Name _____

# 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) |

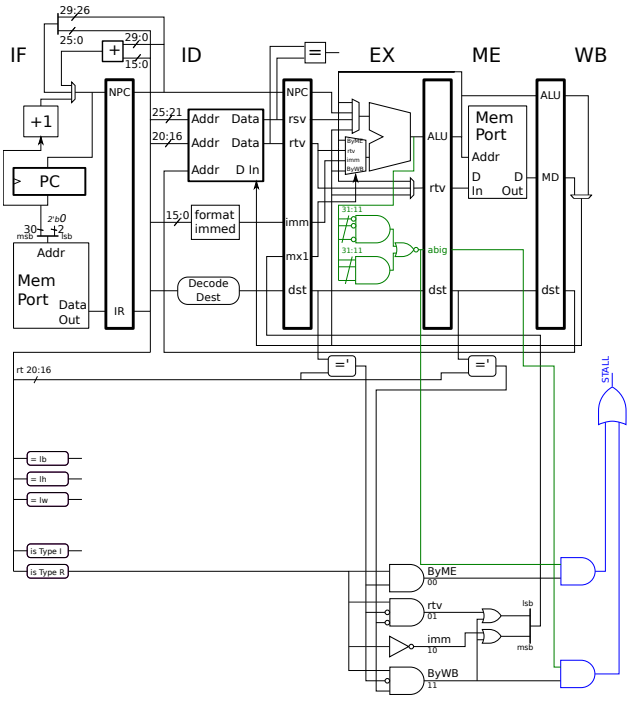
Alias _____

*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.*

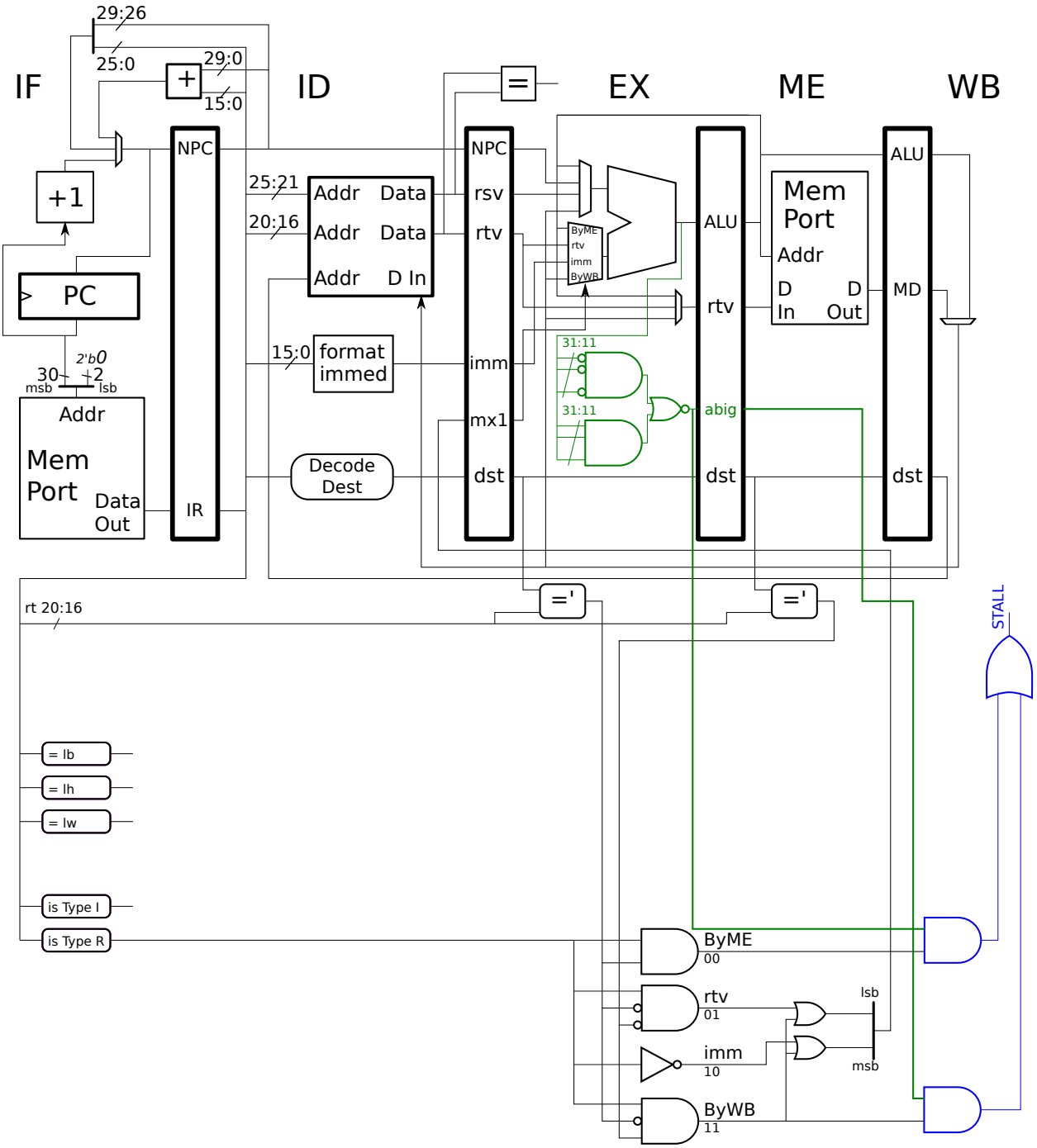



*Use next page for solution.*

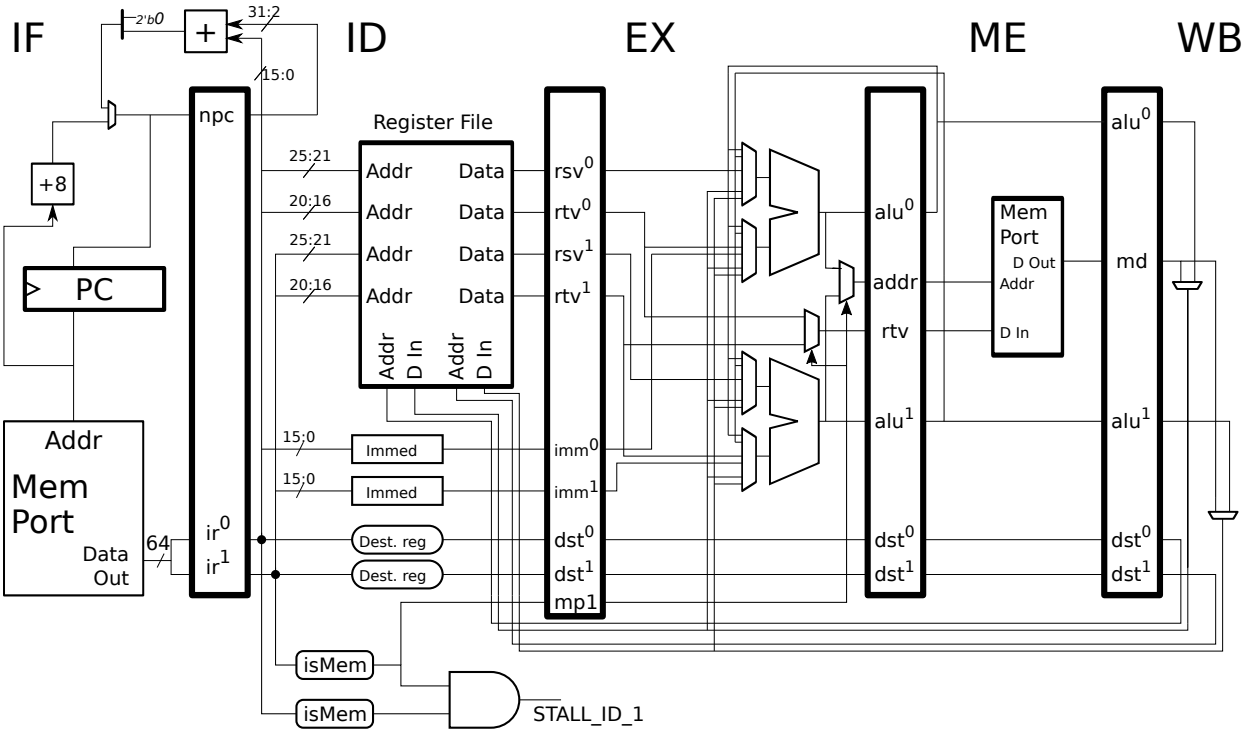
☐ Modify the control logic so that it also generates the stall signal for dependencies through the `rs` register that can't use 12-bit bypasses.

☐ 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.



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.



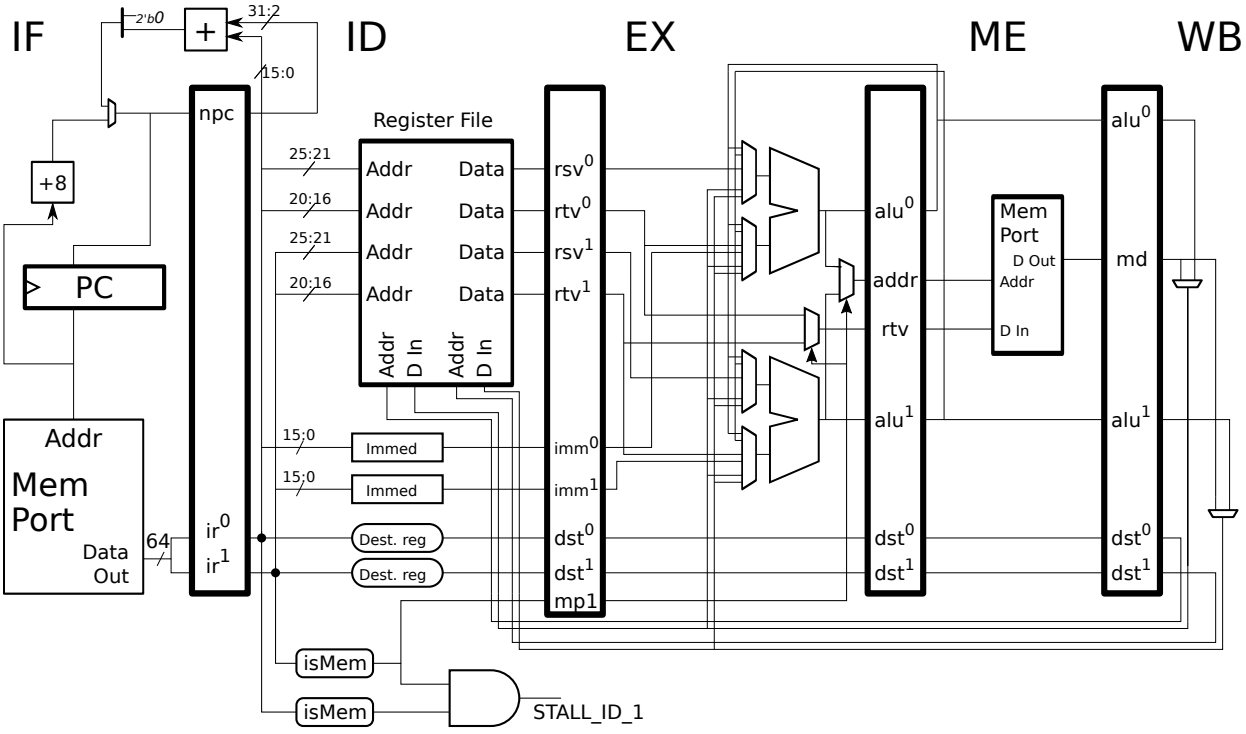

(*a*) Show the execution of the code below on this implementation. Note that the address of the first instruction is `0x1000`.

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

☐ Check for dependencies.

```
START: Address is 0x1000.

 lw r1, 0(r2)

 lw r3, 4(r2)

 lw r4, 8(r2)

 add r5, r1, r5

 add r5, r3, r5

 add r5, r4, r5
```

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.

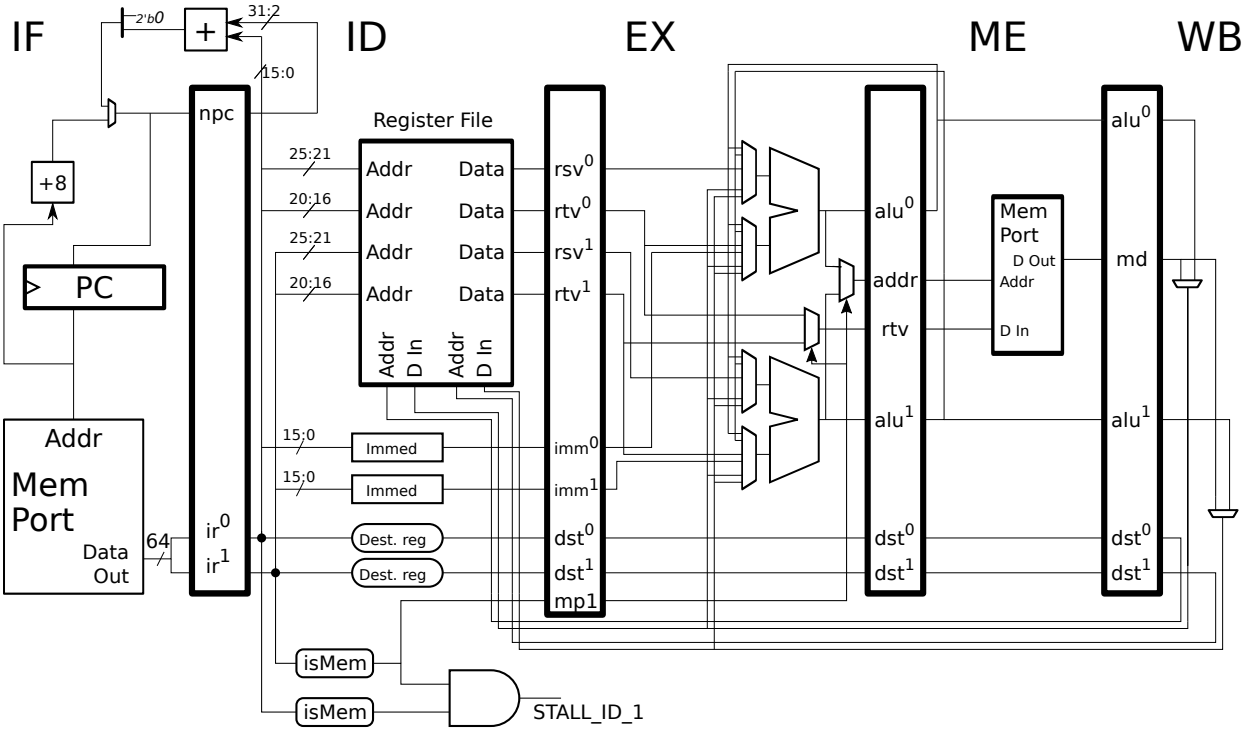
☐ Show execution of the following code sequence. ☐ Check for dependencies.

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

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

```
#       Branch is taken.
0x1000: bne r1, r4  TARG

0x1004: sub  r5, r2, r7

0x1008: xor  r10, r11, r12

0x100c: lbu  r9, 0(r5)

0x1010: andi r8, r9, 12

TARG:
0x1014: or  r11, r5, r12

0x1018: sb  r11, 0(r5)
```

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
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
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
```

☐ CPI for a large number of iterations.

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`?

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

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

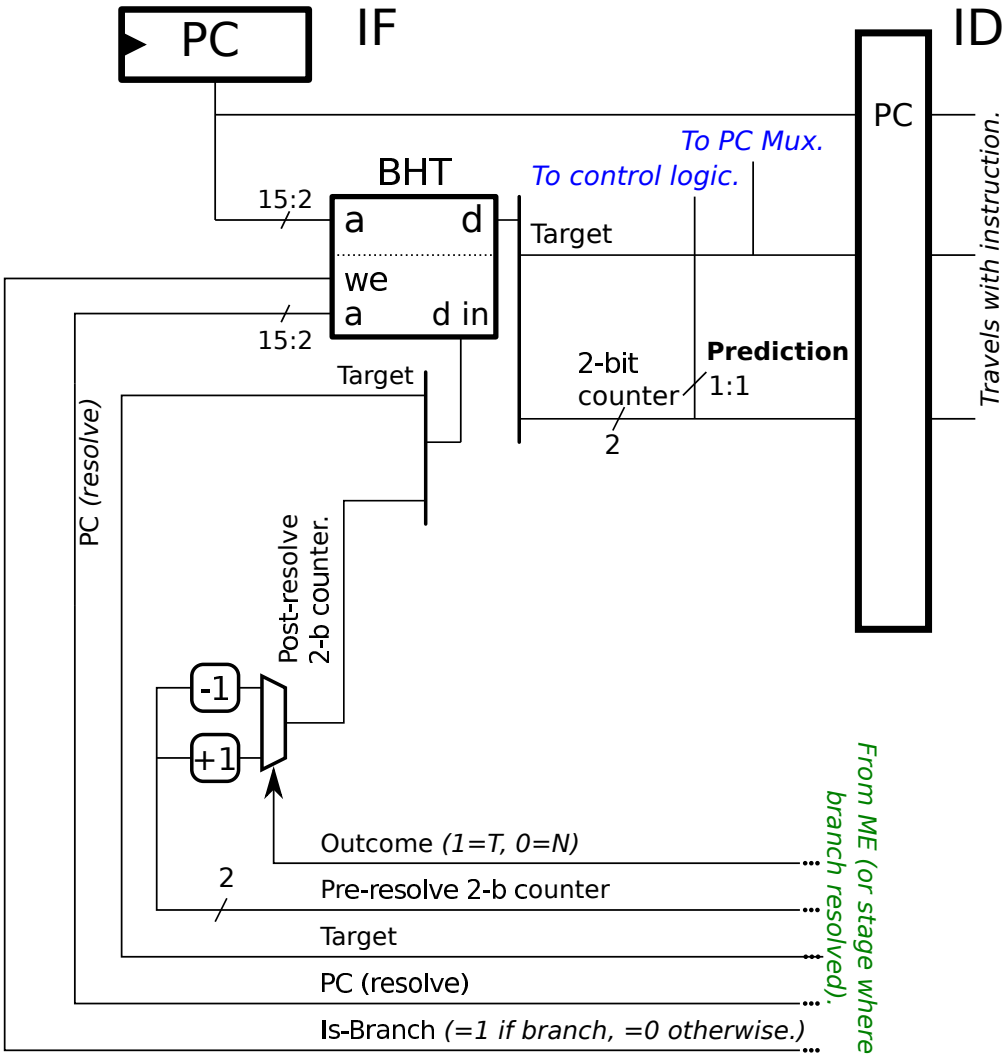
☐ What is the minimum local history size for which branch `B1` and `B2` will not interfere with each other? ☐ Explain.

☐ 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.

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.

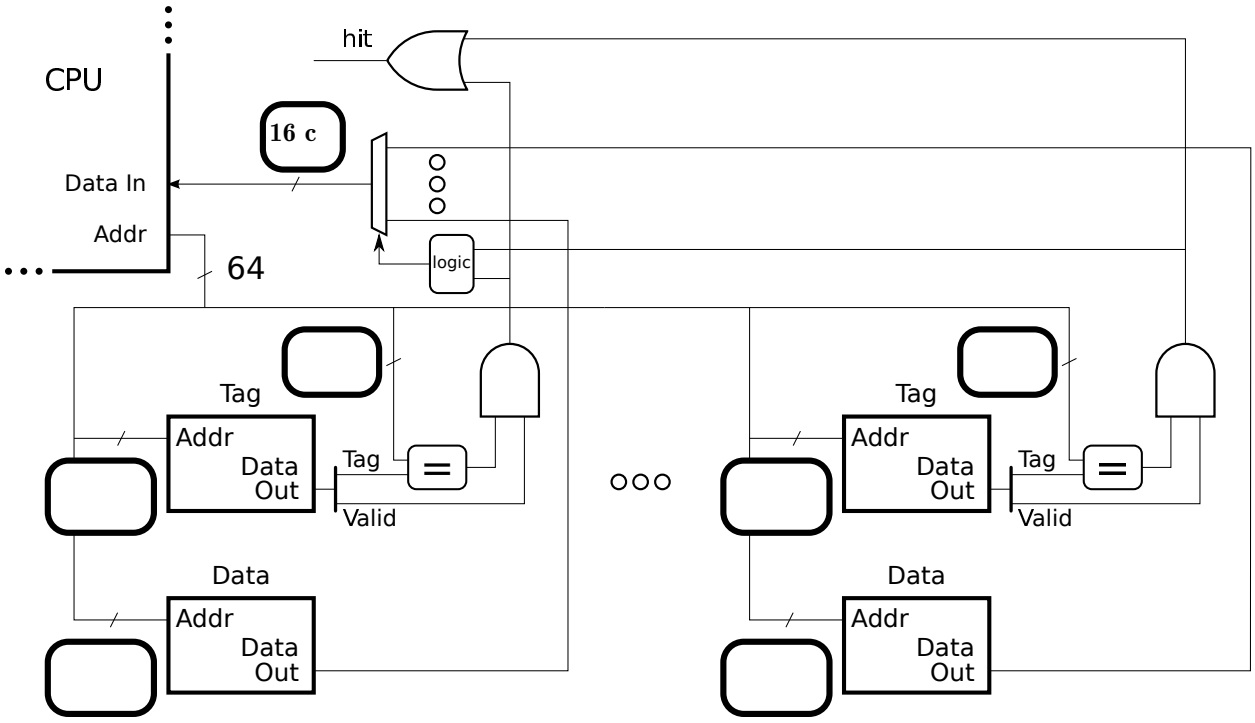
☐ Show the PHT, and connections for ☐ prediction and ☐ update.



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.)

Address: ☐ ☐ ☐

0

☐ Memory Needed to Implement ☐ Indicate Unit!!:

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

Address: ☐ ☐ ☐

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.

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?

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

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

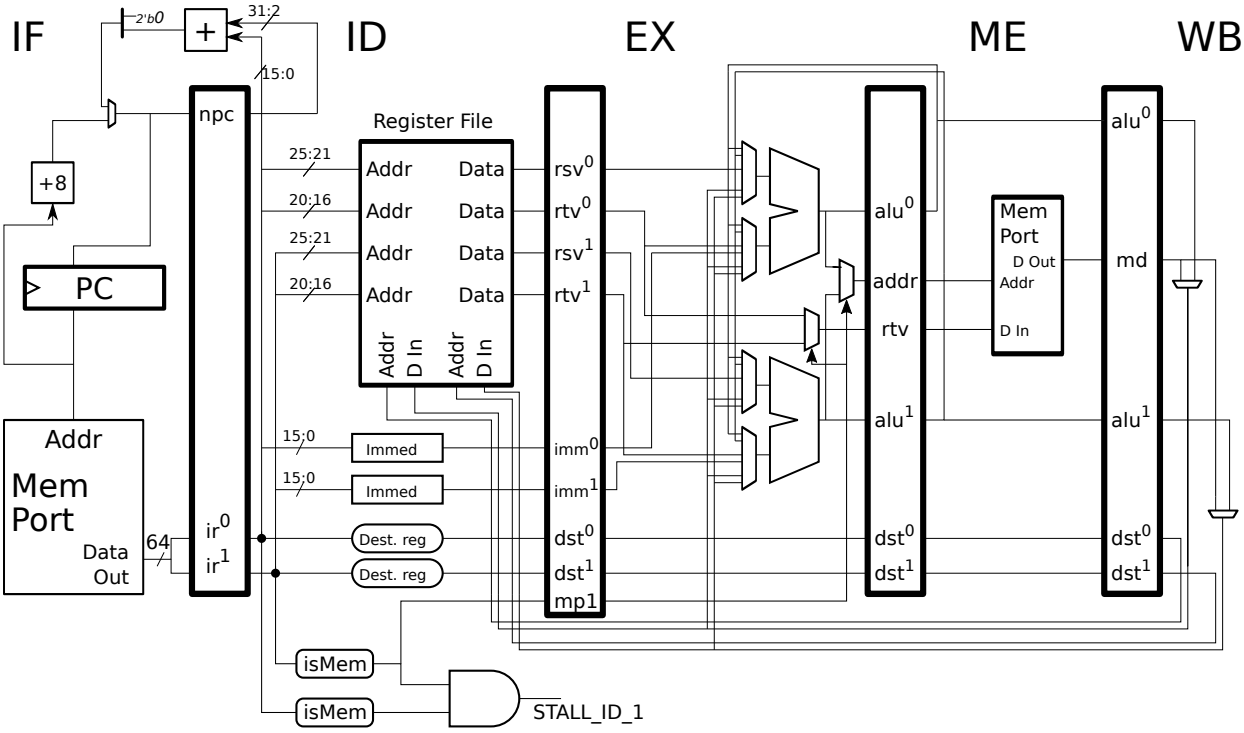
(*b*) Unlike MIPS, ARM A64 has pre-index and post-index load and store instructions. Show two code examples, one in A64 that uses a post-index load, and one in MIPS that does the same thing (but without a post-index load). The exact syntax of the ARM instructions is not important, use comments to clarify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

☐ ARM code and ☐ equivalent MIPS code.

(*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.

(*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.

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

(*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.

(*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:

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