Name Solution

# Computer Architecture
# EE 4720
# Final Examination

8 May 2012, 12:30–14:30 CDT

| | | |
|---|---|---|
| Problem 1 | | (15 pts) |
| Problem 2 | | (10 pts) |
| Problem 3 | | (10 pts) |
| Problem 4 | | (20 pts) |
| Problem 5 | | (15 pts) |
| Problem 6 | | (30 pts) |
| Exam Total | | (100 pts) |

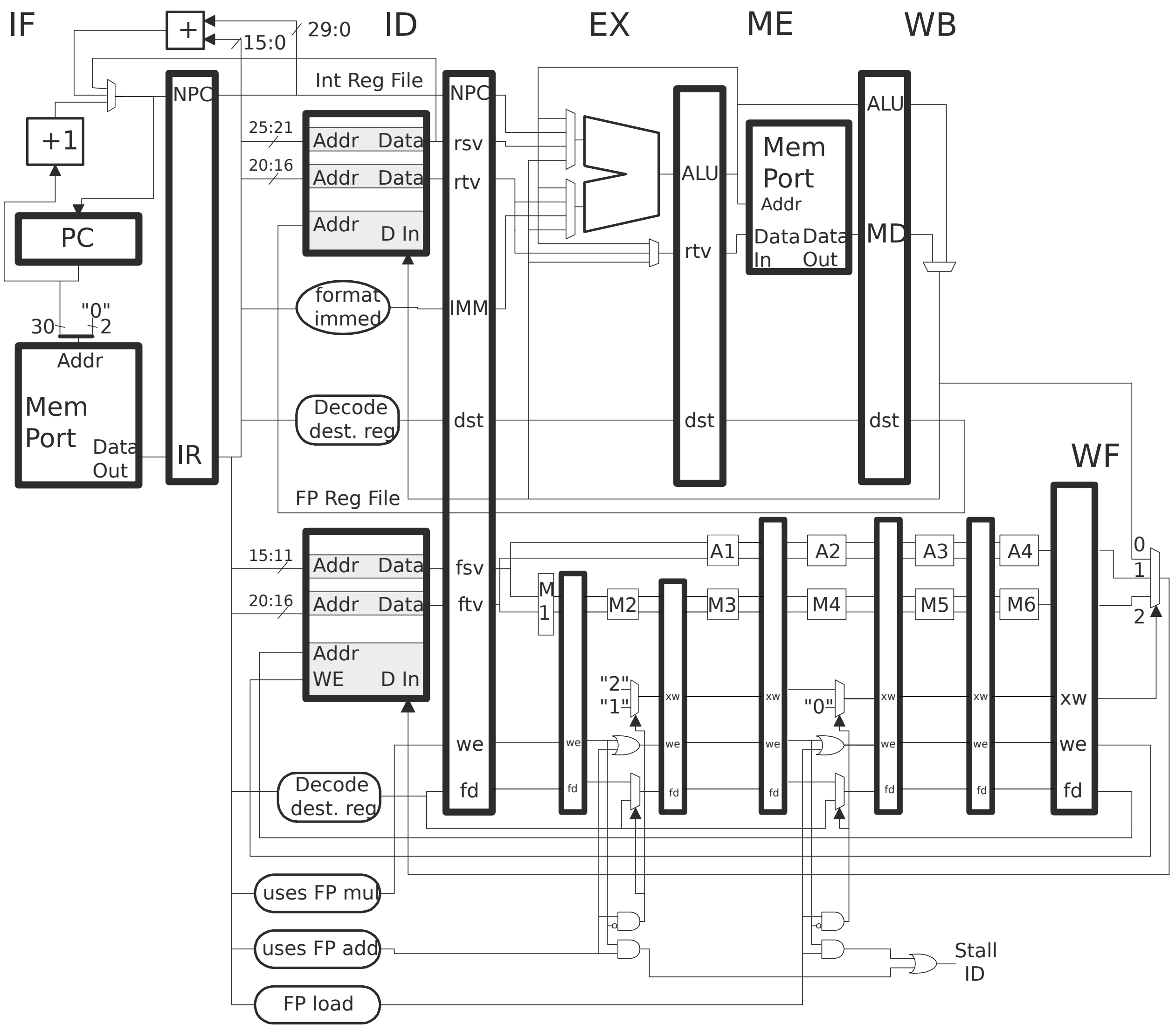
Alias Click Here

*Good Luck!*

Problem 1: (15 pts) Consider the following method for implementing the MIPS32 integer multiply instruction `mul` (the one that writes ordinary registers, not to be confused with `mult`) on the implementation below. The full set of stages `M1` to `M6` perform floating point multiply, but stages `M2` to `M4` perform integer multiplication. The integer multiply `mul` will read and write registers from the integer register file but will use `M2` to `M4` to perform the multiplication. A sample execution appears below. *Grading Note: In the original exam there was an ID-stage stall in cycle 6, implying that there was no* `WB` *to* `EX` *bypass for the* `mul`*.*

```
# Cycle          0  1  2  3  4  5  6  7  8
add r1, r2, r3   IF ID EX ME WB
mul r4, r1, r5      IF ID M2 M3 M4 WB
xor r9, r10, r11       IF ID -> EX ME WB
sub r6, r4, r7            IF -> ID EX ME WB
```

*USE NEXT PAGE FOR SOLUTION*

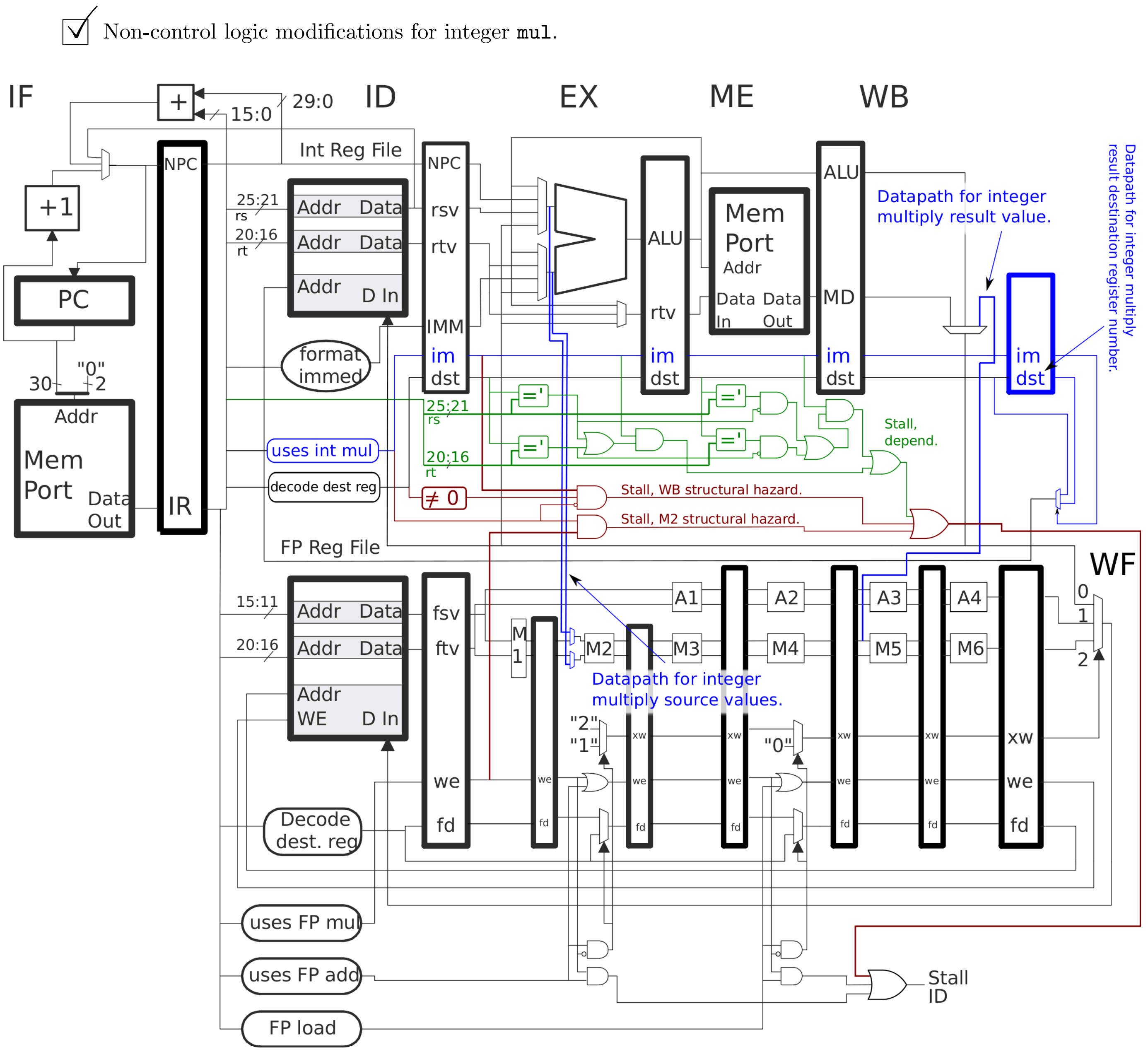


*USE NEXT PAGE FOR SOLUTION*

*CONTINUED ON NEXT PAGE*

(*a*) Modify the implementation below so that `mul` uses M2 to M4. Provide any necessary bypass paths so that the code above executes as shown. For this part do not consider control logic. *Hint: Pay attention to source and destination registers.*

☑ Non-control logic modifications for integer `mul`.



Solution appears in blue above. Connections for the integer multiply source values have been added from the ALU inputs to the inputs of M2. A connection for the result value has been added from the M4/M5 pipeline latch to the WB-stage multiplexor. A new pipeline latch has been added to hold the destination register number (`dst`), creating a sixth stage in the integer pipeline which is only used by the integer multiply instruction. A new mux selects the destination register in the fifth or sixth stage.

Starting the source value connection at the *output* of the ALU multiplexor provides access not just to the `rsv` and `rtv` values, but also to values bypassed from ME and WB. In this solution source values must pass through two multiplexors before reaching M2. A higher-cost, but potentially faster, solution would take the source values from `EX.rsv` and `EX.rtv`, but the M2 multiplexors would need additional inputs for bypassed values. Either solution would get full credit.

The destination value is taken from the output of the M4/M5 pipeline, where it will be available at the beginning of the cycle, to give sufficient time for the register to write back the value. Credit was deducted for solutions that took the value directly from the output of M4 to the WB multiplexor, because this would likely increase the critical path length and so lower clock frequency.

(*b*) Add control logic related to this implementation of `mul` to detect the structural hazard on `M2` and on `WB`. Also add control logic needed for a data dependence between `mul` and an immediately following instruction. All those signals should connect to the existing *Stall ID* signal and use a new $\boxed{\text{uses int mul}}$ logic block.

☑ Control logic for `WB` structural hazard, ☑ `M2` structural hazard, ☑ and the data dependence.

The control logic for the structural hazards appears in maroon and the control logic for the dependence appears in green.

A stall for the WB structural hazard is generated when there is an integer multiply instruction in EX and there is an instruction in ID, except integer multiply, that needs to write back an integer register (in which case the destination register would be nonzero). See the logic generating the signal labeled Stall, WB structural hazard. Such a stall occurs in cycle 2 in Example S1 below.

The stall for the M2 structural hazard is generated when there is an integer multiply in ID and a floating-point multiply in M1, this is detected with the AND gate whose output is labeled Stall, M2 structural hazard. See cycle 2 in Example S2 below.

```
# SOLUTION -- Example S1, for the WB structural hazard.
 # Cycle          0  1  2  3  4  5  6
 mul r1, r2, r3   IF ID M2 M3 M4 WB
 add r4, r5, r6      IF ID -> EX ME WB

# SOLUTION -- Example S2, for the M2 structural hazard.
 # Cycle            0  1  2  3  4  5  6   7  8
 mul.s f4, f5, f6   IF ID M1 M2 M3 M4 M5 M6 WF
 mul r1, r2, r3        IF ID -> M2 M3 M4 WB
```

The dependence control logic, shown in green, checks whether the source register numbers of the instruction in ID matches the destination register numbers of the instructions in EX and ME, and the instruction is an integer multiply. The logic ignores the ME-stage destination register if there is already a dependence with the EX-stage destination register (because of the WB structural hazard this particular situation can't happen for the `mul` but might occur for instructions that write back normally).

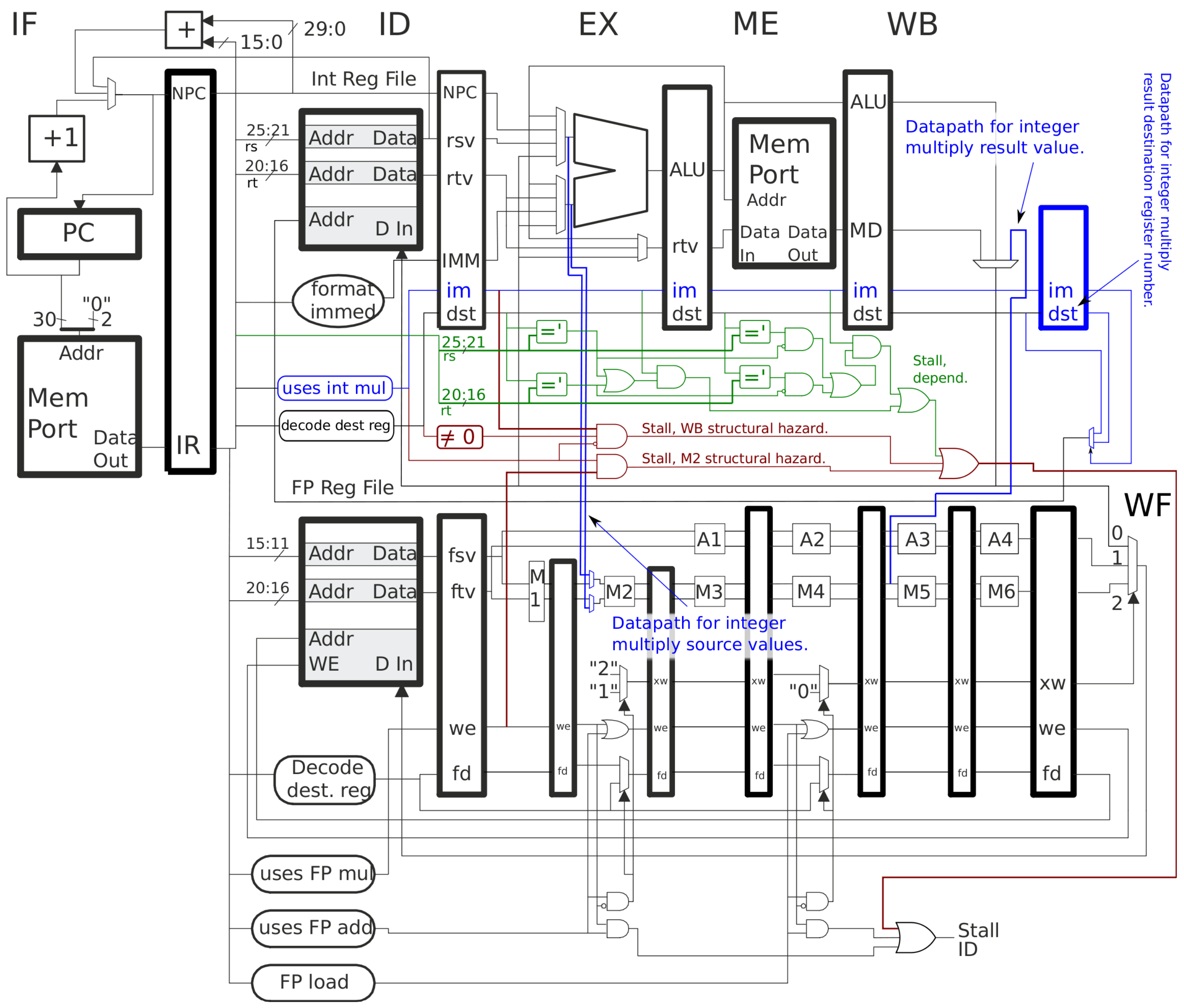
In example D1 below the `add` stalls two cycles due to a dependence with the integer `mul`. In cycle 2 the `rs` register of the instruction in ID, the `add`, matches the destination of the instruction in EX, the `mul`, and so there is a stall. (For the `mul`, EX and M2 are the same stages.) In cycle 3 the `add` is still in ID but the `mul` has moved to ME (equivalent to M3). The logic again detects the dependence and so there is another stall.

Notice that in cycle 2 there are two reasons for the stall, the dependence discussed immediately above, but also the WB structural hazard. If the WB structural hazard stall always occurred when a `mul` instruction were in EX then there would be no reason to check for a dependence with the EX-stage instruction. But the WB structural hazard stall is not generated if the instruction in ID does not write a register, and for those cases the EX stage must be checked for a dependence. See example D2.

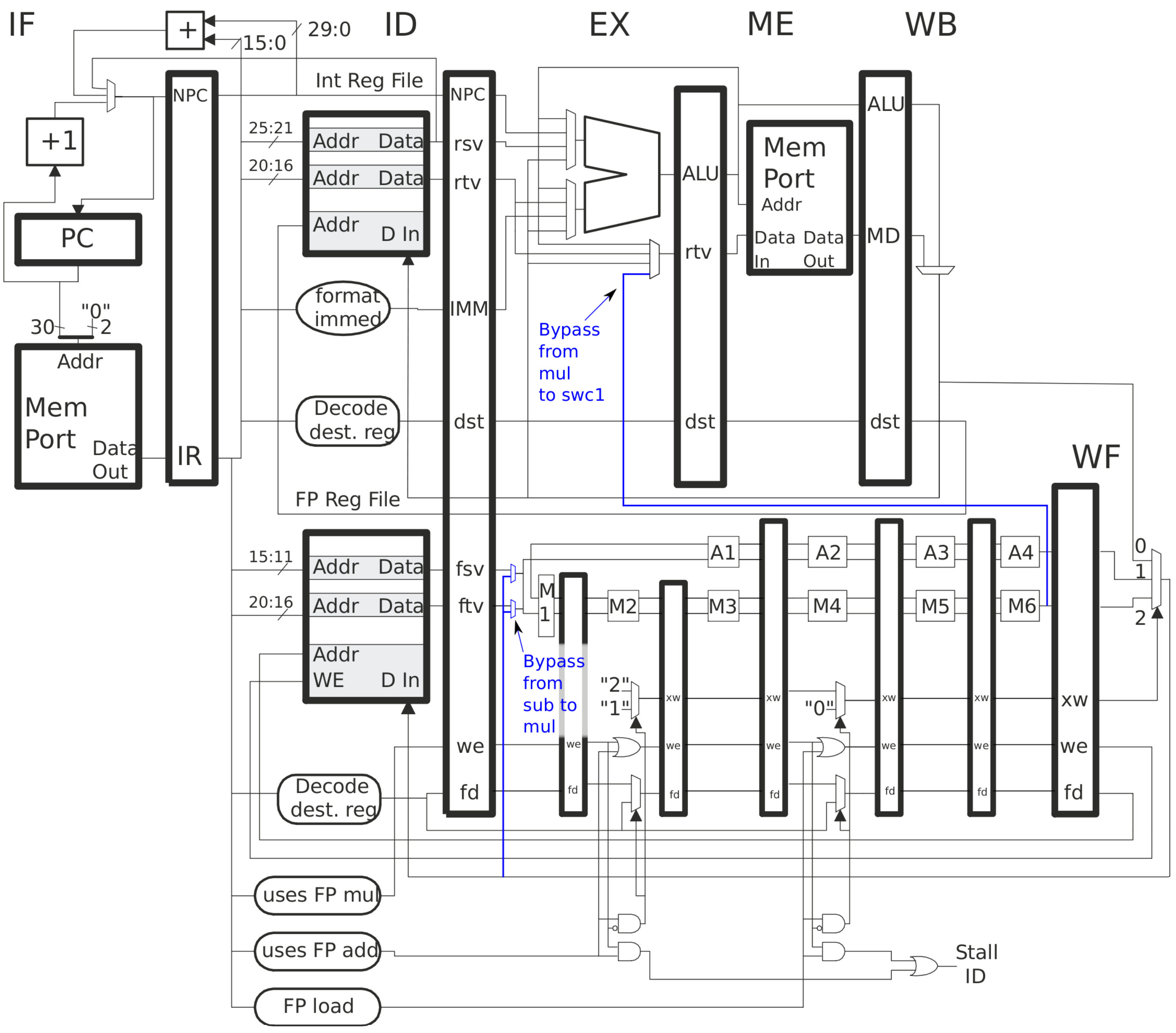
In example D2 the `add` is replaced by a `sw`. The stall in cycle 2 is due only to the dependence through register `r1`, there is no structural hazard. The stall in cycle 3 is also due to the dependence.

```
 # SOLUTION - Example D1, mul dependence with next instruction.
 # Cycle          0  1  2  3  4  5  6  7
 mul r1, r2, r3   IF ID M2 M3 M4 WB
 add r3, r1, r4      IF ID ----> EX ME WB

 # SOLUTION - Example D2, mul dependence with next instruction.
 # Cycle          0  1  2  3  4  5  6  7
 mul r1, r2, r3   IF ID M2 M3 M4 WB
 sw r1, 0(r4)        IF ID ----> EX ME WB
```


IF
ID
EX
ME
WB
WF
29:0
15:0
Int Reg File
FP Reg File
NPC
IR
PC
+1
Mem Port
Addr
Data Out
rsv
rtv
IMM
im
dst
ALU
MD
Data In
format immed
uses int mul
decode dest reg
≠ 0
Datapath for integer multiply result value.
Datapath for integer multiply result destination register number.
Datapath for integer multiply source values.
Stall, depend.
Stall, WB structural hazard.
Stall, M2 structural hazard.
25:21 rs
20:16 rt
15:11
fsv
ftv
we
fd
xw
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
"2"
"1"
"0"
Decode dest. reg
uses FP mul
uses FP add
FP load
Stall ID

Problem 2: (10 pts) Show the execution of the instructions below on the illustrated implementation. Add any needed datapath and reasonable bypass paths.



☑ Show execution. ☑ Add datapath and reasonable bypass paths. ☑ Double-check for dependencies.

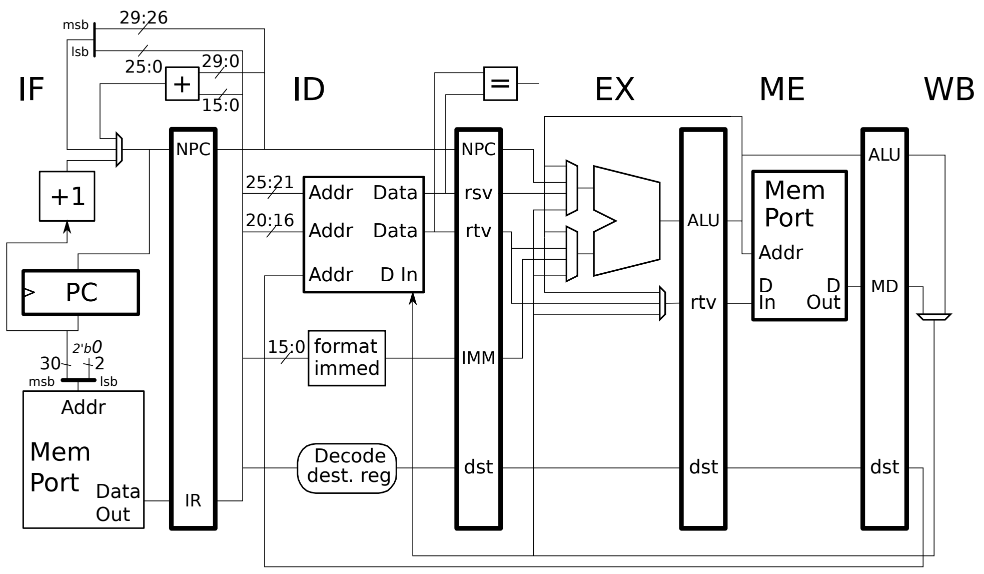
```
# SOLUTION
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
 sub.s f8, f10, f12     IF ID A1 A2 A3 A4 WF
 add r1, r2, r3            IF ID EX ME WB
 mul.s f14, f2, f8            IF ID ----> M1 M2 M3 M4 M5 M6 WF
 swc1 f14, 0(r1)                 IF ----> ID ------------> ME WB
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

The added bypass appears above in blue. A bypass path from `WF` to `M1` (and `A1` too) has been added for the `sub.s` to `mul.s` dependence. For the given code example only the lower multiplexor is needed since the dependence is through the second source operand of `mul.s` (through the `ft` field, set to `f8` in the example).

For the `mul.s` to `swc1` dependence a bypass path has been added from the end of `M6` to `EX`. A lower-performance option would be to bypass from `WF` to `ME`, this is lower performance under the assumption that the memory ports are always on the critical path and so inserting a multiplexor between `ME.rtv` and the memory port `Data In` connection would lower the clock frequency.

*Grading Note: Many solutions had the* `add` *instruction stall until after the* `sub.d` *wrote back.*

Problem 3: (10 pts) The code fragments below execute on several different MIPS implementations. In all cases the loop iterates many times. A five-stage scalar system is shown for reference.



(*a*) Show the execution of the code below on our familiar scalar pipeline, above. Show enough iterations to compute the CPI, and compute the CPI.

[✓] Execution for enough iterations to determine CPI.

See solution below. The first iteration starts in cycle 1 (by definition, when the first instruction of the loop body is in `IF`), the second iteration starts in cycle 8, and the third in cycle 14. We know we have enough iterations when the state of the pipeline is identical at the start of two consecutive iterations; we can use one of those iterations to compute CPI. The second and third iterations start identically: with `lw` in `IF`, `add` in `ID`, and `bne` in `EX`. So we know that the third iteration (for which only the `lw` is shown) will be identical to the second and so there is no need to show any more.

[✓] Find the CPI.

[✓] Doublecheck for dependencies.

[✓] Note that the first instruction in not part of the loop body.

The second iteration is $14 - 8 = 6$ cycles, and contains 4 instructions, and so the CPI is $6/4 = 1.5$.

```
 # SOLUTION
 lw r2, 0(r10) IF ID EX ME WB
LOOP:  #       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)     IF ID -> EX ME WB
 addi r2, r2, 4      IF -> ID EX ME WB
 bne r2, r4 LOOP           IF ID ----> EX ME WB
 add r5, r5, r1               IF ----> ID EX ME WB
LOOP:  #       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)                          IF ID EX ME WB
 addi r2, r2, 4                           IF ID EX ME WB
 bne r2, r4 LOOP                             IF ID ----> EX ME WB
 add r5, r5, r1                                 IF ----> ID EX ME WB
LOOP:  #       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)                                            IF ID EX ME WB
```

Problem 3, continued:

(*b*) Show the execution of the code below on a 4-way superscalar statically scheduled system without branch prediction. The superscalar system has five stages, aligned fetch, and can bypass between the same stages as can our scalar system. This means there are no bypass paths to the branch condition. Compute the CPI.

☑ Execution for enough iterations to determine CPI.

Solution appears below. None of the instructions stall in the first iteration, but in the second iteration the branch stalls due to a dependency with `addi` (remember that there are no bypass paths to the branch condition hardware).

Note that because fetch is aligned and because the first instruction of the loop body (`add`) has an aligned address (the address is a multiple of 16 [superscalar width of 4 times instruction size of 4 bytes]), all four instructions of the loop body are fetched at the same time.

☑ Find the CPI.

☑ The code below is *different* than the previous part.

☑ Doublecheck for dependencies.

☑ Note that first instruction not part of loop body.

The CPI is $(9-6)/4 = 0.75$ or $\frac{4}{3}$ IPC, less than half the 4 IPC potential of the hardware.

```
# SOLUTION
 addi r1, r0, 0     IF ID EX ME WB
 # Note: Address of insn below is 0x1000
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1        IF ID EX ME WB
 lw r1, 0(r2)          IF ID EX ME WB
 bne r2, r4 LOOP       IF ID EX ME WB
 addi r2, r2, 4        IF ID EX ME WB
                          IFx
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1              IF ID EX ME WB
 lw r1, 0(r2)                IF ID EX ME WB
 bne r2, r4 LOOP             IF ID -> EX ME WB
 addi r2, r2, 4              IF ID -> EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
                                IF ->x
 add r5, r5, r1                       IF ID EX ME WB
 lw r1, 0(r2)                         IF ID EX ME WB
 bne r2, r4 LOOP                      IF ID -> EX ME WB
 addi r2, r2, 4                       IF ID -> EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
                                         IF ->x
 add r5, r5, r1                                IF ID EX ME WB
```

(*c*) Show the execution of the code below on a four-way superscalar system with perfect branch prediction.

- ✓ Execution for enough iterations to determine CPI.

Because there is a branch predictor the branch does not need to be resolved in `ID`. Instead it will be resolved in `EX` where it can use the ALU to compute the condition, and where it can benefit from the existing ALU bypass paths. Therefore the branch instruction does not stall at all because the value of `r2` that the branch needs is bypassable starting when `addi` is in `ME` (such as in cycle 4 and 6). (Note that the stalls in cycle 4, 6, 8, etc are due to the dependence between the `lw` and the `add`.)

- ✓ Find the CPI.
- ✓ The code below is *different* than the first part.
- ✓ Doublecheck for dependencies.
- ✓ Note that first instruction not part of loop body.

The CPI is $(5-3)/4 = 0.5$ or $2\,\text{IPC}$, still half the full potential, due to the load-use dependence.

```
 # SOLUTION
 addi r1, r0, 0     IF ID EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1        IF ID EX ME WB
 lw r1, 0(r2)          IF ID EX ME WB
 bne r2, r4 LOOP       IF ID EX ME WB
 addi r2, r2, 4        IF ID EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1           IF ID -> EX ME WB
 lw r1, 0(r2)             IF ID -> EX ME WB
 bne r2, r4 LOOP          IF ID -> EX ME WB
 addi r2, r2, 4           IF ID -> EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1              IF -> ID -> EX ME WB
 lw r1, 0(r2)                IF -> ID -> EX ME WB
 bne r2, r4 LOOP             IF -> ID -> EX ME WB
 addi r2, r2, 4              IF -> ID -> EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1                    IF -> ID -> EX ME WB
 lw r1, 0(r2)                      IF -> ID -> EX ME WB
 bne r2, r4 LOOP                   IF -> ID -> EX ME WB
 addi r2, r2, 4                    IF -> ID -> EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1                          IF -> ID -> EX ME WB
```

Problem 4: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a $2^{10}$-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 12-outcome local history, and one system uses a global predictor with a 12-outcome global history.

```
Insn        Branch
Addr        Outcomes
0x1000: B1  r   r   r   r   r   r   r   r    r   r   r   r   r   r   r   r
0x1010: B2   T   N   N   N   N   T   N   N    T   N   N   N   N   T   N   N
0x1020: B3    R   R   R   R   R   R   R   R    R   R   R   R   R   R   R   R
0x2000: B4     T   T   T   T   T   T   T   T    T   T   T   T   T   T   T   T
```

Branch B1 is random, and can be described by a Bernoulli random variable with $p = .5$ (models a fair coin toss). The outcome of branch B3 is the same as the most recent execution of B1 (perhaps they are testing the same condition). For the questions below accuracy is after warmup.

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

In the diagram below the value of the bimodal predictor's 2-bit counter corresponding to B2 is shown before each execution of the branch. Counter values are shown only for one pass through the TNNNNTNN pattern. At the start of the second pass the counter has the same value, 0, as it had at the start of the first and so there is no need to continue (because we know the counter values will repeat). The bimodal predictor will predict not-taken when the counter value is 0 or 1, and so all predictions will be not-taken, resulting in $\boxed{\text{an accuracy of } 6/8}$.

*Solution Tips:* Be sure to base your answer on a repeating pattern. Obviously it would be wrong to base the accuracy only on the N outcomes (that would be 100% accuracy), but basing a solution on ten outcomes or twelve outcomes would also be wrong. Another thing to look out for is making sure the pattern repeats. Here we only needed to work out the counter values for one iteration to find a repeating pattern. If we used an initial counter value of 3 then we'd need to work counter values out for two passes through the TNNNNTNN pattern, and we'd need to base the prediction accuracy only on the second pass.

```
SOLUTION: Diagram to work out the values of the bimodal 2-bit counter.
BIMODAL COUNTER: 0   1   0   0   0   0   1   0   0
0x1010: B2         T   N   N   N   N   T   N   N    T   N   N   N   N   T   N   N
PRED RESULT:       x                   x
```

☑ What is the accuracy of the bimodal predictor on B3?

The 2-bit counter used to predict B3 is not affected by B1. Because the probability that B3 is taken is .5 $\boxed{\text{the accuracy is .5}}$.

☑ Considering BHT size, what is the approximate accuracy of the bimodal predictor on B4? ☑ Explain.

Since the predictor has $2^{10}$ entries it will be indexed by bits $11 : 2$ of the address of the branch being predicted. Notice that those bits are all zero for both branches B4 (its address is 0x1000) and B1 (its address is 0x2000), and so they will share an entry. Since both branches occur at the same frequency and B4 is always taken, there will be no way for the counter to be decremented. It will change between 3 and 4 and so B4 will be predicted with $\boxed{\text{100\% accuracy}}$. *Grading Note: The problem would have been more interesting if $p < 0.5$ or if* B1 *were executed more frequently than* B4.

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

The pattern length of B2, 8, easily fits in the local history of 12 and so the $\boxed{\text{accuracy is 100\%}}$.

☑ What is the minimum local history size needed to predict B2 with 100% accuracy?

$\boxed{\text{Six outcomes.}}$

Five is insufficient, for example, pattern NNTNN occurs twice, once followed by a N and once by a T.

☑ What is the accuracy of the global predictor B3? ☑ Explain

The 12-bit global history is long enough so that B3 can "see" B1's most recent outcome. The PHT entries will eventually warm up and predict B3 with $\boxed{\text{100\% accuracy}}$.

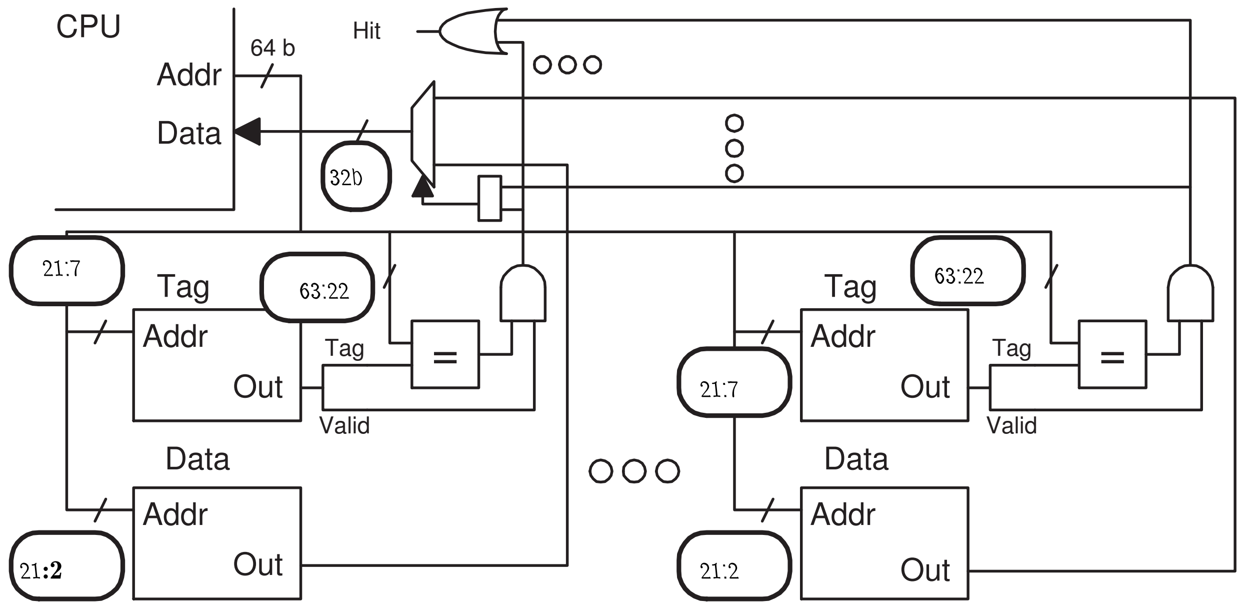
☑ How many PHT entries are used by the global predictor to predict B2?

To answer this question we need a way to represent the GHR contents. Lets use r, R, and T for branches B1, B3, and B4, respectively. For B2 use a 2. A GHR value pattern is then 2RTr2RTr2RTr. Each R can have two values, so for all three there are $2^3 = 8$ possibilities. The first two r's each match one of the R's and so don't contribute additional patterns. The last (rightmost or most recent) r does not correlate with any Rs in the pattern so there are two more possibilities for a total so far of $8 \times 2 = 16$. The three 2's are for branch B2. Those three outcomes can have four possible values, TNN, NNN, NNT, and NTN. (For example, three consecutive outcomes of B2 can never be TTT.) The total number of patterns now is $8 \times 2 \times 4 = 64$. Therefore, $\boxed{\text{64 entries}}$ are used to predict B2.

Problem 5: (15 pts) The diagram below is for an eight-way set-associative cache. The size of a tag is 42 bits and the size of a line is $128 = 2^7$ bytes.

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

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

| | Tag | Index | Offset | |
|---|---|---|---|---|
| Address: | | | | |
| | 63 … 22 | 21 … 7 | 6 … 2 | 1 … 0 |

☑ Cache Capacity (Indicate Unit!!):

The cache capacity can be determined from the following pieces of information: the 42-bit tag (given explicitly in the first paragraph), the 8-way associativity (also given explicitly), and the 64-bit address space (from the number of address bits shown in the diagram near the upper-left). From the 42-bit tag and 64-bit address space we know that the low tag bit is $64 - 42 = 22$, and so the capacity of a way is $2^{22}$ bytes or $4\,\mathrm{MiB}$. Since the cache is 8-way the $\boxed{\text{total capacity is } 2^{22} \times 8 = 2^{25} \text{ bytes or } 32\,\mathrm{MiB}}$.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, $2^{25}$ bytes, plus $8 \times 2^{22-6}\,(64 - 22 + 1)$ bits.

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

| | Tag | Index | Offset | |
|---|---|---|---|---|
| Address: | | | | |
| | 63 … 25 | 24 … 7 | 6 … 2 | 1 … 0 |

Problem 5, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB ($2^{25}$ byte) direct-mapped cache with a 128-byte line size. Each code fragment starts with the cache empty; consider only accesses to the array, `a`.

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

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

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

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

The line size of $2^7 = 128$ bytes is given, the size of an array element, of type half, is $2 = 2^1$ characters, and so there are $2^7/2 = 2^{7-1} = 2^6 = 64$ elements per line. The first access, at `i=0`, will miss but bring in a line with $2^6$ elements, and so the next $2^6 - 1$ accesses will be to data on the line. The access at `i=64` will miss and the process will repeat. Therefore the $\boxed{\text{hit ratio is } \frac{63}{64}}$.

(*c*) Find the minimum positive value of `OFFSET` needed so that the code below experiences a hit ratio of 0% on accesses to `a`. Explain.

☑ Minimum positive `OFFSET` to achieve 0% hit ratio. ☑ Explanation.

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


int OFFSET =   1 << 24;  // <---------------------------- SOLUTION


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

To achieve a zero percent hit ratio each line brought into the cache must be evicted before it is used again. The code above accesses `a` sequentially at two different places: at `i` and at `i+OFFSET`. Sequential accesses of two-byte elements should yield a hit ratio of $\frac{63}{64}$ on this cache, as computed in the previous part. To reduce that to zero one must choose `OFFSET` so that `a[i]` and `a[i+OFFSET]` fall in the same set, meaning that their addresses have the same index and different tags. Since it's a direct-mapped cache there cannot be two different lines with the same index in the cache at the same time. The index bits are at positions `24:7`. Let $x$ be some memory address. Address $y$ will have the same index as $x$ if $y = x + 2^{25}$ (note that adding 1 to the 25th bit position won't affect the index bit positions). For our particular problem we need to choose `OFFSET` such that the difference between `a[i]` and `a[i+OFFSET]` is $2^{25}$. Since the size of an element of `a` is two bytes, that means `OFFSET` must be $2^{25-1} = 2^{24}$ =`1<<24`.

Problem 6: (30 pts) Answer each question below.

(*a*) For the SPECcpu benchmark suite results can be reported using two tuning levels, *base* and *peak*. Consider a new level, *no-opt* in which the code is compiled without optimization. How valuable would no-opt tuning scores be to the people that care about SPECcpu scores? How different is no-opt tuning from base tuning?

✓ Value of no-opt tuning level?

No-opt tuning would not be of much value because it reflects the performance of a system running improperly prepared software. In other words, no-opt tuning would be useful to people who care about performance but for some silly reason don't compile with optimization turned on, or they get software from developers that don't turn optimization on.

*Grading Notes:* Some answered that the performance on unoptimized code might help predict the performance on optimized code, and for that reason no-opt is useful. That's wrong for two reasons. First, there are many reasons why system A can be faster than B on unoptimized code but B can be faster than A with optimization on. Second, peak and base both measure system performance on optimized code, so there is no need to have a third tuning level just to predict the first two.

✓ Difference between no-opt tuning and base tuning?

Programs prepared under the base tuning rules are optimized using normal effort by an experienced programmer. Such a program would at least use a no-brainer flag like `-O3` or `-fast`. That's far from no optimization.

(*b*) The `lw` below will experience a TLB miss exception when it first executes, remember that this exception is considered routine and is not due to any kind of error. The word in memory at location `0x12345678` is `0x222`. Show the execution of this code on our five-stage static pipeline until the `add` instruction reaches writeback, and show the value in register `r1` when the handler starts (see code).

✓ Show execution from `lui` to when `add` reaches `WB`.

✓ `FILL IN` the value that will be in `r1` when the handler starts.

```
# SOLUTION
# Cycles            0  1  2  3  4  5  6  7  8  9    1010 11 12 13 14 15 16
lui r1, 0x1234      IF ID EX ME WB

lw r1, 0x5678(r1)      IF ID EX ME*                      IF ID EX ME WB

add r2, r2, r1            IF ID EX*                         IF ID EX ME WB


HANDLER:
# Cycles            0  1  2  3  4  5  6  7  8  9    1010 11 12 13 14 15 16
# Value of r1 is           0x12340000   <- SOLUTION
sw ...                             IF ID EX ME WB
...
eret                                                  IF ID EX ME WB
```

Solution appears above. The value of `r1` reflects correct execution up to but not including `lw`. The handler ends by executing an `eret` (exception return) instruction which jumps to the `lw`, which executes completely this second time around.

(*c*) The instruction below is not a good candidate for a RISC ISA. Explain why in terms of hardware, not just in terms of a rule the instruction would violate. `add (r1), r2, (r3)`

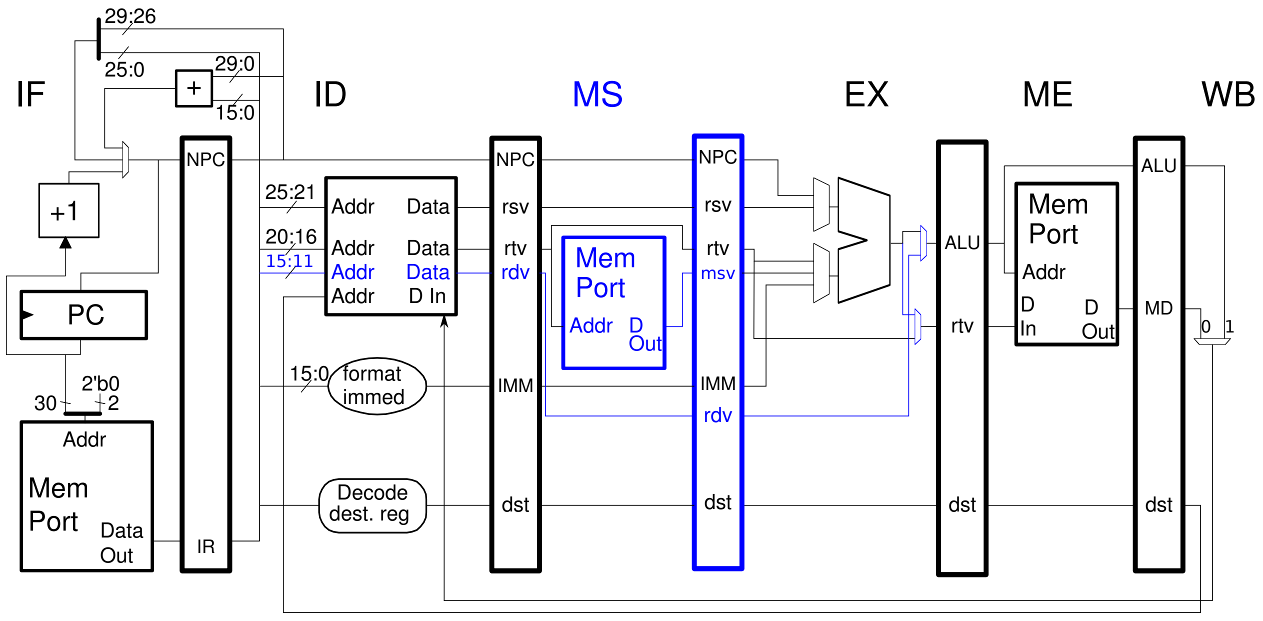
☑ Instruction above unsuitable for RISC because the implementation . . .

. . . would either need two memory ports (memory ports are expensive) or else would require this `add` instruction to use the ME stage twice, which would make the control logic far more complex. The former implementation would be too expensive, the later would go against the goal of simple pipelined implementations.

☑ Provide a quick sketch to illustrate your answer.

See the diagram below in which the added hardware is in blue and in which bypass paths have been omitted for clarity. Assuming that memory addresses for the source and destination operands are just register values (no offsets added), then a sixth memory source stage could be added between ID and EX to fetch a source operand that comes from memory, see stage MS the pipeline diagram below. The existing ME stage can write the result, though additional multiplexors were added so the `rd` register value can be used as a memory address and the ALU output can be used for the write data.

```
# Execution with new memory source stage added.
add (r1), r2, (r3)    IF ID Ms EX ME WB
```



(*d*) With the aid of a diagram, explain why a $5n$-stage pipeline would need fewer bypass paths than a 5-stage, $n$-way superscalar implementation. (Both implementations are statically scheduled.)

☑ Diagram showing why there are fewer bypass paths in $5n$-stage pipeline than 5-way superscalar.

Both systems would have $2n$ results that could be bypassed (from the original ME and WB stages). However the superscalar would have $n$ instructions in EX to which a result could be bypassed, while the deeply pipelined would have just 1. (There are two ALU inputs for each instruction.)

(*e*) Why is it more important to have a good compiler for a superscalar statically scheduled system than a scalar statically scheduled system?

✓ Compiler more important for superscalar because . . .

. . . dependent instructions must be further apart to avoid a stall and so the compiler must be better at scheduling (finding instructions to put between dependent pairs). In the scalar system there is no need to stall even for adjacent dependent ALU instructions (thanks to the bypass paths). In an $n$-way superscalar system there must be up to $n-1$ instructions between dependent ALU instructions to avoid a stall, and so the compiler must be good at finding instructions to put between such dependent pairs.

(*f*) Consider the executions of the MIPS code below on a 4-way superscalar dynamically scheduled system of the type discussed in class (method 3). The first execution is correct, the others, though they would run the program correctly, have problems. Describe the problems by completing the statements below.

```
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
add r3, r1, r4   IF ID Q        RR EX WB C
lh r1, 0(r6)     IF ID Q  RR EA ME WB    C
sub r7, r1, r3   IF ID Q           RR EX WB C
```

✓ The one-commit-per-cycle execution below is silly because . . .

. . . it will execute at a maximum rate of one instruction per cycle, and yet it has enough fetch, decode, and presumably execute hardware to sustain 4-instruction-per-cycle execution. The reasonable thing to do is to allow 4 IPC commit.

```
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
add r3, r1, r4   IF ID Q        RR EX WB C
lh r1, 0(r6)     IF ID Q  RR EA ME WB       C
sub r7, r1, r3   IF ID Q           RR EX WB    C
```

✓ The stalls shown below would be necessary on a statically scheduled pipeline because . . .

. . . there is a single pipeline and instructions must remain in program order within the pipeline. If one instruction must wait, so must the unlucky instructions ahead of it.

✓ . . . but should not occur on a dynamically scheduled one because . . .

. . . it is designed to execute instructions out of program order. The `IF/ID/Q` and `RR/EX/WB` are each separate pipelines, and so the `add` instruction waits for the `lw` in buffers, not in pipelines, allowing the `lh` to proceed without delay.

```
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
add r3, r1, r4   IF ID Q  ----> RR EX WB C
lh r1, 0(r6)     IF ID Q  ----> RR EA ME WB C
sub r7, r1, r3   IF ID Q  ----------> RR EX WB  C
```