This document contains assignments given in LSU EE 4720 over many semesters. It was automatically generated and so some solutions (and possibly some assignments) are likely missing. At the top of each page of each assignment is a link to the original assignment. Those who want to print an assignment might follow that link. All assignments and public solutions are available at https://www.ece.lsu.edu/ee4720/prev.html

## Contents

1 Spring 2023 13
1.1 hw01.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
1.2 hw02.pdf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
1.3 hw03.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
1.4 hw04.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
1.5 hw05.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
1.6 hw06.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
1.7 hw07.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

2 Spring 2022 35
2.1 hw01.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.2 hw02.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.3 hw03.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
2.4 hw04.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
2.5 hw05.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
2.6 hw06.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
2.7 hw07.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
2.8 hw08.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

3 Spring 2021 53
3.1 hw01.pdf|. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.2 hw02.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.3 hw03.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.4 hw04.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
3.5 hw05.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
3.6 hw06.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.7 hw07.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
3.8 hw08.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

4 Spring 2020 74
4.1 hw01.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.2 hw02.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.3 hw03.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
4.4 hw04.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
4.5 hw05.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
4.6 hw06.pdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5 Spring 2019 ..... 91
5.1 hw01.pdf ..... 92
5.2 hw02.pdf ..... 95
5.3 hw03.pdf ..... 97
5.4 hw04.pdf ..... 99
5.5 hw05.pdf ..... 101
5.6 hw06.pdf ..... 103
5.7 hw07.pdf ..... 105
5.8 hw08.pdf ..... 106
5.9 hw09.pdf ..... 108
6 Spring 2018 ..... 109
6.1 hw01.pdf ..... 110
6.2 hw02.pdf ..... 114
6.3 hw03.pdf. ..... 119
6.4 hw04.pdf ..... 123
$6.5 \quad \mathrm{hw} 05 . \mathrm{pdf}$ ..... 124
$6.6 \quad \mathrm{hw} 06 . \mathrm{pdf}$ ..... 127
6.7 hw07.pdf ..... 130
7 Spring 2017 ..... 131
7.1 hw01.pdf. ..... 132
7.2 hw02.pdf. ..... 134
7.3 hw03.pdf ..... 137
7.4 hw04.pdf ..... 139
7.5 hw05.pdf ..... 142
$7.6 \quad \mathrm{hw} 06 . \mathrm{pdf}$ ..... 146
7.7 hw07.pdf ..... 148
$7.8 \mathrm{hw} 08 . \mathrm{pdf}$ ..... 149
8 Spring 2016 ..... 150
8.1 hw01.pdf ..... 151
8.2 hw02.pdf ..... 154
8.3 hw03.pdf ..... 157
8.4 hw04.pdf ..... 159
8.5 hw05.pdf ..... 160
8.6 hw06.pdf ..... 161
$9 \quad$ Spring 2015 ..... 162
9.1 hw01.pdf ..... 163
9.2 hw02.pdf ..... 167
9.3 hw03.pdf ..... 168
9.4 hw04.pdf ..... 170
9.5 hw05.pdf. ..... 173
9.6 hw06.pdf ..... 174
10 Spring 2014 ..... 175
10.1 hw01.pdf ..... 176
10.2 hw02.pdf ..... 178
10.3 hw03.pdf ..... 179
10.4 hw04.pdf ..... 181
10.5 hw05.pdf ..... 184
10.6 hw06.pdf. ..... 185
11 Spring 2013 ..... 186
11.1 hw01.pdff ..... 187
$11.2 \mathrm{hw} 02 . \mathrm{pdf}$ ..... 190
11.3 hw03.pdf ..... 191
11.4 hw04.pdf ..... 193
$11.5 \mathrm{hw} 05 . \mathrm{pdf}$ ..... 195
11.6 hw06.pdf. ..... 196
$11.7 \mathrm{hw} 07 . \mathrm{pdf}$. ..... 198
12 Spring 2012 ..... 199
12.1 hw01.pdf ..... 200
12.2 hw02.pdf ..... 204
12.3 hw03.pdf ..... 205
12.4 hw04.pdf ..... 206
13 Spring 2011 ..... 207
13.1 hw01.pdf ..... 208
13.2 hw02.pdf ..... 209
$13.3 \mathrm{hw} 03 . \mathrm{pdf}$ ..... 210
13.4 hw04.pdf ..... 211
14 Fall 2010 ..... 212
$14.1 \mathrm{hw} 01 . \mathrm{pdf}$ ..... 213
14.2 hw02.pdf ..... 214
$14.3 \mathrm{hw} 03 . \mathrm{pdf}$. ..... 216
14.4 hw04.pdf ..... 217
$14.5 \mathrm{hw} 05 . \mathrm{pdf}$ ..... 219
14.6 hw06.pdf ..... 220
$14.7 \mathrm{hw} 07 . \mathrm{pdf}$ ..... 221
14.8 hw08.pdf ..... 222
15 Spring 2010 ..... 223
15.1 hw01.pdf ..... 224
$15.2 \mathrm{hw} 02 . \mathrm{pdf}$ ..... 226
$15.3 \mathrm{hw} 03 . \mathrm{pdf}$ ..... 228
15.4 hw04.pdf ..... 231
15.5 hw05.pdf ..... 232
16 Spring 2009 ..... 233
16.1 hw01.pdf ..... 234
16.2 hw02.pdf ..... 236
16.3 hw03.pdf ..... 237
16.4 hw04.pdf ..... 238
16.5 hw05.pdf ..... 239
17 Fall 2008 ..... 240
17.1 hw01.pdf ..... 241
17.2 hw02.pdf ..... 243
17.3 hw03.pdf ..... 245
18 Spring 2008 ..... 247
18.1 hw01.pdf ..... 248
18.2 hw02.pdf ..... 249
18.3 hw03.pdf ..... 251
18.4 hw04.pdf. ..... 252
19 Fall 2007 ..... 253
19.1 hw02.pdf ..... 254
19.2 hw03.pdf ..... 256
19.3 hw04.pdf ..... 260
20 Spring 2007 ..... 261
20.1 hw01.pdf ..... 262
20.2 hw02.pdf ..... 264
20.3 hw03.pdf ..... 265
20.4 hw04.pdf ..... 266
21 Fall 2006 ..... 268
21.1 hw01.pdf| ..... 269
21.2 hw02.pdf ..... 270
21.3 hw03.pdf ..... 271
21.4 hw04.pdf ..... 273
22 Spring 2006 ..... 275
22.1 hw01.pdf ..... 276
22.2 hw02.pdf ..... 278
22.3 hw03.pdf ..... 280
22.4 hw04.pdf. ..... 281
22.5 hw05.pdf ..... 283
23 Fall 2005 ..... 285
23.1 hw01.pdf ..... 286
23.2 hw04.pdf ..... 287
23.3 hw05.pdf ..... 289
24 Spring 2005 ..... 290
24.1 hw01.pdf ..... 291
24.2 hw02.pdf ..... 292
24.3 hw03.pdf ..... 293
25 Fall 2004 ..... 294
25.1 hw01.pdf ..... 295
25.2 hw03.pdf ..... 296
26 Spring 2004 ..... 298
26.1 hw01.pdf ..... 299
26.2 hw03.pdf ..... 300
26.3 hw04.pdf ..... 301
26.4 hw05.pdf ..... 303
26.5 hw06.pdf ..... 305
26.6 hw07.pdf ..... 307
27 Fall 2003 ..... 308
27.1 hw01.pdf ..... 309
27.2 hw03.pdf ..... 310
27.3 hw04.pdf ..... 313
28 Spring 2003 ..... 317
28.1 hw01.pdf ..... 318
28.2 hw02.pdf ..... 320
28.3 hw03.pdf ..... 323
28.4 hw04.pdf. ..... 327
28.5 hw05.pdf ..... 329
28.6 hw06.pdf ..... 331
29 Fall 2002 ..... 333
29.1 hw01.pdf ..... 334
29.2 hw02.pdf ..... 336
29.3 hw04.pdf ..... 339
29.4 hw05.pdf ..... 341
29.5 hw06.pdf ..... 342
30 Spring 2002 ..... 343
30.1 hw01.pdf ..... 344
30.2 hw02.pdf ..... 348
30.3 hw03.pdf ..... 351
30.4 hw04.pdf ..... 354
30.5 hw05.pdf. ..... 357
31 Fall 2001 ..... 358
31.1 hw01.pdf ..... 359
31.2 hw02.pdf ..... 361
31.3 hw03.pdf ..... 364
31.4 hw04.pdf ..... 366
$31.5 \mathrm{hw} 05 . \mathrm{pdf}$ ..... 368
32 Spring 2001 ..... 370
32.1 hw01.pdf ..... 371
$32.2 \mathrm{hw} 02 . \mathrm{pdf}$ ..... 373
$32.3 \mathrm{hw} 03 . \mathrm{pdf}$ ..... 377
32.4 hw04.pdf ..... 380
$32.5 \mathrm{hw} 05 . \mathrm{pdf}$ ..... 382
33 Fall 2000 ..... 384
33.1 hw01.pdf ..... 385
33.2 hw02.pdf ..... 386
33.3 hw03.pdf ..... 388
$33.4 \mathrm{hw} 04 . \mathrm{pdf}$ ..... 391
33.5 hw05.pdf ..... 393
34 Spring 2000 ..... 395
$34.1 \mathrm{hw} 01 . \mathrm{pdf}$ ..... 396
34.2 hw02.pdf ..... 397
34.3 hw03.pdf ..... 399
34.4 hw04.pdf ..... 401
$34.5 \mathrm{hw} 05 . \mathrm{pdf}$ ..... 405
34.6 hw06.pdf ..... 407
35 Fall 1999 ..... 408
$35.1 \mathrm{hw} 01 . \mathrm{pdf}$ ..... 409
$35.2 \mathrm{hw} 02 . \mathrm{pdf}$ ..... 411
$35.3 \mathrm{hw} 03 . \mathrm{pdf}$. ..... 416
36 Spring 1999 ..... 420
36.1 hw01.pdf| ..... 421
36.2 hw02.pdf ..... 422
$36.3 \mathrm{hw} 03 . \mathrm{pdf}$ ..... 424
36.4 hw04.pdf. ..... 428
37 Spring 1998 ..... 430
37.1 hw01.pdf. ..... 431
37.2 hw02.pdf ..... 433
$37.3 \mathrm{hw} 03 . \mathrm{pdf}$ ..... 434
37.4 hw04.pdf ..... 435
$37.5 \mathrm{hw} 05 . \mathrm{pdf}$ ..... 436
37.6 hw06.pdf ..... 438
38 Spring 1997 ..... 440
38.1 hw01.pdf ..... 441
38.2 hw02.pdf ..... 442
38.3 hw03.pdf ..... 443
38.4 hw04.pdf ..... 444
38.5 hw05.pdf ..... 445
38.6 hw06.pdf ..... 446
38.7 hw07.pdf ..... 447
38.8 hw08.pdf ..... 448
39 Spring 2023 Solutions ..... 449
39.1 hw01-sol-easy.s.html ..... 450
39.2 hw01-sol-fast.s.html ..... 457
39.3 hw02 sol.pdf ..... 464
39.4 hw03 sol.pdf ..... 468
39.5 hw04 sol.pdf ..... 477
39.6 hw05 sol.pdf ..... 481
39.7 hw06 sol.pdf ..... 485
39.8 hw07 sol.pdf ..... 487
40 Spring 2022 Solutions ..... 488
40.1 hw01 sol.pdf ..... 489
40.2 hw01-sol.s.html ..... 495
40.3 hw02 sol.pdf ..... 501
40.4 hw03 sol.pdf ..... 505
40.5 hw04 sol.pdf ..... 513
40.6 hw05 sol.pdf ..... 518
40.7 hw06 sol.pdf ..... 519
40.8 hw07 sol.pdf ..... 520
40.9 hw08 sol.pdf ..... 521
41 Spring 2021 Solutions ..... 522
41.1 hw01 sol.pdf ..... 523
41.2 hw01-sol.s.html ..... 525
41.3 hw02 sol.pdf ..... 528
41.4 hw02-sol-easy.s.html ..... 540
41.5 hw 02 -sol.s.html ..... 558
41.6 hw03 sol.pdf ..... 576
41.7 hw04 sol.pdf ..... 584
41.8 hw05 sol.pdf ..... 589
42 Spring 2020 Solutions ..... 592
42.1 hw02 sol.pdf ..... 593
42.2 hw03 sol.pdf ..... 596
42.3 hw04 sol.pdf ..... 602
42.4 hw05 sol.pdf ..... 606
42.5 hw06 sol.pdf ..... 609
43 Spring 2019 Solutions ..... 610
43.1 hw01-sol.s.html ..... 611
43.2 hw03 sol.pdf ..... 617
43.3 hw04 sol.pdf ..... 620
43.4 hw05 sol.pdf ..... 623
43.5 hw08 sol.pdf ..... 625
44 Spring 2018 Solutions ..... 628
44.1 hw01-sol.s.html ..... 629
44.2 hw02 sol.pdf ..... 635
44.3 hw03 sol.pdf ..... 643
44.4 hw05 sol.pdf ..... 649
44.5 hw06 sol.pdf ..... 655
45 Spring 2017 Solutions ..... 660
45.1 hw01-sol.s.html ..... 661
45.2 hw02 sol.pdf ..... 664
45.3 hw03 sol.pdf ..... 668
45.4 hw04 sol.pdf ..... 671
45.5 hw05 sol.pdf ..... 674
45.6 hw06 sol.pdf ..... 679
45.7 hw07 sol.pdf ..... 682
45.8 hw08 sol.pdf ..... 683
46 Spring 2016 Solutions ..... 685
46.1 hw01 sol.pdf ..... 686
46.2 hw02 sol.pdf ..... 691
46.3 hw03 sol.pdf ..... 695
46.4 hw04 sol.pdf ..... 698
46.5 hw05 sol.pdf ..... 700
47 Spring 2015 Solutions ..... 701
47.1 hw01 sol.pdf ..... 702
47.2 hw02 sol.pdf ..... 707
47.3 hw03 sol.pdf ..... 710
47.4 hw04 sol.pdf ..... 714
47.5 hw05 sol.pdf ..... 718
47.6 hw06 sol.pdf ..... 719
48 Spring 2014 Solutions ..... 720
48.1 hw01 sol.pdf ..... 721
48.2 hw03 sol.pdf ..... 725
48.3 hw04 sol.pdf ..... 728
48.4 hw05 sol.pdf ..... 733
49 Spring 2013 Solutions ..... 734
49.1 hw01 sol.pdf ..... 735
49.2 hw02 sol.pdf ..... 740
49.3 hw03 sol.pdf ..... 741
49.4 hw04 sol.pdf ..... 744
49.5 hw05 sol.pdf ..... 747
49.6 hw06 sol.pdf ..... 750
49.7 hw07 sol.pdf ..... 753
50 Spring 2012 Solutions ..... 754
50.1 hw01 sol.pdf ..... 755
51 Spring 2011 Solutions ..... 759
51.1 hw01 sol.pdf ..... 760
52 Fall 2010 Solutions ..... 762
52.1 hw01 sol.pdf ..... 763
52.2 hw02 sol.pdf ..... 766
52.3 hw03 sol.pdf ..... 770
52.4 hw04 sol.pdf ..... 772
52.5 hw06 sol.pdf ..... 775
53 Spring 2010 Solutions ..... 777
53.1 hw01 sol.pdf ..... 778
53.2 hw02 sol.pdf ..... 782
53.3 hw03 sol.pdf ..... 785
53.4 hw04 sol.pdf ..... 790
53.5 hw05 sol.pdf ..... 794
54 Spring 2009 Solutions ..... 795
54.1 hw01 sol.pdf ..... 796
54.2 hw02 sol.pdf ..... 800
54.3 hw03 sol.pdf ..... 801
54.4 hw04 sol.pdf ..... 802
54.5 hw05 sol.pdf ..... 804
55 Fall 2008 Solutions ..... 806
55.1 hw01 sol.pdf ..... 807
55.2 hw02 sol.pdf ..... 811
55.3 hw03 sol.pdf ..... 814
56 Spring 2008 Solutions ..... 817
56.1 hw01 sol.pdf ..... 818
56.2 hw02 sol.pdf ..... 820
56.3 hw03 sol.pdf ..... 822
57 Fall 2007 Solutions ..... 824
57.1 hw02 sol.pdf ..... 825
57.2 hw03 sol.pdf ..... 828
57.3 hw04 sol.pdf ..... 835
58 Spring 2007 Solutions ..... 837
58.1 hw01 sol.pdf ..... 838
58.2 hw02 sol.pdf ..... 841
58.3 hw03 sol.pdf ..... 842
58.4 hw04 sol.pdf ..... 844
59 Fall 2006 Solutions ..... 847
59.1 hw02 sol.pdf ..... 848
59.2 hw03 sol.pdf ..... 850
59.3 hw04 sol.pdf ..... 852
60 Spring 2006 Solutions ..... 855
60.1 hw01 sol.pdf ..... 856
60.2 hw02 sol.pdf ..... 859
60.3 hw03 sol.pdf ..... 861
60.4 hw04 sol.pdf ..... 866
60.5 hw05 sol.pdf ..... 870
61 Fall 2005 Solutions ..... 873
61.1 hw01 sol.pdf ..... 874
61.2 hw04 sol.pdf ..... 876
61.3 hw05 sol.pdf ..... 879
62 Spring 2005 Solutions ..... 883
62.1 hw01 sol.pdf ..... 884
62.2 hw02 sol.pdf ..... 886
62.3 hw03 sol.pdf ..... 888
63 Fall 2004 Solutions ..... 890
63.1 hw01 sol.pdf ..... 891
63.2 hw03 sol.pdf ..... 893
64 Spring 2004 Solutions ..... 896
64.1 hw03 sol.pdf ..... 897
64.2 hw04 sol.pdf ..... 899
64.3 hw05 sol.pdf ..... 902
64.4 hw06 sol.pdf ..... 905
64.5 hw 07 sol.pdf ..... 908
65 Fall 2003 Solutions ..... 910
65.1 hw01 sol.pdf ..... 911
65.2 hw03 sol.pdf ..... 914
65.3 hw 04 sol.pdf ..... 918
66 Spring 2003 Solutions ..... 922
66.1 hw01 sol.pdf ..... 923
66.2 hw02 sol.pdf ..... 926
66.3 hw03 sol.pdf ..... 940
66.4 hw04 sol.pdf ..... 945
66.5 hw05 sol.pdf ..... 950
66.6 hw06 sol.pdf ..... 954
67 Fall 2002 Solutions ..... 956
67.1 hw01 sol.pdf ..... 957
67.2 hw02 sol.pdf ..... 959
67.3 hw04 sol.pdf ..... 963
67.4 hw05 sol.pdf ..... 966
67.5 hw06 sol.pdf ..... 971
68 Spring 2002 Solutions ..... 975
68.1 hw01 sol.pdf ..... 976
68.2 hw02 sol.pdf ..... 981
68.3 hw03 sol.pdf ..... 986
68.4 hw04 sol.pdf ..... 991
68.5 hw05 sol.pdf ..... 996
69 Fall 2001 Solutions ..... 997
69.1 hw01 sol.pdf ..... 998
69.2 hw02 sol.pdf ..... 1003
69.3 hw03 sol.pdf ..... 1007
69.4 hw04 sol.pdf ..... 1012
69.5 hw05 sol.pdf ..... 1014
70 Spring 2001 Solutions ..... 1017
70.1 hw01 sol.pdf ..... 1018
70.2 hw02 sol.pdf ..... 1024
70.3 hw03 sol.pdf ..... 1030
70.4 hw04 sol.pdf ..... 1033
71 Fall 2000 Solutions ..... 1038
71.1 hw01 sol.pdf ..... 1039
71.2 hw02 sol.pdf ..... 1041
71.3 hw03 sol.pdf ..... 1044
71.4 hw04 sol.pdf ..... 1048
71.5 hw05 sol.pdf ..... 1050
72 Spring 2000 Solutions ..... 1055
72.1 hw01 sol.pdf ..... 1056
72.2 hw03 sol.pdf ..... 1058
72.3 hw04 sol.pdf ..... 1062
72.4 hw05 sol.pdf ..... 1069
73 Fall 1999 Solutions ..... 1073
73.1 hw01 sol.pdf ..... 1074
73.2 hw02 sol.pdf ..... 1077
73.3 hw03 sol.pdf ..... 1084
74 Spring 1999 Solutions ..... 1089
74.1 hw01 sol.pdf ..... 1090
74.2 hw02 sol.pdf ..... 1093
74.3 hw03 sol.pdf ..... 1096
74.4 hw04 sol.pdf ..... 1102
75 Spring 1998 Solutions ..... 1108
75.1 hw01 sol.html ..... 1109
75.2 hw02 sol.html ..... 1116
75.3 hw 03 sol.html ..... 1118
75.4 hw04 sol.html ..... 1120
75.5 hw05 sol.html ..... 1122
75.6 hw06 sol.html ..... 1126
76 Spring 1997 Solutions ..... 1130
76.1 hw01 sol.pdf| ..... 1131
76.2 hw03 sol.html ..... 1133
76.3 hw04 sol.html ..... 1136
76.4 hw06 sol.html ..... 1139
76.5 hw 07 sol.html ..... 1141
76.6 hw08 sol.html ..... 1143
$\rightarrow$ Spring 2023
$\leftrightarrows \rightarrow$ Homework 1

## 1 Spring 2023

Problem 0: Follow the instructions for class account setup and for homework workflow in https://www.ece.lsu.edu/ee4720/proc.html. Review the comments in hw01.s and look for the areas labeled "Problem 1".

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2023/hw01.s.html. For MIPS references see the course references page,
https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in
https://www.ece.lsu.edu/ee4720/proc.html.
Using LSU version of SPIM
This assignment requires a modified version of the SPIM simulator originally developed by James Larus. Instructions for using this simulator appear on the course procedures page. When running SPIM check the LSU version date, there should be a line reading LSU Version Date: 2022-01-31. Make sure that the date is there and is no earlier than 31 January 2022. (The date will appear on the console output near the top when run non-graphically, and in the lowermost window pane when run graphically.)

## Debugging

To facilitate debugging the code can be run so that the simulator emits a trace of executed instructions, plus an indication of changed register values.

The best way to get a trace is to run the code non-graphically. To do so load the code into an Emacs buffer in a properly set up account. Press Ctrl-F9 to start the simulator non-graphically. That should pop up a window showing a simulator banner followed by a prompt:

```
SPIM Version 6.3.1 lsu of 9 November 2001, 17:34:35 CST
LSU Version Date: 2022-01-31
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Includes LSU modifications.
File loaded.
    Type "run" to run normally.
    Type "step 100" to execute next 100 instructions with tracing.
    Type "help" for more help.
(spim)
```

To see a trace of instructions enter step followed by the number of instructions, say step 100 . This will execute next 100 instructions but will only trace instructions in the lookup routine (as of this writing). Suppose that the lookup routine starts with the following code:
lookup:
addi \$v0, \$0, -1
START_WORD:
addi \$t0, \$a0, 0
addi \$v0, \$v0, 1
Then a trace of execution would produce the following:

| (spim) step 100 |  |  |  |
| :---: | :---: | :---: | :---: |
| [0x004000cc] | 0x4080b000 | mtc0 \$0, \$22 |  |
| [0x00400118] | 0x0c100000 | jal 0x00400000 [lookup] ; 299: jal |  |
| \# Change in \$31 | (\$ra) | 0 -> 0x400120 D | Decimal: 0 -> 4194592 |
| [0x0040011c] | 0x40154800 | mfc0 \$21, \$9 | 300: mfc0 \$s5, \$9 |
| \# Change in \$21 | (\$s5) | $0->0 \times 14$ D | Decimal: 0 -> 20 |
| [0x00400000] | 0x2002ffff | addi \$2, \$0, -1 | ; 16: addi \$v0, \$0, -1 |
| [0x00400004] | 0x20880000 | addi \$8, \$4, 0 | 18: addi \$t0, \$a0, 0 |
| \# Change in \$8 | (\$t0) | 0 -> 0x1001024f D | Decimal: 0 -> 268501583 |
| [0x00400008] | 0x20420001 | addi \$2, \$2, 1 | 19: addi \$v0, \$v0, 1 |

Each line starting with square brackets shows the execution of an instruction. The address of the instruction is shown inside the square brackets. After the square brackets the instruction is shown in three different forms. First encoded, shown in hexadecimal. Then a disassembled form (which is based on the encoded instruction). Finally, after the semicolon the instruction is shown as it appears in the assembler file. Immediately after the semicolon is a line number.

The lines that start with a \# show register values that change. The values are shown both in hexadecimal and decimal. In the example above the first three instructions are from the testbench, the fourth instruction shown, at address $0 \times 400000$, is the first instruction of lookup.

Problem 1: MIPS routine lookup is called with three arguments: $\$ \mathrm{aO}$ holds the address of a string, called the lookup word, \$a1 holds the address of a word table, \$a2 holds the address of a word length table. Complete lookup so that when it returns $\$ \mathrm{v} 0$ is set to the index of the lookup word in the word table (as explained below) or to -1 if the lookup word is not in the word table.

Register $\$$ a1 holds the address of a word table. (Look for table_words: in the file.) The words are stored one after the other in alphabetical order, but without even a null separating them. The first three words are aah, aardvark, able. They are stored in memory as aahaardvarkable. Register $\$ \mathrm{a} 2$ holds the address of a length table. The first entry is the length of the first word, the second entry is the length of the second word, and so on. The lengths are stored as 1-byte unsigned integers. The first three values stored in the length table are 3,8 , and 4 . Note that without the length table it would not be possible to reliably identify the words in the word table.

The index of a word is its position in the word table, with the first word, aah, having index 0 . (The second word, aardvark, has index 1, and so on.)
(The word table, word size table, and the lookup words can be found in hw01.s. Inspecting these might help in understanding the problem. Search for table_words: to find the word table. The other tables are below.)

If lookup is called with $\$ \mathrm{a} 0$ pointing to able then when it returns $\$ \mathrm{v0}$ should hold a 2 . If $\$ \mathrm{a} 0$ points to abacus then $\$ \mathrm{v} 0$ should be set to -1 . (Since the word table is in alphabetical order abacus can't be in the word list even if all we know is that the first three words are aah, aardvark, able.)

The testbench will run lookup on several lookup words and report the number of instructions needed for each lookup word and whether it was correct. Here is the output for the instructor's solution:

| a | Num Insn: | 227 | Index: | -1 | -- Correct |
| :--- | :--- | ---: | :--- | ---: | :--- |
| aah | Num Insn: | 43 | Index: | 0 | -- Correct |
| able | Num Insn: | 62 | Index: | 2 -- Correct |  |
| i | Num Insn: | 230 | Index: | 41 -- Correct |  |



First, complete lookup so that it is correct (shows zero errors). Do not use pseudoinstructions except for nop. Comments in the code show other rules and indicate which registers can be modified. Next, optimize it to reduce the number of executed instructions. There are many ways to do this.

To make the solution fast take advantage of the fact that the word table is in alphabetical order and that the length table provides the length of each word in the table.

Your solution should not rely on extra storage that's set up by an initialization call. For example, if one computes an offset table from the length table then a binary search is possible. That's interesting, but not allowed in this problem. Another possibility is to compute a hash table. (See 2019 Homework 1.) Using a hash table is interesting but not allowed in this assignment either.

Finally, write clear code and add comments appropriate for an expert MIPS programmer. For example, don't explain things that can easily be figured out.

For examples of MIPS program see past Homework 1 assignments in this class.

Problem 1: The code fragment below was taken from the course hex string assembly example. (The hex string example was not covered this semester. The full example can be found at https://www.ece.lsu.edu/ee4720/2022/hex-string.s.html.) The fragment below converts the value in register a0 to an ASCII string, the string is the value in hexadecimal (though initially backward).

LOOP:

```
andi $t0, $a0, Oxf # Retrieve the least-significant hex digit.
srl $a0, $a0, 4 # Shift over by one hex digit.
slti $t1, $t0, 10 # Check whether the digit is in range 0-9
bne $t1, $0, SKIP # Don't forget that delay slot insn always exec.
addi $t2, $t0, 48 # If 0-9, add 48 to make ASCII '0' - '9'.
addi $t2, $t0, 87 # If 10-15, add 87 to make ASCII 'a' - 'z'.
sb $t2, 0($a1) # Store the digit.
bne $a0, $0, LOOP # Continue if value not yet zero.
addi $a1, $a1, 1 # Move string pointer one character to the left.
```

SKIP:
(a) Show the encoding of the MIPS bne a0, 0, LOOP instruction. Note: This is not the same as the instruction used in last year's Homework 2. Include all parts, including - especially-the immediate. For a quick review of MIPS, including the register numbers corresponding to the register names, visit https://www.ece.lsu.edu/ee4720/2023/lmips.s.html.
(b) RISC-V RV32I has a bne instruction too, though it is not exactly the same. Show the encoding of the RV32I version of the bne $a 0,0$, LOOP instruction. For this subproblem assume that the bne will jump backward eight instructions, just as it does in the code sample above.

To familiarize yourself with RISC-V start by reading Chapter 1 of Volume I of the RISC-V specification, especially the Chapter 1 Introduction and Sections 1.1 and 1.3. Skip Section 1.2 unless you are comfortable with operating system and virtualization concepts. Other parts of Chapter 1 are interesting but less relevant for this problem. Also look at Section 2.5 (Control Transfer Instructions). The spec can be found in the class references page at https://www.ece.lsu.edu/ee4720/reference.html.
(c) Consider the four-instruction sequence from the code above:

```
slti $t1, $t0, 10 # Check whether the digit is in range 0-9
bne $t1, $0, SKIP # Don't forget that delay slot insn always exec.
addi $t2, $t0, 48 # If 0-9, add 48 to make ASCII '0' - '9'.
addi $t2, $t0, 87 # If 10-15, add 87 to make ASCII 'a' - 'z'.
```

SKIP:
Re-write this sequence in RISC-V RV32I, and take advantage of RISC-V branch behavior to reduce this to three instructions (plus possibly one more instruction before the loop). For this problem one needs to focus on RISC-V branch behavior. Assume that the RISC-V slti and addi instructions are identical to their MIPS counterparts at the assembly language level. It is okay to retain the MIPS register names. Hint: One change needs to be made for correctness, another for efficiency.

Problem 2: Note: The following problem was assigned in each of the last six years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
# Cycle 0
lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
```

(b) Explain error and show correct execution.

```
# Cycle }\quad0\quad
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB
```

(c) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3
SW r1, O(r4)
IF ID EX ME WB
    IF ID -> EX ME WB
```

(d) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3
xor r4, r1, r5 IF ----> ID EX ME WB
```

Problem 1: Appearing below are incorrect executions on the illustrated implementation. For each execution explain why it is wrong and show the correct execution. Note: This problem was assigned in 2020, 2021, and 2022, and their solutions are available. DO NOT look at the solutions unless you are lost and can't get help elsewhere. Even in that case just glimpse.

(a) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3
IF ID EX ME WB
xor r4, r1, r5 IF ID -> EX ME WB
```

(b) The execution of the branch below has two errors. One error is due to improper handling of the andi instruction. (That is, if the andi were replaced with a nop there would be no problem in the execution below.) The other is due to the way the beq executes. As in all code fragments in this problem, the program is correct, the only problem is with the illustrated execution timing.

```
# Cycle: 0
andi r2, r2, 0xff IF ID EX ME WB
beq r1, r2, TARG IF ID EX ME WB
add r3, r4, r5 IF ID EX ME WB
xor IFx
TARG:
    Sw r6, 7(r8) IF ID EX ME WB
# Cycle: 0
```


(c) Explain error and show correct execution.

```
# Cycle
0
lw r2, O(r4)
add r1, r2, r7
IF ID EX ME WB
    IF ID EX ME WB
```

(d) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3
lw r1, O(r4)
IF ID EX ME WB
    IF ID -> EX ME WB
```

(e) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
    IF ID -> EX ME WB
```

Problem 2: Illustrated below is a MIPS implementation in which each multiplexor has a label, such as a circled A at the multiplexor providing a value for the PC. (The implementation debuted on the 2018 midterm exam.) The multiplexor inputs are also numbered. Below the illustration an execution of the program on the implementation is shown for two iterations of a loop. Below the execution is a table with one row for each labeled multiplexor. Complete the table so that it shows the values on the multiplexors' select signals at each cycle based on the execution. Leave an entry blank if its value does not make a difference.

Wire thicknesses and colors have been varied to make it easier to trace them through the diagram. Before attempting this problem, solve 2018 Midterm Exam Problem 2b, which also appeared as 2022 Homework 3 Problem 2. Also see the 2014 Midterm Exam Problem 1 for a similar problem.


Continued on the next page.

Complete the table (the rows starting with A:, B: etc.) based on the execution below.
Omit select signal values if they do not matter. For example, omit values for $E$ for cycles in which there is not a store instruction in EX.

Assume that the branch is taken the second time it appears. (No assumption needed for its first appearance.)

| addi r1, r1, -4 | IF | ID | EX | ME | WB |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| LOOP: \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |  |  |
| sw r2, 4(r1) |  | IF | ID | EX | ME | WB |  |  |  |  |  |  |  |
| lw r1, 8(r2) |  |  | IF | ID | EX | ME | WB |  |  |  |  |  |  |
| bne r2, r3, LOOP |  |  |  | IF | ID | EX | ME | WB |  |  |  |  |  |
| add $r 2, \mathrm{r} 2$, r6 |  |  |  |  | IF | ID | EX | ME | WB |  |  |  |  |
| LOOP: \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |  |  |
| sw r2, 4(r1) |  |  |  |  |  | IF | ID | EX | ME | WB |  |  |  |
| 1w r1, 8(r2) |  |  |  |  |  |  | IF | ID | EX | ME | WB |  |  |
| bne r2, r3, LOOP |  |  |  |  |  |  |  | IF | ID | EX | ME | WB |  |
| add r2, r2, r6 |  |  |  |  |  |  |  |  | IF | ID | EX | ME | WB |
| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

A:

B:

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

C:

D:

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

E:
F:

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

Problem 3: Show the execution of the code fragments on the following implementations for enough iterations to determine the instruction throughput (IPC). As always, base the behavior of branches and the availability of bypasses on the implementations. Also, don't forget that MIPS branches have a delay slot. Sorry for yelling, but I hate it when students miss things.

This problem appeared as most of Problem 1 on the 2022 Final Exam. A solution is not yet available.


Show execution and $\square$ determine instruction throughput (IPC) based on a large number of iterations.

LOOP:
bne r1, r2, LOOP
addi r1, r1, 4
xor r5, r6, r7
sub r8, r9, r10
Show execution and $\square$ determine instruction throughput (IPC) based on a large number of iterations.

## LOOP:

bne r1, r2, Loop
addi $r 1, r 1,4$
xor r5, r6, r7
sub $\mathrm{r} 8, \mathrm{r} 9$, r10
Show execution and $\square$ determine instruction throughput (IPC) based on a large number of iterations.

```
LOOP:
    bne r1, r2, LOOP
    addi r1, r1, 4
    xor r5, r6, r7
    sub r8, r9, r10
```

In the problems below a new MIPS instruction, integer fmadd, (hypothetical of course) is to be added to our pipelined MIPS implementation. A simpler implement-the-instruction problem was the subject of Fall 2010 Homework 3, in which a shift unit is added to MIPS to implement shift instructions. The 2010 problem is simpler because the shift unit occupies just one stage, while the fmadd for this assignment spans multiple stages. For past assignments in which integer arithmetic hardware spans several stages see 2020 Homework 2, 3, and 4 and 2020 midterm exam Problem 5. In these 2020 problems an integer multiply instruction was to be implemented.

Problem 1: A fused multiply/add instruction, such as fmadd r1, r2, r3, r4, computes $r_{1}=r_{2} r_{3}+r_{4}$. Such instructions are useful for both floating-point and integer calculations, and integer version is considered here. The goal in this problem is to extend MIPS with an integer multiply/add instruction, fmadd. The new fmadd instruction will be encoded in MIPS Format $R$ with the SA field being used to specify the third source register, r4 in the example.


The hardware to compute the multiply/add will consist of two types of units: a carry-save multiplier (CSM) and integer adders (labeled ADD). The connection of these two types of units needed to compute a multiply/add are shown below.


The CSM takes 0.9 clock cycles to compute a result and each adder takes 0.45 clock cycles, so the critical path through the hardware shown above is 1.8 clock cycles. Because the critical path is greater than one clock cycle the hardware cannot be placed in one stage. (Unless the clock frequency were to be decreased from $\phi$ to $\phi / 1.8$, which would slow everything down and so of course we don't want to do it.)

Note: For the three parts below a single hardware solution can be provided. That is, a correct solution to part $c$ also can be a correct solution to parts $b$ and $a$, and so there is no need to draw three hardware designs.
(a) Add the CSM and ADD units to the MIPS implementation below to efficiently implement the fmadd instruction. For this sub-problem provide the hardware needed so that fmadd can execute without stalls when there are no nearby dependencies, such as in the execution below.

```
# There are no dependencies in this code fragment.
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sub r4, r5, r6 IF ID EX ME WB
fmadd r7, r8, r9, r10 IF ID EX ME WB
fmadd r11, r12, r13, r14 IF ID EX ME WB
xori r15, r16, 17 IF ID EX ME WB
# Cycle 0
```

Put your solution on the larger diagram several pages ahead.


Put your solution on the larger diagram several pages ahead.
Add the CSM and ADD units to the implementation above so that the can implement the fmadd instruction.
Provide the datapath needed so that operands can reach the CSM and ADD units andthe result can reach the register file.Don't forget that this instruction has three source operands.Do not increase the critical path.As always, consider cost. Assume that an $n$-bit register costs twice as much as an $n$-bit, 2-input multiplexor.fmadd should execute without stalls when there are no nearby dependencies.
$\square$ Do not design control logic for this assignment.
(b) In the code fragments below the fmadd depends on prior instructions.

Add bypass paths to the fmadd implementation so that all of the executions below are possible.

```
# Fragment A
# Cycle 0
add R1, r2, r3 IF ID EX ME WB
sub R4, r5, r6 IF ID EX ME WB
fmadd r7, R1, R4, r9 IF ID EX ME WB
# Fragment B
# Cycle
sub R9, r5, r6
fmadd R7, r1, r4, R9
fmadd r2, r3, r5, R7
# Cycle
0
    IF ID EX ME WB
        IF ID EX ME WB
        IF ID EX ME WB
0
# Fragment C
# Cycle 0
add R1, r2, r3
IF ID EX ME WB
lw R9, O(r10) IF ID EX ME WB
fmadd r7, R1, r4, R9 IF ID EX ME WB
```

(c) Using additional ADD unit(s) modify the implementation so that it can execute Fragments L and D correctly. This will require some tricky bypassing. Note that stalls will be needed when the dependent instruction following the fmadd does not use the adder, such as in Fragment E. Note: In the original problem just one adder was to be used. That is probably impossible without critical path impact.

Add a second adder and bypass paths so that fragments $L$ and $D$ execute as shown.

```
# Fragment L
# Cycle 0
fmadd R7, r1, r4, r9 IF ID EX ME WB
lw r10, 16(R7) IF ID EX ME WB # No stall!
# Fragment D
# Cycle 0
fmadd R7, r1, r4, r9 IF ID EX ME WB
add r2, R7, r3 IF ID EX ME WB # No stall!
# Fragment E
# Cycle 0
fmadd R7, r1, r4, r9 IF ID EX ME WB
or r2, R7, r3
    IF ID -> EX ME WB # A stall. :-(
```

Use the diagram below for your solution, or download https://www.ece.1su.edu/ee4720/2023/mpipei3.svg and edit with your favorite SVG editor. (The diagram was drawn with Inkscape.)


Problem 1: In this problem consider the encoding of integer and floating-point addition instructions in MIPS and RISC-V. Descriptions of MIPS and RISC-V are linked to the course references page, https://www.ece.lsu.edu/ee4720/reference.html.
(a) Show the encoding of MIPS instructions add r1, r2, r3 and add.s $f 4, f 5, f 6$.
(b) Show the encoding of RISC-V RV32IF instructions add x7, x8, x9 and fadd f10, f11, f12.
(c) Notice that the register fields in the integer and floating-point RISC-V RV32IF instructions are the same, while the register fields in the two MIPS instructions are different. One possible reason for RISC-V's matching fields was to simplify implementations of the Zfinx variant. (Web search for it.) How do the matching fields reduce the cost of implementations of the RISC-V Zfinx variant?

## ARM A64 Background

The following background will help in solving the next problem. MIPS and many other ISAs have a set of integer registers and a set of floating-point registers. Many newer ISAs, including ARM A64, have a set of vector registers in lieu of floating-point registers. Extensions of legacy ISAs, such as Intel 64 AVX2, have vector registers but retain floating-point registers for compatibility.

In many ISAs, including ARM A64, a vector register can be used to hold one FP value, just as a traditional FP register would, or a vector register can hold several values. Scalar instructions read or write one value per vector register, and vector instructions read and write multiple values per register.

In ARM A64 there are 32 128-bit vector registers, named v0 to v31. When used in scalar instructions operating on single-precision FP values they are known by the names s0 to s31 and by the names d0 to d31 by double-precision scalar instructions. For example, the ARM A64 assembler instruction fadd s0, s1, $s 2$ computes $s 0=s 1+s 2$ and fadd $d 0$, $d 1$, $d 2$ computes $d 0=d 1+d 2$. In both cases the operands were taken from vector registers v0, v1, and v2. The assembler name s0 means use the low 32 bits of v0 and interpret the value as an IEEE 754 single. The assembler name d0 means use the low 64 bits of v0 and interpret the value as an IEEE 754 double.

In the next problem, the one with sum_thing_unusual, the ARM code contains only scalar floating point instructions and base (integer register) instructions. To solve the next problem one needs to look up instructions in the ARM Architecture Reference Manual. Instructions that operate on vector registers, including fadd can be found in the Advanced SIMD and Floating Point section, C7.2 for the list of instructions. Other instructions can be found in the A64 Base Instruction section, C6.2 for the list of instructions.

Vector instructions are not needed in this assignment, but they will be briefly described anyway. For vector instructions the vector register name indicates how many elements in the vector to use, and what their format is. For example, v0.4s, means to use vector register v0 and to split its 128 bits into 432 -bit lanes, with each lane holding one float (the s). The names can be used in instructions such as fadd v0.4s, $\mathrm{v} 0.4 \mathrm{~s}, \mathrm{v} 1.4 \mathrm{~s}$. This instruction performs four additions, one on each lane of the vector register.

Problem 2: Appearing below is a $\mathrm{C}++$ procedure with a for loop that computes the sum of elements in an array. This would be a totally ordinary loop were it not for the fact that the iteration variable, i, and the increment, delta, are both floats. Since $i$ is a float the number of iterations, depending on delta, can be less than 1024 (say, if delta=2.3) or more than 1024 (say, if delta=0.25). Below the C code are MIPS-I and ARM A64 assembler versions of the loop. Yes, that means you don't have to write them! (The MIPS-I code was hand written, and the A64 was based on code generated by a compiler.) Notice that the ARM code is shorter than the MIPS code. That's because some of the ARM instructions do the equivalent of several MIPS instructions.

Next to each ARM instruction indicate the MIPS instruction(s) from the MIPS code that it corresponds to.

When an ARM instruction corresponds to more than one MIPS instruction explain what the ARM instruction is doing.

A short reference for MIPS floating-point instructions is the course lfp.s notes. This should be sufficient for all but the MIPS-II trunc instruction. For the trunc instruction see the MIPS documentation (linked to the course ISA page).

```
float sum_thing_unusual( float *a, float delta ) {
    float sum = 0;
    for ( float i = 0; i < 1024; i += delta ) sum += a[int(i)];
    return sum;
}
```

\# MIPS Code for sum_thing_unusual.
\#
\# \$a0: The address of array a.
\# \$f0: i. At this point it contains a zero.
\# \$f4: delta.
\# \$f5: The constant 1024, in FP format.
\# \$f8: sum. At this point it contains a zero.
LOOP:
trunc.w.s \$f6, \$f0 \# Note: This is a MIPS-II instruction.
mfc1 \$t1, \$f6
sll \$t1, \$t1, 2
add \$t2, \$t1, \$a0
lwc1 \$f7, 0(\$t2)
add.s \$f0, \$f0, \$f4
c.lt.s \$f0, \$f5
bc1f LOOP
add.s \$f8, \$f8, \$f7
.arch arm
© ARM A64 Code for sum_thing_unusual.
©
© x0-x31: Integer registers. x 31 is sometimes the zero register.
@ s0-s31: Scalar single-precision floating-point registers.
©
© x0: The address of array a
© s0: sum. At this point it contains a zero.
@ s1: i. At this point it contains a zero.
© s3: delta.
© s4: The contains 1024, in FP format.
LOOP:
fcvtzs x1, s1
fadd s1, s1, s3
fcmpe s1, s4
ldr s2, [x0, x1, lsl 2]
fadd s0, s0, s2
bmi LOOP

Problem 1: Solve 2022 Final Exam Problem 2, in which code fragments are either written or analyzed for our MIPS FP implementation.

Problem 2: Solve the last part of 2022 Final Exam Problem 1, the one with the 4 -way superscalar pipeline. (You can tell it's 4-way because the superscripts range from 0 to 3.) There is no need to show superscripts on the stage labels in your execution diagram. For sample problems see past final exams, such as 2021 Problem 2.

Problem 3: Appearing below is our MIPS FP implementation but with an unpipelined FP add unit. Some of the control logic needed to generate stalls when a FP add instruction is in flight is in the magic cloud labeled "Future HW Solution". Design that logic. For similar logic see the logic on the Partially Pipelined pages from Set 9 slides (about page 14). Hint: This does not require much hardware. For similar problems see 2020 Spring Homework 5 and 2020 Spring Final Exam Problem 2.

Use the execution below to help you design the hardware:

```
# Cycle 
add.d f0, f2, f4 IF ID A A A A WF
add.d f6, f8, f10 IF ID -------> A A A A WF
addi r1, r1, 8 IF -------> ID EX ME WB
add.d f12, f18, f14 IF ID ----> A A A A WF
# Cycle }0
```

An SVG version of the image can be found at https://www.ece.lsu.edu/ee4720/2023/hw06-fp-aaaa.svg, use Inkscape or some other SVG editor, or even a text editor.


Problem 1: Solve 2022 Final Exam Problem 3, in which hardware is added to a variation on a 2-way superscalar MIPS implementation.

## 2 Spring 2022

Problem 0: Follow the instructions for class account setup and for homework workflow in https://www.ece.lsu.edu/ee4720/proc.html. Review the comments in hw01.s and look for the areas labeled "Problem 1" and "Problem 2."

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2022/hw01.s.html. For MIPS references see the course references page,
https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in https://www.ece.lsu.edu/ee4720/proc.html.

This Assignment
In class as MIPS review we wrote a routine, strlen, to find the length of a C string. In our completed routine (shown below) the main loop consisted of three instructions, and would load one character per iteration. Therefore at best it could run at the rate of $\frac{1}{3}$ characters per instructions.

```
strlen:
```

```
# Register Usage
# $a0: Address of first character of string.
# $v0: Return value, the length of the string.
addi $v0, $a0, 1 # Set aside a copy of the string start + 1.
    Ibu $t0, 0($a0) # Load next character in string into $t0
    bne $t0, $0, LOOP # If it's not zero, continue
    addi $a0, $a0, 1 # Increment address. (Note: Delay slot insn.)
jr $ra
sub $v0, $a0, $v0
```

LOOP:

Can we do better? Since the main loop only consists of three instructions there is little that can be done to make it shorter, at least using MIPS I instructions. Notice that a character is loaded using lbu (load byte unsigned). Suppose instead a lw (load word) were used. Then four characters would be loaded. If our loop body contained 12 instructions (including the lw) then it would execute at the same rate as our original strlen because it would operate on 4 characters per 12 instructions or at the rate of $\frac{1}{3}$ characters per instruction. If we could somehow check for a null with fewer than 12 instructions our new code would be faster.

In Problem 1 such a string length routine is to be completed. It is assumed that most students' MIPS skills are rusty so the starting point is code using a lhu instruction. In the solution to Problem 1 I attained a rate of 0.392 char/insn, not much better than .329 attained by our original routine.

In Problem 2 the strlen routine is to be written using additional non MIPS-I instructions. These include orc.b from a RISC-V extension, and clz and clo from MIPS32 (based on their r6 versions). Using these instructions my solution achieves 0.942 chars per insn.

## Test Routine

The code for this assignment includes a test routine that runs three string length routines: the routines to be written for Problems 1 and 2, and the string length routine written in class (called strlen_ref here). Each routine is run on several strings, including all lengths from 0 to 5 , plus strings of length 23 and 196. The shorter-length strings are there to make sure that the routines
are correct and to check how fast they are on short strings. The longest string is there to test performance. The performance numbers from the previous section are based on the longest string.

Here is the output from the unmodified assignment:
** Starting Test of Routine "strlen_p1 (Problem 1 - Bit Ops)" **
String 1: Length 1 is correct. Took 10 insn or 0.100 char/insn
String 2: Length 2 is correct. Took 13 insn or 0.154 char/insn
String 3: Length 3 is correct. Took 16 insn or 0.188 char/insn
String 4: Length 4 is correct. Took 19 insn or 0.211 char/insn
String 5: Length 5 is correct. Took 22 insn or 0.227 char/insn
String 6: Length 0 is correct. Took 7 insn or 0.000 char/insn
String 7: Length 23 is correct. Took 76 insn or 0.303 char/insn
String 8: Length 196 is correct. Took 595 insn or 0.329 char/insn
** Starting Test of Routine "strlen_p2 $\quad$ (Problem $2-\mathrm{RISC}$ V orc insn)" **
String 1: Length 1 is correct. Took 11 insn or 0.091 char/insn
String 2: Length 2 is correct. Took 15 insn or 0.133 char/insn
String 3: Length 3 is correct. Took 19 insn or 0.158 char/insn
String 4: Length 4 is correct. Took 23 insn or 0.174 char/insn
String 5: Length 5 is correct. Took 27 insn or 0.185 char/insn
String 6: Length 0 is correct. Took 7 insn or 0.000 char/insn
String 7: Length 23 is correct. Took 99 insn or 0.232 char/insn
String 8: Length 196 is correct. Took 791 insn or 0.248 char/insn
** Starting Test of Routine "strlen_ref (Simple strlen routine.)" **
String 1: Length 1 is correct. Took 9 insn or 0.111 char/insn

String 2: Length 2 is correct. Took 12 insn or 0.167 char/insn
String 3: Length 3 is correct. Took 15 insn or 0.200 char/insn
String 4: Length 4 is correct. Took 18 insn or 0.222 char/insn
String 5: Length 5 is correct. Took 21 insn or 0.238 char/insn
String 6: Length 0 is correct. Took 6 insn or 0.000 char/insn
String 7: Length 23 is correct. Took 75 insn or 0.307 char/insn
String 8: Length 196 is correct. Took 594 insn or 0.330 char/insn
To see all of this output when running graphically it might be necessary to make the pop-up window larger. It is possible to scroll the text in the pop-up window by focusing the window and using the arrow keys.

Each line shows the result from one string. The length of the string is shown, as well as the number of instructions executed in the string length routine, and the execution rate. If the returned length had been wrong both the returned and correct length would be shown but the instruction count would be omitted.

The strings themselves can be found in the test code after the str label. The testbench does not print out the strings, just their lengths. Feel free to modify the strings if it helps with debugging, but please restore them before the deadline.
Using LSU version of SPIM
This assignment requires a modified version of the SPIM simulator originally developed by James Larus. Instructions for using this simulator appear on the course procedures page. When running SPIM check the LSU version date, there should be a line reading LSU Version Date: 2022-01-31. Make sure that the date is there and is no earlier than 31 January 2022. (The date will appear
on the console output near the top when run non-graphically, and in the lowermost window pane when run graphically.)

Two changes were made for this assignment: implementation of the RISC-V-like orb.c instruction, and implementation of the MIPS32 r6 (revision 6) clo (count leading ones) instruction. Also new is the ability to start and stop tracing.

## Debugging

To facilitate debugging the code can be run so that the simulator emits a trace of executed instructions, plus an indication of changed register values. The trace will mostly include the three string length routines, but it will also include a few testbench instructions. The trace includes line numbers so that there should be no confusion about where an instruction is from.

The best way to get a trace is to run the code non-graphically. To do so load the code into an Emacs buffer in a properly set up account. Press Ctrl-F9 to start the simulator non-graphically. That should pop up a window showing a simulator banner followed by a prompt:

```
SPIM Version 6.3.1 lsu of 9 November 2001, 17:34:35 CST
LSU Version Date: 2022-01-31
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Includes LSU modifications.
File loaded.
    Type "run" to run normally.
    Type "step 100" to execute next 100 instructions with tracing.
    Type "help" for more help.
(spim)
```

At the prompt enter step 100 to run the next 100 instructions. The instructions in the string length routines will be traced, but the count of 100 instructions also includes the test routine (as of this writing). For example:

```
(spim) step 100
[0x00400064] 0x4080b000 mtc0 $0, $22 ; 218: mtc0 $0, $22
** Starting Test of Routine "strlen_p1 (Problem 1 - Bit Ops)" **
[0x004000d0] 0x0100f809 jalr $31, $8 ; 251: jalr $t0
# Change in $31 ($ra) 0x4000bc -> 0x4000d8 Decimal: 4194492 -> 4194520
[0x004000d4] 0x40154800 mfc0 $21, $9 ; 252: mfc0 $s5, $9
# Change in $21 ($s5) 0 -> 0x23 Decimal: 0 -> 35
[0x00400000] 0x20820000 addi $2, $4, 0 ; 84: addi $v0, $a0,
# Change in $2 ($v0) Oxfffffffff -> 0x10010000 Decimal: -1 -> 268500992
[0x00400004] 0x94880000 lhu $8, 0($4) ; 87: lhu $t0, 0($a0)
# Change in $8 ($t0) 0x400000 -> 0x3100 Decimal: 4194304 -> 12544
[0x00400008] 0x3109ff00 andi $9, $8, -256 ; 88: andi $t1, $t0, 0xff00\
# Change in $9 ($t1) 0x100101f0 -> 0x3100 Decimal: 268501488 -> 12544
[0x0040000c] 0x11200006 beq $9, $0, 24 [DONE0-0x0040000c]; 89: beq $t1, $0, DONEO\
[0x00400010] 0x310900ff andi $9, $8, 255 ; 90: andi $t1, $t0, 0xff\square
```

Each line starting with square brackets shows the execution of an instruction. The address of the instruction is shown inside the square brackets. After the square brackets the instruction is
shown in three different forms. First encoded, shown in hexadecimal. Then a disassembled form (which is based on the encoded instruction). Finally, after the semicolon the instruction is shown as it appears in the assembler file. Immediately after the semicolon is a line number.

The lines that start with a \# show register values that change. The values are shown both in hexadecimal and decimal.

Problem 1: Routine strlen_p1 in hw01.s computes the length of a string using a loop that loads two characters at a time. It achieves a rate of .329 char/insn. Modify it so that it uses a lw instead of 1 hu . (Note that there is no such thing as lwu in MIPS I. Such an instruction only makes sense if registers are larger than 32 bits.) It is possible to achieve .393 chars /insn, or maybe even faster.

The string starting address will be in register a0. That address will be a multiple of 4 . Strings end with a null (a zero). The byte after the null is not part of the string and can be of any value. Don't assume it is a particular value.

Your solution should use MIPS-I instructions and should not use pseudo instructions except for nop. See the check-box comments (such as [ ] Code should be efficient.) for additional restrictions, requirements, and reminders.

Problem 2: Complete strlen_p2 so that it determines the string length by loading four characters (using a 1 w ) and checks for the null using the RISC-V-like orc.b (Bitwise OR-Combine, byte granule) instruction. Also helpful will be the MIPS32 r6 clz and clo instructions.

The orc.b instruction is part of the RISC-V bit manipulation ISA extensions. See the documentation for this instruction for details on what it does. The documentation is linked to the course references page and of course can be found on the RISC-V site. The orc.b is in the strlen_p2 routine, but it doesn't do anything useful. Of course, that should be changed as part of the solution.

The MIPS32 clz and clo might also come in handy. Look for the MIPS32 r6 (not the older versions) Volume 2 manuals on the course references page.

It is possible to complete this so that it runs at 0.947 char /insn or faster.

Problem 1: The code fragment below was taken from the course hex string assembly example. (The hex string example was not covered this semester. The full example can be found at https://www.ece.lsu.edu/ee4720/2022/hex-string.s.html.) The fragment below converts the value in register a0 to an ASCII string, the string is the value in hexadecimal (though initially backward).

LOOP:

```
andi $t0, $a0, Oxf # Retrieve the least-significant hex digit.
srl $a0, $a0, 4 # Shift over by one hex digit.
slti $t1, $t0, 10 # Check whether the digit is in range 0-9
bne $t1, $0, SKIP # Don't forget that delay slot insn always exec.
addi $t2, $t0, 48 # If 0-9, add 48 to make ASCII '0' - '9'.
addi $t2, $t0, 87 # If 10-15, add 87 to make ASCII 'a' - 'z'.
sb $t2, 0($a1) # Store the digit.
bne $a0, $0, LOOP # Continue if value not yet zero.
addi $a1, $a1, 1 # Move string pointer one character to the left.
```

SKIP:
(a) Show the encoding of the MIPS bne t1, 0, SKIP instruction. Include all parts, including-especiallythe immediate. For a quick review of MIPS, including the register numbers corresponding to the register names, visit https://www.ece.lsu.edu/ee4720/2022/lmips.s.html.
(b) RISC-V RV32I has a bne instruction too, though it is not exactly the same. Show the encoding of the RV32I version of the bne t1, 0 , SKIP instruction. For this subproblem assume that the bne will jump ahead two instructions, just as it does in the code sample above.

To familiarize yourself with RISC-V start by reading Chapter 1 of Volume I of the RISC-V specification, especially the Chapter 1 Introduction and Sections 1.1 and 1.3. Skip Section 1.2 unless you are comfortable with operating system and virtualization concepts. Other parts of Chapter 1 are interesting but less relevant for this problem. Also look at Section 2.5 (Control Transfer Instructions). The spec can be found in the class references page at https://www.ece.lsu.edu/ee4720/reference.html.
(c) Consider the four-instruction sequence from the code above:

```
slti $t1, $t0, 10 # Check whether the digit is in range 0-9
bne $t1, $0, SKIP # Don't forget that delay slot insn always exec.
addi $t2, $t0, 48 # If 0-9, add 48 to make ASCII '0' - '9'.
addi $t2, $t0, 87 # If 10-15, add 87 to make ASCII 'a' - 'z'.
```

SKIP:
Re-write this sequence in RISC-V RV32I, and take advantage of RISC-V branch behavior to reduce this to three instructions (plus possibly one more instruction before the loop). For this problem one needs to focus on RISC-V branch behavior. Assume that the RISC-V slti and addi instructions are identical to their MIPS counterparts at the assembly language level. It is okay to retain the MIPS register names. Hint: One change needs to be made for correctness, another for efficiency.

Problem 2: Note: The following problem was assigned in each of the last six years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
# Cycle 0
Iw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
```

(b) Explain error and show correct execution.

```
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB
```

(c) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3 IF ID EX ME WB
SW r1, O(r4) IF ID -> EX ME WB
```

(d) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3
xor r4, r1, r5 IF ----> ID EX ME WB
```

Note: The following problems (or very similar problems) were assigned in 2020 and 2021, and their solutions are available. DO NOT look at the solutions unless you are lost and can't get help elsewhere. Even in that case just glimpse.

Problem 1: Appearing below are incorrect executions on the illustrated implementation. Notice that this implementation is different than the one from the previous problem. For each execution explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
# Cycle }\quad0\begin{array}{llllllllll}{1}&{1}&{2}&{3}&{4}&{5}&{6}&{7}
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ID -> EX ME WB
```

(b) The execution of the branch below has two errors. One error is due to improper handling of the andi instruction. (That is, if the andi were replaced with a nop there would be no problem in the execution below.) The other is due to the way the beq executes. As in all code fragments in this problem, the program is correct, the only problem is with the illustrated execution timing.

```
# Cycle: 0
andi r2, r2, 0xff IF ID EX ME WB
beq r1, r2, TARG IF ID EX ME WB
add r3, r4, r5 IF ID EX ME WB
xor IFx
TARG:
    sw r6, 7(r8) IF ID EX ME WB
    # Cycle: 0
```


(c) Explain error and show correct execution.

```
# Cycle
0
lw r2, O(r4)
add r1, r2, r7
IF ID EX ME WB
    IF ID EX ME WB
```

(d) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3
lw r1, O(r4)
IF ID EX ME WB
    IF ID -> EX ME WB
```

(e) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3 IF ID EX ME WB
SW r1, O(r4)
    IF ID -> EX ME WB
```

Problem 2: Appearing below is the labeled MIPS implementation from 2018 Midterm Exam Problem 2(b), and as in that problem each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.

(a) Use F0. Don't be fancy about it, just one instruction is all it takes.
(b) Use F0, C2, and D3 at the same time. The code should not suffer a stall. More than one instruction is needed for the solution. Note: This is new in 2022.
(c) Explain why its impossible to use E0 and D0 at the same time.

Problem 3: This problem appeared as Problem 2c on the 2020 final exam. Appearing below is our bypassed, pipelined implementation followed by a code fragment.

It might be helpful to look at Spring 2019 Midterm Exam Problem 4a. That problems asks for the execution of a loop and for a performance measure based upon how fast that loop executes.

(a) Show the execution of the code below on the illustrated implementation up to the point where the first instruction, addi r2,r2,16, reaches WB in the second iteration.

LOOP:
addi $r 2, r 2,16$
lw r1, 8(r2)
sw r1, 12(r3)
bne r3, r4, LOOP
addi r3, r3, 32
sub r10, r3, r2
(b) Based on your execution determine how many cycles it will take to complete $n$ iterations of the loop.

Problem 1: Appearing below is our familiar five stage MIPS implementation with a new branch bypass path shown in blue. For this problem assume that orc.b is executed by the ALU.

(a) The code below is based on a solution to Homework 1. Show a pipeline execution diagram of this code on the illustrated hardware. Pay close attention to the behavior of the branch including behavior due to dependencies with prior instructions. Show enough of the execution to compute the instruction throughput in units of IPC.

Show execution on the illustrated hardware. $\quad \square$ Compute the instruction throughput (IPC). $\square$ Pay attention to dependencies and available bypass paths.
lw \$to, $0(\$ \mathrm{a} 0)$
LOOPB:

```
addi $a0, $a0, 4
orc.b $t1, $t0
beq $t1, $t3, LOOPB
lw $t0, O($a0)
```

(b) The code below should have executed more slowly on the illustrated implementation. Explain why. Hint: The only difference in the code is the branch instruction.

```
    lw $t0, O($a0)
LOOPB:
    addi $a0, $a0, 4
    orc.b $t1, $t0
    beq $t3, $t1, LOOPB
    lw $t0, O($a0)
```

Explain why the code above executes more slowly.

Problem 2: Appearing below is the implementation used in the previous problem. Add control logic for the branch condition multiplexor (shown in blue). Feel free to insert an is Branch logic block to detect the presence of a branch based on the instruction opcode. For an Inkscape SVG version of the implementation follow https://www.ece.lsu.edu/ee4720/2022/hw04-br-byp.svg.


Problem 3: Appearing below is our MIPS implementation (the one we use, we're not taking credit for inventing it) with an orc.b unit in the EX stage. Unlike the first problem in this assignment, here the orc.b instruction is executed by its own unit, not by the ALU. One reason is because orc.b is fairly easy to compute, and so its output can be available much sooner than the ALU's output. In fact, it will be available early enough to be bypassed to ID for use in determining the branch condition.

Connect the orc.b functional unit so that it can be used by orc.b instructions. Paying attention to cost, connect it so that the following bypasses are possible: (1) A bypass so that an immediately following dependent branch does not stall. This would eliminate a stall in a solution to Problem 1, and avoid a stall in Case 1 in the code fragment below. (2) Bypasses to the next two arithmetic/logical instructions. See Case 2 below.

When weighing design alternatives assume that one pipeline latch bit cost twice as much as one multiplexor bit. Don't overlook opportunities to reuse existing hardware. The Inkscape SVG source for the diagram below is at https://www.ece.lsu.edu/ee4720/2022/hw04-orc.svg.

```
# Case 1
orc.b R1, r9
beq R1, r10, TARG
# Case 2
orc.b R1, r9
add r2, R1, r3 # Bypass from ME
xor r4, R1, r5 # Bypass from WB
or r6, R1, r7 # No bypass needed.
```Connect orc.b unit so code above executes without a stall.
Show control logic for any multiplexors added. (Control logic does not need to be shown for the branch condition mux.)

As always, avoid costly, inefficient, and unclear solutions.

\(\square\)

Problem 1: Solve 2021 Final Exam Problem 2(a) and (b). Problem 2(a) asks for a pipeline execution diagram for the execution of floating-point code on a MIPS implementation which is a little different than the ones in the class notes. Problem 2(a) also asks for additional information, including the instruction throughput. In Problem 2(b) the floating-point code is to be scheduled to improve the throughput. Note: A brief summary of the problem is provided here to reduce the chance that you solve the wrong problem, say by getting the year or problem number wrong.

Problem 1: Solve 2021 Final Exam Problem 2(c). In the problem the execution of a loop on a 4 -way superscalar MIPS implementation is to be shown.

Problem 2: Solve 2021 Final Exam Problem 1. In this problem some features of an unconventional 2-way superscalar processor are to be completed. The solution to this problem is not as long as it might seem.
\(\square\)

Problem 1: Solve 2021 Final Exam Problem 3 (all parts). The problem has some routine predictor analysis questions, how to craft a side-channel attack exploiting of local predictor that does not reset its tables at context switches, and questions about a bimodal predictor with a separate tag store (as covered in class on Friday). For example local predictor analysis problems see prior years' final exams.

Problem 1: Solve 2021 Final Exam Problem 4 parts a,b,c,d. (Don't solve 4e). These are an assortment of short answer questions, covering superscalar and vector processors, and other topics.

\section*{\(3 \quad\) Spring 2021}

Problem 0: Follow the instructions for class account setup and for homework workflow in https://www.ece.1su.edu/ee4720/proc.html. Review the comments in hw01.s and look for the area labeled "Problem 1."

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2021/hw01.s.html. For MIPS references see the course references page, https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in https://www.ece.lsu.edu/ee4720/proc.html.

Problem 1: The hw01.s file has a routine called getbit.
(a) Complete the getbit routine so that it returns the value of a bit from a bit vector that spans one or more bytes. Register \$a0 holds the start address of the bit vector and register \$a1 holds the bit number to retrieve. The most-significant bit of the first byte is bit number 0 . When getbit returns register \(\$ \mathrm{v} 0\) should be set to 0 or 1.

For example, a 16 -bit bit vector is specified in the assembler below starting at the label bit_vector_start:
bit_vector_start:
.byte 0xc5, 0x1f
In binary this would be \(1100010100011111_{2}\). If getbit were called with \(\$ a 1=0\) then bit number zero, meaning the leftmost bit in \(1100010100011111_{2}\), should be returned and so \(\$ v 0=1\). For \(\$ \mathrm{a} 1=2\) a 0 should be returned.

Each memory location holds eight bits of the bit vector. For \$a1 values from 0 to 7 the bit will be in the byte at address \(\$ \mathrm{a} 0\). For \(\$ \mathrm{a} 1\) values from 8 to 15 the bit will be in the byte at address \(\$ \mathrm{a} 0+1\), and so on.

When the the code in hw01.s is run (by pressing F9 for example) a testbench routine will call getbit several times. For each call the testbench will print the value returned by getbit (meaning the value of \(\$ \mathrm{v} 0\) ), whether that value is correct, and if wrong, the correct value. At the end it will print the number of incorrect values returned by getbit, which hopefully will be zero when you're done.

See the checkboxes in the code for more information on what is expected.
(b) The bit vector used by the testbench is specified with:
```

bit_vector_start: \# Note: MIPS is big-endian.
.byte 0xc5, 0x1f
.half 0x05af
.word 0xedcba987
.ascii "123"
bit_vector_end:

```

The assembler will convert the lines following data directives .byte, .half, .word, and .ascii into binary and place them in memory. The total size will be \(2 \times 1+2+4+3=11\) bytes. For the purposes of this problem those 11 bytes form a \(11 \times 8=88\)-bit bit vector. In most circumstances for something like the bit vector above one would use the same kind of data directive for all data, say using only .byte, but mixing directives is not wrong and in some cases may be convenient for
example when the bit vector is constructed by concatenating pieces of different sizes and types. Note that the kind of data directives used above does not affect how getbit is written.

Following the bit vector are the tests for the testbench. For each test there is one line consisting of a bit number and the expected return value. For example, the second test sets \(\$\) a1=4 and expects a return value of \(\$ \mathrm{v} 0=0\).
```

testdata:
.half 0, 1
.half 4, 0
.half 10, 0

```

Add a test to the testdata data to test the part of the bit vector specified using . ascii "123". The test should be written for .ascii "123" and should report an error if the directive were changed to . ascii "213".

Problem 0: Follow the instructions for class account setup and for homework workflow in https://www.ece.lsu.edu/ee4720/proc.html. Review the comments in hw02.s and look for the area labeled "Problem 1."

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2021/hw02.s.html. For MIPS references see the course references page,
https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in https://www.ece.lsu.edu/ee4720/proc.html.

This Assignment
One goal of this assignment is to build assembly language proficiency by working with data at different sizes and by traversing a tree. The sizes are bits for the compressed text, words for the array of compressed text, half (2-byte) for the tree, and bytes for the dictionary. Another goal is to provide a starting point for architectural improvements. That is, ISA and hardware changes to make code go faster.

\section*{Huffman Compression Background}

One way to compress data is to divide it up into pieces, compute a Huffman coding for the pieces, then replace each piece with its Huffman code. The size of a Huffman code can vary from 1 bit (yes, just one), to an arbitrarily long bit vector. Pieces that appear more frequently in the original text will have short codes and pieces that appear less frequently will have longer codes. Consider a file containing English text, such as the source file for the Homework 1 handout. One way of dividing it to pieces is to make each character a piece. For Homework 1 a space was the most frequent piece ( 258 times) followed by the letter "e" ( 174 times). They received codes \(100_{2}\) and \(1110_{2}\), each of which is shorter than the eight bits used to encode each in the original file. The character " 8 " appears just once and gets a long encoding, \(11011000001_{2}\). The compressed data consists of a concatenation of all of the codes. So "e e" would be encoded \(11101001110_{2}\). The encoded data does not contain any separators between the pieces. To decode it one needs to first assume the code is one bit long, see if such a code exists, if not try two bits, and so on. So for the example one would first look for a code for \(1_{2}\). If it didn't exist (and it shouldn't) one checks for \(11_{2}\), which also shouldn't exist, neither does \(111_{2}\) (the third try). One the fourth try we look for \(1110_{2}\) and find that this is a code and the value is "e". The de-coding can continue by trying \(1_{2}\), \(10_{2}\), and finally \(100_{2}\) which is the code for a space.

\section*{Huffman Huff Tree Format for This Assignment}

This assignment will use a format in which text is compressed into three arrays, the compressed text, starting at huff_compressed_text_start, a dictionary of strings, starting at huff_dictionary, and the Huff Tree (a lookup tree), at huff_tree.

The compressed text is a long bit vector. As with Homework 1, bits are numbered in big-endian order. The compressed text is specified using words but of course can be read using other sizes. The dictionary of strings consists of a bunch of null-terminated strings. The Huff Tree is used to decode compressed pieces. It is traversed using bits of the compressed text ( 0 for left child, 1 for right child) and a leaf provides either an index into the dictionary or a character.

Consider the following excerpt from the homework file:
huff_compressed_text_start:
.word 0xd9ac96d8, 0x10b75d4f, 0xa06510d1, 0x7d9961e3, 0xeb6f31f1
```


# Encoding: .word BIT_START, BIT_END, TREE_POS, DICT_POS, FRAG_LENGTH

huff_debug_samples:

# 0: 0 11011 -> "\n"

    .word 0, 5, 0x2ee, 0xa, 1;
    
# 0: 5 001101 -> " ."

    .word 5, 11, 0x32, 0x16, 9;
    
# 0:11 0110010010 -> "text"

    .word 11, 21, 0x110, 0x27b, 4;
    
# 0:21 11011 -> "\n"

    .word 21, 26, 0x2ee, 0xa, 1;
    
# 0:26 011000000 -> "histo"

    .word 26, 35, Oxea, Oxce, 5;
    
# 1: 3 1000010 -> ":\n"

    .word 35, 42, 0x1d6, 0x2e, 2;
    ```

The compressed text is shown as 32 -bit words, under huff_compressed_text_start and as an aid in debugging, the start of the same text is shown under huff_debug_samples. The first piece, \(11011_{2}\), encodes a line feed character (we can see that by looking at the comment). The second piece, \(001101_{2}\), encodes " ." (spaces followed by a period). The first piece is in bits 0 to 4 (inclusive) of the compressed text, and the second piece is in bits 5 to 10. The hexadecimal digits of the compressed text can be found by concatenating the compressed pieces and then grouping them into four-bit hex digits: \(110110011010110010010 \rightarrow 110110011010110010010 \rightarrow \mathrm{~d} 9\) a c 9 ?. That matches the start of the compressed text shown under huff_compressed_text_start.

For this assignment a piece, for example 11011, is decoded by traversing the huff_tree. Each node in the huff_tree is 16 bits and can be one of three possible kinds: A leaf encoding a character, a leaf encoding a dictionary entry, or an internal node (with a left and right child). If the value of a node is \(<128\) it is a leaf encoding a character. Otherwise if the value of a node is \(>=0 \times 7000\) it is a leaf encoding a dictionary entry. Otherwise it is an internal node.
```

huff_tree:

# Huffman Lookup Tree

# 

huff_tree: \# Note: Most entries omitted.
.half 0x01fa \# Tree Idx 0 Pointer to right child.
.half 0x011d \# Tree Idx 1 0 Pointer to right child.

# [Many entries not shown.]

    .half 0x028c # Tree Idx 378 1 Pointer to right child
    
# [Many entries not shown.]

    .half 0x02e2 # Tree Idx 524 11 Pointer to right child.
    .half 0x02c7 # Tree Idx 525 110 Pointer to right child.
    
# [Many entries not shown.]

    .half 0x02e1 # Tree Idx 583 1101 Pointer to right child.
    
# [Many entries not shown.]

    .half 0x000a # Tree Idx 609 11011 Literal "\n"
    ```

The Huff Tree is an array of nodes, each a 16 -bit value. Let \(T\) denote such an array. The root is \(T[0]\). Let \(i\) indicate some position in the tree and \(n=T[i]\) denote the node at position \(i\). The assembler data above shows some elements of a Huff Tree. (The entire tree can be found in hw02.s.) The numbers in binary (following the Tree Idx) show the path to that node.

If \(n<128\) it is a leaf node encoding a character, and the ASCII value is \(n\). If \(n \geq 7000_{16}\) then the node is a leaf encoding a dictionary entry. The address of the first character of the dictionary entry is huff_dictionary + n - 0x7000. The strings in the dictionary are null-terminated.

Let \(n=T[i]\) be a non-leaf node, so that \(n \geq 128\) and \(n<7000_{16}\). Its left child is at \(T[i+1]\) and its right child is at \(T[n-128]\).

Here is how piece 11011 of the compressed text would be decoded based on the data in the example above. Start at the root, retrieving \(T[0]\). The value is \(1 f a_{16}\), which is an internal node. The first bit of 11011 is 1 so we traverse the right child which is at \(1 f a_{16}-80_{16}=17 a_{16}=378\). The entry at tree index 378 (based on the table) is \(28 c_{16}\) which again is an internal node. The second bit of the piece is 1 so we compute the index of the right child: \(28 c_{16}-80_{16}=20 c_{16}=524\). The next compressed bit is zero so we proceed to the left child, at index \(524+1\). The tree excerpt above includes the entry leading to the leaf node.

The routine below (which can be found in huff-decode.cc in the homework package) decodes the piece starting at bit bit_offset and writes the decoded piece at dcd_ptr.
```

void
hdecode(HData\& hd, int\& bit_offset, char*\& dcd_ptr)
{
// Decode one piece, starting at bit position bit_offset and
// write decoded piece starting at dcd_ptr.
// hd.huff_compressed: Compressed text. An array of 32-bit values.
// hd.huff_tree: A tree used to decode the compressed pieces.
// hd.huff_dictionary: Decompressed pieces.
// Start lookup at root of Huffman tree (tree_idx = 0).
//
int tree_idx = 0;
while ( true )
{
// Retrieve node.
uint16_t node = hd.huff_tree[tree_idx];
if ( node < 128 )
{
// Node is a leaf encoding a character.
char c = node; // Node value is an ASCII character.
*dcd_ptr++ = c; // Write character to decoded text pointer ..
return; // .. and return.
}
else if ( node >= 0x7000 )
{
// Node is a leaf holding an index into the dictionary.
// Compute dictionary index.
int idx = node - 0x7000;

```
```

            // Compute address of first character of dictionary entry.
            char* str = hd.huff_dictionary + idx;
            // Copy the dictionary entry.
            while ( *str ) *dcd_ptr++ = *str++;
            return;
        }
        else
            {
            // Node is not a leaf, need to set tree_idx to the index of
            // either the left or right child of the node. The left
            // child is used if the next bit of compressed text is zero
            // and the right child is used if the next bit of compressed
            // text is 1.
            // Get the next bit of compressed text.
            //
            int comp_idx = bit_offset / 32; // Index of word in huff_compressed.
            int bit_idx = bit_offset % 32; // Index of bit. MSB is 0.
            uint32_t comp_word = hd.huff_compressed[ comp_idx ];
            // Move needed bit to LSB in a way that sets other bits to zero.
            bool bit = comp_word << bit_idx >> 31;
            bit_offset++;
            if ( bit )
            {
                    // Set tree_idx to index of the right child.
                    tree_idx = node - 128;
            }
        else
            {
            // Set tree_idx to index of the left child.
            tree_idx++;
            }
        }
    }
    }

```

\section*{Homework Package}

The homework package consists of files to help with your solution and to satisfy curiosity. Your solution, of course, goes in \(\mathrm{hw} 02 . \mathrm{s}\), which is in the usual SPIM assembler format for this class.

The Huffman compression was performed by the huff perlscript. To compress MYFILE invoke it using ./huff MYFILE. With no arguments it compresses itself. It will write two files, encoded.s and encoded.h. The contents of encoded.s could be copied into the hw02.s (replacing what's there). Do this if you'd like to run your code on some other input.

File huff-decode.cc is a C++ routine that includes encoded.h and decodes it. It needs to
be re-built for each new input file. (Sorry, I've already spent too much time on the assignment.) Here is how it might be used on a new file:
```

[koppel@dmk-laptop hw02]\$ ./huff ../../hw02.tex
File ../../hw02.tex
Words 366 Codes 366 Resorts 11
[koppel@dmk-laptop hw02]\$ gmake -j 4
g++ --std=c++17 -Wall -g huff-decode.cc -o huff-decode
[koppel@dmk-laptop hw02]\$ ./huff-decode
Decoded:
\magnification 1095
% TeXize-on
\input r/notes

```

The assignment was created by compressing histo-bare.s.

Problem 1: Complete routine hdecode so that it decodes the piece of Huffman-compressed text starting at bit number a1, writes the decoded text to memory starting at the address in a2, and sets registers v0, v1, a1, and a2 as described below. (Yes, a0 is unused.)

Use symbol huff_compressed_text_start for the address of the start of the compressed text, huff_tree for the address of the start of the Huff Tree, and huff_dictionary for the address of the start of the dictionary.

When hdecode returns set v0, v1, a1, and a2 as follows. Set a1 to the next bit position to use. For example, if the compressed piece were 3 bits and hdecode were called with a1=100 then when hdecode returns a1 should be set to 103 . Set a2 to the address at which to write the next decoded character. For example, if the decoded text is 9 characters (not including the null) and initially a2 \(=0 \times 1000\) then when hdecode returns a2 should be set to \(0 \times 1009\). When hdecode returns v0 should be set to the address of the leaf in the Huff Tree that was used and v1 should be set to either the address of the dictionary entry used or the value of the character.

Note that the return values of a1 and a2 are useful because they are at the values needed to call hdecode again for the next piece. The return values of v0 and v1 are for debugging.

When hw02.s is run hdecode will be called multiple times, the return values checked, and the results printed on the console. It will be called for the first 200 pieces, or until there are three errors, whichever is sooner. A tally of errors is printed at the end, followed by the decoded text.

Pay attention to the error messages. Once syntax and execution errors are fixed, debug your code by tracing. To trace start the simulator using Ctrl F9 if running graphically or just F9 non-graphically. At the spim prompt type step 50 to execute the next 50 instructions. The trace shows line numbers of source assembly to the right of the semicolon. It also shows changed register values.

Single stepping is most useful when the first piece fails, which is likely to happen at first. But before long it will be correct and so viewing the trace will be a pain. To have the testbench start at your erroneous piece first locate the piece after label huff_debug_samples. The first number after . word is the Bit Position referred to in the "Decoding of.." message. Copy that line (perhaps with the comment above it) to just below the label huff_debug_samples.

Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Note: The following problem was assigned in each of the last five years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```


# Cycle }\quad0\begin{array}{llllllllll}{1}\&{2}\&{3}\&{4}\&{5}\&{6}\&{7}

lw r2, 0(r4)
add r1, r2, r7
IF ID EX ME WB
IF ID EX ME WB

```
(b) Explain error and show correct execution.
```


# Cycle }\quad0\quad

add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB

```
(c) Explain error and show correct execution.
```


# Cycle }\quad0\quad

add r1, r2, r3 IF ID EX ME WB
SW r1, O(r4) IF ID -> EX ME WB

```
(d) Explain error and show correct execution.
```


# Cycle }0

add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB

```

Problem 2: Appearing below are incorrect executions on the illustrated implementation. Notice that this implementation is different than the one from the previous problem. For each execution explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```


# Cycle

0
lw r2, 0(r4)
IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB

```
(b) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3
IF ID EX ME WB
lw r1, O(r4)
IF ID -> EX ME WB

```
(c) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3
sw r1, O(r4)
IF ID EX ME WB
IF ID -> EX ME WB

```
(d) Explain error and show correct execution. Note that this execution differs from the one from the previous problem.
```


# Cycle

0
add r1, r2, r3
IF ID EX ME WB
xor r4, r1, r5 IF ID ----> EX ME WB

```

Problem 3: Appearing below is the labeled MIPS implementation from 2018 Midterm Exam Problem 2(b), and as in that problem each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.

(a) Use F0.
(b) Use F0 and C3 at the same time. The code should not suffer a stall.
(c) Explain why its impossible to use E0 and D0 at the same time.

Problem 4: This problem appeared as Problem 2c on the 2020 final exam. Appearing below is our bypassed, pipelined implementation followed by a code fragment.

It might be helpful to look at Spring 2019 Midterm Exam Problem 4a. That problems asks for the execution of a loop and for a performance measure based upon how fast that loop executes.
(a) Show the execution of the code below on the illustrated implementation up to the point where the first instruction, addi \(\mathrm{r} 2, \mathrm{r} 2,16\), reaches WB in the second iteration.
(b) Based on your execution determine how many cycles it will take to complete \(n\) iterations of the loop.

```

LOOP:
addi r2, r2, 16
lw r1, 8(r2)
sw r1, 12(r3)
bne r3, r4, LOOP
addi r3, r3, 32
sub r10, r3, r2

```

Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Recall that code for the solution to Homework 2 included a loop that traversed a tree. The decision on whether to descend to the left or right child of a node was based on the next bit of compressed text. Several instructions were devoted to testing that next bit, and to checking whether a new word of bits needed to be loaded. In this assignment we are going to add a new instruction, bnbb (branch next bit big-endian), to MIPS that will allow such code to be written with fewer instructions.

Instruction bnbb rV, rP, TARG0, TARG1 works as follows. Register rV holds a bit vector, and register \(r \mathrm{P}\) holds a position in the bit vector. (A bit vector is just a number, but it's called a bit vector when we are interested in examining specific bits in the number's binary representation.) If the value of \(r P\) is 0 then it refers to the MSB of \(r V\), if the value of \(r P\) is 1 it refers to position 1 (to the right of the MSB), etc. Let pos refer to bits 5:0 of rP. If pos is in the range 0 to 31 (inclusive) then bnbb will be taken, otherwise (values from 32 to 63 ) bnbb is not taken. When bnbb is taken it will branch to TARGO if bit pos in rV is 0 and to TARG1 if bit pos in \(r V\) is 1. Regardless of whether bnbb is taken register \(r P\) is written with \(r P+1\). See the code and comments below:
```


# With sample values below bnbb is taken to LCHILD since bit 30 of 0x5 is zero.

# \$t8 = 0x5 (bit vector), \$t9 = 30 (pos)

bnbb \$t8, \$t9, LCHILD, RCHILD
addi \$v0, \$t9, 0 \# Delay slot insn. Here t9 is 31.

# This code is only executed when \$t9 in range 32-63 before bnbb executes.

# Fall through. Updates t8 and t9

addi \$t6, \$t6, 4 \# Update address ..
lw $t8, 0($t6) \# No more bits, load a new word.
addi \$t9, \$0, 0

```

The bnbb instruction can be used to eliminate at least two instructions in the hw02 solution. First, there would no longer be a need to shift the bit vector (the sll \$t8, \$t8, 1 instruction). Instead, the bnbb instruction would automatically increment a bit position register. Also, there would no longer be a need for a second branch to check whether all 32 bits in the bit vector were examined. (That was the bne \(\$ \mathrm{a} 1\), \(\$ \mathrm{t} 9\), EXAMINE_NEXT_BIT instruction.)

In the subproblems below complete the specification for bnbb and show hardware to implement it.
An Inkscape SVG version of the hardware diagram can be found at https://www.ece.lsu.edu/ee4720/2021/hw04-br-3way.svg.
(a) The description above leaves out a few details. In this problem fill them in. It may be helpful to attempt a solution to the next parts before answering this part.

Show a possible encoding for bnbb. That possible encoding must be based on format I. Show how the two targets are specified and and whether rV is encoded in the rt or rs fields.
(b) For bbnb to work correctly the \(r P\) register value needs to be incremented. It would be nice if an existing ALU operation could do that. Explain why the add operation, used for the add, addi, lw, and other instructions, would not work.
(c) The diagram below shows a five-stage MIPS implementation including some branch hardware. Also shown is logic to detect the bnbb instruction and two placeholder wires, bnbb-t0-taken and bnbb-t1-taken. Wire bnbb-t0-taken should be set to 1 if there is a bnbb in the ID stage and it should be taken to TARGO. The definition of bnbb-t1-taken is similar. If there is not a bnbb in ID or if there is and it's not taken, then both wires should be 0 .

In this problem design the logic to drive those wires. (The solution to this and the following problem can be done on the same diagram, or on separate diagrams.)
\begin{tabular}{l} 
bnbb-t0-taken \\
bnbb-t1-taken \\
\hline
\end{tabular}

(d) Modify the hardware below so that when bnbb-t0-taken is 1 target TARGO is used and when bnbb-t1taken is 1 target TARG1 is used. Follow the points below.
- Design for lower cost rather than higher performance.
- There is an unused input on the PC mux. That can be used, but does not have to be used.
- As always, hardware must be reasonably efficient.
- As always, do not break other instructions.


Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Look over SPECcpu2017 run and reporting rules, available at http://www.spec.org/cpu2017/Docs/runrules.html. Start with Sections 1.1 to 1.5 and read other sections as needed to answer the questions below.
(a) Section 1.2.3 of the run and reporting rules list several assumptions about the tester.

Consider the following testing scenario: The SUT (system being benchmarked) is a new product and that the tester works for the company that developed it. The company spent lots of money developing the product and their potential customers will use SPECcpu2017 when making buying decisions.
Explain why assumptions b and c seem reasonable given the testing scenario above.

Explain why assumption d also seems reasonable, given other stipulations set forth in the run and reporting rules (and discussed in class).
(b) The SPECcpu benchmarks can be prepared at base and peak tuning levels (or builds). These are described in Section 1.5. Section 2.3 .1 stipulates that base optimizations are expected to be safe.

What is an unsafe optimization? (Points deducted for irrelevant or lengthy answers, especially if they appear copied.)

Does that mean peak optimizations are unsafe? Does that mean peak results can be obtained with unsafe, don't-try-this-at-home optimizations?Why would it be bad if peak results were obtained with unsafe optimizations?
\(\square\) What rules ensure that optimizations used to obtain peak results aren't too unsafe?

Problem 2: The illustration below is our familiar 5-stage MIPS implementation with the destination register mux and an immediate mux shown. Modify it so that it is consistent with the RISC-V RV32I version as described below. The modifications should include datapath and labels, but not control logic. For this problem use RISC-V specification 20191213 available at https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

The Inkscape SVG source for the image is at
https://www.ece.lsu.edu/ee4720/2021/hw05-mips-id-mux.svg. It can be edited with your favorite SVG or plain text editor.

In some ways RISC-V is similar to MIPS, but there are differences. Pay attention to the encoding of the store instructions. Also pay attention to how branch and jump targets are computed.

Be sure to change the following:
Bit ranges at the register file inputs.
The bit ranges used to extract the immediate.
The bit ranges used for the offsets of branch and jump instructions and the hardware used to compute branch and jump targets.

The inputs to the destination register mux (which connects to the dst pipeline latch).
The names used in the pipeline latches.
Add or remove unneeded pipeline latches. (Such changes will be needed for branches and jumps.)
Consider the following instructions:
Two-register and immediate arithmetic instructions, such as add and addi.
The lui instruction (which is similar but not identical to MIPS' lui).
Branch instructions as well as jal and jalr.
The load and store instructions. (Only the store instructions will require a change beyond what is required for arithmetic instructions.)

Note:
- Do not show control logic such as logic driving mux select inputs.
- Do not show the logic to decide whether a branch is taken.

SVG source at https://www.ece.lsu.edu/ee4720/2021/hw05-mips-id-mux.svg.


Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Solve 2020 final exam Problem 2 a and b, in which the execution of code on a MIPS FP implementation is to be shown. Note that the FP adder in this implementation is different than the FP MIPS pipeline used in the classroom examples.

Problem 2: Solve 2020 final exam Problem 2d and 2e, in which the execution of code on a MIPS superscalar implementation is to be shown.

Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Solve 2020 final exam Problem 1 in which a 2-way superscalar implementation is to be improved using the so-called second-chance ALU.

Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Solve 2020 final exam Problem 3a in which a system using a local and a bimodal predictor is to be examined. See past exams and their solutions for similar problems.

Problem 2: Solve 2020 final exam Problem 3b which asks some easy-if-you-understand-things questions about hardware implementing a local predictor.

Problem 3: Solve 2020 final exam Problem 4c, in which contrasts between two machines with equal floatingpoint are to be found.

Problem 4: Solve 2020 final exam Problem 4f, a pandemic-themed short answer question.

\section*{4 Spring 2020}

WARNING: Problem 3 may be the hardest.
Problem 1: Note: The following problem was assigned in each of the last four years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```


# Cycle }\quad0\begin{array}{llllllllll}{1}\&{2}\&{3}\&{4}\&{5}\&{6}\&{7}

lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB

```
(b) Explain error and show correct execution.
```


# Cycle }\quad0\quad

add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB

```
(c) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3
sw r1, O(r4)
IF ID -> EX ME WB

```
(d) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3
IF ID EX ME WB
xor r4, r1, r5
IF ----> ID EX ME WB

```

Problem 2: Appearing below is the labeled MIPS implementation from 2018 Midterm Exam Problem 2(b), and as in that problem each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.

(a) Use F0.
(b) Use F0 and C3 at the same time. The code should not suffer a stall.
(c) Explain why its impossible to use E0 and D0 at the same time.

Problem 3: Solve 2019 final exam Problem 5a, which asks that two MIPS assembly language routines be re-written to be correct given an accompanying C routine.

Problem 1: The illustration below (and on the next page, there's no need to squint) shows our 5-stage MIPS implementation with some new hardware including: a Y1 unit in EX and a Y2 unit in ME. These are the two stages of a pipelined integer multiplication unit. They are to be used to implement a MIPS32 mul instruction (not to be confused with a MIPS-I mult instruction). The mul instruction executes as you would expect it to, for example mul r1, r2, r3 writes r1 with the product of r2 and r3. Because of the need to reduce (add together) all of the partial products, the multiplication hardware spans two stages, in contrast to an integer add which is one in one stage (in the ALU of course). Note: The mult instruction was the subject of 2013 Homework 4.

Here is how mul should execute:
```


# Cycle 0

    add R1, r2, r3 IF ID EX ME WB
    mul r4, R1, r5 IF ID Y1 Y2 WB
    mul R6, r7, R1 IF ID Y1 Y2 WB
    sub r8, R6, r9 IF ID -> EX ME WB
    
# Cycle }\quad0\quad

```

First of all, notice that there is no problem overlapping the two multiplies. Also notice that there is no problem bypassing a value to the source of a multiply.

(a) Add datapath hardware so that the multiply can execute as shown above.
- Assume that the Y1 and Y2 units each have about two multiplexor delays of slack. (Meaning if the path into the inputs of Y1 or out of the output of Y2 passes through more than two multiplexors the clock period would have to be increased, and we don't want that.)
- Pay attention to cost. Assume that the cost of one pipeline latch bit is the same as two multiplexor bits. Make other reasonable cost assumptions.
- Do not lengthen the critical path.
- Make sure that the code fragment above will execute as shown.
- Don't break other instructions.
(b) Add control logic for the existing WB-stage multiplexor and for any new multiplexors you might have added. Hint: This problem is easy, especially if you use two-input muxen.
- Use a pipeline execution diagram (such as the one above) to make sure that the value computed for a multiplexor select signal is the correct value when it is used, perhaps several stages later.
(c) At the lower-right is a big OR gate, its output is labeled STALL. Add an input to that OR gate which will be one when an instruction must stall due to a dependency with a mul. The sub from the execution above suffers such a stall.
(Not interesting enough? There is another problem on the next page!) Use this page for the solution or download illustration Inkscape SVG source from https://www.ece.lsu.edu/ee4720/2020/hw02-p1.svg and use that one way or another to prepare a solution.


Problem 2: Though two stages (Y1 and Y2) may be necessary to compute the product of arbitrary 32-bit signed integers, there are special cases that can be computed in less time, for example when either operand is zero or one.

If the \(Y\) units compute the product then it doesn't matter what operation the ALU is set to, but to handle special case(s) suppose that the control logic set the ALU operation to bitwise AND when decoding a mul instruction. In that case the output of the ALU would be correct for some multiplication operations and so the product would be ready in time to bypass to the next instruction. Add control logic to detect such situations and suppress the stall when present. Don't design the logic to set the ALU operation itself, we'll leave that to the Magic Cloud [tm].

Homework 3
Due: 30 March 2020

Please E-mail solutions of this assignment to koppel@ece.lsu.edu by the evening of the due date. PDF files are preferred. These can be generated by scanning software that you might have installed with a multifunction printer. A PDF can also be assembled from photos of a hand-completed copy. The disorganized homework penalty will be ignored for the remainder of the semester (unless we return early) so an E-mail with multiple image attachments will be accepted without penalty. Do not physically mail them to my office address, I will not be able to pick them up.

Problem 1: Appearing below is the solution to Homework 2 with labels added to some wires, which is followed by an execution of the code showing values on those labeled wires. The execution is based on the code fragment shown plus nop instructions before the first instruction (addi) and after the last instructions (nop).

\begin{tabular}{llllllll} 
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
addi R2, r0, 0 & IF & ID & EX & ME & WB & & \\
mul R1, R2, r3 & & IF & ID & EX & ME & WB & \\
add r4, R2, R1 & & & IF & ID & EX & ME & WB \\
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & & 0 & 1 & 1 & & & \\
B & & 0 & 1 & 0 & & & \\
C & 0 & 1 & 1 & 1 & & & \\
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
D & & & 0 & 1 & 0 & & \\
E & & 0 & 1 & 1 & 1 & & \\
F & 0 & 1 & 2 & 0 & 0 & & \\
\# Cycle & & 2 & 1 & 4 & 4 & 5 & 6 \\
G & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
H Cycle & 0 & & & & & & \\
\# Cycle & & & & &
\end{tabular}
(a) Refer to the table on the previous page for this problem. Notice that the value in the B row (above) in cycle 1 is 0 . According to the problem statement the instruction before addi is a nop.

Why would that value be 0 regardless of what instruction came before addi?

Suppose the addi \(\mathrm{r} 2, \mathrm{r} 0,0\) were changed to addi \(\mathrm{r} 2, \mathrm{r} 7,0\). Why would the value in the B row still be 0 ?
(b) Appearing below is a different code fragment. Complete the table so that it shows the values on the labeled wires.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline ori r2, r6, 7 & IF & ID & EX & ME & WB & & & \\
\hline sub r1, r2, r2 & & IF & ID & EX & ME & WB & & \\
\hline mul r3, r8, r1 & & & IF & ID & EX & ME & WB & \\
\hline mul r5, r3, r4 & & & & IF & ID & EX & ME & WB \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline A & & & & & & & & \\
\hline B & & & & & & & & \\
\hline C & & & & & & & & \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline D & & & & & & & & \\
\hline E & & & & & & & & \\
\hline F & & & & & & & & \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline G & & & & & & & & \\
\hline H & & & & & & & & \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{tabular}
(c) Appearing below are completed tables, but without a code fragment. Show a code fragment that could have produced those table values.

The originally assigned problem contained an error which made it difficult to solve. Shown below is the originally table, followed by the intended table. The solution uses the intended table.


Problem 2: Appearing below and on the next page is the solution to Homework 2 Problem 1. In this problem add hardware to handle a different and less special multiplication special case. Suppose that the middle output of the Y1 stage of the multiplier held the correct product whenever the high 24 bits of its b input are zero. For example, when b is 1,5 , or 255 . Call such values small. In all cases the correct product appears at the output of Y2.

Note: All outputs of Y1 arrive with zero slack, even the center output with the small b special case. That means that nothing can be done with these values until the next clock cycle, at least without reducing the clock frequency.

(a) Add hardware to bypass the product to the ALU and to the rtv mux when b is small. (There is a larger diagram on the next page.) The bypass should allow the first code fragment below to execute without a stall.
(b) Add control logic to suppress the stall when it is possible to bypass.

In the first code fragment below the stall is avoided because the \(b\) value (which is the \(r t v\) ) is small, in the second it is too large.
```


# Cycle }0

addi r1, r0, 23 IF ID EX ME WB
mul r2, r3, r1 IF ID EX ME WB
sub r4, r2, r5 IF ID EX ME WB

# Cycle }\quad

addi r1, r0, 300 IF ID EX ME WB
mul r2, r3, r1 IF ID EX ME WB
sub r4, r2, r5 IF ID -> EX ME WB

```
- Make sure that the changes don't break existing instructions.
- As always avoid costly solutions.
- As always pay attention to critical path.

The SVG source for the illustration below is at https://www.ece.lsu.edu/ee4720/2020/hw03-p2.svg. It can be edited using Inkscape or any other SVG editor, or (not recommended) a text editor.


It's up to all of us: \(\quad r>2 \mathrm{~m} \Rightarrow R_{e}<1 \quad\) where \(r\) is the radius of the largest circle with you at the center and containing only people in your household, and \(R_{e}\) is the effective reproduction number, the number of people infected by an infected person.

Problem 1: Appearing below is the code fragment from Homework 3.
\begin{tabular}{llllllll} 
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
addi R2, r0, 0 & IF & ID & EX & ME & WB & & \\
mul R1, R2, r3 & & IF & ID & EX & ME & WB & \\
add r4, R2, R1 & & & IF & ID & EX & ME & WB \\
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6
\end{tabular}
(a) Does this code fragment look like it was compiled with optimization on?

If your answer is something like "yes, it could be part of optimized code" then explain why you think it so and provide any missing context. (Do not change or re-arrange the three instructions above.)

If your answer is something like "no, it does not appear optimized" then show what the code would look like after optimization. Hint: A correct answer can start with either "Yes it does" or "No it doesn't". The "No" answer is straightforward.

Problem 2: MIPS does not appear to have a muli instruction.
(a) Comment on the following:

MIPS has a mul instruction but does not have a muli instruction because, as the solution to Homework 2 shows, the additional hardware for muli (beyond that used for mul) would be too costly.

Is the statement above reasonable or unreasonable? Explain.
(b) Show the encoding of MIPS instruction mul r1, r2, r3. Show all 32 bits of the instruction, divided into fields (each field can be shown in the radix of your choice). (The MIPS ISA manuals are linked to the course Web page. Instruction encodings are in Volume II.)
(c) Some possible reasons that there is no muli instruction in MIPS is that either there are no Format-I opcodes available (they are all used by other instructions) or that the few remaining opcodes are being kept in reserve for a better instruction than a muli.

Based on the MIPS Architecture Manuals (they are linked to the course references page) how many opcodes are available for new Format-I instructions? The easy way to solve this is to find the right table. The hard way to solve this is to go through the 144 or so pages of instruction descriptions. Hint: Look in volume I.

Problem 3: Perhaps you saw this coming: Time to add muli to MIPS.
(a) Show how a Format-R muli instruction with a ten-bit immediate might be defined using unused fields in the Format-R encoding. Make up your own function field value, but try to pick one that's unused. (See the previous problem.) Show how muli r1, r2, 43 might be encoded for your muli definition.
(b) Modify the hardware below (there's a copy on the next page) to implement this new instruction. The modified hardware should provide the immediate needed by muli. Show datapath but not control logic. Of course, any changes should not break existing instructions.

Pay attention to cost and performance. This can easily be solved by adding a mux in the ID stage. Hint: The solution is not much more than a mux. Be sure to carefully label the inputs.


The SVG source for the diagram below is available at https://www.ece.lsu.edu/ee4720/2020/hw03-p2.svg.


It's up to all of us: \(\quad r>2 \mathrm{~m} \Rightarrow R_{e}<1 \quad\) where \(r\) is the radius of the largest circle with you at the center and containing only people in your household, and \(R_{e}\) is the effective reproduction number, the number of people infected by an infectious person.

Problem 1: Solve 2019 Final Exam Problem 2, which asks for a pipeline execution diagram of FP code on our FP MIPS implementation, but with the comparison functional unit and floating-point condition code register added. For more information on the implementation of the floating-point compare instructions see 2018 Final Exam Problem 3. Please don't get confused about which problem to solve and which to use for background!

Problem 2: The following question appeared as Spring 2010 Homework 3 Problem 3, but in this ten-year anniversary version the solution must contain control logic for the multiplexors at the inputs to the A1 and A2 units. Try to initially solve it without looking at the solution, but use the solution if you get stuck.

Replace the fully pipelined adder in our FP pipeline (which appears on the next page) with one with an initiation interval of two and an operation latency of four. (The existing FP adder has an initiation interval of one and an operation latency of four.) See 2010 Homework 3 Problem 3 for more details.

Show datapath and control logic. Be sure to show control logic for the multiplexors at the inputs to A1 and A2, this control logic does not appear in the solution to the 2010 assignment. Hint: This additional control logic is really easy to do, it can be done just with wires, no gates!

An Inkscape SVG version of the MIPS implementation below can be found at https://www.ece.1su.edu/ee4720/2020/mpipei_fp_by.svg.


It's up to all of us: \(\quad r>2 \mathrm{~m} \Rightarrow R_{e}<1 \quad\) where \(r\) is the radius of the largest circle with you at the center and containing only people in your household, and \(R_{e}\) is the effective reproduction number, the number of people infected by an infectious person.

Problem 1: Solve 2017 Final Exam Problem 2, which asks PED of some code fragments on a 2-way superscalar MIPS implementation. The solution is available, but make every effort to solve it on your own. Use the posted solution only if you get stuck. Solving the 2017 problem will make the problem below easier.

Problem 2: Solve 2019 Final Exam Problem 1, including the bonus question (part d), which asks for datapath and control logic for a 2 -way superscalar implementation, some associated with a dependence leading to a sw instruction. Parts a and b ask for typical hardware. Part c is more interesting because the hardware is essentially avoiding a stall by skipping an instruction in a dependence chain. The dependence chain is or \(\rightarrow\) add \(\rightarrow\) sw and the skipped instruction is the or. Part d, the bonus question, asks whether this is worth it.
An Inkscape SVG version of the MIPS implementation from Problem 1 of the exam can be found at https://www.ece.lsu.edu/ee4720/2019/fe-fuse.svg.
\(\square\)

\section*{\(5 \quad\) Spring 2019}

Problem 1: Follow the instructions for class account setup and for homework workflow in https://www.ece.1su.edu/ee4720/proc.html. Review the comments in hw01.s and look for the area labeled "Problem 1."

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2019/hw01.s.html and the C ++11 code at https://www.ece.lsu.edu/ee4720/2019/hw01-equiv.cc.html. A good reference for \(\mathrm{C}++\) is https://en.cppreference.com/. For MIPS references see the course references page, https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in https://www.ece.lsu.edu/ee4720/proc.html.

File hw01.s in the assignment package is where your solution should go. Briefly, complete routine get_index so that it finds the position of a word in a list. It will use a hash table to speed things up if the same word is used twice. (Yes, you're going to code all that in assembler!) File hw01-equiv. cc contains C++11 code that does the same thing. In particular, routine App: :lookup does much of what get_index is supposed to do.

Routine get_index is called with four arguments. The first, a0, is the address of a C-style string, call this string the lookup word. Argument a1 is the address of a table of C-style strings, call this the word list. (See word_list_start in the code.) Argument a2 is the address of the end of the word list, and a3 holds the address of a 1024-byte area of memory called the hash table.

The routine is supposed to check if the lookup word is in the word list. When the routine returns register v0 should be set to 0 if the lookup word was not found in the word list, otherwise v0 should be set to the position in the word list. The first word (aardvark) is at position 1, the second is at position 2, etc. The word list contains one string after another. That is, the character after the null terminating aardvark is a, the first letter of the second word, ark. The word list layout used in the C++ code, App::lookup, is the same as the layout used in assembler, so inspecting the code in App: :lookup might help in understanding the layout of the word list.
(a) Add code to get_index so that it sets v0 as described above. The testbench should show zero Pos errors.
(b) Complete get_index so that it uses a hash table to speed lookups. Suppose that the number of words in the word list is large (even if it's small for this assignment), and also do not assume that the list is alphabetized. Note: That was the original plan.

Let \(n\) denote the number of bytes in the word list. The value of \(n\) is a2-a1 when get_index starts and always is word_list_end - word_list_start. The amount of work needed to determine whether the lookup word is in the word list is \(O(n)\). Remember, we don't like \(O(n)\), we much prefer \(O(\log n)\) or better yet \(O(1)\). As the alert students suspect, this is where the hash table comes in.

When first called, get_index should check whether there is an entry in the hash table for the lookup word. It should do so by computing a hash of the first three characters of the lookup word using the same hash function as App: :hash_func (shown below). Note that App: :hash_func returns a value in the range \(0-255\), call that the hash index.
```

int hash_func(const char* str)
{
return
( str[0] ^ ( str[1] >> 2 ) ^ ( str[1] << 6 ) ^
( str[2] << 4 ) ^ ( str[2] >> 4 ) ) \& hash_mask;

```
\}
Each entry in the hash table consists of two 16-bit values, see the Hash_Elt structure. The first 16 bits is an offset, the second is the pos.
```

struct Hash_Elt {
int16_t offset;
int16_t pos;
};

```

The pos member indicates the position of the word in the word list, or 0 if that hash entry has not been initialized. Member offset is the character offset in the word list. That is, a1 + offset is the address of the first character of the word.

The \(\mathrm{C}++11\) code below first computes the hash for the lookup word (word), retrieves the hash table entry, then checks to see if the entry is a hit. To be a hit the pos member must be non-zero and the word at offset must match word. (This is necessary because two different words can generate the same hash index.)
```

Lookup_Info lookup(const char *word)
{
// Look up word in hash table.
//
const int hash = hash_func(word);
Hash_Elt\& elt = hash_table[ hash ];
Lookup_Info li;
li.hash = hash;
// Return if hash table hit.
//
if ( elt.pos \&\& streq( \&words[elt.offset], word ) )
{
li.pos = elt.pos;
li.in_hash = true;
return li;
}

```

The li object has information about the match. In the assembler code the same information is provided in v0 and v1.

If there is a miss in the hash table, scan for the word. If it is in the word list then update the hash table so that the next time there will be a hash table hit. That is performed by the App: :lookup by writing the hash table element reference, elt, that it had retrieved.
```

// Scan word list to find word.
//
int offset = 0, pos = 0;
while ( words[offset] )
{
if ( streq( \&words[offset], word ) )
{

```
```

                // Word has been found, add to hash table and return index.
                elt.offset = offset;
            li.pos = elt.pos = pos;
            return li;
        }
        // Word not found, advance to next word. Assembler can be faster.
    offset += strlen( &words[offset] ) + 1;
        pos++;
    }
    // Word not in word list, return 0.
li.pos = -1;
return li;

```

When the the code in hw01.s is run a testbench will call get_index for multiple words. The console will show the position and hash returned for the word, an X to the right of the position indicates that it is wrong, and underscore indicates that it is correct. Two characters are shown to the left of the hash index (labeled Hash). The first character, X or _, indicates whether the hash table lookup result is correct, the second character indicates whether the hash index itself is correct. At the end there will be a tally of the total number of errors.

The solution to several of the problems in this assignment requires material about to be covered in class, in particular, stalling instructions to avoid hazards. For coverage of this material see slide set six, https://www.ece.lsu.edu/ee4720/2019/lsli06.pdf. For a solved problem see 2014 Homework 1 Problem 3. Feel free to look through old homework and exams for other similar problems, but when doing so make sure that the MIPS implementation matches the one in this problem: the muxen at the ALU inputs should each have just 2 inputs.

Problem 1: Note: The following problem was assigned in each of the last three years (though not in color), and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```

LOOP: \# Cycles 0 1 1 2 3 4 4 5 6 7
lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
LOOP: \# Cycles 0

```
(b) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB
LOOP: \# Cycles }0

```
(c) Explain error and show correct execution.
```

LOOP: \# Cycles }0
add r1, r2, r3 IF ID EX ME WB
Sw r1, O(r4)
IF ID -> EX ME WB
LOOP: \# Cycles

```
(d) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3
xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: \# Cycles }0

```

Problem 2: The MIPS code below is taken from the solution to 2018 Homework 1. Show the execution of this MIPS code on the illustrated implementation for two iterations. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be value being written. (In class we called such a register file internally bypassed.)
- Check carefully for dependencies.
- Focus on when the branch target is fetched and on when wrong-path instructions are squashed.
- Be sure to stall when necessary.

```

CLOOP:
lbu $t0, O($t4)
sb $t0, O($a1)
addi \$t4, \$t4, 1
bne \$t4, \$t5, CLOOP
addi \$a1, \$a1, 1

```

Before solving the branch hardware problem below it might be helpful to look at 2016 Homework 2.
Problem 1: The code below should suffer a stall on the illustrated implementation due to a dependency between the addi and bne instructions. The stall can be avoided by scheduling the loop, but lets consider a hardware solution for code fragments like this in which an addi \(r \mathrm{X}, \mathrm{rY}\), IMM is followed by a bne \(\mathrm{rX}, \mathrm{r} 0\), \(T\) or by a beq \(r X, r 0, T\).
```

LOOP:
addi r3, r3, -1
bne r3, r0, LOOP
lw r1, 4(r1)

```

One way to avoid the stall (which would work for more than just the cases outlined above) would be to have the ALU generate an =0 signal which, if the dependencies were right, could be used by the branch hardware. Alas, the ALU people are on vacation, so lets try something else.

As alert students may have realized by now, all the branch hardware has to do is check whether rY == -IMM, which is \(r 3==1\) in the example. The comparison itself can be done using the existing comparison logic. The challenge is delivering the operands to that logic at the right time.


Attention students who have forgotten how to use a pencil (or never learned): An Inkscape SVG version of the implementation can be found at https://www.ece.lsu.edu/ee4720/2019/mpipei3.svg.
(a) Add hardware to the implementation above to deliver the correct operands to the comparison unit so code fragments like the one above can execute without a stall.
- Pay attention to cost, including the number of bits in each wire used. (For example, don't add a second comparison unit.)
- The changes should not prevent other code from executing correctly. (For example, a branch such as beq r1,r2, T should execute correctly.)
- Don't overlook that rX and rY are not necessarily the same register.
(b) Add control logic to generate a BY signal which is set to logic 1 when the branch can use the bypass. The control logic must detect that the correct instructions (including the registers) are present.
(c) If the design above was done correctly the highest cost part is the logic handling the immediate. Show how the cost of that logic can be reduced while still retaining most (but not all) of the benefits of the full-cost design. Your argument should include examples of "typical" code. (Assume [actually assert] that your code samples are typical [reflects what is running by users most of the time]. Later in the semester we'll remove the scare-quotes from "typical".)

Problem 1: The three loops below (probably on the next page) copy an area of memory starting at the address in \(r 2\) to an area of memory starting at the address in \(r 3\). The number of bytes to copy is in r5.

(a) Show a pipeline execution diagram for each loop on the illustrated implementation.
(b) Compute the rate that each loop copies data in units of bytes per cycle. Base this on your execution diagrams.
(c) Loop A has a wasted delay slot and should suffer stalls. Schedule the code (re-arrange instructions) to fill the delay slot and minimize the number of stalls. Feel free to change instructions and to add new ones, though the loop should still copy one byte per iteration and should copy the data as described above.
(d) Loop A can be safely substituted for Loop C. That is, if a program calls Loop C then that call can be changed to a call of Loop A or B and the program will still work correctly. However, if a program calls Loop A, substituting B will not work. Explain why and show sample values for r2, r3, and r5 for which this is true.
(e) If a program calls Loop B substituting C will not work. Explain why and show sample values for r2, r3, and r 5 for which this is true.

Code on next page.
```


# Loop A

add r4, r3, r5
LOOP:
lb r1, O(r2)
sb r1, O(r3)
addi r2, r2, 1
addi r3, r3, 1
bne r3, r4, LOOP
nop

```
\# Loop B
add r4, r3, r5
LOOP:
    lw r1, O(r2)
    sw r1, O(r3)
    addi r2, r2, 4
    addi r3, r3, 4
    bne r3, r4, LOOP
    nop
\# Loop C
add r4, r3, r5
addi r4, r4, -8
LOOP:
    lw r1, O(r2)
    lw r10, 4(r2)
    sw r1, \(0(r 3)\)
    sw r10, 4(r3)
    addi r2, r2, 8
    bne r3, r4, LOOP
    addi r3, r3, 8

Problem 1: Solve Midterm Exam Problem 1b, in which code is to be completed using the \(\jmath^{\jmath}\) r instruction.
Problem 2: In the solution to Midterm Exam Problem 3 the Taken signal is set for a bne using the rsv and \(r t v\) values (from the register file) even if there is a dependence with an slt in the EX stage. Modify the logic to fix this.

For your solving convenience, the solution illustration appears on the next page and in Inkscape SVG at https://www.ece.lsu.edu/ee4720/2019/mt-p3-slt-bne-sol.svg


Problem 1: Solve 2018 Final Exam Problem 2, which asks for an analysis of FP code.
Problem 2: Solve 2018 Final Exam Problem 3, in which a comparison unit is to be added to our FP pipeline. The illustration appears on the next page. For those who prefer to prepare their solution using an image editing program the Inkscape SVG source for the image is available at https://www.ece.lsu.edu/ee4720/2018/fe-fp-cmp.svg.


Problem 1: Solve 2018 Final Exam Problem 1 (a) and (b), in which the execution of code on a 2-way superscalar MIPS implementation is to be shown. Note: do not solve part (c) for this assignment. Part (c) will be in the next assignment.

Problem 2: Solve 2018 Final Exam Problem 6, which are a collection of short-answer questions about interrupts.

Problem 1: The following problem is an enhanced version of 2018 Final Exam Problem 1 (c). Appearing below is our 2-way superscalar MIPS with ID-stage hardware to determine branch direction (near the top in blue) and ID-stage hardware to squash instructions (near the bottom in blue). The Inkscape SVG source for this image can be found at https://www.ece.lsu.edu/ee4720/2019/hw08-ss.svg.

There are two outputs of the branch direction hardware logic, indicating whether the respective ID-stage slot has a taken branch. For example, if Taken0 is 1 then there is a branch in slot 0 and that branch is taken. Of course, assume that this logic is correct.

There is a squash logic with two inputs at the bottom. If input Sq 0 is 1 then the instruction in ID-stage slot 0 will be squashed, likewise for Sq1.

In this implementation fetch groups are not aligned.

(a) When a branch is taken we may need to squash one or two instructions (the number of instructions to squash depends on the whether the branch is in slot 0 or slot 1). Design logic to set the Sq0 and Sq1 inputs so that appropriate instructions are squashed. It will be very helpful to draw pipeline execution diagrams showing a taken branch in slot 0 and slot 1 .Draw PEDs for the two cases.
Add hardware to set SQ signals.
(b) Notice that the branch hardware shown can only provide the target for a branch in slot 1. Add hardware for providing the branch target of a branch in slot 0 . Note that unlike the final exam, in this problem fetches are not aligned. That precludes the more efficient solution given in the final exam.

Do not add hardware for checking the branch condition. Show logic computing the select signals for any multiplexors you add, but do not show any other control logic. Note: In the original assignment the direction to show logic computing select signals was omitted.Add hardware for a slot-0 branch.Pay attention to cost.Be sure the hardware computes the correct target address. Think about the value of NPC (or related value) that's needed.


Problem 1: Solve 2018 Final Exam Problem 4, which asks for analysis of simple bimodal and local branch predictors and for the construction of a branch predictor. The Inkscape SVG source for the local branch predictor can be found at https://www.ece.lsu.edu/ee4720/2018/fe-p4-local-pred.svg

\section*{6 Spring 2018}

Problem 1: Follow the instructions for class account setup and for homework workflow. Review the comments in hw01.s and look for the area labeled "Problem 1," which has procedure named unsqw (unsquish) which initially just has simple placeholder code. Those who want to start before getting a class account the assembler for the entire assignment can be found at http://www.ece.lsu.edu/ee4720/2018/hw01.s.html.

Routine unsqw is called with two arguments. The first, in register a0, holds the address of a C-style string which will be called the input string. The second argument, in register a1, holds the address of a writeable area of memory, which will be called the output buffer.

The input string is an ASCII string that has been compressed using a simplified version of the the Lempel-Ziv LZ77 method. LZ77 compresses text by replacing a substring that has appeared before with a reference to its prior occurrence. The reference consists of the length of the prior occurrence and the distance (number of characters back) of the occurrence. For example, consider:

All work and no play makes Johnny a dull boy.
All work and no play makes Johnny a dull boy.
All work and no play makes Johnny a dull boy.
All work and no play makes Johnny a dull boy.
The compressed string might look something like this:
All work and no play makes Johnny a dull boy.
* Length: 46, Distance: 46 // Replace 46 chars starting 46 characters back.
* Length: 92, Distance: 92 // Replace 92 chars starting 92 characters back.

The first line consists of 46 characters (including the end-of-line character). The second line of the compressed string consists of a reference to the first line in the form of a length, 46 characters, and how far back to find the start of the first line, also 46 characters. This makes a copy of the first line. The third line of the compressed string consists of a reference to the first two lines. A reference to a prior occurrence of text can refer to anything that has already appeared, not just whole lines.

The size of the compressed string depends upon the size of a reference. Obviously it would be inefficient to encode a reference such as * Length: 46, Distance: 46 using regular ASCII text. For this assignment two methods of encoding the reference will be used, Simple and Better.

In both the Simple and Better methods the characters in the original string must be in the range \(1_{16}\) to \(7 f_{16}\) (inclusive). Character 0 is used to terminate a string and characters \(80_{16}\) and higher are used to mark the start of a reference.

In the Simple method a reference is a three-byte sequence: \(80_{16}, L, D\). The sequence always starts with \(80_{16}\). The second byte, \(L\), is the length of the sequence and the third byte, \(D\), is the distance, the number of characters back at which the duplicated text can be found. Both \(L\) and \(D\) are unsigned integers. Let \(p\) denote the number of uncompressed characters generated so far. Then reference \(80_{16}, L, D\) indicates that the uncompressed characters at positions \(p-D\) to \(p-D+L-1\) should be copied to the end of the string of uncompressed characters. Using this method the reference \(*\) Length: 46, Distance: 46 would be encoded as \(0 \times 800 \times 2 \mathrm{e} 0 \times 2 \mathrm{e}\). It is okay for \(L-D>0\).

To help with understanding how the compression works, the compressed string in hw01.s includes comments that show the length, distance, and the referenced text. For example, consider the assembly language source code below showing two strings, uncomp (uncompressed) and comp_simple:
```

uncomp: \# Uncompressed data.
.ascii "\nAll work and no play makes Johnny a dul" \# Idx: 0 - 39
.ascii "l boy.\nAll work and no play makes Johnny" \# Idx: 40 - 79
.ascii " a dull boy.\nAll work and no play makes " \# Idx: 80 - 119
.ascii "Johnny a dull boy.\nAll work and no play " \# Idx: 120 - 159
.asciiz "makes Johnny a dull boy.\n" \# Idx: 160 - 184
comp_simple: \# Compressed data Simple Method.
.ascii "\nAll work and no play makes Johnny a dul" \# Idx: 0 - 39
.ascii "l boy." \# Idx: 40 - 45
\# Idx: 0 = 46 - 46 = 0x2e - 0x2e. Len: 139 = 0x8b.
.byte 0x80 139 46 \# "\nAll work and no play makes Johnny a dull boy.\nAll work and
no play makes Johnny a dull boy.\nAll work and no play makes Johnny a dull boy.\n"
.byte 0

# Original: 185 B, Simple Compressed: 49 B, Ratio: 0.265

```

The references start with assembler directive .byte, other text starts with the directive .ascii. There are two comments associated with a reference. The comment above .byte shows the starting index, \(p-D\), of the text to be copied and the length, \(L\). The comment on the reference line shows the copied text.

The first line of the uncompressed text (uncomp) (index 0 to 45) appears literally in the compressed text (comp_simple). There is a single reference that indicates \(L=139\) characters should be copied starting at \(D=46\) characters back from position \(p=46\). Notice that this is a case where the text to be copied from overlaps the target text, but that works out well for us.

For more examples search for comp_simple in hw01.s.
A disadvantage of the Simple method is that the distance is limited to 255 bytes and that the 7 least-significant bits of the first byte (the \(0 \times 80\) ) are unused. The Better method fixes both problems.

Problem continued on the next page.

In the Better method a reference starts with a byte having bit 7 (the most-significant bit) set to 1 . There are four cases for a reference, the size of a reference ranges in size from 2 to 4 bytes. Call the first byte of a reference \(R\).

Case 1: \(R=80_{16}=10000000_{2}, L, D\). This is identical to the simple case. \(R\) is followed by two bytes, the first is the length \((L)\) and the second is distance \((D)\).

Case 2: \(R=10 \mathrm{ll} l l l l_{2}, D . R\) is followed by one byte \((D)\). The six least-significant bits of \(R\) are the length, and \(D\) is the distance.

Case 3: \(R=c 0_{16}=11000000_{2}, L, D_{h}, D_{l} . R\) is followed by three bytes, the first is the length \((L)\), the second, \(D_{h}\), holds bits 15 to 8 of the distance, and the third, \(D_{l}\), is the 8 last-significant bits (bits 7 to 0 ) of the distance.

Case 4: \(R=11 l l l l l l_{2}, D_{h}, D_{l} . R\) is followed two bytes, the first, \(D_{h}\), holds bits 15 to 8 of the distance, and the second, \(D_{l}\), is the 8 last-significant bits (bits 7 to 0 ) of the distance. The six least-significant bits of \(R\) hold the length.

Examples of the cases are shown below, these are taken from the assignment code in hw01.s.
```

uncomp: \# Uncompressed data.
.ascii "[Note: it has been cold cold cold cold!]" \# Idx: 0 - 39
.ascii "\n==========================================" \# Idx: 40 - 79
.ascii "=================================\nAnother " \# Idx: 80 - 119
.ascii "frigid Arctic airmass is already pushing" \# Idx: 120 - 159
.ascii " into the region\nand will provide bitter" \# Idx: 160 - 199
.ascii "ly cold temperatures. Temperatures will\n" \# Idx: 200 - 239
.ascii "plunge into the teens and 20s tonight an" \# Idx: 240 - 279
.ascii "d could be quite similar\nWednesday night" \# Idx: 280 - 319
.ascii ".\n========================================" \# Idx: 320 - 359
.ascii "===================================\n* TEMPE" \# Idx: 360 - 399
.ascii "RATURE...Lows will fall into the mid tee" \# Idx: 400 - 439
.ascii "ns to lower 20s\nalong and north of the I" \# Idx: 440 - 479
.ascii "-10/12 corridor. South of I-10 lows\nwill" \# Idx: 480 - 519
.ascii " range from 20 to 25. These temperatures" \# Idx: 520 - 559
.ascii " will be similar\nWednesday night.\n\n* DUR" \# Idx: 560 - 599
.ascii "ATION...Freezing conditions will likely " \# Idx: 600 - 639
.ascii "last for 12 to 26\nhours over much of the" \# Idx: 640 - 679
.ascii " warned area tonight and then 12 to 18\nh" \# Idx: 680 - 719
.asciiz "ours Wednesday night." \# Idx: 720 - 740
comp_better: \# Compressed data Better Method.
.ascii "[Note: it has been cold" \# Idx: 0 - 22

# Case 2: \# Idx: 18= 23- 5 = 0x17 - 0x5. Len: 15 = 0xf.

    .byte 0x8f 0x05 # " cold cold cold"
    .ascii "!]\n=" # Idx: 38-41
    
# Case 1: \# Idx: 41 = 42- 1 = 0x2a - 0x1. Len: 69 = 0x45.

    .byte 0x80 0x45 0x01 # "============================================================================"\
    
# Case 3 \# Idx: 40=321-281 = 0x141 - 0x119. Len: 72 = 0x48.

    .byte 0xc0 0x48 0x01 0x19 # "\n================================================================================="吅
    
# Case 4 \# Idx: 157=613-456 = 0x265-0x1c8. Len: 4 = 0x4.

    .byte 0xc4 0x01 0xc8 # "ing "
    ```

The assignment package includes a program, hw01-comp.cc that can be used to encode your own text stings, should you want to.

When the code in hw01.s is run a testbench will run routine unsqw twice, first on code compressed using the simple method and again on text compressed using the better method.

The output of the testbench starts with the uncompressed text (a known correct copy), followed by the results of running unsqw on the two compressed text versions. If there is an error it will show at what index (character position) the error occurs and will show the text before and just beyond this index from the output of unsqw and the known correct text. You may need to scroll up to see the beginning of the compressed text (use the arrow keys).
(a) Complete unsqw so that it uncompresses text compressed using the simple method.
(b) Complete unsqw so that it uncompresses text compressed using the better method.

The solution to several of the problems in this assignment requires material about to be covered in class, in particular, stalling instructions to avoid hazards. For coverage of this material see slide set six, http://www.ece.lsu.edu/ee4720/2018/lsli06.pdf. For a solved problem see 2014 Homework 1 Problem 3. Feel free to look through old homework and exams for other similar problems, but when doing so make sure that the MIPS implementation matches the one in this problem: the muxen at the ALU inputs should each have just 2 inputs.

Problem 1: Recall that in the unsqw program from Homework 1 there was a loop that had to copy the prior occurrence of a piece of text to the output buffer. That loop from the solution appears below, and again re-written to improve performance, at least that was the goal.
```


# Original Code -------------------------------------------------------------------------

    # Copy the prior occurrence of text from some part of the
    # output buffer to the end of the output buffer.
    # At this point in code:
    # Reference marker is in $t0.
    # Length is in register $t3.
    # Distance is in register $t4.
    #
    sub $t4, $a1, $t4 # Compute starting address of prior occurrence.
    add $t5, $t4, $t3 # Compute ending address of prior occurrence.
    addi $a0, $a0, 1
    COPY_LOOP:
lb $t0, O($t4) \# Load character of prior occurrence ..
sb $t0, O($a1) \# .. and write it to the end of the output buffer.
addi \$t4, \$t4, 1
bne \$t4, \$t5, COPY_LOOP
addi \$a1, \$a1, 1
j LOOP
nop

```
```


# Improved Code -----------------------------------------------------------------------------

    # Copy the prior occurrence of text to the end of the output buffer.
    # At this point in code:
    # Reference marker is in $t0.
    # Length is in register $t3.
    # Distance is in register $t4.
    #
    sub $t4, $a1, $t4 # Compute starting address of prior occurrence.
    addi $a0, $a0, 1
    # Round length (L) up to a multiple of 4.
    #
    addi $t7, $t3, 3
    andi $t8, $t7, Oxfffc # Note: this only works if L < 65536
    sub $t6, $t8, $t3
    #
    # At this point:
    # $t8: L', rounded-up length.
    # $t6: Amount added to L to round it up. That is, L' - L
    # t6 can be 0, 1, 2, or 3.
    # If t6 is 0, then L' = L;
    # if t6 is 1, then L' = L + 1; etc.
    # Decrement prior-occurrence and output-buffer pointers.
    #
    sub $t4, $t4, $t6
    sub $a1, $a1, $t6
    # Jump to one of the four lb instructions in the copy loop.
    #
    la $t7, COPY_LOOPd4 # Get address of first lb instruction.
    sll $t6, $t6, 3 # Compute offset to lb that we want to start at.
    add $t7, $t7, $t6 # Compute address of starting lb ..
    jr $t7 # .. and jump to it.
    add $t5, $t4, $t8 # Don't forget to compute stop address.
    COPY_LOOPd4:
lb $t0, 0($t4)
sb $t0, 0($a1)
lb $t0, 1($t4)
sb $t0, 1($a1)
lb $t0, 2($t4)
sb $t0, 2($a1)
lb $t0, 3($t4)
sb $t0, 3($a1)
addi \$t4, \$t4, 4
bne \$t4, \$t5, COPY_LOOPd4
addi \$a1, \$a1, 4
j LOOP
nop

```

Let \(L\) denote the length of the prior occurrence of text to copy.
(a) Determine the number of instructions executed by the original code in terms of \(L\). Include the copy loop and the instructions before it shown above. State any assumptions.
(b) Determine the number of instructions executed by the improved code in terms of \(L\). Include the copy
loop and the instructions before it shown above. State any assumptions.
(c) What is the minimum value of \(L\) for the improved method to actually be faster?
(d) What is it about the improved code that helps performance?

Problem 2: Note: The following problem is identical to 2016 Homework 1 Problem 1. Try to solve it without looking at the solution. Answer each MIPS code question below. Try to answer these by hand (without running code).
(a) Show the values assigned to registers t1 through t8 (the lines with the tail comment Val:) in the code below. Refer to the MIPS review notes and MIPS documentation for details.
```

    .data
    myarray:
.byte 0x10, 0x11, 0x12, 0x13
.byte 0x14, 0x15, 0x16, 0x17
.byte 0x18, 0x19, 0x1a, 0x1b
.byte 0x1c, 0x1d, 0x1e, 0x1f
.text
la \$s0, myarray \# Load \$s0 with the address of the first value above.
\# Show value retrieved by each load below.
lbu $t1, 0($s0) \# Val:
lbu $t2, 1($s0) \# Val:
lbu $t2, 5($s0) \# Val:
lhu $t3, 0($s0) \# Val:
lhu $t4, 2($s0) \# Val:
addi \$s1, \$0, 3
add \$s3, \$s0, \$s1
lbu $t5, 0($s3) \# Val:
sll \$s4, \$s1, 1
add \$s3, \$s0, \$s4
lhu $t6, 0($s3) \# Val:
sll \$s4, \$s1, 2
add \$s3, \$s0, \$s4
lhu $t7, 0($s3) \# Val:
lw $t8, O($s3) \# Val:

```
(b) The last two instructions in the code above load from the same address. Given the context, one of those instructions looks wrong. Identify the instruction and explain why it looks wrong. (Both instructions should execute correctly, but one looks like it's not what the programmer intended.)
(c) Explain why the following answer to the question above is wrong for the MIPS 32 code above: "The 1w instruction should be a lwu to be consistent with the others."

Problem 3: Note: The following problem was assigned in each of the last three years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```

LOOP: \# Cycles }0
lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
LOOP: \# Cycles

```
(b) Explain error and show correct execution.
```

LOOP: \# Cycles }0
add r1, r2, r3
lw r1, O(r4)
IF ID EX ME WB
IF ID -> EX ME WB
LOOP: \# Cycles
0

```
(c) Explain error and show correct execution.
```

LOOP: \# Cycles }0
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
IF ID -> EX ME WB
LOOP: \# Cycles
0

```
(d) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: \# Cycles }0

```

Problem 4: The MIPS code below is taken from the solution to 2018 Homework 1. Show the execution of this MIPS code on the illustrated implementation for two iterations. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be the value being written. (In class we called such a register file internally bypassed.)
- Check carefully for dependencies.
- Focus on when the branch target is fetched and on when wrong-path instructions are squashed.
- Be sure to stall when necessary.

```

CLOOP:
lbu $t0, O($t4)
sb $t0, O($a1)
addi \$t4, \$t4, 1
bne \$t4, \$t5, CLOOP
addi \$a1, \$a1, 1

```

Problem 1: Appearing below are two MIPS implementations, The First Implementation is taken from Homework 2 Problem 4. Branches suffer a two-cycle penalty on this implementation since they resolve in ME. On the The Second Implementation branches resolve in EX reducing the penalty to one cycle. For convenience for those using 2 -sided printers the same implementations are shown again on the next page.

The First (HW2 P4) Implementation


The Second Implementation



The code fragment below and its execution on The First Implementation is taken from the solution to Homework 2 Problem 4. Notice that the branch suffers a two-cycle branch penalty.

```

    lbu $t0, O($t4) IF ID EX ME WB
    sb $t0, 0($a1) IF ID ----> EX ME WB
    addi $t4, $t4, 1 IF ----> ID EX ME WB
    bne $t4, $t5, CLOOP IF ID ----> EX ME WB
    addi $a1, $a1, 1 IF ----> ID EX ME WB
    X1
X2
IF IDx
IFx
CLOOP: \# Cycle 0 1 2 2 3 4 4 5 5 6 6 7 8 8 9 10
lbu $t0, 0($t4) IF ID EX ME WB
sb $t0, 0($a1) IF ID ----> EX ME WB
addi \$t4, \$t4, 1 IF ----> ID EX ME WB
bne \$t4, \$t5, CLOOP IF ID ----> EX ME WB
addi \$a1, \$a1, 1 IF ----> ID EX ME WB
X1
IF IDx
X2
CLOOP: \# Cycle 0 11 2 3 3 4 4 5 6 6 7 % 8 9 10
lbu $t0, 0($t4) IF ID

```
(a) On The Second Implementation the branch penalty would only be one cycle. But, as we discussed in class, moving branch resolution from ME to EX might impact the critical path. Let \(\phi_{1}=\) 1 GHz denote the clock frequency on The First Implementation and call the clock frequency on The Second Implementation \(\phi_{2}\). For what value of \(\phi_{2}\) would the performance of the two implementations be the same when executing the code above for a large number of iterations?

Show your work.

Problem 2: The code below is taken from the solution to Homework 2 Problem 1. Sharp students might remember that the loop can be entered at four places: the COPY_LOOPd4 label (which is the normal way to enter such a loop), the second lb , the third lb , or the fourth lb . For this problem assume that the loop can only be entered at the COPY_LOOPd4 label.
```

COPY_LOOPd4:
lb $t0, 0($t4) \# First lb
sb $t0, 0($a1)
lb $t0, 1($t4) \# Second lb
sb $t0, 1($a1)
lb $t0, 2($t4) \# Third lb
sb $t0, 2($a1)
lb $t0, 3($t4) \# Fourth lb
sb $t0, 3($a1)
addi \$t4, \$t4, 4
bne \$t4, \$t5, COPY_LOOPd4
addi \$a1, \$a1, 4

```

(a) Schedule the code (rearrange the instructions) so that it executes without a stall on the implementation shown above.

Problem 3: Perhaps some students have already wondered why, if the goal were to reduce dynamic instruction count, the previous occurrence loop (the subject of the first two problems and of Homework 2) wasn't written using lw and sw instructions since they handle four times as much data. Such a loop appears below. Alas, the loop won't work for every situation, for one reason due to MIPS' alignment restrictions.

Let \(a_{p}\) denote the address of the previous text occurrence (the value is in t4), let \(a_{o}\) denote the address of the next character to write into the output buffer (the value is in a1), and let \(L\) denote the length of the previous occurrence to copy. (Register t5 is \(a_{p}+L\).)
```

COPY_LOOP44:
lw $t0, 0($t4)
sw $t0, O($a1)
addi \$t4, \$t4, 4
bne \$t4, \$t5, COPY_LOOP44
addi \$a1, \$a1, 4
j LOOP
nop

```
(a) In terms of \(a_{p}, a_{o}\), and \(L\), specify the conditions under which the loop above will run correctly. Also show that the loop would work for about only 1 out of 64 copies assuming that the values of \(a_{p}, a_{o}\), and \(L\), are uniformly distributed over some large range. For this part don't assume any special code added before or after.
(b) Suppose one added prologue code before the loop to copy the first few characters and epilogue code after the loop to copy the last few characters, with the goal of being able to use the loop for more than \(\frac{1}{64}\) th (or \(\frac{100}{64} \%\) ) of copies.

In terms of \(a_{p}, a_{o}\), and \(L\), specify the conditions under which the loop will run correctly and show that the fraction of copies that the loop can handle is about \(\frac{1}{4}\).

Also show the number of characters that should be copied by the prologue code and the number of characters that should be copied by the epilogue code.

Problem 1: Solve Spring 2017 Final Exam Problem 1, in which control logic to generate stall signals is added to a MIPS-I implementation that uses 12-bit bypass paths. See Spring 2017 Homework 5 Problem 3 and its solution, on which the final exam problem is based.
Attention perfectionists: The original Inkscape SVG for illustrations are available for download: the unsolved homework, http://www.ece.lsu.edu/ee4720/2017/mpipei3c.svg, the solved homework, http://www.ece.lsu.edu/ee4720/2017/hw05-p3-sol.svg,
and the exam, http://www.ece.lsu.edu/ee4720/2017/fe-p1.svg. Print them out as is (be sure that the rasterization of the SVG vector format targets the resolution of your printer) or use Inkscape or some other SVG editor to put in your solution. For a real challenge use a plain text editor to add your solution to the SVG source.

This assignment consists of questions on the ARM A64 (AAarch64) ISA. (Not to be confused with ARM A32, which might be called classic ARM. Older information sources that refer to ARM are probably referring to A32, which is not relevant to this assignment.)

A description of the ARM ISA is linked to the course references page, at
http://www.ece.lsu.edu/ee4720/reference.html. Feel free to seek out introductory material as a suppliment.

ARM A64 was used in EE 4720 Spring 2017 Homework 4 and Spring 2017 Midterm Exam Problems 2 and 3. It may be useful to see those assignments for code samples, but the questions themselves are different.

Appearing on the next page is a simple C routine, lookup, that returns a constant from a list. The routine appears to have been written with the expectation that its call argument, i, would be either 0,1 , or 2 . Following the C code is ARM A64 code for lookup as compiled by gcc version 8 .

Use the course reference materials and external sources to understand the ARM code below. The course references page has a link to the ARM ISA manual which should be sufficient to answer questions in this assignment. Feel free to seek out introductory material on ARM A64 (AArch64) assembly language, but after doing so use the ARM Architecture Reference Manual to answer questions in this assignment.

Full-length versions of the code on the next page, along with other code examples can be found at http://www.ece.1su.edu/ee4720/2018/hw05.c.html and http://www.ece.lsu.edu/ee4720/2018/hw05-arm.s.html. These include the pi program and a simple copy program that was a part of the decompress program used in Homeworks 1, 2, and 3.

Code on next page, problems on following pages.
```

int lookup(int i)
{
int c[] = { 0x12345678, 0x1234, 0x1234000 };
return c[i];
}

```
© ARM A64 Assembly Code. C code appears in comments.
lookup:
@@
©
© CALL VALUE
© wO: The value of i (from the \(C\) routine above).
@ RETURN VALUE
© wO: The value of \(c[i]\).
@
© Note: The size of int here is 4 bytes.
© const int \(c[]=\{0 x 12345678,0 x 1234,0 x 1234000\} ;\)
    adrp \(x 1\), .LC1
        mov w2, 0x4000
        ldr do, [x1, \#:lo12:.LC1]
        movk w2, 0x123, lsl 16
        str \(w 2,[s p, 8]\)
        str do, [sp]
© return c[i];
    ldr w0, [sp, w0, sxtw 2]
        ret
    .rodata.cst8,"aM",@progbits,8
. LC1:
    . word 0x12345678
    .word 0x1234

Problem 1: The ARM code above uses three kinds of register names, those starting with d, w, and x .
(a) Explain the difference between each.
(b) MIPS has general-purpose registers and four sets of co-processor registers. Indicate the name of the register set for each of the three types of ARM registers above. Hint: two are part of the same set.

Problem 2: The mov moves constant \(0 \times 4000\) into register w2. Actually, mov is a pseudo instruction.
(a) What are pseudo instructions called in ARM?
(b) What is the real instruction that the assembler will use in this particular case?
(c) Show the encoding for this use of mov. Be sure to show how w2 and \(0 \times 4000\) fit into the fields.

Problem 3: MIPS-I does not have an instruction like adrp.
(a) Describe what the adrp instruction does in general.
(b) Explain what it is doing in the code above. (It might be easier to look at the documentation for adr first.)
(c) Show MIPS code that writes the same value to its destination as adrp. Do not use MIPS pseudo instructions other than la. Assume that MIPS integer registers are 64 bits.

Problem 4: The movk instruction is sort of an improved version of lui.
(a) Describe what the movk instruction does in general.
(b) Explain why a single MIPS lui instruction could not do what the movk is doing in the code above.

Problem 5: Add comments to the ARM code above that explain what the code is doing, rather than what the individual instructions do.

Problem 6: The lookup routine was compiled using gcc at optimization level 3, the highest. Nevertheless, the code appears more complicated than it need to be. Explain what about the code is excessively complicated and how it could be simplified.

Problem 1: Solve 2014 Homework 4 Problem 1.
Problem 2: Appearing on the next page is a MIPS implementation and the execution of some code on that implementation.

(a) Wires in the implementation (on the previous page) are labeled in blue. Show the values on those wires each cycle that they are affected by the executing instructions. The values for label A are already filled in.
```

LOOP: \# Cycle 0
lwc1 f0, O(r1) IF ID EX ME WF
mul.s f1, f0, f10 IF ID -> M1 M2 M3 M4 M5 M6 WF
add.s f2, f2, f1 IF -> ID -------------> A1 A2 A3 A4 WF
lwc1 f0, 4(r1) IF -------------> ID EX ME WF
mul.s f1, f0, f11 IF ID -> M1 M2 M3 M4 M5 M6 WF
add.s f2, f2, f1 IF -> ID -------------> A1 A2 A3 A4 WF
bne r2, r1 LOOP IF -------------> ID EX ME WB
addi r1, r1, 8
IF ID EX ME WB

# Cycle 0

A 1 1 2 1 \# Sample
B
C

# Cycle

D
E

# Cycle

F
G
\# Cycle

```
(b) Schedule the code above so that it suffers fewer stalls. Register numbers can be changed but in the end the correct value must be in register f2. It is okay to add a few instructions before and after the loop. Each iteration must do the same amount of work as the original code.

Show a pipeline execution diagram for two iterations.

Problem 3: Design control logic for the lower M1-stage bypass multiplexor. Note that this is fairly easy since the mux has two inputs, so it's only necessary to detect the dependence.

An SVG source for the FP diagram can be found at:
http://www.ece.lsu.edu/ee4720/2018/mpipei-fp-by.svg.

Problem 1: Solve 2017 Final Exam Problem 2, which asks for or about pipeline execution diagrams for several code fragments on a superscalar MIPS implementation.

Problem 2: Solve 2017 Final Exam:
(a) Problem 5a, comparing superscalar and a scalar processor with a vector unit.
(b) Problem 5d, which asks how a VLIW ISA implementation might be simpler than a given superscalar implementation.

Homework 1

\section*{7 Spring 2017}

Homework 1

Problem 1: Follow the instructions for class account setup and for homework workflow. Review the comments in hw01.s and look for the area labeled "Problem 1," which has an empty procedure named split. The split routine itself is reproduced below, and for those who want to start before getting a class account the assembler for the entire assignment can be found at http://www.ece.lsu.edu/ee4720/2017/hw01.s.html.

Routine split is called with four arguments. Register a0 holds the number of elements in an input array, register a1 holds the address of the input array. The input array consists of 4 -byte elements. Register a2 holds the address of a 4 -byte-element output array and register a3 holds the address of a 2-byte-element output array. Complete split so that it copies elements from the input array to the 2 -byte output array (if it fits), otherwise to the 4 -byte output array. The return value, in register v 0 , should be the number of elements in the 4 -byte output array when copying is done.

When the code in the hw01.s file is run the split routine will be tested. The first line of output indicates whether the return value, v0, was correct. Below that the values written to the 4 - and 2 -byte arrays are shown side-by-side with correct values.
```

split:

## Register Usage

# 

# CALL VALUES:

# \$a0: Number of elements in input array.

# \$a1: Address of input array. Elements are 4 bytes each.

# \$a2: Address of output array, elements are 4 byte each.

# \$a3: Address of output array, elements are 2 byte each.

# 

# RETURN:

# [ ] \$v0, Number of elements in 4-byte output array.

# [ ] Write small elements in \$a3 array and large elements in \$a2.

# 

# Note:

# Can modify $t0-$t9, $a0-$a3

# [ ] Code should be correct.

# [ ] Code should be reasonably efficient.

# SOLUTION GOES HERE

```
```

jr \$ra
nop

```

There's another problem on the next page.

Problem 2: Note: The following problem was assigned last year, two years ago, and three years ago and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```

LOOP: \# Cycles 0
lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
LOOP: \# Cycles

```
(b) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3
IF ID EX ME WB
lw r1, O(r4)
IF ID -> EX ME WB
LOOP: \# Cycles

```
(c) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
IF ID -> EX ME WB
LOOP: \# Cycles

```
(d) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: \# Cycles

```

Problem 1: The following problems are from the 2016 EE 4720 Final Exam.
(a) The following problem appeared as 2016 EE 4720 Final Exam Problem 2a. (Just 2a, not 2b). Show the execution of each of the two code fragments below on the illustrated MIPS implementations. All branches are taken. Don't forget to check for dependencies.

```


# CODE SEQUENCE A

add r1, r2, r3
sub r4, r1, r5

```

Show execution of the following code sequence.
```


# CODE SEqUENCE B

```
beq r1, r1 TARG
or \(r 2, r 3, r 4\)
sub r5, r6, r7
xor r8, r9, r10

TARG:
```

lw r10, O(r11)

```
(b) For each of the two code sequences above label each used wire on the diagram below with: the cycle number, the instruction, and if appropriate a register number or immediate value. For example, the register file input connected to ID. IR bits 25:21 should be labeled 1: add, =2 because in cycle 1 the add instruction is using that register file input to retrieve register r2. See the illustration below. Only label wires that are used in the execution of an instruction. For example, there should not be a label 2: sub, \(=1\) because the value of \(r 1\) will be bypassed. Instead, label the bypass path that is used. Pay particular attention to wires carrying branch information and to bypass paths. Look through old homeworks and exams to find similar problems.
For Code Sequence A:


For Code Sequence B:


Problem 2: The following problem is from 2016 EE 4720 Final Exam Problem 5a. Suppose that a new ISAis being designed. Rather than requiring implementations to include control logic to detect dependencies the ISA will require that dependent instructions be separated by at least six instructions. As a result, less hardware will be used in the first implementation.
(a) Explain why this is considered the wrong approach for most ISAs.
(b) What is the disadvantage of imposing this separation requirement?

To help in solving this problem it might be useful to study the solutions to the following problems which involve hardware implementing branches in the statically scheduled five-stage MIPS implementation we've been working with: Spring 2016 Homework 3 (SPARC-like branch instruction), Spring 2015 Homework 2 Problem 2 (use reg bits for larger displacement) and Problem 3 (logic for IF-stage mux), Spring 2015 Homework 3 Problem 2 (implement bgt, but resolve it in EX), Spring 2011 Final Exam Problem 1 (resolve in ME, with bypass).

Problem 1: Modify the implementation below so that it implements the MIPS II bgezall instruction, see the subproblems for details on the hardware to be designed. See the MIPS ISA documentation linked to the course Web page for a description of the bgezall instruction. An Inkscape SVG version of the illustration below can be found at http://www.ece.lsu.edu/ee4720/2017/hw03-p1.svg. The illustration also appears on the next page.

(a) Design control logic to detect the instruction and connect its output to the br/jump logic cloud. The control logic should consist of basic gates, not a box like bgezall.
(b) Design the control logic to squash the delay slot instruction when bgezall is not taken. The control logic should squash the delay slot instruction by changing its destination register and memory operation. Be sure that the control logic squashes the correct instruction, and does so only when bgezall is not taken. Do not rely on magic clouds [tm].
(c) Add datapath or make other changes needed to compute the return address. Note that NPC is already connected to the ALU. Consider inexpensive ways to compute the second operand. (Adding a 32 -bit ID/EX pipeline latch is not considered inexpensive for this problem.)


To help in solving this problem it might be useful to study the solutions to the following problems: Spring 2014 Homework 3 (ARM A32 instructions, ARM-like scaling instructions). Fall 2010 Homework 3 (shift unit in MIPS).

See the course references page for a link to the ARM v8 ISA, which will be needed to solve the problems below.

Problem 1: In most RISC ISAs register number 0 is not a true register, its value as a source is always zero, and it can be harmlessly used as a destination (for example, for use as a nop instruction).

The ARM A64 instruction set (not to be confused with A32 [arm] or T32 [thumb]) takes a different approach to the zero register.
(a) What register number is the zero register in A64?
(b) Let \(z\) denote the answer to the previous part, meaning that rz can denote the zero register. In an ISA like MIPS, the general purpose register (GPR) file only needs enough storage for 31 registers, since the register zero location can be hardwired to zeros. But in ARM A64 the GPR file needs 32 storage locations because register number \(z\) is the zero register for some instructions, but an ordinary register for others. In ARM notation ZR denotes the zero register, in this problem rz indicates the register number of the zero register, which, depending on the instruction can refer to the zero register or to an ordinary register.

Show two instructions that can read \(r z\) and one that can write \(r z\), for these instructions \(r z\) is an ordinary register (at least for certain operand fields).

Show a one-destination-register, two-source-register instruction for which \(r z\) is the zero register for all operand fields.

Problem 2: The ARM A64 code below computes the sum of an array of 64 -bit integers. The load instruction uses post-index addressing, the behavior of this instruction is shown in the comments. (The @ is the comment character.)
```

LOOP: @ ARM A64

```
```

ldr x1, [x2], \#8 @ x1 = Mem[x2]; x2 = x2 + 8

```
ldr x1, [x2], #8 @ x1 = Mem[x2]; x2 = x2 + 8
    cmp x2, x4
    cmp x2, x4
    add x3, x3, x1
    add x3, x3, x1
    bne LOOP
```

    bne LOOP
    ```
(a) Appearing below is a pipeline based on our MIPS implementation. Add datapath elements so that it can execute the ldr with pre-index, post-index, and immediate addressing. Don't make changes that will break other instructions. Note that the register file has a write-enable input, which is necessary because there is no full time register that acts as register zero.

An Inkscape SVG version of the implementation can be found at https://www.ece.lsu.edu/ee4720/2017/hw04-armskel.svg.

- Show the bits used to connect to the address inputs of register file.
- Show the second write port on the register file needed for the updated address.
- Use fixed bit positions for the destination registers.
- Use the given decode logic to determine a write enable signal for each dest.
- The changes must work well with pipelining.
- As always, avoid excessively costly solutions.
- Do not add hardware for the branch or compare.


Problem 1: Complete MIPS routine fxitos so that it converts a fixed point integer to a singleprecision floating point value as follows. When fxitos starts register a0 will hold a fixed-point value \(i\) and register a1 will hold the number of bits that are to the right of the binary point, \(d\), with \(d \geq 0\). For example, to represent \(9.75_{10}=1001.11_{2}\) we would set a0 to 0 b 100111 and a1 to 2. (Or we could set a 0 to 0 b 100111000 and a1 to 5 .) When fxitos returns register f 0 should be set to \(i / 2^{d}\) represented as a single-precision floating point number.

Solve this problem by using a division instruction for \(i / 2^{d}\). (The floating division instruction can be avoided by performing integer arithmetic on the FP representation, but that's not required in this problem.)

Submit the solution on paper. Your class account can be used to work on the solution. The fxitos routine and a testbench can be found in
/home/faculty/koppel/pub/ee4720/hw/2017/hw05/hw05.s, follow the same instructions as for Homework 1.
fxitos:
```


## 

# 

# CALL VALUES:

# \$a0: Fixed-point integer to convert.

# \$a1: Number of bits to the right of the binary point.

# 

# RETURN:

# [ ] \$f0: The value as a single-precision FP number.

jr \$ra
nop

```

Problem 2: Appearing below is the MIPS hardware needed to implement bgezall from the solution to Homework 3. Recall that with bgezall the delay-slot instruction is annulled if the branch is not taken. Modify the hardware for new instruction bgezalllsu which executes like bgezall when the branch target is at or before the branch, but when the target is after the branch the delay-slot instruction is annulled when the branch is taken and allowed to execute normally if the branch is not taken. The opcode and rt values are the same as for bgezall. Hint: this can be done with very little hardware, a gate or two, if that.


Problem 3: Suppose that an analysis of the execution of benchmark programs on our pipelined MIPS implementation shows that over \(75 \%\) of bypassed values can be represented with 12 bits or fewer. A low-cost implementation takes advantage of this fact by using 12 -bit bypass paths.
(a) The control logic below is intended for bypass paths that can bypass a full 32 -bit value. Modify the control logic shown so that it works for 12-bit bypass paths. In your modified hardware add a stall signal to be used when values are too large to be bypassed.
- Indicate which parts of the added logic, if any, may lengthen the critical path.
- As always, avoid costly or slow hardware.

Attention perfectionists: An Inkscape SVG version of the implementation below can be found at http://www.ece.1su.edu/ee4720/2017/mpipei3c.svg.

It's small on purpose, use next page for solution.


It's small on purpose, use next page for solution.
(b) Why would it be far more challenging for a compiler to optimize for these 12-bit paths than for ordinary full-width bypass paths?


Attention Perfectionists: An Inkscape SVG version of the illustration used in the final exam and this assignment can be found at: https://www.ece.1su.edu/ee4720/2017/mpipei_fp.svg.

Problem 1: Answer Spring 2016 Final Exam Problem 2b and 2c, which ask about the execution of FP MIPS code. The solution to these problems are available. Make a decent attempt to solve these problems on your own, without looking at the solution. Only peek at the solution for hints and use the solution to check your work.

Problem 2: Appearing below are two MIPS code fragments and the MIPS implementation from the final exam. The fragments execute on the illustrated implementation with the addition of the datapath needed for the store instructions that was provided in Final Exam Problem 2c. The fragments are labeled Degree 1 and Degree 2, these refer to an optimization technique called loop unrolling, which has been applied to the Degree 2 loop.

```

LOOP: \# Degree 1
lwc1 f0, O(r1)
add.s f0, f0, f1
swc1 f0, O(r1)
bne r1, r3 LOOP
addi r1, r1, 4
LOOP: \# Degree 2
lwc1 f0, O(r1)
lwc1 f1, 4(r1)
add.s f0, f0, f9
add.s f1, f1, f9
swc1 f0, O(r1)
swc1 f1, 4(r1)
bne r1, r3 LOOP
addi r1, r1, 8

```
(a) Show a pipeline execution diagram of each on the illustrated code fragments. Show enough iterations to compute the CPI. Note that the second loop should have fewer stalls than the first.
(b) Compute the execution efficiency of both loops in CPI. Remember that the number of cycles should be determined by looking at the same point in execution, usually IF of the first instruction, in two different iterations. Put another way, just because the custom car you will order after graduation will take two months to arrive, doesn't mean that the factory makes just one car every two months.
(c) Assume that both loops operate on \(N\)-element arrays (and that \(N\) is even). The Degree-1 loop operates on just one element per iteration, while the Degree-2 loop operates on two elements per iteration.

Devise a performance measure that can be used to compare the two loops based on the work that they do. The improvement of Degree-2 or Degree-1 should be higher with this work-based performance measure than the improvement computed using CPI.
(d) Besides eliminating stalls, what makes Degree 2 faster than Degree 1 even when doing the same amount of work?

Attention Perfectionists: An Inkscape SVG version of the illustration of the superscalar MIPS implementation used in the final exam problems and their solution for this assignment can be found at http://www.ece.lsu.edu/ee4720/2016/fe-ss.svg and http://www.ece.lsu.edu/ee4720/2016/fe-p1abc-sol.svg.

Problem 1: Answer Spring 2016 Final Exam Problem 1 a, b, and c, in which a single memory port is connected to the ME stage of a two-way superscalar MIPS implementation. The solution to this problem is available. Make a decent attempt to solve this problem on your own, without looking at the solution. Only peek at the solution for hints and use the solution to check your work.

Problem 2: Answer Spring 2016 "Final Exam Problem" 1e, which asks for modifications to a 2-way supescalar MIPS implementation that avoids stalls for certain pairs of load instructions. Note: Problem 1d was given on the final exam. Problem 1e, which did not appear on the final, is an expanded version of Problem 1d.

Problem 1: Answer Spring 2016 Final Exam Problem 3, which asks about the performance of various branch predictors.

The solution to this problem is available. Make a decent attempt to solve this problem on your own, without looking at the solution. Only peek at the solution for hints and use the solution to check your work. Credit will only be given if there is some evidence of an attempt to solve the problem.

Problem 2: Compute the amount of storage needed for each predictor described at the beginning of Spring 2016 Final Exam Problem 3 (the same question used in the problem above) accounting for the following additional details: Each BHT stores a six-bit tag and a 16-bit displacement (in addition to whatever other data is needed).

Be sure to show the size of each table (BHT, PHT) that each predictor (bimodal, local, global) uses. Show the size in bits.

Problem 3: In a bimodal predictor the size of the tag and displacement is much larger than the 2-bit counter used to actually make the prediction. Consider a design that uses two tables, a BHT and a Branch Target Buffer (BTB). The BHT stores only the 2-bit counter, the BTB stores the tag and displacement. However, the tag and displacement are only written to the BTB if the branch will be predicted taken.

Draw a sketch of such a system and indicate the number of entries that should be in each table so that the amount of storage is the same as the original bimodal predictor.

\section*{8 Spring 2016}

Problem 1: Answer each MIPS code question below. Try to answer these by hand (without running code).
(a) Show the values assigned to registers t1 through t8 (the lines with the tail comment Val:) in the code below. Refer to the MIPS review notes and MIPS documentation for details.
```

.data
.byte 0x10, 0x11, 0x12, 0x13
.byte 0x14, 0x15, 0x16, 0x17
.byte 0x18, 0x19, 0x1a, 0x1b
.byte 0x1c, 0x1d, 0x1e, 0x1f
.text
la \$s0, myarray \# Load \$s0 with the address of the first value above.
\# Show value retrieved by each load below.
lbu $t1, 0($s0) \# Val:
lbu $t2, 1($s0) \# Val:
lbu $t2, 5($s0) \# Val:
lhu $t3, 0($s0) \# Val:
lhu $t4, 2($s0) \# Val:
addi \$s1, \$0, 3
add \$s3, \$s0, \$s1
lbu $t5, 0($s3) \# Val:
sll \$s4, \$s1, 1
add \$s3, \$s0, \$s4
lhu $t6, 0($s3) \# Val:
sll \$s4, \$s1, 2
add \$s3, \$s0, \$s4
lhu $t7, 0($s3) \# Val:
lw $t8, 0($s3) \# Val:

```
myarray:
(b) The last two instructions in the code above load from the same address. Given the context, one of those instructions looks wrong. Identify the instruction and explain why it looks wrong. (Both instructions should execute correctly, but one looks like it's not what the programmer intended.)
(c) Explain why the following answer to the question above is wrong for the MIPS 32 code above: "The lw instruction should be a lwu to be consistent with the others."

Problem 2: Note: The following problem was assigned last year and two years ago and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```

LOOP: \# Cycles 0
lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
LOOP: \# Cycles }0

```
(b) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3
IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB
LOOP: \# Cycles

```
(c) Explain error and show correct execution.
```

LOOP: \# Cycles }\quad
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID -> EX ME WB
LOOP: \# Cycles }\quad0\quad

```
(d) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: \# Cycles

```

Problem 3: Show the execution of the MIPS code below on the illustrated implementation. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be value being written. (In class we called such a register file internally bypassed.)
- Check carefully for dependencies.
- Pay attention to when the branch target is fetched and to when wrong-path instructions are squashed.
- Be sure to stall when necessary.

```

add r1, r2, r3
sub r4, r1, r5
beq r1, r1, SKIP
lw r6, 0(r4)
xor r7, r8, r9

```

SKIP:
ori r9, r10, 11

Problem 1: The code fragment below is to execute on the illustrated MIPS implementation. Unfamiliar instructions can be looked up on the MIPS ISA manual linked to the course references page. Show the execution of the code fragment below on the illustrated MIPS implementation. All branches are taken.
- Pay close attention to dependencies, including those for the branch.
- Note that unnecessary stalls are just as incorrect as not stalling when a stall is necessary.

```

add r4, r2, r3
1w r6, 8(r4)
sub r1, r6, r5
bltz r1 TARG
and r8, r7, r10
or r11, r12, r13
xor r14, r11, r8

```

TARG:
sw r1, \(0(r 2)\)

Spoiler Alert: Hint about solution on next page.

Problem 2: The implementation below (which is the same as the implementation for the previous problem) lacks hardware needed for the bltz instruction. In this problem design such hardware as described in the parts below. Note: An Inkscape SVG version of the implementation can be found at http://www.ece.lsu.edu/ee4720/2016/mpipei3.svg.

(a) Add the hardware needed to detect when a bltz is taken. The hardware should have an output labeled TAKEN, which should be set to logic 1 if there is a taken bltz in ID. Include control logic, including the logic for detecting bltz.
(b) The solution to the previous problem (not the previous part to this problem) should have included a stall due to the branch instruction. Add a bypass path to the hardware designed above so that the branch from the previous problem can execute without stalling.
(c) Design control logic for the bypass path.

Problem 3: The code below is similar to the code from the first problem, the only difference is in the branch instruction. In this problem explain some bad news and good news about that branch.
```

add r4, r2, r3
lw r6, 8(r4)
sub r1, r6, r5
beq r0, r1 TARG
and r8, r7, r10
or r11, r12, r13
xor r14, r11, r8

```

TARG:
sw r1, \(0(r 2)\)
(a) The bad news is that adding bypass paths for a beq would not be a good idea, even though adding bypass paths for the bltz was a good idea. Explain why.
(b) The good news is that the program above can easily avoid the stalls by just changing the branch instruction. Explain how. (Of course, it should go without saying that the changed program must do the same thing as the original one.)


Problem 1: Illustrated below is our MIPS implementation with some control logic shown. Modify the implementation so that it can execute the SPARC v8 instructions as described below. In your solution ignore register windows, assume that SPARC uses an ordinary 32-register general-purpose register file.

Details of the SPARC ISA (which includes later versions) can be found in http://www.ece.lsu.edu/ee4720/doc/JPS1-R1.0.4-Common-pub.pdf. An Inkscape SVG version of the illustration below can be found at http://www.ece.lsu.edu/ee4720/2016/mpipei3b.svg.

(a) Modify the implementation for format 3 arithmetic instructions. Use add as an example. Show changes in the bits used: to index the register file, to format the immediate, and to generate the writeback register number, dst.
(b) Modify the implementation for branch instructions. Use BPcc as an example. Be sure to make changes for computing the branch target.

Show changes in the hardware to generate the target address. Remove the unneeded MIPS branch comparison hardware and add a CC register.
(c) Modify the implementation for load and store instructions. Use LDUW and STW as examples.

Show changes in the format immediate unit, and make sure that it can handle both ADD and loads and stores.

Problem 2: Section 1.3.1 of the SPARC JPS1 lists features of the ISA.
(a) Indicate which features are typical RISC features and which features are not.
(b) One feature is "Branch elimination instructions" Provide an example of how such an instruction can be used to eliminate a branch.

Problem 1: Problem 2b from the 2015 Final Exam asks about our usual FP MIPS pipeline. An Inkscape SVG version of the FP pipeline can be found at
http://www.ece.lsu.edu/ee4720/2016/mpipei_fp.svg.
(a) Solve Spring 2015 Final Exam problem 2b.
(b) Add bypass paths to the implementation from problem 2 b needed so that the code from 2 b executes without a stall.

Problem 2: Solve Problem 2c from the 2015 Final Exam, which asks about our usual superscalar pipeline.

An Inkscape SVG version of the ordinary 2-way superscalar MIPS pipeline used in 2c can be found at http://www.ece.1su.edu/ee4720/2016/mpipei3ss.svg.

Problem 3: Solve Problem 1 from the 2015 Final Exam, which asks about a modified version of our two-way superscalar MIPS implementation.

An Inkscape SVG version of the fused-add 2-way superscalar MIPS pipeline used in 2c can be found at http://www.ece.1su.edu/ee4720/2015/fess.svg.

Problem 1: Solve Spring 2015 Final Exam Problem 3, which asks about the performance of several branch predictors. See older final exam solutions for more information on how to solve these kinds of problems.

Problem 2: Show major elements of the hardware for each predictor used in Spring 2015 Final Exam Problem 3a. In particular:
- Show the BHT, PHT, and GHR (in those predictors that use them).
- Show the connection from the PC to the appropriate table.
- Show the number of bits in each connection.
- Show the logic generating a "predict taken" signal.

You do not need to show the logic to update the predictor or to generate the target address.

This assignment is not for credit. Any submitted solution will be corrected and returned, but the grade won't count.

Problem 1: Solve Spring 2015 Final Exam Problem 4. Part (a) is a fill-in-the-blanks cache question. Part (b) asks about the hit ratio of a simple program. These two parts are very similar to problems asked on almost every prior final exam. It would be a good idea to study these to the point where they can be answered in under 40 seconds (but on the real exam take it a bit more slowly). Part (c) asks about the performance of a code fragment on a particular cache. This part requires actual thinking.

\section*{\(9 \quad\) Spring 2015}

Problem 1: Answer each MIPS code question below. Try to answer these by hand (without running code).
(a) Where indicated, show the changed register in the following simple code fragments:
```


# r1 = 10, r2 = 20, r3 = 30, etc.

# 

add r1, r2, r3

# 

# Changed register, new value:

# r1 = 10, r2 = 20, etc.

# 

add r1, r1, r2

# 

# Changed register, new value:

```
(b) Show the values assigned to registers s1 through s6 in the code below. Correctly answering this question requires an understanding of MIPS big-endian byte ordering and of the differences between lw, lbu, and lb. Refer to the MIPS review notes and MIPS documentation for details.
```

    .data
    values: .word 0x11121314
.word 0xaabbccdd
.word 0x99887766
.word 0x41424344
.text
la \$s0, values \# Load \$s0 with the address of the first value above.
lw $s1, 0($s0)
lw $s2, 4($s0)
sh $s2, 0($s0) \# Note: this is a store *half*.
lbu $s3, 0($s0)
lbu $s4, 3($s0)
lb $s5, 4($s0)
lb $s6, 7($s0)

```

Problem 2: Note: The following problem was assigned last year and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```

LOOP: \# Cycles
lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
LOOP: \# Cycles

```
(b) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB
LOOP: \# Cycles

```
(c) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID -> EX ME WB
LOOP: \# Cycles

```
(d) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: \# Cycles

```

Problem 3: Show the execution of the MIPS code below on the illustrated implementation. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be value being written. (In class we called such a register file internally bypassed.)
- Check carefully for dependencies.
- Pay attention to which registers are sources and which are destinations, especially for the sw instruction.
- Be sure to stall when necessary.

```

add r1, r2, r3
lw r4, O(r1)
sw r1, O(r1)
sub r5, r4, r1
sh r5, 4(r1)

```

Problem 4: The code below is the same as the code used in the previous problem, but the MIPS implementation is different.
(a) Show the execution of the MIPS code below on the illustrated implementation.
(b) On the diagram label multiplexor data inputs connecting to bypass paths that are used in the execution of this code. The label should include the cycle number, the register, and the instruction consuming the value. For example, the label \(3: \mathrm{r} 1: 1 \mathrm{lw}\) might be placed next to one of the data inputs on the ALU's upper mux.

```

add r1, r2, r3
lw r4, O(r1)
sw r1, O(r1)
sub r5, r4, r1
sh r5, 4(r1)

```

For those preparing electronic submission of a solution (E-mail) and who would like a vectorformat version of the MIPS implementation can find it in Encapsulated Postscript at http://www.ece.lsu.edu/ee4720/2015/mpipei3.eps and for those who would like to edit the image can find it in Inkscape SVG at http://www.ece.lsu.edu/ee4720/2015/mpipei3.svg.

Problem 1: Answer Spring 2014 Final Exam Problem 7(c), which asks about how new load addressing behavior should be added to MIPS.

Problem 2: Answer Spring 2014 Final Exam Problem 5, which asks about a new MIPS branch instruction, bfeq.

Problem 3: Show the control logic for the IF-stage multiplexor in the MIPS implementation below.
- The control logic should work for beq, bne, bgtz, bgez, and j. Assume that any other instruction is not a control transfer.
- Show exactly which IR bits are needed by the control logic that detects bgez (Hint, hint.) and other instructions.
- The control logic should check the condition to determine if the branch is taken.


Problem 1: For the following question refer to the Intel 64 and IA- 32 Architectures Software Developer's Manual linked to the course references page. Intel 64 is an example of a CISC ISA, but not a good example because it evolved from an ISA designed for a 16 -bit address space. Over the years the size of the general purpose registers increased from 16 bits to 64 bits and the number of general-purpose registers increased from 8 to 16 .
(a) Show the 64 -bit names of the general purpose registers provided by Intel 64 . (See Chapter 3 of the manual mentioned above.)
(b) A MIPS assembly language instruction uses the same name for a register regardless of how many bits of the register we use. For example, sb r1, \(0(r 2)\) uses 8 bits of r1 and sw r1, \(0(r 2)\) uses all 32 bits, but in both instructions we refer to \(r 1\). Not so in IA-32/Intel 64, in which the name of the register indicates how many bits to use. Show the names for RAX for the different sizes and positions in the register.

Problem 2: Diagrams of the MIPS implementation for this problem can be found in EPS format at http://www.ece.lsu.edu/ee4720/2015/hw02-p3-if-mux-sol.eps and in Inkscape SVG (which can easily be edited) at http://www.ece.lsu.edu/ee4720/2015/hw02-p3-if-mux-sol.svg.

As has been pointed out in class, MIPS lacks a bgt rs, rt, TARG (branch greater than comparing two registers) instruction because the ISA was designed for a five-stage implementation in which the branch is resolved in ID. To resolve bgt in ID one would have to compare two register values starting about half-way through the cycle, something that might slow down the clock frequency.

In this problem suppose there was a bgt instruction in MIPS. We would like the implementation to have the same clock frequency as our bgt-less implementation. One way of doing that is by resolving bgt in EX (but still resolving the other branches in ID as they are now). If we resolve in EX we can expect a one-cycle branch penalty, as can be seen in the PED below.

```

TARG:
lw r9, O(r10) IF ID EX ME WB

# Cycle 0

```

Note that xor is squashed in cycle 3, which is the behavior we want for a taken bgt (see the second subproblem below). If bgt were not taken then no instruction would be squashed.
(a) Modify the implementation on the next page (taken from the Homework 2 solution) so that bgt is resolved in EX. Note: The original assignment had a very big typo in the previous sentence: giving ID instead of EX as the stage to resolve in.
- Pay attention to cost. Assume that a magnitude comparison (e.g., greater than) is relatively costly.
- Show the control logic for bgt.
- Do not "break" existing instructions.
(b) If bgt is taken an instruction will have to be squashed. (Because bgt has just one delay slot, just like all the other branches.) Add logic so that a one-bit signal sq (squash) is delivered to ID when the instruction in ID needs to be squashed due to a taken bgt.


An EPS version of the MIPS FP implementation used in some of the problems below can be found at http://www.ece.lsu.edu/ee4720/2015/mpipei_fp.eps and an easy-to-edit Inkscape SVG version can be found at http://www.ece.lsu.edu/ee4720/2015/mpipei_fp.svg.

Problem 1: Solve 2014 Midterm Exam Problem 2, which asks for a stall-in-ME version of our floating-point pipeline. A solution to this problem is available but use it only if you are stuck, and after you are finished to check your answer. If you got it wrong, then solve the problem again without looking at the solution.

Problem 2: Solve 2014 Final Exam Problem 1, which asks for an execution diagram of code running on the solution to the 2014 Midterm Problem 2.

There's another problem on the next page.

Problem 3: In the FP implementation on the next page (which is the same as the one used in class) an add.s instruction can stall due to an earlier mul.s, see the example below.
```


# Execution of code on the illustrated implementation.

# Cycle 0

mul.s f0, f1, f2 IF ID M1 M2 M3 M4 M5 M6 WF
add.s f6, f7, f8 IF ID A1 A2 A3 A4 WF
add.s f3, f4, f5 IF ID -> A1 A2 A3 A4 WF
and r6, r7, r8 IF -> ID EX ME WB

```

To avoid the stall consider the fpa- \(4 / 6\) design in which an add.s instruction that would stall taking the usual route instead enters the FP pipeline at the M1 unit. Assume that the M1 unit's control signal (not shown and not part of the problem) will command it to pass the values at its inputs to its outputs unchanged when it is carrying add.s operands. Then at the appropriate time it crosses over to A1 and continues through the remaining adder stages. An add.s not facing a WF structural hazard stall would go from ID to A1, as in the usual design. See the execution below.
```


# Desired execution on the fpa-4/6 implementation.

# Cycle 0

mul.s f0, f1, f2 IF ID M1 M2 M3 M4 M5 M6 WF
add.s f6, f7, f8 IF ID A1 A2 A3 A4 WF \# Uses 4-stage (normal) path.
add.s f3, f4, f5 IF ID M1 M2 A1 A2 A3 A4 WF \# Uses 6-stage (M1 M2..) path.
and r6, r7, r8 IF ID EX ME WB

```
(a) Modify the pipeline to implement fpa- \(4 / 6\).
- Show the datapath for the operands crossing from the multiply to the add unit.
- Show the control logic. The control logic should only send add.s into M1 if it would stall taking the usual route.
- The control logic should include the we, fd, and xw signals, and signals for any multiplexors that you add.
- As always, pay attention to cost and critical path.
(b) In the code fragment above the add.s f 3 goes from ID to M1. If it had gone from ID to M2 it would have still avoided the WF hazard and it also would have finished one cycle earlier. Consider an fpa-4/5/6 design in which an add.s can start at A1, M2, or M1, using the first one that avoids a stall. Provide a code example that would finish sooner on an fpa- \(4 / 5 / 6\) design than on an fpa- \(4 / 6\) design. Hint: A correct answer can add just one more instruction to the code fragment above.
(c) Is the fpa- \(4 / 5 / 6\) design better than the fpa- \(4 / 6\) design? Justify your answer using reasonable cost estimates and made-up properties of typical user programs. Either yes or no is correct, credit will be given for the justification.


An EPS version of the MIPS superscalar implementation used in one of the problems below can be found at http://www.ece.1su.edu/ee4720/2015/mpipei3ss.eps and an easy-to-edit Inkscape SVG version can be found at http://www.ece.lsu.edu/ee4720/2015/mpipei3ss.svg.

Problem 1: Solve 2014 Final Exam Problem 2, which asks for control logic in a 2-way superscalar processor.

Problem 2: Solve 2014 Final Exam Problem 3, in which branch predictors are analyzed. Note: Check earlier final exams for solutions to similar problems.

Problem 1: Solve 2014 Final Exam Problem 4, the cache problem. Homework 1

\section*{10 Spring 2014}

Problem 1: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.

```

    lw r1, -6(r3)
    lw r5, -2(r3)
    LOOP:
add r5, r5, r1
lw r1, 2(r3)
bneq r3, r4, LOOP
addi r3, r3, 4
jr r31
sw r5, 0(r6)

```
(a) Show the execution of the code above on the illustrated implementation up to and including the first instruction of the third iteration (that is, the third time that the add instructions is fetched).
- Carefully check the code for dependencies.
- Be sure to stall when necessary.
- Pay careful attention to the timing of the fetch of the branch target.
(b) Compute the CPI for a large number of iterations.

Problem 2: Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```

LOOP: \# Cycles
lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB
LOOP: \# Cycles

```
(b) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3
IF ID EX ME WB
lw r1, O(r4)
IF ID -> EX ME WB
LOOP: \# Cycles

```
(c) Explain error and show correct execution.
```

LOOP: \# Cycles
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID -> EX ME WB
LOOP: \# Cycles

```
(d) Explain error and show correct execution.
```

LOOP: \# Cycles 0
add r1, r2, r3
xor r4, r1, r5
IF ID EX ME WB
IF ----> ID EX ME WB
LOOP: \# Cycles

```

Problem 1: Solve Spring 2013 Homework 4 Problem 1, in which an integer multiply unit is added to the pipeline. The solution to the problem is available, but please make an honest attempt to solve it yourself. Understanding the solution to this problem will help with Problem 2, below.

Problem 2: Solve Spring 2013 Final Exam Problem 1. This problem is easier than it looks (but a solution will not be available until after the homework due date). Please refer to the Statically Scheduled MIPS study guide, http://www.ece.1su.edu/ee4720/guides/ssched.pdf, for help on solving similar problems and for where to find solutions to such problems.

For this assignment read the ARM Architecture Reference Manual linked to http://www.ece.lsu.edu/ee4720/reference.html. This assignment asks about the ARM A32 instruction set.

Problem 1: Show the encoding of the ARM A32 instruction that is most similar to MIPS instruction add r1, r2, r3.

Problem 2: ARM instructions can shift one of its source operands, something MIPS cannot. With this feature the code below can be executed with a single ARM add instruction. Show the encoding of such an ARM A32 add instruction.
```

sll r1, r2, 12
add r1, r4, r1

```

Problem 3: So, the ARM add instructions can shift one of its operands, something that MIPS would need two instructions to do. Since we have been working with MIPS for so long it would be natural for us to get protective of MIPS and defensive or jealous when hearing about wonderful features of other ISAs that MIPS doesn't have. To relieve these negative emotions lets add operand shifting to MIPS with a new addsc instruction. The addsc instruction will use MIPS' sa field to specify a shift amount. So instead of, for example, the following two instructions:
```

sll r1, r2, 12
add r1, r4, r1

```

We could use just
addsc r1, r4, r2, 12
where the " 12 " indicates that the value in r 2 should be shifted by 12 before the addition.
Modify our five-stage MIPS implementation so that it can implement this instruction. (See below for diagrams.)
- The addsc should execute without a stall.
- Don't break existing instructions.
- Don't increase the critical path by more than a tiny amount.
- Keep an eye on cost.

Assume that both the ALU and shift unit take most of the clock period. This means if the ALU and shifter are in the same stage and output of the shifter is connected to the ALU, the critical path will be doubled. (Of course, doubling the critical path would be disastrous for performance.)

There are several ways to solve this, one possibility includes adding a sixth stage, another possibility uses a plain adder (not a full ALU) in the EX stage.

Add hardware to the implementation below. Source files for the diagram are at: http://www.ece.lsu.edu/ee4720/2013/mpipei3.pdf,
http://www.ece.lsu.edu/ee4720/2013/mpipei3.eps, http://www.ece.lsu.edu/ee4720/2013/mpipei3.svg. The svg file can be edited using Inkscape.


To see how a shift unit can be added to MIPS see Fall 2010 Homework 3.

Problem 1: The following code fragments execute incorrectly on the following pipeline. For each fragment describe the problem and correct the problem.

(a) Describe problem and fix problem.
```

lwc1 f2, 0(r1) IF ID EX ME WF
add.s f1, f2, f3 IF ID A1 A2 A3 A4 WF

```
(b) Describe problem and fix problem.
```

add.s f1, f2, f3
IF ID A1 A2 A3 A4 WF
addi r1, r1, 4
lwc1 f2, O(r1)
IF ID EX ME WB
IF ID EX ME WF

```
(c) Describe problem and fix problem.
```

add.d f1, f2, f3 IF ID A1 A2 A3 A4 WF

```

Problem 2: The code fragment below contains a MIPS floating-point comparison instruction and branch. The pipeline illustrated below does not have a comparison unit, in this problem we will add one. The comparison unit to be used has two stages, named C1 and C2. The output of C2 is one bit, indicating if the comparison was true.

```

c.gt.d f2, f4
bc1t TARG
add.d f2, f2, f10
TARG: xor r1, r2, r3

```
(a) Add the comparison unit to the pipeline above. Also add a new register FCC (floating point condition code) that is written by the comparison instruction and is used by the control logic to determine if a floating-point branch is taken.

The FCC register should have a data and write-enable input, show the control logic generating the write-enable signal. Show a cloud labeled "branch control logic" and connect it to appropriate datapath components.
(b) Show the execution of the code sample above on your modified hardware, but without any bypass paths for the added hardware.
(c) Add whatever bypass paths are needed so that the code executes with as few stalls as possible but without having a major impact on clock frequency. Assume that C2 produces a result in about \(80 \%\) of the clock period.

Source files for the diagram are at: http://www.ece.lsu.edu/ee4720/2014/mpipeifp.eps,
http://www.ece.lsu.edu/ee4720/2014/mpipeifp.svg. The svg file can be edited using Inkscape.

Problem 1: Solve Spring 2013 Final Exam Problem 2, the problem is to design control logic to detect stalls in a 2 -way superscalar system, and to add bypass paths for a special case.

Problem 2: Solve Spring 2013 Final Exam Problem 3, asking an assortment of branch predictor problems. See past final exams for additional problems of this type.

Problem 1：Solve Spring 2013 Final Exam Problem 4（a），asking for details about a partially described cache．

Problem 2：Solve Problem 4（b），a really easy hit ratio question．
Problem 3：Consider the code in Problem 4（c）．For these questions assume that \(\mathrm{b}=0 \times 1000\) ， with that value of b we know that b ［0］starts at the beginning of a line．
（a）What would the hit ratio be if ASIZE were 0 （meaning that the a array is effectively not part of the structure）？When you compute the hit ratio consider both loops（the one with sum＋＝in the loop body and the one with norm＿val in the loop body）．
（b）For what value of ASIZE would Some＿Struct（or b［0］or b［i］，etc．）be the same size as a cache line？
（c）What would the hit ratio be if Some＿Struct were the size of a cache line？Consider both loops．
（d）Find the smallest value for ASIZE that will minimize the hit ratio（make things as bad as they can get）．（This is part（c）from the test．）Don＇t forget that this is a set－associative cache．
\(\square\)

\section*{11 Spring 2013}

Problem 1: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.

```

LOOP:
lw r2, O(r4)
slt r1, r2, r7
bne r1, r0 LOOP
addi r4, r4, 4
sw r4, O(r6)
jr r31

```
(a) Show the execution of the code above on the illustrated implementation up to and including the first instruction of the second iteration.
- Carefully check the code for dependencies.
- Be sure to stall when necessary.
- Pay careful attention to the timing of the fetch of the branch target.
(b) Compute the CPI for a large number of iterations.

Problem 2: The code fragment below is the same as the one used in the last problem, but the implementation is different (most would say better).

```

LOOP:
lw r2, O(r4)
slt r1, r2, r7
bne r1, r0 LOOP
addi r4, r4, 4
sw r4, O(r6)
jr r31

```
(a) Show the execution of the code on this new implementation.
- There will still be stalls due to dependencies, though fewer than before.
(b) Compute the CPI for a large number of iterations.

Problem 3: Consider once again the code fragment from the previous two problems, and the implementation from the previous problem. In this problem consider a MIPS implementation that executes a blt instruction, an instruction that is not part of MIPS. With such an instruction the code fragment from the previous problems can be shortened, and one would hope that the code would take less time to run. In this problem rather than hope we'll figure it out.

```

LOOP:
lw r2, O(r4)
blt r2, r7 LOOP
addi r4, r4, 4
sw r4, O(r6)
jr r31

```
(a) Add the additional datapath (non-control) hardware needed to execute blt. Hint: Just add one unit and a few wires.
(b) Show the execution of the code on the illustrated implementation up until the second fetch of lw.
(c) As we discussed in class, doing a magnitude comparison in ID might stretch the critical path, forcing a reduction in clock frequency. Suppose the clock frequency without blt is 1 GHz . At what clock frequency will the blt implementation, the one in this problem, be just as fast as the implementation from the prior problem on their respective code fragments?
- Be sure to pick a sensible meaning of just as fast. Do not define just-as-fast in terms of CPI.
(d) Explain why the code fragments in these problems might exaggerate the benefit of the blt instruction.

Note: For help with this and similar assignments see the Statically Scheduled MIPS study guide linked to http://www.ece.1su.edu/ee4720/guides.html.

Problem 1: Solve Spring 2012 Midterm Exam Problem 1. Part a is the usual draw-a-pipeline-execution-diagram-and-find-the-CPI problem, but it's on an implementation with some bypass paths removed. For part b you need to design control logic to generate stalls for the missing bypasses.

Problem 2: Solve Spring 2012 Midterm Exam Problem 2. In that problem the memory stage is split in two.

Problem 1: As described in class, SPARC v7 integer branch instructions use a 22 -bit immediate field for the displacement. Branches are typically used in loops and if/else constructs, and so the \(\pm 2097152\) instruction range might be more than is needed. So did the computer engineers at Sun Microsystems (now part of Oracle). Look up the v 7 integer branch instruction in the SPARC Joint Programming Specification (JPS1), linked to the course references page (look for JPS1). You'll find SPARC v7 integer branch under Instruction Definitions in the Deprecated Instructions section. Then look up the replacement integer branch instructions (not in the deprecated section).
(a) Sketch (or cut-and-paste, take a picture with your cell phone, etc.) the format of the three instructions (one old, two new).
(b) Describe how BPr is different than the original v7 integer branch instruction, and point out two benefits.
(c) Describe how BPcc is different than the original v7 integer branch instruction. This instruction shares one benefit with BPr, but it has lost 2 bits of displacement in order to accommodate 64-bit register values. (The other third lost bit has nothing to do with 64 -bit register values). Explain.

Problem 2: For the following assignment familiarize yourself with the VAX ISA by looking in the VAX-11 Architecture Reference Manual (linked to the course references page). In particular, see Section 2.6 for a summary of the instruction format, and Chapter 3 for details on the operand specifiers used in the instruction formats. For examples, look at some past homework assignments in this course: http://www.ece.1su.edu/ee4720/2010f/hw04_sol.pdf,
http://www.ece.lsu.edu/ee4720/2007f/hw03_sol.pdf, and http://www.ece.lsu.edu/ee4720/2002/hw02_sol.pdf.

The VAX format is simple, it consists of a one- or two-byte opcode followed by some number of operand specifiers and any additional fields they may use. The operand specifiers are 8 bits, and are followed by a possible extension and immediates. (See Section 2.6 and Chapter 3 of the VAX-11 Architecture Reference Manual.)
(a) The VAX operand specifier is 8 bits, it includes a 4 -bit mode field, and for literal addressing, a 6 -bit literal field. (A literal in VAX is a small immediate.) Explain how it's possible to fit a 4-bit mode field and a 6 -bit literal field into 8 bits.
(b) Find the best VAX replacement for each of the two MIPS instructions below and show their encoding. The two VAX instructions will be different.
```

addi r1, r2, 1
addi r1, r1, 1

```
(c) Find a VAX instruction to replace the following sequence of MIPS instructions, and show its encoding.
```

lw r1, O(r2)
lui r3, 0x2abb
ori r3, r3, Oxccdd
add r1, r1, r3
sw r1, 0x100(r2)

```
(d) Compare the size of the VAX instruction from the problem above to the size of the MIPS instructions.

Problem 1: Recall that the MIPS-I mult instruction reads two integer registers and writes the product into registers hi and lo. To use the product the values of lo and hi (if needed) have to be moved to integer registers, done using a move from instruction such as mflo. In this problem these instructions will be added to the implementation below.


Consider an integer multiply unit that consists of two stages, Y1 and Y2. The inputs to Y1 are the 32 -bit multiplier and multiplicand, and the output of Y2 is the 64 -bit product. Unit Y1 has three 32 -bit outputs named s0, s1, and s2; unit Y2 has 332 -bit inputs of the same name. As one would guess, the data from the \(\mathbf{s} 0\) output of Y1 should be sent to the \(\mathbf{s} 0\) input to Y2, likewise for s1 and s2.

As with other functional units, such as the ALU, inputs to Y1 and Y2 must be stable near the beginning of the clock cycle and the outputs must be stable near the end of the clock cycle. There is enough time to put a multiplexer before the inputs, or after the outputs (but not both).

Solve the two parts below together. That is, the hardware for part (a) might take advantage of the hardware for part (b) and vice versa.
(a) Add the datapath hardware needed to implement the mtlo, mthi, mflo, and mfhi instructions. Both the ALU and the integer multiply unit have an operation to pass either input to its output unchanged. That is, let \(x\) denote the ALU output and let \(a\) and \(b\) its inputs. In addition to operations like \(x=a+b\) and \(x=a \& b\), the ALU can also perform a pass- \(a\) operation, that is, \(x=a\) and a pass- \(b\) operation, \(x=b\). The integer multiply unit also has pass- \(a\) and pass- \(b\) operations.
- Put the hi and lo registers in the ID stage.
- Do not write the hi and lo registers earlier than the ME stage.
- As always, cost is a criteria.
- Bypass paths will be added in the parts below.
(b) Add the datapath hardware needed to implement the mult instruction. That is, put the Y1 and Y2 units in the appropriate stages, and connect them to the appropriate pipeline latch registers (adding new ones where necessary).
- Don't add new bypass paths, but take advantage of what is available.
(c) Show the execution of the code below on your hardware so far. That is, your hardware should not have any new bypass paths, but existing bypass paths in the implementation can be used.
```

sub r2, r6, r7
mult r1, r2
mflo r3
add r4, r3, r5

```
(d) Add bypass paths so that the code below (which is the same as in the previous part) can execute without a stall. Assume that an additional multiplexer delay is tolerable.
```

sub r2, r6, r7
mult r1, r2
mflo r3
add r4, r3, r5

```

Problem 2: Continue to consider the implementation of the MIPS-I mult instruction. If MIPS designers thought that an integer multiply unit could be built with two stages they might not have used special registers, hi and lo, for the product.
(a) Show how the pipeline would look if the multiply unit had three stages, Y1, Y2, and Y3. There is no need to add bypass paths for this part.
(b) Explain why there is much less of a need for the hi and lo registers with a two-stage multiply unit (the first problem) than with a three-stage unit (this problem).

Problem 1: In the following execution of MIPS code the lw instruction raises a TLB miss exception and the handler is called. A TLB miss is not an error, it indicates that the TLB needs to be updated, which is what the handler will do.

Execution is shown up to the first instruction of the handler. Alert students will recognize that there is something wrong in the execution below: it shows the execution of a deferred exception for an instruction, the 1 w , that should raise a precise exception.
```


# Cycle }\quad0\quad

    sh r1, O(r3) IF ID EX ME WB
    lw r1, O(r2) IF ID EX M*x
    addi r2, r2, 4 IF ID EX ME WB
    sw r7, O(r8) IF ID EX ME WB
    and r4, r1, r6 IF ID EX ME WB
    or r10, r11, r12
    HANDLER:

# Cycle }\quad0\quad

    sw r31,0x100(r0) IF ID EX ME WB
    ... # Additional handler code here.
    eret
    ```
(a) Show the execution of the eret instruction and the instructions that execute after the eret. Assume that eret reaches IF in cycle number 100. The execution should be for a deferred exception, even though memory instruction exceptions should be - must be - precise. A correct solution to this part will result in incorrect execution of the code.
(b) Suppose the execution above is for a computer on Mars, meaning that there is no fast or cheap way of replacing the hardware, and there is no way to turn on precise exceptions for the lw. Happily, it is possible to re-write the handler. Explain what the handler would have to do so that the code above executes correctly. The handler will know the address of the faulting instruction. Optional: explain why the sw r7 is nothing to worry about, at least in the execution above.
(c) Show the execution of the code above, but this time for a system in which lw raises a precise exception. Start at cycle 0 with the sh instruction, and have the lw raising once again a TLB miss exception. The execution should be in two parts, first from the sh up to the first instruction of the handler, then jump ahead to cycle 100 with eret in IF and continue with whatever instructions remain.

Problem 2: Solve Spring 2012 Final Exam Problem 2, which asks for the execution of MIPS floating-point instructions on our FP implementation.

Problem 3: Solve Spring 2012 Final Exam Problem 1 (yes, this is out of order). In this problem parts of the FP multiply unit are used to implement the MIPS integer mul instruction. Note that the mul writes integer registers, unlike mult which writes the hi and lo registers. In other words, do not use hi and lo registers in your solution.

SVG and EPS versions of the superscalar processor illustration are available at http://www.ece.1su.edu/ee4720/2013/mpipei3ss.svg and http://www.ece.lsu.edu/ee4720/2013/mpipei3ss.eps, respectively. Inkscape can be used to edit the SVG version.

Problem 1: The two-way superscalar implementation below has two memory ports in the ME stage, and so it can sustain an execution of 2 IPC on code containing only load and store instructions. Since for many types of programs loads and stores are rarely so dense and because memory ports are costly, it is better to make a 2-way processor with just one memory port in the ME stage.

Modify the implementation below so that it has just one memory port in the ME stage. It should still be possible to execute arbitrary MIPS programs, albeit more slowly.


Problem 2: The datapath hardware for resolving branches in the 2-way superscalar MIPS implementation below is incomplete: it does not show branch target computation for the instruction in Slot 1 (it is shown for Slot 0). (The hardware for determining branch conditions is also not shown, but that's not part of this problem.) The IF-stage memory port can retrieve any 4-byte aligned address. (That is, it does not have the stricter 8 -byte alignment that is assumed by default for 2 -way superscalar processors presented in class.)

(a) Add hardware so that the correct branch target is computed for branches in either slot. The following signals are available: br_slot_0, which is 1 if the instruction in Slot 0 (ir \({ }^{0}\) ) is a branch; br_slot_1, which is 1 if the instruction in Slot 1 is a branch. Assume that there will never be a branch in both Slot 0 and Slot 1.

Design the hardware for low cost. Hint: Adder carry-in inputs can come in handy. The goal of this part is to generate the correct target address, the next part concerns what is done with it.
(b) Add datapath so that the branch target address can be delivered to the PC at the correct time, whether the branch is in Slot 0 or Slot 1. (Earlier in the semester branch delay slots were given as an example of an ISA feature that worked well for the first implementations but that would become a burden in future ones. Welcome to the future.)

Problem 1: Solve Spring 2012 Final Exam Problem 3, in which pipeline execution diagrams are requested for some superscalar systems.

Problem 2: Solve Spring 2012 Final Exam Problem 6 (d) and (e). (Just those two.) These questions concern the techniques of widening (superscalar designs) and deepening (more pipeline stages) our implementation to exploit more instruction-level parallelism.

Problem 3: Solve Spring 2012 Final Exam Problem 4, which asks for performance information about some branch predictors.

\section*{12 Spring 2012}

Problem 1: To save space in a program an array is designed to hold four-bit unsigned integers instead of the usual 32 -bit integers (it is known in advance that their values are \(\in[0,15]\) ). Because this 4 -bit data size is less than the smallest MIPS integer size, 8 -bits, even a load byte instruction will fetch two array elements. Code to read such an array and a test routine appear on the next page, along with a stub for code to write the array. The routine compact_array_read is used to read an element of this array and compact_array_write is the start (mostly comments) of a routine to write an element.
(a) Add comments to compact_array_read appropriate for an experienced programmer. The comments should describe how instructions achieve the goal of reading from the array. The comments should not explain what the instruction itself does, something an experienced program already knows. See the test code for examples of good comments.
(b) Complete the routine compact_array_write, so that it writes data into the array. See the comments for details.
```

\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#

## 

## Test Code

## 

a: \# Array of values to test. Each byte hold two 4-bit elements.
.byte 0x12, 0x34, 0x56
msg: \# Message format string (similar to printf).
.asciiz "Value of array element a[%/s0/d] is 0x%/s3/x\n"
.text
.globl __start
__start:
addi \$s2, \$0, 4 \# Last index in array a.
addi \$s0, \$0, 0 \# Initialize loop index.
LOOP:
la \$a0, a \# First argument, address of array.
jal compact_array_read
addi \$a1, \$s0, 0 \# Second argument, index of element to read.
la \$a0, msg \# Format string for test routine's msg.
addi \$s3, \$v0, 0 \# Move return value (array element) ...
addi \$v0, \$0, 11 \# ... out of \$v0 and replace with 11 ...
syscall \# ... which is the printf syscall code.
bne \$s0, \$s2 LOOP
addi \$s0, \$s0, 1 \# Good Comment: Advance index to next
\# element of test array.
\# Bad Comment: Add 1 to contents of \$s0.
li \$v0, 10 \# Syscall code for exit.
syscall
\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#

## 

## compact_array_read

## 

compact_array_read:
\#\# Register Usage
\#
\# CALL VALUES
\# \$a0: Address of first element of array.
\# \$a1: Index of element to read.
\#
\# RETURN VALUE
\# \$v0: Array element that has been read.
\#
\# Element size: 4 bits.
\# Element format: unsigned integer.
srl \$t0, \$a1, 1
add \$t1, \$a0, \$t0
andi \$t3, \$a1, 1
bne \$t3, \$0 SKIP
lb $t2, 0($t1)
jr \$ra
srl \$v0, \$t2, 4
SKIP: jr \$ra
andi \$v0, \$t2, 0xf

```
```

\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#

# 

# compact_array_write

# 

compact_array_write:
\#\# Register Usage
\#
\# CALL VALUES
\# \$a0: Address of first element of array.
\# \$a1: Index of element to write.
\# \$a2: Value to write.
\#
\# RETURN VALUE
\# None.
\#
\# Element size: 4 bits.
\# Element format: unsigned integer.

```

Problem 2: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.


LOOP:
srl r4, r3, 2
sw r4, O(r3)
bne r3, r2 LOOP
addi r3, r3, 4
(a) Show a pipeline execution diagram for the code above on the illustrated implementation for enough iterations to determine CPI.

Problem 1: Solve Problem 1 of the EE 4720 Spring 2011 Midterm Exam. (Execution on a bypassed implementation.)

Problem 2: Solve Problem 2 of the EE 4720 Spring 2011 Midterm Exam. (Add hardware to implement jalr and jr.

Problem 1: Solve Problem 6 of the EE 4720 Spring 2011 Midterm Exam. These are short-answer questions about ISA, the first question is about ISA variations and starts by listing MIPS and IA-32 variants.

Problem 2: Solve Problem 7 (b) of the EE 4720 Spring 2011 Midterm Exam. This question is about a parallel universe much like our own except that MIPS ipa called ALT-MIPS and it wapa (that's the conjugated form of to be to use when describing events in a parallel universe) designed for a four-stage pipeline.

Problem 1: Solve Problem 3 of the EE 4720 Spring 2011 Final Exam. This is a problem on branch prediction accuracy.

\section*{13 Spring 2011}

Problem 1: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles.

```

    lw r2, O(r5)
    LOOP:
lw r1, O(r2)
lw r3, O(r1)
sw r3, 4(r2)
bne r3, r0 LOOP
addi r2, r3, 0

```
(a) Show a pipeline execution diagram for enough iterations to determine the CPI. Compute the CPI for a large number of iterations.
(b) Show when each bypass path is used. Do so by drawing an arrow to a multiplexor input and labeling it with the cycles in which it was used and the register. For example, something like \(\mathrm{C} 10 / \mathrm{r} 9 \longrightarrow\) to show that the input is used in cycle 10 carrying a value for r 9 .

Problem 2: Continue to consider the pipeline and code from the previous problem. The store instruction and the branch could both benefit from a new bypass connection.
(a) Show a new bypass connection for the store.
(b) Indicate the impact of the new store bypass connection on critical path length.
(c) Show a new bypass connection needed by the branch.
(d) Indicate the impact of the new branch bypass connection on critical path length.
(e) Suppose that the cost of the two bypass connections were equal and that both had no critical path impact. If only one could be added to an implementation which would you add? Base your answer not on the example code above, but on what you consider to be typical programs.

Problem 1: Solve Fall 2010 Final Exam Problem 1.
Problem 2: Solve Fall 2010 Final Exam Problem 3.

Problem 1: Solve Fall 2010 Final Exam Problem 2(a) and 2(b) (but not 2(c)).
Problem 2: Solve Fall 2010 Final Exam Problem 6 (all parts).

Updated, Problem 2 below now refers to the correct final exam problem.
Problem 1: Solve Fall 2010 Final Exam Problem 4.
Problem 2: This problem is based on Fall 2010 Final Exam Problem 7(a).
(a) Solve Fall 2010 Final Exam Problem 7(a).
(b) Illustrate your answer above with a code fragment that includes a loop. The loop should have what we will call a body branch inside the loop body and a loop branch at the end, going to the top of the loop. The loop branch is predicted with \(100 \%\) accuracy. Consider cases in which the body branch is predicted with \(95 \%\) accuracy and \(96 \%\) accuracy. Estimate the iteration time when the body branch is taken and not taken, and when it's predicted correctly or not. Do this both for a 2 -way and 4 -way superscalar system. Based on these numbers estimate performance for a \(95 \%\) prediction accuracy of that branch and a \(96 \%\) prediction accuracy.

\section*{14 Fall 2010}

Problem 1: Diagnose or fix the MIPS-I problems below.
(a) Explain why the code fragment below will not complete execution. Fix the problem, assuming that the load addresses are correct. (Problems such as this occur when operating on data prepared on a different system.)
```

lw r1, O(r2)
lw r3, 6(r2)

```
(b) The code below will execute, but it looks like there might be a bug. Explain.
```

jal subroutine
add r31, r0, r0

```
(c) The two fragments below are almost but not quite MIPS-I. Re-write them using MIPS instructions so they accomplish what the programmer likely intended.
```


# Fragment 1

lw r1, O(r2+r3)

# Fragment 2

bgti r1, 101 target
nop

```
(d) The code fragments below are correct, but not as efficient as they could be. Re-write them using fewer instructions (and without changing what they do).
```


# Fragment 1

addi r1, r0, 0xaabb
sll r1, r1, 16
ori r1, r1, 0xccdd

# Fragment 2

add r1, r0, r0
addi r1, r1, 123

```

Problem 1: Consider the execution of the code fragments below on the illustrated implementation.

- A value written to the register file can be read from the register file in the same cycle. (For example, if instruction \(A\) writes r1 in cycle \(x\) (meaning \(A\) is in WB in cycle \(x\) ) and instruction \(B\) is in ID in cycle \(x\), then instruction \(B\) can read the value of \(r 1\) that \(A\) wrote.)
- As one should expect, the illustrated implementation will execute the code correctly, as defined by MIPS-I, stalling and squashing as necessary.
```

LOOP:
lw r3, O(r1)
add r4, r4, r3
bne r1, r2 LOOP
addi r1, r1, 4
xor r7, r8, r3
sw r4, 16(r5)

```
(a) Show a pipeline execution diagram for this code running for at least two iterations.
- Carefully check the code for dependencies, including dependencies across iterations.
- Base timing on the illustrated implementation, pay particular attention to how the branch executes.
(b) Find the CPI for a large number of iterations.
(c) How much faster would the code run on an implementation similar to the one above, except that it resolved the branch in EX instead of ME? Explain using the pipeline execution diagram above, or using a new one. An answer similar to the following would get no credit because "should run faster" doesn't say much: A resolution of a branch in EX occurs sooner than ME so the code above should run faster.. Be specific, and base your answer on a pipeline diagram.

Problem 2: Apologies in advance to those tired of the previous problem. Consider the execution of the code below on the implementation from the last problem. The code is only slightly modified.
(a) Show a pipeline execution diagram for this code, and compute the CPI for a large number of iterations. It should be faster.
```

LOOP:
add r4, r4, r3
lw r3, O(r1)
bne r1, r2 LOOP
addi r1, r1, 4
add r4, r4, r3
sw r4, 16(r5)

```
(b) How much faster would the code above run on the implementation that resolves branches in EX (from the previous problem)?
(c) Suppose that due to critical path issues, the resolve-in-EX implementation had a slower clock frequency. Let \(\phi_{\text {ME }}\) be the clock frequency of the resolve-in-ME implementation (the one illustrated), and \(\phi_{\mathrm{EX}}\) be the clock frequency of the resolve-in-EX implementation. Find \(\phi_{\mathrm{EX}}\) in terms of \(\phi_{\mathrm{ME}}\) such that both implementations execute the code fragment above in the same amount of time. That is, find a clock frequency at which the benefit of a smaller branch penalty is neutralized by the lower clock frequency on the code fragment above.

Problem 1: Note: Problems like this one have been assigned before. Please solve this problem without looking for a solution elsewhere. If you get stuck ask for hints. Copying a solution will leave you unprepared for exams, and will waste your (or your parents') hard-earned tuition dollars. A shift unit is to be added to the EX stage of the implementation below. The shift unit has a 32-bit data input, VIN, a 5 -bit shift amount input, AMT, a 1-bit input SIN, and a 1-bit control input DIR. There is a 32 -bit data output, VOUT. The DIR input determines whether the shift is left (1) or right ( 0 ). If the shift is right then the value at input SIN is shifted in to the vacated bit positions. The meaning of the other inputs is self-explanatory. For a description of MIPS-I instructions see the MIPS32 Volume 2 linked to the course references page.

(a) Connect the shift unit data inputs so that it can be used for the MIPS sll, sllv, srl, srlv, sra, and srav instructions. Assume that the ALU has plenty of slack (it is not close to carrying the critical path). (Control inputs are in the next part.)
- Be sure your design does not unnecessarily inflate cost or lower performance.
- In your diagrams be sure to use the bit ranges used, for example, 27:21, when connecting a wire to an input with fewer bits than the wire.
(b) Show the logic for control inputs DIR and SIN and any multiplexors that you added.
(c) Repeat the design of the datapath but assuming that the ALU is on the critical path and that we don't want to lower the clock frequency.

Questions in this assignment are about VAX, an ISA that was mentioned in class but for which no details were given. Use the VAX-11 Architecture Reference Manual (Cover, 1982; text, 1980), which is linked to the course references page, as a reference for this assignment. (The VAX MACRO and Instruction Set Reference Manual can be used as a secondary reference; you may also use any other resources that you can find.) Chapter and section numbers in this assignment refer to the VAX-11 manual, not to the VAX MACRO manual.

Problem 1: Compare the design goals for VAX as described in Section 1.1 to the design goals for SPARC as described in the SPARC Architecture Manual V8 Section 1.1 (also linked to the course references page).
(a) List the design goals for each architecture that are considered defining elements of the respective ISA family (CISC and RISC). Explain whether the design goals in VAX and SPARC are mutually exclusive (meaning you can't easily do both).
(b) List a feature or design goal for each ISA that is unrelated to the features of the respective ISA family. Briefly explain why it is unrelated.

Problem 2: Answer the following questions about VAX and RISC instruction formats.
(a) MIPS has three instruction formats for the integer instructions, SPARC has from three to five (depending on how you count). The VAX ISA seems to have a simpler format, according to Section 2.6 (it takes just half a page to describe). Even if the VAX format is conceptually simpler (and many would dispute that), why is it more complex in a way that is important to implementers. Hint: This is an easy question.
(b) In class each operand of a typical CISC instruction had a type and info field to describe its addressing mode. What are the corresponding VAX field names?
(c) Some RISC instructions have something like a type field, though not capable of specifying the wide range of operand types as the VAX type fields (see the previous problem). Find two examples of MIPS instructions that have an equivalent of a type field. Identify the field and explain what operand types it specifies. Hint: Consider instructions that deal with floating-point numbers.
(d) Both MIPS and SPARC have an opcode field that appears in every instruction format and some kind of an opcode extension field that appears in some of the formats. Name the opcode extension fields in MIPS and SPARC. What is the closest equivalent to an opcode extension field in VAX?

Problem 3: Find the VAX addressing modes requested in the problems below. The term addressing mode can refer to registers, immediates, as well as memory addresses.
(a) Find the VAX addressing modes corresponding to the addressing mode used by the indicated operands in each instruction below.

Name the mode, and show how the operand would be encoded in the instruction (there is no need to show the entire instruction).
```

addi r1, r2, 3 \# Both source operands.
lw r1, O(r2) \# Source operand. Note that the displacement is zero.
lw r3, 4(r4) \# Source operand

```
```

ld [11+12], 13 \# SPARC insn, source operand

```
(b) Find the VAX addressing mode that can be used in place of the three instructions below. Name the mode, and show how it is encoded.
```

sll r1, r2, 2
add r3, r1, r4
lw r5, O(r3)

```

Problem 1: Solve EE 4720 Spring 2010 Midterm Exam Problem 1. Please initially attempt to solve this problem without seeking out solutions to similar problems. If you are able to solve it that way, great!, otherwise look in the statically scheduled study guide (accessible from the course Web page) for tips on solving problems of this type and a list of similar problems with solutions.

Links in this assignment are clickable in Adobe Reader. For the questions below refer to the gcc 4.1.2 manual, available via http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/.

Problem 1: Read the introductory text to the optimization options page, 3.10, in the GCC 4.1.2 manual, and familiarize yourself with your Web browser's search function so that you can search the rest of the page. Answer the following questions.
(a) When optimizing gec tries to fill branch delay slots. What option can be used to tell gcc not to fill delay slots, without affecting other optimizations? What option can be used to control how much effort gcc makes to fill delay slots?
(b) A reason given in class for scheduling code was to avoid stalls due to a lack of bypass paths. What reason is given in the description of the -fschedule-insn option?

Problem 2: The POWER and PowerPC ISAs have alot in common, but each has instructions the other lacks. Show the gcc command line switch to compile for both, start looking in section 3.17, Hardware Models and Configurations.

Problem 3: Read the following blog post about the use of profiling in the build of the Firefox Web browser:
http://blog.mozilla.com/tglek/2010/04/12/squeezing-every-last-bit-of-performance-out-of-the-linux-toolchain/.
The post compares the results of profiling optimizations provided by gcc to those obtained using other tools for optimization.
(a) As described in the blog post, what was the training data used for profiling?
(b) Suppose that a Web page with a 5000 -row table performs just as sluggishly with the profileoptimized gcc build described in the blog post (firefox.static.pgo) as the ordinary Firefox build (firefox.stock). Provide a possible reason for this, and a solution.

Problem 4: SPEC recently ended a call for possible programs for their next CPU suite, cpuv6. Read the page describing the call: http://www.spec.org/cpuv6/.
(a) There is a section entitled "Criteria SPEC considers important for the next CPU benchmark suite." Evaluate the suitability of the pi.c program used in class based on each of these criteria.

Problem 1: Do Spring 2010 Final Exam Problem 1 (our MIPS floating-point implementation questions). The following were the criteria used when grading the final. Positive numbers indicate total points for some aspect of the solution. Negative numbers are specific deductions for mistakes. Many of these are based on specific mistakes made by one or more students. In the criteria smoke means logic in which the outputs of two gates are connected together.
```

Problem 1 ( 15 pts )
-~~~~~~-
-2 Smoke
4 Existing path for simple mtc1
-2 Squiggly line starts in MEM.
-2 Squiggly line from EX to fp RF din.
4 Control Logic
-3 Correct unbypassed mtc1, others wrong.
-3 Compare sources to integer dest. Don't check insn types.
-2 Bypass from FP add and M6 we, no test for reg\# or int val. No mtc1.
-4 and=>or of register field bit ranges, but inputs not connected.
-1 Correct test for fd, but uses fs from ID, no test for type
3 Store
4 Bypasses
-1 Store correct, other bypass could induce WF str hazard.
-3 Bypass from output of ALU to A1 input.
-2 Store bypass but no other bypass.
-O Correct mtc1 bypass, but no store bypass (and no store either).
-4 Bypass from WF, with errors. (Mux out is 3rd input; src is fd).
-3 Bypass from mem port data out to a1.
-3 Bypass from EX/MEM.ALU, but connects to 1 -input mux that smokes A1.

```

Problem 2: Do Spring 2010 Final Exam Problem 5 (how instructions given in problem could be added to MIPS). Grading criteria used in final exam (see previous problem):
```

Problem 5 (15 pts)
5 Encoding
-2 No attempt at field uniformity.
-1 Show func as zero.
-2 addsid: "immed=Mem[ ]"
5 Shift Unit Placement
-O Mux rs and rsv; implicit assumption that ALU shifts.
-2 Shift in ID or EX without mentioning critical path impact.
-3 Assume that ALU can both shift and add.
-2 slli: Use rsv for shift amount.
-4 adds: Shift in ID, next cycle bypass from me. Omit stall, etc.
-4 slli: Add a shift unit in ID stage. adds: shift before ALU, costly
-5 sllii: "Only need shifter." adds: "No hardware, use ALU for mult."
-3 Direct connection ALU out to in. "much control logic"
-0 "Control logic .. stalls .. ALU two consecutive cycles".
5 Memory unit position
-3 "Kind of" switching EX and ME.
-3 Bypass from WB, that's hard. Omits stall, bypassing sources.
-4 Bypass from WB, that's easy. Omits stall, bypassing sources.
-4 No description of connections, but does note ME is after EX.
-2 Memory port in ID, difficult. (Does not consider added stage.)
-5 Difficult because of fetch, no datapath description.
-0 "No extra dp.. reuse .. break pipe flow .. sig drops perf"

```

Problem 1: Do Spring 2010 Final Exam Problem 2 (branch prediction). The grading criteria used for the final appear below. Positive numbers indicate total credit for a category, negative numbers are specific deductions.
```

Problem 2 (20 pts)
5 Bimodal B2
-2 Accuracy based on predicting after update.
1 Local B2
4 B2 min local history size
-2 Nine, since pattern repeats every 10 executions
-2 Three, with "example"
-0 Four, with "example"
1 B2 local pht count
1 Bimodal B3
1 Local B3
1 Global B3
4 B3 min global history
-3 Three, needs $\mathrm{b} 1, \mathrm{~b} 2, \mathrm{~b} 3$
2 B3 warmup
-1 2~5, 32 * 18 cycles

```

Problem 2: Do Spring 2010 Final Exam Problem 3 using some additional information provided here: The BHT is usually indexed with a subset of PC address bits, in class we like to use 11:2 (bits 11 to 2) for a \(2^{10}\)-entry table but real systems use more bits. Even so, the entry retrieved from the BHT might not be for the branch being predicted. For example, a branch at PC \(0 \times 1234\) and 0x9234 would have the same index (BHT address) bits, 0x234 (leaving the two least-significant bits on for convenience), but the higher bits, \(31: 12\), would be different: \(0 x 1\) for the first branch and \(0 x 9\) for the second. An ordinary branch predictor would treat these two branches as the same branch, with a possible loss of performance. A solution is to place some or all of the high-order PC bits in the BHT entry, along size the 2-bit counter, target, and other info. The high-order PC bits used this way are referred to as a tag. A full-size tag for MIPS-I and our default predictor would be \(31: 12\) or 20 bits. When predicting a branch the tag in the entry that is retrieved is compared to the corresponding bits in the branch being predicted. If they match the BHT lookup is said to hit, if they don't match it is said to miss. Tags, along with target and other data, are stored when the predictor is updated. If we were predicting \(0 \times 1234\) there would be a hit if the tag were \(0 \times 1\), if the tag were \(0 x 9\) that would be a miss. By using tags a collision between the two branches has been detected. Without tags the predictor would have to assume that the retrieved BHT entry was correct. For structures like branch predictors that don't have to always be right, it's possible to get away with fewer than full-size tags. So in this example, one could use just two tag bits and detect the collision. If one tag bit were used the tags would be the same for the example branches and so the collision would not be detected.

With the information above, solve Problem 3.
Problem 3: Do Spring 2010 Final Exam Problem 7 (short answers). Part (a) is easy, part (b) requires a small amount of thinking, parts (c) and (d) are easy.

Homework 1

\section*{15 Spring 2010}

Problem 1: Re-write each code fragment below so that it uses fewer instructions (but still does the same thing).
```


# Fragment 1

lw r1, O(r2)
addi r2, r2, 4
1w r3, O(r2)
addi r2, r2, 4

# Fragment 2

slt r1, r2, r3
blt r1, r0 TARG
add r4, r5, r6

# Fragment 3

ori r1, r0, 0x1234
sll r1, r1, 16
ori r1, r1, 0x5678

```

Problem 2: The MIPS code below runs on the illustrated implementation. Assume that the number of iterations is very large.


LOOP:
1w r3, \(0(r 1)\)
addi r2, r2, 1
beq r3, r4 LOOP
lw r1, 4(r1)
(a) Show a pipeline execution diagram with enough iterations to determine the CPI.
(b) Determine the CPI.
(c) Schedule (re-arrange) the code to remove as many stalls as possible.

Problem 3: The MIPS implementation below has three multiplexors in the EX stage.
(a) Write a program that executes without stalls and which uses the eight ALU multiplexer inputs in order (perhaps starting at cycle 3) in consecutive cycles. That is, in cycle 3 the top input of the upper ALU mux would be used, (bypass from memory), in cycle 4 the second one would be used (NPC), in cycle 5 rsv , in cycle 6 bypass from WB, in cycle 7 we switch to the lower ALU mux with the bypass from ME input, in cycle 8 rtv , etc.
(b) Explain why it would be impossible to use the EX-stage rtv mux inputs in order in consecutive cycles.

Problem 1: The SPARC jmpl (jump and link) instruction adds the contents of two source registers or a register and an immediate, and jumps to that address. It also puts the address of the instruction in the destination register (usually to be used to compute a return address). For more information, find the description of jmpl in the SPARC V8 ISA description from the references linked to the course home page.

In this problem a similar instruction (or instructions) will be added to MIPS. Like the SPARC jmpl, the MIPS variant can jump to a target determined by the sum of two registers or a register and an immediate, while the address of the instruction is saved in the destination register. (Note that the saved address is different than the address saved by MIPS' jalr and jal instructions. Be sure to save the address indicated by the SPARC definition.)
(a) Show how the MIPS version of these instruction(s) can be encoded. Show a format for the instruction, using the descriptions in the MIPS32 Architecture Volume II (linked to the references page) as an example. The format should show which instruction fields indicate each part of the instruction.
(b) Show datapath changes (that is, omit control) to the implementation below needed to implement this (these) instruction(s). The changes must fit in naturally with what is present and should not risk lowering clock frequency. Do not forget about any changes needed to save a return address.

(c) As discussed in class, a SPARC-style jmpl on something like our 5-stage pipeline would have to be resolved in EX. However, a higher-cost implementation might resolve a jmpl in ID if no addition were necessary.

Identify which of the following cases is the least trouble to detect (shown with SPARC assembler), and explain why it is the least trouble:
```

jmpl %g1, %g0, %o7 ! g0 is the zero register.
jmpl %g1, 0, %o7 ! The immediate is zero.
jmpl %g1, %g2, %o7 ! Contents of g2 is zero.

```

Problem 2: Without looking at the solution, do Fall (November) 2007 Midterm exam Problem 1. Use the Statically Scheduled MIPS study guide, http://www.ece.lsu.edu/ee4720/guides/ssched.pdf, for tips on how to solve this interesting, understanding-building, and fun-to-solve (if one is prepared and not under intense time pressure) problem. Only use the solution if you must. Warning: The test problems will be chosen under the assumption that students really solved this problem.

Problem 1: The code below executes on the illustrated MIPS implementation. Assume that any reasonable bypasses needed for the FP operands are available, even though they are not shown in the illustration. A bypass is reasonable if it does not have a significant impact on clock frequency and if it does not use circuitry that can predict the future.

(a) Show a pipeline execution diagram covering enough iterations to compute the CPI. Don't forget to check code for dependencies.
(b) Compute the CPI.
(c) Remember, that some bypass paths are assumed present though not illustrated. Add the needed paths to the implementation and show when they are used.

Problem 2: Precise exceptions are necessary for integer instructions, but only Nice To Have for floating-point instructions. Suppose exception conditions, such as overflow, were detected in A4 and M6 in the pipeline from the previous problem.
```

mul.d f2, f0, f4
add.d f6, f6, f2
and r3, r3, r5
addi r1, r1, 8

```
(a) For the code fragment above, would a mul.d exception detected in M6 be precise? Explain in terms of architecturally visible storage (register and memory values) when the handler starts. (Note that in general exceptions detected in M6 would not be precise, but the question is only asking about the fragment above.)
(b) For the code fragment above, would a add.d exception detected in A4 be precise? Explain in terms of architecturally visible storage when the handler starts.

Problem 3: The MIPS implementation below has a fully pipelined FP add unit. Replace the FP add unit with one that has an initiation interval of 2 and a total computation time of 4 cycles. Note that the time to compute a floating point sum is the same on the original and replacement adder.


The new adder has two stages, A1 and A2, each has two inputs (like their fully pipelined counterparts), and each has two outputs. In the first cycle of computation the source operands are placed at the inputs to A1, in the second cycle of computation the values at the outputs of A1 at the end of the first cycle are placed at the inputs to A1. In the third cycle the values at the outputs of A1 at the end of the second cycle are placed at the inputs A2, and in the fourth cycle the inputs to A2 are the values at the outputs of A2 at the end of the third cycle. The sum is available from the upper output of A2 at the end of the fourth cycle.
(a) Replace the FP adder datapath with the one described above.
(b) Modify the control logic for the new adder. Be sure to account for the structural hazard when there are two consecutive FP add instructions.

Problem 1: A deeply pipelined MIPS implementation is constructed from our familiar five-stage pipeline by splitting IF, ID, and ME each into two stages, but leaving EX and WB as one stage. The total number of stages will be eight, call them F1, F2, D1, D2, EX, Y1, Y2, and WB. In this system branches are resolved at the end of D2 (rather than at the end of ID). Assume that all reasonable bypass paths are present.
(a) Provide a pipeline execution diagram of the code below for both the 5 -stage and this new implementation, for enough iterations to compute the IPCs.

LOOP :
addi r2, r2, 4
lw r1, \(0(r 2)\)
add r3, r3, r1
bne r5, r4 LOOP
addi r5, r5, 1
(b) Suppose the 5 -stage MIPS runs at 1 GHz . Choose a clock frequency for the 8 -stage system for which the time to execute the code above is the same as for the 5 -stage MIPS.
(c) Consider two ways to make a 7 -stage system from the 8 -stage system. In method ID, the two ID stages (D1 and D2) are merged back into one (or if you prefer, the ID stage was never split in the first place). In method ME, the two ME stages (Y1 and Y2) are merged back into one (or were never split).

Which method is better, and why? Assume that the eight-stage system runs at 1.8 GHz . Consider both the likely impact on clock frequency (remembering that you are at least senior-level computer engineering students) and the benefit for code execution (don't just consider the code above, argue for what might be typical code).

Problem 2: Itanium is a VLIW ISA designed for general-purpose use. Being a VLIW ISA (as defined in class) its features were chosen to simplify superscalar implementations. The questions below are about such features, read the Intel Itanium Architecture Software Developer's Manual Volume 1, Section 3 for details and concentrate on Sections 3.3 and 3.4. The manual is linked to the course references page, http://www.ece.lsu.edu/ee4720/reference.html. Use this copy to be sure that section and table numbering used here match.
(a) Section 3.3 mentions four types of functional unit, and where an instruction using a particular unit can be placed in a bundle (see Table 3-9 and 3-10).

Suppose an Itanium implementation fetches one bundle per cycle. Indicate the maximum number of execution units of each type needed. (That is, there would be no advantage of having more than this number.) Assume that the units all have latency 1 or else are fully pipelined.
(b) For this problem suppose the implementation had the minimum number of units of each type, one. Sketch the pipeline execute stages, and show connections to each of the FU inputs. There should be three sets of source operands flowing down the pipeline. Some (or all) of the execute units should have multiplexors at their inputs to select operands from one of the three instructions in a bundle. Show the multiplexors, and based on the slot restrictions show the minimum number of inputs.
(c) Notice in Table 3-10 that there is no template with a stop right after Slot 0 and right after Slot 1. Provide a possible reason for this.

Suppose there was a template with two such stops (as described above), call this ISA Itanium-stop-stop. Why might code compiled for Itanium-stop-stop be smaller than code compiled for Itanium?

Consider Itanium and Itanium-stop-stop implementations that fetch one bundle per cycle (same as in the prior problems). Explain why Itanium-stop-stop might be no faster than Itanium.

Problem 1: Do Spring 2009 final exam Problem 2.
Problem 2: Do Spring 2009 final exam Problem 3, the branch predictor problem.
Problem 3: Do Spring 2009 final exam Problem 5.

\section*{\(16 \quad\) Spring 2009}

Homework 1
Due: 27 February 2009

Problem 1: Answer each question.
(a) Explain why the code below won't finish running.

LOOP:
lw r1, O(r2)
xor r3, r3, r1
bne \(r 2, r 4\) LOOP
addi r2, r2, 2
(b) Shorten the code below.
```

lui r1, 0x1234
ori r1, r1, 0x5678
lw r1,0(r1)

```
(c) Shorten the code below.
```

xor r1, r2, r3
beq r1, r0 TARG
addi r4, r4, 1

```

Problem 2: Consider the execution code below on the illustrated implementation.
```

LOOP:
lw r2, 0(r4)
slt r1, r2, r3
beq r1, r0 LOOP
addi r4, r4, 4

```

(a) Determine the execution rate in IPC (instructions per cycle) assuming a large number of iterations. Use a pipeline execution diagram to justify your answer. (No credit without one.)
(b) If the previous part was solved correctly there should be a stall due to the branch. Add a bypass path to avoid the branch stall.
(c) Why might the added bypass path impact clock frequency?
(d) Suppose the clock frequency of the original pipeline were 1 GHz , and call the clock frequency of the added-bypass implementation \(\phi\). For what value of \(\phi\) will the run time of the code fragment be the same on the original and added-bypass implementations (assuming a large number of iterations).
(e) Suppose a blt (branch less than) instruction was available that could compare two registers (not just a register to zero). Re-write the code above for this instruction and add bypasses that are no worse than the added-bypass bypass. How would the performance of this blt implementation on the re-written code fragment compare to the added-bypass implementation on the original code fragment? Assume both systems have the same clock frequency.

Problem 1: Solve Fall 2008 EE 4720 midterm exam Problem 1. Solutions have not yet been posted but were given in class in the Fall 2008 semester. Do not look at any solution you might come across. (Solutions will be posted after the homework is collected.) Hints: This sort of problem looks alot harder than it actually is. For a solved problem of this type see the solution to Problem 1 in the Spring 2008 midterm exam. Also look at the Fall 2008 Homework 2 for more on the shift unit.

Problem 1: Consider three ISAs: IA-32 (or 64), IBM POWER6, and Sun SPARC.
(a) Choose the highest-performing system, based on SPECint2006, for each ISA. Print or provide a link to the test disclosure for each one.
(b) Find examples of benchmarks (if any) which favor each ISA based on the test disclosures you found.

Problem 2: Why might a company publish peak SPECcpu scores but not base? Why is it against the rules?

Problem 3: Solve the Fall 2008 Final Exam Problems 1 and 2.

Problem 1: Solve Fall 2008 Final Exam Problem 4 and the additional questions below.
(a) For part (a) provide pipeline execution diagrams for the three systems ( 5 -stage scalar, \(n\)-way superscalar, and \(5 n\)-stage superpipelined) running code of your choosing. Refer to these diagrams when answering part (a).

Problem 2: Consider the three systems from Problem 4 in the final exam. The problem focused on potential (favorable) execution time, which can be achieved when there are few stalls, here we'll be more realistic.
(a) Which system will suffer more stalls on typical code? Explain.
(b) Invent a quantitative measure of implementation (not program) stall potential and apply it to the three systems. The answer should include a formula for each system (giving the stall potential); the superscalar and superpiplined formulas should be in terms of \(n\). Hint: think about the average or minimum distance between two dependent instructions needed to avoid a stall. The formulas should be consistent with your answer to the first part.

Problem 1: Solve Fall 2008 Final Exam Problem 3.
Problem 2: Continue to consider the systems and code from Problem 3.
(a) What is the warmup time of the local predictor on branch B2?
(b) What is the warmup time of the global predictor on branch B2?

Problem 3: Continuing still with Problem 3, suppose the number of iterations of the B1 loop could be 1,2 , or 3 , the probability of each number of iterations is \(\frac{1}{3}\) and the number of iterations is independent of everything. The patterns of B1 for an iteration of BIGLOOP can thus be N or T N or T T N.
(a) What is the accuracy of the bimodal predictor on B1. An exact solution is preferred but an approximate solution is acceptable. Hint: Model the effect of the change of one BIGLOOP iteration on the counter using a Markov chain, something you may have learned about in other courses.
(b) How will B1's behavior impact the accuracy of the local predictor on branch B2? Show an example of execution that would result in a B2 misprediction and compute the probability of that particular execution.
(c) Optional: Find the exact prediction accuracy of B2 on the local predictor with B1's new behavior. This may be very difficult so don't spend too much time on it.

\section*{17 Fall 2008}

To answer the first question below see the MIPS32 Architecture manual linked to the course references page.

Problem 1: The MIPS I bgtz and bltz instructions compare a register to zero, but can't compare two registers (unless the second one is the zero register). Consider an extension of MIPS I that allowed branch greater than and branch less than instructions to compare two registers, call the new instructions bgt and blt. Explain why the existing bgtz opcode could be used for bgt but why the bltz opcode could not be used for blt. Hint: See bltzal.

Problem 2: A C function and a part of a MIPS equivalent are shown below. The C function looks at the attributes of a car and decides what to pack in a promotional giveaway to the car buyer. The assembler code corresponds to the C function up until the last line (checking for a sun roof).
```

\#define FE_SPORTY 0x1
\#define FE_OFF_ROAD 0x2
\#define FE_EFFICIENT 0x4
\#define FE_SUN_ROOF 0x10000
\#define FE_MANUAL_TRANSMISSION Ox20000
enum Giveaways { G_Food, G_Hiking_Boots, G_Sunblock, G_Driving_Gloves };
void prepare_promotion_package(Car_Object *car) {
int car_features = car->features;
if ( car_features \& FE_OFF_ROAD ) pack(car, 1200, G_Hiking_Boots);
if ( car_features \& FE_SUN_ROOF ) pack(car, 200, G_Sunblock);
}

# MIPS-I Equivalent of C code.

# 

# \$a0: Address of car object.

# Notes: Procedure call arguments placed in \$a0, \$a1, ...

# Assume that pack does not change $a0-$a3 or $s0-$s7

    lw $s0, 16($a0) # Load the features bit vector of car object.
    andi $t0, $s0, 2
    beq $t0, $0 SKIP1
    addi $a1, $0, 1200
    jal pack
    addi $a2, $0, 1
    SKIP1:

# PART b SOLUTION STARTS HERE

```
(a) The MIPS code above omits the last line of C code (checking for a sun roof); complete it using MIPS I instructions. (Do this on paper, there is no need to run it.) Hint: A clever solution uses five instructions a straightforward solution uses six instructions. If you have more than ten instructions ask for help.
(b) Add comments to the assembler code above. Write the comments for an experienced MIPS and C programmer, that is, the comments should describe what an instruction is doing in terms of what the \(\mathbf{C}\) code is trying to do. The comments should not just describe how instructions change register values.

For example, a bad comment for the 1 w instruction would be: Compute address \(16+\$ \mathrm{aO}\), retrieve word starting at that address and write into \(\$\) s0. This is a bad comment because an experienced MIPS programmer already knows what an lw instruction does. The comment for 1 w in the code (Load the features...) is good because it tells the reader what the \(\$ \mathbf{s} 0\) value is in terms of what the code is supposed to do.

Problem 3: Consider the code from the previous problem. Invent a new branch instruction that can be used for the kind of branching used in the code: testing if a single bit in a register value is 1.
(a) Show the encoding for the new branch instruction. The new instruction must fit as naturally as possible with other instructions.
(b) Compare the implementation cost and performance of the new instruction to the existing MIPSI bltz and to a hypothetical blt instruction. (With each instruction doing its own thing, not as part of functionally equivalent alternatives.)

Problem 4: Solve Fall 2007 Homework 2 without looking at the solution. Then look at the solution and give yourself a grade on a scale of \([0,1]\). Warning: test questions are based on the assumption that homework problems were completed, so make a full effort to solve it without first consulting the solution.

Problem 1: The hardware needed to implement shift instructions, such as sll, is not shown in the implementation below. (The ALU in the implementation below does not perform shift operations.) Add a separate shift unit to the implementation to implement the MIPS sll and sllv instructions. The shift unit has a shift amount input and an input for the value to be shifted.
- Show exactly where the shift-amount bits come from (including bit positions).
- Add bypass paths so that the code below can execute without a stall.
- The primary goal is to not slow the clock frequency, the secondary goal is to minimize added cost. This might affect where multiplexers are placed.
```

sll r1, r2, 3
add r3, r1, r4

```


Problem 2: To answer this question see the SPARC Joint Programming Specification, a description of the SPARC V9 ISA, linked to the course references page. The SPARC V9 ISA is naturally big endian. Since many programs must read data using little-endian byte order, for example when reading a binary data file that was produced on a little-endian system, the programs need some way to get the data into big-endian order. If loading little-endian data were only a small part of what a program did then it could get by with some combination of ordinary instructions to convert the data to big-endian format. For programs spending substantial time reading little-endian data even a 9 -instruction sequence may take too long.

The first instruction below is an ordinary load in SPARC V9, a 64 -bit ISA (in which addresses and registers are 64 bits). The second instruction, ldxle, is made up; it's a load that assumes data
is in little-endian byte order. The last instructions is a real SPARC instruction for loading little endian data.
```

! All load instructions below load 8 bytes into a register.
! Registers are 64 bits.
ldx [%l1], %12 ! Ordinary load. For big-endian data.
ldxle [%11], %12 ! Not a real SPARC insn. For little-endian data.
ldxa [%l1] 0x88, %12 ! SPARC's load for little-endian data.

```
(a) The ldxa instruction is an example of an alternate load instruction. The alternate load instructions are intended for three kinds of access. Briefly describe the three kinds and indicate which one is used above. What symbolic name does JPS1 give for 0x88 above?

Note: To answer this question one must read through material dealing with topics not yet covered, for example, the concept of multiple address spaces. It is only necessary that the concept of multiple address spaces is vaguely understood. The kind of access done by the ldxa should be clearly understood.
(b) Show the encoding for the three instructions above. The ldx and ldxa are real instructions, so it's just a matter of looking things up. For the ldxle make up an appropriate encoding.

Homework 3
Due: 29 October 2008

Problem 1: Two MIPS implementations appear below, the first is the one presented in class, it will be called the mux-in-EX implementation. The second, the mux-in-ID implementation, has the ALU input multiplexers in the ID stage, to better balance critical paths. The clock frequency of the mux-in-EX implementation is 1 GHz and the clock frequency of the mux-in-ID implementation is 1.1 GHz .

(a) With this change some of the ALU multiplexer inputs are unnecessary. Show which inputs are unnecessary and explain why.
(b) The code below computes the sum of the low 12 bits of elements in an integer array. Compute the performance, in array elements per second, of this code for both the mux-in-EX system and the mux-in-ID system. Assume that the array size is large and that the number of array elements is even.
```


# Call Values

# \$a0, address of start of array of four-byte integers.

# \$a1, number of elements in array. Assume > 0 and multiple of 2.

# Return Value

# \$v0, sum of low 12 bits of integers in array.

    sll $t1, $a1, 2
    add $t1, $a0, $t1
    addi $t1, $t1, -4
    LOOP :
lw $t0, O($a0)
lw $t5, 4($a0)
andi \$t2, \$t0, Oxfff
add \$v0, \$v0, \$t2
andi \$t7, \$t5, Oxfff
add \$v0, \$v0, \$t7
bne \$a0, \$t1 LOOP
addi \$a0, \$a0, 8
jr \$ra
nop

```
(c) If, after double-checking your work, the performance of the mux-in-ID system is faster than the old mux-in-EX system inform the professor that there is a mistake in this problem. Otherwise, schedule (re-arrange instructions) the code above so that it performs faster (while still performing the same computation) on the mux-in-ID system.

Problem 2: You are in an alternate universe where you work for MIPS at a time when its first implementation (mux-in-EX) has been very successful and is in the hands of customers of all types. You are deciding on whether to make mux-in-ID the second implementation to be marketed.
(a) What role do compiler writers have in the success of mux-in-ID? Explain.
(b) If mux-in-ID is faster than mux-in-EX using the old compilers, do compilers still need to be re-written? Explain.

\section*{18 Spring 2008}

Problem 1: Solve Fall 2007 Homework 2 without looking at the solution. Then look at the solution and give yourself a grade on a scale of \([0,1]\). Warning: test questions are based on the assumption that homework problems were completed, so make a full effort to solve it without first consulting the solution.

Problem 2: The MIPS IV movn instruction is an example of a predicated instruction (predication will be covered later in the semester, but that material is not needed to solve this problem).
(a) Show how the movn instruction could be added to the implementation below inexpensively, but without impact on critical path. Take into account the new logic's impact on dependency testing (see the code sample below). Show all added control logic.

(b) Show how the code below would execute on your implementation.
```

add r1, r2, r3
movn r4, r5, r1
xor r6, r4, r7

```
(c) Suggest methods to eliminate any stalls encountered.

\section*{LSUU ĒE \(47 \overline{7} 0\)}

Homework 2
Due: 29 February 2008
For the answers to these questions look at the ARM Architecture Reference Manual linked to the course references page, http://www.ece.lsu.edu/ee4720/reference.html.

Problem 1: The register fields in ARM instructions are four bits and so only 16 integer registers are accessible. The ISA manual describes ARM as having 32 integer registers, however many of them are only accessible in particular modes.

An advantage of fewer registers is that extra bits are available in the instruction encoding, for example, ARM three-register instruction formats would have three more bits available than the MIPS type R format. Where in the ARM formats do you think these bits went? In your answer give the instruction field and its purpose. There should be no equivalent in MIPS.

Problem 2: In MIPS an arbitrary 32-bit constant can be loaded into a register using a lui followed by an ori. In ARM the immediate field for data-processing (integer) instructions is only 8 bits.
(a) Show ARM code to put an arbitrary 32 -bit constant into a register without using a load instruction. Use as few instructions as possible. Hint: take advantage of ARMS shift and rotate capabilities.
(b) Show how ARM can put an arbitrary constant into a register with one load instruction, whereas in MIPS two would be required. The MIPS code is shown below. Do not assume the address of the constant is already in a register, that would make this problem insultingly easy! Hint: Use one of ARM's special purpose registers.
```

.text
lui r1, 0x1111
lw r1, 0x2220(r1)

# ... a few more instructions ..

jr \$ra
nop
.data
my_32_bit_constant: \# Address: 0x11112220
.word 0x12345678

```

Problem 3: In ARM the program counter is register r15, and so as far as instruction encoding goes, is treated as a general-purpose register.
(a) Why would really keeping the program counter in the integer register file add to the cost of an implementation?
(b) How does the ISA manual hint that blue parts of the implementation below is what they had in mind? (Register r15 is not stored in the register file, it will always be bypassed from the real PC.) (Note: The ARM implementation is far from complete and parts may not work.)


For answers to these questions consult the SPECcpu2006 Run and Reporting Rules (which can be found at spec.org).

Problem 1: One way testers can stretch the rules is by using compiler optimizations that give good performance when they work correctly but are too error prone for non-experimental use.
(a) Why would it be a bad idea for SPEC to limit allowable compiler optimizations to those that are already known to be safe? (Say, dead-code elimination based on a SPEC-provided analysis technique.)
(b) Rather than dictate allowable optimizations the rules instead explain that if it's good enough for your customers it's good enough for SPEC, though not in those words. Find the section in the run and reporting rules where this rule is given.
(c) For at least three bullet items in the section (from the last part) explain what sort of unscrupulous action the bullet item is supposed to prevent.

Problem 2: When preparing a run of the SPEC benchmark the tester provides, among other things, libraries (such as the C standard library that contains routines such as strlen, malloc, printf). It is in the testers interest to make sure these library routines run as fast as possible and is free to do so within the SPEC rules.

Section 2.1.2 stipulates that one can't use flags that substitute library routines for routines defined in the benchmark.

In addition to base and peak, imagine a third metric called swap, in which the rule in Section 2.1.2 didn't apply. Testers could abuse the swap metric by substituting routines that merely return the correct value (since input data is known in advance), but for this question suppose testers comply with the spirit of the SPEC rules and substitute routines which provide higher performance for any input data.
(a) Comparing the peak scores to the base scores shows the additional performance that can be obtained by a suitably motivated and resourced expert. Explain what might be learned by comparing swap scores to base and peak scores. (That is, where might the higher performance be coming from.)
(b) Provide an argument that the swap metric is a good test of a system that complements base and peak.
(c) Provide an argument that swap doesn't really tell you anything about the system (CPU, memory, compiler and other build items).

Problem 3: For exceptions the handler needs to know the address of the faulting instruction both so that it can examine the instruction and so that it knows where to return to in case the instruction needs to be re-executed or skipped. For answers to this question consult the ARM and MIPS32 (Volume 3) ISA manuals on the course references page.

A programmer-friendly ISA would provide the handler with the address of the faulting instruction, however in both MIPS32 and ARM may provide an address near the faulting instruction.
(a) In which registers do MIPS and ARM A32 write the approximate faulting instruction address? (For MIPS give the register number as well as its name.)
(b) The address that MIPS provides may be that of the faulting instruction, or it may not be. When is this done, and what is the other address?
(c) ARM A32 also does not provide consistent addresses. What addresses does it provide? Give a credible reason for the differences in addresses.

Problem 1: Solve the Fall 2007 Final Exam problem 1 (floating-point hardware).
Problem 2: Solve the Fall 2007 Final Exam problem 2 (branch prediction).

\section*{19 Fall 2007}

For lecture material relevant to this assignment see
http://www.ece.lsu.edu/ee4720/2007f/lsli06.pdf. For some background and a list of similar problems see the statically scheduled study guide, http://www.ece.lsu.edu/ee4720/guides/ssched.pdf. Please make an effort to solve this problem based on an understanding of the material, use the solution to similar problems (if any) only for hints. Feel free to ask questions using the forums, E-mail, or in person. Exam problems will be based on the assumption that students completed (really completed) homework assignments, so don't short-change yourself!

Problem 1: Consider the following MIPS code and implementation:
\begin{tabular}{lll} 
\# Cycle & 0 & 1 \\
lw r2, 0(r10) & IF ID EX \\
LOOP: & & \\
lw r1, 0(r2) & & IF ID \\
add r3, r1, r4 & & \\
sw r3, 4(r2) & & \\
bne r3, r5 LOOP & & \\
addi r2, r2, 8 & & \\
\# Cycle & 0 & 1
\end{tabular}

(a) Complete the pipeline execution diagram of the execution of the code above on the implementation illustrated for at least the first two iterations. (See the next part for instructions on the "A:", etc.)
(b) After the addi instruction three labels are shown, A:, B:, and C:; similar labels are shown, in blue and circled, in the implementation. On the pipeline execution diagram show the values on the wires (which are multiplexor inputs) that those labels point to only in cycles in which those signals are used. The values are already shown for cycles 0,1 , and 2 . Signals A and B are used in cycle 2 (but not 0 or 1 ), signal C is not used in cycles \(0-2\).

Note that the multiplexor inputs are numbered from the top starting at zero.
(c) Find the CPI of this loop on the illustrated implementation for a large number of iterations.
(d) Add bypass connection(s) so that the loop above executes as quickly as possible. Show the CPI with those connections.
(e) Even with bypass connections the loop above, regrettably, executes with stalls (or at least it should!). Schedule (re-arrange) the code so that it executes without stalls. The scheduled loop should still load and store one value per iteration. Minor changes to the code can be made, such as changing register numbers and immediate values.

\section*{LSU EE 4720}

Homework 3
Due: 15 October 2007
The problems below ask about VAX instructions, which were not yet covered in class. For information on these instructions see the VAX Macro and Instruction Set manual linked to the EE 4720 references page.

Problem 1: The VAX locc instruction finds the first occurrence of a character in a string (see example below). The first operand specifies the character to find (A in the example), the second operand specifies the length of the string (in register r 2 ), and the third operand specifies the address of the first character of the string (register r3 below).
```


# Find first occurrence of 65 (ASCII A) in memory starting at

# address r3 and continuing for the next r2 characters.

locc \#65, r2, (r3)

```
(a) Show how the sample instruction above is encoded. Include the name of each field and its value for the example above, not for the general case. In the original assignment the third argument was shown as r3, not (r3) which is correct.
(b) Provide an example of locc in which the encoded second and third operands each require more space than the example above. At least one of these operands should use a memory addressing mode that is not available in MIPS. Show the instruction in assembler and show its encoding.

For the problems below consider a MIPS implementation similar to the one illustrated below and a DFequivalent VAX implementation. Like the MIPS implementation, the DF-equivalent VAX implementation can read two registers per cycle, write one register per cycle, perform one ALU operation per cycle, and one memory operation per cycle (not including fetch). The DF-equivalent VAX implementation may or may not be pipelined and regardless does not suffer any kind of penalty for the complexity and size of its control logic. Assume that the DF-equivalent VAX takes one cycle to fetch an instruction and one cycle to decode an instruction, regardless of the instruction's size or complexity.

Unlike MIPS the DF-equivalent VAX may be able to simultaneously use its ALU and memory port for the same instruction (in the illustrated MIPS implementation they would be for two different instructions). The 2-read, 1-write register restriction only applies to registers defined by the ISA. As with MIPS pipeline latches, the DF-equivalent VAX can read or write as many temporary registers per cycle that it needs.

When showing the execution of an instruction on the DF-equivalent VAX use something like a pipeline diagram and explain what's going on when things aren't clear. For example, here is how an add instruction might execute:
```


# Note: Destination is rightmost register (r3)

Cycle 0}0
add 123(r1), (r2)+, r3 IF ID EX ME ME EX WB
EX WB
sub IF ID EX
Cycle 2: EX: 123 + r1
Cycle 3: ME load (123+r1)
Cycle 4: ME: load (r2)
Cycle 4: EX: r2 + 4
Cycle 5: EX: add (123+r1) + (r2)
Cycle 5: WB: wb r2+4 to r2
Cycle 6: WB: WB sum to r3.

```

In the example above the add instruction can be said to have taken four cycles since that's how long the sub might have had to wait to execute (to avoid overlap).

Use the following MIPS implementation for comparison:


Problem 2: The MIPS jal instruction supports a procedure call by saving a return address in r31, other activities normally done on a procedure call, such as saving registers to the stack, must be performed using additional MIPS instructions. In contrast the VAX calls instruction not only saves a return address but also saves registers in the stack and performs other common activities.

MIPS and VAX examples are shown below in which the VAX code uses a calls instruction and the MIPS code performs a roughly equivalent operation. In particular, in both code samples three registers must be saved on the stack. (The calls instruction performs additional actions, but for this problem assume it does only what the MIPS code shows.)
(a) Show how the calls instruction would execute in the DF-equivalent VAX implementation. Note that the calls instruction reads the word at the beginning of the called routine to determine which registers to save.
(b) Is the DF-equivalent VAX implementation substantially faster on this instruction, about the same, or slower?
```


# VAX VAX VAX Vax vax vax vax vax Vax vax vax vax vax vax vax vax

    calls $O, myroutine
    myroutine:
.data
.word 0x046
xor ...

# MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS

    jal myroutine
    myroutine:
sw $r6, 0x18($fp)
sw $r2, 0x8($fp)
sw $r1, 0x4($fp)
addi \$fp, \$sp, 0
xor ...
myroutine:

# Cycle 0

    sw $r6, 0x18($fp) IF ID EX ME WB
    sw $r2, 0x8($fp) IF ID EX ME WB
    sw $r1, 0x4($fp) IF ID EX ME WB
    addi $fp, $sp, 0 IF ID EX ME WB
    xor ... IF ID EX ME WB
    ```

Problem 3: The VAX locc instruction is another example of an instruction that would not be included in a RISC ISA because it could not be pipelined in any reasonable way. For this problem assume that implementations of character location can only read one byte at a time. (A fast implementation might read a word and check each position for the sought byte, but not in this problem.)
(a) What is the minimum amount of time that the DF-equivalent VAX implementation might take to execute locc with a length parameter equal to \(n\) ? Show how the instruction would execute.
(b) The MIPS routine below performs the same operation (except for the r0 and r1 return values). In terms of \(n\) how long does it take to compute locc?
locc:
\# Call Values:
\# a0: char: Character to find.
\# a1: len: Length of string.
\# a2: addr: Address of first character of string.
\# Return Value:
\# v0: 0 if character not found, 1 if found.
\# Note: Other locc return values not computed.
j START
add \$t1, \$a1, \$a2 \# \$t1: Stop address ( last char + 1 )
LOOP:
beq \$t0, \$a0 FOUND
addi \$a2, \$a2, 1
START:
bne \$a2, \$t1, LOOP
lb \$t0, 0(\$a2)
jr \$ra
addi \$v0, \$0, 0

FOUND :
jr \$ra
addi \$v0, \$0, 1
(c) Which implementation has the speed advantage? Explain.
(d) Can instructions be added to MIPS consistent with RISC principles that would substantially improve its performance? If not, explain what gives locc an inherent advantage on CISC.

Problem 1: For answers to this problems consult the SPARC Architecture Manual Version V8, linked to the course references page.

Suppose a SPARC V8 trap table has been set up at address \(0 \times 12340000\)
(a) Write a SPARC V8 program that sets the trap base register (TBR) to that address. Assume the processor is already in privileged mode. Hint: A correct solution consists of two instructions, a three-instruction program is okay too.

Call the SPARC V8 instruction that writes the TBR foo. The ISA definition of foo makes it easy to design the control logic and bypassing hardware on certain implementations.
(b) What about the definition of foo makes the control logic and bypassing hardware design easy on those certain implementations?
(c) Why not do the same for, say, the add instruction?
(d) Describe an implementation in which the control logic for foo would not be so simple despite the "help" from the ISA definition.

Problem 2: Solve the EE 4720 Spring 2007 Final Exam problem 1.
Problem 3: Solve the EE 4720 Spring 2007 Final Exam problem 3.

\section*{\(20 \quad\) Spring 2007}

Problem 1: Without looking at the solution solve Spring 2002 Homework 2 Problem 2 parts a-c. Then, look at the solution and assign yourself a grade in the range \([0,1]\).

Problem 2: If the value in register r 2 is not aligned (a multiple of four) the 1 w in the MIPS code below will not complete.
```

lw r1, O(r2)

```
(a) Re-write the code so that r 1 is loaded with the word at the address in \(r 2\), whether or not it is aligned. For this part do not use instructions lwl and lwr (see the next part).
(b) Re-write the code, but this time use MIPS instructions lwl and lwr. Hint: These instructions were not covered in class, try looking them up in the MIPS architecture manual conveniently linked to the http://www.ece.lsu.edu/ee4720/reference.html page..

Problem 3: Consider how the lwl and lwr instructions might be added to the implementation below. There are two pieces of hardware that with minor modification would be able to merge the sub-words in a reasonable solution. Alternatively, a new piece of hardware to perform the merge can be added.

(a) Show how the hardware can be modified.
- If your solution relies on an existing component to perform the merge indicate which component and why that can be easily modified to do the merge.
- As with other MIPS instructions lwl must spend one cycle in each stage (except when stalling).
(b) Show how the code below would execute on your solution. Pay attention to dependencies. Feel free to propose an alternate solution to reduce the number of stalls.
```

add r1, r3, r4
lwl r1, O(r2)
sub r5, r1, r6

```

Problem 4: Consider these options for handling unaligned loads in the MIPS ISA which might have been debated while MIPS was being developed.
- Option Lean: All load addresses aligned. No special instructions for unaligned loads (e.g., no lwl or lwr).
- Option Real: All load addresses aligned. Special instructions for unaligned loads (e.g., lwl or lwr).
- Option Nice: Load addresses do not have to be aligned, however warn programmers that loads of unaligned addresses may take longer in some implementations.
(a) For each option provide an advantage and a disadvantage.
(b) What kind of data would be needed to choose between these options? Consider both software and hardware data, be reasonably specific.
(c) Using made up data pick the best option. Any choice would be correct with the right data.

Problem 1: A manufacturer develops an ISA extension which can dramatically improve the performance of certain benchmarks. The extension includes new instructions which work well with small integers. In what the manufacturer calls well-formed C programs the compiler will find all opportunities where the new instructions can be used and so the dramatic improvement will be realized. On other programs in which the new instructions could be used the compiler won't use them because it can't tell if the resulting machine code would be correct (perhaps because its not sure if values in registers would be small). In such cases the compiler will provide a message for the programmer indicating a list of regions in which there was the possibility of using the instructions. The programmer can then recompile with a special option indicating which of those regions the new instructions can safely be used in. The resulting code would be sped up.

Suppose this all works out very well for developers. They have no problems indicating which regions are safe for the new instructions and their resulting executables are fast and run correctly.

The manufacturer would like to run the SPECcpu2006 benchmarks on their new implementation. Most of the SPECcpu2006 benchmarks are not well formed.
(a) Why couldn't the compiler options (flags) for the SPEC run (base or peak) indicate the safe regions under a reasonable interpretation of the rules? In your answer refer to specific parts of the SPECcpu2006 run and reporting rules,
http://www.spec.org/cpu2006/Docs/runrules.html.
(b) Keeping in mind the goals of the SPECcpu benchmarks argue either that the SPECcpu rules should be changed (perhaps for a future version of the benchmark) or argue that the rules should remain as they are.

Solve the problems below. Then look at the solutions and assign yourself a grade.
Problem 2: Without looking at the solution solve Spring 2006 Midterm Exam Problem 1.
Problem 3: Without looking at the solution solve Spring 2006 Midterm Exam Problem 2.

Some of the questions below are about the interrupt mechanisms defined for the MIPS32, SPARC V8, and PowerPC 2 ISAs. MIPS and SPARC interrupt mechanisms were covered in class, PowerPC's mechanism was not. All are documented in manuals linked to the class references page,
http://www.ece.lsu.edu/ee4720/reference.html. When using these references keep in mind that interrupt terminology differs from ISA to ISA and that you are not expected to understand (at least on first reading) most of what is in these manuals. Finding the right manuals and the relevant pages in those manuals is part of this assignment's learning experience.

Problem 1: Consider a load instruction that raises an exception due to a fixable problem with a memory address (for example, a TLB miss, whatever that is) on an implementation of MIP32, SPARC V8, and PowerPC 2.
(a) Where does each ISA say the address of the faulting instruction (the load) should be put? Give the exact register name, number, or both (if available).
(b) Where does each ISA say to put the memory address that the load attempted to load from?

Problem 2: Is PowerPC's equivalent of a trap table more similar to SPARC's trap table or to MIPS'? Explain and describe how specific elements are the same or different. Look at table placement, size, number of entries, and perhaps other characteristics.

Problem 3: In class a precise exception was defined as one in which, to the handler it appears that all instructions before the faulting instruction have completed normally and that the faulting instruction and those following it have not executed (correctly or otherwise). PowerPC calls certain exceptions precise even though they violate this rule. What are they and how is this violation justified in the manual?

Problem 4: Solve Fall 2006 Final Exam Problem 1. Note: At the time this was assigned the solutions were not available.

Problem 1: Estimate performance of the 8-way superscalar statically scheduled MIPS implementations described here. All are five stages, as used in class, and always hit the cache, as has been the case in class so far. Some of the implementations have no fetch group alignment restrictions, which means any eight contiguous instructions can be fetched. Some impose a fetch group alignment restriction, meaning if a CTI target is address \(a\) IF will fetch eight instructions starting at address \(a \prime=8 \times 4 \times\left\lfloor\frac{a}{8 \times 4}\right\rfloor\) (for those preferring C: \(\quad \mathrm{aa}=\mathrm{a} \& \sim 0 \mathrm{x} 1 \mathrm{f}\) ). Instructions in \([a \prime, a)\) (or from aa to before a) will be squashed before reaching ID.

The implementations include a branch predictor that predicts when a branch is in IF, resolves (checks the prediction) when a branch is in ID, and if necessary recovers (squashes wrong-path instructions) when a branch is in EX. A branch is predicted when it is in IF and the prediction is used in the next cycle. Example 1 , below, illustrates a correct taken prediction. The correctness of the prediction is checked, resolved, when the branch is in ID; if incorrect the wrong-path instructions are squashed and the correct path instructions are fetched in the next cycle (when the branch is in EX). This is illustrated in Example 2 for an incorrect taken prediction.

Note that due to alignment restrictions (if imposed) and branch placement the number of useful instructions fetched in a cycle can vary, and that is something to take into account in the subproblems below. The examples below illustrate when instructions will be fetched and squashed but they do not show how many will be fetched in every situation.
```


# Example 1: Branch correctly predicted taken. Target fetched in next cycle.

# 

# Cycle 0

beq r1,r2, TARG IF ID EX ME WB
nop IF ID EX ME WB
TARG:
add r3, r4, r5 IF ID EX ME WB
\# Example 2: Branch wrongly predicted taken.
\# Target squashed, correct path (fall through) fetched in cycle after ID.
\# Cycle 0
beq r1,r2, TARG IF ID EX ME WB
nop IF ID EX ME WB
sub r6, r7, r8 IF ID EX ME WB
TARG:
add r3, r4, r5
IFx
\# Cycle

```

All implementations run the same program, which has not been specially compiled for the 8 -way machine. In the program, which has no floating-point instructions, any two data-dependent instructions have at least seven instructions between them. This avoids some stalls, assume that there are no other stalls in the superscalar implementation due to data dependencies.

Let \(n_{\mathrm{i}}\) denote the number of dynamic instructions in the program and let \(n_{\mathrm{b}}\) denote the number of dynamic instructions that are branches. For the questions below show answers in terms of these symbols and also show values for \(n_{\mathrm{i}}=10^{10}\) and \(n_{\mathrm{b}}=2 \times 10^{9}\). Assume that half of the times a branch is executed it is taken.
(a) Suppose the 8-way implementation has perfect branch prediction and has no fetch alignment restrictions. Approximately how long (in cycles) will it take to run the program. State any assumptions. Hint: Because
of the branches it's \(>\frac{n_{i}}{8}\).
(b) Repeat the question above for a predictor that always predicts not taken (which essentially means no predictor).
(c) Repeat the question above for a predictor with a \(95 \%\) prediction accuracy. (Yes, that means \(95 \%\) of the predictions are correct.)
(d) Once again, suppose the 8 -way implementation has perfect branch prediction but now fetch is restricted to aligned groups. Approximately how long (in cycles) will it take to run the program. State any assumptions.

Problem 2: Consider a bimodal branch predictor with a \(2^{10}\)-entry branch history table (BHT).
(a) What is the prediction accuracy on the branch below with the indicated behavior assuming no interference. Assume that the pattern continues to repeat. Provide the accuracy after warmup.

Problem 3: Suppose that for some crazy reason it's important that the branch at address \(0 \times 1000\) be predicted accurately, even if that means suffering additional mispredictions elsewhere. The result of this crazyness is the code below, in which the branch in HELPER is intended to help the branch at 0x1000.
(a) Choose an address for HELPER so that \(0 \times 1000\) is helped.
(b) Given a correct choice for the address of HELPER, find the prediction accuracy of the branch at \(0 \times 1000\).
```

    jal HELPER
    nop
    0x1000:
beq r1, r2 TARG t t t n t t n n n t t t n t t n n n m
...
....
HELPER:
beq r1, r2 SKIP
nop
SKIP:
jr r31
nop

```

\section*{21 Fall 2006}

Problem 1: Without looking at the solution solve Spring 2002 Homework 2 Problem 2 parts a-c. Then, look at the solution and assign yourself a grade in the range \([0,1]\).

Problem 1: Section 2.2.2 of the run and reporting rules for SPECcpu2006, http://www.spec.org/cpu2006/Docs/runrules.html, specifies that the optimization flags and options used to obtain the base result must be the same for each benchmark (compiled with the same language, say C). Why must they be the same?

Problem 2: Section 1.2.3 of the run and reporting rules for SPECcpu2006, http://www.spec.org/cpu2006/Docs/runrules.html, assumes that the tester is honest. Provide an argument that many of the run and reporting rules ignore this assumption, or at best are based on the assumption that the tester is honest but sloppy or unmotivated.

Problem 3: Find the SPECcpu2000 CINT2000 disclosure for the fastest systems using each of the chips below. All chips implement some form or superset of IA-32 (also known as 80 x 86 ). All of the implementations are superscalar, meaning they can sustain execution of more than one instruction per cycle. In particular, an \(n\)-way superscalar processor can sustain execution of \(n\) instructions per cycle on ideal code, on real code the sustained execution rate is much lower (for reasons to be covered later in the course, such as cache misses). Some of the implementations are multi-cored. (A core is an entire processor and so a 2-core chip has two complete processors.)
- Pentium III, 1-core, 2-way
- Pentium 4, 1-core, 3-way
- Pentium Extreme, 2-core, 3-way
- Intel Core 2 Extreme X6800, 2-core, 4-way
- Opteron 256, 1-core, 3 -way
- Athlon FX-62, 2-core, 3-way
(a) For each system list the following information:
- The peak (result) ratio (for the suite).
- The clock frequency.
- The gec peak (result) run time (in seconds).
- The maximum number of instructions the system could have executed during the run of gcc assuming all cores were used.
- The maximum number of instructions the system could have executed during the run of gcc assuming one core was used.
- Execution efficiency assuming all cores were used: number of instructions executed divided by maximum number of instructions that could have been executed in the same amount of time. Assume that all systems run the same binary (executable) of gcc and make a guess at how many instructions would be executed when running the binary for the SPEC inputs.
- Execution efficiency assuming a single core was used: Same as previous value, except assume only one core used.
(b) The execution efficiency computation was based on the assumption that the number of executed instructions was the same in all systems. Identify two systems for which this was more likely to be true and two systems where this was less likely to be true.
(c) How much does a dual-core implementation improve the performance of gcc?

Problem 1: Show the changes to the MIPS implementation below needed to implement the SPARC V8 instructions shown in the sub-problems. (See the SPARC Architecture Manual linked to the course references page for a description of SPARC instructions.) Do not show control logic changes or additions. For this problem assume that SPARC has 32 general-purpose registers, just like MIPS. (In reality there are \(16 n\), \(n \geq 4\) general-purpose registers organized into windows. An integer instruction sees only 32 of these but using save and restore instructions a program can replace the values of 16 of them, the feature is intended for procedure calls and returns. To satisfy curiosity, see the description of register windows in the ISA manual.)


For solution can use larger version on next page.
(a) Show the changes for the following instructions. The only changes needed for these are to bit ranges in the ID stage.
```

add %g1, %g2, %g3
sub %g4, 5, %g6

```
(b) Show the changes needed for the store instruction below. This will require more than changing bit ranges.
\[
\text { st } \% \mathrm{~g} 3,[\% \mathrm{~g} 1+\% \mathrm{~g} 2]
\]
(c) Show the changes needed to implement the instructions below. The alert student will have noticed the ALU has a new output labeled cc. That output has condition code values taken from the result of the ALU operation.
- Don't forget the changes needed for the branch target.
- The changes should work correctly whether or not the branch immediately follows the CC instruction.
- Cross out the comparison unit if it's no longer needed.
subcc \(\% \mathrm{~g} 1, \% \mathrm{~g} 2, \% \mathrm{~g} 3\)
bge TARG


Problem 1: The floating point pipeline in the MIPS implementation illustrated below must sometimes stall instructions to avoid the WF structural hazard. The WF structural hazard could be avoided by requiring all instructions that use WF to go through the same number of stages. Note that instructions that use WB all pass through five stages, even though some instructions, such as xor, could write back earlier.

Redesign the illustrated implementation so that the WF structural hazard is eliminated by having WF instructions (consider add.d, sub.d, mul.d, and lwc1) all pass through the same number of stages. The functional units themselves shouldn't change (still six multiply steps and four add steps) but their positions might change.
(a) Show the possibly relocated functional units and their connections. Don't forget connections for the lwc1 instruction.
(b) Show any changes to the logic generating the fd , we, and xw signals. Note: The original assignment did not ask for xw changes.
(c) Show bypass paths needed to avoid stalls between any pair of floating point instructions mentioned above.


Problem 2: Consider the changes to avoid structural hazard stalls from the previous problem. Provide an argument, either for making the changes and or against making the changes. For your argument use whatever cost and performance estimates can be made from the previous problem. Add to that the results of fictitious code analysis experiments and alternative ways of using silicon area to improve performance.

The code analysis experiments might look at the dynamic instruction stream of selected programs. For these experiments explain what programs were used and what you looked for in the instruction stream. Make up results to bolster your argument.

For the alternative ways of using silicon area, consider other ways of avoiding the structural hazard stalls, or other ways of improving performance. This does not have to be very detailed, but it must be specific. (For example, "use the silicon area for pipeline improvement" is too vague.)

The argument should be about a page and built on a few specific elements, rather than meandering long-winded generalities.

Problem 3: In the previous problem structural hazards were avoided by having all WF instructions pass through the same number of stages. If both WB and WF instructions passed through the same number of stages then, were it not for stores, it would easily be possible for floating-point instructions to raise precise exceptions without added stalls (even if exceptions could not be detected until M6).
(a) For this part, ignore store instructions. Explain why having all instructions pass through the same number of stages makes it easier to implement precise exceptions (without added stalls, etc.) for floating point instructions.
(b) For this part, include store instructions. Explain how store instructions preclude precise exceptions for the implementation outlined above, or at least for a simple one.
(c) For this part, include store instructions. Do something about stores so that the all-instructions-use-the-same-number-of-stages implementation can provide precise exceptions to floating point instructions. It is okay if the modified implementation adds stalls around loads and stores. A good solution balances cost with performance.

If your solution is costly say so and justify it. If your solution is low cost but lowers performance say so and show the execution of code samples that encounter stalls.
\(\square\)

\section*{22 Spring 2006}

Several Web links appear below and at least one is long, to avoid the tedium of typing them view the assignment in Adobe reader and click.

Problem 1: Consider these add instructions in three common ISAs. (Use the ISA manuals linked to the references page, http://www.ece.lsu.edu/ee4720/reference.html.)
```


# MIPS64

    addu r1, r2, r3
    
# SPARC V9

    add g2, g3, g1
    
# PA RISC 2

    add r1, r2, r3
    ```
(a) Show the encoding (binary form) for each instruction. Show the value of as many bits as possible.
(b) Identify the field or fields in the SPARC add instruction which are the closest equivalent to MIPS' func field. (A field is a set of bits in an instruction's binary representation.)
(c) Identify the field or fields in the PA-RISC add instruction which are the closest equivalent to MIPS' func field. Hint: Look at similar PA-RISC instructions such as sub and xor.
(d) The encodings of the SPARC and MIPS add instructions have unused fields: non-opcode fields that must be set to zero. Identify them.

Problem 2: Read the Overview section of the PA-RISC 2.0 Architecture manual, http://h21007.www2.hp.com/dspp/files/unprotected/parisc20/PA_1_overview.pdf.
(If that link doesn't work find the overview section from the course references page, http://www.ece.lsu.edu/ee4720/reference.html.)

A consequence of the unused fields in MIPS and SPARC add instructions and RISC's fixedwidth instructions is that the instructions are larger than they need to be.

The PA-RISC overview explains how PA-RISC embodies important RISC characteristics, as do other RISC ISAs, but also has unique features of its own.
(a) It is because of one of those class of features that the PA-RISC 2.0 add instruction lacks an unused field. What is PA-RISC's catchy name for those features?
(b) Provide an objection (from the RISC point of view) to the added functionality of PA-RISC's add instruction. If possible, find places in the overview that provide or at least hint at counterarguments to those objections.

Problem 3: Many ISAs today started out as 32 -bit ISAs and were extended to 64 bits. Two examples are SPARC (v8 is 32 bits, v9 is 64 bits) and MIPS (MIPS32 and MIPS64). One important goal is that code compiled for the 32 -bit version should run unchanged on the 64 -bit version. Another important goal is to add as little as possible in the 64 -bit version. For example, it would be easy maintain compatibility by adding a new set of 64 -bit integer registers and new 64 -bit integer instructions, but that would inflate the cost of the implementation. Another approach would be to extend the existing 32 -bit integer registers to 64 bits and change the existing instructions so they now operate on 64 -bit quantities, but that would break 32 -bit code (consider sll followed by srl).
(a) Does a MIPS32 add instruction, for example, add \$s1, \$s2, \$s3, perform 64-bit arithmetic when run on a MIPS64 implementation? If not, what instruction should be used to perform 64 bit integer arithmetic?
(b) Does a SPARC v8 add instruction, for example, add \(\% \mathrm{~g} 2, \% \mathrm{~g} 3, \% \mathrm{~g} 1\), perform 64 -bit arithmetic when run on a v9 implementation? If not, what instruction should be used?

Problem 4: Continuing with techniques for extending 32 -bit ISAs to 64 bits, consider the problem of floating-point registers. Both MIPS32 and SPARC v8 have 32 32-bit FP registers that can be used in pairs to perform 64 -bit FP arithmetic. Both 64 -bit versions effectively have 3264 -bit FP registers, but using different approaches.
(a) Describe the different approaches.

Problem 1: The code fragment below runs on the illustrated implementation. Assume the branch is always taken.
(a) Show a pipeline execution diagram covering execution to the beginning of the third iteration of the loop.
(b) What is the CPI for a large number of iterations?

Hint: Pay close attention to dependencies and carefully add the stalls to handle them; also pay close attention to the timing of the branch. Work from the illustrated implementation, do not adapt the solution from a similar past assignment, that would be like preparing for a 10 km run by driving around the jogging trail.
```

LOOP:
lw $s0, O($s1)
addi \$s3, \$s0, 4
bneq \$s3, \$0 LOOP
add \$s1, \$s1, \$s2
xor \$t0, \$t1, \$t2
or $t3,$t4, \$t5
and \$t6, \$t7, \$t8

```


Problem 2: The code fragment below (the same as the one above) runs on the illustrated implementation (different than the one above - and better!). Assume the branch is always taken.
(a) Show a pipeline execution diagram covering execution to the beginning of the third iteration of the loop.
(b) What is the CPI for a large number of iterations?
(c) An A points to a wire on the illustration. On the pipeline execution diagram show the value of that wire in every cycle that the corresponding stage holds a "live" instruction.
(d) A \(\sqrt{\mathrm{B}}\) points to a wire on the illustration. On the pipeline execution diagram add a row labeled \(B\), and on it place an \(X\) in a cycle if the value on the wire can be changed without changing the way the program executes.
```

LOOP:
1w $s0, O($s1)
addi \$s3, \$s0, 4
bneq \$s3, \$0 LOOP
add \$s1, \$s1, \$s2
xor \$t0, \$t1, \$t2
or \$t3, \$t4, \$t5
and \$t6, \$t7, \$t8

```


Review Fall 2004 Final Exam Problem 2, which was discussed in class on Monday, 13 March 2006.
Problem 1: Using the solution to Fall 2004 Final Exam problem 2 parts a, b, and d (but not c) as a starting point, make changes to implement a new two-source register MM instruction add.mmr which operates as shown in the example below. Hint: The solution requires a register file modification.
```

add.mmr (r1), (r2), r3 \# Mem[r1] = Mem[r2] + r3

```

Problem 2: Your boss, a stuck-in-the-twentieth-century RISC true believer who only grudgingly agreed to include add.mm, add.mr, add.rm, and add.mmr in MMMIPS, flies into an incoherent rage when you suggest also adding add.mmm to MMMIPS. What pushed your boss over the edge? (That is, why is add.mmm much harder to add to the implementation in the Fall 2004 exam than add.mmr.) Instruction add.mmm operates as shown below:
```

add.mmm (r1), (r2), (r3) \# Mem[r1] = Mem[r2] + Mem[r3]

```

Problem 3: Write a pair of programs intended to show the benefit of MMMIPS. Both programs should do the same thing, program \(A\) should use ordinary MIPS instructions and run on the MIPS pipeline shown below. Program \(B\) should use MMMIPS instructions and run on the implementation shown in the exam solution. Reasonable bypass connections may be added, including those needed for branches.
(a) Show the programs.
(b) Compute the execution time (in cycles) of each program. The comparison should be fair so each program should be producing the same result.


Problem 4: Show a program that will run slower on the MMMIPS implementation that the ordinary MIPS implementation. That program, of course, should not use MMMIPS instructions. Reasonable bypass connections can be added, including those needed for branches. Hint: Branches are important.

Problem 1: The code below executes on the illustrated MIPS implementation. The FP pipeline is fully bypassed but the bypass connections are not shown.

(a) Show a pipeline execution diagram.
(b) Determine the CPI for a large number of iterations.
(c) Add exactly the bypass connections that are needed. LOOP:
mul.d f2, f2, f4
bneq r1,0 LOOP
addi r1, r1, -1

Problem 2: Due to a coffee spill the implementation below has a flaw: The inputs to the M2-stage XW mux have been reversed, the top input should be a 2 but is a 1 , and the lower input should be a 1 but is a 2. There are no other flaws, in particular the control signal for the mux has been designed for a 2 at the upper input and a 1 at the lower input.

You are stranded alone on an island with this flawed implementation and to get off the island you need the result computed by the code below. The code was written for a normal MIPS implementation and will not compute the correct result on the flawed one. Re-write it so that it computes the correct result on the flawed implementation. (The solution must use the FP arithmetic units, do not simply implement IEEE 754 floating point using integer instructions.)


LOOP :
add.d f2, f2, f4
bneq r1,0 LOOP
addi r1, r1, -1

Note: For some sample problems with predictors see the final exam solutions.
Problem 1: The routine samples in the code below is called many times. Consider the execution of the code on three systems, each system using one of the branch predictors below.

All predictors use a \(2^{14}\)-entry branch history table (BHT). (The global predictor does not need its BHT for predicting branch direction.) The three predictors are:
- System B: bimodal
- System G: global, history length 10. (Accuracy can be approximated.)
- System L: local, history length 10.
(a) Determine the amount of memory (in bits) needed to implement each predictor.
(b) For each loop in samples determine the accuracy of the loop branch (the one that tests the value of i) after warmup on each system. The accuracy for the global predictor can be approximated, the others must be determined exactly.
(c) Why would solving the problem above be impossible, or at least tedious, if the BHT size were \(\approx 2^{3}\) entries?
```

void samples(int\& x, int\& y, char **string_array )
{
// Loop 5-xor
for( int i = 0; i < 5; i++ )
x = x `i;     // Loop 5-len     for( int i = 0; i < 5; i++ )         if( strlen( string_array[i] ) < 20 )             return; // Never executes. <- Important.     // Loop 100-xor     for( int i = 0; i < 100; i++ )         y = y` i;
}

```

Problem 2: The code more, below, runs on four systems. All predictors use a \(2^{14}\)-entry branch history table (BHT). (The global and gshare predictors do not need its BHT for predicting branch direction.) The predictors are:
- System B: bimodal
- System G: global, history length 10. (Accuracy can be approximated.)
- System X: gshare, history length 10. (Accuracy can be approximated.)
- System L: local, history length 10.
(a) In the code below estimate the prediction accuracy of the following predictors on Branch B and Branch C (there is no Branch A) after warmup, assuming that more is called many times.
(b) One of the predictors should have a low prediction accuracy. Why? Avoid a sterile description of the hardware, instead discuss the concept the predictor is based on and why that's not working here.
```

void more(int\& x, int\& y, int a, int\& b, int\& c)
{
for( int i=0; i<100; i++ ) x = x ` i;
if( a < 10 ) b++; // Branch B, never taken.
for( int i=0; i<100; i++ ) y = y ^ i;
if( a >= 10 ) c++; // Branch C, always taken.
}

```

\section*{23 Fall 2005}

Problem 1: Suppose the base and result (peak) SPEC CINT2000 benchmark scores were identical on company \(X\) 's new processor. Make up an advertising slogan based on the fact that they were identical. A catchy tune is optional.

Problem 2: According to the CPU performance equation increasing the clock frequency \((\phi)\) by a factor of \(x\) without changing instruction count (IC) or cycles per instruction start (CPI) will reduce execution time by a factor of \(x\). Find two SPEC CINT2000 disclosures (benchmark results) that provide good evidence for this.
(a) Give the CPU, clock frequency, and the base and result CINT2000 scores.
(b) Explain why for these disclosures \(\phi\) is different (obvious) but IC and CPI are probably the same (requires some thinking). It may not be possible to determine this for certain and it may not be possible to find a pair for which they are exactly the same, it's sufficient to find a pair in which they are arguably close.
(c) Based on the assumption of IC and CPI equality, show how closely the CPU performance equation predicts the performance of one of the systems. Suggest reasons for any difference.

Problem 3: In section 1.2 of the SPEC CPU 2000 run and reporting rules, http://www.spec.org/cpu2000/docs/runrules.html, there is a bullet item that states, "The vendor encourages the implementation for general use." Explain what that means and why it is there. Why would it be bad if the "implementation" were not "for general use."

Problem 1: The code below executes on the implementation illustrated.
(a) Draw a pipeline execution diagram up until the first fetch of the third iteration.
(b) What is the CPI for a large number of iterations?
addi r3, \$0, 123
LOOP:
lw r1, \(0(r 2)\)
bne r1, r3, LOOP
lw r2, 4(r1)


Problem 2: Is there any way to add bypass paths to the implementation above so that the code executes with fewer stalls:
(a) Suggest bypass paths that might have critical path impact but which probably won't halve the clock frequency.
(b) Explain why it is impossible to remove all stalls by adding bypass paths.

Problem 3: The beqir instruction from the midterm exam solution compares the contents of the rs register to the immediate, if the two are equal the branch is taken, the address of the branch target is in the rt register. In the code example below beqir compares the contents of r3 to the constant 123, if they are equal the branch is taken with register r1 holding the target address, in this case to TARG. The delay slot, nop, is also executed.
(a) Show the changes needed to implement this instruction on the implementation above.
(b) Include bypass paths so that the code below executes as fast as possible:
```

lui r1, hi(TARG)
ori r1, r1, lo(TARG)
beqir r3, 123, r1
nop

# Lots more code.

TARG:
xor r9, r10, r11

```

Problem 1: The execution of a new MIPS instruction blcz TARG, branch unless loop count register is zero, will result in a delayed control transfer to TARG unless the contents of a new register, lc, is zero; the target is computed in the same way as ordinary branch instructions. Execution of blcz will also decrement lc unless it is already zero. The lc register is loaded by two new instructions mtlc and mtlci. The code below uses some of the new instructions and the diagram shows a possible implementation.
```

    mtlc 100 # Load lc register for a 101-iteration loop
    LOOP:
sw r0, O(r1)
blcz LOOP \# If lc is not zero branch to LOOP, lc = lc - 1.
addiu r1, r1, 4

```

(a) Re-write the code above using ordinary MIPS instructions and write it so that the loop uses as few instructions as possible. Hint: A three-instruction loop body is possible.
(b) Using pipeline execution diagrams determine the speed of the sample program and your program from the previous part. Only use bypass paths that have been provided.
(c) Unless the control logic is appropriately modified the implementation above may not realize precise exceptions for all integer instructions. In fact, the problem could occur in the example program. Explain what the problem is and show a pipeline execution diagram in which the control logic insures that execution proceeds so that exceptions will be precise. Hint 1: The exception does not occur in any of the new instructions. Hint 2: One of the two remaining instructions in the example can not raise an exception so it must be the other one.
(d) Modify the implementation so that precise exceptions are again possible for all integer instructions (while retaining the loop count instructions) without sacrificing performance.

\section*{24 Spring 2005}

Problem 1: POWER is an IBM ISA developed for engineering workstations, PowerPC is an ISA developed by IBM, Apple, and Motorola for personal computers and is based on POWER. POWER and PowerPC have instructions in common but each has instructions the other lacks (and some of the common instructions behave differently). Therefore a POWER implementation could not run every PowerPC program and vice versa.
(a) Show the gcc 3.4.3 compiler switches used to compile code for a POWER implementation. Hint: Google is your friend, look for gcc documentation.
(b) Show the gcc 3.4.3 compiler switches used to compile code for a PowerPC implementation.
(c) Is it possible to use gcc 3.4.3 to compile a program that will run on both? If yes, show the switches.

Problem 2: From the SPEC Web site, http://www.spec.org, find the fastest result on the SPECFP2000 (that's FP, not INT) benchmark for each of the following implementations: IBM POWER5, Intel Itanium2, Intel Pentium 4, Fujitsu SPARC64 v, and AMD FX-55. (Use the configurable search form and have it display the processor name.)
(a) The non-IA-32 implementations (POWER5, Itanium2, and SPARC64 V) blow away the IA-32 implementations on one benchmark. Which one? Which company (of those listed above) would want that benchmark removed?
(b) The POWER5 can decode five instructions per clock, the Itanium 2 can decode six instructions per clock, the Pentium 4 and FX-55 each can decode three (what are essentially) instructions per clock, and the SPARC64 V can decode four per clock. Based on the SPECFP2000 results used in the first part, which processor is making best use of these decode opportunities? In other words, if one processor could decode \(10^{12}\) instructions during execution of the suite and another could decode \(5 \times 10^{12}\) instructions during execution of the suite, the first would be more efficient since it ran the suite using fewer instructions. (See last semester's Homework 1 for a similar problem.)

Problem 3: As pointed out in class a processor's CPI varies depending on the program being executed. For the questions below write a program in MIPS assembler (see
http://www.ece.lsu.edu/ee4720/mips32v2.pdf for a list of instructions), some other assembly language, or assembly pseudocode, as requested below.
(a) Write a program that might be used to determine the minimum possible CPI. Suppose you actually used the program to determine the minimum CPI on processor \(X\). How would the CPI be computed? Show an example using made up numbers based on your program an hypothetical processor \(X\). Explain why the result would be the minimum CPI (or close to it).
(b) Write a program that might be used to determine the maximum possible CPIand as with the previous part, show how CPI is computed. Your answer should include information about instructions in processor \(X\) used in your program. Explain why the result would be the maximum CPI (or close to it).

For answers to the questions below refer to the PowerPC description Book I which can be found on the class references page, http://www.ece.lsu.edu/ee4720/reference.html.

Problem 1: One instruction that MIPS lacks but many RISC ISAs have is an indexed load. Find the closest equivalent PowerPC instruction to SPARC's lw [\% \(\mathrm{r} 2+\% \mathrm{r} 3], \% \mathrm{r} 1\).
(a) Show the instruction in PowerPC assembly language.
(b) Show how the instruction is coded, include the register numbers.

Problem 2: One instruction that MIPS lacks but that a few other RISC ISAs have is autoincrement addressing. PowerPC has an instruction that can be used for autoincrement addressing but is more powerful than the autoincrement addressing described in class. Find the PowerPC instruction.
(a) Show the assembly language for the PowerPC instruction doing the same thing as the following autoincrement instruction: lw r1, (r2)+.
(b) Show the coding for the instruction above.
(c) The PowerPC instruction is more powerful than an ordinary autoincrement instruction. Show a code sample using the PowerPC instruction for which an ordinary autoincrement would not be suitable. Briefly explain why an ordinary autoincrement would not do.

Problem 3: PowerPC has a wide variety of load and store instructions. Find the load instruction that is least suitable for a RISC ISA based upon the criteria discussed in class. Explain why it's least suitable.

Problem 4: Some instructions are more difficult to implement than others, one reason is that the difficult instruction does something very different from normal instructions requiring at least a moderate amount of additional hardware. Some difficult-to-implement instructions are listed below. Explain what the difficulty is (what extra hardware or control complications would be needed).
(a) An indexed store instruction. (An indexed load instruction would not be considered difficult.)
(b) Autoincrement (or PowerPC's version) load instructions. (The autoincrement or PowerPC version of the store instructions are not difficult.)

Problem 1: Do Problem 1 in the Spring 2004 EE 4720 final exam. Grade yourself using the solution, the grade should be out of five points. When grading yourself please explain what the mistakes were and what the correct answer should be, as a helpful grader would. Be polite in your explanations unless there was no serious attempt to solve the problem. In that case point out how final exam study time is being undermined by the need to catch up.

Problem 2: In Method 3 the commit register map is used to recover the state the ID register map was in just after the most recently committed instruction was decoded. In a system in which the ID register map is checkpointed for predicted CTIs, the commit register map won't be used very often.
(a) Describe how a system using Method 3 but without the commit register map could recover the ID map state before a faulting instruction. The ID map would be recovered using information in the ROB at and after the faulting instruction.
- Explain, with an example, what steps the processor takes to recover the information.
(b) Show new connections to the ID register map to implement this. Try to do it without adding new read and write ports (that is, use existing ports).
(c) Describe the impact on performance when the technique is used for exceptions.
(d) Describe the impact on performance when the technique is used for mispredicted branches.

Problem 3: Consider the commit mapless system from the previous problem. Suppose it were possible to sequentially read the ROB from two locations, the head (as is currently done for committing instructions) and some other place, say at a mispredicted branch.
(a) How might this be used to recover the ID map faster than was done in the previous problem.
(b) If this can be made to work for branches then there would be no need for checkpointing the register map. The impact on performance when using the mechanism for mispredicted branches depends on the following factors: how fast instructions are fetched, how many cycles it takes to resolve a branch (determine if the prediction was correct), how long it takes the fetch mechanism to bring correct-path instruction to ID, and how fast recovery can be done. Show a formula that will give the number of extra cycles needed to recover from a branch misprediction using this scheme (compared to checkpointing). For the formula, use the factors listed above and any other that is relevant.

Note that the amount of time to fetch the first correct-path instruction is a variable, it can be more than the one cycle shown in most other problems.

\section*{25 Fall 2004}

\section*{LSU EE 4720}

Homework 1
Due: 15 September 2004
Problem 1: Select two pairs of disclosures (that's four total) from the CPU2000 benchmark results posted at www.spec.org. A pair should be for machines using the same ISA but having different implementations. Make the implementations as different as possible. Explain why you think the implementations are very different.

Some ISAs and implementations are listed in lecture set 1, but the solution is not restricted to those. Feel free to ask if you're not sure what ISA a processor implements or whether two ISAs are considered the same or different.

For each disclosure list: the ISA, the implementation, the peak (result) performance, and file name of the HTML-formatted disclosure.

Problem 2: The processors below have roughly the same SPEC CINT2000 peak (result) scores but are very different. (The links should be clickable in Acrobat Reader.)
ISA: Power, Implementation: POWER5, Decode: 5-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q3/cpu2000-20040804-03314.pdf
ISA: Itanium (IA-64), Implementation: Itanium 2, Decode: 6-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q1/cpu2000-20040126-02775.pdf
ISA: IA-32, Implementation: Xeon, Decode: 3-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q3/cpu2000-20040727-03291.pdf
ISA: \(\approx\) IA-32, Implementation: Athlon, Decode: 3-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030908-02502.pdf
ISA: SPARC V9, Implementation: SPARC64 V, Decode: 4-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q2/cpu2000-20040518-03044.pdf
(a) The performance of the processors, based on the peak result, are roughly the same. On the same graph plot the performance in the following ways:
- Using the SPEC peak (result) scores.
- Assume that performance is proportional to clock frequency. Determine the score of a processor by comparing its clock frequency to that of the SPARC64 and using that to scale the SPARC64 peak result.
- The table above shows how many instructions a processor can decode per cycle. (Four-way superscalar means four per cycle, see explanation below.) Determine the performance by comparing the number of instructions fetched per second to the SPARC64 and use that to scale the SPARC64 peak result.

What conclusions can be drawn from the plotted data?
*The following information is not needed to solve this assignment. The decode widths shown above are the maximum number of instructions that can be decoded per cycle. For any real program the number will be much lower due to a variety of factors, which will be covered later in the semester. One relatively minor factor is the instruction mix. The POWER5 (implementation), Itanium (ISA), and to a lesser extent the others limit the kinds of instructions that can be decoded together. More on this later in the semester. The decode widths for the Xeon and Athlon don't refer to IA-32 instructions: these processors take IA-32 instructions and break them into simpler instructions called micro-ops by Intel and (favoring marketing over descriptive accuracy) macro-ops by \(A M D\). The three instructions per cycle for the Xeon and Athlon are actually three micro- or macro-ops per cycle.
(b) The Xeon and Athlon systems in the disclosures above have about the same performance. AMD might argue that those disclosures don't show the full potential of the Athlon. Find a system that uses an Athlon and scores much better, and explain what accounts for the difference.
(c) There is a system characteristic that affects the performance of benchmark mcf. What is it?
(d) Nominate a disclosure for The Most Desperate Peak Tuning award.

Problem 1: Do Problems 1 and 2 From Spring 2004 Homework 3 http://www.ece.lsu.edu/ee4720/2004/hw03.pdf. After completing the problems look at the solution and assign yourself a grade. The maximum grade should be 10 points, divide the points between problems as you wish.

Problem 2: A new instruction, copyTreg rt, rs, will read the contents of register rt and rs and will write the contents of rs to the register number specified by the contents of register rt (not into register rt). For example,
```


# Before: \$1 = 4, \$2 = 0x1234, \$4 = 0

copyTreg \$1, \$2

# After: \$1 = 4, \$2 = 0x1234, \$4 = 0x1234;

# (Register \$4 written with contents of register 2.)

```

Note that this is a variation on Midterm Exam 1 Problem 3, with the destination, rather than the source, being specified in a register.
(a) Modify the pipeline below to implement this instruction.
(b) Add the bypass connections needed so that the code below executes correctly.
```


# Before: \$1 = 4, \$2 = 0x1234, \$4 = 0, \$5 = 0

addi \$1, \$0, 5
copyTreg \$1, \$2

# After: \$1 = 5, \$2 = 0x1234, \$4 = 0, \$5 = 0x1234;

```


Problem 3: In the problem above the register number to write to is in a register. Here consider copyFreg \(r t, r s\) in which, like the test question, the register to copy from is in a register. That is,
```


# Before: \$1 = 2, \$2 = 0x1234, \$4 = 0

copyFreg \$4, \$1

# After: \$1 = 2, \$2 = 0x1234, \$4 = 0x1234;

# (Register \$4 written with contents of register 2.)

```

Explain a difficulty in implementing this instruction on the pipeline below without vitiating its sublime elegance.


Problem 4: No, we are not vitiators. Instead consider copyBreg rd in which the source register to read is specified in the rs register of the preceding instruction, that value is written into the rd register of this instruction. (Okay, maybe we are vitiators.) For example,
```


# Before: \$1 = 2, \$2 = 0x1234, \$4 = 0

add \$0, \$1, \$0 \# Instruction below uses rs (\$1 here) of this insn.
copyBreg \$4

# After: \$1 = 2, \$2 = 0x1234, \$4 = 0x1234;

# (Register \$4 written with contents of register 2.)

```

Implement this instruction on the pipeline above (from the previous problem).
\(\square\) Homework 1

\section*{26 Spring 2004}

Problem 1：Write a MIPS program to count the number of words in a C－style string． See http：／／www．ece．lsu．edu／ee4720／2004／hw1．html for details．

Problem 2：As of this writing（13 February 2004，13：07：05 CST）the two highest scoring CINT2000 systems use Intel D875PBZ motherboards，one has a＂result＂of 1620，the other 1509 （both October 2003 disclosures）． What accounts for this difference？Some programs in the suite run at about the same speed on the two systems，some take different amounts of time．Why might only some programs show improvement？

Problem 1: The MIPS program below copies a region of memory and runs on the illustrated implementation. In the sub-problems below use only the bypass connections shown in the illustration.
(a) Show a pipeline execution diagram for the code running on the illustrated implementation for two iterations.
(b) Compute the CPI and the rate at which memory is copied in bytes per cycle assuming a large number of iterations.
- Don't forget, when computing the number of cycles per iteration be sure not to count a cycle more, or less, than once.
```

LOOP:
lw $t0, O($a0)
sw O(\$a1), \$t0
addi \$a0, \$a0, 4
bne \$a0, \$a2 LOOP
addi \$a1, \$a1, 4

```


Problem 2: Execution should be inefficient in the problem above.
(a) Add exactly the bypass connections needed so that the program above executes as fast as possible.
- Don't forget that branch uses ID-stage comparison units.
- Don't forget the store.
(b) Show a pipeline execution diagram of the code on the improved implementation.
(c) For each bypass path that you've added show the cycles in which it will be used by writing the cycle number near the bypass path. If a bypass path goes to several places (for example, both ALU muxen) put the cycle number at the place(s) that use the signal.
(d) Re-compute the CPI and the rate at which memory is copied.

Problem 1: Suppose code like the memory copy program above (from Homework 3) appears frequently enough in the execution of programs so that new instructions should be added to the ISA to allow improved execution. (It does and they have been.)

Following the points below devise new instruction(s) that can be used to write a new memory copy loop that would execute more efficiently than is possible with existing MIPS-I instructions. A goal is to copy at the rate of two bytes per cycle. See the subparts after the bulleted points below.
LOOP :
lw \$t0, \(0(\$ a 0)\)
sw \(0(\$ a 1), \$ \mathrm{t0}\)
addi \$a0, \$a0, 4
bne \$a0, \$a2 LOOP
addi \$a1, \$a1, 4

- The instructions must use the existing MIPS formats.
- An instruction can do more than one thing (as long as it follows the points below). For example, an instruction that does more than one thing is a post-increment load. To reach the two bytes / cycle limit one might need to combine a branch with something.
- The instructions cannot use implicit registers. (A register is implicit if it does not appear in the encoded instruction. For example, register 31 is implicit in the jal instruction.)
- To achieve two bytes per cycle the instructions might need to do something unusual with operands. Please ask if you're not sure if something is too unusual.
- As with all other ordinary instructions, the new instructions must advance one stage per cycle (unless stalled, if so they would sit idle).
- The modified pipeline must still use the same memory port and no new memory ports can be added.
- Modifications such as bypass paths can be added to speed the instructions.
(a) Show an example of each new instruction and show how it is coded.
(b) Show how the instructions would be implemented on the pipeline.
(c) Write a memory copy program using the new instructions.
(d) Show a pipeline execution diagram for the memory copy code.
(e) A two bytes per cycle solution would require doing something interesting for the branch. Explain what that is and show a pipeline execution diagram for the memory copy loop finishing a copy (where the interesting stuff would be done).

Problem 1: One question when extending an ISA from 32 to 64 bits is what to do about the shift instructions. Because of the way that the shift instructions are encoded in MIPS two new shifts (of each type) were added to MIPS-64.
(a) What do you think the MIPS-32 sra instruction should do in MIPS-64? Remember that an implementation of MIPS-64 must run MIPS-32 code correctly. Please answer this question before answering the next parts (but feel free to look at the questions). Hint: Any serious answer will get full credit. A smart-alec answer will get full credit only if it's particularly witty.
(b) Give two reasons why the MIPS-32 sra (not srav) instruction could not be used for all right arithmetic shifts needed in a 64 -bit program.
(c) What are the new MIPS-64 shift right arithmetic instructions? Give the mnemonics.
(d) Why were two (as opposed to one) new shift instructions of each type added to MIPS-64?

Problem 2: Do the Problem 2 (a) through (d) from the Fall 2003 EE 4720 final exam (the one on floating point instructions). (http://www.ece.lsu.edu/ee4720/2003f/fe.pdf) Do not look at the solution until after you have solved the problem or gave it a good try.

Problem 3: In the diagram below the we pipeline latches carry write enable signals for use in floating point writeback. If the functional units were arranged differently the we pipeline latches could be used as a reservation register (for detecting WF structural hazards).

(a) Redraw the diagram with that arrangement. Hint: Try to use the we signal in the diagram above for a reservation register. Figure out why that won't work and come up with a solution.
(b) Suppose the ID stage has boxes uses FP ADD and uses FP MUL to detect which (if any) floating point functional unit an instruction would use. Design the control logic to generate a stall signal if there would be a write float structural hazard.
(c) Add the connections necessary for a lwc1 instruction. Include the connections needed to detect a WF structural hazard (as was done for ADD and MUL in the previous part).

Problem 1: Read the Microprocessor Report article on the IBM PowerPC 970 (a.k.a. the G5), used in a popular person computer. The article is available at
http://www.ece.lsu.edu/ee4720/s/mprppc.pdf. If accessing from outside the lsu.edu domain provide user name ee4720 and the password given in class. Answer the following questions: (Please read the entire article, additional questions might be asked in a future assignment.)
(a) One might infer from the second paragraph that deeper pipelines are used to inflate clock frequencies solely for marketing purposes. Why do deeper pipelines allow higher clock frequencies? Are there reasons other than marketing to do that?
(b) The article describes the PPC 970 as a 5 -way superscalar processor, which is consistent with the definition used in class. How could overzealous marketing people inflate that number using features of the microarchitecture? Describe the specific feature. Why would that be overzealous?

The following two problems are nearly identical to Spring 2003 Homework 6. The main difference is in the stages that are used. It is okay to peek at the solutions for hints, for best results leave twelve hours between looking at those solutions (or solutions to similar problems) and completing this assignment.

Problem 2: Show the execution of the MIPS code fragment below for three iterations on a fourway dynamically machine using Method 3 (physical register file) with a 256 -entry reorder buffer. Though the machine is four-way, assume that there can be any number of write-backs per cycle. Use Method 3 as described in the study guide at http://www.ece.lsu.edu/ee4720/guides/ds.pdf with for the following differences:
- The FP multiply functional unit is three stages (M1, M2, and M3) with an initiation interval of 1 .
- Assume that the branch and branch target are always correctly predicted in IF so that when the branch is in ID the predicted target is being fetched.
- There are an unlimited number of functional units.
(a) Show the pipeline execution diagram, indicate where each instruction commits.
(b) Determine the CPI for a large number of iterations. (The method used for statically scheduled systems will work here but will be very inconvenient. There is a much easier way to determine the CPI.)
```

LOOP: \# LOOP = 0x1000
ldc1 f0, O(t1)
mul.d f2, f2, f0
bneq t1, t2 LOOP
addi t1, t1, }

```

Problem 3: The execution of a MIPS program on a one-way dynamically scheduled system is shown below. The value written into the destination register is shown to the right of each instruction. Below the program are tables showing the contents of the ID Map, Commit Map, and Physical Register File (PRF) at each cycle. The tables show initial values (before the first instruction is fetched), in the PRF table the right square bracket "]" indicates that the register is free. (Otherwise the right square bracket shows when the register is freed.)
(a) Show where each instruction commits.
(b) Complete the ID and Commit Map tables.
(c) Complete the PRF table. Show the values and use a "[" to indicate when a register is removed from the free list and a "]" to indicate when it is put back in the free list. Be sure to place these in the correct cycle.


The PPC 970 which was the subject of a question in Homework 6 is very similar to the POWER4 chip, the main differences being that the POWER4 lacks the packed-operand instructions and POWER4 includes two processors on a single chip.

Answer the following questions about the POWER4 based on information in "POWER4 System Microarchitecture," by Tendler et al, available via http://www.ece.lsu.edu/ee4720/doc/power4.pdf. The questions can be answered without reading the entire paper. In particular, there is no need to read past page 17.

Problem 1: Translate the following terms, as used in class, to their nearest equivalent in the paper.
- Integer Instruction
- Instruction Queue
- Reorder Buffer
- Physical Register

Problem 2: The pipeline execution diagram below shows MIPS code on the dynamically scheduled system described in the study guide.
(a) Re-draw the diagram using the stages from POWER4. (Do not translate the instructions into the POWER assembly language.) Just show one iteration and assume that the four instructions are formed into one group. Also assume that the branch does not have a delay slot. Use stages F1, F2, and F3 for the multiply.
(b) In your diagram identify the fetch and execute pipelines, as defined in class.
```


# Cycle

LOOP:
ldc1 f0, 0(t1) IF ID Q RR EA ME WB C
mul f2, f2, f0 IF ID Q RR M1 M2 M3 WB C
addi t1, t1, 8 IF ID Q RR EX WB C
bneq t1, t2 LOOP IF ID Q RR B WB C

```

Problem 3: The POWER4 uses what is commonly called a hybrid predictor in which each branch is predicted by two different predictors and a third predictor predicts the prediction to use. One predictor is something like the bimodal predictor discussed in class and the other is something like the gshare predictor discussed in class.
(a) Provide a code example in which the bimodal predictor described in class will do better than the POWER4's almost equivalent predictor. (Ignore the selector.)
(b) How might the POWER4 designers justify the differences with the bimodal predictor given the lower performance in the example above?
(c) Provide a code example in which the gshare predictor described in class outperforms the POWER4's almost equivalent predictor. (Ignore the selector.)
(d) How might the POWER4 designers justify the differences with the gshare predictor given the lower performance in the example above?

\section*{27 Fall 2003}

Problem 1: Look at the following SPEC CINT2000 disclosures for these Dell and HP Itanium 2 systems: HP: http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030711-02389.html
Dell: http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030701-02367.html. (Note: Links are clickable within Acrobat reader.)

The CPU performance equation decomposes execution time into three components, clock frequency, \(\phi\), instruction count, IC, and CPI. For each component determine if its value on the two systems is definitely the same, probably the same, probably different, definitely different. Hint: The answer for the clock frequency is easy, the others require a little understanding of what IC and CPI are. Briefly justify your answers.

Problem 2: Though one may normally think of an implementation as a microprocessor chip, the definition can also include other parts of the system, such as memory and even disk. Why is that important in the problem above?

Problem 3: Differences in ISA, compiler, and implementation all affect the execution time of programs, and the impact of these factors can vary from program to program. For example, an implementation with faster floating point will have a larger impact on programs that do more floating-point computations.

From a look at the SPEC disclosures one can see that the fastest program on one system may not be the fastest program on another. (Use the int2000 results. From the spec CPU2000 results page find the configurable query form and request a page sorted by "Result" in descending order. It would be helpful to include the processor and compiler in the results. If your system is slow omit results before 2002.) For example, the Dell system from the first problem ran vortex fastest (of all the benchmarks), while the HP system ran mcf fastest. In this case the difference in fastest benchmark could not be the ISA, but it could be the compiler or the implementation. Call the speed ranking of benchmarks for a system its character. The character of the Dell system is vortex, gcc, eon, ... (benchmarks from fastest to slowest) and the character of the HP system is mcf, vortex, gcc, ....

The differences in character are due in part to the ISA, compiler, and implementation. Using the SPEC CINT2000 disclosures determine which is most important in determining character. Please do not try to look at all disclosures, just enough to determine an answer, even if that answer might change if you were to look at more.

In your answer, state which (ISA, compiler, or implementation) is most important, which disclosures you looked at, and how you drew that conclusion. This question is easy to answer (once it's understood).

As best you can explain why a particular factor is most important and why it is least important. You are not expected to answer this question very well, most of the material has not been covered yet. Don't take too much time and do your best with what has been covered and what you already know.

Additional Information:
Here are some ISAs and implementations of processors listed:
IA-32 ISA: (includes variations) implemented by Pentium 4 (and other Pentia) Xeon, Athlon, Opteron. Power Architecture ISA: Implementated by POWER4, RS64IV. Itanium ISA: Implementated by Itanium 2. Alpha ISA: Implementated by Alpha 21164, 21264, 21364. SPARC V9 ISA: Implementated by SPARC64V, UltraSPARC III Cu MIPS ISA: Implementated by R14000

Problem 4: The two procedures below are compiled with optimization on but with no special optimization options. Why might the second one run faster? (This is very similar to the classroom example.)
void add_array_first_one(int \(*\) a, int \(* \mathrm{~b}\), int \(* x\) )
\{
    for ( int \(i=0 ; i<100 ; i++\) ) a[i] = b[i] + *x;
\}
void add_array_second_one(int *a, int *b, int *x)
\{
    int xval \(=\) *x;
    for ( int i = 0; i < 100; i++ ) a[i] = b[i] + xval;
\}

Problem 1: Unlike MIPS, PA-RISC 2.0 has a post-increment load and a load using scaled-index addressing. The code fragments below are from the solution to Problem 2 in the midterm exam the fragments show several MIPS instructions under "Combine" and a new instruction under "Into." For each "Into" instruction show the closest equivalent PA-RISC instructions and show the coding of the PA-RISC instruction. (See the references page for information on PA-RISC 2.0)
(The term offset used in the PA-RISC manual is equivalent to the term effective address used in class, and is not to be confused with offset as used in this class. Assume that the s field and cc fields in the PA-RISC format are zero.)

Show all the fields in the format, including their names and their values.
```

Combine:
lbu $t1, O($t0)
addi \$t0, \$t0, 1
Into:
lbu.ai $t1, O($t0)+ \# Post increment load.
Combine:
sll \$t1, \$t1, 2
add \$t3, \$a1, \$t1
lw $t4, 0($t3)
Into:
lw.si $t4, ($t3,\$t1) \# Scaled index addressing.

```

Problem 2: The code fragment below runs on the implementation illustrated below.
(a) Show a pipeline execution diagram for the code fragment on the implementation up to the second fetch of the sub instruction; assume the branch will be taken.
(b) Show the value of the labeled wires (A, B, and C) at each cycle in which a value can be determined.

For maximum pedagogical benefit please pay close attention to the following:
- As always, look for dependencies.
- Pay attention to the RAW hazard between sub and sw and the RAW hazard between andi and bne.
- Make sure that add is fetched in the right time in the second iteration.
- Base timing on the implementation diagram, not on rules inferred from past solutions.


LOOP:
```

    add r1, r2, r3
    sub r3, r1, r4
    sw r3, 0(r5)
    andi r6, r3, 0x7
    ```
    bne r6, \$0, LOOP
    addi r2, r2, 0x8

Problem 3: Consider the implementation from the previous problem, repeated below.
(a) There is a subtle reason why the implementation cannot execute a jr instruction. What is it? Modify the hardware to correct the problem.
(b) There are two reasons why it cannot execute a jalr instruction, one given in the previous part and a second reason. What is it? Modify the hardware to correct the problem.


Problem 1: Design the control logic for the store value multiplexor (the one that writes pipeline latch ex_mem_rtv). The control logic must be in the ID stage. Hint: This is a fairly easy problem.


Problem 2: One problem with a post-increment load is storing the incremented base register value into a register file with one write port. Suppose a post-increment, register-indirect load were added to MIPS and implemented in the pipeline on the next page. This post-increment load does not use an offset, instead the effective address is just the contents of the rs register.

One option for storing the incremented base register value is to stall the following instruction and write back the value when the bubble reaches WB. We would like to avoid stalls if we have to, so for this problem design hardware that will use the WB stage of the instruction before [sic] or after the post-increment load if one of those instructions does not perform a writeback. For example:
```

bneq \$s0, \$s1, SKIP (Not taken)
lw $t1, ($t2)+
j TARG
add \$s3, \$s1, \$s2
lw $t1, ($t2)+
sub \$s4, \$s5, \$s6

```

The first post-increment load could writeback when either the bneq or the \(j\) were in the WB stage since neither performs writeback. The second post-increment load would have to insert a stall.
(a) Show the hardware needed to implement the post-increment load in this way.
- Remember that this load does not have an offset.
- Use a \(=\) PIL box to identify post-increment loads (input is opcode, output is 1 if it is a post-increment load, 0 otherwise).
- A stall signal is available in each stage; if the signal is asserted the instruction in that and preceding stages will stall and a nop instruction will move into the next stage (for each cycle hold is asserted).
- Show any new paths added for the incremented value, perhaps to the register file write port (which still has one write port).
- Add any new paths needed to get the correct register number to the register file write port.
- Ignore bypassing of the incremented address to other instructions.
- Show the added control logic, which does not have to be in the ID stage. (In fact it would be difficult to put all of control logic for this instruction in the ID stage.)
- Last but not least, a design goal is low cost, so add as little hardware as necessary to implement the instruction.
(b) If you're like most people, you didn't worry about precise exceptions when solving the previous part. Explain how the need for precise exceptions can complicate the design.


Problem 3: Answer these questions about interrupts in the PowerPC, as described in the PowerPC
Programming Environments Manual, linked to
http://www.ece.lsu.edu/ee4720/reference.html.
(a) Listed below are the three types of interrupts using the terminology presented in class. What are the equivalent terms used for the PowerPC.
- Hardware Interrupt
- Exception
- Trap
(b) In which register is the return address saved?
\(\square\)

\section*{28 Spring 2003}

At the time this was assigned computer accounts and solution templates were not ready. If they become available they can be used for the solution, either way a paper submission is acceptable.

Problem 1: When compiling code to be distributed widely one should be conservative when selecting the target ISA but less caution needs to be taken with the target implementation. Explain what "conservative" and less caution mean here, and explain why conservatism and in one case less caution in the other can be taken.

Problem 2: Based on the SPECINT2000 results for the fastest Pentium and the fastest Alpha, which programs would a shameless and unfair Alpha advocate choose if the number of programs in the suite were being reduced to five. Justify your answer.

Problem 3: The Pentium 4 can execute at a maximum rate of three instructions (actually, microops, but pretend they're instructions) per cycle (IPC), the Alpha 21264 can execute at most 4 IPC and the Itanium 2 can execute at most 6 IPC. Assume that the number of instructions for perlbmk, one of the SPECINT2000 programs, is the same for the Alpha, Itanium 2, and Pentium 4 (pretending micro-ops are instructions, if you happen to know what micro-ops are).
(a) Based upon the SPECINT2000 results (not base) for the perlbmk benchmark, which processor comes closest to executing instructions at its maximum rate? ("Its", not "the".)
(b) Are these numbers consistent with the expected tradeoffs for increasing clock frequency (mentioned in class) and for increasing the number of instructions that can be started per cycle?

Problem 4: Complete the lookup routine below so that it counts the number of times an integer appears in an array of 32 -bit integers. Register \(\$ \mathrm{a} 0\) holds the address of the first array element, \$a1 holds the number of elements in the array, and \$a2 holds the integer to look for. The return value should be written into \(\$ \mathrm{v} 0\).

\section*{lookup:}
```


# Call Arguments

# 

# \$a0: Address of first element of array. Array holds 32-bit integers.

# \$a1: Number of elements in array.

# \$a2: Element to count.

# 

# Return Value

# 

# \$v0: Number of times \$a2 appears in the array starting at \$a0

# [ ] Fill as many delay slots as possible.

# [ ] Avoid using too many instructions.

# [ ] Avoid obviously unnecessary instructions.

# A correct solution uses 11 instructions, including 6 in

# the loop body. A different number of instructions can be used.

# Solution Starts Here

```
\# Use the two lines to return, fill the delay slot if possible.
jr \$ra
nop

Design a stack ISA with the following characteristics:
- Memory has a 64 -bit address space and consists of 8 -bit characters.
- The stack consists of 64 -bit registers.
- The ISA uses 2's complement signed integers.
- Only add other data types as necessary.

The stack ISA must realize these goals:
- Small program size.
- Low energy consumption. (For here, assume energy consumption is proportional to dynamic instruction count.)
- Relatively simple implementation. Instructions should be no more complex than RISC instructions.

Design the instruction set based on the sample programs in the problems below and the following:

\section*{Arithmetic and Logical Instructions}

They should read their source operands from the top of stack (top one or two items) and push their result on the top of stack. Arithmetic instructions cannot read memory and they cannot read beyond the top two stack elements. (That is, you can't add an element five registers down to one ten registers down. Instead use rearrangement instructions before the add.) Specify whether the arithmetic and logical instructions pop their source operands. One can have both versions of an instruction. For example, add might pop its two source operands off the stack while addkeep might leave the two operands:
```


# Stack: 26 3 2003

add

# Stack: 29 2003

addkeep

# Stack: 2032 29 2003

```

Remember that arithmetic and logical instructions cannot rearrange the stack and cannot access memory.

\section*{Immediates}

Decide how immediates will be handled. There can be immediate versions of arithmetic instructions or one can have push immediate instructions. See the example below. Keep in mind that the register size is 64 bits.
```


# Stack: 123

addi 3 \# An immediate add.

# Stack: 126

pushi 3

# Stack: 3 126

add

# Stack: 129

```

\section*{Load and Store Instructions}

Memory is read only by load instructions which push the loaded item on the stack. Memory is written only by store instructions which get the data to store from the top of the stack. Determine which addressing modes are needed for loads and stores, and design instructions with those modes.

\section*{Stack Rearrangement Instructions}

The stack rearrangement instructions change the order of items on the stack. Consider adding the following: exch, swaps the top two stack elements. roll \(n j\), remove the top \(j\) stack elements and insert them starting after what was the nth stack element. (See the example.) Other stack rearrangement instructions are possible.
```


# stack 11 22 33 44 55 66

exch

# stack 22 11 33 44 55 66

roll 5 2

# stack 33 44 55 22 11 66

```

\section*{Control Transfer Instructions}

Your ISA must have instructions to perform conditional branches, unconditional jumps, indirect jumps, and procedure calls. It must be possible to jump or make a procedure call to anywhere in the address space. (The only thing special the instruction used for a procedure call has to do is save a return address.) The branch instructions can (but do not have to) use a condition code register. No other registers can be used (other than those in the stack). Don't forget about the target address.

Problem 1: As specified below, describe your ISA and the design decisions used. (Don't completely solve this part until you have solved the other problems.)
(a) For each instruction used to solve the problems below or requested above, show the assembler syntax and the instruction's coding. The coding should show the opcode, immediate, and any other fields that are present. Don't forget the design goals. Also don't forget about control transfer targets.

There is no need to list a complete set of instructions, but for coding purposes assume their existence. (There must be a way of coding a complete set of instructions that realize the goals of this stack ISA.)
(b) Determine the size of the stack. Specify instruction coding and implementation issues used to determine the size.
(c) Explain your decision on whether there are immediate versions of arithmetic instructions. (The alternative is instructions like pushi.)
(d) Explain your selection of memory addressing modes. Also, pick an addressing mode that you did not use and explain why not.
(e) Explain how other design decisions you have made help realize the goals of small program size, low energy, or simple implementation.
( \(f\) ) Describe any design decision you made that involved a tradeoff between code size, energy, or implementation simplicity. (Pick any pair.) The original question asked only about code size and energy. If you didn't make such a decision make one up.

Problem 2: Re-write the following MIPS code in your stack ISA.
```

lui \$a0, %hi(array) \# High 16 bits of symbol array.
ori \$a0, \$a0, %lo(array) \# Low 16 bits of symbol array.
jal lookup \# The name of a routine.
nop

```

Problem 3: Re-write the solution to Homework 1 in your stack ISA, use the template below. (Use your own solution or the one posted.)
lookup:
```


# Call Arguments (TOS is the top of the stack.)

# 

# TOS: Return address

# TOS + 1: ADDR of first element of array. Array holds 64-bit integers.

# TOS + 2: Number of elements in array.

# TOS + 3: TARGET, element to count.

# 

# Return Value

# 

# TOS: Number of times TARGET appears in the array starting at ADDR.

# Solution Here

# 

# [ ] Don't forget the return.

```

Problem 1: Consider the code below.
```


# Cycle 0-1

add \$t1, \$t2, \$t3 IF ID
sub \$t4, \$t5, \$t1
lw $t6, 4($t1)
sw 0($t4),$t6

```
(a) Show a pipeline execution diagram for the code running on the following illustration. Note that the add is fetched in cycle zero.
- Take great care in determining the number of stall cycles.


Problem 2: The code below is the same as in the previous problem.
```


# Cycle 0 1

add \$t1, \$t2, \$t3 IF ID
sub \$t4, \$t5, \$t1
lw $t6, 4($t1)
sw 0(\$t4), \$t6

```
(a) Show a pipeline execution diagram (PED) of the code running on the system below.
(b) In the PED circle each stage that sends a bypassed value. In the diagram label each bypass path with the cycle in which it is used. To avoid ambiguity, label the end of the path (at the mux input).


The problem below is tricky. If necessary use Spring 2001 Homework 2 problem 3 for practice.

Problem 3: The program below has an infinite loop and runs on the bypassed implementation below.
```

    # Initially $t0 = LOOP (address of jalr)
    LOOP:
jalr \$t0
addi \$t0, \$ra, -4
bne \$t0, \$0 LOOP
addi \$t0, \$t0, -4

```

(a) Show a pipeline execution diagram for this program up to a point at which a pattern starts repeating. Beware, the loop is tricky! Read the fine print below for hints.

Note that jalr reads and writes a register. The jalr instruction should be fetched twice per repeating pattern. The addi instruction should be fetched three times per
repeating pattern
(b) In the PED circle each stage that sends a bypassed value. In the diagram label each bypass path with the cycle in which it is used. To avoid ambiguity, label the end of the path (at the mux input).
(c) Determine the CPI for a large number of iterations.

Problem 4: SPARC V9 has multiple floating-point condition code (FCC) registers. See the references pages for more information on SPARC V8 and V9.
(a) Write a program that uses multiple FCC's in a way that reduces program size. As an example, the SPARC program below uses a single FCC. (To solve this problem first find instructions that set and use the multiple FCC registers in the SPARC V9 Architecture Manual. Then write a program that needs the result of one comparison (say, \(a<b\) ) several times while also using the result of another (say, \(c>d\) ). A program not using multiple condition code registers should have to do the comparison multiple times whereas the program you write does each comparison once.)
```

! 10 ! {
! 11 ! sum = sum + 4.0 / i; i += 2;
.L77000016:
/* 0x0020 11 */ fdivd %f4,%f30,%f6
/* 0x0024 */ faddd %f30,%f2,%f8
! 12 ! sum = sum - 4.0/ i; i += 2;
/* 0x0028 12 */ faddd %f8,%f2,%f30
/* 0x002c */ fcmped %f30,%f0
/* 0x0030 */ fdivd %f4,%f8,%f8
/* 0x0034 11 */ faddd %f10,%f6,%f6
/* 0x0038 12 */ fbl .L77000016
/* 0x003c */ fsubd %f6,%f8,%f10
! 13 ! }

```
(b) SPARC V9 is the successor to SPARC V8, which has only one FCC register. (SPARC V9 implementations can run SPARC V8 code.) Did the addition of multiple FCC's require the addition of new instructions or the extension of existing instructions? Answer the question by citing the old and new instructions and details of their coding.
(c) Do you think the designers of SPARC V8 planned for multiple FCC's in a future version of the ISA?

Problem 1: The two code fragments below call trap number 7. How do the respective handlers determine that trap 7 was called?
! SPARC V8
ta \(\% \mathrm{~g} 0,7\)
\# MIPS
teq \$0, \$0, 7

Problem 2: There is a difference between the software emulation of unimplemented SPARC V8 instructions triggered by an illegal opcode exception, such as faddq, and Alpha's use of PALcode for certain instructions. (See the respective ISA manuals on the references Web page. For SPARC, see Appendix G, it should not be difficult to find the PALcode information for Alpha.)
(a) What is similar about the two?
(b) What is the difference between the kinds of instructions emulated using the two techniques? Why would it not make sense to use PALcode for quad-precision arithmetic instructions?

Problem 3: In both SPARC and MIPS each trap table entry contains the first few instructions of the respective trap handler. On some ISAs a vector table is used instead, each vector table entry holds the address of the respective handler.

Why would the use of a vector table (rather than a trap table) be difficult for the MIPS implementation below?


Problem 4: One way of implementing a vector table interrupt system on the MIPS implementation above would be by injecting hardware-generated instructions into the pipeline to initiate the handler. These instructions would be existing ISA instructions or new instructions similar to existing instructions.

What sort of instructions would be injected and how would they be generated? Show changes needed to the hardware, including the injection of instructions. In the hardware diagram the instructions can be generated by a magic cloud [tm] but the cloud must have all the inputs for information it needs.

Include a program and pipeline execution diagram to show how your scheme works.
- Assume an exception code is available in the MEM stage.
- Include a vector base register, (VBR), which holds the address of the first table entry.

Problem 1: [Easy] Complete pipeline execution diagrams for the following code fragments running on the fully bypassed MIPS implementations with floating point units as described below.
```


# One ADD unit, latency 3, initiation interval 1.

add.d f0, f2, f4
sub.d f6, f0, f8
add.d f8, f10, f12

# One ADD unit, latency 3, initiation interval 2.

add.d f0, f2, f4
sub.d f6, f0, f8
add.d f8, f10, f12

# Two ADD units (A and B), latency 3, initiation interval 4.

add.d f0, f2, f4
sub.d f6, f0, f8
add.d f8, f10, f12

```

Problem 2: [Easy] Choose the latency and initiation interval for the add and multiply functional units so that the two instructions stall to avoid a structural hazard. Show a pipeline execution diagram consistent with them. (The easy way to solve it is to do the PED first, then figure out the latency and initiation interval.)
```

mul.d f0, f2, f4
add.d f6, f8, f10

```

Problem 3: The two PEDs below show execution of MIPS code produce that produces wrong answers. For each explain why and show a PED of correct execution.
```


# PED showing a DESIGN FLAW. (The code runs incorrectly.)

add.s f1, f10, f11 IF ID A1 A2 A3 A4 WF
sub.d f2, f0, f4 IF ID A1 A2 A3 A4 WF

# PED showing a DESIGN FLAW. (The code runs incorrectly.)

mul.d f0, f2, f4 IF ID M1 M2 M3 M4 M5 M6 WF
sub.s f1, f10, f11 IF ID A1 A2 A3 A4 WF

```

Problem 4: As described below design the logic for the floating-point register file in the MIPS implementation below. The FP portion shows only part of add functional unit. Assume that is the only functional unit.
- Describe how the FP register file works. For reference, here is a description of the integer register file: The integer register file has two read ports and a write port. Each read port has a five-bit address input and a 32 -bit data output. The write port has a five-bit address input and a 32 -bit data input. Reads from zero retrieve 0 , writes to zero have no effect.
- The following signals are available: is dbl; if 1 the instruction uses double-precision operands, otherwise single-precision. FP dst: if 1 the instruction writes the floating-point register file, otherwise it does not (possibly because it's not a floating-point instruction).
- Show all connections to the FP register file. Show the number of bits or the bit range for each connection.
- The WF stage provides two signals, FPU (the value to write back) and fd, something generated in ID (as part of the solution). Additional signals can be sent down the pipeline.
- Keep In Mind: The hardware should work for both single and double operands. (That's what makes the problem interesting. If you're confused first solve it assuming only double operands, then attempt the full problem.)
- Make sure the fragments from the previous problem would run correctly.


Problem 1: Show the execution of the MIPS code fragment below for three iterations on a fourway dynamically machine using method 3 (physical register file). Though the machine is four-way, assume that there can be any number of write-backs per cycle.
- Assume that the branch and branch target are correctly predicted in IF so that when the branch is in ID the predicted target is being fetched.
- The FP multiply functional unit is three stages (M1, M2, and M3) with an initiation interval of 1 .
- There are an unlimited number of functional units.
(a) Show the pipeline execution diagram, indicate where each instruction commits.
(b) Determine the CPI for a large number of iterations. (The method used for statically scheduled systems will work here but will be very inconvenient. There is a much easier way to determine the CPI.)
LOOP: \# LOOP = 0x1000
ldc1 f0, 0 (t1)
mul.d f2, f2, f0
bneq t1, t2 LOOP
addi t1, t1, 8

Problem 2: The execution of a MIPS program on a one-way dynamically scheduled system is shown below. The value written into the destination register is shown to the right of each instruction. Below the program are tables showing the contents of the ID Map, Commit Map, and Physical Register File (PRF) at each cycle. The tables show initial values (before the first instruction is fetched), in the PRF table the right square bracket"]" indicates that the register is free. (Otherwise the right square bracket shows when the register is freed.)
(a) Show where each instruction commits.
(b) Complete the ID and Commit Map tables.
(c) Complete the PRF table. Show the values and use a "[" to indicate when a register is removed from the free list and a "]" to indicate when it is put back in the free list. Be sure to place these in the correct cycle.

```

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

ID Map
r1 96
r2 92

# Cycle 0

Commit Map
r1 96
r2 92

# Cycle 0

Physical Register File
99 112]
98 583 ]
97 174]
96 309
95 606 ]
94 058 ]
93 285 ]
92 1234
91 518 ]
90 207 ]

# Cycle 0

```

\section*{29 Fall 2002}

\section*{EE 4720}

\section*{Homework 1}

Due: \(\mathbf{1 8}\) September 2002
At the time this was assigned computer accounts and solution templates were not ready. If they become available they can be used for the solution, either way a paper submission is acceptable.

Problem 1: Write a MIPS assembly language program that copies and converts an array of integers to an array of doubles. Use the template below.
```

\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#

## cpy_w_to_dbl

    ## Register Usage
    #
    # $a0: Procedure input: Address of start of integer array (to read).
    # $a1: Procedure input: Length of integer array.
    # $a2: Procedure input: Address of start of double array (to write).
    .globl cpy_w_to_dbl
    cpy_w_to_dbl:
\# Your code can modify $a0-$a2 and \$t registers.
\# A correct solution uses 8 instructions (not including jr, nop),
\# a different number of instructions are okay.
\# Points will be deducted for obviously unnecessary instructions.
\#
\# Solution starts here.
jr \$ra
nop

```

Problem 2: What do the Sun compiler -xarch and -xchip options as used below do, and what are the equivalent gcc 2.95 (GNU C compiler) switches.
```

cc myprog.c -o myprog -xarch=v8 -xchip=super

```

See http://gcc.gnu.org/onlinedocs/ for gcc and http://docs.sun.com for the Sun Forte C 6 / Sun Workshop 6 cc compiler.

Problem 3: In Sun's CINT2000 SPEC Benchmark disclosure for the Sun Blade 1000 Model 900 Cu they specify a -xregs=syst compiler flag for several of the benchmarks compiled under the peak rules. Hint: Use a search engine to find this rare flag. Guess what many of the search hits are to?
(a) What does this flag do?
(b) How does it improve performance? Hint: It affects one of the few low-level optimizations covered in class up to this point.
(c) How often could this option be used in the real world?

Problem 4: Benchmark suites are suites because a single program might run well on a processor that runs most other code poorly.

At http://www.spec.org find the fastest processors using the "result" numbers from the SPEC CINT2000 benchmarks in the following categories: The fastest two Pentium 4s, the fastest Athlon, and the fastest Alpha. (Figure out how to get a result-sorted list of machines that shows processor type.)
(a) What programs might an unfair Intel advocate want removed from the suite?

For the parts below consider the relative performance of the programs in the suite. (Put the bar graphs for two different systems side by side and note the difference in shape.)
(b) Why might one expect the top two Pentia to be very similar? Are they in fact very similar?
(c) Why might one expect the Athlon to be more similar to the Pentium than to the Alpha? Does it?

ISA manuals are needed for some problems below. Links to the ISA manuals can be found on the new references Web page: http://www.ece.lsu.edu/ee4720/reference.html

Problem 1: Consider the following SPARC instructions:
```

sub %g3, %g2, %g1 ! g1 = g3 - g2;
and %g1, 0xf, %g1 ! g1 = g1 \& 0xf

```

Wouldn't it be nice to have a sub.and instruction that would do both:
```

sub.and %g3, %g2, 0xf, %g1 ! g1 = ( g3 - g2 ) \& 0xf

```
(a) Could the SPARC V9 ISA easily be extended to support such double-op instructions? If yes, explain how they would be coded.
(b) Estimate how useful double-op instructions would be, using the data below. Usefulness here is conveniently defined as the dynamic instruction count. Consider a large class of double-op instructions that operate on two source registers and an immediate. For example, add.add, sll.add, and and.or. The data below does not provide important statistics needed to estimate the usefulness. Describe what statistics are needed and make up numbers. The made up numbers can be totally arbitrary (as long as they are possible).

The data below show instruction category and immediate sizes running the gcc compiler (cc1). Assume that this is a representative program and so the results apply to others. The data show the total number of instructions, and the breakdown by category, including ALU instructions that use an immediate, ALU instructions that use two source registers, etc. Following that histograms of the immediate sizes are shown for four instruction categories. This is very similar to the data shown in class. The percentage at each size and a cumulative percentage are shown. For example, \(11.12 \%\) of ALU immediate instructions use two bits and \(55.40 \%\) use two bits or fewer.
```

[drop] % echo /opt/local/lib/gcc-lib/sparc-sun-solaris2.6/2.95.2/cc1 \
els.i -03 -quiet isize
Analyzed 156423240 instructions:
48483403 ( 31.0%) ALU Immediate
23886739 ( 15.3%) ALU Two Source Register
6353567 ( 4.1%) sethi
34039161 ( 21.8%) Loads and Stores
30331049 ( 19.4%) Branches
13329321 ( 8.5%) Other
ALU Immediate Size Distribution
Bits Pct Cum
25.96% 25.96% ********************
18.32% 44.28% **************
11.12% 55.40% ********
15.59% 70.99% ************
3.16% 74.15% ***
6.10% 80.25% *****
7.31% 87.56% ******
6.81% 94.37% *****
0.76% 95.13% *
2.13% 97.26% **
2.64% 99.90% **
0.01% 99.91% *
0.08% 99.99% *
0.01% 100.00% *

```
```

SETHI Immediate Size Distribution
Bits Pct Cum
0 0.00% 0.00% *
1 0.00% 0.00% *
2 0.02% 0.02% *
0.72% 0.74% *
0.43% 1.17% *
0.09% 1.26% *
0.66% 1.93% *
2.00% 3.93% **
4.53% 8.45% ****
3.14% 11.60% ***
10 5.86% 17.46% *****
11 5.54% 23.00% ****
72.67% 95.67% ******************************************************
0.09% 95.76% *
0.13% 95.89% *
0.10% 95.99% *
0.01% 96.00% *
0.00% 96.01% *
0.49% 96.50% *
3.14% 99.63% ***
0.02% 99.66% *
0.33% 99.99% *
0.01% 100.00% *
Memory Offset Distribution
Bits Pct Cum
0 4.93% 4.93% ****
0.11% 5.04% *
2.51% 7.55% **
15.95% 23.50% ************
24.41% 47.91% *******************
10.36% 58.27% *********
4.90% 63.17% ****
6.89% 70.06% *****
8 5.79% 75.85% *****
9 4.98% 80.82% ****
16.09% 96.92% *************
2.38% 99.30% **
0.70% 100.00% *
0.00% 100.00%
Branch Displacement Distribution

| Bits | Pct | Cum |  |
| :---: | :---: | :---: | :---: |
| 0 | 0.00\% | 0.00\% |  |
| 1 | 0.00\% | 0.00\% | * |
| 2 | 13.06\% | 13.06\% | ********** |
| 3 | 22.20\% | 35.26\% | **************** |
| 4 | 16.58\% | 51.84\% | ************ |
| 5 | 18.94\% | 70.79\% | ************** |
| 6 | 5.69\% | 86.48 | *********** |

```
```

    6.74% 93.22% *****
    3.47% 96.69% ***
    1.63% 98.31% **
    0.71% 99.02% *
    0.27% 99.29% *
    0.41% 99.69% *
    0.01% 99.71% *
    0.00% 99.71%
    0.00% 99.71% *
    0.21% 99.91% *
    0.01% 99.92% *
    0.08% 100.00% *
    0.00% 100.00%
    0.00% 100.00%
    0.00% 100.00%
    0.00% 100.00%
    ```

Problem 2: It's time to go instruction hunting!
(a) The Alpha does not have a general set of double-op instructions but it does have one that can replace the two SPARC V9 instructions below. What is it? Replace the two instructions below with the Alpha instruction. (For full credit [another 0.5 point, maybe] take into account that SPARC V9 and not SPARC V8 was specified.)
```

sll %g2, 2, %g1 ! g1 = g2 << 2;
add %g3, %g1, %g1 ! g1 = g3 + g1

```
(b) SPARC V9 does not have a full set of predicated instructions, but it does have a predicated instruction that can replace the code fragment below. What is it?
```

subcc %g1, 0, %g0 ! Set integer condition codes.
be SKIP ! Branch if result equal to zero.
nop
add %g3, 0, %g4 ! g4 = g3 + 0
SKIP:

```

Problem 3: Complete Spring 2002 Homework 2 Problems 2 and 3.
(At http://www.ece.lsu.edu/ee4720/2002/hw02.pdf.) (The Verilog part is optional.) This is a very important type of problem, similar problems will be appearing all semester. You must solve the problem, that is, scratch your head, figure it out, and work it through. If you're stuck, feel free to ask for help. When you're done look at a solution and assign yourself a grade. Grade on a scale of 0 to 1 (real, not integer!)

Not solving it or solving it with too many glances at the solution will leave you ill-prepared for the test. Yes, you can solve it the night before the test (if you have time), but that won't help you understand everything presented in class between now and then. You have been warned.

Problem 1: Consider the solution to Spring 2002 Homework 4, shown on the next page. (The solution was updated 19 November 2002, the PED is shown in dynamic order instead of the nearly-impossible-to-read static order.)
(a) Show the contents of the reorder buffer in cycle 12. For each entry show the values of the fields from the illustration below, for the PC show the instruction (ldc1, mul.d, etc.). (The fields are ST, dst, dstPR, and incumb.) If a field value cannot be determined from the solution leave it blank, that will include fields related to registers \$2 and \$3.
(b) For the solution to the part above, number each instruction. (1, 2, 3, etc.) Show the contents of the instruction queue at cycle 12 identifying each instruction by these numbers.
(c) On the illustration there are three wires labeled with big lower-case letters, \(\mathrm{a}, \mathrm{b}\), and c , and corresponding rows in a table in the middle of the next page. Based on the solution to last semester's problem, show what values are on those wires in each cycle that they are used. Omit cycles where a value cannot be determined. Note that the illustration is for a one-way (non-superscalar) processor but the program runs on a four-way system. That means each wire can hold up to four values in one cycle. Hint: The solution for at least one of the letters already appears. Just label the row(s) in the appropriate table with the letter. At least one of the letters does not appear, so that will have to be written in.



To answer the questions below you need to use the PSE dataset viewer program. PSE (pronounced see) runs on Solaris and Linux; you can use the computer accounts distributed in class to run it, a Linux distribution may also be provided for running it on other systems.

Procedures for setting up the class account and using PSE are at http://www.ece.lsu.edu/ee4720/proc.html; preliminary documentation for PSE is at http://www.ece.1su.edu/ee4720/pse.pdf.

Problem 1: Near the beginning of the semester the performance of a program to compute \(\pi\) was evaluated with and without optimization. It's back, down below.

Follow instructions referred to above to view the execution of the optimized and unoptimized versions of the pi program running on a simulated 4 -way dynamically scheduled superscalar machine with a 48 -instruction reorder buffer. The datasets to use are pi_opt.ds and pi_noopt.ds.
(a) Based on the pipeline execution diagram compute the CPI of the main loop for a large number of iterations in the optimized version. Do not use the IPC displayed by PSE, instead base it on the PED. In your answer describe how the CPI was determined.
(b) Consider first the optimized version of the program. Would it run faster with a larger reorder buffer? Would it run faster on an 8 -way superscalar machine? How else might the processor be modified to improve performance? Explain each answer.
(c) Now consider the un-optimized version. Would it run faster with a larger reorder buffer? Should a computer designer pay attention to the performance of un-optimized code? Explain each answer.
(d) The simulated processors use a gshare branch predictor. Use pi_opt.ds to answer this question. How many bits is the global history register? Entries in the PHT are initialized to 1 and the GHR is initialized to zero. The PHT is updated when the branches resolve (in the cycle after they execute). Explain your answer.
(e) Would execution be any different if the PHT were updated when the instructions commit?
```

\#include <stdio.h>
int
main(int argv, char **argc)
{
double i;
double sum = 0; // Line 7
for(i=1; i<5000;) // Line 9
{
sum = sum + 4.0 / i; i += 2; // Line 11
sum = sum - 4.0 / i; i += 2; // Line 12
}
printf("After %d iterations sum = %.8f\n", (int)(i-1)/2, sum); // Line 15
return 0;
}

```

To answer the questions below you need to use the PSE dataset viewer program. PSE (pronounced see) runs on Solaris and Linux; you can use the computer accounts distributed in class to run it, a Linux distribution may also be provided for running it on other systems.

Procedures for setting up the class account and using PSE are at http://www.ece.lsu.edu/ee4720/proc.html; preliminary documentation for PSE is at http://www.ece.lsu.edu/ee4720/pse.pdf.

Problem 1: The code in http://www.ece.lsu.edu/ee4720/2002f/hw6.pdf includes two routines to perform a linear search, lookup_array and lookup_ll. Routine lookup_array (aws,foo) searches aws for element foo. The list itself is an ordinary C array, structure aws (array with size) includes the array and its size. Routine lookup_ll(head,foo) searches for foo in the linked list starting at head.

The code calls the search routines under realistic conditions: Before the linked list is allocated dynamic storage is fragmented and before the searches are performed the level- 1 cache is flushed. See the code for more details.

The code was executed on a simulated 4 -way superscalar dynamically scheduled machine with a 64 -entry reorder buffer and a two-level cache. The simulation was recorded in hw6.ds; view this dataset file using PSE to answer the questions below.

The code initializes the lists with identical data and then calls the search routines looking for the same value. Answer the following questions about the execution of the two lookup routines. When browsing the dataset be aware that the time spent in the lookup routines is dwarfed by the time needed for setting everything up and so only the last few segments need to be examined.
(a) Would increasing the ROB size improve the performance of the linked list routine, lookup_11? Explain.
(b) Would increasing the ROB size improve the performance of the array routine, lookup_array? Explain.
(c) As can be seen viewing the PED plots, the array routine follows a regular pattern while the linked list code starts off slowly but as it nears completion it runs much faster. Why does the linked list code speed up like that?
(d) How could one determine the line size from the PED plots? Be specific and use numbers from the dataset. (The line size can be found two other ways, if you come upon them by all means use them to check your answer that is based on the PED plot.)
(e) Before people stopped replacing \(\$ 2,500\) computers every six months computer engineers would loose sleep worrying about The Memory Wall, the growing gap in performance between processors and memory (e.g., the number of instructions that could have been executed while waiting for memory). What is it about the array routine that lets it sail over the memory wall while the linked list routine is stopped dead? The answer should take into account certain load instructions and the critical path. Discuss how the performance of the routines change as the L1 miss time gets longer and longer.

\section*{30 Spring 2002}

At the time this was assigned computer accounts and solution templates were not available. If they become available they can be used for the solution, either way a paper submission is acceptable.

Problem 1: The value computed by the program below approaches \(\pi\). Re-write the program in MIPS assembler. The code should execute quickly. Assume that all integer instructions take one cycle, floating-point divides take ten cycles, floating-point compares take one cycle, and all other floating-point instructions, including conversion, take four cycles. Note: As originally assigned only the time for divides and adds was given. Make changes to the code to improve speed (possibly using an integer for i or even using both an integer and double). Do not use a different technique for computing \(\pi\).
```

int
main(int argv, char **argc)
{
double i;
double sum = 0;
for(i=1; i<50000000;)
{
sum = sum + 4.0 / i; i += 2;
sum = sum - 4.0 / i; i += 2;
}
printf("After %d iterations sum = %.8f\n", (int)(i-1)/2, sum);
return 0;
}

```

Problem 2: The program below is used to generate a password based on the outcome of several rolls of a twenty-sided die. The program was compiled using the Sun Workshop Compiler 5.0 targeting SPARC V7 (-xarch=v7) and SPARC V9 (-xarch=v8plus, code which can run on a V9 processor with a 32 -bit OS), the output of the compiler is shown for the for loop.

Use the V8 architecture manual to look up V7 instructions, available at
https://www.ece.lsu.edu/ee4720/samv8.pdf; the V9 architecture manual is available at https://www.ece.1su.edu/ee4720/samv9.pdf.

Here are a few useful facts about SPARC:
Register names for SPARC are: \(\% \mathrm{~g} 0-\% \mathrm{~g} 7\) (global), \(\% 10-\% 17\) (local), \(\% \mathrm{in} 0-\% \mathrm{i} 7\) (input), \(\% 00-\% \mathrm{o} 7\) (output), and \(\% \mathrm{f} 0-\% \mathrm{f} 31\) (floating point). Registers \(\% \mathrm{f} \mathrm{p}\) (frame pointer) and \(\%\) sp are aliases for \(\% \mathrm{i} 6\) and \(\% \circ 6\), respectively. Register \(\%\) g0 is a zero register.

Local variables (the only kind used in the code fragment shown) are stored in memory at some offset from the stack pointer (in \%sp). For example, ldd \([\% s p+96]\), \%f0 loads a local variable into register \%f0.

All V7 and V8 integer registers are 32 bits. V9 registers are 64 bits but with the v8plus option only the 32 lower bits are used.

Unlike MIPS and DLX, the last register in an assembly language instruction is the destination. For example, add \(\% \mathrm{~g} 1, \% \mathrm{~g} 2, \% \mathrm{~g} 3\), puts the sum of g 1 and g 2 in register g 3 .

Like MIPS, SPARC branches are delayed. Unlike MIPS, some delayed branches are annulled, indicated with a ",a" in the mnemonic. In an annulled branch the instruction in the delay slot is executed if and only if the branch is taken.
(a) For each compilation, identify which registers are used for which program variables.
(b) For each instruction used in the V9 version of the code but not in the V7 version, explain what it does and how it improves execution over the V 7 version.
```

int
main(argc, argv)
int argc;
char **argv;
{
int die_rolls[] = {15, 17, 6, 10, 19, 19, 15, 17, 16, 5, 0 };
int *rolls_ptr = \&die_rolls[0];
char pw[8];
char *pw_ptr = \&pw[0];
int faces_per_die = 20; /* Available at Little Wars in Village Square */
double bits_per_roll = log(faces_per_die)/log(2.0);
double bits_per_letter = log(26.0) / log(2.0);
double bits = 0.0;
uint64_t seed = 0; /* A 64-bit integer. */
int roll;
while( ( roll = *rolls_ptr++ ) )
{
seed = faces_per_die * seed + (roll-1);
bits += bits_per_roll;
}
for( ; bits >= bits_per_letter; bits -= bits_per_letter )
{
*pw_ptr++ = 'a' + seed % 26;
seed = seed / 26;
}
*pw_ptr = 0;
printf("The password is %s\n",pw);
return 0;
}
! Compiled with -xarch=v7
!
! 32 ! for( ; bits >= bits_per_letter; bits -= bits_per_letter )
/* 0x010c 32 */ ldd [%sp+96],%f0
/* 0x0110 32 */ fcmped %f30,%f0
/* 0x0114 */ nop
/* 0x0118 */ fbul .L77000009
/* 0x011c */ or %g0,0,%o2
! 33 ! {

```
```

! ! 34 *pw_ptr++ = 'a' + seed % 26;

```

```

! 35 ! seed = seed / 26;

| /* 0x013c | 35 */ | or | \%g0,\%i1,\%o0 |
| :---: | :---: | :---: | :---: |
| /* 0x0140 | */ | or | \%g0,0,\%o2 |
| /* 0x0144 | */ | or | \%g0,26,\%o3 |
| /* 0x0148 | */ | call | __udiv64 |
| sult $=\%$ o0 |  |  |  |
| /* 0x014c | */ | or | \%g0,\%i2,\%o1 |
| /* 0x0150 | */ | ldd | [\%sp+96], \%f0 |
| /* 0x0154 | 34 */ | add | \%i0,1,\%i0 |
| /* 0x0158 | 35 */ | or | \%g0,\%o0,\%i1 |
| /* 0x015c | */ | ldd | [\%sp+104],\%f30 |
| /* 0x0160 | */ | fsubd | \%f30,\%f0,\%f30 |
| /* 0x0164 | */ | fcmped | \%f30, \%f0 |
| /* 0x0168 | */ | or | \%g0,\% $\%$, \%i2 |
| /* 0x016c | */ | fbge | .L900000116 |
| /* 0x0170 | */ | or | \%g0,0,\%o2 |
| .L77000009: |  |  |  |


| ! | 36 | ! |
| :--- | :--- | :---: |
| $!$ | Compiled | With |
| -xarch=v8plus |  |  |

Compiled With -xarch=v8plus
! 32 ! for( ; bits >= bits_per_letter; bits -= bits_per_letter )
/* 0x00e8 32 */ fcmped %fcc0,%f8,%f4
/* 0x00ec 32 */ % fbul,a,pt % %fcc0,.L900000115
/* 0x00f0 */ stb %g0,[%i0]

| $!$ | 33 | $!$ |
| :--- | :--- | :--- |
| $!$ | 34 | $!$ |$\quad$ *pw_ptr++ = 'a' + seed $\% 26 ;$

/* 0x00f4 34 */ udivx %o0,26,%g2
.L900000114:
/* 0x00f8 34 */ mulx %g2,26,%g3
/* 0x00fc */ sub %o0,%g3,%g3
! 35 ! seed = seed / 26;

| /* | $0 \times 0100$ | 35 | $* /$ |
| ---: | ---: | :--- | :--- |
| /* | 0 x 0104 | $* /$ | or |
| /* | $0 x 0108$ | 34 | */ |

```
```

/* 0x010c
/* 0x0110
/* 0x0114
/* 0x0118
/* 0x011c

```
```

    */ stb %g3,[%i0]
    ```
    */ stb %g3,[%i0]
    */ add %i0,1,%i0
    */ add %i0,1,%i0
35 */ fcmped %fcc1,%f8,%f4
35 */ fcmped %fcc1,%f8,%f4
    */ fbge,a,pt %fcc1,.L900000114
    */ fbge,a,pt %fcc1,.L900000114
    */ udivx %o0,26,%g2
    */ udivx %o0,26,%g2
    .L77000009:
```

    .L77000009:
    ```
    \}

Problem 1: Two VAX instructions appear below. VAX documentation can be found via http://www.ece.lsu.edu/ee4720/doc/vax.pdf. Don't print it, it's 544 pages. Take advantage of the extensive bookmarking of the manual to find things quickly. Chapter 5 describes the addressing modes and assembler syntax, Chapter 8 summarizes the VAX ISA, and Chapter 9 lists the instructions. For the instructions look up ext and add then find the mnemonics used below. Pay attention to operand order.
(a) Translate the VAX code below to MIPS (without changing what it does, of course). Ignore overflows and the setting of condition codes.
```

extzv \#10, \#5, r1, r2
addl2 @0x12034060(r3), (r4)+ \# Don't overlook the "@" and "+".

```
(b) (Extra Credit) Show how the instructions above are coded.

Problem 2: A pipelined MIPS implementation and some MIPS code appear below. The results computed by the MIPS instructions are shown in the comments.

```

LOOP: \# LOOP = 0x1000
addi \$1, \$2, 4 \# 0x24
sub \$3, \$0, \$3 \# 0x30
and \$1, \$1, \$6 \# 0x20
or \$4, \$1, \$5 \# 0x70
bne \$4, \$3, LOOP \# Taken
sw \$4, 7(\$8) \# \$8 = 0x801
add \$10, \$11, \$12 \# 0x230
add \$13, \$11, \$12 \# 0x230
add \$14, \$11, \$12 \# 0x230

```
(a) Draw a pipeline execution diagram showing the execution of the code on the implementation. Base your pipeline execution diagram on the illustrated pipeline, do not depend solely on memorized execution timing rules, since they depend on details of the hardware which vary from problem to problem. Show execution until the second fetch of the first instruction.
(b) Determine the CPI for a large number of iterations.
(c) Certain wires in the implementation diagram are labeled with letters. (The circled letters with arrows.) Beneath the pipeline execution diagram show the value on those wires at near the end of each cycle. (Write sideways if necessary.) Do not show values if the corresponding stage holds a bubble or a squashed instruction. Only show immediate values if the corresponding instruction uses one. Hint: Three instructions above use an immediate.
(d) This is a special bonus question that did not appear in the original assignment! For those students who have taken EE 3755 in Fall 2001, identify the Verilog code in http://www.ece.lsu.edu/ee4720/v/mipspipe.html corresponding to each labeled wire.

Problem 3: Add exactly the bypass paths needed so that the code in the previous problem will run on the implementation below (the same as the one above) with the minimum number of stalls. Indicate the cycles in which the bypass paths will be used and the values bypassed on them.

Problem 1: The exception mechanism used in the MIPS 32 ISA differs in some ways from the SPARC V8 mechanism covered in class. See Chapter 7 in http://www.ece.lsu.edu/ee4720/sam.pdf for the SPARC V8 exception information and http://www.ece.lsu.edu/ee4720/mips32v3.pdf for a description of the MIPS mechanism. The MIPS description is a bit dense, so start early and ask for help if needed.
(a) Describe how the methods used to determine which exception was raised differ in SPARC V8 and MIPS 32. Use an illegal (reserved) instruction error as an example. Shorter answers will get more credit so concentrate on explaining how the processor identifies the exception (was it an illegal instruction, an arithmetic overflow, etc) and avoid irrelevant details. For example, details on how the processor switches to system mode is irrelevant.
(b) Where do the two ISAs store the address of the faulting instruction? Both ISAs have delayed branches, so why does SPARC store two return addresses while MIPS gets away with one?
(SPARC registers are organized like a stack, on a procedure call a save instruction "pushes" a fresh set of registers on the stack, and a restore instruction "pops" the registers, returning to the previous set. The set of visible registers is called a window. This mechanism reduces the need to save and restore registers in memory. This piece of information is needed for the previous problem.)

Problem 2: The pipeline execution diagram below is for code running on a MIPS implementation developed just for this homework problem! Note that the program itself is missing. The dog deleted it. The \(M_{-}\)and \(A_{-}\)refer to parts of the multiply and add functional units with segment numbers omitted for this problem. A WBx indicates that an instruction does not write back to avoid a WAW hazard.
\[
\begin{aligned}
& \text { IF ID M_ M_ M_ M_ M_ M_ WB } \\
& \text { IF ID ----> M_ M_ M_ M_ M_ M_ WB } \\
& \text { IF ----> ID ----> } A_{-} A_{-} \text {WB } \\
& \text { IF ----> ID M_ M_ M_ M_ M_ M_ WBx } \\
& \text { IF ID A_ A_ WB } \\
& \text { IF ID A_ A_ WB }
\end{aligned}
\]
(a) Write a program consistent with the diagram. Pay attention to dependencies.
(b) Identify the latency and initiation interval of the functional units. Fill in the segment numbers.

Problem 3: In the MIPS implementation below (also shown in class) branches are resolved in the ID stage. Resolution of a branch direction (determining whether it was taken) must wait for register values to be retrieved and, for some branches, compared to each other. Suppose this takes too long.

(a) Show the modifications needed to do the equality comparison in the EX stage. The modified hardware must use as little additional hardware as possible and, to maximize performance, should only do an EX-stage equality comparison when necessary. Don't forget about branch target address handling. Hint: The modifications are easy.
(b) Write a code fragment that runs differently on the two implementations and show pipeline execution diagrams for the code on the two implementations.
(c) The table below lists SPARC instructions and indicates how frequently they were used when running \(\mathrm{T}_{\mathrm{E}} \mathrm{X}\) to prepare this homework assignment. (Many rows were omitted to save space, so the "\%exec" column will not add to \(100 \%\).) Suppose that the instruction percentages are identical for MIPS (which means totally ignoring the cc instructions). Assume that SPARC be and be, a are equivalent to MIPS beq, SPARC bne and bne, a are equivalent to MIPS bne, and that the other branch instructions (they begin with a b), are equivalent to branch instructions that compare to zero (bgez, etc.).

Suppose the clock frequency of the original design is 1.0000 GHz . Based on the data below and making any necessary assumptions, for what clock frequency would the new design run a program in the same amount of time as the old one? What column would you add (what additional data do you need) to the table to make your answer more precise?

Assume that floating-point instructions are insignificant and that there are no stalls due to memory access.
\begin{tabular}{lrr} 
opcode & \#exec & \(\%\) exec \\
subcc & 4659360 & \(12.6187 \%\) \\
lduw & 4521722 & \(12.2459 \%\) \\
add & 4159629 & \(11.2653 \%\) \\
or & 3110542 & \(8.4241 \%\) \\
sethi & 3066797 & \(8.3056 \%\) \\
stw & 1848293 & \(5.0056 \%\) \\
sll & 1402122 & \(3.7973 \%\) \\
be & 1393475 & \(3.7739 \%\) \\
jmpl & 1140223 & \(3.0880 \%\) \\
call & 1088068 & \(2.9467 \%\) \\
ldub & 1064918 & \(2.8841 \%\) \\
bne & 936493 & \(2.5362 \%\) \\
stb & 687981 & \(1.8632 \%\) \\
srl & 609402 & \(1.6504 \%\) \\
save & 526477 & \(1.4258 \%\) \\
restore & 526474 & \(1.4258 \%\) \\
bne, a & 453545 & \(1.2283 \%\) \\
nop & 433253 & \(1.1734 \%\) \\
bge & 429978 & \(1.1645 \%\) \\
ldsb & 429497 & \(1.1632 \%\) \\
orcc & 382947 & \(1.0371 \%\) \\
and & 370967 & \(1.0047 \%\) \\
be, a & 360057 & \(0.9751 \%\) \\
sub & 354847 & \(0.9610 \%\) \\
ba & 321970 & \(0.8720 \%\) \\
bl & 297715 & \(0.8063 \%\) \\
andcc & 270465 & \(0.7325 \%\) \\
bgu & 235304 & \(0.6373 \%\) \\
bl, a & 216074 & \(0.5852 \%\) \\
sra & 204610 & \(0.5541 \%\) \\
ble & 198154 & \(0.5366 \%\) \\
xor & 185137 & \(0.5014 \%\) \\
bcs & 182153 & \(0.4933 \%\) \\
addcc & 155156 & \(0.4202 \%\) \\
bleu & 142755 & \(0.3866 \%\) \\
bg & mulscc & \(0.3184 \%\) \\
& \(0.2402 \%\) \\
& &
\end{tabular}

To solve Problem 3 and the next assignment a paper has to be read. Do not leave the reading to the last minute, however try attempting the first problem below before reading the paper.

Problem 1: The pipeline below was derived from the five-stage statically scheduled MIPS implementation by splitting each stage (except writeback) into two stages. Each pair of stages (say IF1 and IF2) does the same thing as the original stage (say IF), but because it is broken in to two stages it takes two rather than one clock cycle. The diagram shows only a few details. Bypass connections into the ALU are available from all stages from MEM1 to WB.


The advantage of this pipeline is that the clock frequency can be doubled. (Actually not quite times two.) Perfect execution is shown in the diagram below:
```

add \$1, \$2, \$3 IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB
sub \$4, \$5, \$6 IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB
and \$7, \$8, \$9 IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB

```
(a) Suppose the old five-stage system ran at a clock frequency of 1 GHz and the new system runs at 2 GHz . How does the execution time compare on the new system when execution is perfect?
(b) Show a pipeline execution diagram of the code below on the new pipeline. Note dependencies through registers \(\$ 10\) and \(\$ 11\).
```

add \$10, \$2, \$3
sub \$4, \$10, \$6
and \$11, \$8, \$9
or \$20, \$21, \$22
xor \$7, \$11, \$0

```
(c) In the previous part there should be a stall on the new pipeline that does not occur on the original pipeline. (It's not too late to change your answer!) How does that affect the usefulness of splitting pipeline stages?
(d) (Optional, complete before reading paper.) To get that I'm-so-clever feeling answer the following: Suppose there is no way a 32 -bit add can be completed in less than two cycles. Is there any way to perform addition so that results can be bypassed to an immediately following instruction, as in the example above, but without stalling? The technique must work when adding any two 32 bit numbers. Hint: The adder would have to be redesigned. (A question in the next homework assignment revisits the issue.)

Problem 2: Note: The following problem is similar to one given in the Fall 2001 semester, see http://www.ece.lsu.edu/ee4720/2001f/hw03.pdf (the problem) and http://www.ece.lsu.edu/ee4720/2001f/hw03_sol.pdf (the solution). For best results do not look at the solutions until you're really stuck. This problem below uses MIPS instead of DLX and is for Method 3 instead of Method 1. The code below executes on a dynamically scheduled four-way superscalar MIPS implementation using Method 3, physical register numbers.
- Loads and stores use the load/store unit, which consists of segments L1 and L2.
- The floating-point multiply unit is fully pipelined and consists of six segments, M1 to M6.
- The usual number of instructions (for a 4-way superscalar machine) can be fetched, decoded, and committed per cycle.
- An unlimited number of instructions can complete (but not commit) per cycle. (Not realistic, but it makes the solution easier.)
- There are an unlimited number of reservation stations, reorder buffer entries, and physical registers.
- The target of a branch is fetched in the cycle after the branch is in ID, whether or not the branch condition is available. (We'll cover that later.)
(a) Show a pipeline execution diagram for the code below until the beginning of the fourth iteration. Show where instructions commit.
(b) What is the CPI for a large number of iterations? Hint: There should be less than six cycles per iteration.
(c) Show the entries in the ID and commit register maps for registers \(f 0\) and \(\$ 1\) for each cycle in the first two iterations. If several values are assigned in the same cycle show each one separated by commas.
(d) Show the values in the physical register file for f 0 and \(\$ 1\) for the first two iterations. Use a "]" to show when a physical register is removed from the free list and use a "[" to show when it is put back in the free list.
```

LOOP: \# LOOP = 0x1000
ldc1 f0, 0(\$1)
mul.d f0, f0, f2
sdc1 0(\$1), f0
addi \$1, \$1, 8
bne \$2, \$3 LOOP
sub \$2, \$1, \$3
\# Initial Values
free list: 99, 98, 97, etc. The next free register is 99.
f0: 1.0
\$1: 0x1000
f2: 1.1
Mem[0x1000] = 10.0
Mem[0x1008] = 20.0
Mem[0x1010] = 30.0
etc.

```

The following is an introduction to the next few problems.
As mentioned several times in class many of the performance-enhancing microarchitectural features that came in to wide use in the closing decades of the twentieth century (I love the way that sounds!) are much easier to apply to RISC ISAs than CISC ISAs. Bound by golden handcuffs to the CISCish IA-32 ISA, Intel was forced to get RISC-level performance from IA-32. (Not just Intel, DEC [now Compaq, perhaps soon HP] faced the problem with the VAX ISA and IBM with 360.) The solution chosen by Intel (and also DEC) was to translate individual IA-32 instructions in to one or more \(\mu \mathrm{ops}\) (micro-operations). Each \(\mu \mathrm{op}\) is something like a RISC instruction and so the parts of the hardware beyond the IA-32 to \(\mu\) op translation can employ the same techniques used to implement RISC ISAs.

The paper at http://www.intel.com/technology/itj/q12001/articles/art_2.htm
and http://www.ece.lsu.edu/ee4720/s/hinton_p4.pdf (password needed off campus, will be given in class) describes the Pentium 4 implementation of IA-32, including \(\mu\) ops (which are typeset using "u" instead of the Greek letter " \(\mu\) ", except occasionally in figures). This paper was not written for a senior-level computer architecture class four weeks from the end of the semester and so it will include material which we have not yet covered (caches and TLBs) and some material not covered at all. Some stuff in the paper is not explained (how they do branch prediction or what the Pentium 4 pipeline segments in Figure 3 mean), some of this can be figured out other things have to be found out elsewhere (but not for this assignment).

Read the paper and answer the question below. The next homework assignment will include additional questions on the paper. For this initial reading skip or lightly read material on the L2 cache, L1 data cache, and the ITLB. Questions on the cache material might be asked in a later assignment.

Problem 3: What does the paper call the following actions and components (that is, translate from the terminology used in class to the terminology used in the paper):

Commit
ID Register Map
Commit Register Map
Physical Register File

The following questions are based on the paper at http://www.intel.com/technology/itj/q12001/articles/art_2.htm and http://www.ece.lsu.edu/ee4720/s/hinton_p4.pdf (password needed off campus, will be given in class). See Homework 4 (http://www.ece.lsu.edu/ee4720/2002/hw04.pdf) for an introduction to the paper.

Problem 1: What is the maximum IPC of the IA-32 (in \(\mu \mathrm{ops}\) )? Put another way, the Pentium 4 is an \(n\)-way superscalar processor, what is \(n\) ?

Problem 2: The Pentium 4 can decode no more than one IA-32 instruction per cycle. How then can it execute more than one IA-32 instruction per cycle (at least for small code fragments prepared by a friendly programmer)?

Problem 3: One problem with superscalar systems noted in class is the wasted instructions following the delay slot of a taken branch near the beginning of a fetch group. How does the Pentium 4 avoid this?

Problem 4: The fast \((2 \times)\) integer ALUs have three stages, an initiation interval of 1 fast cycle \(\left(\frac{1}{2}\right.\) processor cycle), and a latency of zero fast cycles. Why is this surprising (not the one half part)? How does it do it?

Problem 5: In describing store-to-load forwarding the paper describes a special case for which data could be forwarded (bypassed) but is not because it would be too costly. Using MIPS code (or IA-32 if you prefer) provide an example of this special case.

Problem 6: In Figure 8 the performance of a 1 GHz Pentium III is compared to a 1.5 GHz Pentium 4. Why is it reasonable for the Pentium 4 to be compared at a higher clock frequency?

\section*{31 Fall 2001}

Problem 1: In DLX the three instructions below, though they do very different things, are of the same type (format).
```

bnez r2, SKIP
lw r1, 1(r2)
addi r1, r2, \#1
SKIP:

```

Because of their similarity their implementations in the diagram below shares a lot of hardware.

(a) Show how these DLX instructions are coded.
(b) Find corresponding instructions in the SPARC V9 ISA. (See the SPARC Architecture Manual V9, http://www.ece.lsu.edu/ee4720/samv9.pdf) (The DLX branch instruction will have to be replaced by two instructions, one to set the condition code registers.)
(c) Show the coding of the SPARC V9 branch, load, and add immediate instructions (but not the condition code setting instruction).
(d) Do these codings allow the same degree of hardware sharing?

Problem 2: Write a DLX assembly language program that determines the length of the longest run of consecutive elements in an array of words. For example, in array \(\{1,7,7,1,5,5,5,7,7\}\) the longest run is three: the three 5's (the four 7's are not consecutive). The comments below show how registers are initialized and where to place the longest run length.
```

! r10 Beginning of array (of words).
! r11 Number of elements.
! r1 At finish, should contain length of the longest run.

```

Problem 3: Small integers can be stored in a packed array to reduce the amount of storage required; the array can be unpacked into an ordinary array when the data is needed. Write a DLX assembly language program to unpack an array containing \(n b\)-bit integers stored as follows. The low \(b\) bits (bits 0 to \(b-1\) ) of the first word of the packed array contain the first integer, bits \(b\) to \(2 b-1\) contain the next, and so on. When the end of the word is reached integers continue on the second word, etc. Size \(b\) is not necessarily a factor of \(n\) and so an integer might span two words.

The diagram below shows how the first 6 integers \(i_{0}, i_{1}, \ldots, i_{5}\) are stored for \(b=12\) bits and \(n \geq 6\).


Write DLX assembly language code to unpack such an array into an array of signed words. The packed array consists of \(n b\)-bit signed numbers, with \(b \in[1,32]\). Initial values of registers are given below.
```

! Initial values
! r10: Address of start of packed array.
! r11: Number of elements (n).
! r12: Size of each element, in bits (b).
! r14: Address of start of unpacked array.

```

Problem 1: Answer the following questions about the MIPS Technologies 4Km processor core. The processor is documented in http://www.mips.com/declassified/Declassified_2000/MD00016-2B-4K-SUM-01.15.pdf.
(a) For each stage in the statically scheduled DLX implementation show where the same work is done in the 4 Km pipeline. Note that work done in one DLX stage might be performed in more than one 4 Km pipeline stage.
(b) The 4 Km documentation uses the term stall differently than used in class. How do their usages differ? What term does the documentation use that is close to stall as used in class? (See section 2.8.1)
(c) A MIPS implementation needs to do all of the following:
(1) arithmetic and logical operations for ordinary instructions
(2) compute the target of a branch
(3) compute the effective address of a load or store

In the first pipelined DLX implementation all of these were performed by the ALU. MIPS has a branch instruction in which a branch is taken if two registers are equal (beq) or not equal (bne). So it must also
(4) determine if two values are equal

How many of these are shared? If they are not shared, why not? (The documentation does not state exactly what hardware is present, answer the question by looking at how instructions execute.)

Problem 2: The program below runs on the DLX implementation shown below. The hardware makes no special provisions for the tricky technique used. The coding for a nop (actually add r0, r0, r0) is all zeros.

Why isn't this an infinite loop? (For those who know why it matters, assume there is no cache.)

Why will the code run for at least one iteration?


LOOP:
lw r1, \(0(r 2)\)
addi r2, r2, \#4
add r3, r3, r1
sw \(0 \times 100(r 0)\), r0
LINE: LINE \(=0 \times 100\)
j LOOP

Problem 3: Show a pipeline execution diagram for the code below running on a 4 -way statically scheduled superscalar processor. All needed bypass paths are available, including one for the branch condition. Determine the CPI for a large number of iterations.
```

and r2, r2, r8
LOOP: ! LOOP = 0x1008
lw r1, O(r2)
add r3, r3, r1
addi r2, r2, \#4
sub r4, r2, r5
bneq r4, LOOP

```

Problem 4: The code from the problem above can be improved (stalls can be removed) to a small extent by scheduling, but that would still leave some stalls. This might see like a good candidate for loop unrolling.
(a) Show why it would take alot of unrolling to eliminate all stalls. (You don't have to show the unrolled code, since it would be long.)
(b) Use software pipelining and scheduling to remove the stalls. (Hint: to software pipeline switch the 1 w and add instructions, and make any other necessary changes.) What is the CPI for a large number of iterations of the modified code?
(c) Would loop unrolling provide further gains?

Problem 1: The code below executes on a dynamically scheduled four-way superscalar DLX implementation that uses reorder buffer entry numbers to name destination registers.
- Loads and stores use the load/store unit, which consists of segments L1 and L2.
- The floating-point multiply unit is fully pipelined and consists of six segments.
- The usual number of instructions (for a 4 -way superscalar machine) can be fetched, decoded, and committed per cycle.
- An unlimited number of instructions can complete per cycle. (This makes the solution easier.)
- There are an unlimited number of reservation stations and reorder buffer entries.
- The target of a branch is fetched in the cycle after the branch is in ID, whether or not the branch condition is available. (We'll cover that later.)
(a) Show a pipeline execution diagram for the code below until the beginning of the fourth iteration. Show where instructions commit.
(b) What is the CPI for a large number of iterations? Hint: There should be less than six cycles per iteration.
(c) Show the entries in the register map for registers \(f 0\) and \(r 1\) for each cycle. (Make up reorder buffer entry numbers.)
```

LOOP: ! LOOP = 0x1000
ld f0, O(r1)
muld f0, f0, f2
sw O(r1), f0
addi r1, r1, \#8
sub r2, r1, r3
bnez r2, LOOP

```
(d) The first two instructions of the code below are different than the code above, the other instructions are identical. It runs on a system identical to the one above except that there are only 1000 reorder buffer entries. (That's actually a lot, but it's not unlimited.) What is the CPI for a large number of iterations? Is the CPI really lower in the period before reorder buffers are used up? If you can, solve the problem without drawing a complete pipeline execution diagram.
```

LOOP: ! LOOP = 0x1000
ld f4, O(r1)
muld f0, f0, f4
sw O(r1), f0
addi r1, r1, \#8
sub r2, r1, r3
bnez r2, LOOP

```

Problem 2: When the MIPS program below starts register \$t0 holds the address of a string, the program converts the string to upper case.
(For MIPS documentation see http://www.ece.lsu.edu/ee4720/mips32v1.pdf and http://www.ece.lsu.edu/ee4720/mips32v2.pdf. Here are the relevant differences with DLX: branches and jumps are delayed ( 1 cycle). Some branch instructions compare two registers. Register \$0 works like DLX ro.)

LOOP:
```

lbu $t1, O($t0)
addi \$t0, \$t0, 1
beq \$t1, \$0, DONE
slti \$t2, \$t1, 97 \# < 'a'
bne \$t2, \$0 LOOP
slti \$t2, \$t1, 123 \# 'z' + 1
beq \$t2, \$0, LOOP
addi \$t1, \$t1, -32
j LOOP
sb $t1,-1($t0)

```
DONE:

Convert the program to IA-64 assembly language using predicated instructions. (You're not expected to know it at this point.) IA-64 is described in the IA-64 Application Developer's Architecture Guide, available at http://www.ece.lsu.edu/ee4720/ia-64.pdf.

For this problem one can ignore alot of IA-64's features. Here is what you will need to know: IA-64 has 64 1-bit predicate registers, p 0 to p 63 , which are written by cmp (compare) and other instructions. Predicates can be specified for most instructions, including cmp itself. See 11.2.2 for a description of how to use IA-64 predicates.

To solve the problem look at the following sections: 9.3, 9.3.1, and 9.3.2 (a brief description of where to place stops); 11.2.2 (predicate description and some more information on stops); and Chapter 7 (for instruction descriptions). The following instructions will be needed: cmp (compare, look at the normal [none] and and comparison types), br (branch), load, store, and add.
- Use general-purpose registers r0-r31 and predicate registers p1-p63 in your solution. (There are 128 general-purpose registers, but those above r31 must be allocated.)
- Minimize the number of instructions per iteration assuming about half the characters are lower case.
- Use predicates to eliminate some branches.
- Make use of post-increment loads or stores.
- Pay attention to data type sizes.
- Show stops but do not show bundle boundaries.

Problem 1: Solve Problems 3 and 4 from Fall 2000 Homework 5, available via http://www.ece.lsu.edu/ee4720/2000f/hw05.pdf. Using the solutions at http://www.ece.1su.edu/ee4720/2000f/hw05_sol.pdf assign yourself a grade in the range [0, 1]. Either: indicate the grade you assigned yourself or write "Did not solve." A solution can be provided along with a grade. It will be corrected but your grade will be used. If you opt not to solve it you will receive full credit but will be hurting your ability to solve future problems.

For the following questions read Kenneth C. Yeager, "The Mips R10000 Superscalar Microprocessr," IEEE Micro, April 1996, pp. 28-40. A restricted-access copy can be found at http://www.ece.lsu.edu/ee4720/s/yeager96.pdf. Access is allowed from within the lsu.edu domain or by using the userid "ee4720" and the correct password. Though not needed for this assignment, information on the MIPS64 4 ISA (implemented by R10000) can be found in
http://www.ece.lsu.edu/ee4720/mips64v1.pdf and http://www.ece.lsu.edu/ee4720/mips64v2.pdf.
Skip over the material on the memory system (under heading "Memory Hierarchy") and the system interface. Material related to memory will be covered later in the semester.

Problem 2: The paper uses the four terms below, for each show the corresponding, or most similar, term used in class.
- Graduate
- Active List
- Tag
- Logical Register

Problem 3: For the superscalar processors described in class taken branches resulted in higher than ideal CPI; the higher the fetch/decode width (the \(n\) in \(n\)-way superscalar) the worse the problem was. Why is this problem not as severe in the R10000? (Branch prediction is not the answer.)

Problem 4: The MIPS R10000 does not have anything like a commit register map or a commit free list. (The register map and free list at the bottom of the figure from the class notes on the next page.) How were those used with exceptions in the Method-3 dynamically scheduled processor described in class? How does the R10000 deal with exceptions given their absence? Do not describe the entire exception process, just those pieces of hardware and steps needed to do what was done with the commit free list and register map.

Problem 5: Figure 5, reproduced below, shows the information that will be placed in the active list and floating-point queue for an instruction being decoded. Various field names are shown along the bottom of the figure. The instruction format fields are shown at the top. Fields fR, fS, fT are source registers (not every instruction uses three); field \(f D\) is the destination register, FLTX is the opcode, and MADD is an extension of the of the opcode field (as func is in DLX). (The figure appears to be using field values rather than names for the first and last fields. MADD is the name of an instruction, multiply-add, and FLTX may be an abbreviation for floating-point extended, though the architecture manual calls the field value COP1X. They probably should have used opcode instead of FLTX and function instead of MADD.)


Write the field names from the bottom of Figure 5 next to the corresponding fields in the figure, from the class notes, below. Though Figure 5 shows a floating-point instruction assume that integer instructions are handled the same way. Some fields have no analog in the figure below, these can be omitted; Tag is not a field that can be omitted.


Problem 1: An ISA has a character size of \(c=9\) bits (one more than most other ISA's!) and a 30 -bit address space \((A)\). An implementation has a bus width of \(w=72\) bits and has no cache. Show how \(2^{20} \times 36\) memory devices can be connected to implement the entire address space for this implementation. Show only the connections needed for loads. Show the alignment network as a box. Label inputs and outputs and be sure to specify which address bits are being used. The solution will require many memory devices so use ellipses \((\cdots)\) between the first and last of a large group of items.

Problem 2: The program below computes the sum of an array of doubles and also computes the sum of the characters in the array. The system uses a direct-mapped cache consisting of 1024 lines with a line size of 256 bits.
```

void p3(double *dstart, double *dend)
{
double dsum = 0.0;
int csum = 0;
double *d = dstart; // sizeof(double) = 8 characters
unsigned char *c = (unsigned char *) dstart; // A character is 8 bits.
unsigned char *cend = (unsigned char *) dend;
int dlength = dend - d;
int clength = cend - c;
while( d < dend ) dsum += *d++;
if( ! LAST_PART ) flush_the_cache(); // Removes all data from the cache.
while( c < cend ) csum += *c++;
}

```

When the procedure is called none of the data in the array is cached. When answering the questions below consider only memory accesses needed for the array (double or character). Assume that the number of iterations is some convenient number, except zero of course.

Note: Though they both access the same amount of data the number of iterations of the two loops are different. The first while loop is equivalent to: for (i=0; i<dlength; i++) dsum = dsum + dstart[i];
(a) What is the hit ratio for the first while loop? Assuming the cache is flushed (emptied) between the two while loops, what is the hit ratio for the second while loop?
(b) Consider a single-issue (one-way) statically scheduled system in which the pipeline stalls on a cache miss. The cache miss delay is 1000 cycles. Roughly how does the time needed to execute the two loops compare? Assume that when there's a cache hit the time needed for one iteration is the same for both loops.
(c) Consider a single-issue (one-way) dynamically scheduled system with perfect branch and branch target prediction, a non-blocking cache, and a reorder buffer that can hold sixteen iterations of the while loops. The miss delay is still 1000 cycles however assume that for cache misses there is an initiation interval of one cycle so that the data for misses at \(t=0\) and \(t=1\) will arrive at \(t=1000\) and \(t=1001\), respectively. Now how do the two loops compare?
(d) Suppose the cache is not flushed before the second while loop executes. What is the smallest value of dlength (dee, not cee) for which the hit ratio of the second loop is less than 1.0?

Problem 3: The SPARC V9 program below adds an array of integers.
(See http://www.ece.lsu.edu/ee4720/samv9.pdf for a description of SPARC V9.) Except for prefetch these instructions (or similar ones) have been covered before. The prefetch instruction is used to avoid the type of cache misses suffered by the program in the previous problem. It is like a nop in that it does not modify registers or memory, however like a load instruction, it moves data into the cache. As used below it will fetch data that will be needed ten iterations later. The data will be moved in to the cache (if not already present) but not in to a register. Ten iterations later the ldx instruction will move the data in to register \%12. Unlike loads, prefetch instructions never raise an exception. If the address is invalid or there is another problem the prefetch instruction does nothing, so there is no danger in prefetching, say, past the end of an array.

Unlike for a load that misses the cache, a statically scheduled processor would not stall on a prefetch miss. (There'd be no point in that!)
```

! Reminder: In SPARC assembler the destination register is on the right side.
LOOP:
ldx [%l1], %12 ! Load extended word (64 bits, same size as reg)
prefetch [80+%l1], 1 ! Prefetch from address 40+%l1, type 1
add %l1, 8, %l1
subcc %l4, %l1, %g0 ! %g0 = %l4 - %l1. (%g0 is zero register.) Set cc.
bpg LOOP,pt ! Branch if condition code >0, predict taken
add %13, %12, %13 ! Branch delay slot.

```
(a) In the code above the prefetch distance is ten iterations. What is the problem with the distance being too large or too small?
(b) Suppose SPARC V9 did not have a prefetch instruction. Explain how ldxa could be used as a prefetch. Show a replacement for prefetch in the program above.

The ldxa and similar instructions include an address space identifier (ASI) which specifies which address space to load or store from. The ASI can be specified with an immediate or the \%asi register. See the architecture manual. Normal loads and stores use the ASI_PRIMARY address space. ldxa lets you specify a different one. A load from a particular address in two different address spaces may load from two different memory locations or may load the same memory location in different ways. For example, an ordinary load of an address, ldx [\% 11 ], \% 12 , would load an integer using big-endian ordering. But a load to the same address using the ASI_PRIMARY_LITTLE, ldxa [\%11] ASI_PRIMARY_LITTLE, \%12 loads an integer using little-endian ordering. Table 12 in the architecture manual lists some of the address spaces.

Hint: Think about the destination register and the ASI.

\section*{32 Spring 2001}

Problem 1: Write a DLX program to reverse a C-style string, as described below. The address of the start of the string is in r1. The string consists of a sequence of characters and is terminated by a zero (NULL). The string length is not stored anywhere, it can only be determined by looking for the NULL. Put the reversed string in memory starting at the address in r2. Be sure to terminate the reversed string.
```

! r1 holds address of first character of original string.
! r2 holds address of first character of reversed string.
! Strings end with a zero (NULL) character.

```

Problem 2: The DLX program below copies a block of memory starting at address r1 to the address r3, the block is of length r2 bytes. The problem is it won't always work. Explain why not and fix the problem without unnecessarily increasing the number of loop iterations. (The program will be slower, except for special cases.) Be sure to modify the program, not a specification of what the program is supposed to do.
```

! r1 Start address of data to copy.
! r2 Number of bytes to copy.
! r3 Start address of place to copy data to.
LOOP:
slti r4, r2, \#4
bnez r4, LOOP2
lw r5, 0(r1)
sw 0(r3), r5
addi r1, r1, \#4
addi r3, r3, \#4
subi r2, r2, \#4
j LOOP
LOOP2:
beqz r2, EXIT
lb r5, 0(r1)
sb 0(r3), r5
addi r1, r1, \#1
addi r3, r3, \#1
subi r2, r2, \#1
j LOOP2

```
EXIT:

Problem 3: Implement the following procedure in DLX assembly language. The procedure is given two ways, both do the same thing, look at either one. The return address is stored in r31. The C short int data type here is two bytes (as it is on many real systems). The registers used for the procedure arguments are specified by the C variable names.
```

    void sum_arrays(short int *s_r1, float *f_r2, double *d_r3, int size_r4)
    {
while( size_r4-- ) *d_r3++ = *s_r1++ + *f_r2++;
}
void sum_arrays(short int *s_r1, float *f_r2, double *d_r3, int size_r4)
{
int i;
for(i=0; i<size_r4; i++) d_r3[i] = s_r1[i] + f_r2[i];
}

```

Problem 4: The code below contains two sets of add instructions, one in DLX assembler, the other in Compaq (née DEC) Alpha assembler. The first instruction in each group adds two integer registers, the second instruction in each group adds an integer to an immediate, the last adds two floating point registers. Information on the Alpha architecture can be found in the Alpha Architecture Handbook, http://www.ee.lsu.edu/ee4720/alphav4.pdf. It's 371 pages, don't print the whole thing.
```

! DLX Assembly Code
add r1, r2, r3 ! r1 = r2 + r3
addi r4, r5, \#6
addf f0, f1, f2
! Alpha Assembly Code (Destination is last operand.)
addq r2, r3, r1 ! r1 = r2 + r3
addq r5, \#6, r4
addt f1, f2, f0

```

Though the DLX and Alpha instructions are similar they are not identical.
- How do the data types and immediates differ between the corresponding DLX and Alpha instructions?
- Show the coding for the DLX and Alpha instructions above. Show the contents of as many fields as possible. For DLX, the addi opcode is 1 . The add func field is 0 and the addf func field is \(1 d_{16}\). For the Alpha fields, see the Alpha Architecture Manual and use the following information: The Trapping mode should be imprecise and the Rounding mode should be Normal. (Trapping [raising an exception] will be covered later in the semester.)
- How do the approaches used to specify the immediate version of an integer instruction differ?
- How is the approach used to code floating-point instructions different in Alpha than DLX?

\section*{EE 4720}

\section*{Homework 2}

\section*{Due: 21 February 2001}

Problem 1: Translate the following C program to DLX assembly, use the minimum number of comparison instructions. Pay attention to data type sizes. The line labels are provided for convenience, please use them in the assembly language version.
```

extern int r1, r2, r3, r10, r11;
extern int *r20, *r21;
/* For DLX: sizeof(int) = sizeof(int*) = 4 */
/* For IA-64: sizeof(int) = sizeof(int*) = 8 */

```
    if( r1 < 3 )
        \{
        LINE1:
            if( r2 == r3 )
            \{
            LINE11: r10 = *r20++;
            \}
            else
                \{
                LINE10: r10 = 4720;
            \}
    LINE1E:
            r11 = r11 + r10;
    \}
    else
    \{
    LINEO:
        r21 = r21 + 7;
        if ( r2 == r3)
            \{
            LINE01: r10 = *r21++;
            \}
        else
            \{
            LINE00: r10 = 7700;
            \}
    \}
DONE:

Problem 2: Translate the C program from the previous problem into IA-64 assembly using predicated instructions. (You're not expected to know it at this point.) IA-64 is described in the IA-64 Application Developer's Architecture Guide, available at http://developer.intel.com/design/ia64/downloads/adag.pdf.

For this problem one can ignore alot of IA-64's features. Here is what you will need to know: IA-64 has 64 1-bit predicate registers, p0 to p63, which are written by cmp (compare) and other instructions. Predicates can be specified for most instructions, including cmp. See 11.2.2 for a description of how to use IA-64 predicates.

To solve the problem look at the following sections: 11.2.2 (predicate description) and Chapter 7 (for instruction descriptions). The following instructions will be needed: cmp (compare, look at the normal [none] and unc comparison types), ld1, ld2,... (loads), and add.

To save time, ignore instruction stops (;;) and consider only normal loads. (Post-increment like loads are considered normal here.)
- Use general-purpose registers r0-r31 and predicate registers p1-p63 in your solution. (There are 128 general-purpose registers, but those above r31 must be allocated.)
- Do not use branches (or any other CTI).
- Ignore stops. (These will be covered later.)
- Use the minimum number of cmp instructions. (Three is possible.)
- Do not assign a value to a register unless it's needed.
- Make use of post-increment loads.
- Pay attention to data type sizes.

Problem 3: Show a pipeline execution diagram of the code below on each implementation. (There should be a total of two diagrams.) The branch is always taken, show the diagram until the second execution of the first instruction reaches WB. If a bypass path is not shown, it's not there. LOOP:
```

addi r2, r2, \#4
lw r1, O(r2)
add r3, r3, r1
slt r4, r2, r5
beqz r4, LOOP
xor r5, r4, r1

```


Problem 4: For each implementation from the problem above, determine the CPI for a large number of iterations.

Problem 5: For the second pipeline execution diagram above, show the location(s) of the latest value of \(r 1\) and \(r 2\) at the beginning of each cycle on the diagram below. For \(r 1\) box the appropriate cycle numbers and draw an arrow to the locations. For r2 circle the cycle numbers and draw an arrow to the locations. In the diagram below this has been completed for cycles zero and two, assuming addi is in IF at cycle zero. The arrows should only point to register values that are valid at the indicated cycles. Note: A valid value can be in more than one location at once.


Problem 1: Consider three variations on the Chapter-3 DLX implementation. In implementation I the FP Add unit has an initiation interval of 2 and a latency of 3 . In implementation II there are two FP Add units, each unit has an initiation interval of 4 and a latency of 3 . In implementation III the FP Add unit has an initiation interval of 1 and a latency of 3 . Other features of the implementations are identical. All implementations are fully bypassed.

Write two programs. Program \(\mathcal{A}\) should run slower on implementation I than on implementations II and III. Program \(\mathcal{B}\) should run the same speed on implementations I and II and faster on implementation III. For this problem base program speed on the time from the fetch of the first instruction to the WB of the last instruction.

Show pipeline execution diagrams for each program on each implementation. The programs need be no longer than four instructions each.
```

! Program \mathcal{A}
! I
addd f0, f2, f4 IF ID A1 A1 A2 A2 WB
addd f6, f8, f10
IF ID -> A1 A1 A2 A2 WB
! II
addd f0, f2, f4 IF ID A A A A WB
addd f6, f8, f10 IF ID B B B B WB
! III
addd f0, f2, f4 IF ID A1 A1 A2 A2 WB
addd f6, f8, f10 IF ID A1 A1 A2 A2 WB
! Program \mathcal{B}
! I
addd f0, f2, f4 IF ID A1 A1 A2 A2 WB
addd f6, f8, f10
addd f12, f14, f16
IF ID -> A1 A1 A2 A2 WB
IF -> ID -> A1 A1 A2 A2 WB
! II
addd f0, f2, f4 IF ID A A A A WB
addd f6, f8, f10
addd f12, f14, f16
IF ID B B B B WB
IF ID ----> A A A A WB
! III
addd f0, f2, f4 IF ID A1 A1 A2 A2 WB
addd f6, f8, f10 IF ID A1 A1 A2 A2 WB
addd f12, f14, f16 IF ID A1 A1 A2 A2 WB

```

Problem 2: Modify the pipeline below so that it can execute jr instructions and add PC mux control logic.
- Modify the pipeline so that it can execute jr instructions. (See Spring 1999 Homework 3, http://www.ee.lsu.edu/ee4720/1999/hw03_sol.pdf.)
- Include control logic for the multiplexor that connects to PC. The control logic should correctly handle branch and jump instructions. Interrupts should be ignored. To recognize instructions use boxes such as \(=\mathrm{bnez}\), the outputs will be 1 if the instruction matches.
- Show the logic for a squash signal for use in EX to squash the fall-through instruction on a taken branch. (The fall through instruction could have been squashed in IF or ID, but for this problem it will be squashed in EX.)


Problem 3: How would the hardware designed above have to be modified if DLX had two-slot (yes, two slots!) delayed branches? Jumps still have no delay slots. Ignore interrupts, they will be considered in the next problem.

Problem 4: In an ISA without delayed branches it would be sufficient for the hardware to save the PC when an exception occurs. Why would this not be sufficient on a system with delayed branches. Provide an example illustrating what might go wrong.

One could not properly resume execution if the faulting instruction were in a branch delay slot. To resume execution properly the exception handler needs the PC of the faulting instruction and the PC of the next instruction to execute. In most cases the next instruction to execute is at PC+4 (assuming four-character instructions) but if the faulting instruction were in the delay slot of a taken branch the next instruction to execute would be the branch target.

Suppose that lw raises an exception in the example below. If the handler only saves the address of \(1 \mathrm{w}, 0 \mathrm{x} 1004\), then when execution resumes the branch will not be taken. By saving \(0 x 1004\) and the address of the next instruction, \(0 \times 2000\), the handler can restore execution so that the branch will be taken.
```

0x1000: beqz r0, TARGET

```
```

0x1004: lw r2, 0(r3)
0x1008: add r3, r3, r4
TARGET:
0x2000: or r5, r6, r7

```

Problem 5: The Hewlett Packard Precision Architecture RISC 2.0 (PA-RISC 2.0) uses an instruction address offset queue rather than a plain-old program counter. See the PA-RISC 2.0 Architecture [Manual], http://devresource.hp.com/devresource/Docs/Refs/PA2_0/acd-1.html. Ignore the material on [address] space IDs and privilege levels. Concentrate on the material in Chapter 4 and 5 and use the index.

PA-RISC 2.0 has delayed branches. Explain how the use of an instruction address offset queue rather than a PC helps with the difficulty alluded to in the previous problem.

The address of the executing instruction and the next instruction can be saved and restored as a unit.
Problem 6: Explain the relationship between the terms interrupt, hw interrupt, exception, and trap provided in class and the terms interruption, fault, interrupt, trap, and check defined for PA-RISC 2.0. Explain the relationships, do not simply provide definitions.

Problem 7: Name a difference between the trap table used in Sun SPARC V8 (presented in class and described in the SPARC Architecture Manual V8, http://www.ee.lsu.edu/ee4720/sam.pdf) and the interruption vector table used in PA-RISC 2.0.

The Sun SPARC table holds four instructions, the PA-RISC table holds eight instructions, otherwise they are very similar.

Problem 1: Complete a pipeline execution diagram for the following code running on a two-way statically scheduled superscalar processor. Show execution until the second fetch of the first add. The processor fetches instructions in aligned groups and is fully bypassed. The branch will be taken. There is no branch prediction hardware.

What is the CPI for a large number of iterations?
```

LOOP: ! LOOP = 0x1004
add r1, r2, r3
add r4, r5, r6
add r9, r4, r7
lw r10, O(r4)
add r11, r11, r10
or r12, r11, r13
xor r15, r16, r17
bnez r10, LOOP

```

Problem 2: Schedule the code from the problem above so that it executes efficiently. The solution can contain added nop instructions. Do not try to unroll the loop. A correct solution contains two stalls plus the branch delay.

Now, what is the CPI for a large number of iterations?
Problem 3: Show the execution of the code from Problem 1 on a two-way superscalar dynamically scheduled machine using Method 1. The number of reservation stations, functional units, and reorder buffer entries is unlimited. Do not show reservation station numbers or reorder buffer entry numbers in the diagram. Do show where instructions commit. Assume that the machine has perfect branch and branch target prediction and so a branch target will be fetched when the branch is in ID. Complete the diagram to the point where all instructions in the first iteration commit, showing what happens to instructions in the second iteration up to that point.

Now, what is the CPI for a large number of iterations?

Problem 4: Convert the code below to VLIW DLX as described in the notes. The maximum lookahead value is 15 , use that for bundles that do not modify any registers. Set the lookahead values and serial bits for maximum performance. (The lookahead values will mostly be small.) How would a modification of the end-of-loop test improve performance on a VLIW implementation?
```

    j TEST
    LOOP:
lw r1, O(r10)
lw r2, 4(r10)
lw r3, 8(r10)
lw r4, 12(r10)
andi r1, r1, \#15
andi r2, r2, \#15
andi r3, r3, \#15
andi r4, r4, \#15
sw O(r10), r1
sw 4(r10), r2
sw 8(r10), r3
sw 12(r10), r4
addi r10, r10, \#16
TEST:
slt r11, r10, r12
bnez r11, LOOP

```

Problem 5: Insert the minimum number of IA-64-style stops in the DLX code below. Do not convert the instructions themselves to IA-64, just insert the stops.

The material on stops was covered in class and will be in the notes. A primary reference is Appendix A of the IA-64 Application Developer's Architecture Guide, available at http://developer.intel.com/design/ia64/downloads/adag.pdf. Appendix A describes how stops affect the execution of code.
```

j TEST
LOOP:
lw r1, O(r10)
lw r2, 4(r10)
lw r3, 8(r10)
lw r4, 12(r10)
andi r1, r1, \#15
andi r2, r2, \#15
andi r3, r3, \#15
andi r4, r4, \#15
sw 0(r10), r1
sw 4(r10), r2
sw 8(r10), r3
sw 12(r10), r4
addi r10, r10, \#16
TEST:
slt r11, r10, r12
bnez r11, LOOP

```

Problem 1: Solve Problems 3 and 4 from Fall 2000 Homework 5, available via http://www.ee.1su.edu/ee4720/2000f/hw05.pdf. Using the solutions at http://www.ee.lsu.edu/ee4720/2000f/hw05_sol.pdf assign yourself a grade in the range \([0,1]\). Either: indicate the grade you assigned yourself or write "Did not solve." A solution can be provided along with a grade. It will be corrected but your grade will be used. If you opt not to solve it you will receive full credit but will be hurting your ability to solve future problems.

Most of the problems below ask about the Alpha 21264 implementation of the Alpha Architecture. The answers to these questions can be found in Kessler 99, R. E. Kessler, "The Alpha 21264 microprocessor," IEEE Micro Magazine, March 1999, vol. 19, no. 2, pp. 24-36, available via http://www.ee.lsu.edu/ee4720/kessler99.pdf

Problem 2: Which of the dynamic scheduling methods described in class most closely matches the 21264? What terminology does Kessler 99 use for the following three terms (as used in class): Reorder Buffer, Commit, and Physical Register Number?

Problem 3: The 21264 is described as a four-way superscalar processor. What are the maximum number of instructions that can issue per cycle? What are the maximum number of instructions that can commit per cycle? How do these numbers differ from the corresponding values for the default dynamically scheduled machine as described in class? Call the higher of these two numbers \(x\). Why would DEC (okay, Compaq) dare not call the 21264 an \(x\)-way superscalar processor?

Problem 4: What is the size of the reorder buffer in the 21264? How is its use slightly different than the one described in class?

Problem 5: The stages making up the 21264 (Figure 2 in Kessler 99) differ from the stages in the dynamically scheduled DLX implementation (using the appropriate method) described in class. For each 21264 stage indicate which DLX stage does the equivalent (or close) work. Consider Slot 1 to be an extension of the fetch stage.

Problem 6: A gshare or gselect two-level predictor can perfectly predict the loop branches for loops with a small, constant number of iterations, such as:
```

    addi r1, r0, #3
    LOOP:
lw r2,0(r3)
beqz r2, SKIP
sw 4(r3), r2
SKIP:
addi r3, r3, \#8
subi r1, r1, \#1
bnez r1, LOOP

```
    Consider a processor using a gshare predictor with a 12 -bit global history register. Would
the processor predict the last branch perfectly (after warmup and assuming no collisions in the
BHT)? Hint: Yes. Modify the loop above by adding code between LOOP and SKIP so that the
gshare predictor no longer predicts the last branch correctly. The number of loop iterations must
not change.

Problem 7: Suppose the code from the previous problem, translated to Alpha, ran on the Alpha 21264. Assuming no collisions, why would the modifications made in the previous problem not remove the perfect prediction of the last branch?
(Information to solve the problem can be found on page 27 (PDF page 4) of http://www.ee.lsu.edu/ee4720/kessler99.pdf. The McFarling paper, not needed to solve this problem but referenced by Kessler, can be found via http://www.ee.lsu.edu/tca/mcf.pdf.)

\section*{33 Fall 2000}

\section*{EE 4720}

Homework 1
Due: 1 September 2000
Problem 1: Find the SPECint2000 results for the API UP2000 750 MHz processor, it can be found at the http://www.spec.org web site. This processor has a SPECint2000 rating of 456. Find another processor with a slower rating but for which individual benchmarks are faster. (Look for different CPU families.) How many of the benchmarks are faster on the slower processor?

Problem 2: Write a DLX assembly language program to convert a string of characters to lower case. The string is NULL-terminated (the character following the end of the string is a zero). Register \(r 1\) contains the address of the start of the string. Any register can be modified. The code for an upper-case A is 65 and the code for a lower-case a is 97 . Modify the string, do not create a new one.

Problem 3: Write a DLX assembly language program that loads an element of a two-dimensional array to a register.

Register r 1 holds address of the start of the array, register r 2 holds the row of the element to retrieve, and register r3 holds the column of the element to retrieve. Put the retrieved element in f0. The array dimensions are 256 rows \(\times 1024\) columns. Each element of the array is a double precision floating point number.

Elements are arranged in memory in the following order:
\[
a_{0,0} a_{0,1} a_{0,2} \cdots a_{1,0} a_{1,1} a_{1,2} \cdots a_{2,0} \cdots
\]
where \(a_{i, j}\) is the element at row \(i\), column \(j\).

\section*{EE 4720}

Homework 2
Due: 22 September 2000
Problem 1: Compare the coding of the DLX instructions:
```

add r1, r2, r3
addi r4, r5, \#6

```
to the corresponding Sun SPARC V8 instructions:
```

add %g3, %g2, %g1 ! g1 = g2 + g3
add %g5, 6, %g4 ! g4 = g5 + 6

```

The definition of the SPARC V8 architecture is available via http://www.ee.lsu.edu/ee4720/sam.pdf or http://www.sparc.com/standards/V8.pdf. Hint: The information needed to solve the problem is in Appendix B.

How are the approaches used to code immediate variants of the add instructions different in the two ISAs?

Problem 2: DLX does not have indexed addressing nor does it have autoincrement addressing. Suppose one wanted to include those addressing modes in an extended version of DLX, call it DLX-BAM (better addressing modes). The addressing modes would be used in load and store instructions. Show how they would best be coded, where the fewer changes to the coding structure the better. (For example, adding a fourth instruction type [say Type-A], would be a big change and so would be bad.) Sample mnemonics for these instructions appear below:
```

! Indexed addressing.
lw r1, (r2+r3) ! r1 = MEM[ r2 + r3 ];
sw (r2+r3), r4 ! MEM[ r2 + r3 ] + r4;
! Autoincrement addressing.
lb r1, 3(+r2) ! r1 = MEM[ r2 + 3 ]; r2 = r2 + 1;
lw r4, 8(+r5) ! r4 = MEM[r5 + 8 ]; r5 = r5 + 4;
sw 4(+r7), r8 ! MEM[r7 + 4 ] = r8; r7 = r7 + 4;

```

Problem 3: Write a C program that does the same thing as the DLX program below.
```

! r2: Start of table of indices, used to retrieve elements
! from the character table.
! r4: Start of table of characters.
! r6: Location to copy characters to.
! r8: Address of end of index table.
LOOP:
lw r1, O(r2)
add r3, r1, r4
lb r5, 0(r3)
sb 0(r6), r5
addi r2, r2, \#4
addi r6, r6, \#1
slt r7, r2, r8
bneq r7, LOOP

```

Solution template available via: http://www.ee.1su.edu/ee4720/2000f/hw02.c
void
untangle(int *r2, char *r4, char *r6, int *r8)
\{
    /* Put solution here. */
\}

Problem 4: Re-code the DLX program above using DLX-BAM, taking advantage of the new instructions.

Problem 1: What changes would have to be made to the pipeline below to add the DLX-BAM indexed addressing instructions (from homework 2). Hint: The load is easy and inexpensive, the store requires a substantial change. Add the changes to the diagram below, but omit the control logic. Do explain how the control logic would have to be changed.
```

! Indexed addressing.
lw r1, (r2+r3) ! r1 = MEM[ r2 + r3 ];
sw (r2+r3), r4 ! MEM[ r2 + r3 ] + r4;

```


Problem 2: For maximum pedagogical benefit solve the problem above before attempting this one. The integer pipeline of the Sun Microsystems microSPARC-IIep implementation of the SPARC V8 ISA is similar to the Chapter-3 implementation of DLX that is being covered in class.

What are the stage names and abbreviations used in the microSPARC-IIep? Hint: This is really easy once you've found the right page.

SPARC V8 includes indexed addressing, for example:
```

ld [%o3+%o0], %o2 ! Load word: %o2 = MEM[ %o3 + %o0 ]
st %o0, [%o1+%g1] ! Store word: MEM[ %o1 + %g1 ] = %o0

```
(Register \(\% \circ 0\) is a real register, not a special zero register.) What are the differences between the micro-SPARC-IIep integer pipeline and the Chapter-3 DLX pipeline that allow it to execute an indexed store? Be sure to answer the question directly, do not copy or paraphrase irrelevant material. A shorter answer is preferred.

Information on the microSPARC-IIep can be found via http://www.sun.com/microelectronics/manuals/microSPARC-IIep/802-7100-01.pdf
or http://www.ee.lsu.edu/ee4720/microsparc-IIep.pdf. Those who enjoy a challenge can study the diagram on page 10, however the material to answer the question can be found early in Chapter 3 . The manual uses many terms which have not yet been covered in class, the question can still be answered once the right page is found. The manual is 256 pages so don't print the whole thing.

Problem 3: The following pipeline execution diagram shows the execution of a program on the DLX implementation shown below. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others; connections needed to implement the jalr instruction are not shown. A value can be read from the register file in the same cycle it is written. Instructions are squashed (nulled) in this problem by replacing them with or \(\mathrm{rO}, \mathrm{rO}, \mathrm{rO}\). All instructions stall in the ID stage.

Add the datapath connections needed so the jalr executes as shown.
```

! Initially, r1=0x100, r2=0x200, r3=0x300, r4 = 0x68
! The lw will read Oxaaa0.
! Cycle 0
sub r0, r0, r0 WB
sub r0, r0, r0 ME WB
sub r0, r0, r0 EX ME WB
sub r0, r0, r0 ID EX ME WB
START: ! START = 0x50
add r2, r2, r3 IF ID EX ME WB
lw r2,4(r2) IF ID EX ME WB
sw 8(r2), r1 IF ID -> EX WE WB
jalr r4 IF -> ID EX ME WB
xor r4, r1, r2 IFx
subi r2, r1, \#0x10
andi r2, r2, \#0x20 IF ID EX ME WB
slti r3, r3, \#0x30 IF ID EX ME

```


The table on the next page shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. The first two columns are completed; fill in the rest of the table. Use a "?" for the value of the "immediate field" of a type R instruction and for the output of the memory when no memory read is performed. Show pipeline register values even if they're not used. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If r0 is written leave the entry blank.

Hint: For hints and confirmation see Spring 1999 HW 3, Fall 1999 HW 2, and Spring 2000 HW 2, linked to http://www.ee.lsu.edu/ee4720/prev.html, for similar problems. It's important that the problem is solved by inspection of the diagram, not by inferring mindless, unworthy-of-an-engineer rules from past solutions. Mindless rules are hard to remember and are useless in new situations.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline PC & 50 & 54 & & & & & & & & \\
\hline IF/ID.IR & sub & add & & & & & & & & \\
\hline Reg. Chng. & \(\mathrm{r} 0 \leftarrow 0\) & \(\mathrm{r} 0 \leftarrow 0\) & & & & & & & & \\
\hline ID/EX.IR & sub & sub & & & & & & & & \\
\hline ID/EX.A & 0 & 0 & & & & & & & & \\
\hline ID/EX.B & 0 & 0 & & & & & & & & \\
\hline ID/EX.IMM & ? & ? & & & & & & & & \\
\hline EX/MEM.IR & sub & sub & & & & & & & & \\
\hline EX/MEM.ALU & 0 & 0 & & & & & & & & \\
\hline EX/MEM.B & 0 & 0 & & & & & & & & \\
\hline MEM/WB. IR & sub & sub & & & & & & & & \\
\hline MEM/WB. ALU & 0 & 0 & & & & & & & & \\
\hline MEM/WB.MD & ? & ? & & & & & & & & \\
\hline
\end{tabular}

Problem 4: Draw a pipeline execution diagram showing the execution of the familiar code below until the second fetch of lw (the beginning of the second iteration). Hint: There are RAW hazards associated with the loads, stores, and the branch. What is the CPI for a large number of iterations?

LOOP:
lw r1, \(0(r 2)\)
add \(r 3\), r1, r4
lb \(\mathrm{r} 5,0(\mathrm{r} 3)\)
sb 0(r6), r5
addi r2, r2, \#4
addi r6, r6, \#1
slt r7, r2, r8
bneq r7, LOOP
xor r10, r11, r12


Problem 5: Rearrange (schedule) the instructions in the program from the previous problem to minimize the number of stalls. Now what is the CPI for a large number of iterations? Hint: The offsets in the load and store instructions can be changed, even to negative numbers.

Homework 4
Due: 3 November 2000
Problem 1: Show a pipeline execution diagram for the execution of the DLX program below on a single-issue statically scheduled (plain old chapter 3) fully bypassed implementation in which the add functional unit is two stages (A1, A2) with an initiation interval of 2 (latency 3) and the multiply unit is six stages (M1 through M6) with an initiation interval of 1 (latency 5). (This problem is very similar to Spring 2000 homework 3 problem 1. Check the solution to that assignment only if completely lost.)
```

addd f0, f2, f4
addd f6, f0, f8
addd f10, f12, f14
multd f16, f18, f20

```

Problem 2: Show a pipeline execution diagram for the execution of the DLX program below on a single-issue statically scheduled fully bypassed implementation in which there are two add units, both consisting of one stage with an initiation interval of 4 (latency 3 , unpipelined). Use symbol A for one adder and B for the other. The program below is slightly different than the one above.
```

addd f0, f2, f4
addd f6, f0, f8
addd f10, f12, f14
addd f16, f18, f20

```

Problem 3: Show a pipeline execution diagram for the execution of the DLX program below on a two-way superscalar statically scheduled fully bypassed implementation in which there are two add units, both consisting of one stage with an initiation interval of 4 (latency 3, unpipelined). Use symbol \(A\) for one adder and \(B\) for the other.
```

LINE1: ! LINE1 = 0x1000
addd f0, f2, f4
addd f6, f0, f8
addd f10, f12, f14
addd f16, f18, f20

```

Problem 4: Show a pipeline execution diagram for the DLX code below executing on a processor with the following characteristics:
- Statically scheduled two-way superscalar.
- Unlimited number of functional units.
- Six stage fully pipelined multiply.
- Can handle an unlimited number of write backs per cycle. (Unrealistic, but reduces adidactic tedium.)
- Fully bypassed, including the branch condition.

The diagram should start at the first iteration and end after 30 cycles or until a repeating pattern is encountered, whichever is sooner. Note that there is a floating-point loop-carried dependency (f2). What is the CPI for a large number of iterations?
```

LOOP: ! LOOP = 0x1004
ld f0, O(r1)
muld f2, f0, f2
addi r1, r1, \#8
sub r2, r1, r3
bneq r2, LOOP
xor r10, r11, r12
and r13, r14, r15
or r16, r17, r18
sgt r19, r20, r21

```

Problem 5: Unroll and schedule the loop from the problem above for maximum efficiency. Unroll the loop four times; the number of iterations will always be a multiple of four. Use software pipelining and take advantage of associativity to overlap the multiply latency. (In software pipelining a computation is spread over several iterations.) Code may be added before the LOOP label.

Problem 1: The familiar loop below executes on a dynamically scheduled machine using a reorder buffer to name destination registers. The machine has the following characteristics:
- Two-way superscalar. An unlimited number of write-backs per cycle.
- A 16 -entry reorder buffer.
- A six-stage fully pipelined floating point multiply unit.
- Perfect branch target prediction. (Branch target in IF when branch is in ID.)

Show a pipeline execution diagram up to the fetch of the third iteration.
Explain why the first two iterations cannot be used to determine the CPI for a large number of iterations in this case. Estimate the CPI for a large number of iterations (a pipeline execution diagram is not necessary).
```

LOOP: ! LOOP = 0x1000
ld f0, O(r1)
muld f2, f0, f2
addi r1, r1, \#8
sub r2, r1, r3
bneq r2, LOOP
xor r10, r11, r12
and r13, r14, r15
or r16, r17, r18
sgt r19, r20, r21

```

Problem 2: Unroll the loop in the problem above twice. (In the last homework it was unrolled four times.) Again exploiting the associativity of multiplication, rearrange the multiplies to improve the performance, but this time without using software pipelining. Why is software pipelining not necessary here?

Problem 3: The code below executes on a system using a one-level branch predictor with a 16 -entry BHT. Which entries will the branches use?

If the number of iterations is large, the prediction accuracy will be high. If a certain number of additional nops are inserted before SKIP1 the prediction accuracy will drop. How many and why?
```

    ! Note: r2 is not modified inside the loop.
    LOOP: ! LOOP = 0x1000
subi r1, r1, \#1
bneq r2, SKIP1
add r10, r10, r11
nop
SKIP1:
beqz r2, SKIP2
add r12, r12, r13
SKIP2
bneq r1, LOOP

```

Problem 4: Determine the prediction accuracy of a one-level branch predictor on each branch in the code below. The predictor uses a 1024 -entry BHT. There is a .5 probability that a loaded value will be zero.
```

LOOP:
addi r2, r2, \#4
lw r1, O(r2)
bneq r1, SKIP1
add r10, r10, r11
SKIP1:
andi r3, r2, \#4
bneq r3, SKIP2
add r11, r11, r12
SKIP2:
beqz r1, SKIP3
add r12, r12, r11
SKIP3:
andi r4, r2, \#12
bneq r4, SKIP4
add r13, r13, r11
SKIP4:
sub r5, r2, r6
bneq r5, LOOP

```

Problem 5: How many BHT entries will the branches in the code above use in the middle of its execution (explained below) in a two-level gselect predictor that uses 10 bits of global branch history and 6 instruction address bits? The loop iterates many times, the middle of its execution starts after many iterations.

How many bits of global branch history are needed so that the branch following SKIP3 is predicted very accurately?

\section*{34 Spring 2000}

Problem 1: Using the SPARC Architecture Manual (SAM) V8 answer the questions below. The SPARC Architecture Manual is distributed with the source for the microSPARC IIep in directory .../models/sparc_v8/docs/pdf of the distribution which can be downloaded from http://www.sun.com/microelectronics/communitysource/sparcv8/.
Alternate instructions will be given in class.
The SAM is 295 pages, so don't print it all out. It is not necessary that you understand everything in the SAM to answer these questions. See Appendix B to answer the last question.
- What size integers does the ISA support?
- What size floating-point numbers does the ISA support?
- How many floating-point registers does the ISA support, how large are they, and how are the different-sized FP numbers placed in them?
- What is the binary coding of the following SPARC v8 instruction:
```

ldsh [%r8 + 2], %r9 ! Load signed half, r9 = Mem[r8 + 2]

```

Problem 2: Find the static and dynamic instruction count for the DLX program below. (DLX is described in Chapter 2 of the text and summarized in the last two pages. Comments, preceded by a !, describe what the instructions do.) The program adds up a table of numbers.
```

lhi r2, \#0x1234 ! Load high: r2 = 0x12340000
ori r2, r2, \#0x5678 ! r2 = r2 0x5678
addi r4, r0, \#10 ! r4 = r0 + 10, r0 always = 0
sub r3, r3, r3 ! r3 = 0. There are lots of ways to do this!
LOOP:
lw r1, 0(r2) ! r1 = Mem[r2+0]
add r3, r3, r1 ! r3 = r3 + r1
addi r2, r2, \#4 ! r2 = r2 + 4
subi r4, r4, \#1 ! r4 = r4 - 1
bneq r4, LOOP ! if r4 != 0 goto LOOP

```

Problem 3: DLX does not allow arithmetic instructions to access memory. Suppose they could and suppose all the addressing modes in Figure 2.5 of the text were available. Re-write the program to use as few instructions as possible (but still perform the same function).

Problem 4: Find the static and dynamic instruction count of the program written for the question above.

Problem 5: What factors (relating to CPI and \(\phi\) ) would one have to take into account to compare the execution time using the dynamic instruction count of the original program and the re-written program?

Problem 1: The program below executes on the DLX implementation shown below. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others. All forwarding paths are shown. (If a needed forwarding path is not there, sorry, you'll have to stall.) A value can be read from the register file in the same cycle it is written. The destination field in the beqz is zero. Instructions are nulled (squashed) in this problem by replacing with slt r0, r0, r0. All instructions stall in the ID stage.


The table below shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. Cycle zero is the time that addi is in instruction fetch. The first two columns are completed; fill in the rest of the table. Use a "?" for the value of the "immediate field" of a type R instruction and for the output of the memory when no memory read is performed. Show pipeline register values even if they're not used. Assume that the ALU performs the branch target computation even though it was already computed in ID. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If r 0 is written leave the entry blank.

Hint: See Spring 1999 HW 3 and Fall 1999 HW 2 for similar problems.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline PC & 0x50 & 0x54 & & & & & & & & & \\
\hline IF/ID.IR & sub & addi & & & & & & & & & \\
\hline Reg. Chng. & r0 \(\leftarrow 0\) & \(\mathrm{r} 0 \leftarrow 0\) & & & & & & & & & \\
\hline ID/EX.IR & sub & sub & & & & & & & & & \\
\hline ID/EX.A & 0 & 0 & & & & & & & & & \\
\hline ID/EX.B & 0 & 0 & & & & & & & & & \\
\hline ID/EX.IMM & ? & ? & & & & & & & & & \\
\hline EX/MEM.IR & sub & sub & & & & & & & & & \\
\hline EX/MEM.ALU & 0 & 0 & & & & & & & & & \\
\hline EX/MEM.B & 0 & 0 & & & & & & & & & \\
\hline MEM/WB.IR & sub & sub & & & & & & & & & \\
\hline MEM/WB. ALU & 0 & 0 & & & & & & & & & \\
\hline MEM/WB.MD & ? & ? & & & & & & & & & \\
\hline
\end{tabular}

Problem 2: The execution of the code in the problem above should suffer a stall (not including the branch delay). Add bypass path(s) to the diagram below needed to avoid the stall(s). Add only the bypass paths needed to avoid the stalls encountered in the problem above, and no others. (The diagram below is the same as the one in the first problem.)

START:
addi r1, r1, \#8
lh r2, 2(r1)
sw 4(r1), r2
bneq r2, START
sub r2, r3, r1


Problem 1: Show a pipeline execution diagram for the following DLX code fragment on a statically scheduled implementation in which the add functional unit has a latency of 3 (four stages) and an initiation interval of 2 (not the usual 1 ) and the multiply unit has a latency of 5 (six stages) and has an initiation interval of 3 (not the usual 1).
```

addf f0, f1, f2
addf f3, f0, f4
addf f5, f0, f7
gtf f0, f8
multf f9, f0, f10

```

Problem 2: Show a pipeline execution diagram for the following DLX code fragment on a statically scheduled implementation in which the add functional unit has a latency of 3 (four stages) and an initiation interval of 1 (the usual 1) and the multiply unit has a latency of 5 (six stages) and has an initiation interval of 1 (the usual 1 ).

The implementation uses ID-stage branch target calculation. As is true for the pipelines used in class, the branch condition is not bypassed.

Instructions stall in ID to avoid structural hazards.
There are bypass paths from the WB stage to the inputs of the floating-point functional units.
(a) What is the CPI for a large number of iterations of the loop?
(b) If the multiply functional unit latency were long enough the second iteration would take longer than the first iteration. (An iteration starts when the first instruction is in IF.) What is the smallest such latency?
```

LOOP:
multd f0, f0, f2
ld f4, 0(r1)
addd f2, f2, f4
addi r1, r1, \#8
sub r2, r1, r3
bneq r2, LOOP
xor r10,r11,r12

```

Problem 3: Schedule - but don't unroll-the code from the problem above to avoid as many stalls as possible. Show a pipeline execution diagram of the scheduled code. Hint: you can change the offset of the load double instruction.

Problem 4: Unroll the loop below so that two iterations of the original loop form one unrolled loop. Schedule the code so that it executes as efficiently as possible. Assume there will be an even number of iterations and that every register not used in the original code is available and so can be used in the unrolled loop. The loop runs on the implementation described in the second problem.

LOOP:
ld fo, \(0(\mathrm{r} 1)\)
multd f0, f0, f2
addd f0, f0, f4
sd 8(r1), f0
addi r1, r1, \#16
sub r2, r1, r3
bneq r2, LOOP

\section*{EE 4720}

Homework 4
Due: 17 April 2000
Problem 1: The diagram below shows the execution of code on a dynamically scheduled machine that uses physical register numbers to name destination operands. Show the state of the ID register map, the commit register map, their free lists, and the physical register file for each cycle of the execution below. In the register maps and file show only values related to registers \(f 0\) and \(f 3\). Initially, \(\mathrm{f} 0=0, \mathrm{f} 1=10, \mathrm{f} 2=20\), etc. Initially, register f 0 is assigned to physical register 12 and f 3 is assigned to physical register 15 (ignore the other architected registers). Initially, both free lists contain physical register numbers \(\{7,8,9,10,11\}\).

Note: As originally assigned the initial free lists did not contain register 11 and the pipeline execution diagram showed reservation station ( \(R S\) ) segments. Both were mistakes and have been corrected.
```

! Cycle 0
multf f0, f1, f2 IF ID Q M0 M1 M2 M3 M4 M5 WC
addf f3, f0, f2 IF ID Q A0 A1 WC
subf f0, f4, f5 IF ID Q AO A1 WB C
addf f3, f0, f5 IF ID Q A0 A1 WB C
addf f0, f2, f1 IF ID Q AO A1 WB C

```

Problem 2: Repeat the problem above assuming that there is an exception in stage A1 of the execution of addf \(f 3, f 0, f 5\), as shown below: The solution can start at the cycle in which the tables will differ from the solution above.
```

! Cycle
0}1
multf f0, f1, f2 IF ID Q M0 M1 M2 M3 M4 M5 WC
addf f3, f0, f2
subf f0, f4, f5
addf f3, f0, f5
addf f0, f2, f1
IF ID Q A0 A1 WC
IF ID Q AO A1 WB C
IF ID Q A0*A1*WB Cx
IF ID Q AO A1 WB

```

Problem 3: The diagram below, of a dynamically scheduled processor, omits hardware that checks whether the register map should be updated in the WB stage. (The hardware was described in class.) Add the hardware to the diagram (at the same level of detail as other parts of the diagram).


Problem 4: Draw a pipeline execution diagram for the DLX code below running on a dynamically scheduled 4 -way superscalar implementation with the following characteristics:
- Dynamically scheduled using a reorder buffer to name registers (method 1).
- One load/store functional unit with stages L1 and L2.
- No dynamic (hardware) branch prediction, all branches are predicted not taken. Branch predictor uses the B functional unit and must wait for its operand like any other instruction.
- Four integer execution units.

Find the IPC for an execution of a large number of iterations. Show the execution for 14 cycles or until there is enough information to compute the IPC, which ever is shorter.
```

! Note: runs for many iterations.
add r3, r0, r0
LOOP:! LOOP = 0x1000
lw r1, 4(r2)
add r3, r3, r1
lw r2, 8(r2)
bneq r2, LOOP
xor r0, r0, r0

```

Problem 5: Repeat the problem above when the branch is statically predicted as taken and the branch target is computed in the ID stage.

Problem 6: Repeat the superscalar problem when the branch is statically predicted taken and in which the address of LOOP it \(0 \times 1004\).
```

! Note: runs for many iterations.
add r3, r0, r0
LOOP:! LOOP = 0x1004
lw r1, 4(r2)
add r3, r3, r1
lw r2, 8(r2)
bneq r2, LOOP
xor r0, r0, r0

```

Problem 1: The code below is run on three machines each using a slightly different one-level branch predictor. Each machine's branch predictor uses a 1024-entry BHT. The first machine uses 2-bit saturating counters (as described in class), the second machine uses the 2-bit prediction scheme illustrated in Figure 4.13 of the text, and the third uses a 3-bit saturating counter. (The scheme illustrated in Figure 4.13 uses two bits, but it's not a saturating counter.) Find the prediction accuracy for each scheme on each branch instruction for a large number of iterations.
```

! r1 is initially set to a large value.
LOOP1:
subi r1, r1, \#1
beqz r1, EXIT
andi r2, r1, \#6
bneq r2, SKIP1
add r3, r3, \#1
SKIP1:
andi r2, r1, \#2
bneq r2, SKIP2
add r3, r3, \#1
SKIP2:
j LOOP1
EXIT:

```

Problem 2: What is the largest BHT size (number of entries) for which there will be collisions between at least two branches in the code above?

Problem 3: The program below runs on a system using a gselect branch predictor with a 14 -bit branch history and a \(2^{22}\)-entry BHT.

Show the value of the global branch history just before executing each branch after a large number of iterations. (The branch can be taken or not taken.) Also show the address used to index (lookup the value in) the BHT.

Determine the prediction accuracy of each branch assuming no collisions in the BHT.
```

    ! r2 is initially set to a large value.
    LOOP1: ! LOOP1 = 0x1000
addi r1, r1, \#2
LOOP2: ! LOOP2 = 0x1080
subi r1, r1, \#1
bneq r1, LOOP2
A: ... ! Nonbranch instructions.
addi r1, r1, \#3
LOOP3: LOOP3 = 0x1100
subi r1, r1, \#1
bneq r1, LOOP3
B: ... ! Nonbranch instructions.
subi r2, r2, \#1
LINE: ! LINE = 0x1180
bneq r2, LOOP1

```

Problem 4: Suppose the problem above ran on a gshare branch predictor with a 10 -bit branch history and a \(2^{10}\)-entry BHT. Determine addresses for LOOP1, LOOP2, LOOP3, and LINE for which there would be collisions in the BHT after a large number of iterations. (Try to retain program order.)

If you only have time for one of these problems, do problem three (the one on connecting memory devices to implement a cache). If you have or are hoping to get a job interview with a company that makes processors or which requires strong familiarity with them, do problems one and two. If you want to practice Verilog or VHDL do problem four. It you had the time to read this far you have no excuse for not doing at least one of the problems.

Base answers to the problems below on information in the white paper at http://www.cpus.hp.com/techreports/PA-8700wp.pdf. This is a very concise technical summary of the distinctive features of the Hewlett Packard PA-8700 processor, which implements the Precision Architecture (PA). It was written to sell processors, so though technically informative certain adjectives must be taken in context. Familiarity with the PA ISA is not needed to solve this assignment. If nevertheless you're curious, there is a brief description of the PA ISA (along with others) in Appendix \(C\) of the text and a complete description can be found at http://www.hp.com/ahp/framed/technology/micropro/architecture/docs/instarch.html.

Problem 1: Describe how dynamic scheduling on the Hewlett Packard PA-8700 is similar to and different from dynamic scheduling methods 1 , 2, and 3 presented in class. The description should include handling of register values and how exception recovery might be performed. The description of dynamic scheduling in the white paper is very brief, get what you can from the description and the figure and make reasonable guesses at the rest. There is additional information on dynamic scheduling at the top of page 12 .

Problem 2: How does the dynamic branch prediction performed by the PA-8700 differ from the one-level (2-bit saturating counter) scheme presented in class? How might a compiler (that knows the microarchitecture of the target) use this difference to eliminate the performance penalty of some collisions? Would this work for all collisions?

The problems below are not based on the white paper.
Problem 3: Show how memory devices using 12-bit addresses can be connected to implement a \(2^{17}\)-byte two-way set-associative cache. The memory devices can store \(2^{12}\) items of \(x\) bits each. Each memory can have its own width ( \(x\) ), (but of course, all the items in a particular memory are the same width). The system has an address space of \(a=32\) bits, a bus width of \(w=8\) bytes, and a block size of \(L=2^{l}=2^{4}=16\) bytes. The character size is one byte. Show which address bits are used to index the memories and which bits are used to determine a hit. Show only the parts used to read data on a cache hit. (This problem would be easier if memory devices with any address size could be used. If your stuck, try that first.)

For students in 4702 or others who know Verilog, VHDL, or some other HDL.
Problem 4: Write a Verilog or VHDL description of a module that will determine which is the least-recently used block in a four-way set-associative cache cache set. This is to be based on information kept in a special memory (like a tag store, but there's only one of them for all the ways). The module will read this information at its inputs. It will generate an updated version of this information at its output for storage. It will also generate an output indicating which set is least recently used. There will be a clock input, inputs to the module must be read on the positive edge. You may specify other inputs that the module might need. (For example, the memory operation.)

\section*{35 Fall 1999}

Problem 1: What are the static and dynamic instruction counts of the two DLX programs below? (DLX is described in Chapter 2 of the text and summarized in the last two pages. Comments, preceded by a !, describe what the instructions do.) Be sure to use the value for r2 specified in the comments. Both programs find the population (number of 1's) in the binary representation of the value in r2. (For example, the population of \(12_{10}=1100_{2}\) is \(2,7_{10}=0111_{2}\) is 3 , and \(d 06 f 00 d_{16}=218558477_{10}=1101000001101111000000001101_{2}\) is 12.)
```

! Program 1.
! r2 = 0xd06f00d
add r1, r0, r0 ! r1 = 0. Initialize total.
LOOP:
andi r3, r2, \#1 ! r3 = r2 \& 0x1. Put least-significant bit in r3.
add r1, r1, r3 ! r1 = r1 + r3. Add to total.
srli r2, r2, \#1 ! r2 = r2 >> 1. Shift right logical. Shift off LSB.
bneq r2, LOOP ! Branch if r2 not zero. Loop if more.

```
```

! Program 2.

```
! Program 2.
! r2 = 0xd06f00d
! r2 = 0xd06f00d
! r4 = Base of table. Entry i is number of 1's in binary i.
! r4 = Base of table. Entry i is number of 1's in binary i.
add r1, r0, r0 ! r1 = 0. Initialize total.
add r1, r0, r0 ! r1 = 0. Initialize total.
LOOP:
LOOP:
andi r3, r2, # 0xff ! r3 = r2 & 0xff. Put 8 least significant bits in r3.
andi r3, r2, # 0xff ! r3 = r2 & 0xff. Put 8 least significant bits in r3.
add r5, r4, r3 ! r5 = r4 + r3. Add to base of population table.
add r5, r4, r3 ! r5 = r4 + r3. Add to base of population table.
lbu r6, 0(r5) ! r6 = Mem[0+r5] Load byte unsigned, Load population of r3
lbu r6, 0(r5) ! r6 = Mem[0+r5] Load byte unsigned, Load population of r3
add r1, r1, r6 ! r1 = r1 + r6. Add to the total.
add r1, r1, r6 ! r1 = r1 + r6. Add to the total.
srli r2, r2, #8 ! r2 = r2 >> 8. Shift right logical. Shift off 8 bits.
srli r2, r2, #8 ! r2 = r2 >> 8. Shift right logical. Shift off 8 bits.
bneq r2, LOOP ! Loop if r2 not zero.
```

bneq r2, LOOP ! Loop if r2 not zero.

```

Problem 2: Suppose the programs above are run on machines that execute one instruction at a time without overlap (unlike most of the examples shown in class) and with no gaps between. Suppose the CPI for all instructions is 1 cycle and the clock frequency is 625 MHz (period is about 1.6 ns ). How long would it take each program to run? Suppose the CPI for the lbu instruction was 3 cycles. How long would program 2 take?

Problem 3: What changes would have to be made to program 2 if the lbu instruction (load byte unsigned) were changed to lhu (load half unsigned)?

Problem 4: The Easy ISA as described in class has only five instructions with no straightforward way of adding new ones. A non-straightforward way of adding instructions is to take advantage of the fact that the coding does not use all possible combination of bits. In particular, it is possible to specify an immediate as the destination of an arithmetic instruction even though the ISA has no corresponding instruction. For example, consider:
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline add & Imm. & 3 & & Reg. & r1 & & Imm & & 12 & \\
\hline 000 & & 01 | & 3 & & 00 । & 1 & & 01 & & 0xc \\
\hline \(0 \quad 2\) & 3 & 45 & 24 & 25 & 2627 & 33 & 34 & & & 55 \\
\hline
\end{tabular}

This could be interpreted as instruction add 3 , r1, 12, however there is no such instruction in the Easy ISA. (If there was, what would it do?)

Explain how this "hole" can be used to code additional instructions. Use this coding to add and, or, sll (shift left logical), and sra (shift right logical) instructions. The new instructions should use the same addressing modes as the existing arithmetic instructions.

Problem 5: Recall that an issue (it's not okay to say problem anymore) with the Easy ISA is that there is no CTI (control-transfer instruction: branch, jump, call, return, etc.) that will branch to an address held in a register. Only self-modifying code can do that. Write such code. The code should branch to an address held in register r100. The solution may use the instructions added above. Addresses in Easy ISA do not have to be aligned. Assume the most significant bit of the address is always zero. Hint: This assumption and the lack of alignment restrictions makes things alot easier.

\section*{EE 4720}

Homework 2
Due: \(t\),
where \(t= \begin{cases}4 \text { October 1999, } & \text { if } 29 \text { September class held; } \\ 6 \text { October 1999, } & \text { if } 29 \text { September class cancelled; } \\ 8 \text { October 1999, } & \text { if } 29 \text { September and } 1 \text { October class cancelled } .\end{cases}\)

Problem 1: Suppose the coding of DLX instructions were changed so the destination appeared before the source operands, as shown in the codings below:

New Type R:


New Type I:
Opcode rd rs1 Immediate
\begin{tabular}{|ll|l|ll|l|}
\hline & & & & \\
\hline 0 & 5 & 6 & 10 & 11 & 15 \\
\hline
\end{tabular}

Type J: (no change)
Opcode Offset
\begin{tabular}{|l|l|}
\hline & \\
\hline 0 & 56
\end{tabular} 31

Show the changes needed to the pipeline below to implement this new ISA. The changes should only effect the ID and WB stages. If there are differences in the control inputs to multiplexors or other units, explain what those differences are. Make sure that your design executes store instructions correctly.


Problem 2: The program below executes on the DLX implementation shown below. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others. All forwarding paths are shown. (If a needed forwarding path is not there, sorry, you'll have to stall.) A value can be read from the register file in the same cycle it is written. The destination field in the beqz is zero. Instructions are nulled (squashed) in this problem by replacing them with slt r0, r0, r0. All instructions stall in the ID stage.
```

! Initially, r1=0x101, r2=0x202, r3=0x303
! MEM[0x103] = Oxfe
sub r0, r0, r0
sub r0, r0, r0
sub r0, r0, r0
sub r0, r0, r0
sub r0, r0, r0
START: ! START = 0x50
lb r1, 2(r1)
addi r1, r1, \#3
or r1, r1, r2
beqz r2, SKIP !(taken)
add r3, r1, r2
sub r0, r0, r0
sub r0, r0, r0
SKIP:
xor r3, r1, r3
sub r0, r0, r0
sub r0, r0, r0
sub r0, r0, r0

```


The table below shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. Cycle zero is the time that lb is in instruction fetch. The first two columns are completed; fill in the rest of the table. Use a "?" for the value of the "immediate field" of a type R instruction and for the output of the memory when no memory read is performed. Show pipeline register values even if they're not used. Assume that the ALU performs the branch target computation even though it was already computed in ID. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If no register value is written leave the entry blank.

Hints: See Spring 1999 HW 3 for a similar problem. One feature of the solution would not be present if lb were replaced by \(a\) addi. Another feature may not be present if lb were replaced by lw.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline PC & 0x50 & 0×54 & & & & & & & & & \\
\hline IF/ID.IR & sub & ID & & & & & & & & & \\
\hline Reg. Chng. & \(\mathrm{r} 0 \leftarrow 0\) & \(\mathrm{r} 0 \leftarrow 0\) & & & & & & & & & \\
\hline ID/EX.IR & sub & sub & & & & & & & & & \\
\hline ID/EX.A & 0 & 0 & & & & & & & & & \\
\hline ID/EX.B & 0 & 0 & & & & & & & & & \\
\hline ID/EX.IMM & ? & ? & & & & & & & & & \\
\hline EX/MEM.IR & sub & sub & & & & & & & & & \\
\hline EX/MEM. ALU & 0 & 0 & & & & & & & & & \\
\hline EX/MEM.B & 0 & 0 & & & & & & & & & \\
\hline MEM/WB. IR & sub & sub & & & & & & & & & \\
\hline MEM/WB . ALU & 0 & 0 & & & & & & & & & \\
\hline MEM/WB. MD & ? & ? & & & & & & & & & \\
\hline
\end{tabular}

Problem 3: Consider the program:
```

LOOP:
lw r1, 0(r2)
add r3, r1, r3
addi r2, r2, \#4
bneq r1, LOOP
or r4, r5, r6

```

For each implementation below provide a pipeline execution diagram showing execution up to the third fetch of 1 w and determine the CPI for a large number of iterations.

Not Bypassed:


Bypassed:


Problem 4: Schedule (rearrange) the instructions in the program used in the previous problem to improve execution speed. (Do not change what the program does!). Show pipeline execution diagrams and determine CPI for the two implementations.

Problem 5: Show the changes needed to implement the predicated instructions presented in class. (Set 4, page 25, as of this writing.) Describe the instruction format and show any datapath and control changes to the implementation below.


Problem 1: Consider the following method of implementing precise exceptions in DLX. An Exception Handler Address (EHA) register holds the address of the exception handler and an Exception Return Address (ERA) register holds the address of the faulting instruction. A new instruction (not in book) set.eha \(\langle\mathbf{r s} \mathbf{1}\rangle\) places the contents of register \(\langle\mathbf{r s} \mathbf{1}\rangle\) in EHA. After an exception occurs the address of the faulting instruction should be put in ERA and control should jump to the address stored in EHA. When an rfe (return from exception) instruction is executed control should jump back to the address stored in ERA.

Each stage has a squash signal that effectively replaces any instruction present with a nop. (See the illustration below.) Each stage also has an EXC signal which, in the middle of the cycle, is true if an exception is discovered in that stage. EXC will not be asserted if the stage contains an already squashed instruction. Registers EHA and ERA will be written with data at their in inputs if en is asserted using the same master /slave timing as the other registers and latches.

The diagram below shows a DLX implementation with the new squash signals (IF.SQ, etc.), exception signals (in every stage except WB), and the two new registers. The hardware shown can implement set.eha but does not implement exceptions or rfe. Add the hardware needed to do these. In particular:
- After an exception occurs control should jump to the address in EHA.
- Exceptions must be precise and handled in program order.
- rfe must return control to the faulting instruction.
- If the multiplexor in IF needs additional inputs, use the Taken signal to create the new multiplexor control signal. Taken is asserted only when the ID-stage adder produces the target address.
- Do not implement instructions that transfer ERA to and from an integer register.
- Assume that exception handlers will never encounter exceptions. (They do in real life, so the handler would need a way to save registers before any exceptions occur.)
- Do not test or set processor status bits for privileged state.


Continued on next page.

Based on your design, show a pipeline execution diagram for the code below in which the 1 w instruction raises a page fault exception in MEM and ant raises an illegal instruction exception in ID. Show the execution through the first two lines of the handler. Also show execution of the return from the handler and the second call of the handler for the ant instruction.
```

lhi r20, hi(HANDLER) ! Put high 16 bits in r20.
or r20, r20, lo(HANDLER) ! Put low 16 bits in r20
set.eha r20 ! In real systems only OS can use this instruction.
add r1, r2, r3
lw r4, O(r5)
ant r6, r7, r8
sub r9, r10, r11
and r12, r13, r15
or r15, r16, r17
HANDLER:
sw 1000(r0), r1
SW 1004(r0), r2
..
! Return address still in ERA.
lw r1, 1000(r0)
rfe ! Handler returns here, code below shouldn't execute.
LINEX:
add r1, r2, r3
sub r4, r5, r6
xor r7, r8, r9

```

In all the problems below all register values are available when the code starts executing. The datapath is fully pipelined so execution of floating point operations can start in the cycle after results are produced, just as the integer instructions do. Unless they are provided, use the following latency and initiation intervals: add unit: latency 3, initiation interval 1; multiply unit: latency 5, initiation interval 1; divide unit: latency 19, initiation interval 20.

Problem 2: Show a pipeline execution diagram for the code below. The branch is not taken.
```

multd f0, f2, f4
beqz r1, SKIP ! Not taken.
multd f0, f2, f6
multd f0, f0, f8
add r1, r1, r2

```

Problem 3: Show a pipeline execution diagram for the code below. The add functional unit has a latency of 3 and an initiation interval of 2. Hint: This problem tests knowledge of initiation intervals, use of functional units by different instructions, and usage of registers by single- and double-precision instructions.
LOOP:
gtd f12, f14
addd f0, f2, f4
addd f6, f8, f10
addf f16, f7, f18
Problem 4: Show a pipeline execution diagram for the code below starting from the first iteration until the CPI for a large number of iterations can be determined. What is that CPI?

The branch condition is bypassed to the ID stage so the branch does not have to stall for r1. (See 1998 HW 3.)
LOOP:
```

subi r1, r1, \#1

```
multd f0, f0, f2
bneq r1, LOOP
and \(r 2, r 3, r 4\)
\(\square\)

\section*{\(36 \quad\) Spring 1999}

The code fragment below, in C source and assembler forms, is referred to in the problems below.
```

for(i=0; i<1000; i++) if( s[i].type == 0 )
suma += s[i].score; else sumb+=s[i].score;
! r3 initialized to address of first element.
add r1, r0, r0 ! i=0
LOOP:
slti r2, r1, \#1000 ! r2 = 1 if r1 < 1000, otherwise r2 = 0.
beqz r2, DONE
lw r4, 0(r3)
ld f0, 16(r3)
bneq r4, SUMB ! Taken half the time.
addd f2, f2, f0
j NEXT
SUMB:
addd f4, f4, f0
NEXT:
addi r3, r3, \#64 ! Size of element is 64 bytes.
addi r1, r1, \#1 ! Increment loop index.
j LOOP
DONE:

```

Problem 1: Determine the static and dynamic instruction count for the DLX program above. The branch that tests r 4 will be taken half the time.

Problem 2: Suppose the program runs for 1 millisecond on a system with a 10 MHz clock. Assuming no cache misses (an assumption that will be made for most of these problems), what is the average CPI?

Problem 3: Divide the instructions into two classes: floating-point and others. (The floatingpoint instructions include the addd and ld instructions.) Suppose on implementation \(A\) the CPI of floating-point instructions, \(\mathrm{CPI}_{\mathrm{fp}}\), is twice the CPI of the other instructions, \(\mathrm{CPI}_{\mathrm{other}}\). If implementation \(A\) uses a 10 MHz clock and runs the program in 1 millisecond (like the previous problem), what would the CPIs be? Implementation \(B\) is the same as implementation \(A\) except floating-point instructions have an average CPI that is 3 times the integer instructions. Estimate how long it will take to run the program on implementation \(B\) using a 10 MHz clock.

Problem 4: Suppose that an implementation executed instructions one after another with no overlapping and no gaps between instructions. If each instruction took five cycles to execute and the clock frequency was 10 MHz , how long would program execution take?

Problem 5: Suppose, somehow, a load double instruction using scaled addressing were added to DLX. The assembler syntax is similar to the one in table 2.5 of the text, except a displacement is included at the end. For example, the execution of \(1 \mathrm{~d} f 0,10(\mathrm{r} 20)[\mathrm{r} 30] 40\) will load \(\mathrm{f0}\) (and f1) with the contents of memory at address \(10+r 20+r 30 * 40\). Rewrite the program above using the new instruction.

The SPARC assembly language program below is used in the problems that follow. SPARC register names are \(\% \mathrm{~g} 0-\% \mathrm{~g} 7, \% \mathrm{i} 0-\% \mathrm{i} 7, \% 10-\% 17\), and \(\% 00-\% \mathrm{o}\); and \(\% \mathrm{~g} 0\) is a zero register (like r0 in DLX). The destination for arithmetic, logical, and load instructions is the rightmost register (add \(\% 11, \% 12, \% 13\) means \(\% 13=\% 11+\% 12\) ). SPARC uses a condition code register and special condition-code-setting instructions for branches. Branches include a delay slot.
```

LOOP:
ld [%l1], %12 ! Load 12 = MEM[ 11 ]
addcc %l2, %g0, %g0 ! g0 = g0 + l2. Sets cond. codes. Note: g0 is zero reg.
be DONE ! Branch if result zero.
nop ! Fill delay slot with nop.
add %16, %12, %16 ! 16 = 16 + 12
andcc %l3, 1, %g0 ! g0 = 1 \& l3. Sets cond. codes. Note: g0 is zero reg.
be SKIP1
nop
add %14, 1, %14
SKIP1:
subcc %13, 1000, %g0
bpos SKIP2 ! Branch if >= 0;
nop
add %14, %13, %14
SKIP2:
andcc %13, 1, %g0
be SKIP3
nop
add %14, %14, %14
SKIP3:
add %11, 4, %11
ba LOOP ! Branch always. (Jump.)
nop
DONE:

```

Problem 1: An execution of the code above on a SPARC implementation takes 1000 cycles. The dynamic instruction count is \(\mathrm{IC}_{\text {all }}\) of which \(\mathrm{IC}_{\text {nop }}\) instructions are nop's. Consider two ways of computing CPI:
\[
\mathrm{CPI}_{\mathrm{A}}=\frac{t}{\mathrm{IC}_{\mathrm{all}}} \quad \text { and } \quad \mathrm{CPI}_{\mathrm{B}}=\frac{t}{\mathrm{IC}_{\text {all }}-\mathrm{IC}_{\mathrm{nop}}},
\]
where \(t\) is the execution time in cycles. Which is better? Justify your answer; an argument for either formula can be correct.

Problem 2: SPARC branches have a one-instruction delay slot, in the code above they are filled with nop's. Re-write the code filling as many slots with useful instructions as possible, reducing the number of instructions in the program.

Problem 3: Re-write the program in DLX, taking advantage of DLX's use of general purpose registers for specifying branch conditions.

Problem 4: The program below executes on the DLX implementation shown below. The comments show the results of the xori, or, and 1 w instructions.
! Initially, r1=11, r2=22, r3=33, etc.
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
START: ! START = 0x50
xori r1, r9, \#7 !99 \(\oplus 7=100\)
or \(r 2\), \(r 3, r 4\) ! 33 or \(44=45\)
lw r5, 9(r6) ! Mem[9+66]=42
sw 10(r7), r8
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0


The table below shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. Cycle zero is the time that xori is in instruction fetch. The first two columns are completed, continue filling the table up until the sw instruction finishes writeback. Ignore values which are not used and which depend on the func field of type-R instructions. Values which are not used and don't depend on the func field should be shown. The output of the data memory is zero when a store or no memory operation is performed. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If no register value is written leave the entry blank.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline PC & 0x50 & 0x5 & & & & & & & & & \\
\hline IF/ID.IR & addi & xori & & & & & & & & & \\
\hline Reg. Chng. & r0 & r0 & & & & & & & & & \\
\hline ID/EX.IR & addi & add & & & & & & & & & \\
\hline ID/EX.A & 0 & 0 & & & & & & & & & \\
\hline ID/EX.B & 0 & 0 & & & & & & & & & \\
\hline ID/EX.IMM & 0 & 0 & & & & & & & & & \\
\hline EX/MEM.IR & addi & add & & & & & & & & & \\
\hline EX/MEM. ALU & 0 & 0 & & & & & & & & & \\
\hline EX/MEM.B & 0 & 0 & & & & & & & & & \\
\hline MEM/WB. IR & addi & add & & & & & & & & & \\
\hline MEM/WB . ALU & 0 & 0 & & & & & & & & & \\
\hline MEM/WB.MD & 0 & 0 & & & & & & & & & \\
\hline
\end{tabular}

\section*{EE 4720}

Homework 3
Due: 8 March 1999
In all problems below assume there are no cache misses and that all register values are available at the beginning of execution.

Problem 1: The pipeline shown below cannot execute the jal or jalr instructions. Identify and fix the problem. (Hint: Think about a difference between jal and beqz besides the fact that jal is unconditional.)


Problem 2: The program below executes on the DLX implementation shown below. The comments show the results of some instructions. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others. The forwarding paths are shown. A value can be read from the register file in the same cycle it is written. The destination field in the bneq is zero. Instructions are nulled (squashed) in this problem by replacing them with or r0, r0, r0.


The table below shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. Cycle zero is the time that add is in instruction fetch. The first two columns are completed, filling up the rest of the table. Ignore values which are not used and which depend on the func field of type-R instructions. Values which are not used and don't depend on the func field should be shown. Don't forget the IMM values for bneq. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If no register value is written leave the entry blank.
\(\left.\begin{array}{|l|llllllllll|}\hline \text { Cycle } & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9\end{array}\right) 10\)

Problem 3: The program below executes on the implementation also shown below.

```

add r1, r2, r3
and r4, r1, r5
sw 0(r4), r1
lw r1, 8(r4)
xori r5, r1, \#1
beqz r5, TARGET
sub r5, r5, r5
...
TARGET:
or r10, r5, r1

```

The implementation includes only the forwarding paths that are shown in the figure. A new register value can be read in the same cycle it is written. Show a pipeline execution diagram for an execution of the code in which the branch is taken.

Problem 4: Add exactly those forwarding paths (but no others) that are needed in the DLX implementation used in the problem above so that the code above executes as quickly as possible. Show a pipeline execution diagram of the code (repeated below) on the modified implementation.
```

add r1, r2, r3
and r4, r1, r5
sw 0(r4), r1
lw r1, 8(r4)
xori r5, r1, \#1
beqz r5, TARGET
sub r5, r5, r5
TARGET:
or r10, r5, r1

```

Problem 5: The code below executes on the DLX implementation shown below which also includes the following floating-point hardware:
- As described in Section 3.7 of the text and in class, there is a four-stage FP add unit, a seven-stage multiply unit, and a 25 -cycle FP divide unit (not used in the code below). The FP add unit also performs FP comparisons, such as eqf.
- The floating-point branch instructions, bfpt and bfpf, are executed in the ID stage just as the integer branches, beqz and bneq. The FP condition code register (also not shown) is updated in the WB cycle but the value to be written is forwarded to the controller at the beginning of WB.
- All stalls are in the ID stage. Floating-point instructions skip the MEM stage.
- Floating-point values are forwarded from the WB stage to the inputs of the FP execution units. A value written to a FP register can be read in the same cycle.
(a) Show a pipeline execution diagram for two iterations of the code below in which bfpt is taken in the first iteration but not taken in the second. (Note: the loop is infinite.)
(b) Determine the CPI of an execution of the code for a large number of iterations in which bfpt is always taken.
(c) Determine the CPI of an execution of the code for a large number of iterations in which bfpt is never taken.
(d) Determine the CPI of an execution of the code for a large number of iterations in which bfpt is taken \(50 \%\) of the time.

```

LOOP:
addi r1, r1, \#8
lf f0, 0(r1)
addf f1, f1, f0
eqf f0, f2
bfpt LOOP
multf f1, f1, f3
beqz r0, LOOP
xor r2, r1, r3

```

In all problems below assume there are no cache misses and that all register values are available at the beginning of execution.

Problem 1: Show a pipeline execution diagram for the first 41 cycles of the code below on a dynamically scheduled implementation of DLX in which:
- There is one floating point multiply unit with a latency of 5 and an initiation interval of \(\mathbf{2}\).
- There is a load/store functional unit with a latency of 1 . The segments are labeled L1 and L2.
- The FP add functional unit has a latency of 3 and an initiation interval of 1 .
- The integer functional unit has a latency of 0 and an initiation interval of 1 .
- The functional units have reservation stations with the following numbers: integer, 6-9; fp add, 0-1; fp multiply, 2-3; load/store, 4-5.
- There is no reorder buffer.
- The branch delay is one. (There are no branch delay slots.)
- Ignore load/store ordering.

Initially all reservation stations are available.
LOOP:
```

addi r1, r1, \#8
sub r2, r1, r3
lf f0, O(r1)
multf f1, f0, f0
multf f2, f0, f1
sf 4(r1), f1
bneq r2, LOOP ! Assume always taken.
xor r4, r5, r6

```

Problem 2: Determine the CPI for a large number of iterations of the loop above (or give a good reason why it would be very difficult to determine the CPI).

Problem 3: What are the minimum number of reservation stations of each type needed so that the code above executes at maximum speed? What is the CPI at maximum speed? (This part was not in the problem as originally assigned:) The CDB can handle any number of writebacks per cycle and there are an unlimited number of functional units.

The problem as originally assigned was more tedious than intended. To solve it one would need to find a repeating pattern of iterations. Because of contention for the \(C D B\), the repeating pattern does not occur in the first few iterations and so one would have to tediously construct the diagram for many iterations.

Problem 4: The code below executes on a machine similar to the type described in the first problem except that it uses a reorder buffer. Draw a pipeline execution diagram for the code below, be sure to show when each instruction commits. Remember that instructions stall in the functional unit if they are not granted access to the CDB.
```

LOOP:
lf f0, 0(r1)
multf f1, f0, f0
multf f2, f0, f1
addf f3, f3, f0
lf f4, 8(r1)
sf 4(r1), f1

```
```

multf f1, f4, f4
multf f2, f4, f1
addi r1, r1, \#16
sub r3, r4, r5
xor r6, r7, r8
or r9, r10, r11

```

Problem 5: Consider the code execution from the problem above. Suppose there is an exception in the L2 segment executing the second lf. At what cycle would the trap instruction be inserted? What might go wrong if a reorder buffer had not been used?

Problem 6: Show the execution of the code below on a dynamically scheduled 4-way superscalar machine using a reorder buffer. Instruction fetch is aligned. There is one of each floating-point functional unit, with latencies and initiation intervals given in the first problem. There are four integer execution units. The reservation station numbers are as given in the first problem.
```

LOOP: = 0x1008
lf f0, 0(r1)
multf f1, f0, f0
multf f2, f0, f1
addf f3, f3, f0
lf f4, 8(r1)
sf 4(r1), f1
multf f1, f4, f4
multf f2, f4, f1
addi r1, r1, \#16
sub r3, r4, r5
xor r6, r7, r8
or r9, r10, r1

```

Problem 7: (Modified 12 November 1999) Rewrite the code below for the VLIW DLX ISA presented in class. Instructions can be rearranged and register numbers changed. In order of priority, try to minimize the number of bundles, minimize the use of the serial bit, and maximize the value of the lookahead field. When determining the lookahead assume that any register can be used following the last bundle in your code. LOOP :
```

lf f0, O(r1)
multf f1, f0, f0
multf f2, f0, f1
addf f3, f3, f0
lf f4, 8(r1)
sf 4(r1), f1
multf f1, f4, f4
multf f2, f4, f1
addi r1, r1, \#16
sub r3, r4, r5
xor r6, r7, r8
or r9, r10, r11

```

\section*{37 Spring 1998}

\section*{EE 4720}

Homework 1
Due: 18 February 1998
The code fragment below, in C source and assembler forms, is referred to in the problems below. The distribution of elements in array is unknown.
```

/* C Source Code Fragment */
\#define ISIZE 10
\#define JSIZE 20
\#define BINS 1024
short int array[ISIZE*JSIZE]; /* sizeof(short int) = 2 */
int hist[BINS]; /* sizeof(int) = 4 */
for(i=0; i<ISIZE; i++)
for(j=0; j<JSIZE; j++)
{
int e = array[ i * JSIZE + j ];
hist[ e ]++;
}
! DLX Assembly Code Below (Simplified)
! Register usage:
! r1 = i, r2 = j, r3 = hist, r4 = array
ADDI r10, r0, \#20 ! r10 = JSIZE
MOVI2FP f0, r10 ! Move r10 to FP register for int mult.
ADDI r1, r0, \#0 ! i=0
NEXTI:
SGEI r10, r1, \#10 ! if i >= ISIZE ...
BNEZ r10, DONEI ! ... exit loop.
ADDI r2, r0, \#0 ! j=0
NEXTJ:
SGEI r10, r2, \#20 ! if j >= JSIZE ...
BNEZ r10, DONEJ ! ... exit loop
MOVI2PF f1, r1 ! Move i to FP register.
MULT f1, f1, f0 ! i * JSIZE
MOVFP2I r10, f1
ADD r10, r10, r2 ! i * JSIZE + j
SLLI r10, r10, \#1 ! ( i * JSIZE + j ) * sizeof(short int)
ADD r10, r10, r4 ! r10 = \&array[ i*JSIZE + j ]
LH r10, 0(r10) ! r10 = array[ i*JSIZE + j ]
SLLI r10, r10, \#2 ! r10 = e = array[ i*JSIZE + j ] * sizeof(int)
ADD r10, r10, r3 ! r10 = \&hist[ e ];
LW r11, O(r10) ! r11 = hist[ e ];
ADDI r11, rll, \#1 ! r11 = hist[ e ] + 1;
SW O(r10), rll ! hist[e] = r11
ADDI r2, r2, \#1 ! r2 = j+1
J NEXTJ
DONEJ:
ADDI r1, r1, \#1 ! r1 = i+1
J NEXTI
DONEI:

```

Problem 1: In the program above, identify memory accesses that exhibit temporal locality, memory accesses that exhibit spatial locality, and memory accesses that may exhibit neither property.

Problem 2: Compute the dynamic and static instruction counts of the DLX program above.
Problem 3: Port the program above to an ISA derived from DLX by adding the addressing modes in Figure 2.5 of the text. In this ISA any operand can use any addressing mode. The ISA also includes an integer multiply instruction that actually uses the integer registers. The ported program should use fewer instructions than the original one, the fewer the better. What are the new static and dynamic instruction counts?

Problem 4: Suppose an implementation of the original DLX is clocked at 1 GHz , and uses 0.25 CPI. Suppose the instruction time of an implementation of the new ISA is also 0.25 CPI. At what clock frequency will the implementation of the new ISA be just as fast as the old one (based on your answers to the previous questions)?

To make headlines nowadays you need at least a \(1-\mathrm{GHz}\) clock frequency. An alternate implementation of the new ISA is clocked at 1 GHz . At what instruction execution time (CPI) will this implementation be the same speed as the original one?

Problem 5: Optional (zero credit, but you'll feel good about it and it may help you on future assignments). Type the C code above into a file and have a C compiler generate assembler code without optimization. From the assembler code, determine the static and dynamic instruction counts. How does the real assembler code differ from the code above? Compile the code with optimization and repeat.

On the Suns running Solaris 2.X, the -S switch directs the C compiler to produce assembler output (in file foo.s). For example,
[sol] \% cc sample.c -S
puts the compiled output in assembler form in file sample.s. The -fast switch (not used) tells the compiler (and linker) to optimize (for speed).

The program below is referred to in the problems.
```

!! r4 holds a limit
!! r5 holds the first array element address
add r2, r0, r0 ! Clear sum register.
add r5, r10, rll ! Set r5 to first element.
lw r6,0(r5)
add r2, r2, r6
slt r3, r2, r4
addi r5, r5, \#4
bneq r3, LOOP:

```
LOOP:

Problem 1: Draw a pipeline execution diagram showing the first three iterations of the program above (assuming that it executes for at least three iterations, of course). The program runs on the DLX implementation shown in Figure 3.4 of the text (and below) in which a register can be read from the register file in the same cycle it was written to the register file. The processor will stall in the face of hazards, but there's no data forwarding or other neat stuff covered later in Chapter 3. Note that the execution of the first iteration is different than subsequent iterations.

Problem 2: Show the contents of the pipeline registers (stage latches) and the register file the first time the addi instruction (just before bneq) is in the memory stage. Assume that the first element of the array is 10 , the second element is 20 , etc. Show only registers that are being used (e.g., don't show r22), use a "??" for pipeline register values that cannot be determined or are not being used.


Problem 3: What is the CPI while running the loop above? Ignore initialization and the first iteration.

The program below is referred to in the problems.
```

    !! r4 holds a limit
    !! r5 holds the first array element address
    add r2, r0, r0 ! Clear sum register.
    add r5, r10, rll ! Set r5 to first element.
    LOOP:
lw r6,0(r5)
add r2, r2, r6
slt r3, r2, r4
addi r5, r5, \#4
bneq r3, LOOP

```

The program executes on the DLX Chapter-3 implementation in which the branch address is computed in the ID stage, as shown in the corrected version of Figure 3.22 (COPYRIGHT 1990, 1996 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED), to the right. The pipeline also includes bypass (forwarding) paths to the ALU (not shown).


FIGURE 3.22 The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.

Problem 1: Draw a pipeline execution diagram showing the first two iterations of the program executing on the implementation above. What is the CPI while executing the loop?

Problem 2: Explain how adding forwarding paths to the ID stage would speed the execution of the branch instruction.

Problem 3: Consider an implementation that uses the ID-stage forwarding paths mentioned in the problem above and which also has a branch delay slot. Re-write the program above so that it executes as fast as possible. Draw a pipeline execution diagram showing the first two iterations and compute the CPI of the loop execution.

Problem 1: The pipeline execution diagram below shows two floating-point instructions on the Chapter-3 implementation of DLX, where the divide unit, which is not pipelined, has latency 24 and initiation interval 24, and the multiply unit, which is fully pipelined, has a latency of 6 and an initiation interval of 1 .
```

div IF ID D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 ... D24 MEM WB
mul IF ID M1 M2 M3 M4 M5 M6 M7 MEM WB

```

Finding structural hazards with the notation above is tricky because there is no indication that D1 and D2 refer to the same piece of hardware while M1 and M2 refer to different hardware. (The code above does not encounter a structural hazard; it would if the second instruction were also a divide.) Develop a notation that fixes the problem and can be used for any latency and initiation interval. Use the notation in the pipeline execution diagram for problem 2.

Problem 2: An implementation of DLX performs in-order execution (but out-of-order completion), is fully bypassed, and properly interlocked (the implementation discussed in Chapter 3). MEM-stage structural hazards are resolved by stalling just before the MEM stage. WAW hazards are handled by nulling the earlier instruction. The floating-point functional units perform the operations as described in Chapter 3, however the timings are as described below:
\begin{tabular}{lll} 
Unit & Init. Inter. & Latency \\
DIV & 16 & 15 \\
MUL & 2 & 3 \\
ADD & 1 & 2
\end{tabular}

Find the pipeline execution diagram for the following code on this system
```

div f3, f4, f5
mul f0, f1, f2
mul f3, f6, f7
sub f8, f9, f10
mul f11, f0, f12

```

Problem 3: Repeat the problem above on an implementation that is the same as the one above except MEM-stage structural hazards are resolved by stalling in ID.

Problem 4: The program below runs on a DLX implementation that uses dynamic scheduling with Tomasulo's algorithm and register renaming. The multiply unit has a latency of 8 and an initiation interval of 1, and has two reservation stations, numbered 1 and 2. Branch targets are computed in the ID stage.

Show the execution of the code below up to the second writeback of the multiply instruction assuming (as we have so far) there are no cache misses.
```

    addi r1, r0, #1000
    LOOP:
ld f1, 1024(r1)
subi r1, r1, \#8
mul f0, f0, f1
bneq r1, LOOP

```

Problem 5: For the problem above, how large can multiply's latency be (with an initiation interval of 1) without running out of reservation stations after some number of iterations?

\section*{EE 4720}

Homework 5
Due: 22 April 1998
The code below is referred to in the problems.
```

LOOP: ! LOOP = 0x5000
lw r1, 0(r2)
addi r2, r2, \#4
beqz r1, TARG
sub r4, r4, r1
j LOOP
TARG:
subi r5, r5, \#1
bnez r5, LOOP
and r4, r4, r6
or r4, r4, r7
sw 0(r8), r4
addi r8, r8, \#4
jr r31

```

Problem 1: Show the execution of the code above up to IF of the third iteration on a 4 -way superscalar implementation of DLX which is statically scheduled, instruction fetches are 4-instruction aligned, is fully bypassed including ID stage bypassing for branch conditions, and in which hardware is fully duplicated (including writeback). Assume that the first conditional branch is not taken in the first iteration, but taken in the second iteration, and that the code executes for many iterations. Branches do not have delay slots, hardware cannot detect branch target/fall through overlap, there is no branch prediction hardware, and no branch target prediction hardware. Indicate the cancelling of an instruction by an x in the earliest cycle that the hardware could cancel it. (Do not assume the implementation can predict the future using unspecified hardware or any other means.) For example in a single-issue implementation:
```

Time
TARG :
sub r1, r3, r4 IF ID EX MEM WB

```

Problem 2: Compute the CPI of the execution of a large number of iterations of the loop above when \(30 \%\) of the words starting at the initial value of \(r 2\) hold zero.

Problems on next page.

Problem 3: How effective and how practical would each of the following be in speeding execution of the loop (compared to the problem-1 DLX implementation):
- Branch delay slots.
- Dynamic scheduling.
- Branch target buffer.
- Predicated execution.
- Loop unrolling. (See section 4.1)

The answer should specify how each technique would avoid specific bubbles (if possible) present in the solution to problem 1.

Problem 4: A 4-way VLIW ISA (derived from DLX) includes predicated execution in the following way: the first instruction of a bundle can be a bundle execution specifier (BX) instruction which specifies whether the remaining three instructions execute. The instruction has three register operands, corresponding to the second, third, and fourth instruction in the bundle. Each register operand has a negation bit, indicated in assembly language by an exclamation point. If the negation bit is zero then the corresponding instruction executes if the register contents is non-zero. If the negation bit is one then the corresponding instruction executes if the register contents is zero.

For example, consider the following bundle:
bx r1,!r2,!r0
add r4, r5, r6
sub r7, r8, r9
div f0, f1, f2
The add executes if \(r 1\) is non-zero, the sub executes if \(r 2\) is zero, and the divide always executes.
If the first instruction of a bundle is not bx all instructions will execute (including the first, as an ordinary instruction). Source registers in a bundle refer to values produced in preceding bundles.

Convert the DLX program below to this VLIW ISA.
```

beqz r1, ELSE
add r2, r3, r4
j ENDIF
ELSE:
add r2, r3, r5
ENDIF:
sub r6, r6, r2
bnez r1, SKIP
addi r7, r7, \#1
SKIP:
addi r8, r2, \#12
slt r1, r8, r9
and r10, r8, r11

```

Problem 1: The program below executes on a single-issue (not superscalar) DLX implementation that uses Tomasulo's algorithm for dynamic execution. It also includes dynamic branch prediction using an ID-stage BHT (but no target prediction) and a reorder buffer to support speculative execution.

Instruction execution proceeds speculatively using a predicted path when a branch condition cannot be immediately determined. When the condition is determined the instructions following the branch are cancelled if the prediction was wrong (otherwise execution proceeds normally). Branch instructions use a special branch functional unit, including reservation stations, located in the ID stage. If a branch condition is available when a branch instruction is in the ID stage (the register value is in the register file or can be bypassed to the ID stage that cycle) the branch is executed normally. Otherwise it waits in the branch functional unit until the register value it needs is ready while following instructions execute. Those following instructions may start at the branch target (as soon as available) or the fall through (the instruction after the branch), depending on how the branch is predicted. When the branch outcome is determined it is compared to the prediction, instructions following the branch are cancelled if the prediction was wrong, otherwise execution proceeds normally.

Assume that the reorder buffer has an unlimited capacity. At most one entry per cycle can be retired from the reorder buffer, but any number of elements can be deleted in one cycle. The system has five reservation stations per functional unit, including the integer functional unit, EX, and the branch functional unit, BR.

Show a pipeline execution diagram for the code below when the branch is mispredicted as taken. (That is, the outcome of the branch is not-taken, but it is predicted taken.) Show the contents of the re-order buffer at each cycle, include only the instructions shown below. Show execution until the reorder buffer is empty of the instructions encountered in the execution of the code below. For each entry in the reorder buffer show the instruction mnemonic and place a check next to it if it has completed execution.

The eqf instruction uses the floating-point add functional unit and bfpt uses the branch functional unit described above. Note that there is a dependency between bfpt and eqf.
```

multf f4, f5, f6
addf f0, f1, f2
eqf f0, f3 ! Set floating point condition code to true iff f0=f3
bfpt TARG ! Branch if floating point condition true.
add r1, r2, r3
sub r4, r5, r6
TARG:
and r1, r2, r3
or r4, r5, r6
...

```

Problem 2: The program below runs on a system using a 3-bit branch history branch predictor, with a branch predicted taken if the count is 5 or greater. Initially all entries in the branch history table are zero. What will the prediction accuracy be during the execution of this program? (The program never finishes, for simplicity consider execution until r1 reaches \(2^{31}-1\).)
```

add r1, r0, r0
LOOP:
andi r2, r1, \#0x10
beqz r2, CONTINUE
addi r3, r3, \#1
CONTINUE:
addi r1, r1, \#1
j LOOP

```

Problem 3: Maybe, just maybe, the behavior of a branch in a procedure depends on where the procedure was called from. Suppose it does. Show how to implement a branch predictor that would use this information (the identity of the branch instruction and the caller of the "procedure" containing the branch instruction) to select branch history. The size of the BHT should be limited to \(2^{12}\) entries. Assume that procedures are always called using the jal and jalr instructions. Be sure to show where the address lines for the BHT come from. The solution does not have to show details of the counter predict and update hardware.

Problem 4: Show how \(32 \mathrm{~b} \times 2^{23}\) memory devices can be connected to implement a 27 -bit, byteaddressed address space in which the CPU fetches 64 -bit aligned doublewords. (Using the notation from class, \(a=27, w=64 \mathrm{~b}, c=8 \mathrm{~b}\).) Indicate which memory devices store each of these addresses: 0x0, 0x7e33b8e, 0x3396891.

\section*{38 Spring 1997}

The table below, used in the next two problems, gives the run times on two machines for programs being considered for a benchmark suite.
\begin{tabular}{rccccccccr} 
Program & A1 & A2 & A3 & B1 & B2 & C1 & C2 & C3 & C4 \\
Base machine & 54 & 40 & 17 & 40 & 71 & 40 & 3 & 111 & 7 \\
Test machine & 20 & 35 & 10 & 20 & 20 & 19 & 1 & 40 & 2
\end{tabular}

Problem 1: Find the arithmetic mean, harmonic mean, and geometric mean of the run times in the table above.

Problem 2: The programs in the table above are being considered for a benchmark suite. Three types of programs are represented: compilers (A1, A2, and A3), database programs (B1 and B2), and floating-point intensive programs ( C 1 through C 4 ). Assume the programs are written using a variety of programming styles and compilers and are a good representation of programs that Real People run. Assume that the base machine used for the numbers above does not run any program particularly fast or particularly slow.

Devise a way of combining the run times of the programs on the base machine and a test machine into a number called the TigerMark so that:
- The TigerMark indicates how much faster the test machine is than the base machine. (E.g., 2.1 times faster for a TigerMark of 2.1.)
- Each type of program (compiler, database, floating-point) is of equal importance.
- As many programs as possible are used.

What is the TigerMark rating of the test machine in the table above?

Problem 3: A new instruction is being considered for an ISA which does the same computation as a sequence of five instructions on the current version of the ISA. (That is, the five-instruction sequence can be replaced with the new instruction.) The new instruction takes nine cycles, while the five-instruction sequence takes twelve cycles.

One benefit of the new instruction is the three-cycle savings. Give another benefit and two reasons why the new instruction might not be such a good idea.

Problem 4: Computer designers are considering two implementations ( \(A\) and \(B\) ) and two compilers (I and II) for an ISA. Instructions are divided into three categories, 1,2 , and 3 . In implementation \(A\), instruction execution times (in cycles) are \(\mathrm{CPI}_{1}(A)=2, \mathrm{CPI}_{2}(A)=2\), and \(\mathrm{CPI}_{3}(A)=3\). The number of executed instructions (by category) of a test program compiled using compiler I are \(\mathrm{IC}_{1}(\mathrm{I})=1500, \mathrm{IC}_{2}(\mathrm{I})=1500\), and \(\mathrm{IC}_{3}(\mathrm{I})=5000\). The number of executed instructions (by category) of a test program compiled using compiler II are \(\mathrm{IC}_{1}(\mathrm{II})=900, \mathrm{IC}_{2}(\mathrm{II})=2500\), and \(\mathrm{IC}_{3}(\mathrm{II})=5000\). For the test program compiled with each compiler, find the average instruction execution time and total run time on a system with a 1 MHz clock. Is CPI a good predictor of implementation performance in this case?

Repeat the problem for implementation \(B\) in which \(\operatorname{CPI}_{1}(B)=3, \mathrm{CPI}_{2}(B)=1\), and \(\mathrm{CPI}_{3}(B)=3\).

Problem 1: The assembly language program below is written using DLX-like mnemonics, but for a richer ISA. Re-write the program in DLX. Be sure to consider immediate sizes.
```

LW R1, (\#0x123456) ! Load R1 with contents of memory location 0x123456.
MOV R3, \#0x1234 ! Load R3 with constant 0x1234.
LW R2, @(R3) ! Memory indirect addressing, R2 = Mem[Mem[R3]]
SUB R4, R1, R2
BN Skip ! Branch if last result negative.
LW R4, \#0
Skip:
RET ! Return from subroutine.

```

Problem 2: Do problem 2.3 (page 12) in the book.

\section*{EE 4720}

Homework 3
Due: 3 March 1997
In predicated execution a condition-code bit or register is used to determine if an instruction will execute. In a simple version some machine instructions have predicated forms, for example, with a predicated load word the following conventional code
```

BNEQZ skip ! Branch if condition codes indicate != 0.
LW R1,12(R2) ! Load register R1.
skip:
ADD R3, R1, R2

```
can be replaced with
```

LW,EQ R1,12(R2) ! Load register R1 if condition codes indicate = 0, otherwise do nothing.
ADD R3, R1, R2

```
saving not just one instruction, but also possible stall cycles.
The branch and LW, EQ use condition code registers, while DLX uses regular registers for conditions. The means that there would be no place to specify a condition register in a DLX LW, EQ without removing the immediate.

Another way of adding predicated execution to DLX makes use of a special PM, predicate mask, instruction. The instruction specifies a condition register and two 8 -bit execution masks, the first mask is used if the register contents is non-zero, the other if the contents is zero. The 8 bits in a mask refer to the next 8 instructions in program order (e.g., following branches) to be executed, the LSB (bit 0 ) refers to the next instruction, bit 1 refers to the instruction after that, and so on. (Control flow instructions make things interesting, for now ignore them.) If a mask bit is zero the corresponding instruction does not execute, if the bit is one it does. An instruction not executed in this manner is said to be nullified. The mnemonic for PM indicates the register to test and the mask to use if the register contents is non-zero and the mask to use if the register contents is zero. For example, if the instruction indicated by
```

PM R1, 0x00, Oxff ! Note, 0x00 = 00000000 (binary) and 0xff = 111111111 (binary)

```
executes and R1 is not equal to zero, the next eight instructions are not executed; if R1 is equal to zero the next eight instructions are executed.

The PM instruction is used in the first code fragment as follows:
```

PM R4, Oxfe, Oxff ! Note: Oxfe = 11111110 (binary)
LW R1,12(R2) ! Load register R1 if R4 equals zero.
ADD R3, R1, R2 ! Always executes.

```

Problem 1: Show two DLX code fragments, one with and one without PM designed to show PM in the best possible light (using analysis from next problem).

Problem 2: Consider the pipelined DLX implementation presented in class and the text. Suppose taken branches stall this implementation by three cycles but no stalls result from PM instructions. (The nullified instructions are not counted as stalls.) Determine the CPI and execution time on this machine of the code fragments developed above. (Nullified instructions are not tallied in the instruction count.)

Problem 3: Show how the pipelined DLX implementation would have to be modified to implement PM. Be sure to show any registers added and to explain what additional actions are performed by the controller.

Problem 4: The description above did not specify constraints on what instructions can follow a PM. Determine which instructions would cause problems and provide suggestions on what to do about them. Problems might include ambiguity on which instructions execute and confusing execution paths with no apparent computational benefit.

Consider a DLX implementation fabricated based on the solution to homework 3 part 3 given in http://www.ee.lsu.edu/koppel/ee4720/1997/hw03_sol.html, and which also includes register forwarding connections so that only load instructions would stall because of RAW hazards.

Problem 1: Consider the fragment below.
```

PM R4, Oxfd, Oxfa ! Note: Oxfd = 11111101 0xfa = 11111010
ADD R3, R1, R2
LW R7, 0(R20)
SUB R6, R7, R8

```

Using a timing diagram (the table with IF, ID, etc. entries for each instruction) show what would happen when the fragment is executed with R4 both equal to zero and not equal to zero. Repeat the problem for the fragment below, in which the contents of R5 is zero.
```

PM R4, 0x13, Oxfc
LOOP:
ADD R3, R1, R2
BEQZ R5, LOOP
SUB R6, R7, R8

```

Problem 2: Suppose a DLX implementation was fabricated based on the solution to homework 3 part 3 given in http://www.ee.lsu.edu/koppel/ee4720/1997/hw03_sol.html, and also included register forwarding connections so that only load instructions would have RAW hazards. Users of this chip are advised never to place a control-transfer or load instruction in the eight instructions following a PM unless the corresponding bits of the execution masks for the instruction and those following are one. For example
```

PM R4, Oxfc, Oxff ! Note: Oxfe = 11111100 (binary)
ADD R3, R1, R2
SUB R6, R7, R8
BNEQ R5, SOMEWHERE

```
is okay because the mask bits will be one for the branch and following instructions. Anyone violating these rules would have to answer to Mike. Since not everyone realizes that this threat has teeth, an illegal instruction exception (non-precise) should be generated when the rule is violated. Modify the solution to HW 3 so that a signal is generated, PM_VIOL, if the rule is violated. How difficult would it be to make this exception precise?

Problem 3: Suppose we don't want to proscribe loads and branches in the scope of a PM. Modify the solution to homework 3 problem 3 so that such loads and branches are handled properly. Note that if a branch is taken, the remaining execution mask bits would apply to the instructions at the target.

Problem 1: Design an interlock mechanism to handle RAW hazards on the DLX pipeline with floating-point execution units. Develop and describe the design so that it can easily be modified to accommodate changes in the timing of the functional units. For example, it should be easy to change the design if the latency and initiation interval of the divide unit were changed to 18 and the initiation interval of the multiplier were raised to 2 .

Problem 1: Show timing diagrams verifying the latencies given in Figure 4.2 on page 224.
Problem 2: (The solution to this problem is based on material in section 3.9, which was not covered in class.) Show the timing of the following code fragment on the MIPS R4000 pipeline based on the pipeline stages given in Figure 3.56. Assume instructions in the RF stage stall (slip in MIPS parlance) only if there is any structural hazard in the floating point execution units or if there are any RAW hazards. (The issue rules in a MIPS R4000 processor are stricter than that. Those who are curious or need material to reinforce section 3.9 can follow
http://www.sgi.com/MIPS/products/r4400/UMan/R4000.book_1.html, keeping in mind that actual R4000 issue rules should not be used in the solution. )

LD F0, O(R2)
```

NEG FO, FO
ADD F2, F0, F4
CGT F6, F8 ! Set FP cond to true if F6 > F8. (FP Compare.)
ADD F10, F12, F14

```

Homework 7
Due: 23 April 1997

Problem 1: Consider a version of the Chapter-4 DLX pipeline that uses the following implementation of Tomasulo's Approach: Each functional unit has two reservation stations, numbered as follows: 1 and 2 for EX, 3 and 4 for ADD, 5 and 6 for MUL, and 7 and 8 for DIV. The integer execution unit is one stage and has an initiation interval of 1 ; all floating point units units are four stages and also have an initiation interval of 1 (these are the Chapter-4 DLX pipeline functionalunit timings). There is no bypassing other than that provided by the common data bus, so the timings given in Table 4.2 don't apply. Assume that all integer instructions use the memory stage but none of the floating-point instructions do. Load and store instructions use the integer pipeline in the same way as other integer instructions (do not assume special load and store buffers, do not worry about cache misses, and especially don't worry about why cache misses are something one might worry about). Show a timing diagram for the code below executing on this system up to (and including) the second fetch of the first ld instruction. Assume that all register values are available when the fragment starts.
```

muld f10, f20, f22
addd f22, f10, f10
muld f24, f10, f26
subd f0, f10, f30
loop:
ld f2, 0(r2)
subd f0, f0, f2
ld f2, 8(r2)
subd f0, f0, f2
ld f2, 16(r2)
subd f0, f0, f2
subi r3, r3, \#1
addi r2, r2, \#24
bnez r3, loop

```

Problem 2: Design hardware implementing ( \(m, n\) )-bit correlating branch prediction for the Chapter 3 DLX pipeline using a \(2^{h}\)-entry BHT. (Ignore the fact that a branch prediction in the Chapter 3 DLX pipeline is not useful.) The hardware should generate a PRED_TAKEN signal in ID when the instruction is a branch and is predicted taken. Don't forget to update the BHT based on branch outcome. Draw a logic diagram showing details of added hardware. Include address, data, and read/write signals for the BHT and any other storage devices used. Show the execution of a branch on the modified pipeline, indicating where important actions take place (e.g., BHT written, prediction ready).

The following code fragment is used in several problems below.
```

add r1, r2, r3
sub r6, r7, r8
lw r10, 0(r20)
add r11, r10, r12
sub r14, r1, r9
add r1, r14, r15
sub r16, r17, r18
add r19, r21, r22
sw 0(r20), r6

```

Problem 1: A 3-way VLIW version of the DLX ISA, 3VDLX, packs three DLX instructions into a single 3VDLX instruction. The values of source registers in a 3VDLX instruction are based on the execution of the previous instructions, not the current one.

An implementation of 3VDLX must be fully pipelined and bypassed. Like the DLX implementations, integer arithmetic values are available in the next cycle, but loads are available to instructions fetched two cycles later (than the 3VDLX instruction containing the load instruction).

Show how the DLX code fragment above can be re-written for 3VDLX, showing one 3VDLX instruction as three DLX instructions on a line separated by semicolons. Show a timing diagram for the execution of the code fragment. The fragment should be written to execute quickly.

Problem 2: Show how the code fragment would execute on a static-issue, 3-way superscalar machine. The superscalar machine is similar to the DLX implementations but has three complete pipelines, it is fully bypassed.

Problem 3: Show how the code fragment would execute on a dynamic-issue, 3-way superscalar machine using Tomasulo's Approach, similar to the DLX implementations. There are three integer execution units, three load/store units, and two reservation stations for each functional unit. The memory stage is used only by load and store instructions.

Problem 4: Using \(2^{18}\)-entry by 16 -bit memory devices, design a memory system that will provide \(16.777216 \mathrm{MB}=2^{24}\) bytes of memory \({ }^{1}\) in 32 -bit words. (This is only slightly more complicated than the simple case given in class.)

\footnotetext{
1 " M " is a combining form meaning \(1,000,000\). Unfortunately, in certain contexts it is used for \(1,048,576\), and it's not always clear which of the two interpretations is intended. In this class "M" will always mean \(1,000,000\).
}

\section*{39 Spring 2023 Solutions}

\section*{排 LSU EE 4720 Spring 2023 Homework 1 －－SOLUTION}非非
\＃Assignment https：／／www．ece．lsu．edu／ee4720／2023／hw01．pdf

非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非

\section*{Problem 1}

非 Instructions：https：／／www．ece．lsu．edu／ee4720／2023／hw01．pdf非

\section*{．text}
lookup：

\section*{非 Register Usage}

非
非 CALL VALUES
非 \＄a0：Word to look up．
非 \＄a1：Start of word table．
非 \＄a2：Start of length table．
非
非 RETURN VALUE
非 \＄v0：Index of word（at \＄a0），or－1 if word is not in table．
非
非［ \(\checkmark\) ］Testbench should show zero errors．
非［厄］Can only modify these registers：\＄t0－\＄t9，\＄a0－\＄a3，\＄v0，\＄v1
非［ ］The fewer instructions executed，the better．
非［ \(\checkmark\) ］Write code clearly，comment for an expert MIPS programmer．
非［ \(\checkmark\) ］Do not use pseudoinstructions except for nop．
非［ \(\checkmark\) ］There is no storage that can be used between calls．
addi \＄t3，\＄a1， 0 非 Word table location．
addi \＄t2，\＄a2， 0 非 Length table．
非 Determine the size of the lookup word．
非
addi \＄t0，\＄a0， 0
SIZE＿LOOP：
lb \＄t1，1（\＄t0）
bne \＄t1，\＄0，SIZE＿LOOP
addi \＄t0，\＄t0， 1
非
sub \＄t8，\＄t0，\＄a0 非 t8：Length of lookup word．

WORD＿LOOP：
非 Check Next Word in Word Table
非
lb \＄t0， \(0(\$ \mathrm{t} 2)\) 非 Length of current word－table word

非 Check whether loop has reached the end of the word table．
beq \＄t0，\＄0，WORD＿NOT＿FOUND
```

    addi $t2, $t2, 1
    addi $t5, $t3, 0 非 First character in table word to use now.
    # Compare length of table word (t0) to length of lookup word (t8)
    非
    bne $t0, $t8, WORD_LOOP
    add $t3, $t3, $t0 非 First character in table word to use later.
    # The lengths match, so need to check the words character by
    # character.
    addi $t4, $a0, 0 非 First character of lookup word.
    非 Compare lookup word and table word.
    非
    CHAR_LOOP:
beq \$t5, \$t3, CASE_TABLE_WORD_END
lb $t7, 0($t4) 非 Lookup word
lb $t6, 0($t5) 非 Table word
addi \$t5, \$t5, 1
beq \$t6, \$t7, CHAR_LOOP
addi \$t4, \$t4, 1
\# Characters don't match. Check whether current position in
\#\# word table alphabetically precedes the lookup word. If so
非 move on to the next word.
非
slt \$t6, \$t6, \$t7
bne \$t6, \$0, WORD_LOOP
nop
WORD_NOT_FOUND:
非 Here word hasn't been and won't be found.
jr \$ra
addi \$v0, \$0, -1
CASE_TABLE_WORD_END:
\# Check whether lookup word has ended, if not no match yet.
bne \$t7, \$0, WORD_LOOP
sub \$v0, \$t2, \$a2
\# Found a match!
jr \$ra
addi \$v0, \$v0, -1

```

非非非非非非非非非非非非非非非非非非非非非非非腓非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非
非
    非 Test Code
非
非 The code below calls the lookup routine.

\section*{非 Word Table}
table＿words：非 Total size 532 characters．
\begin{tabular}{|c|c|c|c|}
\hline ．ascii＂aah＂ & 非 Index & 0 & Length \\
\hline ．ascii＂aardvark＂ & 非 Index & 1 & Length 8 \\
\hline ．ascii＂able＂ & 非 Index & 2 & Length 4 \\
\hline ．ascii＂accoutering＂ & 非 Index & 3 & Length 11 \\
\hline ．ascii＂accouterments＂ & \＃ Index & 4 & Length 13 \\
\hline ．ascii＂altarpieces＂ & Index & 5 & Length 11 \\
\hline ．ascii＂altars＂ & 非 Index & 6 & Length 6 \\
\hline ．ascii＂alter＂ & Index & 7 & Length \\
\hline ．ascii＂alterable＂ & 非 Index & 8 & Length \\
\hline ．ascii＂alteration＂ & \＃ Index & 9 & Length 10 \\
\hline ．ascii＂as＂ & 非 Index & 10 & Length \\
\hline ．ascii＂asbestos＂ & 非 Index & 11 & Length 8 \\
\hline ．ascii＂bibliographic＂ & Index & 12 & Length 13 \\
\hline ．ascii＂bibliographical＂ & \＃\(⿰ ⿰ 三 丨 ⿰ 丨 丨 ⿱ 一 土\) & 13 & Length 15 \\
\hline ．ascii＂bibliographically＂ & \＃ Index & 14 & Length 17 \\
\hline ．ascii＂blur＂ & \＃Index & 15 & Length 4 \\
\hline ．ascii＂blurb＂ & \＃ Index & 16 & Length \\
\hline ．ascii＂blurbs＂ & 非 Index & 17 & Length \\
\hline ．ascii＂counselors＂ & 非 Index & 18 & Length 10 \\
\hline ．ascii＂counsels＂ & \＃Index & 19 & Length 8 \\
\hline ．ascii＂count＂ & 非 Index & 20 & Length \\
\hline ．ascii＂countable＂ & 非 Index & 21 & Length \\
\hline ．ascii＂countably＂ & 非 Index & 22 & Length 9 \\
\hline ．ascii＂countdown＂ & \＃Index & 23 & Length 9 \\
\hline ．ascii＂cross＂ & 非 Index & 24 & Length 5 \\
\hline ．ascii＂crystalline＂ & 非 Index & 25 & Length 11 \\
\hline ．ascii＂crystallization＂ & \＃\(⿰ ⿰ 三 丨 ⿰ 丨 丨 ⿱ 一 土\) & 26 & Length 15 \\
\hline ．ascii＂diagrammatic＂ & 非 Index & 27 & Length 12 \\
\hline ．ascii＂diagrammatically＂ & 非 Index & 28 & Length 16 \\
\hline ．ascii＂diagrammed＂ & 非 Index & 29 & Length 10 \\
\hline ．ascii＂electroencephalogram＂ & 非 Index & 30 & Length 20 \\
\hline ．ascii＂electromagnetically＂ & 非 Index & 31 & Length 19 \\
\hline ．ascii＂elucidations＂ & 非 Index & 32 & Length 12 \\
\hline ．ascii＂elude＂ & 非 Index & 33 & Length 5 \\
\hline ．ascii＂eluded＂ & 非 Index & 34 & Length 6 \\
\hline ．ascii＂fishwives＂ & 非 Index & 35 & Length 9 \\
\hline ．ascii＂fishy＂ & 非 Index & 36 & Length 5 \\
\hline ．ascii＂fissile＂ & 非 Index & 37 & Length 7 \\
\hline ．ascii＂fission＂ & 非 Index & 38 & Length 7 \\
\hline ．ascii＂fissionable＂ & 非 Index & 39 & Length 11 \\
\hline ．ascii＂fissure＂ & 非 Index & 40 & Length 7 \\
\hline ．ascii＂i＂ & 非 Index & 41 & Length 1 \\
\hline ．ascii＂mucky＂ & 非 Index & 42 & Length 5 \\
\hline ．ascii＂mucous＂ & 非 Index & 43 & Length 6 \\
\hline ．ascii＂mucus＂ & 非 Index & 44 & Length 5 \\
\hline ．ascii＂mud＂ & 非 Index & 45 & Length 3 \\
\hline ．ascii＂oversimple＂ & \＃Index & 46 & Length 10 \\
\hline ．ascii＂oversimplification＂ & 非 Index & 47 & Length 18 \\
\hline ．ascii＂oversimplifications＂ & 非 Index & 48 & Length 19 \\
\hline ．ascii＂oversimplified＂ & \＃ Index & 49 & Length 14 \\
\hline ．ascii＂weightlessly＂ & 非 Index & 50 & Length 12 \\
\hline ．ascii＂weightlessness＂ & Index & 51 & Length 14 \\
\hline
\end{tabular}
```

.ascii "weightlifter"
.ascii "zucchinis"
.ascii "zwieback"
.ascii "zydeco"
.ascii "zygote"
.ascii "zygotes"

```
．byte 0

\section*{非 Word Length Table \\ table＿word＿lengths：}
\begin{tabular}{|c|c|c|c|c|c|}
\hline ．byte & 3 & \＃ & Index & 0 & aah \\
\hline ．byte & 8 & 非 & Index & 1 & aardvark \\
\hline ．byte & 4 & 非 & Index & 2 & able \\
\hline ．byte 11 & 11 & 非 & Index & 3 & accoutering \\
\hline ．byte 13 & 13 & 非 & Index & 4 & accouterments \\
\hline ．byte 11 & 11 & 非 & Index & 5 & altarpieces \\
\hline ．byte & 6 & 非 & Index & 6 & altars \\
\hline ．byte & 5 & 非 & Index & 7 & alter \\
\hline ．byte & 9 & 非 & Index & 8 & alterable \\
\hline ．byte 10 & 10 & 非 & Index & 9 & alteration \\
\hline ．byte & 2 & 非 & Index & 10 & as \\
\hline ．byte & 8 & 非 & Index 11 & 11 & asbestos \\
\hline ．byte 13 & 13 & 非 & Index 1 & 12 & bibliographic \\
\hline ．byte 15 & 15 & 非 & Index 1 & 13 & bibliographical \\
\hline ．byte 17 & 17 & 非 & Index 1 & 14 & bibliographically \\
\hline ．byte & 4 & 非 & Index 1 & 15 & blur \\
\hline ．byte & 5 & 非 & Index 1 & 16 & blurb \\
\hline ．byte & 6 & 非 & Index 17 & 17 & blurbs \\
\hline ．byte 10 & 10 & 非 & Index 1 & 18 & counselors \\
\hline ．byte & 8 & 非 & Index 1 & 19 & counsels \\
\hline ．byte & 5 & 非 & Index 20 & 20 & count \\
\hline ．byte & 9 & 非 & Index 2 & 21 & countable \\
\hline ．byte & 9 & 非 & Index 22 & 22 & countably \\
\hline ．byte & 9 & 非 & Index 2 & 23 & countdown \\
\hline ．byte & 5 & 非 & Index 2 & 24 & cross \\
\hline ．byte 11 & 11 & 非 & Index 2 & 25 & crystalline \\
\hline ．byte 15 & 15 & 非 & Index 2 & 26 & crystallization \\
\hline ．byte 12 & 12 & 非 & Index 2 & 27 & diagrammatic \\
\hline ．byte 16 & 16 & 非 & Index 2 & 28 & diagrammatically \\
\hline ．byte 10 & 10 & 非 & Index 2 & 29 & diagrammed \\
\hline ．byte 20 & 20 & 非 & Index 3 & 30 & electroencephalogram \\
\hline ．byte 19 & 19 & 非 & Index 3 & 31 & electromagnetically \\
\hline ．byte 12 & 12 & 非 & Index 32 & 32 & elucidations \\
\hline ．byte & 5 & 非 & Index 3 & 33 & elude \\
\hline ．byte & 6 & 非 & Index 3 & 34 & eluded \\
\hline ．byte & 9 & 非 & Index 3 & 35 & fishwives \\
\hline ．byte & 5 & 非 & Index 36 & 36 & fishy \\
\hline ．byte & 7 & 非 & Index 37 & 37 & fissile \\
\hline ．byte & 7 & 非 & Index 3 & 38 & fission \\
\hline ．byte 11 & 11 & 非 & Index 3 & 39 & fissionable \\
\hline ．byte & 7 & 非 & Index 4 & 40 & fissure \\
\hline ．byte & 1 & 非 & Index 4 & 41 & 1 \\
\hline ．byte & 5 & 非 & Index 4 & 42 & mucky \\
\hline ．byte & 6 & 非 & Index 4 & 43 & mucous \\
\hline
\end{tabular}
\begin{tabular}{ll} 
非 Index 52 & Length 12 \\
\＃Index 53 & Length 9 \\
\＃Index 54 & Length 8 \\
\＃ \\
Index 55 & Length 6 \\
非 Index 56 & Length 6 \\
\＃Index 57 & Length 7 \\
非 Combined & Lengths 532
\end{tabular}

非 Index 52 Length 12
非 Index 53 Length 9
非 Index 54 Length 8
非 Index 55 Length 6
非 Index 56 Length 6
非 Index 57 Length 7
非 Combined Lengths 532
\begin{tabular}{|c|c|c|c|c|}
\hline ．byte & 5 & 非 & Index 44 & mucus \\
\hline ．byte & 3 & 非 & Index 45 & mud \\
\hline ．byte 1 & 10 & 非 & Index 46 & oversimple \\
\hline ．byte 1 & 18 & 非 & Index 47 & oversimplification \\
\hline ．byte 1 & 19 & 非 & Index 48 & oversimplifications \\
\hline ．byte 1 & 14 & 非 & Index 49 & oversimplified \\
\hline ．byte 1 & 12 & 非 & Index 50 & weightlessly \\
\hline ．byte 1 & 14 & 非 & Index 51 & weightlessness \\
\hline ．byte 1 & 12 & 非 & Index 52 & weightlifter \\
\hline ．byte & 9 & 非 & Index 53 & zucchinis \\
\hline ．byte & 8 & 非 & Index 54 & zwieback \\
\hline ．byte & 6 & 非 & Index 55 & zydeco \\
\hline ．byte & 6 & 非 & Index 56 & zygote \\
\hline ．byte & 7 & 非 & Index 57 & zygotes \\
\hline
\end{tabular}

\section*{非 Lookup Words}
test＿words：
\begin{tabular}{|c|c|c|c|}
\hline ．asciiz＂a＂ & 非 & Index & －1 \\
\hline ．asciiz＂aah＂ & 非 & Index & 0 \\
\hline ．asciiz＂able＂ & 非 & Index & 2 \\
\hline ．asciiz＂i＂ & 非 & Index & 41 \\
\hline ．asciiz＂blank＂ & 非 & Index & －1 \\
\hline ．asciiz＂counselor＂ & 非 & Index & －1 \\
\hline ．asciiz＂county＂ & 非 & Index & －1 \\
\hline ．asciiz＂fish＂ & 非 & Index & －1 \\
\hline ．asciiz＂gram＂ & 非 & Index & －1 \\
\hline ．asciiz＂palindromic＂ & 非 & Index & －1 \\
\hline ．asciiz＂zymurgy＂ & 非 & Index & －1 \\
\hline ．asciiz＂bibliographical＂ & 非 & Index & 13 \\
\hline ．asciiz＂bibliographically＂ & 非 & Index & 14 \\
\hline ．asciiz＂cross＂ & 非 & Index & 24 \\
\hline ．asciiz＂zydeco＂ & 非 & Index & 55 \\
\hline ．asciiz＂zygotes＂ & 非 & Index & 57 \\
\hline
\end{tabular}
．align 4
test＿data：
\begin{tabular}{|c|c|}
\hline ．half 0 & －1 非 a \\
\hline ．half & 0 非 aah \\
\hline ．half 6 & 2 非 able \\
\hline ．half 11 & 41 非 i \\
\hline ．half 13 & －1 非 blank \\
\hline ．half 19 & －1 非 counselor \\
\hline ．half 29 & －1 非 county \\
\hline ．half 36 & －1 非 fish \\
\hline ．half 41 & －1 非 gram \\
\hline ．half 46 & －1 非 palindromic \\
\hline ．half 58， & －1 非 zymurgy \\
\hline ．half 66， & 13 非 bibliographical \\
\hline ．half 82 & 14 非 bibliographically \\
\hline ．half 100 & 24 非 cross \\
\hline ．half 106 & 55 非 zydeco \\
\hline ．half 113 & 57 非 zygotes \\
\hline a＿end： & \\
\hline
\end{tabular}
msg＿correct：
```

    .asciiz "%/s0/-17s Num Insn: %/t1/5d Index: %/v1/3d -- Correct\n"
    ```
msg＿incorrect：
．asciiz＂\％／s0／－17s Num Insn：\％／t1／5d Index：\％／v1／3d Er SHOULD BE \％／t2／d\n＂
end＿msg：
．asciiz＂TOTALS：Num Insn：\％／k0／5d Tests：\％／k1／3d Errors：\％／s1／2d n n＂
．text
. globl＿＿start
＿＿start：
mtc0 \＄0，\＄22 ⿰⿰三丨⿰丨三一期 Pause tracing．
非 s0：Address of test word．
addi \＄s1，\＄0， \(0 \quad\) 非 Number of incorrect indices．
la \＄s2，test＿data＿end 非 Holds same value．
非 s5：Number of instructions before．
la \＄s6，test＿words 非 Holds same value．
la \＄s7，test＿data 非 Incremented each iteration．
addi \＄k0，\＄0， 0 非 Total number of instructions．
addi \＄k1，\＄0， 0 非 Total number of lookup words．
非 Number correct．
TB＿LOOP：
lh \＄t0，0（\＄s7）非 Offset in word table．
addi \＄k1，\＄k1， 1
add \＄s0，\＄s6，\＄t0
add \＄a0，\＄s0，\＄0
la \＄a1，table＿words
la \＄a2，table＿word＿lengths
addi \＄v0，\＄0，－1
mtc0 \＄v0，\＄22 非 Resume tracing．（No effect if not stepping．）
jal lookup
mfc0 \＄s5，\＄9 非 Copy current instruction count．（Before．）
mfc0 \＄t0，\＄9 非 Copy current instruction count．（After．）
mtc0 \＄0，\＄22 非 Pause tracing．
addi \＄v1，\＄v0， 0
sub \＄t1，\＄t0，\＄s5
add \＄k0，\＄k0，\＄t1
lh \＄t2，2（\＄s7）非 Correct index．
la \＄a0，msg＿correct
beq \＄t2，\＄v1，TB＿SKIP
nop
addi \＄s1，\＄s1， 1
la \＄a0，msg＿incorrect
TB＿SKIP：
addi \＄v0，\＄0， 11
syscall
addi \＄s7，\＄s7， 4
bne \＄s7，\＄s2，TB＿LOOP
nop
la \＄a0，end＿msg
addi \＄v0，\＄0， 11
syscall
addi \$v0, \$0, 10
syscall
nop

\section*{排 LSU EE 4720 Spring 2023 Homework 1 －－SOLUTION}非非

\author{
非 Assignment https：／／www．ece．lsu．edu／ee4720／2023／hw01．pdf
}

非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非

\section*{Problem 1}

非 Instructions：https：／／www．ece．lsu．edu／ee4720／2023／hw01．pdf非

\section*{．text}
lookup：
```

非 Register Usage

```

非
非 CALL VALUES
非 \＄a0：Word to look up．
非 \＄a1：Start of word table．
\＃\(⿰ ⿰ 三 丨 ⿰ 丨 三 一 灬 ~ \$ a 2: ~ S t a r t ~ o f ~ l e n g t h ~ t a b l e . ~\)
非
\＃RETURN VALUE
非 \＄v0：Index of word（at \＄a0），or－1 if word is not in table．
非
［ \(\quad \checkmark\) ］Testbench should show zero errors．
非 \([\checkmark]\) Can only modify these registers：\＄t0－\＄t9，\＄a0－\＄a3，\＄v0，\＄v1
非［ \(\checkmark\) ］The fewer instructions executed，the better．
\＃［ \(\checkmark\) ］Write code clearly，comment for an expert MIPS programmer．
非［ \(\checkmark\) ］Do not use pseudoinstructions except for nop．
非［ \(\quad\) ］There is no storage that can be used between calls．

非 Fast Solution
非
addi \＄t3，\＄a1， 0 非 Word table location．
addi \＄t2，\＄a2， 0 非 Length table．

非 Determine the size of the lookup word．
非
addi \＄t0，\＄a0， 0
SIZE＿LOOP：
1b \＄t1，1（\＄t0）
bne \＄t1，\＄0，SIZE＿LOOP
addi \＄t0，\＄t0， 1
非
sub \＄t8，\＄t0，\＄a0 非 t8：Length of lookup word．

非 Hold on to the first character of the lookup word．
非
lb \＄t9，0（\＄a0）非 t9：First character of lookup word．

WORD＿LOOP：
lb \＄t0，0（\＄t2）非 Length of current word－table word
WORD＿LOOP2：
beq \＄t0，\＄0，WORD＿NOT＿FOUND
add \＄t3，\＄t3，\＄t0 非 Start of next word．
非 Check whether length of lookup word and table word match．
bne \＄t0，\＄t8，WORD＿LOOP
addi \＄t2，\＄t2， 1 非 Start of next length．
sub \＄t5，\＄t3，\＄t0 非 Re－compute start of table word．
非 Check whether first characters match ．．
非 ．．and if not，whether lookup word is after table word alphabetically．非
lb \＄t6，0（\＄t5）非 Table word
beq \＄t6，\＄t9，CHAR＿LOOP＿PRE
slt \＄t6，\＄t6，\＄t9
非
bne \＄t6，\＄0，WORD＿LOOP2
lb \＄t0，0（\＄t2）非 Word length

非 At this point lookup word follows table word，so the word非 can＇t be in the table．
jr \＄ra
addi \＄v0，\＄0，－1

非 Compare lookup word and table word ．．
非 ．．starting at SECOND character．
非
CHAR＿LOOP＿PRE：
addi \＄t4，\＄a0， 1 非 Addr in lookup word．
CHAR＿LOOP：
addi \＄t5，\＄t5， 1
CHAR＿BODY：
beq \＄t5，\＄t3，CASE＿TABLE＿WORD＿END
lb \＄t7，0（\＄t4）非 Lookup word
lb \＄t6，0（\＄t5）非 Table word
beq \＄t6，\＄t7，CHAR＿LOOP
addi \＄t4，\＄t4， 1
slt \＄t6，\＄t6，\＄t7
bne \＄t6，\＄0，WORD＿LOOP2
lb \＄t0，0（\＄t2）非 Word length
WORD＿NOT＿FOUND：
非 Here word hasn＇t been and won＇t be found．
jr \＄ra
addi \＄v0，\＄0，－1
CASE＿TABLE＿WORD＿END：
非 Found a match！
sub \＄v0，\＄t2，\＄a2
jr \＄ra
addi \＄v0，\＄v0，－1

非 The code below calls the lookup routine．

\section*{．data}

非 Word Table
table＿words：非 Total size 532 characters．
\begin{tabular}{|c|c|c|}
\hline ．ascii＂aah＂ & Index 0 & Length \\
\hline ．ascii＂aardvark＂ & Index 1 & Length 8 \\
\hline ．ascii＂able＂ & 非 Index 2 & Length 4 \\
\hline ．ascii＂accoutering＂ & Index 3 & Length 11 \\
\hline ．ascii＂accouterments＂ & Index 4 & Length 13 \\
\hline ．ascii＂altarpieces＂ & 非 Index 5 & Length 11 \\
\hline ．ascii＂altars＂ & Index 6 & Length 6 \\
\hline ．ascii＂alter＂ & Index 7 & Length 5 \\
\hline ．ascii＂alterable＂ & Index 8 & Length \\
\hline ．ascii＂alteration＂ & Index 9 & Length 10 \\
\hline ．ascii＂as＂ & 非 Index 10 & Length 2 \\
\hline ．ascii＂asbestos＂ & Index 11 & Length 8 \\
\hline ．ascii＂bibliographic＂ & 非 Index 12 & Length 13 \\
\hline ．ascii＂bibliographical＂ & Index 13 & Length 15 \\
\hline ．ascii＂bibliographically＂ & 非 Index 14 & Length 17 \\
\hline ．ascii＂blur＂ & Index 15 & Length 4 \\
\hline ．ascii＂blurb＂ & 非 Index 16 & Length 5 \\
\hline ．ascii＂blurbs＂ & 非 Index 17 & Length 6 \\
\hline ．ascii＂counselors＂ & 非 Index 18 & Length 10 \\
\hline ．ascii＂counsels＂ & 非 Index 19 & Length 8 \\
\hline ．ascii＂count＂ & 非 Index 20 & Length 5 \\
\hline ．ascii＂countable＂ & 非 Index 21 & Length 9 \\
\hline ．ascii＂countably＂ & Index 22 & Length 9 \\
\hline ．ascii＂countdown＂ & Index 23 & Length 9 \\
\hline ．ascii＂cross＂ & 非 Index 24 & Length 5 \\
\hline ．ascii＂crystalline＂ & 非 Index 25 & Length 11 \\
\hline ．ascii＂crystallization＂ & Index 26 & Length 15 \\
\hline ．ascii＂diagrammatic＂ & 非 Index 27 & Length 12 \\
\hline ．ascii＂diagrammatically＂ & 非 Index 28 & Length 16 \\
\hline ．ascii＂diagrammed＂ & Index 29 & Length 10 \\
\hline ．ascii＂electroencephalogram＂ & Index 30 & Length 20 \\
\hline ．ascii＂electromagnetically＂ & 非 Index 31 & Length 19 \\
\hline ．ascii＂elucidations＂ & 非 Index 32 & Length 12 \\
\hline ．ascii＂elude＂ & 非 Index 33 & Length \\
\hline ．ascii＂eluded＂ & 非 Index 34 & Length \\
\hline ．ascii＂fishwives＂ & 非 Index 35 & Length 9 \\
\hline ．ascii＂fishy＂ & Index 36 & Length 5 \\
\hline ．ascii＂fissile＂ & 非 Index 37 & Length 7 \\
\hline ．ascii＂fission＂ & 非 Index 38 & Length 7 \\
\hline ．ascii＂fissionable＂ & 非 Index 39 & Length 11 \\
\hline ．ascii＂fissure＂ & 非 Index 40 & Length 7 \\
\hline ．ascii＂i＂ & 非 Index 41 & Length 1 \\
\hline ．ascii＂mucky＂ & 非 Index 42 & Length \\
\hline ．ascii＂mucous＂ & 非 Index 43 & Length 6 \\
\hline
\end{tabular}
.ascii "mucus"
.ascii "mud"
.ascii "oversimple"
.ascii "oversimplification"
.ascii "oversimplifications"
.ascii "oversimplified"
.ascii "weightlessly"
.ascii "weightlessness"
.ascii "weightlifter"
.ascii "zucchinis"
.ascii "zwieback"
.ascii "zydeco"
.ascii "zygote"
.ascii "zygotes"

\section*{.byte 0}

\section*{非 Word Length Table}
table_word_lengths:

\begin{tabular}{|c|c|c|c|}
\hline ．byte & 5 & 非 Index 36 & fishy \\
\hline ．byte & 7 & 非 Index 37 & fissile \\
\hline ．byte & 7 & 非 Index 38 & fission \\
\hline ．byte 1 & 11 & 非 Index 39 & fissionable \\
\hline ．byte & 7 & 非 Index 40 & fissure \\
\hline ．byte & 1 & 非 Index 41 & i \\
\hline ．byte & 5 & 非 Index 42 & mucky \\
\hline ．byte & 6 & 非 Index 43 & mucous \\
\hline ．byte & 5 & 非 Index 44 & mucus \\
\hline ．byte & 3 & 非 Index 45 & mud \\
\hline ．byte 1 & 10 & 非 Index 46 & oversimple \\
\hline ．byte 1 & 18 & 非 Index 47 & oversimplification \\
\hline ．byte 1 & 19 & 非 Index 48 & oversimplifications \\
\hline ．byte 1 & 14 & 非 Index 49 & oversimplified \\
\hline ．byte 1 & 12 & 非 Index 50 & weightlessly \\
\hline ．byte 1 & 14 & 非 Index 51 & weightlessness \\
\hline ．byte 1 & 12 & Index 52 & weightlifter \\
\hline ．byte & 9 & 非 Index 53 & zucchinis \\
\hline ．byte & 8 & 非 Index 54 & zwieback \\
\hline ．byte & 6 & 非 Index 55 & zydeco \\
\hline ．byte & 6 & 非 Index 56 & zygote \\
\hline ．byte & 7 & 非 Index 57 & zygotes \\
\hline
\end{tabular}
.byte 5 非 Index 36 fishy
.byte 7 非 Index 37 fissile
.byte 7 非 Index 38 fission
.byte 11 非 Index 39 fissionable
.byte 7 非 Index 40 fissure
.byte 1 非 Index 41 i
.byte 5 非 Index 42 mucky
.byte 6 非 Index 43 mucous
.byte 5 非 Index 44 mucus
.byte 3 非 Index 45 mud
.byte 10 非 Index 46 oversimple
.byte 18 非 Index 47 oversimplification
.byte 19 非 Index 48 oversimplifications
.byte 14 非 Index 49 oversimplified
.byte 12 非 Index 50 weightlessly
.byte 14 非 Index 51 weightlessness
.byte 12 非 Index 52 weightlifter
.byte 9 非 Index 53 zucchinis
.byte 8 非 Index 54 zwieback
.byte 6 非 Index 55 zydeco
- Index 56
.byte 7 非 Index 57 zygotes

\section*{非 Lookup Words}
test＿words：
```

.asciiz "a" 非 Index -1
.asciiz "aah" 非 Index 0
.asciiz "able" 非 Index 2
.asciiz "i" 非 Index 41
.asciiz "blank" 非 Index -1
.asciiz "counselor" 非 Index -1
.asciiz "county" 非 Index -1
.asciiz "fish" 非 Index -1
.asciiz "gram" 非 Index -1
.asciiz "palindromic" 非 Index -1
.asciiz "zymurgy" 非 Index -1
.asciiz "bibliographical" 非 Index 13
.asciiz "bibliographically" 非 Index 14
.asciiz "cross" 非 Index 24
.asciiz "zydeco" 非 Index 55
.asciiz "zygotes" 非 Index 57

```
．align 4
test＿data：
```

.half 0, -1 非 a
.half 2, 0 非 aah
.half 6, 2 非 able
.half 11, 41 非 i
.half 13, -1 非 blank
.half 19, -1 非 counselor
.half 29, -1 非 county
.half 36, -1 非 fish
.half 41, -1 非 gram
.half 46, -1 非 palindromic
.half 58, -1 非 zymurgy

```
```

    .half 66, 13 非 bibliographical
    .half 82, 14 非 bibliographically
    .half 100, 24 非 cross
    .half 106, 55 非 zydeco
    .half 113, 57 非 zygotes
    ```
test_data_end:
msg_correct:
    .asciiz "\%/s0/-17s Num Insn: \%/t1/5d Index: \%/v1/3d -- Correct\n"
msg_incorrect:
    .asciiz "\%/s0/-17s Num Insn: \%/t1/5d Index: \%/v1/3d Er SHOULD BE \%/t2/d\n"
end_msg:
    .asciiz "TOTALS: Num Insn: \%/k0/5d Tests: \%/k1/3d Errors: \%/s1/2d\n"
    .text
    .globl __start
__start:
    \(m t c 0\) \$0, \$22 非 Pause tracing.
    非 s0: Address of test word.
    addi \$s1, \$0, \(0 \quad\) 非 Number of incorrect indices.
    la \$s2, test_data_end 非 Holds same value.
    非 s5: Number of instructions before.
    la \$s6, test_words 非 Holds same value.
    la \$s7, test_data 非 Incremented each iteration.
    addi \$k0, \$0, 0 非 Total number of instructions.
    addi \$k1, \$0, 0 非 Total number of lookup words.
    非 Number correct.
TB_LOOP:
    lh \$t0, 0(\$s7) 非 Offset in word table.
    addi \$k1, \$k1, 1
    add \$s0, \$s6, \$t0
    add \$a0, \$s0, \$0
    la \$a1, table_words
    la \$a2, table_word_lengths
    addi \$v0, \$0, -1
    mtc0 \$v0, \$22 非 Resume tracing. (No effect if not stepping.)
    jal lookup
    mfc0 \$s5, \$9 非 Copy current instruction count. (Before.)
    mfc0 \$t0, \$9 非 Copy current instruction count. (After.)
    mtc0 \$0, \$22 非 Pause tracing.
    addi \$v1, \$v0, 0
    sub \$t1, \$t0, \$s5
    add \$k0, \$k0, \$t1
    1h \$t2, 2(\$s7) 非 Correct index.
    la \$a0, msg_correct
    beq \$t2, \$v1, TB_SKIP
    nop
    addi \$s1, \$s1, 1
    la \$a0, msg_incorrect
TB_SKIP:
    addi \$v0, \$0, 11
    syscall
addi \$s7, \$s7, 4
bne \$s7, \$s2, TB_LOOP
nop
Ia \$a0, end_msg addi \$v0, \$0, 11 syscall
addi \$v0, \$0, 10
syscall
nop

Problem 1: The code fragment below was taken from the course hex string assembly example. (The hex string example was not covered this semester. The full example can be found at https://www.ece.lsu.edu/ee4720/2022/hex-string.s.html.) The fragment below converts the value in register a0 to an ASCII string, the string is the value in hexadecimal (though initially backward).

LOOP:
```

andi \$t0, \$a0, Oxf \# Retrieve the least-significant hex digit.
srl \$a0, \$a0, 4 \# Shift over by one hex digit.
slti \$t1, \$t0, 10 \# Check whether the digit is in range 0-9
bne \$t1, \$0, SKIP \# Don't forget that delay slot insn always exec.
addi \$t2, \$t0, 48 \# If 0-9, add 48 to make ASCII '0' - '9'.
addi \$t2, \$t0, 87 \# If 10-15, add 87 to make ASCII 'a' - 'z'.

```
```

sb $t2, 0($a1) \# Store the digit.

```
sb $t2, 0($a1) # Store the digit.
bne $a0, $0, LOOP # Continue if value not yet zero.
bne $a0, $0, LOOP # Continue if value not yet zero.
addi $a1, $a1, 1 # Move string pointer one character to the left.
```

addi \$a1, \$a1, 1 \# Move string pointer one character to the left.

```
SKIP:
(a) Show the encoding of the MIPS bne a0, 0, LOOP instruction. Note: This is not the same as the instruction used in last year's Homework 2. Include all parts, including - especially - the immediate. For a quick review of MIPS, including the register numbers corresponding to the register names, visit https://www.ece.lsu.edu/ee4720/2023/lmips.s.html.

The encoding appears below. Note that register a0 (procedure call argument 0 ) is a helpful name for \(r 4\), and so a 4 is placed in the rs field. If the bne is taken it jumps backward by eight instructions (starting from the delay-slot instruction), and so the immediate filld holds a -8 (which is \(1111111111111000_{2}\) in a two's complement, 16-bit representation).

(b) RISC-V RV32I has a bne instruction too, though it is not exactly the same. Show the encoding of the RV32I version of the bne a0, 0, LOOP instruction. For this subproblem assume that the bne will jump backward eight instructions, just as it does in the code sample above.

To familiarize yourself with RISC-V start by reading Chapter 1 of Volume I of the RISC-V specification, especially the Chapter 1 Introduction and Sections 1.1 and 1.3. Skip Section 1.2 unless you are comfortable with operating system and virtualization concepts. Other parts of Chapter 1 are interesting but less relevant for this problem. Also look at Section 2.5 (Control Transfer Instructions). The spec can be found in the class references page at https://www.ece.lsu.edu/ee4720/reference.html.

The branch instructions are discussed in Section 2.5 under the Conditional Branches heading. There are two significant differences with the MIPS bne. First, there is no delay slot. That's not relevant in this problem. Second, the immediate field value is used differently. Let IMM denote the immediate field value (based on the bits set in the instruction). The braneh target is then PC \(+2 *\) IMM, where PC is the address of the branch. (In MIPS the target would be PC \(+4+4 *\) IMM.) So, we need to set the immediate to the number of bytes to skip divided by two. The problem says to jump back eight instructions, in RISC-V (and most RISC ISAs) that's 32 bytes, and so the immediate field should be set to -16 which is \(111111110000_{2}\) in a two's complement, 12-bit representation.

Though Section 2.5 shows the encoding of a bne instruction, it does not provide the values for opcode filds and their extensions, instead using names: \(B R A N C H\) for the opcode field and \(B N E\) for the funct 3 fild. The values can be found in Chapter 24 .

The encoding appears below. The instruction field names have been abbreviated, such as im12 for imm [12]. Also note that in RISC-V immediate fild names use the bit numbers within the immediate. So, for example, imm [4:1] (abbreviated im4:1 below) indicates that the four bits in the instruction field ( \(0000_{2}\) in the example) are placed in bit positions \(4: 1\) of the immediate. There is no field named \(\operatorname{imm}[0]\) because the corresponding immediate bit is always set to zero. So, the IMM \(* 2\) is computed by putting the twelve immediate bits in the instruction in bits \(12: 1\) of the immediate, setting bit 0 , the LSB, to zero.

In MIPS format I-instructions the 16-bit immediate is put in bits 15:0 of the instruction, which is straightforward and easy to understand. In RISC-V B-format instructions the 12-bit immediate is scrambled into four fields of the instruction. Following the convention of the B -format instruction the immediate bit positions will be numbered 12 to 1 . (There is a bit position zero but it is always zero and so it does not appear in the instruction.) Bit 12 of the immediate is found in bit position 31 of the instruction. Bits \(10: 5\) [sic] of the immediate are in bits \(30: 25\) of the instruction. Where is bit 11 of the immediate? It's at bit 7 , hanging out next to bits \(4: 1\) of the immediate which are at bits \(11: 8\) of the instruction. This bit scrambling is done to simplify hardware, as is explained in section 2.3 of the RISC-V standard. A question about the rationale for this bit scrambling may be asked on the 2023 midterm exam.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline & & 0:5 & rs2 & & rs1 & & & 4:1 & & & \\
\hline \multirow[t]{2}{*}{RISC-V B:} & \(1_{2}\) & \(111111_{2}\) & & 0 & & 4 & 0012 & \(0000_{2}\) & \(1_{2}\) & & \multirow[t]{2}{*}{} \\
\hline & 31 & \(30 \quad 25\) & 524 & \multicolumn{2}{|l|}{2019} & 1514 & 12 & 118 & \multicolumn{2}{|l|}{76} & \\
\hline
\end{tabular}
(c) Consider the four-instruction sequence from the code above:
```

slti \$t1, \$t0, 10 \# Check whether the digit is in range 0-9
bne \$t1, \$0, SKIP \# Don't forget that delay slot insn always exec.
addi \$t2, \$t0, 48 \# If 0-9, add 48 to make ASCII '0' - '9'.
addi \$t2, \$t0, 87 \# If 10-15, add 87 to make ASCII 'a' - 'z'.

```

SKIP:
Re-write this sequence in RISC-V RV32I, and take advantage of RISC-V branch behavior to reduce this to three instructions (plus possibly one more instruction before the loop). For this problem one needs to focus on RISC-V branch behavior. Assume that the RISC-V slti and addi instructions are identical to their MIPS counterparts at the assembly language level. It is okay to retain the MIPS register names. Hint: One change needs to be made for correctness, another for efficiency.

Solution appears below. Before the loop is entered the addi instruction sets t6 to 10, a constant that will come in handy. Inside the loop the four MIPS instructions are replaced by three RISC-V instructions. First, the slti is no longer necessary because RISC-V has a blt (branch less than). The blt itself checks whether to is less than 10 (which is in t6). Using the blt to do the comparison is the efficiency change mentioned in the hint. Because RISC-V lacks delay slots the addi \(\mathrm{t} 2, \mathrm{t} 0,48\) had to be moved before the blt. That's the correctness change mentioned in the hint. Notice that RISC-V has register names that are similar to MIPS, such as t0-t6 for caller-save (temporary) registers.
```


# Instruction inserted before the loop to put 10 into register t6.

addi t6, zero, 10

```
```


# These instructions replace the four MIPS instructions.

```
# These instructions replace the four MIPS instructions.
addi t2, t0, 48 # Convert assuming t0 in range 0-9 ..
addi t2, t0, 48 # Convert assuming t0 in range 0-9 ..
blt t0, t6, SKIP # .. if that's correct, branch ..
blt t0, t6, SKIP # .. if that's correct, branch ..
addi t2, t0, 87 # .. otherwise convert assuming a-f.
```

addi t2, t0, 87 \# .. otherwise convert assuming a-f.

```
LOOP:

SKIP:

Problem 2: Note: The following problem was assigned in each of the last six years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```


# Cycle

0
lw r2, O(r4)
IF ID EX ME WB
add r1, r2, r7
IF ID EX ME WB

```

The add depends on the 1 w through r2, and for the illustrated implementation the add has to stall in ID until the 1 w reaches WB.
```


# Cycle

0
SOLUTION
lw r2, 0(r4)
IF ID EX ME WB
add r1, r2, r7
IF ID ----> EX ME WB

```
(b) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4)
IF ID -> EX ME WB

```

There is no need for a stall because the 1 w writes r 1 , it does not read \(r 1\).
```


# Cycle 0

add r1, r2, r3
lw r1, O(r4) IF ID EX ME WB

```
(c) Explain error and show correct execution.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
IF ID -> EX ME WB

```

A longer stall is needed here because the sw reads \(r 1\) and it must wait until add reaches WB.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
IF ID ----> EX ME WB

```
(d) Explain error and show correct execution.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB

```

The stall above allows the xor, when it is in ID, to get the value of \(r 1\) written by the add; that part is correct. But, the stall starts in cycle 1 before the xor reaches \(I D\), so how could the control logic know that the xor needed r1, or for that matter that it was an xor? The solution is to start the stall in cycle 2, when the xor is in ID.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ID ----> EX ME WB

```

Problem 1: Appearing below are incorrect executions on the illustrated implementation. For each execution explain why it is wrong and show the correct execution. Note: This problem was assigned in 2020, 2021, and 2022, and their solutions are available. DO NOT look at the solutions unless you are lost and can't get help elsewhere. Even in that case just glimpse.

(a) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3
IF ID EX ME WB
xor r4, r1, r5
IF ID -> EX ME WB

```

There is a bypass path available so that there is no need to stall.
```


# Cycle

0
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ID EX ME WB

```
(b) The execution of the branch below has two errors. One error is due to improper handling of the andi instruction. (That is, if the andi were replaced with a nop there would be no problem in the execution below.) The other is due to the way the beq executes. As in all code fragments in this problem, the program is correct, the only problem is with the illustrated execution timing.
```


# Cycle: 0

andi r2, r2, 0xff IF ID EX ME WB
beq r1, r2, TARG IF ID EX ME WB
add r3, r4, r5 IF ID EX ME WB
xor
IFx
TARG:
sw r6, 7(r8) IF ID EX ME WB

# Cycle: 0

```

Briefly, the two problems are the lack of a stall for the andi/beq dependence carried by \(r 2\) and because the branch target is fetched one cycle later than it should be. The correct execution appears below.

Detailed explanation: In the illustrated implementation the \(\triangle\) In ID is used to compute the branch condition for beq (and bne). When the branch reaches \(I D\), in cycle 2 , the value of \(r 2\) retrieved from the register file is outdated, it needs to use the value computed by andi. Since there are no bypass paths to the \(=\) logic the branch will need to stall until andi reaches writeback. The stalls occur in cycles 2 and 3.

The illustrated implementation resolves the branch in \(I D\), and \(S 0\) the branch target should be in \(I F\) when the branch is in EX. In the execution above the target isn't fetched until the branch is in ME , in cycle 4 . That is fixed below by fetching the target a cycle earlier. The xor is no longer fetched and squashed.
```


# Cycle: 0

andi r2, r2, Oxff IF ID EX ME WB
beq r1, r2, TARG IF ID ----> EX ME WB
add r3, r4, r5 IF ----> ID EX ME WB
xor
TARG:
sw r6, 7(r8) IF ID EX ME WB
\# Cycle: 0

```

(c) Explain error and show correct execution.
```


# Cycle

0
lw r2, O(r4)
add r1, r2, r7
IF ID EX ME WB
IF ID EX ME WB

```

The add depends on the 1 w through r2, and for the illustrated implementation the add has to stall in ID until the 1 w reaches ME so that the add can bypass from WB.
```


# Cycle

0
lw r2, O(r4)
add r1, r2, r7
IF ID EX ME WB
IF ID -> EX ME WB

```
(d) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3
lw r1, O(r4)
IF ID EX ME WB
IF ID -> EX ME WB

```

There is no need for a stall because \(r 1\) is not a source register of 1 w . Note that r 1 is a destination of 1 w .
```


# Cycle

0
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID EX ME WB

```
(e) Explain error and show correct execution.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID -> EX ME WB

```

No stall is needed here because the sw can use the ME-to-EX bypass path.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
IF ID EX ME WB

```

Problem 2: Illustrated below is a MIPS implementation in which each multiplexor has a label, such as a circled A at the multiplexor providing a value for the PC. (The implementation debuted on the 2018 midterm exam.) The multiplexor inputs are also numbered. Below the illustration an execution of the program on the implementation is shown for two iterations of a loop. Below the execution is a table with one row for each labeled multiplexor. Complete the table so that it shows the values on the multiplexors' select signals at each cycle based on the execution. Leave an entry blank if its value does not make a difference.

Wire thicknesses and colors have been varied to make it easier to trace them through the diagram. Before attempting this problem, solve 2018 Midterm Exam Problem 2b, which also appeared as 2022 Homework 3 Problem 2. Also see the 2014 Midterm Exam Problem 1 for a similar problem.


Continued on the next page.
\(\checkmark\) Complete the table (the rows starting with A:, B: etc.) based on the execution below.
\(\checkmark\) Omit select signal values if they do not matter. For example, omit values for E for cycles in which there is not a store instruction in EX.
\(\checkmark\) Assume that the branch is taken the second time it appears. (No assumption needed for its first appearance.)
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline addi r1, r1, -4 & IF & ID & EX & ME & WB & & & & & & & & \\
\hline LOOP: \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & & & \\
\hline sw r2, 4(r1) & & IF & ID & EX & ME & WB & & & & & & & \\
\hline lw r1, 8(r2) & & & IF & ID & EX & ME & WB & & & & & & \\
\hline bne r2, r3, LOOP & & & & IF & ID & EX & ME & WB & & & & & \\
\hline add r2, r2, r6 & & & & & IF & ID & EX & ME & WB & & & & \\
\hline LOOP: \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & & & \\
\hline sw r2, 4(r1) & & & & & & IF & ID & EX & ME & WB & & & \\
\hline lw r1, 8(r2) & & & & & & & IF & ID & EX & ME & WB & & \\
\hline bne r2, r3, LOOP & & & & & & & & IF & ID & EX & ME & WB & \\
\hline add r2, r2, r6 & & & & & & & & & IF & ID & EX & ME & WB \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline \# SOLUTION & & & & & & & & & & & & & \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline A : & 3 & 3 & 3 & 3 & 1 & 3 & 3 & 3 & 1 & & & & \\
\hline B : & & & & & & & & & & & & & \\
\hline LOOP: \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline C: & & & 2 & 0 & 2 & & 2 & 2 & 3 & & 2 & & \\
\hline D: & & & 2 & 2 & 2 & & 1 & 2 & 2 & & 1 & & \\
\hline LOOP: \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline E: & & & & 1 & & & & 0 & & & & & \\
\hline F: & & & & & 1 & & 0 & & 1 & & 0 & & 1 \\
\hline LOOP: \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
\end{tabular}

Problem 3: Show the execution of the code fragments on the following implementations for enough iterations to determine the instruction throughput (IPC). As always, base the behavior of branches and the availability of bypasses on the implementations. Also, don't forget that MIPS branches have a delay slot. Sorry for yelling, but I hate it when students miss things.

This problem appeared as most of Problem 1 on the 2022 Final Exam. A solution is not yet available.


Show execution and determine instruction throughput (IPC) based on a large number of iterations.
The solution appears below. The branch is resolved in \(\mathrm{ME}_{\text {, }}\) and so the target is fetched (in IF) in the next cycle, when the branch is in WB. Two wrong-path instructions are fetched, xor and sub. They are squashed when the branch is resolved. (Of course, they would not be squashed if the branch were not taken.)

The instruction throughput is \(\frac{2 \mathrm{insn}}{(8-4) \text { cyc }}=\frac{2}{4} \mathrm{insn} /\) cycle based on the second iteration starting at cycle 4 and the third iteration starting at cycle 8.
```

    # SOLUTION -- Dynamic Instruction Order
    LOOP: \# Cycle 0
bne r1, r2, LOOP IF ID EX ME WB \# First Iteration
addi r1, r1, 4 IF ID EX ME WB
xor r5, r6, r7 IF IDx
sub r8, r9, r10 IFx
LOOP: \# Cycle 0
bne r1, r2, LOOP IF ID EX ME WB \# Second Iteration
addi r1, r1, 4 IF ID EX ME WB
xor r5, r6, r7 IF IDx
sub r8, r9, r10 IFx
LOOP: \# Cycle }0
bne r1, r2, LOOP IF ID EX ME WB

# These instructions will be completely executed after the last iteration.

    xor r5, r6, r7
    sub r8, r9, r10
    ```


Show execution and \(\forall\) determine instruction throughput (IPC) based on a large number of iterations.
The solution appears below. The good news in this pipeline the branch is resolved in ID, meaning that zero wrong-path instructions are fetched. The bad news is that there is a dependence carried by r1 that stalls bne in ID for two cycles. For this reason, the instruction throughput is the same: \(\frac{2 \mathrm{insn}}{(6-2) \text { cyc }}=\frac{2}{4} \mathrm{insn} /\) cycle based on the second iteration starting at cycle 2 and the third iteration starting at cycle 6.
```

LOOP: \# Code in Static Instruction Order
bne r1, r2, LOOP
addi r1, r1, 4
xor r5, r6, r7
sub r8, r9, r10

```
```

    # SOLUTION -- Dynamic Instruction Order
    LOOP: \# Cycle (0 1
bne r1, r2, LOOP IF ID EX ME WB \# First Iteration
addi r1, r1, 4 IF ID EX ME WB
LOOP: \# Cycle 0}0
bne r1, r2, LOOP IF ID ----> EX ME WB \# Second Iteration
addi r1, r1, 4 IF ----> ID EX ME WB
LOOP: \# Cycle }0
bne r1, r2, LOOP IF ID ----> EX ME WB
addi r1, r1, 4
IF ----> ID EX ME WB

```
    \# These instructions will be executed after the last iteration.
    xor r5, r6, r7
    sub r8, r9, r10


Show execution and determine instruction throughput (IPC) based on a large number of iterations.
In this implementation there is a bypass that helps with the branch condition dependence, reducing the stall from two cycles to one cycle. The instruction throughput is higher, \(\frac{2 \mathrm{insn}}{(5-2) \mathrm{cyc}}=\frac{2}{3} \mathrm{insn} /\) cycle based on the second iteration starting at cycle 2 and the third iteration starting at cycle 5.
```

LOOP: \# Code in Static Instruction Order
bne r1, r2, LOOP
addi r1, r1, 4
xor r5, r6, r7
sub r8, r9, r10
\# SOLUTION -- Dynamic Instruction Order
LOOP: \# Cycle }0
bne r1, r2, LOOP IF ID EX ME WB \# First Iteration
addi r1, r1, 4 IF ID EX ME WB
LOOP: \# Cycle 0}0
bne r1, r2, LOOP IF ID -> EX ME WB \# Second Iteration
addi r1, r1, 4 IF -> ID EX ME WB
LOOP: \# Cycle 0
bne r1, r2, LOOP IF ID -> EX ME WB
addi r1, r1, 4 IF -> ID EX ME WB

```

In the problems below a new MIPS instruction, integer fmadd, (hypothetical of course) is to be added to our pipelined MIPS implementation. A simpler implement-the-instruction problem was the subject of Fall 2010 Homework 3, in which a shift unit is added to MIPS to implement shift instructions. The 2010 problem is simpler because the shift unit occupies just one stage, while the fmadd for this assignment spans multiple stages. For past assignments in which integer arithmetic hardware spans several stages see 2020 Homework 2, 3, and 4 and 2020 midterm exam Problem 5. In these 2020 problems an integer multiply instruction was to be implemented.

Problem 1: A fused multiply/add instruction, such as fmadd r1, r2, r3, r4, computes \(r_{1}=r_{2} r_{3}+r_{4}\). Such instructions are useful for both floating-point and integer calculations, and integer version is considered here. The goal in this problem is to extend MIPS with an integer multiply/add instruction, fmadd. The new fmadd instruction will be encoded in MIPS Format R with the SA field being used to specify the third source register, r4 in the example.


The hardware to compute the multiply/add will consist of two types of units: a carry-save multiplier (CSM) and integer adders (labeled ADD). The connection of these two types of units needed to compute a multiply/add are shown below.


The CSM takes 0.9 clock cycles to compute a result and each adder takes 0.45 clock cycles, so the critical path through the hardware shown above is 1.8 clock cycles. Because the critical path is greater than one clock cycle the hardware cannot be placed in one stage. (Unless the clock frequency were to be decreased from \(\phi\) to \(\phi / 1.8\), which would slow everything down and so of course we don't want to do it.)

Note: For the three parts below a single hardware solution can be provided. That is, a correct solution to part \(c\) also can be a correct solution to parts \(b\) and \(a\), and so there is no need to draw three hardware designs.
(a) Add the CSM and ADD units to the MIPS implementation below to efficiently implement the fmadd instruction. For this sub-problem provide the hardware needed so that fmadd can execute without stalls when there are no nearby dependencies, such as in the execution below.
```


# There are no dependencies in this code fragment.

# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sub r4, r5, r6 IF ID EX ME WB
fmadd r7, r8, r9, r10 IF ID EX ME WB
fmadd r11, r12, r13, r14 IF ID EX ME WB
xori r15, r16, 17 IF ID EX ME WB

# Cycle 0

```

Put your solution on the larger diagram several pages ahead.


Put your solution on the larger diagram several pages ahead.
Add the CSM and ADD units to the implementation above so that the can implement the fmadd instruction.
Provide the datapath needed so that operands can reach the CSM and ADD units and \(\forall\) the result can reach the register file.
\(\checkmark\) Don't forget that this instruction has three source operands.
Do not increase the critical path.
As always, consider cost. Assume that an \(n\)-bit register costs twice as much as an \(n\)-bit, 2 -input multiplexor. fmadd should execute without stalls when there are no nearby dependencies.
\(\checkmark\) Do not design control logic for this assignment.
The solution and its discussion appear several pages ahead.
(b) In the code fragments below the fmadd depends on prior instructions.
\(\checkmark\) Add bypass paths to the fmadd implementation so that all of the executions below are possible.
The solution and its discussion appear on the next page.
```


# Fragment A

# Cycle 0

add R1, r2, r3 IF ID EX ME WB
sub R4, r5, r6 IF ID EX ME WB
fmadd r7, R1, R4, r9 IF ID EX ME WB

# Fragment B

# Cycle 0

sub R9, r5, r6
fmadd R7, r1, r4, R9
fmadd r2, r3, r5, R7
0

# Fragment C

# Cycle 0

add R1, r2, r3 IF ID EX ME WB
lw R9, O(r10) IF ID EX ME WB
fmadd r7, R1, r4, R9 IF ID EX ME WB

```
(c) Using additional ADD unit(s) modify the implementation so that it can execute Fragments L and D correctly. This will require some tricky bypassing. Note that stalls will be needed when the dependent instruction following the fmadd does not use the adder, such as in Fragment E. Note: In the original problem just one adder was to be used. That is probably impossible without critical path impact.
\(\checkmark\) Add a second adder and bypass paths so that fragments \(L\) and \(D\) execute as shown.
The solution and its discussion appear on the next page.
```


# Fragment L

# Cycle 0

fmadd R7, r1, r4, r9 IF ID EX ME WB
lw r10, 16(R7) IF ID EX ME WB \# No stall!

# Fragment D

# Cycle 0

fmadd R7, r1, r4, r9 IF ID EX ME WB
add r2, R7, r3 IF ID EX ME WB \# No stall!

# Fragment E

# Cycle 0

fmadd R7, r1, r4, r9 IF ID EX ME WB
or r2, R7, r3 IF ID -> EX ME WB \# A stall. :-(

```

Use the diagram below for your solution, or download
https://www.ece.lsu.edu/ee4720/2023/mpipei3.svg and edit with your favorite SVG editor. (The diagram was drawn with Inkscape.)

Solution appears below. The changes for parts a and \(b\) are shown in blue and the changes for part c are shown in green.
For part \(a\), the most important thing was not to put CSM and an ADD in the same stage, because the delay of CSM already used most of the clock cycle. Notice that CSM's inputs are obtained from the ALU's multiplexors, so that for the multiplier and multiplicand operands of fmadd no further changes are needed for bypassing.

The fmadd instruction uses a third source operand, sa. For this operand a third read port is added to the register file (in ID). In EX the sa value uses the rtv path to reach ME , saving the need for an sav pipeline latch between EX and ME . The output of the second ADD is the result of the fmadd instruction, it is connected to a new mux at the top of ME where it joins the path leading to the register file input.

The changes described above provide the bypass paths needed for Fragment A. For fragments B and C a bypass is needed for the sa value, for example R9 in Fragment C. That bypass is provided by the mux in the lower part of ME. Other bypasses needed for the sa value are provided by the existing mux that provides a value for the ME.rtv pipeline latch.

For part e two ADD units are used, these are shown in green. (The original problem was to use just one, which is probably impossible without critical path impact.)


Problem 1: In this problem consider the encoding of integer and floating-point addition instructions in MIPS and RISC-V. Descriptions of MIPS and RISC-V are linked to the course references page, https://www.ece.lsu.edu/ee4720/reference.html.
(a) Show the encoding of MIPS instructions add r1, r2, r3 and add.s \(f 4, f 5, f 6\).

The encoding of add r1, r2, r3 is:
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline & Opcode & & rs & & rt & & rd & & Sa & & fun & \\
\hline \multirow[t]{2}{*}{MIPS} & & 0 & & 2 & & 3 & & 1 & & 0 & & 100000 \\
\hline & 31 & 26 & 25 & 21 & 20 & 16 & & & 10 & & & 0 \\
\hline
\end{tabular}

Instruction add.s \(\mathrm{f4}, \mathrm{f5}, \mathrm{f} 6\) is floating-point and in MIPS its format differs significantly from the integer instructions. (In many ISAs FP and integer instructions are encoded differently.) The register number fields are in different positions, and there is a format field, fmt, for specifying whether the operands are single- or double-precision. (The values for the fmt field can be found in Volume I Appendix A in Table A-11, in which the fmt field is confusingly called the rs field.)

(b) Show the encoding of RISC-V RV32IF instructions add x 7 , x 8 , x 9 and fadd \(f 10, f 11\), \(f 12\).

The encoding for add \(\mathrm{x} 7, \mathrm{x} 8, \mathrm{x} 9\) appears below. The encoding is most conveniently found in Chapter \(24, \mathrm{RV} 32 / 64 \mathrm{G}\) Instruction Set Listing.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline & \multicolumn{2}{|r|}{rs2} & \multicolumn{2}{|r|}{rs1} & \multicolumn{2}{|c|}{fun3} & rd & \multicolumn{2}{|r|}{opcode} \\
\hline RISC-V R: & 0 & & 9 & & 8 & 0002 & & 7 & \(\mathrm{OP}=0110011_{2}\) \\
\hline & & 24 & 20 & 19 & 15 & 12 & 11 & & 0 \\
\hline
\end{tabular}

The encoding for fadd \(f 10, f 11, f 12\) appears below. The encoding is most conveniently found in Chapter 24, RV32/64G Instruction Set Listing, in the table for RV32F. The problem did not specify which rounding mode (rm)to use, the encoding below uses round-to-nearest, ties to even.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline & func5 & & \multicolumn{2}{|r|}{rs2} & \multicolumn{2}{|r|}{rs1} & \multicolumn{2}{|r|}{rm} & \multicolumn{2}{|r|}{rd} & \multicolumn{3}{|r|}{opcode} \\
\hline RISC-V R: & & 0 & \(00_{2}\) & & 12 & & 11 & & 0002 & & 10 & & OP-FP=10100112 \\
\hline & 31 & & & 24 & 20 & 19 & 15 & & 12 & 11 & & 6 & 0 \\
\hline
\end{tabular}
(c) Notice that the register fields in the integer and floating-point RISC-V RV32IF instructions are the same, while the register fields in the two MIPS instructions are different. One possible reason for RISC-V's matching fields was to simplify implementations of the Zfinx variant. (Web search for it.) How do the matching fields reduce the cost of implementations of the RISC-V Zfinx variant?

In the Zfinx variant there is no separate floating-point register file. Instead, both integer and floating-point instructions' operands are from the same register file, the integer register file. Because integer and FP instructions operand fields are in the same place the connections from needed to retrieve integer instruction operands will also work for floating-point instruction operands.

\section*{ARM A64 Background}

The following background will help in solving the next problem. MIPS and many other ISAs have a set of integer registers and a set of floating-point registers. Many newer ISAs, including ARM A64, have a set of vector registers in lieu of floating-point registers. Extensions of legacy ISAs, such as Intel 64 AVX2, have vector registers but retain floating-point registers for compatibility.

In many ISAs, including ARM A64, a vector register can be used to hold one FP value, just as a traditional FP register would, or a vector register can hold several values. Scalar instructions read or write one value per vector register, and vector instructions read and write multiple values per register.

In ARM A64 there are 32 128-bit vector registers, named v0 to v31. When used in scalar instructions operating on single-precision FP values they are known by the names \(\mathbf{s} 0\) to \(s 31\) and by the names d0 to d31 by double-precision scalar instructions. For example, the ARM A64 assembler instruction fadd s0, s1, \(s 2\) computes \(s 0=s 1+s 2\) and fadd \(d 0\), \(d 1\), \(d 2\) computes \(d 0=d 1+d 2\). In both cases the operands were taken from vector registers v0, v1, and v2. The assembler name s0 means use the low 32 bits of v0 and interpret the value as an IEEE 754 single. The assembler name d0 means use the low 64 bits of v0 and interpret the value as an IEEE 754 double.

In the next problem, the one with sum_thing_unusual, the ARM code contains only scalar floating point instructions and base (integer register) instructions. To solve the next problem one needs to look up instructions in the ARM Architecture Reference Manual. Instructions that operate on vector registers, including fadd can be found in the Advanced SIMD and Floating Point section, C7.2 for the list of instructions. Other instructions can be found in the A64 Base Instruction section, C6.2 for the list of instructions.

Vector instructions are not needed in this assignment, but they will be briefly described anyway. For vector instructions the vector register name indicates how many elements in the vector to use, and what their format is. For example, v0.4s, means to use vector register v0 and to split its 128 bits into 432 -bit lanes, with each lane holding one float (the s). The names can be used in instructions such as fadd v0.4s, \(\mathrm{v} 0.4 \mathrm{~s}, \mathrm{v} 1.4 \mathrm{~s}\). This instruction performs four additions, one on each lane of the vector register.

Problem 2: Appearing below is a \(\mathrm{C}++\) procedure with a for loop that computes the sum of elements in an array. This would be a totally ordinary loop were it not for the fact that the iteration variable, i, and the increment, delta, are both floats. Since i is a float the number of iterations, depending on delta, can be less than 1024 (say, if delta=2.3) or more than 1024 (say, if delta=0.25). Below the C code are MIPS-I and ARM A64 assembler versions of the loop. Yes, that means you don't have to write them! (The MIPS-I code was hand written, and the A64 was based on code generated by a compiler.) Notice that the ARM code is shorter than the MIPS code. That's because some of the ARM instructions do the equivalent of several MIPS instructions.

Next to each ARM instruction indicate the MIPS instruction(s) from the MIPS code that it corresponds to.
When an ARM instruction corresponds to more than one MIPS instruction explain what the ARM instruction is doing.

A short reference for MIPS floating-point instructions is the course lfp.s notes. This should be sufficient for all but the MIPS-II trunc instruction. For the trunc instruction see the MIPS documentation (linked to the course ISA page).
```

float sum_thing_unusual( float *a, float delta ) {
float sum = 0;
for ( float i = 0; i < 1024; i += delta ) sum += a[int(i)];
return sum;
}

```
```


# MIPS Code for sum_thing_unusual.

```
# MIPS Code for sum_thing_unusual.
#
#
# $a0: The address of array a.
# $a0: The address of array a.
# $f0: i. At this point it contains a zero.
# $f0: i. At this point it contains a zero.
# $f4: delta.
# $f4: delta.
# $f5: The constant 1024, in FP format.
# $f5: The constant 1024, in FP format.
# $f8: sum. At this point it contains a zero.
# $f8: sum. At this point it contains a zero.
LOOP:
trunc.w.s $f6, $f0 # Note: This is a MIPS-II instruction.
trunc.w.s $f6, $f0 # Note: This is a MIPS-II instruction.
mfc1 $t1, $f6
mfc1 $t1, $f6
sll $t1, $t1, 2
sll $t1, $t1, 2
add $t2, $t1, $a0
add $t2, $t1, $a0
lwc1 $f7, 0($t2)
lwc1 $f7, 0($t2)
add.s $f0, $f0, $f4
add.s $f0, $f0, $f4
c.lt.s $f0, $f5
c.lt.s $f0, $f5
bc1f LOOP
bc1f LOOP
add.s $f8, $f8, $f7
```

add.s \$f8, \$f8, \$f7

```

The ARM instruction use is described below. Unlike MIPS, the ARM64 conversion instruction converts from FP to integer format, and writes the result to the integer register file. The ARM memory instructions, including ldr, can do more to compute an address than MIPS load instructions. They not only can add two registers, but also shift one of the registers, reducing the amount of code needed for array access.
```

.arch arm
@ ARM A64 Code for sum_thing_unusual.
@
@ x0-x31: Integer registers. x31 is sometimes the zero register.
@ s0-s31: Scalar single-precision floating-point registers.
@
@ x0: The address of array a.
@ s0: sum. At this point it contains a zero.
@ s1: i. At this point it contains a zero.
@ s3: delta.
@ s4: The contains 1024, in FP format.
@@ SOLUTION
LOOP:

```
```

fcvtzs x1, s1
@ MIPS trunc. and mfc1.
@ Convert FP in s1 to an integer, with truncation and write
@ result in integer register x1.
fadd s1, s1, s3 @ MIPS add.s f0
fcmpe s1, s4
@ The closest equivalent instruction is MIPS c.lt.s.
@ But this instruction sets the comparison flags, NZCV, based
@ on the comparison. (The names N, Z, C, V are for the result
@ of an integer operation, but ARM uses the for FP comparisons too.)
ldr s2, [x0, x1, lsl 2]
@ MIPS sll, add, lwc1
@ ldr first computes an address by shifting x1 left two bits,
@ then adds the result to x0. The memory at that address (and
@ the three following) is read and the value placed in
@ register s2. Put another way:
@ s2 = Mem[ x0 + x1*4 ];
fadd s0, s0, s2 @ MIPS add.s f8
bmi LOOP @ MIPS bc1t and partly c.lt.s

```

Problem 1: Solve 2022 Final Exam Problem 2, in which code fragments are either written or analyzed for our MIPS FP implementation.

See the final exam solution at https://www.ece.lsu.edu/ee4720/2022/fe_sol.pdf.

Problem 2: Solve the last part of 2022 Final Exam Problem 1, the one with the 4 -way superscalar pipeline. (You can tell it's 4 -way because the superscripts range from 0 to 3.) There is no need to show superscripts on the stage labels in your execution diagram. For sample problems see past final exams, such as 2021 Problem 2.

See the final exam solution at https://www.ece.lsu.edu/ee4720/2022/fe_sol.pdf.

Problem 3: Appearing below is our MIPS FP implementation but with an unpipelined FP add unit. Some of the control logic needed to generate stalls when a FP add instruction is in flight is in the magic cloud labeled "Future HW Solution". Design that logic. For similar logic see the logic on the Partially Pipelined pages from Set 9 slides (about page 14). Hint: This does not require much hardware. For similar problems see 2020 Spring Homework 5 and 2020 Spring Final Exam Problem 2.

Use the execution below to help you design the hardware:
```


# Cycle 0

add.d f0, f2, f4 IF ID A A A A WF
add.d f6, f8, f10 IF ID -------> A A A A WF
addi r1, r1, 8 IF -------> ID EX ME WB
add.d f12, f18, f14 IF ID ----> A A A A WF

# Cycle }0

```

An SVG version of the image can be found at https://www.ece.lsu.edu/ee4720/2023/hw06-fp-aaaa.svg, use Inkscape or some other SVG editor, or even a text editor.

The solution appears below in blue. The OR gate now has three new inputs, which are the ad signals in the M3, M4, and M5 stages.


Problem 1: Solve 2022 Final Exam Problem 3, in which hardware is added to a variation on a 2-way superscalar MIPS implementation.

See the final exam solution at https://www.ece.lsu.edu/ee4720/2022/fe_sol.pdf.

\section*{40 Spring 2022 Solutions}

Homework 1 solution
Due: 4 February 2022

Problem 0: Follow the instructions for class account setup and for homework workflow in https://www.ece.lsu.edu/ee4720/proc.html. Review the comments in hw01.s and look for the areas labeled "Problem 1" and "Problem 2."

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2022/hw01.s.html. For MIPS references see the course references page,
https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in https://www.ece.lsu.edu/ee4720/proc.html.

This Assignment
In class as MIPS review we wrote a routine, strlen, to find the length of a C string. In our completed routine (shown below) the main loop consisted of three instructions, and would load one character per iteration. Therefore at best it could run at the rate of \(\frac{1}{3}\) characters per instructions.
```

strlen:

```
```


# Register Usage

# \$aO: Address of first character of string.

# \$vO: Return value, the length of the string.

addi \$v0, \$a0, 1 \# Set aside a copy of the string start + 1.
lbu $t0, 0($a0) \# Load next character in string into \$t0
bne \$t0, \$0, LOOP \# If it's not zero, continue
addi \$a0, \$a0, 1 \# Increment address. (Note: Delay slot insn.)
jr \$ra
sub \$v0, \$a0, \$v0

```
LOOP:

Can we do better? Since the main loop only consists of three instructions there is little that can be done to make it shorter, at least using MIPS I instructions. Notice that a character is loaded using lbu (load byte unsigned). Suppose instead a lw (load word) were used. Then four characters would be loaded. If our loop body contained 12 instructions (including the lw) then it would execute at the same rate as our original strlen because it would operate on 4 characters per 12 instructions or at the rate of \(\frac{1}{3}\) characters per instruction. If we could somehow check for a null with fewer than 12 instructions our new code would be faster.

In Problem 1 such a string length routine is to be completed. It is assumed that most students' MIPS skills are rusty so the starting point is code using a lhu instruction. In the solution to Problem 1 I attained a rate of 0.392 char/insn, not much better than .329 attained by our original routine.

In Problem 2 the strlen routine is to be written using additional non MIPS-I instructions. These include orc.b from a RISC-V extension, and clz and clo from MIPS32 (based on their r6 versions). Using these instructions my solution achieves 0.942 chars per insn.

\section*{Test Routine}

The code for this assignment includes a test routine that runs three string length routines: the routines to be written for Problems 1 and 2, and the string length routine written in class (called strlen_ref here). Each routine is run on several strings, including all lengths from 0 to 5, plus strings of length 23 and 196. The shorter-length strings are there to make sure that the routines
are correct and to check how fast they are on short strings. The longest string is there to test performance. The performance numbers from the previous section are based on the longest string.

Here is the output from the unmodified assignment:
** Starting Test of Routine "strlen_p1 (Problem 1 - Bit Ops)" **
String 1: Length 1 is correct. Took 10 insn or 0.100 char/insn
String 2: Length 2 is correct. Took 13 insn or 0.154 char/insn
String 3: Length 3 is correct. Took 16 insn or 0.188 char/insn
String 4: Length 4 is correct. Took 19 insn or 0.211 char/insn
String 5: Length 5 is correct. Took 22 insn or 0.227 char/insn
String 6: Length 0 is correct. Took 7 insn or 0.000 char/insn
String 7: Length 23 is correct. Took 76 insn or 0.303 char/insn
String 8: Length 196 is correct. Took 595 insn or 0.329 char/insn
** Starting Test of Routine "strlen_p2 \(\quad\) (Problem \(2-\mathrm{RISC}\) V orc insn)" **
String 1: Length 1 is correct. Took 11 insn or 0.091 char/insn
String 2: Length 2 is correct. Took 15 insn or 0.133 char/insn
String 3: Length 3 is correct. Took 19 insn or 0.158 char/insn
String 4: Length 4 is correct. Took 23 insn or 0.174 char/insn
String 5: Length 5 is correct. Took 27 insn or 0.185 char/insn
String 6: Length 0 is correct. Took 7 insn or 0.000 char/insn
String 7: Length 23 is correct. Took 99 insn or 0.232 char/insn
String 8: Length 196 is correct. Took 791 insn or 0.248 char/insn
** Starting Test of Routine "strlen_ref (Simple strlen routine.)" **
String 1: Length 1 is correct. Took 9 insn or 0.111 char/insn

String 2: Length 2 is correct. Took 12 insn or 0.167 char/insn
String 3: Length 3 is correct. Took 15 insn or 0.200 char/insn
String 4: Length 4 is correct. Took 18 insn or 0.222 char/insn
String 5: Length 5 is correct. Took 21 insn or 0.238 char/insn
String 6: Length 0 is correct. Took 6 insn or 0.000 char/insn
String 7: Length 23 is correct. Took 75 insn or 0.307 char/insn
String 8: Length 196 is correct. Took 594 insn or 0.330 char/insn
To see all of this output when running graphically it might be necessary to make the pop-up window larger. It is possible to scroll the text in the pop-up window by focusing the window and using the arrow keys.

Each line shows the result from one string. The length of the string is shown, as well as the number of instructions executed in the string length routine, and the execution rate. If the returned length had been wrong both the returned and correct length would be shown but the instruction count would be omitted.

The strings themselves can be found in the test code after the str label. The testbench does not print out the strings, just their lengths. Feel free to modify the strings if it helps with debugging, but please restore them before the deadline.
Using LSU version of SPIM
This assignment requires a modified version of the SPIM simulator originally developed by James Larus. Instructions for using this simulator appear on the course procedures page. When running SPIM check the LSU version date, there should be a line reading LSU Version Date: 2022-01-31. Make sure that the date is there and is no earlier than 31 January 2022. (The date will appear
on the console output near the top when run non-graphically, and in the lowermost window pane when run graphically.)

Two changes were made for this assignment: implementation of the RISC-V-like orb.c instruction, and implementation of the MIPS32 r6 (revision 6) clo (count leading ones) instruction. Also new is the ability to start and stop tracing.

\section*{Debugging}

To facilitate debugging the code can be run so that the simulator emits a trace of executed instructions, plus an indication of changed register values. The trace will mostly include the three string length routines, but it will also include a few testbench instructions. The trace includes line numbers so that there should be no confusion about where an instruction is from.

The best way to get a trace is to run the code non-graphically. To do so load the code into an Emacs buffer in a properly set up account. Press Ctrl-F9 to start the simulator non-graphically. That should pop up a window showing a simulator banner followed by a prompt:
```

SPIM Version 6.3.1 lsu of 9 November 2001, 17:34:35 CST
LSU Version Date: 2022-01-31
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Includes LSU modifications.
File loaded.
Type "run" to run normally.
Type "step 100" to execute next 100 instructions with tracing.
Type "help" for more help.
(spim)

```

At the prompt enter step 100 to run the next 100 instructions. The instructions in the string length routines will be traced, but the count of 100 instructions also includes the test routine (as of this writing). For example:
```

(spim) step 100
[0x00400064] 0x4080b000 mtc0 \$0, \$22 ; 218: mtc0 \$0, \$22
** Starting Test of Routine "strlen_p1 (Problem 1 - Bit Ops)" **
[0x004000d0] 0x0100f809 jalr \$31, \$8 ; 251: jalr \$t0

# Change in $31 ($ra) 0x4000bc -> 0x4000d8 Decimal: 4194492 -> 4194520

[0x004000d4] 0x40154800 mfc0 \$21, \$9 ; 252: mfc0 \$s5, \$9

# Change in $21 ($s5) 0 -> 0x23 Decimal: 0 -> 35

[0x00400000] 0x20820000 addi \$2, \$4, 0 ; 84: addi \$v0, \$a0,

# Change in $2 ($v0) Oxfffffffff -> 0x10010000 Decimal: -1 -> 268500992

[0x00400004] 0x94880000 lhu \$8, 0(\$4) ; 87: lhu $t0, 0($a0)

# Change in $8 ($t0) 0x400000 -> 0x3100 Decimal: 4194304 -> 12544

[0x00400008] 0x3109ff00 andi \$9, \$8, -256 ; 88: andi \$t1, \$t0, 0xff00\

# Change in $9 ($t1) 0x100101f0 -> 0x3100 Decimal: 268501488 -> 12544

[0x0040000c] 0x11200006 beq \$9, \$0, 24 [DONE0-0x0040000c]; 89: beq \$t1, \$0, DONEO\
[0x00400010] 0x310900ff andi \$9, \$8, 255 ; 90: andi \$t1, \$t0, 0xff■

```

Each line starting with square brackets shows the execution of an instruction. The address of the instruction is shown inside the square brackets. After the square brackets the instruction is
shown in three different forms. First encoded, shown in hexadecimal. Then a disassembled form (which is based on the encoded instruction). Finally, after the semicolon the instruction is shown as it appears in the assembler file. Immediately after the semicolon is a line number.

The lines that start with a \# show register values that change. The values are shown both in hexadecimal and decimal.

Problem 1: Routine strlen_p1 in hw01.s computes the length of a string using a loop that loads two characters at a time. It achieves a rate of .329 char/insn. Modify it so that it uses a lw instead of lhu. (Note that there is no such thing as lwu in MIPS I. Such an instruction only makes sense if registers are larger than 32 bits.) It is possible to achieve .393 chars /insn, or maybe even faster.

The string starting address will be in register a0. That address will be a multiple of 4 . Strings end with a null (a zero). The byte after the null is not part of the string and can be of any value. Don't assume it is a particular value.

Your solution should use MIPS-I instructions and should not use pseudo instructions except for nop. See the check-box comments (such as [ ] Code should be efficient.) for additional restrictions, requirements, and reminders.

The solution appears below. The complete solution file is at
https://www.ece.lsu.edu/ee4720/2022/hw01-sol.s.html. The easy part is changing lhu to a lw and changing addi \(\mathrm{a} 0, \mathrm{a} 0,2\) to addi \(\mathrm{a} 0, \mathrm{a} 0,4\). Next we need to modify the code so that it looks at the two most significant bytes. Because MIPS immediates are 16 bits we can't simply use an instruction like andi \(\mathrm{t} 1, \mathrm{t} 1\), \(0 x f f 000000\). But we can use an instructions like lui to load the constant in to a register, t 6 in the solution, then mask using and t 1 , t 1 , t6. Something is similar for mask \(0 x f f 0000\). Finally we need to adjust and add DONE targets for the different cases.

Since performance is important the masks are prepared before the loop is entered. The solution appears below.
```

strlen_p1:
\# CALL VALUE
\# \$a0: Address of first character of string.
\# RETURN
\# \$v0: The length of the string (not including the null).
addi \$v0, \$a0, 0
\# SOLUTION: Prepare masks for two high bytes.
lui \$t6, 0xff00 \# t6 -> 0xff000000
lui \$t7, 0xff \# t7 -> 0xff0000
LOOP: \# SOLUTION described by comments below.
lw $t0, 0($a0) \# Change lhu to lw
and \$t1, \$t0, \$t6 \# Mask off most-significant byte ..
beq \$t1, \$0, DONEO \# .. if it is zero we are done ..
and \$t1, \$t0, \$t7 \# .. otherwise mask off next byte ..
beq \$t1, \$0, DONE1 \# .. and if that one is zero we're done ..
andi \$t1, \$t0, 0xff00 \# .. otherwise mask the next one ......
beq \$t1, \$0, DONE2
andi \$t1, \$t0, 0xff
bne \$t1, \$0, LOOP
addi \$a0, \$a0, 4

```
```

    sub $v0, $a0, $v0
    jr $ra
    addi $v0, $v0, -1
    DONE2: \# SOLUTION: Modify the case for byte 2
sub \$v0, \$a0, \$v0
jr \$ra
addi \$v0, \$v0, 2
DONE1: \# SOLUTION: Add a case for byte 1
sub \$v0, \$a0, \$v0
jr \$ra
addi \$v0, \$v0, 1
DONEO:
jr \$ra
sub \$v0, \$a0, \$v0

```

Problem 2: Complete strlen_p2 so that it determines the string length by loading four characters (using a 1 w ) and checks for the null using the RISC-V-like orc.b (Bitwise OR-Combine, byte granule) instruction. Also helpful will be the MIPS32 6 clz and clo instructions.

The orc.b instruction is part of the RISC-V bit manipulation ISA extensions. See the documentation for this instruction for details on what it does. The documentation is linked to the course references page and of course can be found on the RISC-V site. The orc.b is in the strlen_p2 routine, but it doesn't do anything useful. Of course, that should be changed as part of the solution.

The MIPS32 clz and clo might also come in handy. Look for the MIPS32 r6 (not the older versions) Volume 2 manuals on the course references page.

It is possible to complete this so that it runs at 0.947 char /insn or faster.
The solution appears below. The complete solution file is at
https://www.ece.lsu.edu/ee4720/2022/hw01-sol.s.html.
The orc.b r1, r2 instruction operates on each byte of the \(r 2\) value. If the byte is non-zero it is replaced by \(0 x f f\) otherwise it remains zero. The transformed value is returned. So for \(r 2=0 x 1100 a a 00\) the value assigned to \(r 1\) would be \(0 x f f 00 f f 00\). If \(r 2\) holds four bytes of a string, then a value of \(r 1=0 x f f f f f f f f\) indicates that no null has been found.

The number of bytes before a null can be determined by counting the number of leading ones (the number of consecutive 1's starting at the most-significant bit position). As suggested in the assignment a clo instruction does just that.
since performance is a goal, the main loop must be written using as few instructions as possible. For that reason the code in the main loop checks only that there are no nulls, and leaves the task of determining how many characters preceded the null to the code after the loop exit. In the solution appearing below the main loop consists of just four instructions: 1 w , orc. b , bne, and addi.

After the exit a clo counts the most-significant 1's and a srl is used to divide that value by 8 , converting bits to chars. See the comments in the solution for details.

Grading note: Many seemingly did not take into account that when performance is important nothing should be done in a main loop that could be done either before or after the main loop. For example, in the solution below the main loop uses a value of t 3 that was prepared before the loop was entered.
strlen_p2:
```


# CALL VALUE

# \$a0: Address of first character of string.

# RETURN

# \$v0: The length of the string (not including the null).

addi \$v0, \$a0, 4
addi \$t3, \$0, -1 \# SOLUTION: Prepare value: Oxffffffff

```
```

lw $t0, O($a0)

```
lw $t0, O($a0)
orc.b $t1, $t0
orc.b $t1, $t0
# SOLUTION: Example execution of orc.b:
# SOLUTION: Example execution of orc.b:
# String at address in $a0: "AB"
# String at address in $a0: "AB"
# t0: A B X <-- Value in ASCII. is null, X unknown.
# t0: A B X <-- Value in ASCII. is null, X unknown.
# t0: 0x41420099 <-- Value in hex.
# t0: 0x41420099 <-- Value in hex.
# t0: 0x41 0x42 0x00 0x99 <-- Value separated into four bytes
# t0: 0x41 0x42 0x00 0x99 <-- Value separated into four bytes
# t1: 0xff 0xff 0x00 0xff <-- Result. Non-zero bytes changed to 0xff
# t1: 0xff 0xff 0x00 0xff <-- Result. Non-zero bytes changed to 0xff
# t1: Oxffff00ff <-- Result. Non-zero bytes changed to Oxff
# t1: Oxffff00ff <-- Result. Non-zero bytes changed to Oxff
beq $t1, $t3, LOOPB # If t1 is 0xffffffff then no null found ..
beq $t1, $t3, LOOPB # If t1 is 0xffffffff then no null found ..
addi $a0, $a0, 4 # .. so increment address for next word.
addi $a0, $a0, 4 # .. so increment address for next word.
# A null has been found. Count number of leading 1's to find out.
# A null has been found. Count number of leading 1's to find out.
# t1: Oxffff00ff <-- Continuing with example.
# t1: Oxffff00ff <-- Continuing with example.
clo $t2, $t1
clo $t2, $t1
# t2: 16 <-- Found 16 consecutive 1's starting at MSB.
# t2: 16 <-- Found 16 consecutive 1's starting at MSB.
sra $t2, $t2, 3 # Effectively divide by }8
sra $t2, $t2, 3 # Effectively divide by }8
# t2: 2 <-- 16 1's means 2 bytes
# t2: 2 <-- 16 1's means 2 bytes
sub $v0, $a0, $v0 # Add on number of bytes before null.
sub $v0, $a0, $v0 # Add on number of bytes before null.
jr $ra
jr $ra
add $v0, $v0, $t2
```

add \$v0, \$v0, \$t2

```

LOOPB:

\section*{非 LSU EE 4720 Spring 2022 Homework 1}

非
非
非 SOLUTION
```

    ## Assignment https://www.ece.lsu.edu/ee4720/2022/hw01.pdf
    非 Solution Discussion https://www.ece.lsu.edu/ee4720/2022/hw01 sol.pdf
    ```

非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非
    : Problem 1
非
非 Instructions: https://www.ece.lsu.edu/ee4720/2022/hw01.pdf
strlen_p1:
            非 Register Usage
            非
            非 CALL VALUE
            \$a0: Address of first character of string.
                This address will be a multiple of 4.
                    RETURN
                    \$v0: The length of the string (not including the null).
                    非
非 Note:
Can modify registers \$t0-\$t9, \$a0-\$a3, \$v0, \$v1.
DO NOT modify other registers.
[ \(\checkmark\) ] The testbench should show 0 errors.
[ \(\checkmark\) ] Try to reduce the number of executed instructions.
\([\checkmark]\) Do not use pseudoinstructions except for nop.
            Do not use: li, la, mov, bgt, blt, etc.
[ \(\checkmark\) ] Code should be efficient.
[ \(\checkmark\) ] The code should be clearly written.
[ ] Comments should be written for an experienced programmer.
addi \$v0, \$a0, 0
非 SOLUTION: Prepare masks for two high bytes.
非
lui \$t6, 0xff00 非 t6 -> 0xff000000
lui \$t7, 0xff 非 t7 -> 0xff0000
非
非 In the version using lhu it was possible to mask off each
非 byte using an immediate, such as in "andi \$t1, \$t0, 0xff".
非 But since immediate values are limited to 16 bits we need to
非 use some other method to get the mask in a register. The
非 method used above is an lui instruction.

LOOP：
非 SOLUTION described by comments below．
lw \＄t0，0（\＄a0）非 Change lhu to lw
and \＄t1，\＄t0，\＄t6 非 Mask off most－significant byte ．．
beq \＄t1，\＄0，DONE0 非 ．．if it is zero we are done ．．
and \＄t1，\＄t0，\＄t7 非 ．．otherwise mask off next byte ．．
beq \＄t1，\＄0，DONE1 非 ．．and if that one is zero we＇re done ．．
andi \＄t1，\＄t0，0xff00 非 ．．otherwise mask the next one ．．．．．
beq \＄t1，\＄0，DONE2
andi \＄t1，\＄t0，0xff
bne \＄t1，\＄0，LOOP
addi \＄a0，\＄a0， 4
sub \＄v0，\＄a0，\＄v0
jr \＄ra
addi \＄v0，\＄v0，－1

DONE2：非 SOLUTION：Modify the case for byte 2
sub \＄v0，\＄a0，\＄v0
jr \＄ra
addi \＄v0，\＄v0， 2

DONE1：非 SOLUTION：Add a case for byte 1
sub \＄v0，\＄a0，\＄v0
jr \＄ra
addi \＄v0，\＄v0， 1

DONEO：
```

jr \$ra
sub \$v0, \$a0, \$v0

```

非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非
排 Problem 2
非
strlen＿p2：
```


## CALL VALUE

    $a0: Address of first character of string.
                This address will be a multiple of 4.
    RETURN
    $v0: The length of the string (not including the null).
    Note:
Can modify registers $t0-$t9, $a0-$a3, \$v0, \$v1.
DO NOT modify other registers.

# 

    ## [\checkmark] Use lw to load from string. (Replace the lbu)
    ## [\checkmark] Make use of orc.b insn. Consider using clz, clo.
    ```

```

    # [\checkmark] Try to reduce the number of executed instructions.
    # [\checkmark] Do not use pseudoinstructions except for nop.
    #⿰⿰三丨⿰丨三一笁 Do not use: li, la, mov, bgt, blt, etc.
    
# 

## [\checkmark] Code should be efficient.

\#⿰三丨⿰丨三一灬[\checkmark] The code should be clearly written.
非 [ ] Comments should be written for an experienced programmer.

```
```

\#⿰⿰三丨⿰丨三一湆 SOLUTION Approach

# 

# In main loop check whether there is a null and do so using

## the minimum number of instructions. The orc.b instruction

# returns a 0xffffffff if no null is found, and it's easy to

非 check for that. If a null is found, exit the loop and *then*

## figure out where.

addi \$v0, \$a0, 4
addi \$t3, \$0, -1 非 SOLUTION: Prepare value: 0xffffffff

```

LOOPB：
```

lw $t0, 0($a0)

```
orc.b \$t1, \$t0
非 SOLUTION: Example execution of orc.b:
非 String at address in \$a0: "AB"
非 t0: A B ø X <-- Value in ASCII. \(\varnothing\) is null, X unknown.
非 t0: 0x41420099 <-- Value in hex.
非 t0: 0x41 0x42 0x00 0x99 <-- Value separated into four bytes
非 t1: 0xff 0xff 0x00 0xff <-- Result. Non-zero bytes changed to 0xff
非 t1: 0xffff00ff <-- Result. Non-zero bytes changed to 0xff
beq \$t1, \$t3, LOOPB 非 If t1 is 0xfffffffff then no null found ..
addi \$a0, \$a0, 4 非 .. so increment address for next word.
非 A null has been found. Count number of leading 1's to find out.
非 t1: 0xffff00ff <-- Continuing with example.
clo \$t2, \$t1
非 t2: 16 <-- Found 16 consecutive 1's starting at MSB.
sra \$t2, \$t2, 3 非 Effectively divide by 8.
非 t2: \(2<--16\) 1's means 2 bytes
sub \$v0, \$a0, \$v0 非 Add on number of bytes before null.
jr \$ra
add \$v0, \$v0, \$t2
非
strlen_ref:
    非 Register Usage
    非
    非 \$a0: Address of first character of string.
    非 \$v0: Return value, the length of the string.
    addi \$v0, \$a0, 1 非 Set aside a copy of the string start + 1.
REF_LOOP:
    lbu \$t0, 0(\$a0) 非 Load next character in string into \$t0
```

bne \$t0, \$0, REF_LOOP 非 If it's not zero, continue
addi \$a0, \$a0, 1 非 Increment address. (Note: Delay slot insn.)

```
jr \$ra
sub \$v0, \$a0, \$v0

```


# 

    非 Test Code
    非

# The code below calls the strlen routines.

```
```

.data

```
str:
.align 2
.asciiz "1"
.align 2, -1
.asciiz "12"
.align 2, -1
.asciiz "123"
.align 2, -1
.asciiz "1234"
.align 2, -1
.asciiz "12345"
.align 2, -1
.asciiz ""
.align 2, -1
.asciiz "\"Per aspera, ad astra!\""
.align 2, -1
.ascii "eighteen quintillion, four hundred forty six quadrillion, "
.ascii "seven hundred forty four trillion, seventy three billion, "
.ascii "seven hundred nine million, five hundred fifty one thousand, "
.asciiz "six hundred sixteen"
\(m s g\) ：
```

    .asciiz "String %/s2/2d: Length %/v1/3d is "
    ```
msg＿good：
．asciiz＂correct．Took \％／s4／3d insn or \％／f6／．3f char／insn\n＂
msg＿bad：
．asciiz＂wrong．Should be \％／s7／3d．\n＂
mut＿strlen＿p1：
．word strlen＿p1
．asciiz＂strlen＿p1（Problem 1－Bit Ops）＂
mut＿strlen＿p2：
．word strlen＿p2
．asciiz＂strlen＿p2（Problem 2 －RISC V orc insn）＂
mut＿strlen＿ref：
．word strlen＿ref
．asciiz＂strlen＿ref（Simple strlen routine．）＂
muts：
．word mut＿strlen＿p1
．word mut＿strlen＿p2
．word mut＿strlen＿ref
．word 0
mut＿msg：
．asciiz＂\n＊＊Starting Test of Routine \＂\％／t0／s\＂＊＊\n＂
．text
．globl＿＿start
＿＿start：
mtc0 \＄0，\＄22
非 Pause tracing．
TBOUTER：
la \＄t0，muts
sll \＄t1，\＄s1， 2
add \＄t1，\＄t1，\＄t0
lw \＄s0，0（\＄t1）
bne \＄s0，\＄0，TB＿MORE
nop
addi \＄v0，\＄0， 10 非 System call code for exit．
syscall
TB＿MORE：
la \＄a0，mut＿msg
addi \＄t0，\＄s0， 4
addi \＄v0，\＄0， 11
syscall
la \＄a0，str
addi \＄s6，\＄a0， \(0 \quad\) 非 Save copy of string starting address．
addi \＄s2，\＄0， 0
TBLOOP：
addi \＄a0，\＄s6， 0
jal strlen＿ref
addi \＄s2，\＄s2， 1
addi \＄s7，\＄v0， 0
addi \＄a0，\＄s6， 0
addi \＄v0，\＄0，－1
lw \＄t0，0（\＄s0）
mtc0 \＄v0，\＄22
jalr \＄t0
mfc0 \＄s5，\＄9
\＃Resume tracing．（No effect if not stepping．）
mfc0 \＄s4，\＄9
mtc0 \＄0，\＄22
addi \＄s4，\＄s4，－1
sub \＄s4，\＄s4，\＄s5
mtc1 \＄s4，\＄土4
mtc1 \＄v0，\＄f0
div．d \＄£6，\＄さ0，\＄£4
addi \＄v1，\＄v0， \(0 \quad\) 非 Move length of string to \＄v1
addi \＄v0，\＄0， 11
la \＄a0，msg
```


## Move length of string to \$v1

## System call code for message.

## Address of message.

```
syscall
la \$a0, msg_good
beq \$v1, \$s7 TB_CONTINUE
nop
la \$a0, msg_bad
TB_CONTINUE:
syscall
add \$s6, \$s6, \$s7
ori \$s6, \$s6, 0x3
addi \$s6, \$s6, 1
la \$a0, msg
slt \$t0, \$s6, \$a0
bne \$t0, \$0, TBLOOP
nop
j TBOUTER
addi \$s1, \$s1, 1

Problem 1: The code fragment below was taken from the course hex string assembly example. (The hex string example was not covered this semester. The full example can be found at https://www.ece.lsu.edu/ee4720/2022/hex-string.s.html.) The fragment below converts the value in register a0 to an ASCII string, the string is the value in hexadecimal (though initially backward).

LOOP:
```

andi \$t0, \$a0, Oxf \# Retrieve the least-significant hex digit.
srl \$a0, \$a0, 4 \# Shift over by one hex digit.
slti \$t1, \$t0, 10 \# Check whether the digit is in range 0-9
bne \$t1, \$0, SKIP \# Don't forget that delay slot insn always exec.
addi \$t2, \$t0, 48 \# If 0-9, add 48 to make ASCII '0' - '9'.
addi \$t2, \$t0, 87 \# If 10-15, add 87 to make ASCII 'a' - 'z'.
sb $t2, 0($a1) \# Store the digit.
bne \$a0, \$0, LOOP \# Continue if value not yet zero.
addi \$a1, \$a1, 1 \# Move string pointer one character to the left.

```
SKIP:
(a) Show the encoding of the MIPS bne t1, 0, SKIP instruction. Include all parts, including-especiallythe immediate. For a quick review of MIPS, including the register numbers corresponding to the register names, visit https://www.ece.lsu.edu/ee4720/2022/lmips.s.html.

The encoding appears below. Note that register t1 (temporary 1) is a helpful name for r9, and so a 9 is placed in the rt field. If the bne is taken it advances by two instructions (starting from the delay-slot instruction), and so the immediate field holds a 2.

(b) RISC-V RV32I has a bne instruction too, though it is not exactly the same. Show the encoding of the RV32I version of the bne t1, 0, SKIP instruction. For this subproblem assume that the bne will jump ahead two instructions, just as it does in the code sample above.

To familiarize yourself with RISC-V start by reading Chapter 1 of Volume I of the RISC-V specification, especially the Chapter 1 Introduction and Sections 1.1 and 1.3. Skip Section 1.2 unless you are comfortable with operating system and virtualization concepts. Other parts of Chapter 1 are interesting but less relevant for this problem. Also look at Section 2.5 (Control Transfer Instructions). The spec can be found in the class references page at https://www.ece.lsu.edu/ee4720/reference.html.

The branch instructions are discussed in Section 2.5 under the Conditional Branches neading. There are two significant differences with the MIPS bne. First, there is no delay slot. That's not relevant in this problem. Second, the immediate field value is used differently. Let IMM denote the immediate field value (based on the bits set in the instruction). The branch target is then PC \(+2 *\) IMM, where PC is the address of the branch. (In MIPS the target would be PC \(+4+4 *\) IMM.) So, we need to set the immediate to the number of bytes to skip divided by two. The problem says to jump ahead two instructions, in RISC-V (and most RISC ISAs) that's 8 bytes, and so the immediate field should be set to 4.

Though Section 2.5 shows the encoding of a bne instruction, it does not provide the values for opcode fields and their extensions, instead using names: \(B R A N C H\) for the opcode field and \(B N E\) for the funct3 field. The values can be found in Chapter 24.

The encoding appears below. The instruction field names have been abbreviated, such as im12 for imm [12]. Also note that in RISC-V immediate field names use the bit numbers within the immediate. So, for example, imm [4:1] (abbreviated im \(4: 1\) below) indicates that the four bits in the instruction field ( \(a 4_{10}=0100_{2}\) in the example) are placed in bit positions \(4: 1\) of the immediate. There is no imm [0] field in the instruction because that immediate bit is always set to zero. So, the IMM*2 is computed by putting the twelve immediate bits in the instruction in bits \(12: 1\) of the immediate, setting bit 0 , the LSB, to zero.

(c) Consider the four-instruction sequence from the code above:
```

slti \$t1, \$t0, 10
bne \$t1, \$0, SKIP
addi \$t2, \$t0, 48
addi \$t2, \$t0, 87

# Check whether the digit is in range 0-9

# Don't forget that delay slot insn always exec.

# If 0-9, add 48 to make ASCII '0' - '9'.

# If 10-15, add 87 to make ASCII 'a' - 'z'.

```
SKIP:

Re-write this sequence in RISC-V RV32I, and take advantage of RISC-V branch behavior to reduce this to three instructions (plus possibly one more instruction before the loop). For this problem one needs to focus on RISC-V branch behavior. Assume that the RISC-V slti and addi instructions are identical to their MIPS counterparts at the assembly language level. It is okay to retain the MIPS register names. Hint: One change needs to be made for correctness, another for efficiency.

Solution appears below. Before the loop is entered the addi instruction sets t6 to 10, a constant that will come in handy. Inside the loop the four MIPS instructions are replaced by three RISC-V instructions. First, the slti is no longer necessary because RISC-V has a blt (branch less than). The blt itself checks whether to is less than 10 (which is in t 6 ). Using the blt to do the comparison is the efficiency change mentioned in the hint. Because RISC-V lacks delay slots the addi \(\mathrm{t} 2, \mathrm{t} 0,48\) had to be moved before the blt. That's the correctness change mentioned in the hint. Notice that RISC-V has register names that are similar to MIPS, such as t0-t6 for caller-save (temporary) registers.
```

    # Instruction inserted before the loop to put 10 into register t6.
    addi t6, zero, 10
    LOOP:

```
```


# These instructions replace the four MIPS instructions.

```
# These instructions replace the four MIPS instructions.
addi t2, t0, 48 # Convert assuming t0 in range 0-9 ..
addi t2, t0, 48 # Convert assuming t0 in range 0-9 ..
blt t0, t6, SKIP # .. if that's correct, branch ..
blt t0, t6, SKIP # .. if that's correct, branch ..
addi t2, t0, 87 # .. otherwise convert assuming a-f.
addi t2, t0, 87 # .. otherwise convert assuming a-f.
SKIP:
```

Problem 2: Note: The following problem was assigned in each of the last six years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
# Cycle
0
lw r2, O(r4)
IF ID EX ME WB
add r1, r2, r7
    IF ID EX ME WB
```

The add depends on the 1 w through r2, and for the illustrated implementation the add has to stall in ID until the 1 w reaches WB.

```
# Cycle
0
SOLUTION
lw r2, 0(r4)
IF ID EX ME WB
add r1, r2, r7
    IF ID ----> EX ME WB
```

(b) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4)
    IF ID -> EX ME WB
```

There is no need for a stall because the 1 w writes $r 1$, it does not read $r 1$.

```
# Cycle 0
add r1, r2, r3
lw r1, O(r4) IF ID EX ME WB
```

(c) Explain error and show correct execution.

```
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID -> EX ME WB
```

A longer stall is needed here because the sw reads r1 and it must wait until add reaches WB

```
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
    IF ID ----> EX ME WB
```

(d) Explain error and show correct execution.

```
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB
```

The stall above allows the xor, when it is in ID, to get the value of $r 1$ written by the add; that part is correct. But, the stall starts in cycle 1 before the xor reaches $I D$, so how could the control logic know that the xor needed r1, or for that matter that it was an xor? The solution is to start the stall in cycle 2, when the xor is in ID.

```
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ID ----> EX ME WB
```

Note: The following problems (or very similar problems) were assigned in 2020 and 2021, and their solutions are available. DO NOT look at the solutions unless you are lost and can't get help elsewhere. Even in that case just glimpse.

Problem 1: Appearing below are incorrect executions on the illustrated implementation. Notice that this implementation is different than the one from the previous problem. For each execution explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3
xor r4, r1, r5 IF ID -> EX ME WB
```

There is a bypass path available so that there is no need to stall.

```
# Cycle 0
add r1, r2, r3
IF ID EX ME WB
xor r4, r1, r5 IF ID EX ME WB
```

(b) The execution of the branch below has two errors. One error is due to improper handling of the andi instruction. (That is, if the andi were replaced with a nop there would be no problem in the execution below.) The other is due to the way the beq executes. As in all code fragments in this problem, the program is correct, the only problem is with the illustrated execution timing.

```
# Cycle: }0
andi r2, r2, 0xff IF ID EX ME WB
beq r1, r2, TARG IF ID EX ME WB
add r3, r4, r5 IF ID EX ME WB
xor IFx
TARG:
    Sw r6, 7(r8) IF ID EX ME WB
# Cycle: 0
```

Briefly, the two problems are the lack of a stall for the andi/beq dependence carried by $r 2$ and because the branch target is fetched one cycle later than it should be. The correct execution appears below.

Detailed explanation: In the illustrated implementation the $\square$ in ID is used to compute the branch condition for beq (and bne). When the branch reaches $I D$, in cycle 2 , the value of $r 2$ retrieved from the register file is outdated, it needs to use the value computed by andi. Since there are no bypass paths to the $\quad=$ logic the branch will need to stall until andi reaches writeback. The stalls occur in cycles 2 and 3 .

The illustrated implementation resolves the branch in $I D$, and $s o$ the branch target should be in $I F$ when the branch is in EX. In the execution above the target isn't fetched until the branch is in ME , in cycle 4 . That is fixed below by fetching the target a cycle earlier. The xor is no longer fetched and squashed.

```
# Cycle: 0
andi r2, r2, 0xff IF ID EX ME WB
beq r1, r2, TARG IF ID ----> EX ME WB
add r3, r4, r5 IF ----> ID EX ME WB
xor
TARG:
    sw r6, 7(r8) IF ID EX ME WB
# Cycle: }0
```


(c) Explain error and show correct execution.

```
# Cycle
0
lw r2, O(r4)
add r1, r2, r7
```

$\begin{array}{llllllll}0 & 1 & 2 & 3 & 4 & 5 & 6 & 7\end{array}$
IF ID EX ME WB IF ID EX ME WB

The add depends on the 1 w through r2, and for the illustrated implementation the add has to stall in ID until the lw reaches ME so that the add can bypass from WB.

```
# Cycle
0
IF ID EX ME WB
lw r2, O(r4)
    IF ID -> EX ME WB
```

(d) Explain error and show correct execution.

```
# Cycle
0
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4)
    IF ID -> EX ME WB
```

There is no need for a stall because r 1 is not a source register of 1 w . Note that r 1 is a destination of 1 w .

```
# Cycle
0
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID EX ME WB
```

(e) Explain error and show correct execution.

```
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID -> EX ME WB
```

No stall is needed here because the sw can use the ME-to-EX bypass path.

```
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID EX ME WB
```

Problem 2: Appearing below is the labeled MIPS implementation from 2018 Midterm Exam Problem 2(b), and as in that problem each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.

(a) Use F0. Don't be fancy about it, just one instruction is all it takes.

Solution appears below. F0 is used by values being loaded from memory into the pipeline. Load instructions, include 1 w , use F0.

```
# SOLUTION
# Cycle 
lw r1, O(r2) IF ID EX ME WB
```

(b) Use F0, C2, and D3 at the same time. The code should not suffer a stall. More than one instruction is needed for the solution. Note: This is new in 2022.

The solution appears below. The F0 mux input is used by load instructions. For that a lw instruction is included in the solution. The D3 mux input is used to bypass something from WB to the second ALU operand. To use F0 and D3 at the same time the load instruction must be in WB at the same time as the other instruction (an add in the example below) is in EX. The add instruction uses the D3 bypass to get the value of $r 1$ written by the 1 w . To use C 2 the instruction must use an unbypassed value for the first source. The first source of the add is r9 which has not been written by the two prior instructions, and so it can use the value from the register file.

```
# SOLUTION
# Cycle 0
lw r1, O(r2) IF ID EX ME WB
xor r5, r6, r7 IF ID EX ME WB
add r3, r9, r1 IF ID EX ME WB
# Cycle 
```

(c) Explain why its impossible to use E0 and D0 at the same time.

If E 0 is in use then there must be a store instruction in EX . If D 0 is in use then a value is being bypassed to the second ALU source operand of the instruction in EX. But store instructions use an immediate for the second ALU input, so a store in EX can only use D2, it can't use D0 (nor D1 nor D3).

Problem 3: This problem appeared as Problem 2c on the 2020 final exam. Appearing below is our bypassed, pipelined implementation followed by a code fragment.

It might be helpful to look at Spring 2019 Midterm Exam Problem 4a. That problems asks for the execution of a loop and for a performance measure based upon how fast that loop executes.

(a) Show the execution of the code below on the illustrated implementation up to the point where the first instruction, addi r2,r2,16, reaches WB in the second iteration.

The execution appears below. The execution is shown until the beginning of the third iteration. (A full-credit solution would only need to show execution until cycle 10 , when the addi $r 2, r 2,16$ reaches WB in the second iteration.) The only stall is a 1 -cycle load/use stall suffered by the sw. The first iteration starts in cycle 0 (when the first instruction, addi, is in IF), the second iteration starts at cycle 6, and the third at cycle 12.

Note that the pattern of stalls in the second iteration is the same as the pattern in the first. We can expect this pattern to continue because the contents of the pipeline is the same at the beginning of the second and third iterations. (The second iteration begins in cycle 6. In that cycle the addi r2 is in IF, the addi r3 is in ID, etc. The contents of the pipeline is the same in cycle 12.)

```
        ## SOLUTION
LOOP: # Cycle (lllllllllllllllllllllllllll
    addi r2, r2, 16
    lw r1, 8(r2)
    sw r1, 12(r3)
    bne r3, r4, LOOP
    addi r3, r3, 32
    sub r10, r3, r2
LOOP: # Cycle }0
    addi r2, r2, 16 IF ID EX ME WB Second Iteration
    lw r1, 8(r2)
    Sw r1, 12(r3)
    bne r3, r4, LOOP
    addi r3, r3, 32
LOOP: # Cycle 0
    addi r2, r2, 16 Third Iteration IF ID EX ME WB
        IF ID EX ME WB First Iteration
        IF ID EX ME WB
            IF ID -> EX ME WB
                IF -> ID EX ME WB
                        IF ID EX ME WB
        IF ID EX ME WB
            IF ID -> EX ME WB
                    IF -> ID EX ME WB
                IF ID EX ME WB
```

(b) Based on your execution determine how many cycles it will take to complete $n$ iterations of the loop.

The time for $n$ iterations of the loop is $n$ times the duration of one iteration of the loop. The key to solving this correctly is using the correct duration for an iteration. The duration of an iteration if the time between the start of two consecutive iterations. In this class the start time of an iteration is the time at which the first instruction is in IF. Using that definition the duration of the first iteration is $6-0=6$ cyc and the duration of the second is $12-6=6 \mathrm{cyc}$. So the number of cycles to complete $n$ iterations is $6 n \mathrm{cyc}$.

An important point to understand is that the definition of duration above insures that iterations don't overlap. That is, by defining an iteration duration as starting in the IF of the first instruction of the iteration, there is no possibility that two iterations overlap and there is no time gap between them. That's what enables us to multiply a duration by the number of iterations to get a total time.

Some might be tempted to add another four cycles to account for the addi r3 instruction completing execution. No credit would be lost for that in a solution, but that is not useful for our purposes because we might want to add together the duration of different pieces of code, so for us the important thing is when the next instruction can be fetched.

Homework 4 Solution

Problem 1: Appearing below is our familiar five stage MIPS implementation with a new branch bypass path shown in blue. For this problem assume that orc.b is executed by the ALU.

(a) The code below is based on a solution to Homework 1. Show a pipeline execution diagram of this code on the illustrated hardware. Pay close attention to the behavior of the branch including behavior due to dependencies with prior instructions. Show enough of the execution to compute the instruction throughput in units of IPC.
Show execution on the illustrated hardware. $\triangle$ Compute the instruction throughput (IPC). $\square$ Pay attention to dependencies and available bypass paths.

Solution appears below. The only unbypassable dependency was from orc.b to beq. The beq needs the value of t 1 when beq is in ID but it cannot be bypassed until orc.b reaches ME (using the blue mux) and so the beq stalls one cycle.

The instruction throughput is $\frac{4 \mathrm{insn}}{(6-1) \mathrm{cyc}}=.8 \mathrm{insn} / \mathrm{cycle}$. Note that the number of cycles in an iteration is computed by using the fetch of the first instruction in the loop body, the addi. So the number of cycles is $6-1=5$.
\# SOLUTION
LOOPB: \# Cycle
addi \$a0, \$a0, 4
IF ID EX ME WB
$\begin{array}{lllllllllllllll}0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 15\end{array}$
orc.b \$t1, \$t0
IF ID EX ME WB \# 1st ITERATION
beq \$t1, \$t3, LOOPB
IF ID EX ME WB
lw \$t0, $0(\$ \mathrm{a} 0)$
IF ID $\rightarrow$ EX ME WB
IF $\rightarrow$ ID EX ME WB
LOOPB: \# Cycle
addi \$a0, \$a0, 4
orc.b \$t1, \$t0
beq \$t1, \$t3, LOOPB
$\begin{array}{llllllllllll}4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15\end{array}$
lw \$t0, $0(\$ \mathrm{aO})$ IF ID EX ME WB \# 2nd ITERATION

IF ID EX ME WB
IF ID $\rightarrow$ EX ME WB IF -> ID EX ME WB
(b) The code below should have executed more slowly on the illustrated implementation. Explain why. Hint: The only difference in the code is the branch instruction.

```
lw $t0, O($a0)
```

LOOPB:

```
addi $a0, $a0, 4
orc.b $t1, $t0
beq $t3, $t1, LOOPB
lw $t0, O($a0)
```

$\checkmark$ Explain why the code above executes more slowly.
The comparison unit used to evaluate the branch condition (the $\quad$ 二 box in the ID stage) can bypass an re register value (though only from ME ) but not an rt value. In both code fragments the beq needs the value of t 1 written by the orc. b , but in the code fragment immediately above t 1 is the beq rt register and so there is no bypass path.

Problem 2: Appearing below is the implementation used in the previous problem. Add control logic for the branch condition multiplexor (shown in blue). Feel free to insert an is Branch logic block to detect the presence of a branch based on the instruction opcode. For an Inkscape SVG version of the implementation follow https://www.ece.1su.edu/ee4720/2022/hw04-br-byp.svg.

The solution appears below in blue. Note that it was not necessary to check whether the instruction in ID is a branch because only a branch instruction would use that mux. It is assumed that logic already exists to generate a stall signal when there is a dependence with the instruction in EX.


Problem 3: Appearing below is our MIPS implementation (the one we use, we're not taking credit for inventing it) with an orc.b unit in the EX stage. Unlike the first problem in this assignment, here the orc.b instruction is executed by its own unit, not by the ALU. One reason is because orc.b is fairly easy to compute, and so its output can be available much sooner than the ALU's output. In fact, it will be available early enough to be bypassed to ID for use in determining the branch condition.

Connect the orc.b functional unit so that it can be used by orc.b instructions. Paying attention to cost, connect it so that the following bypasses are possible: (1) A bypass so that an immediately following dependent branch does not stall. This would eliminate a stall in a solution to Problem 1, and avoid a stall in Case 1 in the code fragment below. (2) Bypasses to the next two arithmetic/logical instructions. See Case 2 below.

When weighing design alternatives assume that one pipeline latch bit cost twice as much as one mulltiplexor bit. Don't overlook opportunities to reuse existing hardware. The Inkscape SVG source for the diagram below is at https://www.ece.lsu.edu/ee4720/2022/hw04-orc.svg.

```
# Case 1
orc.b R1, r9
beq R1, r10, TARG
# Case 2
orc.b R1, r9
add r2, R1, r3 # Bypass from ME
xor r4, R1, r5 # Bypass from WB
or r6, R1, r7 # No bypass needed.
```

Connect orch unit so code above executes without a stall.
Show control logic for any multiplexors added. (Control logic does not need to be shown for the branch condition max.)
As always, avoid costly, inefficient, and unclear solutions.


The solution appears on the next page.

The solution appears below in turquoise. The orc.b instruction has one source, the rs register value. That value is taken at the output of the upper ALU mux, this way we can take care of the existing bypass paths and control logic.

The output of orc.b connects to two places. For the important orc.b/beq use case the output is connected directly to the branch = mux for a stall-free bypass to the branch instruction. For other cases we need to put it on a path to the register file. The chosen solution first puts it in the rtv mux, because that path is not otherwise used by the orc. b instruction. Then, in the ME stage a mux is used to put the value on the "main" path back to the register file. The is orc.D signal is used as a select input to the mux. Of course, the signal travels with the instruction by using pipeline latches.

There are two efficiency issues worth noting. First, it is possible to place a mux between the ALU and the pipeline laten, and have the orc.b output connect to an input to that mux. That would have received full credit. But, it would be correct to argue that the output of the ALU was on the critical path and so that should be avoided. That is why in the solution below the path from the ALU output to the latch is not touched. For the same reason the path from ME. ALU to the memory port address input is not touched.


Problem 1: Solve 2021 Final Exam Problem 2(a) and (b). Problem 2(a) asks for a pipeline execution diagram for the execution of floating-point code on a MIPS implementation which is a little different than the ones in the class notes. Problem 2(a) also asks for additional information, including the instruction throughput. In Problem 2(b) the floating-point code is to be scheduled to improve the throughput. Note: A brief summary of the problem is provided here to reduce the chance that you solve the wrong problem, say by getting the year or problem number wrong.

See posted final exam solution at https://www.ece.lsu.edu/ee4720/2021/fe_sol.pdf.

Problem 1: Solve 2021 Final Exam Problem 2(c). In the problem the execution of a loop on a 4 -way superscalar MIPS implementation is to be shown.

See posted final exam solution at https://www.ece.lsu.edu/ee4720/2021/fe_sol.pdf.
Problem 2: Solve 2021 Final Exam Problem 1. In this problem some features of an unconventional 2-way superscalar processor are to be completed. The solution to this problem is not as long as it might seem.

See posted final exam solution at https://www.ece.lsu.edu/ee4720/2021/fe_sol.pdf.
$\square$

Problem 1: Solve 2021 Final Exam Problem 3 (all parts). The problem has some routine predictor analysis questions, how to craft a side-channel attack exploiting of local predictor that does not reset its tables at context switches, and questions about a bimodal predictor with a separate tag store (as covered in class on Friday). For example local predictor analysis problems see prior years' final exams.

See posted final exam solution at https://www.ece.lsu.edu/ee4720/2021/fe_sol.pdf.

Problem 1: Solve 2021 Final Exam Problem 4 parts a,b,c,d. (Don't solve 4e). These are an assortment of short answer questions, covering superscalar and vector processors, and other topics.

See posted final exam solution at https://www.ece.lsu.edu/ee4720/2021/fe_sol.pdf.

## 41 Spring 2021 Solutions

Homework 1 Solution

The solution has been copied into the assignment directory in file hw01-sol.s. An HTML version is at https://www.ece.lsu.edu/ee4720/2021/hw01-sol.s.html.

Problem 0: Follow the instructions for class account setup and for homework workflow in https://www.ece.1su.edu/ee4720/proc.html. Review the comments in hw01.s and look for the area labeled "Problem 1."

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2021/hw01.s.html. For MIPS references see the course references page, https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in https://www.ece.lsu.edu/ee4720/proc.html.

Problem 1: The hw01.s file has a routine called getbit.
(a) Complete the getbit routine so that it returns the value of a bit from a bit vector that spans one or more bytes. Register $\$ \mathrm{aO}$ holds the start address of the bit vector and register $\$ \mathrm{a} 1$ holds the bit number to retrieve. The most-significant bit of the first byte is bit number 0 . When getbit returns register $\$ \mathrm{v} 0$ should be set to 0 or 1 .

For example, a 16 -bit bit vector is specified in the assembler below starting at the label bit_vector_start:
bit_vector_start:
.byte 0xc5, 0x1f
In binary this would be $1100010100011111_{2}$. If getbit were called with $\$ a 1=0$ then bit number zero, meaning the leftmost bit in $1100010100011111_{2}$, should be returned and so $\$ \mathrm{v} 0=1$. For $\$ \mathrm{a} 1=2$ a 0 should be returned.

Each memory location holds eight bits of the bit vector. For \$a1 values from 0 to 7 the bit will be in the byte at address $\$ \mathrm{a} 0$. For $\$ \mathrm{a} 1$ values from 8 to 15 the bit will be in the byte at address $\$ \mathrm{a} 0+1$, and so on.

When the the code in hw01.s is run (by pressing F9 for example) a testbench routine will call getbit several times. For each call the testbench will print the value returned by getbit (meaning the value of $\$ \mathrm{v} 0$ ), whether that value is correct, and if wrong, the correct value. At the end it will print the number of incorrect values returned by getbit, which hopefully will be zero when you're done.

See the checkboxes in the code for more information on what is expected.
The solution appears below:

```
getbit:
    srl $t0, $a1, 3 # Compute byte offset from $a0.
    add $t0, $a0, $t0 # Compute address of byte holding bit.
    lbu $t1, 0($t0) # Load that byte.
    andi $t2, $a1, 0x7 # Compute bit number within loaded byte.
    addi $t2, $t2, 24 # Find left shift amt that puts needed bit in MSB.
    sllv $t1, $t1, $t2 # Shift so that needed bit is most-significant.
    jr $ra
```

```
srl $v0, $t1, 31 # Shift so that needed bit is least significant.
```

The first two instructions, srl and add compute the address of the byte holding the needed bit. The byte is loaded and shifted left by enough so that the needed bit is in the most-significant position of 32 -bit register t 1 . The register is then shifted right by 31 bits so that the needed bit is in the least-significant position and all other bits are zero.
(b) The bit vector used by the testbench is specified with:

```
bit_vector_start: # Note: MIPS is big-endian.
    .byte 0xc5, 0x1f
    .half 0x05af
    .word 0xedcba987
    .ascii "1234"
bit_vector_end:
```

The assembler will convert the lines following data directives .byte, .half, .word, and . ascii into binary and place them in memory. The total size will be $2 \times 1+2+4+3=11$ bytes. For the purposes of this problem those 11 bytes form a $11 \times 8=88$-bit bit vector. In most circumstances for something like the bit vector above one would use the same kind of data directive for all data, say using only .byte, but mixing directives is not wrong and in some cases may be convenient for example when the bit vector is constructed by concatenating pieces of different sizes and types. Note that the kind of data directives used above does not affect how getbit is written.

Following the bit vector are the tests for the testbench. For each test there is one line consisting of a bit number and the expected return value. For example, the second test sets $\$ \mathrm{a} 1=4$ and expects a return value of $\$ \mathrm{v} 0=0$.

```
testdata:
```

```
.half 0, 1
.half 4, 0
.half 10, 0
```

Add a test to the testdata data to test the part of the bit vector specified using .ascii "123". The test should be written for .ascii "123" and should report an error if the directive were changed to .ascii "213".

The 1234 in the data area above follows two bytes, one half, and one word. The total size of these is $2 \times 1+2+4=$ 8 bytes. So the 1 in 1234 starts at bit $8 \times 8=64$, which is the most-significant bit of the 1 . The ASCII for 1 is $31_{16}$ and 2 is $32_{16}$, these differ in the least-significant bit, so we need to check bit $64+7=71$. It will be 1 for 1 . So we add data items for checking bit 71 for a 1 :

```
testdata:
    .half 71, 1 # Solution to 1b.
    .half 0, 1
    .half 4, 0
    .half 10, 0
```

Spring 2021

## 非 LSU EE 4720 Spring 2021 Homework 1 －－SOLUTION

非
非

## 非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非

## 非 Problem 1

非 Instructions：https：／／www．ece．lsu．edu／ee4720／2021／hw01．pdf
非 Solution narrative：https：／／www．ece．lsu．edu／ee4720／2021／hw01 sol．pdf

## ．text

getbit：

```
非 Register Usage
非
## CALL VALUES
# $a0: Address of start of array.
## $a1: Bit number to retrieve.
非
## RETURN
非 $v0: This bit. (Zero or one.)
非
# Note:
#⿰⿰三丨⿰丨三一隹 Can modify $t0-$t9, $a0-$a3
非
# [\checkmark] Test code should show 0 errors.
# [\checkmark] Code should be correct.
## [\checkmark] Code should be reasonably efficient.
# [\checkmark] Do not use pseudoinstructions except for nop and la.
非 SOLUTION -- Problem 1a.
非
srl $t0, $a1, 3 非 Compute byte offset from $a0.
add $t0, $a0, $t0 非 Compute address of byte holding bit.
lbu $t1, 0($t0) 非 Load that byte.
andi $t2, $a1, 0x7 非 Compute bit number within loaded byte.
addi $t2, $t2, 24 非 Find left shift amt that puts needed bit in MSB.
sllv $t1, $t1, $t2 非 Shift so that needed bit is most-significant.
jr $ra
srl $v0, $t1, 31 非 Shift so that needed bit is least significant.
```

```
        .data
        .align 4
    bit_vector_start: 非 Note: MIPS is big-endian.
        .byte 0xc5, 0x1f
        .half 0x05af
        .word 0xedcba987
        .ascii "1234"
bit_vector_end:
testdata:
    非 SOLUTION -- Problem 1b.
    #
    # The "1234" above follows two bytes, one half, and one word.
    非 The total size of these is 2*1 + 2 + 4 = 8 bytes. So the
    #⿰⿰三丨⿰丨三一价" in "1234" starts at bit 8 * 8 = 64, which is the
    #⿰⿰三丨⿰丨三一利 most-significant bit of the "1". The ASCII for "1" is 0x31
    # and "2" is 0x32, these differ in the least-significant bit,
    非 so we need to check bit 64 + 7 = 71. It will be 1 for "1".
    .half 71, 1 非 Solution to 1b.
```

    .half 0, 1
    .half 4, 0
    .half 10, 0
    .half 16, 0, 20,0, 21,1, 非 The part specified using ".half"
    .half 32,1 35,0, 39,1, 63,1, 非 The part specified using ".word"
    .half 5, 1
    .half 6, 0
    .half 1, 1
    .half 2, 0
    .half 11, 1
    .half 12, 1
    .half 13, 1
    .half 3, 0
    .half 7, 1
    .half 8, 0
    .half 9, 0
    .half 14, 1
    .half 15, 1
    .half -1, -1
    msg＿good：
.asciiz "Bit number: \%/a1/3d Val: \%/s4/1d, correct\n"
msg_bad:
.asciiz "Bit number: \%/a1/3d Val: \%/s4/1d, wrong. Correct val: \%/s5/1d\n"
msg_done:
.asciiz "Done with tests, \%/s6/d errors.\n";
.text
.globl __start
__start:
la \$s0, bit_vector_start
la \$s1, testdata
addi \$s6, \$0, 0
TB_LOOP:
addi \$a0, \$s0, 0
lh \$a1, 0(\$s1)
bltz \$a1, TB_DONE
nop
jal getbit
nop
addi \$s4, \$v0, 0
lhu \$s5, 2(\$s1)
la \$a0, msg_good
beq \$v0, \$s5, TB_CORRECT
lh \$a1, 0(\$s1)
la \$a0, msg_bad
addi \$s6, \$s6, 11
TB_CORRECT:
addi \$v0, \$0, 11
syscall
nop
j TB_LOOP
addi \$s1, \$s1, 4
TB_DONE:
la \$a0, msg_done
addi \$v0, \$0, 11
syscall
nop
addi \$v0, \$0, 10
syscall
nop

Problem 0: Follow the instructions for class account setup and for homework workflow in https://www.ece.lsu.edu/ee4720/proc.html. Review the comments in hw02.s and look for the area labeled "Problem 1."

Those who want to start before getting to the lab can find the assembler for the entire assignment at https://www.ece.lsu.edu/ee4720/2021/hw02.s.html. For MIPS references see the course references page,
https://www.ece.lsu.edu/ee4720/reference.html. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in https://www.ece.lsu.edu/ee4720/proc.html.

This Assignment
One goal of this assignment is to build assembly language proficiency by working with data at different sizes and by traversing a tree. The sizes are bits for the compressed text, words for the array of compressed text, half (2-byte) for the tree, and bytes for the dictionary. Another goal is to provide a starting point for architectural improvements. That is, ISA and hardware changes to make code go faster.

## Huffman Compression Background

One way to compress data is to divide it up into pieces, compute a Huffman coding for the pieces, then replace each piece with its Huffman code. The size of a Huffman code can vary from 1 bit (yes, just one), to an arbitrarily long bit vector. Pieces that appear more frequently in the original text will have short codes and pieces that appear less frequently will have longer codes. Consider a file containing English text, such as the source file for the Homework 1 handout. One way of dividing it to pieces is to make each character a piece. For Homework 1 a space was the most frequent piece ( 258 times) followed by the letter "e" ( 174 times). They received codes $100_{2}$ and $1110_{2}$, each of which is shorter than the eight bits used to encode each in the original file. The character " 8 " appears just once and gets a long encoding, $11011000001_{2}$. The compressed data consists of a concatenation of all of the codes. So "e e" would be encoded $11101001110_{2}$. The encoded data does not contain any separators between the pieces. To decode it one needs to first assume the code is one bit long, see if such a code exists, if not try two bits, and so on. So for the example one would first look for a code for $1_{2}$. If it didn't exist (and it shouldn't) one checks for $11_{2}$, which also shouldn't exist, neither does $111_{2}$ (the third try). One the fourth try we look for $1110_{2}$ and find that this is a code and the value is "e". The de-coding can continue by trying $1_{2}$, $10_{2}$, and finally $100_{2}$ which is the code for a space.

## Huffman Huff Tree Format for This Assignment

This assignment will use a format in which text is compressed into three arrays, the compressed text, starting at huff_compressed_text_start, a dictionary of strings, starting at huff_dictionary, and the Huff Tree (a lookup tree), at huff_tree.

The compressed text is a long bit vector. As with Homework 1, bits are numbered in big-endian order. The compressed text is specified using words but of course can be read using other sizes. The dictionary of strings consists of a bunch of null-terminated strings. The Huff Tree is used to decode compressed pieces. It is traversed using bits of the compressed text ( 0 for left child, 1 for right child) and a leaf provides either an index into the dictionary or a character.

Consider the following excerpt from the homework file:
huff_compressed_text_start:
.word 0xd9ac96d8, 0x10b75d4f, 0xa06510d1, 0x7d9961e3, 0xeb6f31f1

```
# Encoding: .word BIT_START, BIT_END, TREE_POS, DICT_POS, FRAG_LENGTH
huff_debug_samples:
# 0: 0 11011 -> "\n"
    .word 0, 5, 0x2ee, 0xa, 1;
# 0: 5 001101 -> " ."
    .word 5, 11, 0x32, 0x16, 9;
# 0:11 0110010010 -> "text"
    .word 11, 21, 0x110, 0x27b, 4;
# 0:21 11011 -> "\n"
    .word 21, 26, 0x2ee, 0xa, 1;
# 0:26 011000000 -> "histo"
    .word 26, 35, Oxea, Oxce, 5;
# 1: 3 1000010 -> ":\n"
    .word 35, 42, 0x1d6, 0x2e, 2;
```

The compressed text is shown as 32 -bit words, under huff_compressed_text_start and as an aid in debugging, the start of the same text is shown under huff_debug_samples. The first piece, $11011_{2}$, encodes a line feed character (we can see that by looking at the comment). The second piece, $001101_{2}$, encodes " ." (spaces followed by a period). The first piece is in bits 0 to 4 (inclusive) of the compressed text, and the second piece is in bits 5 to 10. The hexadecimal digits of the compressed text can be found by concatenating the compressed pieces and then grouping them into four-bit hex digits: $110110011010110010010 \rightarrow 110110011010110010010 \rightarrow \mathrm{~d} 9$ a c 9 ?. That matches the start of the compressed text shown under huff_compressed_text_start.

For this assignment a piece, for example 11011, is decoded by traversing the huff_tree. Each node in the huff_tree is 16 bits and can be one of three possible kinds: A leaf encoding a character, a leaf encoding a dictionary entry, or an internal node (with a left and right child). If the value of a node is $<128$ it is a leaf encoding a character. Otherwise if the value of a node is $>=0 \times 7000$ it is a leaf encoding a dictionary entry. Otherwise it is an internal node.

```
huff_tree:
# Huffman Lookup Tree
#
huff_tree: # Note: Most entries omitted.
    .half 0x01fa # Tree Idx 0 Pointer to right child.
    .half 0x011d # Tree Idx 1 0 Pointer to right child.
# [Many entries not shown.]
    .half 0x028c # Tree Idx 378 1 Pointer to right child
# [Many entries not shown.]
    .half 0x02e2 # Tree Idx 524 11 Pointer to right child.
    .half 0x02c7 # Tree Idx 525 110 Pointer to right child.
# [Many entries not shown.]
    .half 0x02e1 # Tree Idx 583 1101 Pointer to right child.
# [Many entries not shown.]
    .half 0x000a # Tree Idx 609 11011 Literal "\n"
```

The Huff Tree is an array of nodes, each a 16 -bit value. Let $T$ denote such an array. The root is $T[0]$. Let $i$ indicate some position in the tree and $n=T[i]$ denote the node at position $i$. The assembler data above shows some elements of a Huff Tree. (The entire tree can be found in hw02.s.) The numbers in binary (following the Tree Idx) show the path to that node.

If $n<128$ it is a leaf node encoding a character, and the ASCII value is $n$. If $n \geq 7000_{16}$ then the node is a leaf encoding a dictionary entry. The address of the first character of the dictionary entry is huff_dictionary $+\mathrm{n}-0 x 7000$. The strings in the dictionary are null-terminated.

Let $n=T[i]$ be a non-leaf node, so that $n \geq 128$ and $n<7000_{16}$. Its left child is at $T[i+1]$ and its right child is at $T[n-128]$.

Here is how piece 11011 of the compressed text would be decoded based on the data in the example above. Start at the root, retrieving $T[0]$. The value is $1 f a_{16}$, which is an internal node. The first bit of 11011 is 1 so we traverse the right child which is at $1 f a_{16}-80_{16}=17 a_{16}=378$. The entry at tree index 378 (based on the table) is $28 c_{16}$ which again is an internal node. The second bit of the piece is 1 so we compute the index of the right child: $28 c_{16}-80_{16}=20 c_{16}=524$. The next compressed bit is zero so we proceed to the left child, at index $524+1$. The tree excerpt above includes the entry leading to the leaf node.

The routine below (which can be found in huff-decode.cc in the homework package) decodes the piece starting at bit bit_offset and writes the decoded piece at dcd_ptr.

```
void
hdecode(HData& hd, int& bit_offset, char*& dcd_ptr)
{
    // Decode one piece, starting at bit position bit_offset and
    // write decoded piece starting at dcd_ptr.
    // hd.huff_compressed: Compressed text. An array of 32-bit values.
    // hd.huff_tree: A tree used to decode the compressed pieces.
    // hd.huff_dictionary: Decompressed pieces.
    // Start lookup at root of Huffman tree (tree_idx = 0).
    //
    int tree_idx = 0;
    while ( true )
        {
        // Retrieve node.
        uint16_t node = hd.huff_tree[tree_idx];
        if ( node < 128 )
            {
                // Node is a leaf encoding a character.
                char c = node; // Node value is an ASCII character.
                *dcd_ptr++ = c; // Write character to decoded text pointer ..
                return; // .. and return.
            }
        else if ( node >= 0x7000 )
            {
                // Node is a leaf holding an index into the dictionary.
                // Compute dictionary index.
                int idx = node - 0x7000;
```

```
            // Compute address of first character of dictionary entry.
            char* str = hd.huff_dictionary + idx;
            // Copy the dictionary entry.
            while ( *str ) *dcd_ptr++ = *str++;
            return;
        }
        else
            {
            // Node is not a leaf, need to set tree_idx to the index of
            // either the left or right child of the node. The left
            // child is used if the next bit of compressed text is zero
            // and the right child is used if the next bit of compressed
            // text is 1.
            // Get the next bit of compressed text.
            //
            int comp_idx = bit_offset / 32; // Index of word in huff_compressed.
            int bit_idx = bit_offset % 32; // Index of bit. MSB is 0.
            uint32_t comp_word = hd.huff_compressed[ comp_idx ];
            // Move needed bit to LSB in a way that sets other bits to zero.
            bool bit = comp_word << bit_idx >> 31;
            bit_offset++;
            if ( bit )
            {
                    // Set tree_idx to index of the right child.
                    tree_idx = node - 128;
            }
        else
            {
            // Set tree_idx to index of the left child.
            tree_idx++;
            }
        }
    }
}
```


## Homework Package

The homework package consists of files to help with your solution and to satisfy curiosity. Your solution, of course, goes in $\mathrm{hw} 02 . \mathrm{s}$, which is in the usual SPIM assembler format for this class.

The Huffman compression was performed by the huff perlscript. To compress MYFILE invoke it using ./huff MYFILE. With no arguments it compresses itself. It will write two files, encoded.s and encoded.h. The contents of encoded.s could be copied into the hw02.s (replacing what's there). Do this if you'd like to run your code on some other input.

File huff-decode.cc is a C++ routine that includes encoded.h and decodes it. It needs to
be re-built for each new input file. (Sorry, I've already spent too much time on the assignment.) Here is how it might be used on a new file:

```
[koppel@dmk-laptop hw02]$ ./huff ../../hw02.tex
File ../../hw02.tex
Words 366 Codes 366 Resorts 11
[koppel@dmk-laptop hw02]$ gmake -j 4
g++ --std=c++17 -Wall -g huff-decode.cc -o huff-decode
[koppel@dmk-laptop hw02]$ ./huff-decode
Decoded:
\magnification 1095
% TeXize-on
\input r/notes
```

The assignment was created by compressing histo-bare.s.

Problem 1: Complete routine hdecode so that it decodes the piece of Huffman-compressed text starting at bit number a1, writes the decoded text to memory starting at the address in a2, and sets registers v0, v1, a1, and a2 as described below. (Yes, a0 is unused.)

Use symbol huff_compressed_text_start for the address of the start of the compressed text, huff_tree for the address of the start of the Huff Tree, and huff_dictionary for the address of the start of the dictionary.

When hdecode returns set v0, v1, a1, and a2 as follows. Set a1 to the next bit position to use. For example, if the compressed piece were 3 bits and hdecode were called with a1=100 then when hdecode returns a1 should be set to 103. Set a2 to the address at which to write the next decoded character. For example, if the decoded text is 9 characters (not including the null) and initially a2=0x1000 then when hdecode returns a2 should be set to $0 \times 1009$. When hdecode returns v0 should be set to the address of the leaf in the Huff Tree that was used and v1 should be set to either the address of the dictionary entry used or the value of the character.

Note that the return values of a1 and a2 are useful because they are at the values needed to call hdecode again for the next piece. The return values of v0 and v1 are for debugging.

When hw02.s is run hdecode will be called multiple times, the return values checked, and the results printed on the console. It will be called for the first 200 pieces, or until there are three errors, whichever is sooner. A tally of errors is printed at the end, followed by the decoded text.

Pay attention to the error messages. Once syntax and execution errors are fixed, debug your code by tracing. To trace start the simulator using Ctrl F9 if running graphically or just F9 non-graphically. At the spim prompt type step 50 to execute the next 50 instructions. The trace shows line numbers of source assembly to the right of the semicolon. It also shows changed register values.

Single stepping is most useful when the first piece fails, which is likely to happen at first. But before long it will be correct and so viewing the trace will be a pain. To have the testbench start at your erroneous piece first locate the piece after label huff_debug_samples. The first number after . word is the Bit Position referred to in the "Decoding of.." message. Copy that line (perhaps with the comment above it) to just below the label huff_debug_samples.

Two solutions have been prepared and placed in the homework directory, /home/faculty/koppel/pub/ee4720/hw/2021/hw02. The solution in hw02-sol-easy.s (and at https://www.ece.lsu.edu/ee4720/2021/hw02-sol-easy.s.html) is easier to understand but is not as
fast as it could be. The solution in hw02-sol.s (and at
https://www.ece.lsu.edu/ee4720/2021/hw02-sol.s.html) is faster but is harder to understand.
The easy solution executes 25757 instructions and has an efficiency of 22.2 instructions per bit. The faster solution executes 19351 instructions and has an efficiency of 16.7 instructions per bit. (For both measures lower numbers are better.)

Both solutions start by loading the first needed word of compressed text into t 8 and shifting it so that the next needed bit is in the most-significant position. After that the main loop (TREE_LOOP) is entered. Each time a bit is extracted t8 is shifted left so that the next bit is in the MSB position. When (if) all bits are used a new word of compressed text is loaded into t8. Grading note: most solutions would load the compressed text each time a bit was needed.

This assignment was chosen to reinforce understanding of how memory addresses relate to arrays and their indices. Consider $\mathrm{C}++$ statement $\mathrm{x}=\mathrm{a}$ [i];. This loads the element at index i of array a into variable x . A MIPS assembly language equivalent will use some kind of a load instruction to retrieve a [i], the kind of load depends on the size of the elements of a. For example, if a were an array of 4 -byte integers then a lw would be used, if a were an array of 2-byte short integers then a 1 h would be used, etc. The memory address of the load would be computed by adding a to $i$ times the size of an element ( 4 for integers, 2 for shorts, etc.). The solution includes access to arrays of varying size. The huff table uses two-byte elements.

See the code comments for additional description of the solution.

```
    .text
hdecode:
## Register Usage
#
# CALL VALUES
# $a0: Unused
# $a1: Bit position to start at.
# $a2: Address to write decoded data.
#
# RETURN
# $a1: Bit position following encoded piece.
# $a2: Address following decoded piece.
# $v0: Address of leaf of huff tree.
# $v1: Address of dict entry or value of literal character.
```


## \#\# SOLUTION - Easy

```
# Load the word of compressed text that has the next bit we need.
#
srl $t4, $a1, 5 # Determine word index of next word to read.
sll $t6, $t4, 2 # Determine byte offset of next word to read.
la $t5, huff_compressed_text_start
add $t6, $t6, $t5 # Compute address of next word to read.
lw $t8, 0($t6) # Read next word of compressed text.
# Shift the next bit into the most-significant position of $t8.
#
andi $t7, $a1, 0x1f # Determine bit number of next bit to examine.
sllv $t8, $t8, $t7 # Put needed bit into most-significant position.
# It's possible that we'll examine all the bits in t8, at
```

```
# which time the next word needs to be loaded. To check
# whether this has occurred set $t9 to the bit number of the
# first bit in the next word.
#
sll $t9, $t4, 5 # Compute bit number at which a new word ..
addi $t9, $t9, 0x20 # .. will need to be loaded.
la $v0, huff_tree # $v0 will be used for the current node.
addi $t3, $v0, 0 # Note: t3 will not be changed.
TREE_LOOP:
# Live Registers at This Point in Code
#
# $t3: Address of root of huff tree.
# $v0: Address of current node.
#
# $a1: Bit number of next bit (of compressed text) to examine.
# $t8: Compressed text. The next bit to examine is the MSB.
# $t9: Bit number at which the next word must be loaded into t8.
# $t6: Address of the current word of compressed text.
#
# $a2: Address at which to write next decoded character.
# Load value of current node into $t4.
#
lhu $t4, 0($v0)
# Check whether current node is a character leaf.
#
sltiu $t5, $t4, 128
bne $t5, $0, CHAR_LEAF
nop
# Check whether current node is a dictionary leaf.
#
sltiu $t5, $t4, 0x7000
beq $t5, $0, DICT_LEAF
nop
NOT_LEAF:
    # Current node is an internal (non-leaf) node, so:
    # - Get next bit of compressed text.
    # - Based on the value of that bit descent to left or right child.
    # Check whether we already have the bit we need.
#
bne $a1, $t9, GET_NEXT_BIT
nop
```

```
# Load next word of compressed text.
#
addi $t6, $t6, 4 # Update address ..
lw $t8, 0($t6) # .. and load the word.
addi $t9, $t9, 0x20 # Also update t9, the load-next-word value.
GET_NEXT_BIT:
    # Put the next bit into $t5 and shift next-next bit to MSB.
    #
    slt $t5, $t8, $0 # Extract the most-significant bit of $t8 ..
    sll $t8, $t8, 1 # .. and shift the current word by 1.
    # Based on value of next bit descend to either right or left child.
    #
    bne $t5, $0, RCHILD
    addi $a1, $a1, 1 # Increment the bit number of the next bit.
# Descend to Left Child.
#
# The left child is right after the current node.
#
j TREE_LOOP
addi $v0, $v0, 2
RCHILD:
```

```
# Descend to Right Child.
```


# Descend to Right Child.

# 

# 

# The address of the right child is computed using current node value.

# The address of the right child is computed using current node value.

# 

# 

addi \$t4, \$t4, -128 \# Compute index of right child. (0 is root, etc.)
addi \$t4, \$t4, -128 \# Compute index of right child. (0 is root, etc.)
sll \$t4, \$t4, 1 \# Multiply index by 2 because nodes are two bytes.
sll \$t4, \$t4, 1 \# Multiply index by 2 because nodes are two bytes.
j TREE_LOOP
j TREE_LOOP
add \$v0, \$t3, \$t4 \# Add byte offset to address of root.
add \$v0, \$t3, \$t4 \# Add byte offset to address of root.
CHAR_LEAF:
\# Node is a leaf. Its value is a character to append.

# 

addi \$v1, \$t4, 0
sb $t4, O($a2) \# Write the character ..
jr \$ra \# .. and we're done ..
addi \$a2, \$a2, 1 \# .. after we increment the pointer.
DICT_LEAF:
\# Node is a leaf. Its value is a byte offset into dictionary.
\#
addi \$t4, \$t4, -0x7000
la \$t0, huff_dictionary
add \$t4, \$t4, \$t0
add \$v1, \$t4, \$0

```
```

    # Copy string from dictionary to output.
    CPY_LOOP:
lb $t5, 0($t4)
addi \$t4, \$t4, 1
sb $t5, 0($a2)
bne \$t5, \$0, CPY_LOOP
addi \$a2, \$a2, 1
jr \$ra
addi \$a2, \$a2, -1

```

The faster solution fills more delay slots and employs other techniques to reduce instruction count.
Effort was focused on the code descending down non-leaf (internal) nodes in the tree, since that it executed most frequently. So after loading a node the code first checks for a non-leaf, then for a leaf. The check for a non-leaf uses an unsigned comparison, sltiu, to test for values between 128 and \(0 x 7000\) using a single comparison. To do s0 t 4 is set to the node value, v1, minus 128. A sltiu t5, t4, \(0 x 6 f 80\) will set t5 to true if t 4 is between zero and 0 x 6 f 80 and false if t 4 is less than zero (negative numbers are treated as large positive numbers by sltiu) or \(\geq 0 \mathrm{x} 6 \mathrm{f} 80\).
```

    .text
    hdecode:

## Register Usage

# 

# CALL VALUES

# \$a0: Unused

# \$a1: Bit position to start at.

# \$a2: Address to write decoded data.

# 

# RETURN

# \$a1: Bit position following encoded piece.

# \$a2: Address following decoded piece.

# \$v0: Address of leaf of huff tree.

# \$v1: Address of dict entry or value of literal character.

# 

```

\section*{\#\# SOLUTION - Faster}
```


# Load the word of compressed text that has the next bit we need.

# 

srl \$t4, \$a1, 5 \# Determine word index of next word to read.
sll \$t6, \$t4, 2 \# Determine byte offset of next word to read.
la \$t5, huff_compressed_text_start
add \$t6, \$t6, \$t5 \# Compute address of next word to read.
lw $t8, 0($t6) \# Read next word of compressed text.

# Shift the next bit into the most-significant position of \$t8.

# 

andi \$t7, \$a1, 0x1f \# Determine bit number of next bit to examine.
sllv \$t8, \$t8, \$t7 \# Put needed bit into most-significant position.

# It's possible that we'll examine all the bits in t8, at

```
```


# which time the next word needs to be loaded. To check

# whether this has occurred set \$t9 to the bit number of the

# first bit in the next word.

# 

sll \$t9, \$t4, 5 \# Compute bit number at which a new word ..
addi \$t9, \$t9, 0x20 \# .. will need to be loaded.
la \$t3, huff_tree
addi \$v0, \$t3, -2 \# Compensate for crossing left-child code below.
TREE_LOOP_L:
\# Descend to Left Child. (Except in first iteration.)
\#
addi \$v0, \$v0, 2
TREE_LOOP:
\# Live Registers at This Point in Code
\#
\# \$t3: Address of root of huff tree.
\# \$v0: Address of current node.
\#
\# \$a1: Bit number of next bit (of compressed text) to examine.
\# \$t8: Compressed text. The next bit to examine is the MSB.
\# \$t9: Bit number at which the next word must be loaded into t8.
\# \$t6: Address of the current word of compressed text.

# 

# \$a2: Address at which to write next decoded character.

# Load value of current node into \$v1

# 

lhu $v1, 0($v0)

# First check for a non-leaf node, since that's most common.

# 

addi \$t4, \$v1, -128 \# Compute index of right child.
sltiu \$t5, \$t4, 0x6f80 \# Note: Unsigned comparison.
bne \$t5, \$0, NOT_LEAF
sll \$t7, \$t4, 1 \# Compute byte offset of right child.

# The node is a leaf. Check if it's a character or dictionary

# entry and branch to the respective code.

# 

bltz \$t4, CHAR_LEAF
sb $v1, O($a2) \# Write the character just in case.
j DICT_LEAF
addi \$t4, \$v1, -0x7000
NOT_LEAF:
\# Current node is an internal (non-leaf) node, so:

```
```


# - Get next bit of compressed text.

# - Based on the value of that bit descend to left or right child.

# Check whether we already have the bit we need.

# 

bne \$a1, \$t9, EXAMINE_NEXT_BIT
addi \$a1, \$a1, 1

# Load next word of compressed text.

# 

addi \$t6, \$t6, 4 \# Update address ..
lw $t8, 0($t6) \# No more bits, load a new word.
addi \$t9, \$t9, 0x20 \# .. and overflow value.
EXAMINE_NEXT_BIT:
\# Note: Next bit is in the MSB of t8. The bgez \$t8 is taken if
\# the MSB of \$t8 is zero.

# 

bgez \$t8, TREE_LOOP_L
sll \$t8, \$t8, 1 \# Shift left so that next bit is in MSB pos.
RCHILD:
\# Descend to Right Child.

# 

# Register \$t7 already contains the byte offset. Just add it

# to the root to get the address of the right child.

# 

j TREE_LOOP
add \$v0, \$t3, \$t7
CHAR_LEAF:
\# Node is a leaf. Its value is a character to append.
\#
\# Note: char already written by sb above.
\#
jr \$ra \# .. and we're done ..
addi \$a2, \$a2, 1 \# .. after we increment the pointer.
DICT_LEAF:
\# Node is a leaf. Its value is a byte offset into dictionary.
\#
la \$t0, huff_dictionary
add \$v1, \$t4, \$t0
\# Copy string from dictionary to output.
lb $t5, 0($v1)
addi \$t4, \$v1, 1
CPY_LOOP:
sb $t5, 0($a2)

```
```

addi \$a2, \$a2, 1
lb $t5, 0($t4)
bne \$t5, \$0, CPY_LOOP
addi \$t4, \$t4, 1
jr \$ra
nop

```

\section*{}

\section*{婔 Problem 1}

非
```

.text

```

```


# 

## CALL VALUES

# \$a0: Unused

# \$a1: Bit position to start at.

# \$a2: Address to write decoded data.

非

# RETURN

\$a1: Bit position following encoded piece.

# \$a2: Address following decoded piece.

## \$v0: Address of leaf of huff tree.

# \$v1: Address of dict entry or value of literal character.

# 

# Note:

# Can modify registers $t0-$t9, $a0-$a3, \$v0, \$v1.

# DO NOT modify other registers.

# [\checkmark] The testbench should show 0 errors.

# [\checkmark] Code should be reasonably efficient.

# [\checkmark] The code should be clearly written.

非 [\checkmark] Comments should be written for an experienced programmer.

# [\checkmark] Do not use pseudoinstructions except for nop and la.

\#⿰⿰三丨⿰丨三⿰⿰三丨⿰丨三\mp@code{SOLUTION - Easy}

# Load the word of compressed text that has the next bit we need.

# 

srl \$t4, \$a1, 5 非 Determine word index of next word to read.
sll \$t6, \$t4, 2 非 Determine byte offset of next word to read.
la \$t5, huff_compressed_text_start
add \$t6, \$t6, \$t5 非 Compute address of next word to read.
lw $t8, 0($t6) 非 Read next word of compressed text.

# Shift the next bit into the most-significant position of \$t8.

# 

andi \$t7, \$a1, 0x1f 非 Determine bit number of next bit to examine.
sllv \$t8, \$t8, \$t7 非 Put needed bit into most-significant position.
\#⿰㇒一㐄二

## which time the next word needs to be loaded. To check

# whether this has occurred set \$t9 to the bit number of the

## first bit in the next word.

# 

sll \$t9, \$t4, 5 非 Compute bit number at which a new word ..
addi \$t9, \$t9, 0x20 非 .. will need to be loaded.
la \$v0, huff_tree 非 \$v0 will be used for the current node.
addi \$t3, \$v0, 0 非 Note: t3 will not be changed.

```
hdecode:

TREE＿LOOP：
```


# Live Registers at This Point in Code

# 

# \$t3: Address of root of huff tree.

# \$v0: Address of current node.

# \$a1: Bit number of next bit (of compressed text) to examine.

# \$t8: Compressed text. The next bit to examine is the MSB.

# \$t9: Bit number at which the next word must be loaded into t8.

# \$t6: Address of the current word of compressed text.

# 

# \$a2: Address at which to write next decoded character.

```
非 Load value of current node into \$t4.
非
lhu \$t4, 0(\$v0)
```


# Check whether current node is a character leaf.

# 

sltiu \$t5, \$t4, 128
bne \$t5, \$0, CHAR_LEAF
nop

# Check whether current node is a dictionary leaf.

非
sltiu \$t5, \$t4, 0x7000
beq \$t5, \$0, DICT_LEAF
nop

```
NOT_LEAF:
非 Current node is an internal (non-leaf) node, so:
非 - Get next bit of compressed text.
\# - Based on the value of that bit descend to left or right child.
\# Check whether we already have the bit we need.
非
bne \$a1, \$t9, GET_NEXT_BIT
nop
\# Load next word of compressed text.
非
addi \$t6, \$t6, 4 Update address ..
lw \$t8, 0(\$t6) 非 . . and load the word.
addi \$t9, \$t9, 0x20 非 Also update t9, the load-next-word value.
GET_NEXT_BIT:
    \# Put the next bit into \$t5 and shift next-next bit to MSB.
    非
slt \$t5, \$t8, \$0 ⿰⿰三丨⿰丨三一 Extract the most-significant bit of \$t8 ..
sll \$t8, \$t8, \(1 \quad ⿰ ⿰ 三 丨 ⿰ 丨 三 一\). . and shift the current word by 1.
\# Based on value of next bit descend to either right or left child.
非
bne \$t5, \$0, RCHILD
addi \$a1, \$a1, 1 Increment the bit number of the next bit.
\# Descend to Left Child.
非
\# The left child is right after the current node.
非
j TREE_LOOP
addi \$v0, \$v0, 2

RCHILD：
```


# Descend to Right Child.

# 

# The address of the right child is computed using current node value.

# 

addi \$t4, \$t4, -128 非 Compute index of right child. (0 is root, etc.)
sll \$t4, \$t4, 1 非 Multiply index by 2 because nodes are two bytes.
j TREE_LOOP
add \$v0, \$t3, \$t4 非 Add byte offset to address of root.

```
CHAR_LEAF:
非 Node is a leaf. Its value is a character to append.
非
addi \$v1, \$t4, 0
sb \$t4, 0(\$a2) 非 Write the character ..
jr \$ra 非 .. and we're done ..
addi \$a2, \$a2, 1 非 . . after we increment the pointer.
DICT_LEAF:
非 Node is a leaf. Its value is a byte offset into dictionary.
非
addi \$t4, \$t4, -0x7000
la \$t0, huff_dictionary
add \$t4, \$t4, \$t0
add \$v1, \$t4, \$0
\# Copy string from dictionary to output.
CPY_LOOP:
lb \$t5, 0(\$t4)
addi \$t4, \$t4, 1
sb \$t5, 0(\$a2)
bne \$t5, \$0, CPY_LOOP
```

addi \$a2, \$a2, 1
jr \$ra
addi \$a2, \$a2, -1

```

\section*{}

\section*{非 Testbench Routine}
```

    .data
    msg_full_text:
.asciiz "\nNumber correct %/s6/d, number wrong %/s5/d.\nDecoded Text:\n%/s0/s\n";
msg_piece_start:
.asciiz "** Decoding of Bit Position %/a1/d **\n"
msg_bit_pos_ok:
.asciiz "End bit pos ($a1 contents) correct: %/a1/d.\n"
msg_bit_pos_bad:
    .asciiz "End bit pos ($a1 contents) wrong: %/a1/d correct value is %/t0/d.\n"
msg_leaf_addr_ok:
.asciiz "Leaf address ($v0 contents) correct: %/t4/非.\n"
msg_leaf_addr_bad:
    .asciiz "Leaf address ($v0 contents) wrong: %/t4/非x correct value is %/t1/非.\n"
msg_dict_addr_ok:
.asciiz "Dict address ($v1 contents) correct: %/v1/非x Entry: \"%/v1/s\".\n"
msg_dict_addr_bad:
    .asciiz "Dict address ($v1 contents) wrong: %/v1/非x correct value is %/t3/非.\n"
msg_char_val_ok:
.asciiz "Char value ($v1 contents) correct: \"%/v1/c\"\n"
msg_char_val_bad:
    .asciiz "Char value ($v1 contents) wrong: %/v1/d (decimal) correct value is %/t3/d (decimal) or \"%/t3/c\".\n"
msg_char_copied_wrong:
.asciiz "Char value correct, \"%/v1/c\", but not copied, found \"%/\$t4/\".\n"
msg_copied_bad:
.asciiz "Text copied incorrectly starting at character %/t6/d: \"%/t7/s\"\n"
msg_early_exit:
.asciiz "** Error limit exceeded, exiting early.\n";
msg_timing:
.ascii "\nTotal insn %/s2/d, encoded bits %/t0/d, uncompressed bytes %/t1/d.\n"
.asciiz "Efficiency: %/f0/4.1f insn/bit %/f2/4.1f insn/byte.\n"
tb_timing_data:
.word 0 非 Instruction count.
.word 0 非 Sum of bits examined.
.word 0 非 Sum of characters in decoded text.
.text
.globl __start
__start:
la \$s3, huff_debug_samples
addi \$t0, \$0, 0
la \$a2, uncompressed
addi \$s2, \$0, 0 非 Total number of instructions.
addi \$s6, \$0, 0 非 Number correct
addi \$s7, \$0, 0 非 Number checked
TLOOP:
addi \$s7, \$s7, 1
lw $a1, 0($s3)
la \$a0, msg_piece_start
addi \$v0, \$0, 11
syscall
nop
addi \$s4, \$a2, 0
jal hdecode
mfc0 \$s1, \$9 非 Number of insns before hdecode.
mfc0 \$t1, \$9 非 Number of insns after hdecode.

```
```

addi \$t1, \$t1, -1
sub \$s1, \$t1, \$s1 非 Number of insns executed by hdecode.
add \$s2, \$s2, \$s1
sb $0, 0($a2)
addi \$s0, \$a0, 0
addi \$t4, \$v0, 0
lw $t0, 4($s3) 非 Bit end
lw $t1, 8($s3) 非 Byte offset into huff tree.
lw $t3, 12($s3) 非 Byte offset into dictionary.

# Compute and store total compressed bits and decompressed bytes.

# 

lw $t2, 0($s3)
sub \$t2, \$t0, \$t2
la \$a0, tb_timing_data
lw $t5, 4($a0) 非 Number of compressed bits.
add \$t5, \$t5, \$t2
sw $t5, 4($a0)
lw $t5, 8($a0) 非 Total number of decompressed bytes.
lw $t2, 16($s3) 非 Number of bytes in this piece.
add \$t5, \$t5, \$t2
sw $t5, 8($a0)

# Check ending compressed-text bit position.

非
la \$a0, msg_bit_pos_ok
beq \$t0, \$a1, TBIT_SHOW
nop
la \$a0, msg_bit_pos_bad
TBIT_SHOW:
addi \$v0, \$0, 11
syscall
nop

# Check address of huff tree leaf.

# 

la \$t2, huff_tree
add \$t1, \$t1, \$t2
la \$a0, msg_leaf_addr_ok
beq \$t1, \$t4, TLEAF_SHOW
nop
la \$a0, msg_leaf_addr_bad
TLEAF_SHOW:
syscall
nop
slti \$t4, \$v1, 128
bne \$t4, \$0, TCHAR_CHECK

# Check address of dictionary entry.

# 

la \$t2, huff_dictionary
add \$t3, \$t3, \$t2
la \$a0, msg_dict_addr_ok
beq \$v1, \$t3, TDICT_SHOW
nop
la \$a0, msg_dict_addr_bad
TDICT_SHOW:
syscall
nop
bne \$v1, \$t3, TCHECK_NEXT
nop
addi \$t6, \$t3, 0
addi \$t7, \$s4, 0
TCHECK_WORD_LOOP:
lbu $t4, 0($s4)
lbu $t5, 0($t3)
bne \$t4, \$t5, TCHECK_WORD_WRONG
addi \$s4, \$s4, 1
bne \$t5, \$0, TCHECK_WORD_LOOP
addi \$t3, \$t3, 1

# Word copied correctly.

addi \$s6, \$s6, 1

```
j TCHECK＿NEXT
nop
TCHECK＿WORD＿WRONG：
la \＄a0，msg＿copied＿bad
sub \＄t6，\＄t3，\＄t6
syscall
nop
j TCHECK＿NEXT
nop

非 Check value of decoded character．
非
TCHAR＿CHECK：
beq \＄v1，\＄t3，TCHAR＿SHOW
nop
la \＄a0，msg＿char＿val＿bad
syscall
nop
j TCHECK＿NEXT
nop

TCHAR＿SHOW：
\＃Check whether character copied correctly．
lbu \＄t4，0（\＄s4）
bne \＄t4，\＄v1，TCHAR＿COPIED＿WRONG
nop
\＃Character correct
addi \＄s6，\＄s6， 1
la \＄a0，msg＿char＿val＿ok
syscall
nop
j TCHECK＿NEXT
nop

TCHAR＿COPIED＿WRONG：
la \＄a0，msg＿char＿copied＿wrong
syscall
nop
TCHECK＿NEXT：
非 Exit loop if error threshold exceeded．
非
sub \＄s5，\＄s7，\＄s6
slti \＄t1，\＄s5， 3
bne \＄t1，\＄0，TCHECK＿CONTINUE
nop
la \＄a0，msg＿early＿exit
syscall
nop
j TLOOP＿EXIT
nop
TCHECK＿CONTINUE：
非 Advance to next piece．
非
addi \＄s3，\＄s3， 20
la \＄t0，huff＿compressed＿text＿start
slt \＄t1，\＄s3，\＄t0
bne \＄t1，\＄0 TLOOP
nop
非
非 End of Loop
TLOOP＿EXIT：
非 Compute number of incorrect pieces．
非
sub \＄s5，\＄s7，\＄s6

非 Show execution rate．
非
la \＄a0，msg＿timing
la \＄t4，tb＿timing＿data
```

lw $t0, 4($t4)
lw $t1, 8($t4)
mtc1 \$s2, \$f20
cvt.d.w \$£22, \$£20
mtc1 \$t0, \$f10
cvt.d.w \$f12, \$f10
mtc1 \$t1, \$f14
cvt.d.w \$f16, \$f14
div.d \$f0, \$f22, \$f12
div.d \$土2, \$土22, \$f16
addi \$v0, \$0, 11
syscall
nop

# Show decoded text.

# 

la \$a0, msg_full_text
la \$s0, uncompressed
addi \$v0, \$0, 11
syscall
nop
addi \$v0, \$0, 10
syscall
nop

```
data
.align 4
uncompressed:
.space 4000
uncompressed_end:
\# Data from file histo-bare.s
\# Code size \(14253+1679+5176=21108\) orig 40784 b, ratio 0.518 max 11
\# Words 188 Codes 188 Resorts 10
非 Compression Debug Samples
非
\# Encoding: .word BIT_START, BIT_END, TREE_POS, DICT_POS, FRAG_LENGTH
非
huff_debug_samples:
非 0: 011011 -> "\n"
    .word 0, 5, 0x2ea, 0xa, 1;
非 0: 5001011 ->
    .word 5, 11, \(0 \times 2 \mathrm{a}, 0 \times 16,9\);
\# 0:11 0110011000 -> "text"
    .word 11, 21, \(0 \times 14 \mathrm{c}, 0 \times 1 \mathrm{af}, 4\);
非 0:21 11011 -> "\n"
    .word 21, 26, 0x2ea, 0xa, 1;
非 0:26 011010010 -> "histo"
    .word 26, 35, 0x166, 0xc5, 5;
\# 1: 3 1000100 -> ":\n"
    .word 35, 42, 0x1de, 0x28, 2;
\# 1:10 11011 -> " n n"
    .word 42, 47, 0x2ea, 0xa, 1;
非 1:15 10101 -> "
    .word 47, 52, 0x21c, 0, 8;
非 1:20 1101000 -> "addi"
    .word 52, 59, 0x2b4, 0x23, 4;
非 1:27 01111 -> " \$"
    .word 59, \(64,0 \times 1 \mathrm{c} 2,0 \times 9,2\);
\# 2: \(0 \quad 10011\)-> "s"
.word 64, 69, 0x1f0, 0x73, 1;
非 2: 5000011 -> "1"
.word 69, 75, \(0 \times 18,0 \times 31,1\);
\# 2:11 00100 -> ", \$"
.word 75, 80, \(0 \times 24,0 x f, 3\);
\# 2:16 00010 -> "r"
.word 80, 85, \(0 \times 1 \mathrm{c}, 0 \times 72,1 ;\)
\# 2:21 01000 -> "a"
    .word 85, 90, \(0 \times 48,0 \times 61,1 ;\)
非 2:26 00111 -> ", "
    .word 90, 95, 0x40, 0xc, 2;
非 2:31 110000 -> "0"
    .word 95, 101, \(0 \times 24 \mathrm{c}, 0 \times 30,1 ;\)
\# 3: 5 101001000 -> " 非 "
            .word 128, 131, \(0 \times 2 \mathrm{ec}, 0 \times 20,1\);
\# 4: 3 0101110110 -> "copy"
    .word 131, 141, 0xb8, 0x1ca, 4
\# 4:13 111 -> " "
    .word 141, 144, 0x2ec, 0x20, 1;
\# 4:16 0011001 -> "of"
    .word 144, 151, \(0 \times 34,0 \times 37,2\).
非 4:23 111 -> "
    .word 151, 154, \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
\# 4:26 10111010 -> "the"
    .word 154, 162, \(0 \times 23 \mathrm{e}, 0 \times 43,3\)
非 5: 2 111 -> "
    .word 162, 165, \(0 \times 2 \mathrm{ec}, 0 \times 20,1\)
\# 5: 5 0101110000 -> "return"
    .word 165, 175, 0xa4, 0x1d6, 6
非 5:15 111 -> " "
    .word 175, 178, \(0 \times 2 \mathrm{ec}, 0 \times 20,1\);
\# 5:18 101110011 -> "address"
    .word 178, 187, 0x23a, 0x74, 7
非 5:27 1100010 -> ".\n"
    .word 187, 194, \(0 \times 250,0 \times 20,2\)
\# 6: 210101 -> "
    .word 194, 199, 0x21c, 0, 8;
非 6: 7 1101010010 -> "jal"
    .word 199, 209, 0x2c6, 0x197, 3;
非 6:17 111 -> "
    .word 209, 212, \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 6:20 011011101 -> "upper"
    .word 212, 221, 0x19e, 0xcb, 5;
非 6:29 111 -> "
    .word 221, 224, \(0 \times 2 \mathrm{ec}, 0 \times 20,1\)
非 7: 0 111-> " "
    .word 224, 227, \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 7: 3111 -> " "
.word 227, 230, \(0 \times 2 \mathrm{ec}, 0 \times 20,1\);
\# 7: 6 111 -> "
.word 230, 233, \(0 \times 2 \mathrm{ec}, 0 \times 20,1\);
\# 7: 9111 -> " "
.word 233, 236, \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
\# 7:12 111 -> " "
    .word 236, 239, 0x2ec, 0x20, 1
\# 7:15 111 -> " "
    .word 239, 242, 0x2ec, 0x20, 1;
7: 7:18 111 -> " "
    .word 242, 245, 0x2ec, 0x20, 1;
非 7:21 111 -> " "
    .word 245, 248, 0x2ec, 0x20, 1;
非 7:24 111 -> "
    .word 248, 251, \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 7:27 111 -> " "
    .word 251, 254, \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
\# 7:30 0101010 -> "非"
    .word 254, 261, \(0 \times 58,0 \times 23,1\);
非 8: 5 111-> "
    .word 261, 264, \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 8: 8 0110110001 -> "Convert"
    .word 264, 274, \(0 \times 182,0 \times 27 d, 7 ;\)
\# 8:18 111 -> " "
    .word 274, 277, 0x2ec, 0x20, 1
\# 8:21 10110010 -> "to"
    .word 277, 285, 0x228, 0x4d, 2
非 8:29 111 -> "
    .word 285, 288, 0x2ec, 0x20, 1;
\# 9: 0 011011101 -> "upper"
    .word 288, 297, \(0 \times 19 \mathrm{e}, 0 \times \mathrm{cb}, 5\);
非 9: 9111 -> "
    .word 297, 300, \(0 \times 2 \mathrm{ec}, 0 \times 20,1\)
非 9:12 1011000 -> "c"
    .word 300, 307, \(0 \times 224,0 \times 63,1\);
\# 9:19 01000 -> "a"
    .word 307, 312, \(0 \times 48,0 \times 61,1\);
\# 9:24 10011 -> "s"
    .word 312, 317, 0x1f0, 0x73, 1;
\＃9：29 10010－＞＂e＂
．word 317，322，0x1ee， \(0 \times 65,1 ;\)
非 10： 2 1100010－＞＂．\n＂ ．word \(322,329,0 \times 250,0 \times 20,2\)
\＃10： 9 10101－＞＂
word 329，334，0x21c，0， 8
非 10：14 1101000－＞＂addi＂
．word 334，341， \(0 \times 2 \mathrm{~b} 4,0 \times 23,4\)
非 10：21 01111－＞＂\＄＂ ．word 341，346，0x1c2，0x9，2；
\＃10：26 10011 －＞＂s＂ ．word 346，351， \(0 \times 140,0 \times 73,1\)
10：31 110000－＞＂0＂
．word \(351,357,0 \times 24 \mathrm{c}, 0 \times 30,1\)
非 11： 500100 －＞＂，\＄＂
．word 357，362， \(0 \times 24,0 \times 4,3\) ；
非 11：10 0101001 －＞＂a0＂
．word \(362,369,0 \times 54,0 \times 31,2\) ；
非 11：17 00111 －＞＂，＂
．word 369，374， \(0 \times 40,0 \times c, 2\) ；
非 11：22 110000－＞＂0＂ ．word 374，380， \(0 \times 24 c, 0 \times 30,1\)
非 11：28 101001000－＞＂非＂ ．word 380，389，0x204，0x9d， 6
非 12： 50101100111 －＞＂Make＂
．word 389，399，0x7c，0x218， 4
非 12：15 111 －＞＂
．word 399，402，0x2ec，0x20， 1
非 12：18 01000－＞＂a＂
．word 402，407， \(0 \times 48,0 \times 61,1\)
非 12：23 111 －＞＂＂
．word 407，410， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\)
非 12：26 0101110110－＞＂copy＂
．word 410，420，0xb8，0x1ca，4；
非 13： 4 111－＞＂＂
．word 420，423， \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 13：7 0011001 －＞＂of＂
．word 423，430， \(0 \times 34,0 \times 37,2\) ；
非 13：14 111 －＞＂＂
．word 430，433，0x2ec，0x20， 1
非 13：17 01100111－＞＂string＂ ．word 433，441，0x156，0x54，6；
非 13．25 111 －＞＂＂
．word 441，444，0x2ec， \(0 \times 20,1\)
非 13：28 0101101101－＞＂start＂
．word 444，454，0x94，0x238， 5
非 14：6 111 －＞＂
．word 454，457， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\)
\＃14：9 101110011 －＞＂address＂
．word 457，466， \(0 \times 23 \mathrm{a}, 0 \times 74,7\)
14 14：18 1100010－＞＂．\n＂
．word 466，473， \(0 \times 250,0 \times 20,2\) ；
14：25 10101－＞
．word 473，478， \(0 \times 21 \mathrm{c}, 0,8\)
非 14：30 1101000－＞＂addi＂
word 478，485， \(0 \times 2 \mathrm{~b} 4,0 \times 23,4\)
非 15： 501111 －＞＂\＄＂
．word 485，490，0x1c2，0x9，2；
非 15：10 0101001－＞＂a0＂
word 490，497， \(0 \times 54,0 \times 31,2\) ；
非 15：17 00100－＞＂，\＄＂
．word 497，502，0x24，0x£，3；
非 15：22 10011 －＞＂s＂
．word 502，507，0x1f0， \(0 \times 73,1\)
非 15：27 110000－＞＂0＂
．word 507，513， \(0 \times 24 \mathrm{c}, 0 \times 30,1\)
非 16： 100111 －＞＂，＇
．word 513，518， \(0 \times 40,0 \times c, 2\) ；
非 16：6 110000 －＞＂0＂
．word 518，524，0x24c， \(0 \times 30,1\)
非 16：12 101001000－＞＂非＂
．word 524，533， \(0 \times 204,0 \times 9 \mathrm{~d}, 6\) ；
\＃16：21 0110011011－＞＂R＂
．word 533，543， \(0 \times 154,0 \times 52,1\)
非 16：31 10010－＞＂e＂ ．word 543，548，0x1ee，0x65，1；
\＃17： 410011 －＞＂s＂ ．word 548，553，0x1f0， \(0 \times 73,1\)
\＃17： 901001 －＞＂t＂
．word 553，558， \(0 \times 4 \mathrm{a}, 0 \times 74,1 ;\)
非 17：14 101111 －＞＂о＂

非 17：20 00010－＞＂r＂
．word 564，569， \(0 \times 1 \mathrm{c}, 0 \times 72,1\) ；
\＃17：25 10010 －＞＂e＂
．word 569，574， \(0 \times 1 \mathrm{ee}, 0 \times 65,1\) ；
非 17：30 111 －＞＂＂
．word 574，577， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃18： 1 01100111－＞＂string＂
．word 577，585， \(0 \times 156,0 \times 5 f, 6\) ；
非 18： 9 111－＞＂＂
．word 585，588， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 18：12 0101101101－＞＂start＂
．word 588，598，0x94，0x238，5；
非 18：22 111 －＞＂＂
．word 598，601，0x2ec，0x20，1；
非 18：25 101110011 －＞＂address＂ ．word 601，610， \(0 \times 23 a, 0 \times 74,7\) ；
\＃19： 21100010 －＞＂．\n＂
．word 610，617， \(0 \times 250,0 \times 20,2\) ；
非 19： 9111 －＞＂＂
．word 617，620， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 19：12 111 －＞＂＂
．word 620，623， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃19：15 111 －＞＂＂
．word 623，626， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 19：18 111 －＞＂＂
．word 626，629， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 19：21 111 －＞＂＂
．word 629，632，0x2ec，0x20，1；
非 19：24 111 －＞＂＂
．word 632，635，0x2ec，0x20，1；
非 19：27 111 －＞＂＂
．word 635，638， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 19：30 111 －＞＂＂
．word 638，641， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 20： 1 11011－＞＂\n＂
．word 641，646， \(0 \times 2 \mathrm{ea}, 0 \times \mathrm{a}, 1\) ；
\＃20： 6 1101010100－＞＂LOOP＂
．word 646，656， \(0 \times 2\) ce， \(0 \times 179,4\) ；
\＃20：16 1000100－＞＂：\n＂
．word 656，663， \(0 \times 1 \mathrm{de}, 0 \times 28,2\) ；
\＃20：23 10101－＞＂
．word 663，668，0x21c，0，8；
非 20：28 101101 －＞＂l＂
．word 668，674， \(0 \times 22 \mathrm{c}, 0 \times 6 \mathrm{c}, 1\) ；
\＃21： 2 11001101－＞＂b＂
．word 674，682，0x29a，0x62，1；
非 21：10 01111 －＞＂\＄＂
．word 682，687， \(0 \times 1 \mathrm{c} 2,0 \times 9,2\) ；
非 21：15 0011000 －＞＂t0＂
．word 687，694， \(0 \times 32,0 \times 3 a, 2\) ；
非 21：22 00111 －＞＂，＂
．word 694，699，0x40，0xc，2；
非 21：27 110000－＞＂0＂
．word 699，705， \(0 \times 24 \mathrm{c}, 0 \times 30,1\) ；
非 22： 1 10111000－＞＂（\＄＂
．word 705，713， \(0 \times 234,0 \times 4 a, 2\) ；
非 22： 9 0101001－＞＂a0＂
．word 713，720， \(0 \times 54,0 \times 31,2\) ；
非 22：16 0101101001－＞＂）非
．word 720，730， \(0 \times 86,0 \times 24 \mathrm{f}, 9\) ；
\＃22：26 1100101101－＞＂Load＂
．word 730，740， \(0 \times 28 \mathrm{c}, 0 \times 156,4\) ；
非 23： 4 111－＞＂＂
．word 740，743，0x2ec，0x20，1；
非 23：7 1100101010－＞＂next＂
．word 743，753， \(0 \times 282,0 \times 174,4 ;\)
非 23：17 111 －＞＂＂
．word 753，756，0x2ec，0x20，1；
非 23：20 110101111 －＞＂character＂
．word 756，765，0x2e8，0x82，9；
\＃23：29 1100010－＞＂．\n＂
．word 765，772， \(0 \times 250,0 \times 20,2\) ；
非 24： 4 10101－＞＂
．word 772，777，0x21c，0，8；
\＃24： 9 101001001－＞＂beq＂
．word 777，786，0x206，0x99，3；
非 24：18 01111 －＞＂\＄＂
．word 786，791， \(0 \times 1 \mathrm{c} 2,0 \times 9,2\) ；
\＃24：23 0011000 －＞＂t0＂
．word 791，798， \(0 \times 32,0 \times 3 a, 2\) ；
\＃24：30 00100－＞＂，\＄＂
word 798，803，0x24，0xf，3；
非 25： 3 110000－＞＂0＂ ．word 803，809， \(0 \times 24 \mathrm{c}, 0 \times 30,1\) ；
非 25： 9 00111－＞＂，＂ ．word 809，814，0x40，0xc，2；
非 25：14 0101101010－＞＂DONE＂ ．word 814，824，0x8a，0x262，4；
非 25：24 0101110111－＞＂非＂ ．word 824，834，0xba，0x1c4，5；
非 26： 2 01100011000－＞＂J＂
．word 834，845， \(0 \times 126,0 \times 4 a, 1\) ；
\＃26：13 1010001－＞＂u＂
．word 845，852，0x1fc， \(0 \times 75,1\) ；
\＃26：20 1100011－＞＂m＂
．word 852，859，0x252，0x6d，1；
非 26：27 0101011 －＞＂p＂
．word 859，866， \(0 \times 5 a, 0 \times 70,1 ;\)
非 27： 2 111－＞＂＂
．word 866，869， \(0 \times 2\) ec， \(0 \times 20,1\) ；
非 27：5 0101111101－＞＂out＂
．word 869，879，0xd2，0x1e8，3；
非 27：15 111 －＞＂＂
．word 879，882，0x2ec，0x20，1；
非 27：18 0011001 －＞＂of＂
．word 882，889， \(0 \times 34,0 \times 37,2\) ；
非 27：25 111 －＞＂＂
．word 889，892， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 27：28 101101 －＞＂1＂
．word 892，898， \(0 \times 22 \mathrm{c}, 0 \times 6 \mathrm{c}, 1\) ；
⿰⿰三丨⿰丨三一 28： 2101111 －＞＂о＂
．word 898，904， \(0 \times 242,0 \times 6 \pm, 1\) ；
非 28： 8101111 －＞＂o＂
．word 904，910， \(0 \times 242,0 \times 6 \mathrm{f}, 1\) ；
非 28：14 0101011－＞＂p＂
．word 910，917， \(0 \times 5 a, 0 \times 70,1 ;\)
非 28：21 111 －＞＂＂
．word 917，920，0x2ec，0x20，1；
非 28：24 00000－＞＂i＂
．word 920，925，0xa，0x69，1；
非 28：29 1101001－＞＂f＂ ．word 925，932，0x2b6，0x66，1；
非 29： 4111 －＞＂＂ ．word 932，935，0x2ec，0x20，1；
\＃29： 7 01000－＞＂a＂
．word 935，940， \(0 \times 48,0 \times 61,1\) ；
\＃29：12 01001 －＞＂t＂
．word 940，945， \(0 \times 4 a, 0 \times 74,1\) ；
非 29：17 111 －＞＂＂
．word 945，948， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 29：20 10010－＞＂e＂
．word 948，953，0x1ee，0x65，1；
非 29：25 00011 －＞＂n＂
．word 953，958， \(0 \times 1 \mathrm{e}, 0 \times 6 \mathrm{e}, 1 ;\)
\＃29：30 1000010－＞＂d＂ ．word 958，965，0x1d6，0x64，1；
\＃30： 5111 －＞＂＂ ．word 965，968，0x2ec，0x20，1；
非 30： 80011001 －＞＂of＂ ．word 968，975， \(0 \times 34,0 \times 37,2\) ；
非 30：15 111 －＞＂＂
．word 975，978， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 30：18 01100111 －＞＂string＂ ．word 978，986， \(0 \times 156,0 \times 5 f, 6\) ；
\＃30：26 1100010 －＞＂．\n＂ ．word 986，993， \(0 \times 250,0 \times 20,2\) ；
非 31： 1 10101－＞＂
．word 993，998，0x21c，0，8；
\＃31： 6 1101000－＞＂addi＂
．word 998，1005，0x2b4，0x23，4；
非 31：13 01111 －＞＂\＄＂
．word 1005，1010，0x1c2，0x9，2；
非 31：18 0101001 －＞＂a0＂
．word 1010，1017， \(0 \times 54,0 \times 31,2\) ；
非 31：25 00100 －＞＂，\＄＂
．word 1017，1022，0x24，0xf，3；
非 31：30 0101001 －＞＂a0＂ ．word 1022，1029， \(0 \times 54,0 \times 31,2 ;\)
非 32： 500111 －＞＂，＂
．word 1029，1034，0x40，0xc，2；
非 32：10 000011 －＞＂1＂
．word 1034，1040， \(0 \times 18,0 \times 31,1 ;\)
非 32：16 101001000－＞＂非＂
．word 1040，1049，0x204，0x9d，6；
非 32：25 11001100 －＞＂I＂
．word 1049，1057，0x298，0x49，1；
\＃33： 1 00011－＞＂n＂
．word 1057，1062， \(0 \times 1 \mathrm{e}, 0 \times 6 \mathrm{e}, 1\) ；
非 33： 6 1011000－＞＂c＂
．word 1062，1069，0x224，0x63，1；
非 33：13 00010－＞＂r＂
．word 1069，1074， \(0 \times 1 \mathrm{c}, 0 \times 72,1\) ；
非 33：18 10010－＞＂e＂
．word 1074，1079，0x1ee，0x65，1；
非 \(33: 23\) 1100011－＞＂m＂
．word 1079，1086， \(0 \times 252,0 \times 6 d, 1 ;\)
\＃33：30 10010 －＞＂e＂ ．word 1086，1091，0x1ee，0x65，1；
非 34： 300011 －＞＂n＂ ．word 1091，1096， \(0 \times 1 \mathrm{e}, 0 \times 6 \mathrm{e}, 1\) ；
非 34： 8 01001－＞＂t＂
．word 1096，1101， \(0 \times 4 a, 0 \times 74,1\) ；
非 34：13 111－＞＂＂
．word 1101，1104，0x2ec， \(0 \times 20,1\) ；
非 34：16 10110010 －＞＂to＂
．word 1104，1112，0x228，0x4d，2；
非 34：24 111 －＞＂＂
．word 1112，1115， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃34：27 101110011－＞＂address＂
．word 1115，1124，0x23a，0x74，7；
\＃35： 4 111－＞＂＂ ．word 1124，1127，0x2ec，0x20，1；
非 35：7 0011001 －＞＂of＂
．word 1127，1134， \(0 \times 34,0 \times 37,2\) ；
非 35：14 111－＞＂＂
．word 1134，1137，0x2ec，0x20，1；
非 35：17 1100101010－＞＂next＂ ．word 1137，1147， \(0 \times 282,0 \times 174,4\) ；
非 35：27 111 －＞＂＂
．word 1147，1150， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃35：30 110101111 －＞＂character＂ ．word 1150，1159，0x2e8，0x82，9；

非
\＃Huffman Compressed Text
非
huff＿compressed＿text＿start：
．word 0xd96cc6da，0x51375d0f，0x9864120f，0x852167e8，0xebb733ee ．word 0xbae1ee78，0xaba976ef，0xfffffffd，0x576c7d97，0x6efb089c ．word 0xb15743e7，0x81149f0a，0x42cfd1d7，0x6e67b3f5，0xb7dcf157 ．word \(0 \times 43 \mathrm{~d} 49278,0 \times 1 f 0 a 4337,0 \times 29 a 6 f 14 b, 0 x b 3 f 5 b 7 d c, 0 x f 17 f f f f f\) ．word 0xef54895b，0x735e60f8，0x5c295a72，0xdf957d7e，0x2ad25e60 ．word 0x981d6a5d，0xd8c51c6a，0xfafbccfb，0x6fbd5f06，0x9e84f90e ．word 0x1733d9f1，0x5743d491，0x49c3a466，0xec0a58e，0x434fb2f7 ．word 0x3e67e55f，0x5f8aba1e，0x60863206，0x1c873ad5，0xe49798f6 ．word \(0 \times 5 \mathrm{ee} 5 \mathrm{dbd7}, 0 \times 7 \mathrm{eb} 3 \mathrm{~b} 718,0 \times 1 \mathrm{c} 0 \mathrm{ed} 460,0 \times 433 \mathrm{c} 9361,0 \times 61\) aced48 ．word 0x28b£3c86，0xf41873a，0xccd2f31c，0x13ed836d，0xe830f5eb ．word \(0 \times 812 \mathrm{e} 27 \mathrm{~d} 9,0 \times \mathrm{e} 8 \mathrm{f} 6 \mathrm{c} 92\) ， \(0 \times 99096\) fad， \(0 \times 25 \mathrm{f} 9 \mathrm{e} 4 \mathrm{c} 0,0 \times f a a 7 f d 57\) ．word 0x75ded7f3，0xbf9524fb，0xaf957d59，0xdfac60b7，0xa65e88e6 ．word \(0 \times 287285 \mathrm{e}, 0 \times \mathrm{c} 6356 \mathrm{~b} 0 \mathrm{~d}, 0 \times 676 \mathrm{daf30} ,0 \times 4307 \mathrm{a} 1 \mathrm{ff}, 0 x f d 57 \mathrm{~d} 5 \mathrm{ec}\) ．word 0x22d976£7，0xe6a27285，0xef1fb97c，0xa3cc1964，0xbeb21eb5 ．word 0xddd72e7d，0xd910190c，0x2ef16bf1，0xaba2997b，0x9f33ee5f ．word 0x28c55b42，0xbe33e171，0x76b4e5be，0xe5f2877b，0x1945b465 ．word 0xc13eab99，0xe906394f，0xeade7290，0xf15743e3，0x24670e91 ．word 0x9349c0a5，0xe7290065，0x7bafab74，0x14800cf3，0xbfe5df27 ．word 0x5beb4bfa，0xa6eb30af，0x8cf85c5d，0xeb7ffff5，0x5dd77b5f ．word 0xcd378a5e，0xebae3f35，0x16355e01，0xfbaf72c5，0xb5aa2569 ．word 0x627cc3ff，0xffffffff，0xd57667d1，0x980657ba，0xf3459ca1 ．word \(0 \times 75 c 3 d c f 1,0 x 5699 b d e f, 0 x 38 c e 574 e, 0 x c d d f 4520,0 x 1 c b 6 a b 6 a\) ．word 0xbf8b22db，0x67e08269，0xbca1231f，0x562bca12，0x31fd634b ．word 0x5bd7aeac，0x401a67ee，0x58956f79，0x1660d5d1，0x4f67eb47 ．word \(0 x 4 c a 6900 c, 0 x a e d 2 c 412,0 x 5 c 24251 b, 0 x 6 c a 21703,0 x b 9 b 1 b 6 f a\) ．word 0xd1638c05，0x88315622，0xd836cc51，0x83b16626，0xd11b9fea ．word 0xbae9e77e，0x3975f694，0x6694240c，0xa97b1050，0x24584c85 ．word 0x3c45c0ee，0x6d11a82e，0x1b5faff，0xaefc97ad，0x7c09ede7 ．word 0xb2f17eb0，0x75b94170，0xde57e2f，0xc09£2458，0x83c602a0 ．word 0xa7b5fc24，0x1a282e01，0xbccf92fc，0xc718fb35，0xc28af194 ．word 0xbc260827，0xf4ca5680，0x65389f2a，0x494806e3，0xcd22bc65 ．word \(0 \times 484 a 0 b 80,0 \times 6 \pm 24 £ 6 d e, 0 x 32 a 423 e 6,0 \times 79 a 5069 d, 0 \times 15068 a 0\) ．word 0xb806f530，0x4e7d3065，0x8fe3be73，0xf62fb390，0xa50bd980 ．word 0x729eb975，0xd28b6df3，0xa25205d5，0x80729a82，0xe01bc9a0 ．word 0xc852f8ef，0xb1aa806e，0x3e68050a，0x7db3f65e，0x60194a0b
．word 0x806£2bf9，0x8e217b41，0xaacbd979，0xca5f51f6，0xa32c34fa ．word 0x9827304e，0x6056ac1a，0xe41a2211，0xa006e380，0x69bfcd58 ．word 0x80329417，0x3292e01，0xbc93e3be，0xf1da02a0，0xa7de3f75 ．word \(0 x f 350697 d, 0 x 175 d 75 d 7,0 x 5 d 751 e c 0,0 x a 57 d b 2 e 9,0 x a 2393 a e b\) ．word 0xaefd47e9，0x5b6cfdd，0x74004a0b，0x806f33c2，0x14918d78 ．word 0x466df453，0xf1a0a1df，0x68113e01，0xfcdb6fbf，0x1baebaeb ．word \(0 x a e b 5 f e 63,0 x 8 c 7 c 2827,0 x 403 f 99 c, 0 x 2 a 27 d 665,0 x 5 c 0 c a 4 b\) ．word 0x806d7ebf，0xebbf25eb，0x5f027b79，0xecbc5f8e，0x417de2b6 ．word \(0 x 481 c d 417,0 x d e 4 d 06,0 x 43 f a 8 b 89,0 x f 0 b 06429,0 x 7 d 77 d b 2 d\) ．word \(0 x 29 e 697 d 1, ~ 0 x 17832409,0 x 2 a 0 b 806 f, 0 x 2 b e b 3389,0 x f c 9 f a 608\) ．word 0x657c778，0xa5ae03c5，0xfdd7c9fe，0x67b0814b，0xad0a0b80 ．word 0x6f53bc52，0xe98281fc，0xcf17eb07，0x5b941703，0x292d7cee ．word 0x74aa578a，0xbe8827db，0x82c7348a，0xf194a378，0xe65442d7 ．word 0xc371524a，0x642e10cb，0x9ecb63fb，0x17d11a69，0xbc219636 ．word 0x5a742fac，0x865d30d，0x9585da1a，0x420235bc，0x2195390b ．word 0x2cda2373，0x62fbd968，0xb4050fc3，0xdea110b5，0xcb87843d ．word 0xe7710b5c，0xbc3c37b5，0xd62170af，0xa39b6cbe，0x5eee19d6 ．word \(0 x 49 e c 5 e 39, ~ 0 x a 5034 f c b, ~ 0 x c e d 9 f b 2 f, ~ 0 x 383785 e a, ~ 0 x e 42 b 121 a\) ．word 0x678b7b2d，0x98d9655b，0x7e6dbd7b，0xdaf712b6，0xa1ea4ed9 ．word 0x6eb6a1e8，0xcfa86eb，0xa1f19101，0x9f0dd743，0xeec919f2 ．word 0xfdc553a2，0x56615f81，0xf0b88cd6，0xb4f7c64b，0xb3c553b7 ．word \(0 x 5 d 0 f 8 c 91,0 x 9 e 1 e f 75 b, 0 x 50 f 8 c f 9 d, 0 x e e b a 1 f 76,0 x 48 c f 0 f 0 d\) ．word \(0 x d 743 f 3 c 9,0 x 81 e 1 d 0 f d, 0 x 5770 c 19 c, 0 x 6 e 2 a 1 c 4 a, 0 x c f 35 f 9 e 7\) ．word 0xc2e2335a，0xd3df192e，0xcf150£75，0xd0f8c919，0xc3dd679a ．word 0xfc0£85c4，0x66bbaea5，0xf23dd699，0xbaac4fb3，0xc09a686e ．word 0xab63b09d，0xafd3409，0xa686eb6a，0x1f9e75d7，0x75b61580 ．word 0xdfa619f0，0xb8c£35ac，0x2059bb09，0xfa68134c，0x3269a1bd ．word 0xd752ed2d，0xd699bae5，0xcf304c09，0x86eba1f9，0xe4c0ec30 ．word 0xf75d0f4c，0x32679e83，0xeeb6a1f，0x30cfa86e，0xba1f6781 ．word 0x303861ee，0xb6a1f3a0，0x7cef75d0，0xf9d024e8，0x1e187b8a ．word 0xa112b685，0x7c67c2e2，0x619ad743，0xe6192619，0xe1eeb497 ．word 0xc64c0eb9，0xeeba1f9e，0x4cf387ba，0xcc56bd2c，0x248ce1ee word 0xab23d390，0x92740896，0x1dd5627c，0x64d386ea，0xb63b09d0 ．word 0xafd38134，0xe1bae3a3，0x6bb09fa6，0x1934e04d，0x3437558f ．word 0x6b9c4ad3，0xdf9e44c3，0x3c550dd6，0x99bdd743，0xecf02607 ．word 0xf0dd563，0xdef79c67，0x2dd726c7，0xd9763cad，0x5c85e2c8 ．word 0x9cb7b6ec，0x4c89bfeb，0x2ec25914，0xd37eb22c，0x8b6dd6f6 ．word 0xaf33d9fd，0x97b178ec，0xe424e209，0x6452e4d5，0xe67ebfb4 ．word 0xfb431324，0x6e4da769，0xf686264b，0xb728b5f8，0xd5a0409d ．word 0xe3eb8e20，0x9fffffff5，0x5d8d3124，0x63576ca9，0xbfb525a6 ．word 0xf14a9bdc，0xd0d475cf，0xfff7ffff，0xfeabffff，0xfffedcd2 ．word 0x990a9bfb，0x9b71a8eb，0x9edad12a，0xbfbe33e1，0x70a75bff ．word 0xfffffaaf，0x96fd7fd2，0x2bf1f67c，0x55d0f524，0x5270f75a ．word 0x4be32607，0x6d7755e5，0xf7648cec，0x7d807fff，0xffaaedf3 ．word 0xb8438dd6，0x9efbb261，0xdadbaaf2，0xfbb24670，0xe853bfff ．word 0xfeabb8cb，0x1c763bc3，0xdd692fbb，0x26076b6e，0xba1f1923 ．word 0xc829d437，0x5a5f6b6e，0xb3cd7c6c，0x12941dc2，0x935db6b1 ．word 0x2b4b13c4，0x8dd699bd，0xef38ce72，0x372d5d22，0xcaf152
．word 0x194983bd，0xe4713226，0xffacbb09，0x6452e4d5，0xe67e900a ．word 0x69fafe67，0xb3e264b3，0xc1c999f6，0x68b68c8f，0x75ed90ca ．word 0x4c1e67dd，0x767c412c，0x861c9b4e，0xd3ed0c4c，0x91b93579 ．word \(0 x 9 £ 628885,0 x 21 a 7 e b f b, 0 x 4 f b 43104,0 x d 743 e 322,0 x 93 e 1 b 6 d e\) ．word 0x2557f798，0x3e17119a，0xd69ef304，0xc0edbeeb，0xa1f19233 ．word \(0 x 87 b d d 743,0 x e 3247900,0 x 7 b a d 2 c 4 f, 0 x 12375591,0 x f 6781232\) ．word 0x29d80000
huff＿compressed＿text＿end：
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline \multicolumn{7}{|l|}{huff＿dictionary：} \\
\hline ．asciiz＂ & ＂ & 非 Idx & 0 & Freq & 81 & Enc 10101 \\
\hline ．asciiz＂\＄＂ & & 非 Idx & 9 & Freq & 71 & Enc 01111 \\
\hline ．asciiz＂， & & \＃ Idx & 12 & Freq & 57 & Enc 00111 \\
\hline ．asciiz＂，\＄＂ & & \＃\(⿰ ⿰ 三 丨 ⿰ 丨 丨 ⿱ 一 土 寸\) Idx & 15 & Freq & 55 & Enc 00100 \\
\hline ．asciiz＂t1＂ & & 非 Idx & 19 & Freq & 38 & Enc 100011 \\
\hline ．asciiz & ．＂ & \＃ Idx & 22 & Freq & 28 & Enc 001011 \\
\hline ．asciiz＂．\({ }^{\text {n＂}}\) & & 非 Idx & 32 & Freq & 24 & Enc 1100010 \\
\hline ．asciiz＂addi＂ & & \＃1 Idx & 35 & Freq & 24 & Enc 1101000 \\
\hline ．asciiz＂：\n＂ & & 非 Idx & 40 & Freq & 18 & Enc 1000100 \\
\hline ．asciiz＂ascii＂ & & \＃ Idx & 43 & Freq & 18 & Enc 1000000 \\
\hline ．asciiz＂a0＂ & & \＃ Idx & 49 & Freq & 15 & Enc 0101001 \\
\hline ．asciiz＂\＂\n＂ & & 非 Idx & 52 & Freq & 15 & Enc 0101000 \\
\hline ．asciiz＂of＂ & & 非 Idx & 55 & Freq & 14 & Enc 0011001 \\
\hline ．asciiz＂t0＂ & & 非 Idx & 58 & Freq & 14 & Enc 0011000 \\
\hline ．asciiz＂\＂＂ & & \＃ Idx & 61 & Freq & 14 & Enc 0011011 \\
\hline ．asciiz＂t2＂ & & 非 Idx & 64 & Freq & 12 & Enc 10111011 \\
\hline ．asciiz＂the＂ & & \＃1 Idx & 67 & Freq & 12 & Enc 10111010 \\
\hline ．asciiz＂t3＂ & & \＃ Idx & 71 & Freq & 12 & Enc 11001111 \\
\hline ．asciiz＂（\＄＂ & & 非 Idx & 74 & Freq & 11 & Enc 10111000 \\
\hline ．asciiz＂to＂ & & 非 Idx & 77 & Freq & 10 & Enc 10110010 \\
\hline ．asciiz＂：＂ & & \＃\(⿰ ⿰ 三 丨 ⿰ 丨 丨 ⿱ 一 土 寸\) Idx & 80 & Freq & 9 & Enc 01110010 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|}
\hline ．asciiz＂非 \＄＂ & \＃Idx & 83 & Freq & 8 & Enc 01100100 \\
\hline ．asciiz＂string＂ & \＃ 1 Idx & 95 & Freq & 8 & Enc 01100111 \\
\hline ．asciiz＂）\n＂ & \＃ 1 Idx & 102 & Freq & 7 & Enc 00110101 \\
\hline ．asciiz＂非\n＂ & \＃\＃Idx & 105 & Freq & 7 & Enc 00001001 \\
\hline ．asciiz＂address＂ & \＃ Idx & 116 & Freq & 6 & Enc 101110011 \\
\hline ．asciiz＂table＂ & \＃Idx & 124 & Freq & 6 & Enc 101110010 \\
\hline ．asciiz＂character＂ & \＃Idx & 130 & Freq & 6 & Enc 110101111 \\
\hline ．asciiz＂miss＂ & \＃Idx & 140 & Freq & 5 & Enc 100010111 \\
\hline ．asciiz＂nop＂ & \＃ 1 Idx & 145 & Freq & 5 & Enc 101001100 \\
\hline ．asciiz＂bne＂ & \＃ 1 Idx & 149 & Freq & 5 & Enc 101001111 \\
\hline ．asciiz＂beq＂ & \＃Idx & 153 & Freq & 5 & Enc 101001001 \\
\hline ．asciiz＂非 & \＃Idx & 157 & Freq & 5 & Enc 101001000 \\
\hline ．asciiz＂Address＂ & \＃ 1 Idx & 164 & Freq & 4 & Enc 011010101 \\
\hline ．asciiz＂and＂ & \＃ 1 Idx & 172 & Freq & 4 & Enc 011010111 \\
\hline ．asciiz＂ULOOP＂ & \＃Idx & 176 & Freq & 4 & Enc 011010110 \\
\hline ．asciiz＂examined＂ & \＃Idx & 182 & Freq & 4 & Enc 011010000 \\
\hline ．asciiz＂being＂ & \＃ 1 Idx & 191 & Freq & 4 & Enc 011010011 \\
\hline ．asciiz＂histo＂ & \＃Idx & 197 & Freq & 4 & Enc 011010010 \\
\hline ．asciiz＂upper＂ & \＃Idx & 203 & Freq & 4 & Enc 011011101 \\
\hline ．asciiz＂非 & \＃ 1 Idx & 209 & Freq & 3 & Enc 1100111001 \\
\hline
\end{tabular}

．asciiz＂stars＂非 Idx 295 Freq 3 Enc 1100111011
．asciiz＂非＂Idx 301 Freq 3 Enc 1100111010
．asciiz＂know＂非 Idx 306 Freq 3 Enc 1100100101
．asciiz＂The＂非 Idx 311 Freq 3 Enc 1100100100
．asciiz＂one＂非 Idx 315 Freq 3 Enc 1100100111
．asciiz＂，－＂非 Idx 319 Freq 3 Enc 1100100000
．asciiz＂strlen＂非 Idx 323 Freq 3 Enc 1100100011
．asciiz＂Load＂\＃Idx 342 Freq 3 Enc 1100101101
．asciiz＂100＂⿰⿰三丨⿰丨三一 Idx 347 Freq 3 Enc 1100101111
．asciiz＂add＂非 Idx 351 Freq 3 Enc 1100101110
．asciiz＂\＂\＂\n＂非 Idx 355 Freq 3 Enc 1100101001
．asciiz＂element＂非 Idx 360 Freq 3 Enc 1100101000
．asciiz＂And＂非 Idx 368 Freq 3 Enc 1100101011
．asciiz＂next＂非 Idx 372 Freq 3 Enc 1100101010
．asciiz＂LOOP＂非 Idx 377 Freq 3 Enc 1101010100
．asciiz＂char＂非 Idx 382 Freq 3 Enc 1101010110
．asciiz＂that＂非 Idx 387 Freq 3 Enc 1101010001
.asciiz "histogram_data" 非 Idx 392 Freq 3 Enc 1101010000
．asciiz＂jal＂－非 Idx 407 Freq 3 Enc 1101010010
．asciiz＂you＂非 Idx 411 Freq 3 Enc 1101011101
．asciiz＂．．．＂非 Idx 415 Freq 3 Enc 1101011100
．asciiz＂－＞＂非 Idx 419 Freq 2 Enc 0110000001
．asciiz＂Return＂非 Idx 424 Freq 2 Enc 0110011001
．asciiz＂text＂非 Idx 431 Freq 2 Enc 0110011000
．asciiz＂not＂非 Idx 436 Freq 2 Enc 0110011010
．asciiz＂TB＿100＂非 Idx 440 Freq 2 Enc 0101110101
．asciiz＂Test＂非 Idx 447 Freq 2 Enc 0101110100
．asciiz＂非＂非 Idx 452 Freq 2 Enc 0101110111
．asciiz＂copy＂非 Idx 458 Freq 2 Enc 0101110110
．asciiz＂result＂非 Idx 463 Freq 2 Enc 0101110001
．asciiz＂return＂非 Idx 470 Freq 2 Enc 0101110000
．asciiz＂TEST＂非 Idx 477 Freq 2 Enc 0101110011
．asciiz＂space＂非 Idx 482 Freq 2 Enc 0101110010
．asciiz＂out＂非 Idx 488 Freq 2 Enc 0101111101
．asciiz＂asciiz＂非 Idx 492 Freq 2 Enc 0101111100
．asciiz＂lbu＂非 Idx 499 Freq 2 Enc 0101111111
．asciiz＂slti＂非 Idx 503 Freq 2 Enc 0101111001
．asciiz＂．．．＂非 Idx 508 Freq 2 Enc 0101111000
．asciiz＂＿＿start＂非 Idx 513 Freq 2 Enc 0101111011
．asciiz＂then＂非 Idx 521 Freq 2 Enc 0101111010
．asciiz＂Usage＂非 Idx 526 Freq 2 Enc 0101100101
．asciiz＂sub＂非 Idx 532 Freq 2 Enc 0101100100
．asciiz＂Make＂⿰⿰三丨⿰丨三一 Idx 536 Freq 2 Enc 0101100111
．asciiz＂there＂非 Idx 541 Freq 2 Enc 0101100110
．asciiz＂．）\n＂非 Idx 547 Freq 2 Enc 0101100001
．asciiz＂New＂非 Idx 551 Freq 2 Enc 0101100000
．asciiz＂syscall＂非 Idx 555 Freq 2 Enc 0101100011
．asciiz＂mtc1＂非 Idx 563 Freq 2 Enc 0101100010
．asciiz＂start＂非 Idx 568 Freq 2 Enc 0101101101
．asciiz＂cvt＂非 Idx 574 Freq 2 Enc 0101101100
．asciiz＂．．\n＂非 Idx 578 Freq 2 Enc 0101101111
．asciiz＂Orleans＂⿰⿰三丨⿰丨三一 Idx 583 Freq 2 Enc 0101101110
．asciiz＂）非＂非 Idx 591 Freq 2 Enc 0101101001
．asciiz＂for＂非 Idx 601 Freq 2 Enc 0101101000
．asciiz＂what＂非 Idx 605 Freq 2 Enc 0101101011
．asciiz＂DONE＂非 Idx 610 Freq 2 Enc 0101101010
．asciiz＂UDONE＂非 Idx 615 Freq 2 Enc 0110110101
．asciiz＂Character＂非 Idx 621 Freq 2 Enc 0110110100
．asciiz＂SLOOP＂非 Idx 631 Freq 2 Enc 0110110111
\begin{tabular}{llll} 
．asciiz＂Convert＂ & 非 Idx & 637 & Freq \\
．asciiz＂used＂ & 非 Idx & 645 & Freq \\
．asciiz＂doi＂ & 非 Idx & 650 & Freq \\
．asciiz＂index＂ & 非 Idx & 654 & Freq \\
．asciiz＂means＂ & 非 Idx & 660 & Freq \\
．asciiz＂Register＂ & 非 Idx & 666 & Freq
\end{tabular}

\footnotetext{
Enc 0110110001
Enc 0110110011
Enc 0110110010
Enc 0110111101
Enc 0110111100
Enc 0110111111
}
```


# Huffman Index Tree.

# 

huff_tree_right_base:
.half 0x80 非 Base for right child index.
huff_tree_literal_base:
.half 0 非 Base for literal.
huff_tree_dictionary_base:
.half 0x7000 非 Base for dict.

```
\#
非
huff_tree:
\(\begin{array}{lll}. h a l f & 0 \times 0162 & \text { 非 Tree Idx } \\ \text {.half } 0 \times 00 a 1 & \text { 非 Tree Idx } & 1\end{array}\)
    .half 0x0090 非 Tree Idx 200
    .half 0x008d 非 Tree Idx 3000
    .half 0x0086 非 Tree Idx 40000
    .half 0x0069 非 Tree Idx 500000
    .half 0x008c 非 Tree Idx 600001
    .half 0x008b 非 Tree Idx 7000010
    .half 0x008a 非 Tree Idx 80000100
    .half 0x0025 非 Tree Idx 900001000
    .half 0x7069 非 Tree Idx 1000001001
    .half 0x0077 非 Tree Idx 110000101
    .half 0x0031 非 Tree Idx 12000011
    .half 0x008f 非 Tree Idx 130001
    .half 0x0072 非 Tree Idx 1400010
    .half 0x006e 非 Tree Idx 1500011
    .half \(0 \times 0096\) 非 Tree Idx 16001
    .half 0x0093 非 Tree Idx 17001
    .half 0x700f 非 Tree Idx 1800100
    .half \(0 \times 0095\) 非 Tree Idx 19 00101
    .half 0x7016 \(\begin{array}{llll}\text {. } ⿰ ⿰ 三 丨 ⿰ 丨 三 一\end{array}\) Tree Idx 21001011
    .half 0x00a0 非 Tree Idx 220011
    .half 0x009b 非 Tree Idx 2300110
    .half 0x009a 非 Tree Idx \(24 \quad 001100\)
    \(\begin{array}{llll}\text {.half } 0 x 703 a & \text { 非 Tree Idx } & 25 & 0011000 \\ . h a l f & 0 x 7037 & \text { 非 Tree Idx } & 26 \\ 0011001\end{array}\)
    .half 0x009f 非 Tree Idx 27001101
    .half 0x009e 非 Tree Idx 280011010
    .half 0x0079 非 Tree Idx 2900110100
    .half \(0 \times 7066\) 非 Tree Idx 3000110101
    .half 0x703d 非 Tree Idx 310011011
    .half 0x700c 非 Tree Idx 3200111
    .half 0x00ed 非 Tree Idx 3301
    .half 0x00a6 非 Tree Idx 34010
    .half 0x00a5 非 Tree Idx 350100
    .half 0x0061 非 Tree Idx 3601000
    .half 0x0074 非 Tree Idx 3701001
    .half 0x00ae 非 Tree Idx 380101
    .half 0x00ab 非 Tree Idx 3901010
    .half 0x00aa 非 Tree Idx 40010100
    .half 0x7034 非 Tree Idx 410101000
    .half 0x7031 非 Tree Idx 420101001
    .half 0x00ad 非 Tree Idx 43010101
    .half 0x0023 非 Tree Idx 440101010
    .half 0x0070 非 Tree Idx 450101011
    .half 0x00ce 非 Tree Idx 4601011
    .half 0x00bf 非 Tree Idx 47010110
    .half 0x00b8 非 Tree Idx 480101100
    .half 0x00b5 非 Tree Idx 4901011000
    .half 0x00b4 非 Tree Idx 50 010110000
    .half 0x7227 非 Tree Idx 510101100000
    .half 0x7223 非 Tree Idx 520101100001
    .half 0x00b7 非 Tree Idx 53010110001
    .half 0x7233 非 Tree Idx 540101100010
    .half 0x722b 非 Tree Idx 550101100011
    .half 0x00bc 非 Tree Idx 5601011001
    .half 0x00bb 非 Tree Idx 57010110010
    .half 0x7214 非 Tree Idx 580101100100
    .half 0x720e 非 Tree Idx 590101100101
    .half 0x00be 非 Tree Idx 60010110011
    .half 0x721d 非 Tree Idx 610101100110

Pointer to right child．
Pointer to right child． Pointer to right child． Pointer to right child． Pointer to right child． Literal＂i＂
Pointer to right child．
Pointer to right child． Pointer to right child．
Literal＂\％＂
Dict Idx for＂非\n＂
Literal＂w＂
Literal＂1＂
Pointer to right child．
Literal＂r＂
Literal＂n＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂，\＄＂
Pointer to right child．
Literal＂g＂
Dict Idx for＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂t0＂
Dict Idx for＂of＂
Pointer to right child．
Pointer to right child．
Literal＂y＂
Dict Idx for＂）\n＂
Dict Idx for＂\＂＂
Dict Idx for＂，＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂a＂
Literal＂t＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂\＂\n＂
Dict Idx for＂a0＂
Pointer to right child．
Literal＂非＂
Literal＂p＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂New＂
Dict Idx for＂．）\n＂
Pointer to right child．
Dict Idx for＂mtc1＂
Dict Idx for＂syscall＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂sub＂
Dict Idx for＂Usage＂
Pointer to right child．
Dict Idx for＂there＂
．half 0x00c4 ．half 0x00c3 非 Tree Idx 65 ．half 0x7259 非 Tree Idx 66 ．half 0x724f 非 Tree Idx 67 ．half 0x00c6 非 Tree Idx 68 ．half \(0 \times 7262\) 非 Tree Idx 69 ．half 0x725d 非 Tree Idx 70
 ．half \(0 \times 723 \mathrm{e}\) 非 Tree Idx 73 ．half 0x7238 非 Tree Idx 74 ．half 0x00cd 非 Tree Idx 75 ．half \(0 \times 7247\) 非 Tree Idx 76 ．half \(0 \times 7242\) 非 Tree Idx 77 ．half 0x00d7 ．half 0x00d4 ．half \(0 \times 71 \mathrm{~d} 6\) ．half 0x71cf ．half 0x00d6 ．half \(0 \times 71 \mathrm{e} 2\) ．half 0x71dd ．half 0x00db ．half 0x71bf ．half 0x00dd ．half 0x71ca ．half 0x00e6 ．half 0x00e3 ．half 0x00e2 ．half 0x71fc ．half 0x00e5 ．half \(0 \times 7209\) ．half 0x7201 非 Tree Idx 101 ．half 0x00ea 非 Tree Idx 102 ．half 0x00e9 非 Tree Idx 103 ．half 0x71ec 非 Tree Idx 104 ．half 0x71e8 非 Tree Idx 105 ．half 0x00ec 非 Tree Idx 106 ．half 0x0044 非 Tree Idx 107 ．half 0x71f3 非 Tree Idx 108 ．half 0x0159 非 Tree Idx 109 ．half 0x012c 非 Tree Idx 110 ．half \(0 \times 011 \mathrm{f}\) 非 Tree Idx 111 ．half 0x0100 非 Tree Idx 112 ．half 0x00ff 非 Tree Idx 113 ．half 0x00f8 非 Tree Idx 114 ．half 0x00£7 非 Tree Idx 115 ．half 0x00f6 非 Tree Idx 116 ．half 0x0037 非 Tree Idx 117 ．half 0x005f 非 Tree Idx 118 ．half 0x71a3 非 Tree Idx 119 ．half 0x00fc 非 Tree Idx 120 ．half 0x00fb 非 Tree Idx 121 ．half 0x002c 非 Tree Idx 122 ．half 0x004e 非 Tree Idx 123 ．half 0x00fe 非 Tree Idx 124 ．half 0x002a 非 Tree Idx 125 ．half 0x007c 非 Tree Idx 126 ．half 0x0036 非 Tree Idx 127 ．half 0x0110 非 Tree Idx 128 ．half 0x0109 非 Tree Idx 129 ．half 0x0106 非 Tree Idx 130 ．half 0x0105 非 Tree Idx 131 ．half 0x0060 非 Tree Idx 132 ．half 0x003f 非 Tree Idx 133 ．half 0x0108 非 Tree Idx 134 ．half 0x0078 非 Tree Idx 135 ．half 0x007e 非 Tree Idx 136 ．half 0x010d 非 Tree Idx 137 ．half 0x010c 非 Tree Idx 138 ．half 0x007b 非 Tree Idx 139 ．half 0x0040 非 Tree Idx 140 ．half 0x010f 非 Tree Idx 141 ．half 0x007d 非 Tree Idx 142

0101100111 0101101 01011010 010110100 0101101000 0101101001 010110101 0101101010 0101101011 01011011 010110110 0101101100 0101101101 010110111 0101101110 0101101111 010111 0101110 01011100 010111000 0101110000 0101110001 010111001 0101110010 0101110011 01011101 010111010 0101110100 0101110101 010111011 0101110110 0101110111 0101111 01011110 010111100 0101111000 0101111001 010111101 0101111010 0101111011 01011111 010111110 0101111100 0101111101 010111111 0101111110 011 0110 01100 011000 0110000 01100000 011000000 0110000000 01100000000 01100000001 0110000001 011000001 0110000010 01100000100 01100000101 0110000011 01100000110 01100000111 01100001 0110001 01100010 011000100 0110001000 01100010000 01100010001 0110001001 01100010010 01100010011 011000101 0110001010 01100010100 01100010101 01100010110 Literal＂\}"

0110001011 Pointer to right child．

Dict Idx for＂Make＂
Pointer to right child． Pointer to right child． Pointer to right child．
Dict Idx for＂for＂
Dict Idx for＂）非＂ Pointer to right child． Dict Idx for＂DONE＂ Dict Idx for＂what＂ Pointer to right child． Pointer to right child．
Dict Idx for＂cvt＂
Dict Idx for＂start＂ Pointer to right child． Dict Idx for＂Orleans＂ Dict Idx for＂．．\n＂ Pointer to right child． Pointer to right child． Pointer to right child． Pointer to right child．
Dict Idx for＂return＂
Dict Idx for＂result＂
Pointer to right child．
Dict Idx for＂space＂
Dict Idx for＂TEST＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂Test＂
Dict Idx for＂TB＿100＂
Pointer to right child．
Dict Idx for＂copy＂
Dict Idx for＂非＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂．．．＂
Dict Idx for＂slti＂
Pointer to right child．
Dict Idx for＂then＂
Dict Idx for＂＿＿start＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂asciiz＂
Dict Idx for＂out＂
Pointer to right child． Literal＂D＂
Dict Idx for＂lbu＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂7＂
Literal＂＿＂
Dict Idx for＂－＞＂
Pointer to right child．
Pointer to right child．
Literal＂，＂
Literal＂N＂
Pointer to right child．
Literal＂＊＂
Literal＂｜＂
Literal＂6＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂｀＂
Literal＂？＂
Pointer to right child．
Literal＂x＂
Literal＂～＂
Pointer to right child．
Pointer to right child．
Literal＂\｛＂
Literal＂＠＂
\begin{tabular}{|c|c|c|c|}
\hline 0x003b & 143 & 01100010111 & Literal＂；＂ \\
\hline ．half 0x0118 & \＃Tree Idx 144 & 01100011 & Pointer to right child． \\
\hline ．half 0x0115 & 非 Tree Idx 145 & 011000110 & Pointer to right child． \\
\hline ．half 0x0114 & \＃Tree Idx 146 & 0110001100 & Pointer to right child． \\
\hline ．half 0x004a & \＃Tree Idx 147 & 01100011000 & Literal＂J＂ \\
\hline ．half 0x0056 & Tree Idx 148 & 01100011001 & Literal＂V＂ \\
\hline ．half 0x0117 & 非 Tree Idx 149 & 0110001101 & Pointer to right child． \\
\hline ．half 0x0045 & Tree Idx 150 & 01100011010 & Literal＂E＂ \\
\hline ．half 0x005b & \＃Tree Idx 151 & 01100011011 & Literal＂［＂ \\
\hline ．half 0x011c & Tree Idx 152 & 011000111 & Pointer to right child． \\
\hline ．half 0x011b & \＃Tree Idx 153 & 0110001110 & Pointer to right child． \\
\hline ．half 0x005e & Tree Idx 154 & 01100011100 & Literal＂＾＂ \\
\hline ．half 0x002b & Tree Idx 155 & 01100011101 & Literal \\
\hline ．half 0x011e & \＃Tree Idx 156 & 0110001111 & Pointer to right child． \\
\hline ．half 0x0055 & Tree Idx 157 & 01100011110 & Literal＂U＂ \\
\hline ．half 0x0039 & Tree Idx 158 & 01100011111 & Literal＂9＂ \\
\hline ．half 0x0123 & Tree Idx 159 & 011001 & Pointer to right child． \\
\hline ．half 0x0122 & 非 Tree Idx 160 & 0110010 & Pointer to right child． \\
\hline ．half 0x7053 & Tree Idx 161 & 01100100 & Dict Idx for \\
\hline ．half 0x002f & \＃Tree Idx 162 & 01100101 & Literal＂／＂ \\
\hline ．half 0x012b & Tree Idx 163 & 0110011 & Pointer to right child． \\
\hline ．half 0x0128 & \＃Tree Idx 164 & 01100110 & Pointer to right child． \\
\hline ．half 0x0127 & Tree Idx 165 & 011001100 & Pointer to right child． \\
\hline ．half 0x71af & \＃Tree Idx 166 & 0110011000 & Dict Idx for＂text＂ \\
\hline ．half 0x71a8 & Tree Idx 167 & 0110011001 & Dict Idx for＂Return＂ \\
\hline ．half 0x012a & Tree Idx 168 & 011001101 & Pointer to right child． \\
\hline ．half 0x71b4 & \＃Tree Idx 169 & 0110011010 & Dict Idx for＂not＂ \\
\hline ．half 0x0052 & \＃\＃Tree Idx 170 & 0110011011 & Literal＂R＂ \\
\hline ．half 0x705f & Tree Idx 171 & 01100111 & Dict Idx for＂string＂ \\
\hline ．half 0x013c & Tree Idx 172 & 01101 & Pointer to right child． \\
\hline ．half 0x0135 & Tree Idx 173 & 011010 & Pointer to right child． \\
\hline ．half 0x0132 & Tree Idx 174 & 0110100 & Pointer to right child． \\
\hline ．half 0x0131 & \＃Tree Idx 175 & 01101000 & Pointer to right child． \\
\hline ．half 0x70b6 & Tree Idx 176 & 011010000 & Dict Idx for＂examined＂ \\
\hline ．half 0x005c & \＃Tree Idx 177 & 011010001 & Literal＂\134＂ \\
\hline ．half 0x0134 & Tree Idx 178 & 01101001 & Pointer to right child． \\
\hline ．half 0x70c5 & \＃Tree Idx 179 & 011010010 & Dict Idx for＂histo＂ \\
\hline ．half 0x70bf & Tree Idx 180 & 011010011 & Dict Idx for＂being＂ \\
\hline ．half 0x0139 & Tree Idx 181 & 0110101 & Pointer to right child． \\
\hline ．half 0x0138 & 非 Tree Idx 182 & 01101010 & Pointer to right child． \\
\hline ．half 0x0042 & Tree Idx 183 & 011010100 & Literal＂B＂ \\
\hline ．half 0x70a4 & Tree Idx 184 & 011010101 & Dict Idx for＂Address＂ \\
\hline ．half 0x013b & Tree Idx 185 & 01101011 & Pointer to right child． \\
\hline ．half 0x70b0 & 非 Tree Idx 186 & 011010110 & Dict Idx for＂ULOOP＂ \\
\hline ．half 0x70ac & Tree Idx 187 & 011010111 & Dict Idx for＂and＂ \\
\hline ．half 0x014c & \＃Tree Idx 188 & 011011 & Pointer to right child． \\
\hline ．half 0x0145 & Tree Idx 189 & 0110110 & Pointer to right child． \\
\hline ．half 0x0142 & \＃Tree Idx 190 & 01101100 & Pointer to right child． \\
\hline ．half 0x0141 & Tree Idx 191 & 011011000 & Pointer to right child． \\
\hline ．half 0x003e & Tree Idx 192 & 0110110000 & Literal＂＞＂ \\
\hline ．half 0x727d & Tree Idx 193 & 0110110001 & Dict Idx for＂Convert＂ \\
\hline ．half 0x0144 & Tree Idx 194 & 011011001 & Pointer to right child． \\
\hline ．half 0x728a & Tree Idx 195 & 0110110010 & Dict Idx for＂doi＂ \\
\hline ．half 0x7285 & Tree Idx 196 & 0110110011 & Dict Idx for＂used＂ \\
\hline ．half 0x0149 & 非 Tree Idx 197 & 01101101 & Pointer to right child． \\
\hline ．half 0x0148 & Tree Idx 198 & 011011010 & Pointer to right child． \\
\hline ．half 0x726d & \＃Tree Idx 199 & 0110110100 & Dict Idx for＂Character \\
\hline ．half 0x7267 & Tree Idx 200 & 0110110101 & Dict Idx for＂UDONE＂ \\
\hline ．half 0x014b & Tree Idx 201 & 011011011 & Pointer to right child． \\
\hline ．half 0x003d & Tree Idx 202 & 0110110110 & Literal＂＝＂ \\
\hline ．half 0x7277 & Tree Idx 203 & 0110110111 & Dict Idx for＂SLOOP＂ \\
\hline ．half 0x0150 & Tree Idx 204 & 0110111 & Pointer to right child． \\
\hline ．half 0x014f & \＃Tree Idx 205 & 01101110 & Pointer to right child． \\
\hline ．half 0x0041 & \＃Tree Idx 206 & 011011100 & Literal＂A＂ \\
\hline ．half 0x70cb & \＃Tree Idx 207 & 011011101 & Dict Idx for＂upper＂ \\
\hline ．half 0x0154 & \＃Tree Idx 208 & 01101111 & Pointer to right child． \\
\hline ．half 0x0153 & Tree Idx 209 & 011011110 & Pointer to right child． \\
\hline ．half 0x7294 & 非 Tree Idx 210 & 0110111100 & Dict Idx for＂means＂ \\
\hline ．half 0x728e & Tree Idx 211 & 0110111101 & Dict Idx for＂index＂ \\
\hline ．half 0x0158 & \＃Tree Idx 212 & 011011111 & Pointer to right child． \\
\hline ．half 0x0157 & \＃Tree Idx 213 & 0110111110 & Pointer to right child． \\
\hline ．half 0x003c & \＃Tree Idx 214 & 01101111100 & Literal＂＜＂ \\
\hline ．half 0x005d & \＃Tree Idx 215 & 01101111101 & Literal＂］＂ \\
\hline ．half 0x729a & \＃Tree Idx 216 & 0110111111 & Dict Idx for＂Register＂ \\
\hline ．half 0x0161 & \＃Tree Idx 217 & 0111 & Pointer to right child． \\
\hline ．half 0x0160 & \＃Tree Idx 218 & 01110 & Pointer to right child． \\
\hline ．half 0x015d & \＃Tree Idx 219 & 011100 & Pointer to right child． \\
\hline ．half 0x0027 & \＃Tree Idx 220 & 0111000 & Literal＂＇＂ \\
\hline ．half 0x015f & 非 Tree Idx 221 & 0111001 & Pointer to right child． \\
\hline ．half 0x7050 & \＃Tree Idx 222 & 01110010 & Dict Idx for＂： \\
\hline ．half 0x0022 & \＃Tree Idx 223 & 01110011 & Literal＂\＂ \\
\hline
\end{tabular}
half 0x7009 非 Tree Idx 225 － 01111 ．half 0x01a2 非 Tree Idx 226 1 ．half 0x0179 非 Tree Idx 22710 ．half 0x0176 非 Tree Idx 228100 ．half 0x016d 非 Tree Idx 2291000 ．half 0x016a 非 Tree Idx 23010000 ．half 0x0169 非 Tree Idx 231100000 ．half 0x702b 非 Tree Idx 2321000000 ．half 0x0068 非 Tree Idx 233 ．half 0x016c 非 Tree Idx 234 ．half 0x0064 非 Tree Idx 235 ．half 0x0034 非 Tree Idx 236 ．half \(0 \times 0175\) 非 Tree Idx 237 ．half 0x7028 非 Tree Idx 239 ．half 0x0172 非 Tree Idx 240 ．half 0x004c 非 Tree Idx 241 ．half \(0 \times 0174\) 非 Tree Idx 242 ．half 0x0043 非 Tree Idx 243 ．half \(0 \times 7013\) 非 Tree Idx 245 ．half 0x0178 非 Tree Idx 246 ．half 0x0065 非 Tree Idx 247 ．half 0x0073 非 Tree Idx 248 ．half 0x018e 非 Tree Idx 250 ．half 0x017f 非 Tree Idx 251 ．half 0x017e 非 Tree Idx 252 ．half 0x0032 非 Tree Idx 253 ．half 0x0187 非 Tree Idx 255 ．half 0x0184 非 Tree Idx 256 \(\begin{array}{ll}. h a l f & 0 \times 0183 \\ \text { 非 Tree Idx } 257 \\ . h a l f & 0 \times 709 \mathrm{~d} \\ \text { 非 Tree Idx } 258\end{array}\) ．half 0x7099 非 Tree Idx 259 ．half 0x0186 非 Tree Idx 260 ．half 0x002d 非 Tree Idx 261 ．half 0x006a 非 Tree Idx 262 ．half 0x018b 非 Tree Idx 263 ．half 0x018a 非 Tree Idx 264 ．half 0x7091 ⿰⿰三丨⿰丨三一俍 Tree Idx 265 ．half 0x003a 非 Tree Idx 266 ．half 0x0033 非 Tree Idx 268 ．half 0x7095 非 Tree Idx 269 ．half 0x7000 非 Tree Idx 270 ．half 0x0197 非 Tree Idx 271 ．half 0x0196 非 Tree Idx 272 ．half \(0 \times 0193\) 非 Tree Idx 273 ．half 0x0063 非 Tree Idx 274 ．half \(0 \times 0195\) 非 Tree Idx 275 ．half 0x0076 非 Tree Idx 277 ．half 0x006c 非 Tree Idx 278 ．half 0x01a1 非 Tree Idx 279 ．half 0x019e 非 Tree Idx 280 ．half 0x019b 非 Tree Idx 281 ．half 0x704a 非 Tree Idx 282 ．half 0x707c 非 Tree Idx 284 ．half 0x7074 非 Tree Idx 285 ．half 0x01a0 非 Tree Idx 286 ．half 0x7043 非 Tree Idx 287 ．half 0x7040 非 Tree Idx 288 ．half 0x006f 非 Tree Idx 289 ．half 0x01f6 非 Tree Idx 290 ．half 0x01d7 非 Tree Idx 291 ．half 0x01aa 非 Tree Idx 292 ．half 0x01a7 非 Tree Idx 293 ．half 0x0030 非 Tree Idx 294 ．half 0x01a9 非 Tree Idx 295 ．half 0x7020 非 Tree Idx 296 ．half 0x006d 非 Tree Idx 297 ．half 0x01ca 非 Tree Idx 298 ．half 0x01bb 非 Tree Idx 299 ．half 0x01b4 非 Tree Idx 300 ．half 0x01b1 非 Tree Idx 301 ．half 0x01b0 非 Tree Idx 302 ．half 0x713f 非 Tree Idx 303 ．half 0x0035 非 Tree Idx 304

Literal＂．＂
Dict Idx for＂\＄＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂ascii＂
Literal＂h＂
Pointer to right child．
Literal＂d＂
Literal＂4＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂：\n＂
Pointer to right child．
Literal＂L＂
Pointer to right child．
Literal＂C＂
Dict Idx for＂miss＂
Dict Idx for＂t1＂
Pointer to right child．
Literal＂e＂
Literal＂s＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂2＂
Literal＂u＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂非＂
Dict Idx for＂beq＂
Pointer to right child．
Literal＂－＂
Literal＂j＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂nop＂
Literal＂：＂
Pointer to right child．
Literal＂3＂
Dict Idx for＂bne＂
Dict Idx for＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂c＂
Pointer to right child．
Dict Idx for＂to＂
Literal＂v＂
Literal＂l＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂（\＄＂
Pointer to right child．
Dict Idx for＂table＂
Dict Idx for＂address＂
Pointer to right child．
Dict Idx for＂the＂
Dict Idx for＂t2＂
Literal＂o＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂0＂
Pointer to right child．
Dict Idx for＂．\n＂
Literal＂m＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂，－＂
Literal＂5＂
．half 0x01b3 非 Tree Idx 305 ．half 0x714a 非 Tree Idx 306 ．half 0x7143 非 Tree Idx 307 ．half 0x01b8 非 Tree Idx 308 ．half 0x01b7 非 Tree Idx 309 ．half \(0 \times 7137\) 非 Tree Idx 310 ．half 0x7132 非 Tree Idx 311 ．half \(0 \times 0057\) 非 Tree Idx 313 ．half 0x713b 非 Tree Idx 314 ．half 0x01c3 非 Tree Idx 315 ．half 0x01c0 非 Tree Idx 316 ．half \(0 \times 01 b f\) 非 Tree Idx 317 ．half \(0 \times 7168\) 非 Tree Idx 318 ．half \(0 \times 01 \mathrm{c} 2\) 非 Tree Idx 320 ．half \(0 \times 7174\) 非 Tree Idx 321 ．half 0x7170 非 Tree Idx 322 ．half 0x01c6 非 Tree Idx 324 ．half 0x007a 非 Tree Idx 325 ．half \(0 \times 7156\) 非 Tree Idx 326 ．half 0x01c9 非 Tree Idx 327 ．half 0x715b 非 Tree Idx 329 ．half 0x01ce 非 Tree Idx 330 ．half 0x01cd 非 Tree Idx 331 ．half \(0 \times 0049\) 非 Tree Idx 332 ．half 0x0062 非 Tree Idx 333 ．half 0x01d6 非 Tree Idx 334 ．half 0x01d3 非 Tree Idx 335 ．half 0x01d2 非 Tree Idx 336 ．half 0x70d5 非 Tree Idx 337 ．half 0x70d1 非 Tree Idx 338 ．half \(0 \times 712 \mathrm{~d}\) 非 Tree Idx 340 ．half 0x7127 非 Tree Idx 341 ．half 0x7047 ⿰⿰三丨⿰丨三一要 Tree Idx 342 ．half 0x01f5 非 Tree Idx 343 ．half 0x01dc 非 Tree Idx 344 ．half \(0 \times 01 \mathrm{db}\) 非 Tree Idx 345 ．half 0x0066 非 Tree Idx 347 ．half \(0 \times 01 \mathrm{ec}\) 非 Tree Idx 348 ．half 0x01e5 非 Tree Idx 349 ．half \(0 \times 01 \mathrm{e} 1\) 非 Tree Idx 351 ．half 0x7188 非 Tree Idx 352 ．half 0x7183 非 Tree Idx 353 ．half 0x01e4 非 Tree Idx 354 ．half 0x7197 非 Tree Idx 355 ．half 0x004d 非 Tree Idx 356 ．half 0x01e8 非 Tree Idx 358 ．half \(0 \times 7179\) 非 Tree Idx 359 ．half 0x006b 非 Tree Idx 360 ．half 0x01eb 非 Tree Idx 361 ．half 0x717e 非 Tree Idx 362 ．half 0x0053 非 Tree Idx 363 ．half 0x01f0 非 Tree Idx 364 ．half 0x01ef 非 Tree Idx 365 ．half 0x0029 非 Tree Idx 367 ．half 0x01f4 非 Tree Idx 368 ．half 0x01f3 非 Tree Idx 369 ．half 0x719f 非 Tree Idx 370 ．half 0x719b 非 Tree Idx 371 ．half 0x7082 非 Tree Idx 372 ．half 0x000a 非 Tree Idx 373 ．half 0x0020 非 Tree Idx 374

110010001 1100100010 1100100011 11001001 110010010 1100100100 1100100101 110010011 1100100110 1100100111 1100101 11001010 110010100 1100101000 1100101001 110010101 1100101010 1100101011 11001011 110010110 1100101100 1100101101 110010111 1100101110 1100101111 110011 1100110 11001100 11001101 1100111 11001110 110011100 1100111000 1100111001 110011101 1100111010 1100111011 11001111 1101 11010 110100 1101000 1101001 110101 1101010 11010100 110101000 1101010000 1101010001 110101001 1101010010 1101010011 11010101 110101010 1101010100 1101010101 110101011 1101010110 1101010111 1101011 11010110 110101100 110101101 11010111 110101110 1101011100 1101011101 110101111 11011 111

Pointer to right child．
Dict Idx for＂非＂
Dict Idx for＂strlen＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂The＂
Dict Idx for＂know＂
Pointer to right child．
Literal＂W＂
Dict Idx for＂one＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂element＂
Dict Idx for＂\＂\＂\n＂
Pointer to right child．
Dict Idx for＂next＂
Dict Idx for＂And＂
Pointer to right child．
Pointer to right child．
Literal＂z＂
Dict Idx for＂Load＂
Pointer to right child．
Dict Idx for＂add＂
Dict Idx for＂100＂
Pointer to right child．
Pointer to right child．
Literal＂I＂
Literal＂b＂
Pointer to right child．
Pointer to right child．
Pointer to right child．

Dict Idx for＂非＂
Pointer to right child．
Dict Idx for＂非＂
Dict Idx for＂stars＂
Dict Idx for＂t3＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂addi＂
Literal＂f＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂histogram＿data＂
Dict Idx for＂that＂
Pointer to right child．
Dict Idx for＂jal＂
Literal＂M＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂LOOP＂
Literal＂k＂
Pointer to right child．
Dict Idx for＂char＂
Literal＂S＂
Pointer to right child．
Pointer to right child．
Literal＂（＂
Literal＂）＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂．．．＂
Dict Idx for＂you＂
Dict Idx for＂character＂
Literal＂\n＂
Literal＂＂

非 Assignment https：／／www．ece．lsu．edu／ee4720／2021／hw02．pdf
非 Solution Writeup https：／／www．ece．lsu．edu／ee4720／2021／hw02 sol．pdf

排 Problem 1
非

\section*{text}
hdecode：
\＃⿰⿰三丨⿰丨三⿰⿰三丨⿰丨三 Register Usage
非
\＃CALL VALUES
非 \＄a0：Unused
\＃\＄a1：Bit position to start at．
\＃\＄a2：Address to write decoded data．
非
\＃RETURN
\＃\＄a1：Bit position following encoded piece．
\＃\＄a2：Address following decoded piece．
非 \＄v0：Address of leaf of huff tree．
\＃\＄v1：Address of dict entry or value of literal character．
非
\＃Note：
非 Can modify registers \＄t0－\＄t9，\＄a0－\＄a3，\＄v0，\＄v1．
非 DO NOT modify other registers．
\＃［ \(\checkmark\) ］The testbench should show 0 errors．
\＃［ \(\checkmark\) ］Code should be reasonably efficient．
\＃［ \(\checkmark\) ］The code should be clearly written．
\＃［ \(\checkmark\) ］Comments should be written for an experienced programmer．
非［ \(\checkmark\) ］Do not use pseudoinstructions except for nop and la．
非 SOLUTION－Faster
\＃Load the word of compressed text that has the next bit we need．非
srl \＄t4，\＄a1， 5 非 Determine word index of next word to read．
sll \＄t6，\＄t4， 2 非 Determine byte offset of next word to read．
la \＄t5，huff＿compressed＿text＿start
add \＄t6，\＄t6，\＄t5 非 Compute address of next word to read．
lw \＄t8，0（\＄t6）非 Read next word of compressed text．

非 Shift the next bit into the most－significant position of \(\$\) t8．
非
andi \＄t7，\＄a1，0x1f 非 Determine bit number of next bit to examine．
sllv \＄t8，\＄t8，\＄t7 非 Put needed bit into most－significant position．
非 It＇s possible that we＇ll examine all the bits in t8，at
非 which time the next word needs to be loaded．To check
非 whether this has occurred set \＄t9 to the bit number of the
非 first bit in the next word．
非
sll \＄t9，\＄t4， 5 非 Compute bit number at which a new word ．．
addi \＄t9，\＄t9，0x20 非 ．．will need to be loaded．
la \＄t3，huff＿tree
addi \＄v0，\＄t3，－2 非 Compensate for crossing left－child code below．

TREE＿LOOP＿L：
非 Descend to Left Child．（Except in first iteration．）
非
addi \＄v0，\＄v0， 2
TREE＿LOOP：
\＃Live Registers at This Point in Code
非
非 \＄t3：Address of root of huff tree．
\＃\＄v0：Address of current node．
非
\＃\＄a1：Bit number of next bit（of compressed text）to examine．
非 \＄t8：Compressed text．The next bit to examine is the MSB．
\＃\(\$\) t9：Bit number at which the next word must be loaded into t8．
\＃\＄t6：Address of the current word of compressed text．

非 \＄a2：Address at which to write next decoded character．

非 Load value of current node into \＄v1
非
1hu \＄v1，0（\＄v0）
\＃First check for a non－leaf node，since that＇s most common．
非
addi \＄t4，\＄v1，－128 非 Compute index of right child．
sltiu \＄t5，\＄t4，0x6f80 非 Note：Unsigned comparison．
bne \＄t5，\＄0，NOT＿LEAF
sll \＄t7，\＄t4， 1 非 Compute byte offset of right child．

非 The node is a leaf．Check if it＇s a character or dictionary
\＃entry and branch to the respective code．
非
bltz \＄t4，CHAR＿LEAF
sb \＄v1， \(0(\$ a 2)\) 非 Write the character just in case．
j DICT＿LEAF
addi \＄t4，\＄v1，－0x7000

NOT＿LEAF：
非 Current node is an internal（non－leaf）node，so：
非－Get next bit of compressed text．
非－Based on the value of that bit descend to left or right child．
\＃Check whether we already have the bit we need．
非
bne \＄a1，\＄t9，EXAMINE＿NEXT＿BIT
addi \＄a1，\＄a1， 1
非 Load next word of compressed text．
非
addi \＄t6，\＄t6， 4 非 Update address ．．
lw \＄t8，0（\＄t6）非 No more bits，load a new word．
addi \＄t9，\＄t9，0x20 非 ．．and overflow value．

EXAMINE＿NEXT＿BIT：
非 Note：Next bit is in the MSB of t8．The bgez \＄t8 is taken if
非 the MSB of \＄t8 is zero
非
bgez \＄t8，TREE＿LOOP＿L
sll \＄t8，\＄t8， 1 非 Shift left so that next bit is in MSB pos．

RCHILD：
非 Descend to Right Child
非
非 Register \＄t7 already contains the byte offset．Just add it
非 to the root to get the address of the right child．
非
j TREE＿LOOP
add \＄v0，\＄t3，\＄t7

CHAR＿LEAF：
非 Node is a leaf．Its value is a character to append．
非
非 Note：char already written by sb above．
非
jr \＄ra 非 ．．and we＇re done ．．
addi \＄a2，\＄a2， 1 非 ．．after we increment the pointer．

DICT＿LEAF：
非 Node is a leaf．Its value is a byte offset into dictionary．
非
la \＄t0，huff＿dictionary
add \＄v1，\＄t4，\＄t0
非 Copy string from dictionary to output．
1b \＄t5，0（\＄v1）
addi \＄t4，\＄v1， 1
CPY＿LOOP：
sb \＄t5，0（\＄a2）
addi \＄a2，\＄a2， 1
lb \＄t5，0（\＄t4）
bne \＄t5，\＄0，CPY＿LOOP
addi \＄t4，\＄t4， 1

\section*{jr \＄ra}
nop

\section*{ \\ 棑 Testbench Routine}

非
非
```

    .data
    msg_full_text:
.asciiz "\nNumber correct %/s6/d, number wrong %/s5/d.\nDecoded Text:\n%/s0/s\n";
msg_piece_start:
.asciiz "** Decoding of Bit Position %/a1/d **\n"
msg_bit_pos_ok:
.asciiz "End bit pos ($a1 contents) correct: %/a1/d.\n"
msg_bit_pos_bad:
    .asciiz "End bit pos ($a1 contents) wrong: %/a1/d correct value is %/t0/d.\n"
msg_leaf_addr_ok:
.asciiz "Leaf address ($v0 contents) correct: %/t4/非.\n"
msg_leaf_addr_bad:
    .asciiz "Leaf address ($v0 contents) wrong: %/t4/非x correct value is %/t1/非.\n"
msg_dict_addr_ok:
.asciiz "Dict address ($v1 contents) correct: %/v1/非x Entry: \"%/v1/s\".\n"
msg_dict_addr_bad:
    .asciiz "Dict address ($v1 contents) wrong: %/v1/非x correct value is %/t3/非.\n"
msg_char_val_ok:
.asciiz "Char value ($v1 contents) correct: \"%/v1/c\"\n"
msg_char_val_bad:
    .asciiz "Char value ($v1 contents) wrong: %/v1/d (decimal) correct value is %/t3/d (decimal) or \"%/t3/c\".\n"
msg_char_copied_wrong:
.asciiz "Char value correct, \"%/v1/c\", but not copied, found \"%/\$t4/\".\n"
msg_copied_bad:
.asciiz "Text copied incorrectly starting at character %/t6/d: \"%/t7/s\"\n"
msg_early_exit:
.asciiz "** Error limit exceeded, exiting early.\n";
msg_timing:
.ascii "\nTotal insn %/s2/d, encoded bits %/t0/d, uncompressed bytes %/t1/d.\n"
.asciiz "Efficiency: %/f0/4.1f insn/bit %/f2/4.1f insn/byte.\n"
tb_timing_data:
.word 0 非 Instruction count.
.word 0 非 Sum of bits examined.
.word 0 非 Sum of characters in decoded text.
.text
.globl __start
__start:
la \$s3, huff_debug_samples
addi \$t0, \$0, 0
la \$a2, uncompressed
addi \$s2, \$0, 0 非 Total number of instructions.
addi \$s6, \$0, 0 非 Number correct
addi \$s7, \$0, 0 非 Number checked
TLOOP:
addi \$s7, \$s7, 1
lw $a1, 0($s3)
la \$a0, msg_piece_start
addi \$v0, \$0, 11
syscall
nop
addi \$s4, \$a2, 0
jal hdecode
mfc0 \$s1, \$9 非 Number of insns before hdecode.
mfc0 \$t1, \$9 非 Number of insns after hdecode.
addi \$t1, \$t1, -1

```
```

sub \$s1, \$t1, \$s1 非 Number of insns executed by hdecode.
add \$s2, \$s2, \$s1
sb $0, 0($a2)
addi \$s0, \$a0, 0
addi \$t4, \$v0, 0
lw $t0, 4($s3) 非 Bit end
lw $t1, 8($s3) 非 Byte offset into huff tree.
lw $t3, 12($s3) 非 Byte offset into dictionary.

# Compute and store total compressed bits and decompressed bytes.

# 

lw $t2, 0($s3)
sub \$t2, \$t0, \$t2
la \$a0, tb_timing_data
1w $t5, 4($a0) 非 Number of compressed bits
add \$t5, \$t5, \$t2
sw $t5, 4($a0)
lw $t5, 8($a0) 非 Total number of decompressed bytes.
lw $t2, 16($s3) 非 Number of bytes in this piece.
add \$t5, \$t5, \$t2
sw $t5, 8($a0)

# Check ending compressed-text bit position.

# 

la \$a0, msg_bit_pos_ok
beq \$t0, \$a1, TBIT_SHOW
nop
la \$a0, msg_bit_pos_bad
addi \$v0, \$0, 11
syscall
nop

# Check address of huff tree leaf.

# 

la \$t2, huff_tree
add \$t1, \$t1, \$t2
la \$a0, msg_leaf_addr_ok
beq \$t1, \$t4, TLEAF_SHOW
nop
la \$a0, msg_leaf_addr_bad

```
TBIT_SHOW:
TLEAF_SHOW:
    syscall
    nop
    slti \$t4, \$v1, 128
    bne \$t4, \$0, TCHAR_CHECK
    \# Check address of dictionary entry.
非
la \$t2, huff_dictionary
add \$t3, \$t3, \$t2
la \$a0, msg_dict_addr_ok
beq \$v1, \$t3, TDICT_SHOW
nop
la \$a0, msg_dict_addr_bad
TDICT_SHOW:
    syscall
    nop
    bne \$v1, \$t3, TCHECK_NEXT
    nop
    addi \$t6, \$t3, 0
    addi \$t7, \$s4, 0
TCHECK_WORD_LOOP:
    lbu \$t4, 0(\$s4)
    lbu \$t5, 0(\$t3)
    bne \$t4, \$t5, TCHECK_WORD_WRONG
    addi \$s4, \$s4, 1
    bne \$t5, \$0, TCHECK_WORD_LOOP
    addi \$t3, \$t3, 1
    \# Word copied correctly.
    addi \$s6, \$s6, 1
j TCHECK＿NEXT
nop

TCHECK＿WORD＿WRONG：
```

la \$a0, msg_copied_bad
sub \$t6, \$t3, \$t6
syscall
nop
j TCHECK_NEXT
nop

# Check value of decoded character.

非
beq \$v1, \$t3, TCHAR_SHOW
nop
la \$a0, msg_char_val_bad
syscall
nop
j TCHECK_NEXT
nop

```
TCHAR_CHECK:
TCHAR_SHOW:
    非 Check whether character copied correctly.
    lbu \$t4, 0 (\$s4)
    bne \$t4, \$v1, TCHAR_COPIED_WRONG
    nop
    非 Character correct.
    addi \$s6, \$s6, 1
    la \$a0, msg_char_val_ok
    syscall
    nop
    j TCHECK_NEXT
    nop
TCHAR_COPIED_WRONG
    la \$a0, msg_char_copied_wrong
    syscall
    nop
TCHECK_NEXT:
    非 Exit loop if error threshold exceeded.
    非
    sub \$s5, \$s7, \$s6
    slti \$t1, \$s5, 3
    bne \$t1, \$0, TCHECK_CONTINUE
    nop
    la \$a0, msg_early_exit
    syscall
    nop
    j TLOOP_EXIT
    nop
TCHECK_CONTINUE:
    非 Advance to next piece.
    非
    addi \$s3, \$s3, 20
    la \$t0, huff_compressed_text_start
    slt \$t1, \$s3, \$t0
    bne \$t1, \$0 TLOOP
    nop
    非非
    非 End of Loop
TLOOP_EXIT:
    非 Compute number of incorrect pieces.
    非
    sub \$s5, \$s7, \$s6
    非 Show execution rate
    非
    la \$a0, msg_timing
    la \$t4, tb_timing_data
    lw \$t0, 4(\$t4)
```

1w $t1, 8($t4)
mtc1 \$s2, \$f20
cvt.d.w \$£22, \$f20
mtc1 \$t0, \$f10
cvt.d.w \$f12, \$f10
mtc1 \$t1, \$f14
cvt.d.w \$f16, \$f14
div.d \$f0, \$土22, \$f12
div.d \$土2, \$土22, \$f16
addi \$v0, \$0, 11
syscall
nop

# Show decoded text.

# 

la \$a0, msg_full_text
la \$s0, uncompressed
addi \$v0, \$0, 11
syscall
nop
addi \$v0, \$0, 10
syscall
nop

```
.data
.align 4
uncompressed:
. space 4000
uncompressed_end:
\# Data from file histo-bare.s
\# Code size \(14253+1679+5176=21108\) orig 40784 b, ratio 0.518 max 11
\# Words 188 Codes 188 Resorts 10
\# Compression Debug Samples
非
非 Encoding: .word BIT_START, BIT_END, TREE_POS, DICT_POS, FRAG_LENGTH
非
huff_debug_samples:
非 0: 011011 -> "\n"
    .word 0, 5, 0x2ea, 0xa, 1;
非 0: 5001011 -> "
    .word 5, 11, \(0 \times 2 a, 0 \times 16,9\);
非 0:11 0110011000 -> "text"
    .word 11, 21, \(0 \times 14 \mathrm{c}, 0 \times 1 \mathrm{af}, 4 ;\)
非 0:21 11011 -> "\n"
    .word 21, 26, 0x2ea, 0xa, 1;
非 0:26 011010010 -> "histo"
    .word 26, 35, 0x166, 0xc5, 5;
\# 1: 3 1000100 -> ": \n"
    .word 35, 42, \(0 \times 1 \mathrm{de}, 0 \times 28,2\);
非 1:10 11011 -> "\n"
    .word 42, 47, 0x2ea, 0xa, 1;
\# 1:15 10101 -> "
    .word 47, 52, \(0 \times 21 \mathrm{c}, 0,8\);
\# 1:20 1101000 -> "addi"
    .word 52, 59, 0x2b4, 0x23, 4;
\# 1:27 01111 -> " \$"
    .word 59, 64, \(0 \times 1 \mathrm{c} 2,0 \times 9,2\);
非 2: 010011 -> "s"
    .word 64, 69, 0x1f0, 0x73, 1;
非 2: 5000011 -> "1"
    .word 69, 75, \(0 \times 18,0 \times 31,1 ;\)
\# 2:11 00100 -> ", \$"
    .word 75, 80, 0x24, 0xf, 3;
非 2:16 00010 -> "r"
    .word 80, 85, \(0 \times 1 \mathrm{c}, 0 \times 72,1 ;\)
非 2:21 01000 -> "a"
    .word \(85,90,0 \times 48,0 \times 61,1\);
非 2:26 00111 -> ",
    .word 90, 95, 0x40, 0xc, 2;
非 2:31 110000 -> "0"
    .word 95, 101, \(0 \times 24 \mathrm{c}, 0 \times 30,1\);
非 3: 5 101001000 -> " 非 "
    .word 101, 110, 0x204, 0x9d, 6;
\＃3：14 0101100111 －＞＂Make＂
．word 110，120，0x7c，0x218，4；
\＃3：24 111 －＞＂＂
．word 120，123， \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 3：27 01000 －＞＂a＂
．word 123，128， \(0 \times 48,0 \times 61,1\) ；
非 4：0 111 －＞＂＂ ．word 128，131， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃4：3 0101110110－＞＂copy＂ ．word 131，141，0xb8，0x1ca，4；
非 4：13 111 －＞＂ ．word 141，144， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 4：16 0011001 －＞＂of＂ ．word 144，151， \(0 \times 34,0 \times 37,2\) ；
\＃4：23 111 －＞＂＂
．word 151，154， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃4：26 10111010 －＞＂the＂
．word 154，162， \(0 \times 23 \mathrm{e}, 0 \times 43,3\) ；
非 5： 2111 －＞＂＂
．word 162，165， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃5：5 0101110000－＞＂return＂ ．word 165，175，0xa4，0x1d6，6；
非 5：15 111 －＞＂＂ ．word 175，178，0x2ec，0x20，1；
\＃5：18 101110011 －＞＂address＂ ．word 178，187，0x23a，0x74，7；
非 5：27 1100010－＞＂．\n＂ ．word 187，194， \(0 \times 250,0 \times 20,2\) ；
非 6： 2 10101－＞
．word 194，199，0x21c，0，8；
\＃6：7 1101010010－＞＂jal＂
．word 199，209， \(0 \times 2 \mathrm{c} 6,0 \times 197,3\) ；
非 6：17 111 －＞＂＂
．word 209，212， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 6：20 011011101－＞＂upper＂
．word 212，221， \(0 \times 19 \mathrm{e}, 0 \times \mathrm{cb}, 5\) ；
\＃\(\quad\) 6：29 111 －＞＂＂
．word 221，224， \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 7： 0 111－＞＂＂
．word 224，227，0x2ec，0x20，1；
非 7： 3 111－＞＂＂
．word 227，230，0x2ec，0x20，1；
非 7： 6 111－＞＂＂
．word 230，233， \(0 \times 2 e c, 0 \times 20,1 ;\)
非 7： 9111 －＞＂＂
．word 233，236，0x2ec，0x20，1；
非 7：12 111 －＞＂＂
．word 236，239， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 7：15 111 －＞＂＂
．word 239，242， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃7：18 111 －＞＂＂
．word 242，245， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 7：21 111 －＞＂＂
．word 245，248，0x2ec，0x20，1；
\＃7：24 111 －＞＂＂
．word 248，251， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃7：27 111 －＞＂＂
．word 251，254，0x2ec，0x20，1；
非 7：30 0101010－＞＂非＂
．word 254，261， \(0 \times 58,0 \times 23,1 ;\)
非 8： 5111 －＞＂＂
．word 261，264，0x2ec，0x20，1；
非 8： 8 0110110001－＞＂Convert＂
．word 264，274， \(0 \times 182,0 \times 27 d, 7\) ；
非 8：18 111 －＞＂＂
．word 274，277， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 8：21 10110010 －＞＂to＂
．word 277，285，0x228，0x4d，2；
非 8：29 111 －＞＂＂
．word 285，288， \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
\＃9：0 011011101－＞＂upper＂
．word 288，297，0x19e，0xcb，5；
\＃9： 9111 －＞＂＂
．word 297，300， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃\(\quad 9: 12 \quad 1011000\)－＞＂c＂
．word 300，307， \(0 \times 224,0 \times 63,1\) ；
\＃9：19 01000－＞＂a＂
．word 307，312， \(0 \times 48,0 \times 61,1 ;\)
非 9：24 10011－＞＂s＂
．word 312，317，0x1f0，0x73，1；
\＃9：29 10010 －＞＂e＂
\＃10： 2 1100010－＞＂\n＂ ．word 322，329， \(0 \times 250,0 \times 20,2\)
\＃10： 910101 －＞
．word 329，334， \(0 \times 21 \mathrm{c}, 0,8\) ；
非 10：14 1101000 －＞＂addi＂
．word 334，341， \(0 \times 2 \mathrm{~b} 4,0 \times 23,4\) ；
非 10：21 01111 －＞＂\＄＂
．word 341，346，0x1c2，0x9，2；
\＃10：26 10011－＞＂s＂
．word 346，351， \(0 \times 1 \mathrm{f} 0,0 \times 73,1\)
非 10：31 110000－＞＂0＂ ．word 351，357，0x24c，0x30，1；
\＃11： 500100 －＞＂，\＄＂ ．word 357，362，0x24，0xf，3；
非 11：10 0101001 －＞＂a0＂
．word 362，369， \(0 \times 54,0 \times 31,2\) ；
非 11：17 00111 －＞＂，＂
．word 369，374，0x40，0xc，2；
\＃11：22 110000－＞＂0＂
．word \(374,380,0 \times 24 \mathrm{c}, 0 \times 30,1\) ；
非 11：28 101001000－＞＂非＂
．word 380，389， \(0 \times 204,0 x 9 d, 6\) ；
\＃12： 5 0101100111－＞＂Make＂
．word 389，399， \(0 \times 7 \mathrm{c}, 0 \times 218,4\) ；
\＃12：15 111 －＞＂＂
word 399，402，0x2ec，0x20， 1 ．
非 12：18 01000 －＞＂a＂
．word 402，407， \(0 \times 48,0 \times 61,1 ;\)
\＃12：23 111 －＞＂＂
．word 407，410， \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
\＃12：26 0101110110－＞＂copy＂
．word 410，420，0xb8，0x1ca， 4
\＃13： 4 111－＞＂＂
．word 420，423，0x2ec，0x20，1；
非 13： 7 0011001－＞＂of＂
．word 423，430， \(0 \times 34,0 \times 37,2\) ；
非 13：14 111 －＞＂
．word 430，433， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\)
｜13：17 01100111－＞＂string＂
．word 433，441， \(0 \times 156,0 \times 5 f, 6\) ；
13：25 111 －＞＂
．word 441，444， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 13：28 0101101101－＞＂start＂
．word 444，454，0x94，0x238，5；
非 14： 6 111－＞＂＂
．word 454，457，0x2ec，0x20， 1
\＃14： 9 101110011－＞＂address＂
．word 457，466，0x23a，0x74，7
非 14：18 1100010 －＞＂．\n＂
．word 466，473， \(0 \times 250,0 \times 20,2 ;\)
非 14：25 10101 －＞＂
．word 473，478，0x21c，0，8
非 14：30 1101000－＞＂addi＂
．word 478，485， \(0 \times 2 \mathrm{~b} 4,0 \times 23,4\) ；
非 15： 501111 －＞＂\＄＂
．word 485，490， \(0 \times 1 \mathrm{c} 2,0 \times 9,2\) ；
非 15：10 0101001 －＞＂a0＂
．word 490，497， \(0 \times 54,0 \times 31,2\) ；
\＃15：17 00100 －＞＂，\＄＂
．word 497，502，0x24，0xf，3；
非 15：22 10011 －＞＂s＂
．word 502，507， \(0 \times 1 \mathrm{f} 0,0 \times 73,1 ;\)
非 15：27 110000－＞＂0＂ ．word 507，513， \(0 \times 24 \mathrm{c}, 0 \times 30,1\)
非 16： 1 00111－＞＂，＂ ．word 513，518，0x40，0xc，2；
非 16： 6 110000－＞＂0＂ ．word 518，524，0x24c，0x30，1；
非 16：12 101001000－＞＂非＂ ．word 524，533，0x204，0x9d， 6
非 16：21 0110011011 －＞＂R＂
．word 533，543， \(0 \times 154,0 \times 52,1\)
\＃16：31 10010 －＞＂e＂
．word 543，548， \(0 \times 1 \mathrm{ee}, 0 \times 65,1\)
\＃17： 410011 －＞＂s＂
．word 548，553，0x1f0，0x73，1；
\＃17： 9 01001－＞＂t＂
．word 553，558， \(0 \times 4 a, 0 \times 74,1\) ；
非 17：14 101111 －＞＂о＂
．word 558，564， \(0 \times 242,0 \times 6 \pm, 1\) ；

非 17：20 00010－＞＂r＂
．word 564，569， \(0 \times 1 \mathrm{c}, 0 \times 72,1\) ；
非 17：25 10010－＞＂e＂
．word 569，574， \(0 \times 1 \mathrm{ee}, 0 \times 65,1 ;\)
非 17：30 111 －＞＂＂
．word 574，577，0x2ec，0x20，1；
非 18： 1 01100111－＞＂string＂
．word 577，585，0x156，0x5f，6；
非 18： 9 111－＞＂＂
．word 585，588， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 18：12 0101101101－＞＂start＂
．word 588，598，0x94，0x238，5；
非 18：22 111 －＞＂＂
．word 598，601， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 18：25 101110011－＞＂address＂
．word 601，610， \(0 \times 23 a, 0 \times 74,7\) ；
\＃19： 2 1100010－＞＂．\n＂
．word 610，617， \(0 \times 250,0 \times 20,2\) ；
非 19： 9 111－＞＂＂
．word 617，620， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 19：12 111 －＞＂＂
．word 620，623， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 19：15 111 －＞＂＂
．word 623，626，0x2ec，0x20，1；
\＃19：18 111 －＞＂＂
．word 626，629，0x2ec，0x20，1；
非 19：21 111 －＞＂＂
．word 629，632， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 19：24 111 －＞＂＂
．word 632，635， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃19：27 111 －＞＂＂
．word 635，638，0x2ec， \(0 \times 20,1\) ；
非 19：30 111 －＞＂＂
．word 638，641，0x2ec，0x20，1；
非 20： 1 11011－＞＂\n＂ ．word 641，646，0x2ea，0xa，1；
\＃20： 6 1101010100－＞＂LOOP＂ ．word 646，656， \(0 \times 2 \mathrm{ce}, 0 \times 179,4 ;\)
\＃20：16 1000100－＞＂：\n＂ ．word 656，663， \(0 \times 1 \mathrm{de}, 0 \times 28,2\) ；
非 20：23 10101 －＞＂ ．word 663，668，0x21c，0，8；
非 20：28 101101 －＞＂1＂ ．word 668，674，0x22c，0x6c，1；
非 21： 211001101 －＞＂b＂ ．word 674，682， \(0 \times 29 a, 0 \times 62,1\) ；
非 21：10 01111 －＞＂\＄＂ ．word 682，687， \(0 \times 1 \mathrm{c} 2,0 \times 9,2\) ；
\＃21：15 0011000 －＞＂t0＂ ．word 687，694， \(0 \times 32,0 \times 3 a, 2\) ；
非 21：22 00111 －＞＂，＂
．word 694，699，0x40，0xc，2；
非 21：27 110000－＞＂0＂ ．word 699，705， \(0 \times 24 \mathrm{c}, 0 \times 30,1\) ；
非 22： 1 10111000－＞＂（\＄＂
．word 705，713，0x234，0x4a，2；
非 22：9 0101001 －＞＂a0＂ ．word 713，720， \(0 \times 54,0 \times 31,2\) ；
非 22：16 0101101001－＞＂）非＂ ．word 720，730，0x86，0x24f，9；
非 22：26 1100101101－＞＂Load＂ ．word 730，740， \(0 \times 28 \mathrm{c}, 0 \times 156,4\) ；
非 23： 4 111－＞＂＂ ．word 740，743，0x2ec，0x20，1；
非 23：7 1100101010－＞＂next＂
．word 743，753， \(0 \times 282,0 \times 174,4\) ；
非 23：17 111 －＞＂＂
．word 753，756，0x2ec，0x20，1；
\＃23：20 110101111－＞＂character＂
．word 756，765，0x2e8，0x82，9；
\＃23：29 1100010－＞＂．\n＂
．word 765，772，0x250，0x20，2；
\＃24： 4 10101－＞＂
．word 772，777，0x21c，0，8；
非 24： 9 101001001－＞＂beq＂ ．word 777，786，0x206，0x99，3；
非 24：18 01111 －＞＂\＄＂ ．word 786，791， \(0 \times 1 \mathrm{c} 2,0 \times 9,2\) ；
非 24：23 0011000－＞＂t0＂ ．word 791，798， \(0 \times 32,0 \times 3 a, 2 ;\)
非 24：30 00100 －＞＂，\＄＂
．word 798，803，0x24，0xf，3；
非 25：3 110000－＞＂0＂
．word 803，809， \(0 \times 24 \mathrm{c}, 0 \times 30,1\) ；
非 25：9 00111－＞＂，＂
．word 809，814， \(0 \times 40,0 \times c, 2\) ；
非 25：14 0101101010－＞＂DONE＂
．word 814，824，0x8a，0x262，4；
非 25：24 0101110111－＞＂非＂
．word 824，834，0xba，0x1c4，5；
非 26： 201100011000 －＞＂J＂
．word 834，845， \(0 \times 126,0 \times 4 a, 1 ;\)
非 26：13 1010001 －＞＂u＂
word 845，852，0x1fc，0x75，1；
非 26：20 1100011 －＞＂m＂ ．word 852，859，0x252，0x6d，1；
非 26：27 0101011－＞＂p＂ ．word 859，866，0x5a，0x70，1
非 27： 2 111－＞＂＂
．word 866，869， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 27： 50101111101 －＞＂out＂
．word 869，879，0xd2，0x1e8，3；
非 27：15 111 －＞＂＂
．word 879，882， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 27：18 0011001－＞＂of＂
．word 882，889， \(0 \times 34,0 \times 37,2\) ；
非 27：25 111 －＞＂＂
．word 889，892， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃27：28 101101－＞＂1＂
．word 892，898， \(0 \times 22 c, 0 \times 6 \mathrm{c}, 1\) ；
非 28： 2 101111－＞＂o＂
．word 898，904，0x242，0x6f， 1
\＃28： 8101111 －＞＂о＂ ．word 904，910，0x242，0x6x，1
非 28：14 0101011 －＞＂p＂ ．word 910，917， \(0 \times 5 \mathrm{a}, 0 \times 70,1 ;\)
非 28：21 111 －＞＂＂
．word 917，920， \(0 \times 2 \mathrm{ec}, 0 \times 20,1 ;\)
非 28：24 00000－＞＂i＂
．word 920，925， \(0 \times a, 0 \times 69,1\) ；
非 28：29 1101001－＞＂f＂
．word 925，932，0x2b6， \(0 \times 66,1\) ；
非 29： 4111 －＞＂
．word 932，935， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
\＃29：7 01000－＞＂a＂
．word 935，940， \(0 \times 48,0 \times 61,1 ;\)
非 29：12 01001－＞＂t＂
．word 940，945， \(0 \times 4 \mathrm{a}, 0 \times 74,1\) ；
非 29：17 111 －＞＂＂
．word 945，948，0x2ec，0x20，1；
非 29：20 10010 －＞＂e＂
．word 948，953， \(0 \times 1\) ee， \(0 \times 65,1\) ；
非 29：25 00011－＞＂n＂
．word 953，958，0x1e，0x6e，1
非 29：30 1000010－＞＂d＂
．word 958，965，0x1d6， \(0 \times 64,1 ;\)
非 30：5 111 －＞＂＂
．word 965，968， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 30： 8 0011001－＞＂of＂
．word 968，975， \(0 \times 34,0 \times 37,2\) ；
非 30：15 111 －＞＂＂
．word 975，978， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 30：18 01100111－＞＂string＂
．word 978，986，0x156，0x5士，6；
非 \(30: 26\) 1100010 \(->\)＂．\n＂
．word 986，993， \(0 \times 250,0 \times 20,2\) ；
非 31： 110101 －＞
．word 993，998，0x21c，0，8；
非 31： 6 1101000－＞＂addi＂
．word 998，1005，0x2b4，0x23，4；
非 31：13 01111 －＞＂\＄＂
．word 1005，1010，0x1c2，0x9，2；
非 31：18 0101001－＞＂a0＂
．word 1010，1017， \(0 \times 54,0 \times 31,2\) ；
非 31：25 00100 －＞＂，\＄＂
．word 1017，1022， \(0 \times 24,0 x 4,3\) ；
非 31：30 0101001－＞＂a0＂
．word 1022，1029， \(0 \times 54,0 \times 31,2\) ；
非 32： 500111 －＞＂，
．word 1029，1034， \(0 \times 40,0 \times c, 2\) ；
非 32：10 000011－＞＂1＂
．word 1034，1040， \(0 \times 18,0 \times 31,1 ;\)
．word 1040，1049，0x204，0x9d，6；
非 32：25 11001100－＞＂I＂
．word 1049，1057，0x298，0x49，1；
\＃33： 100011 －＞＂n＂
．word 1057，1062，0x1e，0x6e，1；
非 33：6 1011000－＞＂c＂
．word 1062，1069，0x224，0x63，1；
非 33：13 00010 －＞＂r＂ ．word 1069，1074，0x1c，0x72，1；
非 33：18 10010－＞＂e＂ ．word 1074，1079，0x1ee，0x65，1；
\＃33：23 1100011－＞＂m＂ ．word 1079，1086，0x252，0x6d，1；
\＃33：30 10010－＞＂e＂
．word 1086，1091，0x1ee，0x65，1；
非 34： 3 00011－＞＂n＂
．word 1091，1096，0x1e，0x6e，1；
\＃34： 8 01001－＞＂t＂
．word 1096，1101，0x4a，0x74，1；
\＃34：13 111 －＞＂＂
．word 1101，1104，0x2ec，0x20，1；
\＃34：16 10110010－＞＂to＂ ．word 1104，1112，0x228，0x4d，2；
非 34：24 111 －＞＂＂
．word 1112，1115，0x2ec，0x20，1；
非 34：27 101110011 －＞＂address＂
．word 1115，1124，0x23a，0x74，7；
非 35： 4 111－＞＂＂
．word 1124，1127， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 35：7 0011001 －＞＂of＂
．word 1127，1134， \(0 \times 34,0 \times 37,2\) ；
非 35：14 111－＞＂＂
．word 1134，1137，0x2ec，0x20，1；
非 35：17 1100101010－＞＂next＂
．word 1137，1147， \(0 \times 282,0 \times 174,4 ;\)
\＃35：27 111 －＞＂＂
．word 1147，1150， \(0 \times 2 \mathrm{ec}, 0 \times 20,1\) ；
非 35：30 110101111 －＞＂character＂
．word 1150，1159，0x2e8，0x82，9；

非
\＃Huffman Compressed Text
非
huff＿compressed＿text＿start：
．word \(0 x d 96 c c 6 d a, ~ 0 x 51375 d 0 f, ~ 0 x 9864120 f, ~ 0 x 852167 e 8, ~ 0 x e b b 733 e e\) ．word 0xbae1ee78，0xaba976ef，0xfffffffd，0x576c7d97，0x6efb089c ．word \(0 \times b 15743 \mathrm{e} 7,0 \times 81149 \mathrm{f} 0 \mathrm{a}, 0 \times 42 \mathrm{cfd} 1 \mathrm{d7}, 0 \times 6 \mathrm{e} 67 \mathrm{~b} 3 f 5,0 \times b 7 \mathrm{dcf157}\) ．word 0x43d49278，0x1f0a4337，0x29a6f14b，0xb3f5b7dc，0xf17fffff ．word 0xef54895b，0x735e60£8，0x5c295a72，0xdf957d7e，0x2ad25e60 ．word 0x981d6a5d，0xd8c51c6a，0xfafbccfb，0x6fbd5f06，0x9e84f90e ．word \(0 \times 1733 d 9 f 1,0 \times 5743 d 491,0 x 49 c 3 a 466,0 x e c 0 a 58 e, 0 x 434 f b 2 f 7\) ．word 0x3e67e55f，0x5f8aba1e，0x60863206，0x1c873ad5，0xe49798f6 ．word \(0 \times 5 e e 5 d b d 7,0 \times 7 e b 3 b 718,0 \times 1 c 0 e d 460,0 \times 433 c 9361,0 \times 61\) aced48 ．word 0x28bf3c86，0xf41873a，0xccd2f31c，0x13ed836d，0xe830f5eb ．word \(0 x 812 e 27 d 9,0 x a e 8 \pm 6 c 92,0 x 99096 f a d, 0 x 25 f 9 e 4 c 0,0 x f a a 7 f d 57\) ．word 0x75ded7f3，0xbf9524fb，0xaf957d59，0xdfac60b7，0xa65e88e6 ．word 0x287285e，0xc6356b0d，0x676daf30，0x4307a1ff，0xfd57d5ec ．word 0x22d976f7，0xe6a27285，0xef1fb97c，0xa3cc1964，0xbeb21eb5 ．word 0xddd72e7d，0xd910190c，0x2ef16bf1，0xaba2997b，0x9f33ee5f ．word \(0 \times 28 c 55 b 42,0 x b e 33 e 171,0 x 76 b 4 e 5 b e, 0 x e 5 f 2877 b, 0 x 1945 b 465\) ．word 0xc13eab99，0xe906394f，0xeade7290，0xf15743e3，0x24670e91 ．word 0x9349c0a5，0xe7290065，0x7bafab74，0x14800cf3，0xbfe5df27 ．word 0x5beb4bfa，0xa6eb30af，0x8cf85c5d，0xeb7ffff5，0x5dd77b5f ．word 0xcd378a5e，0xebae3£35，0x16355e01，0xfbaf72c5，0xb5aa2569 ．word \(0 x 627 c c 3 f f, ~ 0 x f f f f f f f f f, ~ 0 x d 57667 d 1, ~ 0 x 980657 b a, ~ 0 x f 3459 c a 1\) ．word 0x75c3dcf1，0x5699bdef，0x38ce574e，0xcddf4520，0x1cb6ab6a ．word \(0 x b f 8 b 22 d b, 0 x 67 e 08269,0 x b c a 1231 f, 0 x 562 b c a 12,0 x 31 f d 634 b\) ．word 0x5bd7aeac，0x401a67ee， \(0 \times 58956\) f79， \(0 \times 1660 \mathrm{~d} 5 \mathrm{~d} 1,0 \times 4 \mathrm{f} 67 \mathrm{eb} 47\) ．word 0x4ca6900c，0xaed2c412，0x5c24251b，0x6ca21703，0xb9b1b6fa ．word 0xd1638c05，0x88315622，0xd836cc51，0x83b16626，0xd11b9fea ．word 0xbae9e77e，0x3975f694，0x6694240c，0xa97b1050，0x24584c85 ．word \(0 x 3 c 45 c 0 e e, ~ 0 x 6 d 11 a 82 e, ~ 0 x 1 b 5 f a f f, ~ 0 x a e f c 97 a d, 0 x 7 c 09 e d e 7\) ．word 0xb2f17eb0，0x75b94170，0xde57e2f，0xc09£2458，0x83c602a0 ．word 0xa7b5fc24，0x1a282e01，0xbccf92fc，0xc718fb35，0xc28af194 ．word 0xbc260827，0xf4ca5680，0x65389f2a，0x494806e3，0xcd22bc65 ．word \(0 \times 484 \mathrm{a} 0 \mathrm{~b} 80,0 \times 6 \pm 24 \mathrm{f} 6 \mathrm{de}, 0 \times 32 \mathrm{a} 423 \mathrm{e} 6,0 \times 79 \mathrm{a} 069 \mathrm{~d}, 0 \times 15068 \mathrm{a} 0\) ．word 0xb806f530，0x4e7d3065，0x8fe3be73，0xf62fb390，0xa50bd980 ．word \(0 x 729 e b 975,0 x d 28 b 6 d f 3,0 x a 25205 d 5,0 x 80729 a 82,0 x e 01 b c 9 a 0\) ．word 0xc852f8ef，0xb1aa806e，0x3e68050a，0x7db3f65e，0x60194a0b ．word 0x806£2bf9，0x8e217b41，0xaacbd979，0xca5f51f6，0xa32c34fa
．word 0x9827304e，0x6056ac1a，0xe41a2211，0xa006e380，0x69bfcd58 ．word 0x80329417，0x3292e01，0xbc93e3be，0xf1da02a0，0xa7de3f75 ．word 0xf350697d，0x175d75d7，0x5d751ec0，0xa57db2e9，0xa2393aeb ．word 0xaefd47e9，0x5b6cfdd，0x74004a0b，0x806£33c2，0x14918d78 ．word \(0 \times 466 \mathrm{~d} f 453\) ， \(0 x f 1 \mathrm{a} 0 \mathrm{a} 1 \mathrm{df}, 0 \times 68113 e 01,0 \times f c d b 6 f b f, 0 \times 1\) baebaeb ．word 0xaeb5fe63，0x8c7c2827，0x403f99c，0x2a27d665，0x5c0ca4b ．word 0x806d7ebf，0xebbf25eb，0x5f027b79，0xecbc5f8e，0x417de2b6 ．word 0x481cd417，0xde4d06，0x43fa8b89，0xf0b06429，0x7d77db2d ．word \(0 x 29 e 697 d 1, ~ 0 x 17832409, ~ 0 x 2 a 0 b 806 f, ~ 0 x 2 b e b 3389, ~ 0 x f c 9 f a 608\) ．word 0x657c778，0xa5ae03c5，0xfdd7c9fe，0x67b0814b，0xad0a0b80 ．word 0x6f53bc52，0xe98281fc，0xcf17eb07，0x5b941703，0x292d7cee ．word 0x74aa578a，0xbe8827db，0x82c7348a，0xf194a378，0xe65442d7 ．word 0xc371524a，0x642e10cb，0x9ecb63fb，0x17d11a69，0xbc219636 ．word \(0 \times 5 a 742\) fac， \(0 \times 865 d 30 d, 0 x 9585 d a 1 a, ~ 0 x 420235 b c, 0 \times 2195390 b\) ．word 0x2cda2373，0x62fbd968，0xb4050fc3，0xdea110b5，0xcb87843d ．word 0xe7710b5c，0xbc3c37b5，0xd62170af，0xa39b6cbe，0x5eee19d6 ．word \(0 x 49 e c 5 e 39, ~ 0 x a 5034 f c b, 0 x c e d 9 f b 2 f, ~ 0 x 383785 e a, ~ 0 x e 42 b 121 a\) ．word 0x678b7b2d，0x98d9655b，0x7e6dbd7b，0xdaf712b6，0xa1ea4ed9 ．word 0x6eb6a1e8，0xcfa86eb，0xa1f19101，0x9f0dd743，0xeec919f2 ．word 0xfdc553a2，0x56615f81，0xf0b88cd6，0xb4f7c64b，0xb3c553b7 ．word \(0 \times 5 \mathrm{~d} 0 f 8 \mathrm{c} 91,0 \times 9 e 1 e f 75 \mathrm{~b}, 0 \times 50 f 8 c f 9 \mathrm{~d}, 0 \times e e b a 1 \mathrm{f} 76,0 \times 48 c f 0 f 0 \mathrm{~d}\) ．word 0xd743f3c9，0x81e1d0fd，0x5770c19c，0x6e2a1c4a，0xcf35f9e7 ．word 0xc2e2335a，0xd3df192e，0xcf150f75，0xd0f8c919，0xc3dd679a ．word 0xfc0f85c4，0x66bbaea5，0xf23dd699，0xbaac4fb3，0xc09a686e ．word 0xab63b09d，0xafd3409，0xa686eb6a，0x1f9e75d7，0x75b61580 ．word 0xdfa619f0，0xb8cf35ac，0x2059bb09，0xfa68134c，0x3269a1bd ．word 0xd752ed2d，0xd699bae5，0xcf304c09，0x86eba1f9，0xe4c0ec30 ．word 0xf75d0f4c，0x32679e83，0xeeb6a1f，0x30cfa86e，0xba1f6781 ．word 0x303861ee，0xb6a1f3a0，0x7cef75d0，0xf9d024e8，0x1e187b8a ．word 0xa112b685，0x7c67c2e2，0x619ad743，0xe6192619，0xe1eeb497 ．word 0xc64c0eb9，0xeeba1f9e，0x4cf387ba，0xcc56bd2c，0x248ce1ee ．word 0xab23d390，0x92740896，0x1dd5627c，0x64d386ea，0xb63b09d0 ．word 0xafd38134，0xe1bae3a3，0x6bb09fa6，0x1934e04d，0x3437558f ．word 0x6b9c4ad3，0xdf9e44c3，0x3c550dd6，0x99bdd743，0xecf02607 ．word 0xf0dd563，0xdef79c67，0x2dd726c7，0xd9763cad，0x5c85e2c8 ．word 0x9cb7b6ec，0x4c89bfeb，0x2ec25914，0xd37eb22c，0x8b6dd6f6 ．word 0xaf33d9fd，0x97b178ec，0xe424e209，0x6452e4d5，0xe67ebfb4 ．word 0xfb431324，0x6e4da769，0xf686264b，0xb728b5f8，0xd5a0409d ．word 0xe3eb8e20，0x9fffffff5，0x5d8d3124，0x63576ca9，0xbfb525a6 ．word 0xf14a9bdc，0xd0d475cf，0xfff7ffff，0xfeabffff，0xfffedcd2 ．word 0x990a9bfb，0x9b71a8eb，0x9edad12a，0xbfbe33e1，0x70a75bff ．word 0xfffffaaf，0x96fd7fd2，0x2bf1f67c，0x55d0f524，0x5270f75a ．word 0x4be32607，0x6d7755e5，0xf7648cec，0x7d807fff，0xffaaedf3 ．word 0xb8438dd6，0x9efbb261，0xdadbaaf2，0xfbb24670，0xe853bfff ．word 0xfeabb8cb，0x1c763bc3，0xdd692fbb，0x26076b6e，0xba1f1923 ．word 0xc829d437，0x5a5f6b6e，0xb3cd7c6c，0x12941dc2，0x935db6b1 ．word 0x2b4b13c4，0x8dd699bd，0xef38ce72，0x372d5d22，0xcaf152 ．word 0x194983bd，0xe4713226，0xffacbb09，0x6452e4d5，0xe67e900a ．word 0x69fafe67，0xb3e264b3，0xc1c999f6，0x68b68c8f，0x75ed90ca ．word \(0 x 4 c 1 e 67 d d, ~ 0 x 767 c 412 c, 0 x 861 c 9 b 4 e, 0 x d 3 e d 0 c 4 c, 0 x 91 b 93579\) ．word 0x9f628885，0x21a7ebfb，0x4fb43104，0xd743e322，0x93e1b6de ．word 0x2557f798，0x3e17119a，0xd69ef304，0xc0edbeeb，0xa1f19233 ．word 0x87bdd743，0xe3247900，0x7bad2c4f，0x12375591，0xf6781232 ．word \(0 \times 29 \mathrm{~d} 80000\)
huff＿compressed＿text＿end：
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline \multicolumn{7}{|l|}{huff＿dictionary：} \\
\hline ．asciiz & ＂ & 非 Idx & 0 & Freq & 81 & Enc 10101 \\
\hline ．asciiz＂\＄＂ & & \＃ Idx & 9 & Freq & 71 & Enc 01111 \\
\hline ．asciiz＂， & & \＃1 Idx & 12 & Freq & 57 & Enc 00111 \\
\hline ．asciiz＂，\＄＂ & & 非 Idx & 15 & Freq & 55 & Enc 00100 \\
\hline ．asciiz＂t1＂ & & \＃ Idx & 19 & Freq & 38 & Enc 100011 \\
\hline ．asciiz & ．\({ }^{\prime}\) & 非 Idx & 22 & Freq & 28 & Enc 001011 \\
\hline ．asciiz＂．\({ }^{\text {n＂}}\) & & \＃1 Idx & 32 & Freq & 24 & Enc 1100010 \\
\hline ．asciiz＂addi＂ & & 非 Idx & 35 & Freq & 24 & Enc 1101000 \\
\hline ．asciiz＂：\n＂ & & \＃ Idx & 40 & Freq & 18 & Enc 1000100 \\
\hline ．asciiz＂ascii＂ & & \＃1 Idx & 43 & Freq & 18 & Enc 1000000 \\
\hline ．asciiz＂a0＂ & & 非 Idx & 49 & Freq & 15 & Enc 0101001 \\
\hline ．asciiz＂\＂\n＂ & & \＃1 Idx & 52 & Freq & 15 & Enc 0101000 \\
\hline ．asciiz＂of＂ & & \＃ Idx & 55 & Freq & 14 & Enc 0011001 \\
\hline ．asciiz＂t0＂ & & \＃ Idx & 58 & Freq & 14 & Enc 0011000 \\
\hline ．asciiz＂\＂＂ & & 非 Idx & 61 & Freq & 14 & Enc 0011011 \\
\hline ．asciiz＂t2＂ & & 非 Idx & 64 & Freq & 12 & Enc 10111011 \\
\hline ．asciiz＂the＂ & & \＃1 Idx & 67 & Freq & 12 & Enc 10111010 \\
\hline ．asciiz＂t3＂ & & 非 Idx & 71 & Freq & 12 & Enc 11001111 \\
\hline ．asciiz＂（\＄＂ & & \＃1 Idx & 74 & Freq & 11 & Enc 10111000 \\
\hline ．asciiz＂to＂ & & \＃ Idx & 77 & Freq & 10 & Enc 10110010 \\
\hline ．asciiz＂：＂ & & \＃1 Idx & 80 & Freq & 9 & Enc 01110010 \\
\hline ．asciiz & 非 \＄＂ & 非 Idx & 83 & Freq & 8 & Enc 01100100 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|}
\hline ．asciiz＂string＂ & \＃\(⿰ ⿰ 三 丨 ⿰ 丨 丨 ⿱ 一 土 寸\) Idx & 95 & Freq & 8 & Enc 01100111 \\
\hline ．asciiz＂）\n＂ & \＃Idx & 102 & Freq & 7 & Enc 00110101 \\
\hline ．asciiz＂非\n＂ & \＃\＃Idx & 105 & Freq & 7 & Enc 00001001 \\
\hline ．asciiz＂address＂ & \＃ 1 Idx & 116 & Freq & 6 & Enc 101110011 \\
\hline ．asciiz＂table＂ & \＃ Idx & 124 & Freq & 6 & Enc 101110010 \\
\hline ．asciiz＂character＂ & \＃Idx & 130 & Freq & 6 & Enc 110101111 \\
\hline ．asciiz＂miss＂ & \＃Idx & 140 & Freq & 5 & Enc 100010111 \\
\hline ．asciiz＂nop＂ & \＃ Idx & 145 & Freq & 5 & Enc 101001100 \\
\hline ．asciiz＂bne＂ & \＃Idx & 149 & Freq & 5 & Enc 101001111 \\
\hline ．asciiz＂beq＂ & \＃ Idx & 153 & Freq & 5 & Enc 101001001 \\
\hline ．asciiz＂非 & \＃Idx & 157 & Freq & 5 & Enc 101001000 \\
\hline ．asciiz＂Address＂ & \＃ Idx & 164 & Freq & 4 & Enc 011010101 \\
\hline ．asciiz＂and＂ & \＃Idx & 172 & Freq & 4 & Enc 011010111 \\
\hline ．asciiz＂ULOOP＂ & \＃ Idx & 176 & Freq & 4 & Enc 011010110 \\
\hline ．asciiz＂examined＂ & \＃Idx & 182 & Freq & 4 & Enc 011010000 \\
\hline ．asciiz＂being＂ & \＃Idx & 191 & Freq & 4 & Enc 011010011 \\
\hline ．asciiz＂histo＂ & \＃ 1 Idx & 197 & Freq & 4 & Enc 011010010 \\
\hline ．asciiz＂upper＂ & \＃ Idx & 203 & Freq & 4 & Enc 011011101 \\
\hline asciiz＂非 & \＃ Idx & 209 & Freq & 3 & Enc 110011100 \\
\hline
\end{tabular}

．asciiz＂stars＂非 Idx 295 Freq 3 Enc 1100111011
asciiz＂非＂非 Idx 301 Freq 3 Enc 1100111010
．asciiz＂know＂非 Idx 306 Freq 3 Enc 1100100101
．asciiz＂The＂非 Idx 311 Freq 3 Enc 1100100100
．asciiz＂one＂非 Idx 315 Freq 3 Enc 1100100111
．asciiz＂，－＂非 Idx 319 Freq 3 Enc 1100100000
．asciiz＂strlen＂非 Idx 323 Freq 3 Enc 1100100011
．asciiz＂非＂非 Idx 330 Freq 3 Enc 1100100010
．asciiz＂Load＂非 Idx 342 Freq 3 Enc 1100101101
．asciiz＂100＂非 Idx 347 Freq 3 Enc 1100101111
asciiz＂add＂非 Idx 351 Freq 3 Enc 1100101110
．asciiz＂\＂\＂\n＂非 Idx 355 Freq 3 Enc 1100101001
．asciiz＂element＂非 Idx 360 Freq 3 Enc 1100101000
．asciiz＂And＂⿰⿰三丨⿰丨三一 Idx 368 Freq 3 Enc 1100101011
．asciiz＂next＂非 Idx 372 Freq 3 Enc 1100101010
．asciiz＂LOOP＂非 Idx 377 Freq 3 Enc 1101010100
．asciiz＂char＂非 Idx 382 Freq 3 Enc 1101010110
．asciiz＂that＂非 Idx 387 Freq 3 Enc 1101010001
．asciiz＂histogram＿data＂非 Idx 392 Freq 3 Enc 1101010000
．asciiz＂jal＂非 Idx 407 Freq 3 Enc 1101010010
．asciiz＂you＂⿰⿰三丨⿰丨三一 Idx 411 Freq 3 Enc 1101011101
．asciiz＂．．．＂非 Idx 415 Freq 3 Enc 1101011100
．asciiz＂－＞＂非 Idx 419 Freq 2 Enc 0110000001
．asciiz＂Return＂非 Idx 424 Freq 2 Enc 0110011001
．asciiz＂text＂非 Idx 431 Freq 2 Enc 0110011000
．asciiz＂not＂非 Idx 436 Freq 2 Enc 0110011010
．asciiz＂TB＿100＂非 Idx 440 Freq 2 Enc 0101110101
．asciiz＂Test＂非 Idx 447 Freq 2 Enc 0101110100
．asciiz＂非＂非 Idx 452 Freq 2 Enc 0101110111
．asciiz＂copy＂非 Idx 458 Freq 2 Enc 0101110110
．asciiz＂result＂非 Idx 463 Freq 2 Enc 0101110001
．asciiz＂return＂非 Idx 470 Freq 2 Enc 0101110000
．asciiz＂TEST＂非 Idx 477 Freq 2 Enc 0101110011
．asciiz＂space＂非 Idx 482 Freq 2 Enc 0101110010
．asciiz＂out＂非 Idx 488 Freq 2 Enc 0101111101
．asciiz＂asciiz＂非 Idx 492 Freq 2 Enc 0101111100
．asciiz＂lbu＂非 Idx 499 Freq 2 Enc 0101111111
．asciiz＂slti＂非 Idx 503 Freq 2 Enc 0101111001
．asciiz＂．．．＂非 Idx 508 Freq 2 Enc 0101111000
．asciiz＂＿＿start＂非 Idx 513 Freq 2 Enc 0101111011
．asciiz＂then＂非 Idx 521 Freq 2 Enc 0101111010
．asciiz＂Usage＂非 Idx 526 Freq 2 Enc 0101100101
．asciiz＂sub＂非 Idx 532 Freq 2 Enc 0101100100
．asciiz＂Make＂⿰⿰三丨⿰丨三一 Idx 536 Freq 2 Enc 0101100111
．asciiz＂there＂非 Idx 541 Freq 2 Enc 0101100110
．asciiz＂．）\n＂非 Idx 547 Freq 2 Enc 0101100001
．asciiz＂New＂非 Idx 551 Freq

Enc 0101100000
Enc 0101100011
Enc 0101100010
Enc 0101101101
Enc 0101101100
Enc 0101101111
Enc 0101101110
Enc 0101101001
Enc 0101101000
Enc 0101101011
Enc 0101101010
Enc 0110110101
Enc 0110110100
Enc 0110110111
Enc 0110110001
\begin{tabular}{llll} 
．asciiz＂used＂ & 非 Idx & 645 & Freq \\
．asciiz＂doi＂ & 非 Idx & 650 & Freq \\
．asciiz＂index＂ & 非 Idx & 654 & Freq \\
．asciiz＂means＂ & \＃Idx & 660 & Freq \\
．asciiz＂Register＂ & \＃Idx & 666 & Freq
\end{tabular}
```

Enc 0110110011
Enc 0110110010
Enc 0110111101
Enc 0110111100
Enc 0110111111

```

非 Huffman Index Tree．
非
huff_tree_right_base:
    .half 0x80 非 Base for right child index.
huff_tree_literal_base:
    .half 0 非 Base for literal.
huff_tree_dictionary_base:
            .half \(0 \times 7000\) 非 Base for dict.
非 Huffman Lookup Tree
非
huff_tree:
.half 0x0162 非 Tree Idx 0
        \(\begin{array}{llll}. h a l f & 0 x 00 a 1 & \text { 非 Tree Idx } & 1 \\ 0 \\ . h a l f & 0 x 0090 & \text { 非 Tree Idx } & 2 \\ 00\end{array}\)
        .half 0x008d 非 Tree Idx 3000
        .half 0x0086 非 Tree Idx 40000
        .half 0x0069 非 Tree Idx 500000
        .half 0x008c 非 Tree Idx 600001
        .half 0x008b 非 Tree Idx 7000010
        \(\begin{array}{llll}. h a l f & 0 x 008 a & \text { 非 Tree Idx } & 8 \\ 0000100 \\ . h a l f ~ 0 x 0025 & \text { 非 Tree Idx } & 9 & 00001000\end{array}\)
        .half 0x7069 非 Tree Idx 1000001001
        .half 0x0077 非 Tree Idx 110000101
        .half 0x0031 非 Tree Idx 12000011
        .half 0x008f 非 Tree Idx 130001
        .half 0x0072 非 Tree Idx 1400010
        . half 0x006e 非 Tree Idx 1500011
        .half 0x0096 非 Tree Idx 16001
        .half 0x0093 非 Tree Idx 170010
        .half \(0 \times 700 \pm\) 非 Tree Idx 18
        \begin{tabular}{lll}
. half \(0 \times 0095\) & 非 Tree Idx 19 \\
. half \(0 \times 0067\) & 非 Tree Idx 20 \\
\hline
\end{tabular}
        .half 0x7016 非 Tree Idx 21 0
        .half 0x00a0 非 Tree Idx 220011
        .half 0x009b 非 Tree Idx 2300110
        .half 0x009a 非 Tree Idx 24001100
        .half 0x703a 非 Tree Idx 250011000
        .half 0x7037 非 Tree Idx 260011001
        .half 0x009f 非 Tree Idx 27001101
        .half 0x009e 非 Tree Idx 280011010
        .half 0x0079 非 Tree Idx 2900110100
        .half 0x7066 非 Tree Idx 3000110101
        .half 0x703d 非 Tree Idx 310011011
        .half 0x700c 非 Tree Idx 3200111
        .half 0x00ed 非 Tree Idx 3301
        .half 0x00a6 非 Tree Idx 34010
        .half 0x00a5 非 Tree Idx 350100
        .half 0x0061 非 Tree Idx 3601000
        .half 0x0074 非 Tree Idx 3701001
        .half 0x00ae 非 Tree Idx 380101
        .half 0x00ab 非 Tree Idx 3901010
        .half 0x00aa 非 Tree Idx 40010100
        .half 0x7034 非 Tree Idx 410101000
        .half 0x7031 非 Tree Idx 420101001
        .half 0x00ad 非 Tree Idx 43010101
        .half 0x0023 非 Tree Idx 440101010
        .half 0x0070 非 Tree Idx 450101011
        .half 0x00ce 非 Tree Idx 4601011
        .half 0x00bf 非 Tree Idx 47010110
        .half 0x00b8 非 Tree Idx 480101100
        .half 0x00b5 非 Tree Idx 4901011000
        .half 0x00b4 非 Tree Idx 50010110000
        .half \(0 \times 7227\) 非 Tree Idx 510101100000
        .half 0x7223 非 Tree Idx 520101100001
        .half 0x00b7 非 Tree Idx 53010110001
        .half \(0 \times 7233\) 非 Tree Idx 540101100010
        .half 0x722b 非 Tree Idx 55
        .half 0x00bc \# Tre Idx 560101100011
        n 01011001
        .half 0x00bb 非 Tree Idx 57010110010
        .half \(0 \times 7214\) 非 Tree Idx 580101100100
        .half 0x720e 非 Tree Idx 590101100101
        .half 0x00be 非 Tree Idx 60010110011
        .half 0x721d 非 Tree Idx 610101100110
        .half 0x7218 非 Tree Idx 620101100111

Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂i＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂\％＂
Dict Idx for＂非\n＂
Literal＂w＂
Literal＂1＂
Pointer to right child．
Literal＂r＂
Literal＂n＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂，\＄＂
Pointer to right child．
Literal＂g＂
Dict Idx for＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂t0＂
Dict Idx for＂of＂
Pointer to right child．
Pointer to right child．
Literal＂y＂
Dict Idx for＂）\n＂
Dict Idx for＂\＂＂
Dict Idx for＂，＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂a＂
Literal＂t＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂\＂\n＂
Dict Idx for＂a0＂
Pointer to right child．
Literal＂非＂
Literal＂p＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂New＂
Dict Idx for＂．）\n＂
Pointer to right child．
Dict Idx for＂mtc1＂
Dict Idx for＂syscall＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂sub＂
Dict Idx for＂Usage＂
Pointer to right child．
Dict Idx for＂there＂
Dict Idx for＂Make＂
．half 0x00c7 非 Tree Idx 630101101 ．half 0x00c4 非 Tree Idx 64 ．half 0x00c3 非 Tree Idx 65 ．half 0x7259 非 Tree Idx 66 ．half 0x724f 非 Tree Idx 67 ．half 0x00c6 非 Tree Idx 68 ．half 0x7262 非 Tree Idx 69 \(\begin{array}{lll}\text { ．half } 0 x 725 d & \text { 非 Tree Idx } 70 \\ . h a l f & 0 x 00 c b & ⿰ ⿰ 三 丨 ⿰ 丨 三 一\end{array}\) Tree Idx 71 ．half 0x00ca ⿰⿰三丨⿰丨三一俍 Tree Idx 72 ．half 0x723e ．half 0x7238 非 Tree Idx 74 ．half 0x00cd 非 Tree Idx 75 ．half 0x7247 非 Tree Idx 76 ．half 0x7242 非 Tree Idx 77 ．half 0x00de 非 Tree Idx 78 ．half 0x00d7 非 Tree Idx 79 ．half 0x00d4 非 Tree Idx 80 ．half 0x00d3 非 Tree Idx 81 ．half 0x71d6 非 Tree Idx 82 ．half 0x71cf 非 Tree Idx 83 ．half \(0 \times 71\)非 Tree Idx 85 ．half 0x00db 非 Tree Idx 87 ．half 0x00da 非 Tree Idx 88 ．half \(0 \times 71\) bf 非 Tree Idx 89 ．half 0x71b8 非 Tree Idx 90 ．half 0x00dd ．half 0x71ca 非 Tree Idx ．half 0x71c4 ．half \(0 \times 00 \mathrm{e} 6\) 非 Tree Idx 94 ．half \(0 x 00 \mathrm{e} 3\) 非 Tree Idx 95 ．half 0x00e2 非 Tree Idx 96 ．half \(0 \times 71 \mathrm{fc}\) 非 Tree Idx 97 ．half 0x71さ7 非 Tree Idx 98 ．half 0x7209 非 Tree Idx 100 ．half 0x7201 非 Tree Idx 101 ．half 0x00ea 非 Tree Idx 102 ．half 0x00e9 非 Tree Idx 103 ．half 0x71ec 非 Tree Idx 104 ．half 0x71e8 非 Tree Idx 105 ．half 0x00ec 非 Tree Idx 106 ．half 0x0044 非 Tree Idx 107 ．half 0x71f3 非 Tree Idx 108 ．half 0x0159 非 Tree Idx 109 ．half 0x012c 非 Tree Idx 110 ．half 0x011f 非 Tree Idx 111 ．half 0x0100 非 Tree Idx 112 ．half 0x00ff 非 Tree Idx 113 ．half 0x00f8 非 Tree Idx 114 ．half 0x00f7 非 Tree Idx 115 ．half 0x00f6 非 Tree Idx 116 ．half 0x0037 ⿰⿰三丨⿰丨三一 Tree Idx 117 ．half 0x005f 非 Tree Idx 118 ．half 0x71a3 非 Tree Idx 119 ．half 0x00fc 非 Tree Idx 120 ．half 0x00fb 非 Tree Idx 121 ．half 0x002c 非 Tree Idx 122 ．half 0x004e 非 Tree Idx 123 ．half 0x00fe 非 Tree Idx 124 ．half 0x002a 非 Tree Idx 125 ．half 0x007c 非 Tree Idx 126 ．half 0x0036 非 Tree Idx 127 ．half 0x0110 ⿰⿰三丨⿰丨三一 Tree Idx 128 ．half 0x0109 非 Tree Idx 129 ．half 0x0106 非 Tree Idx 130 ．half 0x0105 非 Tree Idx 131 ．half 0x0060 非 Tree Idx 132 ．half 0x003f 非 Tree Idx 133 ．half 0x0108 非 Tree Idx 134 ．half 0x0078 非 Tree Idx 135 ．half 0x007e 非 Tree Idx 136 ．half 0x010d 非 Tree Idx 137 ．half 0x010c 非 Tree Idx 138 ．half 0x007b 非 Tree Idx 139 ．half 0x0040 非 Tree Idx 140 ．half 0x010f 非 Tree Idx 141 ．half 0x007d 非 Tree Idx 142 ．half 0x003b 非 Tree Idx 143

01011010 010110100 0101101000 0101101001 010110101 0101101010 0101101011 01011011 010110110 0101101100 0101101101 010110111 0101101110 0101101111 010111 0101110 01011100 010111000 0101110000 0101110001 010111001 0101110010 0101110011 01011101 010111010 0101110100 0101110101 010111011 0101110110 0101110111 0101111 01011110 010111100 0101111000 0101111001 010111101 0101111010 0101111011 01011111 010111110 0101111100 0101111101 010111111 0101111110 0101111111 011 0110 01100 011000 0110000 01100000 011000000 0110000000 01100000000 01100000001 0110000001 011000001 0110000010 01100000100 01100000101 0110000011 01100000110 01100000111 01100001 0110001 01100010 011000100 0110001000 01100010000 01100010001 0110001001 01100010010 01100010011 01100010 01100010100 Literal＂ई＂ 01100010101 Literal＂＠＂ 01100010110 Literal＂\}"
01100010111 Literal＂；＂

0110001010 Pointer to right child．

0110001011 Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂for＂
Dict Idx for＂）非＂
Pointer to right child．
Dict Idx for＂DONE＂
Dict Idx for＂what＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂cvt＂
Dict Idx for＂start＂
Pointer to right child．
Dict Idx for＂Orleans＂
Dict Idx for＂．．\n＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂return＂
Dict Idx for＂result＂
Pointer to right child．
Dict Idx for＂space＂
Dict Idx for＂TEST＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂Test＂
Dict Idx for＂TB＿100＂
Pointer to right child．
Dict Idx for＂copy＂
Dict Idx for＂非＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂．．．＂
Dict Idx for＂slti＂
Pointer to right child．
Dict Idx for＂then＂
Dict Idx for＂＿＿start＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂asciiz＂
Dict Idx for＂out＂
Pointer to right child．
Literal＂D＂
Dict Idx for＂lbu＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂7＂
Literal＂＿＂
Dict Idx for＂－＞＂
Pointer to right child．
Pointer to right child．
Literal＂，＂
Literal＂N＂
Pointer to right child．
Literal＂＊＂
Literal＂｜＂
Literal＂6＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂｀＂
Literal＂？＂
Pointer to right child． Literal＂x＂
iteral＂～＂
Pointer to right child．
Pointer to right child．
\begin{tabular}{|c|c|c|c|}
\hline 18 & 非 Tree Idx 144 & 011 & Pointer to right child． \\
\hline ．half 0x0115 & Tree Idx 145 & 011000110 & Pointer to right child． \\
\hline ．half 0x0114 & Tree Idx 14 & 0110001100 & Pointer to right child． \\
\hline ．half 0x004a & Tree Idx 147 & 01100011000 & Literal＂J＂ \\
\hline ．half 0x0056 & Tree Idx 148 & 01100011001 & Literal＂V \\
\hline ．half 0x0117 & Tree Idx 149 & 0110001101 & Pointer to \\
\hline ．half 0x0045 & Tree Idx 150 & 01100011010 & Literal＂E＂ \\
\hline ．half 0x005b & Tree Idx 151 & 01100011011 & Literal \\
\hline ．half 0x011c & Tree Idx 152 & 011 & Pointer to \\
\hline ．half 0x011b & ee Idx 153 & 011000111 & Pointer to right child． \\
\hline ．half 0x005e & Tree Idx 154 & 01100011100 & Literal \\
\hline lf 0x002b & 155 & 01100011101 & Literal \\
\hline ．half 0x011e & 56 & 01100 & Pointer to \\
\hline f 0x0055 & dx 157 & 01100011110 & Literal＂U＂ \\
\hline ．half 0x0039 & ee Idx 158 & 01100011111 & Literal \\
\hline ．half 0x0123 & Tree Idx 159 & 011 & Pointer to right child． \\
\hline ．half 0x0122 & 160 & 0110010 & Pointer to right child． \\
\hline ．half 0x7053 & 161 & 01100100 & Dict Idx for \\
\hline ．half 0x002f & ee Idx 162 & 01100101 & Literal \\
\hline ．half 0x012b & 163 & 01 & Pointer to \\
\hline ．half 0x0128 & Tree Idx 16 & 01100110 & Pointer to right child． \\
\hline ．half 0x0127 & ee Idx 165 & 011001100 & Pointer to right child． \\
\hline ．half 0x71af & ee Idx 166 & 0110011000 & Dict Idx for＂text＂ \\
\hline ．half 0x71a8 & ee Idx 167 & 0110011001 & Dict Idx for＂Return＂ \\
\hline ．half 0x012a & Tree Idx 168 & 011001101 & Pointer to right child． \\
\hline ．half 0x71b4 & ee Idx 169 & 0110011010 & Dict Idx for＂not＂ \\
\hline ．half 0x0052 & 0 & 0110011011 & Literal \\
\hline ．half 0x705f & 171 & 011001 & Dict Idx for＂string＂ \\
\hline ．half 0x & \＃Tree Idx 172 & 01 & Pointer to right child． \\
\hline ．half 0x0135 & ee Idx 173 & 011010 & Pointer to right child． \\
\hline ．half 0x0132 & 非 Tree Idx 174 & 0110100 & Pointer to right child． \\
\hline ．half 0x0131 & d 175 & 01101000 & Pointer to right child． \\
\hline ．half 0x70b6 & ee Idx 176 & 011010000 & Dict Idx for＂examined＂ \\
\hline ．half 0x005c & 非 Tree Idx 177 & 011010001 & Literal＂\134＂ \\
\hline ．half 0x0134 & 178 & 01101001 & Pointer to right chil \\
\hline ．half 0x & 79 & 011010010 & Dict Idx for＂histo＂ \\
\hline ．half 0x70bf & dx 180 & 011010011 & Dict Idx for＂being＂ \\
\hline ．half 0x0139 & 181 & 01 & Pointer to right child． \\
\hline ．half 0x0138 & 182 & 01101010 & Pointer to right child． \\
\hline ．half 0x0042 & dx 183 & 011010100 & Literal＂B＂ \\
\hline ．half 0x70a4 & dx 184 & 011010101 & Dict Idx for＂Address＂ \\
\hline ．half 0x013b & 185 & 01101011 & Pointer to right child． \\
\hline ．half 0x70b0 & dx 186 & 011010110 & Dict Idx for＂ULOOP＂ \\
\hline ．half 0x70ac & ee Idx 187 & 01101011 & Dict Idx for＂and＂ \\
\hline ．half 0x014c & dx 188 & 011 & Pointer to right child． \\
\hline ．half 0x0145 & ＋ 189 & 0110110 & Pointer to right child． \\
\hline ．half 0x0142 & dx 190 & 01101100 & Pointer to right child． \\
\hline ．half 0x0141 & \＃Tree Idx 191 & 011011000 & Pointer to right child． \\
\hline ．half 0x003e & 192 & 0110110000 & \\
\hline ．half 0x727d & dx 193 & 0110110001 & Dict Idx for＂Convert＂ \\
\hline ．h & \＃Tree Idx 194 & 01 & Pointer to right chi \\
\hline ．half 0x728a & dx 195 & 0110110010 & Dict Idx for＂doi＂ \\
\hline ．half 0x7285 & 非 Tree Idx 196 & 0110110011 & Dict Idx for＂used＂ \\
\hline ．half 0x0149 & 197 & 011 & Pointer to right child． \\
\hline ．half 0x0148 & \＃Tree Idx 198 & 011011010 & Pointer to right child． \\
\hline ．half 0x726d & dx 199 & 0110110100 & Dict Idx for＂Characte \\
\hline ．half 0x7267 & 200 & 0110110101 & Dict Idx for＂UDONE＂ \\
\hline ．half 0x014b & 201 & 011 & Pointer to right child． \\
\hline ．half 0x003d & ＋ 202 & 0110110110 & Literal \\
\hline ．half 0x7277 & ee Idx 203 & 0110110111 & Dict Idx for＂SLOOP＂ \\
\hline ．half 0x0150 & Tree Idx 204 & 011 & Pointer to right child． \\
\hline ．half 0x014f & ee Idx 205 & 1101110 & Pointer to right child． \\
\hline ．half 0x0041 & dx 206 & 011011100 & Literal＂A＂ \\
\hline ．half 0x70cb & ee Idx 207 & 011011101 & Dict Idx for＂upper＂ \\
\hline ．half 0x0154 & dx 208 & 01 & Pointer to right child． \\
\hline ．half 0x0153 & Tree Idx 209 & 011011110 & Pointer to right child． \\
\hline ．half 0x7294 & ee Idx 210 & 0110111100 & Dict Idx for＂means＂ \\
\hline ．half 0x728e & dx 211 & 0110111101 & Dict Idx for＂index＂ \\
\hline ．half 0x0158 & ee Idx 212 & 011011111 & Pointer to right child． \\
\hline ．half 0x0157 & Tree Idx 213 & 0110111110 & Pointer to right child． \\
\hline ．half 0x003c & Tree Idx 214 & 01101111100 & Literal＂＜＂ \\
\hline ．half 0x005d & Idx 215 & 01101111101 & Literal＂］＂ \\
\hline ．half 0x729a & Tree Idx 216 & 0110111111 & Dict Idx for＂Register＂ \\
\hline ．half 0x0161 & Tree Idx 217 & 011 & Pointer to right child． \\
\hline ．half 0x0160 & Tree Idx 218 & 01110 & Pointer to right child． \\
\hline ．half 0x015d & Tree Idx 219 & 011100 & Pointer to right child． \\
\hline ．half 0x0027 & Tree Idx 220 & 0111000 & Literal \\
\hline ．half 0x015f & Tree Idx 221 & 0111001 & Pointer to right child． \\
\hline ．half 0x7050 & \＃Tree Idx 222 & 01110010 & Dict Idx for＂： \\
\hline \(f\) 0x0022 & 非 Tree Idx 223 & 01110011 & Literal＂\＂＂ \\
\hline
\end{tabular}
．half 0x01a2 非 Tree Idx 2261 ．half 0x0179 非 Tree Idx 22710 ．half 0x0176 非 Tree Idx 228100 ．half 0x016d 非 Tree Idx 2291000 ．half 0x016a 非 Tree Idx 23010000 ．half 0x0169 非 Tree Idx 231100000 ．half 0x702b 非 Tree Idx 2321000000 ．half 0x0068 非 Tree Idx 2331000001 ．half 0x016c 非 Tree Idx 234100001 ．half 0x0064 非 Tree Idx 235 ．half 0x0034 非 Tree Idx 236 ．half 0x0175 非 Tree Idx 237 ．half 0x0170 非 Tree Idx 238 ．half 0x7028 非 Tree Idx 239 ．half \(0 \times 0172\) 非 Tree Idx 240 ．half 0x0174 非 Tree Idx 242 ．half \(0 x 0043\) 非 Tree Idx 243 \(\begin{array}{ll}\text { ．half 0x708c 非 Tree Idx } 244 \\ . h a l f ~ 0 x 7013 ~ & \text { 非 Tree Idx } 245\end{array}\) ．half 0x0178 非 Tree Idx 246 ．half 0x0065 非 Tree Idx 247 ．half 0x0073 非 Tree Idx 248 ．half 0x018f 非 Tree Idx 249 ．half 0x018e 非 Tree Idx 250 ．half 0x017f 非 Tree Idx 251 ．half 0x017e 非 Tree Idx 252 ．half 0x0032 非 Tree Idx 253 ．half 0x0075 非 Tree Idx 254 ．half 0x0187 非 Tree Idx 255 ．half 0x0184 非 Tree Idx 256 ．half 0x0183 非 Tree Idx 257 \(\begin{array}{ll}. h a l f 0 \times 709 \mathrm{~d} & \text { 非 Tree Idx } 258 \\ . h a l f 0 \times 7099 & \text { 非 Tree Idx } 259\end{array}\) ．half 0x0186 非 Tree Idx 260 ．half 0x002d 非 Tree Idx 261 ．half 0x006a 非 Tree Idx 262 ．half 0x018b 非 Tree Idx 263 ．half 0x018a 非 Tree Idx 264 ．half 0x7091 非 Tree Idx 265 ．half 0x003a 非 Tree Idx 266 ．half 0x018d 非 Tree Idx 267 ．half 0x0033 非 Tree Idx 268 ．half 0x7095 非 Tree Idx 269 ．half 0x7000 非 Tree Idx 270 ．half 0x0197 非 Tree Idx 271 ．half 0x0196 非 Tree Idx 272 ．half 0x0193 非 Tree Idx 273 ．haZf \(0 \times 0063\) 非 Tree Idx 274 ．half 0x704d 非 Tree Idx 276 ．half 0x0076 非 Tree Idx 277 ．half 0x006c 非 Tree Idx 278 ．half 0x01a1 非 Tree Idx 279 ．half 0x019e 非 Tree Idx 280 ．half 0x019b 非 Tree Idx 281 ．half 0x704a 非 Tree Idx 282 ．half 0x019d 非 Tree Idx 283 ．half \(0 \times 7074\) 非 Tree Idx 285 ．half 0x01a0 非 Tree Idx 286 ．half 0x7043 非 Tree Idx 287 ．half 0x7040 非 Tree Idx 288 ．half 0x006f 非 Tree Idx 289 ．half 0x01f6 非 Tree Idx 290 ．half 0x01d7 非 Tree Idx 291 ．half 0x01aa 非 Tree Idx 292 ．half 0x01a7 非 Tree Idx 293 ．half 0x0030 非 Tree Idx 294 ．half 0x01a9 非 Tree Idx 295 ．half 0x7020 非 Tree Idx 296 ．half 0x006d 非 Tree Idx 297 ．half 0x01ca 非 Tree Idx 298 ．half 0x01bb 非 Tree Idx 299 ．half 0x01b4 非 Tree Idx 300 ．half 0x01b1 非 Tree Idx 301 ．half 0x01b0 非 Tree Idx 302 ．half 0x713f 非 Tree Idx 303 ．half 0x0035 非 Tree Idx 304 ．half 0x01b3 非 Tree Idx 305

01111
1 10
100
1000 1000

100001
1000011
10001
100010
1000100
1000101
10001010
10001011
100010110
100010111
100011
1001
10010
10011
101
1010
10100
101000
1010000
1010001
101001
1010010
10100100
101001000
101001001
10100101
101001010
101001011
1010011
10100110
101001100
101001101
10100111
101001110
101001111
10101
1011
10110
101100
1011000
1011001
10110010
10110011
101101
10111
101110
1011100
10111000
10111001
101110010
101110011
1011101
10111010
10111011
101111
11
110
1100
11000
110000
110001
1100010
1100011
11001
110010
1100100
11001000
110010000
1100100000
1100100001
110010001

Dict Idx for＂\＄＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂ascii＂
Literal＂h＂
Pointer to right child．
Literal＂d＂
Literal＂4＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂：\n＂
Pointer to right child． Literal＂L＂
Pointer to right child． Literal＂C＂
Dict Idx for＂miss＂
Dict Idx for＂t1＂
Pointer to right child．
Literal＂e＂
Literal＂s＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂2＂
Literal＂u＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂非＂
Dict Idx for＂beq＂
Pointer to right child．
Literal＂－＂
Literal＂j＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂nop＂
Literal＂：＂
Pointer to right child．
Literal＂3＂
Dict Idx for＂bne＂
Dict Idx for＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂c＂
Pointer to right child．
Dict Idx for＂to＂
Literal＂v＂
Literal＂l＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂（\＄＂
Pointer to right child．
Dict Idx for＂table＂
Dict Idx for＂address＂
Pointer to right child．
Dict Idx for＂the＂
Dict Idx for＂t2＂
Literal＂o＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Literal＂0＂
Pointer to right child．
Dict Idx for＂．\({ }^{\text {n＂}}\)
Literal＂m＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂，－＂
Literal＂5＂
Pointer to right child．
\begin{tabular}{|c|c|c|}
\hline & \＃Tree Idx 306 & 1100100010 \\
\hline f 0x7143 & Tree Idx 307 & 110010001 \\
\hline ．half 0x01b8 & Tree Idx 308 & 11 \\
\hline lf 0x01b7 & Tree Idx 309 & 110010010 \\
\hline f 0x7137 & Tree Idx 310 & 1100100100 \\
\hline f 0x7132 & Tree Idx & 1100100101 \\
\hline f 0x01ba & Tree Idx & \\
\hline lf 0x0057 & Tree Idx 313 & 10 \\
\hline lf 0x713b & Tree Idx & 11 \\
\hline \(f 0 \times 0\) & Idx & \\
\hline lf 0x01c0 & Idx 316 & 11001010 \\
\hline lf 0x01bf & Tree Idx 317 & 110010100 \\
\hline ．half 0x7168 & 非 Tree Idx 318 & 1100101000 \\
\hline ．half 0x7163 & Tree Idx & \\
\hline lf 0x01c2 & Tree Idx 320 & 01 \\
\hline lf \(0 \times 7174\) & Idx 321 & 1100101010 \\
\hline ．half 0x & Idx & \\
\hline ．half 0x01c7 & Tree Idx 323 & 11001011 \\
\hline ．half 0x01c6 & Tree Idx 32 & 110 \\
\hline lf 0x007a & dx 325 & \\
\hline ．half 0x & Idx & \\
\hline lf 0x0 & Idx 327 & 110010111 \\
\hline ．half 0x715f & Idx 328 & 11 \\
\hline ．half 0x & \＃Tree Idx 329 & \\
\hline lf 0x0 & Idx 330 & \\
\hline lf 0x01cd & Tree Idx 331 & 1100110 \\
\hline ．half 0x0049 & Idx 332 & 00 \\
\hline ．half 0x0062 & Idx & \\
\hline ．half 0x0 & Idx & \\
\hline ．half 0x01d3 & Tree Idx 335 & 11001110 \\
\hline ．half 0x01d2 & Idx & 110011100 \\
\hline ．half 0x7 & Idx 33 & \\
\hline ．half 0x & Idx 338 & 1100111001 \\
\hline ．half 0x01d5 & Idx 339 & 110011101 \\
\hline ．half 0x & Idx & \\
\hline ．half 0x7127 & Tree Idx 34 & \\
\hline ．half 0x7047 & e Idx 342 & 11001111 \\
\hline ．half 0x01f5 & Idx & 1101 \\
\hline ．half 0x01dc & Idx & \\
\hline ．half 0x & Idx 345 & 00 \\
\hline ．half 0x7023 & Idx 34 & 000 \\
\hline ．half 0x0066 & Idx & 1101001 \\
\hline ．half 0x01ec & Tree Idx & \\
\hline ．half 0x01e5 & dx 3 & 1101010 \\
\hline ．half 0x01e2 & dx & 11010100 \\
\hline ．half 0x01e1 & Idx 3 & 110101000 \\
\hline ．half 0x7188 & Idx 352 & 1101010000 \\
\hline ．half 0x7183 & dx 353 & 1101010001 \\
\hline ．half 0x01e4 & dx & \\
\hline ．half 0x7197 & Tree Idx 355 & 1101010010 \\
\hline ．half 0x004d & Idx 356 & 11010100 \\
\hline ．half 0x01e9 & dx 357 & \\
\hline ．half 0x01e8 & dx 358 & \\
\hline ．half 0x7179 & Idx 359 & 1101010100 \\
\hline ．half 0x006b & Idx 360 & 110101010 \\
\hline ．half 0x01eb & ee Idx 361 & \\
\hline ．half 0x717e & 非 Tree Idx 362 & \\
\hline ．half 0x0053 & Idx 363 & 11010101 \\
\hline ．half 0x01f0 & ee Idx 364 & 11 \\
\hline ．half 0x01ef & Tree Idx 365 & 11010110 \\
\hline ．half 0x0028 & dx 36 & 110101100 \\
\hline ．half 0x0029 & Tree Idx 367 & 110101101 \\
\hline ．half 0x01f4 & Tree Idx 368 & 11010111 \\
\hline ．half 0x01f3 & 非 Tree Idx 369 & 110101110 \\
\hline ．half 0x719f & Tree Idx 370 & 110101110 \\
\hline ．half 0x719b & Tree Idx 371 & 110101110 \\
\hline ．half 0x7082 & Tree Idx 372 & 110101 \\
\hline \(f\) 0x000a & Tree Idx 373 & 1101 \\
\hline \(f\) 0x0020 & Tree Idx & 111 \\
\hline
\end{tabular}

Dict Idx for＂非＂
Dict Idx for＂strlen＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂The＂
Dict Idx for＂know＂
Pointer to right child．
Literal＂W＂
Dict Idx for＂one＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂element＂
Dict Idx for＂\＂\＂\n＂
Pointer to right child．
Dict Idx for＂next＂
Dict Idx for＂And＂
Pointer to right child．
Pointer to right child．
Literal＂z＂
Dict Idx for＂Load＂
Pointer to right child．
Dict Idx for＂add＂
Dict Idx for＂100＂
Pointer to right child．
Pointer to right child．
Literal＂I＂
Literal＂b＂
Pointer to right child．
Pointer to right child．
Pointer to right child．

Dict Idx for＂非＂
Pointer to right child．
Dict Idx for＂非＂
Dict Idx for＂stars＂
Dict Idx for＂t3＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂addi＂
Literal＂f＂
Pointer to right child．
Pointer to right child．
Pointer to right child．
Pointer to right child．
Dict Idx for＂histogram＿data＂
Dict Idx for＂that＂
Pointer to right child．
Dict Idx for＂jal＂
Literal＂M＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂LOOP＂
Literal＂k＂
Pointer to right child．
Dict Idx for＂char＂
Literal＂S＂
Pointer to right child．
Pointer to right child．
Literal＂（＂
Literal＂）＂
Pointer to right child．
Pointer to right child．
Dict Idx for＂．．．＂
Dict Idx for＂you＂
Dict Idx for＂character＂
Literal＂\n＂
Literal＂＂

Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Note: The following problem was assigned in each of the last five years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```


# Cycle }\quad0\quad

lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID EX ME WB

```

The add depends on the 1 w through r2, and for the illustrated implementation the add has to stall in ID until the 1 w reaches WB.
```


# Cycle 0

lw r2, O(r4) IF ID EX ME WB
add r1, r2, r7 IF ID ----> EX ME WB

```
(b) Explain error and show correct execution.
```


# Cycle }\quad0\begin{array}{llllllllll}{1}\&{2}\&{3}\&{4}\&{5}\&{6}\&{7}

add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID -> EX ME WB

```

There is no need for a stall because the 1 w writes r 1 , it does not read r 1 .
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID EX ME WB

```
(c) Explain error and show correct execution.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4)
IF ID -> EX ME WB

```

A longer stall is needed here because the sw reads r1 and it must wait until add reaches WB
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID ----> EX ME WB

```
(d) Explain error and show correct execution.
```


# Cycle }\quad0\quad

add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ----> ID EX ME WB

```

The stall above allows the xor, when it is in ID, to get the value of \(r 1\) written by the add; that part is correct. But, the stall starts in cycle 1 before the xor reaches ID, so how could the control logic know that the xor needed \(r 1\), or for that matter that it was an xor? The solution is to start the stall in cycle 2, when the xor is in ID.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ID ----> EX ME WB

```

Problem 2: Appearing below are incorrect executions on the illustrated implementation. Notice that this implementation is different than the one from the previous problem. For each execution explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.
```


# Cycle

0
lw r2, O(r4)
IF ID EX ME WB
add r1, r2, r7
IF ID EX ME WB

```

The add depends on the 1 w through r 2 , and for the illustrated implementation the add has to stall in ID until the 1 w reaches ME so that the add can bypass from WB.
```


# Cycle 0

lw r2, 0(r4) IF ID EX ME WB
add r1, r2, r7 IF ID -> EX ME WB

```
(b) Explain error and show correct execution.
```


# Cycle

0
add r1, r2, r3
lw r1, O(r4)
IF ID EX ME WB
IF ID -> EX ME WB

```

There is no need for a stall because \(r 1\) is not a source register of 1 w . Note that r 1 is a destination of 1 w .
```


# Cycle

0
add r1, r2, r3 IF ID EX ME WB
lw r1, O(r4) IF ID EX ME WB

```
(c) Explain error and show correct execution.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID -> EX ME WB

```

No stall is needed here because the sw can use the ME-to-EX bypass path.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
sw r1, O(r4) IF ID EX ME WB

```
(d) Explain error and show correct execution. Note that this execution differs from the one from the previous problem.
```


# Cycle

0
add r1, r2, r3
xor r4, r1, r5
IF ID EX ME WB
IF ID ----> EX ME WB

```

There is a bypass path available so that there is no need to stall.
```


# Cycle 0

add r1, r2, r3 IF ID EX ME WB
xor r4, r1, r5 IF ID EX ME WB

```

Problem 3: Appearing below is the labeled MIPS implementation from 2018 Midterm Exam Problem 2(b), and as in that problem each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.

(a) Use F0.

Solution appears below. Fo is used by values being loaded from memory into the pipeline. Load instructions, include 1 w , use F0.
```


# SOLUTION

# Cycle 0

lw r1, O(r2) IF ID EX ME WB

```
(b) Use F0 and C3 at the same time. The code should not suffer a stall.

The solution appears below. The C3 mux input is used to bypass something from WB to the first ALU operand. To use both at the same time we need some kind of a load instruction in WB writing rX at the same time there is some instruction in EX that uses \(r X\) as the first operand. In the solution below the lw writes \(r 1\) in cycle 4 . The instruction after the instruction after the \(1 w\), an add below, will be in EX at this time (since we are assuming no stalls). The first source operand of the add is set to r1 so that the C3 bypass input will be used.
```


# SOLUTION

# Cycle

lw r1, 0(r2) IF ID EX ME WB
xor r5, r6, r7 IF ID EX ME WB
add r3, r1, r4 IF ID EX ME WB

# Cycle

```
(c) Explain why its impossible to use E0 and D0 at the same time.

If E 0 is in use then there must be a store instruction in EX. If D0 is in use then a value is being bypassed to the second ALU source operand of the instruction in EX. But store instructions use an immediate for the second ALU input, so a store in EX can only use D2, it can't use D0 (nor D1 nor D3).

Problem 4: This problem appeared as Problem 2c on the 2020 final exam. Appearing below is our bypassed, pipelined implementation followed by a code fragment.

It might be helpful to look at Spring 2019 Midterm Exam Problem 4a. That problems asks for the execution of a loop and for a performance measure based upon how fast that loop executes.

(a) Show the execution of the code below on the illustrated implementation up to the point where the first instruction, addi \(\mathrm{r} 2, \mathrm{r} 2,16\), reaches WB in the second iteration.

The execution appears below. The execution is shown until the beginning of the third iteration. (A full-credit solution would only need to show execution until cycle 10 , when the addi \(r 2, r 2,16\) reaches WB in the second iteration.) The only stall is a 1 -cycle load/use stall suffered by the sw. The first iteration starts in cycle 0 (when the first instruction, addi, is in IF), the second iteration starts at cycle 6, and the third at cycle 12.

Note that the pattern of stalls in the second iteration is the same as the pattern in the first. We can expect this pattern to continue because the contents of the pipeline is the same at the beginning of the second and third iterations. (The second iteration begins in cycle 6. In that cycle the addi r2 is in IF, the addi r3 is in ID, etc. The contents of the pipeline is the same in cycle 12.)
```

        ## SOLUTION
    LOOP: \# Cycle 0
addi r2, r2, 16 IF ID EX ME WB First Iteration
lw r1, 8(r2)
Sw r1, 12(r3)
bne r3, r4, LOOP
addi r3, r3, 32
sub r10, r3, r2
LOOP: \# Cycle 0
addi r2, r2, 16 IF ID EX ME WB Second Iteration
lw r1, 8(r2)
sw r1, 12(r3)
bne r3, r4, LOOP
addi r3, r3, 32
LOOP: \# Cycle
addi r2, r2, 16 Third Iteration IF ID EX ME WB

```
(b) Based on your execution determine how many cycles it will take to complete \(n\) iterations of the loop.

The time for \(n\) iterations of the loop is \(n\) times the duration of one iteration of the loop. The key to solving this correctly is using the correct duration for an iteration. The duration of an iteration if the time between the start of two consecutive iterations. In this class the start time of an iteration is the time at which the first instruction is in IF. Using that definition the duration of the first iteration is \(6-0=6\) cyc and the duration of the second is \(12-6=6 \mathrm{cyc}\). So the number of cycles to complete \(n\) iterations is \(6 n\) cyc

An important point to understand is that the definition of duration above insures that iterations don't overlap. That is, by defining an iteration duration as starting in the IF of the first instruction of the iteration, there is no possibility that two iterations overlap and there is no time gap between them. That's what enables us to multiply a duration by the number of iterations to get a total time.

Some might be tempted to add another four cycles to account for the addi r3 instruction completing execution. No credit would be lost for that in a solution, but that is not useful for our purposes because we might want to add together the duration of different pieces of code, so for us the important thing is when the next instruction can be fetched.

Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Recall that code for the solution to Homework 2 included a loop that traversed a tree. The decision on whether to descend to the left or right child of a node was based on the next bit of compressed text. Several instructions were devoted to testing that next bit, and to checking whether a new word of bits needed to be loaded. In this assignment we are going to add a new instruction, bnbb (branch next bit big-endian), to MIPS that will allow such code to be written with fewer instructions.

Instruction bnbb rV, rP, TARG0, TARG1 works as follows. Register rV holds a bit vector, and register \(r P\) holds a position in the bit vector. (A bit vector is just a number, but it's called a bit vector when we are interested in examining specific bits in the number's binary representation.) If the value of \(r P\) is 0 then it refers to the MSB of \(r V\), if the value of \(r P\) is 1 it refers to position 1 (to the right of the MSB), etc. Let pos refer to bits 5:0 of rP. If pos is in the range 0 to 31 (inclusive) then bnbb will be taken, otherwise (values from 32 to 63 ) bnbb is not taken. When bnbb is taken it will branch to TARGO if bit pos in rV is 0 and to TARG1 if bit pos in rV is 1. Regardless of whether bnbb is taken register rP is written with rP+1. See the code and comments below:
```


# With sample values below bnbb is taken to LCHILD since bit 30 of 0x5 is zero.

# \$t8 = 0x5 (bit vector), \$t9 = 30 (pos)

bnbb \$t8, \$t9, LCHILD, RCHILD
addi \$v0, \$t9, 0 \# Delay slot insn. Here t9 is 31.

# This code is only executed when \$t9 in range 32-63 before bnbb executes.

# Fall through. Updates t8 and t9

addi \$t6, \$t6, 4 \# Update address ..
lw $t8, 0($t6) \# No more bits, load a new word.
addi \$t9, \$0, 0

```

The bnbb instruction can be used to eliminate at least two instructions in the hw02 solution. First, there would no longer be a need to shift the bit vector (the sll \(\$ \mathrm{t} 8, \$ \mathrm{t} 8\), 1 instruction). Instead, the bnbb instruction would automatically increment a bit position register. Also, there would no longer be a need for a second branch to check whether all 32 bits in the bit vector were examined. (That was the bne \$a1, \$t9, EXAMINE_NEXT_BIT instruction.)

In the subproblems below complete the specification for bnbb and show hardware to implement it.
An Inkscape SVG version of the hardware diagram can be found at
https://www.ece.lsu.edu/ee4720/2021/hw04-br-3way.svg.
(a) The description above leaves out a few details. In this problem fill them in. It may be helpful to attempt a solution to the next parts before answering this part.

Show a possible encoding for bnbb. That possible encoding must be based on format I. Show how the two targets are specified and and whether rV is encoded in the rt or rs fields.

The format appears below. The RT register field is used for rP and the RS for \(r V\). Placing \(r P\) in the RT field simplifies the decoding logic by a small amount. That's because the hardware needs to write back the incremented value of \(r P\) and there is already logic that sets the dst control signal to the RT field in format I instructions.

As with other branches, the bnbb will use displacement addressing, but unlike other branches there are two displacements (one displacement for TARGO and one displacement for TARG1). To accommodate two displacements the immediate field has been split in two as shown below. Note that with just eight bits of displacement the targets must be within 128 instructions of the branch ( 128 before the delay slot or 127 after the delay slot).
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline & Opcode & RS & RT & \multicolumn{2}{|l|}{Immo} & \multicolumn{2}{|l|}{Imm1} \\
\hline MIPS ID: & 0x2d & rV & rP & & & & \\
\hline & 26 & 21 & 16 & 15 & 8 & 7 & 0 \\
\hline
\end{tabular}

(b) For bbnb to work correctly the rP register value needs to be incremented. It would be nice if an existing ALU operation could do that. Explain why the add operation, used for the add, addi, lw, and other instructions, would not work.

The bbnb instruction needs to add a 1 to the rP value. The ALU add operation used for those instructions computes the sum of the upper and lower ALU inputs. One of those inputs would be the \(r P\) value, but there is no way to set the other ALU input to 1 with the illustrated hardware. The add operation could be used if a new input were added to one of the ALU multiplexors and that input was set to the constant 1.
(c) The diagram below shows a five-stage MIPS implementation including some branch hardware. Also shown is logic to detect the bnbb instruction and two placeholder wires, bnbb-t0-taken and bnbb-t1-taken. Wire bnbb-t0-taken should be set to 1 if there is a bnbb in the ID stage and it should be taken to TARGO. The definition of bnbb-t1-taken is similar. If there is not a bnbb in ID or if there is and it's not taken, then both wires should be 0 .

In this problem design the logic to drive those wires. (The solution to this and the following problem can be done on the same diagram, or on separate diagrams.)

The solution appears after part d.

(d) Modify the hardware below so that when bnbb-t0-taken is 1 target TARG0 is used and when bnbb-t1taken is 1 target TARG1 is used. Follow the points below.
- Design for lower cost rather than higher performance.
- There is an unused input on the PC mux. That can be used, but does not have to be used.
- As always, hardware must be reasonably efficient.
- As always, do not break other instructions.

The solution is on the next page.
\begin{tabular}{l} 
bnbb-t0-taken \\
bnbb-t1-taken \\
\hline
\end{tabular}


The solution for parts c and d appears below. A multiplexor extracts the needed bit from rsv. To keep costs low bnbb computes the target address using the same adder as other branch instructions.


Do not hand in paper copies. Instead, E-mail your solution to koppel@ece.lsu.edu. The preferred format is a PDF file.

Problem 1: Look over SPECcpu2017 run and reporting rules, available at http://www.spec.org/cpu2017/Docs/runrules.html. Start with Sections 1.1 to 1.5 and read other sections as needed to answer the questions below.
(a) Section 1.2.3 of the run and reporting rules lists several assumptions about the tester.

Consider the following testing scenario: The SUT (system being benchmarked) is a new product and that the tester works for the company that developed it. The company spent lots of money developing the product and their potential customers will use SPECcpu2017 when making buying decisions.
Explain why assumptions b and c seem reasonable given the testing scenario above.
It is in the company's interest that benchmark scores are high, so we can safely assume that they choose someone knowledgeable about the SUT (item c) and give that person time and other resources to learn what compilation options and other configuration details can be changed and how to best set them for the SUT(item \(b\) ). Even if the company intended to submit a misleading benchmark run, they would make sure that the tester new what the rules said.
Explain why assumption d also seems reasonable, given other stipulations set forth in the run and reporting rules (and discussed in class).

We can safely assume honesty on the part of the tester because any dishonesty would quickly be caught. The rules require that any publicly divulged results be accompanied by a config file which can be used to reproduce the results. We can safely assume that the company expects its competitors to buy a system and use the config file to reproduce the results. Further, the company would not expect its competitors to keep quite about any discrepancies.
(b) The SPECcpu benchmarks can be prepared at base and peak tuning levels (or builds). These are described in Section 1.5. Section 2.3 .1 stipulates that base optimizations are expected to be safe.
What is an unsafe optimization? (Points deducted for irrelevant or lengthy answers, especially if they appear copied.)

An optimization should transform unoptimized code into optimized code which does exactly the same thing, but does so faster, using less energy, using fewer instructions, or realizing some other benefit. An optimization is unsafe if sometimes the transformed code computes different results than the unoptimized code. In practice, compilers do not attempt unsafe optimizations because programmers have enough problems finding their own bugs. The last thing they need to worry about is a bug being introduced by the compiler.

Item e in Section 2.3.1 states that there is evidence an optimization is unsafe if it is used to prepare a run of the benchmarks and the outputs fail to validate. Presumably a company preparing the benchmark run in this scenario would have their compiler people fix the unsafe optimization so that it is correct (validates) on the SPEC benchmarks, but is still unsafe on other code.
Does that mean peak optimizations are unsafe? Does that mean peak results can be obtained with unsafe, don't-try-this-at-home optimizations?
Why would it be bad if peak results were obtained with unsafe optimizations?
Because in that case peak results would not be useful to many people because they could not rely on their code reaching similar performance levels while computing correct results.
What rules ensure that optimizations used to obtain peak results aren't too unsafe?
Section 1.3.2 stipulates that components used to build the benchmarks (especially compilers) be of production quality, rather than something quickly thrown together to complete one job, or a buggy prototype. It would not be very meaningful if the rules required that components be of production quality but went no further. Otherwise when challenged a tester would respond, "we believe that all of the components used to build the benchmarks are of production quality or better." To provide some objective criteria, the rules require that the components are real products. (A non-real product is one that violates the items in 1.3.2. For example, the customer is not aware of the name of the product, the product will not be available for years, is undocumented and there's no support either.) These criteria are not a guarantee that the optimizations will be safe because there are (or have been) companies that sell lousy products.

Problem 2: The illustration below is our familiar 5-stage MIPS implementation with the destination register mux and an immediate mux shown. Modify it so that it is consistent with the RISC-V RV32I version as described below. The modifications should include datapath and labels, but not control logic. For this problem use RISC-V specification 20191213 available at
https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.
The Inkscape SVG source for the image is at
https://www.ece.lsu.edu/ee4720/2021/hw05-mips-id-mux.svg. It can be edited with your favorite SVG or plain text editor.

In some ways RISC-V is similar to MIPS, but there are differences. Pay attention to the encoding of the store instructions. Also pay attention to how branch and jump targets are computed.

Be sure to change the following:
Bit ranges at the register file inputs.
7 The bit ranges used to extract the immediate.
The bit ranges used for the offsets of branch and jump instructions and the hardware used to compute branch and jump targets.

The inputs to the destination register mux (which connects to the dst pipeline latch).
The names used in the pipeline latches.
Add or remove unneeded pipeline latches. (Such changes will be needed for branches and jumps.)

Consider the following instructions:
Two-register and immediate arithmetic instructions, such as add and addi. The lui instruction (which is similar but not identical to MIPS' lui).
Branch instructions as well as jal and jalr.
The load and store instructions. (Only the store instructions will require a change beyond what is required for arithmetic instructions.)

Note:
- Do not show control logic such as logic driving mux select inputs.
- Do not show the logic to decide whether a branch is taken.

The solution diagram appears on the next page, beneath the original MIPS implementation.
A common mistake was including r31 and some second field in the destination-register mux. RISC-V does not have any assumed destination register, sor31 is not needed. Also, if an instruction does write a register, then the register is always specified in the rd field. (That's why the immediate for store instructions is a special case.)

Another common mistake was using NPC to compute branch and jump targets. In RISC-V branch and jump displacements are added to the PC of the branch or jump.

SVG source at https://www.ece.1su.edu/ee4720/2021/hw05-mips-id-mux.svg.
Solution appears below, beneath the MIPS implementation.


\section*{42 Spring 2020 Solutions}

Problem 1: The illustration below (and on the next page, there's no need to squint) shows our 5 -stage MIPS implementation with some new hardware including: a Y1 unit in EX and a Y2 unit in ME. These are the two stages of a pipelined integer multiplication unit. They are to be used to implement a MIPS32 mul instruction (not to be confused with a MIPS-I mult instruction). The mul instruction executes as you would expect it to, for example mul r1, r2, r3 writes r1 with the product of \(r 2\) and \(r 3\). Because of the need to reduce (add together) all of the partial products, the multiplication hardware spans two stages, in contrast to an integer add which is one in one stage (in the ALU of course). Note: The mult instruction was the subject of 2013 Homework 4.

Here is how mul should execute:
```


# Cycle 0

    add R1, r2, r3 IF ID EX ME WB
    mul r4, R1, r5 IF ID Y1 Y2 WB
    mul R6, r7, R1 IF ID Y1 Y2 WB
    sub r8, R6, r9 IF ID -> EX ME WB
    
# Cycle }\quad0\quad

```

First of all, notice that there is no problem overlapping the two multiplies. Also notice that there is no problem bypassing a value to the source of a multiply.

(a) Add datapath hardware so that the multiply can execute as shown above.
- Assume that the Y1 and Y2 units each have about two multiplexor delays of slack. (Meaning if the path into the inputs of Y1 or out of the output of Y2 passes through more than two multiplexors the clock period would have to be increased, and we don't want that.)
- Pay attention to cost. Assume that the cost of one pipeline latch bit is the same as two multiplexor bits. Make other reasonable cost assumptions.
- Do not lengthen the critical path.
- Make sure that the code fragment above will execute as shown.
- Don't break other instructions.

Solution appears on next page in purple. The inputs to the Y1 unit are provided by the same multiplexors that feed the ALU. This saves the trouble of adding new multiplexors just for the Y1 unit. The output of the Y2 unit is connected to a new ME-stage mux. The added cost is only the new ME-stage mux.

It would be more costly to connect the Y2 output to a new pipeline laten, since the per-bit cost of registers is higher than muxes, and a new mux (or mux input) would be needed anyway in the WB stage.
(b) Add control logic for the existing WB-stage multiplexor and for any new multiplexors you might have added. Hint: This problem is easy, especially if you use two-input muxen.
- Use a pipeline execution diagram (such as the one above) to make sure that the value computed for a multiplexor select signal is the correct value when it is used, perhaps several stages later.

Solution appears in green. For the mul instruction all that's necessary is to send the output of \(=\) mul through the pipeline and use it as the new mux's select signal. Something similar is done for load instructions.
(c) At the lower-right is a big OR gate, its output is labeled STALL. Add an input to that OR gate which will be one when an instruction must stall due to a dependency with a mul. The sub from the execution above suffers such a stall.

Solution appears in turquoise. There is logic to detect a true dependency between the instruction in ID and EX. The stall signal is generated if there is a dependency and if the instruction in EX is a mul.
(Not interesting enough? There is another problem on the next page!) Use this page for the solution or download illustration Inkscape SVG source from https://www.ece.lsu.edu/ee4720/2020/hw02-p1.svg and use that one way or another to prepare a solution.

Solution to both problems appears below.

\(\square\)

Problem 2: Though two stages (Y1 and Y2) may be necessary to compute the product of arbitrary 32-bit signed integers, there are special cases that can be computed in less time, for example when either operand is zero or one.

If the \(Y\) units compute the product then it doesn't matter what operation the ALU is set to, but to handle special case(s) suppose that the control logic set the ALU operation to bitwise AND when decoding a mul instruction. In that case the output of the ALU would be correct for some multiplication operations and so the product would be ready in time to bypass to the next instruction. Add control logic to detect such situations and suppress the stall when present. Don't design the logic to set the ALU operation itself, we'll leave that to the Magic Cloud [tm].

Solution appears in orange.
Since the ALU is computing a bitwise AND we need to detect situations in which a bitwise AND computes the correct product. That is the case if an operand is zero. Logic is added to detect if either operand is zero, and if so a multiply dependency stall is suppressed.

Homework 3 Solution
Due: 30 March 2020

Please E-mail solutions of this assignment to koppel@ece.lsu.edu by the evening of the due date. PDF files are preferred. These can be generated by scanning software that you might have installed with a multifunction printer. A PDF can also be assembled from photos of a hand-completed copy. The disorganized homework penalty will be ignored for the remainder of the semester (unless we return early) so an E-mail with multiple image attachments will be accepted without penalty. Do not physically mail them to my office address, I will not be able to pick them up.

Problem 1: Appearing below is the solution to Homework 2 with labels added to some wires, which is followed by an execution of the code showing values on those labeled wires. The execution is based on the code fragment shown plus nop instructions before the first instruction (addi) and after the last instructions (nop).

\begin{tabular}{llllllll} 
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
addi R2, r0, 0 & IF & ID & EX & ME & WB & & \\
mul R1, R2, r3 & & IF & ID & EX & ME & WB & \\
add r4, R2, R1 & & & IF & ID & EX & ME & WB \\
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & & 0 & 1 & 1 & & & \\
B & & 0 & 1 & 0 & & & \\
C & 0 & 1 & 1 & 1 & & & \\
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
D & & & 0 & 1 & 0 & & \\
E & & 0 & 1 & 1 & 1 & & \\
F & 0 & 1 & 2 & 0 & 0 & & \\
\# Cycle & & 2 & 1 & 4 & 4 & 5 & 6 \\
G & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
H Cycle & 0 & & & & & \\
\# Cycle & & & & &
\end{tabular}
(a) Refer to the table on the previous page for this problem. Notice that the value in the B row (above) in cycle 1 is 0 . According to the problem statement the instruction before addi is a nop.

Why would that value be 0 regardless of what instruction came before addi?
Short Answer: Because the rs register is r 0 , and the output of the \(~=1\) comparison unit is 0 when either input is 0 (even if both inputs are 0 ).

Explanation: The B signal is 1 when the rs register of the instruction in ID is the same as the destination register of the instruction in EX. The rs register is the register specified by bits \(25: 21\) if the instruction, and is usually the first source register of the instruction when written in assembly language. In the code fragment above the rs register for the addi is \(r 0\) and the rs register for the mul and add are r2. (Register r2 is the same as R 2 , upper case is used only for emphasis.) Depending on the instruction the destination register might be in the rd field (most type R instructions), the rt filid (many type I) instructions, or an implicit r31 (the jal instruction). If an instruction does not write any general purpose (integer) register, the destination register 0 is used.

The output of the comparison unit \(\exists^{\prime}\) is 1 when the two inputs (which are 5 -bit quantities) are equal, with one exception: if both inputs are zero the output is 0 . The reason for the exception is that \(\mathbf{r 0}\) is not a real register, meaning that an instruction with r0 as a source (such as addi) does not need to wait for an instruction to write r0 (such as the nop before the addi).

Suppose the addi r2, r0, 0 were changed to addi \(r 2, r 7,0\). Why would the value in the B row still be 0 ?

Because the destination of a nop is ro and \(0 \neq 7\).

(b) Appearing below is a different code fragment. Complete the table so that it shows the values on the labeled wires.

The solution appears below. As a start to understanding the solution examine row G and H , these show the destination register of the instruction in ID and \(M E\), respectively. Row \(D\) is also straightforward, it is 1 when the instruction in EX is a mul.

To solve line \(E\) one needs to determine if the value of at least one of the operands of the instruction in EX is zero. This is certainly false in cycle 3 when sub is in EX because r2 must be non-zero (look at the ori instruction). But, r1 must be zero and so \(B\) is true in cycle 4 . Because the multiplicand of the first multiply is \(r 1\) the product, \(r 3\) must also be zero. Back in cycle 2 , when the ori is in EX, there is no way to tell if \(r 6\) is zero, so the \(B\) value is shown as ?.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline ori r2, r6, 7 & IF & ID & EX & ME & WB & & & \\
\hline sub r1, r2, r2 & & IF & ID & EX & ME & WB & & \\
\hline mul r3, r8, r1 & & & IF & ID & EX & ME & WB & \\
\hline mul r5, r3, r4 & & & & IF & ID & EX & ME & WB \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \#\# SOLUTION \\
\hline A & & 0 & 1 & 1 & 1 & & & \\
\hline B & & 0 & 1 & 0 & 1 & & & \\
\hline C & & 0 & 1 & 1 & 1 & & & \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \\
\hline D & & & 0 & 0 & 1 & 1 & & \\
\hline E & & & ? & 0 & 1 & 1 & & \\
\hline F & & 0 & 0 & 0 & 0 & & & \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \\
\hline G & & 2 & 1 & 3 & 5 & & & \\
\hline H & & & & 2 & 1 & 3 & 5 & \\
\hline \# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \\
\hline
\end{tabular}
(c) Appearing below are completed tables, but without a code fragment. Show a code fragment that could have produced those table values.

The originally assigned problem contained an error which made it difficult to solve. Shown below is the originally table, followed by the intended table. The solution uses the intended table.


Solution appears below. Lower case characters are used for register numbers and instructions which are one of several possible correct answers. For example, the first instruction orI, could have been any type I instruction that wrote a register, such as addI and xorI. The destination of orI must be r2 (and so it is written as R2) but the first source could be any register.
```


# SOLUTION

# Cycle

orI R2, r2, 7 IF ID EX
MUL R3, R2, r2 IF ID EX ME WB
addI R4, R3, 9 IF ID -> EX ME WB
sub R8, r10, R4

```

Problem 2: Appearing below and on the next page is the solution to Homework 2 Problem 1. In this problem add hardware to handle a different and less special multiplication special case. Suppose that the middle output of the Y1 stage of the multiplier held the correct product whenever the high 24 bits of its b input are zero. For example, when b is 1,5 , or 255 . Call such values small. In all cases the correct product appears at the output of Y2.

Note: All outputs of Y1 arrive with zero slack, even the center output with the small b special case. That means that nothing can be done with these values until the next clock cycle, at least without reducing the clock frequency.

(a) Add hardware to bypass the product to the ALU and to the rtv mux when b is small. (There is a larger diagram on the next page.) The bypass should allow the first code fragment below to execute without a stall.
(b) Add control logic to suppress the stall when it is possible to bypass.

In the first code fragment below the stall is avoided because the b value (which is the rtv ) is small, in the second it is too large.
```


# Cycle }\quad0\quad

addi r1, r0, 23 IF ID EX ME WB
mul r2, r3, r1 IF ID EX ME WB
sub r4, r2, r5 IF ID EX ME WB

# Cycle }\quad0\quad

addi r1, r0, 300 IF ID EX ME WB
mul r2, r3, r1 IF ID EX ME WB
sub r4, r2, r5 IF ID -> EX ME WB

```
- Make sure that the changes don't break existing instructions.
- As always avoid costly solutions.
- As always pay attention to critical path.

The SVG source for the illustration below is at https://www.ece.lsu.edu/ee4720/2020/hw03-p2.svg. It can be edited using Inkscape or any other SVG editor, or (not recommended) a text editor.

Solution appears below with part (a), the datapath, in green and part (b), the control logic, in purple.
A path is provided from the middle Y1 output to the ME-to-EX bypass paths and uses a new multiplexor which selects between ME. ALU and ME. Ym. Pipeline latch register ME. mx is used as the select signal for this mux. Note that the mux does not affect the path from ME. ALU to the Mem Port Addr input. That's important because we always assume that the Mem Port Addr input and D Out are on the critical path and so anything that increases the length (time) of the path will slow the clock frequency.

For part (b) we need to suppress the stall if the b input to the multiplier is \(<256\). For unsigned values that is true if bits 8 to 31 are zero, which is easily checked by an NOR gate. The logic that is shown checks whether the value is not small \((\geq 256)\), and uses that as an additional condition for the stall. (So the multiply-dependence stall will not be asserted if the value is small.)

Alternative Solution: Rather than adding a new multiplexor, the ME. Ym signal could have been connected to the three EX-stage muxen. This would have cost more and made the control logie more complicated, but it possibly would be faster, depending on how the five-input multiplexors were synthesized. Such solutions received full credit, even without the control logic for the EX-stage multiplexors.


It's up to all of us: \(\quad r>2 \mathrm{~m} \Rightarrow R_{e}<1 \quad\) where \(r\) is the radius of the largest circle with you at the center and containing only people in your household, and \(R_{e}\) is the effective reproduction number, the number of people infected by an infected person.

Problem 1: Appearing below is the code fragment from Homework 3.
\begin{tabular}{llllllll} 
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
addi R2, r0, 0 & IF & ID & EX & ME & WB & & \\
mul R1, R2, r3 & & IF & ID & EX & ME & WB & \\
add r4, R2, R1 & & & IF & ID & EX & ME & WB \\
\# Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6
\end{tabular}
(a) Does this code fragment look like it was compiled with optimization on?

If your answer is something like "yes, it could be part of optimized code" then explain why you think it so and provide any missing context. (Do not change or re-arrange the three instructions above.)

If your answer is something like "no, it does not appear optimized" then show what the code would look like after optimization. Hint: A correct answer can start with either "Yes it does" or "No it doesn't". The "No" answer is straightforward.

No, because with constant propagation and folding none of the instructions are necessary. We know that \(r 2\) will be assigned 0 , and then \(r 1\) and \(r 4\) will also be zero. Any uses of those registers in the same basic block can be replaced by zero. (See previous answer.)

Problem 2: MIPS does not appear to have a muli instruction.
(a) Comment on the following:

MIPS has a mul instruction but does not have a muli instruction because, as the solution to Homework 2 shows, the additional hardware for muli (beyond that used for mul) would be too costly.

Is the statement above reasonable or unreasonable? Explain.
Unreasonable, because the connection to the lower input of the Y1 unit is from the lower ALU mux, which has a connection to the immediate. Therefore, the implementation of the muli would only require changes to control logic.
(b) Show the encoding of MIPS instruction mul r1, r2, r3. Show all 32 bits of the instruction, divided into fields (each field can be shown in the radix of your choice). (The MIPS ISA manuals are linked to the course Web page. Instruction encodings are in Volume II.)

A quick lookup in the ISA manual reveals that the opcode is \(1 c_{16}\) and the Func field value is 2 . As with most type-R instructions the order of the assembly language arguments are \(r d, r s, r t\). The sa field is-must be-zero.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline & & & r & \multicolumn{2}{|r|}{sa} & \multicolumn{3}{|c|}{Func} \\
\hline MIPS R: & 0x1c & 2 & 3 & 1 & & 0 & & 0x2 \\
\hline & 26 & & 16 & & 10 & & 4 & 0 \\
\hline
\end{tabular}
(c) Some possible reasons that there is no muli instruction in MIPS is that either there are no Format-I opcodes available (they are all used by other instructions) or that the few remaining opcodes are being kept in reserve for a better instruction than a muli.

Based on the MIPS Architecture Manuals (they are linked to the course references page) how many opcodes are available for new Format-I instructions? The easy way to solve this is to find the right table. The hard way to solve this is to go through the 144 or so pages of instruction descriptions. Hint: Look in volume I.

The following solution is based on the MIPS manuals linked to the course Web page, which are Revision 0.95 and deseribe MIPS before Release 6.

Table A-2 shows the encoding of the Opcode field. Twelve entries in the table appear blank. Assuming they are supposed to be blank, and are not a problem with either the PDF encoding of the manual or of the PDF viewer I am using, then there are 12 opcode slots available.

Assuming there is a PDF problem of some kind then the number of free opcodes are the number of entries in Table A-2 with the symbol shown in the first row of Table A-1, the row for "Operations . . . reserved for future use" instructions.

Problem 3: Perhaps you saw this coming: Time to add muli to MIPS.
(a) Show how a Format-R muli instruction with a ten-bit immediate might be defined using unused fields in the Format-R encoding. Make up your own function field value, but try to pick one that's unused. (See the previous problem.) Show how muli r1, r2, 43 might be encoded for your muli definition.

Table A-5 of the Revision 0.95 shows function field values used by the family of instructions that includes mul. The mul instruction is in the first row of the table and the second row appears empty (see the gripe about the PDF manual from previous problem's solution). Lets reserve the second row for immediate-value versions of first-row instructions. So the function field value for mul is \(000010_{2}\). So lets make the function field for muli \(001010_{2}\). The opcode will remain \(1 c_{16}\). To encode the 10 -bit immediate use bits 20:16 (the rt field) for the upper 5 bits and use bits \(10: 6\) (the sa field) for the lower 5 bits. So to encode \(43_{10}=0000101011_{2}\) set the \(r\) field to \(00001_{2}\) and the sa field to \(01011_{2}=b_{16}\).

Opcode rs rt rd sa Func
MIPS R:
\begin{tabular}{rr|r|r|r|r|r|}
\hline & \(0 \times 1 \mathrm{C}\) & 2 & 1 & 1 & \(0 \times \mathrm{l}\) & \(0 \times 2\) \\
\hline 31 & 2625 & 21 & 20 & 16 & 15 & 11 \\
\hline
\end{tabular}
(b) Modify the hardware below (there's a copy on the next page) to implement this new instruction. The modified hardware should provide the immediate needed by muli. Show datapath but not control logic. Of course, any changes should not break existing instructions.

Pay attention to cost and performance. This can easily be solved by adding a mux in the ID stage. Hint: The solution is not much more than a mux. Be sure to carefully label the inputs.


The SVG source for the diagram below is available at https://www.ece.lsu.edu/ee4720/2020/hw03-p2.svg.

Solution appears below in green. Since no other instruction concatenates the rt and sa fields to extract an immediate, that hardware had to be added. The lower input to the new ID-stage mux consists of those concatenated fields and also includes 6 zero bits on the most-significant side. The resulting constant is 16 bits, the same size as ordinary immediate. The format immed block sign-extends the value to 32 bits, which will have no effect for the muli immediate but is still needed for the other immediates.


It's up to all of us: \(\quad r>2 \mathrm{~m} \Rightarrow R_{e}<1 \quad\) where \(r\) is the radius of the largest circle with you at the center and containing only people in your household, and \(R_{e}\) is the effective reproduction number, the number of people infected by an infectious person.

Problem 1: Solve 2019 Final Exam Problem 2, which asks for a pipeline execution diagram of FP code on our FP MIPS implementation, but with the comparison functional unit and floating-point condition code register added. For more information on the implementation of the floating-point compare instructions see 2018 Final Exam Problem 3. Please don't get confused about which problem to solve and which to use for background!

See the posted final exam solution.
Problem 2: The following question appeared as Spring 2010 Homework 3 Problem 3, but in this ten-year anniversary version the solution must contain control logic for the multiplexors at the inputs to the A1 and A2 units. Try to initially solve it without looking at the solution, but use the solution if you get stuck.

Replace the fully pipelined adder in our FP pipeline (which appears on the next page) with one with an initiation interval of two and an operation latency of four. (The existing FP adder has an initiation interval of one and an operation latency of four.) See 2010 Homework 3 Problem 3 for more details.

Show datapath and control logic. Be sure to show control logic for the multiplexors at the inputs to A1 and A2, this control logic does not appear in the solution to the 2010 assignment. Hint: This additional control logic is really easy to do, it can be done just with wires, no gates!

Solution appears below. Note to Spring 2020 students: The control logic for the WF-stage mux is different, pernaps simpler, than in the FP pipelines used elsewhere in class. The ad pipeline latch signal is 1 in stages with an instruction using the FP add functional unit. The connections to the ad pipeline latch are equivalent to connections to the LSB (bit 0:0) of the xw pipeline latch. Note to future students: the WF-stage mux logic here might be used in the default class FP pipeline in future semester.

The changes in green allow data to pass through the A1 and A2 units twice. The control logic, in purple provides the select signal for the new multiplexors at the A1 and A2 inputs.

The stall condition for an add.s in ID must now check for an add. s in the first pass through A1, those two conditions are checked by the purple OR gate.


An Inkscape SVG version of the MIPS implementation below can be found at https://www.ece.1su.edu/ee4720/2020/mpipei_fp_by.svg.


It's up to all of us: \(\quad r>2 \mathrm{~m} \Rightarrow R_{e}<1 \quad\) where \(r\) is the radius of the largest circle with you at the center and containing only people in your household, and \(R_{e}\) is the effective reproduction number, the number of people infected by an infectious person.

Problem 1: Solve 2017 Final Exam Problem 2, which asks for a PED of some code fragments on a 2-way superscalar MIPS implementation. The solution is available, but make every effort to solve it on your own. Use the posted solution only if you get stuck. Solving the 2017 problem will make the problem below easier.

Problem 2: Solve 2019 Final Exam Problem 1, including the bonus question (part d), which asks for datapath and control logic for a 2 -way superscalar implementation, some associated with a dependence leading to a sw instruction. Parts a and b ask for typical hardware. Part c is more interesting because the hardware is essentially avoiding a stall by skipping an instruction in a dependence chain. The dependence chain is or \(\rightarrow\) add \(\rightarrow\) sw and the skipped instruction is the or. Part d, the bonus question, asks whether this is worth it.
An Inkscape SVG version of the MIPS implementation from Problem 1 of the exam can be found at https://www.ece.lsu.edu/ee4720/2019/fe-fuse.svg.

See posted exam solution at https://www.ece.lsu.edu/ee4720/2019/fe_sol.pdf

\section*{43 Spring 2019 Solutions}

\section*{非 LSU EE 4720 Fall 2019 Homework 1}

非
非

\section*{非 SOLUTION}
```

    ## Assignment https://www.ece.lsu.edu/ee4720/2019/hw01.pdf
    ```

非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非排 Problem 1
```

            .text
    ```
get_index:
            非 Register Usage
            非
非 CALL VALUES:
\# \$a0: Address word to lookup. Word will be at least 3 chars.
\# \$a1: Address of start of word table.
\# \$a2: Address of end of word table.
非 \$a3: Address of storage for hash table.
非
非 RETURN:
\$v0: If found, word position (first is 1, second is 2, etc.);
        if not found, 0;
\$v1: If not in hash table, hash index;
        if in hash table, \(0 \times 100\) + hash index.
Note:
非 Can modify \$t0-\$t9, \$a0-\$a3
\# [r] Code should be correct.
非 [ \(\quad\) ] Code should be reasonably efficient.
非 [ \(\checkmark\) ] Do not use pseudoinstructions except for nop and la.
非

\section*{SOLUTION}
addi \＄t6，\＄ra， 0 非 Save return address．
addi \＄t5，\＄a1， 0 非 Save start address of word table．
非 Compute hash．
lb \＄t0，0（\＄a0）
lb \＄t1，1（\＄a0）
sra \＄t2，\＄t1， 2
xor \＄t0，\＄t0，\＄t2
sll \＄t2，\＄t1， 6
xor \＄t0，\＄t0，\＄t2
lb \＄t1，2（\＄a0）
sra \＄t2，\＄t1， 4
xor \＄t0，\＄t0，\＄t2
sll \＄t2，\＄t1， 4
xor \＄t0，\＄t0，\＄t2
```

andi \$v1, \$t0, 0xff
sll \$t0, \$v1, 2
add \$t3, \$a3, \$t0
lhu $v0, 2($t3)
beq \$v0, \$0, NOT_IN_HASH
lhu $t2, 0($t3)
jal streq
add \$a1, \$t5, \$t2
add \$ra, \$t6, \$0
beq \$v0, \$0, NOT_IN_HASH
addi \$a1, \$t5, 0
Ihu $v0, 2($t3)
jr \$ra
ori \$v1, \$v1, 0x100

```
NOT_IN_HASH:
addi \$t2, \$0, 1
LOOP:
    jal streq
    sub \$t4, \$a1, \$t5
    beq \$v0, \$0, MISMATCH
    nop
    非 Match
    sh \$t4, 0(\$t3)
    sh \$t2, 2(\$t3)
    jr \$t6
    addi \$v0, \$t2, 0
MISMATCH:
    非 Scan to next null.
    lbu \$t1, 0(\$a1)
    bne \$t1, \$0 MISMATCH
    addi \$a1, \$a1, 1
    slt \$t1, \$a1, \$a2
    bne \$t1, \$0, LOOP
    addi \$t2, \$t2, 1

DONE:
jr \$t6
addi \$v0, \$0, 0

Spring 2019
```

streq:
\#⿰⿰三丨⿰丨三⿰⿰三丨⿰丨三⿻\mp@code{* Register Usage}
非
\# CALL VALUES:
\#\# \$a0: Address of string 1.
\# $a1: Address of string 2.
    非
    # RETURN:
    #⿰⿰三丨⿰丨三一\mp@code{$v0: If strings match, 1; otherwise, 0;}
\$a1: At null or first mismatched character.
add \$t8, \$0, \$a0
SE_LOOP:
lbu $t7, 0($t8)
lbu $t9, 0($a1)
bne \$t7, \$t9, SE_MISMATCH
addi \$t8, \$t8, 1
bne \$t7, \$0, SE_LOOP
addi \$a1, \$a1, 1
jr \$ra
addi \$v0, \$0, 1
SE_MISMATCH:
jr \$ra
addi \$v0, \$0, 0

```
非
非
.data
word_list_start:
    .asciiz "aardvark"
    .asciiz "ark"
    .asciiz "bark"
    .asciiz "barkeeper"
    .asciiz "persevere"
    .asciiz "bird"
    .asciiz "box"
    .asciiz "sox"
    .asciiz "lox"
    .asciiz "soy"
    .asciiz "sax"
    .asciiz "brain"
word_list_end:
test＿words＿start：
．asciiz＂ark＂
．asciiz＂box＂
．asciiz＂sox＂
．asciiz＂soy＂
．asciiz＂sax＂
．asciiz＂sod＂
．asciiz＂barkeeper＂
．asciiz＂bark＂
．asciiz＂woof＂
．asciiz＂bar＂
．asciiz＂ark＂
．asciiz＂bar＂
．asciiz＂bark＂
．asciiz＂barkeeper＂
．asciiz＂arkansas＂
．asciiz＂lox＂
．asciiz＂box＂
．asciiz＂sox＂
．asciiz＂sax＂
．asciiz＂soy＂
．asciiz＂aardvark＂
．asciiz＂persevere＂
test＿words＿end：
．byte 0000 ．align 4
results＿start：
．word \(20 \times 4 b\) 非 ark
．word 7 0x3e 非 box
．word 8 0x2f 非 sox
．word 10 0x3f 非 soy
．word 11 0xac 非 sax
．word 0 0xee 非 sod
．word 4 0x1d 非 barkeeper
．word 3 0x1d 非 bark
．word 0 0x5a 非 woof
．word 0 0x1d 非 bar
．word 2 0x14b 非 ark
．word 0 0x1d 非 bar
．word 3 0x11d 非 bark
．word 4 0x1d 非 barkeeper
．word 0 0x4b 非 arkansas
．word 9 0x30 非 lox
．word 7 0x13e 非 box
．word 8 0x12f 非 sox
．word \(110 \times 1 \mathrm{ac}\) 非 sax
．word 10 0x13f 非 soy
．word 1 0x1e 非 aardvark
．word 5 0xe 非 persevere
hash＿table：
．space 1024
msg：
．asciiz＂Pos：\％／s1／3d \％／t5／1c Hash 0x\％／v1／3x \％／t3／1c\％／t4／1c Word：\％／s0／s n ＂
msg＿at＿end：
．asciiz＂Done with tests：Errors：\％／s3／d pos，\％／s5／d hash found，\％／s4／d hash idx\n＂
．text
．globl＿＿start
＿＿start：
la \＄s0，test＿words＿start
la \＄s2，test＿words＿end
addi \＄s3，\＄0， 0 非 Word position error count．
addi \＄s4，\＄0， 0 非 Hash index error count．
addi \＄s5，\＄0， 0 非 Hash found error count．
la \＄s6，results＿start

TB＿WORD＿LOOP：
addi \＄a0，\＄s0， 0
la \＄a1，word＿list＿start
la \＄a2，word＿list＿end
la \＄a3，hash＿table
jal get＿index
addi \＄v0，\＄0，－2
lw \＄t0， \(0(\$ s 6)\) 非 Word Position
beq \＄t0，\＄v0，TB＿DONE＿POS＿CHECK
addi \＄t5，\＄0， 95 非＇＿＇
addi \＄t5，\＄0， 88 非＇X＇
addi \＄s3，\＄s3， 1
TB＿DONE＿POS＿CHECK：
lw \＄t0，4（\＄s6）非 Hash
andi \＄t1，\＄t0，0x100
andi \＄t2，\＄v1， \(0 \times 100\)
beq \＄t1，\＄t2，TB＿DONE＿HASH＿FOUND＿CHECK
addi \(\$ \mathrm{t} 3, \$ 0,95\) 非＇＿＇
addi \＄t3，\＄0， 88 非＇X＇
addi \＄s5，\＄s5， 1
TB＿DONE＿HASH＿FOUND＿CHECK：
andi \＄t1，\＄t0，0xff
andi \＄t2，\＄v1，0xff
beq \＄t1，\＄t2，TB＿DONE＿HASH＿IDX＿CHECK
addi \＄t4，\＄0， 95 非＇＿＇
addi \＄t4，\＄0， 88 非＇X＇
addi \＄s4，\＄s4， 1
TB＿DONE＿HASH＿IDX＿CHECK：
addi \＄s6，\＄s6， 8
addi \＄s1，\＄v0， 0
la \＄a0，msg
addi \＄v0，\＄0， 11
syscall

TB＿CHAR＿LOOP：
```

lbu \$t0, 0(\$s0)
bne \$t0, \$0, TB_CHAR_LOOP
addi \$s0, \$s0, 1
slt \$t0, \$s0, \$s2
bne \$t0, \$0, TB_WORD_LOOP
nop

```
la \$a0, msg_at_end
addi \$v0, \$0, 11
syscall
addi \$v0, \$0, 10
syscall
nop

Before solving the branch hardware problem below it might be helpful to look at 2016 Homework 2.
Problem 1: The code below should suffer a stall on the illustrated implementation due to a dependency between the addi and bne instructions. The stall can be avoided by scheduling the loop, but lets consider a hardware solution for code fragments like this in which an addi \(r X, r Y\), IMM is followed by a bne \(r X, r 0\), \(T\) or by a beq \(r X, r 0, T\).
```

LOOP:
addi r3, r3, -1
bne r3, r0, LOOP
lw r1, 4(r1)

```

One way to avoid the stall (which would work for more than just the cases outlined above) would be to have the ALU generate an =0 signal which, if the dependencies were right, could be used by the branch hardware. Alas, the ALU people are on vacation, so lets try something else.

As alert students may have realized by now, all the branch hardware has to do is check whether rY == -IMM, which is r3 == 1 in the example. The comparison itself can be done using the existing comparison logic. The challenge is delivering the operands to that logic at the right time.


Attention students who have forgotten how to use a pencil (or never learned): An Inkscape SVG version of the implementation can be found at https://www.ece.lsu.edu/ee4720/2019/mpipei3.svg.

(a) Add hardware to the implementation above to deliver the correct operands to the comparison unit so code fragments like the one above can execute without a stall.
- Pay attention to cost, including the number of bits in each wire used. (For example, don't add a second comparison unit.)
- The changes should not prevent other code from executing correctly. (For example, a branch such as beq \(\mathrm{r} 1, \mathrm{r} 2\), T should execute correctly.)
- Don't overlook that rX and rY are not necessarily the same register.

Solution appears above in blue. Note that the rs value from the addi is obtained from the register file using a 5 -bit mux to replace the branch's rs with that of the addi. A more costly alternative would be to bypass the entire 32 -bit value EX.rsv. Also note that the immediate value from the EX stage is being used, because that's where the addi will be.
(b) Add control logic to generate a BY signal which is set to logic 1 when the branch can use the bypass. The control logic must detect that the correct instructions (including the registers) are present.

Solution appears above in green. The logic checks that there is a bne in ID, that the branch's rt register is zero, that the branch's rs register matches the addi destination, and that there is an addi in the EX stage.
(c) If the design above was done correctly the highest cost part is the logic handling the immediate. Show how the cost of that logic can be reduced while still retaining most (but not all) of the benefits of the full-cost design. Your argument should include examples of "typical" code. (Assume [actually assert] that your code samples are typical [reflects what is running by users most of the time]. Later in the semester we'll remove the scare-quotes from "typical".)

Solution appears above in purple. It is reasonable to expect that-I assert that!!-many loops will have an addi/bne instruction sequence like the one above in which the immediate value is small, perhaps just -1 . Suppose that the immediate fits in \(w\) bits and is
negative. Then the negation logic need handle only \(w\) bits and the mux going into the comparison unit would only need to provide the \(w\) least significant bits. The remaining \(32-w\) bits would come from the \(r t\) value, which must be zero. The control logic needs to check whether the immediate fits in \(w\) bits, that is done by the purple AND gate which examines bits \(15: w\) of the immediate. (There is no need to examine bits \(31: 0\) since they are all zeros or all ones.)

The least expensive option would be to design the hardware to work only with an immediate value of -1 . In that case the negation logic would no longer be necessary since we know that the result can only be 1.

Problem 1: The three loops below (probably on the next page) copy an area of memory starting at the address in \(r 2\) to an area of memory starting at the address in \(r 3\). The number of bytes to copy is in \(r 5\).

(a) Show a pipeline execution diagram for each loop on the illustrated implementation. Solution appears below.
```


# Loop A -- SOLUTION Part a

add r4, r3, r5 IF ID EX ME WB
LOOP: \# Cycle 0
lb r1, O(r2) IF ID EX ME WB
sb r1, O(r3) IF ID -> EX ME WB
addi r2, r2, 1 IF -> ID EX ME WB
addi r3, r3, 1 IF ID EX ME WB
bne r3, r4, LOOP IF ID ----> EX ME WB
nop IF ----> ID EX ME WB
LOOP: \# Cycle 0
lb r1, O(r2) IF ID EX ME WB

```
\# Loop B -- SOLUTION Part a
add r4, r3, r5 IF ID EX ME WB
LOOP: \# Cycle \(0 \begin{array}{lllllllllllllllll} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & \text {-- } & 1 \text { st Iter }\end{array}\)
    lw r1, O(r2) IF ID EX ME WB
    sw r1, \(0(r 3) \quad\) IF ID \(\rightarrow\) EX ME WB
    addi \(r 2\), \(r 2,4\) IF 4 ID EX ME WB
    addi r3, r3, 4 IF ID EX ME WB
    bne r3, r4, LOOP IF ID ----> EX ME WB
    nop IF ----> ID EX ME WB
LOOP: \# Cycle \(0 \begin{array}{llllllllllllllll} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & -- \\ \text { 2nd Iter }\end{array}\)
    1w r1, 0(r2)
\# Loop C -- SOLUTION Part a
add r4, r3, r5 IF ID EX ME WB
addi r4, r4, -8 IF ID EX ME WB
LOOP: \# Cycle \(0 \begin{array}{llllllllllllllll} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & \text {-- } & \text { 1st Iter }\end{array}\)
    lw r1, \(0(r 2) \quad\) IF ID EX ME WB
    lw r10, 4(r2) IF ID EX ME WB
    sw r1, 0 (r3) IF ID EX ME WB
    sw r10, 4(r3) IF ID EX ME WB
    addi \(r 2\), r2, 8 IF ID EX ME WB
    bne r3, r4, LOOP IF ID EX ME WB
    addi r3, r3, 8 IF ID EX ME WB
LOOP: \# Cycle \(\left.0 \begin{array}{llllllllllllllll} & 0 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & -- & 2 n d\end{array}\right)\)
    lw r1, O(r2) IF ID EX ME WB
(b) Compute the rate that each loop copies data in units of bytes per cycle. Base this on your execution diagrams.

Based on the time that the first instruction of the loop, \(\mathbf{l b}\), is in IF one iteration of Loop A takes \(10-1=9\) cycles. Each iteration copies one byte, so Loop A copies at a rate of \(\frac{1}{9} \approx 0.111\) bytes per cycle.

An iteration of Loop B also takes 9 cycles, but it copies four bytes and so Loop B copies at a rate of \(\frac{4}{9} \approx 0.444\) bytes per cycle, four times faster!

An iteration of Loop C takes \(9-2=7\) cycles and copies 8 bytes, so its rate is \(\frac{8}{7} \approx 1.143\) bytes per cycle, more than twice as fast as Loop B!
(c) Loop A has a wasted delay slot and should suffer stalls. Schedule the code (re-arrange instructions) to fill the delay slot and minimize the number of stalls. Feel free to change instructions and to add new ones, though the loop should still copy one byte per iteration and should copy the data as described above.

The solution appears below. The delay slot was filled by the store instruction, which uses a negative offset because r3 is incremented before it executes. The increment of \(r 3\) is put a the beginning so that its value will be ready when the bne is in ID.
```


# Loop A -- SOLUTION Part c

add r4, r3, r5 IF ID EX ME WB
LOOP: \# Cycle 0
addi r3, r3, 1 IF ID EX ME WB
lb r1, O(r2) IF ID EX ME WB
addi r2, r2, 1 IF ID EX ME WB
bne r3, r4, LOOP IF ID EX ME WB
sb r1, -1(r3) IF ID EX ME WB

# Cycle }0

```
(d) Loop A can be safely substituted for Loop C. That is, if a program calls Loop C then that call can be changed to a call of Loop A or B and the program will still work correctly. However, if a program calls Loop A, substituting B will not work. Explain why and show sample values for \(r 2\), \(r 3\), and \(r 5\) for which this is true.

The number of bytes copied by Loop B is a multiple of 4 and the starting address too must be a multiple of 4. Loop A has no such restrictions. Values that will work for A but not B are \(\mathrm{r} 2=0 \times 1001, \mathrm{r} 3=0 \times 2002, \mathrm{r} 5=3\).
(e) If a program calls Loop B substituting C will not work. Explain why and show sample values for r2, r3, and \(r 5\) for which this is true.

The number of values copied by Loop C must be a multiple of 8 . Sample register values that work for B but not C are \(\mathrm{r} 2=0 \times 1000, \mathrm{r} 3=0 \times 2000\), and \(\mathrm{r} 5=4\).

Homework 5 solution

Problem 1: Solve Midterm Exam Problem 1b, in which code is to be completed using the \(\mathrm{J}_{\mathrm{r}}\) instruction.
see the nidterm exam solution.
Problem 2: The next page shows the original solution to Midterm Exam Problem 3 (the one appearing in the exam before 20 April 2019). In this solution the Taken signal is set for a bne using the rsv and rtv values (from the register file) even if there is a dependence with an slt in the EX stage. Modify the logic to fix this.

For your solving convenience, the original solution illustration appears on the next page and in Inkscape SVG at https://www.ece.lsu.edu/ee4720/2019/mt-p3-slt-bne-sol.svg

See the nidterm exam solution. The solution to 3b now includes a solution to this problem (shown in orange).


Problem 1: The following problem is an enhanced version of 2018 Final Exam Problem 1 (c). Appearing below is our 2-way superscalar MIPS with ID-stage hardware to determine branch direction (near the top in blue) and ID-stage hardware to squash instructions (near the bottom in blue). The Inkscape SVG source for this image can be found at https://www.ece.lsu.edu/ee4720/2019/hw08-ss.svg.

There are two outputs of the branch direction hardware logic, indicating whether the respective ID-stage slot has a taken branch. For example, if Taken0 is 1 then there is a branch in slot 0 and that branch is taken. Of course, assume that this logic is correct.

There is a squash logic with two inputs at the bottom. If input Sq 0 is 1 then the instruction in ID-stage slot 0 will be squashed, likewise for Sq1.

In this implementation fetch groups are not aligned.

(a) When a branch is taken we may need to squash one or two instructions (the number of instructions to squash depends on the whether the branch is in slot 0 or slot 1). Design logic to set the Sq0 and Sq1 inputs so that appropriate instructions are squashed. It will be very helpful to draw pipeline execution diagrams showing a taken branch in slot 0 and slot 1 .
\(\checkmark\) Draw PEDs for the two cases.
Solution appears below. If the branch is in Slot 0 (Case 0) then both IF-stage instructions are squashed, if the branch is in Slot 1 (Case 1) then only the slot-1 instruction in IF is squashed.
Add hardware to set SQ signals.
The solution appears above. Notice that the squash signals are put in the pipeline latch so they move with the IF-stage instruction.
```


# 

# Cycle

    sw r7, 4(r15)
    addi r15, r15, 4
    beq r1, r2, TARG
    add r3, r4, r5
    slti r16, r17, 8
    or r18, r19, r20
xor r10, r11, r12
TARG:
sub r7, r8, r9 IF ID EX ME WB
lw r14, O(r15) IF ID EX ME WB

# Cycle

# 

# Cycle

    sw r7, 4(r15)
    addi r15, r15, 4
    beq r1, r2, TARG
    add r3, r4, r5
    slti r16, r17, 8
    SOLUTION - Case 1: Branch in slot 1.
0
SOLUTION - Case 0: Branch in slot 0.
0
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IFx
IFx
0
xor r10, r11, r12
TARG:
sub r7, r8, r9 IF ID EX ME WB
lw r14, O(r15)
IF ID EX ME WB

# Cycle

```
\(\begin{array}{llllllll}0 & 1 & 2 & 3 & 4 & 5 & 6 & 7\end{array}\)
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB IF ID EX ME WB IFx
xor r10, r11, r12
TARG:
\begin{tabular}{llllll} 
sub r7, r8, r9 & & & IF ID EX ME WB \\
lw r14, & \(0(r 15)\) & & & IF ID EX ME WB \\
\# Cycle & 0 & 1 & 2 & 3 & 4 \\
\hline
\end{tabular}

SOLUTION - Case 0: Branch in slot 0
\(\begin{array}{llllllll}0 & 1 & 2 & 3 & 4 & 5 & 6 & 7\end{array}\)
EX ME

IF ID EX ME WB
IF ID EX ME WB
IFx
IFx
(b) Notice that the branch hardware shown can only provide the target for a branch in slot 1. Add hardware for providing the branch target of a branch in slot 0 . Note that unlike the final exam, in this problem fetches are not aligned. That precludes the more efficient solution given in the final exam.

Do not add hardware for checking the branch condition. Show logic computing the select signals for any multiplexors you add, but do not show any other control logic. Note: In the original assignment the direction to show logic computing select signals was omitted.

Add hardware for a slot-0 branch.

\section*{Pay attention to cost.}

Be sure the hardware computes the correct target address. Think about the value of NPC (or related value) that's needed.

Solution appears below. The slot-1 immediate is selected by the upper-left green mux if there is a branch in slot 1 , otherwise the slot-0 immediate is selected (and is ignored if neither slot holds a branch). If the branch is in slot 1 then ID. npc is the branch PC plus 4 , which is just what we need to calculate \(\operatorname{PC}+4+i m m * 4\). (The actual computation is effectively \(4 *((P C+4) / 4+i m m\) ).) If the branch is in slot 0 then ID. npc is the branch PC plus 8 . In order to compute the correct target 1 is subtracted from the immediate. That is, we compute \(4 *((\mathrm{PC}+8) / 4+\) imm -1\()\).

Note that by subtracting 1 from the immediate we only need a 16 -bit subtractor. Had we nailvey subtracted 4 from ID. npc we would need a 30 -bit (or even a 32 -bit) subtractor.

In the final exam fetch groups were aligned and so a lower-cost solution was possible. See the final exam solution.


\section*{44 Spring 2018 Solutions}
```

\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#

## 

## LSU EE 4720 Fall 2018 Homework 1 -- SOLUTION

## 

## 

# Assignment http://www.ece.lsu.edu/ee4720/2018/hw01.pdf

```

\section*{\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#}
``` \#\# Problem 1 -- unsqw Routine
```

```
        .text
unsqw:
## Register Usage
#
# CALL VALUES
$a0: Address of start of compressed string.
$a1: Address of area to decompress into.
# RETURN:
    [\checkmark] Write memory starting at al with decompressed string.
# Note:
# Can modify $t0-$t9, $a0-$a3
[ }\checkmark\mathrm{ ] Code should be correct
[\checkmark] Code should be reasonably efficient.
# [\checkmark] Do not use pseudoinstructions except for nop and la.
Both Methods
A reference starts with a character of value 0x80 or higher.
This byte is called the reference marker.
Simple Method:
    0x80 B1 B2
= 1000_0000 B1 B2 :
    0\times80- is the reference marker.
        B1 is length
        B2 is distance
    Better Method:
        1000_0000 B1 B2:
        0x80 = 1000_0000 is the reference marker.
            B1 is length
            B2 is distance.
        10LL LLLL B1:
        10LL_LLLL is the reference marker
            LL_LLLL (low 6 bits) are length,
            B1 is distance
        1100 0000 B1 B2 B3:
        0xC0-= 1100_0000 is the reference marker.
            B1 holds Tength.
            B2 holds bits 15:8 of distance and
            B3 holds bits 7:0 of distance.
        11LL LLLL B1 B2:
        11LL_LLLL is the reference marker
            LL_LLLL (low 6 bits) are length
            B1 holds bits 15:8 of distance and
            B2 holds bits 7:0 of distance
```

    \#\# SOLUTION
        j LOOP
    LITERAL_CHAR

| sb \$t0, 0(\$a1) | \# Write literal character to output buffer. |
| :--- | :--- |
| beq \$t0, \$0, DONE |  |
| addi \$a1, \$a1, 1 | \# If 0, we're at the end of the string. |
|  |  |
| lb \$t0, 0(\$a0) \# Load next character of compressed text. <br> bgez \$t0, LITERAL_CHAR \# If it's non-negative, it's an ordinary char. <br> addi \$a0, \$a0, 1 \# Either way, increment address. |  |

## REFERENCE:

\#\# Reference Marker Found
\# At this point in the code:
\# Reference Marker is in \$t0.

```
andi $t3, $t0, 0x3f # Extract length from marker. (Bits 6-0.)
bne $t3, $0, SKIP LB # If this length is non-zero, get distance.
lbu $t4, 0($a0) # Load next byte. Could be length or dist.
```

\#\# Case 1 and 3 -- Length is in byte immediately following marker.
\# At this point in code:
Reference marker is in \$t0
\# Length is in register \$t4

```
addi $t3, $t4, 0 # Move distance into register t3.
lbu $t4, l($a0) # Load next byte, the first distance byte.
addi $a0, $a0, 1
```

SKIP_LB:
\#\# Check whether distance is in one or two bytes (case 3 \& 4).

```
At this point in code
    Reference marker is in $t0.
    Length is in register $t3.
    First distance byte is in register $t4.
```

andi $\$$ t1, $\$ t 0,0 \times 40$ \# Check whether distance is stored in two bytes.
beq \$t1, \$0, SKIP_2B_DISTANCE \# If taken, \$t4 is the distance.
nop
\#\# Cases 3 and 4 -- Length is stored in two bytes.
\#\#
\#
\# At this point in code:
Reference marker is in \$t0.
Length is in register \$t3.
Most (more) significant distance byte is in register $\$ \mathrm{t} 4$.
lbu \$t5, 1(\$a0) \# Load second distance byte.
sll \$t4, \$t4, 8 \# Combine two distance bytes by shifting LSB ..
or \$t4, \$t4, \$t5 \# ... and oring them together.
addi \$a0, \$a0, 1

SKIP_2B_DISTANCE:
\#\# All Cases
\#
\# Copy the prior occurrence of text from some part of the
output buffer to the end of the output buffer.
\# At this point in code:
Reference marker is in $\$$ t0.
Length is in register \$t3.
Distance is in register \$t4.
sub \$t4, \$a1, \$t4 \# Compute starting address of prior occurrence.
add \$t5, \$t4, \$t3 \# Compute ending address of prior occurrence.
addi \$a0, \$a0, 1
COPY_LOOP:
lbu \$t0, 0(\$t4) \# Load character of prior occurrence ..
sb \$t0, 0(\$a1) \# .. and write it to the end of the output buffer.
addi \$t4, \$t4, 1
bne \$t4, \$t5, COPY LOOP
addi \$a1, \$a1, 1
j LOOP
nop

DONE:
jr \$ra
nop

## \#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\# \#\# Testbench Routine <br> \#

## .data

uncomp: \# Uncompressed data.

.byte $0 \times 80 \quad 6174$ \# " cold "
.ascii "temperatures. T" \# Idx: 208-222
\# Idx: 209 = 223 - 14 = 0xdf - 0xe. Len: 11 = 0xb.
.byte $0 \times 80 \quad 1114$ \# "emperatures"
\# Idx: $180=234-54=0 \times e a-0 \times 36$. Len: $5=0 \times 5$. .byte $0 \times 80 \quad 554$ \# " will"
.ascii "\nplunge" \# Idx: 239-245
\# Idx: $160=246-86=0 x f 6-0 x 56$. Len: $10=0 x a$.
.byte $0 \times 80$ 10 86 \# " into the " \# Idx: 256-261
ascii "teens "
\# Idx: $177=262-85=0 \times 106-0 \times 55$. Len: $4=\begin{gathered}\text { \# Idx: } \\ 0 \times 4 \text {. }\end{gathered}$
.byte 0x80 485 \# "and "
.ascii "20s tonight" \# Idx: 266-276
\# Idx: $261=277-16=0 \times 115-0 \times 10$. Len: $5=0 \times 5$. .byte $0 \times 80 \quad 5 \quad 16 \quad \#$ " and "
.ascii "could be quite similar\nWednesday " \# Idx: 282-314
\# Idx: $272=315-43=0 \times 13 b-0 \times 2 b$. Len: $5=0 \times 5$. .byte 0x80 543 \# "night"
ascii ".\n" \# Idx: 320 - 321
\# Idx: $67=322-255=0 \times 142-0 x f f$. Len: $44=0 \times 2 c$.
.byte $0 \times 8044255$ \# "===============================================1
\# Idx: $365=366-1=0 \times 16 e-0 x 1$. Len: $26=0 \times 1 a$.
.byte $0 \times 80261$ \# "=========================="
ascii "\n* TEMPERATURE...Low" \# Idx: 392-411
\# Idx: $233=412-179=0 \times 19 c-0 \times b 3$. Len: $6=0 \times 6$. .byte 0×80 6179 \# "s will"
.ascii " fall" \# Idx: 418-422
\# Idx: 246 = 423 - 177 = 0x1a7 - 0xb1. Len: 10 = 0xa. .byte $0 \times 8010177$ \# " into the "
ascii "mi" \# Idx: 433 - 434
\# Idx: $206=435-229=0 \times 1 b 3-0 x e 5$. Len: $4=0 \times 4$.
.byte 0x80 4229 \# "d te"
\# Idx: 258 = 439 - 181 = 0x1b7 - $0 \times b 5$. Len: $4=0 \times 4$. .byte $0 \times 80 \quad 4181$ \# "ens "
.ascii "to lower" \# Idx: 443-450
\# Idx: $265=451-186=0 \times 1 \mathrm{c} 3-0 \times b a$. Len: $4=0 \times 4$. .byte $0 \times 804186$ \# " 20s"
ascii "\nalong" \# Idx: 455 - 460
\# Idx: $277=461-184=0 \times 1 \mathrm{~cd}-0 \times b 8$. Len: $5=0 \times 5$. .byte $0 \times 80 \quad 5184$ \# " and " .ascii "north of" \# Idx: 466-473
\# Idx: $428=474-46=0 \times 1 d a-0 \times 2 e . ~ L e n: 5=0 \times 5$. .byte $0 \times 80 \quad 546$ \# " the "
.ascii "I-10/12 corridor. Sou" \# Idx: 479 - 499
\# Idx: $469=500-31=0 \times 1 f 4-0 \times 1 f$. Len: $6=0 \times 6$. .byte $0 \times 80 \quad 6 \quad 31$ \# "th of "
\# Idx: 479 = 506 - 27 = 0x1fa - $0 \times 1 \mathrm{~b}$. Len: $4=0 \times 4$. .byte $0 \times 80427$ \# "I-10"
\# Idx: $445=510-65=0 \times 1 \mathrm{fe}-0 \times 41$. Len: $4=0 \times 4$. .byte $0 \times 80 \quad 465$ \# " low"
.ascii "s\n" \# Idx: 514-515
\# Idx: 414 = 516 - $102=0 \times 204$ - $0 \times 66$. Len: $5=0 \times 5$.
.byte 0x80 5102 \# "will
ascii "range from 20" \# Idx: 521 - 533
\# Idx: $442=534-92=0 \times 216-0 \times 5 \mathrm{c}$. Len: $4=0 \times 4$. .byte $0 \times 80 \quad 4 \quad 92$ \# " to " .ascii "25. These temperature" \# Idx: 538-558
\# Idx: $412=559-147=0 \times 22 f-0 \times 93$. Len: $7=0 \times 7$.
.byte $0 \times 80 \quad 7147$ \# "s will "
.ascii "be similar\nWednesday night. $\backslash n \backslash n *$ DURATION." \# Idx: 566 - 605 .ascii "..Freezing condition" \# Idx: 606-625
\# Idx: $559=626-67=0 \times 272$ - $0 \times 43$. Len: $7=0 \times 7$. .byte $0 \times 80767$ \# "s will "
.ascii "likely last for 12" \# Idx: 633 - 650
\# Idx: $534=651-117=0 \times 28 b-0 \times 75$. Len: $5=0 \times 5$.
.byte 0x80 5117 \# " to 2"
.ascii "6\nhours over muc" \# Idx: 656-671
\# Idx: $470=672-202=0 \times 2 a 0-0 x c a$. Len: $9=0 \times 9$.
.byte $0 \times 80 \quad 202$ \# "h of the " $\quad$ \# Idx: 681-694
.ascii "warned area to"

.byte $0 \times 805108$ \# "night"
\# Idx: 461 = $700-239=0 \times 2 b c-0 x e f . ~ L e n: ~ 5=0 \times 5$. .byte $0 \times 80 \quad 5239$ \# " and "
.ascii "then" \# Idx: 705-708
\# Idx: $648=709-61=0 \times 2 c 5-0 x 3 d$. Len: $7=0 \times 7$. .byte $0 \times 80 \quad 761$ \# " 12 to " \# Idx: 716-717
\# Idx: $657=718-61=0 \times 2 \mathrm{ce}-0 \times 3 \mathrm{~d}$. Len: $7=0 \times 7$. .byte $0 \times 80 \quad 761$ \# "\nhours "
\# Idx: $577=725-148=0 \times 2 \mathrm{~d} 5-0 \times 94$. Len: $16=0 \times 10$.
.byte 0x80 16148 \# "Wednesday night."
.byte 0
\# Original: 741 B, Simple Compressed: 508 B, Ratio: 0.686
comp_better: \# Compressed data Better Method.

\# Idx: $28=202-174=0 \times c a-0 \times a e$. Len: $6=0 \times 6$.
.byte $0 \times 86$
.ascii "temperatures. T"
\# Idx: $209=223-14=0 x d f$ - 0xe. Len: $11=0 x b$.
.byte $0 \times 8 \mathrm{~b}$
Idx: $180=234-54=$ \# "emperatures"
Idea $0 \times 0 \times 36 . ~ L e n: ~$
2
\# Idx: $180=234-54=0 \times e a-0 \times 36$. Len: $5=0 \times 5$.

.ascii "\nplunge" \# Idx: 239-245
\# Idx: $160=246-86=0 \times f 6-0 \times 56$. Len: $10=0 \times a$.
.byte $0 \times 8 \mathrm{a}$ 0×56 \# " into the "
.ascii "t" \# Idx: 256-256
\# Idx: $15=257-242=0 \times 101-0 \times f 2$. Len: $3=0 \times 3$.
byte $0 \times 83$ 0xf2 \# "een"
\# Idx: $143=260-117=0 \times 104-0 \times 75$. Len: $3=0 \times 3$.
.byte $0 \times 83$
Idx: $178=263-85=0 \times 107$ \# " $^{2} 0 \times 55$. Len: $3=0 \times 3$.
\# Idx: $178=263-85=0 \times 107-0 \times 55$. Len: $3=0 \times 3$.
.byte $0 \times 83$.
.ascii 20 s tonight" \# Idx: 266 - 276
\# Idx: $261=277-16=0 \times 115-0 \times 10$. Len: $5=0 \times 5$.
$\begin{array}{llll} & & \\ \text {.byte } 0 \times 85 & 0 \times 10 \text { \# " and " } & \\ \text { \# Idx: } 282-284\end{array}$
\# Idx: $205=285-80=0 \times 11 d-0 \times 50$. Len: $3=\begin{aligned} & \text { \# Idx } \\ & 0 \times 3 \text {. }\end{aligned}$
.byte $0 \times 83$ 0×50 \# "ld " 285 . Len: $3=0 \times 3$
.ascii "be quite similar\nWednesday " \# Idx: 288-314
\# Idx: $272=315-43=0 \times 13 \mathrm{~b}-0 \times 2 \mathrm{~b}$. Len: $5=0 \times 5$.
.byte $0 \times 850 \times 2 \mathrm{~b}$ \# "night"
.ascii "." \# Idx: 320-320
\# Idx: $40=321-281=0 \times 141-0 \times 119$. Len: $72=0 \times 48$.
byte $0 \times c 00 \times 480 \times 010 \times 19$ \# " n n============================
$\begin{aligned} \text {.ascii } " * \text { TEMPERATURE...Low" } & \text { \# Idx: } 393-411\end{aligned}$
\# Idx: $233=412-179=0 \times 19 \mathrm{c}-0 \times b 3$. Len: $6=0 \times 6$.
.byte 0x86 0xb3 \# "s will"
ascii " fa" \# Idx: 418-420
\# Idx: $416=421-5=0 \times 1 a 5-0 \times 5$. Len: $3=0 \times 3$.
.byte $0 \times 83$
\# Idx: $247=424-177=0 \times 1$ \#88- $0 \times b 1$. Len: $9=0 \times 9$.
.byte $0 \times 89$ 0xb1 \# "into the " \# Idx: 433 - 434
435 \# Idx: 433 - 434
\# Idx: $206=435-229=0 \times 1 b 3-0 \times e 5$. Len: $4=0 \times 4$.
.byte 0x84 0xe5 \# "d te"
\# Idx: $258=439-181=0 \times 1 b 7-0 \times b 5$. Len: $4=0 \times 4$.
.byte $0 \times 84 \quad 0 \times b 5$ \# "ens "
$\begin{aligned} \text { \# Idx: } 426=443-\quad 17=0 \times 1 \mathrm{bb}-0 \times 11 . ~ L e n: ~ \\ \text {. byte } 0 \times 83\end{aligned} \quad=0 \times 3$.
.byte 0x83 $0 \times 11$ \# "to " \# Idx: 446-450
\# Idx: $265=451-186=0 \times 1 c 3-0 x b a$. Len: $4=0 \times 4$.
.byte 0x84 0xba \# " 20s"
.ascii "\nalong" \# Idx: 455 - 460
\# Idx: $277=461-184=0 \times 1 \mathrm{~cd}-0 \times b 8$. Len: $5=0 \times 5$.

.ascii "north of" \# Idx: 466-473
\# Idx: $428=474-46=0 \times 1 d a-0 \times 2 e$. Len: $5=0 \times 5$.
.byte $0 \times 850 \times 2 \mathrm{e}$ \# " the " $\quad$ Idx: 479
.ascii "T-10/12" \# Idx: 479 - 485

ascii "rridor. Sou" \# \# Idx: 489-499
\# Idx: $469=500-31=0 \times 1 f 4-0 \times 1 f$. Len: $6=0 \times 6$.
byte $0 \times 86$ 0x1f \# "th of "
\# Idx: $479=506-27=0 \times 1 \mathrm{fa}-0 \times 1 \mathrm{~b}$. Len: $4=0 \times 4$.
.byte $0 \times 840506-0 \times 1 \mathrm{~b}$ \# "I-10"
\# Idx: $445=510-65=0 \times 1 \mathrm{fe}-0 \times 41$. Len: $4=0 \times 4$.
.byte $0 \times 84$ 0x41 \# " low" \# Idx
.ascii "s\n" \# Idx: 514 - 515
\# Idx: $414=516-102=0 \times 204-0 \times 66$. Len: $5=\begin{gathered}\text { \# Idx: }\end{gathered}$

.ascii "ra" \# Idx: 521 - 522
\# Idx: $243=523-280=0 \times 20 b-0 \times 118$. Len: $4=0 \times 4$.
byte 0xc4 $0 \times 010 \times 18$ \# "nge "
.ascii "from" \# Idx: 527-530
\# Idx: $451=531-80=0 \times 213-0 \times 50$. Len: $3=0 \times 3$.
.byte $0 \times 830 \times 50$ \# " 20" $0 \times 5$. Len. $3=0 \times 3$.
\# Idx: $442=534-92=0 \times 216-0 \times 5 \mathrm{c}$. Len: $4=0 \times 4$.
.byte $0 \times 84$
. ascii " 25 . Thes" $0 \times 5 \mathrm{c} \#$ " to "
.ascii "25. Thes" \# Idx: 538-545
\# Idx: $254=546-292=0 \times 222-0 \times 124$. Len: $4=0 \times 4$.
.byte 0xc4 $0 \times 010 \times 24$ \# "e te"
$\begin{array}{rl}\text { \# Idx: } & 224= \\ \text {.byte } 0 \times \mathrm{cf} & 550-326=0 \times 226-0 \times 146 . ~ L e n: ~ \\ 0 \times 010 \times 46 \text { \# "mperatures will" } & =0 \times f .\end{array}$
$\begin{array}{rl}\text {. byte } 0 \times \mathrm{cf} & 0 \times 010 \times 46 \text { \# "mperatures will" } \\ \text { \# Idx: } 287 & =565-278=0 \times 235-0 \times 116 \text {. Len: } \quad 4=0 \times 4 \text {. }\end{array}$
.byte $0 \times c 4 \quad 0 \times 010 \times 16$ \# " be "
\# Idx: $297=569-272=0 \times 239-0 \times 110$. Len: $25=0 \times 19$.
.byte $0 \times \mathrm{d} 9 \quad 0 \times 010 \times 10$ \# "similar\nWednesday night. \n"
\# Idx: $392=594-202=0 \times 252-0 \times c a . L e n: 3=0 \times 3$.
$\begin{aligned} & \text { \# Idx: } 392=594-202=0 \times 252 \\ & \text {.byte } 0 \times 83\end{aligned} \quad 0 \times \mathrm{ca}$ \# "\n* "
.ascii "DU" \# Idx: 597 - 598
\# Idx: $400=599-199=0 \times 257-0 \times c 7$. Len: $3=0 \times 3$.
.byte $0 \times 83$ 0xc7 \# "RAT"
.ascii "ION" \# Idx: 602 - 604
\# Idx: $406=605-199=0 \times 25 d-0 \times c 7$. Len: $3=0 \times 3$.
byte $0 \times 83$ 0xc7 \# "..." $0 \times \mathrm{c}$. Len: 3 = $0 \times 3$.
ascii "Freez" \# Idx: 608 - 612
\# Idx: $157=613-456=0 \times 265-0 \times 1 \mathrm{c} 8$. Len: $4=0 \times 4$.
.byte $0 \times \mathrm{c} 4 \mathrm{0x} 010 \times \mathrm{c} 8$ \# "ing "
.ascii "conditio" \# Idx: 617-624
\# Idx: $440=625-185=0 \times 271-0 \times b 9$. Len: $3=0 \times 3$.
byte 0x83 $\quad 625$ - 0xb9 \# "ns "
\# Idx: $561=628-67=0 \times 274-0 \times 43$. Len: $5=0 \times 5$.
.byte $0 \times 85$ 0×43 \# "will "
.ascii "likely last for " \# Idx: 633-648
\# Idx: $484=649-165=0 \times 289-0 x a 5$. Len: $3=0 \times 3$.
byte 0x83 0xa5 \# "12 "
\# Idx: $535=652-117=0 \times 28 \mathrm{c}-0 \times 75$. Len: $4=0 \times 4$. .byte $0 \times 84$ 0x75 \# "to 2" .ascii "6\nhours ov" \# Idx: 656-665
\# Idx: $449=666-217=0 \times 29 a-0 x d 9$. Len: $3=0 \times 3$.
.byte $0 \times 83$
ascii "muc" $0 x d 9$ \# "er " \# Idx: 669 - 671
\# ascii "muc" $672-202=0 \times 2 a 0-0 \times c a$. Len: $9=\begin{gathered}\text { \# Idx: } \\ 0 \times 9 \text {. }\end{gathered}$
.byte $0 \times 890 \times c a$ \# "h of the "
.ascii "warned area" \# Idx: 681 - 691
\# Idx: $269=692-423=0 \times 2 b 4-0 \times 1 a 7$. Len: $13=0 x d$.
.byte 0xcd 0x01 0xa7 \# " tonight and
\# Idx: $677=705-28=0 \times 2 \mathrm{c} 1-0 \times 1 \mathrm{c}$. Len: $3=0 \times 3$.
.byte 0x83 0x1c \# "the"
.ascii "n" \# Idx: 708-708
\# Idx: $648=709-61=0 \times 2 \mathrm{c} 5-0 \times 3 \mathrm{~d}$. Len: $7=0 \times 7$. $\begin{array}{ll}\text {.byte } 0 \times 87 \\ \text {.ascii "18" } 0 \times 3 \mathrm{~d} & \text { \# " } 12 \text { to " } \quad \text { Idx: } 716 \text { - } 717\end{array}$
\# Idx: $657=718-61=0 \times 2 c e-0 x 3 d . \quad$ Len: $7=0 \times 7$.
.byte $0 \times 87$ 0x3d \# "\nhours "
\# Idx: $577=725-148=0 \times 2 d 5-0 \times 94$. Len: $16=0 \times 10$. .byte 0x90 0x94 \# "Wednesday night."
.byte 0
\# Original: 741 B, Better Compressed: 428 B, Ratio: 0.578
msg:
$\qquad$
msg_err:
.asciiz "Input \%/a2/s: Error at idx \%/t4/d. Text below is from idx \%/t6/d - \%/t8/d. \n Found: \"\%/t5/s $\backslash$ " $\mathrm{nCorrect:} \mathrm{\ "} \mathrm{\% /t7/s} \mathrm{\backslash "} \mathrm{\backslash n} \mathrm{\backslash n"}$
msg_correct:
.asciiz "Input \%/a2/s: Decompressed text is all correct.\n\n"
dname_simple:
.asciiz "Simple"
dname_better:
.asciiz "Better"
data_out:
.space 512
.text
.globl __start
__start:
la \$a0, msg
la \$a2, uncomp
addi \$v0, \$0, 11
syscall
la \$a0, comp_simple
la \$a1, data_out
jal unsqw
nop
la \$a2, dname_simple
jal TB_CHECK
nop
la \$a0, comp_better
la \$a1, data-out
addi \$t0, \$a1, 512
TB_LOOP_CLEAR:
sb \$0, 0 (\$a1)
bne \$a1, \$t0, TB_LOOP_CLEAR
addi \$a1, \$a1, 1
la \$a1, data_out
jal unsqw
nop
la \$a2, dname_better
jal TB_CHECK
nop
addi \$v0, \$0, 10
syscall
nop

TB_CHECK:
la \$t0, uncomp
la \$t1, data_out
TB_COMPARE:
lbu \$t2, 0(\$t0)
lbu \$t3, 0(\$t1)
bne \$t2, \$t3, TB_ERROR

```
addi $t0, $t0, 1
bne $t2, $0, TB COMPARE
addi $t1, $t1, \overline{1}
j TB_CORRECT
nop
```


## TB_ERROR:

la \$a0, msg_err
addi $\$ \vee 0, \$ \overline{0}, 11$
la \$t5, data_out
sub \$t4, \$t1, \$t5
addi \$t7, \$0, 10 \# Amount of context sub \$t6, \$t4, \$t7
bgez \$t6, TB_POS
nop
addi \$t6, \$0, 0
TB_POS:
add \$t5, \$t5, \$t6 la \$t7, uncomp add \$t7, \$t7, \$t6 addi \$t8, \$t4, 1 sb \$0, $1(\$ t 0)$
sb \$0, $2(\$+1)$
syscall
nop
j TB DONE
nop
TB CORRECT:
la \$a0, msg correct
addi \$v0, $\$ \overline{0}, 11$
syscall
TB DONE:
jr \$ra nop

The solution to several of the problems in this assignment requires material about to be covered in class, in particular, stalling instructions to avoid hazards. For coverage of this material see slide set six, http://www.ece.lsu.edu/ee4720/2018/lsli06.pdf. For a solved problem see 2014 Homework 1 Problem 3. Feel free to look through old homework and exams for other similar problems, but when doing so make sure that the MIPS implementation matches the one in this problem: the muxen at the ALU inputs should each have just 2 inputs.

Problem 1: Recall that in the unsqw program from Homework 1 there was a loop that had to copy the prior occurrence of a piece of text to the output buffer. That loop from the solution appears below, and again re-written to improve performance, at least that was the goal.

```
# Original Code ------------------------------------------------------------------------
    # Copy the prior occurrence of text from some part of the
    # output buffer to the end of the output buffer.
    # At this point in code:
    # Reference marker is in $t0.
    # Length is in register $t3.
    # Distance is in register $t4.
    #
    sub $t4, $a1, $t4 # Compute starting address of prior occurrence.
    add $t5, $t4, $t3 # Compute ending address of prior occurrence.
    addi $a0, $a0, 1
COPY_LOOP:
    lb $t0, O($t4) # Load character of prior occurrence ..
    sb $t0, O($a1) # .. and write it to the end of the output buffer.
    addi $t4, $t4, 1
    bne $t4, $t5, COPY_LOOP
    addi $a1, $a1, 1
    j LOOP
    nop
```

```
# Improved Code ------------------------------------------------------------------------
    # Copy the prior occurrence of text to the end of the output buffer.
    # At this point in code:
    # Reference marker is in $t0.
    # Length is in register $t3.
    # Distance is in register $t4.
    #
    sub $t4, $a1, $t4 # Compute starting address of prior occurrence.
    addi $a0, $a0, 1
    # Round length (L) up to a multiple of 4.
    #
    addi $t7, $t3, 3
    andi $t8, $t7, Oxfffc # Note: this only works if L < 65536
    sub $t6, $t8, $t3
    #
    # At this point:
    # $t8: L', rounded-up length.
    # $t6: Amount added to L to round it up. That is, L' - L
    # t6 can be 0, 1, 2, or 3.
    # If t6 is 0, then L' = L;
    # if t6 is 1, then L' = L + 1; etc.
    # Decrement prior-occurrence and output-buffer pointers.
    #
    sub $t4, $t4, $t6
    sub $a1, $a1, $t6
    # Jump to one of the four lb instructions in the copy loop.
    #
    la $t7, COPY_LOOPd4 # Get address of first lb instruction.
    sll $t6, $t6, 3 # Compute offset to lb that we want to start at.
    add $t7, $t7, $t6 # Compute address of starting lb ..
    jr $t7 # .. and jump to it.
    add $t5, $t4, $t8 # Don't forget to compute stop address.
```


## COPY LOOPd4:

```
lb $t0, 0($t4)
```

lb $t0, 0($t4)
sb $t0, 0($a1)
sb $t0, 0($a1)
lb $t0, 1($t4)
lb $t0, 1($t4)
sb $t0, 1($a1)
sb $t0, 1($a1)
lb $t0, 2($t4)
lb $t0, 2($t4)
sb $t0, 2($a1)
sb $t0, 2($a1)
lb $t0, 3($t4)
lb $t0, 3($t4)
sb $t0, 3($a1)
sb $t0, 3($a1)
addi \$t4, \$t4, 4
addi \$t4, \$t4, 4
bne \$t4, \$t5, COPY_LOOPd4
bne \$t4, \$t5, COPY_LOOPd4
addi \$a1, \$a1, 4
addi \$a1, \$a1, 4
j LOOP
j LOOP
nop

```
nop
```

Let $L$ denote the length of the prior occurrence of text to copy
(a) Determine the number of instructions executed by the original code in terms of $L$. Include the copy loop and the instructions before it shown above. State any assumptions.

The number of instructions executed is: $3+5 L+2=5+5 L$. This includes the nop since the time to execute the program
includes the time to handle nops.
(b) Determine the number of instructions executed by the improved code in terms of $L$. Include the copy loop and the instructions before it shown above. State any assumptions.

Short Answer: The number of instructions executed is: $13+2 L+3\lceil L / 4\rceil+2=15+2 L+3\lceil L / 4\rceil \approx 15+2.75 L$.
Explanation: The 15 term covers instructions outside of the loop. Each full iteration of the loop executes four $\mathrm{lb} / \mathrm{sb}$ instruction pairs however the loop can be entered at any one of the four lb instructions, with the entry point chosen so that exactly $L$ characters are copied. The $2 L$ term covers the $\mathrm{lb} / \mathrm{sb}$ pairs. The loop also includes two addi instructions and a braneh. These are executed once per iteration and there are $\lceil L / 4\rceil$ iterations. These three instructions are covered by the $3\lceil L / 4\rceil$ term.
(c) What is the minimum value of $L$ for the improved method to actually be faster?

An approximate solution can be found by solving $5+5 L=15+2.75 L$ for $L$, yielding $L=\frac{40}{9} \approx 4 . \overline{44}$. Because $L$ must be an integer and because we ignored the ceiling function we need to investigate values of $L$ around 4. For $L=6$ the improved method is 2 instructions faster, for all $0<L<6$ the original method takes fewer cycles. So the minimum value is $L=6$.
(d) What is it about the improved code that helps performance?

Short Answer: The two addis and the bne are executed just once for each four characters copied.
Long Answer: In both the original and improved code one lb and one sb are exeeuted for each character copied. That is captured in the $2 L$ term in the expression for the improved code and part of the $5 L$ term in the expression for the original code. So for these the two codes are comparable. But in the improved code the two addi instructions and the bne are executed once for every four characters copied while in the original code they are executed four times as often: one for each character copied. That is what is responsible for the improved performance.

Problem 2: Note: The following problem is identical to 2016 Homework 1 Problem 1. Try to solve it without looking at the solution. Answer each MIPS code question below. Try to answer these by hand (without running code).
(a) Show the values assigned to registers t1 through t8 (the lines with the tail comment Val:) in the code below. Refer to the MIPS review notes and MIPS documentation for details.

Solution appears below (to the right of SOLUTION, of course).

```
.data
.byte 0x10, 0x11, 0x12, 0x13
.byte 0x14, 0x15, 0x16, 0x17
.byte 0x18, 0x19, 0x1a, 0x1b
.byte 0x1c, 0x1d, 0x1e, 0x1f
```

myarray:

```
.text
la $s0, myarray # Load $s0 with the address of the first value above.
    # Show value retrieved by each load below.
lbu $t1, 0($s0) # Val: SOLUTION: 0x10
lbu $t2, 1($s0) # Val: SOLUTION: 0x11
lbu $t2, 5($s0) # Val: SOLUTION: 0x15
lhu $t3, 0($s0) # Val: SOLUTION: 0x1011
lhu $t4, 2($s0) # Val: SOLUTION: 0x1213
addi $s1, $0, 3
add $s3, $s0, $s1
lbu $t5, 0($s3) # Val: SOLUTION: 0x13
sll $s4, $s1, 1 SOLUTION: Note: s4 <= 3<<1 = 6
add $s3, $s0, $s4
lhu $t6, 0($s3) # Val: SOLUTION: 0x1617
sll $s4, $s1, 2 SOLUTION: Note: s4 <= 3<<2 = 12
add $s3, $s0, $s4
lhu $t7, 0($s3) # Val: SOLUTION: 0x1c1d
lw $t8, 0($s3) # Val: SOLUTION: 0x1c1d1e1f
```

(b) The last two instructions in the code above load from the same address. Given the context, one of those instructions looks wrong. Identify the instruction and explain why it looks wrong. (Both instructions should execute correctly, but one looks like it's not what the programmer intended.)

Register s0 holds an address that the programmer decided to call myarray, so lets think of the data starting at that address as an array. Normally, to access element i of an array that starts at address a, you load data at address a $+i * s$, where $s$ is the size of an array element. In the code fragment above, register s0 holds the starting address ( $\mathbf{a}$ in the example). From the way the code is written it looks like register s1 is holding the element index ( $i$ in the example). Because the sll in the last group of four instructions is effectively multiplying s1 by 4 , it looks like the load should be of the s1'th element of an array of elements of size 4. That's consistent with the 1 w , which loads a 4 -byte element, and the last 1 hu looks out of place. The 1 hu that loads t 6 looks fine, because its address was computed from a value of s1 multiplied by 2 .
(c) Explain why the following answer to the question above is wrong for the MIPS 32 code above: "The 1w instruction should be a lwu to be consistent with the others."

There is no lwu, because when loading a 32 -bit quantity into a 32 -bit register there is no need to distinguish between a signed and unsigned quantity. In contrast, the 1 hu and 1 h load a 16 -bit quantity into a 32 -bit register, the 1 hu sets the high 16 bits to zero, it zero-pads, while lh sets the high 16 bits to the value of the MSB of the loaded value, it sign extends.

Problem 3: Note: The following problem was assigned in each of the last three years, and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
LOOP: # Cycles 0
    lw r2, O(r4) IF ID EX ME WB
    add r1, r2, r7 IF ID EX ME WB
LOOP: # Cycles 0
```

The add depends on the 1 w through $r 2$, and for the illustrated implementation the add has to stall in ID until the 1 w reaches WB.

```
LOOP: # Cycles 0
    lw r2, 0(r4)
    add r1, r2, r7 IF ID ----> EX ME WB
LOOP: # Cycles 
```

(b) Explain error and show correct execution.

```
LOOP: # Cycles }0
    add r1, r2, r3 IF ID EX ME WB
    lw r1, O(r4) IF ID -> EX ME WB
LOOP: # Cycles 
```

There is no need for a stall because the 1 w writes $r 1$, it does not read $r 1$.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    lw r1, O(r4) IF ID EX ME WB
LOOP: # Cycles 
```

(c) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3 IF ID EX ME WB
    sw r1, O(r4) IF ID -> EX ME WB
LOOP: # Cycles 0
```

A longer stall is needed here because the sw reads r1 and it must wait until add reaches WB.

```
LOOP: # Cycles 0
    add r1, r2, r3
    sw r1, O(r4)
    IF ID EX ME WB
        IF ID ----> EX ME WB
LOOP: # Cycles 0
```

(d) Explain error and show correct execution.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    xor r4, r1, r5 IF ----> ID EX ME WB
    LOOP: # Cycles 0}0
```

The stall above allows the xor, when it is in ID, to get the value of $r 1$ written by the add; that part is correct. But, the stall starts in cycle 1 before the xor reaches ID, so how could the control logic know that the xor needed $r 1$, or for that matter that it was an xor? The solution is to start the stall in cycle 2 , when the xor is in ID.

```
LOOP: # Cycles 0 1 < 2 3 3 4 5 5 6 7 7 SOLUTION
    add r1, r2, r3 IF ID EX ME WB
    xor r4, r1, r5 IF ID ----> EX ME WB
    LOOP: # Cycles 0
```

Problem 4: The MIPS code below is taken from the solution to 2018 Homework 1. Show the execution of this MIPS code on the illustrated implementation for two iterations. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be the value being written. (In class we called such a register file internally bypassed.)

- Check carefully for dependencies.
- Focus on when the branch target is fetched and on when wrong-path instructions are squashed.
- Be sure to stall when necessary.


The solution appears below. Each iteration includes two stalls due to dependencies, and two squashes due to the taken branch. The first stall occurs in cycles 3 and 4 and is due to the dependence between the $l \mathrm{bu}$ and sb carried by to . The second stall occurs in cycles 7 and 8 and is due to the dependence between the first addi and the bne carried by t4. Because branches are resolved in ME (look for the path from EX/ME. ALU back to PC) two wrong-path instructions will be fetched before the control logic can determine that they are indeed wrong-path instructions. During the cycle at which the branch is resolved the two wrong path instructions are squashed, this occurs below in cycle 10. The wrong-path instructions in this example are called X 1 and X 2 , these are whatever instructions appear in memory after the second addi.

```
CLOOP: # Cycle 0
    lbu $t0, O($t4) IF ID EX ME WB
    sb $t0, O($a1) IF ID ----> EX ME WB
    addi $t4, $t4, 1 IF ----> ID EX ME WB
    bne $t4, $t5, CLOOP IF ID ----> EX ME WB
    addi $a1, $a1, 1 IF ----> ID EX ME WB
    X1 IF IDx
    X2 IFx
CLOOP: # Cycle 0
    lbu $t0, 0($t4) IF ID EX ME WB
    sb $t0, 0($a1)
    IF ID ----> EX ME WB
    addi $t4, $t4, 1
    bne $t4, $t5, CLOOP
    addi $a1, $a1, 1
    X1
        IF ----> ID EX ME WB
            IF ID ----> EX ME WB
            IF ----> ID EX ME WB
        X2
            IF IDx
                                    IFx
CLOOP: # Cycle 0 1 2 2 3 4 4 5 6 % 7 8 9 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    lbu $t0, O($t4) IF ID
```

Problem 1: Appearing below are two MIPS implementations, The First Implementation is taken from Homework 2 Problem 4. Branches suffer a two-cycle penalty on this implementation since they resolve in ME. On the The Second Implementation branches resolve in EX reducing the penalty to one cycle. For convenience for those using 2 -sided printers the same implementations are shown again on the next page.

The First (HW2 P4) Implementation


The Second Implementation



The code fragment below and its execution on The First Implementation is taken from the solution to Homework 2 Problem 4. Notice that the branch suffers a two-cycle branch penalty.


```
    lbu $t0, O($t4) IF ID EX ME WB
    sb $t0, 0($a1) IF ID ----> EX ME WB
    addi $t4, $t4, 1 IF ----> ID EX ME WB
    bne $t4, $t5, CLOOP IF ID ----> EX ME WB
    addi $a1, $a1, 1 IF ----> ID EX ME WB
    X1
    X2 IFx
CLOOP: # Cycle 0 1 2 2 3 4 4 5 5 6 7 7 8 9 9 10
    lbu $t0, 0($t4) IF ID EX ME WB
    sb $t0, O($a1) IF ID ----> EX ME WB
    addi $t4, $t4, 1 IF ----> ID EX ME WB
    bne $t4, $t5, CLOOP IF ID ----> EX ME WB
    addi $a1, $a1, 1 IF ----> ID EX ME WB
    X1
    X2
                                    IFx
CLOOP: # Cycle 0
    lbu $t0, 0($t4) IF ID
CLOOP: # Cycle 0 1 1 2 3 3 4 4 5 % 6
    lbu $t0, 0($t4) IF ID EX ME WB
    sb $t0, 0($a1) IF ID ----> EX ME WB
    addi $t4, $t4, 1 IF ----> ID EX ME WB
    bne $t4, $t5, CLOOP IF ID ----> EX ME WB
    addi $a1, $a1, 1 IF ----> ID EX ME WB
    X1
    X2
CLOOP: # Cycle 0 1 2 2 3 4 4 5 6 6 7 % 8 9 0
    lbu $t0, 0($t4) IF ID EX ME WB
    sb $t0, O($a1) IF ID ----> EX ME WB
    addi $t4, $t4, 1
    bne $t4, $t5, CLOOP
    addi $a1, $a1, 1
    X1
    X2
CLOOP: # Cycle 0 1 2 2 3 4 4 5 5 6 7 7 8 9 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    lbu $t0, 0($t4) IF ID EX ME
```

(a) On The Second Implementation the branch penalty would only be one cycle. But, as we discussed in class, moving branch resolution from ME to EX might impact the critical path. Let $\phi_{1}=$ 1 GHz denote the clock frequency on The First Implementation and call the clock frequency on The Second Implementation $\phi_{2}$. For what value of $\phi_{2}$ would the performance of the two implementations be the same when executing the code above for a large number of iterations?

Show your work.
Short answer: $\phi_{2}=\phi_{1} \frac{10}{11}=909.09 \mathrm{MHz}$.
Explanation: One loop iteration on The First Implementation takes 11 clock cycles. (The duration of loop iteration $x$ can be found by subtracting the fetch time of the first instruction of iteration $x$ from the fetch time of the frst instruction of iteration $x+1$. In the loop above iteration 0 takes $11-0=11$ cyc and iteration 1 takes $22-11=11$ cyc. There's no guarantee that iteration 0 and 1 will take the same amount of time. However we can expect iteration 2 to take as much time as iteration 1 because the state of the pipeline is identical at cycles 11 and 22 : 1 lbu in IF , addi in ME , and bne in WB.) By similar reasoning one loop iteration on The Second Implementation takes 10 cycles. Dividing cycles by clock frequency gives time, so 11 cycles on The First Implementation takes $\frac{11}{\phi_{1}}=11 \mathrm{~ns}$. To find the clock frequency for The Second Implementation at which the two perform equally solve $\frac{11}{\phi_{1}}=\frac{10}{\phi_{2}}$ for $\phi_{2}, \phi_{2}=\phi_{1} \frac{10}{11}=909.09 \mathrm{MHz}$.

Grading Note: A common mistake was to assume that in The Second Implementation the sb and bne suffer only 1 -cycle stalls rather than the 2 -cycles that they suffer in The First Implementation. That's not a good mistake to make because the only change between the two implementations is where the branch resolves. The two-cycle stall starting in cycle 3 is due to the sb waiting for the value produced by the 1 bu, the stall starting in cycle 7 is due to the bne waiting for the value of $t 4$ written by addi. The change does not effect either stall.

Problem 2: The code below is taken from the solution to Homework 2 Problem 1. Sharp students might remember that the loop can be entered at four places: the COPY_LOOPd4 label (which is the normal way to enter such a loop), the second lb , the third lb , or the fourth lb . For this problem assume that the loop can only be entered at the COPY_LOOPd4 label.

```
COPY_LOOPd4:
    lb $t0, 0($t4) # First lb
    sb $t0, 0($a1)
    lb $t0, 1($t4) # Second lb
    sb $t0, 1($a1)
    lb $t0, 2($t4) # Third lb
    sb $t0, 2($a1)
    lb $t0, 3($t4) # Fourth lb
    sb $t0, 3($a1)
    addi $t4, $t4, 4
    bne $t4, $t5, COPY_LOOPd4
    addi $a1, $a1, 4
```


(a) Schedule the code (rearrange the instructions) so that it executes without a stall on the implementation shown above.

The solution appears below. The sb instructions have been moved away from the lb instructions, avoiding $\mathrm{lb} / \mathrm{sb}$ stalls. The increment of $t 4$ has been moved up, avoiding the need for the branch to stall. Note that the destination register of all but the first lb and the source of all but the first sb had to be changed to avoid the second lb clobbering the value loaded by the first 1 lb , ete.

Grading Note: One common mistake was to leave the lb and sb registers unchanged.

```
COPY_LOOPd4: SOLUTION
    lb $t0, O($t4) IF ID EX ME WB
    lb $t1, 1($t4) IF ID EX ME WB
    lb $t2, 2($t4) IF ID EX ME WB
    lb $t9, 3($t4)
    addi $t4, $t4, 4
    sb $t0, O($a1)
    sb $t1, 1($a1)
    sb $t2, 2($a1)
    sb $t9, 3($a1)
    bne $t4, $t5, COPY_LOOPd4
    addi $a1, $a1, 4
```

Problem 3: Perhaps some students have already wondered why, if the goal were to reduce dynamic instruction count, the previous occurrence loop (the subject of the first two problems and of Homework 2) wasn't written using lw and sw instructions since they handle four times as much data. Such a loop appears below. Alas, the loop won't work for every situation, for one reason due to MIPS' alignment restrictions.

Let $a_{p}$ denote the address of the previous text occurrence (the value is in t4), let $a_{o}$ denote the address of the next character to write into the output buffer (the value is in a1), and let $L$ denote the length of the previous occurrence to copy. (Register t5 is $a_{p}+L$.)

## COPY_LOOP44:

```
lw $t0, 0($t4)
sw $t0, 0($a1)
addi $t4, $t4, 4
bne $t4, $t5, COPY_LOOP44
addi $a1, $a1, 4
j LOOP
nop
```

(a) In terms of $a_{p}, a_{o}$, and $L$, specify the conditions under which the loop above will run correctly. Also show that the loop would work for about only 1 out of 64 copies assuming that the values of $a_{p}, a_{o}$, and $L$, are uniformly distributed over some large range. For this part don't assume any special code added before or after.

The loop will only run correctly if $a_{p}, a_{o}$, and $L$ are all multiples of 4 and if $L \geq 4$. That is $a_{p} \bmod 4=0$, $a_{o} \bmod 4=0, L \bmod 4=0$, and $L \geq 4$.

Suppose that previous occurrence lengths are uniformly distributed over range $[1,4 M]$, for some integer $M$. Because they are uniformly distributed the probability of each value in the range is $P(L=x)=\frac{1}{4 M}$ for $1 \leq x \leq 4 M$. Since $L \bmod 4=0$ for $M$ values in the range, those values occur $\frac{1}{4}$ of the time. The same argument can be made for the address of the previous occurrence and the current buffer location. If each is a multiple of four $\frac{1}{4}$ of the time and assuming their values are independent, all of them are multiples of four $\left(\frac{1}{4}\right)^{3}=\frac{1}{64}$ of the time.
(b) Suppose one added prologue code before the loop to copy the first few characters and epilogue code after the loop to copy the last few characters, with the goal of being able to use the loop for more than $\frac{1}{64}$ th (or $\frac{100}{64} \%$ ) of copies.

In terms of $a_{p}, a_{o}$, and $L$, specify the conditions under which the loop will run correctly and show that the fraction of copies that the loop can handle is about $\frac{1}{4}$.

Also show the number of characters that should be copied by the prologue code and the number of characters that should be copied by the epilogue code.

If the only requirement were that $L$ be a multiple of 4 , then prologue code could copy the first $L \bmod 4$ characters and the loop would handle the remaining $L^{\prime}=L-L \bmod 4$ characters. But t 4 and a1 must both also be a multiple of 4 when the COPY_LOOP44 loop is entered. Botn!

$$
\text { Suppose }\left(a_{p} \bmod 4\right)=\left(a_{o} \bmod 4\right) \text { and let } m=\left(a_{o} \bmod 4\right) \text { and } b=\left\{\begin{array}{ll}
0, & \text { if } m=0 ; \\
4-m, & \text { otherwise. }
\end{array}\right. \text {. If the prologue }
$$ loop executes $b$ iterations (and advances t 4 and a1) then when the COPY_LOOP44 is entered both t 4 and a1 will be multiples of 4. Execute $L^{\prime}=\lfloor(L-b) / 4\rfloor$ iterations in COPY_LOOP44 and then execute $L-4 L^{\prime}-b$ iterations in the epilogue code.

If $a_{p} \bmod 4 \neq a_{o} \bmod 4$ then COPY_LOOP44 cannot work.
Suppose $a_{p}, a_{0}$, and $L$ are uniformly distributed and independent, as considered in the second part. Then $a_{p} \bmod 4=a_{o} \bmod 4$ occurs $\frac{1}{4}$ of the times because there are 4 possible values of $a_{p} \bmod 4$ and 4 values for $a_{o} \bmod 4$, for a total of 16 pairs. Each pair is equally likely with probability $\frac{1}{16}$. For four of those pairs $a_{p} \bmod 4=a_{o} \bmod 4$,
the probability of any of those favorable pairs occurring is $4 \frac{1}{16}=\frac{1}{4}$. The value of $L$ does not matter so long as it is $L \geq 10$ since the epilogue will handle the last 1-3 characters.

Homework 5 solution
This assignment consists of questions on the ARM A64 (AAarch64) ISA. (Not to be confused with ARM A32, which might be called classic ARM. Older information sources that refer to ARM are probably referring to A32, which is not relevant to this assignment.)

A description of the ARM ISA is linked to the course references page, at http://www.ece.lsu.edu/ee4720/reference.html. Feel free to seek out introductory material as a suppliment.

ARM A64 was used in EE 4720 Spring 2017 Homework 4 and Spring 2017 Midterm Exam Problems 2 and 3. It may be useful to see those assignments for code samples, but the questions themselves are different.

Appearing on the next page is a simple C routine, lookup, that returns a constant from a list. The routine appears to have been written with the expectation that its call argument, i, would be either 0,1 , or 2 . Following the C code is ARM A64 code for lookup as compiled by gcc version 8 .

Use the course reference materials and external sources to understand the ARM code below. The course references page has a link to the ARM ISA manual which should be sufficient to answer questions in this assignment. Feel free to seek out introductory material on ARM A64 (AArch64) assembly language, but after doing so use the ARM Architecture Reference Manual to answer questions in this assignment.

Full-length versions of the code on the next page, along with other code examples can be found at http://www.ece.1su.edu/ee4720/2018/hw05.c.html and http://www.ece.lsu.edu/ee4720/2018/hw05-arm.s.html. These include the pi program and a simple copy program that was a part of the decompress program used in Homeworks 1, 2, and 3.

Code on next page, problems on following pages.

```
int lookup(int i)
{
    int c[] = { 0x12345678, 0x1234, 0x1234000 };
    return c[i];
}
```

@ ARM A64 Assembly Code. C code appears in comments.
lookup:
@®
©
© CALL VALUE
© w0: The value of $i$ (from the $C$ routine above).
@ RETURN VALUE
© w0: The value of $c[i]$.
@
© Note: The size of int here is 4 bytes.
© const int $c[]=\{0 x 12345678,0 x 1234,0 x 1234000\} ;$
adrp x1, .LC1
mov w2, 0x4000
ldr do, [x1, \#:lo12:.LC1]
movk w2, 0x123, Isl 16
str $w 2,[s p, 8]$
str do, [sp]
© return c[i];
ldr wo, [sp, w0, sxtw 2]
ret
.LC1 :
. word 0x12345678
.word 0x1234

Problem 1: The ARM code above uses three kinds of register names, those starting with d, w, and x .
(a) Explain the difference between each.

The d registers are registers in the SIMD \& floating-point register file. The w and x are in the general-purpose register file. The GPR file contains 3264 -bit registers, r0 to r31. In assembly language $x 0$ to $x 30$ refer to the entire 64 bits of r0 to r30, while w0 to w30 refer to the lower 32-bits of r0 to r30. In assembly language w31 and x31 refer to the constant zero and SP refers to register r31. The SIMD \& FP register file contains 32128 -bit registers, v0 to v31. In assembly language d0 to d31 refer to the low 64 bits. See section B1.2.1 of the ARM V8 Architecture Reference Manual (2017).
(b) MIPS has general-purpose registers and four sets of co-processor registers. Indicate the name of the register set for each of the three types of ARM registers above. Hint: two are part of the same set.

See answer above.
Problem 2: The mov moves constant $0 \times 4000$ into register w2. Actually, mov is a pseudo instruction. (a) What are pseudo instructions called in ARM?

They are called aliases.
(b) What is the real instruction that the assembler will use in this particular case?

The real instruction is movz with the shift amount set to zero.
(c) Show the encoding for this use of mov. Be sure to show how w2 and 0 x 4000 fit into the fields.

Solution appears below. The sf field is set to 0 because we are using a w register, which is 32 bits. The opcode fields are set based on the description in the architecture manual. The hw field is set to zero to indicate that the immediate is not shifted. The imm16 is the value appearing in the assembly code and $R d$ is the register number.

|  | Sf | ope | opc2 | nw | imm16 |  | R |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| movz: | 0 | ) 102 | $100101_{2}$ | 0 |  | $4000{ }_{16}$ |  | 2 |
|  | 31 | 3029 | 2823 | 2221 | 20 |  | 4 | 0 |

Problem 3: MIPS-I does not have an instruction like adrp.
(a) Describe what the adrp instruction does in general.

The adrp instruction writes its destination register with ( PC \& bitwise not 0xfff ) + ( imm << 12 ), where PC \& bitwise not $0 x f f f$ is the address of the adrp instruction with the 12 least-significant bits set to zero. See the next part for an example.
(b) Explain what it is doing in the code above. (It might be easier to look at the documentation for adr first.)

It is setting register x 1 to the address. .LC1 \& bitwise not $0 x f f f$ (address . LC1 with the 12 least-significant bits set to zero). Instruction ldr d0, [x1, \#:1012:. LC1] computes its address by adding the least significant bits of . LC1, which is the same as .LC1 \& Oxfff, to x1, the result of which is .LC1. The immediate value in the adrp instruction was set to (.LC1 >> 12 ) - ( PC >> 12 ).

The alert student may wonder why the compiler didn't choose an instruction that would set x1 to exactly .LC1, such as adr. It's hard to know for sure, but one possible reason was so the same register, x1 could be used as a base for accesses to multiple addresses, say .LC1, .LC2, etc., all of which had the same values for bits 63 to 12.
(c) Show MIPS code that writes the same value to its destination as adrp. Do not use MIPS pseudo instructions other than la. Assume that MIPS integer registers are 64 bits.

```
subi r3, r0, 0x1000 # Write r3 with Oxfffff000
la r2, .LC1
and r1, r2, r3 # Write r1 with low 12 bits set to zero.
```

Problem 4: The movk instruction is sort of an improved version of lui.
(a) Describe what the movk instruction does in general.

It moves a 16-bit immediate into a 16-bit section of its destination register, leaving the other 16 or 48 bits unchanged.
(b) Explain why a single MIPS lui instruction could not do what the movk is doing in the code above.

Because the lui always zeros the low 16 bits.

Problem 5: Add comments to the ARM code above that explain what the code is doing, rather than what the individual instructions do.

```
// SOLUTION
int lookup(int i) { int c[] = { 0x12345678, 0x1234, 0x1234000 }; return c[i]; }
@ ARM A64 Assembly Code. C code appears in comments.
lookup:
@@
@
@ CALL VALUE
@ wO: The value of i (from the C routine above).
@
@ RETURN VALUE
    w0: The value of c[i].
    Note: The size of int here is 4 bytes.
const int c[] = { 0x12345678, 0x1234, 0x1234000 };
    @
    @ Read the first two elements of the c array from memory at
    @ address .LC1, and compute the third element using immediates.
    @
    adrp x1,.LC1 @ Load x1 with base of c original array.
    mov w2, 0x4000 @ Compute lower 16-bits of 3rd c array elt.
    ldr d0, [x1, #:lo12:.LC1] @ Load 1st and 2nd elts of c array.
    movk w2, 0x123, lsl 16 @ Compute 3rd elt, 0x1234000
    @
    @ Write a copy of the c array to memory ...
    @ ... starting at the address in register sp.
    @
    str w2, [sp, 8] @ Write 3rd elt to a c array copy.
    str d0, [sp] @ Write 1st and 2nd elts to c array copy.
@ return c[i];
    ldr w0, [sp, w0, sxtw 2] @ Load the i'th (w0'th) element.
    ret
    .rodata.cst8,"aM",@progbits,8
.LC1:
\begin{tabular}{ll}
.word 0x12345678 & @ First element of c array \\
.word \(0 \times 1234\) & @ Second element of c array. \\
& @ This element intentionally left blank.
\end{tabular}
```

H

Problem 6: The lookup routine was compiled using gcc at optimization level 3, the highest. Nevertheless, the code appears more complicated than it need to be. Explain what about the code is excessively complicated and how it could be simplified.

Because the compiler, for whatever reason, decided to construct the carray using the first two elements stored at . LC1 and an instruction that wrote the third array element.

Most humans would simply put the entire c array at .LC1 so that the routine would consist of just two instructions, an adr instruction and an ldr instruction.

Problem 1: Solve 2014 Homework 4 Problem 1.
Problem 2: Appearing on the next page is a MIPS implementation and the execution of some code on that implementation.

(a) Wires in the implementation (on the previous page) are labeled in blue. Show the values on those wires each cycle that they are affected by the executing instructions. The values for label A are already filled in.

Solution appears below.
A common mistake for signals $B, C$, and $D$ was to omit the lwc instructions.
Signal $E$ is 1 when there is an add.s or similar instruction in ID that (that, not which) will enter A1 in the next cycle. Because both add.s instructions below are stalled signal E affects the respective instructions in the last cycle of the stall and so E is 1 in cycles 9 and 18 instead of 4 and 13 .

Signal $F$ stalls the pipeline if a lwc1 can't enter the pipeline due to a structural hazard at WF . This doesn't happen for the two lwc1 instructions below and so the signals are 0 in cycles 1 and 10 , when the respective instructions are in ID.

Signal G carries the register number to write back into the integer register file. The only instruction below that actually writes an integer register is addi. It writes register 1 and so G is 1 in cycle 22 , which is when addi is in ME.

```
LOOP: # Cycle 0
    lwc1 f0, O(r1) IF ID EX ME WF
    mul.s f1, f0, f10 IF ID -> M1 M2 M3 M4 M5 M6 WF
    add.s f2, f2, f1 IF -> ID ------------> A1 A2 A3 A4 WF
    lwc1 f0, 4(r1) IF ------------> ID EX ME WF
    mul.s f1, f0, f11 IF ID -> M1 M2 M3 M4 M5 M6 WF
    add.s f2, f2, f1
    bne r2, r1 LOOP
    addi r1, r1, 8
IF -> ID -----------> A1 A2 A3 A4 WF
                                    IF -------------> ID EX ME WB
                                    IF ID EX ME WB
```


(b) Schedule the code above so that it suffers fewer stalls. Register numbers can be changed but in the end the correct value must be in register f2. It is okay to add a few instructions before and after the loop. Each iteration must do the same amount of work as the original code.

Show a pipeline execution diagram for two iterations.
Two solutions appear below. The first reorganizes instructions within an iteration, but stalls six cycles per iteration. The second solution stalls just three cycles per iteration by executing the add. s instructions in the iteration after the corresponding mul.s was executed. All of the stalls in the second solution are due to structural hazards. (This is an example of software pipelining. Software pipelining was not covered in this course, though a related technique, loop unrolling, was covered.)

```
# SOLUTION: Easy, but not best solution.
LOOP: # Cycle: 0
    lwc1 f0, O(r1) IF ID EX ME WF
    IF ID EX ME WF
    lwc1 f5, 4(r1)
    mul.s f1, f0, f10
    mul.s f6, f5, f11
    addi r1, r1, 8
    add.s f2, f2, f1
    bne r1, r2, LOOP
    add.s f2, f2, f6
    LOOP: # Cycle:
    lwc1 f0, O(r1)
    lwc1 f5, 4(r1)
    mul.s f1, f0, f10
    mul.s f6, f5, f11
    addi r1, r1, 8
    add.s f2, f2, f1
    bne r1, r2, LOOP
    add.s f2, f2, f6
    LOOP: # Cycle: 0
    lwc1 f0, 0(r1) THIRD ITERATION IF ----> ID EX ME WF
```

```
# SOLUTION: Better solution: avoid more stalls by executing add.s's in next iteration.
# Prologue Code. Performs first iteration (except for adds).
    lwc1 f0, 0(r1)
    lwc1 f5, 4(r1)
    mul.s f1, f0, f10
    mul.s f6, f5, f11
    addi r1, r1, 8
    addi r2, r2, 8 # Increment because branch now checks r2 after r1 is incremented.
LOOP: # Cycle: 0
    lwc1 f0, O(r1) IF ID EX ME WF FIRST ITERATION
    add.s f2, f2, f1 IF ID A1 A2 A3 A4 WF
    lwc1 f5, 4(r1) IF ID EX ME WF
    addi r1, r1, 8 IF ID EX ME WB
    mul.s f1, f0, f10 IF ID M1 M2 M3 M4 M5 M6 WF
    add.s f2, f2, f6 IF ID A1 A2 A3 A4 WF
    bne r1, r2, LOOP IF ID EX ME WB
    mul.s f6, f5, f11 IF ID M1 M2 M3 M4 M5 M6 WF
LOOP: # Cycle: 
    lwc1 f0, 0(r1) IF ID -> EX ME WF
    add.s f2, f2, f1 IF -> ID A1 A2 A3 A4 WF
    lwc1 f5, 4(r1) IF ID ----> EX ME WF
    addi r1, r1, 8
    mul.s f1, f0, f10 IF ID M1 M2 M3 M4 M5 M6 WF
    add.s f2, f2, f6 IF ID A1 A2 A3 A4 WF
    bne r1, r2, LOOP IF ID EX ME WB
    mul.s f6, f5, f11 SECOND ITERATION IF ID M1 M2 M3 M4 M5 M6 WF
LOOP: # Cycle: 
    lwc1 f0, 0(r1) IF ID -> EX ME WF
# Epilogue Code: Performs add.s's for last iteration
    add.s f2, f2, f1
    add.s f2, f2, f6
```

Problem 3: Design control logic for the lower M1-stage bypass multiplexor. Note that this is fairly easy since the mux has two inputs, so it's only necessary to detect the dependence.

An SVG source for the FP diagram can be found at: http://www.ece.lsu.edu/ee4720/2018/mpipei-fp-by.svg.

Solution appears below in blue. The control signal is named bft. Its value is computed in ID and used in M1A1. The logic compares the ft register of the instruction in ID with the fd register of the instruction in A4M6, as well as checking whether the instruction in A4M6 is going to write a FP register (the we [write enable] bit is set). The logic drawn with dashed lines checks whether the instruction in ID uses an ft source register (uses the multiply or add pipeline). Conceptually this should be AND'ed with the output of the register comparison and the we signal, but in the solution its value is assumed true. That eliminates an OR gate and an input on the AND gate and does not interfere with correct operation.


## 45 Spring 2017 Solutions

## ．data

name：
．asciiz＂＂非 Put your name between the quotes

非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非非渄非非非
排 Problem 1 －－Split Routine－－SOLUTION

```
    .text
split:
    #⿰⿰三丨⿰丨三⿰⿰三丨⿰丨三\mp@code{Register Usage}
    #
    # CALL VALUES:
    # $a0: Number of elements in input array.
    ## $a1: Address of input array. Elements are 4 bytes each.
    # $a2: Address of output array, elements are 4 byte each.
    # $a3: Address of output array, elements are 2 byte each.
    # RETURN:
    ## $v0, Number of elements in 4-byte output array
    # Note:
    # Can modify $t0-$t9, $a0-$a3
    非 SOLUTION
    sll $t1, $a0, 2
    add $t9, $a1, $t1
    addi $v0, $a2, 0
LOOP:
    beq $a1, $t9, DONE
    lw $t0, 0($a1)
    srl $t1, $t0, 16
    beq $t1, $0, SMALL
    addi $a1, $a1, 4
    sw $t0, 0($a2)
    j LOOP
    addi $a2, $a2, 4
SMALL:
sh $t0, 0($a3)
j LOOP
addi $a3, $a3, 2
DONE：
sub \＄v0，\＄a2，\＄v0
jr \＄ra
srl \＄v0，\＄v0， 2
```



## 非 Testbench Routine

非
非
．data
array＿a：
．word $0 \times 1 \mathrm{ae} 2 \mathrm{~d} 15,0 \times \mathrm{c} 87,0 \times 0,0 \times 2 \mathrm{e} 89,0 \times 11 \mathrm{c} 70,0 \times 7 a 36 \mathrm{~b}, 0 \times 5,0 \times 1 \mathrm{e} 78 \mathrm{aa} 7,0 \times 1457 \mathrm{c} 01 \mathrm{f}, 0 \times 12,0 \times 21623,0 \times 8 c 86,0 \times 5,0 \times 2139 \mathrm{f} 25,0 \times 0,0 x$
array＿b1：
．space 32
array＿b4：
．space 64
ans＿b1：
．half $0 \times c 87,0 \times 0,0 \times 5,0 \times 12,0 \times 8 c 86,0 \times 5,0 \times 0,0 \times 10$
ans＿b4：
．word $0 \times 1 \mathrm{ae} 2 \mathrm{~d} 15,0 \times 2 \mathrm{ae} 89,0 \times 11 c 70,0 \times 7 a 36 \mathrm{~b}, 0 \times 1 \mathrm{e} 78 \mathrm{aa} 7,0 \times 1457 c 01 \mathrm{f}, 0 \times 21623,0 \times 2139 f 25$
msg＿oor：
．asciiz＂The number of elements in b4 is out of range：\％／s4／d．\n＂
msg＿elt：
msg_b1_head:
.asciiz " Should Be Written Array B1 \n"
msg_b4_head:
asciiz " Should Be Written Array B4\n"
text_xxx:
text_z:
.asciiz ""

## .text

.globl__start
__start:
addi \$a0, \$0, 16
la \$a1, array_a
la \$a2, array_b4
la \$a3, array_b1
jal split
addi \$v0, \$0, 4720
addi \$s4, \$v0, 0
addi \$s1, \$0, 16
sub \$s1, \$s1, \$v0
sltiu \$t0, \$s4, 17
bne \$t0, \$0, okay_range
nop
la \$a0, msg_oor
addi \$v0, \$0, 11 syscall
addi \$v0, \$0, 10 syscall
okay_range:
addi \$t0, \$0, 0
la \$t0, ans_b1
la \$t1, array_b1
add \$t4, \$t0, \$s1
add \$t4, \$t4, \$s1
la \$a0, msg_b1_head
addi \$v0, \$0, 11
syscall
la \$t6, text_z
la \$t7, text_xxx
la \$a0, msg_elt
loop_1:
Ih \$t2, 0(\$t0)
lh \$t3, 0(\$t1)
beq \$t2, \$t3, skip_1
add \$t5, \$t6, \$0
add \$t5, \$t7, \$0
addi \$t8, \$t8, 1
skip_1:
syscall
addi \$t0, \$t0, 2
bne \$t0, \$t4, loop_1
addi \$t1, \$t1, 2
la \$a0, msg_b4_head
addi \$v0, \$0, 11
syscall
addi \$t0, \$0, 0
la \$t0, ans_b4
la \$t1, array_b4
sll \$t4, \$s1, 2
add \$t4, \$t4, \$t0
la \$t6, text_z
la \$t7, text_xxx
la \$a0, msg_elt
loop_2:
1w \$t2, 0(\$t0)
lw \$t3, 0(\$t1)
beq \$t2, \$t3, skip_2
add \$t5, \$t6, \$0
add \$t5, \$t7, \$0
addi \$t8, \$t8, 1
skip_2:
syscall
addi \$t0, \$t0, 4
bne \$t0, \$t4, loop_2

Problem 1: The following problems are from the 2016 EE 4720 Final Exam.
(a) The following problem appeared as 2016 EE 4720 Final Exam Problem 2a. (Just 2a, not 2b). Show the execution of each of the two code fragments below on the illustrated MIPS implementations. All branches are taken. Don't forget to check for dependencies.


The solutions appear below. Notice that there is no need for a stall in the first example because $r 1$ can be bypassed and that in the second example because beq is resolved in ID the target can be fetched while the branch is EX, cycle 2 below. Instruction addresses have been added to the second code fragment for use in the next subproblem's solution.

```
# CODE SEQUENCE A -- SOLUTION
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sub r4, r1, r5 IF ID EX ME WB
```

Show execution of the following code sequence.

```
# CODE SEQUENCE B -- SOLUTION
# Cycle 0
beq r1, r1 TARG IF ID EX ME WB
or r2, r3, r4 IF ID EX ME WB
sub r5, r6, r7
xor r8, r9, r10
TARG:
lm r10, O(r11) IF ID EX ME WB
# Cycle 0
```

(b) For each of the two code sequences above label each used wire on the diagram below with: the cycle number, the instruction, and if appropriate a register number or immediate value. For example, the register file input connected to ID. IR bits 25:21 should be labeled 1: add, $=2$ because in cycle 1 the add instruction is using that register file input to retrieve register r2. See the illustration below. Only label wires that are used in the execution of an instruction. For example, there should not be a label 2: sub, $=1$ because the value of $r 1$ will be bypassed. Instead, label the bypass path that is used. Pay particular attention to wires carrying branch information and to bypass paths. Look through old homeworks and exams to find similar problems.

The solutions appear on the next page. For clarity, some paths are highlighted with the same color as the instruction that uses the path.

For Solution for Code Sequence A:


Solution For Code Sequence B:


Problem 2: The following problem is from 2016 EE 4720 Final Exam Problem 5a. Suppose that a new ISAis being designed. Rather than requiring implementations to include control logic to detect dependencies the ISA will require that dependent instructions be separated by at least six instructions. As a result, less hardware will be used in the first implementation.
(a) Explain why this is considered the wrong approach for most ISAs.

Short Answer: Because ISA features should be tied as little as possible to implementation features. The separation requirement in the new ISA is based on a particular implementation, one with five or six stages and that lacks dependency checking.

Long Answer: An ISA should be designed to enable good implementations over a range of cost and performance goals. It is reasonable to assume that low hardware cost was a goal of the first implementation and, as stated in the problem, the ISA's separation requirement helped achieve that goal. So the separation requirement would be a good idea if there was only going to be one implementation or if cost constraints and technology wouldn't change. However, ISAs are usually designed for a wide product line and a long lifetime. The separation requirement would become a burden if in the future a higher-performance, higher-cost implementation was needed.
(b) What is the disadvantage of imposing this separation requirement?

Short Answer: It would make a higher-performance implementation that included bypass paths pointless since the compiler would be forced to separate dependent instructions, possibly by inserting nops, and so the bypass paths would never be used.

Long Answer: Call the ISA with the separation requirement, ISA $S$ (strict), and one that lacked the separation requirement but was otherwise identical, ISA $N$ (normal). Consider a low-cost implementation of each ISA. Neither implementation would include bypass paths but the implementation of ISA $N$ would need to check for dependencies and stall if any were found, making the cost slightly higher than that of the implementation if ISA $S$. Now consider a code fragment compiled for both ISAs in which the version for ISA $S$ requires nops to meet the separation requirement. (The version for ISA $N$ has fewer instructions.) Now consider their execution on their respective implementations. The execution times should be the same because for each nop in the ISA $S$ implementation there will be a stall in the ISA $N$ implementation.

So the performance of the low-cost implementations of the ISAs are about the same, but the cost of the ISA $N$ version is slightly higher. (Note that the hardware for checking dependencies is of relatively low cost since it operates on register numbers, 5 bits each in many ISAs. That is in contrast to the cost of the hardware for bypasses, which are 32 or 64 bits wide and include multiplexors.)

Next, consider high-performance implementations. For ISA $N$ we can add bypass paths, as we've done in class. As a result, some instructions that would stall in the low-cost implementation would not stall in the high-performance implementation, and so code would run faster. In contrast, code for ISA $S$ would still have nop instructions since the ISA itself imposes the instruction separation requirement. That would make it much harder to design higher performance implementations.

Therefore, the disadvantage of the instruction separation requirement is that it leads to much higher-cost highperformance implementations with only a small cost beneft for the low-cost implementation.

Grading Note: Several students incorrectly answered that a disadvantage of the separation requirement is that it would be tedious and error prone for hand (human) coding, and that it would require that the compiler schedule instructions rather than having the hardware check for dependencies and stall when necessary.

It is $100 \%$ true that hand-coding assembly language with such a requirement would be tedious, especially considering branch targets. However, an ISA is designed to facilitate efficient implementations, not to improve assembly language programmers' productivity. (An assembler can still be helpful by pointing out likely separation issues.) Also, if something can be done equally well in the compiler and in hardware, it should be done in the compiler because the cost of compiling is borne beforehand, by the developer. In contrast, the higher cost of the hardware is payed by every customer and the energy of execution is expended each time the program is run.

To help in solving this problem it might be useful to study the solutions to the following problems which involve hardware implementing branches in the statically scheduled five-stage MIPS implementation we've been working with: Spring 2016 Homework 3 (SPARC-like branch instruction), Spring 2015 Homework 2 Problem 2 (use reg bits for larger displacement) and Problem 3 (logic for IF-stage mux), Spring 2015 Homework 3 Problem 2 (implement bgt, but resolve it in EX), Spring 2011 Final Exam Problem 1 (resolve in ME, with bypass).

Problem 1: Modify the implementation below so that it implements the MIPS II bgezall instruction, see the subproblems for details on the hardware to be designed. See the MIPS ISA documentation linked to the course Web page for a description of the bgezall instruction. An Inkscape SVG version of the illustration below can be found at http://www.ece.lsu.edu/ee4720/2017/hw03-p1.svg. The illustration also appears on the next page.


Solution on next page.

(a) Design control logic to detect the instruction and connect its output to the br/jump logic cloud. The control logic should consist of basic gates, not a box like bgezall.

Solution appears above in green. The green logic also computes a taken signal, and it is that which is connected to the br/jumplogic. Note that the $\geq 0$ condition is true when the rs value is not negative, which can be inexpensively and quickly checked by looking at bit 31 of rsv.
(b) Design the control logic to squash the delay slot instruction when bgezall is not taken. The control logic should squash the delay slot instruction by changing its destination register and memory operation. Be sure that the control logic squashes the correct instruction, and does so only when bgezall is not taken. Do not rely on magic clouds [tm].

Solution appears above in red. The squash signal is asserted when there is a bgezall instruction in ID and when it is not taken. (A common mistake was to ignore whether or not there was a bgezall in ID when generating the
squash signal.) The squash signal is put in a new pipeline latch where it joins other signals related to the instruction in IF. In the next cycle the sq signal is used to replace the memory op with a nop memory operation (the name nop is whatever the no memory operation code is called). The sq signal also selects register zero as an output register. In the solution the contents of the decode dest reg logic block is shown, which was not necessary in a submitted solution.
(c) Add datapath or make other changes needed to compute the return address. Note that NPC is already connected to the ALU. Consider inexpensive ways to compute the second operand. (Adding a 32 -bit ID/EX pipeline latch is not considered inexpensive for this problem.)

Solution appears below (and above) in blue. As described in the ISA manual, the return address is PC+8 or NPC+4. In ID the immediate is replaced with the constant 4. This is done before the fmt imm logic so that a 16 -bit mux rather than a 32-bit mux can be used. Also, the decode dest logic is modified so that 31 is used as a destination when bgezall is in ID. When bgezall is in EX the ALU will be used to add NPC and the 4.


To help in solving this problem it might be useful to study the solutions to the following problems: Spring 2014 Homework 3 (ARM A32 instructions, ARM-like scaling instructions). Fall 2010 Homework 3 (shift unit in MIPS).

See the course references page for a link to the ARMv8 ISA, which will be needed to solve the problems below.

Problem 1: In most RISC ISAs register number 0 is not a true register, its value as a source is always zero, and it can be harmlessly used as a destination (for example, for use as a nop instruction).

The ARM A64 instruction set (not to be confused with A32 [arm] or T32 [thumb]) takes a different approach to the zero register.
(a) What register number is the zero register in A64?
(b) Let $z$ denote the answer to the previous part, meaning that rz can denote the zero register. In an ISA like MIPS, the general purpose register (GPR) file only needs enough storage for 31 registers, since the register zero location can be hardwired to zeros. But in ARM A64 the GPR file needs 32 storage locations because register number $z$ is the zero register for some instructions, but an ordinary register for others. In ARM notation ZR denotes the zero register, in this problem rz indicates the register number of the zero register, which, depending on the instruction can refer to the zero register or to an ordinary register.

Show two instructions that can read $r z$ and one that can write $r z$, for these instructions $r z$ is an ordinary register (at least for certain operand fields).

Show a one-destination-register, two-source-register instruction for which $r z$ is the zero register for all operand fields.

Problem 2: The ARM A64 code below computes the sum of an array of 64 -bit integers. The load instruction uses post-index addressing, the behavior of this instruction is shown in the comments. (The @ is the comment character.)

```
LOOP: @ ARM A64
    ldr x1, [x2], #8 @ x1 = Mem[x2]; x2 = x2 + 8
    cmp x2, x4
    add x3, x3, x1
    bne LOOP
```

(a) Appearing below is a pipeline based on our MIPS implementation. Add datapath elements so that it can execute the ldr with pre-index, post-index, and immediate addressing. Don't make changes that will break other instructions. Note that the register file has a write-enable input, which is necessary because there is no full time register that acts as register zero.

An Inkscape SVG version of the implementation can be found at https://www.ece.lsu.edu/ee4720/2017/hw04-armskel.svg.


- Show the bits used to connect to the address inputs of register file.
- Show the second write port on the register file needed for the updated address.
- Use fixed bit positions for the destination registers.
- Use the given decode logic to determine a write enable signal for each dest.
- The changes must work well with pipelining.
- As always, avoid excessively costly solutions.
- Do not add hardware for the branch or compare.


Problem 1: Complete MIPS routine fxitos so that it converts a fixed point integer to a singleprecision floating point value as follows. When fxitos starts register a0 will hold a fixed-point value $i$ and register a1 will hold the number of bits that are to the right of the binary point, $d$, with $d \geq 0$. For example, to represent $9.75_{10}=1001.11_{2}$ we would set a0 to 0 b 100111 and a1 to 2. (Or we could set a 0 to 0 Ob 100111000 and a1 to 5 .) When fxitos returns register f 0 should be set to $i / 2^{d}$ represented as a single-precision floating point number.

Solve this problem by using a division instruction for $i / 2^{d}$. (The floating division instruction can be avoided by performing integer arithmetic on the FP representation, but that's not required in this problem.)

Submit the solution on paper. Your class account can be used to work on the solution. The fxitos routine and a testbench can be found in
/home/faculty/koppel/pub/ee4720/hw/2017/hw05/hw05.s, follow the same instructions as for Homework 1.
fxitos:

```
##
#
# CALL VALUES:
# $a0: Fixed-point integer to convert.
# $a1: Number of bits to the right of the binary point.
#
# RETURN:
# [ ] $f0: The value as a single-precision FP number.
```

```
##
#
# Let d denote value of $a1, # of digits to the right of binary point.
# Let i denote value of $a0, the fixed point number to convert.
# Need to set $f0 to i / 2^d, where 2^d is 2 to the d'th power.
addi $t0, $0, 1
sllv $t2, $t0, $a1 # Construct value 2^d
mtc1 $a0, $f1 # Move i to a FP register. (But it's still an int.)
mtc1 $t2, $f2 # Move 2^d to a FP register. (Still an int too.)
cvt.s.w $f11, $f1 # Convert i from an integer to SP FP.
cvt.s.w $f12, $f2 # Convert 2^d from an integer to SP FP.
jr $ra
div.s $f0, $f11, $f12 # Compute i / 2^d.
```

Problem 2: Appearing below is the MIPS hardware needed to implement bgezall from the solution to Homework 3. Recall that with bgezall the delay-slot instruction is annulled if the branch is not taken. Modify the hardware for new instruction bgezalllsu which executes like bgezall when the branch target is at or before the branch, but when the target is after the branch the delay-slot instruction is annulled when the branch is taken and allowed to execute normally if the branch is not taken. The opcode and rt values are the same as for bgezall. Hint: this can be done with very little hardware, a gate or two, if that.


Solution appears above. Whether the branch target is before or after the branch can be determined by looking at the sign bit of the immediate value, which is bit 15 of ID. IR and is labeled backward in the diagram above. If that bit is

1 it means the immediate is negative and so the branch target is before the branch (to be $100 \%$ precise, it means that the branch target is before the delay slot instruction). Branch bgezalllsu is taken if rsv is not positive, that condition is provided at the output of the NOT gate and labeled $>=0$ above. Based on the description of bgezalllsu given in the problem statement, the delay-slot instruction should be annulled (squashed) if the XOR of these two conditions is true.

Note that in the solution to Homework 3 Problem 1 the NOT gate used to determine the branch-taken condition is shown there as a bubble at an input to the TAKEN AND gate. Here it is shown as a free-standing NOT gate so that its value can be used for the XOR. The TAKEN signal itself could have been used instead of the NOT gate output, but that would have resulted in a slightly longer critical path. Since a synthesis program could easily optimize the logic the variation to use is the one that's easier for humans to understand.

Problem 3: Suppose that an analysis of the execution of benchmark programs on our pipelined MIPS implementation shows that over $75 \%$ of bypassed values can be represented with 12 bits or fewer. A low-cost implementation takes advantage of this fact by using 12 -bit bypass paths.
(a) The control logic below is intended for bypass paths that can bypass a full 32 -bit value. Modify the control logic shown so that it works for 12-bit bypass paths. In your modified hardware add a stall signal to be used when values are too large to be bypassed.

- Indicate which parts of the added logic, if any, may lengthen the critical path.
- As always, avoid costly or slow hardware.

Attention perfectionists: An Inkscape SVG version of the implementation below can be found at http://www.ece.1su.edu/ee4720/2017/mpipei3c.svg.


Solution appears above. Logic has been added at the output of the ALU (in the EX stage, shown in green) that checks whether the value is too big for the 12-bit bypass paths. It does so by checking whether bits 31 through 11 are either all 0 (a small positive value) or all 1 (a small negative value). (The reason that bits $31: 11$ rather than $31: 12$ are checked is to make sure that the sign bit of the 12-bit bypassable portion matches the parts that we won't bypass.) If the value at the output of the ALU is too big to bypass then the output of the NOR gate is 1 , that signal is put in the abig (ALU output is big) pipeline latch.

Added control logic, shown in blue, checks this abig signal. If ByME is 1 that means the instruction in ID will need to use the ByME bypass path in the next cycle (when it is in EX). The upper blue AND gate checks whether the value in EX is too large to bypass, if so the stall signal is 1. If the instruction in ID will need to use the ByWB path and the abig bit in the ME stage is 1 we will also need to stall.

This logic only works for dependencies to the rt register (of the instruction in ID), and only when the producing instruction (the instruction in EX or ME while the consuming instruction is in ID) uses the ALU (not the memory port) to produce a value. See the examples below.

It would be a simple matter-simple enough for a midterm exam problem-to modify the logic to handle a dependency to the rs register of the instruction in ID, such as Example II below. On the other hand, bypassing for a 1 w is hopeless due to our usual critical path assumptions.

The logic generating the abig signal might be stretching the critical path since it doesn't get started until the ALU produces a value. If we really need such a signal we might ask the ALU guys whether they can design an ALU that produces an abig-like signal without threatening the critical path, perhaps taking advantage of carry-lookahead logic, for example.

```
# Example I
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sub r4, r5, r1 IF ID EX ME WB # Logic active in cyc 2, bypass in cyc 3.
# Example II
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sub r4, r1, r5 IF ID EX ME WB
# Example III
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
sub r4, r5, r6 IF ID EX ME WB
xor r7, r8, r1 IF ID EX ME WB # Logic active in cyc 3, bypass in cyc 4.
# Example IV
# Cycle 0
lw r1, O(r2) IF ID EX ME WB
sub r4, r5, r6 IF ID EX ME WB
xor r7, r8, r1 IF ID EX ME WB
```

(b) Why would it be far more challenging for a compiler to optimize for these 12-bit paths than for ordinary full-width bypass paths?

The compiler would need to know whether it was possible to use a bypass path for some pair of dependent instructions, which means the compiler would need to know whether the register value would fit in 12 bits. In most cases the compiler will not be able to tell the size of a value in a register because that can depend on input data that can vary from run to run or it might depend on other pieces of code that the compiler does not have access to. There are a few situations in which the compiler could figure it out. For example:

```
andi r1, r2, 0xff # Value in r1 must be between 0-255.
addi r3, r1, 3 # Value in r3 must be between 3-258.
sub r4, r5, r3 # Can use 12-bit bypass here for r3.
```

In the example above the values in r1 and r3 can easily fit in 12 bits (since the $r 1$ value was masked down to 8 bits). Therefore the compiler can assume a bypass can be used from the addi to the sub and so it will not was time finding instructions to put between them.

The compiler could also use profiling to determine at least a range of values for registers. The compiler doesn't need to be $100 \%$ sure that register values are small, since the hardware will stall if the values are too large.

The problem statement mentioned that $75 \%$ of bypassed values fit within 12 bits, and we might expect that a profiling analysis to come to the same conclusion. However that doesn't really help us because that doesn't mean that $75 \%$ of dependent instruction pairs pass values that fit within 12 bits.

Attention Perfectionists: An Inkscape SVG version of the illustration used in the final exam and this assignment can be found at: https://www.ece.1su.edu/ee4720/2017/mpipei_fp.svg.

Problem 1: Answer Spring 2016 Final Exam Problem 2b and 2c, which ask about the execution of FP MIPS code. The solution to these problems are available. Make a decent attempt to solve these problems on your own, without looking at the solution. Only peek at the solution for hints and use the solution to check your work.

Problem 2: Appearing below are two MIPS code fragments and the MIPS implementation from the final exam. The fragments execute on the illustrated implementation with the addition of the datapath needed for the store instructions that was provided in Final Exam Problem 2c. The fragments are labeled Degree 1 and Degree 2, these refer to an optimization technique called loop unrolling, which has been applied to the Degree 2 loop.

(a) Show a pipeline execution diagram of each on the illustrated code fragments. Show enough iterations to compute the CPI. Note that the second loop should have fewer stalls than the first.

Solution appears below. Note that the state of the pipeline at the start of the second iteration, in cycle 9, is the same as the state of the pipeline at the start of the third iteration, in cycle 18. In both cases the addi is in ID, the bne is in EX, and the swc1 is in ME. Therefore we can use the second iteration to compute CPI. (It just so happens that we could also the first iteration, but we didn't prove that we could with the argument above.)

```
# Degree 1 -- SOLUTION
LOOP: # First Iteration
    # Cycle 0
    lwc1 f0, O(r1) IF ID EX ME WF
    add.s f0, f0, f1 IF ID -> A1 A2 A3 A4 WF
    swc1 f0, 0(r1) IF -> ID -------> EX ME WB
    bne r1, r3 LOOP IF -------> ID EX ME WB
    addi r1, r1, 4 IF ID EX ME WB
LOOP: # Second Iteration
    # Cycle 0
    lwc1 f0, 0(r1) IF ID EX ME WF
    add.s f0, f0, f1 IF ID -> A1 A2 A3 A4 WF
    swc1 f0, 0(r1) IF -> ID -------> EX ME WB
    bne r1, r3 LOOP IF -------> ID EX ME
    addi r1, r1, 4 IF ID EX ME WB
        # Cycle 
LOOP: # Third Iteration
    lwc1 f0, O(r1) IF ID EX ME..
```

\# Degree 2 -- SOLUTION
LOOP: \# First Iteration
\# Cycle $\begin{array}{lllllllllllllllllllll} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 29\end{array} 20$
lwc1 f0, O(r1) IF ID EX ME WF
lwc1 f1, 4(r1) IF ID EX ME WF
add.s $f 0, f 0, f 9 \quad$ IF ID A1 A2 A3 A4 WF
add.s f1, f1, f9 IF ID A1 A2 A3 A4 WF
swc1 f0, 0(r1) IF ID ----> EX ME WB
swc1 f1, 4(r1) IF ----> ID EX ME WB
bne r1, r3 LOOP IF ID EX ME WB
addi r1, r1, 8 IF ID EX ME WB
$\begin{array}{lllllllllllllllllllllll}\text { \# Cycle } & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 29 & 20\end{array}$
LOOP: \# Second Iteration
lwc1 f0, O(r1) IF ID EX ME WF
lwc1 f1, 4(r1) IF ID EX ME WF
add.s f0, f0, f9 IF ID A1 A2 A3 A4 WF
(b) Compute the execution efficiency of both loops in CPI. Remember that the number of cycles should be determined by looking at the same point in execution, usually IF of the first instruction, in two different iterations. Put another way, just because the custom car you will order after graduation will take two months to arrive, doesn't mean that the factory makes just one car every two months.

The Degree 1 loop has a CPI of $\frac{18-9}{5}=1.8$ CPI based on the iteration start times of cycle 9 and cycle 18. The Degree 2 loop has a CPI of $\frac{10-0}{8}=1.25 \mathrm{CPI}$, under the assumption that all iterations will execute in the same way as the first.
(c) Assume that both loops operate on $N$-element arrays (and that $N$ is even). The Degree-1 loop operates on just one element per iteration, while the Degree-2 loop operates on two elements per iteration.

Devise a performance measure that can be used to compare the two loops based on the work that they do. The improvement of Degree-2 or Degree-1 should be higher with this work-based performance measure than the improvement computed using CPI.

A reasonable measure would be cycles per element or CPE. The Degree 1 loop computes one element of the array per iteration, and so it computes $\frac{18-9}{1}=9 \mathrm{CPE}$. The Degree 2 loop computes two elements per iteration, and so it computes at $\frac{10-0}{2}=5 \mathrm{CPE}$, almost twice as fast.

Based on CPI the Degree 2 loop is $\frac{1.8}{1.25}=1.44$ times better than the Degree 1 loop. (Remember that lower CPI is better.) But based on CPE the Degree 2 loop is $\frac{9}{5}=1.8$ times better.

Note that CPI is appropriate for comparing two implementations that run the same program, but it's not useful for comparing two different programs.
(d) Besides eliminating stalls, what makes Degree 2 faster than Degree 1 even when doing the same amount of work?

There are fewer instructions per element. For example, an addi is executed for each array element in Degree 1, but it is executed for every two array elements in Degree 2.

Attention Perfectionists: An Inkscape SVG version of the illustration of the superscalar MIPS implementation used in the final exam problems and their solution for this assignment can be found at http://www.ece.lsu.edu/ee4720/2016/fe-ss.svg and http://www.ece.lsu.edu/ee4720/2016/fe-p1abc-sol.svg.

Problem 1: Answer Spring 2016 Final Exam Problem 1 a, b, and c, in which a single memory port is connected to the ME stage of a two-way superscalar MIPS implementation. The solution to this problem is available. Make a decent attempt to solve this problem on your own, without looking at the solution. Only peek at the solution for hints and use the solution to check your work.

See posted final exam solution at http://www.ece.lsu.edu/ee4720/2016/fe_sol.pdf.
Problem 2: Answer Spring 2016 "Final Exam Problem" 1e, which asks for modifications to a 2-way supescalar MIPS implementation that avoids stalls for certain pairs of load instructions. Note: Problem 1d was given on the final exam. Problem 1e, which did not appear on the final, is an expanded version of Problem $1 d$.

See posted inal exam solution at http://www.ece.1su.edu/ee4720/2016/fe_sol.pdf.

Problem 1: Answer Spring 2016 Final Exam Problem 3, which asks about the performance of various branch predictors.

The solution to this problem is available. Make a decent attempt to solve this problem on your own, without looking at the solution. Only peek at the solution for hints and use the solution to check your work. Credit will only be given if there is some evidence of an attempt to solve the problem.

Problem 2: Compute the amount of storage needed for each predictor described at the beginning of Spring 2016 Final Exam Problem 3 (the same question used in the problem above) accounting for the following additional details: Each BHT stores a six-bit tag and a 16-bit displacement (in addition to whatever other data is needed).

Be sure to show the size of each table (BHT, PHT) that each predictor (bimodal, local, global) uses. Show the size in bits.
Bimodal Predictor:
Short Answer: BHT, $2^{14}(2+6+16)$ b.
Long Answer: The BHT, as stated in the problem, has $2^{14}$ entries. Each entry stores a 2 -bit counter, a 6 -bit tag, and a 16 -bit displacement. The total size of an entry is therefore 24 bits, and the total storage used for the BHT is $24 \mathrm{~b} \times 2^{14}$. Note that bimodal predictors do not have a PHT.

Local Predictor:
Short Answer: BHT, $2^{14}(10+6+16) \mathrm{b} ;$ PHT, $2^{10} 2 \mathrm{~b}$.
Long Answer: The BHT contents is the same as the BHT used by the bimodal predictor except that the 2 bit counter is replaced by a 10 -outcome local history (which is encoded in 10 bits). The size of a BHT entry is thus $10+6+16=32 \mathrm{~b}=4 \mathrm{~B}$ and the total storage is $4 \times 2^{10} \mathrm{~B}$. Since the local history length is 10 outcomes the PHT (pattern history table) has $2^{10}$ entries. Each entry is a 2-bit counter, for a total size of $2 \times 2^{10}=2048 \mathrm{~b}$.

Global Predictor:
Short Answer. BHT, $2^{14}(6+16) \mathrm{b} ;$ PHT, $2^{10} 2 \mathrm{~b}$.
Long Answer: The BHT contents is the same as the contents of the BHT in the bimodal predictor except that there is no 2 -bit counter. The total size of an entry is therefore $6+16=22 \mathrm{~b}$ and there are $2^{14}$ entries for a total size of $2^{14}(6+16) \mathrm{b}=360448 \mathrm{~b}=45056 \mathrm{~B}=44 \mathrm{kiB}$. Because the global history size, 10 outcomes, is the same as the history size in the local predictor the PHT contents the same as the PHT in the local predictor, $2 \times 2{ }^{10} \mathrm{~b}$.

Problem 3: In a bimodal predictor the size of the tag and displacement is much larger than the 2-bit counter used to actually make the prediction. Consider a design that uses two tables, a BHT and a Branch Target Buffer (BTB). The BHT stores only the 2-bit counter, the BTB stores the tag and displacement. However, the tag and displacement are only written to the BTB if the branch will be predicted taken.

Draw a sketch of such a system and indicate the number of entries that should be in each table so that the amount of storage is the same as the original bimodal predictor.

Let $m$ denote the number of bits in the original BHT address or, put another way, let $m$ denote ceiling log-base-2 of the BHT size. For the exam problem $m=14$ for a BHT size of $2^{m}=2^{10}$ entries.

For our new predictor let $m_{h}$ denote the number of bits in the BHT address and let $m_{t}$ denote the number of bits in the BTB address. If $m_{h}=m_{t}=m$ then the two predictors are equivalent. The size of the new predictor is $2^{m_{h}} \times 2+2^{m_{t}}(6+16) \mathrm{b}$.

Since the BTB will only be used for taken branches, we can have fever entries. Assuming half of all branch outcomes are taken we would need half the number of BTB entries as BHT entries. With this condition we can set $m_{t}=m_{h}-1$.

To properly size the new predictor solve

$$
2^{m}(2+6+16)=2^{m_{h}} \times 2+2^{m_{t}}(6+16) \mathrm{b}
$$

for $m_{h}$ after substituting $m_{t}=m_{h}-1$ :

$$
\begin{aligned}
2^{m}(2+6+16) & =2^{m_{h}} \times 2+2^{m_{t}}(6+16) \\
& =2^{m_{h}} \times 2+2^{m_{h}-1}(6+16) \\
& =2 \times 2^{m_{h}-1} \times 2+2^{m_{h}-1}(6+16) \\
& =2^{m_{h}-1}(4+6+16) \\
2^{m_{h}-1} & =2^{m} \frac{2+6+16}{4+6+16}
\end{aligned}
$$

Taking the log base 2 of both sizes:

$$
\begin{aligned}
& m_{h}-1=m+\lg \frac{2+6+16}{4+6+16} \\
& m_{h}=m+\lg \frac{2+6+16}{4+6+16}+1
\end{aligned}
$$

Substituting $m=14$ gives us $m_{h}=14.88$ and $m_{t}=13.88$. If $m_{t}$ is rounded up to 14 and if we limit the amount of storage to no more than the original bimodal predictor then we could not set $m_{h}=m_{t}+1$ without running out of storage. So, lets set $m_{t}=13$ and choose $m_{h}$ to use of the remaining storage, $m_{h}=16$.

An ordinary bimodal predictor appears below on the left, the bimodal predictor with the BTB appears below to the right. Notice that the bits indexing (connecting to the address input) of the BTB and BHT are different, and are based on $m_{t}=13$ and $m_{h}=16$ chosen above. For the BTB bits $14: 2$ of the address are used, a total of 13 bits. (The two low bits are omitted because the address of an instruction must be a multiple of 4 and so those two bits will always be zero.) Also notice that when the branch is resolved the new 2 -bit counter value is checked to see whether the branch is taken. The BTB is only written if the branch is taken.


## 46 Spring 2016 Solutions

Homework 1 solution

Problem 1: Answer each MIPS code question below. Try to answer these by hand (without running code).
(a) Show the values assigned to registers t1 through t8 (the lines with the tail comment Val:) in the code below. Refer to the MIPS review notes and MIPS documentation for details.

Solution appears below (to the right of SOLUTION, of course).

```
    .data
myarray:
    .byte 0x10, 0x11, 0x12, 0x13
    .byte 0x14, 0x15, 0x16, 0x17
    .byte 0x18, 0x19, 0x1a, 0x1b
    .byte 0x1c, 0x1d, 0x1e, 0x1f
```

```
.text
la $s0, myarray # Load $s0 with the address of the first value above.
    # Show value retrieved by each load below.
lbu $t1, 0($s0) # Val: SOLUTION: 0x10
lbu $t2, 1($s0) # Val: SOLUTION: 0x11
lbu $t2, 5($s0) # Val: SOLUTION: 0x15
lhu $t3, 0($s0) # Val: SOLUTION: 0x1011
lhu $t4, 2($s0) # Val: SOLUTION: 0x1213
addi $s1, $0, 3
add $s3, $s0, $s1
lbu $t5, 0($s3) # Val: SOLUTION: 0x13
sll $s4, $s1, 1 SOLUTION: # Note: s4 <= 3<<1 = 6
add $s3, $s0, $s4
lhu $t6, 0($s3) # Val: SOLUTION: 0x1617
sll $s4, $s1, 2 SOLUTION: Note: s4 <= 3<<2 = 12
add $s3, $s0, $s4
lhu $t7, 0($s3) # Val: SOLUTION: 0x1c1d
lw $t8, 0($s3) # Val: SOLUTION: 0x1c1d1e1f
```

(b) The last two instructions in the code above load from the same address. Given the context, one of those instructions looks wrong. Identify the instruction and explain why it looks wrong. (Both instructions should execute correctly, but one looks like it's not what the programmer intended.)

Register s0 holds an address that the programmer decided to call myarray, so lets think of the data starting at that address as an array. Normally, to access element i of an array that starts at address a, you load data at address a + i $* \mathbf{s}$, where $\boldsymbol{s}$ is the size of an array element. In the code fragment above, register so holds the starting address (a in the example). From the way the code is written it looks like register s1 is holding the element index ( $i$ in the example). Because the sll in the last group of four instructions is effectively multiplying s1 by 4 , it looks like the load should be
of the s1'th element of an array of elements of size 4 . That's consistent with the 1 w , which loads a 4 -byte element, and the last lhu looks out of place. The lhu that loads t 6 looks fine, because its address was computed from a value of s1 multiplied by 2.
(c) Explain why the following answer to the question above is wrong for the MIPS 32 code above: "The lw instruction should be a lwu to be consistent with the others."

There is no lwu, because when loading a 32-bit quantity into a 32 -bit register there is no need to distinguish between a signed and unsigned quantity. In contrast, the 1 hu and 1 lh load a 16 -bit quantity into a 32 -bit register, the lhu sets the high 16 bits to zero, it zero-pads, while 1 h sets the high 16 bits to the value of the MSB of the loaded value, it sign extends.

Problem 2: Note: The following problem was assigned last year and two years ago and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
LOOP: # Cycles 0
    lw r2, 0(r4) IF ID EX ME WB
    add r1, r2, r7 IF ID EX ME WB
LOOP: # Cycles 
```

The add depends on the 1 w through $r 2$, and for the illustrated implementation the add has to stall in ID until the 1 w reaches WB.

```
LOOP: # Cycles 0
    lw r2, O(r4) IF ID EX ME WB
    add r1, r2, r7 IF ID ----> EX ME WB
LOOP: # Cycles 0
```

(b) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3
    IF ID EX ME WB
    lw r1, 0(r4)
    IF ID -> EX ME WB
LOOP: # Cycles
0
```

There is no need for a stall because the $1 w$ writes $r 1$, it does not read $r 1$.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    lw r1, O(r4) IF ID EX ME WB
LOOP: # Cycles 0
```

(c) Explain error and show correct execution.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    sw r1, O(r4) IF ID -> EX ME WB
LOOP: # Cycles 0
```

A longer stall is needed here because the sw reads r1 and it must wait until add reaches WB.

```
LOOP: # Cycles 0
    add r1, r2, r3
    sw r1, O(r4)
    IF ID ----> EX ME WB
LOOP: # Cycles
    0
```

(d) Explain error and show correct execution.

```
LOOP: # Cycles 0
    add r1, r2, r3
    IF ID EX ME WB
    xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: # Cycles 0
```

The stall above allows the xor, when it is in ID, to get the value of $r 1$ written by the add; that part is correct. But, the stall starts in cycle 1 before the xor reaches ID, so how could the control logic know that the xor needed $r 1$, or for that matter that it was an xor? The solution is to start the stall in cycle 2, when the xor is in ID.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    xor r4, r1, r5 IF ID ----> EX ME WB
LOOP: # Cycles 0}0
```

Problem 3: Show the execution of the MIPS code below on the illustrated implementation. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be value being written. (In class we called such a register file internally bypassed.)

- Check carefully for dependencies.
- Pay attention to when the branch target is fetched and to when wrong-path instructions are squashed.
- Be sure to stall when necessary.


The solution appears below. Note that because MIPS branches are delayed, the 1 w instruction is allowed to complete execution even though the branch is taken. The xor, in contrast is on the wrong path and so is squashed after the branch resolves. Also note the timing of the fetch of the branch target based on this particular implementation: the branch target, ori, enters IF in the cycle after the branch leaves ME.

```
# SOLUTION
```



```
add r1, r2, r3
IF ID EX ME WB
sub r4, r1, r5 IF ID ----> EX ME WB
beq r1, r1, SKIP IF ----> ID EX ME WB
lw r6, O(r4) IF ID -> EX ME WB
    xor r7, r8, r9 IF -> x
SKIP:
    ori r9, r10, 11 IF ID EX ME WB
# Cycle 
```

Problem 1: The code fragment below is to execute on the illustrated MIPS implementation. Unfamiliar instructions can be looked up on the MIPS ISA manual linked to the course references page. Show the execution of the code fragment below on the illustrated MIPS implementation. All branches are taken.

- Pay close attention to dependencies, including those for the branch.
- Note that unnecessary stalls are just as incorrect as not stalling when a stall is necessary.


Solution appears above. The 1w does not stall because the r4 register value can be bypassed. That's not possible for the sub, since there is no bypass for the r6 value it needs because the r6 value isn't available until the end of cycle 4, too late to bypass. The bltz stalls two cycles, waiting for the value of $r 1$ that it needs to arrive in $I D$, where it needs it. The and is in the delay slot and so executes normally. Since the branch resolves in ID (as we can tell by the connections from ID to the PC mux) the target, sw, will be in IF when the branch is in EX.

Problem 2: The implementation below (which is the same as the implementation for the previous problem) lacks hardware needed for the bltz instruction. In this problem design such hardware as described in the parts below. Note: An Inkscape SVG version of the implementation can be found at https://www.ece.1su.edu/ee4720/2016/mpipei3.svg.

(a) Add the hardware needed to detect when a bltz is taken. The hardware should have an output labeled TAKEN, which should be set to logic 1 if there is a taken bltz in ID. Include control logic, including the logic for detecting bltz.

The solution appears above. The logic in blue is used for detecting a bltz instruction. That instruction has an opcode of 1 , but also requires that the rt field be zero. In the solution above AND gates are used to detect these, rather than the usual $=$ bitz and $=0$ boxes. Either would be correct. The branch is taken if the rsv is negative, that can be detected by looking at the MSB. That's shown in green (the solution to the part below is also shown). The rightmost AND gate detects the taken condition for this branch.
(b) The solution to the previous problem (not the previous part to this problem) should have included a stall due to the branch instruction. Add a bypass path to the hardware designed above so that the branch from the previous problem can execute without stalling.

Logic also shown above in green. In particular the green mux and the connection to the output of the ALU. We can get away with using the ALU output in this case because we only need examine one bit. If we needed to look at all 32 bits there would not be enough time left in the clock cycle.

An important thing to remember is that this is one of the few cases where we can get away with using the ALU output. See Problem 3 a.
(c) Design control logic for the bypass path.

Logic shown in purple.

Problem 3: The code below is similar to the code from the first problem, the only difference is in the branch instruction. In this problem explain some bad news and good news about that branch.

```
add r4, r2, r3
lw r6, 8(r4)
sub r1, r6, r5
beq r0, r1 TARG
and r8, r7, r10
or r11, r12, r13
xor r14, r11, r8
```

TARG:

```
sw r1, O(r2)
```

(a) The bad news is that adding bypass paths for a beq would not be a good idea, even though adding bypass paths for the bltz was a good idea. Explain why.

Testing the condition for a bltz instruction is simple: just check if the MSB of the rs value is 1 . That's a good thing because the bypass path added in the previous problem was from the output of the ALU, and so was available at the end of the cycle. In contrast, the beq must compare two registers, which requires about 7 layers of logic. (An XNOR to test equality of each bit pair and a 32 -input AND gate to check that all XNOR output are 1 . The 32 -input AND gate might be realized with five layers of two-input and gates.)

As pointed out in class, the ALU output will not be available until close to the end of the clock period, and so there is no time for testing equality.

Grading Note: Many answered that the difficulty was in the bypass paths since two values had to be bypassed. Though that's true it ignores two things. First, for bltz all we need is the MSB, so the difference between the two cases is more like a factor of 64 than a factor of 2 in bypass path cost. It also ignores the more significant problem of testing equality, which stretches the critical path. Nevertheless, full credit was given for the "two values" answer.
(b) The good news is that the program above can easily avoid the stalls by just changing the branch instruction. Explain how. (Of course, it should go without saying that the changed program must do the same thing as the original one.)

Change the branch to a beq r6, r5 TARG, thus avoiding the need for a bypass. The sub can not be removed because the sw uses the value of $r 1$.


Problem 1: Illustrated below is our MIPS implementation with some control logic shown. Modify the implementation so that it can execute the SPARC v8 instructions as described below. In your solution ignore register windows, assume that SPARC uses an ordinary 32-register general-purpose register file.

Details of the SPARC ISA (which includes later versions) can be found in http://www.ece.1su.edu/ee4720/doc/JPS1-R1.0.4-Common-pub.pdf. An Inkscape SVG version of the illustration below can be found at http://www.ece.1su.edu/ee4720/2016/mpipei3b.svg.

(a) Modify the implementation for format 3 arithmetic instructions. Use add as an example. Show changes in the bits used: to index the register file, to format the immediate, and to generate the writeback register number, dst.

Solution appears in blue on the previous page. The bits at the input to the register file have been changed, and notice also that the pipeline latches at the register file outputs were renamed, for example, from rsv to rsiv. Since all SPARC integer instructions that write a value, use the rd field for that value. For that reason the dst mux input for rt has been removed. Also note that the immediate unit uses fewer bits. (If the sethi instruction were implemented then the number of bits might be expanded.)
(b) Modify the implementation for branch instructions. Use BPcc as an example. Be sure to make changes for computing the branch target.

Show changes in the hardware to generate the target address. Remove the unneeded MIPS branch comparison hardware and add a CC register.

Solution shown in purple. The branch displacement bits were changed at the input to the ID-stage adder. Also, a CC output was added to the ALU and the value is carried through the pipeline to the ID stage where it's used to write a new CC register. (It goes without saying that bypass paths could be added to the branch control logic.) Also notice that he comparison unit in ID has been removed (shown with a red ex).
(c) Modify the implementation for load and store instructions. Use LDUW and STW as examples.

Show changes in the format immediate unit, and make sure that it can handle both ADD and loads and stores.

Changes shown in green. Only the STW instruction requires further changes. An instruction like stw r1, [r2+r3] has three source registers, and so a third read port has been added to the register file. A path for the retrieved value, rdv, is provided to the Mem Port Din and red ex breaks the old path (from what was rtv).

Problem 2: Section 1.3.1 of the SPARC JPS1 lists features of the ISA.
(a) Indicate which features are typical RISC features and which features are not.

The typical RISC features are those that facilitate pipelined implementations and make it easy to compile code. They are 32-bit instructions (as opposed to variable-sized instructions), few addressing modes, and triadic register addresses (as opposed to having arithmetic instructions read memory or forcing an arithmetic instruction to use the same register for its first source and destination, or something like that).
(b) One feature is "Branch elimination instructions" Provide an example of how such an instruction can be used to eliminate a branch.

The conditional moves are the branch elimination instructions. They write a register if a condition is true. For example, movg r1, r2 (move greater than) will write $r 2$ with the value of r1 only if the ICC register $Z$ (zero) and $N$ (negative) bits are both zero (meaning the last CC instruction result was strictly greater than zero). If the condition is not true r2 is unchanged.

See page 276 of the SPARC JPS1 for an example.
Note: other ISAs, such as ARM, achieve the same result using predication.

For the solution to the final exam questions used in this assignment visit
http://www.ece.lsu.edu/ee4720/2015/fe_sol.pdf

Problem 1: Problem 2b from the 2015 Final Exam asks about our usual FP MIPS pipeline.
An Inkscape SVG version of the FP pipeline can be found at http://www.ece.lsu.edu/ee4720/2016/mpipei_fp.svg.
(a) Solve Spring 2015 Final Exam problem 2b.

See the posted solution at the link above.
(b) Add bypass paths to the implementation from problem 2 b that were omitted but are needed in the execution of the code sample.

Note: The original assignment asked for bypasses needed so that the code would execute without a stall. That is obviously impossible since with zero stalls the multiply would have to start before the add finished.

The solution appears below in blue. Only one bypass connection is needed, for the value of $f 9$ from the 1 wc1 to the mul.s. The value of $f 1$ needed by mul.s will be read from the register file during the stall.


Problem 2: Solve Problem 2c from the 2015 Final Exam, which asks about our usual superscalar pipeline.

An Inkscape SVG version of the ordinary 2-way superscalar MIPS pipeline used in 2c can be found at http://www.ece.1su.edu/ee4720/2016/mpipei3ss.svg.

See the posted solution at the link above.
Problem 3: Solve Problem 1 from the 2015 Final Exam, which asks about a modified version of our two-way superscalar MIPS implementation.

An Inkscape SVG version of the fused-add 2-way superscalar MIPS pipeline used in 2c can be found at http://www.ece.1su.edu/ee4720/2015/fess.svg.

See the posted solution at the link above. Perhaps you are wondering why I didn't just put the link right here. Because the link is to the entire exam solution, and so the same link would be repeated three times which on balance might be a tiny bit more irritating than having to find the link near the top of this page.

Problem 1: Solve Spring 2015 Final Exam Problem 3, which asks about the performance of several branch predictors. See older final exam solutions for more information on how to solve these kinds of problems.

See the final exam solution at http://www.ece.lsu.edu/ee4720/2015/fe_sol.pdf.
Problem 2: Show major elements of the hardware for each predictor used in Spring 2015 Final Exam Problem 3a. In particular:

- Show the BHT, PHT, and GHR (in those predictors that use them).
- Show the connection from the PC to the appropriate table.
- Show the number of bits in each connection.
- Show the logic generating a "predict taken" signal.

You do not need to show the logic to update the predictor or to generate the target address. Solution appears below. The prediction is made using the MSB of the 2 -bit counter, it it's 1 predict taken. Note that the size of the BHT here is ridiculously large, realistically the number of BHT entries would be on the order of $2^{14}$.


## 47 Spring 2015 Solutions

Problem 1: Answer each MIPS code question below. Try to answer these by hand (without running code).
(a) Where indicated, show the changed register in the following simple code fragments:

```
# r1 = 10, r2 = 20, r3 = 30, etc.
#
add r1, r2, r3
#
# Changed register, new value: SOLUTION: r1 from 10 to 50.
# r1 = 10, r2 = 20, etc.
#
add r1, r1, r2
#
# Changed register, new value: SOLUTION: r1 from 10 to 30.
```

(b) Show the values assigned to registers s1 through s6 in the code below. Correctly answering this question requires an understanding of MIPS big-endian byte ordering and of the differences between lw, lbu, and lb. Refer to the MIPS review notes and MIPS documentation for details.

```
    .data
values: .word 0x11121314
    .word 0xaabbccdd
    .word 0x99887766
    .word 0x41424344
    .text
    la $s0, values # Load $s0 with the address of the first value above.
    lw $s1, 0($s0) SOLUTION: 0x11121314
    lw $s2, 4($s0) SOLUTION: 0xaabbccdd
    sh $s2, 0($s0) # Note: this is a store *half*.
    lbu $s3, 0($s0) SOLUTION: 0xcc
    lbu $s4, 3($s0) SOLUTION: 0x14
    lb $s5, 4($s0) SOLUTION: 0xffffffaa
    lb $s6, 7($s0) SOLUTION: 0xfffffffdd
```

Problem 2: Note: The following problem was assigned last year and its solution is available. DO NOT look at the solution unless you are lost and can't get help elsewhere. Even in that case just glimpse. Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
LOOP: # Cycles 
    lw r2, O(r4) IF ID EX ME WB
    add r1, r2, r7 IF ID EX ME WB
LOOP: # Cycles 
```

The add depends on the 1 w through r 2 , and for the illustrated implementation the add has to stall in ID until the 1 w reaches WB.

```
LOOP: # Cycles 0
    lw r2, O(r4)
    add r1, r2, r7 IF ID ----> EX ME WB
LOOP: # Cycles }\quad0\begin{array}{llllllll}{0}&{1}&{2}&{3}&{4}&{5}&{6}&{7}
```

(b) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3 IF ID EX ME WB
    lw r1, O(r4) IF ID -> EX ME WB
LOOP: # Cycles 
```

There is no need for a stall because the $1 w$ writes $r 1$, it does not read $r 1$.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    lw r1, O(r4) IF ID EX ME WB
LOOP: # Cycles 
```

(c) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3 IF ID EX ME WB
    sw r1, O(r4) IF ID -> EX ME WB
LOOP: # Cycles }\quad0\quad
```

A longer stall is needed here because the sw reads r1 and it must wait until add reaches WB.

```
LOOP: # Cycles 0
    add r1, r2, r3
    sw r1, O(r4)
    IF ID ----> EX ME WB
LOOP: # Cycles
    0
```

(d) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3 IF ID EX ME WB
    xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: # Cycles 
```

The stall above allows the xor, when it is in ID, to get the value of r1 written by the add; that part is correct. But, the stall starts in cycle 1 before the xor reaches ID, so how could the control logic know that the xor needed r1, or for that matter that it was an xor? The solution is to start the stall in cycle 2 , when the xor is in ID.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    xor r4, r1, r5 IF ID ----> EX ME WB
LOOP: # Cycles 
```

Problem 3: Show the execution of the MIPS code below on the illustrated implementation. The register file is designed so that if the same register is simultaneously written and read, the value that will be read will be value being written. (In class we called such a register file internally bypassed.)

- Check carefully for dependencies.
- Pay attention to which registers are sources and which are destinations, especially for the sw instruction.
- Be sure to stall when necessary.


Solution appears below. Since there are no bypass paths the lw must stall in ID until add reaches WB. (If the register file were not internally bypassed the add would have to stall in ID one more cycle than it does below.) The sub stalls for $r 4$ produced by $1 w$, and sh stalls for r5 produced by sub.

Common Mistakes:
One common mistake is forgetting that store instructions, such as sw and sh, do not write registers. That's why sub does not need to wait for $r 1$.

Another common mistake is assuming that a standard set of bypass paths is available. The implementation in this problem does not have bypass paths which is why the code below suffers from so many stalls.

```
# SOLUTION
# Cycle 
add r1, r2, r3 IF ID EX ME WB
lw r4, O(r1) IF ID ----> EX ME WB
sw r1, O(r1) IF ----> ID EX ME WB
sub r5, r4, r1 IF ID -> EX ME WB
sh r5, 4(r1) IF -> ID ----> EX ME WB
# Cycle 
```

Problem 4: The code below is the same as the code used in the previous problem, but the MIPS implementation is different.
(a) Show the execution of the MIPS code below on the illustrated implementation.

Solution appears below.
(b) On the diagram label multiplexor data inputs connecting to bypass paths that are used in the execution of this code. The label should include the cycle number, the register, and the instruction consuming the value. For example, the label $3: \mathrm{r} 1: \mathrm{lw}$ might be placed next to one of the data inputs on the ALU's upper mux.

Solution appears below in blue. Notice that in this case there are no stalls.


```
# SOLUTION
# Cycle 0
    add r1, r2, r3
    lw r4, O(r1)
    sw r1, O(r1)
    sub r5, r4, r1
    sh r5, 4(r1)
# Cycle 0
    IF ID EX ME WB
    IF ID EX ME WB
        IF ID EX ME WB
                        IF ID EX ME WB
                                IF ID EX ME WB
```

For those preparing electronic submission of a solution (E-mail) and who would like a vectorformat version of the MIPS implementation can find it in Encapsulated Postscript at http://www.ece.lsu.edu/ee4720/2015/mpipei3.eps and for those who would like to edit the image can find it in Inkscape SVG at http://www.ece.lsu.edu/ee4720/2015/mpipei3.svg.

Problem 1: The following problem appeared on the Spring 2014 Final Exam as Problem 7c. Suppose the 16 -bit offset in MIPS 1w instructions was not large enough. Consider two alternatives. In alternative 1 the offset in the existing lw instruction is the immediate value times 4 . So, for example, to encode instruction lw r1, 32 (r2) the immediate would be 8 . In alternative 2 the behavior of the existing lw is not changed but there is a new load lws r1, $32(r 2)$, in which the immediate is multiplied by 4 . Note that alternative 2 requires a new opcode. Which instruction should be added to a future version of MIPS, alternative 1 or alternative 2? Explain.

Alternative 2,1 ws, should be chosen so that existing software continues to run correctly.
Problem 2: The following problem appeared on the Spring 2014 Final Exam as Problem 5. The displacement in MIPS branches is 16 bits. Consider a new MIPS branch instruction, bfeq rsn, $r \operatorname{tn}$ (branch far), where rsn and rtn are 2-bit fields that refer to registers 4-7. As with beq, branch bfeq is taken if the contents of registers rsn and rtn are equal. With six extra bits bfeq can branch 64 times as far.
(a) Show an encoding for this instruction which requires as few changes to existing hardware as possible. Explain how your choice of encoding minimizes changes.

The encoding for bfeq is shown below. The 22-bit displacement is is3, it3, Immed (in Verilog notation).


In the encoding above the rs and rt register fields each have been shortened from five to two bits. (The original format I encoding is shown below.) Since the fields have been shortened but not moved the four remaining bits can be connected directly to the register file. (See the solution to the next part.)


With the encoding above we need two 3 -bit multiplexors at the register file address inputs (see the next subproblem). The encoding below, which would receive partial credit, would require two 5 -bit multiplexors at the register file address inputs and so would be more costly. This inferior encoding is certainly more organized with rs2 and rt2 next to each other and with the immediate occupying 22 contiguous bits. However the multiplexors at the register file inputs would need to be five bits instead of three bits. Notice that it does not make a difference whether or not the immediate bits are contiguous, as they are below, or split, as they are in the solution above.


Grading Note: Despite the hint about cost, almost no one used the preferred solution, the one with the immediate field split.
(b) Modify the pipeline below to implement the new instruction. Use as little hardware as possible. Changes appear below. At the register file address inputs, shown in blue, the high three bits of each register number is determined by a multiplexor. If a bf eq is present the upper inputs are used, making the upper three bits of each register number $001_{2}$, otherwise the upper bits come from the instruction. The lower two bits are taken from the instruction regardless of whether bfeq is present. If a bfeq is present the displacement includes the extra six bits, these changes are shown in green.

Grading Note: Many solutions did show logic selecting a larger displacement size, but that larger displacement was sent to the EX stage as an immediate rather than to the adder feeding the IF-stage multiplexor. We are proud of our resolve-branch-in-ID pipeline, so of course points must be deducted when this feature is ignored. Better to have points deducted on a homework than on a test.


Problem 3 on next page.

Problem 3: Show the control logic for the IF-stage multiplexor in the MIPS implementation below.

- The control logic should work for beq, bne, bgtz, bgez, and j. Assume that any other instruction is not a control transfer.
- Show exactly which IR bits are needed by the control logic that detects bgez (Hint, hint.) and other instructions.
- The control logic should check the condition to determine if the branch is taken.
- Pay attention to cost.

Solution appears below in blue. The input numbers on the IF-stage mux were chosen so that the input for a taken branch (shown as $\mathrm{t}-\mathrm{br}$ ) is $01_{2}$, the input for a jump is $10_{2}$, and the input for the incremented PC (inc) is $00_{2}$.

The logic detecting a $j$ is connected directly to the most-significant bit of the multiplexor select input. The logic detecting each of the four kinds of branches (which obviously is not a complete list) connects to an AND gate which detects whether the respective condition is true.

The full opcode for the bgez instruction is in the opcode and rt fields. (The rt field for the bgez plays the same role as the func field for the type R instructions.) For the jump and the other branches listed it is sufficient to only look at the opcode field.

A lower-cost solution would use just one comparison unit with a mux at the lower input selecting either rtv or the constant 0 . That would make the critical path in ID a bit longer.


Problem 1: For the following question refer to the Intel 64 and IA- 32 Architectures Software Developer's Manual linked to the course references page. Intel 64 is an example of a CISC ISA, but not a good example because it evolved from an ISA designed for a 16-bit address space. Over the years the size of the general purpose registers increased from 16 bits to 64 bits and the number of general-purpose registers increased from 8 to 16 .
(a) Show the 64 -bit names of the general purpose registers provided by Intel 64. (See Chapter 3 of the manual mentioned above.)

The 64-bit names are: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15.
(b) A MIPS assembly language instruction uses the same name for a register regardless of how many bits of the register we use. For example, sb r1, $0(r 2)$ uses 8 bits of r1 and sw r1, $0(r 2)$ uses all 32 bits, but in both instructions we refer to r1. Not so in IA-32/Intel 64, in which the name of the register indicates how many bits to use. Show the names for RAX for the different sizes and positions in the register.

Parts of RAX are known as EAX (bits 0 to 31 ), AX (bits 0 to 15), AL (bits 0 to 7), and AH (bits 8 to 15).
Problem 2: Diagrams of the MIPS implementation for this problem can be found in EPS format at http://www.ece.lsu.edu/ee4720/2015/hw02-p3-if-mux-sol.eps and in Inkscape SVG (which can easily be edited) at http://www.ece.lsu.edu/ee4720/2015/hw02-p3-if-mux-sol.svg.

As has been pointed out in class, MIPS lacks a bgt rs, rt, TARG (branch greater than comparing two registers) instruction because the ISA was designed for a five-stage implementation in which the branch is resolved in ID. To resolve bgt in ID one would have to compare two register values starting about half-way through the cycle, something that might slow down the clock frequency.

In this problem suppose there was a bgt instruction in MIPS. We would like the implementation to have the same clock frequency as our bgt-less implementation. One way of doing that is by resolving bgt in EX (but still resolving the other branches in ID as they are now). If we resolve in EX we can expect a one-cycle branch penalty, as can be seen in the PED below.

```
# Cycle }\quad0\quad
    bgt r1, r2, TARG IF ID EX ME WB
    add r3, r4, r5 IF ID EX ME WB
    xor r6, r7, r8 IF IDx
TARG:
    lw r9, O(r10) IF ID EX ME WB
# Cycle 0
```

Note that xor is squashed in cycle 3, which is the behavior we want for a taken bgt (see the second subproblem below). If bgt were not taken then no instruction would be squashed.
(a) Modify the implementation on the next page (taken from the Homework 2 solution) so that bgt is resolved in EX. Note: The original assignment had a very big typo in the previous sentence: giving ID instead of EX as the stage to resolve in.

- Pay attention to cost. Assume that a magnitude comparison (e.g., greater than) is relatively costly.
- Show the control logic for bgt.
- Do not "break" existing instructions.

Solution appears below. The bgt branch is taken if the rs register contents is greater than the rt register contents. Since we are resolving the branch in EX we can use the ALU to detect whether this condition is true. When a bgt is in ID the control logic, shown in dark yellow, chooses a greater-than operation for the ALU. A greater-than operation sets the ALU output to 1 if the upper input is strictly larger than the lower input, and to 0 otherwise. (It is like the less than operation that must be used for the slt instruction.) An AND gate added to the EX stage, shown in blue checks whether a bgt is in EX and if the condition is true (notice that the logic only needs to look at the LSB of the ALU output).

The adder computing the branch target can now get its inputs from either ID or EX, controlled by the is BGT, these changes are shown in purple. The reason for using is BGT (from EX!) and not the taken bgt signal (the output of the AND gate) is timing. is BGT is available at the beginning of EX, giving the adder plenty of time. The bgt taken signal is available near the end of EX (since it depends on the ALU output), so it would be too late to start computing the target address.

Because the same adder is used for ID- and EX-stage branches no changes need to be made to the IF-stage mux. However, we do need to add a condition for selecting input $01_{2}$, that's shown with the green OR gate which merges the ID-stage or EX-stage condition. (Fortunately for us, MIPS disallows two consecutive branches. Otherwise, a branch in ID would have to stall if there was a taken bgt in EX.)

An alternative solution is to compute the target address in ID and pass it to EX, that is discussed further below.
Grading Notes.
In some solutions the is BGT logic is in EX, with its input connected to a new ID/EX. IR pipeline latch. Adding a 32 -bit pipeline latch is more expensive than adding just one bit. Of course, one only really needs to send the opcode field, but if a solution does not explicitly state that it, is at risk of having points deducted for waste.
(b) If bgt is taken an instruction will have to be squashed. (Because bgt has just one delay slot, just like all the other branches.) Add logic so that a one-bit signal sq (squash) is delivered to ID when the instruction in ID needs to be squashed due to a taken bgt.

The logic for the squash signal is shown in red. It is the bgt taken signal sent to the new IF/ID. SQ pipeline latch.


In an alternative design, the target computed in ID is passed to EX, where it is used for taken bgt instructions. That design appears below. This alternative design is more costly because the 32 bits of new pipeline latch and 32 bits of mux input are more costly than the $30+16$ bits of mux input for the shared adder used in the first solution.


An EPS version of the MIPS FP implementation used in some of the problems below can be found at http://www.ece.lsu.edu/ee4720/2015/mpipei_fp.eps and an easy-to-edit Inkscape SVG version can be found at http://www.ece.lsu.edu/ee4720/2015/mpipei_fp.svg.

Problem 1: Solve 2014 Midterm Exam Problem 2, which asks for a stall-in-ME version of our floating-point pipeline. A solution to this problem is available but use it only if you are stuck, and after you are finished to check your answer. If you got it wrong, then solve the problem again without looking at the solution.

See the posted final exam solution.
Problem 2: Solve 2014 Final Exam Problem 1, which asks for an execution diagram of code running on the solution to the 2014 Midterm Problem 2.

See the posted final exam solution.

Problem 3: In the FP implementation on the next page (which is the same as the one used in class) an add.s instruction can stall due to an earlier mul.s, see the example below.

```
# Execution of code on the illustrated implementation.
# Cycle 0
mul.s f0, f1, f2 IF ID M1 M2 M3 M4 M5 M6 WF
add.s f6, f7, f8 IF ID A1 A2 A3 A4 WF
add.s f3, f4, f5 IF ID -> A1 A2 A3 A4 WF
and r6, r7, r8 IF -> ID EX ME WB
```

To avoid the stall consider the fpa-4/6 design in which an add.s instruction that would stall taking the usual route instead enters the FP pipeline at the M1 unit. Assume that the M1 unit's control signal (not shown and not part of the problem) will command it to pass the values at its inputs to its outputs unchanged when it is carrying add.s operands. Then at the appropriate time it crosses over to A1 and continues through the remaining adder stages. An add.s not facing a WF structural hazard stall would go from ID to A1, as in the usual design. See the execution below.

```
# Desired execution on the fpa-4/6 implementation.
# Cycle 0
mul.s f0, f1, f2 IF ID M1 M2 M3 M4 M5 M6 WF
add.s f6, f7, f8 IF ID A1 A2 A3 A4 WF # Uses 4-stage (normal) path.
add.s f3, f4, f5 IF ID M1 M2 A1 A2 A3 A4 WF # Uses 6-stage (M1 M2..) path.
and r6, r7, r8 IF ID EX ME WB
```

(a) Modify the pipeline to implement fpa-4/6.

- Show the datapath for the operands crossing from the multiply to the add unit.
- Show the control logic. The control logic should only send add.s into M1 if it would stall taking the usual route.
- The control logic should include the we, fd, and xw signals, and signals for any multiplexors that you add.
- As always, pay attention to cost and critical path.

Solution shown on the next page. The datapath changes appear in blue. Note that the values sent to A1 are taken from the output of the pipeline latches, which are available in the beginning of the clock cycle.

Grading Note: A common mistake was to connect the outputs of M2 to the multiplexors added before A1. That's wrong because the data would arrive one cycle early and because it assumes that M2 has a fast path for unmodified data.

The signal indicating that an add should pass through M1 and M2 is labeled lpa, for long-path add. Signal lpa is generated by the same logic that detected the add.s WF structural hazard condition, but now the connection to the Stall ID signal is broken (with the red ex).

The lpa signal travels with the add.s. In M3 it is used to select the multiplier value as inputs to A1. In M3 lpa also changes the xw signal to 1 , indicating that the result will come from the adder. Also notice that in ID the we signal is set if the lpa is needed or if the instruction is a multiply.
(b) In the code fragment above the add.s $f 3$ goes from ID to M1. If it had gone from ID to M2 it would have still avoided the WF hazard and it also would have finished one cycle earlier. Consider
an fpa-4/5/6 design in which an add.s can start at A1, M2, or M1, using the first one that avoids a stall. Provide a code example that would finish sooner on an fpa- $4 / 5 / 6$ design than on an fpa- $4 / 6$ design. Hint: A correct answer can add just one more instruction to the code fragment above.

Solution appears below. There is a dependency between the sub.s and the add.s that uses the long path. If the add.s $f 3$ when from M1 to A1 the sub. s would stall one cycle less.

```
# Cycle 0
mul.s f0, f1, f2 IF ID M1 M2 M3 M4 M5 M6 WF
add.s f6, f7, f8 IF ID A1 A2 A3 A4 WF # Uses 4-stage (normal) path.
add.s f3, f4, f5 IF ID M1 M2 A1 A2 A3 A4 WF # Uses 6-stage (M1 M2..) path.
sub.s f9, f3, f10 IF ID -------------> A1 A2 A3 A4 WF
```

(c) Is the fpa- $4 / 5 / 6$ design better than the fpa- $4 / 6$ design? Justify your answer using reasonable cost estimates and made-up properties of typical user programs. Either yes or no is correct, credit will be given for the justification.

The $\mathrm{fpa}-4 / 5 / 6$ design would cost more because the multiplexors at the adder inputs would need to have one more input each. Multiplexors would also have to be added for the fd signal, among other complications. The cost could only be justified if instructions reading the result of add instructions frequently stalled due to data dependencies.


An EPS version of the MIPS superscalar implementation used in one of the problems below can be found at http://www.ece.1su.edu/ee4720/2015/mpipei3ss.eps and an easy-to-edit Inkscape SVG version can be found at http://www.ece.lsu.edu/ee4720/2015/mpipei3ss.svg.

Problem 1: Solve 2014 Final Exam Problem 2, which asks for control logic in a 2-way superscalar processor.

See posted exam solution.
Problem 2: Solve 2014 Final Exam Problem 3, in which branch predictors are analyzed. Note: Check earlier final exams for solutions to similar problems.

See posted exam solution.

Problem 1: Solve 2014 Final Exam Problem 4, the cache problem. See the posted final exam solution.

## 48 Spring 2014 Solutions

Problem 1: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.


```
lw r1, -6(r3)
lw r5, -2(r3)
LOOP:
    add r5, r5, r1
    lw r1, 2(r3)
bneq r3, r4, LOOP
addi r3, r3, 4
jr r31
sw r5, O(r6)
```

(a) Show the execution of the code above on the illustrated implementation up to and including the first instruction of the third iteration (that is, the third time that the add instructions is fetched).

- Carefully check the code for dependencies.
- Be sure to stall when necessary.
- Pay careful attention to the timing of the fetch of the branch target.

Solution appears below.

```
lw r1, -6(r3) IF ID EX ME WB
lw r5, -2(r3) IF ID EX ME WB
LOOP: # Cycles 0
    add r5, r5, r1 IF ID ----> EX ME WB
    lw r1, 2(r3) IF ----> ID EX ME WB
    bneq r3, r4, LOOP IF ID EX ME WB
    addi r3, r3, 4 IF ID EX ME WB
    jr r31
    sw r5, O(r6)
LOOP: # Cycles
    add r5, r5, r1
    lw r1, 2(r3)
    bneq r3, r4, LOOP
    addi r3, r3, 4
    jr r31
    sw r5, O(r6)
LOOP: # Cycles (0) 1 2 2 3 3 4 4 5
    add r5, r5, r1
```

```
                IF IDx
```

                IF IDx
                                    IFx
                                    IFx
    0}1
0}1
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB

```
                                    IF ID EX ME WB
```

                                    IF ID EX ME WB
                                    IF IDx
                                    IF IDx
                                    IFx
                                    IFx
                                    IF ID ..
    ```
                                    IF ID ..
```

(b) Compute the CPI for a large number of iterations.

Recall that we define an iteration to start when the first instruction is in IF. In the execution above the first iteration starts in cycle 2 and the second iteration starts in cycle 10, and the third starts in cycle 16.

Notice that the first and second iterations are different: in the first there is a stall, in the second there isn't a stall. The first iteration takes $10-2=8$ cycles and the second takes $16-10=6$ cycles.

To compute the CPI for a large number of iterations we need a repeating pattern. The stalls in the first iteration are caused by instructions before the loop, they won't affect subsequent iterations. Even so, how can we be sure that the third and subsequent iterations will be like the second? By looking at the state of the pipeline when the first instruction in the loop is fetched. For the first iteration the state is add in IF, lw r5 in ID and lw r1 in EX. In the second iteration we have add in IF , addi in ME and bneq in WB. The third iteration starts exactly the same way as the second. Therefore the third will look like the second, and by induction all future iterations. So we can use 6 cycles as the iteration time.

The CPI is then $\frac{6 \mathrm{cyc}}{4 \mathrm{insn}}=1.5 \mathrm{CPI}$.

Problem 2: Appearing below are incorrect executions on the illustrated implementation. For each one explain why it is wrong and show the correct execution.

(a) Explain error and show correct execution.

```
LOOP: # Cycles 
    lw r2, O(r4) IF ID EX ME WB
    add r1, r2, r7 IF ID EX ME WB
LOOP: # Cycles 
```

The add depends on the 1 w through r 2 , and for the illustrated implementation the add has to stall in ID until the 1 w reaches WB.

| LOOP: \# Cycles | 0 | 1 | 2 | 3 | 4 | 5 |  | 6 | 7 | SOLUTION |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| lw r2, O(r4) | IF | ID | EX | ME |  |  |  |  |  |  |
| add r1, r2, r7 |  | IF | ID | - | -> | EX |  | ME | WB |  |
| LOOP: \# Cycles | 0 |  | 2 | 3 | 4 | 5 |  | 6 | 7 |  |

(b) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3
        IF ID EX ME WB
    IF ID -> EX ME WB
lw r1, O(r4)
    0
```

There is no need for a stall because the $1 w$ writes $r 1$, it does not read $r 1$.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    lw r1, O(r4) IF ID EX ME WB
LOOP: # Cycles 
```

(c) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3 IF ID EX ME WB
    sw r1, O(r4) IF ID -> EX ME WB
LOOP: # Cycles 
```

A longer stall is needed here because the sw reads r1 and it must wait until add reaches WB.

```
LOOP: # Cycles 0
    add r1, r2, r3
    sw r1, O(r4)
    IF ID ----> EX ME WB
LOOP: # Cycles
    0
```

(d) Explain error and show correct execution.

```
LOOP: # Cycles 
    add r1, r2, r3 IF ID EX ME WB
    xor r4, r1, r5 IF ----> ID EX ME WB
LOOP: # Cycles 
```

The stall above allows the xor, when it is in ID, to get the value of r1 written by the add; that part is correct.
But, the stall starts in cycle 1 before the xor reaches $I D$, so how could the control logic know that the xor needed r1, or for that matter that it was an xor? The solution is to start the stall in cycle 2 , when the xor is in ID.

```
LOOP: # Cycles 0
    add r1, r2, r3 IF ID EX ME WB
    xor r4, r1, r5 IF ID ----> EX ME WB
LOOP: # Cycles 
```

For this assignment read the ARM Architecture Reference Manual linked to http://www.ece.lsu.edu/ee4720/reference.html. This assignment asks about the ARM A32 instruction set.

Problem 1: Show the encoding of the ARM A32 instruction that is most similar to MIPS instruction add r1, r2, r3.

```
# Arm Add
add r1, r2, r3 # Registers
add Rd, Rn, Rm # Register field symbols.
```

The encoding appears below. The cond field is set to $1110_{2}$ because the instruction should execute unconditionally. The fmt field (name made up) set to zero to indicate a data processing instruction. The opcode and bit position 4 fields are set based on the description of the add instruction. The $S$ is set to 0 because we don't want to write condition code registers (since the MIPS add doesn't either). The instruction should not shift, so we set type and imm5 to zero. The $\mathrm{Rn}, \mathrm{Rd}$, and Rm fields values are based on the register numbers in the example.
cond fmt opcode S Rn Rd imm5 type Rm

| $1110_{2}$ |  | 0 | $0100_{2}$ | 0 |  | 2 |  | 1 |  | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

Problem 2: ARM instructions can shift one of its source operands, something MIPS cannot. With this feature the code below can be executed with a single ARM add instruction. Show the encoding of such an ARM A32 add instruction.

```
sll r1, r2, 12
add r1, r4, r1
```

The assembly language for the ARM A32 equivalent is:
\# Arm assembler
add r1, r4, r2, LSL \#12
The encoding of the ARM instruction appears below. Notice that it is the same as the ordinary add, but with a shift specified by putting a non-zero value in the imm5 field.


Problem 3: So, the ARM add instructions can shift one of its operands, something that MIPS would need two instructions to do. Since we have been working with MIPS for so long it would be natural for us to get protective of MIPS and defensive or jealous when hearing about wonderful features of other ISAs that MIPS doesn't have. To relieve these negative emotions lets add operand shifting to MIPS with a new addsc instruction. The addsc instruction will use MIPS' sa field to specify a shift amount. So instead of, for example, the following two instructions:

```
sll r1, r2, 12
```

```
add r1, r4, r1
```

We could use just
addsc r1, r4, r2, 12
where the " 12 " indicates that the value in $r 2$ should be shifted by 12 before the addition.
Modify our five-stage MIPS implementation so that it can implement this instruction. (See below for diagrams.)

- The addsc should execute without a stall.
- Don't break existing instructions.
- Don't increase the critical path by more than a tiny amount.
- Keep an eye on cost.

Assume that both the ALU and shift unit take most of the clock period. This means if the ALU and shifter are in the same stage and output of the shifter is connected to the ALU, the critical path will be doubled. (Of course, doubling the critical path would be disastrous for performance.)

There are several ways to solve this, one possibility includes adding a sixth stage, another possibility uses a plain adder (not a full ALU) in the EX stage.

Add hardware to the implementation below. Source files for the diagram are at:
http://www.ece.lsu.edu/ee4720/2013/mpipei3.pdf, http://www.ece.lsu.edu/ee4720/2013/mpipei3.eps, http://www.ece.lsu.edu/ee4720/2013/mpipei3.svg. The svg file can be edited using Inkscape.


To see how a shift unit can be added to MIPS see Fall 2010 Homework 3.

One big choice is between adding a stage or adding an adder. If a stage is added then additional pipeline latches are needed, so the decision should be based on the cost of the latches versus the cost of the adder. An additional factor is the number of bypass paths needed. In the solution below the decision was made to add and adder, but in two different ways.

In the first version rsv is added to the EX/ME pipeline latch so that rsv will be available in the ME stage.


In the second version (below) the ALU is used to bring rsv to the ME stage. In this second version, when an addsc instruction is in the EX stage the ALU will perform a Pass $A$ operation in which the upper ALU input is passed to the output unchanged. Also, a multiplexor is saved by using the ME-stage adder to pass results of non-scaled add instructions, this is done by setting the Left Shift unit to output a zero when a non-scaled add instruction is in EX.


Problem 1: The following code fragments execute incorrectly on the following pipeline. For each fragment describe the problem and correct the problem.

(a) Describe problem and fix problem.

```
lwc1 f2, 0(r1) IF ID EX ME WF
add.s f1, f2, f3 IF ID A1 A2 A3 A4 WF
```

There is a dependence between the $1 w c 1$ and the add.s through $f 2$ and $s 0$ the add.s should have stalled. Correct execution appears below for a pipeline in which bypass paths exist from WF into A1 and M1.

```
# SOLUTION 0
lwc1 f2, 0(r1) IF ID EX ME WF
add.s f1, f2, f3 IF ID -> A1 A2 A3 A4 WF
```

(b) Describe problem and fix problem.

```
# Cycle 0
add.s f1, f2, f3 IF ID A1 A2 A3 A4 WF
addi r1, r1, 4 IF ID EX ME WB
lwc1 f2, 0(r1)
```

```
        IF ID EX ME WF
```

```
        IF ID EX ME WF
```

There are two instructions in WF in cycle 6, which is impossible on this pipeline. The solution is to stall 1 wc 1 by one cycle, shown below.

```
# SOLUTION 0
add.s f1, f2, f3 IF ID A1 A2 A3 A4 WF
addi r1, r1, 4 IF ID EX ME WB
lwc1 f2, O(r1) IF ID -> EX ME WF
```

Note that if the addi and the 1 wc1 changed places there would be no reason to stall. However the problem wasn't a stall, the problem was that the hardware should have stalled and didn't, presumably resulting in an incorrect value in either f 2 or f 1 . So swapping the two instruction doesn't fix the problem it just works around (avoids) it.
(c) Describe problem and fix problem.

```
add.d f1, f2, f3 IF ID A1 A2 A3 A4 WF
```

The instruction above is a double-precision add (notice the . d at the end). In MIPS-I and other 32-bit RISC ISAs double precision instructions can only use even-numbered registers as operands. (Each 64-bit operand is obtained from two registers, the even-numbered register and the next register, for example, $f 10$ and $f 11$.

Lets fix this under the assumption that the programmer meant to use a double-precision add but used the wrong registers. In that case make the odd registers even:

```
add.d f10, f2, f4 IF ID A1 A2 A3 A4 WF
```

Problem 2: The code fragment below contains a MIPS floating-point comparison instruction and branch. The pipeline illustrated below does not have a comparison unit, in this problem we will add one. The comparison unit to be used has two stages, named C1 and C2. The output of C2 is one bit, indicating if the comparison was true.

c.gt.d f2, f4
bc1t TARG
add.d f2, f2, f10
...
TARG: xor r1, r2, r3
(a) Add the comparison unit to the pipeline above. Also add a new register FCC (floating point condition code) that is written by the comparison instruction and is used by the control logic to determine if a floating-point branch is taken.

The FCC register should have a data and write-enable input, show the control logic generating the write-enable signal. Show a cloud labeled "branch control logic" and connect it to appropriate datapath components.

## Solution appears below in green. (The solution to part c is in blue.)

The comparison units were added to the two stages before WF . The write enable signal enters the pipeline in M 4 , using a dedicated latch segment named $f w$. The $f w$ signal is set by logic in the ID stage detecting instructions that write the FCC register, including c . (The c instruction cannot use the we signal used by other FP instructions because that is meant for the FP register file.)

The new FCC register is shown in the ID stage, with its output connected to a new branch control logic cloud. The output of that cloud connects to the PC mux.

Notice that there is no stall logic for the cinstruction. That was not asked for in this problem, and it would not be needed if only five-stage instructions could write FCC.

(b) Show the execution of the code sample above on your modified hardware, but without any bypass paths for the added hardware.

Solution appears below. Note that the bc1t instruction must wait in ID until the c instruction reaches WF .

```
# SOLUTION 0
c.gt.d f2, f4 IF ID C1 C2 WF
bc1t TARG IF ID ----> EX ME WB
add.d f2, f2, f10 IF ----> ID A1 A2 A3 A4 WF
TARG: xor r1, r2, r3
IF ID EX ME WB
```

(c) Add whatever bypass paths are needed so that the code executes with as few stalls as possible but without having a major impact on clock frequency. Assume that C2 produces a result in about $80 \%$ of the clock period.

The added bypass path are in sky blue on the diagram above. (The path goes from the output of C 2 to the branch-control-logic cloud.) With this bypass path one fewer stall is needed. That execution is shown below.

```
# SOLUTION 0
c.gt.d f2, f4 IF ID C1 C2 WF
bc1t TARG IF ID -> EX ME WB
add.d f2, f2, f10 IF -> ID A1 A2 A3 A4 WF
TARG: xor r1, r2, r3 IF ID EX ME WB
```

Source files for the diagram are at:
http://www.ece.lsu.edu/ee4720/2014/mpipeifp.eps, http://www.ece.lsu.edu/ee4720/2014/mpipeifp.svg.
The svg file can be edited using Inkscape.

Problem 1: Solve Spring 2013 Final Exam Problem 2, the problem is to design control logic to detect stalls in a 2 -way superscalar system, and to add bypass paths for a special case.

See the posted exam solution at http://www.ece.lsu.edu/ee4720/2013/fe_sol.pdf.
Problem 2: Solve Spring 2013 Final Exam Problem 3, asking an assortment of branch predictor problems. See past final exams for additional problems of this type.
see the posted exam solution at http://www.ece.lsu.edu/ee4720/2013/fe_sol.pdf.

## 49 Spring 2013 Solutions

Problem 1: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.


LOOP:
lw r2, $0(r 4)$
slt r1, r2, r7
bne r1, r0 LOOP
addi $\mathrm{r} 4, \mathrm{r} 4,4$
sw r4, 0 (r6)
jr r31
(a) Show the execution of the code above on the illustrated implementation up to and including the first instruction of the second iteration.

- Carefully check the code for dependencies.
- Be sure to stall when necessary.
- Pay careful attention to the timing of the fetch of the branch target.

Solution appears below. Notice that there is a dependence between the slt and the bne (sometimes people forget that branches can have dependencies too).

```
LOOP: \# Cycles \(\quad 0 \quad 1 \quad 2 x_{1}\)
    lw r2, O(r4)
    slt r1, r2, r7
    bne r1, r0 Loop
    addi \(\mathrm{r} 4, \mathrm{r} 4,1\)
    sw r4, 0 (r6)
    jr r31
```



```
    lw r2, 0 (r4)
```

```
    IF ID EX ME WB
```

    IF ID EX ME WB
    IF ID ----> EX ME WB
    IF ID ----> EX ME WB
    ```
        IF ----> ID ----> EX ME WB
```

        IF ----> ID ----> EX ME WB
            IF ----> ID EX ME WB
            IF ----> ID EX ME WB
        IF IDx
        IF IDx
            IFx
            IFx
                    IF ID EX ME WB
    ```
                    IF ID EX ME WB
```

(b) Compute the CPI for a large number of iterations.

Recall that we define an iteration to start when the first instruction is in IF. In the execution above the first iteration starts in cycle 0 and the second iteration starts in cycle 10, and so an iteration takes 10 cycles. There are four instructions in an iteration, so the CPI is $\frac{10}{4}=2.5 \mathrm{CPI}$.

Problem 2: The code fragment below is the same as the one used in the last problem, but the implementation is different (most would say better).


```
LOOP:
    lw r2, O(r4)
    slt r1, r2, r7
    bne r1, r0 LOOP
    addi r4, r4, 4
    sw r4, O(r6)
    jr r31
```

(a) Show the execution of the code on this new implementation.

- There will still be stalls due to dependencies, though fewer than before.

Solution appears below. Notice that fewer stalls are eliminated than one might have hoped because the load value is available later in the pipeline and because there are no bypasses for the branch.

```
    # SOLUTION
LOOP: # Cycles 0
    lw r2, O(r4)
    slt r1, r2, r7
    bne r1, r0 LOOP
    addi r4, r4, 1
```



```
    lw r2, O(r4)
    IF ID EX ME WB
        IF ID -> EX ME WB
                IF -> ID ----> EX ME WB
        IF ----> ID EX ME WB
        IF ID EX ME WB
```

(b) Compute the CPI for a large number of iterations.

The iteration time is now 7 cycles, and so the CPI is $\frac{7}{4}=1.75 \mathrm{CPI}$.

Problem 3: Consider once again the code fragment from the previous two problems, and the implementation from the previous problem. In this problem consider a MIPS implementation that executes a blt instruction, an instruction that is not part of MIPS. With such an instruction the code fragment from the previous problems can be shortened, and one would hope that the code would take less time to run. In this problem rather than hope we'll figure it out.

(a) Add the additional datapath (non-control) hardware needed to execute blt. Hint: Just add one unit and a few wires.

Solution appears above in green, where in the ID stage a less-than comparison unit was added above the equality unit.
(b) Show the execution of the code on the illustrated implementation up until the second fetch of lw.

Solution appears below. It looks like we saved two cycles by eliminating the slt and the stall it suffered.

```
# SOLUTION
LOOP: # Cycles (0 1 2 3 3 4 4 5 6 6 7 8 8 9 10
    lw r2, O(r4)
    blt r2, r7 LOOP
    addi r4, r4, 4 IF ----> ID EX ME WB
LOOP: # Cycles 0
    lw r2, O(r4)
    sw r4, O(r6)
    jr r31
```

(c) As we discussed in class, doing a magnitude comparison in ID might stretch the critical path, forcing a reduction in clock frequency. Suppose the clock frequency without blt is 1 GHz . At
what clock frequency will the blt implementation, the one in this problem, be just as fast as the implementation from the prior problem on their respective code fragments?

- Be sure to pick a sensible meaning of just as fast. Do not define just-as-fast in terms of CPI.

Most would agree that the important measure of computer performance is how long it takes to finish your program. We have two implementations, the original bypassed MIPS, and the blt version. Lets call the code fragments used in this assignment our programs. The first program in this assignment is for the original MIPS, the program with blt is for the blt version of MIPS. What's important to us is how long it takes to run these programs on their respective implementations.

Lets assume that in a run of either program the loop iterates 1000 times. The original MIPS implementation takes 7 cycles per iteration, for a total of 7000 cycles, and that corresponds to a time of $t_{\text {original }}=\frac{7000 \mathrm{cycles}}{\phi_{\text {orig }}}=\frac{7000 \mathrm{cycles}}{1 \mathrm{GHz}}=$ $7 \mu \mathrm{~s}$, where $\phi_{\text {orig }}=1 \mathrm{GHz}$ is the clock frequency of the original implementation.

In the blt version of MIPS an iteration takes only 5 cycles and so execution takes 5000 cycles. Execution time is $t_{\mathrm{blt}}=\frac{5000 \text { cycles }}{\phi_{\mathrm{b}}}$, where $\phi_{\mathrm{blt}}$ is the clock frequency of the blt version. For this problem we need to find a value of $\phi_{\mathrm{blt}}$ that will make the execution time of the blt version $7 \mu \mathrm{~s}$. That is we need to solve $\frac{5000 \text { cycles }}{\phi_{\mathrm{blt}}}=7 \mu \mathrm{~s}$, for $\phi_{\mathrm{blt}}$, which is $\phi_{\mathrm{blt}}=714 \mathrm{MHz}$.

This means that if the hardware needed to implement blt slows down the clock frequency, but the clock frequency is still $>714 \mathrm{MHz}$ the blt implementation will be faster.

Note that one cannot just look at something like CPI, since that would ignore the fact that the two different programs execute a different number of instructions.
(d) Explain why the code fragments in these problems might exaggerate the benefit of the blt instruction.

The code fragment was short, could use a blt, and the blt reduced execution by two cycles rather than the one cycle it would for other cases. If we look at other code samples we will probably find that a blt is usable less frequently than every five instructions, in part because not every branch is based on a magnitude comparison (see the example below).

That means the actual reduction in the number of cycles will not be as great as 7 to 5 . Suppose the number of clock cycles drops from 100 trillion to 98 trillion. In that case, we can tolerate a much smaller drop in clock frequency and still get better performance

Grading Notes: Many answered that the beneffit is exaggerated because the clock frequency would be lower. That is the right answer to a different question. This question asked about the code fragments in these problems. The lower clock frequency would affect any code.

```
LOOP:
    lw r2, O(r4)
    beq r2, r7 LOOP # blt won't help here.
    addi r4, r4, 4
    sw r4, O(r6)
    jr r31
```

Note: For help with this and similar assignments see the Statically Scheduled MIPS study guide linked to http://www.ece.1su.edu/ee4720/guides.html.

Problem 1: Solve Spring 2012 Midterm Exam Problem 1. Part a is the usual draw-a-pipeline-execution-diagram-and-find-the-CPI problem, but it's on an implementation with some bypass paths removed. For part b you need to design control logic to generate stalls for the missing bypasses.

Problem 2: Solve Spring 2012 Midterm Exam Problem 2. In that problem the memory stage is split in two.

Problem 1: As described in class, SPARC v7 integer branch instructions use a 22 -bit immediate field for the displacement. Branches are typically used in loops and if/else constructs, and so the $\pm 2097152$ instruction range might be more than is needed. So did the computer engineers at Sun Microsystems (now part of Oracle). Look up the v7 integer branch instruction in the SPARC Joint Programming Specification (JPS1), linked to the course references page (look for JPS1). You'll find SPARC v7 integer branch under Instruction Definitions in the Deprecated Instructions section. Then look up the replacement integer branch instructions (not in the deprecated section).
(a) Sketch (or cut-and-paste, take a picture with your cell phone, etc.) the format of the three instructions (one old, two new).

The formats appear below. The dhi and dlo in the last format refer to the 16 -bit displacement, split across two fields.

Op a cond op2 disp22
Bice:


Op a reond op2 dhi $P$ rs1 dlo

BPr:

| 00 |  | 0 |  |  | 011 |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

(b) Describe how BPr is different than the original v7 integer branch instruction, and point out two benefits.

The BPr instruction can test the value of a register (the one specified by the rs1 field), avoiding the need for an instruction like subcc to set the condition code register. It also provides, in field p , a hint to the hardware on whether the branch will be taken. The hardware may choose to ignore the hint (relying on its own branch predictor). If the hardware uses the hint and it is wrong performance will be slightly lower than if the hint were right.

Grading Note: Many incorrectly answered that with BPcc and Bicc the condition setting instruction would have to immediately precede the branch. That's not true, the condition setting instruction, such as subcc, can be placed far away from the branch.
(c) Describe how BPcc is different than the original v7 integer branch instruction. This instruction shares one benefit with BPr , but it has lost 2 bits of displacement in order to accommodate 64 -bit register values. (The other third lost bit has nothing to do with 64 -bit register values). Explain.

The shared benefft is the prediction hint, using a bit in the p field. The two "lost" bits are for the cc 1 and cc 0 fields. These specify which condition code register to use. There are two integer condition code registers, icc and xcc. The icc is set based on the low 32 bits of a result (and so ignore the upper 32 bits), while the xcc is set based on the full 64 bits. The icc register is need for code compiled for SPARC v7, which used 32-bit registers. An arithmetic operation might overflow a 32 -bit register, and SPARC v7 code would expect that overflow. If run on v9, the operation would not cause an overflow. The overflow ( V ) bit on the xcc register would not be set, but the V bit on the icc would be set. So, when developing a 64 -bit version of the SPARC ISA, Sun engineers added those cc1 and cc0 bits to maintain compatibility. (They could have used just one bit, but two bits were used so that BPcc would be similar to the floating-point version, FBPfcc, which does need two bits because their are 4 FP condition code registers.)

Problem 2: For the following assignment familiarize yourself with the VAX ISA by looking in the VAX-11 Architecture Reference Manual (linked to the course references page). In particular, see Section 2.6 for a summary of the instruction format, and Chapter 3 for details on the operand specifiers used in the instruction formats. For examples, look at some past homework assignments in this course: http://www.ece.lsu.edu/ee4720/2010f/hw04_sol.pdf,
http://www.ece.lsu.edu/ee4720/2007f/hw03_sol.pdf, and http://www.ece.lsu.edu/ee4720/2002/hw02_sol.pdf.

The VAX format is simple, it consists of a one- or two-byte opcode followed by some number of operand specifiers and any additional fields they may use. The operand specifiers are 8 bits, and are followed by a possible extension and immediates. (See Section 2.6 and Chapter 3 of the VAX-11 Architecture Reference Manual.)
(a) The VAX operand specifier is 8 bits, it includes a 4 -bit mode field, and for literal addressing, a 6 -bit literal field. (A literal in VAX is a small immediate.) Explain how it's possible to fit a 4-bit mode field and a 6 -bit literal field into 8 bits.

The two least-significant bits of the mode field is used for part of the literal. Therefore literal mode actually uses four mode values, 0 through 3.
(b) Find the best VAX replacement for each of the two MIPS instructions below and show their encoding. The two VAX instructions will be different.

The VAX instructions appear right after the corresponding MIPS instruction, followed by the VAX instruction coding. For the first MIPS instruction a 3-operand VAX add was used. But for the second, a VAX increment (INCL) instruction was used, which requires only a single operand, the register number. The INCL instruction is one byte shorter than an ADDL2 which needs a one-byte specifier for the immediate value, 1.

```
addi r1, r2, 1 # MIPS
ADDL3 r2, S^#1, r1 # VAX (destination register is last)
```



```
addi r1, r1, 1 # MIPS
INCL r1 # VAX
\begin{tabular}{|c|c|c|}
\hline Opcode & mode & reg \\
\hline 0xd6 & 5 & 1 \\
\hline 70 & 74 & 30 \\
\hline
\end{tabular}
```

(c) Find a VAX instruction to replace the following sequence of MIPS instructions, and show its encoding.

```
lw r1, O(r2)
lui r3, 0x2abb
ori r3, r3, 0xccdd
add r1, r1, r3
sw r1, 0x100(r2)
```

Solution appears below. Note that the load performed by the first MIPS 1 w instruction uses VAX register deferred mode, rather than displacement mode, since the displacement is zero. The VAX encoding shown below on several lines, with one line for the opcode and one line for each operand.

```
# SOLUTION
ADDL3 (r2), I^#0x2abbccdd, W^0x100(r2) # Destination is last.
```

Opcode

(d) Compare the size of the VAX instruction from the problem above to the size of the MIPS instructions.

There are five MIPS instructions, for a total size of 20 bytes. The single VAX instruction is 10 bytes, half the size.

Problem 1: Recall that the MIPS-I mult instruction reads two integer registers and writes the product into registers hi and lo. To use the product the values of lo and hi (if needed) have to be moved to integer registers, done using a move from instruction such as mflo. In this problem these instructions will be added to the implementation below.


Consider an integer multiply unit that consists of two stages, Y1 and Y2. The inputs to Y1 are the 32 -bit multiplier and multiplicand, and the output of Y2 is the 64 -bit product. Unit Y1 has three 32 -bit outputs named $\mathbf{s} 0$, s 1 , and $\mathbf{s} 2$; unit Y2 has 332 -bit inputs of the same name. As one would guess, the data from the s0 output of Y1 should be sent to the s0 input to Y2, likewise for s1 and s2.

As with other functional units, such as the ALU, inputs to Y1 and Y2 must be stable near the beginning of the clock cycle and the outputs must be stable near the end of the clock cycle. There is enough time to put a multiplexer before the inputs, or after the outputs (but not both).

Solve the two parts below together. That is, the hardware for part (a) might take advantage of the hardware for part (b) and vice versa.
(a) Add the datapath hardware needed to implement the mtlo, mthi, mflo, and mfhi instructions. Both the ALU and the integer multiply unit have an operation to pass either input to its output unchanged. That is, let $x$ denote the ALU output and let $a$ and $b$ its inputs. In addition to operations like $x=a+b$ and $x=a \& b$, the ALU can also perform a pass- $a$ operation, that is, $x=a$ and a pass- $b$ operation, $x=b$. The integer multiply unit also has pass- $a$ and pass- $b$ operations.

- Put the hi and lo registers in the ID stage.
- Do not write the hi and lo registers earlier than the ME stage.
- As always, cost is a criteria.
- Bypass paths will be added in the parts below.

Solution appears in blue above. The mflo and mfhi instructions, using a new ID-stage multiplexor, route the hi or lo value to the existing (though renamed) ID/EX.ihl pipeline latch, where it can easily take a path through the ALU (using a pass b operation) and continue on to write back the integer register file.

The mtlo and mthi use the ordinary multiply unit inputs (see the next problem), but the multiply unit uses a pass a (since the register is in the rs field). The multiply unit would need to have two versions of pass a, once to pass to the lower 32 bits of its output, and one to pass to the upper 32 bits. The control logic would also have to enable the appropriate register (lo or hi).
(b) Add the datapath hardware needed to implement the mult instruction. That is, put the Y1 and Y2 units in the appropriate stages, and connect them to the appropriate pipeline latch registers (adding new ones where necessary).

- Don't add new bypass paths, but take advantage of what is available.

Solution appears in blue, where Y1 is placed in EX and Y2 in ME. Notice that the multiply unit takes advantage of the multiplexors at the ALU's inputs. Also notice that the writeback occurs in the WB stage, but that the outputs connect directly to the hi and lo registers.

Because the output of the multiply unit is written to a fixed register pair the data can arrive at those registers, hi and lo, close to the end of the cycle. This is different to the writeback of the integer (general-purpose) register file, where one of 31 registers might be written and bypassing might also be performed and so more time is needed. For that reason, it might be possible to write back hi and lo in the ME stage, and such an answer did not loose points.
(c) Show the execution of the code below on your hardware so far. That is, your hardware should not have any new bypass paths, but existing bypass paths in the implementation can be used.

Solution appears below. Notice that there is a dependence between sub and mult, but that it can be handled by the bypass paths shared with the ALU and so the mult instruction does not stall. There is also a dependence between the mult and the mflo. Since there are no bypass paths for that, the mflo must stall two cycles.

```
# SOLUTION
# Cycle 0
sub r2, r6, r7 IF ID EX ME WB
mult r1, r2 IF ID Y1 Y2 WB
mflo r3 IF ID ----> EX ME WB
add r4, r3, r5 IF ----> ID EX ME WB
```

(d) Add bypass paths so that the code below (which is the same as in the previous part) can execute without a stall. Assume that an additional multiplexer delay is tolerable.

A bypass path has been added from WB to ME , the appears in green in the diagram. Notice that this bypass path leads to another bypass path, and so the add instruction receives the correct value of r4.

Grading Note: Some solutions had a bypass from WB to EX. Such a bypass would be from the mult instruction to the add, which are not directly dependent. To be completely correct such solutions would have to explain how the control logic can detect such a bypassing opportunity.

```
# SOLUTION
# Cycle 
sub r2, r6, r7 IF ID EX ME WB
mult r1, r2 IF ID Y1 Y2 WB
mflo r3 IF ID EX ME WB
add r4, r3, r5 IF ID EX ME WB
```

Problem 2: Continue to consider the implementation of the MIPS-I mult instruction. If MIPS designers thought that an integer multiply unit could be built with two stages they might not have used special registers, hi and lo, for the product.
(a) Show how the pipeline would look if the multiply unit had three stages, Y1, Y2, and Y3. There is no need to add bypass paths for this part.

Solution appears in orange where a Y3 stage has been added in WB and a fifth pipeline latch has been added. Because the multiply unit writes a fixed register (as opposed to the register file) the timing constraints for the writes are less severe and so the added pipeline latch might not be necessary. If would be necessary if the physical distance between the multiplier output and the ID stage were large.
(b) Explain why there is much less of a need for the hi and lo registers with a two-stage multiply unit (the first problem) than with a three-stage unit (this problem).

One reason for having special hi and lo registers is to avoid the structural hazard during writeback. Consider the example below for the three-stage multiply unit, in which wy indicates the stage in which mult writes back. Both the mult and add instruction writeback in cycle 5. But because mult is writing back into its special registers there is no conflict. If mult wrote to the general-purpose registers there would have to be two write ports on the GPR file, which would be more costly than having the two special registers.

With a two-stage multiply unit, the multiply instruction can write back at the same time as other instructions, so there would not be a need for a second write port. There is still the problem of the multiply writing back 64 bits. The expensive solution is to widen the write port to 64 bits (and perhaps write a pair of registers, as is done for floating point). Another possibility is to have just one special register, for the high 32 bits.

```
# SOLUTION: Code execution for the three-stage multiply unit.
# Cycle 0
mult r1, r2 IF ID EX ME Y3 WY
add r4, r5, r6 IF ID EX ME WB
```

Homework 5 solution
Due: 27 March 2013

Problem 1: In the following execution of MIPS code the 1 l instruction raises a TLB miss exception and the handler is called. A TLB miss is not an error, it indicates that the TLB needs to be updated, which is what the handler will do.

Execution is shown up to the first instruction of the handler. Alert students will recognize that there is something wrong in the execution below: it shows the execution of a deferred exception for an instruction, the 1 w , that should raise a precise exception.

```
# Cycle }\quad0\quad
    sh r1, O(r3) IF ID EX ME WB
    lw r1, O(r2) IF ID EX M*x
    addi r2, r2, 4 IF ID EX ME WB
    sw r7, O(r8) IF ID EX ME WB
    and r4, r1, r6 IF ID EX ME WB
    or r10, r11, r12
HANDLER:
# Cycle 
    sw r31,0x100(r0) IF ID EX ME WB
    ... # Additional handler code here.
    eret
```

(a) Show the execution of the eret instruction and the instructions that execute after the eret. Assume that eret reaches IF in cycle number 100. The execution should be for a deferred exception, even though memory instruction exceptions should be-must be-precise. A correct solution to this part will result in incorrect execution of the code.

Solution appears below. Note that the cycle after 100 is 101 , but is written as 1 to save space.
In a deferred exception the handler starts several instructions after the faulting instruction. In the example above the addi, sw, and and execute before the handler starts. The return point would then be after the and instruction, that is what is shown below.

Also, notice that eret does not have a delay slot.
The solution below assumes that there is a connection from the register file (actually coprocessor set 0 , which contains the exception return address) to the IF-stage multiplexor. That would enable the first user instruction to reach IF when eret is in EX.


```
    sh r1, O(r3) IF ID EX ME WB
    lw r1, O(r2) IF ID EX M*x
    addi r2, r2, 4 IF ID EX ME WB
    sw r7, O(r8) IF ID EX ME WB
    and r4, r1, r6 IF ID EX ME WB
    or r10, r11, r12 IF ID EX ME WB SOLUTION
```

HANDLER:
$\begin{array}{llllllllllllllllllllll}\text { \# Cycle } & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \ldots & 100 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8\end{array}$

```
sw r31,0x100(r0) IF ID EX ME WB
```

... \# Additional handler code here.

(b) Suppose the execution above is for a computer on Mars, meaning that there is no fast or cheap way of replacing the hardware, and there is no way to turn on precise exceptions for the lw. Happily, it is possible to re-write the handler. Explain what the handler would have to do so that the code above executes correctly. The handler will know the address of the faulting instruction. Optional: explain why the sw r7 is nothing to worry about, at least in the execution above.

Because the 1 w did not make it to writeback, r1 and $r 4$ will have incorrect values when the handler starts. Re-write the handler so that it puts the correct values in $r 1$ and $r 4$ and then returns to the or instruction (as it might for a deferred exception).

The handler will update the TLB, as the original handler did. But then it will load the word from memory that the lw would have loaded, using address $r 2-4$, and put it in $r 1$. and then recompute $r 4$. The sw $r 7$ would only be a problem if it wrote the same address as the 1 w , making it impossible to retrieve the prior word at that address. However, if the addresses were the same the sw r7 would also raise an exception, so they must be different. After this, execution can return to the or and continue as though nothing happened.
(c) Show the execution of the code above, but this time for a system in which 1 w raises a precise exception. Start at cycle 0 with the sh instruction, and have the lw raising once again a TLB miss exception. The execution should be in two parts, first from the sh up to the first instruction of the handler, then jump ahead to cycle 100 with eret in IF and continue with whatever instructions remain.

Solution appears below. Since the exception is precise the faulting instruction $* l^{w}$ ) and those that follow it are squashed and all instructions before the faulting instruction finishes normally. The handler has the option of re-executing the faulting instruction or skipping it. For a TLB miss the usual practice is to re-execute it.

SOLUTION


HANDLER:


Problem 2: Solve Spring 2012 Final Exam Problem 2, which asks for the execution of MIPS floating-point instructions on our FP implementation.

See the posted final exam solution.
Problem 3: Solve Spring 2012 Final Exam Problem 1 (yes, this is out of order). In this problem parts of the FP multiply unit are used to implement the MIPS integer mul instruction. Note that the mul writes integer registers, unlike mult which writes the hi and lo registers. In other words, do not use hi and lo registers in your solution.

See the posted final exam solution.

SVG and EPS versions of the superscalar processor illustration are available at http://www.ece.lsu.edu/ee4720/2013/mpipei3ss.svg and
http://www.ece.lsu.edu/ee4720/2013/mpipei3ss.eps, respectively. Inkscape can be used to edit the $S V G$ version.

Problem 1: The two-way superscalar implementation below has two memory ports in the ME stage, and so it can sustain an execution of 2 IPC on code containing only load and store instructions. Since for many types of programs loads and stores are rarely so dense and because memory ports are costly, it is better to make a 2-way processor with just one memory port in the ME stage.

Modify the implementation below so that it has just one memory port in the ME stage. It should still be possible to execute arbitrary MIPS programs, albeit more slowly.

Solution on next page.


Solution appears below, original design above. One of the memory ports was removed, and multiplexors were used to provide paths from either slot to the Addr and D In inputs to the memory ports, these are shown in blue. These are placed in the EX stage (rather than ME ) under the assumption that the memory port takes the most time and so anything added between its connections and the pipeline latches would lower the clock frequency. Also note that the output of the memory port connects to the multiplexor for each slot.


Problem 2: The datapath hardware for resolving branches in the 2-way superscalar MIPS implementation below is incomplete: it does not show branch target computation for the instruction in Slot 1 (it is shown for Slot 0). (The hardware for determining branch conditions is also not shown, but that's not part of this problem.) The IF-stage memory port can retrieve any 4 -byte aligned address. (That is, it does not have the stricter 8 -byte alignment that is assumed by default for 2 -way superscalar processors presented in class.)

(a) Add hardware so that the correct branch target is computed for branches in either slot. The following signals are available: br_slot_0, which is 1 if the instruction in Slot 0 (ir ${ }^{0}$ ) is a branch; br_slot_1, which is 1 if the instruction in Slot 1 is a branch. Assume that there will never be a branch in both Slot 0 and Slot 1.

Design the hardware for low cost. Hint: Adder carry-in inputs can come in handy. The goal of this part is to generate the correct target address, the next part concerns what is done with it.

Solution appears above in blue and green. Multiplexors have been placed before both adder inputs, and the br_slot_1 signal is used as a control input. One mux provides the corresponding displacement value. The other provides either pc or npc. For the branch in slot 0 we need to compute pc + immed0 +4 , so the upper input to the mux is pc and the carry in bit is set to 1 . (Note that the new ID.pc latch holds the address of the instruction in slot 0.) For the branch in slot 1 we need to compute pc_1 + immed1 + 4, where pc_1 is the address of the instruction in slot 1, but pc_1 is not available anywhere. However, npc is pc_0+8 = pc_1 + 4. So for a branch in slot 1 we compute npc + immed1 (in part by setting the carry in to 0 ).
(b) Add datapath so that the branch target address can be delivered to the PC at the correct time, whether the branch is in Slot 0 or Slot 1. (Earlier in the semester branch delay slots were given as an example of an ISA feature that worked well for the first implementations but that would become a burden in future ones. Welcome to the future.)

No datapath needs to be added. The question should have asked for control logic to identify instructions to squash.

Problem 1: Solve Spring 2012 Final Exam Problem 3, in which pipeline execution diagrams are requested for some superscalar systems.

See the Spring 2012 Final Exam solution at http://www.ece.lsu.edu/ee4720/2012/fe_sol.pdf.
Problem 2: Solve Spring 2012 Final Exam Problem 6 (d) and (e). (Just those two.) These questions concern the techniques of widening (superscalar designs) and deepening (more pipeline stages) our implementation to exploit more instruction-level parallelism.

See the Spring 2012 Final Exam solution at http://www.ece.lsu.edu/ee4720/2012/fe_sol.pdf.
Problem 3: Solve Spring 2012 Final Exam Problem 4, which asks for performance information about some branch predictors.

See the Spring 2012 Final Exam solution at http://www.ece.lsu.edu/ee4720/2012/fe_sol.pdf.

50 Spring 2012 Solutions

Problem 1: To save space in a program an array is designed to hold four-bit unsigned integers instead of the usual 32 -bit integers (it is known in advance that their values are $\in[0,15]$ ). Because this 4 -bit data size is less than the smallest MIPS integer size, 8 -bits, even a load byte instruction will fetch two array elements. Code to read such an array and a test routine appear on the next page, along with a stub for code to write the array. The routine compact_array_read is used to read an element of this array and compact_array_write is the start (mostly comments) of a routine to write an element.
(a) Add comments to compact_array_read appropriate for an experienced programmer. The comments should describe how instructions achieve the goal of reading from the array. The comments should not explain what the instruction itself does, something an experienced program already knows. See the test code for examples of good comments. Note: The code in the original assignment had a bug: lb should have been lbu.

See comments below.
(b) Complete the routine compact_array_write, so that it writes data into the array. See the comments for details.

Solution appears below.

```
###############################################################################
##
## Test Code
##
a: # Array of values to test. Each byte hold two 4-bit elements.
        .byte 0x12, 0x34, 0x56
msg: # Message format string (similar to printf).
        .asciiz "Value of array element a[%/s0/d] is 0x%/s3/x\n"
        .text
        .globl __start
__start:
    addi $s2, $0, 4 # Last index in array a.
    addi $s0, $0, 0 # Initialize loop index.
LOOP:
    la $a0, a # First argument, address of array.
    jal compact_array_read
    addi $a1, $s0, 0 # Second argument, index of element to read.
    la $a0, msg # Format string for test routine's msg.
    addi $s3, $v0, 0 # Move return value (array element) ...
    addi $v0, $0, 11 # ... out of $v0 and replace with 11 ...
    syscall # ... which is the printf syscall code.
    bne $s0, $s2 LOOP
    addi $s0, $s0, 1 # Good Comment: Advance index to next
    # element of test array.
    # Bad Comment: Add 1 to contents of $s0.
    li $v0, 10 # Syscall code for exit.
    syscall
```

```
#################################################################################
##
## compact_array_read
##
compact_array_read:
    ## Register Usage
    #
    # CALL VALUES
    # $a0: Address of first element of array.
    # $a1: Index of element to read.
    #
    # RETURN VALUE
    # $v0: Array element that has been read.
    #
    # Element size: 4 bits.
    # Element format: unsigned integer.
```

    \#\# SOLUTION
    srl \$t0, \$a1, 1 \# Scale array index to byte offset.
    add \$t1, \$a0, \$t0 \# Compute address of element.
    andi \$t3, \$a1, 1 \# Determine if loading upper or lower 4 bits.
    bne \$t3, \$0 SKIP
    lb \$t2, 0 (\$t1) \# Load a pair of elements.
    jr \$ra \# Return for upper-4-bits case.
    srl \$v0, \$t2, 4 \# Move upper 4 bits into position.
    SKIP: jr \$ra \# Return for lower-4-bits case
andi \$v0, \$t2, 0xf \# Just want lower bits, so mask off rest.

```
###############################################################################
#
# compact_array_write
#
compact_array_write:
    ## Register Usage
    #
    # CALL VALUES
    # $a0: Address of first element of array.
    # $a1: Index of element to write.
    # $a2: Value to write.
    #
    # RETURN VALUE
    # None.
    #
    # Element size: 4 bits.
    # Element format: unsigned integer.
    srl $t0, $a1, 1
    add $t1, $a0, $t0
    andi $t3, $a1, 1
    lb $t2, 0($t1)
    bne $t3, $0 SKIPw
    andi $a2, $a2, 0xf # Make sure that write value is 4 bits.
    # Even element, put $a2 value in bits 7-4.
    #
    andi $t2, $t2, 0xf # Write zeros in bits 31-4 (keep only 3-0)
    j FINISH
    sll $a2, $a2, 4 # Put write value in correct position.
SKIPw:
    # Odd element, put $a2 in bits 3-0.
    #
    andi $t2, $t2, 0xf0 # Write zeros everywhere except bits 7-4.
FINISH:
    or $t2, $t2, $a2 # Combine write value with value already present.
    jr $ra
    sb $t2, 0($t1)
```

Problem 2: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles. The register file bypasses data from the write ports to the read port in the same cycle.


```
LOOP: # 1st Iter 0
    srl r4, r3, 2 IF ID EX ME WB
    sw r4, 0(r3) IF ID ----> EX ME WB
    bne r4, r2 LOOP IF ----> ID EX ME WB
    addi r3, r3, 4 IF ID EX ME WB
    nop IF ID EXx
    nop IF IDx
# Second Iteration 0
    srl r4, r3, 2 IF ID EX ME WB
    sw r4, 0(r3) IF ID ----> EX ME WB
    bne r4, r2 LOOP IF ----> ID EX ME WB
    addi r3, r3, 4 IF ID EX ME WB
# Third Iteration 0}1
    srl r4, r3, 2 IF
```

(a) Show a pipeline execution diagram for the code above on the illustrated implementation for enough iterations to determine CPI.

Solution appears above. The implementation lacks bypass paths, forcing the store to stall two cycles, waiting for the result of the srl. Also notice that in this implementation the branch resolves in ME and so the branch target is not fetched until the branch is in WB. Grading Note: In most submissions the branch target was fetched one cycle too early.

The second iteration starts at cycle 8 , the third at cycle 16 . The state of the pipeline is identical in cycles 8 and 16 (addi in ME and bne in WB ), and so the third iteration will execute identically to the second. The time for the second iteration is $16-8=8$ cycles, so the third and subsequent iterations will be 8 cycles. The CPI is then $\frac{8}{4}=2 \mathrm{CPI}$.

## 51 Spring 2011 Solutions

Problem 1: The MIPS code below executes on the illustrated implementation. The loop iterates for many cycles.


```
# SOLUTION
    lw r2, O(r5) IF ID EX ME WB
LOOP: # Cycles 0
    lw r1, 0(r2) IF ID -> EX ME WB FIRST ITERATION
    lw r3, O(r1) IF -> ID -> EX ME WB
    sw r3, 4(r2) IF -> ID -> EX ME WB
    bne r3, r0 LOOP IF -> ID EX ME WB
    addi r2, r3, 0
# Cycles
    1w r1, O(r2) SECOND ITERATION IF ID EX ME WB I
    lw r3, O(r1) IF ID -> EX ME WB
    sw r3, 4(r2)
    bne r3, r0 LOOP
    addi r2, r3, 0
# Cycles 
    lw r1, 0(r2) THIRD ITERATION IF ID
```

(a) Show a pipeline execution diagram for enough iterations to determine the CPI. Compute the CPI for a large number of iterations.

Pipeline diagram appears above. Note that execution is shown to the start of the third iteration. That was necessary to insure that a repeating pattern has been established, meaning that the state of the pipeline was the same in consecutive iterations. The first iteration starts at cycle 1 (by definition with the fetch of the first instruction of the loop), the pipeline
has the first instruction in IF and a non-loop instruction in ID. The second iteration starts in cycle 9, there the ID stage has a different instruction than in the first iteration. The third iteration starts in cycle 16 , the stage contents here are the same as the start of the second iteration, addi in ID, bne in EX, and sw in ME. Therefore whatever happens in the second iteration will happen in the third iteration, and all following iterations (so long as the loop branch is taken).

The second iteration takes $16-9=7$ cycles, so the CPI is $\frac{7}{5}=1.4$.
(b) Show when each bypass path is used. Do so by drawing an arrow to a multiplexor input and labeling it with the cycles in which it was used and the register. For example, something like $\mathrm{C} 10 / \mathrm{r} 9 \longrightarrow$ to show that the input is used in cycle 10 carrying a value for r 9 .

The labels are shown in the diagram above in blue.
Problem 2: Continue to consider the pipeline and code from the previous problem. The store instruction and the branch could both benefit from a new bypass connection.
(a) Show a new bypass connection for the store.

The store needs the value of the preceding load. That's available too late for the bypass connection in the EX stage. A new bypass has been added to the ME stage, that is shown in green.
(b) Indicate the impact of the new store bypass connection on critical path length.

The memory port is assumed to be on the critical path, however it would be reasonable to assume that it's the address input and data output that are critical. If so, the added multiplexor would not increase eritical path.
(c) Show a new bypass connection needed by the branch.

Bypass needed from ME to the comparison unit in ID. Though it is impractical, it's shown in red.
(d) Indicate the impact of the new branch bypass connection on critical path length.

The output of the memory port will not be available until the very end of the cycle, so this bypass would certainly lengthen the critical path and so should not be added.
(e) Suppose that the cost of the two bypass connections were equal and that both had no critical path impact. If only one could be added to an implementation which would you add? Base your answer not on the example code above, but on what you consider to be typical programs.

The branch bypass. Branches occur frequently and its reasonable that it would use a value loaded by a nearby instruction. The store bypass, since it's only useful when the bypassed value is from an immediately preceding load, would only be useful for programs that are copying data from one area of memory to another, and there are ways of separating such load/store pairs.

## 52 Fall 2010 Solutions

Problem 1: Diagnose or fix the MIPS-I problems below.
(a) Explain why the code fragment below will not complete execution. Fix the problem, assuming that the load addresses are correct. (Problems such as this occur when operating on data prepared on a different system.)

```
lw r1, O(r2)
lw r3, 6(r2)
```

MIPS Ioads and stores must be to aligned addresses, meaning that the address must be a multiple of the data size. In this case the data size is 4 (because the instructions are 1 w ). Because they both use the same base register, r 2 , at most one of the load addresses can be a multiple of 4.

The code below fixes the problem by using 1 b instead of 1 w and sliding the bytes into the respective destination registers. A faster solution is possible: based on the two least-significant bits of r2 and branch to one of four routines. (For example, if the two-least significant bits were zero then 1 w r1, 0 (r2) would work and two 1 hu could be used for 6(r2).)

```
# Solution
    addi r4, r2, 4
LOOP:
    sll r1, r1, 8
    lbu r11, O(r2)
    or r1, r1, r11
    sll r3, r3, 8
    lbu r13, 6(r2)
    or r3, r3, r13
    bne r2, r4 LOOP
    addi r2, r2, 1
```

(b) The code below will execute, but it looks like there might be a bug. Explain.

```
jal subroutine
add r31, r0, r0
```

The jal instruction writes the return address in register r31, but the instruction in the delay slot, which is executed immediately after the $j a l$, overwrites $r 31$. If the programmer didn't care about the return address then a $j$ instruction would be used, so the code above probably has a bug.
(c) The two fragments below are almost but not quite MIPS-I. Re-write them using MIPS instructions so they accomplish what the programmer likely intended.

```
# Fragment 1
lw r1, O(r2+r3)
# Fragment 2
bgti r1, 101 target
nop
```

```
# Solution - Fragment 1
#
# MIPS does not have a load that uses two source registers.
add r1, r2, r3
lw r1, O(r1)
# Solution - Fragment 2
#
# MIPS branches cannot perform magnitude comparisons (gt between two
# registers) nor can they compare to an immediate (the 101).
slti r2, r1, 102
bne r1, r0 target
nop
```

(d) The code fragments below are correct, but not as efficient as they could be. Re-write them using fewer instructions (and without changing what they do).

```
# Fragment 1
addi r1, r0, 0xaabb
sll r1, r1, 16
ori r1, r1, 0xccdd
# Fragment 2
add r1, r0, r0
addi r1, r1, 123
# Fragment 1 - Solution
#
# Hel-llow, lui is in the instruction set for a reason!
lui r1, 0xaabb
ori r1, r1, 0xccdd
# Fragment 2 - Solution
#
# Too much exposure to accumulator-style ISAs can ingrain bad habits.
# In particular a register does not need to be cleared before it is
# written.
addi r1, r0, 123
```

Problem 1: Consider the execution of the code fragments below on the illustrated implementation.


- A value written to the register file can be read from the register file in the same cycle. (For example, if instruction $A$ writes r1 in cycle $x$ (meaning $A$ is in WB in cycle $x$ ) and instruction $B$ is in ID in cycle $x$, then instruction $B$ can read the value of r 1 that $A$ wrote.)
- As one should expect, the illustrated implementation will execute the code correctly, as defined by MIPS-I, stalling and squashing as necessary.

```
# SOLUTION Execution on the resolve-in-ME (illustrated) pipeline
```



```
lw r3, O(r1) IF ID EX ME WB FIRST ITERATION
add r4, r4, r3
bne r1, r2 LOOP
addi r1, r1, 4
    xor r7, r8, r3
sw r4, 16(r5)
LOOP: # Cycles 
    lw r3, O(r1)
    SECOND ITERATION IF ID EX ME WB
    add r4, r4, r3
    bne r1, r2 LOOP
    addi r1, r1, 4
    xor r7, r8, r3
    sw r4, 16(r5)
LOOP: # Cycles (0)1
    lw r3, O(r1)
```

(a) Show a pipeline execution diagram for this code running for at least two iterations.

Solution appears above.
Grading Note: A common mistake this semester was counting the squashed instructions, xor and sw, in the formula for the CPI. The CPI is a measure of performance, so it does not make sense to count instructions that were not supposed to be executed.

- Carefully check the code for dependencies, including dependencies across iterations.
- Base timing on the illustrated implementation, pay particular attention to how the branch executes.
(b) Find the CPI for a large number of iterations.

The iteration start times of the first three iterations (based on the IF of the first instruction) are 0,8 , and 16 . The first iteration and the second iteration each take 8 cycles. The states of the pipeline at the start of the second and third iterations are identical ( 1 w in ID , addi in ME , etc.) and therefore the third iteration will take the same amount of time as the second. Therefore we can safely say that there are 8 cycles per iteration. Since there are four instructions in the loop the CPI is $\frac{8}{4}=2$.
(c) How much faster would the code run on an implementation similar to the one above, except that it resolved the branch in EX instead of ME? Explain using the pipeline execution diagram above, or using a new one. An answer similar to the following would get no credit because "should run faster" doesn't say much: A resolution of a branch in EX occurs sooner than ME so the code above should run faster.. Be specific, and base your answer on a pipeline diagram.

With the branch resolved in EX rather than ME the target would be fetched while the branch is in ME rather than WB, that's one cycle earlier. Sounds good so far. But let's not be hasty, let's do a pipeline diagram, that's shown below.

The diagram shows that the second iteration starts one cycle earlier than before, but there is also a stall in the 1 w because of the dependence with the addi. Because of that stall the execution is no faster. The first iteration takes just 7 cycles, but the second takes 8 , and so will subsequent iterations. So it's no faster.


Problem 2: Apologies in advance to those tired of the previous problem. Consider the execution of the code below on the implementation from the last problem. The code is only slightly modified.
(a) Show a pipeline execution diagram for this code, and compute the CPI for a large number of iterations. It should be faster.

```
LOOP:
    add r4, r4, r3
    lw r3, O(r1)
    bne r1, r2 LOOP
    addi r1, r1, 4
    add r4, r4, r3
    sw r4, 16(r5)
```

The code has been scheduled to avoid dependence stalls, the execution appears below.

```
    # SOLUTION Execution on the resolve in ME (illustrated) pipeline
LOOP: # Cycle (0)1
    add r4, r4, r3 IF ID EX ME WB
    lw r3, O(r1) IF ID EX ME WB
    bne r1, r2 LOOP IF ID EX ME WB
    addi r1, r1, 4 IF ID EX ME WB
    add r4, r4, r3 IF IDx
    sw r4, 16(r5) IFx
LOOP: # Cycle (0)1
    add r4, r4, r3 IF ID EX ME WB
    lw r3, O(r1) IF ID EX ME WB
    bne r1, r2 LOOP IF ID EX ME WB
    addi r1, r1, 4 IF ID EX ME WB
    add r4, r4, r3 IF IDx
    sw r4, 16(r5) IFx
LOOP: # Cycle (0)1
    add r4, r4, r3 IF ...
```

(b) How much faster would the code above run on the implementation that resolves branches in EX (from the previous problem)?

An iteration takes 6 cycles on the resolve-in-ME version (shown above); for a CPI of $\frac{6}{4}$. On the resolve-in-EX version the second iteration would start in cycle 5 , on cycle earlier, and would not suffer stalls (unlike it's problem 1 counterpart). So it would run with a CPI of $\frac{5}{4}$. The question asked how much faster. A correct answer might be $\frac{5}{4}$ versus $\frac{6}{4}$. (Any reasonable comparison would be just as correct.)
(c) Suppose that due to critical path issues, the resolve-in-EX implementation had a slower clock frequency. Let $\phi_{\text {ME }}$ be the clock frequency of the resolve-in-ME implementation (the one illustrated), and $\phi_{\mathrm{EX}}$ be the clock frequency of the resolve-in-EX implementation. Find $\phi_{\mathrm{EX}}$ in terms of $\phi_{\mathrm{ME}}$ such that both implementations execute the code fragment above in the same amount of time. That is, find a clock frequency at which the benefit of a smaller branch penalty is neutralized by the lower clock frequency on the code fragment above.

Let $c_{\text {EX }}$ denote the number of cycles per iteration of the resolve-in-EX version and define $c_{\text {ME }}$ similarly. The execution time per iteration for the two systems are $\frac{c_{\mathrm{EX}}}{\phi_{\mathrm{EX}}}$ and $\frac{c_{\mathrm{ME}}}{\phi_{\text {VE }}}$. Equate the two quantities, $\frac{c_{\mathrm{EX}}}{\phi_{\mathrm{EX}}}=\frac{c_{\mathrm{ME}}}{\phi_{\text {YE }}}$, and solve for $\phi_{\mathrm{EX}}$ :

$$
\phi_{\mathrm{EX}}=\phi_{\mathrm{ME}} \frac{c_{\mathrm{EX}}}{c_{\mathrm{ME}}}=\phi_{\mathrm{ME}} \frac{5}{6}
$$

Grading Note: Too many students apparently did not give their answers some does-this-makesense scrutiny. We know that the resolve-in-EX system is faster. Therefore we should expect that it can run at a lower clock frequency and still equal the performance of the resolve-in-ME system.

Problem 1: Note: Problems like this one have been assigned before. Please solve this problem without looking for a solution elsewhere. If you get stuck ask for hints. Copying a solution will leave you unprepared for exams, and will waste your (or your parents') hard-earned tuition dollars. A shift unit is to be added to the EX stage of the implementation below. The shift unit has a 32-bit data input, VIN, a 5 -bit shift amount input, AMT, a 1 -bit input SIN, and a 1 -bit control input DIR. There is a 32 -bit data output, VOUT. The DIR input determines whether the shift is left (1) or right (0). If the shift is right then the value at input SIN is shifted in to the vacated bit positions. The meaning of the other inputs is self-explanatory. For a description of MIPS-I instructions see the MIPS32 Volume 2 linked to the course references page.

(a) Connect the shift unit data inputs so that it can be used for the MIPS sll, sllv, srl, srlv, sra, and srav instructions. Assume that the ALU has plenty of slack (it is not close to carrying the critical path). (Control inputs are in the next part.)

- Be sure your design does not unnecessarily inflate cost or lower performance.
- In your diagrams be sure to use the bit ranges used, for example, 27:21, when connecting a wire to an input with fewer bits than the wire.
Solution appears above in blue and purple. The blue material applies to all parts.
The shift input and amount are taken from the ALU input mux outputs, this enables bypassing into the shift unit.
The shift output is muxed with the ALU output before the pipeline latch, so a new EX/ME latch is not needed.
Grading Note: There was no deduction for the following minor issue: Not using the bypassed values for shifter inputs (that would make stalls necessary in some cases).
(b) Show the logic for control inputs DIR and SIN and any multiplexors that you added.

Solution appears above in green. The control logic is simple due to the careful choice of function field values for the different shifts.

The box is Shift has an output of 1 if any of the shifts are present. If the implementation is of MIPS I then for the is Shift we can nor-together bits $31: 26$ (the opcode, to check for format R ) and bits $5: 3$ (the function field, to test for a shift instruction). If MIPS IV is being implemented then this won't work because the output would be 1 for movt and movn instructions.

The shifts have been conveniently encoded so that if IR bit 0:0 is 1 then the shift is arithmetic, otherwise it is logical, that simplifies the SIN logic. Similarly bit $1: 1$ is 1 if the direction is right (but the problem was made up by a left-handed person, so an inverter is needed). A 1 in bit $2: 2$ indicates the shift amount comes from a register.

The s and a control signals are shown using their own pipeline latch bits, but it would be better to get their values from IMM, as is already done for the sa field value.

Grading Note: There was no deduction for the following minor issue: Computing the control signals in $E X$ (doing that could lengthen critical path).
(c) Repeat the design of the datapath but assuming that the ALU is on the critical path and that we don't want to lower the clock frequency.

Solution appears in red. For part c, the ALU would once again be connected directly to the EX/ME. ALU latch. A new EX/ME. shf latch has been added to carry the shift value to ME where it enters a new mux. It is important that the output of this new mux NOT connect to the ME to EX bypass path (since the ALU is critical, as stated above) and that it not connect to the memory port Addr input (because the memory port is always assumed to be critical).

Questions in this assignment are about VAX, an ISA that was mentioned in class but for which no details were given. Use the VAX-11 Architecture Reference Manual (Cover, 1982; text, 1980), which is linked to the course references page, as a reference for this assignment. (The VAX MACRO and Instruction Set Reference Manual can be used as a secondary reference; you may also use any other resources that you can find.) Chapter and section numbers in this assignment refer to the VAX-11 manual, not to the VAX MACRO manual.

Problem 1: Compare the design goals for VAX as described in Section 1.1 to the design goals for SPARC as described in the SPARC Architecture Manual V8 Section 1.1 (also linked to the course references page).
(a) List the design goals for each architecture that are considered defining elements of the respective ISA family (CISC and RISC). Explain whether the design goals in VAX and SPARC are mutually exclusive (meaning you can't easily do both).

The solution below is based on the EE 4720 ISA Families Overview notes. One VAX goals that is consistent with defining elements of CISC ISAS is "High bit efficiency," which implies variable instruction size. Another "Systematic, elegant instruction set ..." can be interpreted to mean a large variety of immediate sizes and addressing modes, which is a CISC characteristic. (Of course, with no context "systematic, elegant" can mean anything.) If instead one interprets the "systematic, elegant" goal only to mean that there are few special-purpose registers and data types, then the goal is consistent with both CISC and RISC.

The SPARC goal of being "Easily pipelined" matches a goal and characteristic of RISC ISAs.
Mutual exclusivity will be discussed for family-consistent goals (the ones mentioned above): High bit efficiency, which implies variable instruction size, is not consistent with easy pipelining because instruction fetch of one instruction would depend upon the decoding of the prior instruction, making it cumbersome to do both (fetch and decode) at the same time.
(b) List a feature or design goal for each ISA that is unrelated to the features of the respective ISA family. Briefly explain why it is unrelated.

For VAX: Extensibility, because that only indicates that new instructions can be added, it says nothing about what those instructions might be.

For SPARC: Register Windows, because that would be just as useful on a CISC ISA.

Problem 2: Answer the following questions about VAX and RISC instruction formats.
(a) MIPS has three instruction formats for the integer instructions, SPARC has from three to five (depending on how you count). The VAX ISA seems to have a simpler format, according to Section 2.6 (it takes just half a page to describe). Even if the VAX format is conceptually simpler (and many would dispute that), why is it more complex in a way that is important to implementers. Hint: This is an easy question.

A VAX instruction can have zero to six operands, and each operand can be a variety of sizes. In a RISC ISA given a format, one knows exactly which bits a particular piece of information (say, a register number) will occupy. That's why the address inputs of the register file in our MIPS implementation can connect directly to the instruction register, no decode logic is needed. In VAX to find, say, a register number for the third operand one must look at the opcode, and the first two operands just to determine where to look. For this one needs decode logic and a shifter or multiplexor to extract the bits that are needed.
(b) In class each operand of a typical CISC instruction had a type and info field to describe its addressing mode. What are the corresponding VAX field names?

The type field is part of the operand specifier in VAX, and info field is part of the operand specifier field and the specifier extension field.

Grading Note: Many answered access type and data type for this question. Those refer to the two general pieces of information that an operand specifier conveys, they do not refer to specific fields in the instruction.
(c) Some RISC instructions have something like a type field, though not capable of specifying the wide range of operand types as the VAX type fields (see the previous problem). Find two examples of MIPS instructions that have an equivalent of a type field. Identify the field and explain what operand types it specifies. Hint: Consider instructions that deal with floating-point numbers.

The format for floating-point operate instructions, such as add.d, has a fmt field which indicates whether the operand is single- or double-precision.
(d) Both MIPS and SPARC have an opcode field that appears in every instruction format and some kind of an opcode extension field that appears in some of the formats. Name the opcode extension fields in MIPS and SPARC. What is the closest equivalent to an opcode extension field in VAX?

The opcode extension in MIPS is called func, the opcode extension in SPARC is called op2 in format 2 and op3 in format 3. The closest VAX equivalent is the second byte of a two-byte opcode.

Problem 3: Find the VAX addressing modes requested in the problems below. The term addressing mode can refer to registers, immediates, as well as memory addresses.
(a) Find the VAX addressing modes corresponding to the addressing mode used by the indicated operands in each instruction below.

Name the mode, and show how the operand would be encoded in the instruction (there is no need to show the entire instruction).

```
# SOLUTION
addi r1, r2, 3 # Both source operands.
#
# r2: Register mode.
# 76 5 4 3 2 1 0 <- Bit positions
# ! 5 ! 2 ! <- Field values.
#
# 3: Literal Mode
# 7 6 5 4 3 2 1 0 <- Bit positions
# ! 0 ! 3 ! <- Field values.
lw r1, O(r2) # Source operand. Note that the displacement is zero.
#
# O(r2): Register Deferred
# 7 6 5 4 3 2 1 0 <- Bit positions
# ! 6 ! 2 ! <- Field values.
lw r3, 4(r4) # Source operand
#
# 4(r2): Displacement Mode, with a byte displacement
# First Byte Second Byte
# 76543210 76543210 <- Bit positions
# ! 10 ! 2 ! ! 4 ! < <- Field values.
ld [l1+12], 13 # SPARC insn, source operand
# No equivalent VAX addressing mode. (Sorry)
```

(b) Find the VAX addressing mode that can be used in place of the three instructions below. Name the mode, and show how it is encoded.

```
sll r1, r2, 2
add r3, r1, r4
lw r5, 0(r3)
# Solution
# Indexed addressing: VAX assembler: (r4) [r2]
#
# 76543210 7654 3 2 1 0 <- Bit positions
# ! 6 ! 4 ! ! 4 ! 2 ! <- Field values.
#
# Note: The amount by which r2 is multiplied is determined by the
# instruction that uses the operand. For example for ADDB3 (add byte
# with two source operands) the value of r2 is multiplied by 1, for
# ADDL3 the value is multiplied by 4 (a VAX longword).
```

Links in this assignment are clickable in Adobe Reader. For the questions below refer to the gcc 4.1.2 manual, available via http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/.

Problem 1: Read the introductory text to the optimization options page, 3.10, in the GCC 4.1.2 manual, and familiarize yourself with your Web browser's search function so that you can search the rest of the page. Answer the following questions.
(a) When optimizing gec tries to fill branch delay slots. What option can be used to tell gcc not to fill delay slots, without affecting other optimizations? What option can be used to control how much effort gcc makes to fill delay slots?

The option for not filling delay slots is -fno-delayed-branch. At least two options can be used for effort. They are --param max-delay-slot-insn-search=N and --param max-delay-slot-live-search=N, where $N$ is a number of instructions, a higher number means more effort. These options indicate how many instructions to look at in a search to fill a branch delay slot.
(b) A reason given in class for scheduling code was to avoid stalls due to a lack of bypass paths. What reason is given in the description of the -fschedule-insn option?

The documentation attributes stalls to slow floating point instructions (which most of them are) and for memory instructions. The assumption is that bypass paths that would be frequently used are already there. (That is, there are no "missing" bypass paths to schedule around, or more precisely the missing bypass paths are used so infrequently they are not worth mentioning.)

Problem 2: The POWER and PowerPC ISAs have alot in common, but each has instructions the other lacks. Show the gcc command line switch to compile for both, start looking in section 3.17, Hardware Models and Configurations.

To compile for both use two switches, each eliminating instructions limited to one of the ISAs: -mcpu=common one could also use the pair of switches: -mno-power -mno-powerpc.

Problem 3: Read the following blog post about the use of profiling in the build of the Firefox Web browser:
http://blog.mozilla.com/tglek/2010/04/12/squeezing-every-last-bit-of-performance-out-of-the-linux-toolchain/. The post compares the results of profiling optimizations provided by gcc to those obtained using other tools for optimization.
(a) As described in the blog post, what was the training data used for profiling?

The training data was just the "Quit" command. That is, Firefox was started and then exited. One might expect they would profile Firefox rendering some Ajaxy Web page, but they didn't.
(b) Suppose that a Web page with a 5000 -row table performs just as sluggishly with the profileoptimized gcc build described in the blog post (firefox.static.pgo) as the ordinary Firefox build (firefox.stock). Provide a possible reason for this, and a solution.

The default startup code either did not have table, or did not have a table elaborate enough to guide optimization. The solution would be to profle on the 5000 -row table.

Problem 4: SPEC recently ended a call for possible programs for their next CPU suite, cpuv6. Read the page describing the call: http://www.spec.org/cpuv6/.
(a) There is a section entitled "Criteria SPEC considers important for the next CPU benchmark suite." Evaluate the suitability of the pi.c program used in class based on each of these criteria.

The pi program would not be suitable. Here is how the pi program meets each of the criteria:

A good benchmark candidate is:

- Used by real users
:- ( Criterion not met: No "real" user because no one uses that program to compute $\pi$.
- Compute bound, or can have its compute bound portion excerpted
:-) Criterion met: The program is compute bound.
- Portable or can be ported to multiple hardware architectures and operating systems with reasonable effort
:-) Criterion met: Written in standard C, easy to port.
- Represents the state of the art for the given field
:- ( Criterion not met: Much better ways to compute $\pi$.
- Derived from a representative application
:- ( Criterion not met: Nope.
- Capable of solving problems of varying sizes. SPEC CPU2006 used 3 workloads, in various capacities, for its benchmarks.
:-) Criterion met: Can vary the number of iterations.
- Reasonably predictable as to its code path. For example, minor differences in floating point accuracy across platforms should not cause the program/application to do wildly different work on those platforms.
:- ( Criterion not met: Not sure, but given the answers above not worth it to find out.


## 53 Spring 2010 Solutions

Problem 1: Re-write each code fragment below so that it uses fewer instructions (but still does the same thing). Note: In the original assignment the branch instruction was blt r1, r0 TARG.

```
# Fragment 1
lw r1, O(r2)
addi r2, r2, 4
lw r3, O(r2)
addi r2, r2, 4
#
# SOLUTION
lw r1, O(r2)
lw r3, 4(r2)
addi r2, r2, 8
# Fragment 2
sub r1, r2, r3
bne r1, r0 TARG
add r1, r5, r6
#
# SOLUTION
bne r2, r3 TARG
add r1, r5, r6
# Fragment 3
ori r1, r0, Ox1234
sll r1, r1, 16
ori r1, r1, 0x5678
#
# SOLUTION
lui r1, 0x1234
ori r1, r1, 0x5678
```

Problem 2: The MIPS code below runs on the illustrated implementation. Assume that the number of iterations is very large.


LOOP:
lw r3, O(r1)
addi r2, r2, 1
beq $r 3$, r4 LOOP
lw r1, 4(r1)
(a) Show a pipeline execution diagram with enough iterations to determine the CPI.

Diagram shown below. To determine the CPI we need a repeating pattern of iterations. An iteration begins when the first instruction of the loop is in IF , the first, second, and third iterations begin in cycle 0,5 , and 11 , respectively. At the beginning of the first iteration only $\mathrm{l}_{\mathrm{w}} \mathrm{r} 3$ is in the pipeline, at the beginning of the second iteration 1 w r3 is in IF, $1 \mathrm{w} r 1$ is in ID, etc. (look at the stages directly above the IF in cycle 5 ). So the pipeline state is different at the beginning of the first and second iterations. At the beginning of the third iteration, in cycle 11 , the pipeline contents (state) is the same as the beginning of the second. Therefore we expect the pattern to repeat and so can determine the CPI using the second iteration.

```
LOOP: # Cycle 0
    lw r3, O(r1) IF ID EX ME WB
    addi r2, r2, 1 IF ID EX ME WB
    beq r3, r4 LOOP IF ID -> EX ME WB
    lw r1, 4(r1) IF -> ID EX ME WB
LOOP: # Cycle 0
    lw r3, O(r1)
    addi r2, r2, 1
    beq r3, r4 LOOP
    lw r1, 4(r1)
LOOP: # Cycle 0
    lw r3, O(r1) IF ID -> EX ME WB
```

(b) Determine the CPI.

The number of cycles is $11-5=6$ (the difference between the start times of the second and third iterations), so
the CPI is $\frac{6}{4} \mathrm{CPI}$.
(c) Schedule (re-arrange) the code to remove as many stalls as possible.

There are two solutions below. The first removes one of the two stalls, the second code fragment removes both stalls.
The first code fragment loads exactly the same items as the original code, but the second one loads an extra 0 (r1), which can possibly result in loading an illegal memory address. Either solution would get full credit.

```
LOOP: # Solution 1, still has 1 stall.
    lw r3, O(r1)
    lw r1, 4(r1)
    beq r3, r4 LOOP
    addi r2, r2, 1
    # Solution 2, no stalls, but risks bad addr on last iter.
    lw r3, O(r1)
    bne r3, r4 DONE
    nop
LOOP:
    lw r1, 4(r1) IF ID EX ME WB
    addi r2, r2, 1 IF ID EX ME WB
    beq r3, r4 LOOP IF ID EX ME WB
    lw r3, O(r1) IF ID EX ME WB
DONE:
```

Problem 3: The MIPS implementation from the previous problem has three multiplexors in the EX stage.
(a) Write a program that executes without stalls and which uses the eight ALU multiplexer inputs in order (perhaps starting at cycle 3) in consecutive cycles. That is, in cycle 3 the top input of the upper ALU mux would be used, (bypass from memory), in cycle 4 the second one would be used (NPC), in cycle 5 rsv, in cycle 6 bypass from WB, in cycle 7 we switch to the lower ALU mux with the bypass from ME input, in cycle 8 rtv , etc.

> Solution shown below. Note that the jal instruction writes register r31.

```
# Bypass Upper-Mux-- Lower-Mux--
# Bypass ME NP RS WB ME RT IM WB
# Cycle 0
add r1, r2, r3 IF ID EX ME WB
add r4, r1, r5 IF ID EX ME WB
jal IF ID EX ME WB
add r6, r7, r4 IF ID EX ME WB
add r1, r31, r8 IF ID EX ME WB
add r10, r11, r1 IF ID EX ME WB
add r12, r13, r14 IF ID EX ME WB
addi r15, r16, 123 IF ID EX ME WB
add r17, r18, r12 IF ID EX ME WB
# Cycle 
# Bypass ME NP RS WB ME RT IM WB
# Bypass Upper-Mux-- Lower-Mux--
```

(b) Explain why it would be impossible to use the EX-stage rtv mux inputs in order in consecutive cycles.

The middle rtv mux input is the bypass from the memory stage. For that input to be used the immediately preceding instruction would have to write a register, which stores don't do. Therefore it is impossible. If they didn't have to be in order (but still consecutive) then it would be easy, see the code below.

```
# Code using all the RTV inputs in consecutive cycles but not in order.
# Bypass RT ME WB <- Mux inputs, in order.
# Bypass ME WB RT <- Mux inputs, but not in order.
# Cycle }\quad0\quad
add r1, r3, r4 IF ID EX ME WB
sw r1, O(r2) IF ID EX ME WB
sw r1, 4(r2) IF ID EX ME WB
sw r1, 8(r2) IF ID EX ME WB
# Cycle 
```

Problem 1: The SPARC jmpl (jump and link) instruction adds the contents of two source registers or a register and an immediate, and jumps to that address. It also puts the address of the instruction in the destination register (usually to be used to compute a return address). For more information, find the description of jmpl in the SPARC V8 ISA description from the references linked to the course home page.

In this problem a similar instruction (or instructions) will be added to MIPS. Like the SPARC $j m p l$, the MIPS variant can jump to a target determined by the sum of two registers or a register and an immediate, while the address of the instruction is saved in the destination register. (Note that the saved address is different than the address saved by MIPS' jalr and jal instructions. Be sure to save the address indicated by the SPARC definition.)
(a) Show how the MIPS version of these instruction(s) can be encoded. Show a format for the instruction, using the descriptions in the MIPS32 Architecture Volume II (linked to the references page) as an example. The format should show which instruction fields indicate each part of the instruction.

Two instructions are needed, jmpl for the two-source-register form and jmpli for the source-register-plusimmediate form. Assembly syntax:

```
jmpl rd, rs, rt
Description:
    pc_cpy <- PC
    PC <- rs + rt
    rd <- pc_cpy
jmpli rt, rs, immed
Description:
    pc_cpy <- PC
    PC <- rt + sign_extend(immed)
    rt <- pc_cpy
```

(b) Show datapath changes (that is, omit control) to the implementation below needed to implement this (these) instruction(s). The changes must fit in naturally with what is present and should not risk lowering clock frequency. Do not forget about any changes needed to save a return address.


Solution appears above. The changes needed for the target address appear in green, the changes needed to save the return address appear in blue. The target address is computed by the ALU, it is sent to the IF-stage PC mux from the ME stage to preserve clock frequency. If the output of the ALU connected directly to the IF-stage PC mux the clock frequency might have to be lowered because the ALU output would not be ready until the end of the cycle (based on the pre-change clock frequency).

The SPARC jmpl instruction saves the address of the instruction itself, new pipeline latches were added to carry that. Note that this is probably wasteful, since there are already latches to carry NPC needed by the existing jal and jalr instructions. The alternative would be to subtract four from the NPC value.

To save some hardware, the PC value joins the rtv in EX, making use of a new ME-stage mux.
(c) As discussed in class, a SPARC-style jmpl on something like our 5 -stage pipeline would have to be resolved in EX. However, a higher-cost implementation might resolve a jmpl in ID if no addition were necessary.

Identify which of the following cases is the least trouble to detect (shown with SPARC assembler), and explain why it is the least trouble:

```
jmpl %g1, %g0, %o7 ! g0 is the zero register.
jmpl %g1, 0, %o7 ! The immediate is zero.
jmpl %g1, %g2, %o7 ! Contents of g2 is zero.
```

Trouble in this context is time and hardware cost. Both hardware cost and time are determined by how many bits need to be checked for a zero value. Another factor for time is when during a clock cycle the check can start.

For the first case we need to check for the zero register in the rt field. That entails checking five bits, and these bits are available at the beginning of the clock cycle.

For the second case we need to check whether the immediate is zero. That means checking 16 bits, also starting at the start of the clock cycle.

For the last case we need to check 32 bits (the rt value at the output of the register file), and those bits are not available until near the end of the clock cycle.

Therefore the first case is the best (least trouble), and the last case is worst, by far.

Problem 2: Without looking at the solution, do Fall (November) 2007 Midterm exam Problem 1. Use the Statically Scheduled MIPS study guide, http://www.ece.1su.edu/ee4720/guides/ssched.pdf, for tips on how to solve this interesting, understanding-building, and fun-to-solve (if one is prepared and not under intense time pressure) problem. Only use the solution if you must. Warning: The test problems will be chosen under the assumption that students really solved this problem.

Problem 1: The code below executes on the illustrated MIPS implementation. Assume that any reasonable bypasses needed for the FP operands are available, even though they are not shown in the illustration. A bypass is reasonable if it does not have a significant impact on clock frequency and if it does not use circuitry that can predict the future.


```
# SOLUTION
LOOP: # 0 0 1 2 2 3 3 4 4 5 6 6 7 7 8 8
    ldc1 f0, O(r1) IF ID EX ME WF
    mul.d f2, f0, f4 IF ID -> M1 M2 M3 M4 M5 M6 WF
    add.d f6, f6, f2 IF -> ID ------------>> A1 A2 A3 A4 WF
    bne r1, r2, LOOP
    addi r1, r1, 8
        IF -------------> ID EX ME WB
        IF ID EX ME WB
#
    Second Iteration Below
LOOP: # 0 0 1 2 2 3 4 4 5 5 6 7 7 8 % 9 10 11 12 13 14 15 16 17 18 19 20 21 22
    ldc1 f0, O(r1)
    mul.d f2, f0, f4
    add.d f6, f6, f2
    bne r1, r2, LOOP
    addi r1, r1, 8
```



```
#
    ldc1 f0, O(r1)
(a) Show a pipeline execution diagram covering enough iterations to compute the CPI. Don't forget to check code for dependencies.

Solution appears above. The state of the pipeline is the same at the start of the second and third iteration (cycles 11 and 22) and so the second iteration can be used to compute CPI....
(b) Compute the CPI.
\(\ldots\) which is \(\frac{22-11}{5}\).
(c) Remember, that some bypass paths are assumed present though not illustrated. Add the needed paths to the implementation and show when they are used.

The bypass paths are from WF to the M1 and A1 inputs, they appear in blue. They are used by the 1dc1 bypassing to the mul.d and mul.d bypassing to the add.d.

Problem 2: Precise exceptions are necessary for integer instructions, but only Nice To Have for floating-point instructions. Suppose exception conditions, such as overflow, were detected in A4 and M6 in the pipeline from the previous problem.
```


# Pipeline diagram for solution.

# Cycle: }\quad0\quad1

mul.d f2, f0, f4 IF ID M1 M2 M3 M4 M5 M6 WF
add.d f6, f6, f2 IF ID ------------> A1 A2 A3 A4 WF
and r3, r3, r5 IF ------------> ID EX ME WB
addi r1, r1, 8 IF ID EX ME WB

```
(a) For the code fragment above, would a mul.d exception detected in M6 be precise? Explain in terms of architecturally visible storage (register and memory values) when the handler starts. (Note that in general exceptions detected in M6 would not be precise, but the question is only asking about the fragment above.)

Yes, because when the multiply is in M6 none of the following instructions has written a register, so they could be squashed. Registers \(f 6, r 3\), and \(r 1\) will reflect execution up to the mul.d instruction, a requirement of precise exceptions.
(b) For the code fragment above, would a add.d exception detected in A4 be precise? Explain in terms of architecturally visible storage when the handler starts.

No, because it would be too late to prevent the and instruction from writing r3.

Problem 3: The MIPS implementation below has a fully pipelined FP add unit. Replace the FP add unit with one that has an initiation interval of 2 and a total computation time of 4 cycles. Note that the time to compute a floating point sum is the same on the original and replacement adder.


The new adder has two stages, A1 and A2, each has two inputs (like their fully pipelined counterparts), and each has two outputs. In the first cycle of computation the source operands are placed at the inputs to A1, in the second cycle of computation the values at the outputs of A1 at the end of the first cycle are placed at the inputs to A1. In the third cycle the values at the outputs of A1 at the end of the second cycle are placed at the inputs A2, and in the fourth cycle the inputs to A2 are the values at the outputs of A2 at the end of the third cycle. The sum is available from the upper output of A2 at the end of the fourth cycle.
(a) Replace the FP adder datapath with the one described above.

Solution appears above in blue. Multiplexors at the adder FU inputs select the proper values to process. Note that some cost is saved by having fewer pipeline latches for intermediate data but that is partly offset by the need for the multiplexors.
(b) Modify the control logic for the new adder. Be sure to account for the structural hazard when there are two consecutive FP add instructions.

Solution appears above in green. Assuming the multiply is still fully pipelined the pipeline latches carrying xw, we, and \(f d\) are still needed and so can't be removed. (If the multiply also had an initiation interval of two then half the number of pipeline latches would be needed.) The control logic for detecting the WF structural hazard does not need to change since operations take the same number of cycles to reach WF. It is only necessary to add control logic to detect the structural hazard, and that is done by examining the we bit in the M3 stage, to see if that stage is occupied, and the LSB of xw, to determine if M3 is occupied by an add operation.

Homework 4 Solution
Due: 28 April 2010
Problem 1: A deeply pipelined MIPS implementation is constructed from our familiar five-stage pipeline by splitting IF, ID, and ME each into two stages, but leaving EX and WB as one stage. The total number of stages will be eight, call them F1, F2, D1, D2, EX, Y1, Y2, and WB. In this system branches are resolved at the end of D2 (rather than at the end of ID). Assume that all reasonable bypass paths are present.
(a) Provide a pipeline execution diagram of the code below for both the 5 -stage and this new implementation, for enough iterations to compute the IPCs.

Solution appears below. Note that for the 8 -stage system the second iteration, starting in cycle 9 , starts with the processor in the same state as the third iteration, starting in cycle 18 , and so the second iteration can be used to compute IPC. The execution rate is \(\frac{5}{18-9}=\frac{5}{9} \mathrm{insn} /\) cycle for the eight-stage system. By a similar argument the rate for the five-stage system is
```

    \frac{5}{12-6}=\frac{5}{6}\mathrm{ insn/cycle}\mathrm{ .}
    
# Solution

# Eight-Stage Pipeline

# 

LOOP: }\quad0\quad
addi r2, r2, 4 F1 F2 D1 D2 EX Y1 Y2 WB
lw r1, O(r2) F1 F2 D1 D2 EX Y1 Y2 WB
add r3, r3, r1 F1 F2 D1 D2 ----> EX Y1 Y2 WB
bne r5, r4 LOOP F1 F2 D1 ----> D2 EX Y1 Y2 WB
addi r5, r5, 1 F1 F2 ----> D1 D2 EX Y1 Y2 WB
F1 ----> F2x
F1x

```

```

    addi r2, r2, 4 F1 F2 D1 D2 EX Y1 Y2 WB
    lw r1, O(r2) F1 F2 D1 D2 EX Y1 Y2 WB
    add r3, r3, r1 F1 F2 D1 D2 ----> EX Y1 Y2..
    bne r5, r4 LOOP F1 F2 D1 ----> D2 EX Y1..
    addi r5, r5, 1 F1 F2 ----> D1 D2 EX..
                                    F1 ----> F2x
                                    F1x
    LOOP: 0}\quad0\quad
addi r2, r2, 4 F1 F2..

```
\# Five-Stage Pipeline
\#
LOOP: \# \(\quad 0 \quad 1 \quad 2 \begin{array}{llllllllllllll} & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15\end{array} 16\)
    addi r2, r2, 4 IF ID EX ME WB
    lw r1, \(0(r 2) \quad\) IF ID EX ME WB
    add \(r 3, r 3, r 1 \quad\) IF ID \(\rightarrow\) EX ME WB
    bne r5, r4 LOOP IF -> ID EX ME WB
    addi \(\mathrm{r} 5, \mathrm{r} 5,1\) IF ID EX ME WB

    addi \(r 2, r 2,4\) IF ID EX ME WB
    lw r1, 0 (r2) IF ID EX ME WB
    add r3, r3, r1 IF ID \(\rightarrow\) EX ME WB
    bne r5, r4 LOOP IF -> ID EX ME WB
    addi r 5 , \(\mathrm{r} 5,1\) IF ID EX ME WB

    addi \(\mathrm{r} 2, \mathrm{r} 2,4\) IF ID EX ME WB
(b) Suppose the 5 -stage MIPS runs at 1 GHz . Choose a clock frequency for the 8 -stage system for which the time to execute the code above is the same as for the 5 -stage MIPS.

First, express the execution rates for the two systems above in instructions per second. Let \(\phi_{5}\) and \(\phi_{8}\) denote the clock frequencies of the 5 -stage and 8 -stage systems. The execution rate in instructions per second is the CPI times the clock frequency: \(\mathrm{CPI} \times \phi=\frac{\phi}{\text { IPC }}\). Solving \(1.8 \phi_{8}=1.2 \phi_{5}\) for \(\phi_{8}\) yields \(\phi_{8}=\frac{1.8}{1.2} \phi_{5}=1.5 \mathrm{GHz}\).
(c) Consider two ways to make a 7 -stage system from the 8 -stage system. In method ID, the two ID stages (D1 and D2) are merged back into one (or if you prefer, the ID stage was never split in the first place). In method ME, the two ME stages (Y1 and Y2) are merged back into one (or were never split).

Which method is better, and why? Assume that the eight-stage system runs at 1.8 GHz . Consider both the likely impact on clock frequency (remembering that you are at least senior-level computer engineering students) and the benefit for code execution (don't just consider the code above, argue for what might be typical code).

As mentioned in class a number of times, it is the memory stage which would take the most time. Here are sample stage timings with memory taking the most: \(\mathrm{IF}, 1 \mathrm{~ns}\); \(\mathrm{ID}, 0.4 \mathrm{~ns} ; \mathrm{EX}, 0.56 \mathrm{~ns} ; \mathrm{ME}, 1 \mathrm{~ns}\); and \(\mathrm{WB}, 0.2 \mathrm{~ns}\). For the 5 -stage system memory determines the clock frequency. On the 8 -stage system, assuming the memory stages timings are split, clock is determined by the EX stage.

Based only on clock frequency it would be better to merge the D1 and D2 stages, since that would not impact clock frequency (the original ID could handle 1.8 GHz ). In contrast, if the memory stages were merged the clock would be back to 1 GHz and so there would be no performance gain.

Splitting stages can introduce stalls or squashes. Because memory was split, the 8 -stage system suffers two stall cycles on a load/use pair. Because IF and ID were split there are two squashes after each taken branch.

Merging Y1 and Y2 would eliminate one stall cycle for each load/use pair. However, scheduling can eliminate many such stalls. Merging D1 and D2 would eliminate one squash on every taken branch. Since many taken branches can't be avoided and occur about 1 out of every 12 instructions in integer code, it would be better to merge the D1 and D2 based on code considerations.

In summary, based on both clock frequency impact and stall/squash benefit, it would be better to merge the D1 and D2 stages.
Problem 2: Itanium is a VLIW ISA designed for general-purpose use. Being a VLIW ISA (as defined in class) its features were chosen to simplify superscalar implementations. The questions below are about such features, read the Intel Itanium Architecture Software Developer's Manual Volume 1, Section 3 for details and concentrate on Sections 3.3 and 3.4. The manual is linked to the course references page, http://www.ece.lsu.edu/ee4720/reference.html. Use this copy to be sure that section and table numbering used here match.
(a) Section 3.3 mentions four types of functional unit, and where an instruction using a particular unit can be placed in a bundle (see Table 3-9 and 3-10).

Suppose an Itanium implementation fetches one bundle per cycle. Indicate the maximum number of execution units of each type needed. (That is, there would be no advantage of having more than this number.) Assume that the units all have latency 1 or else are fully pipelined.

The table below shows the types of execution units that can be used by each slot, as well as the maximum number of units of each type that any bundle can need. The table was constructed by examining table 3-10, and noting that the L-unit/X-unit instructions really use an I or B unit.

The bottom of the table shows the maximum number of times a unit will appear in a bundle. There would be no benefft in having more than that number. For example, there would be no point in having three M -units (assuming fully pipelined) because an instruction in slot 2 would never use it.
```

Execution Unit

```
```

Slot 0: M B
Slot 1: M B I F
Slot 2: B I F*
Max: 2 3 2 1 <- Maximum useful number of units.

```
(b) For this problem suppose the implementation had the minimum number of units of each type, one. Sketch the pipeline execute stages, and show connections to each of the FU inputs. There should be three sets of source operands flowing down the pipeline. Some (or all) of the execute units should have multiplexors at
their inputs to select operands from one of the three instructions in a bundle. Show the multiplexors, and based on the slot restrictions show the minimum number of inputs.

Solution shown below. The ID stage provides register values for instructions in each slot (RS1V, etc). Note that if there were no restrictions on which instruction could appear in each slot then each mux would have three inputs. Also note that bypass paths are not shown.

(c) Notice in Table 3-10 that there is no template with a stop right after Slot 0 and right after Slot 1. Provide a possible reason for this.

With at most one internal stop a bundle can be issued in at most two pieces. With two internal stops the hardware would have to allow three pieces, one for each slot, to move separately. The added hardware cost would not be worth the small benefit in code size.

Suppose there was a template with two such stops (as described above), call this ISA Itanium-stop-stop. Why might code compiled for Itanium-stop-stop be smaller than code compiled for Itanium?

Because the compiler (or human) would no longer need to insert nops (as a last resort) to comply with the one-internal-stop restriction.

Consider Itanium and Itanium-stop-stop implementations that fetch one bundle per cycle (same as in the prior problems). Explain why Itanium-stop-stop might be no faster than Itanium.

Bundle placement and execution (assuming five stages) are shown for code for the regular Itanium and Itanium stop-stop. The last instruction, or, executes at the same time on both systems, so stop-stop has no execution time advantage. (If the hardware complexity results in a lower clock frequency then it would have performance disadvantage.)
```

// Itanium Scheduling
// Bundle 1 - Template 03
nop // Slot 0
add r1 = r2, r3 ;; // Slot 1
sub r4 = r1, r5 ;; // Slot 2
// Bundle 2 - Template 02
nop // Slot 0
xor r6 = r4, r7 ;; // Slot 1
or r8 = r6, r9 // Slot 2
// Itanium Stop-Stop
// Bundle 1 - Template 03 0 0 1 1 2 2 3
// Bundle 1 - Template SS-1
add r1 = r2, r3 ;;

```
0
```

0
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID -> EX ME WB
IF ID -> EX ME WB
IF -> ID EX ME WB
IF -> ID EX ME WB
IF >> ID EX ME WB
IF >> ID EX ME WB
IF -> ID -> EX ME WB

```
    IF -> ID -> EX ME WB
```

```
IF ID EX ME WB
```

```
IF ID EX ME WB
```

```
sub r4 = r1, r5 ;;
xor r6 = r4, r7 ;;
// Bundle 2 - Template SS-2
or r8 = r6, r9
```

IF ID -> EX ME WB
IF ID ----> EX ME WB

IF ----> ID EX ME WB

Problem 1: Do Spring 2009 final exam Problem 2. See the posted solution at http://www.ece.lsu.edu/ee4720/2009/fe_sol.pdf

Problem 2: Do Spring 2009 final exam Problem 3, the branch predictor problem. See the posted solution at http://www.ece.lsu.edu/ee4720/2009/fe_sol.pdf.

Problem 3: Do Spring 2009 final exam Problem 5.
See the posted solution at http://www.ece.lsu.edu/ee4720/2009/fe_sol.pdf.

## 54 Spring 2009 Solutions

Problem 1: Answer each question.
(a) Explain why the code below won't finish running.

```
LOOP:
    lw r1, O(r2)
    xor r3, r3, r1
    bne r2, r4 LOOP
    addi r2, r2, 2
```

The lw effective address must be a multiple of four (the address alignment restriction) but it can't always be in the code fragment above since $r 2$ is incremented by 2 each iteration. The code won't finish because the 1 w will raise some kind of address misalignment exception at either the first or second iteration.
(b) Shorten the code below.

```
lui r1, 0x1234
ori r1, r1, 0x5678
lw r1,0(r1)
# Solution
lui r1, 0x1234
lw r1,0x5678(r1)
```

(c) Shorten the code below.

```
xor r1, r2, r3
beq r1, r0 TARG
addi r1, r4, 1
```

In the code above $r 1$ will be zero only if $r 2$ is equal to $r 3$, so there is no need for the xor. Note: In the original assignment the last instruction did not modify $r 1$, so one could not safely remove the xor.

```
# Solution.
beq r2, r3 TARG
addi r1, r4, 1
```

Problem 2: Consider the execution code below on the illustrated implementation.

```
LOOP:
    lw r2, 0(r4)
    slt r1, r2, r3
    beq r1, r0 LOOP
    addi r4, r4, 4
```


(a) Determine the execution rate in IPC (instructions per cycle) assuming a large number of iterations. Use a pipeline execution diagram to justify your answer. (No credit without one.)

```
LOOP:
    # Solution
    # Cycle 
    lw r2, O(r4) IF ID EX ME WB
    slt r1, r2, r3 IF ID -> EX ME WB
    beq r1, r0 LOOP IF -> ID ----> EX ME WB
    addi r4, r4, 4 IF ----> ID EX ME WB
    # Cycle 
    lw r2, O(r4) IF ID EX ME WB
    slt r1, r2, r3 IF ID -> EX ME WB
    beq r1, r0 LOOP IF -> ID ----> EX ME WB
    addi r4, r4, 4 IF ----> ID EX ME WB
    # Cycle 
    lw r2, O(r4) IF ID EX ME WB
```

The code suffers two stalls, the first because the slt needs the 1 w value and the second (a two-cycle stall) because the beq needs the slt value. Iterations start (first instruction of the loop is in IF) in cycles 0,7 , and 14 . Since the pipeline is in the same state in cycles 7 and 14 ( 1 w in IF, addi in ID, and beq in EX) we can expect the iteration that
starts at 14 to be identical to the one that starts at 7 . The time for these iterations is $14-7=7$ cycles, and so the execution rate is $\frac{4}{7}$ IPC(or if you prefer, the instruction initiation interval is $\frac{7}{4} \mathrm{CPI}$ ).
(b) If the previous part was solved correctly there should be a stall due to the branch. Add a bypass path to avoid the branch stall.

The added bypass path appears below in blue. This bypass path eliminates the stall but would likely lower the clock frequency. (See the next problem.)


```
LOOP:
    # Solution - Execution with the bypass.
    # Cycle 
    lw r2, O(r4) IF ID EX ME WB
    slt r1, r2, r3 IF ID -> EX ME WB
    beq r1, r0 LOOP IF -> ID EX ME WB
    addi r4, r4, 4 IF ID EX ME WB
```

(c) Why might the added bypass path impact clock frequency?

The slt at the ALU output would be ready late in the cycle, and these signals would still have to pass through the ID-stage comparison unit, then some control logic, the IF-stage mux, finally reaching the PC input. Since it's reasonable that the critical path passed through the ALU without this bypass, adding the bypass would increase the critical path and therefore reduce clock frequency. (A solution that bypassed from ME to ID would not impact clock frequency [unless it were taken from the memory port output] but it would only reduce the number of stall cycles from 2 to 1 .)
(d) Suppose the clock frequency of the original pipeline were 1 GHz , and call the clock frequency of the added-bypass implementation $\phi$. For what value of $\phi$ will the run time of the code fragment be the same on the original and added-bypass implementations (assuming a large number of iterations).

Without the bypass the code executes at $\frac{4}{7} 1 \mathrm{GHz}$ IPS (instruction per second). With the bypass the code will execute at a rate of $\frac{4}{5} \mathrm{IPCor} \frac{4}{5} \phi$ IPS. Solving $\frac{4}{7} 1 \mathrm{GHz}=\frac{4}{5} \phi$ yields $\phi=\frac{5}{7} \mathrm{GHz}$.
(e) Suppose a blt (branch less than) instruction was available that could compare two registers (not just a register to zero). Re-write the code above for this instruction and add bypasses that are no worse than the added-bypass bypass. How would the performance of this blt implementation on the re-written code fragment compare to the added-bypass implementation on the original code fragment? Assume both systems have the same clock frequency.

The bypass needed for this part would be from ME to ID, since one value to compare arrives at through memory port. The execution rate of the original code on the bypassed pipeline is 5 cycles per loop iteration. The code with blt still suffers one stall and so executes at 4 cycles per iteration (see diagram below). The net result is improved performance. Note that the speedup (performance ratio) is $\frac{5}{4}=1.25$ while the improvement in IPC is only $\frac{4 / 5}{3 / 4}=1.0667$.

```
# Solution: Code using blt
LOOP:
    lw r2, O(r4)
    blt r2, r3 LOOP
    addi r4, r4, 4
    IF ID EX ME WB
    IF ID -> EX ME WB
        IF -> ID EX ME WB
```

Problem 1: Solve Fall 2008 EE 4720 midterm exam Problem 1. Solutions have not yet been posted but were given in class in the Fall 2008 semester. Do not look at any solution you might come across. (Solutions will be posted after the homework is collected.) Hints: This sort of problem looks alot harder than it actually is. For a solved problem of this type see the solution to Problem 1 in the Spring 2008 midterm exam. Also look at the Fall 2008 Homework 2 for more on the shift unit.

See the posted exam solution.

Problem 1: Consider three ISAs: IA-32 (or 64), IBM POWER6, and Sun SPARC.
(a) Choose the highest-performing system, based on SPECint2006, for each ISA. Print or provide a link to the test disclosure for each one.

The highest performing IA-32 (loosely defned) implementation is the Xeon X 5570 at 3.33 GHz achieving a peak score of 36.3: http://spec.org/cpu2006/results/res2009q1/cpu2006-20090316-06691.html

The highest performing POWER implementation is the POWER6 at 4.2 GHz achieving a peak score of 19.2: http://spec.org/cpu2006/results/res2008q3/cpu2006-20080623-04639.html

The highest performing SPARC implementation is the SPARCVII at 2.52 GHz achieving a peak score of 12.6 : http://spec.org/cpu2006/results/res2008q4/cpu2006-20081023-05678.html
(b) Find examples of benchmarks (if any) which favor each ISA based on the test disclosures you found.

There are two ways to answer this question: is a benchmark faster on one system than on the other two, or does a benchmark help one system's score more than the other two.

The Xeon X5570 system runs each of the benchmarks faster than the other two systems and so in the first sense all benchmarks favor the X5570 system (and so no benchmarks favor the POWER6 or SPARCVII systems).

For the second sense consider the perlbench benchmark. On the SPARCVII system perlbench's peak ratio, 14.3, is above the overall peak score of 12.6 and so perlbench is helping the SPARCVII. On the other systems the perbench peak score is below the overall peak score and so perlbench favors the SPARCVII system. (That's not true when looking at base scores.) Another benchmark favoring SPARCVII is xalancbmk.

The mef benchmark favors the POWER6 system, where mcf's peak score is $50 \%$ higher than the overall peak score. On the other systems mef's peak score is above the overall peak score, but by less than $50 \%$.

The $\mathbf{X} 5570$ is favored by libquantum with a peak score over 8.5 times higher than the overall peak. The other systems trounce the reference system on libquantum too, but for them the peak scores are 4.4 times higher (for POWER6) and 2.5 times higher (SPARCVII).

Problem 2: Why might a company publish peak SPECcpu scores but not base? Why is it against the rules?

One reason to publish peak but not base is because peak is much nigher than base, the difference indicates that it takes great skill or care to achieve the peak score performance, an effort that many users won't or can't make.

It is against the rules because some users might be mislead into believing that they can achieve the same performance with normal program development effort.

Problem 3: Solve the Fall 2008 Final Exam Problems 1 and 2. See the posted final exam solution.

Problem 1: Solve Fall 2008 Final Exam Problem 4 and the additional questions below.
(a) For part (a) provide pipeline execution diagrams for the three systems ( 5 -stage scalar, $n$-way superscalar, and $5 n$-stage superpipelined) running code of your choosing. Refer to these diagrams when answering part (a).

Solution shown below. In the superscalar solution $n$ instructions reside in a stage at one time. in the superpipelined system each stage is split into $n$ stages and the clock frequency is increased by a factor of $n$.

```
# Scalar System
#
# Cycle 1 1 2 3 4 5 6 \ldots.. n n+1 \ldots..
1: add r1, r2, r3 IF ID EX ME WB
2: or r9, r2, r3 IF ID EX ME WB
    ...
n: sub r4, r5, r6
n+1: xor r6, r7, r8
    ...
# n-way Superscalar
#
#Cycle 1 2 3 4 4 5 6 \ldots.. n n+1 \ldots
1: add r1, r2, r3 IF ID EX ME WB
2: or r9, r2, r3 IF ID EX ME WB
n: sub r4, r5, r6 IF ID EX ME WB
n+1: xor r6, r7, r8 IF ID EX ME WB
    ...
# Superspipelined
#
```



```
1: add r1, r2, r3 IF1 IF2 .. IFn ID1 ID2 .. IDn EX1 EX2 .. EXn ..
2: or r9, r2, r3 IF1 IF2 .. IFn ID1 ID2 .. IDn EX1 EX2 .. EXn ..
n: sub r4, r5, r6
n+1: xor r6, r7, r8
    IF1 IF2 .. IFn ID1 ID2 .. IDn EX1 ..
    IF1 IF2 .. IFn ID1 ID2 .. IDn ..
```

Problem 2: Consider the three systems from Problem 4 in the final exam. The problem focused on potential (favorable) execution time, which can be achieved when there are few stalls, here we'll be more realistic.
(a) Which system will suffer more stalls on typical code? Explain.

The dependence in the code below will not stall the sealar system, will always stall the superpipelined system (since sub needs a value in EX1 at the latest, but the add doesn't have it ready until ME1, $n-1$ cycles too late), and will sometimes stall the superscalar system. The non-stall superscalar case is shown below, note that the add is the last instruction of a group so the sub starts one cycle later. Therefore the superpipelined system suffers the most stalls.

```
n: add r1, r2, r3 IF ID EX ME WB
n+1: sub r4, r1, r5 IF ID EX ME WB
```

(b) Invent a quantitative measure of implementation (not program) stall potential and apply it to the three systems. The answer should include a formula for each system (giving the stall potential); the superscalar and superpiplined formulas should be in terms of $n$. Hint: think about the average or minimum distance between two dependent instructions needed to avoid a stall. The formulas should be consistent with your answer to the first part.

Call the measure the average stall distance. Let $a \in\{0, \ldots, A-1\}$ be the set of possible instruction locations (the address divided by 4 in MIPS) and let $s(a)$ denote the minimum number of instructions between an add instruction at $a$ and a dependent sub instruction (so that sub would be at location $a+1+s(a)$ ). For the scalar MIPS system $s(a)=0$ for all $a$ (there are no possible stalls between an add and a subtract). For the supersealar system

$$
s(a)= \begin{cases}0, & \text { if } a \bmod n=n-1 \\ 1, & \text { otherwise }\end{cases}
$$

and for the superpipelined system $s(a)=n$.
Define the average stall distance to be $\frac{1}{A} \sum_{a=0}^{A-1} s(a)$. The average stall distance for the sealar system is 0 , for the superscalar system it is $\frac{n-1}{n}$ and for the superpipelined system it is $n$.

Problem 1: Solve Fall 2008 Final Exam Problem 3.
See exam solution,
Problem 2: Continue to consider the systems and code from Problem 3.
(a) What is the warmup time of the local predictor on branch B2?

It will take 10 B 2 executions to bring the entire local history into the BHT. There are 7 distinct outcome patterns, each will take at most two outcomes to warm up. The total warmup time is $10+2 \times 7=24$.
(b) What is the warmup time of the global predictor on branch B2?

When predicting B2 the contents of the GHR will look something like XXtXXXnXXX, where $n$ and $t$ are possible B 2 outcomes and X are B1 outcomes. Only two B2 outcomes can fit. It will take two executions of B2 to bring the 2 B 2 outcomes into the GHR. There are three local history patterns (TT never occurs), it will take 6 executions to warm them up (though nN won't be followed by accurate predictions). The total warmup time is then approximately $10+2 \times 6$.

The table below shows the local outcome patterns sorted and broken at two outcomes to emphasize predictability. NT and TN are predictable since they are consistently followed by N , while NN it followed by N or T in equal proportions.

```
1234567
```

nN NNTN NT
nN NTNN TN
nN TNNN NT
NN TNNT NN
NT NNNN TN
NT NNTN NN
TN NNNT NN
TN NTNN NN

Problem 3: Continuing still with Problem 3, suppose the number of iterations of the B1 loop could be 1,2 , or 3 , the probability of each number of iterations is $\frac{1}{3}$ and the number of iterations is independent of everything. The patterns of B1 for an iteration of BIGLOOP can thus be N or T N or T T N.
(a) What is the accuracy of the bimodal predictor on B1. An exact solution is preferred but an approximate solution is acceptable. Hint: Model the effect of the change of one BIGLOOP iteration on the counter using a Markov chain, something you may have learned about in other courses.

Let $p_{i}$ denote the probability that the BHT entry for B 1 is $i$ at the top of BIGLOOP.
Consider the change in the counter (the BHT entry) between BIGLOOP and B2. If B1 executes a 1 -tteration loop, N , the counter will be decremented by 1 , a 2 -iteration loop will leave it unchanged (because the counter can never be 3), and a 3 -iteration loop, TTN, will change it from 0 to 1 , or from 1 to 2 , but leave it unchanged at 2 .

Based on these counter changes probability of a counter transition from 0 to 1,1 to 0,1 to 2 , and 2 to 1 are all $\frac{1}{3}$. The rate of transitions from 0 to 1 is $\frac{1}{3} p_{0}$ and the rate of transitions from 1 to 0 is $\frac{1}{3} p_{1}$. The two must balance and so $\frac{1}{3} p_{0}=\frac{1}{3} p_{1}$ and therefore $p_{0}=p_{1}$. Similarly, $p_{1}=p_{2}$. Since $p_{0}+p_{1}+p_{2}=1, p_{0}=p_{1}=p_{2}=\frac{1}{3}$.

When the counter is zero the number of correct predictions for the 1 -teration loop is 1 , for the 2 -iteration loop there is 1 correct prediction, and zero correct predictions for the 3 -teration loop. The total accuracy for this case is $\frac{2}{6}$.

Similarly, when the counter is 1 the numbers of correct predictions are 1,0 , and 1 , and when the counter is 2 the numbers of correct predictions are 0,1 and 2 . Since the probability of each counter value is identical, the overall prediction accuracy is $\frac{2+2+3}{18}=\frac{7}{18}$.
(b) How will B1's behavior impact the accuracy of the local predictor on branch B2? Show an example of execution that would result in a B2 misprediction and compute the probability of that particular execution.

Some of B1's local history patterns will match those of B2, however the subsequent outcomes may not maten and so B1 will pollute B2's PHT entries.

Consider, for example, B2 pattern nnntnntnnn. That could be reproduced with an execution in which B1 patterns are $n \mathrm{n} \mathrm{n} \operatorname{tn} \mathrm{n} \operatorname{tn} \mathrm{n} \mathrm{n}$. B2's next outcome would be at but B1 in this case might have an n .

This particular sequence includes 9 B1 loops, and there is only one way for B1 to do this. The probability is $\left(\frac{1}{3}\right)^{9}$, which is pretty unlikely and is not enough to cause a misprediction of the PHT entry is already warmed up. See the next problem.
(c) Optional: Find the exact prediction accuracy of B2 on the local predictor with B1's new behavior. This may be very difficult so don't spend too much time on it.

For B1 to induce a misprediction in B2 it must mimic a B2 pattern twice before between encounters of the real B2 pattern. It is possible for B1 to produce two patterns between one of B2's because it can produce two outcomes (the ttn pattern isn't useful) in a BIGLOOP iteration where B1 always just inserts 1 . The remainder of the solution is left as an exercise to the reader.

## 55 Fall 2008 Solutions

To answer the first question below see the MIPS32 Architecture manual linked to the course references page.

Problem 1: The MIPS I bgtz and bltz instructions compare a register to zero, but can't compare two registers (unless the second one is the zero register). Consider an extension of MIPS I that allowed branch greater than and branch less than instructions to compare two registers, call the new instructions bgt and blt. Explain why the existing bgtz opcode could be used for bgt but why the bltz opcode could not be used for blt. Hint: See bltzal.

The opcode for bgtz is $0 x 07$ and the value that the ISA specifies for the $r t$ field is 0 . Assuming that no other instruction uses opcode $0 x 07$, the rt field could be used for the second comparison register. (This would not be incompatible with its current use because in its current use it is comparing the rs register to 0 so it wouldn't matter if rt held a register number.)

In the bltz instruction the rt field is being used as an extension of the opcode field and so it cannot be used for a register number. A new blt instruction would need its own opcode.

Problem 2: A C function and a part of a MIPS equivalent are shown below. The C function looks at the attributes of a car and decides what to pack in a promotional giveaway to the car buyer. The assembler code corresponds to the C function up until the last line (checking for a sun roof).

```
#define FE_SPORTY 0x1
#define FE_OFF_ROAD 0x2
#define FE_EFFICIENT Ox4
#define FE_SUN_ROOF Ox10000
#define FE_MANUAL_TRANSMISSION Ox20000
enum Giveaways { G_Food, G_Hiking_Boots, G_Sunblock, G_Driving_Gloves };
void prepare_promotion_package(Car_Object *car) {
    int car_features = car->features;
    if ( car_features & FE_OFF_ROAD ) pack(car, 1200, G_Hiking_Boots);
    if ( car_features & FE_SUN_ROOF ) pack(car, 200, G_Sunblock);
}
# MIPS-I Equivalent of C code.
#
# $a0: Address of car object.
# Notes: Procedure call arguments placed in $a0, $a1, ...
# Assume that pack does not change $a0-$a3 or $s0-$s7
    lw $s0, 16($a0) # Load the features bit vector of car object.
    andi $t0, $s0, 2
    beq $t0, $0 SKIP1
    addi $a1, $0, 1200
    jal pack
    addi $a2, $0, 1
SKIP1:
# PART b SOLUTION STARTS HERE
```

(a) The MIPS code above omits the last line of C code (checking for a sun roof); complete it using MIPS I instructions. (Do this on paper, there is no need to run it.) Hint: A clever solution uses five instructions a straightforward solution uses six instructions. If you have more than ten instructions ask for help.

The solution appears below.
This would be an insultingly trivial problem were it not for the fact that an andi instruction can't be used to mask off the FE SUN ROOF bit because the constant, $0 \times 10000$, is too large for the immediate fild.

The solution uses an sll (shift left logical) instruction instead of an andi instruction to move the sun roof bit to the most significant bit position, making it into a sign bit. Replacing beq with bgez achieves the desired functionality.

```
# MIPS-I Equivalent of C code.
#
# $a0: Address of car object.
# Notes: Procedure call arguments placed in $a0, $a1, ...
# Assume that pack does not change $a0-$a3 or $s0-$s7
    lw $s0, 16($a0) # Load the features bit vector of car object.
    andi $t0, $sO, 2 # Extract OFF_ROAD bit from vector.
    beq $tO, $0 SKIP1 # If OFF_ROAD bit not set, skip ahead.
    addi $a1, $0, 1200 # Arg 1: Promotional item weight
    jal pack # Insert promotional item in Car_Object
    addi $a2, $0, 1 # Arg 2: Promotional item model number.
SKIP1:
# PART b SOLUTION STARTS HERE
# SOLUTION BELOW
#
    sll $t0, $s0, 15 # Make SUN_ROOF bit the sign bit.
    bgez $t0 SKIP2 # If SUN_ROOF not set (t0 >=0) skip ahead.
    addi $a1, $0, 200 # Arg 1: Promotional item weight.
    jal pack # Insert promotional item in Car_Object
    addi $a2, $0, 2 # Arg 2: Promotional item model number.
SKIP2:
```

(b) Add comments to the assembler code above. Write the comments for an experienced MIPS and C programmer, that is, the comments should describe what an instruction is doing in terms of what the C code is trying to do. The comments should not just describe how instructions change register values.

For example, a bad comment for the 1 w instruction would be: Compute address $16+\$ \mathrm{aO}$, retrieve word starting at that address and write into $\$$ s 0 . This is a bad comment because an experienced MIPS programmer already knows what an lw instruction does. The comment for lw in the code (Load the features...) is good because it tells the reader what the $\$ \mathbf{s} 0$ value is in terms of what the code is supposed to do.

The comments have been added to the solution code above.
Grading Notes: Many solutions included something like the following comment for the jal pack routine: "Save the return address and call the pack routine." Points were deducted for such comments because an experienced MIPS programmer, and even beginners, already know what jal does. Comments like that increase the amount of time it takes someone to read (and write) the code.

Problem 3: Consider the code from the previous problem. Invent a new branch instruction that can be used for the kind of branching used in the code: testing if a single bit in a register value is 1.
(a) Show the encoding for the new branch instruction. The new instruction must fit as naturally as possible with other instructions.

Call the new instruction bbit, branch if bit set. The rs register has the value to test and the rt field holds the bit number in the rs value to test. The immed field holds the displacement which is used like the displacement in any other branch.

In the example assembler below the branch is taken if bit position 16 in register $\mathbf{s} 0$ is 1 . (This instruction could have been used in the first problem).

```
bbit $s0, 16, SKIP2
```

The encoding appears below:
Opcode RS RT Immed

|  |  | Source reg. | Bit position. | Displacement |  |
| ---: | :--- | :--- | :--- | :--- | :--- |
|  | 31 | 26 | 25 | 21 | 20 |

(b) Compare the implementation cost and performance of the new instruction to the existing MIPSI bltz and to a hypothetical blt instruction. (With each instruction doing its own thing, not as part of functionally equivalent alternatives.)

Cost of bbit: A $32 \times 1$ bit multiplexer could be used to extract the desired bit position. The logic for that mux would implement the expression

$$
b_{0} \overline{p_{4}} \overline{p_{3}} \overline{p_{2}} \overline{p_{1}} \overline{p_{0}}+b_{1} \overline{p_{4}} \overline{p_{3}} \overline{p_{2}} \overline{p_{1}} p_{0}+b_{2} \overline{p_{4}} \overline{p_{3}} \overline{p_{2}} p_{1} \overline{p_{0}} \cdots+b_{31} p_{4} p_{3} p_{2} p_{1} p_{0}
$$

where $b_{i}$ is the bit at position $i$ in the rs register value, and $p_{j}, 0 \leq j<5$ is the bit at position $j$ in the binary representation of $r t$ (the field value, not the register value).

Cost of blt: A magnitude comparison unit is needed, the complexity of which is similar to an adder (or subtractor). Since it must be made fast, the cost would be comparable to the adder in the ALU. A ripple adder (or subtractor) is one of the least expensive designs, that requires about five gates per bit (counting an exclusive or as one gate). The higher-speed design would cost more.

The bbit logic could use a single six-input gate per bit (a term in the expression above), but to be conservative one might count it as five two-input gates. This cost is comparable to that of a binary full subtractor and so is certainly lower than the cost of the lookahead subtractor needed by blt to perform the comparison in ID. Therefore the cost of the hardware needed for bbit is less than the hardware needed for blt .

The bit test logic is two levels though it has a large fan in. The number of logic levels for the comparison depends on cost but is certainly greater than two levels. Therefore the bbit logic is probably faster than blt .

The bltz instruction only needs to test the sign bit, so the hardware cost of bltz is lower than bbit and blt
and the logic for bltz is much faster than bbit and blt
Problem 4: Solve Fall 2007 Homework 2 without looking at the solution. Then look at the solution and give yourself a grade on a scale of $[0,1]$. Warning: test questions are based on the assumption that homework problems were completed, so make a full effort to solve it without first consulting the solution.

Problem 1: The hardware needed to implement shift instructions, such as sll, is not shown in the implementation below. (The ALU in the implementation below does not perform shift operations.) Add a separate shift unit to the implementation to implement the MIPS sll and sllv instructions. The shift unit has a shift amount input and an input for the value to be shifted.

- Show exactly where the shift-amount bits come from (including bit positions).
- Add bypass paths so that the code below can execute without a stall.
- The primary goal is to not slow the clock frequency, the secondary goal is to minimize added cost. This might affect where multiplexers are placed.
sll r1, r2, 3
add $\mathrm{r} 3, \mathrm{r} 1, \mathrm{r} 4$


The added hardware for the shift appears in blue and the needed bypass path appears in green.
The shifter would stall for any close dependencies on the shift amount, but such dependencies do not appear in the sample code.

A low-cost solution would have the ALU and shifter outputs go to a EX-stage mux, this would eliminate the need for any added logic beyond the EX stage. But since performance was the primary goal and the ALU output was likely on the critical path such a mux could not be added. Instead, the mux is placed in the ME stage.

The new ME-to-EX bypass path adds cost. If performance were not the primary goal that added bypass could be avoided by moving the existing bypass connection to the mux output. But that would add to the critical path in precisely the same way the EX-stage mux discussed above would.

Common Mistakes:
Many solutions had an unnecessary "format sa" block.
Many solutions passed an sa value through the ID/EX pipeline latch even though the sa bits were part of the IMM value.

Some solutions passed the shifted value through the ME/WB latch even though such lateh bits could be avoided by using an ME-stage mux.

Problem 2: To answer this question see the SPARC Joint Programming Specification, a description of the SPARC V9 ISA, linked to the course references page. The SPARC V9 ISA is naturally big endian. Since many programs must read data using little-endian byte order, for example when reading a binary data file that was produced on a little-endian system, the programs need some way to get the data into big-endian order. If loading little-endian data were only a small part of what a program did then it could get by with some combination of ordinary instructions to convert the data to big-endian format. For programs spending substantial time reading little-endian data even a 9 -instruction sequence may take too long.

The first instruction below is an ordinary load in SPARC V9, a 64 -bit ISA (in which addresses and registers are 64 bits ). The second instruction, ldxle, is made up; it's a load that assumes data is in little-endian byte order. The last instructions is a real SPARC instruction for loading little endian data.

```
! All load instructions below load 8 bytes into a register.
! Registers are 64 bits.
ldx [%l1], %12 ! Ordinary load. For big-endian data.
ldxle [%l1], %l2 ! Not a real SPARC insn. For little-endian data.
ldxa [%l1] 0x88, %12 ! SPARC's load for little-endian data.
```

(a) The ldxa instruction is an example of an alternate load instruction. The alternate load instructions are intended for three kinds of access. Briefly describe the three kinds and indicate which one is used above. What symbolic name does JPS1 give for 0x88 above?

Note: To answer this question one must read through material dealing with topics not yet covered, for example, the concept of multiple address spaces. It is only necessary that the concept of multiple address spaces is vaguely understood. The kind of access done by the ldxa should be clearly understood.

The symbolic name for $0 \times 88$ is ASI PRIMARY LITTLE.
The three kinds of access are:
Accesses to an alternate address space. The ASI acts like extra bits to put on the end of an address. For example, suppose a 1 dxa speciffed an ASI of $0 x 12$ and an address of $0 x 00000000 \mathrm{abcd0124}$. The full address would be $0 \times 1200000000$ abcd0124. There are many uses of such ASIs, one is to allow OS code to access its own memory space and the memory space of a process it needs to work with.

Access to special machine registers. The ASI indicates which set of machine registers, and the address specifies a particular register. The machine registers are used to control hardware, for example, the memory system or a video card.

Variations on a normal memory access. This includes little-endian byte ordering, and also includes things like fault-free loads, and new data sizes (such as byte loads to a floating point register). In this use the ASI acts like an extension of the opcode field.

Grading Note: This was much harder than intended.
(b) Show the encoding for the three instructions above. The ldx and ldxa are real instructions, so it's just a matter of looking things up. For the ldxle make up an appropriate encoding.

The encodings appear below. There are two possible ways to encode the ldx: with an immediate (shown below) or with rs2 set to g0. The only difference between ldx and ldxle is the opcode. An unused opcode was found for ldxle using the opcode map, in particular table E-4.


Problem 1: Two MIPS implementations appear below, the first is the one presented in class, it will be called the mux-in-EX implementation. The second, the mux-in-ID implementation, has the ALU input multiplexers in the ID stage, to better balance critical paths. The clock frequency of the mux-in-EX implementation is 1 GHz and the clock frequency of the mux-in-ID implementation is 1.1 GHz .

(a) With this change some of the ALU multiplexer inputs are unnecessary. Show which inputs are unnecessary and explain why.

The WB bypass multiplexer inputs are unnecessary because the register file can already bypass from the data in port (which connects to WB) to the data out port.

Problem continued on next page.
(b) The code below computes the sum of the low 12 bits of elements in an integer array. Compute the performance, in array elements per second, of this code for both the mux-in-EX system and the mux-in-ID system. Assume that the array size is large and that the number of array elements is even.

Note that the code computes two array elements per loop iteration. The solution strategy is to determine the number of cycles per iteration, then use the clock frequency to compute the performance in array elements per second.

The pipeline diagrams appear below. Execution is shown until a repeating pattern is encountered (by examining the pipeline state present at the first instruction in an iteration). For the mux-in-EX system there are no stalls, the mux-in-ID system has several stalls.

The code for the mux-in-EX system enjoys smooth, stall-free execution and so takes 8 cycles per iteration, 4 cycles per array element, and computes at a rate of $\frac{1.0 \times 10^{9}}{\frac{1}{2} \times(8-0)}=250 \times 10^{6}$ array elements per second.

The code for the mux-in-ID system suffers dependence stalls. From the pipeline execution diagram below one can see that it takes $23-11=12$ cycles per iteration or 6 cycles per element. It computes at a rate of $\frac{1.1 \times 10^{9}}{\frac{1}{2} \times(23-11)}=$ $183.3 \times 10^{6}$ elements per second. The benefit of the higher clock frequency has been undermined by the stalls for this code.

```
# Performance on mux-in-EX system
LOOP: # Cycle 0
lw $t0, 0($a0) IF ID EX ME WB
lw $t5, 4($a0) IF ID EX ME WB
andi $t2,$t0, 0xfff IF ID EX ME WB
add $v0, $v0, $t2
andi $t7, $t5, 0xfff
add $v0, $v0, $t7
bne $a0, $t1 Loop
addi $a0, $a0, 8 IF ID EX ME WB
# Cycle (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
lw $t0, O($a0) IF ID EX ME WB
# Performance on mux-in-ID system
LOOP: # Cycle 0
lw $t0, O($a0) IF ID EX ME WB
lw $t5, 4($aO) IF ID EX ME WB
andi $t2, $t0, 0xfff IF ID -> EX ME WB
add $v0, $v0, $t2 IF -> ID -> EX ME WB
andi $t7, $t5, 0xfff IF -> ID EX ME WB
add $v0, $v0, $t7 IF ID -> EX ME WB
bne $a0, $t1 LOOP IF -> ID EX ME WB
addi $a0, $a0, 8 IF ID EX ME WB
# Cycle 
lw $t0, 0($a0) IF ID -> EX ME WB
lw $t5, 4($a0) IF -> ID EX ME WB
andi $t2, $t0, 0xfff
    IF ID -> EX ME WB
add $v0, $v0, $t2
IF -> ID -> EX ME WB
andi $t7, $t5, 0xfff
                                    IF -> ID EX ME WB
add $v0, $v0, $t7
bne $a0, $t1 LOOP
addi $a0, $a0, 8
lllllllllllllllllllllllll
lw $t0, 0($a0) IF ID -> EX.
```

(c) If, after double-checking your work, the performance of the mux-in-ID system is faster than the old mux-in-EX system inform the professor that there is a mistake in this problem. Otherwise, schedule (re-arrange instructions) the code above so that it performs faster (while still performing the same computation) on the mux-in-ID system.

The solution appears below. The instructions can easily be rearranged to avoid the stalls. Now the system computes at a rate of 275 million array elements per second, outperforming the mux-in-EX system.

```
LOOP: # Cycle 0
lw $t0, O($a0) IF ID EX ME WB
lw $t5, 4($a0) IF ID EX ME WB
addi $a0, $a0, 8 IF ID EX ME WB
andi $t2, $t0, Oxfff IF ID EX ME WB
andi $t7, $t5, Oxfff IF ID EX ME WB
add $v0, $v0, $t2
bne $a0, $t1 LOOP
add $v0, $v0, $t7 IF ID EX ME WB
# Cycle 0
lw $t0, 0($a0) IF ID EX ME WB
```

Problem 2: You are in an alternate universe where you work for MIPS at a time when its first implementation (mux-in-EX) has been very successful and is in the hands of customers of all types. You are deciding on whether to make mux-in-ID the second implementation to be marketed.
(a) What role do compiler writers have in the success of mux-in-ID? Explain.

They must be able to write optimizers that can successfully schedule the code to avoid the "new" stalls. A compiler writer of average skill should be able to schedule away the stalls in the sample code above. Other situations are more difficult.
(b) If mux-in-ID is faster than mux-in-EX using the old compilers, do compilers still need to be re-written? Explain.

Yes. The old code might be faster because fewer than $10 \%$ of instructions use a source register produced by the immediately preceding instruction. Suppose that number were $5 \%$. Then re-writing the compiler could reduce or eliminate stalls due to these instructions, yielding further performance gains.

56 Spring 2008 Solutions

Problem 1: Solve Fall 2007 Homework 2 without looking at the solution. Then look at the solution and give yourself a grade on a scale of $[0,1]$. Warning: test questions are based on the assumption that homework problems were completed, so make a full effort to solve it without first consulting the solution.

Problem 2: The MIPS IV movn instruction is an example of a predicated instruction (predication will be covered later in the semester, but that material is not needed to solve this problem).
(a) Show how the movn instruction could be added to the implementation below inexpensively, but without impact on critical path. Take into account the new logic's impact on dependency testing (see the code sample below). Show all added control logic.

Changes for the movn instruction are shown in blue bold below. The logic shown in green was put in for the solution but would have appeared on the unsolved diagram if control logic were shown. That is, the green logic would be there with or without the movn.

The movn implementation below works as follows. The output of =movn box in ID is 1 if a movn instruction is present. In EX the ALU passes the rs value unchanged while the added $=0$ unit tests whether the $r t$ value is zero. The and gate checks whether a movn instruction is in EX and whether the move should be cancelled, if so the mux substitutes a 0 for the destination register (suppressing the writeback), otherwise the dst register number is passed through unchanged. Note that the control logic for detecting bypasses examines the output of the mux.

This implementation will execute the code below without a stall.
In a lower-cost implementation (not illustrated) a comparison unit in the ID stage, already needed for branches, would be used. The code below would stall on such an implementation unless bypasses were added from EX.

(b) Show how the code below would execute on your implementation.

```
# Solution
# Cycle }\quad0\quad
    add r1, r2, r3 IF ID EX ME WB
    movn r4, r5, r1 IF ID EX ME WB
    xor r6, r4, r7 IF ID EX ME WB
```

(c) Suggest methods to eliminate any stalls encountered.

There are no stalls.

For the answers to these questions look at the ARM Architecture Reference Manual linked to the course references page, http://www.ece.lsu.edu/ee4720/reference.html.

Problem 1: The register fields in ARM instructions are four bits and so only 16 integer registers are accessible. The ISA manual describes ARM as having 32 integer registers, however many of them are only accessible in particular modes.

An advantage of fewer registers is that extra bits are available in the instruction encoding, for example, ARM three-register instruction formats would have three more bits available than the MIPS type R format. Where in the ARM formats do you think these bits went? In your answer give the instruction field and its purpose. There should be no equivalent in MIPS.

Every instruction format uses a cond (condition field), there is no counterpart to this in MIPS. The condition field is used to predicate instructions, that is, control whether or not an instruction has any effect.
(See section A3-1, which conveniently lists the instruction coding for many instructions.)
Problem 2: In MIPS an arbitrary 32-bit constant can be loaded into a register using a lui followed by an ori. In ARM the immediate field for data-processing (integer) instructions is only 8 bits.
(a) Show ARM code to put an arbitrary 32 -bit constant into a register without using a load instruction. Use as few instructions as possible. Hint: take advantage of ARMS shift and rotate capabilities.

A move followed by three or's with shifts can do the trick.

```
# Solution:
# Note: The arbitrary constant is 0x12345678
mov r1, # 0x78, 0
orr r1, r1, # 0x56, 12
orr r1, r1, # 0x34, 8
orr r1, r1, # 0x12, 4
```

(b) Show how ARM can put an arbitrary constant into a register with one load instruction, whereas in MIPS two would be required. The MIPS code is shown below. Do not assume the address of the constant is already in a register, that would make this problem insultingly easy! Hint: Use one of ARM's special purpose registers.

```
.text
    lui r1, 0x1111
    lw r1, 0x2220(r1)
    # ... a few more instructions ..
    jr $ra
    nop
.data
my_32_bit_constant: # Address: 0x11112220
.word 0x12345678
```

Solution shown below. As in the MIPS example the constant is stored in memory near the code. MIPS code requires two instructions, one to load the high 16 bits of the address, the second to load the data (using the load offset for the low 16 bits of the address). In ARM the program counter is one of the data processing (general purpose in MIPS) registers, r15. This makes something like a lui unnecessary in ARM because the program counter can serve as the load's base
register. The code below is in pseudo assembly language, the assembler would convert $-8+$ my_32_bit_constant

- HERE into the correct offset.
\# Solution
HERE:
ldr r1, [r15 - 8 + my_32_bit_constant - HERE ]

Problem 3: In ARM the program counter is register r15, and so as far as instruction encoding goes, is treated as a general-purpose register.
(a) Why would really keeping the program counter in the integer register file add to the cost of an implementation?

Because to maintain an execution rate of one instruction per cycle one would need an additional read port and an additional write port on the register file to accommodate the program counter.
(b) How does the ISA manual hint that blue parts of the implementation below is what they had in mind? (Register r15 is not stored in the register file, it will always be bypassed from the real PC.) (Note: The ARM implementation is far from complete and parts may not work.)

Because instructions that use their source operands in the EX stage (ordinary arithmetic and logical instructions and address operands for loads and stores) get not the value of PC , but $\mathrm{PC}+8$, which is what you'd get in the diagram below. According to the ARM ISA stores of PC might result in PC+12 being written, which is consistent with the memory stage being three stages ahead of IF.


For answers to these questions consult the SPECcpu2006 Run and Reporting Rules (which can be found at spec.org).

Problem 1: One way testers can stretch the rules is by using compiler optimizations that give good performance when they work correctly but are too error prone for non-experimental use.
(a) Why would it be a bad idea for SPEC to limit allowable compiler optimizations to those that are already known to be safe? (Say, dead-code elimination based on a SPEC-provided analysis technique.)
(In the question "known to be safe" is not the same as "safe." A tester might have good reason to believe that an optimization is safe but such an optimization is not known by SPEC to be safe and so, based on the question above [but not in real life], could not be used.)

Limiting allowable optimizations to a short, conservative list would discourage development of new compiler optimization techniques. Compiler optimization is a perfectly legitimate way of achieving performance goals (and is in fact preferred over elaborate hardware techniques).
(b) Rather than dictate allowable optimizations the rules instead explain that if it's good enough for your customers it's good enough for SPEC, though not in those words. Find the section in the run and reporting rules where this rule is given.

Section 1.3.2.
(c) For at least three bullet items in the section (from the last part) explain what sort of unscrupulous action the bullet item is supposed to prevent.

- be specified using customer-recognizable names

The compiler is available and the company would sell it to any customer than can provide its name, but the name is kept secret from the customer.

- be generally available within certain time frames

The compiler is never made available.

- provide documentation

The compiler can't be used because it is undocumented.

- provide an option for customer support

The compiler can't be used because there is no support if a customer can't figure out how to use it.

- be of production quality

The optimizations are too buggy for reasonable use.

- provide a suitable environment for programming

The optimizations can only de used for very narrow purposes (the particular benchmarks).
Problem 2: When preparing a run of the SPEC benchmark the tester provides, among other things, libraries (such as the C standard library that contains routines such as strlen, malloc, printf). It is in the testers interest to make sure these library routines run as fast as possible and is free to do so within the SPEC rules.

Section 2.1.2 stipulates that one can't use flags that substitute library routines for routines defined in the benchmark.

In addition to base and peak, imagine a third metric called swap, in which the rule in Section 2.1.2 didn't apply. Testers could abuse the swap metric by substituting routines that merely return the correct value (since input data is known in advance), but for this question suppose testers
comply with the spirit of the SPEC rules and substitute routines which provide higher performance for any input data.
(a) Comparing the peak scores to the base scores shows the additional performance that can be obtained by a suitably motivated and resourced expert. Explain what might be learned by comparing swap scores to base and peak scores. (That is, where might the higher performance be coming from.)

If the benchmarks were well-written the swap result might show performance obtainable by structuring the computation for the implementation. For example, the code might be re-written so that it could use packed-operand (sometimes called multimedia) instructions, something a compiler couldn't always do because, for one reason, it doesn't know if using lower-precision and saturating arithmetic is okay.

If the benchmarks are not well written the swap result might show how poorly written they were. (That is, the re-writing would benefit many systems, not just the one it was re-written for.)
(b) Provide an argument that the swap metric is a good test of a system that complements base and peak.

The compiler is not smart enough to use some special instructions, such as packed-operand instructions, in many programs and so neither base nor peak would show the true potential of the system.
(c) Provide an argument that swap doesn't really tell you anything about the system (CPU, memory, compiler and other build items).

The swapped routines might improve the performance of any system and so the swap result would just show how many skilled programmers the tester was able to use to prepare the test.

Problem 3: For exceptions the handler needs to know the address of the faulting instruction both so that it can examine the instruction and so that it knows where to return to in case the instruction needs to be re-executed or skipped. For answers to this question consult the ARM and MIPS32 (Volume 3) ISA manuals on the course references page.

A programmer-friendly ISA would provide the handler with the address of the faulting instruction, however in both MIPS32 and ARM may provide an address near the faulting instruction.
(a) In which registers do MIPS and ARM A32 write the approximate faulting instruction address? (For MIPS give the register number as well as its name.)

MIPS writes the address to register c14 (co-processor 0 register 14), named EPC. ARM writes the address to r14.
(b) The address that MIPS provides may be that of the faulting instruction, or it may not be. When is this done, and what is the other address?

If the faulting instruction is in the delay slot of a CTI (control-transfer instruction) then register c14 is written with the branch address. The handler will be able to determine which instruction raised the exception, but it will return to the CTI, executing it a second time.
(c) ARM A32 also does not provide consistent addresses. What addresses does it provide? Give a credible reason for the differences in addresses.

Let PC denote the address of the faulting instruction if a load or store raised the exception r14 is written with PC +8 , for most other instructions it writes PC+4.

One reason for this inconsistency is that the implementation is expected to branch to the handler as soon as the exception is discovered, and for loads or stores the discovery might be one cycle later. The implementation does not bother sending an instruction PC down the pipeline so the exception mechanism uses the current PC value.

Note that ARM implementations would have to write these addresses whether or not the reasoning above is correct.

## 57 Fall 2007 Solutions

Homework 2 solution
Due: 1 October 2007
For lecture material relevant to this assignment see
http://www.ece.lsu.edu/ee4720/2007f/lsli06.pdf. For some background and a list of similar problems see the statically scheduled study guide, http://www.ece.lsu.edu/ee4720/guides/ssched.pdf. Please make an effort to solve this problem based on an understanding of the material, use the solution to similar problems (if any) only for hints. Feel free to ask questions using the forums, E-mail, or in person. Exam problems will be based on the assumption that students completed (really completed) homework assignments, so don't short-change yourself!

Problem 1: Consider the following MIPS code and implementation:

| \# Cycle | 0 | 1 |
| :--- | :--- | :--- |
| lw r2, 0(r10) | IF ID EX |  |
| LOOP: |  |  |
| lw r1, 0(r2) |  | IF ID |
| add r3, r1, r4 |  |  |
| sw r3, 4(r2) |  |  |
| bne r3, r5 LOOP |  |  |
| addi r2, r2, 8 |  |  |
| \# Cycle | 0 | 1 |


(a) Complete the pipeline execution diagram of the execution of the code above on the implementation illustrated for at least the first two iterations. (See the next part for instructions on the "A:", etc.)

Solution appears after part b, below. Note that the branch stalls due to a dependency on the add instruction which produces one of the branch source registers, r3.
(b) After the addi instruction three labels are shown, A:, B:, and C:; similar labels are shown, in blue and circled, in the implementation. On the pipeline execution diagram show the values on the wires (which are multiplexor inputs) that those labels point to only in cycles in which those signals are used. The values are already shown for cycles 0,1 , and 2 . Signals A and B are used in cycle 2 (but not 0 or 1 ), signal C is not used in cycles $0-2$.

Note that the multiplexor inputs are numbered from the top starting at zero.

```
    # SOLUTION
    # Cycle 
    lw r2, O(r10) IF ID EX ME WB
LOOP:
# First Iteration XX
    lw r1, O(r2) IF ID -> EX ME WB
    add r3, r1, r4 IF -> ID -> EX ME WB
    sw r3, 4(r2) IF -> ID EX ME WB
    bne r3, r5 LOOP IF ID -> EX ME WB
    addi r2, r2, 8 IF -> ID EX ME WB
# Cycle 
    A: 2 3 3 3 2 < 2 0 % 3 2 %
```



```
    C: 1 1 ...
# Cycle 
# Second Iteration XX
    lw r1, O(r2) IF ID EX ME WB
    add r3, r1, r4 IF ID -> EX ME WB
    sw r3, 4(r2) IF -> ID EX ME WB
    bne r3, r5 LOOP IF ID -> EX ME WB
    addi r2, r2, 8 IF -> ID EX ME WB
# Cycle 
# Third Iteration XX
    lw r1, O(r2) IF ID EX ME WB
```

(c) Find the CPI of this loop on the illustrated implementation for a large number of iterations.

In the first iteration the load in the loop body stalls because of a dependency with a load outside the loop, obviously that won't happen on subsequent iterations and so the first iteration is not representative. The second iteration starts at cycle 9 (when the first instruction is fetched), the third iteration starts in cycle 16 , and so the second iteration takes $16-9=7$ cycles. Both of these iterations start with the pipeline in an identical state: the addi is in ID, the bne is in EX, etc. Therefore the third iteration will take exactly the same amount of time as iteration 2 , as will all subsequent iterations. Therefore the CPI for a large number of iterations is $\frac{16-9}{5}=1.4$.
(d) Add bypass connection(s) so that the loop above executes as quickly as possible. Show the CPI with those connections.

The stalls in cycles 5 and 12 can't be eliminated by bypasses because the data arrives at the end of cycle 5 and 12, but it would be needed at the beginning of cycle 5 and 12 to avoid the stall.

The stall at cycles 8 and 15 can be eliminated because the data is available at the end of cycles 6 and 13 , and the branch needs it in the middle of cycles 7 and 14. The added bypasses, shown in green, eliminate the stall.

(e) Even with bypass connections the loop above, regrettably, executes with stalls (or at least it should!). Schedule (re-arrange) the code so that it executes without stalls. The scheduled loop should still load and store one value per iteration. Minor changes to the code can be made, such as changing register numbers and immediate values.

The code below executes without a stall with the bypasses added above.

```
# Scheduled Code
    lw r2, O(r10)
    lw r1, O(r2)
LOOP:
    # Cycle 0
    addi r2, r2, 8 IF ID EX ME WB
    add r3, r1, r4 IF ID EX ME WB
    sw r3, -4(r2)
    bne r3, r5 LOOP IF ID EX ME WB
    lw r1, O(r2) IF ID EX ME WB
    # Cycle }0
```


## LSU EE 4720

Homework 3 Solution
Due: 15 October 2007
The problems below ask about VAX instructions, which were not yet covered in class. For information on these instructions see the VAX Macro and Instruction Set manual linked to the EE 4720 references page.

Problem 1: The VAX locc instruction finds the first occurrence of a character in a string (see example below). The first operand specifies the character to find (A in the example), the second operand specifies the length of the string (in register r 2 ), and the third operand specifies the address of the first character of the string (register r3 below).

```
# Find first occurrence of 65 (ASCII A) in memory starting at
# address r3 and continuing for the next r2 characters.
locc #65, r2, (r3)
```

(a) Show how the sample instruction above is encoded. Include the name of each field and its value for the example above, not for the general case. In the original assignment the third argument was shown as r 3 , not (r3) which is correct.

Solution appears below. Note that immediate mode PC-addressing is used to specify the constant 65. In $P C$ addressing the register field of the operand specifier is set to 15 (the VAX PC is register 15 ), this changes the meaning of some of the modes. For general addressing (the register field is not 15) mode 8 is autoincrement mode, for PC addressing mode 8 is immediate mode.

```
SOLUTION:
Instruction: locc #65, r2, (r3)
Syntax: locc char.rb, len.rw, addr.ab
Sections: opcode immediate_mode_op register_mode_op register_deferred_op
opcode -> 8 bits: 0x3a
immediate_mode_op -> operand_specifier immediate
    operand_specifier -> mode(=immediate) reg(=PC) -> (4 bits) 0x8 (4 bits) 0xf
    immediate -> (8 bits) 0x41
register_mode_op -> operand_specifier -> mode(=register) reg(=2) -> 0x5 0x2
register_deferred_op -> operand_specifier
    -> mode(=register deferred) reg_num(=3) -> 0x6 0x3
Instruction Encoding:
```


(b) Provide an example of locc in which the encoded second and third operands each require more space than the example above. At least one of these operands should use a memory addressing mode that is not available in MIPS. Show the instruction in assembler and show its encoding.

The second operand now uses byte displacement deferred (shown as bdd below), and the third operand uses absolute addressing.

```
.data
STR_ADDR: # Assume address is 0x1234
.asciiz "My string."
.text
locc #65, @B^8(r2), @#STR_ADDR
```



For the problems below consider a MIPS implementation similar to the one illustrated below and a DFequivalent VAX implementation. Like the MIPS implementation, the DF-equivalent VAX implementation can read two registers per cycle, write one register per cycle, perform one ALU operation per cycle, and one memory operation per cycle (not including fetch). The DF-equivalent VAX implementation may or may not be pipelined and regardless does not suffer any kind of penalty for the complexity and size of its control logic. Assume that the DF-equivalent VAX takes one cycle to fetch an instruction and one cycle to decode an instruction, regardless of the instruction's size or complexity.

Unlike MIPS the DF-equivalent VAX may be able to simultaneously use its ALU and memory port for the same instruction (in the illustrated MIPS implementation they would be for two different instructions). The 2-read, 1-write register restriction only applies to registers defined by the ISA. As with MIPS pipeline latches, the DF-equivalent VAX can read or write as many temporary registers per cycle that it needs.

When showing the execution of an instruction on the DF-equivalent VAX use something like a pipeline diagram and explain what's going on when things aren't clear. For example, here is how an add instruction might execute:

```
# Note: Destination is rightmost register (r3)
Cycle }0
add 123(r1), (r2)+, r3 IF ID EX ME ME EX WB
sub IF ID EX
Cycle 2: EX: 123 + r1
Cycle 3: ME load (123+r1)
Cycle 4: ME: load (r2)
Cycle 4: EX: r2 + 4
Cycle 5: EX: add (123+r1) + (r2)
Cycle 5: WB: wb r2+4 to r2
Cycle 6: WB: WB sum to r3.
```

In the example above the add instruction can be said to have taken four cycles since that's how long the sub might have had to wait to execute (to avoid overlap).

Use the following MIPS implementation for comparison:


Problem 2: The MIPS jal instruction supports a procedure call by saving a return address in r31, other activities normally done on a procedure call, such as saving registers to the stack, must be performed using additional MIPS instructions. In contrast the VAX calls instruction not only saves a return address but also saves registers in the stack and performs other common activities.

MIPS and VAX examples are shown below in which the VAX code uses a calls instruction and the MIPS code performs a roughly equivalent operation. In particular, in both code samples three registers must be saved on the stack. (The calls instruction performs additional actions, but for this problem assume it does only what the MIPS code shows.)
(a) Show how the calls instruction would execute in the DF-equivalent VAX implementation. Note that the calls instruction reads the word at the beginning of the called routine to determine which registers to save.

Solution appears below. An xor is shown following the calls to show how long the calls would take.
(b) Is the DF-equivalent VAX implementation substantially faster on this instruction, about the same, or slower?

One cycle slower, because it has to check the mask to determine which registers to save.

```
# VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX VAX
    calls $O, myroutine
myroutine:
    .data
    .word 0x046
    xor ...
# MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS MIPS
    jal myroutine
    SOLUTION:
# Cycle 0
calls $0, myroutine IF ID ME EX ME ME ME
    RR RR EX EX EX WB
    RR RR
xor IF ID EX ...
Cycle 2: ME Load first word of myroutine, which specifies which regs to save.
        Word loaded into special 16-bit $m reg.
        Initial value of $m register is 0000 0000 0100 0110
            RR Retrieve fp register
Cycle 3: EX Set $r = clz($m) (Count number of leading zeros, $r will be 1)
            Compute: $addr = fp + ( $r << 2 ).
            RR Retrieve from register file: $rx = $r1
            At end of cycle set least-significant "1" bit of $m to 0.
            New value of $m: 0000 0000 0100 0100
Cycle 4: ME Store $rx ($r1) at address $addr
    EX, RR: Same operation as in cycle 3. (But $r will be 2)
    New value of $m: 0000 0000 0100 0000
Cycle 5: ME Store $rx ($r2) at address $addr
    EX: Same operation as in cycle 3. (But $r will be 6)
    RR: Read r6 and also $sp
    New value of $m: 0000 0000 0000 0000
Cycle 6: ME Store $rx ($r6) at address $addr
    EX Add $sum = $sp + 0
Cycle 7: WB Write $sum to register $fp
```

Problem 3: The VAX locc instruction is another example of an instruction that would not be included in a RISC ISA because it could not be pipelined in any reasonable way. For this problem assume that implementations of character location can only read one byte at a time. (A fast implementation might read a word and check each position for the sought byte, but not in this problem.)
(a) What is the minimum amount of time that the DF-equivalent VAX implementation might take to execute locc with a length parameter equal to $n$ ? Show how the instruction would execute.

The instructions appear below. Two cycles per character are needed because there is one comparison unit but two comparisons are needed: the character loaded and the character count. The worst-case time to find a character is $4+2 n$ cycles.

```
SOLUTION
# Cycle 0}0
    locc #65, r2, (r3) IF ID RR CM ME CM ME CM ... WB
    EX CM EX CM EX ...
                        EX EX ...
Cycle 1: ID: $char = lit. Assume literal addressing for character.
Cycle 2: RR: $len = $rx, assuming register addressing; $addr = $ry
Cycle 3: CM: Check if the $len is non-zero (if so proceed to X).
EX: Decrement $len
Cycle 4: ME: Retrieve byte at $addr.
    CM: Check if the $len is non-zero (if so proceed to X).
    EX: $len = $len - 1;
Cycle 5: CM: Check if byte equals $char, if so proceed to X.
    EX: $addr = $addr + 1
Cycle 6: ME: Retrieve byte at $addr.
    CM: Check if the $len is non-zero (if so proceed to X).
    EX: $len = $len - 1;
Cycle X: WB: Write condition code with 1 if char found, O otherwise.
```

(b) The MIPS routine below performs the same operation (except for the r0 and r1 return values). In terms of $n$ how long does it take to compute locc?
locc:
\# Call Values:
\# a0: char: Character to find.
\# a1: len: Length of string.
\# a2: addr: Address of first character of string.
\# Return Value:
\# v0: 0 if character not found, 1 if found.
\# Note: Other locc return values not computed.
j START
add \$t1, \$a1, \$a2 \# \$t1: Stop address ( last char + 1 )
LOOP :
beq \$t0, \$a0 FOUND
addi \$a2, \$a2, 1
START:
bne \$a2, \$t1, LOOP
lb \$t0, 0(\$a2)
jr \$ra
addi \$v0, \$0, 0
FOUND:
jr \$ra
addi \$v0, \$0, 1

From the diagram below it can be seen that an iteration takes 8 cycles (cycle 6 to 14), and so the routine takes $4+8 n$ cycles to find the character in the worst case (when the character is not in the string).

SOLUTION: Analyze the loop:

```
LOOP: beq $t0, $aO FOUND IF ID EX ME WB
    addi $a2, $a2, 1
START: bne $a2, $t1, LOOP
    lb $t0, 0($a2)
# Cycle
LOOP: beq $t0, $aO FOUND IF ID ----> EX ME WB
    addi $a2, $a2, 1
START: bne $a2, $t1, LOOP
    lb $t0, 0($a2)
LOOP: beq $t0, $aO FOUND
    IF ID EX ME WB
        IF ID ----> EX ME WB
        IF ----> ID EX ME WB
    0}1
    ID ----> EX ME WB
# Cycle 0
```

(c) Which implementation has the speed advantage? Explain.

Based on the given code fragments, the VAX. However the MIPS code can be re-written to reduce execution time on the given implementation. For example, the code below, an unrolled version of the code above, searches at the rate of $\frac{10}{4}$ cycles per character.
$\begin{array}{lll}\text { lb } & \text { \$t0, } & 0(\$ \mathrm{a} 2) \\ \text { lb } & \$ \mathrm{t} 5, & 1(\$ \mathrm{a} 2) \\ \text { lb } & \$ \mathrm{t} 6, & 2(\$ \mathrm{a} 2)\end{array}$
LOOP: beq \$t0, \$a0, FOUND
lb \$t7, 3(\$a2)
beq \$t5, \$a0, FOUND
addi \$a2, \$a2, 4
beq \$t6, \$a0, FOUND
lb \$t0, 0 (\$a2)
beq \$t7, \$a0, FOUND
lb \$t5, 1 (\$a2)
bne \$a2, \$t1, LOOP
lb \$t6, 2(\$a2)

IF ID EX ME WB
IF ID EX ME WB IF ID EX ME WB

IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
IF ID EX ME WB
(d) Can instructions be added to MIPS consistent with RISC principles that would substantially improve its performance? If not, explain what gives locc an inherent advantage on CISC.

Auto-increment addressing would save one instruction. This problem specifically disallows loading a word. If it were allowed an instruction could test each byte position for a match.

Homework 4 solution

Problem 1: For answers to this problems consult the SPARC Architecture Manual Version V8, linked to the course references page.

Suppose a SPARC V8 trap table has been set up at address $0 \times 12340000$
(a) Write a SPARC V8 program that sets the trap base register (TBR) to that address. Assume the processor is already in privileged mode. Hint: A correct solution consists of two instructions, a three-instruction program is okay too.

Browsing the SPARC Architecture Manual Version V8 for information on the trap table one should soon come across the wrtbr (write trap base register) instruction. The instruction is used in the code fragment below.
! SOLUTION
sethi \%hi (0x12345000), \%10
wrtbr \%10, 0, \%tbr

Call the SPARC V8 instruction that writes the TBR foo. The ISA definition of foo makes it easy to design the control logic and bypassing hardware on certain implementations.
(b) What about the definition of foo makes the control logic and bypassing hardware design easy on those certain implementations?

The V8 architecture manual definition of wrtbr states that it is a delayed-write instruction, meaning that if any of the next three instructions attempt to read the TBR the result is undefined. (See the third paragraph on page 134 of the posted version of the manual.)

A five-stage implementation similar to the one used in class would not need to check for true dependencies with the TBR, nor would bypass paths be needed, reducing the cost. For example, in the code below the TBR is written and then read (using rdtbr). According to the definition the code below would be correct regardless of what the first rdtbr writes in g 4 (because it is within 3 instructions of the most recent wrtbr). The same is true for the rdtbr that writes g5. Therefore the implementation does not need to check for a dependency between rdtbr and preceding instructions in the pipeline. The value placed in g6 by the last rdtbr must be the value written to the TBR by the wrtbr, for this five-stage implementation that requires no special hardware because the TBR write is complete when the last rdtbr reaches ID.

```
! Cycle 0
wrtbr %l0, 0, %tbr IF ID EX ME WB
add %g1, %g2, %g3 IF ID EX ME WB
rdtbr %tbr, %g4
rdtbr %tbr, %g5
rdtbr %tbr, %g6
        IF ID EX ME WB
        IF ID EX ME WB
                                IF ID EX ME WB
```

(c) Why not do the same for, say, the add instruction?

Because it would be difficult to schedule code without slowing execution by adding lots of nop instructions. The first code sample below shows ordinary MIPS code. With sufficient bypass paths the code should execute without a stall on a scalar 5 -stage statically scheduled implementation. In the second code fragment there must be at least a 3 -instruction separation between an instruction that writes a register and the instruction that reads it. To maintain correctness under that restriction nop's were added, slowing down execution.

It's okay to impose the three-instruction separation restriction on rarely used instructions, such as wrtbr because the impact on performance will be tiny. Imposing such a restriction on frequently executed instructions would have too large an impact on execution, not worth the small savings in control and bypass logic.

```
# MIPS as is: No restriction on placement.
LOOP:
    lw r1, O(r2)
    addi r2, r2, 4
    and r1, r1, r3
    bne r2, r4 LOOP
    add r5, r5, r1
# MIPS with 3-insn separation:
LOOP:
    lw r1, O(r2)
    addi r2, r2, 4
    nop
    nop
    and r1, r1, r3
    nop
    nop
    bne r2, r4 LOOP
    add r5, r5, r1
```

(d) Describe an implementation in which the control logic for foo would not be so simple despite the "help" from the ISA definition.

Any implementation in which wrtbr writes the TBR after the fourth following instruction reads it. This can definitely occur in a 5 -way superscalar implementation. In such an implementation logic would be needed to detect the dependency, or else always assume there is such a dependency and stall the pipeline after every wrtbr instruction.

Problem 2: Solve the EE 4720 Spring 2007 Final Exam problem 1.
Problem 3: Solve the EE 4720 Spring 2007 Final Exam problem 3.

58 Spring 2007 Solutions

Problem 1: Without looking at the solution solve Spring 2002 Homework 2 Problem 2 parts a-c. Then, look at the solution and assign yourself a grade in the range [0,1].

Problem 2: If the value in register r 2 is not aligned (a multiple of four) the 1 w in the MIPS code below will not complete.

```
lw r1, O(r2)
```

(a) Re-write the code so that r 1 is loaded with the word at the address in r 2 , whether or not it is aligned. For this part do not use instructions lwl and lwr (see the next part).

Solution shown below.
Grading Note: No one submitted a solution like the one below, that is, one that (correctly) combined data from just two lw instructions. Most submitted solutions used four lb instructions, some of these used shift and OR instructions to insert each loaded byte into what would be the full word, one solution followed each lb by $a \mathrm{sb}$, the sb instructions were relative to an aligned address.

```
# Solution shown below.
# Comments show register contents for this example:
#
# r2 = 0x1001
# Memory contents:
# Mem[0x1000] = x00, Mem[0x1001] = x11, Mem[0x1002] = x22, .. Mem[0x1007] = x77
# Therefore, want r1 = 0x11223344.
#
andi r4, r2, 3 # Extract "misalignment", m. r4 = 1 (call this m)
sub r5, r2, r4 # Round down to aligned address. r5 = 0x1000
lw r6, 0(r5) # Load first part. r6 = 0x00112233
# This has 4-m = 3 bytes we need.
lw r7, 4(r5) # Load second part. r7 = 0x44556677
sll r4, r4, 3 # Shift amt for 1st part. r4 = 8*m = 8.
sll r6, r6, r4 # Shift 1st part into place. r6 = 0x11223300
addi r8, r0, 32 # Constant for 2nd shift amt.
sub r8, r8, r4 # Shift amt for 2nd part. r8 = 32 - 8m = 24.
srl r7, r7, r8 # Shift 2nd part into place. r7 = 0x00000044
or r1, r7, r6 # Finally, combine them. Voila! r1 = 0x11223344
```

(b) Re-write the code, but this time use MIPS instructions lwl and lwr. Hint: These instructions were not covered in class, try looking them up in the MIPS architecture manual conveniently linked to the http://www.ece.lsu.edu/ee4720/reference.html page..

The solution is shown below.
Grading Note: About the only wrong solution was my own (the one below is correct), in which the lwr used the same offset as the lwl. I hope at least some got the answer correct because they carefully read the lwr description and saw that the effective address for this instruction is of the least significant byte of the word, not the most significant byte required of $1 \mathrm{wl}, 1 \mathrm{w}$, 1 h , and 1 hu .

```
# Solution:
#
lwl r1, O(r2)
lwr r1, 3(r2)
```

Problem 3: Consider how the lwl and lwr instructions might be added to the implementation below. There are two pieces of hardware that with minor modification would be able to merge the sub-words in a reasonable solution. Alternatively, a new piece of hardware to perform the merge can be added.

(a) Show how the hardware can be modified.

- If your solution relies on an existing component to perform the merge indicate which component and why that can be easily modified to do the merge.
- As with other MIPS instructions lwl must spend one cycle in each stage (except when stalling).

An implementation must merge some bytes of the value loaded from memory with the existing register contents.
Here are three reasonable solutions:
Merge in memory port: For an ordinary load instruction the data-in to the memory port in the ME stage has the rt register value, which is ignored. Since the memory port "automatically" gets the old value and it retrieves the new value, have it merge the two together. Some of the hardware needed to do that, namely hardware to shift the loaded value into the correct place, is already there. (More on this later in the semester.) Making whatever changes are necessary to do the merge might not add to the critical path because most of the hardware is already there.

Merge in the WB stage using new hardware: The new hardware would have as inputs the WB. MD pipeline latch, a new WB.rtv pipeline latch, the low two bits of WB. alu (for alignment), and control bits. The output would connect to a new input of the WB mux. Because of the time needed for the merge with this alternative it might not be possible to bypass to EX from WB.

Merge within the register file: A slightly modified memory port shifts the loaded value into the correct position (it already does that for lb and lh ) and the loaded value reaches the register file following the existing data path. A modified register file will write only those bytes of the register which are to change, the others will remain unchanged. New control logic (pernaps placed in the WB stage) will examine address bits to see which bytes of the register to write.

Here is a solution that would work but would be too slow:
Merge in the ME stage (not the mem port): New hardware merges the output of the memory port and ME.rtv. (The hardware would otherwise be the same as the merge-in-the-WB-stage option above.) This would be too slow because memory is considered to be on the critical path and anything delaying a signal between, say, the memory port out and the pipeline latch would be lengthening the critical path.
(b) Show how the code below would execute on your solution. Pay attention to dependencies. Feel free to propose an alternate solution to reduce the number of stalls.

```
# Execution for merge-in-memory-port solution.
# Cycle 0
add r1, r3, r4 IF ID EX ME WB
lwl r1, O(r2) IF ID ----> EX ME WB
sub r5, r1, r6 IF ----> ID -> EX ME WB
```

Problem 4: Consider these options for handling unaligned loads in the MIPS ISA which might have been debated while MIPS was being developed.

- Option Lean: All load addresses aligned. No special instructions for unaligned loads (e.g., no lwl or lwr).
Pro: Simple implementation. Con: Slower handling of unaligned loads.
- Option Real: All load addresses aligned. Special instructions for unaligned loads (e.g., lwl or lwr). Pro: Quick handling of unaligned loads. Con: Small amount of additional data path complexity.
- Option Nice: Load addresses do not have to be aligned, however warn programmers that loads of unaligned addresses may take longer in some implementations.

Pro: Simpler and shorter code. Con: Larger amount of additional datapath complexity. Slower code if implementations do take Ionger.
(a) For each option provide an advantage and a disadvantage.

See above.
(b) What kind of data would be needed to choose between these options? Consider both software and hardware data, be reasonably specific.

One needs to weigh the cost against the benefft for a typical implementation. The cost is the engineering effort and chip area needed for each option, the beneft is how much faster code will run.

For the cost one needs an estimate of how large and how complex the additional hardware would be.
For the benefit one would need to know how often code performs unaligned access. That data could be obtained by finding an existing ISA that imposes alignment restrictions and then analyzing benchmarks compiled for that ISA to determine how frequently they perform unaligned accesses. Then estimate the relative performance of the three options.

Evaluating the nice option requires another piece of data: how many programmers would avoid unaligned access because it might be slower. Gentle reader, I'm sure you would-or if you didn't you'd have a good reason, but what about the typical programmer? If programmers made no effort to avoid unaligned accesses then their code would be slower on some implementations. Fortunately, most machine language is "written" by compilers and compiler writers would likely pay attention to the warning, so for this analysis assume in nice option unaligned access is avoided as it should be.

Grading Note: Many answers discussed the conditions under which unaligned accesses would be made, rather than just suggesting measuring how much existing programs made unaligned access. Many solutions correctly noted that unaligned accesses are needed when data is to be packed tightly (avoiding the padding needed to meet alignment requirements). Nevertheless its much better to directly measure how often something is done (when possible) then to estimate how often it would be done. In this case because one would still have to gauge how often misalignment was actually necessary to save space (this would happen with arrays of structures of certain members (say, one integer and one character). One also might not account for unanticipated reasons for unaligned access.
(c) Using made up data pick the best option. Any choice would be correct with the right data.

Made up cost data: Cost of lean option, 0 ; cost of real option, 10 person-weeks +1000 gates; cost of nice option, 20 person-weeks +1200 gates.

Best proposed feature (nothing to do with unaligned loads): $5 \%$ improvement using 10 person-weeks and 1000 gates.
Made up benchmark data: Over all benchmarks, $20 \%$ of instructions are loads. Of these, $0.0001 \%$ are unaligned. With lean option each unaligned load would take about 5 times as many instructions as real option but that would still have negligible impact on performance, so choose lean.

Different made up benchmark data: Over all benchmarks, $20 \%$ of instructions are loads. Of these, $10 \%$ are unaligned. Assuming instruction count is proportional to performance for lean and real Lean, $0.8+0.18+0.02 * 10=1.18$; real, $0.8+0.18+0.02 * 2=$ 1.02, so the performance improvement of real or lean would $13.6 \%$ faster, well over the $5 \%$ needed to justify the cost (compared to the best proposed feature). Assuming no penalty for unaligned access, the nice option would yield about a further $2 \%$ improvement, which would not be worth the cost.

Grading Note: Many solutions provided made-up data describing only a small code fragment. The idea was to pick the kind of data a CPU manufacturer would use to actually decide which option to pursue, and so it would have to be based on real benchmarks or something equally convincing.

Problem 1: A manufacturer develops an ISA extension which can dramatically improve the performance of certain benchmarks. The extension includes new instructions which work well with small integers. In what the manufacturer calls well-formed C programs the compiler will find all opportunities where the new instructions can be used and so the dramatic improvement will be realized. On other programs in which the new instructions could be used the compiler won't use them because it can't tell if the resulting machine code would be correct (perhaps because its not sure if values in registers would be small). In such cases the compiler will provide a message for the programmer indicating a list of regions in which there was the possibility of using the instructions. The programmer can then recompile with a special option indicating which of those regions the new instructions can safely be used in. The resulting code would be sped up.

Suppose this all works out very well for developers. They have no problems indicating which regions are safe for the new instructions and their resulting executables are fast and run correctly.

The manufacturer would like to run the SPECcpu2006 benchmarks on their new implementation. Most of the SPECcpu2006 benchmarks are not well formed.
(a) Why couldn't the compiler options (flags) for the SPEC run (base or peak) indicate the safe regions under a reasonable interpretation of the rules? In your answer refer to specific parts of the SPECcpu2006 run and reporting rules, http://www.spec.org/cpu2006/Docs/runrules.html.

The rules state that compiler flags cannot use names. Strictly speaking, the list of regions provided by the compiler is probably not a list of names but what is being asked of the programmer is similar to what spec rule 2.1.1 forbids: the use of variable or subroutine names in optimization flags. Flags with variable or subroutine names might be used to tell the compiler to apply a dangerous optimization only to that code, such flags are probably forbidden because few programmers would make the effort to use them properly (especially when using the flags inappropriately would lead to incorrect execution). The compiler deseribed in the problem provides a list of regions and asks the programmer to make the same kind of decision, one that would require familiarity with both the optimization and with the code. For that reason the safe regions flags could arguably be forbidden under SPEC rules.
(b) Keeping in mind the goals of the SPECcpu benchmarks argue either that the SPECcpu rules should be changed (perhaps for a future version of the benchmark) or argue that the rules should remain as they are.

Argument for changing the rules:
SPECcpu is supposed to show the performance potential of new systems, including CPU, memory, and compilers. By providing a short list of regions the programmer is not burdened with scouring the code for special optimization opportunities, and so many programmers would use it. In fact, many programmers do use it and it would make no sense to forbidding optimization techniques that are becoming common practice in production environments.

Solve the problems below. Then look at the solutions and assign yourself a grade.
Problem 2: Without looking at the solution solve Spring 2006 Midterm Exam Problem 1.
Problem 3: Without looking at the solution solve Spring 2006 Midterm Exam Problem 2.

Some of the questions below are about the interrupt mechanisms defined for the MIPS32, SPARC V8, and PowerPC 2 ISAs. MIPS and SPARC interrupt mechanisms were covered in class, PowerPC's mechanism was not. All are documented in manuals linked to the class references page,
http://www.ece.lsu.edu/ee4720/reference.html. When using these references keep in mind that interrupt terminology differs from ISA to ISA and that you are not expected to understand (at least on first reading) most of what is in these manuals. Finding the right manuals and the relevant pages in those manuals is part of this assignment's learning experience.

Problem 1: Consider a load instruction that raises an exception due to a fixable problem with a memory address (for example, a TLB miss, whatever that is) on an implementation of MIP32, SPARC V8, and PowerPC 2.
(a) Where does each ISA say the address of the faulting instruction (the load) should be put? Give the exact register name, number, or both (if available).

MIPS32: Coprocessor 0 register EPC, register number 14. (If the faulting instruction is not in a delay slot then EPC is set to the address of the faulting instruction, otherwise the address of previous instruction).

SPARC V8: After advancing to a new register window, the address of the faulting instruction is put in register 11 (also called $r 17$ ). The address of the next instruction (which could be a CTI target) is put in register 12 (also called $r 18$ ).

PowerPC: The address of the load instruction is written to SRRO.
(b) Where does each ISA say to put the memory address that the load attempted to load from?

MIPS32: Coprocessor 0 register BadVAddr, register number 8.
SPARC V8: The memory management unit's (MMU) fault status register.
PowerPC: Register DAR.
Problem 2: Is PowerPC's equivalent of a trap table more similar to SPARC's trap table or to MIPS'? Explain and describe how specific elements are the same or different. Look at table placement, size, number of entries, and perhaps other characteristics.

Here is a summary of table characteristics.
MIPS32: The trap "table" is actually spread across several locations, defined by the ISA. At some locations there is a single handler, at others a set of four of handlers (as MIPS and PowerPC). Normally the cache exception table starts at 0xa0000000 and the table for other non-reset, non-debug exceptions starts at $0 x 80000000$. The number of entries under normal operations is about 8 , the smallest gap between entries is 32 instructions.

SPARC V8: Base address is value stored in TBR (trap base register), written by software during OS startup. Table size: 256 entries, each entry is four instructions.

PowerPC: Table starts at address 0 and spans 4096 characters. There are 15 entries with defined uses. Based on Figure 29 one might assume handler size is 32 instructions based on the smallest space between defined entries.

Comparison: SPARC has the largest table (256 entries) and one that can be placed anywhere. The benefit of many entries is that the handler might spend less branching to the appropriate code (because for each entry there is just one place to jump). (This is probably not that big an advantage since the table would not be used that often.)

PowerPC is similar to SPARC in that all entries are in one table, not spread across the address space as in MIPS. However like MIPS the location of PowerPC's table is defined by the ISA. The number of entries is closer to that of MIPS than SPARC. Overall the PowerPC table seems more similar to MIPS.

Grading Note: If reasonably argued, "closer to SPARC" would also be considered a correct answer.
Problem 3: In class a precise exception was defined as one in which, to the handler it appears that all instructions before the faulting instruction have completed normally and that the faulting instruction and those following it have not executed (correctly or otherwise). PowerPC calls certain exceptions precise even though they violate this rule. What are they and how is this violation justified in the manual?

Data segment exceptions raised by certain load and store instructions. Under some circumstances when the handler starts the faulting load or store will have written some registers, violating a condition normally associated with precise exceptions. The ISA "book" points out that what's important is the ability to re-execute the instruction, and so implementations can have such loads and stores write registers so long as no information needed to re-execute them is lost.

Problem 4: Solve Fall 2006 Final Exam Problem 1. Note: At the time this was assigned the solutions were not available.
see posted solution to exam.

Problem 1: Estimate performance of the 8-way superscalar statically scheduled MIPS implementations described here. All are five stages, as used in class, and always hit the cache, as has been the case in class so far. Some of the implementations have no fetch group alignment restrictions, which means any eight contiguous instructions can be fetched. Some impose a fetch group alignment restriction, meaning if a CTI target is address $a$ IF will fetch eight instructions starting at address $a \prime=8 \times 4 \times\left\lfloor\frac{a}{8 \times 4}\right\rfloor$ (for those preferring $\mathrm{C}: ~ \mathrm{aa}=\mathrm{a} \&{ }^{\sim} 0 \mathrm{x} 1 \mathrm{f}$ ). Instructions in $[a \prime, a)$ (or from aa to before a) will be squashed before reaching ID.

The implementations include a branch predictor that predicts when a branch is in IF, resolves (checks the prediction) when a branch is in ID, and if necessary recovers (squashes wrong-path instructions) when a branch is in EX. A branch is predicted when it is in IF and the prediction is used in the next cycle. Example 1 , below, illustrates a correct taken prediction. The correctness of the prediction is checked, resolved, when the branch is in ID; if incorrect the wrong-path instructions are squashed and the correct path instructions are fetched in the next cycle (when the branch is in EX). This is illustrated in Example 2 for an incorrect taken prediction.

Note that due to alignment restrictions (if imposed) and branch placement the number of useful instructions fetched in a cycle can vary, and that is something to take into account in the subproblems below. The examples below illustrate when instructions will be fetched and squashed but they do not show how many will be fetched in every situation.

```
# Example 1: Branch correctly predicted taken. Target fetched in next cycle.
#
# Cycle 0
beq r1,r2, TARG IF ID EX ME WB
nop IF ID EX ME WB
TARG:
    add r3, r4, r5 IF ID EX ME WB
    # Example 2: Branch wrongly predicted taken.
    # Target squashed, correct path (fall through) fetched in cycle after ID.
    # Cycle 0
    beq r1,r2, TARG IF ID EX ME WB
    nop IF ID EX ME WB
    sub r6, r7, r8 IF ID EX ME WB
TARG:
    add r3, r4, r5
        IFx
    # Cycle 
```

All implementations run the same program, which has not been specially compiled for the 8 -way machine. In the program, which has no floating-point instructions, any two data-dependent instructions have at least seven instructions between them. This avoids some stalls, assume that there are no other stalls in the superscalar implementation due to data dependencies.

Let $n_{\mathrm{i}}$ denote the number of dynamic instructions in the program and let $n_{\mathrm{b}}$ denote the number of dynamic instructions that are branches. For the questions below show answers in terms of these symbols and also show values for $n_{\mathrm{i}}=10^{10}$ and $n_{\mathrm{b}}=2 \times 10^{9}$. Assume that half of the times a branch is executed it is taken.
(a) Suppose the 8-way implementation has perfect branch prediction and has no fetch alignment restrictions. Approximately how long (in cycles) will it take to run the program. State any assumptions. Hint: Because
of the branches $i t$ 's $>\frac{n_{i}}{8}$.
First of all, if it weren't for branches execution would take $\frac{n_{i}}{8}$ cycles. Performance is lost when a branch is not in the penultimate slot in a group. For example, suppose a taken branch is the first of eight instructions in ID. Six of those eight would have to be squashed. Because there is perfect branch prediction none of the instructions in IF are squashed. Only when a taken branch is in the last slot do instructions in IF get squashed, the seven following the delay-slot instruction. If a taken branch is in position $i$ then the number of squashed instructions is $\left\{\begin{array}{cc}6-i & i<7 \\ 7 & i=7\end{array}\right.$, where $i=0$ is the first slot.

Since the program has not been specially compiled the branch is equally likely to be in any slot, so the average number of squashed instructions for a taken branch is 3.5 .

The execution time is then $\left(n_{\mathrm{i}}+\frac{n_{\mathrm{b}}}{2} 3.5\right) / 8$ cycles. For the sample numbers the execution time is $1.69 \times 10^{9}$ cycles. (b) Repeat the question above for a predictor that always predicts not taken (which essentially means no predictor).

If a branch outcome is not taken no instructions are squashed. If a branch in ID is taken then seven or eight instructions in IF must be squashed (seven when the branch is in the last slot). The number of squashed instructions for a branch in slot $i$ is $8+6-i$ for $0 \leq i<8$. The average is 10.5 and so the execution time is $\left(n_{\mathrm{i}}+\frac{n_{\mathrm{b}}}{2} 10.5\right) / 8$ cycles , for the sample numbers the execution time is $2.56 \times 10^{9}$ cycles. Note the difference between perfect branch prediction (previous subproblem) and no branch prediction. (c) Repeat the question above for a predictor with a $95 \%$ prediction accuracy. (Yes, that means $95 \%$ of the predictions are correct.)

There are four cases to consider. Predicted not taken, outcome not taken (call that nn), $n \mathrm{t}$, tn , and tt . In case $n n$ nothing is squashed. Case $n t$ is what was analyzed in part $b$, the average number of squashes was 10.5 . Case $t t$ was analyzed in part $a$, the average number of squashes was 3.5 .

For this part we need to find the number of squashes when a branch is predicted taken but its not (the tn case). Consider such a branch when it is in ID and assume the branch is not in slot 7 . Because the branch was predieted taken $6-i$ instructions in ID will be squashed. When the branch is resolved not taken all the instructions in IF (which are on the taken path) will be squashed. The total number of squashed instructions is then $8+6-i$ for $i<7$. It is possible to actually not squash the instructions in ID in this case, but here we will assume its too diffficult. (For one, because the branch is resolved at the end of ID so they'd have to be squashed in EX in the tt case.)

If the branch is in slot 7 then IF will be fetehing the delay slot instruction along with 7 which based on the prediction are to be squashed. If the hardware is smart enough these can de saved and so that zero instructions are squashed in the tn case if the branch is in slot 7 . The average number across all slots is 9.625 instructions squashed.

Assuming prediction accuracy is the same for taken- and not-taken branches one can find a weighted average over the four casses:

$$
\overbrace{0 \times \frac{0.95}{2}}^{\text {nn case }}+\overbrace{10.5 \times \frac{0.05}{2}}^{\text {nt case }}+\overbrace{9.625 \times \frac{0.05}{2}}^{\text {tn case }}+\overbrace{3.5 \times \frac{0.95}{2}}^{\text {tt case }}=2.17 \text { cycles. }
$$

The execution time is $\left(n_{\mathrm{i}}+n_{\mathrm{b}} 2.17\right) / 8$ cycles, for the sample numbers the execution time is $1.52 \times 10^{9}$ cycles. Note that this calculation uses the total number of branches, not just the taken ones.
(d) Once again, suppose the 8-way implementation has perfect branch prediction but now fetch is restricted to aligned groups. Approximately how long (in cycles) will it take to run the program. State any assumptions.

Each time a branch is taken 3.5 instructions near the branch will be squashed (see part a). Because fetch isn't aligned, on average 3.5 instructions at the target will be squashed, for a total of 7 instructions.

The execution time is $\left(n_{\mathrm{i}}+\frac{n_{\mathrm{b}}}{2} 7\right) / 8$ cycles, for the sample numbers the execution time is $2.13 \times 10^{9}$ cycles.

Problem 2: Consider a bimodal branch predictor with a $2^{10}$-entry branch history table (BHT).
(a) What is the prediction accuracy on the branch below with the indicated behavior assuming no interference. Assume that the pattern continues to repeat. Provide the accuracy after warmup.

```
0x1000 beq r1, r2 TARG t t t n t t n n n t t t n t t n n n ...
```

The counter values, prediction and outcome are shown below. Note that the counter values will repeat. The prediction accuracy is $4 / 9=.444$, which is not good at all.

```
    Counter: }\begin{array}{llllllllllll}{0}&{1}&{2}&{3}&{2}&{3}&{3}&{2}&{1}&{0}
```



```
    Prediction n n t t t t t t n
    Prediction wrong: X X X X X
```

Problem 3: Suppose that for some crazy reason it's important that the branch at address $0 \times 1000$ be predicted accurately, even if that means suffering additional mispredictions elsewhere. The result of this crazyness is the code below, in which the branch in HELPER is intended to help the branch at $0 x 1000$.
(a) Choose an address for HELPER so that $0 \times 1000$ is helped.

Choose the address so that HELPER and the branch at $0 \times 1000$ use the same entry in the BHT. (This isn't something that's ordinarily done.) Since the BHT has $2^{10}$ entries the address of HELPER must match $0 \times 1000$ in the lower 12 bits. One possible address is $0 \times 2000$.
(b) Given a correct choice for the address of HELPER, find the prediction accuracy of the branch at $0 \times 1000$.

```
    jal HELPER
    nop
0x1000:
    beq r1, r2 TARG t t t n t t n n n t t t n t t n n n m
    ...
    ....
HELPER:
    beq r1, r2 SKIP
    nop
SKIP:
    jr r31
    nop
```

The solution is worked out below. The counter is modified twice, before and after the branch (both times using its outcome). The accuracy is now $6 / 9=.667$, better but still not that good.

```
            Counter: 01 23 33 32 12 33 32 10 00 01
0x1000 beq r1, r2 TARG t t t n t t n n n t t t n t t n n n m
    Prediction n t t t t t t n n
    Prediction wrong: X X X
```


## 59 Fall 2006 Solutions

Problem 1: Section 2.2.2 of the run and reporting rules for SPECcpu2006, http://www.spec.org/cpu2006/Docs/runrules.html, specifies that the optimization flags and options used to obtain the base result must be the same for each benchmark (compiled with the same language, say C). Why must they be the same?

Base scores are supposed to reflect the amount of optimization effort a programmer would make under ordinary circumstances. In that case the programmer might not find a best set of optimization flags for each program. Even if they did, there is no workable way to write a rule that would distinguish between compiler switches found with ordinary and extraordinary effort, and so requiring switches to be the same on all benchmarks is a reasonable substitute.

Problem 2: Section 1.2.3 of the run and reporting rules for SPECcpu2006, http://www.spec.org/cpu2006/Docs/runrules.html, assumes that the tester is honest. Provide an argument that many of the run and reporting rules ignore this assumption, or at best are based on the assumption that the tester is honest but sloppy or unmotivated.

The rules are written to ensure that anyone (with the budget, skills, and time) can reproduce any disclosed result. So the rules don't assume the tester is honest, they assume that the tester doesn't want to be caught lying.

Problem 3: Find the SPECcpu2000 CINT2000 disclosure for the fastest systems using each of the chips below. All chips implement some form or superset of IA-32 (also known as 80x86). All of the implementations are superscalar, meaning they can sustain execution of more than one instruction per cycle. In particular, an $n$-way superscalar processor can sustain execution of $n$ instructions per cycle on ideal code, on real code the sustained execution rate is much lower (for reasons to be covered later in the course, such as cache misses). Some of the implementations are multi-cored. (A core is an entire processor and so a 2-core chip has two complete processors.)

- Pentium III, 1-core, 2-way
- Pentium 4, 1-core, 3-way
- Pentium Extreme, 2-core, 3-way
- Intel Core 2 Extreme X6800, 2-core, 4-way
- Opteron 256, 1-core, 3-way
- Athlon FX-62, 2-core, 3-way
(a) For each system list the following information:
- The peak (result) ratio (for the suite).
- The clock frequency.
- The gcc peak (result) run time (in seconds).
- The maximum number of instructions the system could have executed during the run of gcc assuming all cores were used.
- The maximum number of instructions the system could have executed during the run of gcc assuming one core was used.
- Execution efficiency assuming all cores were used: number of instructions executed divided by maximum number of instructions that could have been executed in the same amount of time. Assume that all systems run the same binary (executable) of gcc and make a guess at how many instructions would be executed when running the binary for the SPEC inputs.
- Execution efficiency assuming a single core was used: Same as previous value, except assume only one core used.
The information is listed in the table below. Those viewing this with Adobe Reader can view the SPECint2000 disclosure by clicking the names in the Chip column.

The Cores column shows the number of cores on the tested system, the Width column shows the decode width per core (the $n$ in $n$-way superscalar), the Peak column shows the SPECint2000 result (peak) ratio, the Clock column shows the clock frequency, and the gec time column shows the execution time of the gee benchmark. Values for the columns mentioned so far are found in the SPEC disclosure or in this assignment.

The next two columns, both headed Max Insn, show the maximum number of instructions that the respective processor could have executed in the time need for the run of the gee benchmark using all cores and one core. Those are found by multiplying the maximum number of instructions per second for the processor by the benchmark run time for each case; the formulæ are shown below the column heading.

To find the efficiency one must estimate the number of instructions executed in a run of the benchmark, call that number $I$. It can't be larger than any of the entries under Max Insn, and realistically will be much smaller (due to stalls and squashes). Since there is no way to tell exactly what $I$ is with what is given here, $I$ will be set to the minimum Max Insn of the single-core cases (because multiple cores don't help gec). That is $I=366 \times 10^{9}$ instructions based on the Core $2 \times 6800$ at 2.933 GHz .

The efficiencies are computed by dividing $I$ by Max Insn. Using this method the efflciency of the Core $2 \times 6800$ is 1 , meaning only that the X 6800 is the most efficient, not that it's perfect.

|  |  |  |  |  |  | All Cores |  |  |  |
| :--- | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: | ---: |
| Chip |  |  |  |  |  |  |  |  |  | Cone Core

(b) The execution efficiency computation was based on the assumption that the number of executed instructions was the same in all systems. Identify two systems for which this was more likely to be true and two systems where this was less likely to be true.

In all eases gec was compiled using the same source code and run using the same inputs, as provided by SPEC. What differs is the compilers (and other build tools) used to test each system, as well as how those compilers were used. The number of instructions are most likely identical where the compiler and options are most similar.

The Pentium Extreme and Core 2 X6800 systems both use the Intel Compiler 9.1 and MieroQuill SmartHeap Library 8.0. These are the exact same compiler and heap (malloc and friends) libraries. The compile flags for gec are: -fast shlW32M-80.lib PASS1=-Qprof gen PASS2=-Qprof use for the Pentium Extreme and -fast sh1W32M.lib PASS1=-Qprof gen PASS2=-Qprof use for the Core $2 \times 6800$. These are as similar as one could expect, the factor that might nevertheless result in different instruction counts are how the compiler will respond to the -fast flag. That tells the compiler to emit the fastest code for the host system, since the two chips are different that might result in different instruction counts.

The number of instructions are less likely identical when the compilers and options are less similar. The Pentium III system uses older version of the Intel compiler, 5.0, and heap library, 5.0 , so the generated code is more likely to differ.
(c) How much does a dual-core implementation improve the performance of gcc?

Not much, based on a comparison of the single-core Pentium 4 and the dual-core Pentium Extreme. The performance is nearly the same. (The mieroarchitectures of the two processors' cores are very similar, so it's not just that each core of the Pentium Extreme is half the speed of the single core of the Pentium 4.)

Problem 1: Show the changes to the MIPS implementation below needed to implement the SPARC V8 instructions shown in the sub-problems. (See the SPARC Architecture Manual linked to the course references page for a description of SPARC instructions.) Do not show control logic changes or additions. For this problem assume that SPARC has 32 general-purpose registers, just like MIPS. (In reality there are $16 n$, $n \geq 4$ general-purpose registers organized into windows. An integer instruction sees only 32 of these but using save and restore instructions a program can replace the values of 16 of them, the feature is intended for procedure calls and returns. To satisfy curiosity, see the description of register windows in the ISA manual.)

Register for CC bits; we is write enable.


For solution can use larger version on next page.
(a) Show the changes for the following instructions. The only changes needed for these are to bit ranges in the ID stage.

```
add %g1, %g2, %g3
sub %g4, 5, %g6
```

Changes shown in blue in the diagram. The instruction bits used for the two existing read ports on the ID-stage register file changed to $18: 14$ and $4: 0$, so that the SPARC rs1 and rs2 registers would be retrieved. The bits in to the ID-stage format-immed logic changed to $12: 0$, reflecting the SPARC immediate field.
(b) Show the changes needed for the store instruction below. This will require more than changing bit ranges.

```
st %g3, [%g1+%g2]
Changes shown in green in the diagram. The store instruction has three source operands so a third read port added to the register file. In EX, the MUX leading to the ME-stage Data In port now gets its input from rdv (the new register read port value).
```

(c) Show the changes needed to implement the instructions below. The alert student will have noticed the ALU has a new output labeled cc. That output has condition code values taken from the result of the ALU operation.

- Don't forget the changes needed for the branch target.
- The changes should work correctly whether or not the branch immediately follows the CC instruction.
- Cross out the comparison unit if it's no longer needed.
subcc $\% \mathrm{~g} 1, \% \mathrm{~g} 2, \% \mathrm{~g} 3$
bge TARG
Changes shown in purple in the diagram. The ce instructions write the condition-code register, which is like any other register and so is placed in the ID stage. The CC value is computed by the ce output of the ALU, and that is carried along the pipeline in new CC pipeline latches to the WB stage where a new CC register is written. That new register has a write-enable (we) input so that only ce instructions (such as subcc) will write it. The output of the ce register connects to the branch control logic, a ce value is bypassed from the EX stage so that a branch immediately after a ce instruction (as in the example above) doesn't have to stall.

SPARC branch instruction targets are computed as a displacement from PC rather than NPC, so IF/ID latch changed. The ID-stage branch target adder lower input changed to reflect the position and size of the displacement field in SPARC instructions, bits 21:0.

Grading Notes: Some submitted solutions have the ce register bits compared to the comp field in the branch instruction. That won't work because the comp field does not specify exactly what the ce bits should be. For example, be (branch equal to zero) just checks if the Z bit is set.

Some submitted solutions show the CC register being written in the instruction's EX stage. That won't work because the instruction may be squashed after EX (and so it would not easily be possible to recover the old ce value). Given what's been covered so far it may seem like a squash won't happen to an instruction that reaches EX, but that will change when we cover exceptions.

Problem 1: The floating point pipeline in the MIPS implementation illustrated below must sometimes stall instructions to avoid the WF structural hazard. The WF structural hazard could be avoided by requiring all instructions that use WF to go through the same number of stages. Note that instructions that use WB all pass through five stages, even though some instructions, such as xor, could write back earlier.

Redesign the illustrated implementation so that the WF structural hazard is eliminated by having WF instructions (consider add.d, sub.d, mul.d, and lwc1) all pass through the same number of stages. The functional units themselves shouldn't change (still six multiply steps and four add steps) but their positions might change.
(a) Show the possibly relocated functional units and their connections. Don't forget connections for the lwc1 instruction.

The add unit steps now share stages with like-numbered multiply unit steps, see the illustration below. After A4 the FP add result is ready but it continues down the pipeline so that it reaches WF two cycles later than usual. There is no way there can be a WF structural hazard with a add. d and a mul. d because such a mul . d would have to be in ID in the same cycle as an add.d.

The result of a lwc1 joins the FP pipeline at the A3 stage. As with the add. d, the lwc1 cannot have a structural hazard with a mul .d or add.d. The add and load results are combined using a mux in the M5 stage. This reduces the number of pipeline latches used and also simplifies the control logic.
(b) Show any changes to the logic generating the fd , we, and xw signals. Note: The original assignment did not ask for xw changes.

Because there can be no WF structural hazards all of the structural hazard logic has been eliminated. Because the fd and we signals now only enter in ID the logic is much simpler. As before, we is set to 1 if any instruction using WF is present, which for this implementation means an instruction using the FP Add Unit, FP Multiply Unit, or a FP load instruction. The $f d$ signal is just the output of Decode dest. reg. logic.

The WF three-input mux in the original design has been replaced by two two-input muxen: one in M5 and one in WF. This simplifies the control logic.
(c) Show bypass paths needed to avoid stalls between any pair of floating point instructions mentioned above.

The solution below shows the pre-existing (but not usually shown) bypass path in green, and new bypass paths for this problem in blue. In the sample execution below bypasses are used in cycle 4 (ldc1 to add.d) and cycle 8 (add.d to sub.d).

```
# Sample Execution:
# Cycle 0
    ldc1 f2, O(r1) IF ID EX ME _3 _4 _5 _6 WF
    add.d f0, f2, f4 IF ID -> A1 A2 A3 A4 _5 _6 WF
    sub.d f6, f0, f8 IF -> ID -------> A1 A2 A3 A4 _5 _6 WF
```



Problem 2: Consider the changes to avoid structural hazard stalls from the previous problem. Provide an argument, either for making the changes and or against making the changes. For your argument use whatever cost and performance estimates can be made from the previous problem. Add to that the results of fictitious code analysis experiments and alternative ways of using silicon area to improve performance.

The code analysis experiments might look at the dynamic instruction stream of selected programs. For these experiments explain what programs were used and what you looked for in the instruction stream. Make up results to bolster your argument.

For the alternative ways of using silicon area, consider other ways of avoiding the structural hazard stalls, or other ways of improving performance. This does not have to be very detailed, but it must be specific. (For example, "use the silicon area for pipeline improvement" is too vague.)

The argument should be about a page and built on a few specific elements, rather than meandering long-winded generalities.

It's not worth it. Improvement is limited to a handful of programs, while the cost is substantial.
The benefit is only practically realized for a few FP-dense programs. Normally our compilers will schedule (arrange) FP instructions so that WF structural hazards are avoided. When analyzing SPECcpu2000 FP programs, we found only one program in which WF structural hazards remained after compiler scheduling. Even so, the resulting stalls caused only
a $10 \%$ increase in execution time and a competent program could have re-cast that region of code to avoid any stalls. Note: The remarks above about "our compilers" are fictional.

The modified hardware includes three new pipeline latches (M3/M4 to M5/M6) plus their bypass connections.
The added cost might be used to increase the L2 cache size, beneffing all programs. Another option would be to add a second FP register write port, though that would add to the complexity of the control logie.

Problem 3: In the previous problem structural hazards were avoided by having all WF instructions pass through the same number of stages. If both WB and WF instructions passed through the same number of stages then, were it not for stores, it would easily be possible for floating-point instructions to raise precise exceptions without added stalls (even if exceptions could not be detected until M6).
(a) For this part, ignore store instructions. Explain why having all instructions pass through the same number of stages makes it easier to implement precise exceptions (without added stalls, etc.) for floating point instructions.

Implementing precise exceptions for FP instructions is difficult because FP instructions write back out of order, so that should it be necessary to squash instructions following a faulting instruction some of those following instructions may have already written back. If all instructions use the same number of stages then instructions will write back in order and so it will always be possible to squash instructions following a faulting instruction.
(b) For this part, include store instructions. Explain how store instructions preclude precise exceptions for the implementation outlined above, or at least for a simple one.

Consider first an implementation in which the memory port is in M2 (as in the solution to the first problem). After a store instruction passes through M2 it will have written memory and so it will be too late to squash it, precluding precise exceptions.

If instead, the memory port is in M6 (the stage before WB) it will be possible to have precise exceptions but instructions dependent on loads will have to stall. See the example below.

```
# Alternate solution, ME in stage _6.
#
# Cycle 0
    ldc1 f2, O(r1) IF ID EX _2 _3 _4 _5 ME WF
    add.d f0, f2, f4 IF ID -----------> A1 A2 A3 A4 _5 _6 WF
    sub.d f6, f0, f8 IF -----------> ID -------> A1 A2 A3 A4 _5 _6 WF
```

(c) For this part, include store instructions. Do something about stores so that the all-instructions-use-the-same-number-of-stages implementation can provide precise exceptions to floating point instructions. It is okay if the modified implementation adds stalls around loads and stores. A good solution balances cost with performance.

If your solution is costly say so and justify it. If your solution is low cost but lowers performance say so and show the execution of code samples that encounter stalls.

One solution would be the ME in stage M6 variation shown above. Though the cost of this solution is low its need for frequent stalls would make performance too low.

Another solution would be to have the memory port provide the overwritten data on a store. That is, when a store instruction is in ME the Data Out of the memory port will hold the data which the store is about to overwrite. That data will proceed down the pipeline and ordinarily will just be disearded when the instruction reached WB. Going down the pipeline with the replaced data will be the effective address (this would require a new set of pipeline latches). If an instruction raises an exception, these replaced data and effective addresses will be used to write memory, undoing the effect of the stores.

60 Spring 2006 Solutions

Several Web links appear below and at least one is long, to avoid the tedium of typing them view the assignment in Adobe reader and click.

Problem 1: Consider these add instructions in three common ISAs. (Use the ISA manuals linked to the references page, http://www.ece.lsu.edu/ee4720/reference.html.)

```
# MIPS64
    addu r1, r2, r3
# SPARC V9
    add g2, g3, g1
# PA RISC 2
    add r1, r2, r3
```

(a) Show the encoding (binary form) for each instruction. Show the value of as many bits as possible.

Codings shown below. For PA-RISC the ea, ec, and ed fields are labeled e1, e2, and e3, respectively in the instruction deseriptions. There is no label for the eb field in the instruction description of the add and yes it would make sense to use e0 instead of e1 but they didn't for whatever reason.

(b) Identify the field or fields in the SPARC add instruction which are the closest equivalent to MIPS' func field. (A field is a set of bits in an instruction's binary representation.)

SPARC op3 is closest to MIPS func.
(c) Identify the field or fields in the PA-RISC add instruction which are the closest equivalent to MIPS' func field. Hint: Look at similar PA-RISC instructions such as sub and xor.

Fields ea (e1), eb, ec (e2), and ed (e3) are closest to the func field. The "e" is for extension. (In the instruction descriptions the fields are called e1, etc, but in the instruction formats chapter they are called ea.)
(d) The encodings of the SPARC and MIPS add instructions have unused fields: non-opcode fields that must be set to zero. Identify them.

For SPARC the asi field must be set to zero. The i field must also be set but that determines whether the instruction uses a register or immediate, so it's not unused. In the MIPS instruction the sa field is unused.

Problem 2: Read the Overview section of the PA-RISC 2.0 Architecture manual, http://h21007.www2.hp.com/dspp/files/unprotected/parisc20/PA_1_overview.pdf.
(If that link doesn't work find the overview section from the course references page, http://www.ece.lsu.edu/ee4720/reference.html.)

A consequence of the unused fields in MIPS and SPARC add instructions and RISC's fixedwidth instructions is that the instructions are larger than they need to be.

The PA-RISC overview explains how PA-RISC embodies important RISC characteristics, as do other RISC ISAs, but also has unique features of its own.
(a) It is because of one of those class of features that the PA-RISC 2.0 add instruction lacks an unused field. What is PA-RISC's catchy name for those features?

Pathlength Reduction. Rather than let the $c, f$, and $d$ fields go to waste, PA RISC uses them to include additional functionality, conditionally squashing the next instruction. With the additional functionality some programs would need fewer instructions, hence the term pathlength reduction.
(b) Provide an objection (from the RISC point of view) to the added functionality of PA-RISC's add instruction. If possible, find places in the overview that provide or at least hint at counterarguments to those objections.

Adding functionality complicates the pipeline, increasing engineering time and reducing chip area available for things like caches. Counterarguments might be found in the second instruction where they argue "reduced" should not be considered above any other quality.

Problem 3: Many ISAs today started out as 32 -bit ISAs and were extended to 64 bits. Two examples are SPARC (v8 is 32 bits, v9 is 64 bits) and MIPS (MIPS32 and MIPS64). One important goal is that code compiled for the 32 -bit version should run unchanged on the 64 -bit version. Another important goal is to add as little as possible in the 64 -bit version. For example, it would be easy maintain compatibility by adding a new set of 64 -bit integer registers and new 64 -bit integer instructions, but that would inflate the cost of the implementation. Another approach would be to extend the existing 32 -bit integer registers to 64 bits and change the existing instructions so they now operate on 64 -bit quantities, but that would break 32 -bit code (consider sll followed by srl).
(a) Does a MIPS32 add instruction, for example, add \$s1, \$s2, \$s3, perform 64-bit arithmetic when run on a MIPS64 implementation? If not, what instruction should be used to perform 64 bit integer arithmetic?

No. The daddy instruction does 64 -bit integer arithmetic.
(b) Does a SPARC v8 add instruction, for example, add $\% \mathrm{~g} 2, \% \mathrm{~g} 3, \% \mathrm{~g} 1$, perform 64 -bit arithmetic when run on a v9 implementation? If not, what instruction should be used?

Yes.
Problem 4: Continuing with techniques for extending 32 -bit ISAs to 64 bits, consider the problem of floating-point registers. Both MIPS32 and SPARC v8 have 32 32-bit FP registers that can be used in pairs to perform 64-bit FP arithmetic. Both 64 -bit versions effectively have 32 64-bit FP registers, but using different approaches.
(a) Describe the different approaches.

MIPS takes the simpler approach: In mode 0 (FR 0) there are 32 32-bit FP registers, as in MIPS32. In mode 1 there are 3264 -bit FP registers and double-precision instructions can use odd-numbered registers.

In SPARC V9 there are 3264 -bit registers, numbered f0, $f 2, \ldots, f 62$; registers $\uparrow 1, \ldots$, $\mathfrak{f 3 1}$ are also defined but these overlap the even registers. Register numbers above 931 are encoded in double-precision FP instructions by putting the MSB of the register number in the LSB of the field. That is, a value of 1 in the register field (which would be illegal in SPARC V8 double precision instruction) actually refers to register f32; a value of 5 refers to f 96 , etc.

Problem 1: The code fragment below runs on the illustrated implementation. Assume the branch is always taken.
(a) Show a pipeline execution diagram covering execution to the beginning of the third iteration of the loop. see below.
(b) What is the CPI for a large number of iterations?

Hint: Pay close attention to dependencies and carefully add the stalls to handle them; also pay close attention to the timing of the branch. Work from the illustrated implementation, do not adapt the solution from a similar past assignment, that would be like preparing for a 10 km run by driving around the jogging trail.

An iteration has four instructions. The first iteration takes $10-0=10$ cycles as does the second iteration: $20-10=10$ cycles. Both the second and third iterations start with the pipeline in the same state ( IW in IF , add in ME , bneq in WB ) and so the third iteration will be identical as will every subsequent iteration and so the CPI is $\frac{10}{4}$.

```
# Cycle:
0
LOOP:
    lw $s0, O($s1)
    addi $s3, $s0, 4
    bneq $s3, $0 LOOP
    add $s1, $s1, $s2
    xor $t0, $t1, $t2
    or $t3, $t4, $t5
# Cycle: 
    lw $s0, O($s1)
    addi $s3, $s0, 4
    bneq $s3, $0 LOOP
    add $s1, $s1, $s2
    xor $t0, $t1, $t2
    or $t3, $t4, $t5
IF ID EX ME WB
        IF ID ----> EX ME WB
        IF ----> ID ----> EX ME WB
            IF IDx
        IFx
IF ID EX ME WB
IF ID ----> EX ME WB
IF ----> ID ----> EX ME WB
                                    IF ----> ID EX ME WB
                                    IF IDx
                                    IFx
                                    IF ID ----> ...
    lw $s0, O($s1)
# Cycle:
0
```



Problem 2: The code fragment below (the same as the one above) runs on the illustrated implementation (different than the one above - and better!). Assume the branch is always taken.
(a) Show a pipeline execution diagram covering execution to the beginning of the third iteration of the loop. See below. Note that there is no bypass for the branch condition.
(b) What is the CPI for a large number of iterations?

An iteration is $14-7=7$ cycles, the CPI is $\frac{7}{4}$.
(c) An A points to a wire on the illustration. On the pipeline execution diagram show the value of that wire in every cycle that the corresponding stage holds a "live" instruction.
see diagram. The A points to the integer register number to write. Both the value, and for convenience, the register name are shown. Note that the branch specifies register 0 as a destination, because it does not write any real register.
(d) A B points to a wire on the illustration. On the pipeline execution diagram add a row labeled B, and on it place an X in a cycle if the value on the wire can be changed without changing the way the program executes.

Through $B$ the rs register value from the register file goes to the ALU. It is only used if the instruction uses the rs register value and if that value is not bypassed. A lower-case ex ( x ) is placed in positions where there is no instruction or if the instruction does not use the rs register value (bypassed or note). An upper-case ex ( X ) is placed in positions where the instruction uses a bypassed rs value (the value from the register file is outdated).

```
# Cycle:
LOOP:
    A:
```



```
    lw $s0, O($s1)
    addi $s3, $s0, 4
    bneq $s3, $0 LOOP
    add $s1, $s1, $s2
# Cycle:
    lw $s0, O($s1)
    addi $s3, $s0, 4
    bneq $s3, $0 LOOP
    add $s1, $s1, $s2
        0
        16 19 0
        s0 s3 r0 s1 s0 s3 r0 s1 s0
    IF ID EX ME WB
        IF ID -> EX ME WB
            IF -> ID ----> EX ME WB
                IF ----> ID EX ME WB
0
IF ID EX ME WB
                                IF ID -> EX ME WB
                                IF -> ID ----> EX ME WB
                                IF ----> ID EX ME WB
    lw $s0, 0($s1) IF ID EX ME WB
# Cycle: 0
```



Review Fall 2004 Final Exam Problem 2, which was discussed in class on Monday, 13 March 2006.
Problem 1: Using the solution to Fall 2004 Final Exam problem 2 parts a, b, and d (but not c) as a starting point, make changes to implement a new two-source register MM instruction add.mmr which operates as shown in the example below. Hint: The solution requires a register file modification.

```
add.mmr (r1), (r2), r3 # Mem[r1] = Mem[r2] + r3
```

Solution shown below. The add.mmr instructions use three register source operands and so a third read port must be added to the register file, that is shown below in blue. The "new" register value, $r d v$, is used as the store address for the sum.

The diagram below also shows, in green, the bypass connections used in the solution to Problem 3 (but not the branch condition bypasses used in Problem 4).


Problem 2: Your boss, a stuck-in-the-twentieth-century RISC true believer who only grudgingly agreed to include add.mm, add.mr, add.rm, and add.mmr in MMMIPS, flies into an incoherent rage when you suggest also adding add.mmm to MMMIPS. What pushed your boss over the edge? (That is, why is add.mmm much harder to add to the implementation in the Fall 2004 exam than add.mmr.) Instruction add.mmm operates as shown below:

```
add.mmm (r1), (r2), (r3) # Mem[r1] = Mem[r2] + Mem[r3]
```

Unlike the other memory-memory instructions, add.mmm must read two source operands from memory. To do that without stalling the pipeline would require a second memory port in the MS stage, which is expensive. The alternative is having add.mmm spend two cycles in MS, but that would mean stalling the pipeline which is not something you want to do for reasons other than dependencies.

Problem 3: Write a pair of programs intended to show the benefit of MMMIPS. Both programs should do the same thing, program $A$ should use ordinary MIPS instructions and run on the MIPS pipeline shown below. Program $B$ should use MMMIPS instructions and run on the implementation shown in the exam solution. Reasonable bypass connections may be added, including those needed for branches.
(a) Show the programs.

Two pairs of programs are shown below, each program adds 7 to all the elements of an array. Program $A-1$ has 5 instructions in a loop body and executes at 1.2 CPI; program $B$-1 has 3 instructions and executes at 1.67 CPI . Since the programs are different (albeit accomplishing the same thing) CPI cannot be used to compare them. Instead, performance will be measured in cycles per element. (Think of it as execution time divided by the number of iterations in the array.) Both programs handle one element per iteration, $A$-1 completes an iteration in 6 cycles and $B$-1 completes an iteration in 5 cycles, so that $B-1$ is faster despite having a higher CPI.
(Program $A-1$ stalls in cycle 5 so that the branch can get r2; program $B$-1 stalls in 5 so that add.mm can get r2 and again in cycle 7 so the branch can get r2.)

Programs $A$-2 and $B$-2 perform the same function but operate on two elements per iteration (using a technique called loop unrolling). This eliminates all stalls and so both run at a CPI of 1 , however the margin of CPE of $B$-2 over $A-1$ is now higher (since $B-2$ had more stalls to eliminate).

Both pairs of programs show an advantage for MMMIPS.
(b) Compute the execution time (in cycles) of each program. The comparison should be fair so each program should be producing the same result.

See above.


Programs on next page.

```
## A-1: Regular MIPS. Loop handles one element per iteration.
    #
    # Cycles per instruction: 6 / 5 = 1.2
    # Cycles per element: 6 / 1=6
#
LOOP:
# Cycle
    0
    lw r1,0(r2)
    IF ID EX ME WB
    addi r2, r2, 4
        IF ID EX ME WB
            IF ID EX ME WB
    addi r1, r1, 7
                                IF ID -> EX ME WB
    bneq r2, r9, LOOP
    sw r1, -4(r2)
                                IF -> ID EX ME WB
    # Second iteration below.
    lw r1,0(r2) IF ID EX ME WB
    addi r2, r2, 4
                                    IF ID EX ME WB
# Cycle
            0
## B-1: Memory-memory MIPS (MMMIPS). Loop handles one element per iteration.
    #
    # Assumes bypass from ME to MS to avoid stalls. (Not on exam soln.)
#
    # Cycles per instruction: (8-3)/3 = 5/3 = 1.67
    # Cycles per element: (8-3)/1 = 5/1 = 5
#
LOOP:
# Cycle 0
    add.mm (r2),(r2), 7 IF ID MS EX ME WB
bneq r2, r9, LOOP IF ID MS EX ME WB
addi r2, r2, 4 IF ID MS EX ME WB
# Second iteration below.
    add.mm (r2),(r2), 7 IF ID -> MS EX ME WB
    bneq r2, r9, LOOP IF -> ID -> MS EX ME WB
    addi r2, r2, 4
    IF -> ID MS EX ME WB
# Third iteration below.
    add.mm (r2),(r2), 7 IF ID -> MS EX ME WB
bneq r2, r9, LOOP IF -> ID -> MS EX ME WB
    addi r2, r2, 4
# Cycle 0
```

```
## A-2: Regular MIPS. Loop handles two elements per iteration.
    #
    # Cycles per instruction: 8 / 8 = 1
    # Cycles per element: 8/2 = 4
    #
LOOP: # 8 / 2 = 4
# Cycle 0
    lw r1,0(r2) IF ID EX ME WB
    lw r11,4(r2) IF ID EX ME WB
    addi r2, r2, 8 IF ID EX ME WB
    add r1, r1, 7 IF ID EX ME WB
    add r11, r11, 7 IF ID EX ME WB
    Sw r1, -8(r2) IF ID EX ME WB
    bneq r2, r9, LOOP IF ID EX ME WB
    sw r1, -4(r2) IF ID EX ME WB
    # Second iteration below.
    lw r1,0(r2) IF ID EX ME WB
# Cycle 
## B-2: Memory-memory MIPS (MMMIPS). Loop handles two elements per iteration.
    #
    # Assumes bypass from ME and WB to MS to avoid stalls. (Not on exam soln.)
    #
    # Cycles per instruction: 5/5 = 1
    # Cycles per element: }\quad5/2=2.
    #
LOOP: # CPE: 5/2 = 2.5
# Cycle 0
    add.mm (r2),(r2), 7 IF ID MS EX ME WB
    add.mm (r12),(r12), 7 IF ID MS EX ME WB
    addi r2, r2, 8 IF ID MS EX ME WB
    bneq r12, r9, LOOP IF ID MS EX ME WB
    addi r12, r12, 8
                                IF ID MS EX ME WB
    # Second iteration below.
    add.mm (r2), (r2), 7 IF ID MS EX ME WB
    add.mm (r12), (r12), 7 IF ID MS EX ME WB
    addi r2, r2, 8
    bneq r12, r9, LOOP
    addi r12, r12, 8
    0
```

Problem 4: Show a program that will run slower on the MMMIPS implementation that the ordinary MIPS implementation. That program, of course, should not use MMMIPS instructions. Reasonable bypass connections can be added, including those needed for branches. Hint: Branches are important.

The program is a single loop, the key feature being that the branch depends upon the immediately preceding instruction. It is assumed that the implementations have a bypass from ME to ID. With this the MIPS only needs one stall for the branch to resolve the condition. MMMIPS needs two stalls because its EX stage is one stage more distant from ID.

```
## Program on regular MIPS.
    #
    # Bypass from ME to ID for branches assumed.
#
    # Cycles per instruction: 4 / 3 = 1.333
#
#
LOOP:
# Cycle 0
    xor r1, r1, r2 IF ID EX ME WB
    bneq r1, r3, LOOP IF ID -> EX ME WB
    sll r1, r1, 3 IF -> ID EX ME WB
    # Second iteration.
    xor r1, r1, r2 IF ID EX ME WB
    bneq r1, r3, LOOP IF ID -> EX ME WB
    sll r1, r1, 3 IF -> ID EX ME WB
# Cycle }0
## Program on MMMIPS.
    #
    # Bypass from ME to ID for branches assumed.
#
    # Cycles per instruction: 5 / 3 = 1.667
#
# Cycle 0
    xor r1, r1, r2 IF ID MS EX ME WB
    bneq r1, r3, LOOP IF ID ----> MS EX ME WB
    sll r1, r1, 3 IF ----> ID MS EX ME WB
    # Second iteration.
    xor r1, r1, r2 IF ID MS EX ME WB
    bneq r1, r3, LOOP IF ID ----> MS EX ME WB
    sll r1, r1, 3 IF ----> ID MS EX ME WB
# Cycle 0
```

Problem 1: The code below executes on the illustrated MIPS implementation. The FP pipeline is fully bypassed but the bypass connections are not shown.

(a) Show a pipeline execution diagram. Solution shown below. The stall is for the dependency through register f 2 .
(b) Determine the CPI for a large number of iterations.

Because the second and third iterations start with the pipeline in the same state, the time for the second iteration can be used as a basis for computing CPI. The second iteration starts in cycle 3 (first instruction in IF), the third iteration starts in cycle 9, each iteration is 3 instructions to the CPI is $\frac{9-3}{3}=2$.
(c) Add exactly the bypass connections that are needed.

Solution shown in the diagram above. Added bypass connection shown in blue bold.

See next page for solution to first part.

```
Solution to Problem 1a
LOOP:
# Cycle: 0
    mul.d f2, f2, f4 IF ID M1 M2 M3 M4 M5 M6 WF
    bneq r1,0 LOOP IF ID EX ME WB
    addi r1, r1, -1 IF ID EX ME WB
# Cycle: 0
    mul.d f2, f2, f4 IF ID -------> M1 M2 M3 M4 M5 M6 WF
    bneq r1,0 LOOP IF -------> ID EX ME WB
    addi r1, r1, -1 IF ID EX ME WB
# Cycle: 0
    mul.d f2, f2, f4 IF ...
```

Problem 2: Due to a coffee spill the implementation below has a flaw: The inputs to the M2-stage XW mux have been reversed, the top input should be a 2 but is a 1 , and the lower input should be a 1 but is a 2. There are no other flaws, in particular the control signal for the mux has been designed for a 2 at the upper input and a 1 at the lower input.

You are stranded alone on an island with this flawed implementation and to get off the island you need the result computed by the code below. The code was written for a normal MIPS implementation and will not compute the correct result on the flawed one. Re-write it so that it computes the correct result on the flawed implementation. (The solution must use the FP arithmetic units, do not simply implement IEEE 754 floating point using integer instructions.)


Solution on next page.

The problem with the implementation above is that when an add. d is in WF the mux will send the output of the FP multiply unit, not the FP add unit, to the register file. When a mul. d reaches WF the mux will choose the FP add unit for writeback.

Simply substituting a mul.d for the add. d won't work because the add unit would have gotten the wrong inputs. This happens in the example below where in cycle 8 the output of the adder, not the multiplier, is written back. The input values for the adder are determined by the instruction that was in ID in cycle 3, which is not mul.d.

```
# Cycle 0
mul.d f2, f4, f6 IF ID M1 M2 M3 M4 M5 M6 WF
ID A1 A2 A3 A4
```

To get the correct input to the add functional unit a second mul. d needs to be added. Suppose our goal is to execute add. d $\mathrm{f} 2, \mathrm{f} 4, \mathrm{f} 6$. Then start with a mul.d with the desired destination register but using dummy source registers. (See the code below.) Follow that with a nop and then another mul. d having a dummy destination but the desired source registers; when this second multiply reaches M1 its source operands will go to both the multiply unit (M1) and the add unit (A1). The first multiply will write the output of that add unit to the register file and so it will be as though add. $\mathrm{d} \mathrm{f} 2, \mathrm{f} 4, \mathrm{f} 6$ were executed.

```
# Code below effectively executes add.d f2, f4, f6
# Cycle 0
mul.d f2, f30, f30 IF ID M1 M2 M3 M4 M5 M6 WF # Dummy sources.
nop
mul.d f30, f4, f6 IF ID M1 M2 M3 M4 M5 M6 WF # Dummy destination
```

To get the original code below running on the faulty computer replace the add. d with a pair of multiplies. The only additional complication is that a source and destination register match, and that would create a confounding dependence stall if the exact technique above were used. Instead, the loop is unrolled so that one iteration of the re-written loop does the work of two iterations in the original loop. The first half-iteration writes $f 12$ instead of $f 2$, the second half-iteration reads $f 12$ instead of $f 2$.

The timing for the first iteration is shown. In the second iteration the first multiply should stall. The code is written assuming an even number of iterations in the original loop.

```
# Original code to be executed on the faulty computer.
LOOP:
    add.d f2, f2, f4
    bneq r1,0 LOOP
    addi r1, r1, -1
# Modified code that produces the same result as the code above.
    # Dummy registers: F26, F28, F30
    # Cycle 
LOOP:
    mul.d f12, f2, F30 IF ID M1 M2 M3 M4 M5 M6 WF # Dummy sources, but f2 needed for stall
    nop IF ID EX ME WB
    mul.d F26, f2, f4 IF ID M1 M2 M3 M4 M5 M6 WF # Dummy dest.
mul.d f2, f12, F30 IF ID ------> M1 M2 M3 M4 M5 M6 WF # Dummy source.
nop
    mul.d F28, f12, f4 IF ID M1 M2 M3 M4 M5 M6 WF # Dummy dest.
    beq r1, O LOOP IF ID EX ME WB
    addi r1, r2, -2 IF ID EX ME WB
    # Cycle 
```

Note: For some sample problems with predictors see the final exam solutions.
Problem 1: The routine samples in the code below is called many times. Consider the execution of the code on three systems, each system using one of the branch predictors below.

All predictors use a $2^{14}$-entry branch history table (BHT). (The global predictor does not need its BHT for predicting branch direction.) The three predictors are:

- System B: bimodal
- System G: global, history length 10. (Accuracy can be approximated.)
- System L: local, history length 10.

```
void samples(int& x, int& y, char **string_array )
{
    // Loop 5-xor
    for( int i = 0; i < 5; i++ )
        x = x ^ i;
    // Loop 5-len
    for( int i = 0; i < 5; i++ )
        if( strlen( string_array[i] ) < 20 )
            return; // Never executes. <- Important.
    // Loop 100-xor
    for( int i = 0; i < 100; i++ )
        y = y ^ i;
}
```

(a) Determine the amount of memory (in bits) needed to implement each predictor.

Bimodal: The bimodal predictor just uses the BHT, each entry is two bits. Bimodal predictor size $2^{14} \times 2$ bits.
Global: The global predictor uses only a pattern nistory table, the number of entries is $2^{h}$, where $h$ is the history length, 10 in this case; each table entry is 2 bits. Global predictor size $2^{10} \times 2$ bits.

Local: The local predictor uses both a BHT and PHT. The BHT stores the local history, which is the outcome of the last 10 branches, so each entry is 10 bits. The PHT is indexed using this 10 -bit history so there are $2^{10}$ entries, each entry of this table is 2 bits. Local predictor size $2^{14} \times 10+2^{10} \times 2$ bits.
(b) For each loop in samples determine the accuracy of the loop branch (the one that tests the value of i) after warmup on each system. The accuracy for the global predictor can be approximated, the others must be determined exactly.

Bimodal: Once warmed up the bimodal predictor will only mispredict the last loop iteration. Loops 5 -xor and 5 -len are 5 iterations and 100 -xor is 100 iterations. Bimodal accuracy on 5 -xor and 5 -len is $\frac{4}{5}=0.8$.
Bimodal accuracy on 100 -xor is $\frac{99}{100}=0.99$.
A solution for 5 -xor is worked out in detail below. Accuracy is computed over a repeating region, in this case outcome \# 5 to 10. At these outcomes the loop has just started and in both cases the counter value is 2.

```
Loop 5-xor - Bimodal
    Outcome # 0
    Branch Outcomes T T T T N T T T T N T...: N, Not taken; T, Taken
2-bit Counter 0
Prediction N N T T T T T T T T T T : T if counter > 1
Mispredicted X X X X : X means misprediction
```

Accuracy: 4 correct predictions / 5 predictions

Local: The local predictor can predict short loops perfectly so long as the number of iterations is not larger than the history length. That is the case for 5 -xor and 5 -len. Local predictor accuracy on 5 -xor and 5 -len is 1.0 .
For loop 100-xor the local predictor will mispredict the last iteration. Local predictor accuracy on 100-xor is 0.99 .
Global: The global predictor can also predict short loops as long as the outcome of the first loop branch is present when predicting the last iteration. For loop 5-xor there is no other branch in the loop and so the global history is long enough to distinguish the last iteration from others. Global predictor accuracy on 5-xor is 1.0 .

Loop 5 -len includes a call to the strlen routine, and that most certainly has branches. There is also the branch testing the string length. In the diagram below the 5-len branch outcomes, the if statement branch outcomes, and the strlen branch outcomes are shown. The global history is a concatenation of these outcomes. When trying to predict outcome 1 the global history will be TTTTTTTTNN (from the loop in the strlen routine), that's the same global history when trying to predict outcomes 2,3, and 4 , so there is no way to distinguish outcome 4 from outcomes 1,2 , and 3 . Since most of those are taken the PHT entry will be 2 or 3 and Taken will be predicted every time, the same prediction a bimodal would make. Global predictor accuracy on 5 -len is 0.8 .

(c) Why would solving the problem above be impossible, or at least tedious, if the BHT size were $\approx 2^{3}$ entries?

If the BHT were that small there would be a reasonable chance that more than one of the branches above would occupy the same BHT entry, possibly reducing accuracy. Since the code is given in high-level form there is no way to tell what the branch addresses were. Unless you compiled the code and examined the branch addresses. That would be tedious.

Problem 2: The code more, below, runs on four systems. All predictors use a $2^{14}$-entry branch history table (BHT). (The global and gshare predictors do not need their BHT for predicting branch direction.) The predictors are:

- System B: bimodal
- System G: global, history length 10. (Accuracy can be approximated.)
- System X: gshare, history length 10. (Accuracy can be approximated.)
- System L: local, history length 10.
(a) In the code below estimate the prediction accuracy of the following predictors on Branch B and Branch C (there is no Branch A) after warmup, assuming that more is called many times.

Bimodal: Each branch's outcome is always the same (they are highly biased) and the BHT is large enough so that it's unlikely that any branch shares B or C 's entry. $\quad$ Bimodal predictor accuracy on B and C is 1.0.

Local: The local history for B after warmup will always be "NNNNNNNNNN" and the corresponding PHT entry will have sunk down to zero (if not already reduced to zero by other biased not taken branches); C's local history after warmup will always be "TTTTTTTTTT" and the corresponding PHT entry will be 3 . Local predictor accuracy on B and C is 1.0 .

Global: The global history used when predicting B consists of the outcomes from the 100 -iteration loop above B , that history will be "TTTTTTTTTN". The global history used when predicting C consists of the outcomes from the 100 -iteration loop above C; it's a different but similar loop and so the outcomes will be the same "TTTTTTTTTN". Since both B and C are predicted using the same global history they will share a PHT entry, branch B will decrement the entry and branch C will increment it. Assume that no other branch uses that PHT entry. If the entry starts out at 0 or 1 branch B will be predicted with $100 \%$ accuracy and C will be predicted with $0 \%$ accuracy. If the entry starts out at 3 branch B will be predicted at $0 \%$ accuracy and C will be predicted at $100 \%$ accuracy. If the entry starts out at 2 both B and C will be predicted at $0 \%$ accuracy. Global predictor accuracy on B and $\mathrm{C}: 1$ and 0 or 1 and 0 or 0 and 0 .
gshare: In a global predictor the PHT is indexed using the global history register. A gshare predictor is identical to a global predictor except that the PHT is indexed using the bitwise exclusive or of the global history and the branch address. As a result the PHT entries used for predicting branches B and C will be different. (The global histories are still the same but of course the addresses of B and C are different.) Assuming no other branches share these PHT entries, the prediction accuracy will be $100 \%$. gshare accuracy on B and C: 1.0.
(b) One of the predictors should have a low prediction accuracy. Why? Avoid a sterile description of the hardware, instead discuss the concept the predictor is based on and why that's not working here.

The global predictor has the low prediction accuracy. See the solution to the previous part for the global and gshare predictors.

```
void more(int& x, int& y, int a, int& b, int& c)
{
    for( int i=0; i<100; i++ ) x = x ^ i;
    if( a < 10 ) b++; // Branch B, never taken.
    for( int i=0; i<100; i++ ) y = y ~ i;
    if( a >= 10 ) c++; // Branch C, always taken.
}
```


## 61 Fall 2005 Solutions

Problem 1: Suppose the base and result (peak) SPEC CINT2000 benchmark scores were identical on company $X$ 's new processor. Make up an advertising slogan based on the fact that they were identical. A catchy tune is optional.

Performance you don't have to work for!
Meaning you don't need to spend hours trying out different compiler optimization options to get the advertised performance.

Grading Notes: Many answered that "extreme optimizations were not necessary," which is not correct (or at least misleading). As used in class "extreme" referred to the effort by the programmer to get the compiler to produce the fastest code. Many programmers will not make an extreme effort at optimization and so the performance they see might be along the lines of the base number, which is fine on systems in which it's the same as peak. In a system with identical base and peak numbers the compiler may well be doing optimizations that could be described as extreme, however since the compiler can do them when given only basic optimization flags their benefts are be seen in the base numbers.

Problem 2: According to the CPU performance equation increasing the clock frequency ( $\phi$ ) by a factor of $x$ without changing instruction count (IC) or cycles per instruction start (CPI) will reduce execution time by a factor of $x$. Find two SPEC CINT2000 disclosures (benchmark results) that provide good evidence for this.
(a) Give the CPU, clock frequency, and the base and result CINT2000 scores.

To show the effect of clock frequency pick two systems which are close to identical in every way except clock frequency. For example, the 2.16 GHz and 1.87 GHz Fujitsu SPARC64 V chips:
http://www.spec.org/osg/cpu2000/results/res2005q2/cpu2000-20050419-04024.html http://www.spec.org/osg/cpu2000/results/res2005q1/cpu2000-20050208-03825.html

The 1.87 system scores 1594 peak and 1456 base, the 1.87 GHz system scores 1341 peak and 1254 base.
Other than clock frequency the major differences between the systems are in the L 2 cache size ( 3 MiB V .4 MiB ) and the number of CPUs the system can handle ( 2 V . 16, though the systems tested each had one).
(b) Explain why for these disclosures $\phi$ is different (obvious) but IC and CPI are probably the same (requires some thinking). It may not be possible to determine this for certain and it may not be possible to find a pair for which they are exactly the same, it's sufficient to find a pair in which they are arguably close.

The clock frequency is listed in the disclosure and they are different. The IC is probably the same because the same compiler was used, Fujitsu Parallelnavi 2.3 with Sun Studio 9 . The CPI is probably the same because the compiled code is the same (for reasons just given) and because the code is running on chips of the same microarchitecture (that's assumed because of the same processor name).
(c) Based on the assumption of IC and CPI equality, show how closely the CPU performance equation predicts the performance of one of the systems. Suggest reasons for any difference.

If clock frequency were the only differing factor then based on the 1341 peak performance of the 1.87 GHz chip one would predict a peak performance of $\frac{2.16 \mathrm{GHz}}{1.87 \mathrm{GHz}} 1341=1549$ on the 2.16 GHz chip. The performance is actually higher, a possible reason might be the larger L2 cache.

Problem 3: In section 1.2 of the SPEC CPU 2000 run and reporting rules, http://www.spec.org/cpu2000/docs/runrules.html, there is a bullet item that states, "The vendor encourages the implementation for general use." Explain what that means and why it is there. Why would it be bad if the "implementation" were not "for general use."

In the disclosure "implementation" refers to a tool or library used to build (compile, etc) the benchmarks. (It might also refer to the hardware but most of that section talks about compilers and related tools.) The statement says that SPEC expects that anything used to build the benchmark should be a product of the company (or offered by others) and
that the company should make a serious attempt to sell it. It would be bad if the "implementation" were not for general use because that might mean it was too unreliable for customer use and so the benchmark scores achieved using it are higher than a typical user, who would avoid unreliable products, could expect to achieve.

Another danger is that a compiler could be too benchmark specific. In an extreme case an assembly language programmer could hand-code parts of the benchmarks and the compiler would insert that code wherever it recognized the benchmark. That would only work on those specific programs (even minor modifications to the benchmark would render such an optimization useless) and would not work on other code.

Homework 4 solution

Problem 1: The code below executes on the implementation illustrated.
(a) Draw a pipeline execution diagram up until the first fetch of the third iteration.
(b) What is the CPI for a large number of iterations?


```
# Solution:
# Cycle 0
addi r3, $0, 123 IF ID EX ME WB
LOOP:
lw r1, O(r2) IF ID EX ME WB
bne r1, r3, LOOP IF ID ----> EX ME WB
lw r2, 4(r1) IF ----> ID EX ME WB
# Cycle 0
LOOP:
```

lw r1, 0(r2) IF ID -> EX ME WB

```
lw r1, 0(r2) IF ID -> EX ME WB
bne r1, r3, LOOP IF -> ID ----> EX ME WB
bne r1, r3, LOOP IF -> ID ----> EX ME WB
lw r2, 4(r1)
lw r2, 4(r1)
# Cycle
# Cycle
lw r1, O(r2)
```

```
lw r1, O(r2)
```

```

In the first iteration lw r1, \(0(r 2)\) executes without a stall but in the second iteration it stalls in ID and so the first iteration cannot be used to compute CPI. The second and third iterations start with the processor in the same state (the branch in EX and the second load in ID, see cycles 6 and 12), and so the second iteration can be used to compute the CPI. The CPI is \(\frac{12-6}{3}=\frac{6}{3}=2\).

Problem 2: Is there any way to add bypass paths to the implementation above so that the code executes with fewer stalls:
(a) Suggest bypass paths that might have critical path impact but which probably won't halve the clock frequency.

To avoid the branch stall bypass the value from the output of the MEM-stage memory port to the comparator in ID, shown in blue below. This will probably impact critical path because the memory port is probably using the whole cycle. It won't halve the clock frequency because the comparison can be done quickly (certainly less than a cycle because it ordinarily waits for the register file).
(b) Explain why it is impossible to remove all stalls by adding bypass paths.

A bypass path for the \(\mathrm{lw} \mathrm{r} 2,4(\mathrm{r} 1)\) to \(\mathrm{lw} \mathrm{r} 1, \mathrm{O}(\mathrm{r} 2)\) dependence would go from the output of the memory port to the ALU input (shown in red below), each of those devices uses most of its clock cycle and so clock frequency would be halved. That's really bad and one should never do it, but it's not impossible.

A bypass to handle the \(\mathrm{lw} \mathrm{r} 1, \mathrm{O}(\mathrm{r} 2)\) to bne r1, r3, LOOP dependence \(i s\) impossible because the branch needs the loaded value one cycle before its available. For example, in the solution the branch needs the loaded value in cycle 3, but the load instruction has not yet reached MEM.


Problem 3: The beqir instruction from the midterm exam solution compares the contents of the rs register to the immediate, if the two are equal the branch is taken, the address of the branch target is in the rt register. In the code example below beqir compares the contents of r3 to the constant 123 , if they are equal the branch is taken with register \(r 1\) holding the target address, in this case to TARG. The delay slot, nop, is also executed.
(a) Show the changes needed to implement this instruction on the implementation above.

Changes shown below in blue. Two changes were made. First, a multiplexer is put before comparison unit in ID to select either the rt register value (regular branches) and the immediate (beqir). Second, the rt register value is sent to the PC mux in IF.
(b) Include bypass paths so that the code below executes as fast as possible:
```

lui r1, hi(TARG)
ori r1, r1, lo(TARG)
beqir r3, 123, r1
nop

# Lots more code.

TARG:
xor r9, r10, r11

```

The code above has a dependence from ori to beqir. To bypass the value a bypass path was added from EX to ID, shown in green. This bypass path may stretch the critical path because two multiplexers have been added to the path at the output of the ALU.


Problem 1: The execution of a new MIPS instruction blcz TARG, branch unless loop count register is zero, will result in a delayed control transfer to TARG unless the contents of a new register, lc, is zero; the target is computed in the same way as ordinary branch instructions. Execution of blcz will also decrement lc unless it is already zero. The lc register is loaded by two new instructions mtlc and mtlci. The code below uses some of the new instructions and the diagram shows a possible implementation.
```

mtlc 100 \# Load lc register for a 101-iteration loop
LOOP:
sw r0, O(r1)
blcz LOOP \# If lc is not zero branch to LOOP, lc = lc - 1.
addiu r1, r1, 4

```

(a) Re-write the code above using ordinary MIPS instructions and write it so that the loop uses as few instructions as possible. Hint: A three-instruction loop body is possible.

The solution is on the next page.
```


# Solution

# 

# Re-written code and pipeline execution diagram.

# Cycle

addiu r2,r1,400 IF ID EX ME WB
LOOP:
sw r0, O(r1) IF ID EX ME WB
bne r1, r2, LOOP IF ID -> EX ME WB
addiu r1, r1, 4 IF -> ID EX ME WB

# Code below is a repeat of the code above.

    sw r0, O(r1) IF ID EX ME WB
    bne r1, r2, LOOP IF ID -> EX ME WB
    addiu r1, r1, 4 IF -> ID EX ME WB
sw r0, O(r1) IF ID EX ME WB
bne r1, r2, LOOP IF ID -> EX ME WB
addiu r1, r1, 4 IF -> ID EX ME WB

# Cycle

# Original code and pipeline execution diagram.

# Cycle

mtlc 100 IF ID EX ME WB
LOOP:
sw r0, O(r1) IF ID EX ME WB
blcz LOOP IF ID EX ME WB
addiu r1, r1, 4 IF ID EX ME WB

# Code below is a repeat of the code above.

sw r0, O(r1) IF ID EX ME WB
blcz LOOP IF ID EX ME WB
addiu r1, r1, 4 IF ID EX ME WB

# Cycle

```
(b) Using pipeline execution diagrams determine the speed of the sample program and your program from the previous part. Only use bypass paths that have been provided.

The original code, which uses blcz, executes without a stall and so it executes at a rate of 0.333 stores per cycle ( 1 CPI ). The re-written code suffers stalls and so only executes at a rate of 0.24 stores per cycle ( 1.333 CPI ).

Note that the question asked for the speed of the programs, not the CPI. Since the two programs do the same thing what's important is which one is faster (lower execution time). Since they both do the same number of stores the speed could be measured by stores per cycle. CPI is not a good measure because it only indicates how efficiently the code is running. If two identical programs are running on different processors the lower CPI (higher efficiency) would be faster. But since the programs are different CPI is not useful in predicting speed.
(c) Unless the control logic is appropriately modified the implementation above may not realize precise exceptions for all integer instructions. In fact, the problem could occur in the example program. Explain what the problem is and show a pipeline execution diagram in which the control logic insures that execution proceeds so that exceptions will be precise. Hint 1: The exception does not occur in any of the new instructions. Hint 2: One of the two remaining instructions in the example can not raise an exception so it must be the other one.
```


# Part of Solution

# Cycle 0

mtlc 100 IF ID EX ME WB
LOOP:
sw r0, O(r1) IF ID EX M*x
blcz LOOP IF ID --> EX ME WB
addiu r1, r1, 4
IF --> ID EX ME WB

```

The integer instructions cannot raise precise exceptions with the blcz changes because the le register is modified in ID, when preceding instructions can still raise exceptions. If they do the handler will see the wrong value in Ic.

In the example above sw raises an exception in cycle 4 . For the exception to be precise the handler must see the execution of only instructions up to sw, which here is only mtle, and so the handler should see an le value of 100 . If blez did not stall in cycle 3 Ic would have been decremented and so the handler would have seen an Ic value of 99, meaning the exception would not have been precise.
(d) Modify the implementation so that precise exceptions are again possible for all integer instructions (while retaining the loop count instructions) without sacrificing performance.

The modifications are shown below. Before the modifications ic would be both read and written in ID. To allow for stall-free precise exceptions modify the implementation so that Ic is written in WB, as are the other registers. Those changes are shown in blue. When an instruction that modifies \(1 c\), blcz, mtle, and mtlci, is in ID the new value of \(I c\), rather than writing \(I C\), is put in the ID/EX.rtv pipeline latch. When such an instruction is in EX the ALU is set to pass the lower input through unchanged so that when the instruction reaches WB the new Ic value will be in the MEM/WB.ALU latch, that latch is connected to le's data input. A write enable (we) input is also shown, it is based on the output of Writes IC, which is 1 for instructions that modify Ic.

Bypass paths for the le register are shown in purple.


\section*{Grading Notes:}

Some solutions tried to repair Ic by decrementing its value when an exception is detected. This would fix the problem with sw but won't handle cases in which mtlc or mtlci write lc. Also, control logic would have to check if a blcz is in the EX and possibly ID stages (depending on timing); many solutions did not point this out.

Another common incorrect solution was keeping the lc logic in ID but not modifying lc if a preceding instruction raises an exception. That won't work because when an instruction is in ID a doomed preceding instructions might not yet have raised an exception, sw is an example. One solution was specific in specifying EX-stage hardware to see if the instruction their would raise a memory-related exception when it reached MEM. That's not possible without doing major damage to the critical path.

\section*{62 Spring 2005 Solutions}

Problem 1: POWER is an IBM ISA developed for engineering workstations, PowerPC is an ISA developed by IBM, Apple, and Motorola for personal computers and is based on POWER. POWER and PowerPC have instructions in common but each has instructions the other lacks (and some of the common instructions behave differently). Therefore a POWER implementation could not run every PowerPC program and vice versa.
(a) Show the gcc 3.4.3 compiler switches used to compile code for a POWER implementation. Hint: Google is your friend, look for gcc documentation.

Either of the following switches compiles for a generic POWER implementation -mpower, -mcpu=power. The compiler can also be told to target a particular implementation, for example, -mepu=rios2.
(b) Show the gcc 3.4.3 compiler switches used to compile code for a PowerPC implementation.

Either of the following switches compiles for a generic PowerPC implementation-mpowerpe, -mepu=powerpe. The compiler can also be told to target a particular implementation, for example, -mepu \(=620\)
(c) Is it possible to use gcc 3.4.3 to compile a program that will run on both? If yes, show the switches.

Yes, one way is to specify two switches: -mno-power -mnopowerpe, the other uses the single switch -mepu=common.

Problem 2: From the SPEC Web site, http://www.spec.org, find the fastest result on the SPECFP2000 (that's FP, not INT) benchmark for each of the following implementations: IBM POWER5, Intel Itanium2, Intel Pentium 4, Fujitsu SPARC64 v, and AMD FX-55. (Use the configurable search form and have it display the processor name.)
(a) The non-IA-32 implementations (POWER5, Itanium2, and SPARC64 V) blow away the IA-32 implementations on one benchmark. Which one? Which company (of those listed above) would want that benchmark removed?

The Art benchmark runs much faster on non-IA 32 systems: SPARC64 12.3. POWER5, 23.1; Itanium 2, 21.0 Pentium 4, 52.6, Opteron, 78. AMD would most want it removed.
(b) The POWER5 can decode five instructions per clock, the Itanium 2 can decode six instructions per clock, the Pentium 4 and FX-55 each can decode three (what are essentially) instructions per clock, and the SPARC64 V can decode four per clock. Based on the SPECFP2000 results used in the first part, which processor is making best use of these decode opportunities? In other words, if one processor could decode \(10^{12}\) instructions during execution of the suite and another could decode \(5 \times 10^{12}\) instructions during execution of the suite, the first would be more efficient since it ran the suite using fewer instructions. (See last semester's Homework 1 for a similar problem.)

To solve this one needs to multiply the instructions-per-second potential of the processor by the run time for the suite. The instructions-per-second potential is the product of the decode rate given above (say, 5 per second for POWER5) and the clock frequency. The run times are given in the disclosure and can be added, but a less time consuming method would be to use the reciprocal of the score. So for the POWER5 the result would be \(\frac{5 \times 1900 \mathrm{MHz}}{2796}=3.40\), where 2796 is the SPECfp2000 result. For the Itanium 2, \(\frac{6 \times 1600}{2712}=3.54\); Athlon, \(\frac{3 \times 2600}{2012}=3.88 ;\) Pentium \(4, \frac{3 \times 3733}{2016}=5.56\); and SPARC64 \(\mathrm{V}, \frac{4 \times 1870}{1973}=3.79\). The POWER5 is the "winner" here because it uses the fewest decode slots to execute the SPECfp2000 benchmarks. This can be beeause POWER5 programs have fewer instructions or because the POWER5 implementation wastes the fewest decode slots (or a combination of the two). In this case the POWER5 is both the most frugal and the fastest. Note that the Athlon and Pentium are almost tied in performance but that the Pentium uses alot more decode slots to attain that performance.

Problem 3: As pointed out in class a processor's CPI varies depending on the program being executed. For the questions below write a program in MIPS assembler (see
http://www.ece.lsu.edu/ee4720/mips32v2.pdf for a list of instructions), some other assembly language, or assembly pseudocode, as requested below.
(a) Write a program that might be used to determine the minimum possible CPI. Suppose you actually used the program to determine the minimum CPI on processor \(X\). How would the CPI be computed? Show an example using made up numbers based on your program an hypothetical processor \(X\). Explain why the result would be the minimum CPI (or close to it).

Many processors do not attain their peak CPI because of program characteristics. Two causes are using instructions that take a long time and having nearby instructions depend on each other, forcing the processor to delay the start of the later instructions. When writing a program to determine peak cpi avoid long-executing instructions and close dependencies. The program below uses integer add instructions, which are fast and which lacks dependencies.
```

LOOP:
add r1, r2, r3
add r4, r5, r6
add r7, r8, r9
\# ...
j LOOP
add r28, r29, r30

```
(b) Write a program that might be used to determine the maximum possible CPIand as with the previous part, show how CPI is computed. Your answer should include information about instructions in processor \(X\) used in your program. Explain why the result would be the maximum CPI (or close to it).

Include long-executing instructions that depend on each other.
LOOP:
div.d f0, f2, f4
div.d f0, f0, f4
div.d f0, f0, f4
div.d f0, f0, f4
\# ...
j loop
div.d f0, f0, f4

For answers to the questions below refer to the PowerPC description Book I which can be found on the class references page, http://www.ece.lsu.edu/ee4720/reference.html.

Problem 1: One instruction that MIPS lacks but many RISC ISAs have is an indexed load. Find the closest equivalent PowerPC instruction to SPARC's lw [\%r2+\%r3], \(\% r 1\).
(a) Show the instruction in PowerPC assembly language.
```


# Solution:

lwzx r1, r2, r3

```

Note: Instruction ldx, which loads 64 bits) would also be graded correct, but since SPARC's 1 w is a 32 -bit unsigned load (in SPARC V9), PPC's 32-bit unsigned indexed load, 1 wzx , is more correct.
(b) Show how the instruction is coded, include the register numbers.


Problem 2: One instruction that MIPS lacks but that a few other RISC ISAs have is autoincrement addressing. PowerPC has an instruction that can be used for autoincrement addressing but is more powerful than the autoincrement addressing described in class. Find the PowerPC instruction.
(a) Show the assembly language for the PowerPC instruction doing the same thing as the following autoincrement instruction: lw r1, (r2)+.
```


# Solution

lwzu r1,4(r2)

```

The PowerPC instruction above is not \(100 \%\) equivalent because it uses \(r 2+4\) as the effective address whereas a typical autoincement would use \(r 2\) as the effective address. This is not a practical problem because \(r 2\) could be initialized to four less that the first address to be loaded.
(b) Show the coding for the instruction above.

(c) The PowerPC instruction is more powerful than an ordinary autoincrement instruction. Show a code sample using the PowerPC instruction for which an ordinary autoincrement would not be suitable. Briefly explain why an ordinary autoincrement would not do.

LOOP:
lwzu r1, 16(r2)
cmpdi r1,0
bne LOOP
The loop above loads words separated by 16 bytes. An ordinary autoincrement would require an extra add instruction (otherwise it would load words separated by 4 bytes (that is, consecutive words).

Problem 3: PowerPC has a wide variety of load and store instructions. Find the load instruction that is least suitable for a RISC ISA based upon the criteria discussed in class. Explain why it's least suitable.

Instruction 1swx is un RISC like because it can load several registers and so an implementation would have to send the instruction through part of the pipeline several times, unlike conventional RISC instructions. This would make the pipeline control much more complicated, but not too complicated since PPC is a real ISA with fast implementations.

Problem 4: Some instructions are more difficult to implement than others, one reason is that the difficult instruction does something very different from normal instructions, requiring at least a moderate amount of additional hardware. Some difficult-to-implement instructions are listed below. Explain what the difficulty is (what extra hardware or control complications would be needed).
(a) An indexed store instruction. (An indexed load instruction would not be considered difficult.)

An indexed store would have three source operands, two for the effective address and one for the store value. That would require three read ports from the register file, it no other instruction has three source register operands then implementing the indexed store would increase the cost by a significant amount.
(b) Autoincrement (or PowerPC's version) load instructions. (The autoincrement or PowerPC version of the store instructions are not difficult.)

An autoincrement load writes two register values, the loaded value and the incremented address. That would require two register write ports, if no other instruction wrote two or more operands than that would increase the cost by a significant amount.

Homework 3 solution

Problem 1: Do Problem 1 in the Spring 2004 EE 4720 final exam. Grade yourself using the solution, the grade should be out of five points. When grading yourself please explain what the mistakes were and what the correct answer should be, as a helpful grader would. Be polite in your explanations unless there was no serious attempt to solve the problem. In that case point out how final exam study time is being undermined by the need to catch up.

Problem 2: In Method 3 the commit register map is used to recover the state the ID register map was in just after the most recently committed instruction was decoded. In a system in which the ID register map is checkpointed for predicted CTIs, the commit register map won't be used very often.
(a) Describe how a system using Method 3 but without the commit register map could recover the ID map state before a faulting instruction. The ID map would be recovered using information in the ROB at and after the faulting instruction.
- Explain, with an example, what steps the processor takes to recover the information.

A ROB entry for an instruction includes, among other things, the architected destination register (call it rd), the new physical register that rd maps to (and the one the instruction will write), and the incumbent, that is, the physical register that rd mapped to before the instruction was decoded. When (and if) the instruction commits it is the incumbent that will be put back on the free list.

When recovery is necessary the incumbents can be used to return the ID map to the state it was before any in-fight instruction was decoded by writing the incumbents back into the register file starting from the tail of the ROB up to the faulting or mispredicted instruction.

The example below shows the state of a system with five instructions in flight. The numbers on the left show the architected destination, physical destination, and incumbent physical destination registers. Suppose the sw raises an exception. The ID map can be recovered by removing the tail entry from the ROB, and writing the incumbent back into the register map (using the architected destination as a key). This process is repeated until the faulting instruction is removed. In the example below ID shows normal activity. Squash recovery starts in cycle 10 and continues in 11 and 12; SQ shows a tail instruction being removed and the incumbent being written back, this happens for each of the instructions to be squashed.

If it is only possible to remove instructions from the head then recovery is a bit trickier since only the first incumbent (for each architected register) should be written to the ID register map. This can be done by keeping track of which architected registers have been written (using a bit vector) and not writing one twice. Apart from needing a bit vector, a disadvantage recovering from the head is that recovery can't start until after the faulting or mispredicted instruction reaches the head.
```

dst PR inc Normal Decode Recovery----
r1 11 10 add r1, r2, r3 ID HEAD END
sw O(r0), r0
r4 16 15 sub r4, r2, r3 ID
r1 4 11 xor r1, r2, r
r1 7 4 or r1, r2, r3
Cycle
ID Register Map

| r1 | 10 | 11 |  | 4 | 7 | 4 | 11 |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| r4 | 15 |  | 16 |  |  |  | 15 |  |

```
(b) Show new connections to the ID register map to implement this. Try to do it without adding new read and write ports (that is, use existing ports).

Add a mux to the Addr and \(D\) in ports currently used to write the new physical register. On input to each mux is the existing connection to the port, the other is taken from either the head output (this part) or tail output (next problem).
(c) Describe the impact on performance when the technique is used for exceptions.

Since instructions must be squashed one at a time rather than all at once recovery will take longer. However, since exceptions are rare that won't have a big impact on performance.
(d) Describe the impact on performance when the technique is used for mispredicted branches. If there are alot of instructions to squash then this will slow down the recovery process.

Problem 3: Consider the commit mapless system from the previous problem. Suppose it were possible to sequentially read the ROB from two locations, the head (as is currently done for committing instructions) and some other place, say at a mispredicted branch.
(a) How might this be used to recover the ID map faster than was done in the previous problem.

As explained in the previous problem, immediately start recovery by removing instructions from the tail (that's the other place) rather than waiting for the branch to reach the head.
(b) If this can be made to work for branches then there would be no need for checkpointing the register map. The impact on performance when using the mechanism for mispredicted branches depends on the following factors: how fast instructions are fetched, how many cycles it takes to resolve a branch (determine if the prediction was correct), how long it takes the fetch mechanism to bring correct-path instruction to ID, and how fast recovery can be done. Show a formula that will give the number of extra cycles needed to recover from a branch misprediction using this scheme (compared to checkpointing). For the formula, use the factors listed above and any other that is relevant.

Note that the amount of time to fetch the first correct-path instruction is a variable, it can be more than the one cycle shown in most other problems.

When a misprediction occurs two things have to happen: correct path instructions need to be fetched and the ID register map needs to be recovered. (Other things need to be done, but not for this problem.) The formula should indicate how much longer ID map recovery takes than getting the correct path instructions.

Let \(i_{\mathrm{f}}\) denote the number of instructions that can be fetched per cycle and let \(i_{\mathrm{r}}\) denote the number of instructions that can be squashed per cycle (as described in the previous problem). Let \(t_{\mathrm{BR}-\mathrm{ID}-\mathrm{wB}}\) denote the average number of cycles a branch takes to resolve (move from ID to WB ) and let \(t_{\mathrm{BR} \text {-wB-ID }}\) denote the average number of cycles from when a mispredicted branch reaches writeback until the first correct path instruction is fetched and reaches ID.

When a misprediction occurs the number of instructions that need to be squashed is \(i_{\mathrm{f}} \times t_{\mathrm{BR}-\mathrm{ID}-\mathrm{wB}}\); the ID map recovery will take \(i_{\mathrm{f}} \times t_{\mathrm{BR}-\mathrm{ID}-\mathrm{wB}} / i_{\mathrm{r}}\) cycles. The amount of extra waiting time is then
\[
\max \left\{0, \frac{i_{\mathrm{f}} \times t_{\mathrm{BR}-\mathrm{ID}-\mathrm{WB}}}{i_{\mathrm{r}}}-t_{\mathrm{BR}-\mathrm{WB}-\mathrm{ID}}\right\} \quad \text { cycles }
\]

In an \(n\)-way superscalar processor \(i_{\mathrm{f}}=n\) and it would be reasonable to expect \(i_{\mathrm{r}}\) also to be \(n\). Therefore there will be a performance penalty if average resolution time is Ionger than fetch time, which is only sometimes true. If it is not be too expensive to provide extra ports so that \(i_{\mathrm{r}}>n\) then the penalty can be reduced further or eliminated.

\section*{63 Fall 2004 Solutions}

\section*{LSU EE 4720}

Homework 1 solution
Due: 15 September 2004

Problem 1: Select two pairs of disclosures (that's four total) from the CPU2000 benchmark results posted at www.spec.org. A pair should be for machines using the same ISA but having different implementations. Make the implementations as different as possible. Explain why you think the implementations are very different.

Some ISAs and implementations are listed in lecture set 1, but the solution is not restricted to those. Feel free to ask if you're not sure what ISA a processor implements or whether two ISAs are considered the same or different.

For each disclosure list: the ISA, the implementation, the peak (result) performance, and file name of the HTML-formatted disclosure.

First pair: ISA IA-32, implementations:
Xeon, Peak 1402, file res2004q3/cpu2000-20040727-03291.pdf and Athlon, Peak 1395, results/res2003q3/cpu2000-2003090802502.pdf.

The two implementations are very different because they come from two different companies (and one was not simply licensing the processor design from another). One obvious evidence of difference is that the two achieve similar performance though having vastly different clock frequencies.

Second pair: ISA SPARC V9, implementations:
SPARC64 V, Fujitsu, Peak 1345, 1.89 GHz, res2004q2/cpu2000-20040518-03044.pdf and UItraSPARC III Cu, Sun Microsystems, Peek 722, 1.2 GHz, http://www.spec.org/cpu2000/results/res2003q2/cpu2000-20030326-01999.html

The two are from different companies. Some evidence of their difference is that the performance of the SPARC64 is faster than what one would expect by scaling clock frequency, so something about the systems other than clock frequency must be different. (The material has not been covered yet, but the Sun chip is statically scheduled while the Fujitsu chip uses a more advanced dynamically scheduled organization.)

Problem 2: The processors below have roughly the same SPEC CINT2000 peak (result) scores but are very different. (The links should be clickable in Acrobat Reader.)

ISA: Power, Implementation: POWER5, Decode: 5-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q3/cpu2000-20040804-03314.pdf
ISA: Itanium (IA-64), Implementation: Itanium 2, Decode: 6-way Superscalar* Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q1/cpu2000-20040126-02775.pdf

ISA: IA-32, Implementation: Xeon, Decode: 3-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q3/cpu2000-20040727-03291.pdf
ISA: \(\approx\) IA-32, Implementation: Athlon, Decode: 3-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030908-02502.pdf
ISA: SPARC V9, Implementation: SPARC64 V, Decode: 4-way Superscalar*
Disclosure: http://www.spec.org/osg/cpu2000/results/res2004q2/cpu2000-20040518-03044.pdf
(a) The performance of the processors, based on the peak result, are roughly the same. On the same graph plot the performance in the following ways:
- Using the SPEC peak (result) scores.
- Assume that performance is proportional to clock frequency. Determine the score of a processor by comparing its clock frequency to that of the SPARC64 and using that to scale the SPARC64 peak result.
- The table above shows how many instructions a processor can decode per cycle. (Four-way superscalar means four per cycle, see explanation below.) Determine the performance by comparing the number of instructions fetched per second to the SPARC64 and use that to scale the SPARC64 peak result.

What conclusions can be drawn from the plotted data?
*The following information is not needed to solve this assignment. The decode widths shown above are the maximum number of instructions that can be decoded per cycle. For any real program the number will be much lower due to a variety of factors, which will be covered later in the semester. One relatively minor factor is the instruction mix. The POWER5 (implementation), Itanium (ISA), and to a lesser extent the others limit the kinds of instructions that can be decoded together. More on this later in the semester. The decode widths for the Xeon and Athlon don't refer to IA-32 instructions: these processors take IA-32 instructions and break them into simpler instructions called micro-ops by Intel and (favoring marketing over descriptive accuracy) macro-ops by AMD. The three instructions per cycle for the Xeon and Athlon are actually three micro- or macro-ops per cycle.

The "Norm Clk" plot shows performance sealed using clock frequency. The value is \(s_{X}^{\prime}=s_{\text {sparc }} \frac{\phi_{X}}{\phi_{\text {sparc }}}\), where \(s_{X}^{\prime}\) is the scaled performance of system \(X\) and \(\phi_{X}\) is the clock frequency of system \(X\). The "Norm Ded" plot shows the performance sealed using decodes per second (not the same as decodes per cycle). The value is \(s_{X}^{\prime \prime}=s_{\text {sparc }} \frac{\phi_{X} d_{X}}{\phi_{\text {sparc }}}\), , whereare \(s_{X}^{\prime \prime}\) is the scaled performance of system \(X\) and \(d_{X}\) is the decode width (decodes per evcle) of system \(X\).

Scaled SPEC int Scores


Conclusions:
The obvious conclusion is that clock frequency is a poor indicator of performance since the predictions based on clock frequency are way off. Taking into account the number of instructions decoded per cycle is a better predictor of performance (than just clock frequency) but still far from perfect.

Taking into account decode rate, the performance of the Athlon system is underpredicted, whereas the others are overpredicted. This hints that the Athlon makes the best use of each instruction decode slot. The worst use of slots is made by the Xeon. Note that there are many other differences, for example, the number of instructions in a program, and the way the IA-32 processors split instructions, so that one can't conclude for sure that the Athlon is making the best use of its slots.
(b) The Xeon and Athlon systems in the disclosures above have about the same performance. AMD might argue that those disclosures don't show the full potential of the Athlon. Find a system that uses an Athlon and scores much better, and explain what accounts for the difference.

AMD might argue that the gec compiler used in the test system does not make the best use of the processor, and so the tested system does not show the full potential. The AMD system using a ASUS SK8N motherboard uses an identical Athlon FX-51 processor, but scores a higher 1447. The difference is probably due to the compiler. The higher-scoring system uses an Intel IA-32 compiler, the lower-seoring system uses gee which is not known for speed. (What it lacks in speed it makes up for in portability.)
(c) There is a system characteristic that affects the performance of benchmark mcf. What is it?

Cache size. The systems with the larger cache size do much better.
(d) Nominate a disclosure for The Most Desperate Peak Tuning award.

Of the five above I nominate the Primepower 650 for the award, because of the longest list of compiler flagg.

Problem 1: Do Problems 1 and 2 From Spring 2004 Homework 3 http://www.ece.1su.edu/ee4720/2004/hw03.pdf. After completing the problems look at the solution and assign yourself a grade. The maximum grade should be 10 points, divide the points between problems as you wish.

Problem 2: A new instruction, copyTreg rt, rs, will read the contents of register rt and rs and will write the contents of rs to the register number specified by the contents of register rt (not into register rt). For example,
```


# Before: \$1 = 4, \$2 = 0x1234, \$4 = 0

copyTreg \$1, \$2

# After: \$1 = 4, \$2 = 0x1234, \$4 = 0x1234;

# (Register \$4 written with contents of register 2.)

```

Note that this is a variation on Midterm Exam 1 Problem 3, with the destination, rather than the source, being specified in a register.
(a) Modify the pipeline below to implement this instruction.
(b) Add the bypass connections needed so that the code below executes correctly.
```


# Before: \$1 = 4, \$2 = 0x1234, \$4 = 0, \$5 = 0

addi \$1, \$0, 5
copyTreg \$1, \$2

# After: \$1 = 5, \$2 = 0x1234, \$4 = 0, \$5 = 0x1234;

```

Changes shown in red below. The major change is providing a path for the \(r t\) value to be used as a destination register, that is through the added multiplexer. It is connected to the output of the ALU mux so that it can make use of the already existing bypass connections. When copyTreg is in EX the ALU will be set to add its \(a\) input to zero (or otherwise set output \(x\) to whatever is on input \(a\) ). The ALU might already be using such an operation for the jal and jalr instructions. (The modification solves both parts (a) and (b) of this problem.)


Problem 3: In the problem above the register number to write to is in a register. Here consider copyFreg \(r t\), rs in which, like the test question, the register to copy from is in a register. That is,
```


# Before: \$1 = 2, \$2 = 0x1234, \$4 = 0

copyFreg \$4, \$1

# After: \$1 = 2, \$2 = 0x1234, \$4 = 0x1234;

# (Register \$4 written with contents of register 2.)

```

Explain a difficulty in implementing this instruction on the pipeline below without vitiating its sublime elegance.

The pipeline and ISA have been designed so that register read can start in the ID stage as soon as the instruction arrives (and in parallel with decoding activities). To implement copyFreg the register file has to be read twice, first using the register number found in the rs field of the instruction, call the retrieved value rsv, and a second time using rsv as the register number. The two register reads obviously cannot be done at the same time. There is no way to retrieve the register without either reading the register file twice in the same cycle (stretching the clock), having the instruction spend two cycles in ID (which would be inelegant, providing a new kind of stall), or having the instruction read the register file a second time while it is in EX (requiring a third read port or else introducing new structural hazards as instructions in ID and EX both vie for the same register port).


Problem 4: No, we are not vitiators. Instead consider copyBreg rd in which the source register to read is specified in the rs register of the preceding instruction, that value is written into the rd register of this instruction. (Okay, maybe we are vitiators.) For example,
```


# Before: \$1 = 2, \$2 = 0x1234, \$4 = 0

add \$0, \$1, \$0 \# Instruction below uses rs (\$1 here) of this insn.
copyBreg \$4

# After: \$1 = 2, \$2 = 0x1234, \$4 = 0x1234;

# (Register \$4 written with contents of register 2.)

```

Implement this instruction on the pipeline above (from the previous problem).
Changes shown in red below. The added multiplexer does have a small impact because its control signal needs to be generated close to the beginning of the cycle. As with the previous problem, by taking the signal at the output of the ALU mux bypassed values are available. In the code sample above the bypass connections are not needed. The "maybe we are vitiators' remark is the comment on the awkwardness of having one instruction use as a source operand the source operand of a preceding instruction. Such instructions are rare if they exist at all.


\section*{64 Spring 2004 Solutions}

Problem 1: The MIPS program below copies a region of memory and runs on the illustrated implementation. In the sub-problems below use only the bypass connections shown in the illustration.
(a) Show a pipeline execution diagram for the code running on the illustrated implementation for two iterations.

See below.
(b) Compute the CPI and the rate at which memory is copied in bytes per cycle assuming a large number of iterations.

Each iteration takes 9 cycles and contains 5 instructions so the CPI is \(\frac{9}{5}=1.8\). Each iteration copies four bytes of data and so the data copy rate is \(\frac{4}{9}\) bytes per cycle.
- Don't forget, when computing the number of cycles per iteration be sure not to count a cycle more, or less, than once.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline LOOP: \# Cycle & & 1 & 2 & & 34 & 5 & 6 & & 8 & 9 & & 01 & 11 \\
\hline 1w \$t0, 0 (\$a0) & IF & I & D E & & ME WB & & & & & IF & I & D & EX ME \\
\hline sw 0 (\$a1), \$t0 & & I & F ID & & & EX & ME & & VB & & & F & ID \\
\hline addi \$a0, \$a0, 4 & & & & F & & ID & EX & & & & & & IF -> \\
\hline \$ \$a0, \$a2 LOOP & & & & & & IF & ID & & & & & & \\
\hline ddi \$a1, \$a1, 4 & & & & & & & & & & & & & ME WB \\
\hline
\end{tabular}


Problem 2: Execution should be inefficient in the problem above.
(a) Add exactly the bypass connections needed so that the program above executes as fast as possible.
- Don't forget that branch uses ID-stage comparison units.
- Don't forget the store.

There are two solutions, conservative and aggressive. With the conservative solution the number of stall cycles will be reduced from two to one (zero would be better) in each case and the critical path length will not be increased (which is good, of course). In the aggressive solution all stalls will be eliminated but there might be critical-path impact, and so the clock frequency might be reduced.

The aggressive solution improves performance on the program above because the number of cycles per iteration is reduced from 9 to 5 , while any reduction in clock frequency would not be as drastic. For programs which do not make frequent use of the new bypasses performance would be lower because there would not be enough of a reduction in stall cycles to compensate for the lower clock frequency.

Note that one could have an aggressive load-store bypass and a conservative branch condition bypass, and vice versa.
In the illustration below the bypass paths for the conservative solution are shown in green, the bypass paths for the aggressive solution are shown in red, and the comparison unit (which was present but not shown in the original diagram) appears in blue.

(b) Show a pipeline execution diagram of the code on the improved implementation.
```


# Aggressive.

LOOP: \# Cycle 0
lw $t0, O($a0) IF ID EX ME WB IF ID EX ME ...
sw O(\$a1), \$t0
addi \$a0, \$a0, 4
bne \$a0, \$a2 LOOP
addi \$a1, \$a1, 4

```
```

0

```
0
    IF ID EX ME WB IF ID EX ...
    IF ID EX ME WB IF ID EX ...
        IF ID EX ME WB IF ID ...
        IF ID EX ME WB IF ID ...
            IF ID EX ME WB IF ...
            IF ID EX ME WB IF ...
                        IF ID EX ME WB
                        IF ID EX ME WB
# Conservative
LOOP: # Cycle 0
    lw $t0, O($a0) IF ID EX ME WB IF ID EX ME ...
    sw 0($a1), $t0
    addi $a0, $a0, 4
    bne $a0, $a2 LOOP
    addi $a1, $a1, 4
```

```
    IF ID -> EX ME WB IF ID -> ...
```

    IF ID -> EX ME WB IF ID -> ...
        IF -> ID EX ME WB IF -> ...
        IF -> ID EX ME WB IF -> ...
            IF ID -> EX ME WB
            IF ID -> EX ME WB
                        IF -> ID EX ME WB
    ```
                        IF -> ID EX ME WB
```

(c) For each bypass path that you've added show the cycles in which it will be used by writing the cycle number near the bypass path. If a bypass path goes to several places (for example, both ALU muxen) put the cycle number at the place(s) that use the signal.
(d) Re-compute the CPI and the rate at which memory is copied.

With the aggressive solution the CPI is $\frac{5}{5}=1$ and the copy rate is $\frac{4}{5}$ bytes per cycle. With the conservative solution the CPI is $\frac{7}{5}=1.4$ and the copy rate is $\frac{4}{7}$ bytes per cycle.

Problem 1: Suppose code like the memory copy program below (from Homework 3) appears frequently enough in the execution of programs so that new instructions should be added to the ISA to allow improved execution. (It does and they have been.)

Following the points below devise new instruction(s) that can be used to write a new memory copy loop that would execute more efficiently than is possible with existing MIPS-I instructions. A goal is to copy at the rate of two bytes per cycle. See the subparts after the bulleted points below.

- The instructions must use the existing MIPS formats.
- An instruction can do more than one thing (as long as it follows the points below). For example, an instruction that does more than one thing is a post-increment load. To reach the two bytes / cycle limit one might need to combine a branch with something.
- The instructions cannot use implicit registers. (A register is implicit if it does not appear in the encoded instruction. For example, register 31 is implicit in the jal instruction.)
- To achieve two bytes per cycle the instructions might need to do something unusual with operands. Please ask if you're not sure if something is too unusual.
- As with all other ordinary instructions, the new instructions must advance one stage per cycle (unless stalled, if so they would sit idle).
- The modified pipeline must still use the same memory port and no new memory ports can be added.
- Modifications such as bypass paths can be added to speed the instructions.
(a) Show an example of each new instruction and show how it is coded.
(b) Show how the instructions would be implemented on the pipeline.
(c) Write a memory copy program using the new instructions.
(d) Show a pipeline execution diagram for the memory copy code.
(e) A two bytes per cycle solution would require doing something interesting for the branch. Explain what that is and show a pipeline execution diagram for the memory copy loop finishing a copy (where the interesting stuff would be done).

Solution starts on next page (not counting this sentence).

The solution adds two instructions, an indexed-looping (IL) load, and an indexed-incrementing (II) store:

```
($s0) lw,il $t0, ($a2-$s0)
($s0-) sw,ii ($a3-$s0), $t0
```

The IL load, a Format $R$ instruction, has two source operands, a base ( $r$ s, in the example a2) and an index (rd, in the example $\mathbf{s 0}$ ), and a destination ( $\mathbf{r t}$, in the example t 0 ).

If the index is zero the instruction does nothing (it acts like a nop). If the index is non-zero then it loads the value at address base - index (in the example a2-s0) into register rt. It also does a delayed control transfer to its own address (it branches to itself). In the example, the 1 w , il will jump to itself (with sw, il executing in the delay slot). Note that the rd register works something like a predicate, which is why it is written using the syntax of predicated instructions (the ( s 0 ) at the start of the instruction).

The indexed-incrementing store is also a Format $R$ instruction, it has three source operands, a base (rs), an index $(r d)$, and a store value ( $r t$ ). If the index is zero the instruction does nothing. Otherwise, it stores the rt value at address rsv - rdv and it writes register rd with rdv - 4. (Unlike most other MIPS instructions, a single register field ( $r$ d ) is being used to specify both a source and destination.) Note that the $1 \mathrm{w}, \mathrm{il}$ and sw , ii instructions compute their effective addresses in the same way but that the sw, ii decrements the index while the lw,il does not change the index.

The program below uses the new instructions to copy a region of memory. The program starts with the same register values $(a 0, a 1$, and $a 2)$ as the original program and does the same thing. Unlike the original program it uses two instructions before the loop. The first computes the size of the region to copy. The second computes the end of the region to copy data to. (The end of the region to copy data from is already provided, in a2.) The loop label (LOOP) is shown for illustrative purposes, but the assembler ignores it because the $1 \mathrm{w}, \mathrm{il}$ always branches to itself.)

```
    # $a0 Start address of region to copy.
    # $a1 Address of memory to copy to.
    # $a2 Address at end of region to copy. (Don't copy $a2, do copy $a2-4.)
    sub $s0, $a2, $a2 # Size of region to copy.
    add $a3, $a1, $s0 # Address at end of region to copy to.
LOOP :
    ($s0) lw,il $t0, ($a2-$s0) # Load word and branch if $s0 not zero.
    ($s0-) sw,ii ($a3-$s0), $t0 # Store and decrement $s0.
    # Same program, with pipeline execution diagram.
    # Cycle 0
    sub $s0, $a2, $a2
    IF ID EX ME WB
    add $a3, $a1, $s0
    IF ID EX ME WB
LOOP:
    ($s0) lw,il $t0, ($a2-$s0) IF ID EX ME WB (1st iteration)
    IF ID EX ME WB (2nd iteration)
    ($s0-) sw,ii ($a3-$s0), $t0 IF ID EX ME WB (1st iteration)
    IF ID EX ME WB (2nd iteration)
    # Cycle 0
```

The diagram below shows how the new instructions might be implemented, changes are shown in blue. A third read port is added to the register file so that the store instruction can read the index, base, and store value simultaneously. A comparison unit is added to the ID stage to check for the end of loop condition (index zero); note that several bypass connections are needed. A decrementer ( -4 ) is added to the EX stage (used by sw,ii) and pipeline latch registers are added to pass the new value of the index down the pipeline.

The cycle numbers, in purple show when the labeled lines will be used for the pipeline execution diagram above.

The hardware for predication is not shown. That hardware would replace the dst value with a zero and change the memory operation to a nop. (The memory operation input is also not shown.)


## LSUEE 4720

Homework 5 solution
Due: 21 April 2004
Problem 1: One question when extending an ISA from 32 to 64 bits is what to do about the shift instructions. Because of the way that the shift instructions are encoded in MIPS two new shifts (of each type) were added to MIPS-64.
(a) What do you think the MIPS-32 sra instruction should do in MIPS-64? Remember that an implementation of MIPS-64 must run MIPS-32 code correctly. Please answer this question before answering the next parts (but feel free to look at the questions). Hint: Any serious answer will get full credit. A smart-alec answer will get full credit only if it's particularly witty.

Just shift the lower 32 bits, sign extend using bit position 31 . Leave the high 32 bits unchanged. For example:

```
# My idea for how sra on a 64-bit machine should work.
# Before $r1 = 0x8888 8888 8888 8888
sra $r1, $r1, 1
# After $r1 = 0x8888 8888 C444 4444 (Spaces added for clarity.)
```

(b) Give two reasons why the MIPS-32 sra (not srav) instruction could not be used for all right arithmetic shifts needed in a 64 -bit program.

It can't specify a shift of more than 32 bits since the sa field is only five bits. If it sign extended using bit 63 as the sign it would not work for 32 -bit code, if it sign extended on bit 31 it would not be appropriate for 64 -bit code.
(c) What are the new MIPS-64 shift right arithmetic instructions? Give the mnemonics. DSRA and DSRA32.
(d) Why were two (as opposed to one) new shift instructions of each type added to MIPS-64? Because they use the existing five-bit sa field which can't specify a full range of shifts.

Problem 2: Do the Problem 2 (a) through (d) from the Fall 2003 EE 4720 final exam (the one on floating point instructions). (http://www.ece.lsu.edu/ee4720/2003f/fe.pdf) Do not look at the solution until after you have solved the problem or gave it a good try.

See http://www.ece.lsu.edu/ee4720/2003f/fe_sol.pdf.

Problem 3: In the diagram below the we pipeline latches carry write enable signals for use in floating point writeback. If the functional units were arranged differently the we pipeline latches could be used as a reservation register (for detecting WF structural hazards).

(a) Redraw the diagram with that arrangement. Hint: Try to use the we signal in the diagram above for a reservation register. Figure out why that won't work and come up with a solution. See the next page.
(b) Suppose the ID stage has boxes uses FP ADD and uses FP MUL to detect which (if any) floating point functional unit an instruction would use. Design the control logic to generate a stall signal if there would be a write float structural hazard.

See the next page.
(c) Add the connections necessary for a lwc1 instruction. Include the connections needed to detect a WF structural hazard (as was done for ADD and MUL in the previous part).

See the next page.

The multiply and add functional units are rearranged so that they finish "together" rather than start "together." In the original arrangement a particular we pipeline latch is always a fixed distance from ID , but their distance from WF depends upon which instruction they carry. For example, the first we latch is always one cycle from ID, if it is carrying an add instruction it is four cycles from WF but if it is carrying a mul it is six cycles from WF. In a reservation register a particular bit position describes an instruction a particular distance from WF and so the we pipeline latches in the original diagram cannot be used as a reservation register.

In the modified pipeline, below, the we is a fixed distance from WF and so it can, and is, used as a reservation register. Instructions using the FP add check the second we pipeline latch and if it holds a 1 an ID stall signal is asserted, otherwise the add (in particular, the we signal, register number ( fd ), and a control signal for the WF multiplexor) is inserted into the pipeline. Note that the operands themselves enter whether or not ID is stalled, if stalled then later at WF the result will be ignored. Similar logic is shown for the lwc1 (and other FP loads) instruction. Note that instructions using the multiply unit never check the reservation register, that is because they take the longest (we're ignoring divide) and so no other instruction can use WF at the same time.

The diagram also shows the control logic for the WF multiplexor.


Problem 1: Read the Microprocessor Report article on the IBM PowerPC 970 (a.k.a. the G5), used in a popular person computer. The article is available at
http://www.ece.lsu.edu/ee4720/s/mprppc.pdf. If accessing from outside the lsu.edu domain provide user name ee4720 and the password given in class. Answer the following questions: (Please read the entire article, additional questions might be asked in a future assignment.)
(a) One might infer from the second paragraph that deeper pipelines are used to inflate clock frequencies solely for marketing purposes. Why do deeper pipelines allow higher clock frequencies? Are there reasons other than marketing to do that?

The clock frequency is set to the highest value that will allow logic at the end of the critical path (from one pipeline latch to another) to stabilize at the correct output (with a suitable margin). To increase the number of stages, logic that spanned one stage is split into pieces (or redesigned as several pieces), (hopefully) reducing the length of the critical path and so allowing for a higher clock frequency.

The marketing benefft of a deeper pipeline is the higher clock frequency, because unprepared buyers might have no other way to estimate performance. Deeper pipelines (with their higher clock frequencies) actually do give higher performance (as long as they are not too deep) because instructions are fetched at a faster rate (the higher clock frequency) and because the number of stalls (in dynamically scheduled systems due to full a ROB because of scheduling constraints on dependent instructions), while higher, is not high enough to eliminate the benefit of more frequent fetches. Therefore there is more than just a marketing benefit to higher clock frequencies. (Deepening pipeline depth further will yield diminishing returns as the pipeline latch overhead becomes a larger fraction of the clock period and as dependent instructions must be scheduled further apart.)
(b) The article describes the PPC 970 as a 5 -way superscalar processor, which is consistent with the definition used in class. How could overzealous marketing people inflate that number using features of the microarchitecture? Describe the specific feature. Why would that be overzealous?

The PPC can fetch eight instructions per cycle, so that could be the rationale for calling it an eight-way superscalar machine. However there are at least two five instruction per cycle bottlenecks and so no program could execute at eight instructions per cycle. Within the fetch pipeline instructions are formed into five-instruction groups. At most one group per cycle can be dispatched to the issue queues, that's one bottleneck. At most one group per cycle can commit, that's another bottleneck.

The following two problems are nearly identical to Spring 2003 Homework 6. The main difference is in the stages that are used. It is okay to peek at the solutions for hints, for best results leave twelve hours between looking at those solutions (or solutions to similar problems) and completing this assignment.

Problem 2: Show the execution of the MIPS code fragment below for three iterations on a fourway dynamically machine using Method 3 (physical register file) with a 256 -entry reorder buffer. Though the machine is four-way, assume that there can be any number of write-backs per cycle. Use Method 3 as described in the study guide at http://www.ece.lsu.edu/ee4720/guides/ds.pdf with for the following differences:

- The FP multiply functional unit is three stages (M1, M2, and M3) with an initiation interval of 1 .
- Assume that the branch and branch target are always correctly predicted in IF so that when the branch is in ID the predicted target is being fetched.
- There are an unlimited number of functional units.
(a) Show the pipeline execution diagram, indicate where each instruction commits.
(b) Determine the CPI for a large number of iterations. (The method used for statically scheduled systems will work here but will be very inconvenient. There is a much easier way to determine the CPI.)

```
    # Solution
    # Cycle 
LOOP:
    ldc1 f0, O(t1) IF ID Q RR EA ME WB C
    mul f2, f2, f0 IF ID Q RR M1 M2 M3 WB C
    bneq t1, t2 LOOP IF ID Q RR B WB C
    addi t1, t1, 8 IF ID Q RR EX WB C
    ldc1 f0, O(t1) IF ID Q RR EA ME WB C
    mul f2, f2, f0 IF ID Q RR M1 M2 M3 WB C
    bneq t1, t2 LOOP IF ID Q RR B WB C
    addi t1, t1, 8 IF ID Q RR EX WB C
    ldc1 f0, 0(t1) IF ID Q RR EA ME WB C
    mul f2, f2, f0 IF ID Q RR M1 M2 M3 WB C
    bneq t1, t2 LOOP IF ID Q RR B WB C
    addi t1, t1, 8 IF ID Q RR EX WB C
    # Cycle 
```

The CPI is $\frac{3}{4}=0.75$. The hard way of computing the CPI is completing the pipeline execution diagram until there is a repeating pattern. With a 256 -entry reorder buffer that will take a long time. Don't even try! The easy way is to find the critical path through the program (not the hardware logic). The critical path must be through loop carried dependencies, for this loop there are two, carried by t 1 and f 2 . There is a single instruction per iteration that updates t1 and that has a latency of zero, so the path through t1 can execute at a rate of one iteration per cycle, which is the same as the fetch rate. The path through $f 2$ is also through a single instruction, the multiply, however that has a latency of 2 (takes 3 cycles to compute) and so the fastest it can execute is 3 cycles per iteration. The processor will initially fetch one iteration per cycle and the addi instruction will be able to keep up, while the mul. d will fall behind. Eventually the reorder buffer will fill, when that happens instructions will only be fetched when new space opens up, which will be when the multiply instructions commit. Therefore fetch will drop to three cycles per iteration or a CPI of $\frac{3}{4}$.

Note that the load is not on the critical path. It does provide data for the multiply and it is dependent on data from a previous iteration, t 1 , but it has its data ready before the multiply needs it. (This is only so because of the assumption that the load always hits the cache. With cache misses the situation is more complex.)

The Spring 2003 version of this problem did not include the RR stage but the CPI in both cases is the same. Though not in this case, deepening the pipeline (here with the RR stage) can have an impact on performance, for example, when there are branch mispredictions.

Problem 3: The execution of a MIPS program on a one-way dynamically scheduled system is shown below. The value written into the destination register is shown to the right of each instruction. Below the program are tables showing the contents of the ID Map, Commit Map, and Physical Register File (PRF) at each cycle. The tables show initial values (before the first instruction is fetched), in the PRF table the right square bracket "]" indicates that the register is free. (Otherwise the right square bracket shows when the register is freed.)
(a) Show where each instruction commits.
(b) Complete the ID and Commit Map tables.
(c) Complete the PRF table. Show the values and use a "[" to indicate when a register is removed from the free list and a "]" to indicate when it is put back in the free list. Be sure to place these in the correct cycle.

Solution shown below. In this solution the RR stage is not used because it was not shown in the original assignment. (That's not wrong, it just means that RR can overlap with $Q_{1}$ meaning that an instruction entering the Q stage can read the physical register file in the same cycle if it's ready to go.)

```
# Solution
```



```
# Cycle 
Physical Register File
```



```
96 309
95 606 ]
94 058 ]
93 285 ]
92 1234
91 518 ]
90 207 ]
# Cycle 
```


## LSUEE 4720

Homework 7 solution
Due: 5 May 2004
The PPC 970 which was the subject of a question in Homework 6 is very similar to the POWER4 chip, the main differences being that the POWER4 lacks the packed-operand instructions and POWER4 includes two processors on a single chip.

Answer the following questions about the POWER4 based on information in "POWER4 System Microarchitecture," by Tendler et al, available via http://www.ece.1su.edu/ee4720/doc/power4.pdf. The questions can be answered without reading the entire paper. In particular, there is no need to read past page 17.

Problem 1: Translate the following terms, as used in class, to their nearest equivalent in the paper.

- Integer Instruction $\rightarrow$ Fixed-point instruction.
- Instruction Queue $\rightarrow$ Issue queue.
- Reorder Buffer $\rightarrow$ Group completion table.
- Physical Register $\rightarrow$ Rename register.

Problem 2: The pipeline execution diagram below shows MIPS code on the dynamically scheduled system described in the study guide.
(a) Re-draw the diagram using the stages from POWER4. (Do not translate the instructions into the POWER assembly language.) Just show one iteration and assume that the four instructions are formed into one group. Also assume that the branch does not have a delay slot. Use stages F1, F2, and F3 for the multiply.

See diagram. Note that in POWER4 there is a mandatory one-cycle gap between two dependent instructions. Yuk!
(b) In your diagram identify the fetch and execute pipelines, as defined in class.

See diagram. Instructions enter the fetch pipeline in IF and exit the fetch pipeline in MP where they are put in (dispatched to) issue queues. They wait in the issue queues until the scheduler chooses them, at which time they enter the execute pipeline.

```
    # Solution
    # Cycle 
LOOP:
ldc1 f0, O(t1) IF IC DO D1 D2 D3 XF GD MP IS RF EA DC FM WB XF CP
mul f2, f2, f0 IF IC D0 D1 D2 D3 XF GD MP IS RF F1 F2 F3 WB XF CP
addi t1, t1, 8 IF IC DO D1 D2 D3 XF GD MP IS RF EX WB XF CP
bneq t1, t2 LOOP IF IC DB D1 D2 D3 XF GD MP IS RF EX WB XF CP
# Note: DB is an abbreviation for BP and DO (branch uses both.)
# XF -> Xfer; FM -> Fmt; IS -> ISS
# Fetch Pipeline Stages: IF IC DO D1 D2 D3 XF GD MP
# Execute Pipeline Stages: IS RF EX,EA,DC,FM,F1-F6 WB XF
```

Problem 3: The POWER4 uses what is commonly called a hybrid predictor in which each branch is predicted by two different predictors and a third predictor predicts the prediction to use. One
predictor is something like the bimodal predictor discussed in class and the other is something like the gshare predictor discussed in class.
(a) Provide a code example in which the bimodal predictor described in class will do better than the POWER4's almost equivalent predictor. (Ignore the selector.)

What the PPC paper calls a local predictor is called a bimodal predictor in class, except that the PPC local predictor uses only a 1-bit entry in the BHT (rather than 2). Consider the loop below:

```
# Three-iteration loop.
LOOP:
bneq r1, r2, LOOP T T N T T N T T N
nop
```

The predictor used in class would have an accuracy of $66.7 \%$, misspredicting the not-taken executions of the branch. The local predictor would have an accuracy of $33.3 \%$, because it would only correctly predict the second consecutive taken branch.
(b) How might the POWER4 designers justify the differences with the bimodal predictor given the lower performance in the example above?

For a given amount of storage, the PPC local predictor can have twice as many entries. Compared to one using a two-bit counter, the PPC would make more mispredictions due to using just one bit, but it would make fewer mispredictions due to collisions and so overall it would perform better (if the bimodal had many mispredictions due to collisions).
(c) Provide a code example in which the gshare predictor described in class outperforms the POWER4's almost equivalent predictor. (Ignore the selector.)

One difference between the PPC's global predictor and gshare is that in PPC the global history register has one bit for each fetch group (not the same as a dispatch group), whether or not it includes a branch. Fetch groups can be as large as eight instructions, dispatch groups can be as large as five instructions. When updating the GHR a fetch group without a CTI is treated like one containing a not-taken branch.

Consider the loop below:

```
# Five-iteration loop.
LOOP:
    # ...
    # ... seventeen instructions, none of them are CTIs ...
    #
    bneq r1, r2, LOOP T T T T N T T T T N T T T T N ...
    nop
```

Each iteration would span at least three groups ( at least three, because cache line boundaries might force contiguous instructions to be in separate fetch groups). Assume that the loop can be fetched in exactly three groups. For each iteration three bits would be shifted into the GHR, one for each group. A possible GHR value used for predicting the loop branch would be TnnTnnTnnTn, where n is the value inserted for groups not holding a branch (it would actually be a zero, for not taken). Since the GHR is eleven bits it can see three and part of a fourth iteration. Therefore the predictor would not be able to tell whether it was in the fourth iteration (where the branch would be taken) or the fifth iteration (where the branch would not be taken), in both cases the GHR would be nnTnnTnnTnn.

In a conventional gshare predictor the GHR would only include branch outcomes, and so for the code above it could easily distinguish the fourth and fifth iterations.
(d) How might the POWER4 designers justify the differences with the gshare predictor given the lower performance in the example above?

The logic to update and recover the GHR might be simpler.

## 65 Fall 2003 Solutions

Problem 1: Look at the following SPEC CINT2000 disclosures for these Dell and HP Itanium 2 systems: HP: http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030711-02389.html
Dell: http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030701-02367.html. (Note: Links are clickable within Acrobat reader.)

The CPU performance equation decomposes execution time into three components, clock frequency, $\phi$, instruction count, IC, and CPI. For each component determine if its value on the two systems is definitely the same, probably the same, probably different, definitely different. Hint: The answer for the clock frequency is easy, the others require a little understanding of what IC and CPI are. Briefly justify your answers.

Clock frequency: The same, the clock frequency is given in the disclosures.
Instruction count (IC): definitely different because the benchmarks were compiled with different compilers.
CPI: probably different. The HP system is faster, some of that difference may be due to the compiler and some may be due to the system (amount of memory, speed of memory bus, etc.). Both system details (such as memory speed) and compiler details (instruction mix and scheduling) affect CPI and since both are different on the two systems the only way CPI could be identical would be by coincidence. (If the CPIS' were identical then the HP systems's better performance would be due solely to the compiler.)

Problem 2: Though one may normally think of an implementation as a microprocessor chip, the definition can also include other parts of the system, such as memory and even disk. Why is that important in the problem above?

Because the two systems used the same implementation, a 1.5 GHz Itanium 2, but other parts of the system, such as memory and disk, differed and that could lead to different execution times (and CPIs).

Problem 3: Differences in ISA, compiler, and implementation all affect the execution time of programs, and the impact of these factors can vary from program to program. For example, an implementation with faster floating point will have a larger impact on programs that do more floating-point computations.

From a look at the SPEC disclosures one can see that the fastest program on one system may not be the fastest program on another. (Use the int2000 results. From the spec CPU2000 results page find the configurable query form and request a page sorted by "Result" in descending order. It would be helpful to include the processor and compiler in the results. If your system is slow omit results before 2002.) For example, the Dell system from the first problem ran vortex fastest (of all the benchmarks), while the HP system ran mcf fastest. In this case the difference in fastest benchmark could not be the ISA, but it could be the compiler or the implementation. Call the speed ranking of benchmarks for a system its character. The character of the Dell system is vortex, gcc, eon, ... (benchmarks from fastest to slowest) and the character of the HP system is mcf, vortex, gcc, ....

The differences in character are due in part to the ISA, compiler, and implementation. Using the SPEC CINT2000 disclosures determine which is most important in determining character. Please do not try to look at all disclosures, just enough to determine an answer, even if that answer might change if you were to look at more.

In your answer, state which (ISA, compiler, or implementation) is most important, which disclosures you looked at, and how you drew that conclusion. This question is easy to answer (once it's understood).

As best you can explain why a particular factor is most important and why it is least important. You are not expected to answer this question very well, most of the material has not been covered yet. Don't take too much time and do your best with what has been covered and what you already know.

Additional Information:
Here are some ISAs and implementations of processors listed:
IA-32 ISA: (includes variations) implemented by Pentium 4 (and other Pentia) Xeon, Athlon, Opteron. Power Architecture ISA: Implementated by POWER4, RS64IV. Itanium ISA: Implementated by Itanium 2. Alpha ISA: Implementated by Alpha $21164,21264,21364$. SPARC V9 ISA: Implementated by SPARC64V, UltraSPARC III Cu MIPS ISA: Implementated by R14000

The solution compares the characters of pairs of disclosures, one pair to bring out ISA effects, a pair for compiler effects, and a pair for implementation effects. When selecting a pair to compare to systems are found which differ on the thing being looked at (ISA, compiler, or implementation) but are as similar as possible with everything else. Once a pair is found the ranking for the integer benchmarks in each system is found.

To show how different the character is, the sum of the absolute difference in ranks of the first five programs for one system to the other is found. (This is probably easier then it sounds.) For example, when looking at ISA effects, the ranking of the first three Xeon programs is the same as in the Itanium system. The fourth program for Xeon is ranked 12 th for Itanium, so the distance is 8. The fifth Xeon program is ranked eighth in the Itanium system, for a difference of 3 . The sum of differences is 11.

Note: To draw any real conclusions from this analysis one would need to look at more than three pairs of systems, which is too much work for one homework assignment.

ISA Effects. Compare two systems with different ISAs but otherwise as similar as possible. One system below implements Itanium 2, the other IA-32. They both use the Intel Intel $\mathrm{C} / \mathrm{C}++$ compiler.

Dell PowerEdge 3250, Itanium 2 :
http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030701-02367.html. Intel Xeon (3.06GHz, 533MHz bus)
http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030630-02338.html

Program Ranks: (First is fastest.)
Xeon: vortex, gcc, eon, gap, perlbmk, twolf, gzip, crafty, parser, bzip2, vpr, mcf
Itanium: vortex, gcc, eon, mcf, crafty, twolf, bzip2, perlbmk, vpr, gzip, parser, gap
Rank Diffs: $0 \quad 0 \quad 0 \quad 8 \quad 3=11$
The fastest three programs in the two systems (above) are the same, but the order of the other programs is very different.

Compiler affects: Look at two systems with the same implementations (and of course the same ISA), but different compilers. A possible pair are the two systems used in Problem 2 (the difference in memory and IO are still there, systems with identical hardware would be better, but even so there would be differences due to the operating system).

Both systems use the same implementation of Itanium, Itanium 2. Differences: compiler, also operating system and hardware (other than Itanium 2). (In a better comparison the same operating system would be used. However it seems that for most (maybe all) systems, only one compiler is used for a particular operating system [gee for Linux, Intel for Windows, etc] ).

Top programs on HP (Unix): mcf, vortex, gcc, eon, twolf
Top programs on Dell: vortex, gcc, eon, mcf, crafty, twolf
Rank Differences: 3 1 $10 \begin{array}{llll} & 3 & 1\end{array}$

Implementation Effects. Compare two systems with the same ISA but different implementations. Two implementations from the same manufacturer may be similar, but one can be pretty certain that intel's lawyers made sure the AMD Athlon had little in common with Intel's Xeon (similar to the Pentium). Therefore, will compare Xeon and Athlon.

AMD Athlon FX-51, 2.2 GHz,
http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030908-02472.html
Intel Xeon (3.06GHz, 533MHz bus)
http://www.spec.org/osg/cpu2000/results/res2003q3/cpu2000-20030630-02338.html
Common IA-32 ISA (with small variations), Intel $\mathrm{C} / \mathrm{C}++7.0$ compiler (and libraries). Differences: Implementation. (The Athlon FX actually implements a 64-bit extension of IA-32, however it's unlikely that the intel compiler is emitting any AMD 64-bit instructions.)

Top programs on Athlon: vortex, eon, twolf, gcc, perlbmk
Top programs on Xeon: vortex, gcc, eon, gap, perlbmk, twolf, gzip
Distance: $0 \quad 1 \quad 3 \quad 2 \quad 0=6$
Using just this data, it would appear that ISA is most important for character, followed by compiler, then implementation. As stated before, there are way too few data points to draw any real conclusion.

Problem 4: The two procedures below are compiled with optimization on but with no special optimization options. Why might the second one run faster? (This is very similar to the classroom example.)

```
void add_array_first_one(int *a, int *b, int *x)
```

\{
for $(i=0 ; i<100 ; i++\quad$ $\quad a[i]=b[i]+* x ;$
\}
void add_array_second_one(int *a, int *b, int *x)
\{
int xval $=$ *x;
for $(\mathrm{i}=0 ; \mathrm{i}<100 ; \mathrm{i}++$ ) $\mathrm{a}[\mathrm{i}]=\mathrm{b}[\mathrm{i}]+\mathrm{xval}$;
\}

First program has to de-reference x (load value from memory) each iteration while the second program can load the value into a register before entering the loop, then just use the value in the register.

Problem 1: Unlike MIPS, PA-RISC 2.0 has a post-increment load and a load using scaled-index addressing. The code fragments below are from the solution to Problem 2 in the midterm exam the fragments show several MIPS instructions under "Combine" and a new instruction under "Into." For each "Into" instruction show the closest equivalent PA-RISC instructions and show the coding of the PA-RISC instruction. (See the references page for information on PA-RISC 2.0)
(The term offset used in the PA-RISC manual is equivalent to the term effective address used in class, and is not to be confused with offset as used in this class. Assume that the s field and cc fields in the PA-RISC format are zero.)

Show all the fields in the format, including their names and their values.

```
Combine:
    lbu $t1, 0($t0)
    addi $t0, $t0, 1
Into:
    lbu.ai $t1, 0($t0)+ # Post increment load.
; Solution:
    ldb,ma 1(%r2),%r1
    ; %r1 is the equivalent of $t1 above.
    ; %r2 is the equivalent of $t0 above.
    PA-RISC Completer Descriptions:
        m: Modify base register (r2 in example, modify it by adding displacement, 1).
        a: After. (Add the displacement after computing the address.)
    PA-RISC Format 5 Field Descriptions
    opcode: Opcode.
    rb: Register holding address base. (Address in this case.)
    im5: Increment amount. One, to match the MIPS addi instruction.
    s: * Space register number. The space registers allow 32-bit programs
                to address more than 4 GiB of memory by holding
                the high 64 bits of a 96-bit address. Not used in 64-bit code, in
                which case the s field is just used for two more bits of displacement.
    a: After. If 0, add displ. after load, if 1, add displ. before load.
    l: Always 1 for format 5 (displacement).
    cc: * Cache control hint. (0, no hint; 2, spatial locality; 1,3, reserved).
    ext4: Memory operation. O indicates load byte unsigned.
    m: Modify base register. If 1, write modified address to same register.
    t: Register in which to write loaded value.
    * You don't need to understand the description of this field.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline opcode & & rb & & im5 & & S & a & 1 & cc & ext4 & & m & \\
\hline & 3 & & 2 & & 1 & 0 & 0 & 1 & 0 & & 0 & 1 & 1 \\
\hline 31 & & 25 & & 20 & 16 & 1514 & 1313 & 1212 & 1110 & 9 & 6 & 55 & \\
\hline
\end{tabular}
```

```
Combine:
    sll $t1, $t1, 2
    add $t3, $a1, $t1
    lw $t4, 0($t3)
Into:
    lw.si $t4, ($a1,$t1) # Scaled index addressing.
; Solution
    ldw.s %r1(%r2), %r4
    ; %r1 is index register (equivalent to $t1 above, before the shift).
    ; %r2 is the base register (equivalent to $a1 above).
    ; %r4 is the destination (equivalent to $t4 above).
    ; Effective address (offset in HP terminology) is: ( %r1 * 4 ) + %r2
PA-RISC Completer Descriptions:
    s: Scale index. Multiply the contents of the index register (r1
            here) by the data size, (in this case multiply by 4).
```

    PA-RISC Format 4 Field Descriptions
        opcode: Opcode.
    rb: Register holding address base. (Address in this case.)
    rx: Register holding index.
    s: * Space register number. The space registers allow 32-bit programs
        to address more than 4 GiB of memory by holding
                        the high 64 bits of a 96-bit address. Not used in 64-bit code, in
                        which case the s field is just used for two more bits of displacement.
        u: Scale. If 1, shift index by "data size". Shift by 2 for 4-bytes, etc.
    l: Always 0 for format 4 (indexed addressing).
    cc: * Cache control hint. (0, no hint; 2, spatial locality; 1,3, reserved).
    ext4: Memory operation. 2 indicates load word (32 bits) unsigned.
    m : Modify base register. If 1, write modified address to same register.
    t: Register in which to write loaded value.
    * You don't need to understand the description of this field.


Problem 2: The code fragment below runs on the implementation illustrated below.
(a) Show a pipeline execution diagram for the code fragment on the implementation up to the second fetch of the sub instruction; assume the branch will be taken.
(b) Show the value of the labeled wires (A, B, and C) at each cycle in which a value can be determined.

For maximum pedagogical benefit please pay close attention to the following:

- As always, look for dependencies.
- Pay attention to the RAW hazard between sub and sw and the RAW hazard between andi and bne.
- Make sure that add is fetched in the right time in the second iteration.
- Base timing on the implementation diagram, not on rules inferred from past solutions.


Problem 3: Consider the implementation from the previous problem, repeated below. For the jr instruction the ALU sets its output to whatever is at its top input. Note: This was omitted from the original problem.
(a) There is a subtle reason why the implementation cannot execute a jr instruction. What is it? Modify the hardware to correct the problem.

The PC holds bits $31: 2$ of the address, but the register value sent through the ALU to the PC will be the entire address. If nothing special is done then the jump will be to the address times four. The solution is to have the ALU perform a two bit right shift.
(b) There is a reason why it cannot execute a jalr instruction. What is it? Modify the hardware to correct the problem.

The ex_mem_alu pipeline latch is on the path used to put the jump target in the PC and to put the return address in the register file. Therefore, the jalr instruction can't do both. A solution would be to add a path from EX to the PC multiplexor for the jump address. Changes shown in red bold.


Problem 1: Design the control logic for the store value multiplexor (the one that writes pipeline latch ex_mem_rtv). The control logic must be in the ID stage. Hint: This is a fairly easy problem.

Changes for solution shown in red. Notice that the control logic doesn't check whether the instruction is a store. If it's not a store the store value multiplexor is not used and so it doesn't matter how it's set.


Problem 2: One problem with a post-increment load is storing the incremented base register value into a register file with one write port. Suppose a post-increment, register-indirect load were added to MIPS and implemented in the pipeline on the next page. This post-increment load does not use an offset, instead the effective address is just the contents of the rs register.

One option for storing the incremented base register value is to stall the following instruction and write back the value when the bubble reaches WB. We would like to avoid stalls if we have to, so for this problem design hardware that will use the WB stage of the instruction before [sic] or after the post-increment load if one of those instructions does not perform a writeback. For example:

```
bneq $s0, $s1, SKIP (Not taken)
lw $t1, ($t2)+
j TARG
add $s3, $s1, $s2
lw $t1, ($t2)+
sub $s4, $s5, $s6
```

The first post-increment load could writeback when either the bneq or the $j$ were in the WB stage since neither performs writeback. The second post-increment load would have to insert a stall.
(a) Show the hardware needed to implement the post-increment load in this way.

- Remember that this load does not have an offset.
- Use a $=$ PIL box to identify post-increment loads (input is opcode, output is 1 if it is a post-increment load, 0 otherwise).
- A stall signal is available in each stage; if the signal is asserted the instruction in that and preceding stages will stall and a nop instruction will move into the next stage (for each cycle hold is asserted).
- Show any new paths added for the incremented value, perhaps to the register file write port (which still has one write port).
- Add any new paths needed to get the correct register number to the register file write port.
- Ignore bypassing of the incremented address to other instructions.
- Show the added control logic, which does not have to be in the ID stage. (In fact it would be difficult to put all of control logic for this instruction in the ID stage.)
- Last but not least, a design goal is low cost, so add as little hardware as necessary to implement the instruction.
(b) If you're like most people, you didn't worry about precise exceptions when solving the previous part. Explain how the need for precise exceptions can complicate the design.

If the post-increment raises an exception in Mem then it might be too late to prevent writing back the incremented address if it's using the previous instruction. Another case is the post-increment load writing back using the next instruction. If the next instruction raises an exception one would have to make sure that despite being squashed it still wrote back the incremented address.

Changes shown in red. In this solution the ALU computes the incremented address, a multiplexor is added so the unincremented address can be sent to the memory's address input. An alternative solution would have an adder dedicated to incrementing the address; with such an adder one would not need the multiplexor at the memory address input.

The control logic, in EX, generates three signals, next, prev, and stall EX. Signals next and prev are put in pipeline latches, stall EX goes to control logic (not shown) to stall EX, ID, and IF. The prev signal sets the pipeline to write back the incremented address in the WB of next previous instruction (which is in the next stage), this is done by having the incremented address and the rs field skip ahead one stage. The next signal has the rs field and incremented address hold back one cycle by routing them through an extra set of registers, ALU, rs, and $n$ in the MEM stage.

For lower cost, the ALU register added to the Mem stage can be eliminated. Instead, one could use an enable signal to hold the value in the Mem/WB.ALU latch.


Problem 3: Answer these questions about interrupts in the PowerPC, as described in the PowerPC
Programming Environments Manual, linked to
http://www.ece.lsu.edu/ee4720/reference.html.
(a) Listed below are the three types of interrupts using the terminology presented in class. What are the equivalent terms used for the PowerPC.

- Hardware Interrupt

Asynchronous Exception

- Exception

Synchronous Exception

- Trap

Trap?
(b) In which register is the return address saved?

It is saved in SSRO (save/restore register 0).

66 Spring 2003 Solutions

At the time this was assigned computer accounts and solution templates were not ready. If they become available they can be used for the solution, either way a paper submission is acceptable.

Problem 1: When compiling code to be distributed widely one should be conservative when selecting the target ISA but less caution needs to be taken with the target implementation. Explain what "conservative" and less caution mean here, and explain why conservatism and in one case less caution in the other can be taken.

Conservative means choosing an ISA variation that most users' computers implement (rather than the variation that would give best performance). Less caution means selecting an implementation that fewer people have but that would give better performance.

Conservatism is necessary for ISA selection because a computer won't run software for an incompatible ISA. On the other hand, a computer will run software compiled for a different implementation, though not as fast as if compiled for the same implementation.

Problem 2: Based on the SPECINT2000 results for the fastest Pentium and the fastest Alpha, which programs would a shameless and unfair Alpha advocate choose if the number of programs in the suite were being reduced to five. Justify your answer.

At the time this solution was written, the fastest Pentium ran at 1130 SPECint2000's and the fastest Alpha ran at 928 SPECint2000's.

The advocate would choose the five programs which performed best compared to the Pentium. They are from best to worst (of 5) mef, vpr, bzip2, twolf, crafty. The last, crafty, is faster on the Pentium so the advocate might want just four programs. The table below shows the benchmark run times and the ratio of run times (Pentium divided by Alpha), sorted by ratio.

| Benchm. | Pentium Alpha | Ratio |  |
| :--- | :--- | :--- | :--- |
| mcf | 231 | 123 | 1.88 |
| vpr | 210 | 159 | 1.32 |
| bzip2 | 174 | 150 | 1.16 |
| twolf | 326 | 292 | 1.12 |
| crafty | 84.8 | 98.4 | 0.86 |
| gcc | 88.7 | 112 | 0.79 |
| vortex | 113 | 145 | 0.78 |
| parser | 170 | 256 | 0.66 |
| eon | 82.5 | 132 | 0.63 |
| perlbmk | 130 | 208 | 0.63 |
| gzip | 111 | 240 | 0.46 |
| gap | 75.2 | 164 | 0.46 |

Problem 3: The Pentium 4 can execute at a maximum rate of three instructions (actually, microops, but pretend they're instructions) per cycle (IPC), the Alpha 21264 can execute at most 4 IPC and the Itanium 2 can execute at most 6 IPC . Assume that the number of instructions for perlbmk, one of the SPECINT2000 programs, is the same for the Alpha, Itanium 2, and Pentium 4 (pretending micro-ops are instructions, if you happen to know what micro-ops are).
(a) Based upon the SPECINT2000 results (not base) for the perlbmk benchmark, which processor comes closest to executing instructions at its maximum rate? ("Its", not "the".)

The first thing to figure out is just what is meant by "closest to ... its maximum rate?" Another way of stating the question is: which wastes the fewest instructions?

Lets suppose that each processor was executing at its maximum rate. The total number of instructions executed would be the IPC $\times \phi \times t$, where $\phi$ is the clock frequency and $t$ is the execution time. According to the SPEC disclosure the Pentium 4 runs at 3066 MHz and takes 130 seconds to execute perlbmk and so it could execute at most 1.195740 trillion $\left(10^{12}\right)$ instructions. Similarly, the Alpha could execute $1250 \mathrm{MHz} \times 4$ inst $/$ cycle $\times 208 \mathrm{~s}=1.04$ trillion instructions and the Itanium2 could execute $900 \mathrm{MHz} \times 6 \mathrm{inst} /$ cycle $\times 251 \mathrm{~s}=1.3554$ trillion instructions. Since we are assuming they all execute the same number of instructions, the Alpha, which could execute the fewest instructions, comes closest to its potential.
(b) Are these numbers consistent with the expected tradeoffs for increasing clock frequency (mentioned in class) and for increasing the number of instructions that can be started per cycle?

Assume that the technology is fixed. (The transistors are not getting faster.) To increase the clock frequency we need to break instructions up into more steps, and that means fewer opportunities for overlap. That means there will be more times when one cannot find an instruction to execute (because the source operands that it needs are not yet ready). So with a higher clock frequency we would expect execution to be further from its maximum. This is true for the Pentium compared to the Alpha, but does not hold for the Itanium2.

A processor that can handle more instructions per cycle will have a more difficult time overlapping them, so one would expect it also to execute further from its maximum. This holds for the Itanium 2 compared to the Alpha and Pentium 4, but does not hold for the Alpha compared to the Pentium 4.

Problem Discussion:
The analysis is based on the assumption that the same number of instructions are executed on each system, a bad assumption to make. (Sorry, I don't have the numbers.) There is also an important difference between the processors' ability to overlap instructions. The Pentium 4 and Alpha 21264 are dynamically scheduled, which means they have greater freedom in overlapping instructions. (Instructions do not have to start in program order on these processors, while they do on the statically scheduled Itanium2.) Using a technique called predication, the Itanium2 can avoid branches, an advantage the others don't have. The Itanium (VLIW) ISA has several other features designed to provide performance on modern implementations, features the older Alpha (RISC) and ancient IA-32 (maybe CISC) ISAs lack.

Problem 4: Complete the lookup routine below so that it counts the number of times an integer appears in an array of 32-bit integers. Register \$a0 holds the address of the first array element, \$a1 holds the number of elements in the array, and \$a2 holds the integer to look for. The return value should be written into $\$ \mathrm{v} 0$.
lookup:

```
# Call Arguments
#
# $a0: Address of first element of array. Array holds 32-bit integers.
# $a1: Number of elements in array.
# $a2: Element to count.
#
# Return Value
#
# $v0: Number of times $a2 appears in the array starting at $a0
# [ ] Fill as many delay slots as possible.
# [ ] Avoid using too many instructions.
# [ ] Avoid obviously unnecessary instructions.
# A correct solution uses 11 instructions, including 6 in
# the loop body. A different number of instructions can be used.
# Solution Starts Here
addi $v0, $0, 0
sll $t0, $a1, 2
add $t1, $t0, $a0
beq $a0, $t1, DONE
lw $s0,0($a0)
bne $s0,$a2 LOOP
addi $a0, $a0, 4
j LOOP
addi $v0, $v0, 1
```

LOOP:
DONE:

```
# Use the two lines to return, fill the delay slot if possible.
jr $ra
nop
```

Design a stack ISA with the following characteristics:

- Memory has a 64 -bit address space and consists of 8 -bit characters.
- The stack consists of 64 -bit registers.
- The ISA uses 2's complement signed integers.
- Only add other data types as necessary.

The stack ISA must realize these goals:

- Small program size.
- Low energy consumption. (For here, assume energy consumption is proportional to dynamic instruction count.)
- Relatively simple implementation. Instructions should be no more complex than RISC instructions.

Design the instruction set based on the sample programs in the problems below and the following:

## Arithmetic and Logical Instructions

They should read their source operands from the top of stack (top one or two items) and push their result on the top of stack. Arithmetic instructions cannot read memory and they cannot read beyond the top two stack elements. (That is, you can't add an element five registers down to one ten registers down. Instead use rearrangement instructions before the add.) Specify whether the arithmetic and logical instructions pop their source operands. One can have both versions of an instruction. For example, add might pop its two source operands off the stack while addkeep might leave the two operands:

```
# Stack: 26 3 2003
add
# Stack: 29 2003
addkeep
# Stack: 2032 29 2003
```

Remember that arithmetic and logical instructions cannot rearrange the stack and cannot access memory.

## Immediates

Decide how immediates will be handled. There can be immediate versions of arithmetic instructions or one can have push immediate instructions. See the example below. Keep in mind that the register size is 64 bits.

```
# Stack: 123
addi 3 # An immediate add.
# Stack: 126
pushi 3
# Stack: 3 126
add
# Stack: 129
```


## Load and Store Instructions

Memory is read only by load instructions which push the loaded item on the stack. Memory is written only by store instructions which get the data to store from the top of the stack. Determine which addressing modes are needed for loads and stores, and design instructions with those modes.

## Stack Rearrangement Instructions

The stack rearrangement instructions change the order of items on the stack. Consider adding the following: exch, swaps the top two stack elements. roll $n j$, remove the top $j$ stack elements and insert them starting after what was the nth stack element. (See the example.) Other stack rearrangement instructions are possible.

```
# stack 11 22 33 44 55 66
exch
# stack 22 11 33 44 55 66
roll 5 2
# stack 33 44 55 22 11 66
```


## Control Transfer Instructions

Your ISA must have instructions to perform conditional branches, unconditional jumps, indirect jumps, and procedure calls. It must be possible to jump or make a procedure call to anywhere in the address space. (The only thing special the instruction used for a procedure call has to do is save a return address.) The branch instructions can (but do not have to) use a condition code register. No other registers can be used (other than those in the stack). Don't forget about the target address.

Problem 1: As specified below, describe your ISA and the design decisions used. (Don't completely solve this part until you have solved the other problems.)
(a) For each instruction used to solve the problems below or requested above, show the assembler syntax and the instruction's coding. The coding should show the opcode, immediate, and any other fields that are present. Don't forget the design goals. Also don't forget about control transfer targets.

There is no need to list a complete set of instructions, but for coding purposes assume their existence. (There must be a way of coding a complete set of instructions that realize the goals of this stack ISA.)

The solution to this part appears after the last problem.
(b) Determine the size of the stack. Specify instruction coding and implementation issues used to determine the size.

Stack size was set to 32 elements. The solution to Problem 3 used six stack elements (at most), if fewer were available additional instructions would be needed to move items from and to memory. Too many stack elements, say
thousands, would be difficult to implement and a nightmare to program (or would go unused). Additional variations of the index and roll instructions would be needed to handle thousands of registers since one byte could not reference them all. A stack size of 32 was chosen to match the number of general purpose registers in RISC ISAS.
(c) Explain your decision on whether there are immediate versions of arithmetic instructions. (The alternative is instructions like pushi.)

A design goal was to include one-byte immediate arithmetic and other instructions. To do this only a few instructions could use immediates. This was limited to only a few arithmetic and logical instructions, the others could only get operands from the stack.
(d) Explain your selection of memory addressing modes. Also, pick an addressing mode that you did not use and explain why not.

The variable instruction size made it easy to include direct addressing (instruction holds entire memory address). Displacement addressing was included because it is so commonly used but the offset was limited to one byte to save opcodes. (RISC ISAs need larger offsets since they lack a direct addressing mode and so the offset is used as the second half of an address, the first being loaded into a register with an instruction like lui.) Indirect addressing was included to keep code size down. (That is, a load. o. word 0 loads the same address as loadr. sw but it uses two bytes instead of one.)

Memory indirect, and postincrement addressing would have helped in reducing code size, they were not included since instructions could not exceed RISC-like complexity.
(e) Explain how other design decisions you have made help realize the goals of small program size, low energy, or simple implementation.

The five-bit-opcode, three-bit-operand format allowed many instructions to be coded in one byte, reducing program size. The use of a single 3 -bit immediate field for many instructions simplifies implementation.
( $f$ ) Describe any design decision you made that involved a tradeoff between code size, energy, or implementation simplicity. (Pick any pair.) The original question asked only about code size and energy. If you didn't make such a decision make one up.

The ISA described above does not involve energy and code size tradeoffs, so here's a made-up decision. In ISA $A$ every instruction has a one-byte opcode and any immediates must start in the second byte of an instruction. In ISA $B$ there is a five-bit opcode and a three-bit immediate. With the exception of a push the three-bit immediate is the only kind of immediate an instruction can use (unlike the ISA described elsewhere in this homework). The push uses as many bytes as it needs for the immediate. Suppose $20 \%$ of dynamic instructions in ISA $A$ could use the 3 -bit immediate (if it were available), $10 \%$ require a one-byte immediate, and the remainder don't use immediates. The dynamic instruction count for ISA $B$ would be $10 \%$ longer because of the added push instructions. On the other hand while those extra $10 \%$ instructions are two bytes each, the $20 \%$ of instructions that use a 3 -bit immediate are 1 byte in ISA $B$ but are two bytes in ISA $A$. Assuming the dynamic count is a reasonable predictor of the static code size, ISA $B$ has smaller code size.

Problem 2: Re-write the following MIPS code in your stack ISA.

```
    lui $a0, %hi(array) # High 16 bits of symbol array.
    ori $a0, $a0, %lo(array) # Low 16 bits of symbol array.
    jal lookup # The name of a routine.
nop
# Push handle large immediates no need to use two instructions.
push.v array
jl lookup
```

Problem 3: Re-write the solution to Homework 1 in your stack ISA, use the template below. (Use your own solution or the one posted.)
lookup:

```
    # Call Arguments (TOS is the top of the stack.)
#
    # TOS: Return address
    # TOS + 1: ADDR of first element of array. Array holds 64-bit integers.
    # TOS + 2: Number of elements in array.
    # TOS + 3: TARGET, element to count.
    #
    # Return Value
    #
    # TOS: Number of times TARGET appears in the array starting at ADDR.
    # Solution Here
    #
    # [ ] Don't forget the return.
# Solution
```

\# ra ptr size target
push. 0
\# count ra ptr size target
rolls 25
\# ptr size target count ra
rollu. 3
\# size target ptr count ra
sll. 3
index. 2
\# ptr sizex8 target ptr count ra
add
\# end target ptr count ra
rolld. 3
\# ptr end target count ra
cmpk.eq
b. 1 DONE
LOOP:

```
# ptr end target count ra
index.0
loadr.sw
# data ptr end target count ra
index.3
# target data ptr end target count ra
cmp.eq
# target=data ptr end target count ra
rolld.5
# count target=data ptr end target ra
add
# count ptr end target ra
rollu.4
```

```
# ptr end target count ra
addpower.3
cmpk.eq
b.1 LOOP
# ptr end target count ra
pop pop pop
# count ra
rollu.2
# ra count
j
```


## Comments on Solutions

Most solutions to this assignment included substantially correct programs, however several common mistakes or less-than-optimal choices were made in the stack ISAs. The following are common mistakes:

Lack of one-byte instructions. Since program size is a goal frequently used instructions should take one byte, whenever possible. Some solutions omitted any one-byte instructions, increasing code size.

Lack of immediate arithmetic instructions. A design goal was to reduce program size, including the ones in the assignment. If there are no immediate arithmetic or logical instructions then whenever an immediate is needed a push instruction must also be included, adding to program size. A justification given in some solutions for omitting immediate arithmetic instructions is a reduction in complexity or instruction count. Though these would be reduced, it would be at the expense of code size and energy, two other design goals.

Lack of one-byte immediates. Many solutions had ISAs with a single immediate size, sometimes very large. Since the goal is small program size and since many instructions can use small immediates, there should be some instructions using one-byte immediates. Other instructions could use larger immediates. (There is no reason why there should be a single immediate size.)

Lack of a 64 -bit immediate. Since the register size is 64 -bits there should be an instruction that can load a 64-bit constant, for example, pushi 0xfedcba9876543210.

Inclusion of lui-like instructions. Many solutions included instructions similar to MIPS loadupper immediate. Such instructions make sense in RISC ISAs because with their fixed instruction size there is no way to load a 32 -bit constant (or whatever the instruction size is) or larger with one instruction. An lui paired with an or or some other instruction can load a 32-bit constant. With variable size instructions one can simply have a pushi (or other instruction) that uses a 64-bit immediate, there is no need for anything like lui.

Inclusion of delayed branches. Delayed branches make sense only in certain pipelined implementations. (Such as those discussed in class so far.) On other implementations delayed branches add to complexity without adding much to performance. (This was mentioned in class several times.) For that reason, delayed branches should have been omitted or their inclusion should have been justified.

## Solution to Problem 1a

A feature of stack ISAs and a design goal in this problem is small program size. Small program size is realized by choosing instructions that minimize static instruction count and by coding instructions so they are as small as practical.

## Instruction Choice

The choice of instructions was based on those needed for the solution to the last problem in this assignment (for example, add and branch instructions). Other commonly needed instructions were added (for example, xor and store).

Though powerful instructions (for example, those that perform multiple operations such as shift and add) would help reduce the static instruction count they were not added because the problem restricted the ISA to instructions that are no more complex than typical RISC instructions.

## Data Types

The ISA uses 64 -bit signed and unsigned integers. The memory is byte-addressed and items are in big-endian byte order.

## Instruction Coding Overview

To minimize program size the coding was chosen so that as many instructions as possible were only one byte. The Problem 3 solution had several arithmetic and other instructions that used immediate operands. To squeeze them down to one byte the coding was based on a five-bit opcode and a three-bit extension field. The extension field holds an immediate or other constant data, or can be used for an extension of the opcode field (as is the function field in MIPS).

Instructions that use the extension field for anything other than an opcode (such as an immediate) are called Type 1, the rest are called Type 2. Let $i_{1}$ denote the number of Type- 1 instructions. Clearly $i_{1} \leq 32$ and the number of possible Type- 2 instructions is $8\left(32-i_{1}\right)$.

The maximum number of Type 1 instructions is small and so only those instructions which occurred frequently and needed a small (or other) immediate were given Type 1 codings. The other instructions either did not use immediates or had immediates in following bytes.

The size of the extension field was a tradeoff between the number of possible Type 1 instructions and the usefulness of the immediate. (That is, with a 1 -bit immediate there could be as many as 128 Type 1 instructions but there would be few cases where the 1-bit immediate would be useful.)

An example of a Type 1 instruction is add.i:

```
add.4 # Add 4 to the element at the top of stack.
# Coding of add.4
Field Name: | opcode | ext |
Field Value: | 0 | 100 |
Bit Number: 7 6 5 4 3 2 1 0
```

The extension field can hold data other than an immediate. The subtype of an instruction specifies what kind of data the immediate field holds. An add.i is subtype i. The table below shows the subtypes, the subtypes are explained in detail further below. In the table EX refers to the entire extension field (all 3 bits), EX10 refers to bits 1 and 0 and EX2 refers to bit 2.

```
1i Use EX as immediate. Whether it's sign extended depends on the instruction.
1v Immediate in following bytes. EX gives immediate size and padding.
1s Size and padding of data item loaded or stored from memory.
1c Comparison. EX specifies type of comparison.
1t How jump target is determined.
1b Branch condition and size of displacement.
```

10 Use EX for additional opcode bits. (Type 2 instruction.)
The assembly language syntax for Type 1 instructions consists of the mnemonic (such as add) followed by a dot and the extension field value, if known, otherwise the subtype name. For example, add. 3 means add the immediate 3 to the TOS while add.i refers to a Type 1i add instruction without specifying what the immediate is (as one does when describing the syntax).

## Instructions

A complete list of instructions appears below, starting with Type 1 instructions.
Opcode 1 push.i Push immediate on TOS.
Opcode 2 push.v IMM Push immediate on TOS.
Instruction push.i pushes value in extension field, $i$, on the stack. Instruction push.v pushes IMM on the stack. IMM is computed using the next 0 to 8 bytes based on the EXT field value as specified in the table below:

```
Type 1v -- Immediate follows first byte of instruction.
EX: Sz: Description
    0: 2: IMM is one byte, no sign extension.
    1: 3: IMM is two bytes, no sign extension.
    2: 5: IMM is four bytes, no sign extension.
    3: 9: IMM is eight bytes.
    4: 2: IMM is one byte, sign extend.
    5: 3: IMM is two bytes, sign extend.
    6: 5: IMM is four bytes, sign extend.
    7: 1: For push, use 0 as value; for others use TOS + 1.
EX is value of extension field.
Sz is total instruction size.
```

```
Execution Examples:
```

Execution Examples:
\# 111 222 333 444 555 666
\# 111 222 333 444 555 666
push.7 \# Type 1i
push.7 \# Type 1i
\# 7 111 222 333 444 555 666
\# 7 111 222 333 444 555 666
push 0x11 \# Type 1v, EX = 0 (one byte immediate, don't sign extend)
push 0x11 \# Type 1v, EX = 0 (one byte immediate, don't sign extend)
\# 0x1 7 111 222 333 444 555 666
\# 0x1 7 111 222 333 444 555 666
push -10 \# Type 1v, EX = 4 (one byte immediate, sign extend)
push -10 \# Type 1v, EX = 4 (one byte immediate, sign extend)
\# -10 0x1 7 111 222 333 444 555 666
\# -10 0x1 7 111 222 333 444 555 666
push 0x123 \# Type 1v, EX = 1 (two byte immediate, don't sign extend)
push 0x123 \# Type 1v, EX = 1 (two byte immediate, don't sign extend)
\# 0x123 -10 0x1 7 111 222 333 444 555 666
\# 0x123 -10 0x1 7 111 222 333 444 555 666
push 0x123456789 \# Type 1v, EX = 3 (Eight byte immediate.)
push 0x123456789 \# Type 1v, EX = 3 (Eight byte immediate.)
\# 0x123456789 0x123 -10 0x1 7 111 222 333 444 555 666

```
    # 0x123456789 0x123 -10 0x1 7 111 222 333 444 555 666
```

Coding Examples:
push. 7 \# Type 1i
\# Coding of instruction above.
First Byte
| opcode | ext |
| 00001 | 111 |
76543210

```
push 0x123 # Type 1v, EX = 1 (two byte immediate, don't sign extend)
# Coding of instruction above.
First Byte Second and Third Byte
| opcode | ext | | IMM |
| 00010 | 001 | | 0x123 |
    00000 000 1111100000000000
    76543 210 5432109876543210
```

Opcode 3 rollu.i Roll up by 1, width $i$.
Pop the TOS and insert it so that it becomes the $i$ 'th element.

```
Example:
# 111 222 333 444 555 666
rollu.3
# 222 333 111 444 555 666
Coding Example:
    rollu.3 # Type 1i
    # Coding of instruction above.
    First Byte
    | opcode | ext |
    | 00011 | 011 |
        76543 210
```

Opcode 4 rolld.i Roll down by 1, width $i$. Remove the i'th element and push it on the TOS.

```
Opcode 5 index.i Push a copy of element i (the (i+1)'th element).
    Example:
    # 111 222 333 444 555 666
    index.0
    # 111 111 222 333 444 555 666
    index.4
    # 444 111 111 222 333 444 555 666
Coding Example:
    index.4 # Type 1i
    # Coding of instruction above.
    First Byte
    | opcode | ext |
    | 00101 | 100 |
        76543 210
```

Opcode 0 add.i Remove the TOS add it to i push the result.
Opcode 6 add.v IMM Remove the TOS add it to IMM push the result.

See the Type 1v table above for sizes and padding of IMM.

```
Coding Examples
    add.0 # Type 1i
    # Coding of instruction above.
    First Byte
    | opcode | ext |
    | 00000 | 000 |
        76543 210
    add.v 0x12345678 # Type 1v
    # Coding of instruction above.
    First Byte Following four bytes.
    | opcode | ext | | IMM |
    | 00110 | 011 | | 0x12345678 |
        00000 000 332222222222111111111110000000000
        76543 210 10987654321098765432109876543210
```

Opcode 7 addpower.i Remove the TOS add $2^{i}$ to it and push the result.
Opcode 8 sub.i Remove the TOS subtract $i$ from it and push the result.
A sub.v is not included because it would not be used often enough to justify a Type- 1 coding.
Opcode 9 sll.i Shift left logical.
Opcode 10 srl.i Shift right logical.
Opcode 11 sra.i Shift right arithmetic.
Remove the TOS, perform the shift by $i+1$ bits and push the result. The assembly language syntax shows the shift amount while the EX field will be coded with the shift amount plus 1. For example, sll. 1 shifts left by one bit and the EX field holds a zero. There is a shift instruction for shifts beyond 9 bits.

Opcode 12 b.b DISP Branch if TOS non-zero $b=1$ or if TOS zero ( $b=0$ ).
A displacement is found in the following $2^{\text {ex10 }}$ bytes, the next instruction is the PC plus the displacement.

Examples:

```
b.1 TARGET # Branch if TOS non-zero. TARGET is 20 bytes ahead.
# Coding of instruction above.
First Byte Second Byte
    | opcode | ext | | DISP |
    | 01100 | 100 | | 10100 |
        00000 000 00000000
        76543 210 76543210
    b.0 TARGET2 # Branch if TOS zero. TARGET2 is 0x1234 bytes ahead
# Coding of instruction above.
First Byte Second and Third Byte
    | opcode | ext | | DISP |
```

| $\mid 01100$ | $\mid 101$ | $\mid 0 \times 1234$ |
| :---: | :---: | :---: |
| 00000 | 000 | 1111100000000000 |
| 76543 | 210 | 5432109876543210 |

Opcode 13 j.t DISPorTARGET Jump.
Opcode 14 jal.t DISPorTARGET Jump and link.
In both instructions the extension field specifies how to find target address, see the table below.
The jal.t instructions push a return address on the stack.

```
Type 1t -- Jump Targets
EX: As: Sz: Description
    ds: 2 : Displacement target, DISPorTARGET is one byte signed.
    ds: 3 : Displacement target, DISPorTARGET is two bytes signed.
    ds: 5 : Displacement target, DISPorTARGET is four bytes signed.
    ds: 9 : Displacement target, DISPorTARGET is eight bytes signed.
    in: 1 : Target is register indirect, address on TOS.
    ix: 1 : Target is indexed, sum of top two stack elements.
        1 : Illegal, reserved for future extension.
    di: 9 : Direct target, DISPorTARGET is eight bytes unsigned.
EX is value of extension field.
As is assembly language characters for corresponding value.
Sz is the total instruction size, including the first byte.
```

| Opcode 15 | cmp.c | Compare. |
| :--- | :--- | :--- |
| Opcode 16 | cmpk.c | Compare and keep. |
| Opcode 17 | cmpz.c | Compare with zero. |
| Opcode 18 | cmpzk.c | Compare with zero and keep. |

Instructions cmp.c and cmpk.c compare the top two elements using the comparison specified by c (see the table below). Instruction cmp.c removes the top two elements cmpk.c does not. Instructions cmpz.c and cmpkz.c are similar except that they compare the TOP element to zero. All instructions push the result of the comparison (zero or one) on the stack.

```
Type 1c -- Conditions. a is TOS, b is 0 or TOS+1
EX: As: Description
    0: eq: a = b
    1: ne: a != b
    2: lt: a<b
    3: le: a<=b
    4: gt: a>b
    5: ge: a>=b
    6: ov: overflow
    ca: carry
```

Note: All instructions are 1 byte.

Opcode 19 loadd.s IMM64 Load direct.
Opcode 20 stored.s IMM64 Store direct.

Load or store from memory using address IMM64 (the immediate found in the following 8 bytes). The size and padding of the element to load is specified by s. (See the table below.)

Opcode 21 loado.s OFF8 Load offset.
Opcode 22 storeo.s 0FF8 Store offset.
Load or store using the TOS + OFF8 as the address. The size and padding of the element to load is specified by s. (See the table below.)

Opcode 23 loadr.s Load register-indirect.
Opcode 24 storer.s Store register-indirect.
Load or store using the TOS as the address. The size and padding of the element to load is specified by s. (See the table below.)

```
Type 1s -- Memory access size and padding.
EX: As:
    ub: One byte, unsigned.
    uq: Two bytes (quarter word), unsigned.
    uh: Four bytes (half word), unsigned.
    uw: Eight bytes (word).
    sb: One byte, signed. Illegal for stores.
    sq: Two bytes, signed. Illegal for stores.
    sh: Four bytes, signed. Illegal for stores.
        : Illegal, reserved for future expansion.
Sizes: 9 bytes: loadd.s and stored.s
            2 bytes: loado.s and storeo.s
            1 byte : loadr.s and storer.s
Examples:
    # Stack: 0x1234 111 222
    index.i
    # Stack: 0x1234 0x1234 111 222
    loadr.uw
    # Stack 543210 0x1234 111 222 # 543210 is contents at memory 0x1234.
    rollu.2
    # Stack 0x1234 543210 111 222
    loado.uw 0x10
    # Stack 540000 543210 111 222 # 540000 is contents at memory 0x1244.
Coding Examples:
    loadr.uw
    # Coding of instruction above.
    First Byte
    | opcode | ext |
    | 10111 | 011 |
        00000 000
        76543 210
    loado.uw 0x10
    # Coding of instruction above.
```

| First Byte |  |  |  |
| :--- | :---: | :---: | :---: |
| \| opcode | Second Byte |  |  |
| \| ext | | \| OFF |  |  |
| 10011 | $\mid 100$ | $\mid$ | $\mid$ |
| 00000 | 000 | 00000000 |  |
| 76543 | 210 | 76543210 |  |

## Type 2 Instructions

There are 24 Type 1 instructions, leaving space for $8(32-24)=64$ Type 2 instructions.
The Type 2 instructions use the extension field as part of the opcode. Some Type 2 instructions have immediates, and some do not.

Opcode 30, Ext 0 pop Pop the stack.

```
Coding Example:
    pop
    # Coding of instruction above.
    First Byte
    | opcode | ext |
    | 11110 | 000 |
        0 0 0 0 0 0 0 0
        76543 210
```

Opcode 30, Ext 1 rolls SHIFT3 SIZE5 Roll small amount.
Opcode 30, Ext 2 roll SHIFT8 SIZE8 Roll.
Rearrange the stack. Instruction rolls is two bytes but cannot perform all rolls (on a 32element stack) whereas roll can perform any roll. A size exceeding 32 or a shift exceeding $\pm 32$ is illegal.

Note that the two immediates used by rolls fit in one byte.

```
Coding Examples:
    rolls 2 19
    # Coding of instruction above.
    First Byte Second Byte
    | opcode | ext | | SHIFT3 SIZE5 |
    | 11110 | 001 | | 010 10011 |
        00000 000 000 00000
        76543 210 765 43210
```

Opcode 30, Ext 3 index DEPTH8 Push a copy of the stack entry at DEPTH8.
Opcode 30, Ext 4 sllv Shift left logical variable.
Opcode 30, Ext 5 srlv Shift left logical variable.
Opcode 30, Ext 6 srav Shift left logical variable.
The TOS is shift by the amount specified in the low six bits of TOS +1 . TOS +1 is removed.
Opcode 30, Ext 7 shift PAD1 DIR1 AMT6 Shift. (Combined shift left, right.)
If PAD1 is 1 shift is arithmetic. If DIR1 is 1 shift is left otherwise it is right. AMT6 is the number of bits to shift. Note that the immediates fit in oe byte.

Opcode 29, Ext x sub, mul, div Arithmetic operations.

Opcode 29, Ext x and, or, xor Logical operations.
The indicated operation is performed on TOS and TOS +1 .
Opcode 31, Ext x illegal
Reserved for future second-byte opcode extension.

Problem 1: Consider the code below.

```
# Cycle 0 1
add $t1, $t2, $t3 IF ID
sub $t4, $t5, $t1
lw $t6, 4($t1)
sw O($t4), $t6
```

(a) Show a pipeline execution diagram for the code running on the following illustration. Note that the add is fetched in cycle zero.

- Take great care in determining the number of stall cycles.

```
# Solution
#
# Cycle 0
add $t1, $t2, $t3 IF ID EX ME WB
sub $t4, $t5, $t1 IF ID ----> EX ME WB
lw $t6, 4($t1) IF ----> ID EX ME WB
sw 0($t4),$t6 IF ID ----> EX ME WB
```



Problem 2: The code below is the same as in the previous problem.

```
# Cycle 0 1
add $t1, $t2, $t3 IF ID
sub $t4, $t5, $t1
lw $t6, 4($t1)
sw 0($t4), $t6
```

(a) Show a pipeline execution diagram (PED) of the code running on the system below.

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| :--- | :--- | :--- | :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| add \$t1, \$t2, \$t3 | IF | ID | EX | (ME) | (WB) |  |  |  |  |  |
| sub \$t4, \$t5, \$t1 |  | IF | ID | EX | ME | WB |  |  |  |  |
| lw $\$ \mathrm{t} 6,4(\$ \mathrm{t} 1)$ |  |  | IF | ID | EX | ME | WB |  |  |  |
| sw | $0(\$ \mathrm{t} 4), \$ \mathrm{t} 6$ |  |  |  | IF | ID | $----->$ | EX | ME | WB |

(b) In the PED circle each stage that sends a bypassed value. In the diagram label each bypass path with the cycle in which it is used. To avoid ambiguity, label the end of the path (at the mux input).

In the PED parenthesis are used instead of circles.


The problem below is tricky. If necessary use Spring 2001 Homework 2 problem 3 for practice.

Problem 3: The program below has an infinite loop and runs on the bypassed implementation below.

```
    # Initially $t0 = LOOP (address of jalr)
LOOP:
        jalr $t0
        addi $t0, $ra, -4
        bne $t0, $0 LOOP
        addi $t0, $t0, -4
```


(a) Show a pipeline execution diagram for this program up to a point at which a pattern starts repeating. Beware, the loop is tricky! Read the fine print below for hints.

```
# Code in dynamic order. (Same four static instructions repeated.)
#
# Cycle (0, 1 2 2 3 4 4 5 5 6 7 7 8 8 9 10
jalr $t0 IF ID EX(ME)WB
addi $t0, $ra, -4 IF ID EX ME WB
bne $t0, $0 LOOP IFx
addi $t0, $t0, -4
jalr $t0 IF ID -> EX(ME)WB
addi $t0, $ra, -4 IF -> ID EX ME WB
bne $t0, $0 LOOP IFx
addi $t0, $t0, -4
jalr $t0
addi $t0, $ra, -4 IF ID EX ME WB
bne $t0, $0 LOOP IF ID ----> EX ME WB
addi $t0, $t0, -4 IF ----> ID EX ME WB
# Cycle (0)1
jalr $t0 IF ID ----> EX(ME)WB
addi $t0, $ra, -4
bne $t0, $0 LOOP
addi $t0, $t0, -4
jalr $t0 IF ID -> EX(ME)WB
addi $t0, $ra, -4 IF -> ID EX ME WB
bne $t0, $0 LOOP
addi $t0, $t0, -4
# Cycle 
```

(b) In the PED circle each stage that sends a bypassed value. In the diagram label each bypass path with the cycle in which it is used. To avoid ambiguity, label the end of the path (at the mux input).
(c) Determine the CPI for a large number of iterations.

Iteration types:
First: A: Starts at cycle 0 , no other loop instruction in pipeline.
Second: B: Starts at cycle 3, pipeline contents: jalr in ME, first addi in EX.
Third: C: Starts at cycle 7 with addi, pipeline contents: jalr in ME, addi in EX.
Fourth: D: Starts at cycle 12, pipeline contents: bne in EX, second addi in ID.
Fifth: B: Starts at cycle 17, pipeline contents: jalr in ME, first addi in EX.
Because state of pipeline at the beginning of second and fifth iterations are identical and because to has the same values at those iterations, the pattern BCD will repeat. (The entire loop: ABCDBCDBCD...) The number of cycles in this three-iteration set is $17-3=14$ and the number of instructions is 7 and so the CPI is $\frac{14}{7}=2$.

Problem 4: SPARC V9 has multiple floating-point condition code (FCC) registers. See the references pages for more information on SPARC V8 and V9.
(a) Write a program that uses multiple FCC's in a way that reduces program size. As an example, the SPARC program below uses a single FCC. (To solve this problem first find instructions that set and use the multiple FCC registers in the SPARC V9 Architecture Manual. Then write a program that needs the result of one comparison (say, $a<b$ ) several times while also using the result of another (say, $c>d$ ). A program not using multiple condition code registers should have to do the comparison multiple times whereas the program you write does each comparison once.)

The solution appears below. Note that it is possible to re-cast the code so that on a system with one FCC only one of each comparison is done. The point is to demonstrate use of the registers.

```
# Solution
#
    fcmpd %fcc0, %f0, %f2
    fcmpd %fcc1, %f4, %f6
    fbg %fcc0, SKIP1
    nop
    faddd %f10, %f10, %f14
SKIP1:
    fbg %fcc1, SKIP2
    nop
    fdivd %f10, %f10, %f12
SKIP2:
    fbg %fcc0, SKIP3
    nop
    faddd %f10, %f10, %f16
SKIP3:
```

(b) SPARC V9 is the successor to SPARC V8, which has only one FCC register. (SPARC V9 implementations can run SPARC V8 code.) Did the addition of multiple FCC's require the addition of new instructions or the extension of existing instructions? Answer the question by citing the old and new instructions and details of their coding.

Yes and no.
Yes, the SPARC V9 floating-point compare instructions (fcmpd, etc) are extensions of SPARC V9 instructions. (They have the same opcodes, the only difference is that the V 9 version uses two bits of the rd field (bits 29-25) to specify the condition code register.)

No, the SPARC V9 floating-point branch instructions that can specify an FCC are different than the SPARC V8 branch instructions. (They have a different opcode.)
(c) Do you think the designers of SPARC V8 planned for multiple FCC's in a future version of the ISA?

Probably not, otherwise the V8 branch instructions would have bits reserved for a condition code register number (with instructions to set them to zero). It would take two bits away from the offset, but a 20 -bit offset can still span over a million instructions, enough for a vast majority of branches.

## ISU EE 4720

Homework 4 solution
Due: 31 March 2003

Problem 1: The two code fragments below call trap number 7. How do the respective handlers determine that trap 7 was called?

```
    ! SPARC V8
```

    ta \(\% \mathrm{gO}, 7\)
    \# MIPS
    teq \$0, \$0, 7
    In SPARC V8 each trap number has its own handler routine, the first four instructions of which are in the trap table. Since a particular handler routine is only called for a particular exception number, there is no need for the handler to determine which exception occurred. That is, if the trap 7 handler is running trap 7 must have occurred.

In MIPS the only way for the hander routine to get the trap number is to load the trap instruction itself and look at the field holding the code, bits 15:6. The handler can get the address of the instruction from the EPC register.

Problem 2: There is a difference between the software emulation of unimplemented SPARC V8 instructions triggered by an illegal opcode exception, such as faddq, and Alpha's use of PALcode for certain instructions. (See the respective ISA manuals on the references Web page. For SPARC, see Appendix G, it should not be difficult to find the PALcode information for Alpha.)
(a) What is similar about the two?

In both cases a single instruction in a program can trigger something like a subroutine that has privileged access to machine state.
(b) What is the difference between the kinds of instructions emulated using the two techniques? Why would it not make sense to use PALcode for quad-precision arithmetic instructions?

PALCode instructions are intended for functions that are too complex for a single RISC instruction and which vary from machine to machine (because the way the function is coded depends upon the underlying hardware). Among other things, PALCode instructions are used the way trap instructions are used in other ISAs, to perform system calls.

A PALCode instruction has a particular opcode, and an immediate operand specifying which PAL routine to execute. The PALCode instructions are used something like trap instructions, in which a trap code is specified in the instruction and operands are placed in fixed registers.

An illegal opcode exception can be raised by any instruction that the implementation does not recognize. If such an instruction is defined in the ISA the handler could emulate it, putting the correct result in the destination register. Illegal opcodes exceptions can be used to emulate instructions that would require alot of hardware and are expected to be rarely used in the implementation.

It would not make sense to emulate quad precision instructions with PALCode because they would not look like other arithmetic instructions and so would be awkward to use. In particular, source operands would have to be placed in fixed registers, say $f 0$ and $f 2$ and destinations would be written to another fixed register, say $f 4$. PALCode instructions are always intended for software emulation, and so in an implementation that had quad precision hardware the PALCode would be called anyway. (It could use the new instruction to do the arithmetic, but it would not be quick as just having a quad precision instruction.)

Quad precision instructions emulated using illegal instruction exceptions look like normal instructions, for example, the source operands can come from any register (perhaps the register number must be a multiple of 4). If an implementation does have quad-precision hardware, the instructions execute normally.

Problem 3: In both SPARC and MIPS each trap table entry contains the first few instructions of the respective trap handler. On some ISAs a vector table is used instead, each vector table entry holds the address of the respective handler.

Why would the use of a vector table (rather than a trap table) be difficult for the MIPS implementation below?


When an exception occurs the processor must branch to the handler routine. With a trap table the address of the handler routine can be determined by combining the trap base register (SPARC) or a fixed address (MIPS) with the exception code, this requires little or no hardware. If a vector table were used the address of the trap handler would have to be read from memory. First, the address of the vector table entry (holding the handler address) would need to be computed, that can also be done easily. Next the vector table entry must be read from memory. That would require the use of the memory stage which would complicate things because ( 1 ) the memory stage is not being used to execute an ordinary instruction (complicating control), (2) a new path must be added for sending the vector table address to the memory address input, and (3) a path must be added from the memory output port to the PC input. All of this can be done of course, but at best it might save only a few cycles from a rarely occurring event.

Problem 4: One way of implementing a vector table interrupt system on the MIPS implementation above would be by injecting hardware-generated instructions into the pipeline to initiate the handler. These instructions would be existing ISA instructions or new instructions similar to existing instructions.

What sort of instructions would be injected and how would they be generated? Show changes needed to the hardware, including the injection of instructions. In the hardware diagram the instructions can be generated by a magic cloud [tm] but the cloud must have all the inputs for information it needs.

Include a program and pipeline execution diagram to show how your scheme works.

- Assume an exception code is available in the MEM stage.
- Include a vector base register, (VBR), which holds the address of the first table entry.

The solution is on the next page.

The solution appears below. Two new registers, VBR (vector table base register) and Exc Code (exception code) are shown. The inputs to these registers are omitted for clarity. Register VBR is loaded by a system instruction (not shown or discussed further) while Exc Code is loaded by the hardware when an instruction raising an exception passes through the MEM stage. The hardware injects three instructions, lui, lw, and jsysr. Instruction jsysr is new, it jumps to the address in its operand register and switches the processor to system mode. It does not have a delay slot. The code uses register k0 which ordinary code must not use.

Normally the top input of the multiplexor is used. When an exception occurs the other inputs are used in sequence, injecting the three instructions. The immediate portion of the lui and lw instructions are inserted by the hardware, based on the contents of the VBR and Exc. Code registers.

A pipeline execution diagram is shown below. Notice that the injected instructions do not use IF.


```
# Solution, Continued
# Cycle 0
# Part of program
    ant $s0, $s1, $s2 IF ID*EX ME WB
    or $t0, $t1, $t2 IF ID EXx
    xori $t3, $t3, 1 IF IDx
    andi $s3, $s3, 7 IFx
```

    \# Injected by hardware. Assumed exception code is 1.
    lui \$k0, 0x1234
    lw \$k0, 0x5404
    ID EX ME WB
        ID EX ME WB
                            ID EX ME WB
    \# Handler
lui \$k0 \$0x9000 IF ID
sw 0(\$k0), \$r1 IF

Problem 1: [Easy] Complete pipeline execution diagrams for the following code fragments running on the fully bypassed MIPS implementations with floating point units as described below.

```
# Solution
# One ADD unit, latency 3, initiation interval 1.
add.d f0, f2, f4 IF ID A1 A2 A3 A4 WF
sub.d f6, f0, f8 IF ID -------> A1 A2 A3 A4 WF
add.d f8, f10, f12 IF -------> ID A1 A2 A3 A4 WF
# One ADD unit, latency 3, initiation interval 2.
add.d f0, f2, f4 IF ID A1 A1 A2 A2 WF
sub.d f6, f0, f8 IF ID ------> A1 A1 A2 A2 WF
add.d f8, f10, f12 IF -------> ID -> A1 A1 A2 A2 WF
# Two ADD units (A and B), latency 3, initiation interval 4.
add.d f0, f2, f4 IF ID A A A A WF
sub.d f6, f0, f8 IF ID ------> A A A A WF
add.d f8, f10, f12 IF ------>> ID B B B B WF
```

Problem 2: [Easy] Choose the latency and initiation interval for the add and multiply functional units so that the second instruction stalls to avoid a structural hazard. Show a pipeline execution diagram with this execution. (The easy way to solve it is to do the PED first, then figure out the latency and initiation interval.)

```
mul.d f0, f2, f4
```

add.d f6, f8, f10

Both functional units have an initiation interval of 1 . The multiply unit has a latency of 3 and the add unit has a latency of 2, so if it were not for the stall they would encounter a structural hazard (the two instructions trying to write their results at the same time).

```
# Solution
mul.d f0, f2, f4 IF ID M1 M2 M3 M4 WF
add.d f6, f8, f10 IF ID -> A1 A2 A3 WF
```

Problem 3: The two PEDs below show execution of MIPS code that produces wrong answers. For each explain why and show a PED of correct execution.

```
# PED showing a DESIGN FLAW. (The code runs incorrectly.)
# Cycle 0
add.s f1, f10, f11 IF ID A1 A2 A3 A4 WF
sub.d f2, f0, f4 IF ID A1 A2 A3 A4 WF
# PED showing a DESIGN FLAW. (The code runs incorrectly.)
# Cycle 0
mul.d f0, f2, f4 IF ID M1 M2 M3 M4 M5 M6 WF
sub.s f1, f10, f11 IF ID A1 A2 A3 A4 WF
In both cases the problem is due to the fact that double-precision instructions (sub.d and mul. d here) actually read and write registers in pairs. The sub. d , for example, reads f 0 and f 1 as the first operand ( 32 bits from each register), \(f 4\) and \(f 5\) as the second operand, and write the result in registers \(f 2\) and \(f 3\).
```

The first code fragment does not run correctly because the sub. d read f 1 in cycle 1 , that is before it is written by the proceeding instruction, in cycle 6 . (Note that using instructions this way is unusual, but they still must execute correctly.)

In the second code fragment the mul. d overwrites, in cycle 8 , the result written by sub. in cycle 7 .

```
# Solution. (Runs correctly assuming a very complete set of bypass paths.)
add.s f1, f10, f11 IF ID A1 A2 A3 A4 WF
sub.d f2, f0, f4 IF ID -------> A1 A2 A3 A4 WF
# Solution. (Runs correctly.)
mul.d f0, f2, f4 IF ID M1 M2 M3 M4 M5 M6 WF
sub.s f1, f10, f11 IF ID ----> A1 A2 A3 A4 WF
```

Problem 4: As directed to below, design the logic for the floating-point register file in the MIPS implementation illustrated below. The FP portion shows only part of add functional unit. Assume that is the only functional unit.

- Describe how the FP register file works. For reference, here is a description of the integer register file: The integer register file has two read ports and a write port. Each read port has a five-bit address input and a 32 -bit data output. The write port has a five-bit address input and a 32 -bit data input. Reads from zero retrieve 0 , writes to zero have no effect.
- The following signals are available: is dbl; if 1 the instruction uses double-precision operands, otherwise single-precision. FP dst: if 1 the instruction writes the floating-point register file, otherwise it does not (possibly because it's not a floating-point instruction).
- Show all connections to the FP register file. Show the number of bits or the bit range for each connection.
- The WF stage provides two signals, FPU (the value to write back) and fd, something generated in ID (as part of the solution). Additional signals can be sent down the pipeline.
- Keep In Mind: The hardware should work for both single and double operands. (That's what makes the problem interesting. If you're confused first solve it assuming only double operands, then attempt the full problem.)
- Make sure the fragments from the previous problem would run correctly.

Solution shown below. It is assumed that the functional units have 64 -bit inputs. If they perform 32 -bit operations then they operate on the high bits, bits 63:32.

The register file stores 1664 -bit numbers, each 64 -bit number is two registers, say $f 0$ and $f 1$. Notice that the address inputs use just four bits, omitting the LSB of the register number. The outputs of the register file are 64 bits, a multiplexor selects the full 64 bits if the register number is even (LSB 0) or it moves the low 32 bits to the high 32 bits if the register number is odd.

The register file uses a write-enable (WE) signal to control register writes. This was not needed in the integer register file because register zero could be used if nothing was to be written. There are actually two write enable signals, for the high 32 bits ( $63: 32$ ) and for the low bits (31:0). If a double operand is written then both write enables are asserted. If a single is written and the register is even WE high is asserted, otherwise WE low is asserted. The write enable signals are computed in ID and sent down the pipeline to be used in the WF stage.


## LSUEE 4720

Homework 6 solution
Due: 25 April 2003
Problem 1: Show the execution of the MIPS code fragment below for three iterations on a fourway dynamically machine using method 3 (physical register file) with a 256 -entry reorder buffer. Though the machine is four-way, assume that there can be any number of write-backs per cycle.

- Assume that the branch and branch target are correctly predicted in IF so that when the branch is in ID the predicted target is being fetched.
- The FP multiply functional unit is three stages (M1, M2, and M3) with an initiation interval of 1 .
- There are an unlimited number of functional units.
(a) Show the pipeline execution diagram, indicate where each instruction commits.
(b) Determine the CPI for a large number of iterations. (The method used for statically scheduled systems will work here but will be very inconvenient. There is a much easier way to determine the CPI.)

```
# Solution
    # Cycle 
LOOP:
    ldc1 f0, O(t1) IF ID Q L1 L2 WC
    mul f2, f2, f0 IF ID Q M1 M2 M3 WC
    bneq t1, t2 LOOP IF ID Q B WB C
    addi t1, t1, 8 IF ID Q EX WB C
    ldc1 f0, 0(t1) IF ID Q L1 L2 WB C
    mul f2, f2, f0 IF ID Q M1 M2 M3 WC
    bneq t1, t2 LOOP IF ID Q B WB C
    addi t1, t1, 8 IF ID Q EX WB C
    ldc1 f0, 0(t1) IF ID Q L1 L2 WB C
    mul f2, f2, f0 IF ID Q M1 M2 M3 WC
    bneq t1, t2 LOOP IF ID Q B WB C
    addi t1, t1, 8 IF ID Q EX WB C
    # Cycle 
```

The CPI is $\frac{3}{4}=0.75$. The hard way of computing the CPI is completing the pipeline execution diagram until there is a repeating pattern. With a 256 -entry reorder buffer that will take a long time. (The ROB size was not given in the original problem.) The easy way is to find the critical path. The critical path must be through loop carried dependencies, for this loop there are two, t 1 and f 2 . There is a single instruction per iteration that updates t 1 and that has a latency of zero, so the path through t 1 can execute at a rate of one iteration per cycle, which is the same as the fetch rate. The path through f 2 is also through a single instruction, the multiply, however that has a latency of 2 (takes 3 cycles to compute) and so the fastest it can execute is 3 cycles per iteration. The processor will initially fetch one iteration per cycle and the addi instruction will be able to keep up, while the mul.d will fall behind. Eventually the reorder buffer will fill, when that happens instructions will only be fetched when new space opens up, which will be when the multiply instructions commit. Therefore fetch will drop to three cycles per iteration or a CPI of $\frac{3}{4}$.

Note that the load is not on the critical path. It does provide data for the multiply and it is dependent on data from a previous iteration, t 1 , but it has its data ready before the multiply needs it. (This is only so because of the assumption that the load always hits the cache. With cache misses the situation is more complex.)

Problem 2: The execution of a MIPS program on a one-way dynamically scheduled system is shown below. The value written into the destination register is shown to the right of each instruction. Below the program are tables showing the contents of the ID Map, Commit Map, and Physical Register File (PRF) at each cycle. The tables show initial values (before the first instruction is fetched), in the PRF table the right square bracket "]" indicates that the register is free. (Otherwise the right square bracket shows when the register is freed.)
(a) Show where each instruction commits.
(b) Complete the ID and Commit Map tables.
(c) Complete the PRF table. Show the values and use a "[" to indicate when a register is removed from the free list and a "]" to indicate when it is put back in the free list. Be sure to place these in the correct cycle.

```
# Solution
```



```
# Cycle 
ID Map
llllllllll
Commit Map
llllllllllll
Physical Register File
99 112 ] [ 100]
```



```
96 309
94 058 ] [ 285 ] [ [ [ 
92 1234
91 518 ]
90 207 ]
# Cycle 
```


## 67 Fall 2002 Solutions

## EE 4720

Homework 1 solution Due: 18 September 2002
At the time this was assigned computer accounts and solution templates were not ready. If they become available they can be used for the solution, either way a paper submission is acceptable.

Problem 1: Write a MIPS assembly language program that copies and converts an array of integers to an array of doubles. Use the template below.

```
################################################################################
## cpy_w_to_dbl
    ## Register Usage
    #
    # $a0: Procedure input: Address of start of integer array (to read).
    # $a1: Procedure input: Length of integer array.
    # $a2: Procedure input: Address of start of double array (to write).
    .globl cpy_w_to_dbl
cpy_w_to_dbl:
    # Your code can modify $a0-$a2 and $t registers.
    # A correct solution uses 8 instructions (not including jr, nop),
    # a different number of instructions are okay.
    # Points will be deducted for obviously unnecessary instructions.
    #
    # Solution starts here.
sll $a1, $a1, 2
add $a1, $a1, $a0
LOOP:
```

```
lwc1 $f0, 0($a0)
```

lwc1 $f0, 0($a0)
cvt.d.w \$f0, \$f0
cvt.d.w \$f0, \$f0
sdc1 $f0, O($a2)
sdc1 $f0, O($a2)
addi \$a0, \$a0, 4
addi \$a0, \$a0, 4
bne \$a0, \$a1, LOOP
bne \$a0, \$a1, LOOP
addi \$a2, \$a2, 8
addi \$a2, \$a2, 8
jr \$ra
jr \$ra
nop

```
    nop
```

Problem 2: What do the Sun compiler -xarch and -xchip options as used below do, and what are the equivalent gcc 2.95 (GNU C compiler) switches.

```
cc myprog.c -o myprog -xarch=v8 -xchip=super
```

See http://gcc.gnu.org/onlinedocs/ for gcc and http://docs.sun.com for the Sun Forte C 6 / Sun Workshop 6 cc compiler.

Option -xarch=v8 specifies that the compiler should emit v8 instructions. The closest equivalent gee switeh is $-\mathrm{mcpu}=\mathrm{v} 8$. The -xchip=super option tells the compiler to select and arrange instructions for a superspare chip. The equivalent gcc option is -mtune=supersparc.

Problem 3: In Sun's CINT2000 SPEC Benchmark disclosure for the Sun Blade 1000 Model 900 Cu they specify a -xregs=syst compiler flag for several of the benchmarks compiled under the peak rules. Hint: Use a search engine to find this rare flag. Guess what many of the search hits are to?
(a) What does this flag do?

It tells the compiler that it can use registers normally reserved for the system.
(b) How does it improve performance? Hint: It affects one of the few low-level optimizations covered in class up to this point.

It eases the register assignment problem. That is, because more registers are available for compiler use the compiler can leave more values in registers rather than writing and reading them from memory.
(c) How often could this option be used in the real world?

The Sun documentation explains that the option cannot be used when system libraries are used, which almost every program uses. So the option could not be used very often.

Problem 4: Benchmark suites are suites because a single program might run well on a processor that runs most other code poorly.

At http://www.spec.org find the fastest processors using the "result" numbers from the SPEC CINT2000 benchmarks in the following categories: The fastest two Pentium 4s, the fastest Athlon, and the fastest Alpha. (Figure out how to get a result-sorted list of machines that shows processor type.)

The solutions below are based on: Intel D850EMV2 motherboard ( 2.8 GHz , Pentium 4 processor) (http://www.spec.org/osg/cpu2000/results/res2002q3/cpu2000-20020827-01581.html), Intel D850EMV2 motherboard ( 2.67 GHz , Pentium 4 processor) (http://www.spec.org/osg/cpu2000/results/res2002q3/cpu2000-20020827-01583.html), HP AlphaServer ES45 68/1250
(http://www.spec.org/osg/cpu2000/results/res2002q3/cpu2000-20020801-01512.html), and the Epox 8KHA+ Motherboard, AMD Athlon (TM) XP 2600+
(http://www.spec.org/osg/cpu2000/results/res2002q3/cpu2000-20020812-01551.html). (Note, the links won't work forever.)
(a) What programs might an unfair Intel advocate want removed from the suite?

The Alpha outperforms the Pentium on the vpr and mof benchmarks, so the unfair Intel advocate would want those removed. The Athlon outperforms the Pentium on crafty and eon, so the unfair person would want those removed too.

For the parts below consider the relative performance of the programs in the suite. (Put the bar graphs for two different systems side by side and note the difference in shape.)
(b) Why might one expect the top two Pentia to be very similar? Are they in fact very similar?

It appears from the name that they only differ in clock frequency. (This could be verified from the Processors' datasheets.) The internal design of the two processors are the same and so they would share the same strengths and weaknesses running programs. However, though the clock frequencies of the processor cores are different, other parts of the system, for example, the memory bus, are the same and so the performance of the faster chip will not be $\frac{2.8}{2.67}$ faster on every benchmark.
(c) Why might one expect the Athlon to be more similar to the Pentium than to the Alpha? Does it?

The Pentium and Athlon have similar ISAs, while the Pentium and Alpha have very different ISAs. If the instruction set was the only thing that determined performance then the Pentium and Athlon would be identical. If the instruction set had no impact on performance the Pentium would differ from the Athlon as much as it does from Alpha. The reality is between the two, the instruction set has some (perhaps relatively small) impact on performance.

The Athlon does appear more similar to the Pentium than the Alpha.

ISA manuals are needed for some problems below. Links to the ISA manuals can be found on the new references Web page: http://www.ece.lsu.edu/ee4720/reference.html

Problem 1: Consider the following SPARC instructions:

```
sub %g3, %g2, %g1 ! g1 = g3 - g2;
and %g1, 0xf, %g1 ! g1 = g1 & 0xf
```

Wouldn't it be nice to have a sub. and instruction that would do both:
sub. and $\% \mathrm{~g} 3, \% \mathrm{~g} 2,0 \mathrm{xf}, \% \mathrm{~g} 1 \quad$ ! $\mathrm{g} 1=(\mathrm{g} 3-\mathrm{g} 2) \& 0 x f$
(a) Could the SPARC V9 ISA easily be extended to support such double-op instructions? If yes, explain how they would be coded.

Discussion: In the problem "easily be extended" means extending the ISA without adding entirely new instruction formats. The instructions above take two register source operands and an immediate source operand. Ordinary two-source register, one destination register instructions are coded using Format 3 with the $i$ (immediate) bit set to zero. That format has an unused field, asi, which can be used for the immediate in the double-op instructions. The asi field is only eight bits, but that's enough for the immediates in the examples above.

Solution: Yes. Code the double-op instructions using SPARC Format $3(i=0)$ and placing the immediate value in the asi field. (b) Estimate how useful double-op instructions would be, using the data below. Usefulness here is conveniently defined as the dynamic instruction count. Consider a large class of double-op instructions that operate on two source registers and an immediate. For example, add.add, sll.add, and and.or. The data below does not provide important statistics needed to estimate the usefulness. Describe what statistics are needed and make up numbers. The made up numbers can be totally arbitrary (as long as they are possible).

The data below show instruction category and immediate sizes running the gcc compiler (cc1). Assume that this is a representative program and so the results apply to others. The data show the total number of instructions, and the breakdown by category, including ALU instructions that use an immediate, ALU instructions that use two source registers, etc. Following that histograms of the immediate sizes are shown for four instruction categories. This is very similar to the data shown in class. The percentage at each size and a cumulative percentage are shown. For example, $11.12 \%$ of ALU immediate instructions use two bits and $55.40 \%$ use two bits or fewer.

Quantifying usefulness means determining how many pairs of dynamic instructions can be combined into a double-op instruction. They can be combined if the first of the pair writes a register that is only used by the second of the pair (otherwise the first instruction could not be eliminated) and if the immediate will fit in the asi field (see the solution to the previous part).

The data below can be used to determine how many immediates would fit in the asi field, which is eight bits. The "ALU Immediate Size Distribution" table indicates that $95.13 \%$ of ALU instruction immediates are eight bits or less, which is good for the double-op instructions.

Using the data below one can only get a rough estimate of how many combinable pairs there are. One of each pair will be a two-source register ALU instruction, which represents $15.3 \%$ of all instructions. Assuming that each of these can be one of a pair (a bad assumption, but perhaps the best we can do with the data other than guessing) and assuming that $95.13 \%$ of the immediate instructions have small enough immediates yields $14.6 \%$ of the dynamic instructions. Assuming instruction count is proportional to execution time, the double-op instructions will reduce execution time from 1.0 to 854.
[drop] \% echo /opt/local/lib/gcc-lib/sparc-sun-solaris2.6/2.95.2/cc1 \}
els.i -03 -quiet isize
Analyzed 156423240 instructions:
48483403 ( $31.0 \%$ ) ALU Immediate
23886739 ( $15.3 \%$ ) ALU Two Source Register
6353567 ( $4.1 \%$ ) sethi
34039161 ( $21.8 \%$ ) Loads and Stores
30331049 ( 19.4\%) Branches
13329321 ( 8.5\%) Other

```
ALU Immediate Size Distribution
    Bits Pct Cum
        0 25.96% 25.96% ********************
        1 18.32% 44.28% **************
        2 11.12% 55.40% ********
        3 15.59% 70.99% ************
        4 3.16% 74.15% ***
        5 6.10% 80.25% *****
        6 7.31% 87.56% ******
        7 6.81% 94.37% *****
        8 0.76% 95.13% *
        9.13% 97.26% **
        10 2.64% 99.90% **
        11 0.01% 99.91% *
        12 0.08% 99.99% *
        13 0.01% 100.00% *
SETHI Immediate Size Distribution
    Bits Pct Cum
        0 0.00% 0.00% *
        1 0.00% 0.00% *
        2 0.02% 0.02% *
        3 0.72% 0.74% *
        4 0.43% 1.17% *
        5 0.09% 1.26% *
        6 0.66% 1.93% *
        7 2.00% 3.93% **
        8 4.53% 8.45% ****
        9 3.14% 11.60% ***
        10 5.86% 17.46% *****
        11 5.54% 23.00% ****
        2 72.67% 95.67% ******************************************************
        13 0.09% 95.76% *
        14 0.13% 95.89% *
        15 0.10% 95.99% *
        16 0.01% 96.00% *
        17 0.00% 96.01% *
        18 0.49% 96.50% *
        19 3.14% 99.63% ***
        20 0.02% 99.66% *
        21 0.33% 99.99% *
        22 0.01% 100.00% *
Memory Offset Distribution
    Bits Pct Cum
        0 4.93% 4.93% ****
        0.11% 5.04% *
        2.51% 7.55% **
        15.95% 23.50% ************
        24.41% 47.91% *******************
        10.36% 58.27% ********
        4.90% 63.17% ****
```

| 7 | 6.89\% | 70.06\% | ***** |
| :---: | :---: | :---: | :---: |
| 8 | 5.79\% | 75.85\% | ***** |
| 9 | 4.98\% | 80.82\% | **** |
| 10 | 16.09\% | 96.92\% | ************ |
| 11 | 2.38\% | 99.30\% | ** |
| 12 | 0.70\% | 100.00\% | * |
| 13 | 0.00\% | 100.00\% |  |
| Branch | Displac | ement Di | istribution |
| Bits | Pct | Cum |  |
| 0 | 0.00\% | 0.00\% |  |
| 1 | 0.00\% | 0.00\% | * |
| 2 | 13.06\% | 13.06\% | ********** |
| 3 | 22.20\% | 35.26\% | **************** |
| 4 | 16.58\% | 51.84\% | ************ |
| 5 | 18.94\% | 70.79\% | ************** |
| 6 | 15.69\% | 86.48\% | *********** |
| 7 | 6.74\% | 93.22\% | ***** |
| 8 | 3.47\% | 96.69\% | *** |
| 9 | 1.63\% | 98.31\% | ** |
| 10 | 0.71\% | 99.02\% | * |
| 11 | 0.27\% | 99.29\% | * |
| 12 | 0.41\% | 99.69\% | * |
| 13 | 0.01\% | 99.71\% | * |
| 14 | 0.00\% | 99.71\% |  |
| 15 | 0.00\% | 99.71\% | * |
| 16 | 0.21\% | 99.91\% | * |
| 17 | 0.01\% | 99.92\% | * |
| 18 | 0.08\% | 100.00\% | * |
| 19 | 0.00\% | 100.00\% |  |
| 20 | 0.00\% | 100.00\% |  |
| 21 | 0.00\% | 100.00\% |  |
| 22 | 0.00\% | 100.00\% |  |

Problem 2: It's time to go instruction hunting!
(a) The Alpha does not have a general set of double-op instructions but it does have one that can replace the two SPARC V9 instructions below. What is it? Replace the two instructions below with the Alpha instruction. (For full credit [another 0.5 point, maybe] take into account that SPARC V9 and not SPARC V8 was specified.)

```
sll %g2, 2, %g1 ! g1 = g2 << 2;
add %g3, %g1, %g1 ! g1 = g3 + g1
S4ADDQ r1, r2, r3
```

(b) SPARC V9 does not have a full set of predicated instructions, but it does have a predicated instruction that can replace the code fragment below. What is it?

```
subcc %g1, 0, %g0 ! Set integer condition codes.
be SKIP ! Branch if result equal to zero.
nop
add %g3, 0, %g4 ! g4 = g3 + 0
SKIP:
movrnz %g1, %g3, %g4
```

Problem 3: Complete Spring 2002 Homework 2 Problems 2 and 3.
(At http://www.ece.lsu.edu/ee4720/2002/hw02.pdf.) (The Verilog part is optional.) This is a very important type of problem, similar problems will be appearing all semester. You must solve the problem, that is, scratch your head, figure it out, and work it through. If you're stuck, feel free to ask for help. When you're done look at a solution and assign yourself a grade. Grade on a scale of 0 to 1 (real, not integer!)

Not solving it or solving it with too many glances at the solution will leave you ill-prepared for the test. Yes, you can solve it the night before the test (if you have time), but that won't help you understand everything presented in class between now and then. You have been warned.

Problem 1: Consider the solution to Spring 2002 Homework 4, shown on the next page. (The solution was updated 19 November 2002, the PED is shown in dynamic order instead of the nearly-impossible-to-read static order.)
(a) Show the contents of the reorder buffer in cycle 12. For each entry show the values of the fields from the illustration below, for the PC show the instruction (ldc1, mul.d, etc.). (The fields are ST, dst, dstPR, and incumb.) If a field value cannot be determined from the solution leave it blank, that will include fields related to registers $\$ 2$ and $\$ 3$.

Note: A solution not showing instructions 1-4 would also be correct.

| Solution "PC" | ST | dst | dstPR | incumb |
| :---: | :---: | :---: | :---: | :---: |
| 1 sdc1 $0(\$ 1), \mathrm{fo}$ |  |  |  |  |
| 2 addi \$1, \$1, 8 | C | \$1 | 95 | 98 |
| 3 bne \$2, \$0 LOOP | C |  |  |  |
| 4 sub \$2, \$1, \$3 | C |  |  |  |
| 5 ldc1 f0, $0(\$ 1)$ | C | f0 | 94 | 96 |
| 6 mul.d f0, f0, f2 |  | f0 | 93 | 94 |
| 7 sdc1 0(\$1), f0 |  |  |  |  |
| 8 addi \$1, \$1, 8 | C | \$1 | 92 | 95 |
| 9 bne \$2, \$0 LOOP | C |  |  |  |
| 10 sub \$2, \$1, \$3 | C |  |  |  |
| $11 \mathrm{ldc1} \mathrm{f0} ,0(\$ 1)$ | C | f0 | 91 | 93 |
| 12 mul.d f0, f0, f2 |  | f0 | 90 | 91 |
| 13 sdc1 0(\$1), f0 |  |  |  |  |
| 14 addi \$1, \$1, 8 | C | \$1 | 89 | 92 |
| 15 bne \$2, \$0 LOOP |  |  |  |  |
| 16 sub \$2, \$1, \$3 | C |  |  |  |
| 17 ldc1 f0, 0(\$1) |  | f0 |  |  |
| 18 mul.d f0, f0, f2 |  | f0 |  |  |

(b) For the solution to the part above, number each instruction. (1, 2, 3, etc.) Show the contents of the instruction queue at cycle 12 identifying each instruction by these numbers.

The instruction queue holds instructions waiting to execute, at cycle 12 only the 18th instruction above, multiply is waiting.

## Solution

18 mul.d f0, f0, f2 f0
(c) On the illustration there are three wires labeled with big lower-case letters, $\mathrm{a}, \mathrm{b}$, and c , and corresponding rows in a table in the middle of the next page. Based on the solution to last semester's problem, show what values are on those wires in each cycle that they are used. Omit cycles where a value cannot be determined. Note that the illustration is for a one-way (non-superscalar) processor but the program runs on a four-way system. That means each wire can hold up to four values in one cycle. Hint: The solution for at least one of the letters already appears. Just label the row(s) in the appropriate table with the letter. At least one of the letters does not appear, so that will have to be written in.

Row $b$ is the same as the commit map (with the two commit map rows merged into one.)



To answer the questions below you need to use the PSE dataset viewer program. PSE (pronounced see) runs on Solaris and Linux; you can use the computer accounts distributed in class to run it, a Linux distribution may also be provided for running it on other systems.

Procedures for setting up the class account and using PSE are at http://www.ece.lsu.edu/ee4720/proc.html; preliminary documentation for PSE is at http://www.ece.lsu.edu/ee4720/pse.pdf.

Problem 1: Near the beginning of the semester the performance of a program to compute $\pi$ was evaluated with and without optimization. It's back, down below.

Follow instructions referred to above to view the execution of the optimized and unoptimized versions of the pi program running on a simulated 4-way dynamically scheduled superscalar machine with a 48 -instruction reorder buffer. The datasets to use are pi_opt.ds and pi_noopt.ds.
(a) Based on the pipeline execution diagram compute the CPI of the main loop for a large number of iterations in the optimized version. Do not use the IPC displayed by PSE, instead base it on the PED. In your answer describe how the CPI was determined.

To find the precise CPI first find a repeating pattern. Fortunately, once the branch predictor warms up and the ROB fills each iteration is identical so a unit of the repeating pattern is one iteration long. One such iteration (not the first) starts at cycle (time) 339, the next starts at 345, for a time of 6 cycles. There are 9 instructions (including the nop), so the CPI is $\frac{6}{9}=\frac{2}{3}$.
(b) Consider first the optimized version of the program. Would it run faster with a larger reorder buffer? Would it run faster on an 8 -way superscalar machine? How else might the processor be modified to improve performance? Explain each answer.

An important feature to notice is that, except for nop, instructions wait many cycles before executing. All of the waiting instructions are waiting for operands and so execution time is limited by the critical path through the code. (No instruction in the loop waits for a functional unit, there are enough for this loop.)

Rank: 4/7 Pos. 1/7
0.76 IPC over 38 cycles.

State: Execute

First Instruction:.LLM7 main+11 pi.c:11
fdivd \%f12, \%f6, \%f2


Time 266 Tag 234 PC 0x00010804
Grid 20 insn X 5 cyc

With a larger ROB there will be more instructions in flight, but all these "new" instructions will do is wait. Similarly, an 8-way machine will fetch instructions twice as fast and provide additional functional units, but that won't change the critical path. All an 8 -way processor will do for this loop is fill the ROB more quickly.

The problem can be more precisely solved by constructing a dataflow graph (DFG) and inding the critical path. The dataflow graph for this loop appears below:
$\sqrt{6}$ Cycles on critical path for one iteration.


The loop critical path is two floating point adds, taking three cycles each. Actually two paths are tied for the critical path award: one path is $0 \times 10810 \rightarrow 0 \times 10824$ and the other is $0 \times 10808 \rightarrow 0 \times 10814$. Since the add instructions wait only for operands, performance can be improved if the floating-point adder takes fewer cycles (say, two), based on what was covered in the class. A more exotic solution would be to have a floating-point functional unit that can perform three-source-operand instructions and a processor that could recognize pairs of instructions that could be replaced by three-source-operand instructions. (The question asked about processor modifications, not compiler or ISA modifications, so one could not just re-compile the pi program for a three-source ISA.) Real processors don't do this yet, but research is being conducted in the area.
(c) Now consider the un-optimized version. Would it run faster with a larger reorder buffer? Should a computer designer pay attention to the performance of un-optimized code? Explain each answer.

Rank: 4/14 Pos. 1/14
First Instruction:.LLM5+13 main+34 pi.c:11 0.82 IPC over 49 cycles. State: Cache Done std \%f2, [ \%fp +-24]


Time 583 Tag 641 PC 0x00010860
Grid 20 insn X 5 cyc

As can be seen from the PED, most instructions in the loop do not wait for dependencies, that is, they execute as soon as they are decoded, renamed, and scheduled. Some of these instructions had to wait for space in the reorder buffer, and so if the reorder buffer were larger they would be executed sooner. If some of these previously-waiting-but-now-executing-sooner-than-before instructions are on the critical path then execution speed would improve. (In the optimized case the nop would be executed earlier with a larger ROB but, of course, it's not on the critical path.) One way to find the critical path is to draw and analyze a dataflow graph, but an easier way is by inspection of the PED: instructions that complete just before they commit may be on the critical path. In this case the faddd at $0 \times 10844$; preceded by a fdivd, a ldd at $0 x 838$ which is dependent on the std at $0 x 898$. That store has its data ready two cycles before the load is ready to load it; the ROB was full between the store and the load, delaying the execution of the load and so increasing the ROB size will shorten this critical path.

A complete analysis shows again that there are two critical paths, this time eight cycles. (Two adds on one, an add and a sub on the other; both paths have two loads and a store. A store/load bypass takes one cycle, for a total of eight cycles.)

Since the critical paths are eight cycles and the loop spans 37 instructions the maximum IPC is 4.625 . Therefore, increasing the decode width from 4 - to 8 -way superscalar will improve performance if the ROB has also been enlarged. (Because of the difficulty of the analysis the problem did not ask if there would be performance gain in an 8 -way system.)

Computer engineers should not pay attention to unoptimized code. Most programs that are run are optimized code. (Homework assignments in beginning computer classes are an exception.) Improving the performance of un-optimized code would only help programmers debugging programs (since when debugging one usually doesn't optimize) and yes, students working on homework. If engineers' time and silicon area are limited, its better to use them to speed the kind of code most people run.
(d) The simulated processors use a gshare branch predictor. Use pi_opt.ds to answer this question. How many bits is the global history register? Entries in the PHT are initialized to 1 and the GHR is initialized to zero. The PHT is updated when the branches resolve (in the cycle after they execute). Explain your answer.


The global history register is four bits. The easiest way to find the answer is to examine execution at the start of the program, when the state of the GHR and PHT are known. Luckily, the program starts with five consecutive branch mispredictions, all for the same static instruction, $0 \times 10820$ fbl (floating-point branch less than). Each time this branch is mispredicted not taken but is in fact taken. The PHT entries are initialized to 1 and so a particular PHT entry can only contribute to one misprediction. Therefore five PHT entries are being used. Since only one static branch is executed through these five mispredictions, the GHR contents must be four bits. Here is the GHR contents used to predict the branch the first six times: $0_{2}, 1_{2}, 11_{2}, 111_{2}, 1111_{2}, 1111_{2} \ldots$. The fifth and sixth contents are identical, and so the branch is correctly predicted the sixth time.
(e) Would execution be any different if the PHT were updated when the instructions commit? Explain.

Yes. Execution is different because the PHT table is being updated after the prediction is made. For example, consider the branch that commits in cycle 253, with tag 197. The PHT would be updated at the end of cycle 253, however by that time the branch has already been predicted for the next iteration (tag 241). Had the PHT been updated at commit time the branch with tag 241 would be predicted using the old PHT value and would have been predicted not taken. Note that branches are predicted before decode and the one-cycle delay in fetching the branch target (at cycle 254) is due to another misprediction by the instruction fetch mechanism, something not covered in the class.

```
#include <stdio.h>
int
main(int argv, char **argc)
{
    double i;
    double sum = 0; // Line 7
    for(i=1; i<5000;) // Line 9
        {
            sum = sum + 4.0 / i; i += 2; // Line 11
            sum = sum - 4.0 / i; i += 2; // Line 12
        }
    printf("After %d iterations sum = %.8f\n", (int)(i-1)/2, sum); // Line 15
    return 0;
}
```

To answer the questions below you need to use the PSE dataset viewer program. PSE (pronounced see) runs on Solaris and Linux; you can use the computer accounts distributed in class to run it, a Linux distribution may also be provided for running it on other systems.

Procedures for setting up the class account and using PSE are at http://www.ece.lsu.edu/ee4720/proc.html; preliminary documentation for PSE is at http://www.ece.lsu.edu/ee4720/pse.pdf.

Problem 1: The code in http://www.ece.lsu.edu/ee4720/2002f/hw6.pdf includes two routines to perform a linear search, lookup_array and lookup_ll. Routine lookup_array (aws,foo) searches aws for element foo. The list itself is an ordinary C array, structure aws (array with size) includes the array and its size. Routine lookup_ll(head,foo) searches for foo in the linked list starting at head.

The code calls the search routines under realistic conditions: Before the linked list is allocated dynamic storage is fragmented and before the searches are performed the level- 1 cache is flushed. See the code for more details.

The code was executed on a simulated 4 -way superscalar dynamically scheduled machine with a 64 -entry reorder buffer and a two-level cache. The simulation was recorded in hw6.ds; view this dataset file using PSE to answer the questions below.

The code initializes the lists with identical data and then calls the search routines looking for the same value. Answer the following questions about the execution of the two lookup routines. When browsing the dataset be aware that the time spent in the lookup routines is dwarfed by the time needed for setting everything up and so only the last few segments need to be examined.
(a) Would increasing the ROB size improve the performance of the linked list routine, lookup_11? Explain.

No. The speed of the linked list routine depends upon how often the load hits the level-one cache. In the first PED below there are many misses, in the second there are many hits.

Without resorting to a dataflow graph one can conclude that a larger ROB won't help by noting that in the first case the only instruction that executes early (execution is shown in yellow), inc, does not provide a value needed by the instructions blocking the head of the reorder buffer. So, with a larger ROB inc would be fetched earlier but that would do nothing for the instructions blocking the head.

In the second case the ROB does not fill, and so a larger ROB will make no difference.
Using a dataflow graph (or just eyeballing the seven instructions) one finds that the critical path for an iteration of this loop is one instruction long, $0 x 1083 \mathrm{c} 1 \mathrm{ld}[00+4,00]$. (If the critical path were measured in instructions rather than cycles then inc would also be a critical path, but it only takes one cycle. The other instructions are not on the critical path because they do not produce a value that is needed in the next cycle. Note that if branches weren't predicted they would be on the critical path.) If the load hits the level 1 cache then the critical path is just two cycles long (one cycle for the address, here address computation is on the critical path unlike the unoptimized $\pi$ routine from Homework 5), if the load misses it is much longer, 23 cycles.

With a critical path of 23 cycles per iteration and seven instructions per iteration, the processor is already executing the loop as fast as it can. The ROB just fills with mostly waiting instructions. A larger ROB won't help.

When there are lots of hits, near the end of the loop, the critical path is just two cycles per iteration. As noted earlier, the ROB is less than half full, so increasing its size won't help. This wasn't asked, but since we're here we might as will determine if this code is executing as fast as it can. The ideal CPI for this code is $\frac{2}{7} \approx 0.286$, which is attainable on a 4 -way superscalar processor. The code is actually executing at $\frac{3}{7} \approx 0.429$, the problem is fetch inefficiency. That is, because the three instructions starting at address $0 \times 10848$ lie on two aligned groups it takes three cycles to fetch the 7-cycle loop.

Rank: 1/122 Pos. 120/122 0.14 IPC over 219 cycles.

State: L1 Miss

First Instruction:.LLM10 lookup_Il+3 hw6.c:74 ld [ \%o0 ], \%g2


Time 238,182 Tag 425,274 PC 0x00010830

Rank: 46/122 Pos. 121/122

(b) Would increasing the ROB size improve the performance of the array routine, lookup_array? Explain.

Yes, because the instructions allowed in with a larger reorder buffer would be able to execute. As can be seen from the PED below, there are many instructions that execute (shown in yellow) right after the pre-ready state. When the ROB fills there are instructions that could execute but aren't being fetched because of the full ROB, and so a larger ROB would help these. Unlike the linked list routine, the load that misses the cache is not on the critical path, so a load miss does not leave the processor with little to do.

Rank: 116/122 Pos. 119/122 2.17 IPC over 96 cycles.

State: Commit


Time 236,216 Tag 421,656 PC 0x00010800
(c) As can be seen viewing the PED plots (for example, the one below), the array routine follows a regular pattern while the linked list code starts off slowly but as it nears completion it runs much faster. Why does the linked list code speed up like that?


A line can hold multiple linked list elements. An L1 miss brings in the linked list element which is currently needed but also brings in others that will be needed later. As the search proceeds more and more of the elements will be found in the cache.
(d) How could one determine the line size from the PED plots? Be specific and use numbers from the dataset. (The line size can be found two other ways, if you come upon them by all means use them to check your answer that is based on the PED plot.)

The array routine accesses memory consecutively and each element is four bytes. It is accessed by the load instruction at $0 \times 107 \mathrm{f} 8$. Looking at the PED above it is easy to find load misses. Since the array is not in the level-one cache when the routine starts every access to a new line must miss. To find the line size count the number of loads between misses. There are 16, and so the line size must be 64 bytes.
(e) Before people stopped replacing $\$ 2,500$ computers every six months computer engineers would loose sleep worrying about The Memory Wall, the growing gap in performance between processors and memory (e.g., the number of instructions that could have been executed while waiting for memory). What is it about the array routine that lets it sail over the memory wall while the linked list routine is stopped dead? The answer should take into account certain load instructions and the critical path. Discuss how the performance of the routines change as the L1 miss time gets longer and longer.

The reason the linked list suffers is because the load for the second element can't start until the load for the first element finishes. In contrast, since an array is laid out sequentially in memory, there is no need to load the first element to find out where the second is located. The array code enjoys two advantages here. First, a miss to a single line will bring several array elements, second, it is possible to overlap misses to two lines (if the ROB is large enough, which it isn't). Therefore, even if the gap between memory and processor speed continues to widen, the performance of the array code won't suffer much as long as the ROB is made larger and the memory can handle multiple overlapping misses.
(Linked lists do have their advantages, but should not be used where an array would suffice.)

68 Spring 2002 Solutions

At the time this was assigned computer accounts and solution templates were not available. If they become available they can be used for the solution, either way a paper submission is acceptable.

Problem 1: The value computed by the program below approaches $\pi$. Re-write the program in MIPS assembler. The code should execute quickly. Assume that all integer instructions take one cycle, floating-point divides take ten cycles, floating-point compares take one cycle, and all other floating-point instructions, including conversion, take four cycles. Note: As originally assigned only the time for divides and adds was given. Make changes to the code to improve speed (possibly using an integer for i or even using both an integer and double). Do not use a different technique for computing $\pi$.

```
int
main(int argv, char **argc)
{
    double i;
    double sum = 0;
    for(i=1; i<50000000;)
        {
            sum = sum + 4.0 / i; i += 2;
            sum = sum - 4.0 / i; i += 2;
        }
    printf("After %d iterations sum = %.8f\n", (int)(i-1)/2, sum);
    return 0;
}
```

The code appears on the next page. But first, here are some reminders based on submitted solutions:
Double-precision values must be placed in an even-numbered ip register.
When speed is a goal delay slots should be filled!
Immediates are limited to 16 bits.

```
# Solution to problem 1.
    .data
ITERATIONS:
    .double 50000000.0
MSG:
    .asciiz "After %/f0/.Of iterations sum = %/f2/.8f\n";
    .text
    .globl __start
__start:
    addi $t1, $0, 1
    mtc1 $t1, $f0
    cvt.d.w $f0, $f0 # f0 -> i
    add.d $f12, $f0, $f0 # f12 <- 2 (constant)
    add.d $f14, $f12, $f12 # f14 <- 4 (constant)
    la $t2, ITERATIONS
    ldc1 $f16, 0($t2) # f16 -> number of iterations.
    sub.d $f2, $f0, $f0 # f2 -> sum <- 0 initialize
LOOP:
    div.d $f4, $f14, $f0 # 4.0 / i
    add.d $f0, $f0, $f12 # i+=2
    add.d $f2, $f2, $f4 # sum += 4.0/i
    div.d $f4, $f14, $f0 # 4.0 / i
    add.d $f0, $f0, $f12 # i+=2
    c.lt.d $f0, $f16
    bc1t LOOP
    sub.d $f2, $f2, $f4 # sum -= 4.0/i
    div.d $f16, $f16, $f12
    addi $v0, $0, 11
    la $a0, MSG
    syscall
    addi $v0, $0, 10
    syscall
```

Problem 2: The program below is used to generate a password based on the outcome of several rolls of a twenty-sided die. The program was compiled using the Sun Workshop Compiler 5.0 targeting SPARC V7 (-xarch=v7) and SPARC V9 (-xarch=v8plus, code which can run on a V9 processor with a 32 -bit OS), the output of the compiler is shown for the for loop.

Use the V8 architecture manual to look up V7 instructions, available at https://www.ece.lsu.edu/ee4720/samv8.pdf; the V9 architecture manual is available at https://www.ece.1su.edu/ee4720/samv9.pdf.

Here are a few useful facts about SPARC:
Register names for SPARC are: \%g0-\%g7 (global), \% $10-\% 17$ (local), $\% \mathrm{i} 0-\% \mathrm{i} 7$ (input), \% $00-\% \mathrm{o} 7$ (output), and $\% \mathrm{f0} 0$ \% 31 (floating point). Registers $\% \mathrm{fp}$ (frame pointer) and $\% \mathrm{sp}$ are aliases for $\% \mathrm{i} 6$ and $\% 06$, respectively. Register $\% \mathrm{~g} 0$ is a zero register.

Local variables (the only kind used in the code fragment shown) are stored in memory at some offset from the stack pointer (in \%sp). For example, ldd $[\% s p+96]$, \% f 0 loads a local variable into register \% f 0 .

All V7 and V8 integer registers are 32 bits. V9 registers are 64 bits but with the v8plus option only the 32 lower bits are used.

Unlike MIPS and DLX, the last register in an assembly language instruction is the destination. For example, add $\% \mathrm{~g} 1, \% \mathrm{~g} 2, \% \mathrm{~g} 3$, puts the sum of g 1 and g 2 in register g 3 .

Like MIPS, SPARC branches are delayed. Unlike MIPS, some delayed branches are annulled, indicated with a ",a" in the mnemonic. In an annulled branch the instruction in the delay slot is executed if and only if the branch is taken.
(a) For each compilation, identify which registers are used for which program variables.

See comments in the code on the following pages.
(b) For each instruction used in the V9 version of the code but not in the V7 version, explain what it does and how it improves execution over the V7 version.
udivx: performs 64-bit unsigned integer division. It's probably faster than the divide routine called in the V7 code. mulx: performs 64 -bit multiplications, used for finding the remainder. It's probably faster than the remainder routine called in the V7 code.

Because the v9 uses divx and mulx it makes no procedure calls in the loop and so there is no need to save and restore the floating-point registers.
fbul, a,pt and fbge, a,pt: branches with prediction hints. Can speed execution if predictions correct and needed by hardware.

```
! SOLUTION.
! Register Variable
! f0: bits_per_letter
! f30: bits
! {i1,i2}: seed (seed is 64 bits so two 32-bit registers used).
! i0: pw_ptr
! Note: fp registers are caller-saved (and caller-restored).
! Also see comments.
! Compiled with -xarch=v7
! 32 ! for( ; bits >= bits_per_letter; bits -= bits_per_letter )
/* 0x010c 32 */ ldd [%sp+96],%f0 ! Load f0 from stack.
        .L900000118:
/* 0x0110
/* 0x0114
/* 0x0118
/* 0x011c
! 33
! 34
/* 0x0120
/* 0x0124
/* 0x0128
/* 0x012c
sult = %o0
/* 0x0130 */ std %f30,[%sp+104] ! Caller save.
/* 0x0134 */ add %o1,97,%%2 ! {seed%26} + 'a'
/* 0x0138 */ stb %g2,[%i0] ! *pw_ptr={'a'+seed%26}
```



```
! SOLUTION.
!
! Register Variable
! f4: bits_per_letter
! f8: bits
! o0,g2: seed
! i0: pw_ptr
Note: fp registers are still caller-saved (and caller-restored)
        but since there are no calls in the loop there is no need
        to save and restore fp regs. (The V7 code used calls for
        64-bit integer division and remainder, v9 has 64-bit
        divide and multiply instructions it can use instead.)
    Also see comments.
! Compiled With -xarch=v8plus
!
! ! for( ; bits >= bits_per_letter; bits -= bits_per_letter )
/* 0x00e8 32 */ fcmped %fcc0,% %8,%f4
/* 0x00ec 32 */ fbul,a,pt %fcc0,.L900000115
/* 0x00f0 */ stb %g0,[%i0]
! 33 ! ! { { *pw_ptr++ = 'a' + seed % 26;
/* 0x00f4 34 */ udivx %o0,26,%g2 ! %g2 = seed / 26
/* 0x00f8 34 */ mulx %g2,26,%g3 ! g3 = 26 ( seed / 26 )
/* 0x00fc */ sub %o0,%g3,%g3 ! g3 = seed % 26
! 35 ! seed = seed / 26;
/* 0x0100 35 */ or %g0,%g2,%o0 ! o0 = seed / 26
/* 0x0104 */ fsubd %f8,%f4,%f8 ! bits -= bits_per_letter
/* 0x0108 34 */ add %g3,97,%g3 ! 'a' + seed % 26
/* 0x010c */ stb %g3,[%i0] ! *pw_ptr = 'a'
/* 0x0110 */ add %i0,1,%i0 ! pw_ptr++
/* 0x0114 35 */ fcmped %fcc1,%f8,%f4
/* 0x0118 */ fbge,a,pt %fcc1,.L900000114
/* 0x011c */ udivx %o0,26,%g2 ! g2 = seed / 26
    .L77000009:
! 36 ! }
```

Problem 1: Two VAX instructions appear below. VAX documentation can be found via http://www.ece.lsu.edu/ee4720/doc/vax.pdf. Don't print it, it's 544 pages. Take advantage of the extensive bookmarking of the manual to find things quickly. Chapter 5 describes the addressing modes and assembler syntax, Chapter 8 summarizes the VAX ISA, and Chapter 9 lists the instructions. For the instructions look up ext and add then find the mnemonics used below. Pay attention to operand order.
(a) Translate the VAX code below to MIPS (without changing what it does, of course). Ignore overflows and the setting of condition codes.

```
extzv #10, #5, r1, r2
addl2 @0x12034060(r3), (r4)+ # Don't overlook the "@" and "+".
    The solution appears below. Common mistakes are noted in the comments (the code shown is correct).
srl $2, $1, 10
andi $2, $2, 31
lui $10, 0x1203
add $10, $10, $3
lw $10, 0x4060($10)
lw $10, 0($10) # The @ is for indirect, so load again!
lw $11, 0($4)
add $10, $10, $11
sw $10, 0($4)
addi $4, $4, 4 # Increment r4 by the size of the data item.
```

(b) (Extra Credit) Show how the instructions above are coded.

Problem 2: A pipelined MIPS implementation and some MIPS code appear below. The results computed by the MIPS instructions are shown in the comments.

(a) Draw a pipeline execution diagram showing the execution of the code on the implementation. Base your pipeline execution diagram on the illustrated pipeline, do not depend solely on memorized execution timing rules, since they depend on details of the hardware which vary from problem to problem. Show execution until the second fetch of the first instruction.

Diagram shown above.
(b) Determine the CPI for a large number of iterations.
$\mathrm{CPI}=\frac{13}{6}=2.16667$.
(c) Certain wires in the implementation diagram are labeled with letters. (The circled letters with arrows.) Beneath the pipeline execution diagram show the value on those wires at near the end of each cycle. (Write sideways if necessary.) Do not show values if the corresponding stage holds a bubble or a squashed instruction. Only show immediate values if the corresponding instruction uses one. Hint: Three instructions above use an immediate.

Diagram shown above. The immediate holds the branch displacement, which is the number of instructions to skip. Many solutions incorrectly showed the branch target ( $0 \times 1000$ ) in the E row (the immediate value). Some solutions omitted the effective address computed by the sw instruction ( 808 in the C and D rows).
(d) This is a special bonus question that did not appear in the original assignment! For those students who have taken EE 3755 in Fall 2001, identify the Verilog code in http://www.ece.lsu.edu/ee4720/v/mipspipe.html corresponding to each labeled wire.
// Verilog lines shown below (without much context).
// A:
id_ex_npc <= if_id_npc;
// B: The line with the B comment.
always @( id_ex_alu_a_src or id_ex_rs_val or id_ex_sa or id_ex_npc ) case( id_ex_alu_a_src )

SRC_rs: alu_a = id_ex_rs_val;
SRC_np: alu_a = id_ex_npc; // B
SRC_sa: alu_a = \{27'd0, id_ex_sa\};
default: 'UNEXPECTED(alu_a,id_ex_alu_a_src); endcase
// C: The line with the C comment.
always @( posedge clk ) begin
ex_me_npc <= id_ex_npc;
ex_me_pc <= id_ex_pc;
ex_me_alu <= alu_out; // C
ex_me_rt_val <= id_ex_rt_val;
// D: The line with the D comment
always @( posedge clk ) begin
me_wb_npc <= ex_me_npc;
me_wb_pc <= ex_me_pc;
me_wb_dst <= next_me_wb_exc $\phi \varnothing$ reset ? 5'd0 : ex_me_dst;
me_wb_from_mem <= ex_me_size != 0;
me_wb_alu <= ex_me_alu; // D

```
        me_wb_md <= data_in_2;
        me_wb_exc <= ex_me_exc ? ex_me_exc : next_me_wb_exc;
        me_wb_occ <= ~reset & ex_me_occ;
        tb_me_wb_din <= tb_ex_me_din;
    end
```

// E: The case statement and the assignment.

```
// E
case( immed_fmt )
IMM_s: next_id_ex_imm = { immed[15] ? 16'hffff : 16'h0, immed };
IMM_l: next_id_ex_imm = { immed, 16'h0 };
IMM_u: next_id_ex_imm = { 16'h0, immed };
IMM_j: next_id_ex_imm = { if_id_npc[31:28], ii, 2'b0 };
IMM_b: next_id_ex_imm = { immed[15] ? 14'h3fff : 14'h0, immed, 2'b0 };
default: 'UNEXPECTED(next_id_ex_imm,immed_fmt);
endcase
```

// Further below
id_ex_imm <= next_id_ex_imm; // E

Problem 3: Add exactly the bypass paths needed so that the code in the previous problem will run on the implementation below (the same as the one above) with the minimum number of stalls. Indicate the cycles in which the bypass paths will be used and the values bypassed on them.

Solution shown below, added bypass paths are in red bold. A pipeline execution diagram is also shown.
Almost all submitted solutions included the bypass path from the MEM stage to the upper ALU mux. Very few properly included the bypass path for the branch conditions. (Some incorrectly showed the bypass path into the ALU, which is used here to compute the branch target.) No submitted solution included a bypass path for the store value.


```
# Solution. (Goes a bit past the sec- ond fetch of the first instruction.)
LOOP:
    # Cycle 
addi $1, $2, 4 IF ID EX ME WB IF ID EX ME WB
sub $3, $0, $3 IF ID EX ME WB IF ID EX ME
and $1, $1, $6 IF ID EX ME WB IF ID EX
or $4,$1,$5 IF ID EX ME WB IF ID
bne $4, $3, LO IF ID EX ME WB IF
sw $4, 7($8) IF ID EX ME WB
add $10, $11, IF IDx
    # Cycle 
```

Problem 1: The exception mechanism used in the MIPS 32 ISA differs in some ways from the SPARC V8 mechanism covered in class. See Chapter 7 in
http://www.ece.lsu.edu/ee4720/sam.pdf for the SPARC V8 exception information and http://www.ece.lsu.edu/ee4720/mips32v3.pdf for a description of the MIPS mechanism. The MIPS description is a bit dense, so start early and ask for help if needed.
(a) Describe how the methods used to determine which exception was raised differ in SPARC V8 and MIPS 32. Use an illegal (reserved) instruction error as an example. Shorter answers will get more credit so concentrate on explaining how the processor identifies the exception (was it an illegal instruction, an arithmetic overflow, etc) and avoid irrelevant details. For example, details on how the processor switches to system mode is irrelevant.

The way the question should be answered:
The difference is that in SPARC a particular exception causes a particular handler to run, so that if the illegal instruction handler is run it must mean that an illegal instruction exception occurred. In MIPS the handler must read a cause register to find out which exception occurred.

Additional Information:
In both ISAs a number is associated with each kind of exception, in SPARC it is called the trap type, and in MIPS it is referred to as an exception type; it will be called an exception code here. The question is asking how the methods used by the handler to determine the exception code differ. In both cases the hardware generates an exception code.

In SPARC V8 the exception code is used to form a trap table entry address; a control transfer is made to this address. At this address is the first four instructions of the handler (first eight in V9). The handler for the illegal instruction exception "knows" an illegal instruction exception occurred because that's the only exception that would cause it to run. (The trap type or an illegal instruction exception is 16.)

MIPS 32 also has an exception table but it has far fewer entries. To determine which exception type caused it to run the MIPS handlers read a cause register which contains the exception type.
(b) Where do the two ISAs store the address of the faulting instruction? Both ISAs have delayed branches, so why does SPARC store two return addresses while MIPS gets away with one?
(SPARC registers are organized like a stack, on a procedure call a save instruction "pushes" a fresh set of registers on the stack, and a restore instruction "pops" the registers, returning to the previous set. The set of visible registers is called a window. This mechanism reduces the need to save and restore registers in memory. This piece of information is needed for the previous problem.)

In SPARC V8 an exception will cause the current window pointer to advance, saving the interrupted code's registers and providing a fresh set of registers to the handler. The PC and NPC of the faulting instruction will be stored in registers 11 and 12. In MIPS 32 only the PC is saved, it is saved in a special EPC register. If the faulting instruction is in a branch delay slot the PC of the branch is saved, otherwise the PC of the faulting instruction is saved.

Suppose the instruction in a branch delay slot of a taken branch raises an exception and is to be re-executed. In MIPS 32 control returns to the branch before the instruction so both the branch and the faulting instruction re-execute. (Since the instruction before the faulting instruction re-executes this is not a precise exception by the definition given in class. Since the branch does not modify registers [other than PC ] or memory it can be used in the same way a precise exception is used, and so in MIPS 32 such exceptions are called precise.) Control is returned to the branch using something like an ordinary jump instruction, except that the processor switches back to user mode. Jumping directly to the faulting instruction would be a challenge because after the faulting instruction is executed the branch target needs to be executed. MIPS has no way to do these kinds of jumps and so there is no need to store NPC.

SPARC on the other hand can return directly to an instruction in the delay slot. It does so using two consecutive control transfer instructions, something forbidden in MIPS. A jmpl instruction jumps to the saved PC, a rett (return from trap) instruction jumps to the NPC.

To summarize, SPARC saves two addresses because it needs both of them to restart an instruction in a branch delay slot. MIPS stores only one because it never returns from exceptions to a branch in a delay slot, instead it re-executes the branch.

Problem 2: The pipeline execution diagram below is for code running on a MIPS implementation developed just for this homework problem! Note that the program itself is missing. The dog deleted it. The $M_{-}$and $A_{-}$refer to parts of the multiply and add functional units with segment numbers omitted for this problem. A WBx indicates that an instruction does not write back to avoid a WAW hazard.

```
IF ID M_ M_ M_ M_ M_ M_ WB
    IF ID ----> M_ M_ M_ M_ M_ M_ WB
        IF ----> ID ----> A_ A_ WB
            IF ----> ID M_ M_ M_ M_ M_ M_ WBx
            IF ID A_ A_ WB
                        IF ID A_ A_ WB
```

(a) Write a program consistent with the diagram. Pay attention to dependencies.

```
# Solution
mul.d f0, f2, f4 IF ID M1 M1 M1 M2 M2 M2 WB
mul.d f6, f8, f10 IF ID ----> M1 M1 M1 M2 M2 M2 WB
add.d f12, f0, f14 IF ----> ID ----> A1 A2 WB
mul.d f16, f18, f20 IF ----> ID M1 M1 M1 M1 M1 M1 WBx
add.d f16, f22, f24 IF ID A1 A2 WB
add.d f26, f28, f30 IF ID A1 A2 WB
```

(b) Identify the latency and initiation interval of the functional units. Fill in the segment numbers. Multiply: Iatency, 5 ; initiation interval, 3 . Add: Iatency, 1 ; initiation interval, 1.

Problem 3: In the MIPS implementation below (also shown in class) branches are resolved in the ID stage. Resolution of a branch direction (determining whether it was taken) must wait for register values to be retrieved and, for some branches, compared to each other. Suppose this takes too long.

(a) Show the modifications needed to do the equality comparison in the EX stage. The modified hardware must use as little additional hardware as possible and, to maximize performance, should only do an EX-stage equality comparison when necessary. Don't forget about branch target address handling. Hint: The modifications are easy.

The ID-stage adder that computes branch displacements is also connected to a new ID/EX latch, the output of this new latch is connected to the PC multiplexor. The ALU in the EX stage does the register comparison for the branch. Note that the only added hardware is the latch and the new paths. (A diagram may be added to this solution at some point.)
(b) Write a code fragment that runs differently on the two implementations and show pipeline execution diagrams for the code on the two implementations.

```
# Solution
    # Execution on original system.
    #
    # Cycle 0
    beq $2, $3 TARG IF ID EX ME WB
    add $4, $5, $6 IF ID EX ME WB
    #...
TARG:
    xor $6, $7 IF ID
    # Execution on modified system.
#
    # Cycle 0
    beq $2, $3 TARG IF ID EX ME WB
    add $4, $5, $6 IF ID EX ME WB
    sub $7, $8, $9 IFx
```

\#. .
TARG:
xor $\$ 6$, $\$ 7$ IF ID
(c) The table below lists SPARC instructions and indicates how frequently they were used when running $\mathrm{T}_{\mathrm{E}} \mathrm{X}$ to prepare this homework assignment. (Many rows were omitted to save space, so the "\%exec" column will not add to $100 \%$.) Suppose that the instruction percentages are identical for MIPS (which means totally ignoring the cc instructions). Assume that SPARC be and be, a are equivalent to MIPS beq, SPARC bne and bne, a are equivalent to MIPS bne, and that the other branch instructions (they begin with a b), are equivalent to branch instructions that compare to zero (bgez, etc.).

Suppose the clock frequency of the original design is 1.0000 GHz . Based on the data below and making any necessary assumptions, for what clock frequency would the new design run a program in the same amount of time as the old one? What column would you add (what additional data do you need) to the table to make your answer more precise?

Assume that floating-point instructions are insignificant and that there are no stalls due to memory access.

| opcode | \#exec | $\%$ exec |
| :--- | ---: | ---: |
| subcc | 4659360 | $12.6187 \%$ |
| lduw | 4521722 | $12.2459 \%$ |
| add | 4159629 | $11.2653 \%$ |
| or | 3110542 | $8.4241 \%$ |
| sethi | 3066797 | $8.3056 \%$ |
| stw | 1848293 | $5.0056 \%$ |
| sll | 1402122 | $3.7973 \%$ |
| be | 1393475 | $3.7739 \%$ |
| jmpl | 1140223 | $3.0880 \%$ |
| call | 1088068 | $2.9467 \%$ |
| ldub | 1064918 | $2.8841 \%$ |
| bne | 936493 | $2.5362 \%$ |
| stb | 687981 | $1.8632 \%$ |
| srl | 609402 | $1.6504 \%$ |
| save | 526477 | $1.4258 \%$ |
| restore | 526474 | $1.4258 \%$ |
| bne,a | 453545 | $1.2283 \%$ |
| nop | 433253 | $1.1734 \%$ |
| bge | 429978 | $1.1645 \%$ |
| ldsb | 429497 | $1.1632 \%$ |
| orcc | 382947 | $1.0371 \%$ |
| and | 370967 | $1.0047 \%$ |
| be, a | 360057 | $0.9751 \%$ |
| sub | 354847 | $0.9610 \%$ |
| ba | 321970 | $0.8720 \%$ |
| bl | 297715 | $0.8063 \%$ |
| andcc | 270465 | $0.7325 \%$ |
| bgu | 235304 | $0.6373 \%$ |
| bl,a | 216074 | $0.5852 \%$ |
| sra | 204610 | $0.5541 \%$ |
| ble | 198154 | $0.5366 \%$ |
| xor | 185137 | $0.5014 \%$ |
| bcs | 182153 | $0.4933 \%$ |
| addcc | 155156 | $0.4202 \%$ |
| bleu | 142755 | $0.3866 \%$ |
| bg | 117582 | $0.3184 \%$ |
| mulscc | 88681 | $0.2402 \%$ |
|  |  |  |

In the new design there will be a bubble added for taken branches that compare two registers. Assume the original system has a CPI of 1 and that half the branches are taken. The percentage of branches that add a bubble is found by adding the percentages for be, bne, bne, a , and be, a : and dividing by two: $\frac{8.5135 \%}{2}=4.25675 \%$. The new CPI will then be 1.0425675 . To find the clock frequency of the new system for which it will run as fast as the old system solve: $\frac{1.0}{\phi_{\text {old }}}=\frac{1.0425675}{\phi_{\text {new }}}$ for $\phi_{\text {new }}$ to get $\phi_{\text {new }}=1.0425675 \mathrm{GHz}$, where $\phi_{\text {old }}=1 \mathrm{GHz}$.

To make the answer more precise two things are needed, the CPI on the original system and the fraction of times a branch is taken.

To solve Problem 3 and the next assignment a paper has to be read. Do not leave the reading to the last minute, however try attempting the first problem below before reading the paper.

Problem 1: The pipeline below was derived from the five-stage statically scheduled MIPS implementation by splitting each stage (except writeback) into two stages. Each pair of stages (say IF1 and IF2) does the same thing as the original stage (say IF), but because it is broken in to two stages it takes two rather than one clock cycle. The diagram shows only a few details. Bypass connections into the ALU are available from all stages from MEM1 to WB.


The advantage of this pipeline is that the clock frequency can be doubled. (Actually not quite times two.) Perfect execution is shown in the diagram below:

```
add $1, $2, $3 IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB
sub $4, $5, $6 IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB
and $7, $8, $9 IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB
```

(a) Suppose the old five-stage system ran at a clock frequency of 1 GHz and the new system runs at 2 GHz . How does the execution time compare on the new system when execution is perfect? It's half! That is, performance scaled with clock frequency.
(b) Show a pipeline execution diagram of the code below on the new pipeline. Note dependencies through registers \$10 and \$11.

```
add $10, $2, $3 IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB
sub $4, $10, $6 IF1 IF2 ID1 ID2 --> EX1 EX2 ME1 ME2 WB
and $11, $8, $9 IF1 IF2 ID1 --> ID2 EX1 EX2 ME1 ME2 WB
or $20, $21, $22 IF1 IF2 --> ID1 ID2 EX1 EX2 ME1 ME2 WB
xor $7, $11, $0 IF1 --> IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB
```

(c) In the previous part there should be a stall on the new pipeline that does not occur on the original pipeline. (It's not too late to change your answer!) How does that affect the usefulness of splitting pipeline stages?

Assuming not all adjacent instructions are truely dependent, splitting is still useful, but performance does not seale with clock frequency. (It never does.)
(d) (Optional, complete before reading paper.) To get that I'm-so-clever feeling answer the following: Suppose there is no way a 32 -bit add can be completed in less than two cycles. Is there any way to perform addition so that results can be bypassed to an immediately following instruction, as
in the example above, but without stalling? The technique must work when adding any two 32 bit numbers. Hint: The adder would have to be redesigned. (A question in the next homework assignment revisits the issue.)

Split the ALU in to sixteen-bit parts and bypass the low and high parts separately.

Problem 2: Note: The following problem is similar to one given in the Fall 2001 semester, see http://www.ece.lsu.edu/ee4720/2001f/hw03.pdf (the problem) and http://www.ece.lsu.edu/ee4720/2001f/hw03_sol.pdf (the solution). For best results do not look at the solutions until you're really stuck. This problem below uses MIPS instead of DLX and is for Method 3 instead of Method 1. The code below executes on a dynamically scheduled four-way superscalar MIPS implementation using Method 3, physical register numbers.

- Loads and stores use the load/store unit, which consists of segments L1 and L2.
- The floating-point multiply unit is fully pipelined and consists of six segments, M1 to M6.
- The usual number of instructions (for a 4-way superscalar machine) can be fetched, decoded, and committed per cycle.
- An unlimited number of instructions can complete (but not commit) per cycle. (Not realistic, but it makes the solution easier.)
- There are an unlimited number of reservation stations, reorder buffer entries, and physical registers.
- The target of a branch is fetched in the cycle after the branch is in ID, whether or not the branch condition is available. (We'll cover that later.)
(a) Show a pipeline execution diagram for the code below until the beginning of the fourth iteration. Show where instructions commit.

See diagram below.
(b) What is the CPI for a large number of iterations? Hint: There should be less than six cycles per iteration.

The CPI is $\frac{3}{6}=0.5$.
(c) Show the entries in the ID and commit register maps for registers $f 0$ and $\$ 1$ for each cycle in the first two iterations. If several values are assigned in the same cycle show each one separated by commas.
(d) Show the values in the physical register file for f 0 and $\$ 1$ for the first two iterations. Use a "]" to show when a physical register is removed from the free list and use a "[" to show when it is put back in the free list.

See pipeline execution diagram on the next page.

```
! Solution
LOOP: ! Instructions shown in dynamic order. (Instructions repeated.)
    ! Cycle 0
    ldc1 f0, 0($1) IF ID Q L1 L2 WC
    mul.d f0, f0, f2 IF ID Q M1 M2 M3 M4 M5 M6 WC
    sdc1 0($1), f0 IF ID Q L1 L2 WC
    addi $1, $1, 8 IF ID Q EX WB
    bne $2, $0 LOOP IF ID Q B WB
    sub $2, $1, $3 IF ID Q EX WB C
    # Cycle 
    ldc1 f0, 0($1) IF ID Q L1 L2 WB C
    mul.d f0, f0, f2 IF ID Q M1 M2 M3 M4 M5 M6 WC
    sdc1 0($1), f0 IF ID Q L1 L2 WC
    addi $1, $1, 8 IF ID Q EX WB C
    bne $2, $0 LOOP IF ID Q B WB C
    sub $2, $1, $3 IF ID Q EX WB C
    # Cycle 
    ldc1 f0, 0($1) IF ID Q L1 L2 WB C
    mul.d f0, f0, f2 IF ID Q M1 M2 M3 M4 M5 M6 WC
    sdc1 0($1), f0 IF ID Q L1 L2 WC
    addi $1, $1, 8 IF ID Q EX WB C
    bne $2, $0 LOOP IF ID Q B WB C
    sub $2, $1, $3 IF ID Q EX WB C
    # Cycle 
    ldc1 f0, 0($1) IF ID Q L1 L2 WB
    mul.d f0, f0, f2 IF ID Q M1 M2 M3 M4 M5
    # Cycle 
    ID Map
    f0 99 97,96 94,93 91,90
    $1 98 95 92 89
    # In cycle one first 97 is assigned to f0, then 96 (replacing 97). The
    # same sort of replacement occurs in cycles 4 and 7.
    # Cycle 
    Commit Map
```



```
    ! Cycle 
    # Cycle 
    Physical Register File
    99 1.0 ]
    98 0x1000 ]
    97 [ 10 ]
    96 [ 11 ]
    95 [ 0x1008 ]
```



```
    92 [ 0x1010 ]
    ! Cycle 
```

The following is an introduction to the next few problems.
As mentioned several times in class many of the performance-enhancing microarchitectural features that came in to wide use in the closing decades of the twentieth century (I love the way that sounds!) are much easier to apply to RISC ISAs than CISC ISAs. Bound by golden handcuffs to the CISCish IA-32 ISA, Intel was forced to get RISC-level performance from IA-32. (Not just Intel, DEC [now Compaq, perhaps soon HP] faced the problem with the VAX ISA and IBM with 360.) The solution chosen by Intel (and also DEC) was to translate individual IA-32 instructions in to one or more $\mu \mathrm{ops}$ (micro-operations). Each $\mu \mathrm{op}$ is something like a RISC instruction and so the parts of the hardware beyond the IA-32 to $\mu$ op translation can employ the same techniques used to implement RISC ISAs.

The paper at http://www.intel.com/technology/itj/q12001/articles/art_2.htm
and http://www.ece.lsu.edu/ee4720/s/hinton_p4.pdf (password needed off campus, will be given in class) describes the Pentium 4 implementation of IA-32, including $\mu$ ops (which are typeset using "u" instead of the Greek letter " $\mu$ ", except occasionally in figures). This paper was not written for a senior-level computer architecture class four weeks from the end of the semester and so it will include material which we have not yet covered (caches and TLBs) and some material not covered at all. Some stuff in the paper is not explained (how they do branch prediction or what the Pentium 4 pipeline segments in Figure 3 mean), some of this can be figured out other things have to be found out elsewhere (but not for this assignment).

Read the paper and answer the question below. The next homework assignment will include additional questions on the paper. For this initial reading skip or lightly read material on the L2 cache, L1 data cache, and the ITLB. Questions on the cache material might be asked in a later assignment.

Problem 3: What does the paper call the following actions and components (that is, translate from the terminology used in class to the terminology used in the paper):

Commit - Retire
ID Register Map - Frontend RAT
Commit Register Map - Retirement RAT
Physical Register File - Physical Register File

The following questions are based on the paper at http://www.intel.com/technology/itj/q12001/articles/art_2.htm and http://www.ece.lsu.edu/ee4720/s/hinton_p4.pdf (password needed off campus, will be given in class). See Homework 4 (http://www.ece.lsu.edu/ee4720/2002/hw04.pdf) for an introduction to the paper.

Problem 1: What is the maximum sustainable IPC of the IA-32 (in $\mu$ op per cycle)? Put another way, the Pentium 4 is an $n$-way superscalar processor, what is $n$ ?

Maximum is $3 \mu \mathrm{PC}(\mu 0 \mathrm{p}$ per cycle), limited by the $3-\mu 0 \mathrm{p}$ decode limit and also the $3-\mu 0 \mathrm{p}$ retire limit.

Problem 2: The Pentium 4 can decode no more than one IA-32 instruction per cycle. How then can it execute more than one IA-32 instruction per cycle (at least for small code fragments prepared by a friendly programmer)?

Decoded instructions are stored in the trace cache. A trace cache line might contain $\mu$ ops spanning more than one IA-32 instruction. Though it took at least two cycles to decode them, once stored in the trace cache they can be used multiple times, each time multiple IA-32 instructions are issued in one cycle.

Problem 3: One problem with superscalar systems noted in class is the wasted instructions following the delay slot of a taken branch near the beginning of a fetch group. How does the Pentium 4 avoid this?

By placing instructions in a trace cache line in dynamic order, so that the target of a branch is right after the branch, there is no need to separately fetch it.

Problem 4: The fast $(2 \times)$ integer ALUs have three stages, an initiation interval of 1 fast cycle ( $\frac{1}{2}$ processor cycle), and a latency of zero fast cycles. Why is this surprising (not the one half part)? How does it do it?

It's surprising because normally something with three stages would have a latency of two. It works because each stage can bypass partial results (the low and high half of the result) which are enough for the dependent instruction.

Problem 5: In describing store-to-load forwarding the paper describes a special case for which data could be forwarded (bypassed) but is not because it would be too costly. Using MIPS code (or IA-32 if you prefer) provide an example of this special case.

```
sb $2, 1($3)
sb $4, 2($3)
lw $5, 0($3)
```

Problem 6: In Figure 8 the performance of a 1 GHz Pentium III is compared to a 1.5 GHz Pentium 4. Why is it reasonable for the Pentium 4 to be compared at a higher clock frequency?

Because the Pentium 4's shorter stages enable a higher clock frequency. When implemented on the same process technology, the Pentium 4 would have the higher clock frequency.

## 69 Fall 2001 Solutions

## EE 4720

Homework 1 solution
Due: 10 October 2001
Problem 1: In DLX the three instructions below, though they do very different things, are of the same type (format).

```
bnez r2, SKIP
lw r1, 1(r2)
addi r1, r2, #1
SKIP:
```

Because of their similarity their implementations in the diagram below shares a lot of hardware.

(a) Show how these DLX instructions are coded.

DLX:
bnez r2, SKIP

(b) Find corresponding instructions in the SPARC V9 ISA. (See the SPARC Architecture Manual V9, http://www.ece.lsu.edu/ee4720/samv9.pdf) (The DLX branch instruction will have to be replaced by two instructions, one to set the condition code registers.)

```
! In this solution the DLX branch is replaced by a single instruction.
brnz %g1, SKIP
ldsw [%g2+1],%g1
add %g2, 1, %g1
! In this solution the DLX branch is replaced by two instructions.
addcc %g1, 0, %g0
bne SKIP
ldsw [%g2+1],%g1
add %g2, 1, %g1
```

(c) Show the coding of the SPARC V9 branch, load, and add immediate instructions (but not the condition code setting instruction).
brnz g1, SKIP

| op | a | 0 | reond | op2 | dh | p | rs1 |  | disp |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | 0 | 5 | 3 | 0 | 0 |  | 2 |  | 2 |
| $\begin{array}{lllllllllllll}3130 & 29 & 29 & 28 & 28 & 27 & 25 & 24 & 22 & 21 & 20 & 19 & 19 \\ 18 & 14 & 13 & 0\end{array}$ |  |  |  |  |  |  |  |  |  |  |

bne SKIP

ldsw [g2+1],g1

add g2, 1, g1

| op | rd | op3 |  | rs1 | 1 |  | simm13 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 2 | 2 | 1 | 0 |  | 2 | 1 |  | 1 |
| 3130 |  |  | $24 \quad 19$ |  |  | 1313 |  | 0 |

(d) Do these codings allow the same degree of hardware sharing?

Because the DLX codings are identical an implementation could use the same datapath for computing the immediate add, load effective address, and branch target. The SPARC V9 add and lduw codings are identical and so hardware can be shared but the placement of the displacement is different for the branch instruction (either one) and so additional hardware would be needed to select the immediate (or displacement) bits corresponding to the instruction.

Problem 2: Write a DLX assembly language program that determines the length of the longest run of consecutive elements in an array of words. For example, in array $\{1,7,7,1,5,5,5,7,7\}$ the longest run is three: the three 5's (the four 7's are not consecutive). The comments below show how registers are initialized and where to place the longest run length.

```
! r10 Beginning of array (of words).
! r11 Number of elements.
! r1 At finish, should contain length of the longest run.
! r10 Beginning of array
! r11 Number of elements
! r1 At finish, should contain length of longest run.
! r1 Longest run encountered.
! r2 Size of this run so far.
! r3 Last element.
add r1, r0, r0
add r2, r0, r0
lw r5, O(r10)
addi r3, r5, #1
LOOP:
beqz r11, DONE
lw r5, O(r10)
addi r10, r10, #4
subi r11, r11, #1
seq r6, r5, r3
beqz r6, NEW_RUN
add r2, r2, #1
j LOOP
NEW_RUN:
add r7, r2, r0
addi r2, r0, #1
sgt r6, r7, r1
add r3, r5, r0
beqz r6, LOOP
add r1, r7, r0
j LOOP
```

Problem 3: Small integers can be stored in a packed array to reduce the amount of storage required; the array can be unpacked into an ordinary array when the data is needed. Write a DLX assembly language program to unpack an array containing $n b$-bit integers stored as follows. The low $b$ bits (bits 0 to $b-1$ ) of the first word of the packed array contain the first integer, bits $b$ to $2 b-1$ contain the next, and so on. When the end of the word is reached integers continue on the second word, etc. Size $b$ is not necessarily a factor of $n$ and so an integer might span two words.

The diagram below shows how the first 6 integers $i_{0}, i_{1}, \ldots, i_{5}$ are stored for $b=12$ bits and $n \geq 6$.


Write DLX assembly language code to unpack such an array into an array of signed words. The packed array consists of $n b$-bit signed numbers, with $b \in[1,32]$. Initial values of registers are given below.

```
    ! Initial values
    ! r10: Address of start of packed array.
    ! r11: Number of elements (n).
    ! r12: Size of each element, in bits (b).
    ! r14: Address of start of unpacked array.
    ! Initial values
    ! r10: Address of start of packed array.
    ! r11: Number of elements (n).
    ! r12: Size of each element, in bits (b).
    ! r14: Address of start of unpacked array.
    ! r1: Current word.
    ! r2: Mask
    ! r3: Unpacked item
    ! r4: Bits remaining in current word.
    ! r8, r9: Miscellaneous
    ! r5: 32 - size of each element.
```

    add \(r 4, r 0, r 0\)
    add r8, r12, r0
    addi r5, r0, 32
    sub r5, r5, r12
    addi r2, r0, \#1
    sll r2, r2, r12
    subi r2, r2, \#1
    LOAD_MAYBE:
bnez r4, LOWPART
lw r1, 0(r10)
addi r10, r10, \#4
addi r4, r0, \#32
LOWPART:
and $r 3, r 1, r 2$
srl r1, r1, r12
sub r4, r4, r12
slt r9, r4, r0

```
bnez r9, SPAN
j STORE
SPAN:
lw r1, O(r10)
addi r10, r10, #4
add r8, r12, r4
sll r9, r1, r8
or r3, r3, r9
and r3, r3, r2
addi r4, r4, #32
! Fall through to store
STORE:
sll r3, r3, r5
sra r3, r3, r5 ! Sign extend
sw O(r14), r3
addi r14, r14, #4
subi r11, r11, #1
bnez r11, LOAD_MAYBE
```

DONE:

Problem 1: Answer the following questions about the MIPS Technologies 4Km processor core. The processor is documented in http://www.mips.com/declassified/Declassified_2000/MD00016-2B-4K-SUM-01.15.pdf.
(a) For each stage in the statically scheduled DLX implementation show where the same work is done in the 4 Km pipeline. Note that work done in one DLX stage might be performed in more than one 4 Km pipeline stage.

IF: 1
ID: E
EX: Part of the ALU operation done in E , part in M . Address calculations for load and stores done in E .
MEM: Memory load and store done in M stage. Alignment done in A stage.
WB: W.
(b) The 4 Km documentation uses the term stall differently than used in class. How do their usages differ? What term does the documentation use that is close to stall as used in class? (See section 2.8.1)

By stall the 4 Km documentation means take more than one cycle to complete a computation, as do the floating-point units in DLX. Unlike the use in class, it does not mean that instructions following (more recently fetched than) the stalled instruction are stopped. The documentation uses the term slip for what is meant by stall in class.
(c) A MIPS implementation needs to do all of the following:
(1) arithmetic and logical operations for ordinary instructions
(2) compute the target of a branch
(3) compute the effective address of a load or store

In the first pipelined DLX implementation all of these were performed by the ALU. MIPS has a branch instruction in which a branch is taken if two registers are equal (beq) or not equal (bne). So it must also
(4) determine if two values are equal

How many of these are shared? If they are not shared, why not? (The documentation does not state exactly what hardware is present, answer the question by looking at how instructions execute.)

The ALU, effective address computation, and part of a branch target computation (I-AC2) may be shared. All of these are done in the second half of E (the ALU is also used in the first half of M ). An instruction needs to do at most one of these things. (e.g., load or store instructions, which compute effective addresses, do not need to compute branch targets or need to use the ALU for other arithmetic or logical operations.) Therefore these [(1), (2), and (3)] can be shared.

According to 2.1.2 an instruction address is determined in E , and so the condition must be evaluated in E. Register values are not available until the second half of E so the register comparison to determine the branch condition must also be evaluated in the second half of E , the same time as branch target address computation (assuming that's what I-AC2 does). Therefore separate hardware is needed for the branch condition.

Problem 2: The program below runs on the DLX implementation shown below. The hardware makes no special provisions for the tricky technique used. The coding for a nop (actually add r0, r0, r0) is all zeros.

Why isn't this an infinite loop? (For those who know why it matters, assume there is no cache.)

The sw instruction replaces the j instruction with a nop.
Why will the code run for at least two iterations?
Because in the first iteration the sw instruction reaches MEM after the $j$ is already fetched. If a write to the MEM-stage memory port is seen by the IF-stage memory port then the loop will perform only two iterations.


```
LOOP:
    lw r1, O(r2)
    addi r2, r2, #4
    add r3, r3, r1
    sw 0x100(r0), r0
LINE: LINE = 0x100
    j LOOP
```

Problem 3: Show a pipeline execution diagram for the code below running on a 4 -way statically scheduled superscalar processor. All needed bypass paths are available, including one for the branch condition. Determine the CPI for a large number of iterations.

```
    and r2, r2, r8
LOOP: ! LOOP = 0x1008
    lw r1, O(r2)
    add r3, r3, r1
    addi r2, r2, #4
    sub r4, r2, r5
    bneq r4, LOOP
        Based on the PED below the CPI is }\frac{7}{5}=1.4\mathrm{ . The pipeline execution diagram is for the second (or later) iteration.
    and r2, r2, r8
LOOP: ! LOOP = 0x1008
    ! Cycle 
    lw r1, O(r2) IF ID EX ME WB IF
    add r3, r3, r1 IF ID ----> EX ME WB IF
    addi r2, r2, #4 IF ----> ID EX ME WB
    sub r4, r2, r5 IF ----> ID -> EX ME
    bneq r4, LOOP IF ----> ID ----> EX
```

Problem 4: The code from the problem above can be improved (stalls can be removed) to a small extent by scheduling, but that would still leave some stalls. This might see like a good candidate for loop unrolling.
(a) Show why it would take alot of unrolling to eliminate all stalls. (You don't have to show the unrolled code, since it would be long.)

Because of the 1-cycle load latency the consuming add instruction would have to be placed seven instructions away. Two of those can be an addi and sub, the rest would be 1 w , so the loop would be unrolled six times. This is shown below. The code has been slightly re-structured To facilitate unrolling positions of the sub and addi have been switched, with a compensating instruction added before the loop. To avoid added dependencies six running sums are computed, at the end of the loop these are added together.

```
and r2, r2, r8
subi r5, r5, #24 ! Compensate for switching position of sub and addi below.
nop
LOOP: ! LOOP = 0x1010
lw r1, O(r2) IF ID EX ME WB
lw r11, 4(r2) IF ID EX ME WB
lw r12, 8(r2) IF ID EX ME WB
lw r13, 12(r2) IF ID EX ME WB
lw r14, 16(r2) IF ID EX ME WB
lw r15, 20(r2) IF ID EX ME WB
sub r4, r2, r5 IF ID EX ME WB
addi r2, r2, #24 IF ID EX ME WB
add r3, r3, r1 IF ID EX ME WB
add r21, r21, r11 IF ID EX ME WB
add r22, r22, r12 IF ID EX ME WB
add r23, r23, r13 IF ID EX ME WB
add r24, r24, r14 IF ID EX ME WB
add r25, r25, r15 IF ID EX ME WB
bneq r4, LOOP IF ID EX ME WB
```

```
! Note: Could add differently to avoid stalls.
    add r3, r3, r21 IF IDx
    add r3, r3, r22 IFx
    add r3, r3, r23 IFx
    add r3, r3, r24 IFx
    add r3, r3, r25 IFx
```

(b) Use software pipelining and scheduling to remove the stalls. (Hint: to software pipeline switch the 1 l and add instructions, and make any other necessary changes.) What is the CPI for a large number of iterations of the modified code?

The loop below runs with a CPI of $\frac{3}{5}=0.6$. The add and $1 w$ were switched and prolog and epilog code, instructions before and after the loop to compensate, was added. Software pipelining was also used for the branch condition: the sub and addi were reversed.

```
and r2, r2, r8
add r1, r0, r0
subi r5, r5, #4
LOOP: ! LOOP = 0x1010
    ! Cycle 0
    add r3, r3, r1 IF ID EX ME WB
                            IF ID
    lw r1, O(r2) IF ID EX ME WB
                            IF ID
    sub r4, r2, r5 IF ID EX ME WB
                            IF ID
    addi r2, r2, #4 IF ID EX ME WB
                            IF ID
    bneq r4, LOOP IF ID EX ME
    add r3, r3, r1 IF IDx
```

(c) Would loop unrolling provide further gains?

It always does. As always, unrolling would reduce the proportion of loop index instructions (those computing the address of the load and the branch condition). Unrolling might place the branch in the last position in a group, reducing fetch waste. Because there are fewer iterations, it will reduce the number of instructions squashed due to the taken branch.

Problem 1: The code below executes on a dynamically scheduled four-way superscalar DLX implementation that uses reorder buffer entry numbers to name destination registers.

- Loads and stores use the load/store unit, which consists of segments L1 and L2.
- The floating-point multiply unit is fully pipelined and consists of six segments.
- The usual number of instructions (for a 4-way superscalar machine) can be fetched, decoded, and committed per cycle.
- An unlimited number of instructions can complete per cycle. (This makes the solution easier.)
- There are an unlimited number of reservation stations and reorder buffer entries.
- The target of a branch is fetched in the cycle after the branch is in ID, whether or not the branch condition is available. (We'll cover that later.)
(a) Show a pipeline execution diagram for the code below until the beginning of the fourth iteration. Show where instructions commit. See diagram below.
(b) What is the CPI for a large number of iterations? Hint: There should be less than six cycles per iteration.

The CPI is $\frac{3}{6}=0.5$.
(c) Show the entries in the register map for registers $f 0$ and $r 1$ for each cycle. (Make up reorder buffer entry numbers.)

See pipeline execution diagram on the next page.

```
    ! Solution
LOOP:
    \(\begin{array}{lllllllllllllllll}\text { ! Cycle } & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15\end{array}\)
    ld f0, O(r1) IF ID L1 L2 WC
        IF ID L1 L2 WB C
        IF ID L1 L2 WB C
        IF ID L1 L2 WB
    muld f0, f0, f2 IF ID RS RS M1 M2 M3 M4 M5 M6 WC
                        IF ID RS RS M1 M2 M3 M4 M5 M6 WC
                            IF ID RS RS M1 M2 M3 M4 M5 M6 WC
                                IF ID RS RS M1 M2 M3 M4 M5 M6 WB
    sw \(0(r 1), f 0\) IF ID L1
            IF ID L1
                                IF ID L1
    addi r1, r1, \#8 IF ID EX WB
            IF ID EX WB
                            IF ID EX WB
    sub r2, r1, r3 IF ID EX WB
                            IF ID EX WB
                            IF ID EX WB
                                C
                                L2 WC
```


## IF ID L1

IF ID L1
addi r1, r1, \#8 IF ID EX WB
IF ID EX WB
IF ID EX WB
sub r2, r1, r3 IF ID EX WB
IF ID EX WB
IF ID EX WB
bnez r2, LOOP

```
L2 WC
L2 WC
C

C
C
C
```


## C

```
IF ID RS B WB
IF ID RS B WB
\(\begin{array}{lllllllllllllllllll}1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17\end{array}\)
ID Map
f0 1.0* \#7,8 \#13,14 \#19,20 \#25,26 \#32,\#33 \#38,39
r1 0x1000* \#10 0x1008 \#22 0x1018 \#34 0x1028
r1
! Note: Because of space restrictions r1 is shown on two lines. The
! first character of an entry is the cycle number for the entry. For
! example, \(0 \times 1008\) is written in to the map at cycle 3 and \#16 at cycle 4.
Commit Map
f0 1.0* \(10.0 \quad 11 \quad 2022 \quad 3033\)
r1 0x1000*
\(\begin{array}{lllllllllllllllllll} & 0 & 1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16\end{array} 17\)
* Initial Values
f0: 1.0
r1: 0x1000
f2: 1.1
\(\operatorname{Mem}[0 \times 1000]=10.0\)
Mem \([0 \times 1008]=20.0\)
\(\operatorname{Mem}[0 \times 1010]=30.0\)
```

(d) The first two instructions of the code below are different than the code above, the other instructions are identical. It runs on a system identical to the one above except that there are only 1000 reorder buffer entries. (That's actually a lot, but it's not unlimited.) What is the CPI for a large number of iterations? Is the CPI really lower in the period before reorder buffers are used up? If you can, solve the problem without drawing a complete pipeline execution diagram.

```
LOOP: ! LOOP = 0x1000
    ld f4, O(r1)
    muld f0, f0, f4
    sw O(r1), f0
    addi r1, r1, #8
    sub r2, r1, r3
    bnez r2, LOOP
```

The CPI is $\frac{6}{6}=1$. Though iterations start every three cycles before the reorder buffer fills, the state of the system is different at each start (in particular, the number of instructions waiting in the reorder buffer increases), and so one cannot base CPI on an iteration time of three cycles. The number of cycles per iteration is limited by the time needed to multiply, which is six.
ld f0, $0(r 1)$
muld f0, f0, f2 M1 M2 M3 M4 M5 M6 WC
M1 M2 M3 M4 M5 M6 WC
M1 M2 M3 M4 M5 M6 WC
sw $0(r 1)$, f0
addi r1, r1, \#8
sub r2, r1, r3
bnez r 2 , LOOP

Problem 2: When the MIPS program below starts register \$t0 holds the address of a string, the program converts the string to upper case.
(For MIPS documentation see http://www.ece.lsu.edu/ee4720/mips32v1.pdf and http://www.ece.lsu.edu/ee4720/mips32v2.pdf. Here are the relevant differences with DLX: branches and jumps are delayed ( 1 cycle). Some branch instructions compare two registers. Register \$0 works like DLX ro.)

LOOP:

```
lbu $t1, O($t0)
addi $t0, $t0, 1
beq $t1, $0, DONE
slti $t2, $t1, 97 # < 'a'
bne $t2, $0 LOOP
slti $t2, $t1, 123 # 'z' + 1
beq $t2, $0, LOOP
addi $t1, $t1, -32
j LOOP
sb $t1,-1($t0)
```

DONE:

Convert the program to IA-64 assembly language using predicated instructions. (You're not expected to know it at this point.) IA-64 is described in the IA-64 Application Developer's Architecture Guide, available at http://www.ece.lsu.edu/ee4720/ia-64.pdf.

For this problem one can ignore alot of IA-64's features. Here is what you will need to know: IA-64 has 64 1-bit predicate registers, p 0 to p 63 , which are written by cmp (compare) and other instructions. Predicates can be specified for most instructions, including cmp itself. See 11.2.2 for a description of how to use IA-64 predicates.

To solve the problem look at the following sections: 9.3, 9.3.1, and 9.3.2 (a brief description of where to place stops); 11.2.2 (predicate description and some more information on stops); and Chapter 7 (for instruction descriptions). The following instructions will be needed: cmp (compare, look at the normal [none] type) br (branch), load, store, and add.

- Use general-purpose registers r0-r31 and predicate registers p1-p63 in your solution. (There are 128 general-purpose registers, but those above r31 must be allocated.)
- Minimize the number of instructions per iteration assuming about half the characters are lower case.
- Use predicates to eliminate some branches.
- Make use of post-increment loads or stores.
- Pay attention to data type sizes.
- Show stops but do not show bundle boundaries.

Solution on next page.

```
// Solution
LOOP:
    ld1 r1 = [r2];;
    cmp.eq p3,p4 = r0,r1
    cmp.le p1,p2 = 97,r1;; // p1 = r1 >= 97; p2 = !p1 = r1 < 97
    (p1) cmp.ge p1,p2 = 122,r1;; // p1 = r1 <= 122; p2 = r1 > 122
    (p1) add r1 = -32, r1;;
    (p4) st1 [r2],1 = r1
    (p4) br LOOP;;
        br DONE
```

Problem 1: Solve Problems 3 and 4 from Fall 2000 Homework 5, available via http://www.ece.lsu.edu/ee4720/2000f/hw05.pdf. Using the solutions at http://www.ece.lsu.edu/ee4720/2000f/hw05_sol.pdf assign yourself a grade in the range [0, 1]. Either: indicate the grade you assigned yourself or write "Did not solve." A solution can be provided along with a grade. It will be corrected but your grade will be used. If you opt not to solve it you will receive full credit but will be hurting your ability to solve future problems.

For the following questions read Kenneth C. Yeager, "The Mips R10000 Superscalar Microprocessr," IEEE Micro, April 1996, pp. 28-40. A restricted-access copy can be found at
http://www.ece.lsu.edu/ee4720/s/yeager96.pdf. Access is allowed from within the lsu.edu domain or by using the userid "ee4720" and the correct password. Though not needed for this assignment, information on the MIPS64 4 ISA (implemented by R10000) can be found in
http://www.ece.lsu.edu/ee4720/mips64v1.pdf and http://www.ece.1su.edu/ee4720/mips64v2.pdf.
Skip over the material on the memory system (under heading "Memory Hierarchy") and the system interface. Material related to memory will be covered later in the semester.

Problem 2: The paper uses the four terms below, for each show the corresponding, or most similar, term used in class.

- Graduate $\longrightarrow$ Commit
- Active List $\longrightarrow$ Reorder Buffer
- Tag $\longrightarrow$ Reorder Buffer Entry Number
- Logical Register $\longrightarrow$ Architecturally visible register number.

Problem 3: For the superscalar processors described in class taken branches resulted in higher than ideal CPI; the higher the fetch/decode width (the $n$ in $n$-way superscalar) the worse the problem was. Why is this problem not as severe in the R10000? (Branch prediction is not the answer.)

Because a group of fetched instructions does not have to be 16 -byte aligned (that is, the address of the first instructions need not be a multiple of 16 ).

Problem 4: The MIPS R10000 does not have anything like a commit register map or a commit free list. (The register map and free list at the bottom of the figure from the class notes on the next page.) How were those used with exceptions in the Method-3 dynamically scheduled processor described in class? How does the R10000 deal with exceptions given their absence? Do not describe the entire exception process, just those pieces of hardware and steps needed to do what was done with the commit free list and register map.

In the implementation described in class, when the faulting instruction reaches the head of the reorder buffer the reorder buffer is flushed and the commit register map and free list are copied to the ID register map and free list, respectively.

In the R10000 the active list holds previous physical register assigned to the destination. When the faulting instruction reaches the head of the active list, rather than flushing the active list, the hardware uses the elements in the active list to repair the register map. For each active list element starting with the one holding the faulting instruction, the register map element corresponding to the destination register is written with the previous physical register. This processes reverses the changes to the register map made by the faulting instructions and those that followed it.

Problem 5: Figure 5, reproduced below, shows the information that will be placed in the active list and floating-point queue for an instruction being decoded. Various field names are shown along the bottom of the figure. The instruction format fields are shown at the top. Fields fR, fS, fT are source registers (not every instruction uses three); field $f D$ is the destination register, FLTX is the opcode, and MADD is an extension of the of the opcode field (as func is in DLX). (The figure appears to be using field values rather than names for the first and last fields. MADD is the name of an instruction, multiply-add, and FLTX may be an abbreviation for floating-point extended, though the architecture manual calls the field value COP1X. They probably should have used opcode instead of FLTX and function instead of MADD.)


Write the field names from the bottom of Figure 5 next to the corresponding fields in the figure, from the class notes, below. Though Figure 5 shows a floating-point instruction assume that integer instructions are handled the same way. Some fields have no analog in the figure below, these can be omitted; Tag is not a field that can be omitted.

Field names shown in blue


Problem 1: An ISA has a character size of $c=9$ bits (one more than most other ISA's!) and a 30 -bit address space $(A)$. An implementation has a bus width of $w=72$ bits and has no cache. Show how $2^{20} \times 36$ memory devices can be connected to implement the entire address space for this implementation. Show only the connections needed for loads. Show the alignment network as a box. Label inputs and outputs and be sure to specify which address bits are being used. The solution will require many memory devices so use ellipses $(\cdots)$ between the first and last of a large group of items.

See the figure below. The 128 -input multiplexor is a bit large for a real system. When one of that many inputs need to be selected a bus would be used, and pernaps several levels of selection.


Problem 2: The program below computes the sum of an array of doubles and also computes the sum of the characters in the array. The system uses a direct-mapped cache consisting of 1024 lines with a line size of 256 bits.

```
void p3(double *dstart, double *dend)
{
    double dsum = 0.0;
    int csum = 0;
    double *d = dstart; // sizeof(double) = 8 characters
    unsigned char *c = (unsigned char *) dstart; // A character is 8 bits.
    unsigned char *cend = (unsigned char *) dend;
    int dlength = dend - d;
    int clength = cend - c;
    while( d < dend ) dsum += *d++;
    if( ! LAST_PART ) flush_the_cache(); // Removes all data from the cache.
    while( c < cend ) csum += *c++;
}
```

When the procedure is called none of the data in the array is cached. When answering the questions below consider only memory accesses needed for the array (double or character). Assume that the number of iterations is some convenient number, except zero of course.

Note: Though they both access the same amount of data the number of iterations of the two loops are different. The first while loop is equivalent to: for ( $\mathrm{i}=0$; $\mathrm{i}<\mathrm{dlength} ; \mathrm{i}++$ ) dsum $=$ dsum + dstart[i];
(a) What is the hit ratio for the first while loop? Assuming the cache is flushed (emptied) between the two while loops, what is the hit ratio for the second while loop?

Line size is 256 bits, which is $\frac{256}{8}=32$ characters or $\frac{32}{8}=4$ doubles. The first loop sequentially loads doubles; the first access to a double on a line will miss, the next three will hit, and so on. The hit ratio for the first loop is thus $\frac{3}{4}=.75$. The second loop sequentially loads characters, the first access to a character on a line will miss, the subsequent 31 will hit, followed by 1 miss, etc. The hit ratio is thus $\frac{31}{32}=.96875$.
(b) Consider a single-issue (one-way) statically scheduled system in which the pipeline stalls on a cache miss. The cache miss delay is 1000 cycles. Roughly how does the time needed to execute the two loops compare? Assume that when there's a cache hit the time needed for one iteration is the same for both loops.

The first loop iterates dlength times, and so it will encounter $\frac{1}{4}$ dlength misses. The total time waiting for misses will be $1000 \frac{1}{4} \mathrm{~d}$ length cycles.

The second loop iterates clength times, and so it will encounter $\frac{1}{32}$ clength misses. The total time waiting for misses will be $1000 \frac{1}{32}$ clength cycles.

Though there is no way to determine how large dlength and clength are from the code above there relative sizes can be determined: clength $=8$ dlength, because a double is eight characters.

Substituting, the second loop spends $1000 \frac{1}{32} 8$ dlength cycles waiting for misses, the same as the first loop.
Assuming the time needed to execute instructions is small compared to the time spent waiting for misses, the two loops take about the same amount of time.
(c) Consider a single-issue (one-way) dynamically scheduled system with perfect branch and branch target prediction, a non-blocking cache, and a reorder buffer that can hold sixteen iterations of the while loops. The miss delay is still 1000 cycles however assume that for cache misses there is an initiation interval of one cycle so that the data for misses at $t=0$ and $t=1$ will arrive at $t=1000$ and $t=1001$, respectively. Now how do the two loops compare?

Sixteen iterations of the first loop covers four lines. When the ROB fills the cache will be working on four misses, the system will stall for a bit less than 1000 cycles, the sixteen iterations will finish and the ROB will fill with the next 16 iterations. The total time spent waiting for misses here is about $1000 \frac{1}{16}$ dlength cycles, about one quarter the time in the statically scheduled system.

When the second loop encounters a miss the ROB will fill with the next 15 iterations, all of which access the line that has missed. As a result the cache will be working on only one miss at a time. The total time spent waiting for misses is therefore unchanged and so the first loop is about four times faster than the second.
(d) Suppose the cache is not flushed before the second while loop executes. What is the smallest value of dlength (dee, not cee) for which the hit ratio of the second loop is less than 1.0?

Since both loops access the same data, the second loop can potentially have a $100 \%$ hit ratio. The hit ratio of the second loop is less than $100 \%$ when a later iteration of the first loop replaces data brought in by an earlier iteration.

The cache capacity is $1024 \times 32=2^{10+5}=2^{15}$ characters. The corresponding number of doubles is $\frac{1}{8} 2^{15}=$ $2^{-3} 2^{15}=2^{12}$. If dlength $=2^{12}+1$ the last iteration of the first loop would replace the line in the cache loaded on the first iteration. As a result, the first iteration of the second loop would miss. (Were dlength $=2^{12}$ the second loop would not miss.)

Problem 3: The SPARC V9 program below adds an array of integers.
(See http://www.ece.1su.edu/ee4720/samv9.pdf for a description of SPARC V9.) Except for
prefetch these instructions (or similar ones) have been covered before. The prefetch instruction is used to avoid the type of cache misses suffered by the program in the previous problem. It is like a nop in that it does not modify registers or memory, however like a load instruction, it moves data into the cache. As used below it will fetch data that will be needed ten iterations later. The data will be moved in to the cache (if not already present) but not in to a register. Ten iterations later the $l d x$ instruction will move the data in to register \% 12. Unlike loads, prefetch instructions never raise an exception. If the address is invalid or there is another problem the prefetch instruction does nothing, so there is no danger in prefetching, say, past the end of an array.

Unlike for a load that misses the cache, a statically scheduled processor would not stall on a prefetch miss. (There'd be no point in that!)

```
! Reminder: In SPARC assembler the destination register is on the right side.
LOOP:
    ldx [%11], %12 ! Load extended word (64 bits, same size as reg)
    prefetch [80+%11], 1 ! Prefetch from address 40+%11, type 1
    add %11, 8, %l1
    subcc %14, %11, %g0 ! %g0 = %14 - %11. (%g0 is zero register.) Set cc.
    bpg LOOP,pt ! Branch if condition code >0, predict taken
    add %13, %12, %13 ! Branch delay slot.
```

(a) In the code above the prefetch distance is ten iterations. What is the problem with the distance being too large or too small?

If the distance is too low the data will arrive after it's needed. (The goal is to get it before it's needed.) There will be a miss, but the miss delay will not be as long because the data is on its way. If the distance is too large the prefetched data may be replaced before its accessed. The data would arrive in the cache and would have to wait a long time before being accessed. In that time the data can be replaced because of a miss with the same index (but a different tag).
(b) Suppose SPARC V9 did not have a prefetch instruction. Explain how ldxa could be used as a prefetch. Show a replacement for prefetch in the program above.

The ldxa and similar instructions include an address space identifier (ASI) which specifies which address space to load or store from. The ASI can be specified with an immediate or the \%asi register. See the architecture manual. Normal loads and stores use the ASI_PRIMARY address space. ldxa lets you specify a different one. A load from a particular address in two different address spaces may load from two different memory locations or may load the same memory location in different ways. For example, an ordinary load of an address, ldx [\% 11 ], \%12, would load an integer using big-endian ordering. But a load to the same address using the ASI_PRIMARY_LITTLE, ldxa [\%11] ASI_PRIMARY_LITTLE, \%12 loads an integer using little-endian ordering. Table 12 in the architecture manual lists some of the address spaces.

Hint: Think about the destination register and the ASI.
One of the alternate address spaces allows a load from the primary address space without risking faults. To prefetch use one of those loads and put the data in the zero register, g 0 .

```
ldxa [80+%11], ASI_PRIMARY_NOFAULT, %g0
```


## 70 Spring 2001 Solutions

## EE 4720

Homework 1 solution
Due: 7 February 2001
Problem 1: Write a DLX program to reverse a C-style string, as described below. The address of the start of the string is in r1. The string consists of a sequence of characters and is terminated by a zero (NULL). The string length is not stored anywhere, it can only be determined by looking for the NULL. Put the reversed string in memory starting at the address in r2. Be sure to terminate the reversed string.

```
! r1 holds address of first character of original string.
! r2 holds address of first character of reversed string.
! Strings end with a zero (NULL) character.
    add r3, r1, r0 ! Copy of r1
SIZE_LOOP:
    lb r4, O(r3)
    addi r3, r3, #1
    bnez r4, SIZE_LOOP
    subi r3, r3, #1 ! Change r3 to address of null.
REV_LOOP:
    sub r6, r3, r1
    beqz r6, EXIT
    subi r3, r3, #1
    lb r4, O(r3)
    sb 0(r2), r4
    addi r2, r2, #1
    j REV_LOOP
EXIT:
    sb O(r2), 0
```

Problem 2: The DLX program below copies a block of memory starting at address r1 to the address r3, the block is of length r2 bytes. The problem is it won't always work. Explain why not and fix the problem without unnecessarily increasing the number of loop iterations. (The program will be slower, except for special cases.) Be sure to modify the program, not a specification of what the program is supposed to do.

```
! r1 Start address of data to copy.
! r2 Number of bytes to copy.
! r3 Start address of place to copy data to.
LOOP:
    slti r4, r2, #4
    bnez r4, LOOP2
    lw r5, O(r1)
    sw 0(r3), r5
    addi r1, r1, #4
    addi r3, r3, #4
    subi r2, r2, #4
    j LOOP
LOOP2:
    beqz r2, EXIT
    lb r5, O(r1)
    sb 0(r3), r5
    addi r1, r1, #1
    addi r3, r3, #1
    subi r2, r2, #1
    j LOOP2
```

EXIT:

It won't work if either the source or target addresses is unaligned and at least one word copy is attempted.

```
! r1 Start address of data to copy.
! r2 Number of bytes to copy.
! r3 Start address of place to copy data to.
    andi r4, r1, #3
    andi r5, r3, #3
    subi r6, r4, r5
    bnez r6, LOOP2 ! Word alignment of source and destination are different.
    slti r7, r2, r4
    bnez r7, LOOP2 ! Data ends before next aligned address.
    j LOOPBENTER
LOOPB: ! Copy until both addresses are word-aligned..
lb r5, 0(r1)
sb 0(r3), r5
addi r1, r1, #1
addi r3, r3, #1
subi r2, r2, #1
```

```
LOOPBENTER:
    andi r4, r3, #3 ! r4 is zero if r3 is word-aligned.
    bnez r4, LOOPB
LOOP:
    slti r4, r2, #4
    bnez r4, LOOP2
    lw r5, 0(r1)
    sw 0(r3), r5
    addi r1, r1, #4
    addi r3, r3, #4
    subi r2, r2, #4
    j LOOP
LOOP2:
    beqz r2, EXIT
    lb r5, 0(r1)
    sb 0(r3), r5
    addi r1, r1, #1
    addi r3, r3, #1
    subi r2, r2, #1
    j LOOP2
EXIT:
```

Problem 3: Implement the following procedure in DLX assembly language. The procedure is given two ways, both do the same thing, look at either one. The return address is stored in r31. The C short int data type here is two bytes (as it is on many real systems). The registers used for the procedure arguments are specified by the C variable names.

```
    void sum_arrays(short int *s_r1, float *f_r2, double *d_r3, int size_r4)
{
    while( size_r4-- ) *d_r3++ = *s_r1++ + *f_r2++;
}
void sum_arrays(short int *s_r1, float *f_r2, double *d_r3, int size_r4)
{
    int i;
    for(i=0; i<size_r4; i++) d_r3[i] = s_r1[i] + f_r2[i];
}
```

```
    j TEST
```

LOOP:
lh r6, $O(r 1)$
addi r1, r1, \#2
movi2fp f6, r6
cvti2d f6, f6
lf $\mathrm{f} 8,0(\mathrm{r} 2)$
addi r2, r2, \#4
cvtf2d f8, f8
addd f10, f6, f8
sd $0(r 3), f 10$
addi r3, r3, \#8
subi r4, r4, \#1
TEST:
bnez r4, LOOP

Problem 4: The code below contains two sets of add instructions, one in DLX assembler, the other in Compaq (née DEC) Alpha assembler. The first instruction in each group adds two integer registers, the second instruction in each group adds an integer to an immediate, the last adds two floating point registers. Information on the Alpha architecture can be found in the Alpha Architecture Handbook, http://www.ee.lsu.edu/ee4720/alphav4.pdf. It's 371 pages, don't print the whole thing.

```
! DLX Assembly Code
    add r1, r2, r3 ! r1 = r2 + r3
    addi r4, r5, #6
    addf f0, f1, f2
! Alpha Assembly Code (Destination is last operand.)
    addq r2, r3, r1 ! r1 = r2 + r3
    addq r5, #6, r4
    addt f1, f2, f0
```

Though the DLX and Alpha instructions are similar they are not identical.

- How do the data types and immediates differ between the corresponding DLX and Alpha
instructions?
The DLX integers are 32 bits, the Alpha integers are 64 bits, a size DLX does not have. The DLX floating-point data type used is IEEE 754 single ( 32 bits), the Alpha floating-point values are IEEE 754 double ( 64 bits), a type DLX does have. The DLX immediate size is 16 bits, the Alpha immediate size is 8 bits.
- Show the coding for the DLX and Alpha instructions above. Show the contents of as many fields as possible. For DLX, the addi opcode is 1 . The add func field is 0 and the addf func field is $1 d_{16}$. For the Alpha fields, see the Alpha Architecture Manual and use the following information: The Trapping mode should be imprecise and the Rounding mode should be Normal. (Trapping [raising an exception] will be covered later in the semester.)
DLX:
add r1, r2, r3

| opcode |  | rs1 $\rightarrow$ r2 |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 0 |  | 2 |  | 3 |  | 1 |  | 0 |
| 0 | 56 | 6 | 10 | 11 | 15 | 16 |  | 21 | 31 |

addi r4, r5, \#6


Alpha:
addq $r 2$, r3, r1

| opcode | $\mathrm{Ra} \rightarrow \mathrm{r} 2$ | $\mathrm{Rb} \rightarrow \mathrm{r} 3$ | SBZ |  | 1 |  | fune |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0x10 | 2 | 3 |  | 0 |  | 0 |  | 0x20 |  | 1 |
| 3126 |  | - 16 |  |  |  |  |  |  |  | 0 |

addq r5, \#6, r4

| opcode | $\mathrm{Ra} \rightarrow \mathrm{r} 5$ | LIT $\rightarrow 6$ |  |  | func |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0x10 | 5 |  | 6 | 1 |  | 0x20 |  | 4 |
| $31 \quad 26$ | $25 \quad 21$ |  |  | 12 |  |  |  | 0 |

addt f1, f2, f0

| opcode | Fa $\rightarrow$ f1 F | $\mathrm{Fb} \rightarrow \mathrm{f} 2$ |  | $\mathrm{Fc} \rightarrow \mathrm{fo}$ |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $0 \times 16$ | 1 | 2 |  |  |  |  |
| 3126 | $25 \quad 2120$ | $20 \quad 16$ | 15 |  | 4 | 0 |

- How do the approaches used to specify the immediate version of an integer instruction differ?

In DLX immediate variants of integer instructions use a different instruction format. In Alpha the same format is used, the immediate variant is indicated by setting an immediate bit, 12.

- How is the approach used to code floating-point instructions different in Alpha than DLX?

In DLX floating-point and three-register integer instructions share the same format. In Alpha they use a different format.

Problem 1: Translate the following C program to DLX assembly, use the minimum number of comparison instructions. Pay attention to data type sizes. The line labels are provided for convenience, please use them in the assembly language version.

```
    extern int r1, r2, r3, r10, r11;
    extern int *r20, *r21;
    /* For DLX: sizeof(int) = sizeof(int*) = 4 */
    /* For IA-64: sizeof(int) = sizeof(int*) = 8 */
```

    if( r1 < 3 )
        \{
        LINE1:
            if ( r2 == r3 )
                \{
                LINE11: r10 = *r20++;
                \}
            else
                \{
                LINE10: r10 = 4720;
                \}
    LINE1E:
        r11 = r11 + r10;
    \}
    else
    \{
    LINEO:
        r21 = r21 + 7;
        if ( r2 == r3 )
            \{
                LINE01: r10 = *r21++;
            \}
        else
            \{
            LINE00: r10 = 7700;
            \}
    \}
    DONE:
!! DLX
slti r8, r1, \#3
seq r9, r2, r3
beqz r8, LINE0
beqz r9, LINE10
LINE11:
lw r10, $0(r 20)$
addi r20, r20, \#4
j LINE1E
LINE10:

```
    addi r10, r0, #4720
LINE1E
    add r11, r11, r10
    j DONE
LINE0:
        addi r21, r21, #28
        beqz r9, LINE00
LINE01:
    lw r10, O(r21)
    addi r21, r21, #4
    j DONE
LINE00:
    addi r10, r0, #4720
DONE:
```

Problem 2: Translate the C program from the previous problem into IA-64 assembly using predicated instructions. (You're not expected to know it at this point.) IA-64 is described in the IA-64 Application Developer's Architecture Guide, available at http://developer.intel.com/design/ia64/downloads/adag.pdf.

For this problem one can ignore alot of IA-64's features. Here is what you will need to know: IA-64 has 64 1-bit predicate registers, p0 to p63, which are written by cmp (compare) and other instructions. Predicates can be specified for most instructions, including cmp. See 11.2.2 for a description of how to use IA-64 predicates.

To solve the problem look at the following sections: 11.2.2 (predicate description) and Chapter 7 (for instruction descriptions). The following instructions will be needed: cmp (compare, look at the normal [none] and unc comparison types), ld1, ld2,... (loads), and add.

To save time, ignore instruction stops (;;) and consider only normal loads. (Post-increment like loads are considered normal here.)

- Use general-purpose registers r0-r31 and predicate registers p1-p63 in your solution. (There are 128 general-purpose registers, but those above r31 must be allocated.)
- Do not use branches (or any other CTI).
- Ignore stops. (These will be covered later.)
- Use the minimum number of cmp instructions. (Three is possible.)
- Do not assign a value to a register unless it's needed.
- Make use of post-increment loads.
- Pay attention to data type sizes.

```
!! IA-64
    cmp.gt p1,p2 = 3,r1
(p1) cmp.eq.unc p3,p4 = r2,r3
(p2) cmp.eq.unc p5,p6 = r2,r3
(p3) ld8 r10 = [r20],4
(p4) adds r10 = 4720,r0
(p1) add r11 = r11,r10
(p2) add r21 = 56,r21
(p5) ld8 r10 = [r21],4
(p6) adds r10 = 7700,r0
```

Problem 3: Show a pipeline execution diagram of the code below on each implementation. (There should be a total of two diagrams.) The branch is always taken, show the diagram until the second execution of the first instruction reaches WB. If a bypass path is not shown, it's not there. LOOP:

```
addi r2, r2, #4
lw r1, 0(r2)
add r3, r3, r1
slt r4, r2, r5
beqz r4, LOOP
xor r5, r4, r1
```



```
! Solution
LOOP:
! Cycle: 0
addi r2, r2, #4 IF ID EX ME WB IF ID
lw r1, O(r2) IF ID ----> EX ME WB IF
add r3, r3, r1 IF ----> ID ----> EX ME WB
    slt r4, r2, r5 IF ----> ID EX ME WB
    beqz r4, LOOP
    xor r5, r4, r1
    IF ID ----> EX ME WB
    IF ----> IDx
```



```
! Solution
LOOP:
! Cycle: 0
    addi r2, r2, #4 IF ID EX ME WB IF ID EX
lw r1, O(r2) IF ID EX ME WB IF ID
add r3, r3, r1 IF ID -> EX ME WB
slt r4, r2, r5 IF -> ID EX ME WB
beqz r4, LOOP
xor r5, r4, r1
    IF ID ----> EX ME WB
        IF ---->x
```

Problem 4: For each implementation from the problem above, determine the CPI for a large number of iterations.

First implementation, average instruction execution time is $\frac{14-0}{5} \mathrm{CPI}=2.8 \mathrm{CPI}$. Second implementation, average instruction execution time is $\frac{9-0}{5} \mathrm{CPI}=1.8 \mathrm{CPI}$.

Problem 5: For the second pipeline execution diagram above, show the location(s) of the latest value of $r 1$ and $r 2$ at the beginning of each cycle on the diagram below. For r1 box the appropriate cycle numbers and draw an arrow to the locations. For r2 circle the cycle numbers and draw an arrow to the locations. In the diagram below this has been completed for cycles zero and two, assuming addi is in IF at cycle zero. The arrows should only point to register values that are valid at the indicated cycles. Note: A valid value can be in more than one location at once.


Problem 1: Consider three variations on the Chapter-3 DLX implementation. In implementation I the FP Add unit has an initiation interval of 2 and a latency of 3 . In implementation II there are two FP Add units, each unit has an initiation interval of 4 and a latency of 3 . In implementation III the FP Add unit has an initiation interval of 1 and a latency of 3 . Other features of the implementations are identical. All implementations are fully bypassed.

Write two programs. Program $\mathcal{A}$ should run slower on implementation I than on implementations II and III. Program $\mathcal{B}$ should run the same speed on implementations I and II and faster on implementation III. For this problem base program speed on the time from the fetch of the first instruction to the WB of the last instruction.

Show pipeline execution diagrams for each program on each implementation. The programs need be no longer than four instructions each.

```
! Solution
! Program \mathcal{A}
! I
addd f0, f2, f4 IF ID A1 A1 A2 A2 WB
addd f6, f8, f10
    IF ID -> A1 A1 A2 A2 WB
! II
addd f0, f2, f4 IF ID A A A A WB
addd f6, f8, f10 IF ID B B B B WB
! III
addd f0, f2, f4 IF ID A1 A2 A3 A4 WB
addd f6, f8, f10 IF ID A1 A2 A3 A4 WB
! Program \mathcal{B}
! I
addd f0, f2, f4 IF ID A1 A1 A2 A2 WB
addd f6, f8, f10
addd f12, f14, f16
    IF ID -> A1 A1 A2 A2 WB
        IF -> ID -> A1 A1 A2 A2 WB
! II
addd f0, f2, f4 IF ID A A A A WB
addd f6, f8, f10
addd f12, f14, f16
    IF ID B B B B WB
        IF ID ----> A A A A WB
! III
addd f0, f2, f4 IF ID A1 A2 A3 A4 WB
addd f6, f8, f10 IF ID A1 A2 A3 A4 WB
addd f12, f14, f16 IF ID A1 A2 A3 A4 WB
```

Problem 2: Modify the pipeline below so that it can execute jr instructions and add PC mux control logic.

- Modify the pipeline so that it can execute jr instructions. (See Spring 1999 Homework 3, http://www.ee.1su.edu/ee4720/1999/hw03_sol.pdf.)
- Include control logic for the multiplexor that connects to PC. The control logic should correctly handle branch and jump instructions. Interrupts should be ignored. To recognize instructions use boxes such as $=\mathrm{bnez}$, the outputs will be 1 if the instruction matches.
- Show the logic for a squash signal for use in EX to squash the fall-through instruction on a taken branch. (The fall through instruction could have been squashed in IF or ID, but for this problem it will be squashed in EX.)


## Changes shown in red.



Problem 3: How would the hardware designed above have to be modified if DLX had two-slot (yes, two slots!) delayed branches? Jumps still have no delay slots. Ignore interrupts, they will be considered in the next problem.

Problem 4: In an ISA without delayed branches it would be sufficient for the hardware to save the PC when an exception occurs. Why would this not be sufficient on a system with delayed branches. Provide an example illustrating what might go wrong.

One could not properly resume execution if the faulting instruction were in a branch delay slot. To resume execution properly the exception handler needs the PC of the faulting instruction and the PC of the next instruction to execute. In most cases the next instruction to execute is at $\mathrm{PC}+4$ (assuming four-character instructions) but if the faulting instruction were in the delay slot of a taken branch the next instruction to execute would be the branch target.

Suppose that 1 w raises an exception in the example below. If the handler only saves the address of $1 \mathrm{w}, 0 \mathrm{x} 1004$, then when execution resumes the branch will not be taken. By saving $0 x 1004$ and the address of the next instruction, 0x2000, the handler can restore execution so that the branch will be taken.

```
0x1000: beqz r0, TARGET
0x1004: lw r2, 0(r3)
0x1008: add r3, r3, r4
TARGET:
    0x2000: or r5, r6, r7
```

Problem 5: The Hewlett Packard Precision Architecture RISC 2.0 (PA-RISC 2.0) uses an instruction address offset queue rather than a plain-old program counter. See the PA-RISC 2.0 Architecture [Manual], http://devresource.hp.com/devresource/Docs/Refs/PA2_0/acd-1.html. Ignore the material on [address] space IDs and privilege levels. Concentrate on the material in Chapter 4 and 5 and use the index.

PA-RISC 2.0 has delayed branches. Explain how the use of an instruction address offset queue rather than a PC helps with the difficulty alluded to in the previous problem.

The address of the executing instruction and the next instruction can be saved and restored as a unit.
Problem 6: Explain the relationship between the terms interrupt, hw interrupt, exception, and trap provided in class and the terms interruption, fault, interrupt, trap, and check defined for PA-RISC 2.0. Explain the relationships, do not simply provide definitions.

A PA-RISC interruption is analogous to the term interrupt used in class.
A PA-RISC fault is a category of exception, as used in class. A PA-RISC trap is another category of exception, as used in class. That is, some of what are called exceptions in class are called faults in PA-RISC, and other exceptions are called traps. (Traps are usually due to programmer error or bad input, while faults indicate that the OS has to take some routine action to keep the program running.) Note that the meaning of the term trap used in class is completely different from a PA-RISC trap.

A PA-RISC interrupt is analogous to the term hardware interrupt used in class.
A PA-RISC check is a specific type of hardware interrupt, no special term was used in class.
Problem 7: Name a difference between the trap table used in Sun SPARC V8 (presented in class and described in the SPARC Architecture Manual V8, http://www.ee.lsu.edu/ee4720/sam.pdf) and the interruption vector table used in PA-RISC 2.0.

The Sun SPARC table holds four instructions, the PA-RISC table holds eight instructions, otherwise they are very similar.

Problem 1: Complete a pipeline execution diagram for the following code running on a two-way statically scheduled superscalar processor. Show execution until the second fetch of the first add. The processor fetches instructions in aligned groups and is fully bypassed. The branch will be taken. There is no branch prediction hardware.

What is the CPI for a large number of iterations?

```
    ! Solution
LOOP: ! LOOP = 0x1004
    ! Cycle 0
    add r1, r2, r3 IF ID EX ME WB IF ID
    add r4, r5, r6 IF ID EX ME WB
    add r9, r4, r7 IF ID -> EX ME WB
    lw r10, 0(r4) IF -> ID EX ME WB
    add r11, r11, r10 IF -> ID ----> EX ME WB
    or r12, r11, r13 IF ----> ID EX ME WB
    xor r15, r16, r17 IF ----> ID EX ME WB
    bnez r10, LOOP IF ID EX ME WB
```

        The CPI for a large number of iterations is \(\frac{9}{8}\).
    Problem 2: Schedule the code from the problem above so that it executes efficiently. The solution can contain added nop instructions. Do not try to unroll the loop. A correct solution contains two stalls plus the branch delay.

Now what is the CPI for a large number of iterations?

```
! Solution
nop ! nop inserted to align first and last loop instructions.
LOOP: ! LOOP = 0x1008
    ! Cycle 0
    add r4, r5, r6 IF ID EX ME WB IF ID EX
    add r1, r2, r3 IF ID EX ME WB IF ID EX
    add r9, r4, r7 IF ID EX ME WB IF ID
    lw r10, O(r4) IF ID EX ME WB IF ID
    add r11, r11, r10 IF ID -> EX ME WB
    xor r15, r16, r17 IF ID -> EX ME WB
    or r12, r11, r13 IF -> ID EX ME WB
    bnez r10, LOOP IF -> ID EX ME WB
```

The CPI for a large number of iterations is $\frac{6}{8}=0.75$.

Problem 3: Show the execution of the code from Problem 1 on a two-way superscalar dynamically scheduled machine using Method 1. The number of reservation stations, functional units, and reorder buffer entries is unlimited. Do not show reservation station numbers or reorder buffer entry numbers in the diagram. Do show where instructions commit. Assume that the machine has perfect branch and branch target prediction and so a branch target will be fetched when the branch is in ID. Complete the diagram to the point where all instructions in the first iteration commit, showing what happens to instructions in the second iteration up to that point.

Now what is the CPI for a large number of iterations?

```
    ! Solution
LOOP: ! LOOP = 0x1004
```



```
    add r1, r2, r3 IF ID EX WC IF ID EX WB C IF
    add r4, r5, r6 IF ID EX WC IF ID EX WB C
    add \(r 9, r 4, r 7\) IF ID RS EX WC
                                IF ID EX WB C
    lw r10, O(r4) IF ID L1 L2 WC
                            IF ID L1 L2
add r11, r11, r10 IF ID RS EX WC
                            IF ID RS
    or r12, r11, r13 IF ID RS EX WC
                                IF ID RS
    xor r15, r16, r17
    IF ID EX WB C
    bnez r10, LOOP
        IF ID B WB C
                            IF ID
```

Because the iterations starting at cycles 5 and 10 start the same way (with corresponding previous instructions being in the same stages of execution) following iterations will take the same number of cycles and so the second (or any following) iteration (starting at cycle 5) can be used to compute the CPI. The CPI is $\frac{5}{8}$.

More problems on the next page.
$\rightarrow \mathrm{H}$

Problem 4: Convert the code below to VLIW DLX as described in the notes. The maximum lookahead value is 15 , use that for bundles that do not modify any registers. Set the lookahead values and serial bits for maximum performance. (The lookahead values will mostly be small.) How would a modification of the end-of-loop test improve performance on a VLIW implementation?

```
    j TEST
LOOP:
    lw r1, O(r10)
    lw r2, 4(r10)
    lw r3, 8(r10)
    lw r4, 12(r10)
    andi r1, r1, #15
    andi r2, r2, #15
    andi r3, r3, #15
    andi r4, r4, #15
    sw O(r10), r1
    sw 4(r10), r2
    sw 8(r10), r3
    sw 12(r10), r4
    addi r10, r10, #16
TEST:
    slt r11, r10, r12
    bnez r11, LOOP
    ! Solution
    { s 15
        j TEST
        nop
        nop }
LOOP:
{ p 0
    lw r1, O(r10)
    lw r2, 4(r10)
    lw r3, 8(r10)
}
{ p 0
    lw r4, 12(r10)
    andi r1, r1, #15
    andi r2, r2, #15
}
{ p 0
    andi r3, r3, #15
    andi r4, r4, #15
    sw O(r10), r1
}
{ p 15
    sw 4(r10), r2
    sw 8(r10), r3
    sw 12(r10), r4
}
```

```
\{ p 0
    addi r10, r10, \#16
    nop
    nop
\}
TEST:
\{ s 15
    slt r11, r10, r12
    bnez r11, LOOP
    nop
\}
! Assuming that r11 not referenced on fall-through path.
```

Problem 5: Insert the minimum number of IA-64-style stops in the DLX code below. Do not convert the instructions themselves to IA-64, just insert the stops.

The material on stops was covered in class and will be in the notes. A primary reference is Appendix A of the IA-64 Application Developer's Architecture Guide, available at http://developer.intel.com/design/ia64/downloads/adag.pdf. Appendix A describes how stops affect the execution of code.

```
    j TEST
```

LOOP:
lw r1, $0(r 10)$
lw r2, 4(r10)
lw r3, 8(r10)
lw r4, 12(r10)
andi r1, r1, \#15
andi r2, r2, \#15
andi r3, r3, \#15
andi r4, r4, \#15
sw $0(r 10), r 1$
sw 4(r10), r2
sw 8(r10), r3
sw 12(r10), r4
addi r10, r10, \#16
TEST:
slt r11, r10, r12
bnez r11, LOOP
! Solution
j TEST
LOOP:
lw r1, $0(r 10)$
lw r2, 4(r10)
lw r3, 8(r10)
lw r4, 12(r10) ; ;
andi r1, r1, \#15
andi r2, r2, \#15
andi r3, r3, \#15
andi r4, r4, \#15 ; ;
sw $0(r 10), r 1$
sw 4(r10), r2
sw 8(r10), r3
sw 12(r10), r4
addi r10, r10, \#16 ;;
TEST:
slt r11, r10, r12 ; ;
bnez r11, LOOP

## 71 Fall 2000 Solutions

Problem 1: Find the SPECint2000 results for the API UP2000 750 MHz processor, it can be found at the http://www.spec.org web site. This processor has a SPECint2000 rating of 456. Find another processor with a slower rating but for which individual benchmarks are faster. (Look for different CPU families.) How many of the benchmarks are faster on the slower processor?

Problem 2: Write a DLX assembly language program to convert a string of characters to lower case. The string is NULL-terminated (the character following the end of the string is a zero). Register r1 contains the address of the start of the string. Any register can be modified. The code for an upper-case A is 65 and the code for a lower-case a is 97 . Modify the string, do not create a new one.

```
! ** Solution **
!
! Register r1 contains address of first character of string.
LOOP:
lbu r2, 0(r1)
beqz r2, DONE
slti r3, r2, #65
bneq r3, CONTINUE
sgti r3, r2, #90
bneq r3, CONTINUE
addi r3, r3, #32
sb 0(r1), r3
CONTINUE:
addi r1, r1, #1
j LOOP
DONE:
```

Problem 3: Write a DLX assembly language program that loads an element of a two-dimensional array to a register.

Register r 1 holds address of the start of the array, register r 2 holds the row of the element to retrieve, and register r3 holds the column of the element to retrieve. Put the retrieved element in f0. The array dimensions are 256 rows $\times 1024$ columns. Each element of the array is a double precision floating point number.

Elements are arranged in memory in the following order:

$$
a_{0,0} a_{0,1} a_{0,2} \cdots a_{1,0} a_{1,1} a_{1,2} \cdots a_{2,0} \cdots
$$

where $a_{i, j}$ is the element at row $i$, column $j$.

```
! ** Solution **
!
! r1: address of the start of the array.
! r2: row of element to retrieve.
! r3: column of element to retrieve.
! Put element in f0.
! Array dimensions are 256 rows x }1024\mathrm{ columns
! Each element of the array is a double precision floating point number.
! Elements are arranged in memory in the following order
    a_{0,0} a_{0,1} a_{0,2} ... a_{ {1,0} a_{1,1} a_{1,2} ... a_{2,0} ...
where a_{i,j} is the element at row i and column j.
slli r4, r2, #10
or r4, r4, r3
slli r4, r4, #3
add r4, r4, r1
ld f0, O(r4)
```


## EE 4720

Homework 2 solution Due: $\mathbf{2 2}$ September 2000
Problem 1: Compare the coding of the DLX instructions:

```
add r1, r2, r3
addi r4, r5, #6
```

to the corresponding Sun SPARC V8 instructions:

```
add %g3, %g2, %g1 ! g1 = g2 + g3
add %g5, 6, %g4 ! g4 = g5 + 6
```

The definition of the SPARC V8 architecture is available via
http://www.ee.lsu.edu/ee4720/sam.pdf or http://www.sparc.com/standards/V8.pdf. Hint: The information needed to solve the problem is in Appendix B.

How are the approaches used to code immediate variants of the add instructions different in the two ISAs?

In DLX the immediate variant of the add uses a different format than the two-source-register version (Type I vs. Type R). In SPARC V8 the immediate and two register adds use the same instruction type and used the same opcode, they are distinguished by a single-bit i field in the instruction word.

Problem 2: DLX does not have indexed addressing nor does it have autoincrement addressing. Suppose one wanted to include those addressing modes in an extended version of DLX, call it DLX-BAM (better addressing modes). The addressing modes would be used in load and store instructions. Show how they would best be coded, where the fewer changes to the coding structure the better. (For example, adding a fourth instruction type [say Type-A], would be a big change and so would be bad.) Sample mnemonics for these instructions appear below:

```
! Indexed addressing.
lw r1, (r2+r3) ! r1 = MEM[ r2 + r3 ];
sw (r2+r3), r4 ! MEM[ r2 + r3 ] + r4;
! Autoincrement addressing.
lb r1, 3(+r2) ! r1 = MEM[ r2 + 3 ]; r2 = r2 + 1;
lw r4, 8(+r5) ! r4 = MEM[ r5 + 8 ]; r5 = r5 + 4;
sw 4(+r7), r8 ! MEM[ r7 + 4 ] = r8; r7 = r7 + 4;
```

The indexed load has two source operands and a destination, so it is natural to code it as a type R instruction. The indexed store has three source operands, but like the displacement store, one of the operands can be placed in the rd position, and so the indexed store can also be coded as a type R instruction.

A poor solution would be to code using a modified type I format, call that type Im. The new register number would be placed in the immed field. This solution is poor because it adds a new type (when type R is perfectly suitable.

Problem 3: Write a C program that does the same thing as the DLX program below.

```
! r2: Start of table of indices, used to retrieve elements
! from the character table.
! r4: Start of table of characters.
! r6: Location to copy characters to.
! r8: Address of end of index table.
LOOP:
    lw r1, O(r2)
    add r3, r1, r4
    lb r5, 0(r3)
    sb 0(r6), r5
    addi r2, r2, #4
    addi r6, r6, #1
    slt r7, r2, r8
    bneq r7, LOOP
```

        Solution template available via: http://www.ee.lsu.edu/ee4720/2000f/hw02.c
        There are two solutions below, a compact one, and one that's easier to understand.
    void
untangle(int $* r 2$, char $* r 4$, char $* r 6$, int $* r 8$ )
\{
do $\{$ *r6++ $=r 4[* r 2++] ;\}$ while ( $\mathrm{r} 2<\mathrm{r} 8$ );
\}
void
untangle_easy(int *r2, char *r4, char *r6, int *r8)
\{
do \{
int table_index $=* r 2$;
char $c=r 4\left[t a b l e \_i n d e x\right] ;$
*r6 = c;
/* Because r2 is declared int* the line below adds 4 (sizeof(int) = 4
on Solaris 2.6) to r2. */
r2 = r2 + 1;
/* Because r6 is declared char* the line below adds 1 (sizeof(char) = 1
probably by definition) to r6. */
r6 = r6 + 1;
\} while ( r2 < r8 ) ;
\}

Problem 4: Re-code the DLX program above using DLX-BAM, taking advantage of the new instructions.

```
LOOP:
```

    lw r1, 0(+r2)
    lb r5, (r1+r4)
    sb \(0(+\mathrm{r} 6)\), r5
    slt r7, r2, r8
    bneq r 7 , LOOP
    
## EE 4720

Homework 3 solution
Due: 2 October 2000
Problem 1: What changes would have to be made to the pipeline below to add the DLX-BAM indexed addressing instructions (from homework 2). Hint: The load is easy and inexpensive, the store requires a substantial change. Add the changes to the diagram below, but omit the control logic. Do explain how the control logic would have to be changed.

```
! Indexed addressing.
lw r1, (r2+r3) ! r1 = MEM[ r2 + r3 ];
sw (r2+r3), r4 ! MEM[ r2 + r3 ] + r4;
```

No datapath changes are needed to implement the indexed load. The control logic must recognize the new instruction type and use the A and B inputs to the ALU rather than the A and IMM that are used for ordinary loads.

The changes needed to implement the indexed store are shown in red bold below. A third read port is added to the register file in ID and a multiplexor is added to route either the ID/EX. B or the new ID/EX. C latch to the memory data in in the MEM stage. Control logic changes are similar to the indexed load, with the addition of control for the new multiplexor.


Problem 2: For maximum pedagogical benefit solve the problem above before attempting this one. The integer pipeline of the Sun Microsystems microSPARC-IIep implementation of the SPARC V8 ISA is similar to the Chapter-3 implementation of DLX that is being covered in class.

What are the stage names and abbreviations used in the microSPARC-IIep? Hint: This is really easy once you've found the right page.

SPARC V8 includes indexed addressing, for example:
ld $[\% \circ 3+\% \circ 0]$, $\%$ o2 ! Load word: $\%$ o2 $=$ MEM $[\% \circ 3+\% \circ 0$ ]
st $\% \circ 0$, $[\% \circ 1+\%$ g1] ! Store word: MEM[ $\% \circ 1+\%$ g1 ] = $\% \circ 0$
(Register $\% \circ 0$ is a real register, not a special zero register.) What are the differences between the micro-SPARC-IIep integer pipeline and the Chapter-3 DLX pipeline that allow it to execute an indexed store? Be sure to answer the question directly, do not copy or paraphrase irrelevant material. A shorter answer is preferred.

Information on the microSPARC-IIep can be found via http://www.sun.com/microelectronics/manuals/microSPARC-IIep/802-7100-01.pdf
or http://www.ee.lsu.edu/ee4720/microsparc-IIep.pdf. Those who enjoy a challenge can study the diagram on page 10 , however the material to answer the question can be found early in Chapter 3 . The
manual uses many terms which have not yet been covered in class, the question can still be answered once the right page is found. The manual is 256 pages so don't print the whole thing.

The register file has a third read port, used for store data. The store data is read in the E stage, rather than in D , as it would in the DLX implementation.

Problem 3: The following pipeline execution diagram shows the execution of a program on the DLX implementation shown below. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others; connections needed to implement the jalr instruction are not shown. A value can be read from the register file in the same cycle it is written. Instructions are squashed (nulled) in this problem by replacing them with or $\mathrm{r} 0, \mathrm{rO}, \mathrm{rO}$. All instructions stall in the ID stage.

Add the datapath connections needed so the jalr executes as shown.
Instruction addresses are shown below, to the left of the instructions.


Changes for jalr are show below in red bold. For the jalr instruction the ALU will pass through the top input unchanged. As an alternative, EX/MEM.NPC and MEM/WB.NPC registers could be included, with the output of MEM/WB.NPC going into the same multiplexor as MEM/WB. ALU and MEM/WB. MD. This would require two more registers, but those NPC registers might be needed for exception processing.


The table on the next page shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. The first two columns are completed; fill in the rest of the table. Use a "?" for the value of the "immediate field" of a type R instruction and for the output of the memory when no memory read is performed. Show pipeline register values even if they're not used. The row labeled "Reg.

Chng." shows a new register value that is available at the beginning of the cycle. If r0 is written leave the entry blank.

Hint: For hints and confirmation see Spring 1999 HW 3, Fall 1999 HW 2, and Spring 2000 HW 2, linked to http://www.ee.1su.edu/ee4720/prev.html, for similar problems. It's important that the problem is solved by inspection of the diagram, not by inferring mindless, unworthy-of-an-engineer rules from past solutions. Mindless rules are hard to remember and are useless in new situations.

Completed table appears below. The numbers in the table are in hexadecimal. An "x" after an instruction name indicates it nas been squashed.

| Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| PC | 50 | 54 | 58 | 50 | 50 | 60 | 68 | 60 |  |  |
| IF/ID. IR | sub | add | Iw | sw | sw | jalr | xor | andi | siti |  |
| Reg. Chng. | r0 $<0$ | ro $\leftarrow 0$ | r0 $\leftarrow 0$ | ro $<0$ | r2 $<500$ | r2 $\leftarrow$ aaa 0 | $\mathrm{r} 0 \leftarrow 0$ | ro $\leftarrow 0$ | $r 31 \leftarrow 60$ | r0 $<0$ |
| ID/EX.IR | sub | sub | add | Iw | swx | sw | jalr | xorx | andi | siti |
| ID/EX.A | 0 | 0 | 200 | 200 | 200 | 500 | 68 | 0 | 2 a20 | 300 |
| ID/EX.B | 0 | 0 | 300 | 200 | 100 | 100 | ? | 0 | $22 a 0$ | 300 |
| ID/EX.IMM | ? | ? | ? | 4 | 8 | 8 | ? | ? | 20 | 30 |
| EX/MEM.IR | sub | sub | sub | add | Iw | swx | sw | jalr | xorx | andi |
| EX/MEM.ALU | 0 | 0 | 0 | 500 | 504 | 300 | aaas | 60 | 0 | 20 |
| EX/MEM.B | 0 | 0 | 0 | 300 | 200 | 100 | 100 | ? | 0 | a 2 a 0 |
| MEM/WB.IR | sub | sub | sub | sub | add | Iw | swx | sw | jalr | xorx |
| MEM/WB.ALU | 0 | 0 | 0 | 0 | 500 | 504 | 300 | $2 a 28$ | 60 | 0 |
| MEM/WB.MD | ? | ? | ? | ? | ? | a 2 a 0 | ? | ? | ? | ? |

Problem 4: Draw a pipeline execution diagram showing the execution of the familiar code below until the second fetch of 1 w (the beginning of the second iteration). Hint: There are RAW hazards associated with the loads, stores, and the branch. What is the CPI for a large number of iterations?

```
LOOP:
    lw r1, O(r2)
    add r3, r1, r4
    lb r5, 0(r3)
    sb 0(r6), r5
    addi r2, r2, #4
    addi r6, r6, #1
    slt r7, r2, r8
    bneq r7, LOOP
    xor r10, r11, r12
```



The pipeline execution diagram appears below. The CPI is $\frac{14}{8}=1.75 \mathrm{CPI}$.
! Solution.
LOOP:
! Cycle
$\begin{array}{llllllllllllllllll}0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17\end{array}$
lw r1, O(r2)
IF ID EX ME WB IF ID EX ME
add $r 3$, r1, r4
IF ID $\rightarrow$ EX ME WB
IF ID ->
lb r5, 0(r3)
IF -> ID EX ME WB
IF ->
sb $0(r 6), r 5$
IF ID ----> EX ME WB
addi r2, r2, \#4
IF ----> ID EX ME WB
addi r6, r6, \#1
IF ID EX ME WB
slt r7, r2, r8
IF ID EX ME WB
bneq r 7 , LOOP
xor r10, r11, r12
IF ID ----> EX ME WB

Problem 5: Rearrange (schedule) the instructions in the program from the previous problem to minimize the number of stalls. Now what is the CPI for a large number of iterations? Hint: The offsets in the load and store instructions can be changed, even to negative numbers.

The pipeline execution diagram appears below. The CPI is $\frac{9}{8}=1.25 \mathrm{CPI}$.
LOOP:

| Cycle |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| lw r1, O(r2) | IF | ID | EX | ME | WB |  |  |  |  | IF |  |  |
| addi r2, r2, \#4 |  | IF | ID | EX | ME | WB |  |  |  |  |  |  |
| add r3, r1, r4 |  |  | IF | ID | EX | ME | WB |  |  |  |  |  |
| lb r5, 0(r3) |  |  |  | IF | ID | EX | ME | WB |  |  |  |  |
| slt r7, r2, r8 |  |  |  |  | IF | ID | EX | ME | WB |  |  |  |
| addi r6, r6, \#1 |  |  |  |  |  | IF | ID | EX | ME | WB |  |  |
| sb -1(r6), r5 |  |  |  |  |  |  | IF | ID | EX | ME |  |  |
| bneq r7, LOOP |  |  |  |  |  |  |  | IF | ID | EX |  |  |
| xor r10, r11, r12 |  |  |  |  |  |  |  |  | IF |  |  |  |

## EE 4720

Homework 4 Solution
Due: 3 November 2000
Problem 1: Show a pipeline execution diagram for the execution of the DLX program below on a single-issue statically scheduled (plain old chapter 3) fully bypassed implementation in which the add functional unit is two stages (A1, A2) with an initiation interval of 2 (latency 3) and the multiply unit is six stages (M1 through M6) with an initiation interval of 1 (latency 5). (This problem is very similar to Spring 2000 homework 3 problem 1. Check the solution to that assignment only if completely lost.)

```
! Solution
addd f0, f2, f4 IF ID A1 A1 A2 A2 WB
addd f6, f0, f8 IF ID -------> A1 A1 A2 A2 WB
addd f10, f12, f14 IF -------> ID -> A1 A1 A2 A2 WB
multd f16, f18, f20 IF -> ID M1 M2 M3 M4 M5 M6 WB
```

Problem 2: Show a pipeline execution diagram for the execution of the DLX program below on a single-issue statically scheduled fully bypassed implementation in which there are two add units, both consisting of one stage with an initiation interval of 4 (latency 3, unpipelined). Use symbol A for one adder and B for the other. The program below is slightly different than the one above.

```
! Solution
addd f0, f2, f4 IF ID A A A A WB
addd f6, f0, f8 IF ID -------> A A A A WB
addd f10, f12, f14 IF -------> ID B B B B WB
addd f16, f18, f20 IF ID ----> A A A A WB
```

Problem 3: Show a pipeline execution diagram for the execution of the DLX program below on a two-way superscalar statically scheduled fully bypassed implementation in which there are two add units, both consisting of one stage with an initiation interval of 4 (latency 3, unpipelined). Use symbol $A$ for one adder and $B$ for the other.

```
! Solution
LINE1: ! LINE1 = 0x1000
addd f0, f2, f4 IF ID A A A A WB
addd f6, f0, f8 IF ID ---------> A A A A WB
addd f10, f12, f14 IF --------> ID B B B B WB
addd f16, f18, f20 IF ---------> ID -------> A A A A WB
```

Problem 4: Show a pipeline execution diagram for the DLX code below executing on a processor with the following characteristics:

- Statically scheduled two-way superscalar.
- Unlimited number of functional units.
- Six stage fully pipelined multiply.
- Can handle an unlimited number of write backs per cycle. (Unrealistic, but reduces adidactic tedium.)
- Fully bypassed, including the branch condition.

The diagram should start at the first iteration and end after 30 cycles or until a repeating pattern is encountered, whichever is sooner. Note that there is a floating-point loop-carried dependency (f2). What is the CPI for a large number of iterations?

```
LOOP: ! LOOP = 0x1004
! Cycle 
ld f0, O(r1) IF ID EX ME WB IF ID EX ME WB IF ID EX ME WB IF ID EX ME WB
muld f2, f0, f2 IF ID -> M1 M2 M3 M4 M5 M6 WB
                                    IF ID -> M1 M2 M3 M4 M5 M6 WB
                                    IF ID -> M1 M2 M3 M4 M5 M6 WB
addi r1, r1, #8 IF ID EX ME WB IF ID EX ME WB IF ID EX ME WB
sub r2, r1, r3 IF -> ID EX ME WB IF -> ID EX ME WB IF -> ID EX ME WB
bneq r2, LOOP IF -> ID -> EX ME WB
                                    IF -> ID -> EX ME WB
                                    IF -> ID -> EX ME WB
xor r10, r11, r12 IF ->x
and r13, r14, r15 IF ->x
```

Problem 5: Unroll and schedule the loop from the problem above for maximum efficiency. Unroll the loop four times; the number of iterations will always be a multiple of four. Use software pipelining and take advantage of associativity to overlap the multiply latency. (In software pipelining a computation is spread over several iterations.) Code may be added before the LOOP label.

```
nop
LOOP: ! LOOP = 0x1008
```



Problem 1: The familiar loop below executes on a dynamically scheduled machine using a reorder buffer to name destination registers. The machine has the following characteristics:

- Two-way superscalar. An unlimited number of write-backs per cycle.
- A 16 -entry reorder buffer.
- A six-stage fully pipelined floating point multiply unit.
- Perfect branch target prediction. (Branch target in IF when branch is in ID.)

Show a pipeline execution diagram up to the fetch of the third iteration.
Explain why the first two iterations cannot be used to determine the CPI for a large number of iterations in this case. Estimate the CPI for a large number of iterations (a pipeline execution diagram is not necessary).

```
LOOP: ! LOOP = 0x1000
! Cycle 0
    ld f0, O(r1) IF ID L1 L2 WB IF ...
    IF ID L1 L2 WB
    muld f2, f0, f2 IF ID RS M1 M2 M3 M4 M5 M6 WB
        IF ID RS M1 M2 M3 M4 M5 M6 WB
                        IF ...
    addi r1, r1, #8 IF ID EX WB
                            IF ID EX WB
    sub r2, r1, r3 IF ID RS EX WB
                            IF ID RS EX WB
    bneq r2, LOOP IF ID RS B WB
        IF ID RS B WB
    xor r10, r11, r12 IF x IF x
    and r13, r14, r15
    or r16, r17, r18
    sgt r19, r20, r21
```

For clarity the first iteration is shown in black, the second in blue, and the third (just IF's) in orange. The first two iterations can't be used to determine CPI because they start differently, for example, in the first $f 2$ is available, but at the beginning of the second (cycle 3 ) the value for $f 2$ is not yet ready.

The CPI for a large number of iterations would be limited by the multiply unit. The hardware can fetch and decode at a rate of 3 cycles per iteration, but the multiply latency is 6 . Because there is a loop-carried dependency on the multiplier input the multiplies have to be done one after another, and so execution is limited to 6 cycles per iteration (after the reorder buffer fills). Since there are five instructions in an iteration the CPI is limited to $\frac{6}{5}$.

Problem 2: Unroll the loop in the problem above twice. (In the last homework it was unrolled four times.) Again exploiting the associativity of multiplication, rearrange the multiplies to improve the performance, but this time without using software pipelining. Why is software pipelining not necessary here?

```
! Solution
LOOP: ! LOOP = 0x1000
```



```
addi r1, r1, #16 IF ID EX WB IF ID EX WB IF ID EX WB
sub r2, r1, r3 IF ID RS EX WB IF ID RS EX WB
    IF ID RS EX WB
bneq r2, LOOP IF ID B WB IF ID B WB IF ID B WB
xor r10, r11, r12 IF x IF x IF x
and r13, r14, r15
or r16, r17, r18
sgt r19, r20, r21
```

For clarity the first iteration is shown in black, the second in blue, the third in orange, and the fourth (just an IF) in purple. (A pipeline diagram was not required for the solution, but is given here to help describe the solution.)

An important feature of the solution is the way the multiplies are done. The code above is limited to execute at a rate of six cycles per iteration because of the loop-carried dependency in the second multiply. (But this does twice as much work as the original code.) In the poor solution below the code is half as fast, limited to twelve cycles per iteration because the loop-carried dependency is a source in the first multiply and a destination in the second:

```
! WARNING: POOR Solution below!
LOOP: ! LOOP = 0x1000
    ld f0, O(r1)
    ld f10, 8(r1)
    muld f2, f0, f2
    muld f2, f10, f2
    addi r1, r1, #16
    sub r2, r1, r3
    bneq r2, LOOP
    xor r10, r11, r12
    and r13, r14, r15
    or r16, r17, r18
    sgt r19, r20, r21
! WARNING: POOR Solution above!
Refer to the good solution for the following discussion.
Software pipelining is not needed because dynamic scheduling allows instructions after the second multiply to start execution even before the second multiply starts. On a statically scheduled machine instructions after the second multiply would have to wait. Software pipelining can be used to reduce the wait by moving the second multiply to the next iteration.
```

Problem 3: The code below executes on a system using a one-level branch predictor with a 16 -entry BHT. Which entries will the branches use?

The BHT entry numbers are shown in the leftmost column below. The entry numbers are bits $2: 5$ in the instruction address, shown in the second column.

If the number of iterations is large, the prediction accuracy will be high. If a certain number of additional nops are inserted before SKIP1 the prediction accuracy will drop. How many and why?

By inserting nop instructions the BHT entry used by the second and third branches will change. Prediction accuracy will fall if the first and second branch use the same entry since their outcomes are always different from each other. Each inserted nop increases the BHT entry number by one, 13 nop's would put the second branch in entry 1 , the same as the first.

```
! Note: r2 is not modified inside the loop.
BHT En Addr LOOP: ! LOOP = 0x1000
    0x1000: subi r1, r1, #1
1 0x1004: bneq r2, SKIP1
    0x1008: add r10, r10, r11
    0x100c: nop
SKIP1:
4 0x1010: beqz r2, SKIP2
    0x1014: add r12, r12, r13
    SKIP2
6 0x1018: bneq r1, LOOP
```

H

Problem 4: Determine the prediction accuracy of a one-level branch predictor on each branch in the code below. The predictor uses a 1024 -entry BHT. There is a .5 probability that a loaded value will be zero.

Because random numbers are loaded, the first branch (following LOOP) and the branch following SKIP2 can't be predicted, so the accuracy will be about $50 \%$.

The second branch (following SKIP1) follows the pattern N T N T .... Depending on how the BHT entry is initialized, the prediction accuracy will be $50 \%$ or $0 \%$.

The third branch (following SKIP3) follows the pattern N T T T N T T T .... The prediction accuracy will be $75 \%$ (the not taken is predicted taken after warm up).

The last branch (following SKIP4) is taken for all but the last iteration, the prediction accuracy will be $100 \%$ for branches predicted after the first two iterations of the loop.

```
LOOP:
    addi r2, r2, #4
    lw r1, O(r2)
    bneq r1, SKIP1
    add r10, r10, r11
SKIP1:
    andi r3, r2, #4
    bneq r3, SKIP2
    add r11, r11, r12
SKIP2:
    beqz r1, SKIP3
    add r12, r12, r11
SKIP3:
    andi r4, r2, #12
    bneq r4, SKIP4
    add r13, r13, r11
SKIP4:
    sub r5, r2, r6
    bneq r5, LOOP
```

Problem 5: How many BHT entries will the branches in the code above use in the middle of its execution (explained below) in a two-level gselect predictor that uses 10 bits of global branch history and 6 instruction address bits? The loop iterates many times, the middle of its execution starts after many iterations.

The global history has the following repeating pattern:
rNrNT rTrTT rNrTT rTrTT rNrNT rTrTT rNrTT $\mathrm{rTrTT} \ldots$, where r is random and can be either T or N . Each group corresponds to an iteration. The global history register contains ten outcomes. The global history when predicting the first branch in the loop might see rNrNT rTrTT, the global history for the second branch might see NrNT rTrTT $r$, and so on.

Ignoring the r's, each branch can see four possible global history patterns (since there are four sets of branch outcomes in an iteration such as rNrNT and they occur in the same order each time). Taking the global history into account, there are 16 variation on each pattern (since each pattern contains 4 r's). Therefore each branch can see 64 different patterns. There will be a different BHT entry for each branch and each pattern (since there are no collisions) and so the total number of BHT entries is $16 \times 4 \times 5=320$.

How many bits of global branch history are needed so that the branch following SKIP3 is predicted very accurately?

The branch following SKIP3 follows the pattern N T T T N T T T .... To distinguish the not taken case from the others the branch predictor might look at the three previous outcomes of the SKIP3 branch. If they are all taken it would predict not taken. That would require a global history length of 15 . However, it's possible to use a shorter global history: look at the two previous outcomes of the SKIP3 and the SKIP1 branch. If the two last SKIP3 branches are TT and the two last SKIP1 branches are TN, predict not taken. (Don't forget that the global history contains all branches in this loop, but the other branches here are just noise.) So the minimum global history size is just 10 outcomes.

## 72 Spring 2000 Solutions

## EE 4720

Homework 1 solution
Due: 9 February 2000
Problem 1: Using the SPARC Architecture Manual (SAM) V8 answer the questions below. The SPARC Architecture Manual is distributed with the source for the microSPARC IIep in directory .../models/sparc_v8/docs/pdf of the distribution which can be downloaded from http://www.sun.com/microelectronics/communitysource/sparcv8/.
Alternate instructions will be given in class.
The SAM is 295 pages, so don't print it all out. It is not necessary that you understand everything in the SAM to answer these questions. See Appendix B to answer the last question.

- What size integers does the ISA support?

8, 16, 32, and 64 bits.

- What size floating-point numbers does the ISA support?

32, 64, and 128 bits.

- How many floating-point registers does the ISA support, how large are they, and how are the different-sized FP numbers placed in them?
Thirty-two 32 -bit registers which can be used as 1664 -bit registers or 8128 -bit registers.
- What is the binary coding of the following SPARC v8 instruction:


Problem 2: Find the static and dynamic instruction count for the DLX program below. (DLX is described in Chapter 2 of the text and summarized in the last two pages. Comments, preceded by a !, describe what the instructions do.) The program adds up a table of numbers.

```
lhi r2, #0x1234 ! Load high: r2 = 0x12340000
ori r2, r2, #0x5678 ! r2 = r2 0x5678
addi r4, r0, #10 ! r4 = r0 + 10, r0 always = 0
sub r3, r3, r3 ! r3 = 0. There are lots of ways to do this!
LOOP:
    lw r1, 0(r2) ! r1 = Mem[r2+0]
    add r3, r3, r1 ! r3 = r3 + r1
    addi r2, r2, #4 ! r2 = r2 + 4
    subi r4, r4, #1 ! r4 = r4 - 1
    bneq r4, LOOP ! if r4 != 0 goto LOOP
        Static: 9 instructions.
        Dynamic: }4+10\times5=54\mathrm{ .
```

Problem 3: DLX does not allow arithmetic instructions to access memory. Suppose they could and suppose all the addressing modes in Figure 2.5 of the text were available. Re-write the program to use as few instructions as possible (but still perform the same function).

## Solution:

```
lhi r2, #0x1234 ! Load high: r2 = 0x12340000
ori r2, r2, #0x5678 ! r2 = r2 0x5678
addi r4, r0, #10 ! r4 = r0 + 10, r0 always = 0
sub r3, r3, r3 ! r3 = 0. There are lots of ways to do this!
LOOP:
add r3, r3, (r2)+
subi r4, r4, #1 ! r4 = r4 - 1
bneq r4, LOOP ! if r4 != 0 goto LOOP
```

Problem 4: Find the static and dynamic instruction count of the program written for the question above.

Static: 7 , dynamic: $4+3 \times 10=34$.
Problem 5: What factors (relating to CPI and $\phi$ ) would one have to take into account to compare the execution time using the dynamic instruction count of the original program and the re-written program?

With the new add instruction the dynamic instruction count drops from 54 to 34 . If the clock frequency and CPI of the two systems are the same execution time is $\frac{54-34}{54} \times 100 \% \approx 37 \%$ lower on the new system.

The CPI of the add instruction that accesses memory would likely be higher than the instructions it replaces and so the performance improvement may not be as low as the new, lower dynamic instruction count suggests.

It's also possible that to accommodate the new add instruction the clock frequency $(\phi)$ had to be lowered, another reason why the lower dynamic count is optimistic.

Problem 1: Show a pipeline execution diagram for the following DLX code fragment on a statically scheduled implementation in which the add functional unit has a latency of 3 (four stages) and an initiation interval of 2 (not the usual 1 ) and the multiply unit has a latency of 5 (six stages) and has an initiation interval of 3 (not the usual 1).

```
! Solution
! Cycle 0
addf f0, f1, f2 IF ID A1 A1 A2 A2 WB
addf f3, f0, f4 IF ID ------> A1 A1 A2 A2 WB
addf f5, f0, f7 IF -------> ID -> A1 A1 A2 A2 WB
gtf f0, f8 IF -> ID -> A1 A1 A2 A2 WB
multf f9, f0, f10 IF -> ID M1 M1 M1 M2 M2 M2 WB
```

Problem 2: The following DLX code fragment executes on a statically scheduled implementation in which the add functional unit has a latency of 3 (four stages) and an initiation interval of 1 (the usual 1) and the multiply unit has a latency of 5 (six stages) and has an initiation interval of 1 (the usual 1).

The implementation uses ID-stage branch target calculation. As is true for the pipelines used in class, the branch condition is not bypassed.

Instructions stall in ID to avoid structural hazards.
There are bypass paths from the WB stage to the inputs of the floating-point functional units.
(a) Show a pipeline execution diagram for the code.

```
LOOP: ! Solution
! Cycle 0
multd f0, f0, f2 IF ID M0 M1 M2 M3 M4 M5 WB IF ID
ld f4, O(r1) IF ID EX ME WB IF
addd f2, f2, f4 IF ID -> A0 A1 A2 A3 WB
addi r1, r1, #8 IF -> ID ----> EX ME WB
sub r2, r1, r3 IF ----> ID EX ME WB
bneq r2, LOOP IF ID ----> EX ME
xor r10,r11,r12 IF ----> x
```

(b) What is the CPI for a large number of iterations of the loop?

The first iteration takes 12 cycles. The state of the pipeline at the beginning of the second iteration (cycle 12) is different then the state at the beginning of the first (cycle 0) because the branch instruction from the first iteration is still present. That branch instruction finishes at the end of cycle 14 and will not change the way the second iteration executes, and so the second iteration will also take 12 cycles. Therefore the CPI for a large number of iterations is $\frac{12}{6}=2$ cycles per instruction.
(c) If the multiply functional unit latency were long enough the second iteration would take longer than the first iteration. (An iteration starts when the first instruction is in IF.) What is the smallest such latency?

The multiply uses values produced in a previous iteration (that is, it has a loop-carried dependency). If those values aren't ready execution will stall. In the example below the execution of multiply in the second iteration is stalled for one cycle (at cycle 15) because the result from the previous iteration is not ready. In this example the multiply unit has a latency of 13 cycles, is the latency were 12 cycles there would be no stall, and so the smallest latency that will increase the duration of the second iteration is 13 cycles.

```
! Part of Solution
! Cycle 0
multd f0, f0, f2 IF ID M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10M11WB
                                    IF ID -> MO M1
ld f4, O(r1) IF ID EX ME WB
addd f2, f2, f4 IF ID -> AO A1 A2 A3 WB
addi r1, r1, #8 IF -> ID EX ME WB
sub r2, r1, r3 IF ID -> EX ME WB
bneq r2, LOOP IF -> ID ----> EX ME -> WB
xor r10,r11,r12 IF ----> x
```

Problem 3: Schedule - but don't unroll-the code from the problem above to avoid as many stalls as possible. Show a pipeline execution diagram of the scheduled code. Hint: you can change the offset of the load double instruction.
LOOP: ! Solution

```
! Cycle 0
    addi r1, r1, #8 IF ID EX ME WB IF ID -> EX ME WB IF ID -> EX ME WB
    sub r2, r1, r3 IF ID EX ME WB IF -> ID EX ME WB
    ld f4, -8(r1) IF ID EX ME WB IF ID EX ME WB
multd f0, f0, f2 IF ID M0 M1 M2 M3 M4 M5 WB
                                    IF ID M0 M1 M2 M3 M4 M5 WB
addd f2, f2, f4 IF ID A0 A1 A2 A3 WB IF ID A0 A1 A2 A3 WB
bneq r2, LOOP IF ID EX ME WB IF ID EX ME WB
xor r10,r11,r12 IFx IFx
```

Problem 4: Unroll the loop below so that two iterations of the original loop form one unrolled loop. Schedule the code so that it executes as efficiently as possible. Assume there will be an even number of iterations and that every register not used in the original code is available and so can be used in the unrolled loop. The loop runs on the implementation described in the second problem.

```
LOOP:
    ld f0, 0(r1)
    multd f0, f0, f2
    addd f0, f0, f4
    sd 8(r1), f0
    addi r1, r1, #16
    sub r2, r1, r3
    bneq r2, LOOP
```

Two solutions are provided below. In the first the loop is unrolled without software pipelining. That is, the work done by one iteration of the unrolled loop is exactly the work done by two iterations of the original loop. This solution has several stall cycles, as can be seen in the pipeline execution diagram.

The second solution also uses software pipelining. A single iteration of this loop does the work of four half-iterations of the original loop. Instructions addd f6, f5, f4 and sd $8(r 1)$, f6 are part of one half-iteration, addd f16, f15, f4 and sd 8(r1), f16 are part of another half-iteration, ldf0, $32(r 1)$ and multd f5, f0, f2 are part of a third half-iteration, and ld $f 10,32$ (r1) and multd f15, f10, f2 are part of a fourth half-iteration. This solution suffers no stalls, it only looses a cycle due to the branch penalty. In class, software pipelining was covered in the context of IA-64 register rotation, but as shown below it can also be used with conventional ISAs.

```
! Solution 1: Unrolled, but no software pipelining.
LOOP:
```



```
! Solution 2:
! Unrolled loop with software pipelining. No stalls.
! Prologue
    ld f0, O(r1)
    multd f8, f0, f2
    ld f0, 16(r1)
    multd f18, f0, f2
    subi r13, r3, #32 ! In loop position of addi and sub swapped.
! Cycle }0
LOOP :
addd f6, f8, f4 IF ID A0 A1 A2 A3 WB IF ID A0 A1 A2 A3
ld f0, 32(r1) IF ID EX ME WB
addd f16, f18, f4
ld f10, 48(r1)
multd f8, f0, f2
sub r2, r1, r13
sd 8(r1), f6
sd 24(r1), f16
multd f18, f10, f2
addi r1, r1, #32
bneq r2, LOOP
! Epilogue
addd f0, f8, f4
sd 8(r1), f0
addd f0, f18, f4
sd 24(r1), f0
```


## EE 4720

Homework 4 solution
Due: 17 April 2000
Problem 1: The diagram below shows the execution of code on a dynamically scheduled machine that uses physical register numbers to name destination operands. Show the state of the ID register map, the commit register map, their free lists, and the physical register file for each cycle of the execution below. In the register maps and file show only values related to registers $f 0$ and $f 3$. Initially, $f 0=0, \mathrm{f} 1=10, \mathrm{f} 2=20$, etc. Initially, register f 0 is assigned to physical register 12 and f 3 is assigned to physical register 15 (ignore the other architected registers). Initially, both free lists contain physical register numbers $\{7,8,9,10,11\}$.

Note: As originally assigned the initial free lists did not contain register 11 and the pipeline execution diagram showed reservation station ( $R S$ ) segments. Both were mistakes and have been corrected.


The solution appears below. Blank entries in the tables below indicate that the value has not changed. The free lists (shown in braces, or curly brackets) are for the cycle in which the opening brace appears. For example, in cycle 3 the ID free list is 10,11 and the completion free list is $7,8,9,10,11$ (because there was no change since cycle 0 ). The row in which a free list appears is not significant, there is only one ID free list and one completion free list.


Problem 2: Repeat the problem above assuming that there is an exception in stage A1 of the execution of addf $f 3, f 0, f 5$, as shown below: The solution can start at the cycle in which the tables will differ from the solution above.

The solution appears below. The exception is not handled until the instruction reaches completion, at cycle 17. (So the solution below is identical to the one above up to cycle 17.) At cycle 17 the controller recovers from the exception by copying the completion map and completion free list to the ID map and free list. The diagram below shows this recovery being done in one cycle, but real system might take Ionger. Because the add encountered an exception the value it writes into the register file may not be valid, that is indicated by question marks.


Problem 3: The diagram below, of a dynamically scheduled processor, omits hardware that checks whether the register map should be updated in the WB stage. (The hardware was described in class.) Add the hardware to the diagram (at the same level of detail as other parts of the diagram).

Solution diagram not yet available.


Problem 4: Draw a pipeline execution diagram for the DLX code below running on a dynamically scheduled 4 -way superscalar implementation with the following characteristics:

- Dynamically scheduled using a reorder buffer to name registers (method 1 ).
- One load/store functional unit with stages L1 and L2.
- No dynamic (hardware) branch prediction, all branches are predicted not taken. Branch predictor uses the B functional unit and must wait for its operand like any other instruction.
- Four integer execution units.

Find the IPC for an execution of a large number of iterations. Show the execution for 14 cycles or until there is enough information to compute the IPC, which ever is shorter.

```
! Note: runs for many iterations.
    add r3, r0, r0
LOOP:! LOOP = 0x1000
    lw r1, 4(r2)
    add r3, r3, r1
    lw r2, 8(r2)
    bneq r2, LOOP
    xor r0, r0, r0
```

The pipeline execution diagram is shown below. The misprediction is detected in cycle 7 and the correct path is fetched in cycle 8. The xor and following instructions get squashed (or flushed from the reorder buffer). Since the iteration that starts at cycle 8 will take the same number of cycles as the one that starts at cycle 1 the IPC is $\frac{4}{7} \approx 0.571$.

```
! Solution
! Cycle 0
    add r3, r0, r0 IF ID EX WC
LOOP: ! LOOP = 0x1000
    lw r1, 4(r2) IF ID L1 L2 WC IF ...
    add r3, r3, r1 IF ID RS RS EX WC IF ...
    lw r2, 8(r2) IF ID RS L1 L2 WC IF ...
bneq r2, LOOP IF ID RS RS RS B WC
    IF ...
    xor r0, r0, r0 IF ID EX WB x
```

Problem 5: Repeat the problem above when the branch is statically predicted as taken and the branch target is computed in the ID stage.

The pipeline execution diagram is shown below. Since the branch target is computed in ID the target instruction is fetched two cycles after the branch. (With a branch target buffer it would be fetched one cycle after the branch is fetched.) The hardware is able to fetch and decode instructions in this loop at the rate of 2 IPC, but the completion rate is lower due to dependencies between the loads. The second load must wait one cycle for the first load to move out of L1, as it does in cycle 3. The first load must wait for the second load from the previous iteration to enter WB, as it does in cycle 5 . Because instructions are being fetched faster than they are begin committed some resource (such as reorder buffer slots or reservation stations) will be used up. When that happens (not shown below) instructions will stall in ID and fetch will drop to a rate of $\frac{4}{3}$ instructions per cycle. This is much faster than $\frac{4}{7}$ from the previous problem but still less than the 4 IPC that the processor is capable of.

```
! Solution
! Cycle 0
    add r3, r0, r0 IF ID EX WC
LOOP: ! LOOP = 0x1000
    lw r1, 4(r2) IF ID L1 L2 WC
    IF ID RS L1 L2 WC
                                IF ID RS RS L1 L2 WC
                            IF ID RS RS RS L1 L2 WC
! Cycle 0
    add r3, r3, r1 IF ID RS RS EX WC
                        IF ID RS RS RS EX WC
                                    IF ID RS RS RS RS EX WC
                                    IF ID RS RS RS RS RS EX WC
    lw r2, 8(r2) IF ID RS L1 L2 WC
                        IF ID RS RS L1 L2 WC
                                IF ID RS RS RS L1 L2 WC
                                    IF ID RS RS RS RS L1 L2 WC
    bneq r2, LOOP IF ID RS RS RS B WC
        IF ID RS RS RS RS B WC
                IF ID RS RS RS RS RS B WC
                    IF ID RS RS RS RS RS RS B WC
    xor r0, r0, r0 IF x IF x IF x IF x
! Cycle 0
```

Problem 6: Repeat the superscalar problem when the branch is statically predicted taken and in which the address of LOOP it $0 \times 1004$.

```
! Note: runs for many iterations.
    add r3, r0, r0
LOOP:! LOOP = 0x1004
    lw r1, 4(r2)
    add r3, r3, r1
    lw r2, 8(r2)
    bneq r2, LOOP
    xor r0, r0, r0
```

The pipeline execution diagram is shown below. Because of alignment the instructions for one iteration are fetched in two groups. (In the previous example the four instructions in an iteration neatly fit on one group.) This adds an extra cycle, so instructions are fetched at a rate of $\frac{4}{3}$ IPC, which is the same rate at which they are executed. So, even though instructions are fetched at a lower rate execution occurs at the same rate because of dependencies.

```
! Solution
! Cycle 0
    add r3, r0, r0 IF ID EX WC
LOOP: ! LOOP = 0x1004
    lw r1, 4(r2) IF ID L1 L2 WC
                        IF ID L1 L2 WC
                                IF ID L1 L2 WC
                        IF ID L1 L2 WC
    add r3, r3, r1 IF ID RS RS EX WC
                        IF ID RS RS EX WC
                                    IF ID RS RS EX WC
                                    IF ID RS RS EX WC
    lw r2, 8(r2) IF ID RS L1 L2 WC
        IF ID RS L1 L2 WC
                            IF ID RS L1 L2 WC
                                    IF ID RS L1 L2 WC
    bneq r2, LOOP IF ID RS RS B WC
        IF ID RS RS B WC
                                    IF ID RS RS B WC
                                    IF ID RS RS B WC
    xor r0, r0, r0 IF IDx IF IDx IF IDx IF IDx IF IDx IF IDx
! Cycle 0
```


## EE 4720

Homework 5 Solution
Due: 24 April 2000
Problem 1: The code below is run on three machines each using a slightly different one-level branch predictor. Each machine's branch predictor uses a 1024 -entry BHT. The first machine uses 2 -bit saturating counters (as described in class), the second machine uses the 2-bit prediction scheme illustrated in Figure 4.13 of the text, and the third uses a 3 -bit saturating counter. (The scheme illustrated in Figure 4.13 uses two bits, but it's not a saturating counter.) Find the prediction accuracy for each scheme on each branch instruction for a large number of iterations.

```
! r1 is initially set to a large value.
LOOP1:
    subi r1, r1, #1
    beqz r1, EXIT
    andi r2, r1, #6
    bneq r2, SKIP1
    add r3, r3, #1
SKIP1:
    andi r2, r1, #2
    bneq r2, SKIP2
    add r3, r3, #1
SKIP2:
    j LOOP1
EXIT:
```

Solution on next page.

The branch outcomes are shown below, the horizontal position indicates the order in which the branches are executed (the distance between them is not drawn to scale). Beneath the branch outcomes are the values of the branch counter (or state) for the prediction scheme indicated. An $x$ appears beneath a branch outcome if it was mispredicted. If the prediction accuracy for a branch and scheme depends on how the counter is initialized then counter values and outcomes are shown for several possible initializations. The prediction scheme illustrated in Figure 4.13 is called 2-bit state; the states are numbered 0 through 3 , with 0 being the state illustrated in the lower right, 1 lower left, 2 upper right, and 3 upper left. At least enough outcomes are shown to reveal a repeating sequence.

```
LOOP1:
    subi r1, r1, #1
    beqz r1, EXIT N N N N N N N N N N N
    ! 2-bit counter 3 x 2 x 1 0 0 0 0 0 0 0 ... 100%
    ! 2-bit state 3 x 2 x 0 0 0 0 0 0 0 0 ... 100%
    ! 3-bit counter 7 x 6 x 5 x 4 x 3 2 1 0 0 0 ... 100%
    andi r2, r1, #6
    bneq r2, SKIP1 N N T T T T T T T T T T T N N N N T T T T
    ! 2-bit counter 3 x 2 x 1 x 2 3 3 3 3 3 x 2 x 1 x 2 3 3 ... 5/8 = 62.5%
    ! 2-bit state 3 x 2 x 0 x 1 x 3 3 3 3 3 x 2 x 0 x 1 x 3 3 ... 4/8= 50%
    ! 3-bit counter 7 x 6 x 5 6 7 7 7 7 7 7 x 6 x 5 6 7 7 7 ... 6/8 = 75% 
    add r3, r3, #1
SKIP1:
    andi r2, r1, #2
    bneq r2, SKIP2 N N T T N N N T T N
    ! 2-bit counter 3 x 2 x 1 x 2 3 x 2 x 1 x 2 3 ... 1/4 = 25%
    ! 2-bit counter 0 0 0 x 1 x 2 x 1 0 x 1 x 2 ... 1/4 = 25%
    ! 2-bit state 3 x 2 x 0 x 1 x 3 x 2 x 0 x 1 x 3 ... 0/4 = 0%
    ! 2-bit state 0 0 0 x 1 x 3 x 2 x 0 x 1 x 3 ... 0/4 = 0%
    ! 3-bit counter 7 x 6 x 5 6 7 ... 2/4 = 50%
    ! 3-bit counter 5 x 4 x 3 x 4 5 ... 1/4 = 25%
    ! 3-bit counter 4 x 3 2 x 3 x 4 ... 1/4 = 25%
    ! 3-bit counter 0 0 0 x 1 x 2 1 0 x 1 x 2 ... 2/4 = 50%
    add r3, r3, #1
SKIP2:
    j LOOP1
EXIT:
```

Problem 2: What is the largest BHT size (number of entries) for which there will be collisions between at least two branches in the code above?

For there to be a collision the BHT address must de the same for at least two branches. If the BHT had just one entry it would use zero address bits and so all branches would have the same BHT address. But the problem asked for the maximum table size. The address of each instruction (using a made-up value of LOOP1) is shown below next to the instruction. The low-order bits of branch instruction addresses (skipping alignment) are shown to the left. If the BHT used three address bits then these would be the addresses and there would be no collisions. If two bits were used (the two least significant bits) the three address would still be different. If one address bit were used then the first two branches would be indistinguishable, there'd be collisions. So the answer is two entries.

Note that the BHT address is constructed using the address (PC) of the branch instruction, not the branch target address.
! r1 is initially set to a large value. LOOP1: 0x1000 0x1000: subi r1, r1, \#1
001 0x1004: beqz r1, EXIT

$$
0 \times 1008 \text { : andi r2, r1, \#6 }
$$

011 0x100c: bneq r2, SKIP1 0x1010: add r3, r3, \#1 SKIP1: 0x1014: andi r2, r1, \#2
110
0x1018: bneq r2, SKIP2

0x101c: add r3, r3, \#1 SKIP2:
0x1020: j LOOP1
EXIT:

Problem 3: The program below runs on a system using a gselect branch predictor with a 14 -bit branch history and a $2^{22}$-entry BHT.

Show the value of the global branch history just before executing each branch after a large number of iterations. (The branch can be taken or not taken.) Also show the address used to index (lookup the value in) the BHT.

Determine the prediction accuracy of each branch assuming no collisions in the BHT.

```
    ! r2 is initially set to a large value.
    add r1, r0, r0
LOOP1: ! LOOP1 = 0x1000
    addi r1, r1, #2
LOOP2: ! LOOP2 = 0x1080
    subi r1, r1, #1
    bneq r1, LOOP2
A: ... ! Nonbranch instructions.
    addi r1, r1, #3
LOOP3: LOOP3 = 0x1100
    subi r1, r1, #1
    bneq r1, LOOP3
B: ... ! Nonbranch instructions.
    subi r2, r2, #1
LINE: ! LINE = 0x1180
    bneq r2, LOOP1
```

The branch outcomes are shown in the diagram below. Each branch outcome appears twice, once in the line labeled "Global history," and once in the line holding the corresponding branch. The global branch history at a particular cycle consists of the last 14 branch outcomes.

```
    ! r2 is initially set to a large value.
LOOP1: ! LOOP1 = 0x1000
    ! Cycle: 
    addi r1, r1, #2
LOOP2: ! LOOP2 = 0x1080
    subi r1, r1, #1
    bneq r1, LOOP2 T N T N T N T N
A: ...
    addi r1, r1, #3
LOOP3: LOOP3 = 0x1100
    subi r1, r1, #1
    bneq r1, LOOP3 T T N T T N T T T N T T N
B: ...
    subi r2, r2, #1
LINE: ! LINE = 0x1180
    bneq r2, LOOP1 
T
T
```

For the first branch the global history at cycle x (see diagram above) is NTTNTTNTTNTTNT. The address of the first branch is $0 \times 1084$. The BHT address is constructed by concatenating the branch history with $22-14=8$ bits of the branch address: $00100001_{2}: 01101101101101_{2}$, where $00100001_{2}$ is the 8 low bits of $0 x 1084$ skipping alignment and $01101101101101_{2}$ is the binary representation of the branch history (obtained by changing " N "s to 0 and "T"s to 1 ), and ":" is a concatenation operator.

For the second branch the global history at cycle y is TNTTNTTNTTNTTN and the BHT address is $01000001_{2}$ : $10110110110110_{2}$.

For the third branch the global history at cycle $\mathbf{z}$ is TNTTNTTNTTNTTN and the BHT address is
$01100000_{2}: 10110110110110_{2}$.
Because the outcome is always the same (before the outer loop is exited) the last branch will be predicted with $100 \%$ accuracy.
Consider the first branch at positions a and b . Position a is at the first iteration of LOOP2 and position b is at the second (last) iteration. As can be seen the global history is the same whenever execution is at the first iteration of LOOP2 (except for the first few iterations of the outer loop). The same holds for the second iteration.

At position a the last two outcomes in the global history are NT, at position b the last two outcomes are TT, and so different BHT entries will be used. At the first iteration the branch is always taken so that BHT entry will saturate at 3 , at the second iteration the branch is never taken so that entry will saturate at 0 ; both branches will be predicted perfectly (after warmup). Here the global branch history holds 14 outcomes, if it held only one outcome the same BHT entry would be used for both iterations and prediction accuracy would suffer.

A similar argument holds for the second branch. Therefore the branch prediction accuracy will approach $100 \%$ for a large number of iterations.

Problem 4: Suppose the problem above ran on a gshare branch predictor with a 10-bit branch history and a $2^{10}$-entry BHT. Determine addresses for LOOP1, LOOP2, LOOP3, and LINE for which there would be collisions in the BHT after a large number of iterations. (Please retain program order.)

The global history at positions $y$ and $z$ are the same. Therefore if the index part of the address of the second and third branches were the same the same BHT entry would be used. Keep LOOP3 at 0x1100 (and the second branch at 0x1104) and change LINE to $1104_{16}+2^{10+2}=2104_{16}$.

## 73 Fall 1999 Solutions

Problem 1: What are the static and dynamic instruction counts of the two DLX programs below? (DLX is described in Chapter 2 of the text and summarized in the last two pages. Comments, preceded by a !, describe what the instructions do.) Be sure to use the value for r2 specified in the comments. Both programs find the population (number of 1's) in the binary representation of the value in r2. (For example, the population of $12_{10}=1100_{2}$ is $2,7_{10}=0111_{2}$ is 3 , and $d 06 f 00 d_{16}=218558477_{10}=1101000001101111000000001101_{2}$ is 12 .)

```
! Program 1.
! r2 = 0xd06f00d
add r1, r0, r0 ! r1 = 0. Initialize total.
LOOP:
    andi r3, r2, #1 ! r3 = r2 & 0x1. Put least-significant bit in r3.
    add r1, r1, r3 ! r1 = r1 + r3. Add to total.
srli r2, r2, #1 ! r2 = r2 >> 1. Shift right logical. Shift off LSB.
bneq r2, LOOP ! Branch if r2 not zero. Loop if more.
! Program 2.
! r2 = 0xd06f00d
! r4 = Base of table. Entry i is number of 1's in binary i.
add r1, r0, r0 ! r1 = 0. Initialize total.
LOOP:
andi r3, r2, # 0xff ! r3 = r2 & 0xff. Put 8 least significant bits in r3.
add r5, r4, r3 ! r5 = r4 + r3. Add to base of population table.
lbu r6, 0(r5) ! r6 = Mem[0+r5] Load byte unsigned, Load population of r3
add r1, r1, r6 ! r1 = r1 + r6. Add to the total.
srli r2, r2, #8 ! r2 = r2 >> 8. Shift right logical. Shift off 8 bits.
bneq r2, LOOP ! Loop if r2 not zero.
```

Program 1: static count, 5 instructions. Using data above program iterates 28 times. With four instructions per iteration dynamic count is $1+4 \times 28=113$ instructions.

Program 2: static count, 7 instructions. Using data above program iterates four times, with six instruction per iteration dynamic count is $1+6 \times 4=25$ instructions.

Problem 2: Suppose the programs above are run on machines that execute one instruction at a time without overlap (unlike most of the examples shown in class) and with no gaps between. Suppose the CPI for all instructions is 1 cycle and the clock frequency is 625 MHz (period is 1.6 ns ). How long would it take each program to run? Suppose the CPI for the lbu instruction was 3 cycles. How long would program 2 take?

If all instructions have a CPI of 1 , program 1 would take 113 inst $\times 1 \mathrm{CPI} \times 1.6 \mathrm{~ns} /$ cycle $=180.8 \mathrm{~ns}$ and program 2 would take only 40 ns .

With a CPI of three for lbu, program 2 would take

$$
((1+4 \times 5) \text { inst } \times 1 \mathrm{CPI}+4 \mathrm{inst} \times 3 \mathrm{CPI}) \times 1.6 \mathrm{~ns} / \text { cycle }=52.8 \mathrm{~ns},
$$

still faster than one. (Program 1 is not affected by the change in CPI for 1 bu. )

Problem 3: What changes would have to be made to program 2 if the lbu instruction (load byte unsigned) were changed to lhu (load half unsigned)?

The lbu instruction loads one byte, the lhu instruction loads two bytes. Assume the change was made because the table contains two-byte, rather than one-byte, entries. Then to find the $i$ th index one would look at address $\mathrm{r} 4+2 \times i$ rather than $r 4+i$. In the program $i$ is the contents of $r 3$, so we would have to multiply that by 2 . The modiffed program appears below.

```
! Modified Program 2.
! r2 = 0xd06f00d
! r4 = Base of table. Entry i is number of 1's in binary i.
add r1, r0, r0 ! r1 = 0. Initialize total.
LOOP:
andi r3, r2, # 0xff ! r3 = r2 & 0xff. Put 8 least significant bits in r3.
add r3, r3, r3 ! Multiply r3 by 2. (Using an add for speed.)
add r5, r4, r3 ! r5 = r4 + r3. Add to base of population table.
lhu r6, O(r5) ! r6 = Mem[0+r5] Load byte unsigned, Load population of r3
add r1, r1, r6 ! r1 = r1 + r6. Add to the total.
srli r2, r2, #8 ! r2 = r2 >> 8. Shift right logical. Shift off 8 bits.
bneq r2, LOOP ! Loop if r2 not zero.
```

Problem 4: The Easy ISA as described in class has only five instructions with no straightforward way of adding new ones. A non-straightforward way of adding instructions is to take advantage of the fact that the coding does not use all possible combination of bits. In particular, it is possible to specify an immediate as the destination of an arithmetic instruction even though the ISA has no corresponding instruction. For example, consider:

| add | Imm. | 3 |  | Reg. | r1 |  | Imm |  | 12 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 000 |  | 01 | 3 |  | 00 - | 1 |  | 01 |  | 0xc |
| $0 \quad 2$ | 3 | 45 | 24 | 25 | 2627 | 33 | 34 |  |  | 55 |

This could be interpreted as instruction add $3, r 1,12$, however there is no such instruction in the Easy ISA. (If there was, what would it do?)

Explain how this "hole" can be used to code additional instructions. Use this coding to add and, or, sll (shift left logical), and srl (shift right logical) instructions. The new instructions should use the same addressing modes as the existing arithmetic instructions.

If the addressing mode for the destination is immediate, interpret the immediate value as an extended opcode and interpret the next three operand fields as destination, source 1 , and source 2. Codings for the logical instructions: and, 0 ; or, 1; sll, 2; and slr, 3 .

Problem 5: Recall that an issue (it's not okay to say problem anymore) with the Easy ISA is that there is no CTI (control-transfer instruction: branch, jump, call, return, etc.) that will branch to an address held in a register. Only self-modifying code can do that. Write such code. The code should branch to an address held in register r100. The solution may use the instructions added above. Addresses in Easy ISA do not have to be aligned. Assume the most significant bit of the address is always zero. Hint: This assumption and the lack of alignment restrictions makes things alot easier.

The branch instruction contains its target address in bits 15 through 78. Those bits will have to be overwritten with the target address held in r100. Call the address of the braneh instruction BLINE. All Easy ISA instructions write 64-bit words starting at any address. Suppose r0 held a zero. Instruction add [BLINE+r0], r10, 0 would write the contents of r10 to the first 64 bits of the branch instruction. If r1 holds a 1 then add [BLINE+r1], r10, 0 would write the contents of $r 10$ to bits 8 through 71 of the branch instruction. If $r 2$ held a 2 then add [BLINE+r2], r10, 0 would modify bits 16 through 79 which is almost what we need. Since there is no way to exactly write bits 15 through 78 , the target address will have to be prepared. In this case preparation merely consists of shifting it one position to the left. Because the MSB of addresses are always zero there is no need to modify bit 15 and nothing need be done with the MSB bit of r100. The solution appears below:

```
add r3, 0, 2 ! Put constant 2 in r3.
! Shift the address by 1, store in branch, starting at byte 2.
    sll [r3+BLINE], r100, 1
    add r1, 0, 1 ! Set branch condition.
BLINE:
    b r1, r101, DONTCARE ! DONTCARE is changed by the time it executes.
```

Problem 1: Suppose the coding of DLX instructions were changed so the destination appeared before the source operands, as shown in the codings below:

New Type R:


New Type I:

| Opcode |
| :--- |
| rd |

Type J: (no change)
Opcode Offset


Show the changes needed to the pipeline below to implement this new ISA. The changes should only effect the ID and WB stages. If there are differences in the control inputs to multiplexors or other units, explain what those differences are.

Make sure your design executes store instructions correctly.
Changes shown in red.


Problem 2: The program below executes on the DLX implementation shown below. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others. All forwarding paths are shown. (If a needed forwarding path is not there, sorry, you'll have to stall.) A value can be read from the register file in the same cycle it is written. The destination field in the beqz is zero. Instructions are nulled (squashed) in this problem by replacing them with slt r0, r0, r0. All instructions stall in the ID stage.

```
! Initially, r1=0x101, r2=0x202, r3=0x303
! MEM[0x103] = 0xfe
sub r0, r0, r0
sub r0, r0, r0
sub r0, r0, r0
sub r0, r0, r0
sub r0, r0, r0
START: ! START = 0x50
    lb r1, 2(r1)
    addi r1, r1, #3
    or r1, r1, r2
    beqz r2, SKIP !(taken)
    add r3, r1, r2
    sub r0, r0, r0
    sub r0, r0, r0
SKIP:
    xor r3, r1, r3
    sub r0, r0, r0
    sub r0, r0, r0
    sub r0, r0, r0
```



The table below shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. Cycle zero is the time that lb is in instruction fetch. The first two columns are completed; fill in the rest of the table. Use a "?" for the value of the "immediate field" of a type $R$ instruction and for the output of the memory when no memory read is performed. Show pipeline register values even if they're not used. Assume that the ALU performs the branch target computation even though it was already computed in ID. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If no register value is written leave the entry blank.

Hints: See Spring 1999 HW 3 for a similar problem. One feature of the solution would not be present if lb were replaced by $a$ addi. Another feature may not be present if lb were replaced by lw.

Completed table appears below.

| Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| PC | 0x50 | 0x54 | 0x58 | $0 \times 58$ | 0x5c | 0x60 | 0x6c | 0x70 | 0x74 | 0x78 | 0x7e |
| IF/ID.IR | sub | Ib | addi | addi | or | beqz | add | xor | sub | sub | sub |
| Reg. Chng. | r0 $<0$ | ro $<0$ | $\mathrm{r} 0 \leftarrow 0$ | ro $<0$ | r0 $\leftarrow 0$ | $\mathrm{r} 1 \leftarrow-2$ | $\mathrm{r} 0 \leftarrow 0$ | $\mathrm{r} 1 \leftarrow 1$ | $r 1 \leftarrow 203$ | $\mathrm{r} 0 \leftarrow 0$ | r0 $\leftarrow 0$ |
| ID/EX.IR | sub | sub | 1 b | sit | addi | or | beqz | sit | X0r | sub | sub |
| ID/EX.A | 0 | 0 | 0x101 | 0 | 0x101 | -2 | 0x202 | 0 | 0x203 | 0 | 0 |
| ID/EX.B | 0 | 0 | 0x101 | 0 | 0x101 | $0 \times 202$ | ? | 0 | $0 \times 303$ | 0 | 0 |
| ID/EX.IMM | ? | ? | 2 | ? | 3 | ? | 3 | ? | ? | ? | ? |
| EX/MEM.IR | sub | sub | sub | Ib | sit | addi | or | beqz | sit | xor | sub |
| EX/MEM.ALU | 0 | 0 | 0 | $0 \times 103$ | 0 | 1 | 0x203 | 0x6c/4 | 0 | 0x100 | 0 |
| EX/MEM.B | 0 | 0 | 0 | 0x101 | 0 | 0x101 | 0x202 | 0 | 0 | 0 | 0 |
| MEM/WB.IR | sub | sub | sub | sub | Ib | sit | addi | or | beqz | sit | xor |
| MEM/WB.ALU | 0 | 0 | 0 | 0 | $0 \times 103$ | 0 | 1 | 0x203 | 0x6c/4 | 0 | 0x100 |
| MEM/WB. MD | ? | ? | 0 | 0 | -2 | ? | ? | ? | ? | ? | ? |

To help solve the problem, find a pipeline execution diagram for the code (shown below). Cycle numbers in diagram and table match.

```
    ! Initially, r1=0x101, r2=0x202, r3=0x303
    ! MEM[0x103] = 0xfe
    sub r0, r0, r0
    sub r0, r0, r0
! Cycle 
START: ! START = 0x50
    lb r1, 2(r1) IF ID EX MEM WB
    addi r1, r1, #3 IF ID --> EX MEM WB
    or r1, r1, r2 IF --> ID EX MEM WB
    beqz r2, SKIP !(taken) IF ID EX MEM WB
    add r3, r1, r2
    sub r0, r0, r0
    sub r0, r0, r0
SKIP:
\begin{tabular}{llllllll} 
xor & \(r 3, r 1, ~ r 3\) \\
sub & r0, r0, r0 & IF & ID & EX & MEM & WB \\
sub & r0, r0, r0 & & IF & ID & EX & MEM \\
sub r0, r0, r0 & & & IF & ID & EX \\
\end{tabular}
```

Problem 3: Consider the program:

```
LOOP:
    lw r1, O(r2)
    add r3, r1, r3
    addi r2, r2, #4
    bneq r1, LOOP
    or r4, r5, r6
```

For each implementation below provide a pipeline execution diagram showing execution up to the third fetch of 1 w and determine the CPI for a large number of iterations.

Not Bypassed:


Solution:

| ! Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

LOOP:
lw r1, O(r2) IF ID EX MEM WB IF ID EX MEM WB IF
add r3, r1, r3 IF ID -----> EX MEM WB IF ID -----> EX MEM WB
addi r2, r2, \#4 IF -----> ID EX MEM WB IF -----> ID EX MEM WB
bneq r1, LOOP IF ID EX MEM WB IF ID EX MEM WB
or $\mathrm{r} 4, \mathrm{r} 5, \mathrm{r} 6$
IFx
IFx

Each iteration takes the same amount of time, 7 cycles, and contains 4 instructions, for a CPI of $\frac{7}{4} \mathrm{CPI}=$ 1.75 CPI .

Bypassed:


Solution:

| ! Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| LOOP: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| lw r1, $0(r 2)$ | IF | ID | EX | MEM | WB |  | IF | ID | EX | MEM | WB |  | IF |  |
| add r3, r1, r3 |  | IF | ID | $-->$ | EX | MEM | WB | IF | ID | $-->$ | EX | MEM | WB |  |
| addi r2, r2, \#4 |  |  | IF | $-->$ | ID | EX | MEM | WB | IF | $-->$ | ID | EX | MEM |  |
| bneq r1, LOOP |  |  |  |  | IF | ID | EX | MEM | WB |  | IF | ID | EX |  |
| or r4, r5, r6 |  |  |  |  |  | IFx |  |  |  |  | IFx |  |  |  |

Each iteration takes the same amount of time, 6 cycles, and contains 4 instructions, for a CPI of $\frac{6}{4} \mathrm{CPI}=1.5 \mathrm{CPI}$.
Problem 4: Schedule (rearrange) the instructions in the program used in the previous problem to improve execution speed. (Do not change what the program does!). Show pipeline execution diagrams and determine CPI for the two implementations.

Solution:

| ! Not bypassed. |  |  |  |  |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| ! Cycle |  |  |  |  |  |  |  |  |

Each iteration takes the same amount of time, 6 cycles, and contains 4 instructions, for a CPI of $\frac{6}{4} \mathrm{CPI}=1.5 \mathrm{CPI}$. Solution:

```
! Bypassed Implementation
! Cycle 0
LOOP:
    lw r1, 0(r2) IF ID EX MEM WB IF ID EX MEM WB IF
    addi r2, r2, #4 IF ID EX MEM WB IF ID EX MEM WB
    add r3, r1, r3 IF ID EX MEM WB IF ID EX MEM
    bneq r1, LOOP IF ID EX MEM WB IF ID EX
    or r4, r5, r6 IFx IFx
```

Each iteration takes the same amount of time, 5 cycles, and contains 4 instructions, for a CPI of $\frac{5}{4} \mathrm{CPI}=$ 1.25 CPI .

Problem 5: Show the changes needed to implement the predicated instructions presented in class. (Set 4, page 25, as of this writing.) Describe the instruction format and show any datapath and control changes to the implementation below.

The solution described below adds predicated type-R instructions.
First, an instruction coding needs to be found. The coding should fit naturally into the DLX ISA such that implementations would be changed as little as possible. Since this is an addition to DLX, existing DLX instructions must not be changed.

Predicated versions of type-R instructions will be added. The predicated instructions have new opcodes, the new opcodes will not be listed. (As with other type-R instructions, the opcode is in the Fune field.)

Unless the format is changed, there is no room to specify the predicate register. Rather than changing the format, the interpretation of the fields will be changed. The destination filed $(\langle\mathbf{r d}\rangle)$ will specify the predicate and $\langle\mathbf{r s} \mathbf{1}\rangle$ will specify the first source operand (as usual) and the destination (they will always be the same). For example, instruction (r1) add r2, r2, r3 is coded:
Type R:

where opcode add.pn indicates a predicated add which writes its result of the predicate is non-zero. (If the Fune field contained an ordinary add the instruction would be add $r 1, r 2, r 3$.)

Here are some not-so-good alternative codings: Add a third source operand field, increasing the instruction size to 40 bits (maybe use the 3 left over bits for more opcode space). If all instructions are 40 bits, then old code won't work and so this is really a new ISA, not an extension of an existing one. If only predicated instructions are 40 bits, then implementation will be a challenge. First (this will be covered later in the semester) it's alot harder to build a memory system that returns any five consecutive bytes. It's much easier to fetch a power-of-two bytes at an aligned address. Another problem is that before the PC is incremented one has to find the instruction size, in the implementations considered size is determined in the cycle after its needed. If the PC were incremented in the beginning of the fetch cycle we could determine whether the previous instruction (in ID) was predicated, but the IF critical path length would be long in that case.

Now that the coding is determined, the pipeline must be modifed to implement it. Predicated instructions need three register values, that can't be avoided so a third read port must be added to the register file (see the illustration below). (Some real ISAs have special predicate registers, so additional general purpose register file ports are not needed.)

Predicated instructions have the destination register in a different place (bits 6 to 10) than other type-R instructions (16 to 20). The decode logie must recognize predicated instructions and place the correct destination register in the ID/EX.RD pipeline latch. (See illustration.)

An instruction is called predicated because its result isn't written back if the predicate is false. This will be implemented by replacing the destination register with a 0 in the EX stage. An $=0$ checks the predicate to see if it's zero. The predicate may come from the register file or be bypassed from MEM or WB. In ordinary predicated instructions the predicate is false if the predicate register is zero. In inverted predicate instructions ( (!r1) add r2, r2, r3) the predicate is false if the predicate register is non-zero. An exclusive or gate is used to invert the output of $=0$ for inverted instructions. (The output of $=$ Pred 0 is true if an inverted predicated instruction is present.)

If the predicate were tested in ID then it would not be possible to use the result of an immediately preceding instruction.

Changes are shown in red:


Problem 1: Consider the following method of implementing precise exceptions in DLX. An Exception Handler Address (EHA) register holds the address of the exception handler and an Exception Return Address (ERA) register holds the address of the faulting instruction. A new instruction (not in book) set.eha $\langle\mathbf{r s} \mathbf{1}\rangle$ places the contents of register $\langle\mathbf{r s} \mathbf{1}\rangle$ in EHA. After an exception occurs the address of the faulting instruction should be put in ERA and control should jump to the address stored in EHA. When an rfe (return from exception) instruction is executed control should jump back to the address stored in ERA.

Each stage has a squash signal that effectively replaces any instruction present with a nop. (See the illustration below.) Each stage also has an EXC signal which, in the middle of the cycle, is true if an exception is discovered in that stage. EXC will not be asserted if the stage contains an already squashed instruction. Registers EHA and ERA will be written with data at their in inputs if en is asserted using the same master /slave timing as the other registers and latches.

The diagram below shows a DLX implementation with the new squash signals (IF.SQ, etc.), exception signals (in every stage except WB), and the two new registers. The hardware shown can implement set.eha but does not implement exceptions or rfe. Add the hardware needed to do these. In particular:

- After an exception occurs control should jump to the address in EHA.
- Exceptions must be precise and handled in program order.
- rfe must return control to the faulting instruction.
- If the multiplexor in IF needs additional inputs, use the Taken signal to create the new multiplexor control signal. Taken is asserted only when the ID-stage adder produces the target address.
- Do not implement instructions that transfer ERA to and from an integer register.
- Assume that exception handlers will never encounter exceptions. (They do in real life, so the handler would need a way to save registers before any exceptions occur.)
- Do not test or set processor status bits for privileged state.


Based on your design, show a pipeline execution diagram for the code below in which the 1 w instruction raises a page fault exception in MEM and ant raises an illegal instruction exception in ID. Show the execution through the first two lines of the handler. Also show execution of the return from the handler and the second call of the handler for the ant instruction.

| ! Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| lhi r20, hi (HANDLER) | IF | ID | EX | MEM | WB |  |  |  |  |  |  |  |  |  |  |
| or r20, r20, lo(HANDLER) |  | IF | ID | EX | MEM | WB |  |  |  |  |  |  |  |  |  |
| set.eha r20 |  |  | IF | ID |  | MEM |  |  |  |  |  |  |  |  |  |
| add r1, r2, r3 |  |  |  | IF | ID | EX | MEM | WB |  |  |  |  |  |  |  |
| lw r4, 0(r5) |  |  |  |  |  |  |  | *MEM |  |  |  |  |  |  |  |
| ant r6, r7, r8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| sub r9, r10, r11 |  |  |  |  |  |  | IFx |  |  |  |  |  |  |  |  |
| and r12, r13, r15 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| or r15, r16, r17 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| HANDLER: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| sw 1000 (r0), r1 |  |  |  |  |  |  |  |  |  | IF | ID | EX | MEM | WB |  |
| sw 1004(r0), r2 |  |  |  |  |  |  |  |  |  |  | IF | ID | EX | MEM | WB |
| ! Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

! First lines duplicated
lhi r20, hi (HANDLER)
or r20, r20, lo(HANDLER)
set.eha r20
!Cycle $\quad 100101102103104105106107$
add r1, r2, r3
lw $\mathrm{r} 4,0(r 5) \quad$ IF ID EX MEM WB
ant r6, r7, r8
IF *ID* EXx
sub r9, r10, r11
IFx
and r12, r13, r15
or r15, r16, r17
! Return address still in ERA.
lw r1, 1000 (r0) IF ID EX MEM WB
rfe IF ID EX MEM WB
LINEX:
add r1, r2, r3 IFx
sub r4, r5, r6
xor $r 7, r 8, r 9$

In all the problems below all register values are available when the code starts executing. The datapath is fully pipelined so execution of floating point operations can start in the cycle after results are produced, just as the integer instructions do. Unless they are provided, use the following latency and initiation intervals: add unit: latency 3, initiation interval 1; multiply unit: latency 5, initiation interval 1; divide unit: latency 19, initiation interval 20.

Problem 2: Show a pipeline execution diagram for the code below. The branch is not taken.

```
! Cycle 0
multd f0, f2, f4 IF ID M0 M1x
beqz r1, SKIP IF ID EX ME WB
multd f0, f2, f6 IF ID M0 M1 M2 M3 M4 M5 WB
multd f0, f0, f8 IF ID ------------> MO M1 M2 M3 M4 M5 WB
add r1, r1, r2 IF -------------> ID EX ME WB
```

Problem 3: Show a pipeline execution diagram for the code below. The add functional unit has a latency of 3 and an initiation interval of 2. Hint: This problem tests knowledge of initiation intervals, use of functional units by different instructions, and usage of registers by single- and double-precision instructions.

```
! Cycle 0
LOOP:
gtd f12, f14 IF ID A1 A1 A2 A2 WB
addd f0, f2, f4 IF ID -> A1 A1 A2 A2 WB
addd f6, f8, f10 IF -> ID >> A1 A1 A2 A2 WB
addf f16, f7, f18 IF -> ID -------> A1 A1 A2 A2 WB
```

Problem 4: Show a pipeline execution diagram for the code below starting from the first iteration until the CPI for a large number of iterations can be determined. What is that CPI?

The branch condition is bypassed to the ID stage so the branch does not have to stall for r1. (See 1998 HW 3.)

```
!Cycle
    0
LOOP:
```


CPI is $\frac{6}{3}=0.5$.

## 74 Spring 1999 Solutions

The code fragment below, in C source and assembler forms, is referred to in the problems below.

```
for(i=0; i<1000; i++) if( s[i].type == 0 )
    suma += s[i].score; else sumb+=s[i].score;
! r3 initialized to address of first element.
    add r1, r0, r0 ! i=0
LOOP:
    slti r2, r1, #1000 ! r2 = 1 if r1 < 1000, otherwise r2 = 0.
    beqz r2, DONE
    1w r4, 0(r3)
    ld f0, 16(r3)
    bneq r4, SUMB ! Taken half the time.
    addd f2, f2, f0
    j NEXT
SUMB:
    addd f4, f4, f0
NEXT:
    addi r3, r3, #64 ! Size of element is 64 bytes.
    addi r1, r1, #1 ! Increment loop index.
    j LOOP
DONE:
```

Problem 1: Determine the static and dynamic instruction count for the DLX program above. The branch that tests r 4 will be taken half the time.

The dynamic count for each instruction is shown in the first column:

```
! r3 initialized to address of first element.
    add r1, r0, r0 ! i=0
LOOP:
    slti r2, r1, #1000 ! r2 = 1 if r1 < 1000, otherwise r2 = 0.
    beqz r2, DONE
    lw r4, 0(r3)
f 1000 ld f0, 16(r3)
    1000 bneq r4, SUMB ! Taken half the time.
f 500 addd f2, f2, f0
        500 j NEXT
            SUMB :
    addd f4, f4, f0
        NEXT:
    1000 addi r3, r3, #64 ! Size of element is 64 bytes.
    1000 addi r1, r1, #1 ! Increment loop index.
```

f 500
1000 j LOOP
DONE:

Static: 12 instructions. Dynamic: 9,503 instructions (totaling dynamic counts above).

Problem 2: Suppose the program runs for 1 millisecond on a system with a 10 MHz clock. Assuming no cache misses (an assumption that will be made for most of these problems), what is the average CPI?

Answer: $\mathrm{CPI}=1 \mathrm{~ms} 10 \mathrm{MHz} / 9503$ inst $=10000$ cycles $/ 9503$ inst $=1.0523 \mathrm{CPI}$.
Problem 3: Divide the instructions into two classes: floating-point and others. (The floatingpoint instructions include the addd and ld instructions.) Suppose on implementation $A$ the CPI of floating-point instructions, $\mathrm{CPI}_{\mathrm{fp}}$, is twice the CPI of the other instructions, $\mathrm{CPI}_{\mathrm{other}}$. If implementation $A$ uses a 10 MHz clock and runs the program in 1 millisecond (like the previous problem), what would the CPIs be? Implementation $B$ is the same as implementation $A$ except floating-point instructions have an average CPI that is 3 times the other instructions. Estimate how long it will take to run the program on implementation $B$ using a 10 MHz clock.

Let $t_{A}$ denote the execution time on implementation $A$ (which can be expressed in cycles or seconds).

$$
\begin{aligned}
t_{A} & =\mathrm{CPI}_{\mathrm{fp}} \mathrm{I} \mathrm{C}_{\mathrm{fp}}+\mathrm{CPI}_{\text {other }} \mathrm{IC}_{\text {other }} \\
& =2 \mathrm{CPI}_{\text {other }} \mathrm{I} \mathrm{C}_{\mathrm{fp}}+\mathrm{CPI}_{\text {other }} \mathrm{IC}_{\text {other }}
\end{aligned}
$$

Solving for $\mathrm{CPI}_{\text {other }}$ :

$$
\mathrm{CPI}_{\text {other }}=\frac{t_{A}}{2 \mathrm{C}_{\mathrm{fp}}+\mathrm{IC}_{\text {other }}}=\frac{10000 \text { cycles }}{2 \times 2000+7503}=0.8692 \mathrm{CPI}
$$

Then $\mathrm{CPI}_{\mathrm{fp}}=2 \mathrm{CPI}_{\text {other }}=1.7387 \mathrm{CPI}$. Let $t_{B}$ denote the execution time estimate for implementation $B$. Then

$$
\begin{aligned}
t_{B} & =\mathrm{CPI}_{\text {other }}\left(3 \mathrm{IC}_{\mathrm{fp}}+\mathrm{IC}_{\text {other }}\right)=0.8693(3 \times 2000+7503) \\
& =11738.7 \text { cycles }=1.17387 \mathrm{~ms}
\end{aligned}
$$

Problem 4: Suppose that an implementation executed instructions one after another with no overlapping and no gaps between instructions. If each instruction took five cycles to execute and the clock frequency was 10 MHz , how long would program execution take?

It would take $5 \times 9503=47515$ cycles $=4.7515 \mathrm{~ms}$.
$\rightarrow \mathrm{H}$

Problem 5: Suppose, somehow, a load double and load word instruction using scaled addressing were added to DLX. The assembler syntax is similar to the one in table 2.5 of the text, except a displacement is included at the end. For example, the execution of ld f0, 10 (r20) [r30] 40 will load f0 (and f1) with the contents of memory at address $10+r 20+r 30 * 40$. Rewrite the program above using the new instruction.

```
! r3 initialized to address of first element.
    add r1, r0, r0 ! i=0
LOOP:
    slti r2, r1, #1000 ! r2 = 1 if r1 < 1000, otherwise r2 = 0.
    beqz r2, DONE
    lw r4, 0(r3)[r1]64
    ld f0, 16(r3)[r1]64
    bneq r4, SUMB ! Taken half the time.
    addd f2, f2, f0
    j NEXT
SUMB:
    addd f4, f4, f0
NEXT:
    ! Note that r3 is no longer changed.
    addi r1, r1, #1 ! Increment loop index.
    j LOOP
DONE:
```

The SPARC assembly language program below is used in the problems that follow. SPARC register names are $\% \mathrm{~g} 0-\% \mathrm{~g} 7, \% \mathrm{iO}-\% \mathrm{i} 7, \% 10-\% 17$, and $\% 00-\% \mathrm{o}$; and $\% \mathrm{~g} 0$ is a zero register (like r0 in DLX). The destination for arithmetic, logical, and load instructions is the rightmost register (add $\% 11, \% 12, \% 13$ means $\% 13=\% 11+\% 12$ ). SPARC uses a condition code register and special condition-code-setting instructions for branches. Branches include a delay slot.

```
LOOP:
    ld [%l1], %12 ! Load 12 = MEM[ l1 ]
    addcc %l2, %g0, %g0 ! g0 = g0 + l2. Sets cond. codes. Note: g0 is zero reg.
    be DONE ! Branch if result zero.
    nop ! Fill delay slot with nop.
    add %16, %12, %16 ! 16 = 16 + 12
    andcc %l3, 1, %g0 ! g0 = 1 & l3. Sets cond. codes. Note: g0 is zero reg.
    be SKIP1
    nop
    add %14, 1, %14
SKIP1:
    subcc %13, 1000, %g0
    bpos SKIP2 ! Branch if >= 0;
    nop
    add %14, %13, %14
SKIP2:
    andcc %13, 1, %g0
    be SKIP3
    nop
    add %14, %14, %14
SKIP3:
    add %l1, 4, %l1
    ba LOOP ! Branch always. (Jump.)
    nop
DONE:
```

Problem 1: An execution of the code above on a SPARC implementation takes 1000 cycles. The dynamic instruction count is $\mathrm{IC}_{\text {all }}$ of which $\mathrm{IC}_{\text {nop }}$ instructions are nop's. Consider two ways of computing CPI:

$$
\mathrm{CPI}_{\mathrm{A}}=\frac{t}{\mathrm{IC}_{\text {all }}} \quad \text { and } \quad \mathrm{CPI}_{\mathrm{B}}=\frac{t}{\mathrm{IC}_{\text {all }}-\mathrm{IC}_{\text {nop }}}
$$

where $t$ is the execution time in cycles. Which is better? Justify your answer; an argument for either formula can be correct.
$\mathrm{CPI}_{\mathrm{A}}$ is better because it measures how efficiently a processor executes instructions, including nop instructions which are part of the code.

Problem 2: SPARC branches have a one-instruction delay slot, in the code above they are filled with nop's. Re-write the code filling as many slots with useful instructions as possible, reducing the number of instructions in the program.

```
    Solution:
```

```
    ld [%11], %12 ! Load 12 = MEM[ l1 ]
LOOP:
    addcc %12, %g0, %g0 ! g0 = g0 + l2. Sets cond. codes. Note: g0 is zero reg.
    be DONE ! Branch if result zero.
    andcc %13, 1, %g0 ! g0 = 1 & 13. Sets cond. codes. Note: g0 is zero reg.
    add %16, %12, %16 ! 16 = 16 + 12
    be SKIP1
    subcc %13, 1000, %g0
    add %14, 1, %14
SKIP1:
    bpos SKIP2 ! Branch if >= 0;
    andcc %13, 1, %g0
    add %14, %13, %14
SKIP2:
    be SKIP3
    add %l1, 4, %l1
    add %14, %14, %14
SKIP3:
    ba LOOP ! Branch always. (Jump.)
    ld [%11], %12 ! Load 12 = MEM[ 11 ]
DONE:
```

Problem 3: Re-write the program in DLX, taking advantage of DLX's use of general purpose registers for specifying branch conditions.

## Solution:

LOOP:
lw r2, O(r1)
beqz 2 , DONE
add r6, r2, r6
andi r10, r3, \#1
beqz r10, SKIP1
addi r4, r4, \#1
SKIP1:
sgei r11, r3, \#1000
bneq r11, SKIP2
add r4, r3, r4
SKIP2:
beqz r10, SKIP3 ! r10 computed before SKIP1.
add r4, r4, r4
SKIP3:
addi r1, r1, \#4
beqz r0, LOOP
DONE:

Problem 4: The program below executes on the DLX implementation shown below. The comments show the results of the xori, or, and 1 w instructions.
! Initially, $r 1=11$, $\mathrm{r} 2=22, r 3=33$, etc.
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
START: ! START = 0x50
xori r1, r9, \#7 !99 $\oplus 7=100$
or $r 2, r 3, r 4!33$ or $44=45$
1w r5, 9(r6) ! Mem $[9+66]=42$
sw 10(r7), r8
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0
addi r0, r0, \#0


The table below shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. Cycle zero is the time that xori is in instruction fetch. The first two columns are completed, continue filling the table up until the sw instruction finishes writeback. Ignore values which are not used and which depend on the func field of type-R instructions. Values which are not used and don't depend on the func field should be shown. The output of the data memory is zero when a store or no memory operation is performed. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If no register value is written leave the entry blank.

| Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| PC | 0x50 | $0 \times 54$ | $0 \times 58$ | 0x5c | $0 \times 60$ | $0 \times 64$ | $0 \times 68$ | 0x6c |  | . |  |
| IF/ID.IR | addi | Xori | or | IW | SW | addi |  |  |  |  |  |
| Reg. Chng. | r0 | r0 | r0 | r0 | r0 | r 1 | O0r 2 | r5 |  |  |  |
| ID/EX.IR | addi | addi | xori | or | IW | SW | addi | $\ldots$ |  |  |  |
| ID/EX.A | 0 | 0 | 99 | 33 | 66 | 77 | 0 | $\ldots$ |  |  |  |
| ID/EX.B | 0 | 0 | 11 | 44 | 55 | 88 | 0 | $\cdots$ |  |  |  |
| ID/EX.IMM | 0 | 0 | 7 | X | 9 | 10 | 0 | $\ldots$ |  |  |  |
| EX/MEM.IR | addi | addi | addi | Xori | or | IW | SW | addi |  |  |  |
| EX/MEM.ALU | 0 | 0 | 0 | 100 | 45 | 75 | 87 | 0 |  |  |  |
| EX/MEM. B | 0 | 0 | 0 | 11 | 44 | 55 | 88 | 0 |  |  |  |
| MEM/WB. IR | addi | addi | addi | addi | xori | or | IW | SW |  | . |  |
| MEM/WB . ALU | 0 | 0 | 0 | 0 | 100 | 45 | 75 | 87 | 0 | . |  |
| MEM/WB. MD | 0 | 0 | 0 | 0 | 0 | 0 | 42 | 0 | 0 | . |  |

In all problems below assume there are no cache misses and that all register values are available at the beginning of execution.

Problem 1: The pipeline shown below cannot execute the jal or jalr instructions. Identify and fix the problem. (Hint: Think about a difference between jal and beqz besides the fact that jal is unconditional.)


The problem: The jal and jalr instructions are supposed to save the return address (NPC) in r31 but in the pipeline above there is no path that NPC can take to the writeback stage. (The path through the ALU could be used if it wasn't already being used to compute the target address.)

The solution: provide EX/MEM. NPC and MEM/WB. NPC pipeline latches and connect them so that the return address can move to the writeback stage without having to go through the ALU. Connect the output of MEM/WB. NPC to the multiplexor leading to the register file. (MEM/WB. ALU and MEM/WB. MD are already connected to this multiplexor.)

Problem 2: The program below executes on the DLX implementation shown below. The comments show the results of some instructions. The implementation uses forwarding (bypassing) to avoid some data hazards and stalls to avoid others. The forwarding paths are shown. A value can be read from the register file in the same cycle it is written. The destination field in the bneq is zero. Instructions are nulled (squashed) in this problem by replacing them with or r0, r0, r0.


The table below shows the contents of pipeline registers and changes to architecturally visible registers r1-r31 over time. Cycle zero is the time that addi is in instruction fetch. The first two columns are completed; fill up the rest of the table. Ignore values which are not used and which depend on the func field of type-R instructions. Values which are not used and don't depend on the func field should be shown. Don't forget the IMM values for bneq. The row labeled "Reg. Chng." shows a new register value that is available at the beginning of the cycle. If no register value is written leave the entry blank.

| Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| PC | 0x50 | 0x54 | $0 \times 58$ | 0x5c | $0 \times 60$ | 0x50 | 0x54 | 0x58 | 0x5c | $0 \times 60$ | 0x50 |
| IF/ID. IR | sub | addi | add | xor | bneq | sub | addi | add | xor | bneq | sub |
| Reg. Chng. | r0 $<0 \times 0$ | r0 $<0 \times 0$ | r0 $\leftarrow 0 \times 0$ | r0 $<0 \times 0$ | $\mathrm{r} 1 \leftarrow 0 \times 2$ | $3 \mathrm{r} 2 \leftarrow 0 \times 89$ | r2 $\leftarrow 0 \times 2 a$ | $x$ | $\mathrm{r} 0 \leftarrow 0 \times 0$ | $\mathrm{r} 1 \leftarrow 0 \mathrm{xab}$ | r2 $\leftarrow 0 \times 111$ |
| ID/EX.IR | sub | sub | addi | add | xor | bneq | or | addi | add | xor | bneq |
| ID/EX.A | 0x0 | 0x0 | 0x22 | $0 \times 11$ | $0 \times 11$ | $0 \times 23$ | 0x0 | 0xaa | $0 \times 23$ | 0x23 | 0xab |
| ID/EX.B | 0x0 | 0x0 | $0 \times 11$ | 0x66 | 0x22 | $0 \times 0$ | 0x0 | $0 \times 23$ | $0 \times 66$ | ab | 0x0 |
| ID/EX.IMM | 0x0 | 0x0 | 0x1 | ? | ? | -0x10 | 0x0 | $0 \times 1$ | ? | ? | 0x14 |
| EX/MEM.IR | sub | sub | sub | addi | add | xor | bneq | or | addi | add | xor |
| EX/MEM.ALU | 0x0 | 0x0 | 0x0 | $0 \times 23$ | $0 \times 89$ | 0xaa | ? | 0x0 | 0xab | $0 \times 111$ | 0x1ba |
| EX/MEM. ${ }^{\text {B }}$ | 0x0 | 0x0 | 0x0 | $0 \times 11$ | $0 \times 66$ | 0x22 | 0x0 | 0x0 | 0x23 | $0 \times 66$ | 0xaa |
| MEM/WB. IR | sub | sub | sub | sub | addi | add | xor | bneq | or | addi | add |
| MEM/WB.ALU | 0x0 | 0x0 | 0x0 | 0x0 | 0x23 | $0 \times 89$ | 0xaa | ? | 0x0 | 0xab | $0 \times 111$ |
| MEM/WB.MD | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 |

Problem 3: The program below executes on the implementation also shown below.


```
add r1, r2, r3
and r4, r1, r5
sw 0(r4),r1
lw r1, 8(r4)
xori r5, r1, #1
beqz r5, TARGET
sub r5, r5, r5
...
TARGET:
    or r10, r5, r1
```

The implementation includes only the forwarding paths that are shown in the figure. A new register value can be read in the same cycle it is written. Show a pipeline execution diagram for an execution of the code in which the branch is taken.

## Solution:

```
Cycle: }\quad0\begin{array}{llllllllllllllllll}{12}
    add r1, r2, r3 IF ID EX MEM WB
    and r4, r1, r5 IF ID EX MEM WB
    sw O(r4),r1 IF ID --> EX MEM WB
    lw r1, 8(r4)
    xori r5, r1, #1
    beqz r5, TARGET
    sub r5, r5, r5
TARGET:
    or r10, r5, r1
    IF ID EX MEM WB
```

Problem 4: Add exactly those forwarding paths (but no others) that are needed in the DLX implementation used in the problem above so that the code above executes as quickly as possible. Show a pipeline execution diagram of the code (repeated below) on the modified implementation.

```
add r1, r2, r3
and r4, r1, r5
sw 0(r4), r1
lw r1, 8(r4)
xori r5, r1, #1
beqz r5, TARGET
sub r5, r5, r5
TARGET:
    or r10, r5, r1
```

The execution in the previous problem suffers three stalls, starting at cycles 4,7 , and 9 .
Without the stall at cycle 4 there would be no way for the data (the new value of $r$ 1) to reach the EX/MEM. B pipeline lateh when sw is at the MEM stage. This can be fixed with a bypass connection from the output of the writeback-stage multiplexor to a new multiplexor placed at the inputs to the EX/MEM. B pipeline latch.

The stall at cycle 7 cannot be avoided since the data is first available at the end of cycle 7 but would be needed at the beginning of cycle 7 (if the stall were removed).

The stall at cycle 9 provides time for the new value of r 5 to reach WB where it meets beqz at cycle 10 . One or both stall cycles can be eliminated by inserting bypass paths. To eliminate one stall cycle insert a bypass path from EX/MEM.ALU to the input of the $=0$ box in ID. To eliminate both stall cycles (while possibly lengthening the critical path) insert a bypass path from the ALU output (before the EX/MEM pipeline latch) to the $=0$ box.

The pipeline execution diagram below uses the conservative approach for the $=0$ bypass, from EX/MEM. ALU:

```
Cycle: 
    add r1, r2, r3 IF ID EX MEM WB
    and r4, r1, r5 IF ID EX MEM WB
    SW O(r4), r1 IF ID EX MEM WB
    lw r1, 8(r4) IF ID EX MEM WB
    xori r5, r1, #1 IF ID --> EX MEM WB
    beqz r5, TARGET IF --> ID --> EX MEM WB
    sub r5, r5, r5
    IF --> x
TARGET:
    or r10, r5, r1 IF ID EX MEM WB
```

Problem 5: The code below executes on the DLX implementation shown below which also includes the following floating-point hardware:

- As described in Section 3.7 of the text and in class, there is a four-stage FP add unit, a seven-stage multiply unit, and a 25 -cycle FP divide unit (not used in the code below). The FP add unit also performs FP comparisons, such as eqf.
- The floating-point branch instructions, bfpt and bfpf, are executed in the ID stage just as the integer branches, beqz and bneq. The FP condition code register (also not shown) is updated in the WB cycle but the value to be written is forwarded to the controller at the beginning of WB.
- All stalls are in the ID stage. Floating-point instructions skip the MEM stage.
- Floating-point values are forwarded from the WB stage to the inputs of the FP execution units. A value written to a FP register can be read in the same cycle.
(a) Show a pipeline execution diagram for two iterations of the code below in which bfpt is taken in the first iteration but not taken in the second. (Note: the loop is infinite.)
(b) Determine the CPI of an execution of the code for a large number of iterations in which bfpt is always taken.
(c) Determine the CPI of an execution of the code for a large number of iterations in which bfpt is never taken.
(d) Determine the CPI of an execution of the code for a large number of iterations in which bfpt is taken $50 \%$ of the time.


```
LOOP:
    addi r1, r1, #8
    lf f0, 0(r1)
    addf f1, f1, f0
    eqf f0, f2
    bfpt LOOP
    multf f1, f1, f3
    beqz r0, LOOP
    xor r2, r1, r3
```

Solution: (The label for the memory (MEM) stage has been shortened to ME. Three iterations (rather than two) are shown; they are needed to solve part (c).) LOOP:

addi r1, r1, \#8 IF ID EX ME WB IF ID EX ME WB IF ID EX ME WB IF ...
lf $f 0,0(r 1)$ IF ID EX

IF ID EX ME WB
IF ID EX ME WB
IF ID -> AO A1 A2 A3 WB
IF ID ----> A0 A1 A2 A3 WB IF $\rightarrow$ ID A0 A1 A2 A3 WB IF ----> ID A0 A1 A2 A3 WB
eqf f0, f2
IF ID $\rightarrow$ A0 A1 A2 A3 WB

## bfpt LOOP multf f1, f1, f3

beqz r0, LOOP
IF ID ----------> EX ME WB
IF ID ----------> EX ME WB
IF ID ----------> EX ME WB ...
IF ----------> X
IF ----------> ID M0 M1 M2 M3 M4 M5 M6 WB IF ----------> ID M0 M1 ...
xor r2, r1, r3
IF x
IF x
Part (b): If bfpt is taken the iteration consists of 5 instructions. If the branch is always taken each iteration will execute as the first above, and so there will be 11 cycles per iteration. The CPI is $11 / 5=2.2 \mathrm{CPI}$.

Part (c): If bfpt is not taken the iteration consists of 7 instructions. In the second iteration above the branch is not taken and so multf is executed, producing a new value of $f 1$. That new value is needed in the third iteration, stalling addf an extra cycle (the stall occurs in cycles 28 and 29 ). The second iteration takes 13 cycles (from cycle 11 to 24 ) but due to the extra cycle the third iteration takes 14 cycles (from cycle 24 to 38 ). Because iteration 3 and 4 start the same way (as can be determined by examining the state of execution [a vertical strip] at cycles 24 and 38 ) they should take the same number of cycles as should following iterations as Iong as the branch is not taken. (Note that iteration 2 at cycle 11 starts differently.) Therefore the CPI is $14 / 7=2 \mathrm{CPI}$.

Part (d): An iteration where bfpt is taken that follows an iteration where it isn't would take 12 cycles (such a pair is not shown in the diagram above). An iteration where bfpt is not taken that follows an iteration where it is would take 13 cycles; for example, the second iteration above. For a large number of iterations the CPI would be $(12+13) /(5+7)=2.083 \mathrm{CPI}$.

## EE 4720

In all problems below assume there are no cache misses and that all register values are available at the beginning of execution.
Problem 1: Show a pipeline execution diagram for the first 41 cycles of the code below on a dynamically scheduled implementation of DLX in which:

- There is one floating point multiply unit with a latency of 5 and an initiation interval of $\mathbf{2}$.
- There is a load/store functional unit with a latency of 1 . The segments are labeled L1 and L2.
- The FP add functional unit has a latency of 3 and an initiation interval of 1 .
- The integer functional unit has a latency of 0 and an initiation interval of 1 .
- The functional units have reservation stations with the following numbers: integer, 6-9; fp add, 0-1; fp multiply, 2-3; load/store, 4-5.
- There is no reorder buffer.
- The branch delay is one. (There are no branch delay slots.)
- Ignore load/store ordering.

Initially all reservation stations are available.

## LOOP:

addi r1, r1, \#8
sub r2, r1, r3
lf f0, $0(r 1)$
multf f1, f0, f0
multf $f 2, f 0, f 1$
sf 4(r1), f1
bneq r2, LOOP ! Assume always taken.
xor r4, r5, r6

| Cycle |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| addi | r1, r1, \#8 | IF | ID | 6:EX | 6:WB |  |  |  |  | IF | ID | 6:EX | 6:WB |  |  |  |  |  |  |  |  |  | IF | ID |
| sub | r2, r1, r3 |  | IF | ID | 7:EX | 7:WB |  |  |  |  | IF | ID | 7:EX | 7:EX | 7:WB |  |  |  |  |  |  |  |  | IF |
| $1 f$ | f0, O(r1) |  |  | IF | ID | 4:L1 | 4:L2 | 4:WB |  |  |  | IF | ID | 5:RS | 5:L1 | 5:L2 | 5:WB |  |  |  |  |  |  |  |
| multf | f1, f0, f0 |  |  |  | IF | ID | 2:RS | 2:M1 | 2:M1 | 2:M2 | 2:M2 | 2:M3 | 2:M3 | 2:WB |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID | 2:RS | 2:RS | 2:M1 | 2:M1 | 2:M2 | 2:M2 | 2:M3 | 2:M3 | 2:WB |  |
| multf | f2, f0, f1 |  |  |  |  | IF | ID | 3:RS | 3:RS | 3:RS | 3:RS | 3:RS | 3:RS | 3:M1 | 3:M1 | 3:M2 | 3:M2 | 3:M3 | 3:M3 | 3:WB |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID |  |  |  |  | -> | 3:RS | 3:RS | 3:M1 | 3:M1 |
| sf | 4(r1), f1 |  |  |  |  |  | IF | ID | 4:RS | 4:RS | 4:RS | 4:RS | 4:RS | 4:L1 | 4:L2 | 4:WB |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF |  |  |  |  | -> | ID | 4:RS | 4:L1 | 4:L2 |
| bneq | r2, LOOP |  |  |  |  |  |  | IF | ID |  |  |  |  |  |  |  |  |  |  |  | IF | ID |  |  |
| xor | r4, r5, r6 |  |  |  |  |  |  |  | IF | x |  |  |  |  |  |  |  |  |  |  |  | IF | x |  |
| LOOP: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Cycle |  | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 |
| addi | r1, r1, \#8 |  | IF | ID | 6:EX | 6:WB |  |  |  |  |  | IF | ID | 6:EX | 6:WB |  |  |  |  |  |  |  |  |  |
| sub | r2, r1, r3 |  |  | IF | ID | 7:EX | 7:WB |  |  |  |  |  | IF | ID | 7:EX | 7:EX | 7:WB |  |  |  |  |  |  |  |
| $1 f$ | f0, O(r1) |  |  |  | IF | ID | 4:L1 | 4:L2 | 4:L2 | 4:WB |  |  |  | IF | ID | 5:RS | 5:L1 | 5:L2 | 5:WB |  |  |  |  |  |
| multf | f1, f0,f0 | 2:M3 | WB |  |  | IF | ID | 2:RS | 2:RS | 2:M1 | 2:M1 | 2:M2 | 2:M2 | 2:M3 | 2:M3 | 2:WB |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID | 2:RS | 2:RS | 2:M1 | 2:M1 | 2:M2 | 2:M2 | 2:M3 | 2:M3 |
| multf | f2, f0,f1 3 | 3:RS | 3:M1 | 3:M1 | 3:M2 | 3:M2 | 3:M3 | 3:M3 | 3:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  | IF | ID | ---> | 3:RS | 3:RS | 3:RS | 3:RS | 3:RS | 3:RS | 3:M1 | $\begin{array}{r} \text { 3:M1 } \\ \text { ID } \end{array}$ | 3:M2 |  |  | 3:M3 | 3:--> | 3:RS | 3:RS |
| sf | 4(r1), f1 | 4:RS | 4:L1 | 4:L2 | 4:WB |  |  | IF | ---> | ID | 4:RS | 4:RS | 4:RS | 4:RS | 4:RS | 4:L1 | 4:L2 | 4:WB |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ---- |  | - | -- | --> | ID | 4:L1 |
| bneq | r2, LOOP | ID |  |  |  |  |  |  |  | IF | ID |  |  |  |  |  |  |  |  |  |  |  | IF | ID |

Note: In cycle 12 the load waits an extra cycle because L1 is being used by the store. (As a general rule, the instruction waiting longer should start first. When contending for the CDB , the functional unit with the longer latency gets priority.)

Problem 2: Determine the CPI for a large number of iterations of the loop above (or give a good reason why it would be very difficult to determine the CPI).

Consider the state of the machine when fetching the first instruction of the loop, addi. It is the same at cycle 8 and 30 (for example, in both cases the first multiply from the previous iteration started the second multiply stage, the sf and second multf are sitting in reservation stations, and the number of free reservation stations at each functional unit is the same in both cycles). There are 7 instructions per iteration, so the CPI is $(30-8) /(2 \times 7)=1.571$ CPI.

Problem 3: What are the minimum number of reservation stations of each type needed so that the code above executes at maximum speed? What is the CPI at maximum speed? (This part was not in the problem as originally assigned:) The CDB can handle any number of writebacks per cycle and there are an unlimited number of functional units.

The problem as originally assigned was more tedious than intended. To solve it one would need to find a repeating pattern of iterations. Because of contention for the $C D B$, the repeating pattern does not occur in the first few iterations and so one would have to tediously construct the diagram for many iterations.

To solve this problem construct a pipeline execution diagram assuming an unlimited number of reservation stations. The diagram should continue until every instruction in the first iteration completes. (This loop does not have inter-iteration dependencies, but if it did [e.g., if the second multiply were multf $f 2$, $f 1$, $f 2$ ] the diagram would continue until every instruction in the second iteration finished.) From the diagram find the maximum number of reservation stations used. For the code above the diagram should be continued until cycle 18 (a few extra cycles are shown):


[^0]Problem 4: The code below executes on a machine similar to the type described in the first problem except that it uses a reorder buffer. Draw a pipeline execution diagram for the code below, be sure to show when each instruction commits. Remember that instructions stall in the functional unit if they are not granted access to the CDB.
LOOP:
lf f0, $0(r 1)$
multf f1, f0, f0
multf f2, f0, f1
addf f3, f3, f0
lf $\mathrm{f} 4,8(\mathrm{r} 1)$
sf 4(r1), f1
multf $f 1, f 4, f 4$
multf f2, f4, f1
addi $\mathrm{r} 1, \mathrm{r} 1, \# 16$
sub r3, r4, r5
xor r6, r7, r8
or r9, r10, r11
Since a re-order buffer is being used instruction results will be identiffed by their reorder buffer entry number rather than their reservation station number. For that reason reservation stations are only held until execution initiation. For example, the first multf only needs a RS in cycle 3.
LOOP:

| Cycle |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 2425 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $1 f$ | f0, 0 (r1) | IF | ID | L1 | L2 | WC |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| multf | $\mathrm{f} 1, \mathrm{f0}, \mathrm{f0}$ |  | IF | ID | 2:RS | M1 | M1 | M2 | M2 | M3 | M3 | WC |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| multf | f2, f0, f1 |  |  | IF | ID | 3:RS | 3:RS | 3:RS | 3:RS | 3:RS | 3:RS | M1 | M1 | M2 | M2 | M3 | M3 | WC |  |  |  |  |  |  |  |  |
| addf | f3, f3, f0 |  |  |  | IF | ID | A1 | A2 | A3 | A4 | WB |  |  |  |  |  |  |  | C |  |  |  |  |  |  |  |
| lf | f4, 8(r1) |  |  |  |  | IF | ID | L1 | L2 | WB |  |  |  |  |  |  |  |  |  | C |  |  |  |  |  |  |
| sf | 4(r1), f1 |  |  |  |  |  | IF | ID | 4:RS | 4:RS | 4:RS | L1 | L2 | WB |  |  |  |  |  |  | C |  |  |  |  |  |
| multf | $\mathrm{f} 1, \mathrm{f} 4, \mathrm{f} 4$ |  |  |  |  |  |  | IF | ID | M1 | M1 | M2 | M2 | M3 | M3 | WB |  |  |  |  |  | C |  |  |  |  |
| multf | f2, f4, f1 |  |  |  |  |  |  |  | IF | ID | 3:RS | 3:RS | 3:RS | 3:RS | 3:RS | M1 | M1 | M2 | M2 | M3 | M3 | WB | C |  |  |  |
| addi | r1, r1, \#16 |  |  |  |  |  |  |  |  | IF | ID | EX | WB |  |  |  |  |  |  |  |  |  |  | C |  |  |
| sub | r3, r4, r5 |  |  |  |  |  |  |  |  |  | IF | ID | EX | ---> | WB |  |  |  |  |  |  |  |  |  | C |  |
| xor | r6, r7, r8 |  |  |  |  |  |  |  |  |  |  | IF | ID | 7:RS | EX | ---> | WB |  |  |  |  |  |  |  |  | C |
| or | r9, r10, r11 |  |  |  |  |  |  |  |  |  |  |  | IF | ID | 6:RS | 6:RS | EX |  | WB |  |  |  |  |  |  | C |

Problem 5: Consider the code execution from the problem above. Suppose there is an exception in the L2 segment executing the second lf. At what cycle would the trap instruction be inserted? What might go wrong if a reorder buffer had not been used?

The trap will be inserted when If reaches the head of the reorder buffer, at cycle 18. If a reorder buffer were not used and the preceding multiply raised an exception, the trap handler for lf might run before the one for multf.

Problem 6: Show the execution of the code below on a dynamically scheduled 4-way superscalar machine using a reorder buffer. Instruction fetch is aligned. There is one of each floating-point functional unit, with latencies and initiation intervals given in the first problem. There are four integer execution units. The reservation station numbers are as given in the first problem.
LOOP: = 0x1008
lf fo, $0(r 1)$
multf f1, f0, f0
multf f2, f0, f1
addf f3, f3, f0
lf $\quad \mathrm{f} 4,8(\mathrm{r} 1)$
sf 4(r1), f1
multf f1, f4, f4
multf f2, f4, f1
addi $\mathrm{r} 1, \mathrm{r} 1, \# 16$
sub $r 3, r 4, r 5$
xor $r 6, r 7, r 8$
or r9, r10, r1
LOOP: $=0 \times 1008=1000000001000$
$\begin{array}{llllllllllllllllllllllllllll}\text { Cycle } & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22\end{array}$
$\begin{array}{lllllrllllll}\text { lf } & \text { f0, } 0(r 1) & \text { IF } & \text { ID } & \text { L1 } & \text { L2 } & \text { WC } & & & & & \\ \text { multf } & \text { f1, f0, f0 } & \text { IF } & \text { ID } 2: R S & 2: R S & \text { M1 } & \text { M1 } & \text { M2 } & \text { M2 } & \text { M3 } & \text { M3 } & \text { WC }\end{array}$
multf f2, f0, f1 IF ID 3:RS 3:RS 3:RS 3:RS 3:RS 3:RS 3:RS M1 M1 M2 M2 M3 M3 WC
addf $f 3, f 3, f 0$
lf f4, 8(r1)
IF ID 0:RS A1 A2 A3 A4 WB
C
C
sf $\quad 4(\mathrm{r} 1), \mathrm{f} 1$
multf f1, f4, f4
multf f2, f4, f1
addi $\mathrm{r} 1, \mathrm{r} 1, \# 16$
sub $\mathrm{r} 3, \mathrm{r} 4, \mathrm{r} 5$
xor $r 6, r 7, r 8$
or r9, r10, r11
$\begin{array}{ll}\text { IF } & \text { ID } 11 \\ \text { L2 } & \text { WB } \\ \text { IF } & \text { ID } \\ 4: \text { RS } & 4: \text { RS } \\ 4: \text { RS }\end{array}$
$\begin{array}{llllllllll}\text { IF ID 2:RS } & \text { 2:RS } & \text { 4:RS } & \text { 4:RS } & \text { 4:RS } & 4: \text { RS } & \text { L1 } & \text { L2 } & \text { WB } & \text { C } \\ \text { IF } & \text { M1 } & \text { M2 } & \text { M2 } & \text { M3 } & \text { M3 } & \text { WB } & \text { C }\end{array}$
IF ID EX WB
M1 M2 M2 M3 M3 WC instructions are fetched in cycle 0 .

Problem 7: (Modified 12 November 1999) Rewrite the code below for the VLIW DLX ISA presented in class. Instructions can be rearranged and register numbers changed. In order of priority, try to minimize the number of bundles, minimize the use of the serial bit, and maximize the value of the lookahead field. When determining the lookahead assume that any register can be used following the last bundle in your code.
LOOP:
lf f0, $0(r 1)$
multf f1, f0, f0
multf f2, f0, f1
addf f3, f3, f0
lf f4, 8(r1)
sf 4(r1), f1
multf f1, f4, f4
multf f2, f4, f1
addi r1, r1, \#16
sub r3, r4, r5
xor r6, r7, r8
or r9, r10, r11
Solution:
LOOP:
\{ P 0
lf f0, $0(r 1)$
lf f4, 8(r1)
sub r3, r4, r5
\}
\{ P 0
multf f1, f0, f0
multf f11, f4, f4
addf f3, f3, f0
\}
\{ P 1
sf $4(\mathrm{r} 1), \mathrm{f} 1$
multf f12, f0, f1
multf f2, f4, f11
\}
\{ P 0
addi r1, r1, \#16
xor $r 6, r 7, r 8$
or r9, r10, r11
\}

## 75 Spring 1998 Solutions

# EE 4720 Computer Architecture - HW 1 Solution (Spring 1998) 

## Problem 1

## Temporal Locality

Instruction fetches for instructions within loop. (Since they are fetched multiple times.)
The store of the incremented histogram element, since the same address was used to load the element.
SW 0(r10), rll ! hist[e] = rl1

## Spatial Locality

Accesses to the array elements. (Since they are done sequentially.) For example:
LH r10, 0(r10) ! r10 = array[ i*JSIZE + j ]

## Possibly Neither Temporal Nor Spatial Locality

Accesses to the hist array, since the pattern of access depends on the value of elements of array, which are not known in advance.

LW r11, 0(r10) ! r11 = hist[ e ];

## Problem 2

The static count is simply the number of instructions, 24 . The dynamic count is 3275 instructions, the table below shows the dynamic count of each instruction. The labels (e.g., NEXTI:) are included for clarity.

Dynamic Instruction
1 ADDI r10, r0, \#20
1 MOVI2FP f0, r10
1 ADDI r1, r0, \#0 NEXTI:
11 SGEI r10, r1, \#10
11 BNEZ r10, DONEI
10 ADDI r2, r0, \#0
NEXTJ:
210 SGEI r10, r2, \#20
210 BNEZ r10, DONEJ
200 MOVI2PF f1, r1
200 MULT f1, f1, f0
200 MOVFP2I r10, f1
200 ADD r10, r10, r2

```
200 SLLI r10, r10, #1
200 ADD r10, r10, r4
200 LH r10, 0(r10)
200 SLLI r10, r10, #2
200 ADD r10, r10, r3
200 LW r11, 0(r10)
200 ADDI r11, rll, #1
200 SW 0(r10), rll
200 ADDI r2, r2, #1
200 J NEXTJ
    DONEJ:
10 ADDI r1, r1, #1
10 J NEXTI
DONEI:
```


## Problem 3

The scaled addressing mode makes accessing both arrays much easier. Element array [i*]SIZE +j$]$ is still loaded into a register, but the histogram element is incremented in a memory-memory add instruction.

```
ADDI r1, r0, #0 ! i=0
NEXTI:
    SGEI r10, r1, #10 ! if i >= ISIZE ...
    BNEZ r10, DONEI ! ... exit loop.
    ADDI r2, r0, #0 ! j=0
NEXTJ:
    SGEI r10, r2, #20 ! if j >= JSIZE ...
    BNEZ r10, DONEJ ! ... exit loop
    MULTI r10, r1, JSIZE ! r10 = i * JSIZE
    ADD r10, r10, r2 ! r10 = i * JSIZE + j
    LH r10, 0(r4)[r10] ! d = 2; r10 = array[r10]
    ADDI O(r3)[r10],0(r3)[r10],#1 ! d=4; hist[r10] = hist[r10]+1;
    ADDI r2, r2, #1 ! r2 = j+1
    J NEXTJ
DONEJ:
    ADDI r1, r1, #1 ! r1 = i+1
    J NEXTI
DONEI:
```

The new static count is 14 , the dynamic count is 1674 , the table below lists count by instruction.
1 ADDI r1, r0, \#0

11 SGEI r10, r1, \#10
11 BNEZ r10, DONEI
11 ADDI r2, r0, \#0
NEXTJ:
210 SGEI r10, r2, \#20
210 BNEZ r10, DONEJ
200 MULTI r10, r1, JSIZE
200 ADD r10, r10, r2

```
200 LH r10, 0(r4)[r10]
200 ADDI 0(r3)[r10],0(r3)[r10],\#1
200 ADDI r2, r2, \#1
200 J NEXTJ
    DONEJ:
    10 ADDI r1, r1, \#1
    10 J NEXTI
        DONEI:
```


## Problem 4

## Clock frequency for fixed CPI

Need to solve: $\mathrm{IC}_{\text {new }} \mathrm{CPI}_{\text {new }} / \mathrm{f}_{\text {new }}=\mathrm{IC}_{\text {old }} \mathrm{CPI}_{\text {old }} / \mathrm{f}_{\text {old }}$ for $\mathrm{f}_{\text {new }}$. Solving yields, $\mathrm{f}_{\text {new }}=1 \mathrm{GHz} \mathrm{IC}_{\text {new }} / \mathrm{IC}_{\text {old }}=$ 511 MHz

So, even though an implementation using complex addressing modes might have a slower clock, it can still run faster. Note that the assumption that the CPI is the same is unrealistic, since it would take much longer to execute, for example, an instruction that had three memory operands.

## CPI for fixed clock frequency.

Need to solve: $\mathrm{IC}_{\text {new }} \mathrm{CPI}_{\text {new }} / \mathrm{f}_{\text {new }}=\mathrm{IC}_{\text {old }} \mathrm{CPI}_{\text {old }} / \mathrm{f}_{\text {old }}$ for $\mathrm{CPI}_{\text {new }}$. The resulting instruction execution time is $\mathrm{CPI}_{\text {new }}=\mathrm{CPI}_{\text {old }} \mathrm{IC}_{\text {old }} / \mathrm{IC}_{\text {new }}=0.489 \mathrm{CPI}$.

So, even though CPI will suffer in an ISA using elaborate addressing, it is still possible to gain performance.

## Problem 5

## Unoptimized Assembler

Some notes on Sparc assembler:
 letters are for input, local, output, and general. Register \%i6 is also called $\% \mathrm{fp}$, frame pointer, and points to the current stack frame. Register \%g0, like r0 in DLX, is always zero.

- Operands are backwards (compared to DLX), so add $\% l 0, \% l 1, \% l 2$ means $\% l 2=\% l 1+\% l 0$.
- Sparc branches are delayed, in the unoptimized code below the delay slots are filled with nops.
- In the code below global variable values have to be loaded into registers before use (unlike the DLX code in the assignment). This is a two-step process: first the address of the variable must be loaded, then the memory at that address is loaded. This is done for array and histo. (Variables i, j, and e are loaded differently since they are local to the procedure and so are stored on the stack.)
- In unoptimized code, any change in a variable is immediately written back to memory, and memory is read whenever a variable is used, even if it was written in the preceding instruction. This makes the program easier to debug.
- Instruction mov \%r1,\%r2 is a synthetic instruction, the assembler emits code for a or $\% \mathrm{~g} 0, \% \mathrm{r} 1, \% \mathrm{r} 2$

The output of the Sun compiler is shown below. The compiler helpfully shows the source lines corresponding to the assembler instructions, and also breaks code into basic blocks which is convenient for finding the dynamic
instruction count. (A basic block is a group of instructions that can only be entered from the first instructions [can't branch into the middle] and can only be exited from the last instruction [only the last instruction can be a CTI]. All instructions in a basic block are executed the same number of times.)

Static count: 55 (yes, nops count, they take space). Dynamic count, 6938 (yes, nops count, they at least take fetch and decode space). The count is detailed below, by basic block, followed by an annotated assembly language listing.

| Block <br> Number | Static <br> Count |  |  |
| :---: | ---: | ---: | ---: |
| Dynamic Count <br> (Per instr) | Dynamic Count <br> (Total) |  |  |
| 1 | 6 | 1 | 6 |
| 2 | 6 | 10 | 60 |
| 3 | 34 | 200 | 6800 |
| 4 | 7 | 10 | 70 |
| 5 | 2 | 1 | 2 |

```
! File hw1.c:
    extern short int *array;
    extern int *hist;
    #define ISIZE 10
    #define JSIZE 20
    void histo()
    {
        int i,j;
        for(i=0; i<ISIZE; i++)
    mov 0,%l0
    st %l0,[%fp-4] ! Store reg. l0 into frame pointer - 4 = &i.
    ld [%fp-4],%l0 ! Load the 0 just written.
    cmp %l0,10 ! Compare i to 10, set condition bits.
    bge .L19 ! Branch if i greater than 10.
    nop ! Sparc has a branch delay slot, wasted here.
    ! block 2
.L20:
.L17:
! 13 for(j=0; j<JSIZE; j++)
    mov 0,%l0
    st %l0,[%fp-8] ! Store reg l0 info frame pointer - 8 = &j...
    ld [%fp-8],%l0 ! ...and load it back in.
    cmp %l0,20 ! Compare to 20
    bge .L23 ! Branch if not smaller.
    nop
    ! block 3
.L24:
.L21:
! 14 15 { int e= array[ i * JSIZE + j ];
    sethi %hi(array),%l0 ! Load hi bits of array pointer addr.
    or %l0,%lo(array),%l0 ! Load rest of array pointer address.
    ld [%l0+0],%l3 ! Load address of first element.
```

| $l d$ | $[\% f p-4], \% l 0$ |
| :--- | :--- |
| sll | $\% l 0,2, \% l 2$ |
| sll | $\% l 0,4, \% l 1$ |
| add | $\% l 2, \% l 1, \% l 0$ |
| ld | $[\% f p-8], \% l 1$ |
| add | $\% l 0, \% l 1, \% 10$ |
| sll | $\% l 0,1, \% l 1$ |
| add | $\% l 3, \% l 1, \% l 0$ |
| dsh | $[\% l 0+0], \% 10$ |
| sll | $\% l 0,16, \circ l 0$ |
| sra | $\% l 0,16, \% l 0$ |
| st | $\% l 0,[\% f p-12]$ |

## ! Load i into 10

sll \%l0,2, \%l2
$12=\mathrm{i} * 4$
! $11=$ i * 16
! 10 = i * 20 = i * JSIZE;
$11=j$
! l0 = j + i * JSIZE
! l1 = l0 * $2=10$ * sizeof(short int)
! l0 = \&array[ j + i * JSIZE ]
l0 = array[ j + i * JSIZE ] = e
l0 = 10 << 16 (Part 1 of sign ext.)
! $10=10 \gg 16$ (Part 2 of sign ext.)
! e = l0
!
16
hist[ e ]++;
sethi \%hi(hist), \%l0
or $\quad \% l 0, \%$ lo(hist), $\%$ l0
ld [\%10+0], $\%$ l2
ld [\%fp-12], \%l0
sll \%l0,2, \%l1
$\% l 2, \% l 1, \% l 0$
st $\quad$ \%l0, [ $\% \mathrm{fp}-16]$
[\%fp-16], \%l0
$\begin{array}{ll}l d & {[\% l 0+0], \% l 0} \\ \text { add } & \% 10,1, \% l 1\end{array}$
ld $\quad[\% f p-16]$, \% 10
st $\quad$ oll, [ $\% 10+0]$
17
$!$
\}

| ld | $[\% f p-8], \% l 0$ |
| :--- | :--- |
| add | $\% \% 0,1, \% l 0$ |
| st | $\% 10,[\% f p-8]$ |
| ld | $[\% f p-8], \% 10$ |
| cmp | $\% 10,20$ |
| bl | .$L 21$ |
| nop |  |

```
! block 4
```

.L25:
.L23:

| ld | $[\%$ |
| :--- | :--- |
| add | $\% l$ |
| st | $\% l$ |
| ld | $[\%$ |
| cmp | $\% l$ |
| bl | .$L$ |
| nop |  |
| ! block | 5 |

.L26:
.L19:
.L15:
${ }_{\text {jmp }}^{\text {restore }}$

## Optimized (for speed) Assembler

Notes on optimized code

- Better use of registers is made. There is less saving and loading variable values.
- Instructions are rearranged, sometimes to make more efficient use of the pipeline. (Do arithmetic several instructions before the result is needed.)
- The block labels below are instructor annotations. (The block labels in the unoptimized code were generated by the compiler.)
- The "volatile" comment was placed by the compiler. That indicates a variable is being loaded even though it could have been left in a register because it's possible that the value changed in memory while the code was executing. Were array and hist declared static, no other procedure would "know" there names, and the compiler would not have to load there values multiple times.

Static count, 26, dynamic count 2476. Alot fewer!

! FILE hw1.c

| $!$ | 2 |
| ---: | ---: |
| $!$ | 3 |
| $!$ | 5 |
| $!$ | 6 |
| $!$ | 8 |
| $!$ | 9 |
| $!$ | 10 |
| $!$ | 12 |

```
!extern short int *array;
!extern int *hist;
!#define ISIZE 10
!#define JSIZE 20
!void histo()
!{
    int i,j;
    for(i=0; i<ISIZE; i++)
```

! Block 1

| /* 000000 |  | or | \%g0,0,\%g4 |  |
| :---: | :---: | :---: | :---: | :---: |
| /* 0x0004 | */ | or | \%g0, 0, \%o5 |  |
| 13 | ! | for (j=0; j | SIZE; j++) |  |
| /* 0x0008 | 13 */ | or | \%g0, 0, \%g1 | ! j = 0; |
| /* 0x000c | */ | sethi | \%hi(hist),\%g3 |  |
| /* 0x0010 | */ | sethi | \%hi(array), \%g2 |  |

! Block 2

| $!$ | 14 |
| :--- | :--- |
| $!$ | 15 |
| $!$ | 16 |

/* 0x0014
/* 0x0018
! Block 3
/* 0x001c
17
/* 0x001c
.L900000109:
! \{
! int e= array[ i * JSIZE + j ];
! hist[ e ]++;

$$
\begin{aligned}
& 16 \text { */ sll } \quad \mathrm{sll}, 1, \% 04 \\
& \text {.L77000005: } \\
& \text { ! o3 = \&array[0] } \\
& \text { */ ld [\%g2+\%lo(array)],\%o3 ! volatile }
\end{aligned}
$$

    .L9000000110:
        ! o3 \(=\operatorname{array}[\mathrm{i} *\) JSIZE \(+\underset{\text { j }}{ }]=\mathrm{e}\)
        */ idsh [\%03+\%04],\% \(\%\) ! volatile
    \}
    

## EE 4720 Computer Architecture - HW 2 Solution (Spring 1998)

## Problem 1

Execution is shown below, the diagram is wide so you'll have to scroll or maximize your browser window. Tree killers should remember that many browsers have a landscape option in their print command. Two instructions, i1 and i2, are included after the branch. They never execute but they are fetched.

Note that the execution of a single instruction uses just one line, and MEM is abbreviated to ME. Abandoned instruction are shown in gray.


## Problem 2

First, certain values need to be assumed since they weren't provided in the problem, the values are in the table below.
Location

| L00P | Assumed Value Comment |
| :--- | :---: |
| r3 | 0x2000 Address of lw instruction. |
| r4 | 2 Branch condition register before being set. |
| r10 | 50 Sum limit. |
| r11 | 0x1000 Part of array address. |
| Mem $[0 \times 1010]$ | $0 \times 10$ Another part of array address. |

The code fragment is shown below along with the register values that change in the first iteration:


The code above doesn't show the state of the pipeline when addi is in the MEM stage, for example, r3 still contains the old value, not the value specified by the slt instruction. The register values are:
Location Value Comment


## Problem 3

The CPI can easily be found if each iteration of the loop executes the same way, as happens here after the first iteration. (Assuming no cache misses.) To find the CPI find the number of cycles separating two corresponding points in consecutive iterations. A convenient corresponding point for the code above is the cycle when $l w$ is in the IF stage. This occurs at cycles 2,17 , and 30 . Since iteration one is different than the others, and since it is clear that future iterations will look like iterations 2 and 3 , the corresponding points in iterations 2 and 3 will be used. The number of cycles is $30-17=13$, the number of instructions is 5 , so the CPI is 2.6.

David M. Koppelman - koppel@ee.Isu.edu

# EE 4720 Computer Architecture - HW 3 Solution (Spring 1998) 

## Problem 1, EE 4720 HW 3 Solution

The pipeline execution diagram appears below. With bypass paths many of the stalls present in the hw 2 execution are not present here, but two remain. The add following lw must stall one cycle, and the bneq also stalls one cycle. The bneq instruction stalls, in the ID stage, because when it first arrives, at cycle 8, the value it needs to make a decision (prodcued by the slt instruction) is in the MEM stage. At cycle 9 the needed value (r3) arrives and so the ID stage of bneq can complete. At the end of bneq's ID stage the target address is clocked into PC, and so at cycle 10 the target, lw, starts IF. The total delay for the branch here is two cycles.
The execution of a single instruction uses just one line, and MEM is abbreviated to ME. Abandoned instruction are shown in gray.

```
Cycle 0 1 0 2 3 4 4 5 5 6 7 7 8 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
add IF ID EX ME WB
add IF ID EX ME WB
Loop:
lw IF ID EX ME WB IF ID EX ME WB
add IF ID -> EX ME WB IF ID -> EX ME WB
slt IF -> ID EX ME WB IF -> ID EX ME WB
addi IF ID EX ME WB IF ID EX ME WB
bneq IF ID -> EX ME WB IF ID -> EX ME WB
i1
IF -> ID EX ME WB IF -> ID EX ME WB
```

The CPI is based on corresponding points in consecutive iterations, here when the first instruction (lw) is in IF, the occurs at cycle 2 and cycle 10. The average instruction execution time (CPI) is thus ( $10-2$ )/5 $=1.6 \mathrm{CPI}$.

Problem 2, EE 4720 HW 3 Solution

As illustrated in the assignment, the branch condition is read from the register file and so branches must stall until the value they need is written or in the WB stage (assuming simultaneous read and write). As was done for arithmetic and logical instructions, bypass paths can be added to the ID stage. For the code fragment in problem 1, bypass paths from MEM to ID would be needed. A path from EX to ID would help when the register is written by an instruction immediately preceding the branch (although it might lengthen critical paths):

```
slt r3, r1, r2
bneq r3, TARGET
```

With the forwarding paths the branch could determine if r3 were zero in cycle 8, avoiding the stall.

Problem 3, EE 4720 HW 3 Solution
Top Previous

The key here is to re-write the code to avoid the stalls. The stall after lw can be avoided by placing some instruction between lw and the add that uses the fetched result, one possibility is
the addi:


The next problem is, what to put in the branch delay slot. There seems to be no way to fill it by just rearranging the instructions or filling it with addi, which would eliminate one bubble, but open another:

```
add r2, r0, r0 ! Clear sum register.
add r5, r10, rll ! Set r5 to first element.
LOOP:
    lw r6, 0(r5)
        add r2, r2, r6
        slt r3, r2, r4
        bneqd r3, LOOP
        addi r5, r5, #4 ! In delay slot, part of loop.
```

(bneqd is the delayed branch mnemonic.) If just rearranging instructions won't work, then adding some might:

|  | add | r2, r0, r0 | Clear sum register. |
| :---: | :---: | :---: | :---: |
|  | add | r5, r10, rll | ! Set r5 to first element. |
|  | lw | r6, 0(r5) | ! Added instruction. |
| LOOP: |  |  |  |
|  | addi | r5, r5, \#4 |  |
|  | add | r2, r2, r6 |  |
|  | slt | r3, r2, r4 |  |
|  | bneqd | r3, L00P |  |
|  | lw | r6, 0(r5) | ! In delay slot, part of loop. |

In the code above, lw has been duplicated. The first lw initializes r6, the second one executes at the end of the loop. The code above executes bubble free, with a CPI of 1 . The pipeline execution diagram appears below:

```
Cycle 0 1 0 2 3 3 4 5 5 6 7 7 8 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
add IF ID EX ME WB
add IF ID EX ME WB
lw IF ID EX ME WB
Loop:
addi IF ID EX ME WB IF ID EX ME WB
add IF ID EX ME WB IF ID EX ME WB
slt IF ID EX ME WB IF ID EX ME WB
bneqd IF ID EX ME WB IF ID EX ME WB
lw
                                    IF ID EX ME WB IF ID EX ME WB
```

A repeating iteration starts on cycle 3 and ends before cycle 8 , for a total of 5 cycles. Containing 5 instructions, the CPI is 1.0 .

# EE 4720 Computer Architecture - HW 4 Solution (Spring 1998) 

Problem 1, EE 4720 HW 4 Solution

The most informative notation would indicate both the hardware the instruction were in and how far along execution it was. (The existing notation shows only how far along execution an instruction is, which unambiguously indicates the hardware only when the initiation interval is one). In one possible solution the location is indicated with a 3-part label. The first part shows the functional unit using a capital letter (A for add, etc.). The second part shows which part of the functional unit the instruction is in, in parenthesis. The third part shows the execution step. For example, $\mathrm{A}(2) 2$ shows an instruction in the second adder segment, which is also the second step of execution. (When the initiation interval is one the execution unit part and step will always be the same.) As another example, $D(1) 20$ shows an instruction in the one-and-only divide unit part, in the twentieth step of execution.
The notation is used in the pipeline execution below, which runs on the implementation described in problem 2.


Problem 2, EE 4720 HW 4 Solution
The problem did not specify how WAW hazards were to be resolved. They could be resolved by stalling the second multiply so it writes after the divide or by cancelling the divide when the multiply is either in ID or WB. (The divide can be canceled because no instruction reads the value it produces.) The second approach will be used since it does not stall following instructions. The execution diagram appears below, using the notation from part 1

| Time 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| div f3, f4, f5 IF | ID | D(1)1 | D(1)2 | D(1)3 | D(1)4 | D(1) 5 | D(1)6 | D(1)7 | D(1)8 | x |  |  |
| mul f0, f1, f2 | IF | ID | M(1) 1 | M(1)2 | M(2)3 | M(2) 4 | MEM | WB |  |  |  |  |
| mul f3, f6, f7 |  | IF | ID | ----> | M(1)1 | $\mathrm{M}(1) 2$ | $\mathrm{M}(2) 3$ | M(2) 4 | MEM | WB |  |  |
| sub f8, f9, f10 |  |  | IF | ---> | ID | A(1) 1 | A(2) 2 | A(3) 3 | -- | MEM | WB |  |
| mul f11, f0, f12 |  |  |  |  | IF | ID | M(1)1 | M(1)2 | M(2)3 | M(2)3 | MEM | WB |

The execution above has two stalls, one in cycle 4 due to the multiply-unit structural hazard, the other in cycle 9 due to the memory stage structural hazard.

Problem 3, EE 4720 HW 4 Solution
Top Previous Next
As with the previous problem, WAW hazards will be handled by cancelling the first instruction writing a register when the second instruction writing that register is in the WB stage.

| Time 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 111 | 12 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| div f3, f4, f5 IF | ID | D(1) 1 | D(1)2 | D(1)3 | D(1)4 | D(1) 5 | D(1) 6 | D(1)7 | D(1)8 | x |  |  |  |
| mul f0, f1, f2 | IF | ID | M(1)1 | M(1)2 | M(2)3 | M(2)4 | MEM | WB |  |  |  |  |  |
| mul f3, f6, f7 |  | IF | ID | ----> | M(1)1 | M(1)2 | M(2)3 | M(2) 4 | MEM | WB |  |  |  |
| sub f8, f9, f10 |  |  | IF | > | ID | ----> | A(1) 1 | A(2) 2 | A(3) 3 | MEM | WB |  |  |
| mul f11, f0, f12 |  |  |  |  | IF | ----> | ID | M(1) 1 | M(1) 2 | M(2)3 | M (2) 3 | 3 MEM | WB |

Notice that the ID-stage stall delays the third multiply by one cycle.

Problem 4, EE 4720 HW 4 Solution
Top Previous Next
In the execution below, the integer unit uses reservation stations 3 and 4. Branch instructions are shown stopping after ID since they don't do anything useful after that and so reservation stations are not shown.
(If the branch had to wait for $|\mathrm{r} 1|$ it would sit in a reservation station which would be shown.) The execution is shown until cycle 25 , two cycles after the second multiply writeback.

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| addi | IF | ID | 3:EX | 3:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LOOP: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ld |  | IF | ID | 4:EX | 4:ME | 4:WB | IF | ID | 4:EX | 4:ME | 4:WB | IF | ID | 4:EX | 4:ME | 4:WB | IF | ID | 4:EX | 4:ME | 4:WB |  |  |  |  | IF |
| subi |  |  | IF | ID | 3:EX | ---> | 3:WB | IF | ID | 3:EX | -> | 3:WB | IF | ID | 3:EX | -> | 3:WB | IF | ID | 3:EX | ---> | 3:WB |  |  |  |  |
| mul |  |  |  | IF | ID | 1:M1 | 1:M2 | 1:M3 | 1:M4 | 1:M5 | 1:M6 | 1:M7 | 1:M8 | 1:M9 | 1:WB |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | IF | ID | 2:RS | 2:RS | 2:RS | 2:RS | 2:M1 | 2:M2 | 2:M3 | 2:M4 | 2:M5 | 2:M6 | 2:M7 | 2:M8 | 2:M9 | 2:WB |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID | 1:RS | 1:RS | 1:RS | 1:RS | 1:RS | 1:RS | 1:RS | 1:RS | 1:M1 | 1:M2 | 1:M3 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID |  |  |  |  | 2:RS | 2:RS |
| bneq |  |  |  |  | IF | ID |  |  |  | IF | ID |  |  |  | IF | ID |  |  |  | IF |  |  |  | -> | ID |  |
| I1 |  |  |  |  |  | IF |  |  |  |  | IF |  |  |  |  | IF |  |  |  |  |  |  |  |  | IF |  |

Before reservation stations run out, each iteration of the loop above takes five cycles, after they run out nine cycles per iteration will be needed. The reservation stations allow the "integer" part of the loop to get several cycles ahead of the floating point part.

Problem 5, EE 4720 HW 4 Solution
Top Previous
In the solution to the previous problem, in cycle 9 the multiply instruction is in ID for the second time. It can move into the execute stage only if the result from the previous iteration is ready. Since the first multiply is in ID in cycle 4, the multiply unit would have to produce a result in 5 (or fewer) cycles to avoid delaying the second multiply. If the execution of the multiply is delayed by any amount, all reservation stations will eventually be used up. A multiply unit that produces a result in 5 cycles has a latency of 4.

# EE 4720 Computer Architecture - HW 5 Solution (Spring 1998) 

## Problem 1, EE 4720 HW 5 Solution

The execution is shown below. The first iteration takes five cycles (5-0), the second iteration takes 7 (12-5). The second iteration takes two cycles longer because of the first branch's onecycle delay and the true dependency between the subi and bneq instructions.


Problem 2, EE 4720 HW 5 Solution

The loop in the program above is quite inefficient. There are many bubbles, and many instructions are cancelled because of the branch delay following the taken branches. When branching from an instruction other than the last in an aligned group (bneq and beqz are the third, $j$ is the first) instructions following the CTI (branch or jump) are cancelled. Finally, when branching to TARG, which is not the first instruction in an aligned group, instructions before TARG are cancelled. Clearly, this loop makes poor use of our 4-way superscalar machine. How poor?
An iteration of the loop takes 5 cycles when the branch is not taken, and takes 7 cycles when taken. Either way, the loop has 5 instructions. These numbers can be used to compute the CPI of the execution of a large number of iterations because they do not depend on what happened in the previous iterations. There can be inter-iteration hazards when long latency functional units are used because the result of an operation started in one iteration might be needed in the next, or the functional unit itself might be needed. These problems do not occur here.
The first branch in the loop is taken if the value read from memory is zero. According to the problem statement $30 \%$ of those words are zero. The CPI is found by taking a weighted average as shown:
$\mathrm{CPI}=0.3 \mathrm{t}_{\text {taken }} \mathrm{IC}_{\text {taken }}+0.7 \mathrm{t}_{\text {not taken }} \mathrm{IC}_{\text {not taken }}$

$$
=0.37 / 5+0.75 / 5
$$

$=1.12$
Keep in mind that the ideal CPI for a 4-way machine is 0.25 .

Problem 3, EE 4720 HW 5 Solution
Top Previous Next

## Branch Delay Slots

Branch delay slots would have very little effectiveness for the code above. Branch delay slots might seem like a promising enhancement for superscalar machines because a branch can have a large delay: 7 instructions if the branch is the first instruction in an aligned group. With a 7-delay-slot branch these bubbles would be avoided. Branch delay slots are only useful if they can be filled with something other than nop's, in the code above (without major re-writing) the only candidate for a branch delay slot is the addi instruction. If the loop is unrolled there might be more opportunities for filling delay slots.

## Dynamic Scheduling

Dynamic scheduling would have very little effectiveness for the code above, though it would be more useful on other code. Dynamic scheduling helps avoid stalls due to name dependencies and allows instructions to start out of order. In the code above the stalls are due to true dependencies (between lw and addi, for example), and control dependencies. Without branch prediction and speculative execution there would be no benefit.

## Branch Target Buffer

A branch target buffer would be moderately effective for the code above, but its execution would still be well below the maximum issue rate. As stated above, the branch execution is slowed due to branches. A branch target buffer would help by eliminating the branch delay slot caused by the beqz, bnez, and jinstructions, the overall effectiveness would depend on the branch prediction accuracy. The bneq branch would probably be correctly predicted because it is only not taken on the last iteration. The behavior of the beqz branch depends on the characteristics of the values read. If the zeros are bunched ( $0,0,0,0,0,1,3,2$ ) then prediction should work well, if they're randomly interspersed the predictor won't work.

## Predicated Execution

Predicated execution would have limited effectiveness Predicated execution can be used to eliminate one or two branches from the loop, thus eliminating wasted issue slots from taken branches. The loop with one branch removed might look like:


The predicate bit is set by the set_c instruction, here that value is being bypassed to the WB stage so there is no additional stall for sub_pn. The branch delay due to the removed branch is
no longer present.

## Loop Unrolling

Loop unrolling can be very effective on the code above. Loop unrolling is effective when several iterations can be overlapped. One problem with loop unrolling is that the number of iterations may not be the multiple of the degree of unrolling. For example, the code above was unrolled 4 times, but the number of iterations may not be a multiple of 4 . The problem is easiest to fix if the number of iterations is known at compile time: the compiler would end the loop at the largest multiple smaller than the number of iterations, and fix-up code following the unrolled loop would perform the remaining few iterations. In the loop below, not only is the number of iterations not known at compile time (presumably), the number of iterations may not be known until the loop finishes its last iteration. (The number of iterations depends on the data fetched.) That's still no problem as long as the it's okay to make unneeded memory accesses.
In the unrolling below the branch in the middle of the loop is eliminated. The sub r4, r4, r1 can be performed whether or not the branch is taken, and seqi and add instructions are used to decrement r5 for four elements at once. The unrolled loop has only a single branch, iteration continues if $r 5$ is positive. When r5 will be zero or negative the loop exits and fix-up code (not shown) re-loads words and adjusts $r 4$. (It's possible that too many elements were subtracted.)
In the unrolled loop values are fetched one iteration ahead of time, so when it finishes at least four extra values will have been fetched.

Using sle r30, r5, r25 the branch condition can be determined one cycle earlier than if a sle r30, r5, r0 instruction were used after sub r5, r5, r25.
There are no stalls to avoid RAW hazards, the only wasted issue slots (bubbles) are due to the branch. An iteration of the unrolled loop takes 6 cycles, for a CPI of $6 / 19=0.316$. The CPI is not useful for comparing with the original loop since the instruction counts are different. A better metric would be cycles per element (an iteration in the original loop). The original loop took a weighted average of $0.75+0.37=5.6$ cycles per element. The unrolled loop processes four elements per iteration, so it takes $6 / 4=1.5$ cycles per element.

| $l w$ | $r 11$, | $0(r 2)$ |
| :--- | :--- | :--- |
| $l w$ | $r 12$, | $4(r 2)$ |
| lw | $r 13$, | $8(r 2)$ |
| lw | $r 14$, | $12(r 2)$ |

```
ULOOP: Assume ULOOP aligned.
    seqi r21, r11, #0 ! Check if elements are zero.
    seqi r22, r12, #0 ! =1 if element is zero, 0 otherwise.
    seqi r23, r13, #0 !
    seqi r24, r14, #0 !
```

    add r15, r11, r12 ! Sum of element values.
    add r16, r13, r14
    add r25, r21, r22 ! Count of zero elements.
    add r26, r23, r24
    add r15, r15, r16 ! Sum of element values.
    add r25, r25, r26 ! Count of zero elements.
    lw r11, 16(r2) ! For next iteration, if needed.
    lw r12, 20(r2) ! For next iteration, if needed.
    sle r30, r5, r25 ! Branch condition.
    sub r5, r5, r25 ! Update r5.
    lw r13, 24(r2) ! For next iteration, if needed.
    ```
lw r14, 28(r2) ! For next iteration, if needed.
    addi r2, r2, #16
    sub r4, r4, r15 ! Maybe over-subtracting. Fix-up code should fix.
    bneq r30, ULOOP
    ! Fix-up code starts. (not shown)
    ! Need to re-load last four words and adjust r4 and r5
    ! Fix up code ends.
    and r4, r4, r6
    or r4, r4, r7
    sw 0(r8), r4
    addi r8, r8, #4
    jr r31
```

Problem 4, EE 4720 HW 5 Solution

Instructions can be rearranged so that all conditionally executed instructions are in the first bundle. There is a dependency between the second addi and and and slt, so they must be placed in separate bundles. There are no other instructions to put in the second and third bundles, so they are padded with nops.
bx r1, ! r1, ! r1
add r2, r3, r4
add r2, r3, r5
addi r7, r7, \#1
sub r6, r6, r2
addi r8, r2, \#12
nop
nop
slt r1, r8, r9
and r10, r8, rll
nop
nop

# EE 4720 Computer Architecture - HW 6 Solution (Spring 1998) 

Problem 1, EE 4720 HW 6 Solution

In the execution below, the branch outcome is not available when the branch instruction reaches ID, at cycle 4, and so it waits in a reservation station. The system predicts the branch as taken and so fetching starts at the branch target in cycle 5. (If the system had an IF-stage branch target buffer it would have been possible to fetch the target at cycle 4, eliminating a cycle of delay.) Instructions at the target are executed speculatively, that is they don't do anything that can't be undone. The branch condition is finally available at cycle 11, the speculatively executed instructions are deleted from the reorder buffer and fetching starts at the target in cycle 12.

```
sub r4, r5, r6 [10cllllllllllllllllll
multf f4, f5, f6 IF ID 1:M1 1:M2 1:M3 1:M4 1:M5 1:M6 1:M7 1:WB
addf f0, f1, f2 IF ID 6:A1 6:A2 6:A3 6:A4 6:WB
eqf f0, f3 IF ID 7:RS 7:RS 7:RS 7:A1 7:A2 7:A3 7:A4 7:WB
bfpt TARG IF ID 2:RS 2:RS 2:RS 2:RS 2:RS 2:RS 2:BR 2:WB
add r1, r2, r3
sub r4, r5, r6
TARG:
    and r1, r2, r3
    or r4, r5, r6
        IF ID 3:EX 3:WB
```

Execution notes: With dynamic execution load and store instructions use their own functional units, so other instructions do not pass through the MEM stage. The stage labeled WB is actually a write to the common data bus which will write into any waiting reservation stations or functional units (as would happen in cycle 7 when addf completes) and also into the reorder buffer. The data is written to the register file when the instruction is retired from the reorder buffer.
The reorder buffer contents are shown by cycle below. An entry for a cycle includes instructions that are inserted during that cycle and instructions that will be removed at the end of the cycle. The reorder buffer fills until cycle 9 as instructions wait for multf to complete, after that instructions up to the branch retire one-by-one. (To be retired [as opposed to being deleted] from the reorder buffer an instruction must be the least-recent entry [the bottom entry as illustrated below] and must have completed execution [indicated by a $\downarrow$ ]). The branch condition is resolved in cycle 11, since the branch was mispredicted the speculatively executed and and or instructions are removed from the reorder buffer and fetching is restarted at the add instruction. The pipeline and reorder buffer might contain instructions before multf and after sub and or, these are omitted for clarity and to save time.

| Cycle <br> Reorder <br> Buffer <br> Contents | $\begin{aligned} & \mathbf{0} \\ & \text { Empty }^{*} \text { ) } \end{aligned}$ | 1 multf | 2 <br> addf <br> multf | 3 <br> eqf <br> addf <br> multf | 4,5 <br> bfpt <br> eqf <br> addf <br> multf | 6 and bfpt eqf addf multf | 7 <br> or <br> and <br> bfpt <br> eqf <br> addf $\checkmark$ <br> multf | 8 <br> or <br> and $\checkmark$ <br> bfpt <br> eqf <br> addf $\checkmark$ <br> multf | 9 <br> or <br> and $\checkmark$ <br> bfpt <br> eqf <br> addf $\checkmark$ <br> multf $\checkmark$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |


| Cycle <br> Reorder <br> Buffer <br> Contents | $\mathbf{1 0}$ <br> or $\checkmark$ <br> and $\checkmark$ <br> bfpt <br> eqf <br> addf $\checkmark$ | $\mathbf{1 1}$ <br> or $\checkmark$ <br> and $\checkmark$ <br> bfpt <br> eqf $\checkmark$ | 12 <br> bfpt $\checkmark$ | 13 <br> add | 14 <br> sub <br> add | sub <br> add $\checkmark$ | 15 <br> sub $\checkmark$ | 16 <br> (Empty $^{*}$ ) |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

*Empty of any instructions included above.
$\checkmark$ Completed execution.

## Problem 2, EE 4720 HW 6 Solution

The branch behavior here is very easy to determine: the branch will not be taken in the first 16 iterations then taken in the next 16 iterations, after which the cycle repeats.

```
add r1, r0, r0
L00P:
    andi r2, r1, #0x10
    beqz r2, CONTINUE
    addi r3, r3, #1
CONTINUE:
addi r1, r1, #1
    j LOOP
```

To determine the branch prediction accuracy the value of the branch history counter at each iteration is needed. The counter value is initially 0 , it's incremented in the first 16 iterations though it saturates at 7 . During the second 16 iterations the counter is decremented, reaching zero by the beginning of iteration 23, and remaining at zero through iteration 31. Note that these are the values of the counter before the branch executes, which of course is used for prediction. As with the branch outcome, the branch history counter value also repeats. The table below shows the counter values, prediction, outcome, and number of correct predictions by iteration. At the beginning of iteration 32 the BHT counter is zero, since the branch behavior repeats every 32 cycles the prediction behavior will also repeat. The branch prediction accuracy is the number of correct predictions divided by the number of predictions, $24 / 32=0.75$.
Iteration Count Value Prediction Outcome Num. Correct

| $0-4$ | $0-4$ | NT | T | 0 |
| :--- | :--- | :--- | :--- | :--- |
| $5-7$ | $5-7$ | T | T | 3 |
| $8-15$ | 7 | T | T | 8 |
| $16-18$ | $7-5$ | T | NT | 0 |
| $19-23$ | $4-0$ | NT | NT | 5 |
| $24-31$ | 0 | NT | NT | 8 |

Problem 3, EE 4720 HW 6 Solution
The solution is shown in the figure below, with additions in red. Not shown are details of the stack implementation, clocking inputs for the BHT, hardware to support context switches, prediction and target address hardware.


The key to solving the problem is choosing the correct address for the branch history table (BHT) because the branch predictor matches a particular history (counter value) with a particular address. Normally, the BHT is addressed with the branch instruction's address, so that each branch instruction has its own history (ignoring collisions). In the problem a branch predictor is to be designed that also takes into account the calling procedure. Therefore, the BHT address must be constructed using something that could identify the calling procedure; that something is the return address that is saved in r31 by the jal and jalr instructions. (The return address is available in the IF/ID. NPC register when the call instruction is in ID.)
In the solution above, part of the most-recent calling address is available at the ToS (top of stack) output of the Call Stack box. That is combined with the branch instruction address to construct a BHT address. Because DLX instructions are aligned, bits 0 and 1 from both addresses are omitted (since they would always be zero, and if included would leave $75 \%$ of the BHT unused). Four bits (2:5) are taken from the calling procedure address and eight bits from the branch instruction (2:9), for a total of 12 bits. Only four calling-address bits are included because the number of different places that a procedure can be called from would be much smaller than the number of branch instructions in the program. As with an ordinary BHT and especially with a correlating predictor there will be collisions in the BHT, that is, multiple branch instructions will use the same entry. This is acceptable because it doesn't happen too often and the consequences of a collision is only misprediction, not incorrect execution. Note that multiple branch instructions with identical address bits $2: 9$ would not collide in the BHT if the calling address bits 2:5 were all different.
The box labeled Caller Stack stores bits 2:5 of the addresses of assumed procedure call instructions. Whenever a jal or jalr instruction is in ID, the instruction address is pushed on the stack, it will appear at the ToS (top of stack) output in the next cycle. When a jr r31 instruction, the assumed procedure return, is in ID the stack is popped. (That is, the most recently pushed address is removed and the next-most-recent one is moved to the top-of-stack position.) If the stack is empty the empty line is asserted, which might trigger a trap. (The operating system might fill the stack with data that had overflowed when too many items were pushed.)

As with all chapter-3 DLX implementations, there is no actual reason to predict branches since the branch condition is available early. The chapter-3 DLX implementation was used above for simplicity.

Problem 4, EE 4720 HW 6 Solution

The connections and address locations are shown below.


## 76 Spring 1997 Solutions

Problem 1: (2 pts) Just plug the run times into these equations

$$
\mathrm{AM}=\frac{1}{n} \sum_{i=1}^{n} t_{i} \quad \mathrm{HM}=\left(\frac{1}{n} \sum_{i=1}^{n} \frac{1}{t_{i}}\right)^{-1} \quad \mathrm{GM}=\sqrt[n]{\prod_{i=1}^{n} t_{i}}
$$

to obtain $42.6,13.8$, and 27.7 for the arithmetic, harmonic, and geometric mean (respectively) of the program run times on the base machine and 18.6, 4.8, 11.7 for the arithmetic, harmonic, and geometric mean (respectively) of the program run times on the test machine.

Problem 2: (3 pts) The key phrase in the problem is, "each type of program is of equal importance." This means that, say, if machine $A$ runs the compilers, A1, A2, and A3, $10 \%$ faster, and machine $B$ runs the databases $10 \%$ faster, both would have the same TigerMark rating (assuming they ran the other programs as fast as the base machine). A common, and incorrect, solution was taking the geometric mean of the speedups of each program. That is,

$$
\begin{array}{r}
\left(\frac{t_{\mathrm{A} 1}(\text { Base })}{t_{\mathrm{A} 1}(\text { Test })} \times \frac{t_{\mathrm{A} 2}(\text { Base })}{t_{\mathrm{A} 2}(\text { Test })} \times \frac{t_{\mathrm{A} 3}(\text { Base })}{t_{\mathrm{A} 3}(\text { Test })} \times \frac{t_{\mathrm{B} 1}(\text { Base })}{t_{\mathrm{B} 1}(\text { Test })} \times \frac{t_{\mathrm{B} 2}(\text { Base })}{t_{\mathrm{B} 2}(\text { Test })} \times\right. \\
\left.\frac{t_{\mathrm{C} 1}(\text { Base })}{t_{\mathrm{C} 1}(\text { Test })} \times \frac{t_{\mathrm{C} 2}(\text { Base })}{t_{\mathrm{C} 2}(\text { Test })} \times \frac{t_{\mathrm{C} 3}(\text { Base })}{t_{\mathrm{C} 3}(\text { Test })} \times \frac{t_{\mathrm{C} 4}(\text { Base })}{t_{\mathrm{C} 4}(\text { Test })}\right)^{1 / 9}
\end{array}
$$

Because the number of programs of each type is different a $10 \%$ change in each program of one type will have a different impact than a $10 \%$ change in each program of another type, which is not acceptable in this case.

In one correct solution, the average speedup of programs of each type is computed, yielding three speedups. These three speedups are averaged to get the TigerMark. Symbolically,

$$
\begin{array}{r}
\mathrm{TM}(\text { Test })=\frac{1}{3}\left(\frac{1}{3}\left(\frac{t_{\mathrm{A} 1}(\text { Base })}{t_{\mathrm{A} 1}(\text { Test })}+\frac{t_{\mathrm{A} 2}(\text { Base })}{t_{\mathrm{A} 2}(\text { Test })}+\frac{t_{\mathrm{A} 3}(\text { Base })}{t_{\mathrm{A} 3}(\text { Test })}\right)+\frac{1}{2}\left(\frac{t_{\mathrm{B} 1}(\text { Base })}{t_{\mathrm{B} 1}(\text { Test })}+\frac{t_{\mathrm{B} 2}(\text { Base })}{t_{\mathrm{B} 2}(\text { Test })}\right)+\right. \\
\left.\frac{1}{4}\left(\frac{t_{\mathrm{C} 1}(\text { Base })}{t_{\mathrm{C} 1}(\text { Test })}+\frac{t_{\mathrm{C} 2}(\text { Base })}{t_{\mathrm{C} 2}(\text { Test })}+\frac{t_{\mathrm{C} 3}(\text { Base })}{t_{\mathrm{C} 3}(\text { Test })}+\frac{t_{\mathrm{C} 4}(\text { Base })}{t_{\mathrm{C} 4}(\text { Test })}\right)\right)
\end{array}
$$

Problem 3: Another possible benefit is code density. That is, it's quite likely that the space needed for the single new instruction is less than the five instructions it replaces, so less space is needed to store the program.

One drawback is that the benefit does not justify the cost. The new instruction may only be rarely used while the hardware cost might be substantial.

Another drawback is that it might be difficult to quickly execute the new instruction on future implementations. The instruction might do $100 \%$ of what the programmer wants and $10 \%$ more. The $10 \%$ more might preclude a faster future implementation. A sequence of simpler instructions might do exactly what the programmer wants and could be executed quickly.

Wrong answers:
(Benefit) Lower instruction count means lower execution time. This is wrong because it implies that fewer instructions will always lead to improved performance. (In this case it does.)
(Drawback) Lower instruction execution rate (MIPs). This is wrong because lower or higher MIPs does not mean lower or higher performance in general, and in this case.
(Drawback) Higher CPI. This is wrong because lower or higher CPI does not mean lower or higher performance in general, and in this case. Note that MIPs $=10^{6} / \mathrm{CPI}$.

Problem 4: ( 3 pts ) For implementation $A$ and compiler I, average instruction execution time is $2.625 \mathrm{CPI}=2.625 \mu \mathrm{~s}$ (either answer is acceptable). Total execution time is 21.0 ms . For implementation $A$ and compiler II, average instruction execution time is $2.595 \mathrm{CPI}=2.595 \mu \mathrm{~s}$. Total execution time is 21.8 ms . In these cases compiler II had the lower CPI (good) but the higher execution time (bad), and so CPI is not a good predictor.

For implementation $B$ and compiler I, average instruction execution time is $2.625 \mathrm{CPI}=$ $2.625 \mu \mathrm{~s}$ (either answer is acceptable). Total execution time is 21.0 ms . For implementation $B$ and compiler II, average instruction execution time is $2.405 \mathrm{CPI}=2.405 \mu \mathrm{~s}$. Total execution time is 20.2 ms . In these cases compiler II had the lower CPI and the lower execution time. CPI does agree with execution time here.

Wrong answer explained: $\mathrm{CPI}=(2+2+3) / 3=(3+1+3) / 3=7 / 3$ is wrong because it gives equal weight to all instruction categories. Average instruction execution time (CPI) is based on the mix of instructions actually executed, so frequently executed instructions should be counted more than infrequent ones.

## EE 4720 Computer Architecture - HW 3 Solution

## Problem 1

Conventional implementation. If r10=0, skip every other instruction:

```
beqz r10,A
add r3,r2,r1
A:
addi r4,r5,#12
beqz r10,B
addi r4,r6,#13
B:
addi r5,r4,#14
beqz r10,C
addi r5,r7,#15
C:
    addi r8,r5,#16
    beqz r10,D
    addi r8,r2,#17
D:
addi r9,r8,#18
```

(Note that a good programmer would implement the code fragment above differently.)
Notice that all branches use the some register. That makes it easy to replace all branches with a single PM:

```
pm R10, 0xFF, 0xAA
add r3,r2,r1
addi r4,r5,#12
addi r4,r6,#13
addi r5,r4,#14
addi r5,r7,#15
addi r8,r5,#16
addi r8,r2,#17
addi r9,r8,#18
```


## Problem 2

Consider execution of the first four instructions of the conventional code fragment when r10=0. The add instruction, which started execution under the predict-not-taken assumption, is abandoned. The addi instruction starts three cycles after the IF for the add. The process repeats with the second beqz instruction.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |
| :--- | :---: | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| beqz | IF | ID | EX | MEM | WB |  |  |  |  |
| add |  | IF | ID | - |  |  |  |  |  |
| A: |  |  |  |  |  |  |  |  | IF |
| addi |  |  |  |  | ID | EX | MEM WB |  |  |
| beqz |  |  |  |  |  | IF | ID | EX | MEM WB |

The time between the beqz instructions is five cycles, the number of instructions executed is 2 . This sequence repeats four times in the entire code fragment, and the sequence takes another four cycles to complete (for the pipeline to drain. So:
$\mathrm{IC}=2 \times 4=8$.
$\mathrm{t} /$ cycles $=2 \times 5+4=24$.
$\mathrm{CPI}=24 / 8=3$

With the PM instruction there are no stalls, so the total execution time is $9+4=13$ cycles. When $\mathrm{r} 10=0$, four instructions are nulled, so $\mathrm{IC}=5$. Then
$\mathrm{CPI}=13 / 5=2.6$
The execution time is almost twice as good (which is what counts), and the CPI is slightly better.

## Problem 3

The solution below is incomplete: it will not work properly with control-transfer instructions, nor will it work if the pipeline stalls. (The complete solution might give away a future homework assignment.)

As shown in red below, add an eight-bit shift register to the EX stage. When a PM instruction is in EX, the appropriate 8-bit execution mask (from the immediate portion of the ID/EX latch) is loaded into the shift register.

At every cycle, the least significant bit of the shift register is checked. If it's zero, the instruction in the EX stage is nulled by replacing it with a NOP. (This works because instructions do not change "state" until the MEM and WB stages.) At the end of the cycle the shift register is shifted right (removing the least significant bit) with a 1 shifted into the left side.


## Problem 4

If loads were allowed, then operation of the shift register would be complicated by stall cycles (but this would not be too great a problem). Allowing branches would make programming tricky since instructions at the branch target would be affected by the execution mask. The PM description did not describe what would happen if a PM were encountered within 8 instructions of another PM. One possibility is that the mask in the second PM could overwrite the first, assuming the second PM was not nullified itself.

Things would be simplest if only arithmetic (including logical and compare) instructions were allowed after a PM. So a reasonable restriction is that the PM works as described above until a non-arithmetic instruction is encountered, in which case the remainder of the mask is set to ones (i.e., the PM instruction is cancelled) or reloaded with a new mask if the non-arithmetic instruction is another PM.

David M. Koppelman - koppel@ee.lsu.edu

## EE 4720 Computer Architecture - HW 4 Solution

## Problem 1

In the timing diagrams below the active PM mask bit is indicated for each cycle. In the solution to homework 3, instructions are nullified (replaced with NOPs) in the EX stage; the segments used by nullified instructions are surrounded by X's. Note that the mask bit is checked in the EX stage and that because of pipeline stalls, mask bits become misaligned with instructions. For example, in the execution below, because the SUB instruction stalls it "misses" its zero.

With R4 = 0:

| Time: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Mask: |  |  | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| PM | IF | ID | EX | MEM | WB |  |  |  |  |  |
| ADD |  | IF | ID | XEX XMX | XWX |  |  |  |  |  |
| LW |  | IF | ID | EX | MEM WB |  |  |  |  |  |
| SUB |  |  | IF | ID |  | EX | MEM WB |  |  |  |

Below, the SUB instruction executes normally (albeit with a stall) because the NOP was inserted in the EX stage while SUB was stalled in ID.

With R4 = 1

| Time: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Mask: |  |  | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| PM | IF | ID | EX | MEM | WB |  |  |  |  |  |
| ADD |  | IF | ID | EX | MEM | WB |  |  |  |  |
| LW |  | IF | ID | XEX XMX | XWX |  |  |  |  |  |
| SUB |  |  | IF | ID |  | EX | MEM WB |  |  |  |

Note that, if the controller doesn't know that LW will be nullified, it will have to stall SUB to avoid the possible RAW hazard. (The stall is unnecessary in this case.)

For R4 = 1, R5 = 0, Mask = 00010011

| Time: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Mask: |  |  |  | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| PM | IF | ID | EX | MEM | WB |  |  |  |  |  |  |
| ADD |  | IF | ID | EX | MEM | WB | IF | ID | XEX | XMX | XWX |
| BEQZ |  |  | IF | ID | EX | MEM | WB | IF | ID | XEX XMX XWX |  |
| SUB |  |  |  | IF | ID |  |  |  | IF | ID | XEX XMX XWX |

For R4 = 1, R5 = 1, Mask $=00010011$

| Time: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Mask: |  |  |  | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| PM | IF | ID | EX | MEM | WB |  |  |  |  |  |  |
| ADD |  | IF | ID | EX | MEM | WB |  |  |  |  |  |
| BEQZ |  |  | IF | ID | EX | MEM | WB |  |  |  |  |
| SUB |  |  |  | IF | ID | XEX XMX XWX |  |  |  |  |  |

For R4 = 0, R5 = 1 or R5 = 0, Mask $=11111100$

| Time: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Mask: |  |  |  | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| PM | IF | ID | EX | MEM | WB |  |  |  |  |  |  |
| ADD |  |  | IF | ID | XEX XMX | XWX |  |  |  |  |  |


| BEQZ | IF | ID | XEX XMX XWX |
| :--- | :--- | :--- | :--- | :--- |
| SUB |  | IF | ID EX MEM WB |

## Problem 2

Use a shift register in which all all bits are available, not just the bit at the end of the register. And these bits together, call the result DONE. DONE indicates that the current and next seven instructions will execute. Generate a second signal, PM_ILL, by detecting an LW or CTI instruction opcode in the EX stage. Then PM_VIOL is the and of PM_ILL and NOT DONE. The exception can easily be made precise since it is detected while the the faulting instruction is in the EX stage, and so the following instructions can easily be abandoned since they are only in the IF and ID segments.

These changes are shown below; blue indicates changes for homework 3 and red indicates changes for this problem. Note the exaggerated inversion bubble at the shift register shift input.
IF
ID
EX
MEM
WB


## Problem 3

For loads and branches to be properly nullified, the shift register must not be clocked when bubbles are passing though the EX stage. The simplest solution is to assume that the controller can provide a "bubble bit" in the EX stage; when such a bit is 1 the shift register is not clocked.

If a bubble bit is not already provided, it can be synthesized by checking for the two stall conditions: a load instruction with a RAW hazard, and a taken branch or any other CTI. A bubble shift register is set to the number of stall cycles, one bit for a load, and three for a taken branch. The register is either loaded, as described above, or
instruction reading registers in the ID stage. If the register written by the load is the same as either of the registers read by the instructions in ID, then one bit in the bubble register is set. Instructions using one and two source registers need to be distinguished.

CTIs can be detected in the EX stage. If instruction in EX is a jump, jump/link, or a taken branch, three bits in the bubble shift register are set.

These changes are shown below; blue indicates changed for homework 3 and red indicates changes for this problem. The solution below assumes that
shifted. If the bit out is one, then the PM shift register is not shifted.

The load stall is detected by checking for a load instruction in the EX stage and any

NOPs are inserted into the IR in place of the stalled instructions. What if that assumption is not correct?

IF
ID
EX
MEM
WB


ECE 4720
HOME PAGE HOME PAGE

## EE 4720 Computer Architecture - HW 6 Solution

## Problem 1

An illustritive program and its execution is shown for each entry in the table. The "Lat" row shows the cycles counted in the latency. Note that the floating-point units use their own memory and writeback stages, called MF and WF, respectively.

## Program for Figure 4.2 Entry 1

```
addd f0, f2, f4
addd f6, f0, f2
```

Timing:

| Time 0 | $\mathbf{1}$ | $\mathbf{2}$ | $\mathbf{3}$ | $\mathbf{4}$ | $\mathbf{5}$ | $\mathbf{6}$ | $\mathbf{7}$ | $\mathbf{8}$ | $\mathbf{9}$ | $\mathbf{1 0}$ | $\mathbf{1 1}$ |
| :--- | :--- | :--- | :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Lat |  |  | $\mathbf{1}$ | $\mathbf{2}$ | $\mathbf{3}$ |  |  |  |  |  |  |
| addd | IF | ID | $\mathrm{A}_{1}$ | $\mathrm{~A}_{2}$ | $\mathrm{~A}_{3}$ | $\mathrm{~A}_{4}$ | MF | WF |  |  |  |
| addd | IF | ID |  |  |  | $\mathrm{A}_{1}$ | $\mathrm{~A}_{2}$ | $\mathrm{~A}_{3}$ | $\mathrm{~A}_{4}$ | MF | WF |

## Program for Figure 4.2 Entry 2

```
addd f0, f2, f4
sd 0(r1), f0
```

Timing:

| Time 0 | $\mathbf{1}$ | $\mathbf{2}$ | $\mathbf{3}$ | $\mathbf{4}$ | $\mathbf{5}$ | $\mathbf{6}$ | $\mathbf{7}$ | $\mathbf{8}$ | $\mathbf{9}$ | $\mathbf{1 0}$ | 11 |  |
| :--- | :--- | :--- | :--- | :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Lat |  |  |  | 1 | 2 |  |  |  |  |  |  |  |
| addd | IF | ID | $\mathrm{A}_{1}$ | $\mathrm{~A}_{2}$ | $\mathrm{~A}_{3}$ | $\mathrm{~A}_{4}$ | MF | WF |  |  |  |  |
| sd |  | IF | ID | EX |  |  | MEM WB |  |  |  |  |  |

Note that store stalls in the EX stage.
Program for Figure 4.2 Entry 3

```
ld f2, 0(r1)
addd f0, f2, f4
```

Timing:

| Time | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{2}$ | $\mathbf{3}$ | $\mathbf{4}$ | $\mathbf{5}$ | $\mathbf{6}$ | $\mathbf{7}$ | $\mathbf{8}$ | $\mathbf{9}$ | $\mathbf{1 0}$ | $\mathbf{1 1}$ |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Lat |  |  |  | $\mathbf{1}$ |  |  |  |  |  |  |  |  |
| ld | IF | ID | EX | MEM | WB |  |  |  |  |  |  |  |
| addd |  | IF | ID |  | $A_{1}$ | $A_{2}$ | $A_{3}$ | $A_{4}$ | MF | WF |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

Program for Figure 4.2 Entry 4

```
ld f2, 0(r1)
addd f0, f2, f4
```

Timing:

| Time 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

Lat

```
ld IF ID EX MEM WB
sd IF ID EX MEM WB
```


## Problem 2

Note: solution based on simplified R4000 issue rules, so timing will not match a real R4000.

| Time | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{2}$ | $\mathbf{3}$ | $\mathbf{4}$ | $\mathbf{5}$ | $\mathbf{6}$ | $\mathbf{7}$ | $\mathbf{8}$ | $\mathbf{9}$ | $\mathbf{1 0}$ | $\mathbf{1 1}$ | $\mathbf{1 2}$ | $\mathbf{1 3}$ | $\mathbf{1 4}$ | $\mathbf{1 5}$ | $\mathbf{1 6}$ | $\mathbf{1 7}$ | $\mathbf{1 8}$ |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| ld | IF | IS | RF | EX | DF | DS | TC | WB |  |  |  |  |  |  |  |  |  |  |  |
| neg |  | IF | IS | RF |  |  | U | S | DF | DS | TC | WB |  |  |  |  |  |  |  |
| add |  |  | IF | IS |  |  | RF |  | U | S+A A+R | R+S | DF | DS | TC | WB |  |  |  |  |
| cgt |  |  |  | IF |  |  | IS |  | RF | IF | U | A | R | DF | DS | TC | WB |  |  |
| add |  |  |  |  |  |  | IF |  | IS |  | RF | U | S+A | A+R | R+S | DF | DS | TC | WB |

## EE 4720 Computer Architecture - HW 7 Solution

## Problem 1

The solution below is partly repeated. The first table shows the entire solution. (You might need to maximize your browser window.) The second table, for convenience, shows the solution starting at cycle 11.

Some points:

- The implementation has no bypass paths, so newly computed operands are not available until the end of the WB cycle.
- Only one FU at a time can write the common data bus (CDB). The second muld is forced to wait a cycle.
- The branch target, ld, is not fetched until after the branch leaves ID.

| Time 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 262 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| muld IF | ID | 5:A1 | 5:A2 | 5:A5 | 5:A4 | 5:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| addd | IF | ID | 3:RS | 3:RS | 3:RS | 3:RS | 3:A1 | 3:A2 | 3:A3 | 3:A4 | 3:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| muld |  | IF | ID | 6:RS | 6:RS | 6:RS | 6:M1 | 6:M2 | 6:M3 | 6:M4 | 6:WB | 6:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| subd |  |  | IF | ID | 4:RS | 4 :RS | 4:RS | 4:A1 | 4:A2 | 4:A3 | 4:A4 | 4:WB | 4:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |
| loop: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ld |  |  |  | IF | ID | 2:EX | 2:MI | 2:WI |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID |  |  |
| subd |  |  |  |  | IF | ID |  |  |  |  |  | 3:RS | 3:RS | 3:A1 | 3:A2 | 3:A3 | 3:A4 | 3:WB |  |  |  |  |  |  |  |  |
| ld |  |  |  |  |  | IF |  |  |  |  |  | ID | 1:EX | 1:MI | 1:WB |  |  |  |  |  |  |  |  |  |  |  |
| subd |  |  |  |  |  |  |  |  |  |  |  | IF | ID | 4:RS | 4:RS |  |  |  | 4:A1 | 4:A2 | 4:A3 | 4:A4 | 4 :WB |  |  |  |
| ld |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID | 2:EX | 2:MI | $2: W B$ |  |  |  |  |  |  |  |  |  |
| subd |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID |  |  |  | 3:RS |  |  |  |  | 3:A1 | 3:A2 | 3:A3 3 |
| subi |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF |  |  |  | ID | 1: EX | 1:MI | 1:WB |  |  |  |  |
| addi |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID | 2:EX | 2:MI | 2:WB |  |  |  |
| bnez |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID |  | 1:EX | 1:MI | 1 WB |  |
| Time 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |  |  |  |  |  |  |  |  |  |
| muld |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| addd 3:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| muld 6:WB | 6:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| subd 4:A4 | 4:WB | 4:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| loop: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ld |  |  |  |  |  |  |  |  |  |  |  | IF | ID |  |  |  |  |  |  |  |  |  |  |  |  |  |
| subd | 3:RS | 3:RS | 3:A1 | 3:A2 | 3:A3 | 3:A4 | 3:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ld | ID | 1: EX | 1:MI | 1:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| subd | IF | ID | 4:RS | 4:RS |  |  |  | 4:A1 | 4:A2 | 4:A3 | 4:A4 | 4:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ld |  | IF | ID | 2:EX | 2:MI | 2:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| subd |  |  | IF | ID |  |  |  | 3:RS |  |  |  |  | 3:A1 | 3:A2 | 3:A3 | 3:A4 | $3: W B$ |  |  |  |  |  |  |  |  |  |
| subi |  |  |  | IF |  |  |  | ID | 1:EX | 1:MI | 1:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| addi |  |  |  |  |  |  |  | IF | ID | 2:EX | 2:MI | 2:WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| bnez |  |  |  |  |  |  |  |  | IF | ID |  | 1:EX | 1:MI | $1: W B$ |  |  |  |  |  |  |  |  |  |  |  |  |

## Problem 2

The modifications to the pipeline are shown below. This solution works in most situations. (See end of description.)
The BHT has a separate read and write port (so that one entry can be read while another is updated). The BHT address is the concatenation of h-m PC bits and mits from the shift register. The BHT entry and the BHT entry address are clocked into the IF/ID latch.

The prediction is made in the ID stage simply by looking at the most significant bit of the BHT entry. An updated BHT entry is generated by adding or subtracting 1 , based on the branch outcome. (In any real branch prediction scheme we would not know the outcome until several cycles later.) Using the address in the IF/ID latch, the updated entry is written to the BHT (at the end of ID) using the address from the latch. (It's important to write the entry to the same address it was read from, of course.)

The shift register is updated at the end of ID. The register shifts only if the "enable" line is asserted.
The solution below would have received full credit, but it does have a problem: If there are two consecutive branches, the shift register value used for the second branch will not be correct. How could this problem be fixed? Also, how would the design be changed if the "taken" signal were available only after ID?


## EE 4720 Computer Architecture - HW 8 Solution

## Problem 1

The original program with line numbers added:

```
1: add r1, r2, r3
2: sub r6, r7, r8
3: lw r10, 0(r20)
4: add r11, r10, r12
5: sub r14, r1, r9
6: add r1, r14, r15
7: sub r16, r17, r18
8: add r19, r21, r22
9: sw 0(r20), r6
```

The instructions are rearranged and placed in groups of 3 for 3VDLX. The first three instructions can be, and are, grouped together because they don't use any values they produce. The next three, on lines 4,5 , and 6 , cannot be grouped together because the value produced on line 5 is needed on line 6 . The inefficient solution is to keep the instructions in the same order and use nops to avoid hazards, starting with the load latency:

```
add r1, r2, r3; sub r6, r7, r8; lw r10, 0(r20)
nop; nop; nop
add r11, r10, r12; sub r14, r1, r9; nop
add r1, r14, r15; sub r16, r17, r18; add r19, r21, r22
sw 0(r20), r6 nop; nop
```

The first set of nops is needed because of the load instruction. The nop in the third 3VDLX instruction is needed because of the true dependency between lines 5 and 6 in the original program. The nops in the last 3VDLX instruction would be needed if we had no further instructions (which is unlikely when the last instruction is not a CTI, but that's where the problem ends).

The instructions can be rearranged for efficient execution. The instructions at lines 5, 7 and 8 can be placed in the second 3VDLX instruction. The remaining can be placed in the third 3VDLX instruction yielding:

```
add r1, r2, r3; sub r6, r7, r8; lw r10, 0(r20)
sub r16, r17, r18; add r19, r21, r22; sub r14, r1, r9
add r11, r10, r12; add r1, r14, r15; sw 0(r20), r6
```

Since the VLIW instructions execute as a unit, there is no need to distinguish between the separate parts of a pipeline stage, as is done for superscalar.

| add | sub | lw | IF | ID | EX | MEM | WB |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| sub | add | sub |  | IF | ID | EX | MEM WB |
| add | add | sw |  |  | IF | ID | EX |
|  |  |  |  |  |  |  |  |

## Problem 2

Each pipeline stage has room for three instructions, superscripts are used to distinguish the parts. To save space, M is used for the MEM stage.

The first three instructions can start executing without delay. Of the next three, only the one on line 5 can start, the add on line 4 must wait for the load and the add on line 6 must wait for the sub on line 5 . With two instructions in ID the last group of three instructions are stalled at time 3.

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| add | IF ${ }^{1}$ | ID ${ }^{1}$ | EX ${ }^{1}$ | $M^{1}$ | WB ${ }^{1}$ |  |  |  |
| sub | $\mathrm{IF}^{2}$ | $\mathrm{ID}^{2}$ | $E X^{2}$ | $M^{2}$ | WB ${ }^{2}$ |  |  |  |
| lw | $\mathrm{IF}^{3}$ | $\mathrm{ID}^{3}$ | $E X^{3}$ | $\mathrm{M}^{3}$ | WB ${ }^{3}$ |  |  |  |
| add |  | $\mathrm{IF}^{1}$ | ID ${ }^{1}$ |  | $E X^{1}$ | $M^{1}$ | WB ${ }^{1}$ |  |
| ub |  | $\mathrm{IF}^{2}$ | $\mathrm{ID}^{2}$ | $E X^{2}$ | $M^{2}$ | WB ${ }^{2}$ |  |  |
| dd |  | $\mathrm{IF}^{3}$ | $I^{3}$ |  | $E X^{3}$ | $\mathrm{M}^{3}$ | WB ${ }^{3}$ |  |
| ub |  |  | $\mathrm{IF}^{1}$ |  | ID ${ }^{1}$ | $E X^{1}$ | $\mathrm{M}^{1}$ | WB ${ }^{1}$ |
| add |  |  | $\mathrm{IF}^{2}$ |  | ID ${ }^{2}$ | $E X^{2}$ | $M^{2}$ | $W^{2}{ }^{2}$ |
| sw |  |  | $\mathrm{IF}^{3}$ |  | ID ${ }^{3}$ | $E X^{3}$ | $\mathrm{M}^{3}$ | $\mathrm{WB}^{3}$ |

## Problem 3

Reservation stations, as described in class, add an extra cycle of latency because of the complexity of using the CDB (in the WB "stage"). The reservation station numbers associated with the functional units are:

|  |  |  |
| :--- | ---: | ---: |
| Int | 1 | 2 |
| L/S | 3 | 4 |
| Int | 5 | 6 |
| L/S | 7 | 8 |
| Int | 9 | 10 |
| L/S | 11 | 12 |

where Int refers to an integer execution unit and L/S refers to a load/store unit. Note that reservation stations are dedicated to functional units. If ready instructions were waiting in reservation stations 1 and 2 , only one could start in a cycle, even if the other two integer execution units were free.

The pipeline notation is the same used in class, for example, $5: \mathrm{EX}^{3}$ indicates that execution unit 3 is executing an instruction from reservation station 5.

Load and store instructions need an ALU to compute addresses, in this solution a load/store unit has its own ALU, that stage is indicated by AD.

Execution is given below:

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| add | $\mathrm{IF}^{1}$ | ID ${ }^{1}$ | 1:EX ${ }^{1}$ | 1: WB ${ }^{1}$ |  |  |  |  |
| sub | $\mathrm{IF}^{2}$ | ID ${ }^{2}$ | 5:EX ${ }^{2}$ | 5: WB ${ }^{2}$ |  |  |  |  |
| lw | $\mathrm{IF}^{3}$ | ID ${ }^{3}$ | 11:AD ${ }^{3}$ | $11: \mathrm{M}^{3}$ | 11:WB ${ }^{3}$ |  |  |  |
| add |  | $\mathrm{IF}^{1}$ | ID ${ }^{1}$ | 2:RS | 2:RS | 2:EX ${ }^{1}$ | $2: W^{1}$ |  |
| sub |  | $\mathrm{IF}^{2}$ | ID ${ }^{2}$ | 6:RS | 6:EX ${ }^{2}$ | 6:WB ${ }^{2}$ |  |  |
| add |  | $\mathrm{IF}^{3}$ | ID ${ }^{3}$ | 9:RS | 9:RS | 9:RS | 9:EX ${ }^{3}$ | $9: W B^{3}$ |
| sub |  |  | $\mathrm{IF}^{1}$ | ID ${ }^{1}$ | 1:EX ${ }^{1}$ | $1: W^{1}$ |  |  |
| add |  |  | $\mathrm{IF}^{2}$ | ID ${ }^{2}$ | 5: $\mathrm{EX}^{2}$ | 5:WB ${ }^{2}$ |  |  |
| sw |  |  | $\mathrm{IF}^{3}$ | ID ${ }^{3}$ | 12:AD ${ }^{3}$ | $12: \mathrm{M}^{3}$ | $12: W^{3}$ |  |

## Problem 4

Memory devices, which have 16-bit entries, are paired to provide the 32-bit words needed. Since 4-byte words are being fetched, use address bits $2 . .19$ to index (as the address for) the memory devices. Bits $20 . .22$ of the address can be used to select the memory pair with the correct data. The solution should include a diagram.

David M. Koppelman - koppel@ee.lsu.edu


[^0]:    Two integer RS are needed (cycle 3), zero FP add RS are needed, three FP multiply units are needed (cycle 14),

