Name ____________________

Computer Architecture

EE 4720

Final Examination

4 May 2016, 15:00–17:00 CDT

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

Alias ____________________

Good Luck!

Problem 1: (20 pts) Appearing below is our two-way superscalar MIPS implementation with a single, unconnected memory port in the `ME` stage. Since there is only one memory port there will have to be a slot-1 `ID`-stage stall whenever `ID` contains two memory instructions, for example:

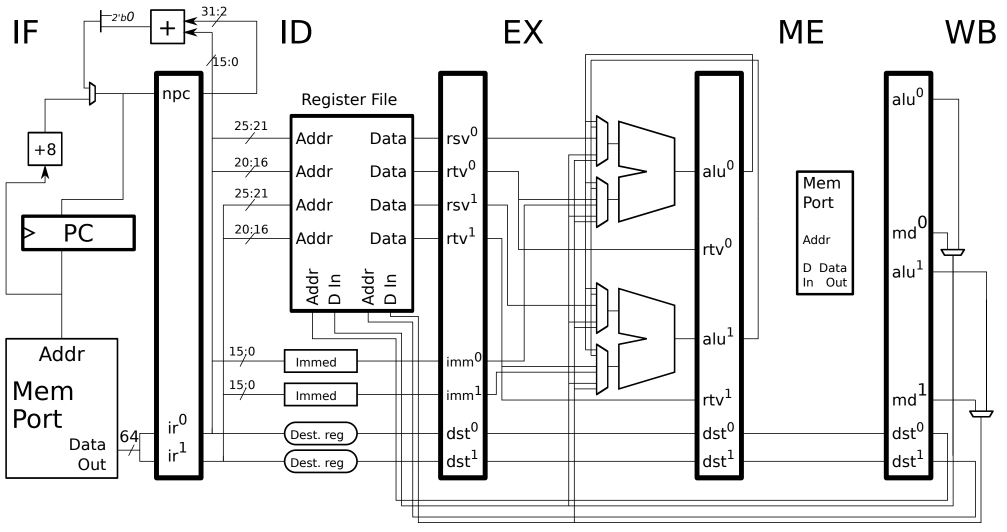
```
# Cycle       0  1  2  3  4  5
lw r1, 0(r2)  IF ID EX ME WB
lw r3, 0(r4)  IF ID -> EX ME WB
```

(*a*) On the next page add datapath to the implementation so that the memory port can be used by a load or store instruction in either slot. Pay attention to the cost of pipeline latches.

(*b*) On the next page add control logic needed for the datapath changes. The control logic should be for any multiplexors that you've added, don't include control logic for the memory port itself.

(*c*) On the next page add control logic to generate a `STALL_ID_1` signal when there are two memory instructions in `ID`, as occurs in cycle 1 to the `lw r3` in the example above.

Use next page for solution.



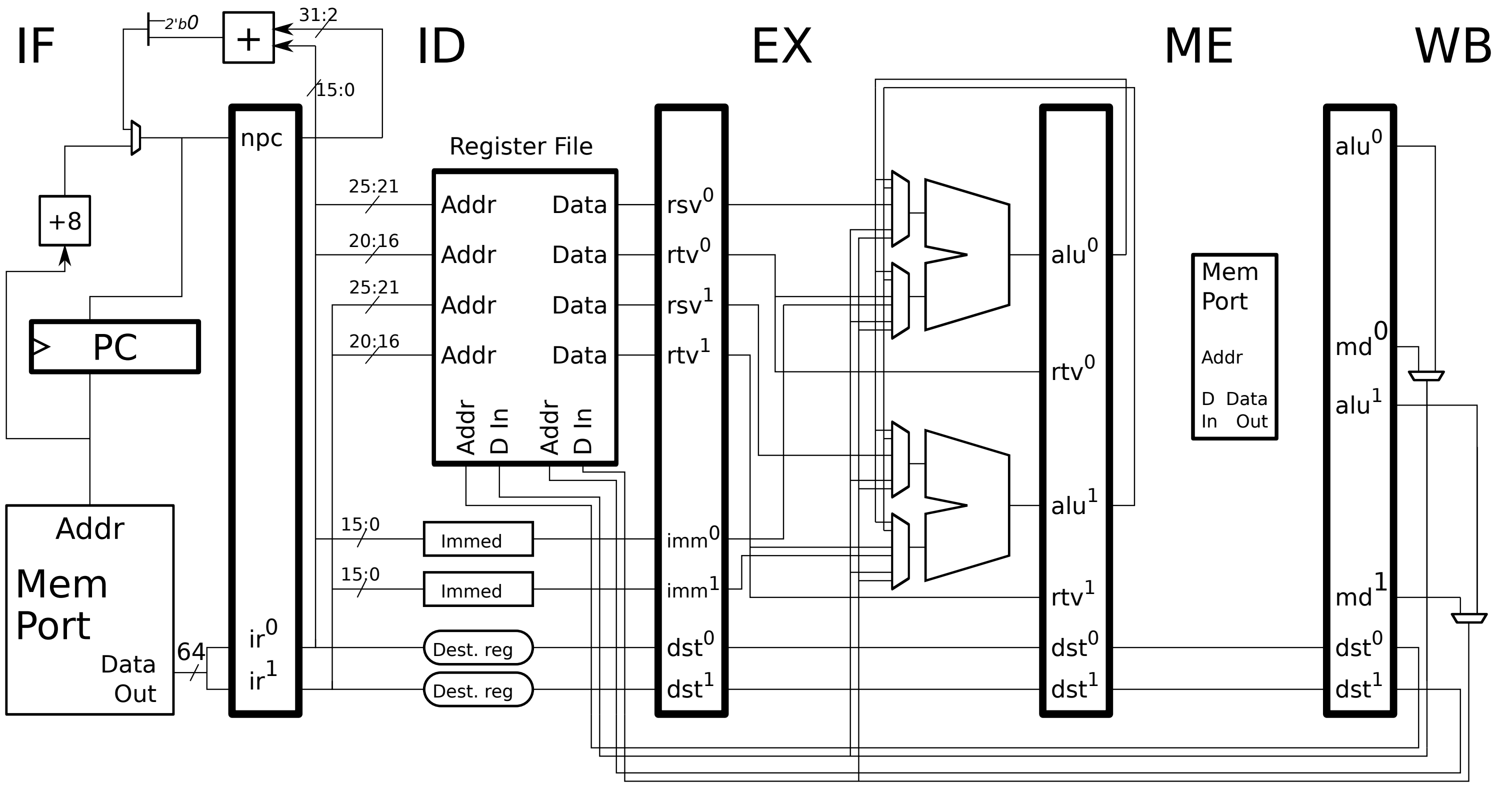
Use next page for solution.

Problem 1, continued:

☐ Datapath so that memory port can be used by a ☐ load or ☐ store in either slot.

☐ Control logic for multiplexors that you've added. ☐ Control logic to generate `STALL_ID_1`.

☐ Pay attention to ☐ **pipeline latch cost** and ☐ the critical path.



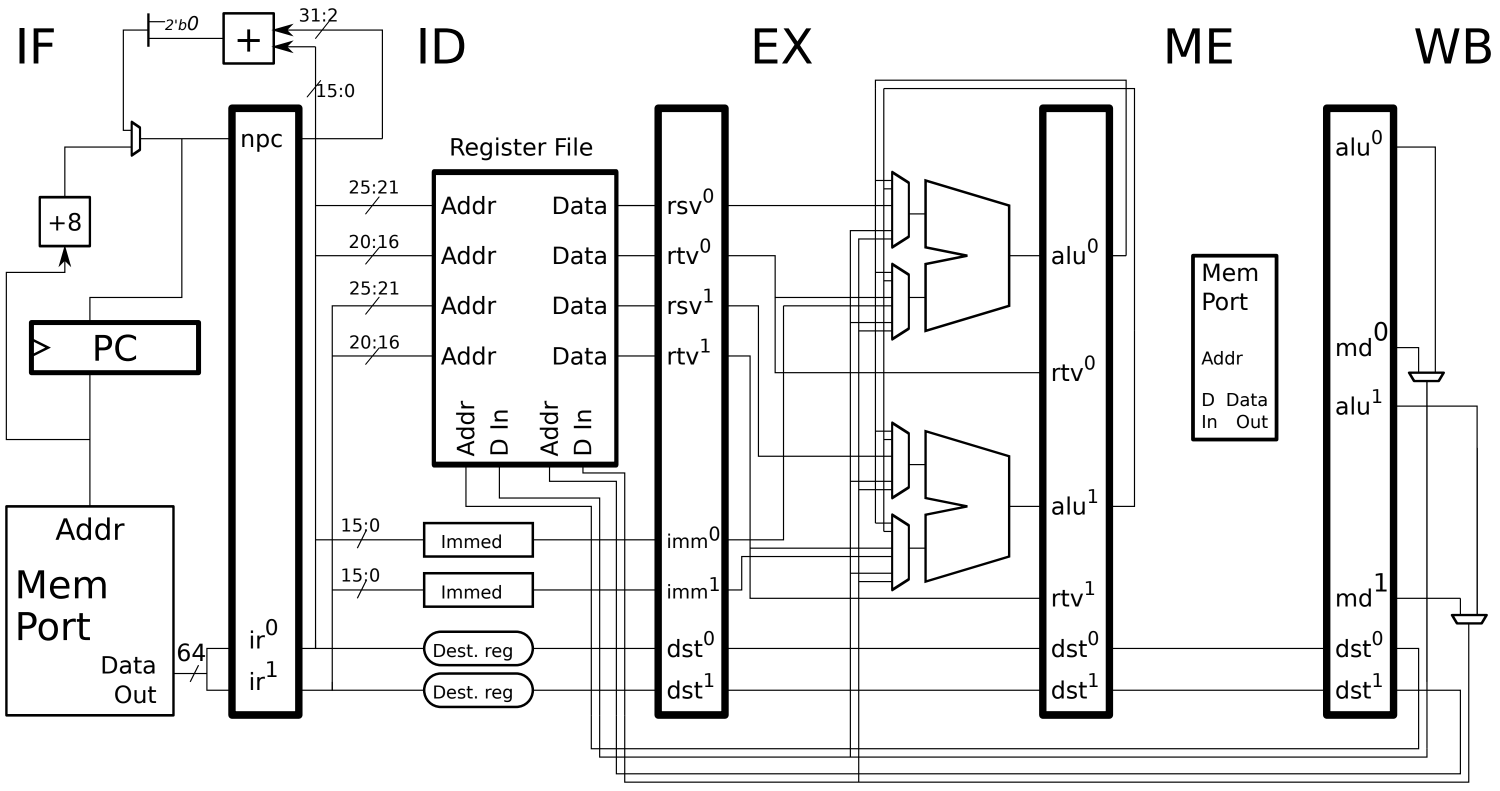
(*d*) Notice that the two instructions in the code below load from adjacent addresses. The superscalar implementation from the previous page would stall the `lbu r3`. But that might not be necessary because the memory port retrieves 32 b of data. A properly designed alignment network could provide data for both instructions, in some cases. Let's call such nice situations, *shared loads*.

```
lbu r1, 0(r2)
lbu r3, 1(r2)
```

Only show `ID`-stage changes. Show control logic to detect possible shared loads in `ID`. Generate a signal named `SHARED_LOAD` which is `1` if the instructions in `ID` could form a shared load, based on register numbers and immediate values.

☐ Control logic to detect possible shared loads.

☐ Explain why alignment and critical path make it impossible to know for sure in `ID` that a shared load is possible.



The following part was NOT on the final exam as given. It will be assigned as homework in 2017.

(*e*) Notice that the two instructions in the code below load from adjacent addresses. The superscalar implementation from the previous page would stall the `lbu r3`. But that might not be necessary because the memory port retrieves 32 b of data. A properly designed alignment network could provide data for both instructions, in some cases. Let's call such nice situations, *shared loads*.

```
lbu r1, 0(r2)
lbu r3, 1(r2)
```

Here's the plan: In `ID` detect whether a shared load is possible. In `EX` check addresses. In `ME` have memory port perform only one kind of load, 32 b (stores are unaffected). In `WB` have a separate alignment network for each slot.

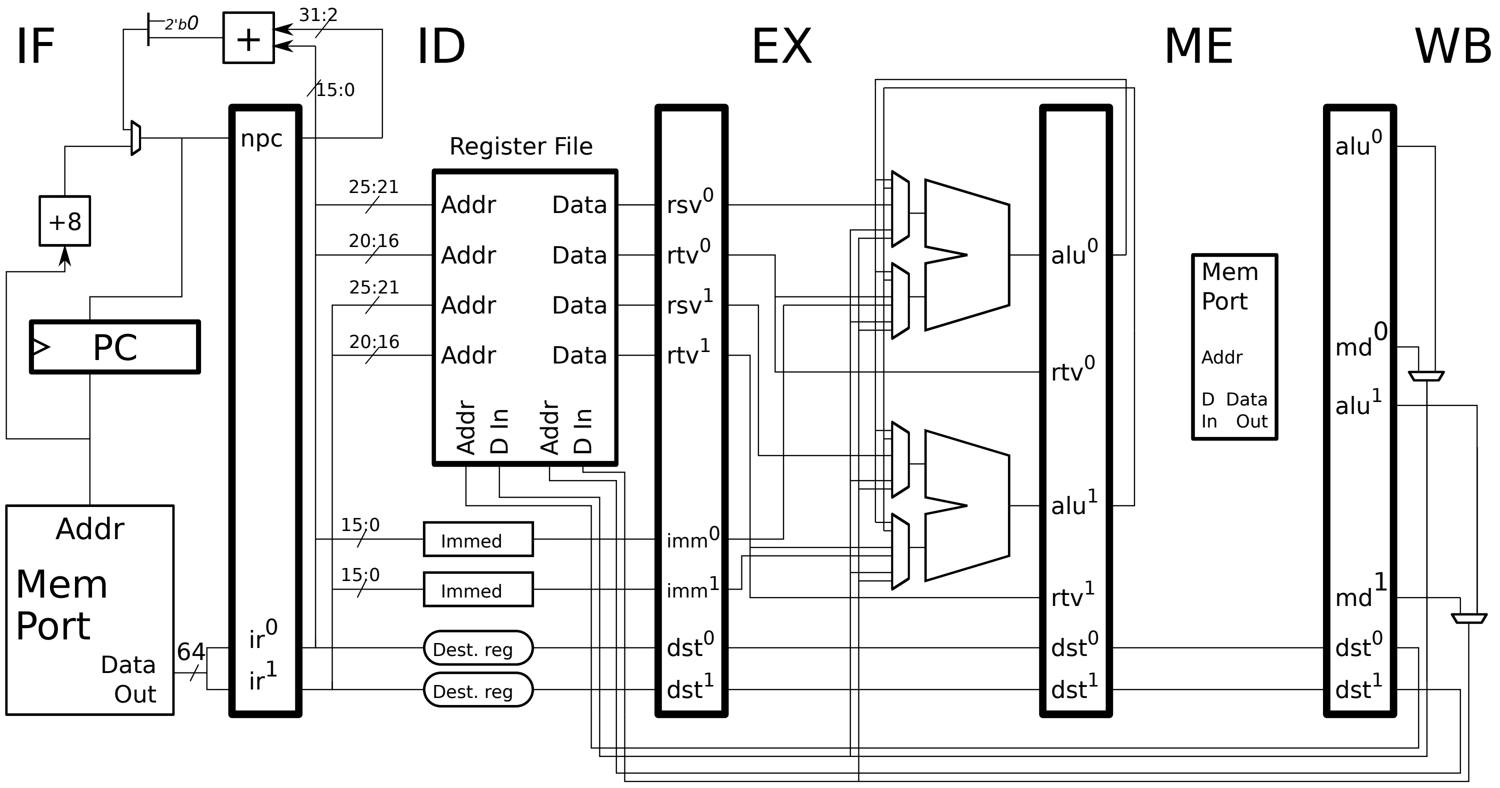
Show control logic to detect possible shared loads in `ID`. Generate a signal named `SHARED_LOAD` which is `1` if the instructions in `ID` could form a shared load, based on register numbers and immediate values. In `EX` generate a `STALL-EX-1-SL` signal if the shared load cannot be performed. This will stall the pipeline allowing the slot-0 load to proceed normally in the current cycle and the slot-1 load to be executed in the next cycle. Show connections for the alignment units in the appropriate places.

☐ Control logic to detect possible shared loads.

☐ Explain why alignment and critical path make it impossible to know for sure in `ID` that a shared load is possible.

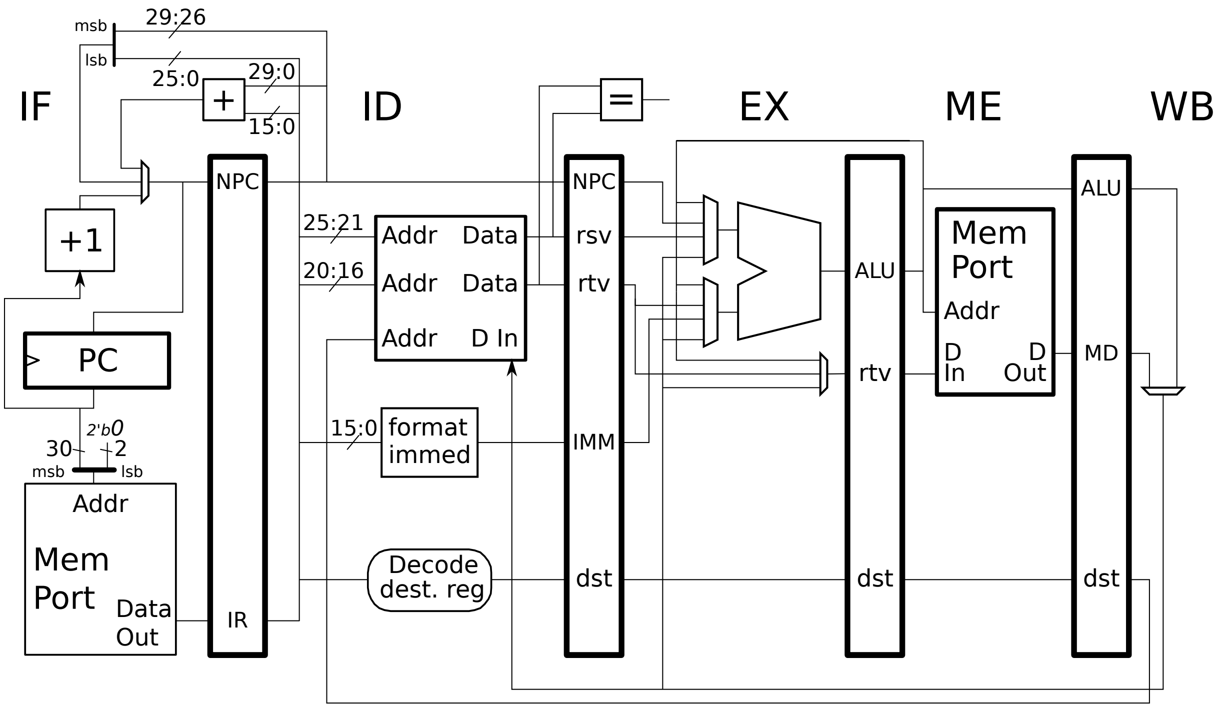
☐ `EX`-stage hardware to generate `STALL-EX-1-SL` if can't do shared load based on addresses.

☐ Add remaining `ME`- and `WB`-stage hardware.



Problem 2: (20 pts) Show the execution of the code fragments below on the illustrated MIPS implementations. All branches are taken. Don't forget to check for dependencies.

(*a*) Show executions.



☐ Show execution of this simple code sequence.

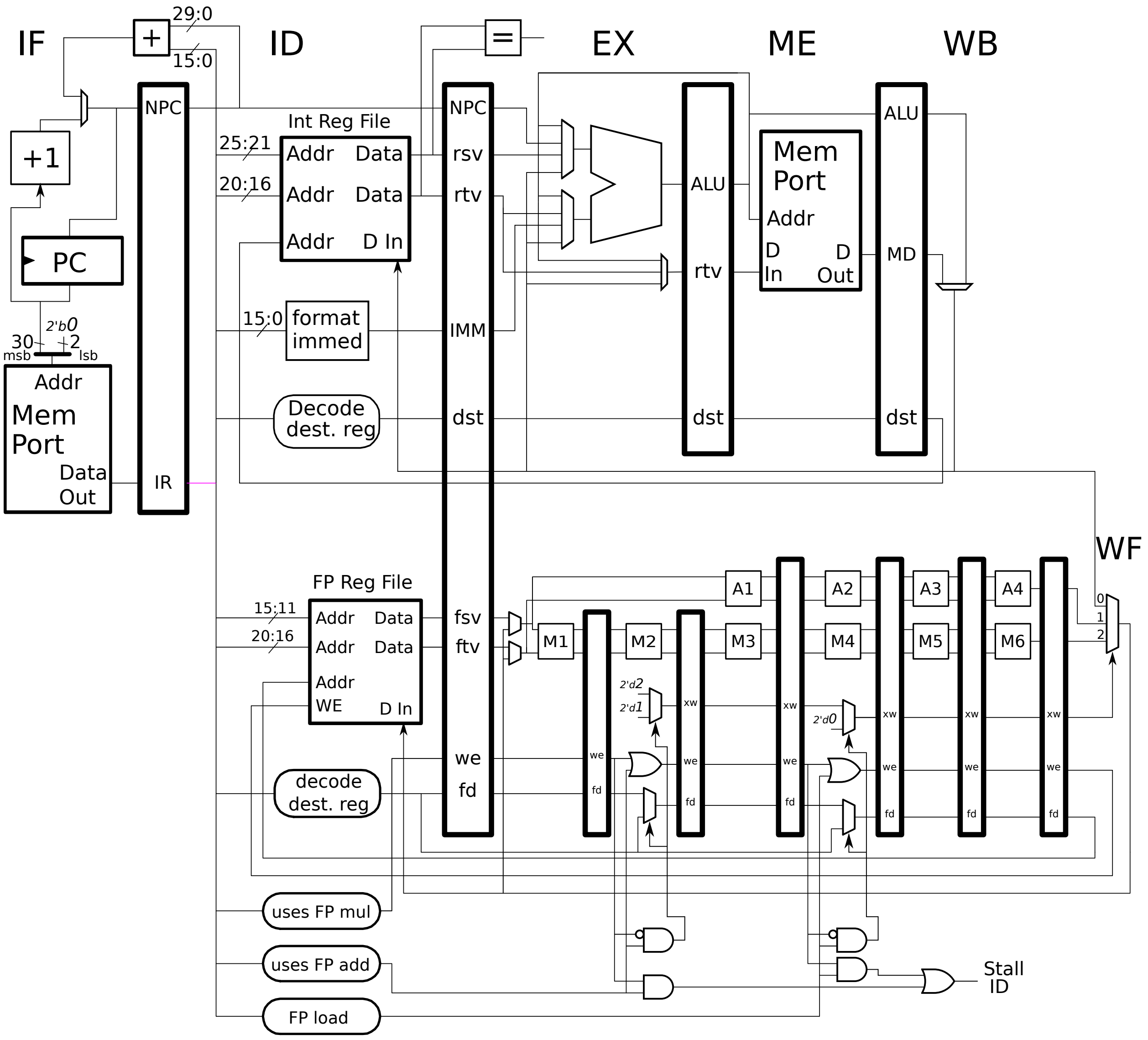
```
add r1, r2, r3

sub r4, r1, r5
```

☐ Show execution of the following code sequence.

```
beq r1, r1 TARG

or  r2, r3, r4

sub r5, r6, r7

xor r8, r9, r10
```

TARG:
```
lw r10, 0(r11)
```

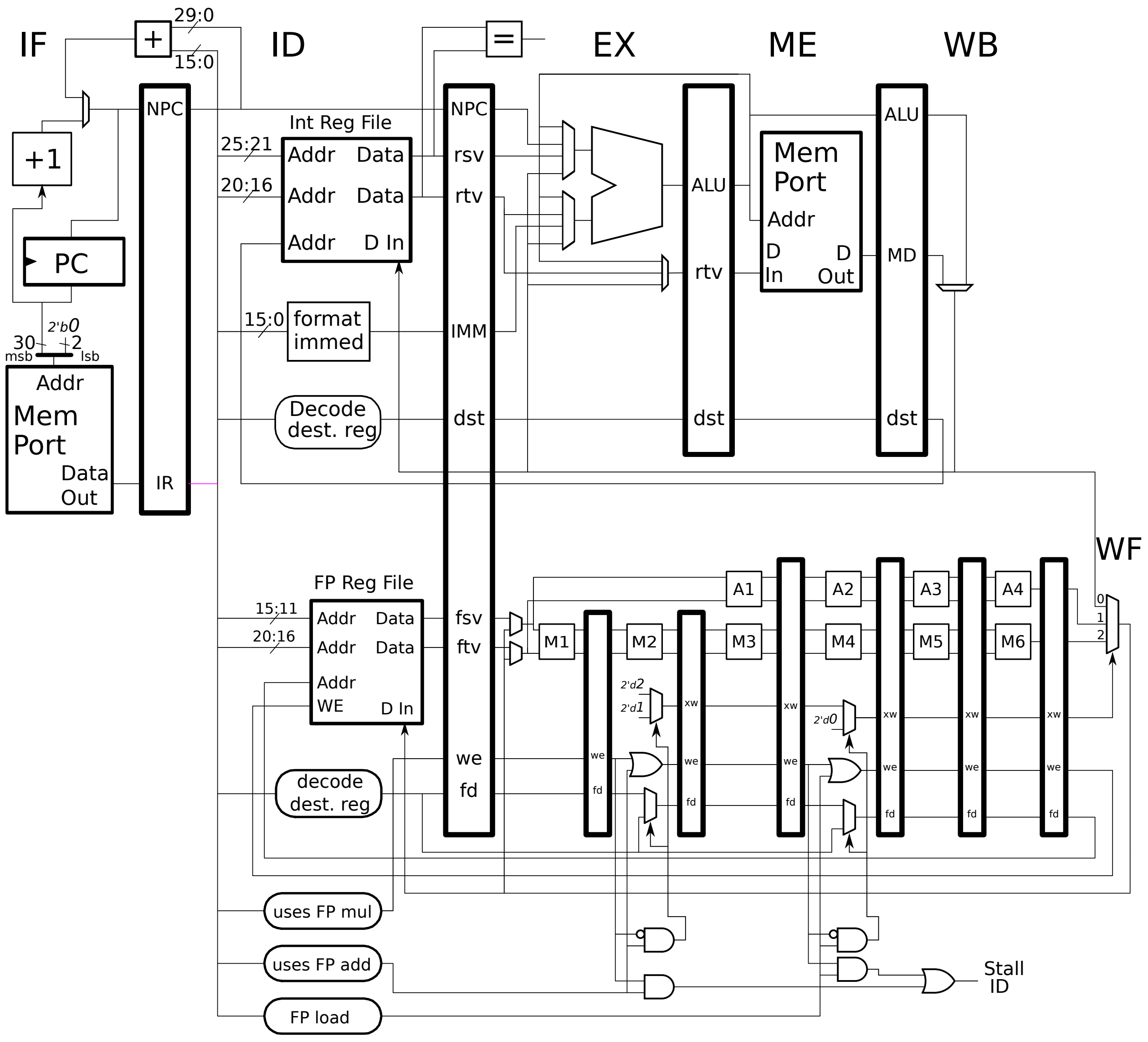
(*b*) Show the execution of the code sequences below on the illustrated MIPS implementation. Don't forget to check for dependencies.



☐ Show execution of the following code sequence.

```
mul.s f5, f6, f7

add.s f1, f2, f3

sub.s f4, f8, f5
```

(*c*) Add any datapath hardware needed to execute the code sequence below, and show its execution. (Control logic is not needed.) Don't forget to check for dependencies.



☐ Add hardware needed by, and ☐ show execution of, the following code sequence.

☐ Consider both stores and, as always, avoid unnecessary costs.

```
add.s f2, f3, f4

swc1 f1, 0(r7)

swc1 f2, 4(r8)
```

Problem 3: (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^{14} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 10-outcome local history, and one system has a global predictor with a 10-outcome global history.

(*a*) Branch behavior is shown below. Two repetitions of a repeating sequence are shown for branches `B1` and `B2`. Branch `B2` generates the following repeating sequence: the eight outcomes `TNTNTNTN` followed by either `NN`, probability .2, or `TT`, probability .8. These last two outcomes are shown below as `r r` and `q q`.

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

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

B2:    T   N   T   N   T   N   T    N   r   r   T   N   T   N T N T N q q ···
```

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

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

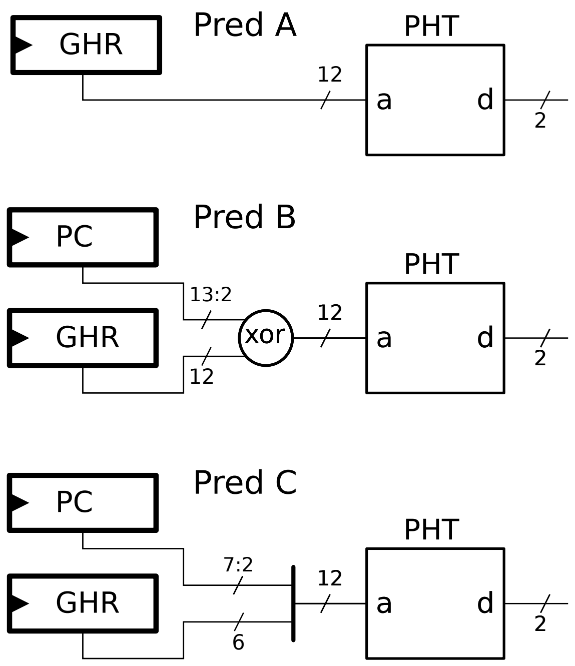
☐ What is the accuracy of the local predictor on `B2`?

☐ What is the minimum local history size needed to predict `B1` with 100% accuracy?

☐ What is the minimum local history size needed to predict `B2` with maximum accuracy?

(*b*) The diagram below shows four branches. Branches B1 and B3 are loop branches, each in a 30-iteration loop. Branches B2 and B4 are perfectly biased branches (B2 always taken, B4 never taken). Consider their execution on the three variations of the global predictor shown below.

```
B1:   TT ··· TN                 TT ··· TN              TT ··· TN
B2:            T                        T                       T
B3:                TT ··· TN                 TT ··· TN
B4:                         N                         N
```



☐ Which variation (A, B, or C) will predict B2 or B4 poorly?

☐ Why does the variation do poorly on one or both of these simple to predict branches?

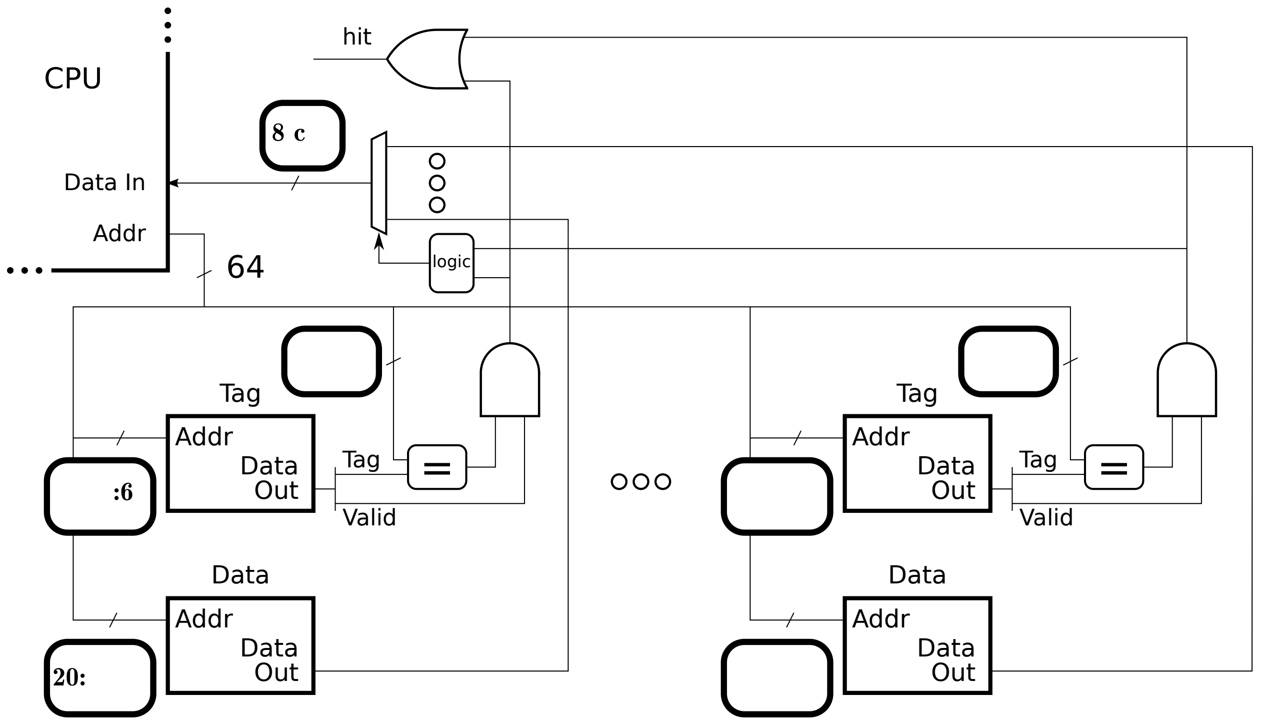
☐ Which variation (A, B, or C) is best for B2 and B4 and also for branches with **repeating patterns**?

☐ Show an example that one variation predicts accurately that the other can not.

Problem 4: (20 pts) The diagram below is for a three-way set-associative cache. Hints about the cache are provided in the diagram.

(*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. Label the sections appropriately. (Index, Offset, Tag.)

Address:

0

☐ Cache Capacity ☐ Indicate Unit!!:

☐ Memory Needed to Implement ☐ Indicate Unit!!:

☐ Line Size ☐ Indicate Unit!!:

☐ Show the bit categorization for the **smallest** direct mapped cache with the same line size and which can hold at least as much data as the cache above.

Address:

Problem 4, continued: The problems on the following pages are **not** based on the cache from Part a. The code in the problems below run on a 16 MiB (2^{24} byte) two-way set-associative cache with a line size of $2^8 = 256$ bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

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

```
long double sum = 0;
long double *a = 0x2000000; // sizeof(long double) == 16
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 4, continued: The code in the problem below runs on a 16 MiB (2^{24} byte) two-way set-associative cache with a line size of $2^8 = 256$ bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*c*) The code below scans through an array (or arrays) twice, once in the `First Loop` and a second time in the `Second Loop`. Three different variations are shown, Plan A, Plan C, and Plan D. For each Plan, find the largest value of `BSIZE` for which the second loop will have a 100% hit ratio.

```
// Note:  sizeof(double) == 8

// Plan A
struct Some_Struct   { double val, norm_val;   double stuff[14];   };

// Plan C
struct Some_Struct   { double val, norm_val; };
struct Some_Struct_2 { double stuff[14];};

// Plan D
double *vals, *norm_vals;
struct Some_Struct_2 { double stuff[14]; };

// Plan A and Plan C
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val + b[i].norm_val;   // First Loop.
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;   // Second Loop.

// Plan D
  for ( int i = 0;  i < BSIZE;  i++ ) sum += vals[i] + norm_vals[i];   // First Loop.
  for ( int i = 0;  i < BSIZE;  i++ ) norm_vals[i] = vals[i] / sum;   // Second Loop.
```

☐ Largest `BSIZE` for Plan A. ☐ Show work or explain.

☐ Largest `BSIZE` for Plan C. ☐ Show work or explain.

☐ Largest `BSIZE` for Plan D: ☐ Show work or explain.

(*d*) What's wrong with the statement below:

In the first iteration of the First Loop under Plan C there will be one miss but with Plan D there will be two misses. Therefore Plan C is better, at least for the First Loop.

☐ What's wrong with the statement?

Problem 5: (20 pts) Answer each question below.

(*a*) Suppose that a new ISA is being designed. Rather than requiring implementations to include control logic to detect dependencies the ISA will require that dependent instructions be separated by at least six instructions. As a result, less hardware will be used in the first implementation.

☐ Explain why this is considered the wrong approach for most ISAs.

☐ What is the disadvantage of imposing this separation requirement?

(*b*) In the execution below the `add.s` instruction encounters an arithmetic overflow when in the `A4` stage and as a result raises an exception in cycle 6.

```
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add.s  f1, f2, f3  IF ID A1 A2 A3 A* WFx
 lwc1 f3, 0(r2)        IF ID EX ME WB
 lw r4, 0(r5)             IF ID EX ME WBx
 add r6, r7, r8              IF ID EX MEx

HANDLER:
 sw                                      IF ID ..
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
```

☐ Explain why this exception can't be precise.

☐ Describe something the handler could do if the exception were precise.

(*c*) Chip A has 60 2-way superscalar cores and Chip B has 20 4-way superscalar cores. Both chips run at 3 GHz, and so Chip A has a higher peak instruction throughput. The cost and power consumption of the two chips are identical. Why might someone prefer Chip B over Chip A?

☐ Chip B preferred, despite lower peak throughput, because:

(*d*) The SPECcpu suite is used to test new chips that might have cost hundreds of millions of dollars to develop, and test results are closely watched.

☐ Why might we trust the SPECcpu selection of benchmarks and rules for building and running the benchmarks?

☐ Why might we trust the SPECcpu scores that Company X publishes?