Name Solution

Computer Architecture
EE 4720
Final Examination

1 May 2017, 10:00–12:00 CDT

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(15 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

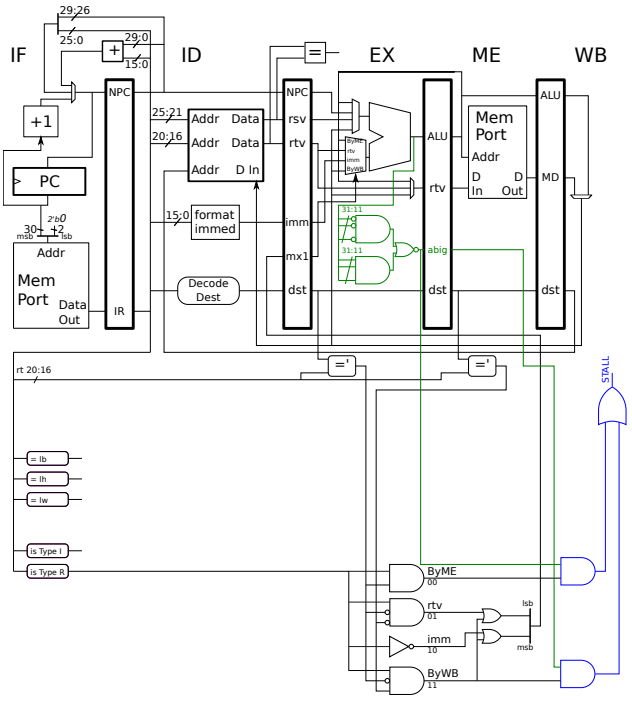
Alias My Alias Placeholder

Good Luck!

Problem 1: (20 pts) The diagram below, based on the solution to Homework 5, shows control logic that generates a stall signal when the value to be bypassed is too large for 12-bit bypass paths. The logic only works when the dependency is with the `rt` register of the consuming instruction and when the producing instruction is not a load. Modify the control logic so that it will generate a stall signal for a dependency to an `rs` register (first example below) and dependencies with loads. Pay attention to the load sizes.

```
# Dependency through rs register (r1 in the sub).
add r1, r2, r3
sub r4, r1, r5

# Producing instruction is a load word.
lw r1, 2(r3)
and r4, r5, r1

# Producing instruction is a load half.
lh r1, 2(r3)
and r4, r5, r1

# Producing instruction is a load byte.
lb r1, 2(r3)
and r4, r5, r1
```

Use next page for solution.



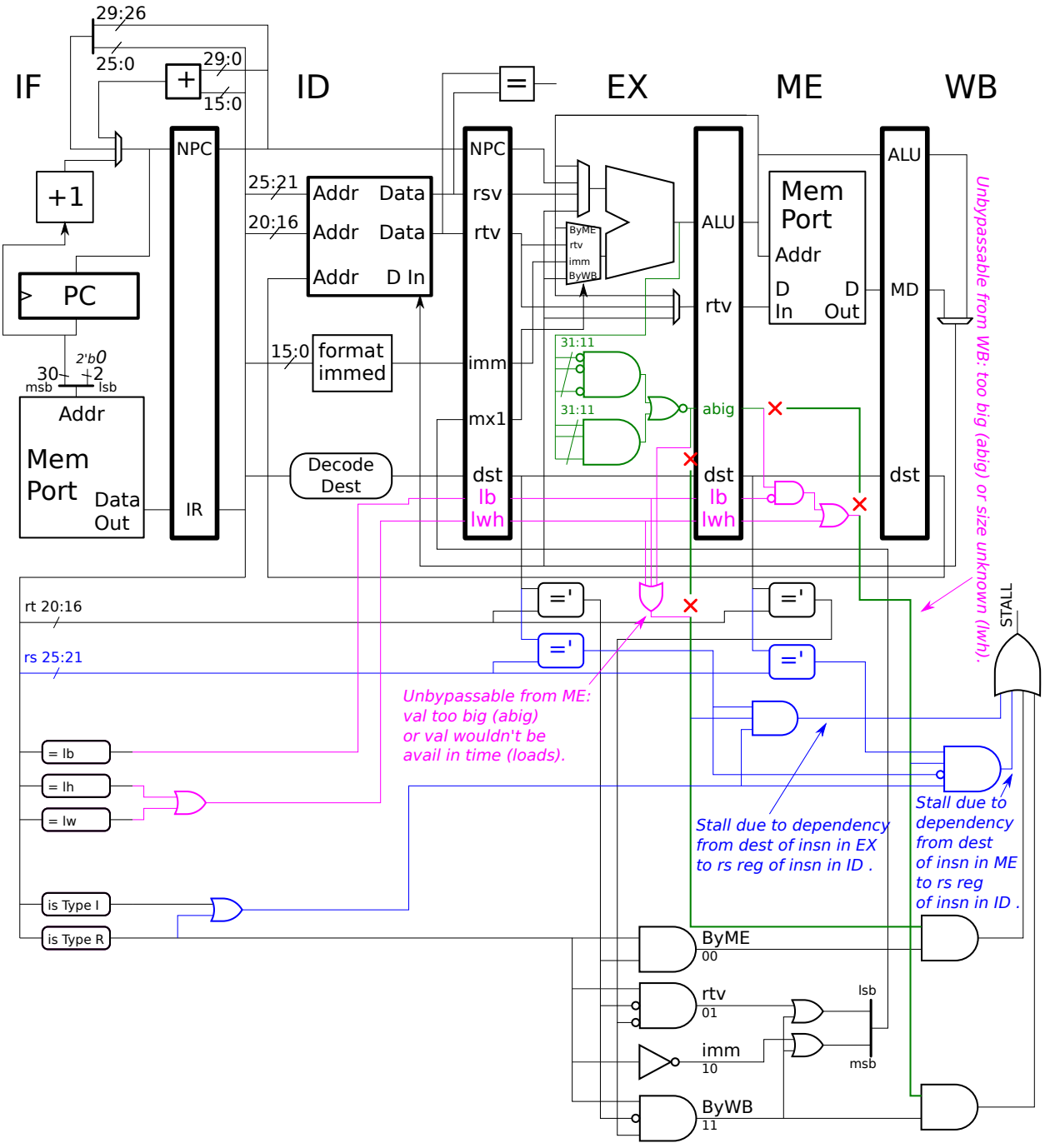
Use next page for solution.

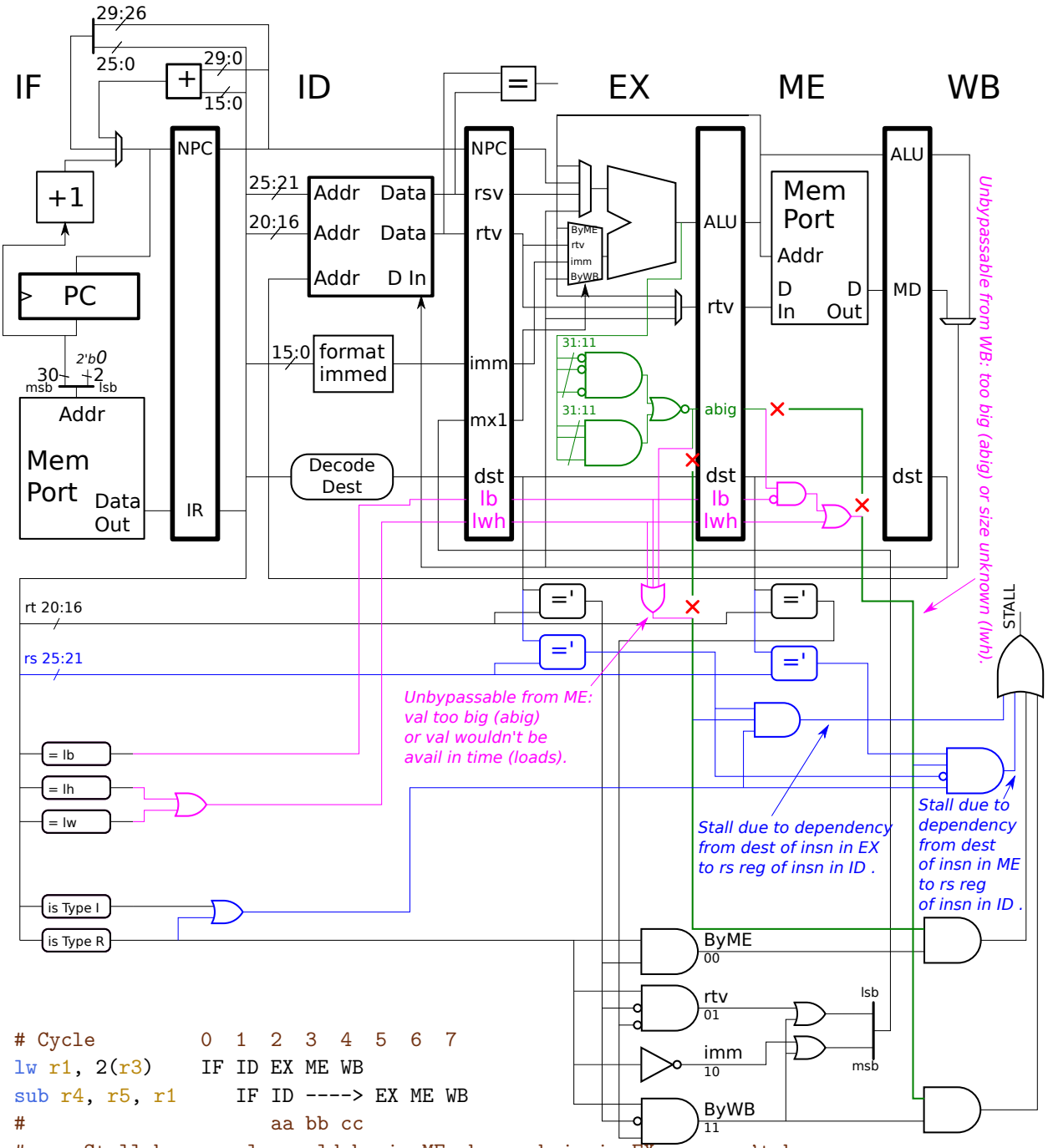
☑ Modify the control logic so that it also generates the stall signal for dependencies through the `rs` register that can't use 12-bit bypasses.

☑ Modify the stall control logic for when loads `lb`, `lh`, and `lw` produce the value to bypass, ☑ take into account whether value can use the 12-bit bypasses and ☑ whether the instructions are too close to bypass.

☑ Do not break existing control logic. As always ☑ consider cost and performance.

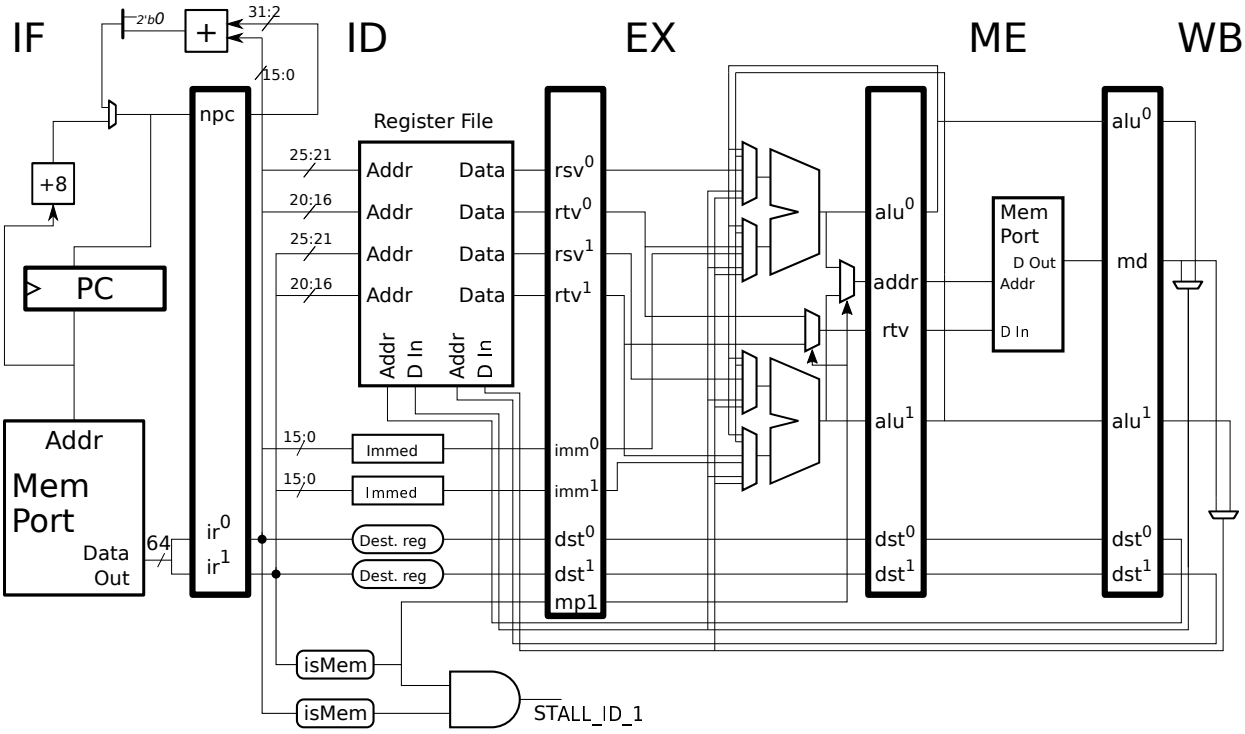
Solution appears below. The logic for stalling due to dependencies through the `rs` register appears in blue, and the logic for dependencies related to loads appears in purple.






```
# Cycle          0  1  2  3  4  5  6  7
lw r1, 2(r3)     IF ID EX ME WB
sub r4, r5, r1      IF ID ----> EX ME WB
#                      aa bb cc
# aa: Stall because lw would be in ME when sub is in EX, so can't bypass.
# bb: Stall because don't know if loaded value will fit in 12-bit bypass paths.
# cc: Don't stall, loaded value now is available from register file.

# Cycle          0  1  2  3  4  5  6  7
lb r1, 2(r3)     IF ID EX ME WB
sub r4, r5, r1      IF ID -> EX ME WB
#                      aa bb
# aa: Stall because lb would be in ME when sub is in EX, so can't bypass.
# bb: Don't stall, can bypass WB->EX and lb-loaded value can fit.
```

Problem 2: (15 pts) Illustrated below is a superscalar implementation taken from the solution to last year's final exam and the subject of this semester's Homework 7. Show the execution of the code sequences below on the illustrated superscalar MIPS implementation. Don't forget to check for dependencies.



(*a*) Show the execution of the code below on this implementation. Note that the address of the first instruction is `0x1000`.

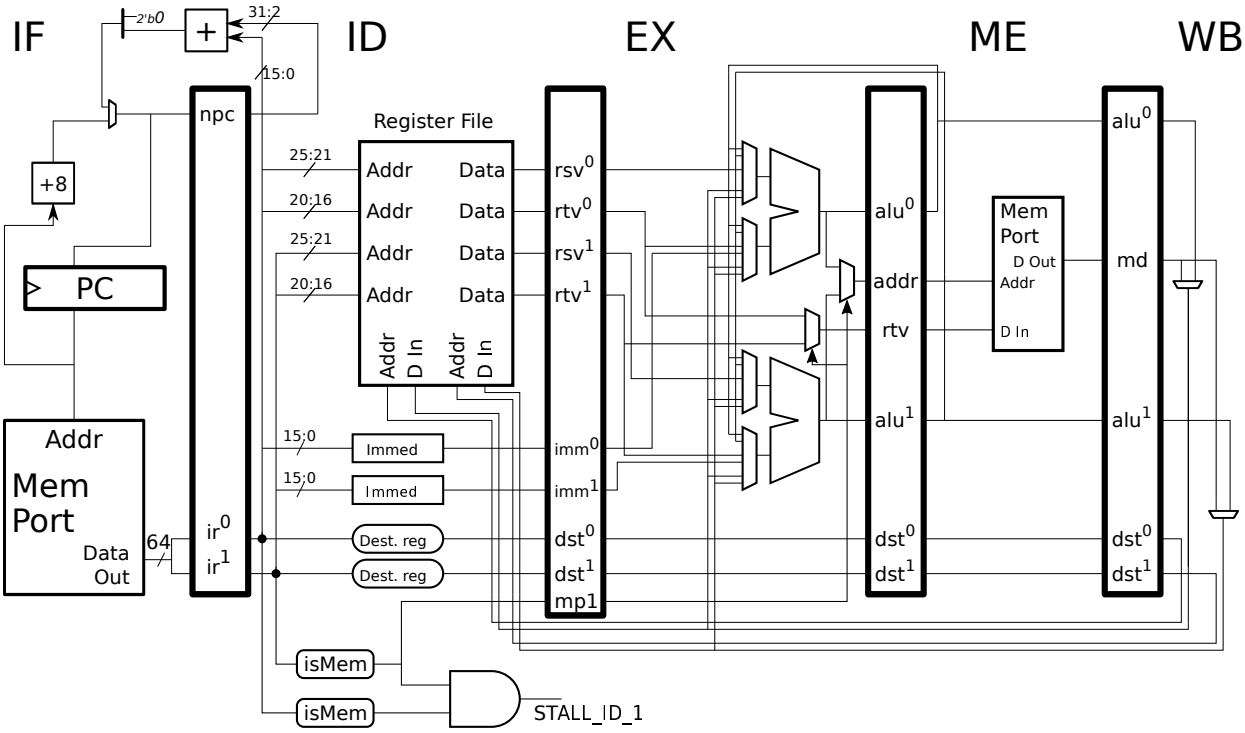
☑ Show execution of the following code sequence. ☑ Pay attention to ME in the diagram.

☑ Check for dependencies.

The solution appears below. The `lw r3` stalls because the ME stage can only accommodate one memory instruction. The last `add` stalls due to a dependence.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID EX ME WB
lw r3, 4(r2)     IF ID -> EX ME WB
lw r4, 8(r2)        IF -> ID EX ME WB
add r5, r1, r5      IF -> ID EX ME WB
add r5, r3, r5            IF ID EX ME WB
add r5, r4, r5            IF ID -> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8
```

Problem 2, continued: The illustration below is the same as the one on the previous page.



(*b*) Show the execution of the code below on the illustrated implementation when the branch is taken. Use the classroom default assumption: fetches are aligned.

☑ Show execution of the following code sequence. ☑ Check for dependencies.

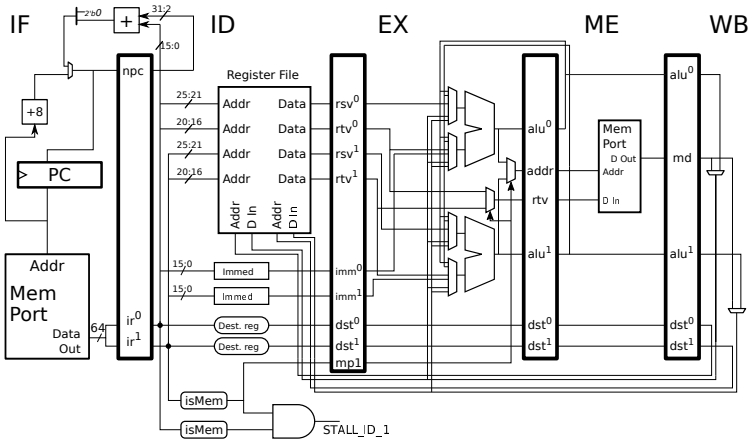
☑ Show all instructions that enter the pipeline, even those that are squashed in IF or later.

☑ Pay attention to instruction addresses, such as `0x1000`.

Solution appears below. Because the branch is resolved in **ID** the branch target is not fetched until the branch reaches **EX**, in cycle 2, and therefore two wrong-path instructions are fetched in cycle 1. Because the memory port in **IF** can only fetch aligned groups (meaning that the address of the instruction in slot 0 must be a multiple of 8) the **andi** instruction is fetched and then squashed as soon as it arrives. The **sb** stalls because of a dependency with **or**. Because there is no bypass into **EX/ME.rtv** the **sb** must stall in **ID** until **or** reaches **WB**. (The implementation used in Problem 1 has a bypass that would eliminate this stall.)

```
# SOLUTION
# Branch is taken.      Cycle  0  1  2  3  4  5  6  7  8  9
0x1000: bne  r1, r4  TARG      IF ID EX ME WB
0x1004: sub  r5, r2, r7        IF ID EX ME WB
0x1008: xor  r10, r11, r12        IFx
0x100c: lbu  r9, 0(r5)            IFx
0x1010: andi r8, r9, 12              IFx
# Cycle                        0  1  2  3  4  5  6  7  8  9
TARG:
0x1014: or   r11, r5, r12            IF ID EX ME WB
0x1018: sb   r11, 0(r5)                 IF ID ----> EX ME WB
# Cycle                        0  1  2  3  4  5  6  7  8  9
```

Problem 2, continued: The illustration below is the same as on the previous page.



(*c*) Appearing below is an execution of MIPS code on the illustrated superscalar implementation shown for the first two iterations. Compute the CPI for a large number of iterations. If necessary extend the execution diagram.

```
 lw r1, 0(r2)     IF ID EX ME WB
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4     IF ID -> EX ME WB                    # First Iteration
 lw r1, 0(r2)       IF ID -> EX ME WB
 bne  r2, r3  LOOP     IF -> ID EX ME WB
 addi r2, r2, 4        IF -> ID EX ME WB
 ???                         IFx                        # Fallthrough insn.
 ???                         IFx                        # Fallthrough insn.
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4                IF ID EX ME WB           # Second Iteration
 lw r1, 0(r2)                  IF ID EX ME WB
 bne  r2, r3  LOOP                IF ID EX ME WB
 addi r2, r2, 4                   IF ID EX ME WB
 ???                                 IFx                # Fallthrough insn.
 ???                                 IFx                # Fallthrough insn.
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4                           IF ID EX ME WB  # Third Iteration
#        Cycle   0  1  2  3  4  5  6  7  8  9  10 11 12
                 ! 1st Itr   ! 2nd Itr! 3rd Itr!
                 ! 4 cyc     ! 3 cyc  ! 3 cyc  !
```

☑ CPI for a large number of iterations.

To determine the number of cycles in an iteration we need to find a repeating pattern. The state of the pipeline at the start of the first iteration, in cycle 1, is clearly different than the state at the start of the second iteration, in cycle 5. Therefore in the solution above the start of a third iteration has been added. We can see that the third iteration starts in cycle 8. In both cycles 5 and 8 the pipeline contains: `add` in `IF0`, `addi` in `EX1`, `bne` in `EX0`, `lw` in `ME1`, and `add` in `ME0`. Since the states are identical we can expect the third iteration to take the same time as the second.

The duration of iteration i is the time from when the first instruction of iteration i enters `IF`, to the time when the first instruction of iteration $i+1$ enters `IF`. That time is highlighted above for the second iteration, which has a duration of $8-5=3$ cycles.

Therefore the CPI is $\boxed{\frac{8-5}{4} = \frac{3}{4} = 0.75\,\text{CPI}}$.

Problem 3: (20 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 8-outcome local history, and one system has a global predictor with a 8-outcome global history. Branch `B2` consists of a repeating pattern that starts with `TNTT` and is either followed by three not-taken outcomes, `nnn`, or four taken outcomes, `tttt`. (They are shown in lower case for clarity.) The `nnn` sequence occurs with probability .4, and is not correlated with anything.

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

```
                              ↓
B1:   T  N  T  T  N          T  N  T  T  N           T  N  T  T  N
B2:    T  N  T  T  n  n  n    T  N  T  T  t  t  t  t
```

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

The diagram below shows the counter values starting from an initial value of 0. A repeating pattern starts at the third group because the counter value is the same, 2, at the beginning and end of the group. So based on that five-outcome group the accuracy is $\boxed{\frac{3}{5}}$.

```
     0  1  0  1  2  1          2  1  2  3  2           3  2  3  3  2  <-- Counter
B1:   T  N  T  T  N          T  N  T  T  N           T  N  T  T  N
      x     x  x  x          x  x  x     x              x        x    <-- Pred. Outcome
```

☑ What is the accuracy of the bimodal predictor on branch B2? ☑ Account for the variable pattern length.

This is best analyzed by considering the four possible cases of the way the random sequence can occur before and after the fixed sequence (**TNTT**). These are shown in the table below. For each case the number of mispredictions is computed starting at the fixed sequence and continuing into the second random sequence. What makes this easy (relatively) is that when the fixed sequence starts the counter will be either 0 or 3. Therefore we can compute an exact prediction ratio for each of the four cases. These are shown under the **Pred** column. The **Prob** column is the probability that the fixed sequence will be surrounded with the particular random outcomes. The numbers under the weight column give something like the space taken up by the particular case. These are used to weight the prediction accuracies. In particular the value under **Weight** is the product of the value under **Pred** and the value under **Weight**. The sums are shown at the bottom. The prediction accuracy is the weighted value divided by the weight: $\boxed{\frac{4.92}{7.6} = .647368}$.

											Pred	Prob	Weight	Weighted
			0	1	0	1	2	1	0					
n	n	n	T	N	T	T	n	n	n					
			x		x	x	x				3/7	.4 * .4	.4 * .4 * 7	.4 * .4 * 3
			3	3	2	3	3	2	1					
t	t	t	t	T	N	T	T	n	n	n				
				x			x	x			3/7	.6 * .4	.6 * .4 * 7	.6 * .4 * 3
			0	1	0	1	2	3	3	3				
n	n	n	T	N	T	T	t	t	t	t				
			x		x	x					5/8	.4 * .6	.4 * .6 * 8	.4 * .6 * 5
			3	3	2	3	3	3	3	3				
t	t	t	t	T	N	T	T	t	t	t	t			
				x							7/8	.6 * .6	.6 * .6 * 8	.6 * .6 * 7
											---	--------	----------	----------
												1	7.6	4.92

☑ What is the accuracy of the local predictor on branch B2? ☑ Account for the variable pattern length.

Short Answer: Assuming that it always predicts **t** for the outcome after **TNTT**, the accuracy will be

$$\frac{7 \times .4 \times \frac{6}{7} + 8 \times .6 \times \frac{8}{8}}{7 \times .4 + 8 \times .6} = .947368$$

where the prediction accuracy for **TNTTnnn**, $\frac{6}{7}$, and for **TNTTtttt**, $\frac{8}{8}$, have been weighted by the probability that a B2 outcome is part of **TNTTnnn**, $\frac{7 \times .4}{7 \times .4 + 8 \times .6}$, or part of **TNTTtttt**, $\frac{8 \times .6}{7 \times .4 + 8 \times .6}$.

Long Answer: Because the local history length, 8, is long enough to identify the position within the pattern, the only outcome that can't be predicted with 100% accuracy is the first branch after the fixed sequence, **TNTT**. For example, consider **TNTTnnn**. It will correctly predict the fixed-sequence outcomes, **TNTT** and it will correctly predict the last two **n**s because once it sees the first of the three **n**s it will recognize that there will be two more. Or, to put it more precisely, when the local history contains **tttTNTTn** or **nnnTNTTn** the corresponding PHT entries will hold a zero because each time either of the two local histories was encountered in the past the B2 outcome would be **n** (that's the second **n**) and so the PHT entry would be decremented. By the same logic the third **n** would always be correctly predicted (after warmup) as would the second, third, and fourth **t**. When predicting the first outcome after the fixed sequence the local history will be either **TnnnTNTT** or **ttttTNTT**. We know that 60% of the time the outcome is **t**. As an approximation we can assume that the PHT entry would be 2 or 3 since 60% of the time it is incremented and 40% of the time it is decremented. It is possible to compute an exact probability distribution for the counter values by constructing a four-state Markov chain and solving the balanced flow equations $ap_i = (1-a)p_{i+1}$ for $0 \geq i \leq 2$, where a is the probability that the branch is taken, $a = .6$ here. Solving these yields $p_0 = \frac{\frac{a}{1-a} - 1}{\left(\frac{a}{1-a}\right)^4 - 1}$ and $p_i = \left(\frac{a}{1-a}\right)^i p_0$. From this we get $p_0 = .123077$ and

the probability of a taken prediction $p_2 + p_3 = .692308$ and a not taken prediction is $p_0 + p_1 = .307692$. We can use these numbers to compute an overall prediction accuracy

$$\frac{7 \times .4 \times \frac{6.307692}{7} + 8 \times .6 \times \frac{7.692308}{8}}{7 \times .4 + 8 \times .6} = .939271,$$

which is only slightly lower than the estimated accuracy.

☑ ☑ What is the minimum local history size for which branch `B1` and `B2` will not interfere with each other? Explain.

Seven outcomes. With seven outcomes the `B2` local history must contain either three consecutive `t`s or two consecutive `n`s, which never occur in a `B1` local history. This means that `B1` and `B2` will never use the same `PHT` entries and so won't interfere with each other with a seven-outcome local history. Now consider six outcomes. Local history `nTNTTn` could be for `B1` and `B2`, and so they would both use the same `PHT` entry. For `B1` the next outcome would be `T`, but for `B2` the next outcome would be `n`, and so the shared `PHT` entry could not predict both branches accurately. (Remember that there's no difference between `n` and `N` and no difference between `t` and `T`, so a local history of `nTNTTn` is exactly the same as `NTNTTN`. Upper and lower case are only being used to show which branch outcomes belong to the fixed part (upper case) and which belong to the repeating part (lower case).

☑ Note that an arrow (↓) points at an execution of `B1`. Show the value of the GHR at the time that that execution is being predicted.

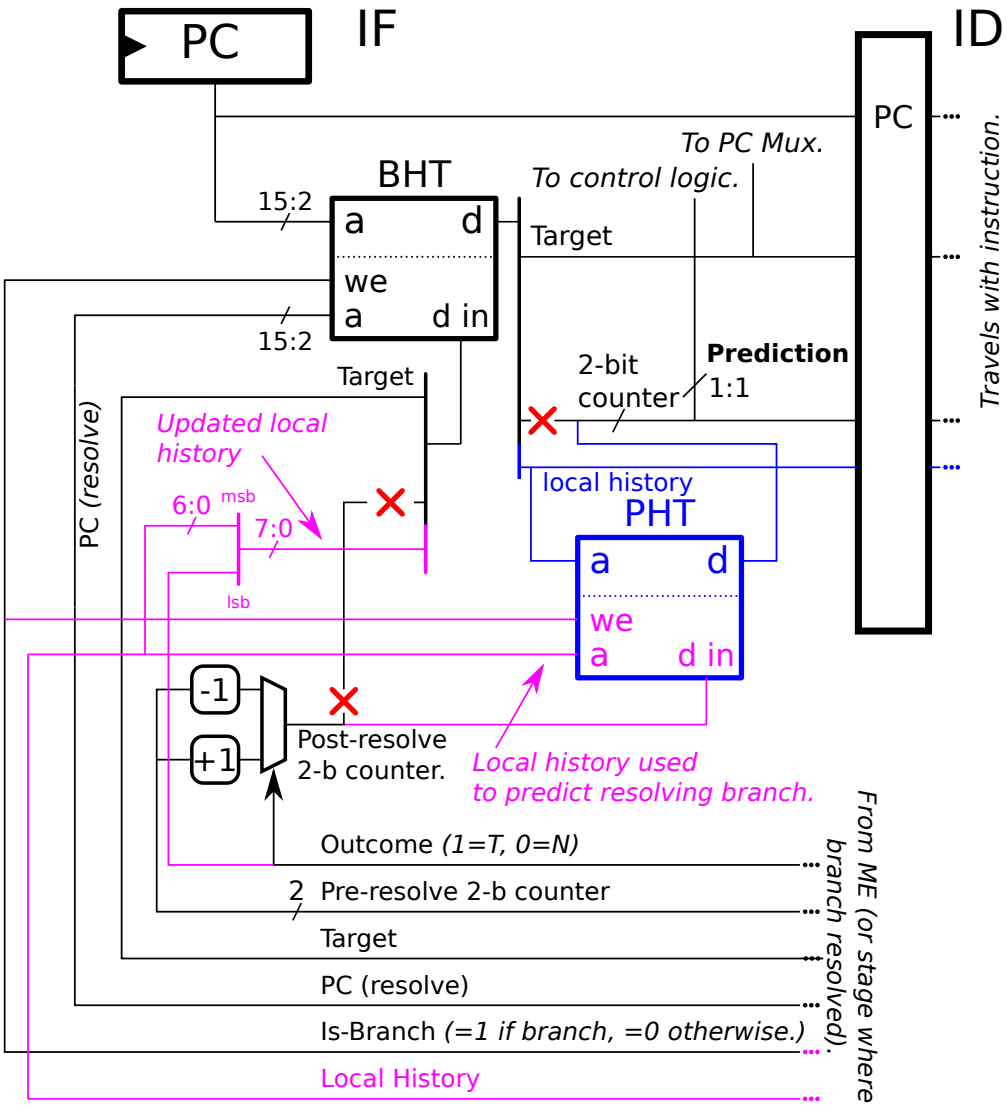
The local history will contain `TTTTNnnn`.

Problem 3, continued:

(*b*) Appearing below is a diagram of a bimodal predictor, showing in detail the logic for predicting the instruction in `IF` and for updating the predictor for the resolving branch. Modify the diagram so that it is a local predictor with an 8-outcome local history.

☑ Show the PHT, and connections for ☑ prediction and ☑ update.

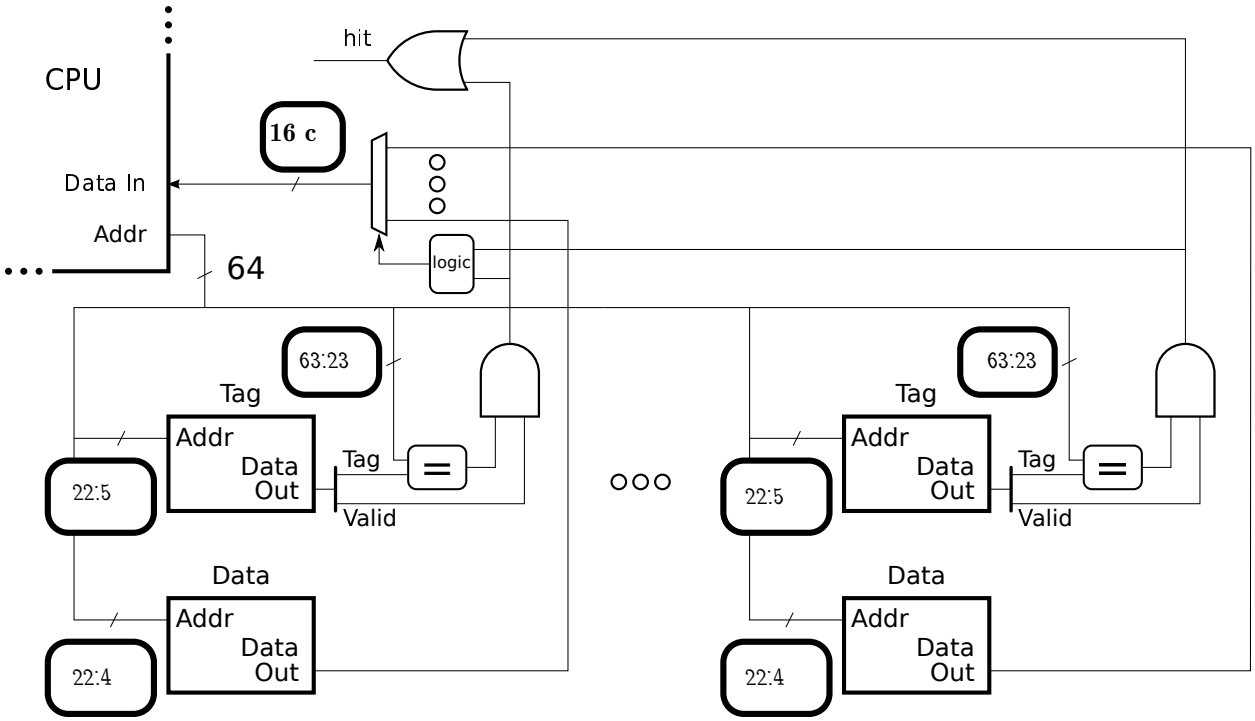
Solution appears below. The changes needed to predict the branch appear in blue. The BHT now stores local history, and that is connected to a newly added PHT. The prediction comes from the 2-bit counter in the PHT rather than the BHT. Update hardware appears in purple. Note that for update the original local history is used to index the PHT, but the updated local history is written into the BHT.



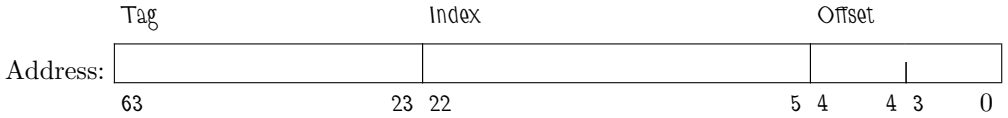
Problem 4: (15 pts) The diagram below is for a 32 MiB (2^{25} B) four-way set-associative cache with a line size of 32 B.

(*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.)

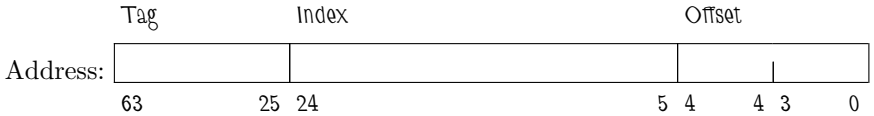


☑ Memory Needed to Implement ☑ Indicate Unit!!:

It's the cache capacity, 32 MiB, plus $4 \times 2^{23-5}\,(64 - 23 + 1)$ bits.

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

The cache above is 32 MiB and 4-way set associative. In a direct mapped cache there is just one way with four times the storage of a way in the cache above. To get four times the number of entries the number of index bits is increased by two, and so the index bits will start at position 24 instead of 22. The other bit positions remain the same.



Problem 4, continued: 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 1024 B (2^{10} B). Each code fragment starts with the cache empty; consider only accesses to the arrays.

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

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

for (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 $2^{10} = 1024$ bytes is given. The size of an array element, which is of type int, is $4 = 2^2$ B, and so there are $2^{10}/2^2 = 2^{10-2} = 2^8 = 256$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^8 elements, and so the next $2^8 - 1 = 255$ accesses will be to data on the line, hits. The access at `i=256` will miss and the process will repeat. Therefore the hit ratio is $\frac{255}{256}$.

Problem 5: (30 pts) Answer each question below.

(*a*) Consider a 4-way superscalar system and a scalar system with a 4-lane vector unit. Both can compute arithmetic at a rate of 4 operations per cycle. The vector system is cheaper but the superscalar system is more flexible.

☑ Why is the vector system less costly?

The vector system only needs to fetch and decode one instruction per cycle, unlike four for the superscalar system and so the superscalar system uses four times as much decode hardware. In a 4-way superscalar processor we expect there to be floating-point bypass paths from each of 4 slots in WF to the two functional unit (A1/M1) inputs for each slot. That's a total of $4 \times 4 \times 2 = 32$ bypass multiplexor inputs. But in a system with a vector unit we don't expect cross-lane bypasses. That is, there's no bypass from the lane-2 value of a result in WF to, say, the lane-1 input, though we do expect a bypass from the lane-2 value in WF to the lane-2 unit input. Each lane requires 2 multiplexor inputs for bypass (one input to each mux), therefore the number of bypass paths is just $2 \times 4 = 8$.

☑ Show something the superscalar system can do that the vector system cannot.

☑ Explain why vector system can't execute equivalent vector code as efficiently.

Vector instructions must apply the same operation to each lane of its operands. A 4-way superscalar system could execute four different operations, for example, the set of operations below.

```
# SOLUTION
 add.d f0, f2, f4
 sub.d f6, f8, f10
 mul.d f12, f14, f16
 add.s f18, f19, f20
```

(*b*) Unlike MIPS, ARM A64 has pre-index and post-index load and store instructions. Show two code examples, one in A64 that uses a post-index load, and one in MIPS that does the same thing (but without a post-index load). The exact syntax of the ARM instructions is not important, use comments to clarify instructions.

☑ ARM code and ☑ equivalent MIPS code.

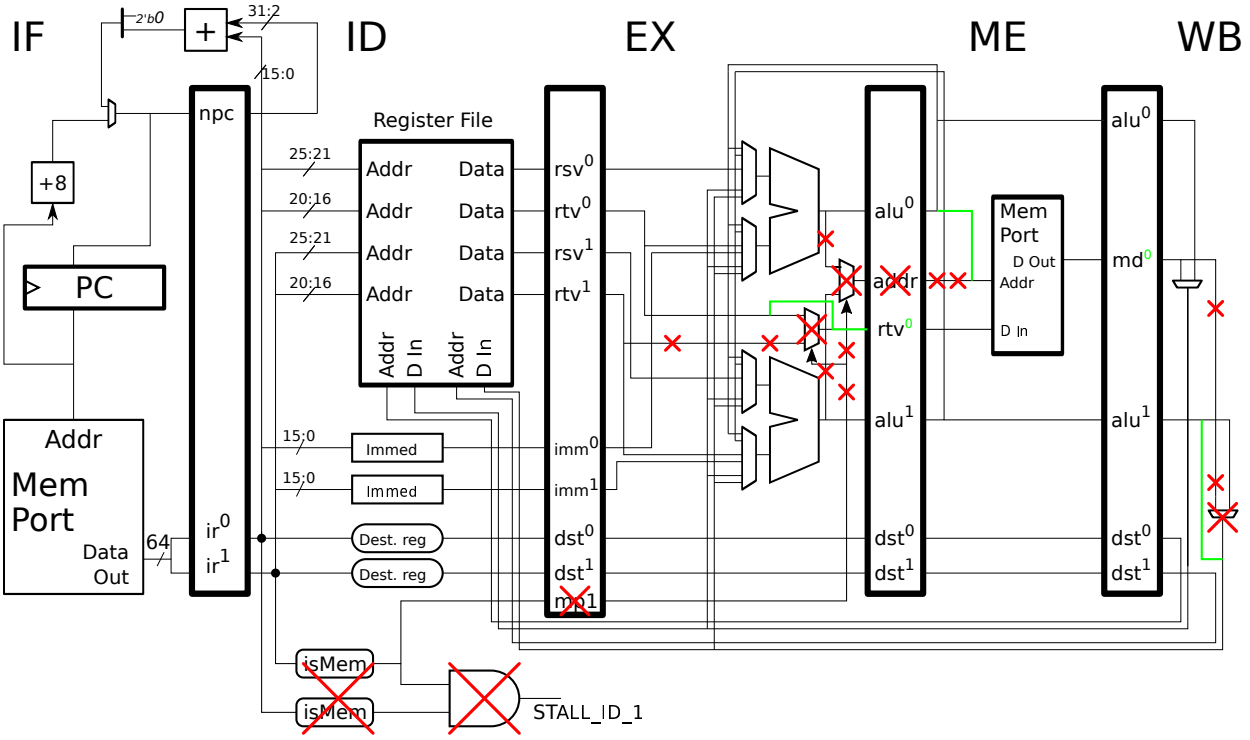
```
@ SOLUTION

@ ARM A64
 ldr x1, [x2], #8     @  x1 = Mem[x2];  x2 = x2 + 8

# MIPS
 lw r1, 0(r2)         #  r1 = Mem[r2]
 addi r2, r2, 4       #  r2 = r2 + 4
```

(*c*) What substantial additional hardware is needed to implement ARM A64 pre- and post-index loads when starting with something like our five-stage MIPS implementation. (Think about Homework 4.) *Note: The words "substantial" and "costly" were not included in the original exam.*

☑ Costly additional hardware for pre- and post-index loads.

Both the pre- and post-index loads need a second write port to the register file. The post-index load might require an additional pipeline latch so that both the unincremented value and incremented value can be passed from EX to ME, or a separate adder in the ME stage to do the post-increment, or some other costly addition.

(*d*) VLIW ISAs are supposed to do for superscalar implementations what RISC ISAs did for pipelined implementations. The diagram below shows our 2-way superscalar MIPS. Show how a 2-slot-bundle VLIW ISA (perhaps one a lot like MIPS) could simplify hardware in this implementation related to the sharing in `ME`.



☑ Modify the hardware above.

☑ Explain the bundle slot restrictions based on modified hardware.

Solution appears above. Unnecessary hardware is crossed out with red exes and where necessary wires reconnected in green.

A 2-slot bundle VLIW ISA designed for implementations like the one above in which there was one `ME`-stage memory port would require that a memory (load or store) instruction be placed only one slot (slot 0 in the solution above). That is, a memory instruction in slot 1 would be invalid and raise an illegal instruction exception. With that restriction the memory port `D In` input only needs a connection to slot 0, and so the `EX`-stage mux providing a path from slot 1 can be removed. Similarly, other connections to or from the memory port to slot 1 have been removed.

☑ Explain why the control logic driving `STALL_ID_1` would no longer be needed.

The bundle slot restrictions forbid two memory instructions in a bundle, and so there's no need to check for it. (In fact, there can't be a memory instruction in slot 1 even without a memory instruction in slot 0.)

(*e*) When an exception occurs (or a trap instruction is executed) the processor switches from user mode into privileged mode (also called system mode). Explain how privileged mode affects instruction execution, including loads, compared to user mode.

☑ Effect of privileged mode on instruction execution including ☑ effect on load instruction execution.

In privileged mode *all* instructions can be executed, but in user mode only a subset of instructions can be executed. Similarly, in privileged mode a load or store can access any valid memory address, in user mode loads and stores can only accesses addresses to which they have been granted access.

(*f*) It's hard to choose a line size that makes everyone happy. Explain how a long line size might slow down some programs in a small cache in comparison to the right line size (for those programs).

☑ With a small cache large lines can slow some programs because:

Short Answer: because the S bytes of data (say) that some program needs cached won't fit in a $4S$ byte cache because the program only uses $\frac{1}{8}$ of the data in a line. It would take an $8S$-byte cache to hold the S bytes in such a case.

For example, consider a $64\,\mathrm{kiB}$ cache with a $1024\,\mathrm{B}$ cache. Such a cache can hold 64 lines. Consider a program that frequently needs to access 100 bytes of data where the address of byte i is $2000i$, for $0 \leq i < 100$. Each line holds just one byte of the needed data plus 1023 unneeded bytes. Now consider a cache with 32-byte lines, but also of $64\,\mathrm{kiB}$. This cache can hold $\frac{64\times 2^{10}}{32} = 2048$ lines. That's more than enough for the program.

☑ Describe the characteristics of code that works well with long lines.

Code that accesses data sequentially. For example, `for (i=0; i<1000000; i++ ) sum += a[i];` . Here, the access to data is sequential.