Name _____

Computer Architecture
EE 4720
Midterm Examination

17 March 1998, 19:00–21:00 CST

Modified

Preliminary, Partial Solutions

Problem 1 _____ (42 pts)

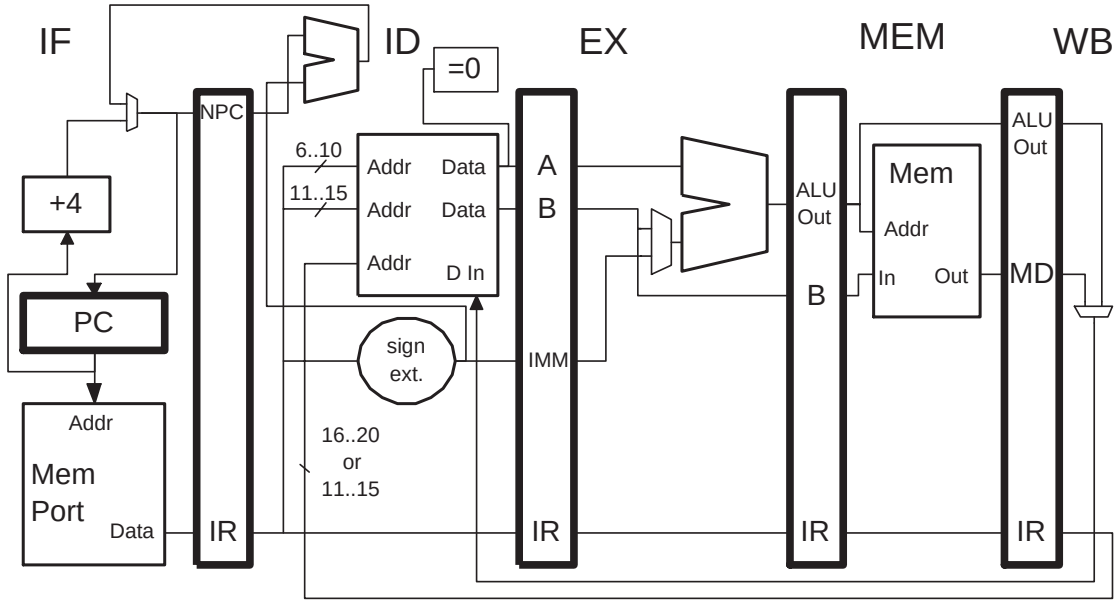
Problem 2 _____ (32 pts)

Problem 3 _____ (26 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: The DLX implementation below uses ID-stage address calculation (as illustrated) and bypassing (which is not illustrated) including the branch condition bypassing developed in homework 3. The branches do not include delay slots.




```
! Initial values: r1=2028, r2=0xf11, r3=5, r4=5004, r5=-1000, LOOP=0x3000, MEM[0xf11]=0x97
LOOP:  ! Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
 lb r1, 0(r2)       IF  ID  EX  ME  WB          IF  ID  EX  ME  WB

 add r2, r2, r3         IF  ID  EX  ME  WB          IF  ID  EX  ME  WB

 lw r4, 3(r2)               IF  ID  EX  ME  WB          IF  ID  EX  ME  WB

 sub r5, r1, r4                 IF  ID  ->  EX  ME  WB      IF  ID  ->  EX  ME  WB

 bneq r5, LOOP                      IF  ->  ID  EX  ME  WB      IF  ->  ID  EX  ME  WB

 addi r2, r2, #1                            IF                      IF
! Cycle             0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

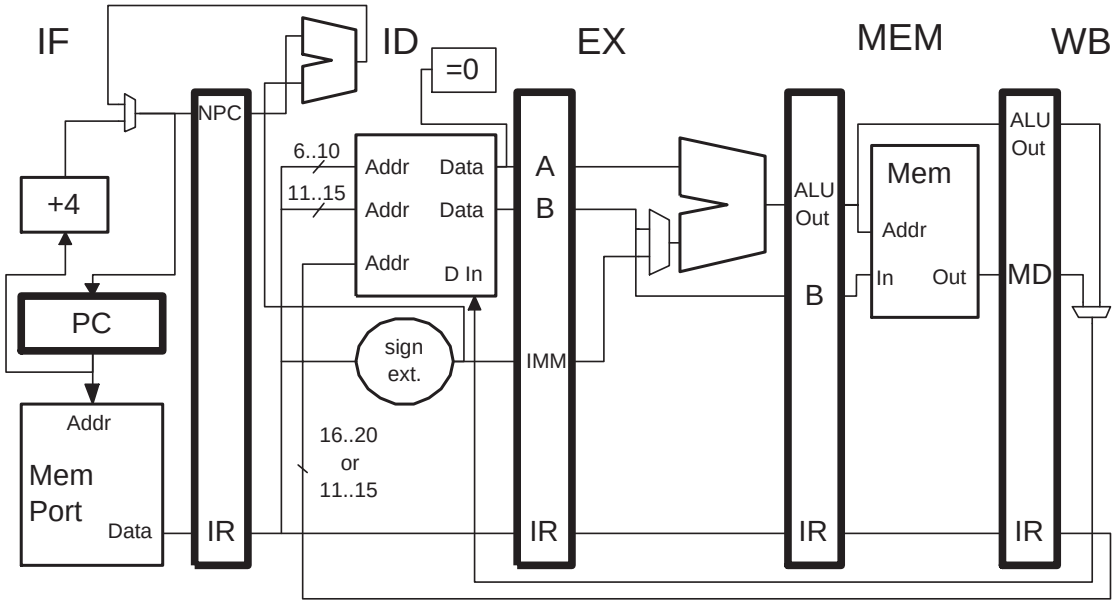
(*a*) Show a pipeline execution diagram for two iterations of the code on the DLX implementation. *Don't forget, bypass paths are present but not shown.* The space to the right of the program or a separate sheet may be used. A description of some mnemonics appears on the next page. (7 pts)

(*b*) On the figure above show exactly those bypass paths needed to execute the code. (7 pts)

(*c*) What is the CPI while executing the loop (assuming a large number of iterations)? (7 pts) $7/5 = 1.4$ CPI.

(*d*) Show the contents (values, not functions of register names) of the pipeline latches at the middle of the first cycle that `lb` is in WB. Include PC and NPC. For IR contents, just show the mnemonic; if an IR contains a nulled instruction show the original mnemonic and "(nulled)." Show the values on the diagram above, write the value within the stage to which it applies with an arrow pointing to the latch or register. (7 pts)

(*e*) Show a pipeline execution diagram of two iterations of the loop below on the DLX implementation illustrated (it's the same as the earlier one). As before, bypass paths are provided but not shown, including hw3 bypass paths. The target of the `beqz r2, SKIP2` instruction is really just the next instruction. Assume that no special hardware optimizations have been made. (7 pts)




```
! Cycle            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20

 addi r1,r0,#0x200  IF  ID  EX  ME  WB
SKIP1:
 andi r2, r1, #1        IF  ID  EX  ME  WB      IF  ID  EX  ME  WB  IF

 beqz r2,SKIP2              IF  ID  EX  ME  WB      IF  ID  EX  ME  WB
SKIP2:
 subi r1, r1, #1                IF  IF  ID  EX  ME  WB  IF  ID  EX  ME  WB

 bneq r1,SKIP1                          IF  ID  EX  ME  WB  IF  ID  EX  ME  WB

 xor  r1, r1, r1                            IF
! Cycle            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

(*f*) Compute the CPI of the execution of the loop above for a large number of iterations. (7 pts)

Note that first branch executes every other iteration, so CPI should be based on two consecutive iterations. $\frac{11}{8} = 1.375$ CPI.

For Reference:

Mnemonic	Description
`addi`	Add immediate
`andi`	Logical "and" immediate
`beqz r1, TARGET`	Branch if `r1` equals zero.
`bneq r1, TARGET`	Branch if `r1` not zero.
`lbu`	Load byte unsigned.
`lb`	Load byte.
`lw`	Load word.

Problem 2: Consider an ISA, DLXPC, which is like DLX except it uses a condition code bit and predicated execution. The condition code bit is set by the execution of new integer arithmetic instructions, to zero if the result is zero and to one if the result is nonzero. The ISA also includes predicated arithmetic instructions which complete only if the condition code bit (set by the most recent condition-code setting instruction) is 1, otherwise they have no effect. The mnemonic for a predicated instruction has a "p" appended, *e.g.*, `add_p`, and the mnemonic for an instruction that sets the condition code has a "`c`" appended, *e.g.*, `add_c`. An instruction can be both predicated and set the condition code; for example, `add_pc` only executes if the condition code (set by a previous instruction) is 1, and sets the condition code itself.

(*a*) Using these instructions rewrite the code below so that it uses fewer instructions and registers. Assume that the value in register `r1` is not used after `beqz`.(6 pts)

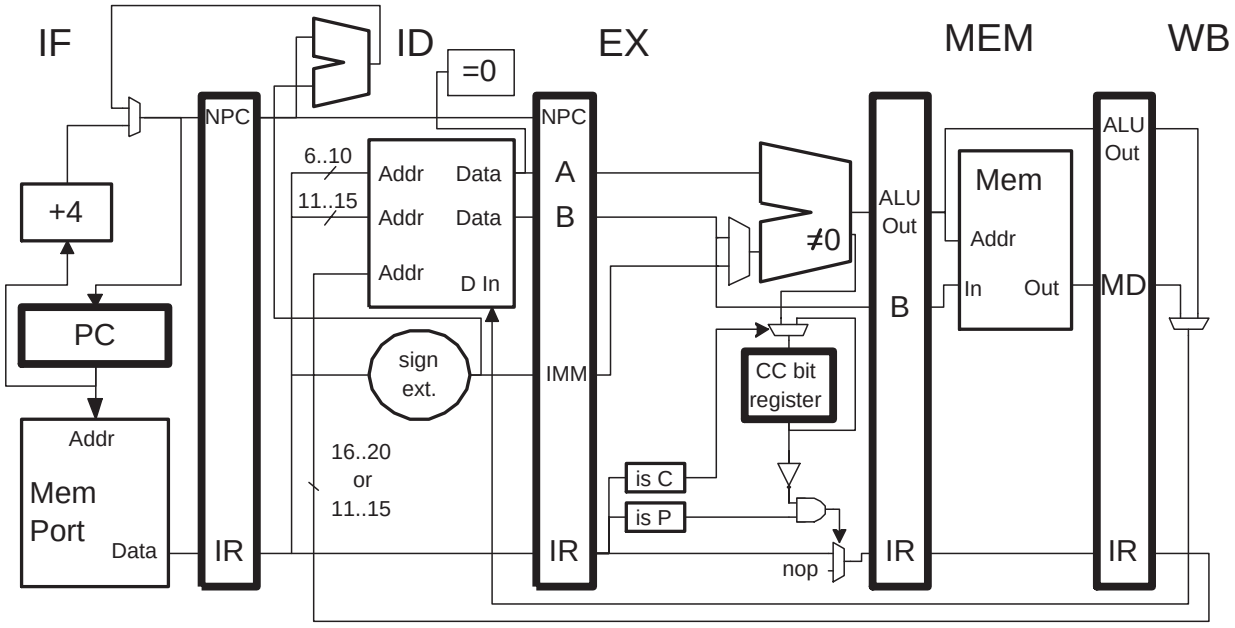
```
 sub r1, r2, r3
 beqz r1, SKIP
 add r4, r5, r6
SKIP:
 addi r4, r4, #1
```

Solution below:

```
sub_c r0, r2, r3
add_p r4, r5, r6
addi r4, r4, \#1
```

(*b*) The pipeline below implements DLXPC—almost: it will not execute predicated, condition-code setting instructions (such as `xor_pc`) correctly if an exception occurs at a certain time. Show code and a pipeline execution diagram that illustrates the problem. Be sure to point out where the exception occurs and what goes wrong. *Hint: the problem would not occur if the execution of some other instructions could be stopped in the cycle that an exception was detected.* (9 pts)

In the figure below the ALU has a second output, which is 1 if the main output is not equal to zero, 0, otherwise. The output of is C is 1 if the instruction is a type that sets the condition code (*e.g.*, `add_c`); the output of is P is 1 if the instruction is predicated, zero otherwise.



Condition-bit instructions change processor state (the condition bit register) one cycle earlier than other instructions.

```
Cycle   0   1   2   3     4   5
lw      IF  ID  EX  *ME*  WB
xor_pc      IF  ID   EX   ME  WB
```

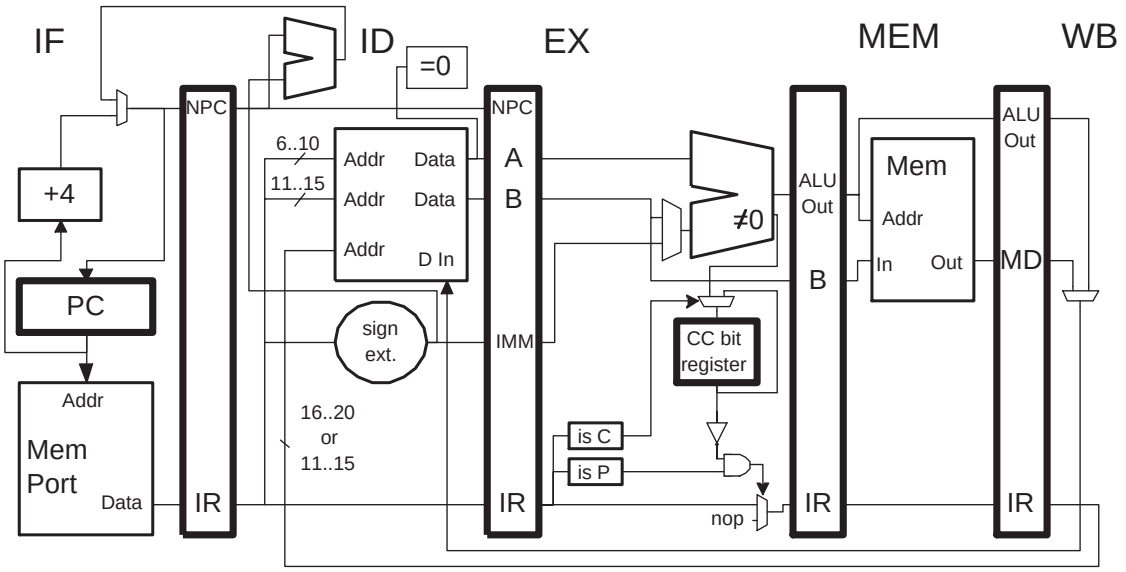
In cycle 3, the lw instruction raises a page-fault exception in the same cycle xor_pc is writing the condition bit. If the condition bit is overwritten then it will be impossible to get the previous value and so when the trap handler returns, xor_pc may not execute, even though the condition bit was 1 (before being overwritten).

(*c*) Modify the pipeline to fix the problem. (9 pts)

One possibility is to add a condition-bit-in-last cycle register. (This would have the condition bit value that was valid in the previous cycle.) It would be used to restore the old bit value, details would be needed for a complete solution.

(*d*) Suppose the ISA also includes a new branch instruction, mnemonic `b_p`, that tests the condition bit (instead of a register) and branches if it's true. To implement the branch instruction an ID-stage `take_branch` signal is needed. The signal should be 1 when a `b_p` instruction that will be taken is in the ID stage (zero otherwise). Show how the pipeline below would have to be modified to provide this signal. Be sure that the modified pipeline executes the code below correctly and without adding an unnecessary stalls. (8 pts)

```
sub_c r0, r1, r2  ! Set condition code.
b_p TARGET
add_c r3, r4, r5  ! Set condition code.
add r6, r7, r8    ! Doesn't affect condition code.
b_p TARGET2
```



Outline: In most cases the CC bit could be used to generate `take_branch`, the exception is when the bit is set by the instruction immediately preceding the branch. For that case the input to the condition bit register would be used (perhaps stretching the clock cycle). A multiplexor would select the proper signal, controlled by the $\boxed{=C}$ box.

Problem 3: Answer each question below.

(*a*) In class execution time has been modeled using the equation $t = \frac{1}{\phi} \sum_i \mathrm{IC}_i \mathrm{CPI}_i$. Given what has been covered in class so far, to what degree is CPI_i a function of the implementation and to what degree is CPI_i a function of the program? Explain. (9 pts)

CPI is strongly determined by implementation, for example, bypass paths would reduce the CPI. CPI is effected to a small degree by the program, since different instruction orderings might effect their execution. For example, if a program has a load instruction immediately followed by an instruction that uses the loaded value than the CPI_l, where l is the class of `lw` would be larger than a program in which loads were followed by non-dependent instructions.

(*b*) Synthetic instruction `neg r1` assigns register `r1` to the negation of its previous value, that is, `r1 = -r1`. How would such a synthetic instruction be defined in DLX? (8 pts)

Solution: `sub r1, r0, r1`

(*c*) Packed-operand instructions (as found in MMX or VIS) and elaborate procedure call instructions (that perform extensive stack-frame preparation actions including register saves) are similar in that they can replace many individual instructions. Provide contrasting reasons on whether or not such instructions should be added to an ISA, including implementation factors. For example, _____instructions should be added because the implementation _____while _____should not be added because _____though certainly if _____it would _____, but that's not enough[1]. (9 pts)

The procedure call instruction should not be added because not much time would be saved over individual instructions that perform the same actions. A procedure call function would constrain how procedure could be called. Packed operand instructions would (usually) take no more time to execute than ordinary arithmetic instructions. The instructions that a packed-operand instruction replace would take many cycles to execute, so there is a good reason to add them to an ISA.

[1] This is just an example, don't try to fill in the blanks!