Name Solution

Computer Architecture
LSU EE 4720
Solve-Home Final Examination

Wednesday, 6 May 2020 to Saturday, 9 May 2020 5:00 (5 AM) CDT

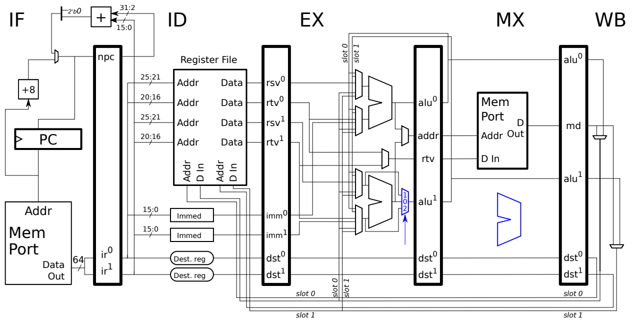
Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as MIPS tutorials, digital logic design guides, and computer architecture references can also be used. Do not try to directly seek out solutions to any question here. That is, don't Web-search the text of a problem. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1		(30 pts)
Problem 2		(25 pts)
Problem 3		(15 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

$r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Help Keep Everybody Safe!

Problem 1: (30 pts) The two-way superscalar MIPS implementation below has an ALU in the `MX` (née `ME`) stage, call it the *second-chance ALU*. For this problem the second-chance ALU will be connected so that two stall situations are avoided. *A slightly similar problem appeared on the Spring 2006 Midterm Exam in Problem 2. It's okay to look at the problem and solution.*



Yes, it's small! Use the next page for the solution.

(*a*) The `sub` in the code below suffers a stall due to a dependence with the other instruction in the same fetch group. Connect the second-chance ALU so that the stall is avoided. The changes must not break existing functionality and must not result in stalls for unrelated code. In particular note that the `add` does not stall in either version.

```
# Cycle          0  1  2  3  4  5  6  7   # Unmodified Implementation
ori R3, r1, 0xff IF ID EX ME WB
xor R2, r8, r9   IF ID EX ME WB
add R1, R2, R3      IF ID EX ME WB
sub R4, R1, r5      IF ID -> EX ME WB
and r7, r8, R4         IF -> ID EX ME WB
# Cycle          0  1  2  3  4  5  6  7

# Cycle          0  1  2  3  4  5  6  7   # With second-chance ALU.
ori R3, r1, 0xff IF ID EX MX WB
xor R2, r8, r9   IF ID EX MX WB
add R1, R2, R3      IF ID EX MX WB        # No stall in either implementation.
sub R4, R1, r5      IF ID EX MX WB        # No stall due to second-chance ALU!
and r7, r8, R4         IF ID -> EX MX WB  # Stall due to second-chance ALU.
# Cycle          0  1  2  3  4  5  6  7
```

☑ Connect second-chance ALU to avoid the stall by the `sub` and allowing code to execute as in the sample above. ☑ The connections should work for any pair of dependent, ALU-using, non-memory instructions.

☑ Pay attention to cost. Assume that a pipeline latch bit costs twice as much as a multiplexor bit.

☑ Do not add unneeded bypass paths. ☑ Don't break existing functionality.

Solution appears in blue in the diagram several pages ahead. The muxes at the second-chance ALU inputs are needed because the `MX.alu0` value could be needed by the `rs` or `rt` operands (or both). Many solved the problem assuming that the `MX.alu0` value is only needed for the `rs` source.

(*b*) The code below suffers a load/use stall. Add the minimum number of connections to the second-chance ALU so that such load/use stalls (in which the using instruction is in slot 1) can be avoided.

```
add r3, r2, r3   IF ID EX ME WB
lw r4, 5(r1)     IF ID EX ME WB
ori r6, r1, 0xff    IF ID EX ME WB
sub r5, r3, r4      IF ID -> EX ME WB
```

☑ Add connections to the second-chance ALU to avoid load/use stalls when the using instruction (such as the `sub` in the example) is in slot 1.

☑ Pay attention to cost, use the same cost assumption as given in the previous part.

See solution several pages ahead.

(*c*) In the code below the **sub** does not stall due to the second-chance ALU but the **and** does stall. Add control logic to generate a stall signal for cases such as this.

```
# Cycle            0  1  2  3  4  5  6  7
ori R3, r1, 0xff   IF ID EX MX WB
xor r2, r8, r9     IF ID EX MX WB
add R1, r2, R3        IF ID EX MX WB
sub R4, R1, r5        IF ID EX MX WB        # No stall due to second-chance ALU!
and r7, r8, R4           IF ID -> EX MX WB  # Stall due to second-chance ALU.
# Cycle            0  1  2  3  4  5  6  7
```

☑ Add logic to generate a stall signal for the situation described above. ☑ The logic should work for any instruction dependent on an instruction using the second-chance ALU.

☑ Pay attention to cost, use the same cost assumption as given in the previous part.

See solution on next page.

(*d*) Generate the select signal for the **EX** stage multiplexor shown in blue. The control logic should work for the intra-group dependence case. (The control logic does not need to work for the load/use case.)

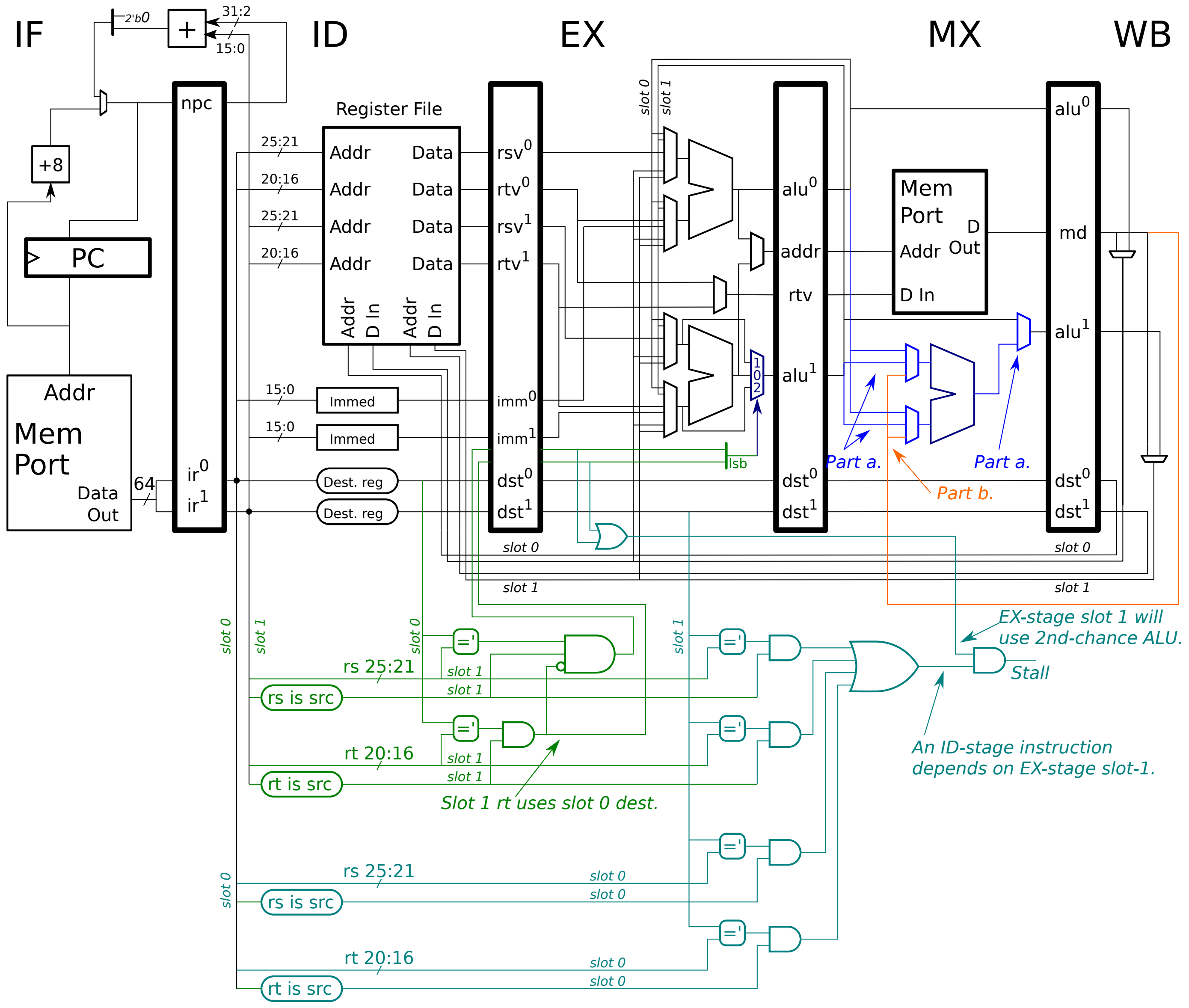
☑ Add logic for the select signal for the intra-group dependence. ☑ The logic should work for any pair of dependent, ALU-using, non-memory instructions.

☑ Pay attention to cost, use the same cost assumption as given in the previous part.

You're almost there! The solution is on the very next page!!

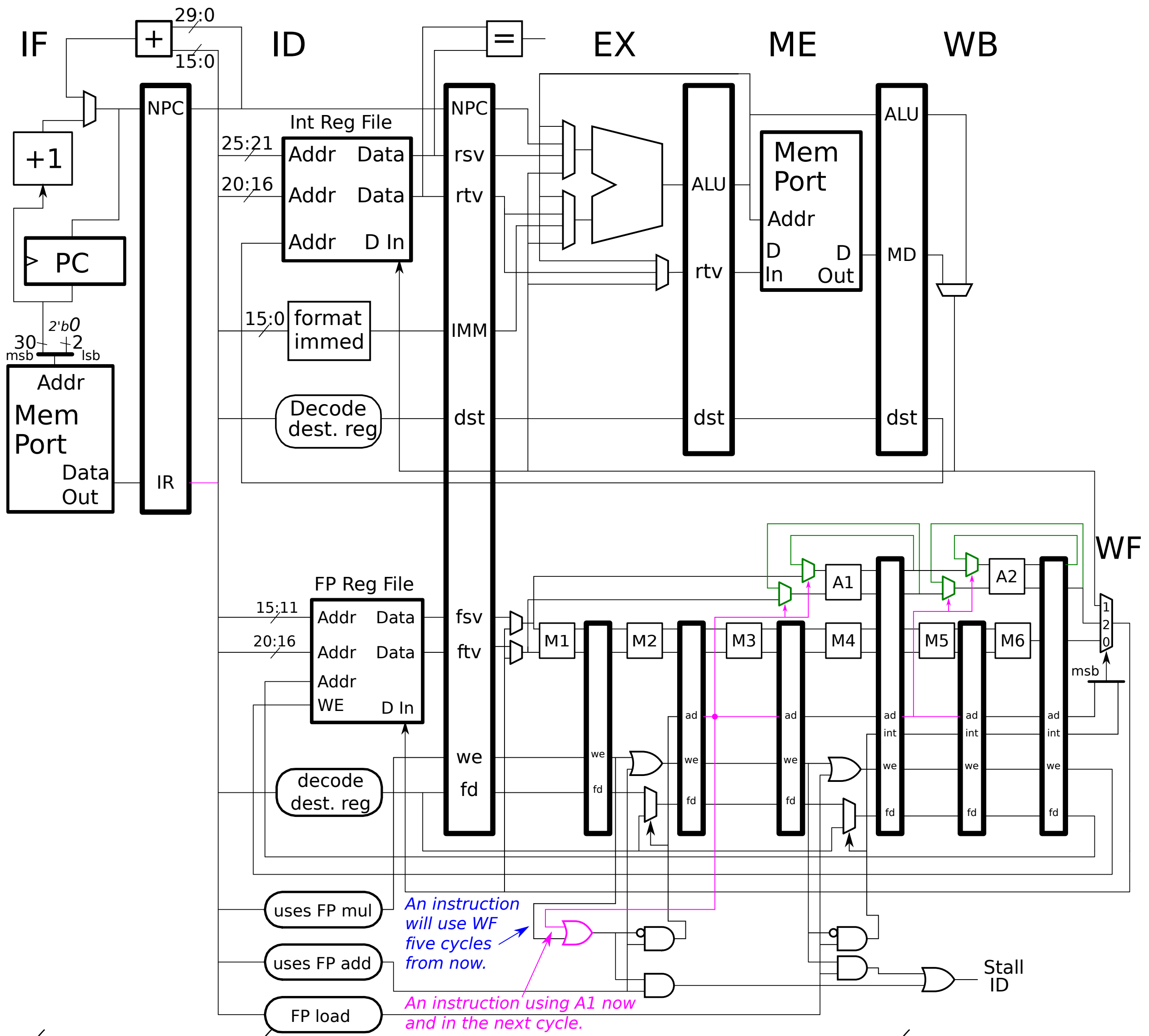
The Inkscape SVG source is at `https://www.ece.lsu.edu/ee4720/2020/fe-p1-v2-ss.svg`.

Solution appears below in blue (part a and b), green (part c or d, depending on whether I swap them), and turquoise (part d or c).



Problem 2: (25 pts) Show the execution of the code fragments as requested below.

(*a*) Show the execution on the FP pipeline below, note that the adder unit has an initiation interval of 2.

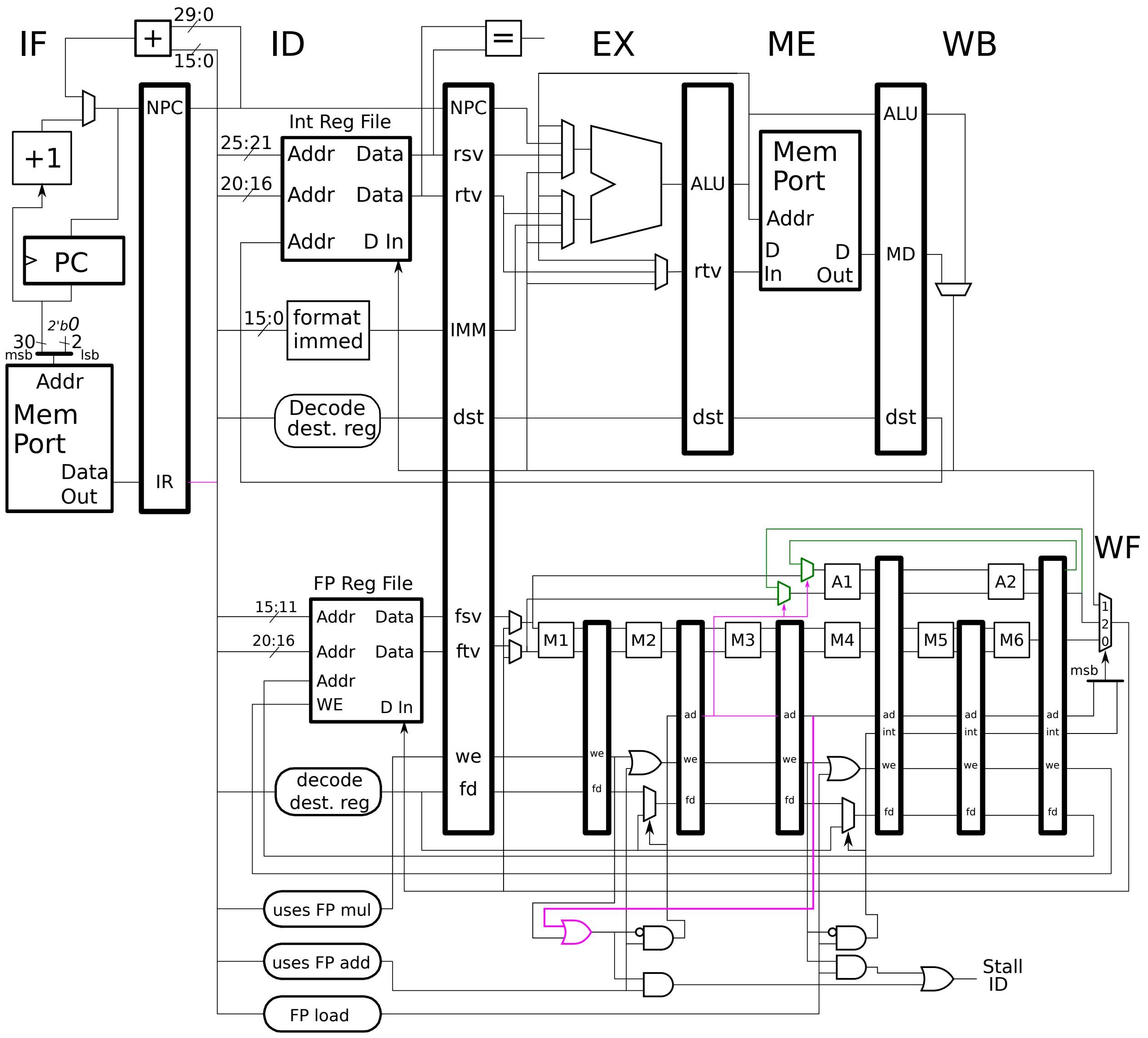


☑ Show execution. ☑ Pay attention to how the FP add unit should operate. ☑ Don't forget to check for dependencies.

Solution appears below. The second `add.s` stalls one cycle because of the initiation interval of the FP add unit. You get what you pay for.

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14  SOLUTION
lwc1 f2, 0(r1)     IF ID EX ME WF
add.s f0, f2, f4      IF ID -> A1 A1 A2 A2 WF
add.s f1, f2, f5         IF -> ID -> A1 A1 A2 A2 WF
add.s f3, f1, f6               IF -> ID -------> A1 A1 A2 A2 WF
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

(*b*) Show the execution on the FP pipeline below, note that the adder unit is different than the previous problem and from other examples covered in class.

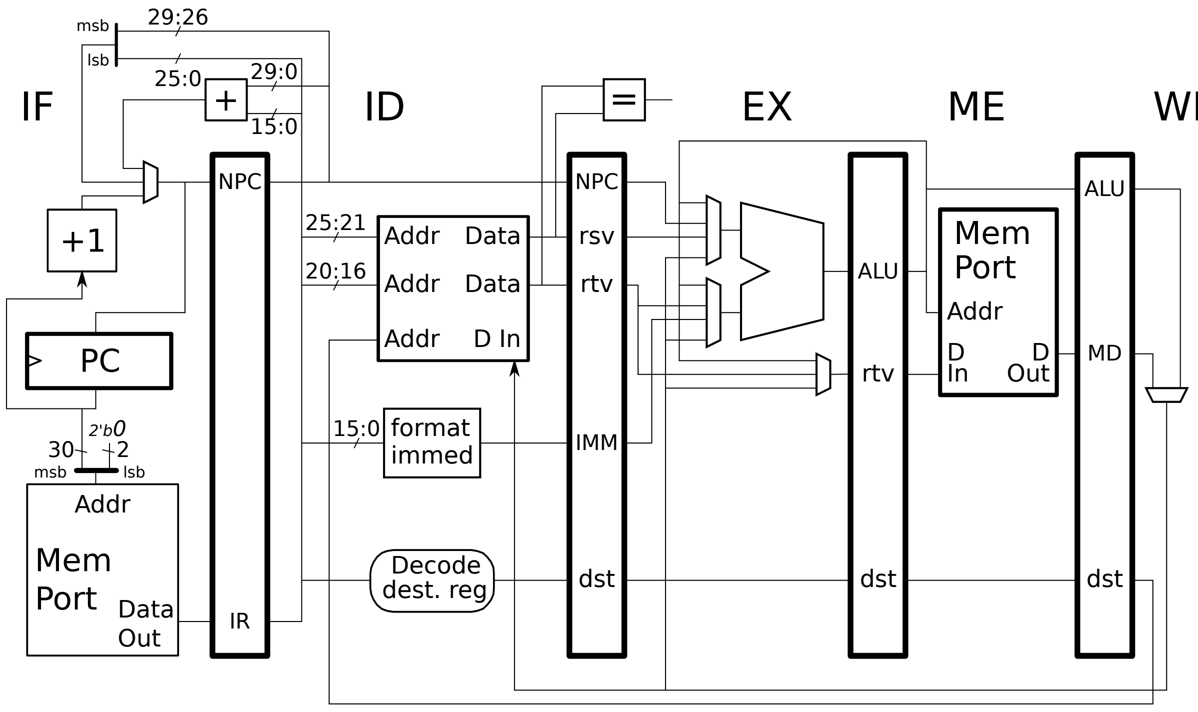


☑ Show execution. ☑ Pay attention to how the FP add unit should operate. ☑ Don't forget to check for dependencies.

Solution appears below. As can be inferred from the control signals, a FP add instruction passes through the A1 and A2 stages twice as shown in the solution below. Note that the second `add.s` no longer stalls waiting for A1!

```
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13  SOLUTION
lwc1 f2, 0(r1)    IF ID EX ME WF
add.s f0, f2, f4     IF ID -> A1 A2 A1 A2 WF
add.s f1, f2, f5        IF -> ID A1 A2 A1 A2 WF
add.s f3, f1, f6              IF ID -------> A1 A2 A1 A2 WF
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

(c) Show the execution of the code on the implementation below. Find the CPI for a large number of iterations.

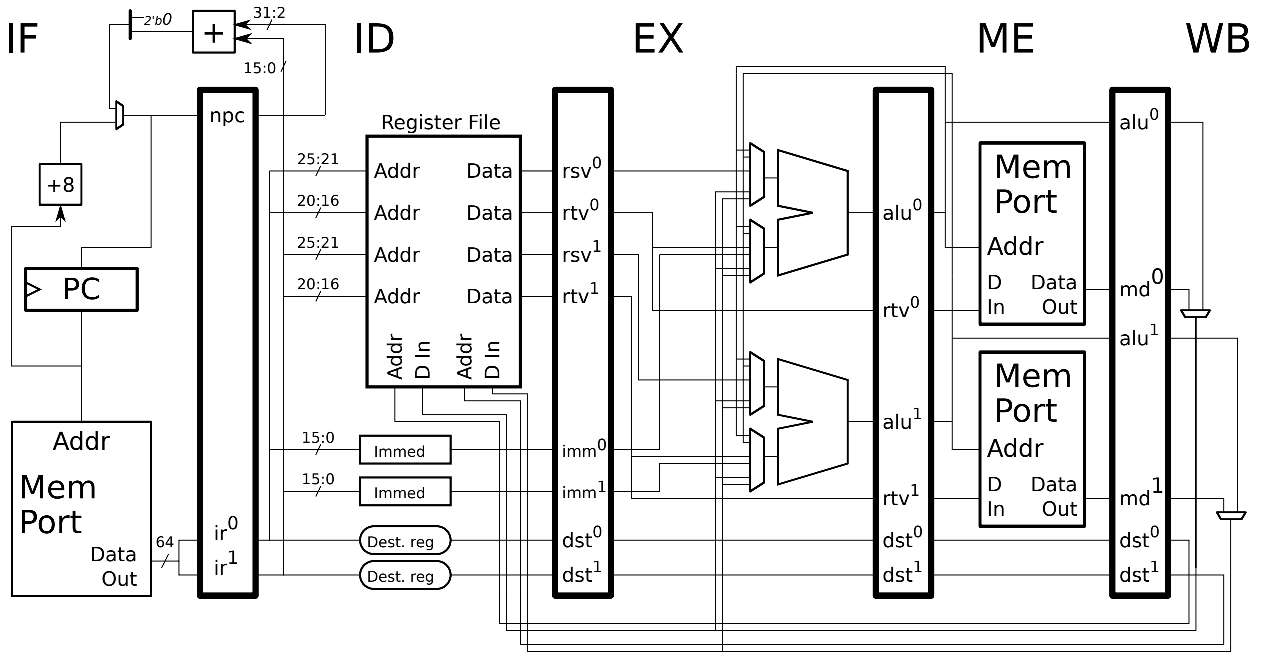


☑ Show execution on the illustrated implementation with the branch taken. ☑ Find the CPI for a large number of iterations.

The solution appears below. 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`) and second at cycle 6. The second iteration is the same as the first, so all iterations will take $6 - 0 = 6$ cycles. An iteration has 5 instructions and so the CPI is $\frac{6}{5} = 1.2\,\mathrm{CPI}$, and the instruction throughput is $\frac{5}{6}\,\mathrm{insn/cycle}$.

```
      # SOLUTION
LOOP: # Cycle        0  1  2  3  4  5  6  7  8  9  10 11
 addi r2, r2, 16     IF ID EX ME WB                      First Iteration
 lw r1, 8(r2)           IF ID EX ME WB
 sw r1, 12(r3)             IF ID -> EX ME WB
 bne r3, r4, LOOP             IF -> ID EX ME WB
 addi r3, r3, 32                    IF ID EX ME WB
 sub r10, r3, r2
LOOP: # Cycle        0  1  2  3  4  5  6  7  8  9  10 11
 addi r2, r2, 16                       IF ID EX ME WB    Second Iteration
 lw r1, 8(r2)                             IF ID EX ME WB
 sw r1, 12(r3)                               IF ID -> EX ME WB
 bne r3, r4, LOOP                               IF -> ID EX ME WB
 addi r3, r3, 32                                      IF ID EX ME WB
#       Cycle        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

(*d*) Show the execution of the code on the 2-way superscalar MIPS implementation illustrated below, and find the CPI for a large number of iterations. This is not the same as the implementation from Problem 1. Instruction fetch is of aligned groups.



[✓] Show execution on the illustrated implementation. [✓] Find the CPI for a large number of iterations.

[✓] Take aligned fetch into account, the address of `LOOP` is `0x1000`. [✓] Pay attention to available bypass paths.

Solution appears below. Because there is no bypass path for the store value the `sw` must stall until the `lw` reaches WB.

The CPI is $\frac{6}{5} = 1.2$, which is disappointing because the implementation is capable of a CPI of $\frac{1}{2}$.

```
LOOP:  #     Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 #  SOLUTION
 addi r2, r2, 16    IF ID EX ME WB                         #  First Iteration
 lw r1, 8(r2)       IF ID -> EX ME WB
 sw r1, 12(r3)         IF -> ID ----> EX ME WB
 bne r3, r4, LOOP      IF -> ID ----> EX ME WB
 addi r3, r3, 32             IF ----> ID EX ME WB
 sub r10, r3, r2             IFx
LOOP:  #     Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12
 addi r2, r2, 16                      IF ID EX ME WB       #  Second Iteration
 lw r1, 8(r2)                         IF ID -> EX ME WB
 sw r1, 12(r3)                           IF -> ID ----> EX ME WB
 bne r3, r4, LOOP                        IF -> ID ----> EX ME WB
 addi r3, r3, 32                               IF ----> ID EX ME WB
 sub r10, r3, r2                               IFx
#            Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

(*e*) The code fragment below is the same as the one from the previous problem and is to run on the same superscalar system. Re-write the code so that it runs with fewer stalls (and of course does the same thing), and compute the CPI for a large number of iterations. Extra instructions can be added before or after the loop. Do not unroll the loop.

☑ Re-write code to minimize stalls on the superscalar implementation.

☑ Compute the CPI of the re-written code for a large number of iterations.

```
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
```

Two solutions are given here, Partial Credit and Full Credit (starting on the next page). An easy change was moving the `addi r2,r2,16` increment after the `lw` and adding 16 to the load offset. This is shown as the Partial Credit solution—partial credit because the `sw` still stalls.

```
 # SOLUTION - Partial Credit
LOOP:
 lw r1, 24(r2)     # Move lw before addi and increase offset to 24.
 addi r2, r2, 16
 sw r1, 12(r3)     # Still lots of stall cycles.
 bne r3, r4, LOOP
 addi r3, r3, 32
 sub r10, r3, r2
```

In the full-credit solution the `sw` is put before the `lw` where it will store the `r1` value *that was loaded in the previous iteration* or by the *prologue* instruction added before the loop. The `lw` in the loop is loading a value for the next iteration, which is why its offset is increased by another 16 to 40.

To compute the instruction throughput (or CPI) a repeating pattern needs to be found. The repeating pattern is established in the second iteration and verified in the third by noting that the pipeline state at cycles 10 and 14 are identical. So the iteration time is $14 - 10 = 4$ cycles and so the instruction throughput is $\frac{5}{4} = 1.25\,\text{insn/cycle}$ or the CPI is $\frac{4}{5} = .8\,\text{CPI}$, an improvement.

```
 # SOLUTION - Full Credit
 #
 lw r1, 24(r2)     # Prologue: Load initial value of r1.

LOOP:
 sw r1, 12(r3)     # This finishes up the previous iteration.
 lw r1, 40(r2)     # Load value to be used in next iteration, if any.
 addi r2, r2, 16   # Increment r2 after lw to avoid a stall.
 bne r3, r4, LOOP
 addi r3, r3, 32
 sub r10, r3, r2

 # Pipeline Execution Diagram
 #
 lw r1, 24(r2)    IF ID EX ME WB
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)       IF ID ----> EX ME WB       First Iteration
 lw r1, 40(r2)       IF ID ----> EX ME WB
 addi r2, r2, 16        IF ----> ID EX ME WB
 bne r3, r4, LOOP       IF ----> ID EX ME WB
 addi r3, r3, 32                 IF ID EX ME WB
 sub r10, r3, r2                 IFx
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)    Second Iteration  IF ID EX ME WB
 lw r1, 40(r2)                      IF ID EX ME WB
 addi r2, r2, 16                       IF ID EX ME WB
 bne r3, r4, LOOP                      IF ID -> EX ME WB
 addi r3, r3, 32                          IF -> ID EX ME WB
 sub r10, r3, r2                          IFx
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)    Third Iteration               IF ID EX ME WB
 lw r1, 40(r2)                                  IF ID EX ME WB
 addi r2, r2, 16                                   IF ID EX ME WB
 bne r3, r4, LOOP                                  IF ID -> EX ME WB
 addi r3, r3, 32                                      IF -> ID EX ME WB
 sub r10, r3, r2                                      IFx
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)                                              IF ID EX ME WB
 lw r1, 40(r2)                                              IF ID EX ME WB
```

Problem 3: (15 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and the other systems have a local predictor, the length of the local history is given in the questions below.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   N  N  N  N  T  T   N  N  N  N  T  T  ...

B2:     T  T  T  T  T  T   T  T  T  T  T  T  ...
```

☑ What is the accuracy of the bimodal predictor on branch B1?

Short answer: $\frac{3}{6}$, see work below.

```
    SOLUTION WORK
    0  0  0  0  0  1  2    1  0  0  0  1  2   <-  Two-bit counter
B1:   N  N  N  N  T  T    N  N  N  N  T  T   ...
                  x  x    x           x  x    <-  Prediction Outcome
                       --------------------   <-  Repeating Pattern
```

Explanation: The line just below `SOLUTION WORK` (above) shows 2-bit counter values for B1, for when the counter starts at zero. The prediction outcomes are shown below the branch outcomes. To compute the prediction ratio we need to use a repeating pattern. A pattern is repeating if the branch outcomes pattern repeats and if the two-bit counter value is the same before the pattern starts and after it ends. Such a pattern is underlined, the counter is two at the start and the end. The prediction accuracy is $\frac{3}{6} = \frac{1}{2}$.

☑ What is the accuracy of a local predictor with a 12-outcome local history on branch B1 and ignoring B2.

Branch B1's pattern repeats and has a length of 6 outcomes and so it can easily be predicted with 100% accuracy by a 12-outcome local predictor.

[✓] What is the accuracy of a local predictor with a 2-outcome local history on branch B1 and ignoring B2.

The accuracy is worked out below. The local histories table shows the 6 local histories used when predicting B1. The labels (1-6) refer to the outcome being predicted, and the local history (LH) consists of the 2 preceding outcomes. For example, then predicting the first T, label 3, the local history is NN. The Pattern History Table Analysis table has one row for each possible PHT entry, all $2^2 = 4$ of them. (In this case the branch affects every entry in the table, but usually branches use only a small fraction of the entries. For example, if the local history size were 10 there would be 1024 PHT entries but branch B1 would use only 6 of them [after warmup].) A PHT entry holds a two-bit value (called a counter), and the table shows values for these counters for each entry as well as the outcome pattern used to update the counter. For TN the outcome pattern is just a sequence of Ns so the counter stays at zero. The NT and TT entries are similarly well-behaved. But the NN entry is used to predict three outcomes, those at label 1, 2, and 3. The counter will change form 1 to 0 and back, and so N will be consistently predicted, which will be correct 2 out of 3 times. These accuracies are shown in the last column and totaled at the bottom. Note that the total is computed by adding the numerators and denominators: $\frac{2+1+1+1}{3+1+1+1} = \frac{5}{6}$, and that it is important that the denominator is set to the number of times the local history is seen in the repeating pattern. Based on this analysis the $\boxed{\text{prediction accuracy is } \frac{5}{6}}$.

```
      SOLUTION WORK
B1:   N  N  N  N  T  T    N  N  N  N  T  T
      5  6  1  2  3  4    5  6  1  2  3  4  <- Label of predicted branch.

Local Histories (LH) and Outcome
   LH Outcome
1: NN N
2: NN N
3: NN T
4: NT T
5: TT N
6: TN N

Pattern History Table Analysis
LH  Pattern               Two-Bit Counter Evolution   Accuracy
--  ------------------    -------------------------   --------
NN: N N T    N N T        1,0,0,      1,0,0,          2/3
NT:      T        T             3,          3,        1/1
TT:       N        N              0,          0,      1/1
TN:        N        N               0,          0,    1/1
--  ------------------    -------------------------   --------
                                                      5/6
```

☑ What is the accuracy of a local predictor with a 2-outcome local history on branch B1 and taking into account B2.

The local history for B2 is consistently TT and of course the outcome too is consistently T. Branch B1 and B2 will share the TT entry, but not nicely. When B1 is resolved the entry is decremented but when B2 is resolved the entry is incremented. We can infer from the branch patterns that B2 will use the TT local history six times more frequently than B1 and so B1 at label 5 will retrieve a 3 and predict T and be wrong. That's shown in the table below, with the B2 outcomes shown in lower case, t, for clarity. So, $\boxed{\text{the accuracy of B1 taking into account B2 is } \frac{4}{6}}$. Branch B2 is predicated at 100% accuracy.

```
Pattern History Table Analysis                           B1
LH  Pattern                   Two-Bit Counter Evolution   Accuracy
--  ------------------------  -------------------------   --------
NN: N N T       N N T         1,0,0,      1,0,0,          2/3
NT:       T           T             3,          3,        1/1
TT:  t t t tNt t t t t tNt t  3,3,3,3,2,3,3,3,3,3,2,3,3   0/1
TN:           N           N               0,          0,  1/1
--  ------------------------  -------------------------   --------
                                                          4/6
```

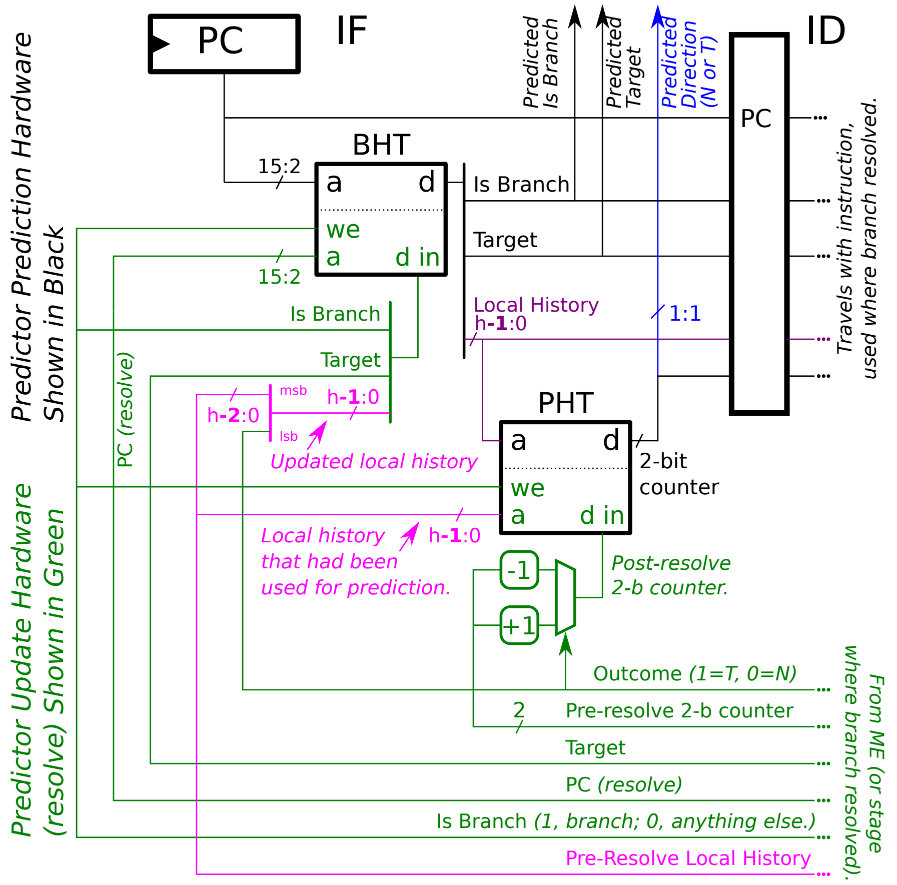
☑ What is the minimum local history size so that branch B1 is predicted with 100% accuracy, taking into account B2.

To solve this problem increase the local history size until the local histories are followed by consistent outcomes (unlike NN above). Because of NNNNT a local history of 3 is too short because NNN can be followed by N or T. $\boxed{\text{A local history size of four will be sufficient.}}$ Note that this time B1 uses 6 out of $2^4 = 16$ entries and that B2 no longer causes trouble since its one local history, TTTT, does not match any of B1's six local histories.

```
Local Histories (LH) and Outcome
   LH Outcome
1: TTNN N
2: TNNN N
3: NNNN T
4: NNNT T
5: NNTT N
6: NTTN N
```

```
Pattern History Table Analysis                 B1         B2
LH    Pattern       Two-Bit Counter Evolution  Accuracy   Accuracy
----  ------------  -------------------------  --------   --------
NNNN:    T          3,3,...                    1/1
NNNT:      T        3,3,...                    1/1
NNTT:        N      0,0,...                    1/1
NTTN:          N    0,0,...                    1/1
TNNN:   N           0,0,...                    1/1
TTNN: N             0,0,...                    1/1
TTTT: t t t t t t   3,3,...                               6/6
----  ------------  -------------------------  --------   --------
                                               6/6        6/6
```

(b) Appearing below is a diagram of a local predictor. The local predictor illustrated has a specific BHT size and an h-outcome local history size. The BHT size does not necessarily match the BHT in the prior part. Determine the amount of storage, in bits, used by the BHT and PHT for a 16-bit local history. Assume that Target is stored efficiently.



Predictor Prediction Hardware Shown in Black
PC
IF
Predicted Is Branch
Predicted Target
Predicted Direction (N or T)
ID
PC
BHT
15:2
a
d
we
a
d in
15:2
Is Branch
Target
Local History h-1:0
1:1
Travels with instruction, used where branch resolved.
Is Branch
Target
msb
h-1:0
h-2:0
lsb
Updated local history
PHT
a
d
we
a
d in
2-bit counter
PC (resolve)
Local history that had been used for prediction.
h-1:0
-1
+1
Post-resolve 2-b counter.
Outcome (1=T, 0=N)
2
Pre-resolve 2-b counter
Target
PC (resolve)
Is Branch (1, branch; 0, anything else.)
Pre-Resolve Local History
From ME (or stage where branch resolved).
Predictor Update Hardware (resolve) Shown in Green

☑ Amount of storage for PHT is:

Short answer: $\boxed{\text{The amount of storage is } 2^{16} \times 2 = 2^{17}\,\text{b} = 2^{14}\,\text{B.}}$

Each PHT entry holds a 2-bit counter, and for a 16-bit local history there are $2^{16} = 65536$ entries. The total size is $65536 \times 2\,\text{b} = 131072\,\text{b} = 16384\,\text{B}$. Note: full credit would be given for an answer in bits, no need to convert to bytes. That said, it is a good idea to include the unit (bits or bytes).

☑ Assumption about Target:

☑ Amount of storage for BHT is:

Short answer: $(1 + 16 + 16)\,2^{14}\,\text{b}$ assuming that the Target field is 16 bits and is used, after retrieval from the BHT, as a displacement from PC+4 to compute the full 32-bit target address.

From the diagram it can be seen that $16-2 = 14$ bits are used for the BHT address (a) input, and so there must be 2^{14} BHT entries. The diagram shows that an entry consists of three fields: *Is Branch*, *Target*, and *Local History*. Assume 1 bit for Is Branch because nothing was mentioned about predicting other kinds of control-transfer instructions. For the target assume that just 16 bits are stored, and that the full 32-bit target is computed using PC+4, just as it is for a branch. The local history is given as 16 bits. Totaling these yields the of a BHT entry, $(1+16+16)\,\text{b}$. The sum of the 2^{14} entry sizes yields the BHT size, $(1+16+16)2^{14}\,\text{b}$.

That's all that's needed for full credit. But since we're here, let's convert it to bytes and compare the size to other parts of the CPU to see how much the BHT is costing us. So, $(1+16+16)2^{14}\,\mathrm{b} = (1+16+16)2^{14}/2^3\,\mathrm{B} = (1+16+16)2^{11}\,\mathrm{B} = 67584\,\mathrm{B}$. That's roughly the size of a typical L1 data cache, so it is on the large size. Real BHTs are smaller.

Problem 4: (30 pts) Answer each question below.

(*a*) Appearing below are the three integer MIPS I instruction formats. Consider a modified form of MIPS in which there are 64 rather than 32 integer registers. A goal is compatibility with MIPS-I code and to use as few new opcodes and function field values as possible. Modify each format so that it can use 64 registers and explain what new opcodes (if any) are needed and any assumptions about existing MIPS-I instructions. *Hint: For one case there's nothing to do, for one case many opcodes will be needed, for one case only a few.*

To encode 64 registers a 6-bit register field is needed. Since a goal is compatibility we want to make as few changes to the instruction format as possible.

[✓] Modification for 64-register MIPS. [✓] Describe what new opcode/func values are needed for, if any.

	Opcode	RS	RT	RD	SA	Function
MIPS R:						
	31 26	25 21	20 16	15 11	10 6	5 0

For most MIPS-I Type R instructions the **SA** field is unused and must be zero. So for MIPS-I64r use the **SA** field for the three extra register field bits, one for **RS**, one for **RT**, and one for **RD**. New `Function` values will be needed for the shift instructions because they use the **SA** field. The existing shift instructions, such as `sll r1, r2, 3` must use the **SA** field in the same way as MIPS-I (otherwise MIPS-I64r would not be compatible). New shift instructions, such as `sll64 r40, r1, 3` will use the **RT** field for the shift amount. No new opcodes or `Function` values will be needed for the other Type R instructions.

[✓] Modification for 64-register MIPS. [✓] Describe what new opcode/func values are needed for, if any.

	Opcode	RS	RT	Immed
MIPS I:				
	31 26	25 21	20 16	15 0

There is no free space in most type I instructions, so new opcodes will be needed. Since there are not many opcodes available a new 3-bit `fuI` field was added (see below) to provide more opcode space, serving the role that `Function` places in type R instructions. The extra bit for the **RS** and **RT** registers are in the same place they are for MIPS I64r instructions, they are labeled **RSh** and **RTh**. The immediate, now 11 bits, is split between two fields, **ImmHi** and **ImmLo**. If a compiler or human needs an instruction with an immediate value of 12 or more bits it will need to use a MIPS-I instruction and so will have to limit itself to registers `r0` to `r31`.

	Opcode	RS	RT	ImmHi	RSh	RTh	fuI	ImmLo
MIPS I64r I:								
	31 26	25 21	20 16	15 11	10 10	9 9	8 6	5 0

[✓] Modification for 64-register MIPS. [✓] Describe what new opcode/func values are needed for, if any.

	Opcode	II
MIPS J:		
	31 26	25 0

No changes need to be made because there are no register fields.

(*b*) Chip A has five 4-way superscalar cores. Chip B has 20 scalar cores. The cores are similar to our pipelined MIPS implementations. All cores use a 1 GHz clock. *Yes, up until this comment the question is identical to one asked on the Spring 2019 final exam.* The SPECcpu benchmarks are run on each chip. Recall that SPECcpu can be run to compute a speed score and a rate score. (Don't confuse speed/rate with base/peak or int/FP.) Feel free to visit the SPEC site to help answering this question.

[✓] Which chip would likely score higher (better) on the SPECspeed2017int benchmarks,
[✓ *Chip A*] or ◯ *Chip B* . [✓] Explain.

Chip A because SPECspeed2017 is run with one benchmark at a time and the SPECcpu benchmarks use few threads, maybe one. For one thread Chip A has a peak performance of 4 insn/cycle, much better than Chip B's peak of 1 insn/cycle.

[✓] Which chip would likely score higher (better) on the SPECrate2017int benchmarks,
◯ *Chip A* or [✓ *Chip B*] . [✓] Explain.

Chip B because a 1-way core would run code more efficiently (fewer stalls) than a 4-way core, and the SPECrate scores are obtained by running many copies of an individual benchmark and so all cores would be used.

(*c*) Our goal is to build a machine that can execute eight floating-point operations per cycle. Two machines are under consideration, an 8-way superscalar system implementing ISA I, and a 2-way superscalar system with an 8-lane vector unit implementing ISA IV, which is like I but with vector instructions. Both machines run at 1 GHz, and both can sustain eight billion floating-point operations per second.

☑ Which machine is likely to be more expensive? ☑ Explain.

The 8-way superscalar. Both machines will have eight sets of FP functional units. But the 8-way superscalar has hardware for decoding eight instructions versus two for the 2-way system. Also the cost of bypass paths in the 8-way superscalar system will be higher than bypass paths in the 2-way system, even when including the vector unit. That's because the vector unit would not have inter-lane bypass paths.

☑ Which machine is likely to be faster on typical code? ☑ Explain with ☑ a code example.

Short answer: The peak FP operation rate of the two systems is identical, but the ISA I implementation is likely faster on typical code because unlike ISA IV, ISA I does not need sets of eight identical operations to realize eight operation per cycle performance.

Description: The ISA I (8-way superscalar) implementation is likely faster than the ISA IV (2-way superscalar plus vector unit) because the ISA IV system can sustain 8 billion floating-point operations (plus one instruction) per second only on code containing one vector instruction per fetch group. On code which is not *vectorizable*, meaning for which vector instructions are not useful or on which vector instructions operate on only one lane, the ISA IV will execute at most only 2×10^9 insn/s. Since most code is not vectorizable the ISA I implementation will be faster. For example, the eight instructions in the code fragment below could be executed at the same time on ISA I. Those eight instructions could not be replaced by a single vector instruction because there are eight different instructions, not the same instruction, say `add`, operating on eight different sets of operands.

```
# Code not vectorizable because each instruction performs a different operation.
 add.d f0, f2, f4
 andi r1, r2, 0xeof
 sub.d f6, f8, f10
 ori r3, r2, 0xfood
 mul.d f12, f14, f16
 xori r4, r2, 0xa8
 add.s f18, f19, f20
 slt r9, r10, r11
```

(*d*) In MIPS and many other RISC ISAs memory accesses must be aligned. For example, a `lw` instruction, which loads a four-byte value, must load from an address that is a multiple of 4. The execution of a `lw` loading from an address that is not a multiple of 4 will result in an exception (and on Linux system resulting in the Bus Error signal handler being called). As we pointed out in class, integer instructions, and especially load and store instructions, in any reasonable ISA would be required to raise precise exceptions. MIPS is certainly reasonable in this respect.

Suppose that a program uses non-aligned addresses in memory accesses, but is otherwise correct. That is, the program would run correctly if the load could handle a non-aligned address. (After all, CISC ISA load instructions can do it.) But on MIPS it raises an exception as soon as a non-aligned load or store is attempted. Suppose further that re-writing the program is not feasible.

☑ Explain how we can take advantage of precise exceptions so that this program would run correctly. A code example would be nice but not necessary.

The execution of a `lw` with a misaligned address will cause execution to reach the exception handler. We will re-write the exception handler so that when a misaligned address exception occurs it will performs the unaligned `lw` using four load byte unsigned instructions (for which there is no alignment restriction). The exception handler will get the attempted load address and the address of the faulting instruction (the load) from MIPS' co-processor 0 registers (or from the appropriate place in some other ISA). Rather than using a `lw` to load the data, the handler will load the data using four `lbu` instructions since they don't have alignment restrictions. The four loaded values will be put together in one register (see the code sample below). Next, the handler routine will need to load the faulting `lw` instruction and determine which register it intended to write. (That is, the instruction will be read from memory as though it were data and the `rt` field value would be extracted using a `srl` and an `andi`.) The handler will place the value in that register, or in the part of the exception handler stack holding the saved value of that register. Finally, the exception handler will resume execution of the interrupted code at the instruction after the faulting load. See the example below.

(In this solution four `lbu` instructions were used. Some ISAs provide instructions specifically intended to load and piece together a misaligned value.)

```
# SOLUTION -- Example of code raising exception.
#
 add r1, r1, r2
 lw r3, 0(r1)       # Note: r1 may not be aligned. If not, handler called ..
 and r8, r8, r3     # .. and later returns to this instruction with r3 loaded.

# SOLUTION -- Part of handler that loads 32-bit value in four pieces.
# Register usage:
#   r1: Attempted load address.
#   r3: Register to put load value in.

# Load the value without risk of misalignment. (For big-endian byte order.)
  lbu r10, 0(r1)    # Most significant byte.
  lbu r11, 1(r1)
  lbu r12, 2(r1)
  lbu r3,  3(r1)    # Least significant byte.

  sll r10, r10, 24  # Move byte to most-significant position.
  or r3, r3, r10    # Combine with least-significant byte.
  sll r11, r11, 16
  or r3, r3, r11
  sll r12, r12, 8
  jr r31
  or r3, r3, r12
```

✓ Explain why it would be impossible if loads only raised deferred exceptions. (Assume that aligned accesses work fine with such loads.)

In a deferred exception the handler starts after the faulting instruction and at least one following instruction finishes execution. Any value written by the faulting instruction will not be correct. (In a precise exception the handler is called after instructions up to but not including the faulting instruction finish execution. All values written will be correct.)

For the handler to emulate unaligned loads it needs to resume execution at the instruction following the unaligned load. In the solution example above the `lw` is the faulting instruction and the **`and`** is the instruction at which execution needs to resume for the program to work correctly. If the exception were deferred then by definition at least the **`and`** and possibly several other instructions would execute before the handler were called. In that case the old value of `r8` would be lost and so there would be no way to return to the **`and`** and execute the code correctly. For example, suppose `r8` were 7 (a correct value) and `r3` were 0 (incorrect, assume that the correct value is 17) when the **`and`** executes the first time (before the handler is called). The execution of **`and`** sets `r8` to 0. Suppose that the handler is called and sets `r3` to 17 (a correct value) and then returns to the **`and`**. The second execution of **`and`** sets `r8` to 0 (the value of `0 & 17`) when it should be set to 1 (the value of `7 & 17`).

(*e*) MIPS has a `slt` (set less than) instruction, but doesn't have a `sge` (set greater than or equal to) instruction. Why not?

☑ Why doesn't MIPS have an `sge` instruction?

Because the exact same operation can be obtained by swapping the two source operands of `slt`. For example, if you need a `sge r1, r2, r3` just use `slt r1, r3, r2`. There is no need to waste a precious opcode.

(*f*) *Note: The following question was asked about two months after in-person classes were ended for the CoViD-19 pandemic.*

Perhaps many of us are wishing we could go back in time. (Not to warn people, that's obviously futile.) Wish granted. You are in a meeting (in person, not Zoom) with future Turing Prize winners discussing which features to put into their new [airquotes] "RISC" ISA, MIPS.

One attendee is advocating for the inclusion of magnitude comparison branch instructions such as `bgt r1,r2 TARG` (branch if `r1` greater than `r2`). But many others oppose the idea because it would have too much critical path impact. "We can include `bgt` with zero critical-path impact if we use a surprisingly simple but effective technique called branch prediction," you say.

☑ Explain how branch prediction can remove the critical path impact that was a concern at the meeting.

MIPS was designed so that branches could be resolved in the `ID` stage of a five-stage pipeline. For any branch instruction that tests register values, the test (such as greater-than) can't start until the register values are retrieved. Since the register values are retrieved in the `ID` stage the critical path includes both the retrieval and the test. This limited the kinds of two-register tests that branches could do to equality comparison but not magnitude comparison. With branch prediction the test is moved from the `ID` stage to the `EX` stage. In the `EX` stage the test starts near the beginning of the clock cycle (as do all other ALU operations) and so more time-consuming tests such as magnitude comparison can easily be done.

(In class we talk about performing branch prediction in the `IF` stage, which is the appropriate place to predict for most implementations.)

☑ Was the phrase *branch prediction* an anachronism at that fictional meeting? Web-search freely to answer this question.

No, branch prediction was in use for years before MIPS was designed (in the mid 1980s). For example, see James Smith's 1981 survey [1] appearing in the 8th International Symposium on Computer Architecture, one of the leading conferences in the area.

[1] Smith, J. E. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Washington, DC, USA, 1981), ISCA '81, IEEE Computer Society Press, p. 135148. `https://dl.acm.org/doi/10.5555/800052.801871`.