Name Solution

Computer Architecture
LSU EE 4720
Final Examination

1 May 2019, 12:30–14:30 CDT

Problem 1		(22 pts)
Problem 2		(22 pts)
Problem 3		(21 pts)
Problem 4		(10 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias Before

Good Luck!

Problem 1: (22 pts) Notice that the execution of the code fragment below suffers two stalls when executing on our 2-way superscalar MIPS implementation. The `add` stalls due to a dependence with `or` and the `sw` stalls due to a dependence with `add`. The MIPS implementation has three unconnected logic blocks that may be useful. Each must be connected to the opcode and func of the instruction in the appropriate slot. The output of the =or is 1 if the instruction is an `or`. The output of uses rs is 1 if the instruction uses the `rs` register as a source, likewise for uses rt.

```
# Cycle           0  1  2  3  4  5  6  7  8
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID -> EX ME WB
 sw r3, 4(r6)        IF -> ID ----> EX ME WB
 addi r6, r6, 8      IF -> ID ----> EX ME WB
```

(*a*) In the execution below the `sw` no longer stalls for `r3`. Add a bypass path that can be used by `sw` to get the `r3` value **in the execution below** but not for other cases.

☑ Add bypass path for `r3` so `sw` executes as shown. ☑ Label the path "Part a", and **do not add** unneeded bypass paths.

Solution appears in blue (several pages ahead). The `sw` needs the value of `r3` written by the `add`. A bypass path was added to the `rtv` mux in the `EX` stage, the path connects to the slot 1 instruction in the `ME` stage. As can be seen from the execution below the `add` is in slot 1 and when the `sw` is in `EX` (in cycle 4) the `add` is in `ME`, and so it can use the added bypass path.

```
# Cycle           0  1  2  3  4  5  6  7  8
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID -> EX ME WB
 sw r3, 4(r6)        IF -> ID EX ME WB
 addi r6, r6, 8      IF -> ID EX ME WB
```

(*b*) The `add` stalls due to the dependence with `or` carried by `r1`. Add control logic that detects such a dependence and connect it to the Stall ID OR gate at the lower right. The output of the logic should be `1` for any true dependence between two instructions in a group.

☑ Provide a stall signal when there is a dependence between the two instructions in `ID`.

Solution appears in green on the diagram a page or two ahead. The =' comparison units check whether the destination of the instruction in slot 0 matches the `rs` or `rt` register of the instruction in slot 1. The uses logic checks whether the `rs` and `rt` fields of the slot-1 instruction specify sources. (In many classroom examples it is assumed that every instruction uses `rs` as a source, here we are being more careful.)

(*c*) Notice that because the second operand is `r0`, the `or` just copies the value in `r2` to `r1`. Therefore the `add` could have used `r2` instead of `r1` and avoided the stall. Design hardware to perform such *substitutions*. The hardware, including control logic, should detect when an `or` is used as a copy (as above) and if so avoid the stall and deliver the correct source operand to the slot-1 instruction.

```
# Cycle           0  1  2  3  4  5
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID EX ME WB
 sw r3, 4(r6)        IF ID EX ME WB
 addi r6, r6, 8      IF ID EX ME WB
```

☑ Detect the substitution opportunity and ☑ suppress the Stall ID signal (from the previous part).

☑ Make sure the slot-1 instruction uses the correct value ☑ and that both instructions execute correctly.

☑ Of course, pay attention to cost. Nothing added for this problem should touch 32 bits.

Solution appears a page or two ahead in purple. The logic at the lower left checks for a slot-0 `or` using `r0` as the `rt` operand. If such a copy is found and if the `rs` source of the slot-1 instruction uses `or` destination the stall is suppressed and the `rs` register *number* of the slot 1 instruction is substituted. The register number is 5 bits, while the value is 32 bits, so it is far less expensive to substitute the number than the value.

(*d*) *The following is a bonus question that did not appear on the original exam. Bonus for whom you ask? Definitely a bonus for those who took the class in the Spring 2019 semester and took a look at the posted exam. Those (you) will have an opportunity to make connections between concepts learned in the class and that will provide a deeper understanding and longer retention.* Yes, the substitution hardware eliminates a stall. Suppose that `r2` had to be copied into `r1`. Provide an argument that substitution hardware is a waste of resources, illustrate with an example. Provide another argument—also with an example—that substitution hardware eliminates a stall that cannot be eliminated in another way. Whether substitution is a good idea will depend on whether the example illustrating its utility is representative of realistically compiled actual code.

☑ Argument against substitution hardware. ☑ Code example.

The substitution hardware is not needed if the compiler can use the source register of the substitution instruction. For the code sample above the compiler would emit `add r3, r2, r4`, eliminating the dependence. See the code below. (It is always better to avoid a stall using a compiler optimization than by hardware changes.)

```
# Cycle          0  1  2  3  4  5   # SOLUTION. No stall, no costly hardware needed!
 or r1, r2, r0   IF ID EX ME WB
 add r3, r2, r4  IF ID EX ME WB     # Replacement for add r3, r1, r4.
 sw r3, 4(r6)       IF ID EX ME WB
 addi r6, r6, 8     IF ID EX ME WB
```

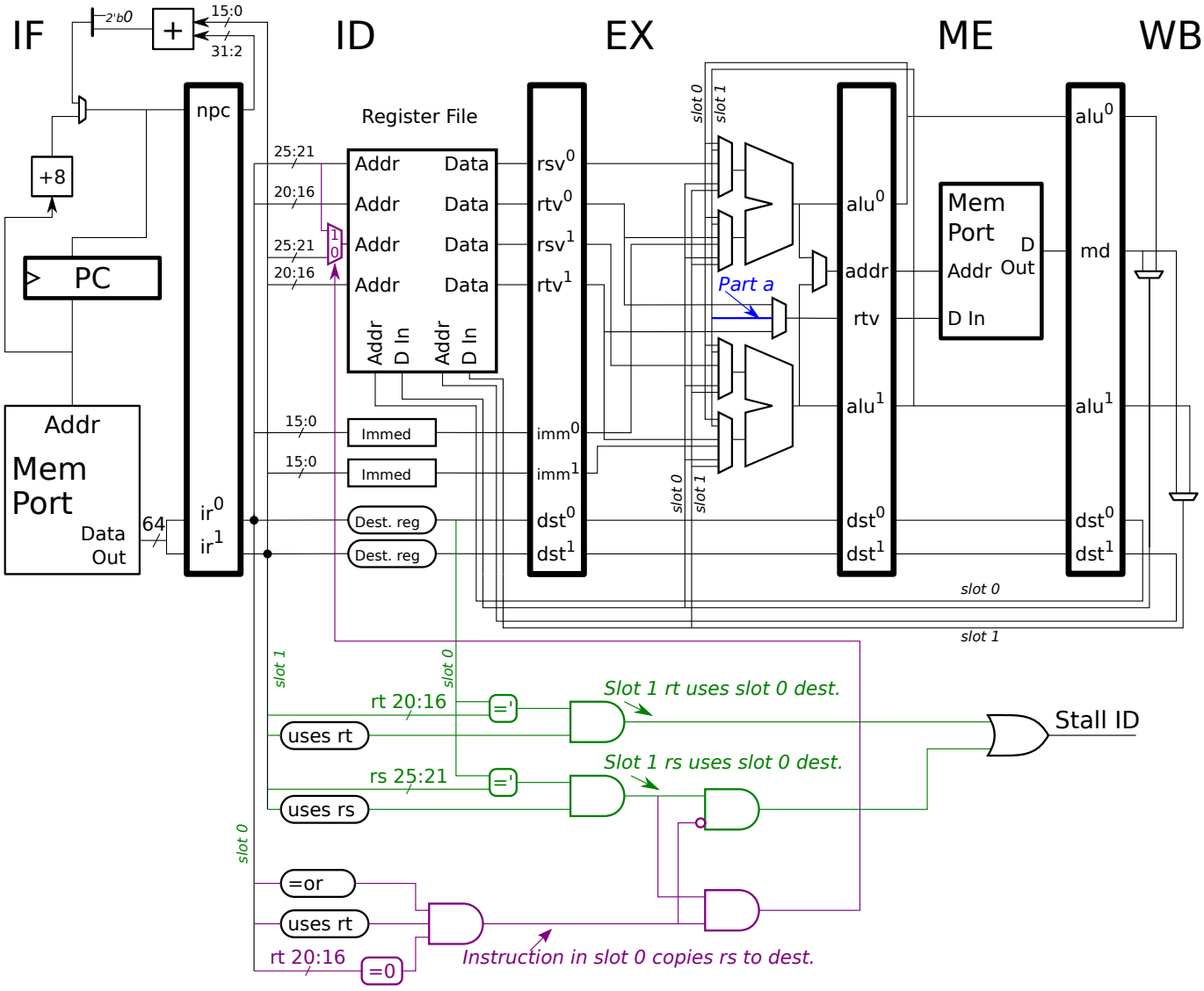
☑ Argument for substitution hardware. ☑ Code example.

The compiler cannot easily perform substitution when the two instructions are in different basic blocks. For example, in the code below the compiler cannot just replace `add r3, r1, r4` with `add r3, r2, r4` because that would not be correct if the `bne` were not taken. This is a weak example for the substitution hardware because the compiler could prepare two code sequences, both containing the `add`, `sw`, and `addi` instructions.

```
 # SOLUTION
 bne r8, r9 TARG  # If branch taken add uses r1, if not taken r2.
 nop
 or r1, r2, r0
TARG:
 add r3, r1, r4
 sw r3, 4(r6)
 addi r6, r6, 8
```

(A basic block is a sequence of instruction that can only have a control transfer [such as a branch] at the end, and which can only have a branch label [labels `TARG` and `LOOP` are frequently used in class] at the beginning. A basic block always starts with the first instruction and always ends with the last instruction.)

Solution to hardware questions appears below. A discussion of the solution appears on prior pages.




```
# SOLUTION:       Use this execution to help in understanding the solution.
# Cycle           0  1  2  3  4  5
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID EX ME WB
 sw r3, 4(r6)        IF ID EX ME WB
 addi r6, r6, 8      IF ID EX ME WB
# Cycle           0  1  2  3  4  5
```

Problem 2: (22 pts) Appearing below is our MIPS FP pipeline with the comparison units added.

(*a*) Show the execution of the following fragment on this hardware.

☑ Show execution up to second fetch of `lwc1`. ☑ Pay attention to dependencies, including the FP condition.

Solution appears below. The second fetch of `lwc1` is shown using static instruction order. Note that `c.lt.s` depends on `add.s f2` (carried by `f2`) and that `bc1t` depends on `c.lt.s` (carried by `fcc`).

```
#                 SOLUTION
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
LOOP:
 lwc1 f1, 0(r1)   IF ID EX ME WF                   IF
 add.s f2, f1, f2    IF ID -> A1 A2 A3 A4 WF
 add.s f4, f1, f3       IF -> ID A1 A2 A3 A4 WF
 c.lt.s f2, f6             IF ID ----> C1 C2 WF
 bc1t LOOP                    IF ----> ID ----> EX ME WB
 addi r1, r1, 4                        IF ----> ID EX ME WB
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 swc1 f2, 4(r1)
 and r1, r1, r9
```

(*b*) Notice that there are two circled letters (in blue) in the lower part of the diagram. For each letter provide a code fragment that causes the labeled wire to go to logic 1.

☑ Code fragment that makes A logic 1.

☑ Show its execution and ☑ indicate cycle at which A is 1.

Solution appears below. Wire A will be 1 in cycle 3, that's when the `lwc1` is in `ID` while the `add` is in `A2`. The `lwc1` must stall one cycle to avoid a `WF` structural hazard in cycle 6.

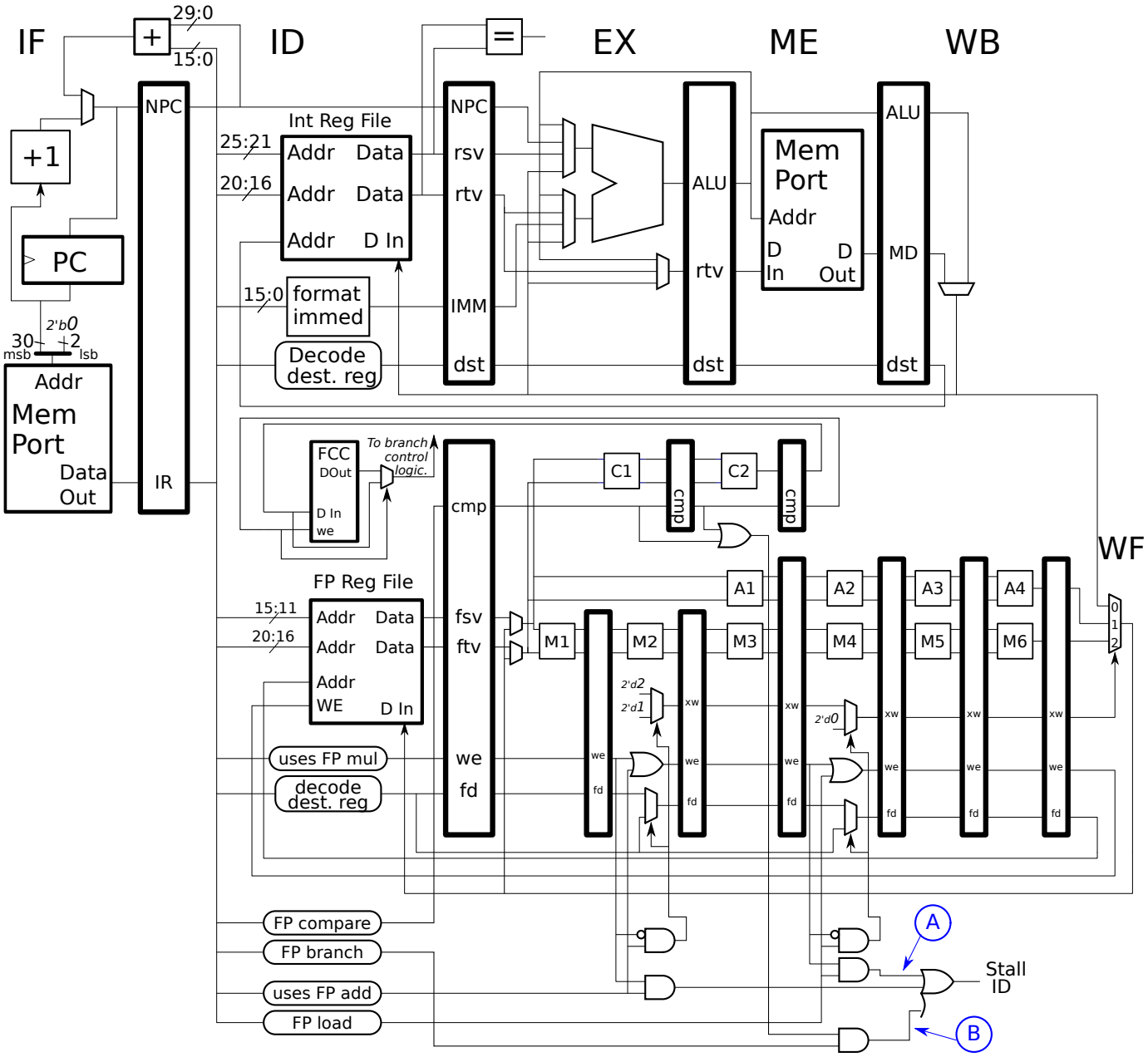
```
#                 SOLUTION
# Cycle           0  1  2  3  4  5  6  7
 add.s f1, f2, f3 IF ID A1 A2 A3 A4 WF
 xor r1, r2, r3      IF ID EX ME WB
 lwc1 f4, 0(r4)         IF ID -> EW ME WF
# Cycle           0  1  2  3  4  5  6  7
```

☑ Code fragment that makes B logic 1.

☑ Show its execution and ☑ indicate cycle at which B is 1.

Solution appears below. Wire B will be 1 in cycles 2 and 3, that's when `bc1t` has to wait for the comparison result to reach the FCC.

```
#              SOLUTION
# Cycle        0  1  2  3  4  5  6  7
 c.lt.s f0, f1 IF ID C1 C2 WF
 bc1t TARG        IF ID ----> EX ME WB
```


IF
ID
EX
ME
WB
WF
29:0
15:0
+
=
+1
PC
NPC
IR
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out
Int Reg File
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
15:0
format
immed
Decode
dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
FCC
DOut
D In
we
To branch
control
logic.
cmp
C1
C2
cmp
cmp
FP Reg File
15:11
20:16
Addr
Data
Addr
Data
Addr
WE
D In
fsv
ftv
we
fd
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
2'd2
2'd1
2'd0
xw
we
fd
0
1
2
uses FP mul
decode
dest. reg
FP compare
FP branch
uses FP add
FP load
A
B
Stall
ID

Problem 3: (21 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on two systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and one system has a local predictor with an 8-outcome local history.

Branch `B1` has a repeating pattern, two repetitions are shown. Branch `B2` repeatedly and randomly emits three sequences, `a`, `b`, and `c`. Sequence `a` is `NT`, sequence `b` is `NNNTT` (five outcomes), and sequence `c` is `NNNN NTTT` (eight outcomes). After finishing one sequence, a new one is started. Sequence `a` is chosen with probability .4, sequence `b` with probability .5, and `c` with probability .1.

Here are some examples of `B2` outcomes, with spaces placed between the sequences for clarity. Example 1: `NT NNNTT NT NNNNNTTT` (that's `a`, `b`, `a`, `c`). Example 2: `NNNNNTTT NT NT NNNTT NNNTT` (that's `c`, `a`, `a`, `b`, `b`).

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

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

B2:    (a, p=.4): NT    (b, p=.5): NNNT T    (c, p=.1): NNNN NTTT
```

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

```
    SOLUTION WORK
    0   1  0  0  0  0  1   2   3  2  1  0  0  1  2  <- 2-bit counter values
B1:   T  N  N  N  N  T  T    T  N  N  N  N  T  T
      x              x  x       x  x        x  x   <- Prediction outcomes
                          ----------------------  <- First occurrence repeating pattern.
```

Short answer: $\frac{3}{7}$, see work above.

Explanation: The line just below `SOLUTION WORK` (above) shows 2-bit counter values for `B1`, assuming that 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. Such a pattern is underlined. Note that the counter value at the start and end is the same and that the underlined part corresponds to a set of repeating branch outcomes. The prediction accuracy is $\frac{(7-4)}{7} = \frac{3}{7}$.

☑ What is the accuracy of the local predictor on branch `B1`?

The pattern can easily fit the 8-outcome local history, so the accuracy is 100%.

☑ What is the accuracy of the local predictor on branch `B2`?

First, lets number the outcomes on each pattern:

```
a: 01
   NT

b: 0123 4
   NNNT T

c: 0123 4567
   NNNN NTTT
```

Start by considering the prediction of the first outcome of each of the three patterns. Then because one of the three patterns has just finished the local history at this point will be either `*****TNT`, `**TNNNTT`, or `NNNNNTTT`, where the `*` can be either `N` or `T`. The first outcome of all three are `N` and so the `PHT` entries for all the local histories will be zero (after warmup) and so the first outcome of all three patterns will be predicted correctly (assuming that the local histories only occur at the beginning, which is easy to verify). After the first outcome is predicted the local histories will be one of `****TNTN`, `*TNNNTTN`, or `NNNNTTTN`. If pattern `a` is occurring the next outcome is `T`, otherwise `N`. Pattern `a` occurs with probability .4 and so it is more likely that an `N` occurs. The `PHT` entries for the local histories are more likely to be 0 or 1, but it's not certain because it is possible that there are two consecutive occurrences of `a`. So for pattern `a` the first outcome is predicted with 100% accuracy and the second outcome at a lower accuracy, which can be approximated as 50% and which can be solved exactly using a Markov chain.

If outcome 1, the second outcome using the number above, is `N` then outcome 2 will be predicted with 100% accuracy because it is always `N`. That is the local histories `***TNTNN`, `TNNNTTNN`, `NNNTTTNN` only appear when the next outcome is `N` and so the `PHT` entries will reach zero after warmup. Outcome 3 will be predicted `T` nearly 100% of the time because `b` is much more likely than `c`. Outcomes 4 and later will be predicted perfectly because they are unambiguous.

Here is the number of correct predictions for each pattern: `a`, $1 + 0.5 = 1.5$; `b` $1 + 0.5 + 1 + 1 + 1 = 4.5$, and `c`=$1 + 0.5 + 1 + 0 + 1 + 1 + 1 + 1 = 6.5$. The accuracy during `a` is $\frac{1.5}{2} = .75$, during `b` is $\frac{4.5}{5} = .9$, and during `c` is $\frac{6.5}{8} = .8125$. To compute the overall prediction accuracy we need to weight these by how frequently they occur. That is:

$$\frac{.4 \times 1.5 + .5 \times 4.5 + .1 \times 6.5}{.4 \times 2 + .5 \times 5 + .1 \times 8} = .853659$$

.

☑ What is the accuracy of the bimodal predictor on branch `B2`?

The problem is solved by first considering the effect that each pattern has on the 2-bit counter. Consider pattern `b`. It starts with three `N`'s guaranteeing that the counter will be zero when the first `T` is predicted, and that the counter will be 2 after the second `T`. Similarly, the counter will be 3 after sequence `c` completes. Unless the counter is zero, pattern `a` does not change the counter. Therefore, at the start of a sequence the counter will be either 2 or 3. It will be 2 with probability $P(k = 2) = \frac{.5}{.5+.1} = \frac{5}{6}$ and 3 with probability $P(k = 3) = \frac{.1}{.5+.1} = \frac{1}{6}$, where $P(k = x)$ is the probability that the 2-bit counter is x. Let $P(a|k)$ denote the prediction accuracy during pattern `a` given that the 2-bit counter value is k and remember that $k \in \{2, 3\}$. Then $P(a|2) = 0$, $P(a|3) = \frac{1}{2}$, $P(b|2) = \frac{2}{5}$, $P(b|3) = \frac{1}{5}$, $P(c|2) = \frac{5}{8}$, and $P(c|3) = \frac{4}{8}$. The probabilities need to be properly combined. First,

$$P(\checkmark|a) = P(a|2)P(k = 2) + P(a|3)P(k = 3) = 0 \times \frac{5}{6} + \frac{1}{2}\frac{1}{6},$$

where $P(\checkmark|a)$ is the prediction accuracy during pattern `a`. Similarly, $P(\checkmark|b) = \frac{2}{5}\frac{5}{6} + \frac{1}{5}\frac{1}{6} = \frac{11}{30}$ and $P(\checkmark|c) = \frac{5}{8}\frac{5}{6} + \frac{4}{8}\frac{1}{6} = \frac{29}{48}$.

Each pattern's probability needs to be weighted by how often the pattern occurs. Pattern `a` has a respectable probability of .4 but only two outcomes, so its relative contribution is $.4 \times 2$. The relative contribution of patterns `b` and `c` are $.5 \times 5$ and $.1 \times 8$. If we arrived at some random time and waited for the next `B2` execution the probability of `B2` being in pattern `a` would be $\frac{.4\times2}{.4\times2+.5\times5+.1\times8} = \frac{.8}{4.1} \approx .195$.

So the overall branch prediction ratio for `B2` is

$$\begin{aligned} P(\checkmark|\texttt{B2}) &= \frac{P(\checkmark|a).4 \times 2 + P(\checkmark|b).5 \times 5 + P(\checkmark|c).1 \times 8}{.4 \times 2 + .5 \times 5 + .1 \times 8} \\ &= \frac{44}{123} \approx .357724 \end{aligned}$$

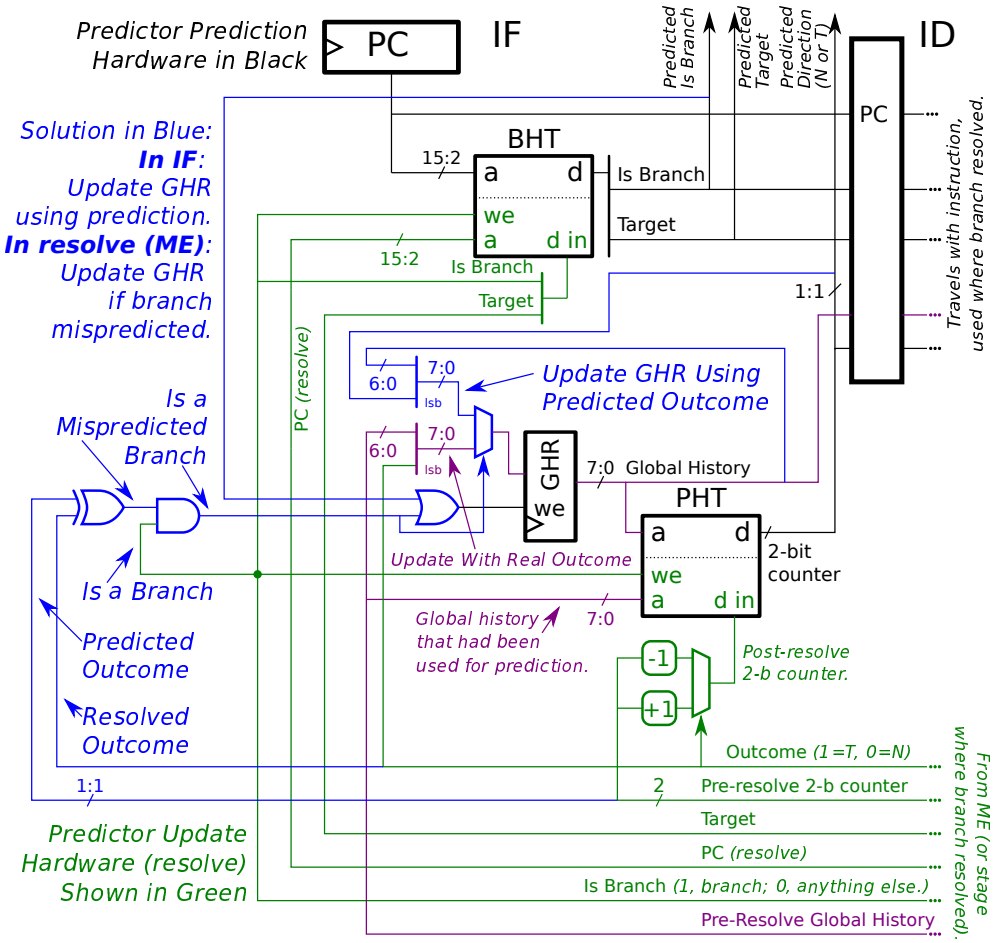
(*b*) Appearing below is a diagram of our global predictor. Notice that the GHR is not updated until the branch resolves. Modify the predictor so that the GHR is updated when the branch is being predicted (in `IF`) using the *predicted* outcome. When the branch resolves check whether the prediction was correct, and if not (if it was mispredicted) write the correct history into the GHR.

The following is interesting background material omitted from the original exam. The importance of updating the GHR using the predicted outcome increases with the number of post-branch instructions that are in the pipeline at the time a branch resolves. Consider our five-stage pipeline with branches resolving in `ME`. In that case there are just three post-branch instructions. For an 8-way superscalar pipeline there would be $3 \times 8 = 24$ instructions. One or more of those 24 instructions could itself be a branch. In the unmodified design below those branches would have been predicted with a GHR that lacked the outcome of the resolving branch and those that followed. The problem is much greater in dynamically scheduled systems where over 100 instructions can be *in flight*. For that reason global-like predictors in dynamically scheduled systems use designs like the one requested for this problem.

☑ Add hardware to detect whether the resolving branch has been mispredicted.

☑ During prediction write GHR based on prediction, ☑ during resolve apply corrected GHR if branch mispredicted.

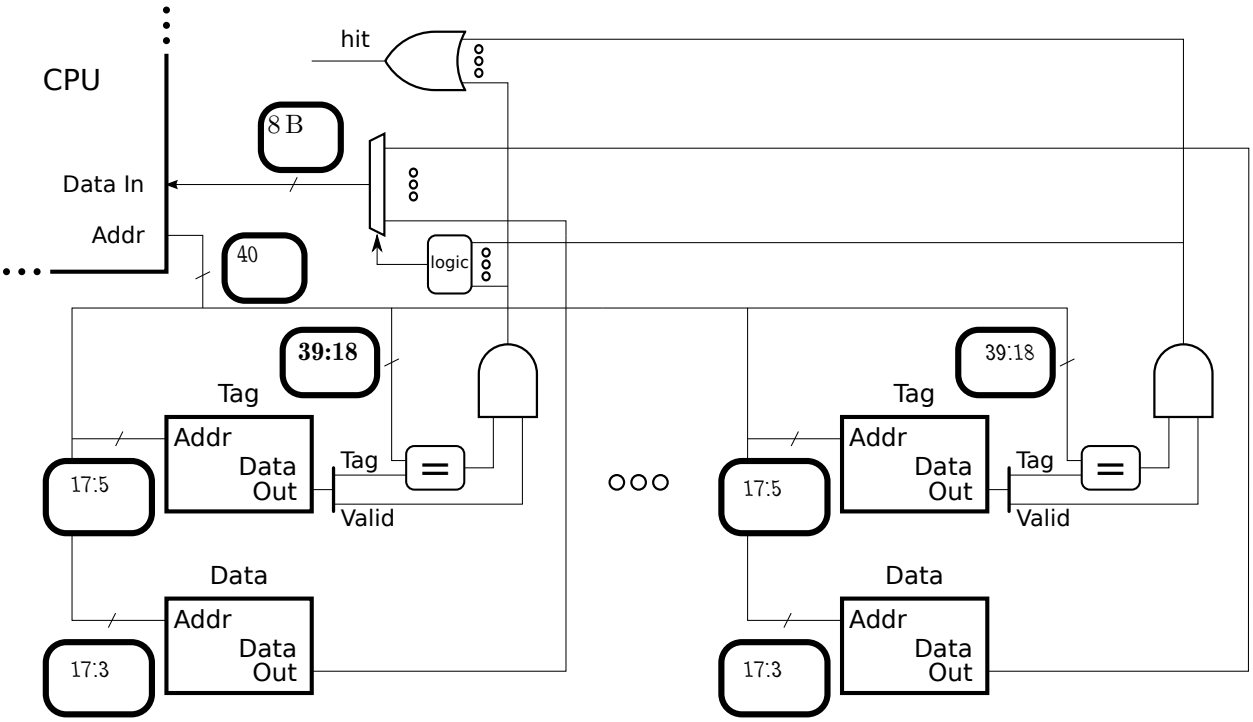
Solution appears below in blue. The mux at the input to the GHR selects either an updated global history for the branch in `IF` (which we hope to be the frequent case) or a corrected global history for the branch in `ME` (also identified as *resolve*). The latter case is only used if the branch is mispredicted, which is determined by XORing the predicted and resolved outcomes (if they are different the result is 1) and also making sure that the instruction in `ME` is a branch.



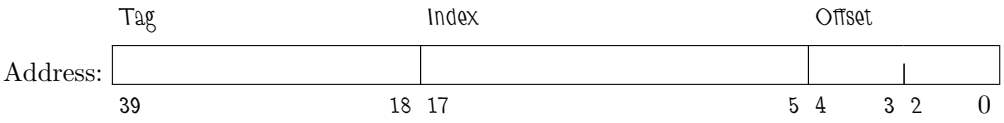
Problem 4: (10 pts) The diagram below is for a 4 MiB set-associative cache with a line size of 32 B. The character size is the usual 8 bits. Other information about the cache can be deduced using hints in the diagram. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $32 = 2^5$.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)



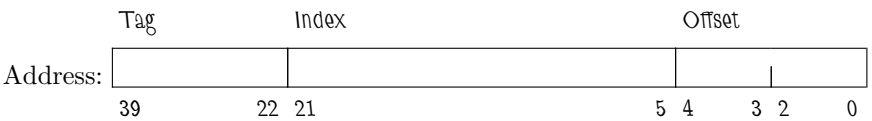
☑ Associativity:

The associativity is 16. The associativity is determined based on the given cache capacity, 2^{22} bytes, and the capacity of an individual data store, 2^{18} bytes. Since the cache capacity is the sum of the data store sizes, the associativity must be $\frac{2^{22}}{2^{18}} = 2^{22-18} = 16$.

☑ Memory Needed to Implement ☑ Indicate Unit!!:

It's the cache capacity, 4 MiB plus $16 \times 2^{18-5}\,(40 - 18 + 1)$ b = 4 MiB + 3014656 b.

☑ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.



The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 64 B (which is 2^6 B). The code fragment starts with the cache empty; consider only accesses to the array.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
int ILIMIT = 1 << 11;    // = 2^11

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^6 = 64$ bytes is given. The size of an array element, which is of type int, is $4 = 2^2$ B, and so there are $2^6/2^2 = 2^{6-2} = 2^4 = 16$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, and so the next $2^4 - 1 = 15$ accesses will be to data on the line, hits. The access at `i=16` will miss and the process will repeat. Therefore the hit ratio is $\frac{15}{16}$.

Problem 5: (25 pts) Answer each question below.

(*a*) Appearing below are simple C routines and corresponding MIPS assembler code. C variable names match the MIPS registers to which they were assigned. Register `v0` is used for the return value. The first C routine, `proc1`, operates on 32-bit signed integers. Further below are two similar C routines, `proc2` and `proc3`, each followed by the MIPS routine written for `proc1`—which is wrong because the MIPS routine is only correct for `proc1`. Rewrite those MIPS routines for `proc2` and `proc3`. Note that `int16_t` is a signed 16-bit integer and `uint8_t` is an unsigned 8-bit integer.

```
int32_t proc1(int32_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Code below is correct for proc1.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

☑ Modify MIPS code for `proc2`. Pay attention to ☑ size and ☑ sign. ☑ Eliminate any unneeded instructions.

```
int16_t proc2(int16_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc2.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

```
## SOLUTION
sll $t0,   $a1,   1    # Element size is 16 bits which is 2 chars, so mult by 2.
add $t0,   $t0,   $a0
lh $v0,   0($t0)       # Because element is two bytes change lw to lh ..
lh $t1,   2($t0)       # .. and change offset to 2.
jr $ra
add $v0,   $v0,   $t1
```

☑ Modify MIPS code for `proc2`. Pay attention to ☑ size and ☑ sign. ☑ Eliminate any unneeded instructions.

```
uint8_t proc3(uint8_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc3.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

```
## SOLUTION
#                         Element size is 8 bits, or one byte.
add $t0,   $a1,   $a0   # No need to scale a1 (mult by elt size) before adding it to a0.
lbu $v0,   0($t0)       # Element size is one byte and unsigned ..
lbu $t1,   1($t0)       # .. so load using lbu and use offset of 1.
jr $ra
add $v0,   $v0,   $t1
```

(*b*) The statement below is based on a lack of understanding of how compilers work. Explain the misunderstanding and otherwise correct the statement.

> *It takes a great deal of effort to write a correct and effective compiler optimizer. Therefore optimizers are written for popular high-level languages such as C++11 but not for less popular languages such as COBOL.*

☑ The misunderstanding about compilers is:

...that optimization is performed on high-level code. In fact, optimization is mostly performed on an *intermediate representation* which typically is the same for all high-level languages the compiler can handle. So effort on optimizing the intermediate representation would benefit all those high-level languages.

It is the compiler *front end* that translates high-level languages into the intermediate representation.

☑ How does that change the conclusion about which languages get better optimization?

The conclusion should be that improvements to a compiler's optimizer benefit all high-level languages that the compiler supports.

(*c*) 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.

☑ Compute the peak execution rate in units of instructions per second of ☑ Chip A and ☑ Chip B.

Each chip can execute 20 instructions per cycle or 20×10^9 insn/s.

☑ Why would Chip A run faster on simple code, such as the routines used in the homework assignments?

It's reasonable to assume that "simple" code is single-threaded (not parallelized) and so it will run on only one core. A Chip A core executes at up to 4 IPC, ideally four times faster than a Chip B core which runs at just 1 IPC. Though in typical circumstances Chip A won't be 4 times faster, it will still be better than Chip B for single-thread code.

☑ Which chip might be less expensive? Explain.

Chip B. The number of ALU bypass paths (multiplexor inputs) for a w-way, five-stage implementation might be $4w^2$. The total number of bypass paths for Chip A is $5 \times 4 \times 4^2 = 320$ and the total number for Chip B is $20 \times 4 \times 1^2 = 80$, which is a lot less. What would you rather do, find the money to buy Chip A or parallelize your code so that it can run faster on Chip B? Of course, the answer depends on the situation.

(*d*) Answer the following questions about ISAs.

☑ Implementations of VLIW ISAs are supposed to be less costly and have higher performance than superscalar implementations of conventional ISAs. Is Intel Itanium a good example of that? Explain

No. Intel loaded Itanium with costly features, such as a large number of registers with rotating windows (for use in software pipelining), so there was no apparent cost benefit. The performance was not spectacular either, some blamed contemporary compilers for not being able to properly exploit those costly features.

☑ What important concept came out of the development IBM System/360?

That of an "Instruction Set Architecture (ISA)" (spoken using air-quotes) which would describe what the hardware would do, but not how it would do it. The ISA was intended for implementations across a product line (low- to high-end) and for implementations in the future, perhaps as long as 15 years.

They also made use of computers to prepare documentation. [I'm tempted to use my air quotes again for the phrase "word processing" but I'm not sure it originated with the 360 project, and if I start looking into it now who knows when I'd get back to real work.]

☑ VAX is a good example of which ISA type?

CISC.

☑ True or false: IA-32 (a.k.a. x86) was widely adopted because of its elegant design? Explain.

False! The original ISA was not designed for a long life. It had many compromise features such as requiring a pair of registers to specify a 32-bit memory address.

(*e*) The SPECcpu benchmarks can be run at two tuning levels, *base* and *peak*. Base scores are useful to those running software developed using typical practices.

☑ What kind of computer buyers should use peak scores?

Buyers who intend to develop or buy code highly tuned for the machine it will run on. Such buyers don't mind spending lots of money or effort to achieve 5% higher performance than those other guys get using standard development practices.

☑ How do the SPEC rules for preparing base and peak runs differ?

The base rules dictate that all benchmarks using the same language shall use the same optimization flags. This might reflect the practice of an experienced programmer having a good set of flags (and for most, that would be -O3) that would be used from project to project. The peak rules allow each benchmark to use different optimization flags. That might reflect the practice of a programmer obsessively tweaking flags to get the best performance. (Most programmers in that situation should look to their own code first to find opportunities for improving performance.)