Name _____

Computer Architecture
EE 4720
Final Examination

11 December 2007, 15:00–17:00 CST

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

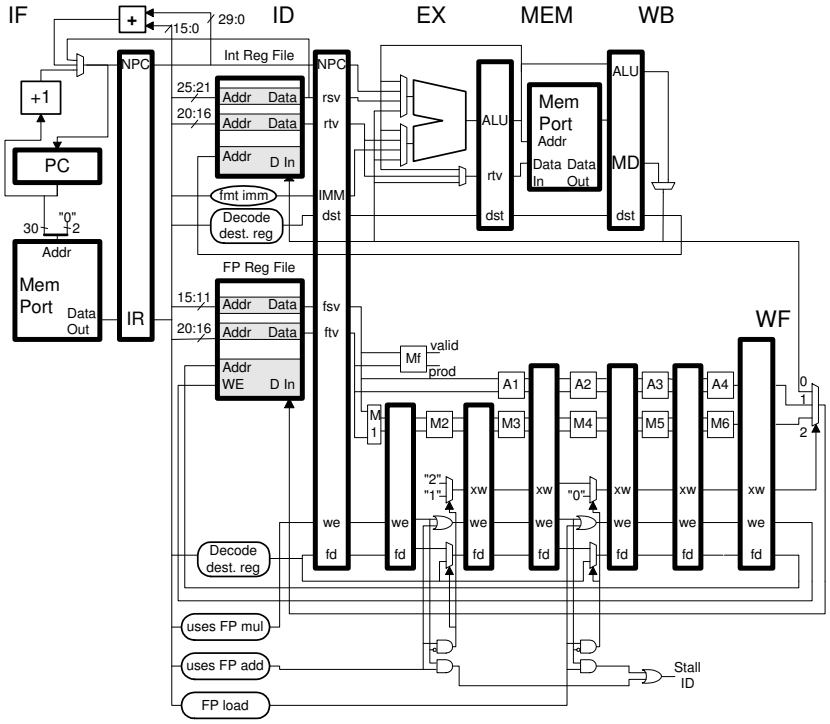
Good Luck!

Problem 1: (25 pts) Suppose it turns out that many `mul.d` instructions are executed with a 0 or 1 for one of the operands, in that case the product could be computed in less than the six stages used in the class implementation. The MIPS implementation below includes a new multiply fast unit, `Mf`, for such situations.

`Mf` has two inputs (unlabeled) and two outputs, `valid` and `prod` (they are not yet connected to anything). As with `A1` and `M1`, IEEE 754 doubles are expected at the inputs of `Mf`. If one of the inputs is 0 or 1 then the `valid` output will be `1` (otherwise it is `0`). If `valid` is `1` the `prod` output is the product of the two inputs; both outputs are available by the end of the cycle.

Call a multiply *fast* if one of its operands is 0 or 1.

USE NEXT PAGE FOR SOLUTION!



USE NEXT PAGE FOR SOLUTION!

(*a*) Add datapath and control logic for `Mf` meeting the requirements below:

☐ Add datapath so the `Mf` output will be written to the FP register file at the right time.

☐ In order of decreasing priority: the added datapath must be correct, must not increase critical paths, and should use as little new hardware as possible.

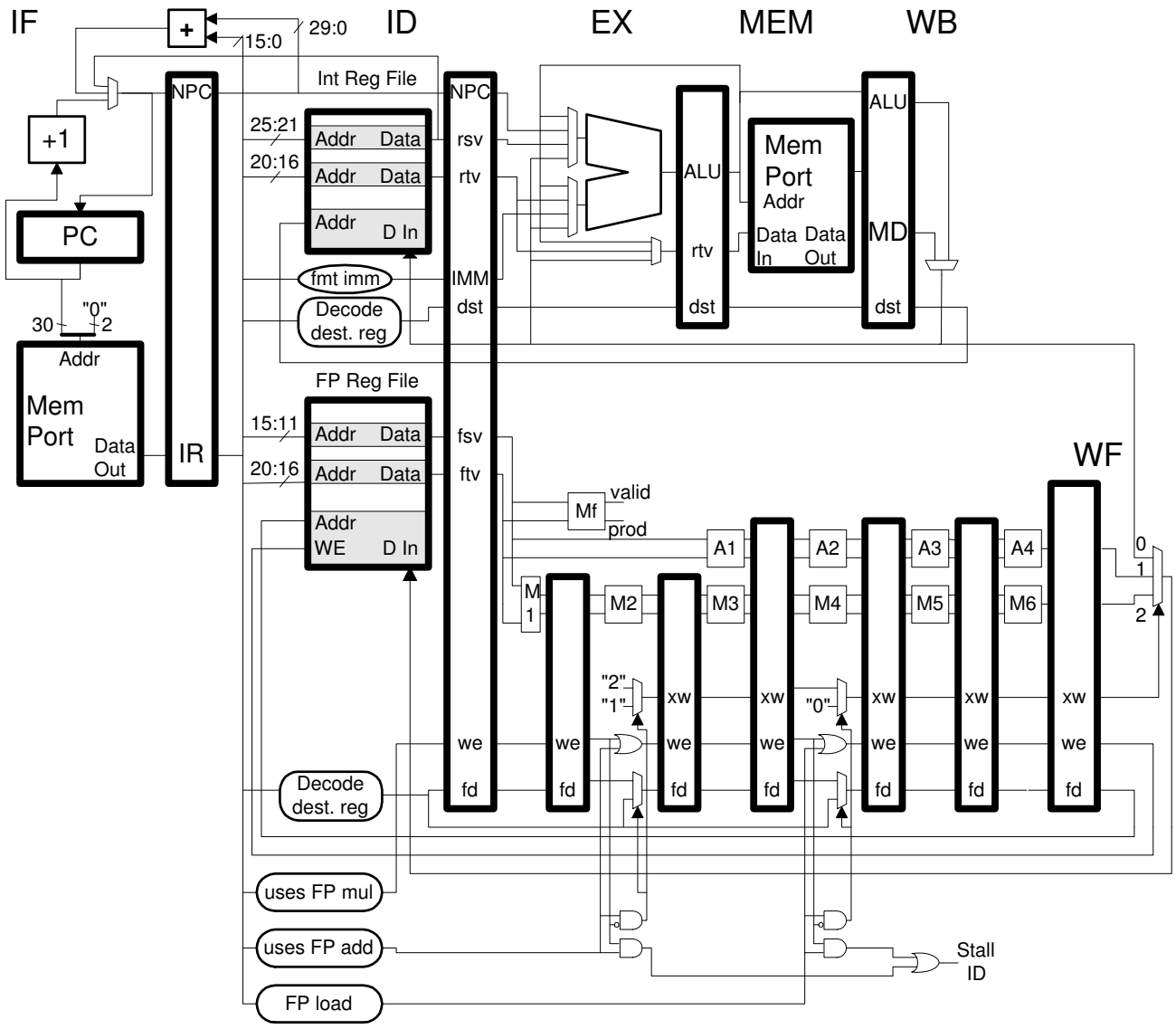
☐ The control logic must detect and handle the new `WF` structural hazard with preceding instructions that's possible when writing a fast product.

☐ If `Mf` is used then the `mul.d`'s usual `WF` slot should be available for other instructions. For example, suppose the second instruction after `mul.d` is an `add.d`. If the multiply is normal the `add.d` would stall, if the multiply is fast the `add.d` shouldn't stall (at least for the `WF` conflict with `mul.d`).

(*b*) Add bypass hardware and control logic so that the code below executes without a stall if the `mul.d` turns out to be fast. For this part assume that multiplexors at FP unit inputs won't add to critical path.

```
mul.d f2, f4, f6
add.d f8, f2, f10
```

☐ Bypass hardware for case above.



Problem 2: (25 pts) Answer the following predictor questions.

(*a*) The code below executes on three systems, one uses a bimodal predictor with a 2^{14}-entry BHT, one uses a local predictor with a 2^{14}-entry BHT and a 7-outcome local history, and one uses a global predictor with a 7-outcome global history. Consider the execution of the code below, all branches are shown.

```
BIGLOOP:
...
B1: beq r1, r2, SKIP1   T T T T N T T T T N T T T T N T T T T N ...
...
SKIP1:
...
B2: bne r3,r4 SKIP2      T T T N N N T T T N N N T T T N N N T T T N N N ...
...
SKIP2:
...
j BIGLOOP
nop
```

For partial credit show work, not just answer.

☐ Accuracy of bimodal predictor on B1 after warmup:

☐ Accuracy of bimodal predictor on B2 after warmup:

☐ Accuracy of local predictor on B2 after warmup:

☐ Warmup time of local predictor on B2:

☐ Minimum local history size for 100% accuracy of local predictor **on B2** (without ignoring B1) (show work):

☐ Accuracy of global predictor on B2 after warmup:

☐ Warmup time of global predictor on B2 (explain):

(*b*) Consider the execution of the code fragment below on a system using branch prediction. The value of `c` used in the switch statement is random, uniformly distributed over `a` to `z`, and outcomes are independent (like a 26-sided die). The branch implementing the `if ( x < 5 )` statement, which will be called the *if* branch, is taken 50% of the time.

```
int c = getchar(); // Unpredictable
switch (c) {
 case 'a': x = 3; j++; break;
 case 'b': x = 7; break;
 ...
 case 'z': x = 1; i++; break;
}
if ( x < 5 ) foo(); else bar();
```

As shown below, the `switch` construct is implemented using a dispatch table and a `jr`, and other jumps. The `switch` construct itself and the case blocks use no branches.

```
 # Part of code implementing switch construct.
 # Value in register $t1 has been computed using variable c.
 lw $t0, 0($t1)   # Load the address of the case statement corresponding to c.
 jr $t0           # Jump to case statement.
 nop
# Case 'a'
 addi $t5, $0, 3  # x = 3;
 j endswitch
 addi $t7, $t7, 1 # j++
...
```

The predictors covered in class would all achieve just a 50% prediction accuracy on the if branch.

Modify one of the predictors used in class so that it does better than 50% on the if branch. *Hint: Note that* `x` *is assigned a different constant in each case statement. A correct solution requires just a small modification of one of the predictors shown in class.*

☐ Briefly explain the idea behind your predictor.

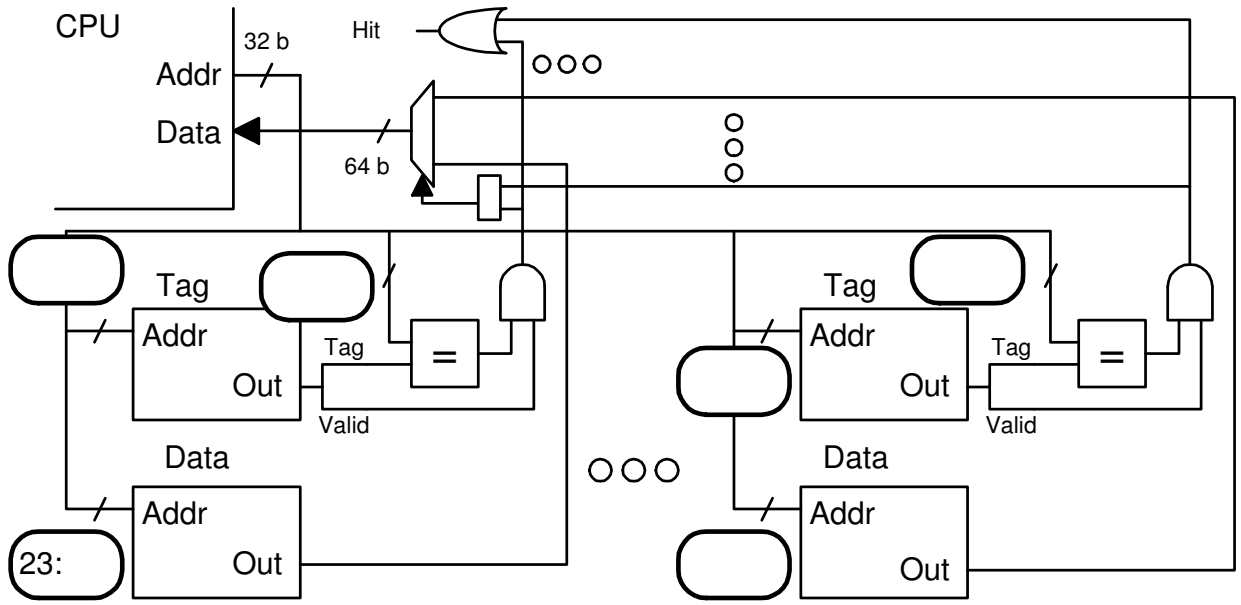
☐ Draw a diagram of the predictor.

☐ Show tables such as BHT and PHT.

Problem 3: (20 pts) The diagram below is for a 256-MiB (2^{28}-character) set-associative cache with a line size of 64 characters on a system with the usual 8-bit characters.

(*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.)

Address: | | | |

☐ Associativity:

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

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

Address: | | | |

Problem 3, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
long *a = 0x2000000;    // sizeof(long) = 8 characters.
int i;
int ILIMIT = 1 << 10;    // = 2^10

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

(*c*) Consider a 1 MiB (2^{20} byte) *direct mapped cache* with a line size of 256 characters (not the same as the one from parts a and b) for a system with a 32-bit address space. Suppose this cache has the following defect: a particular bit position in the tag comparison unit will match even if the tags differ. That is, if the bad bit position were 2 then tags `0x5` and `0x1` would match. Other cache hardware functions correctly. The cache is write through and write around.

☐ Complete the program below so that it finds the bad bit position in a small amount of time and assigns it to `badbit`.

☐ For maximum partial credit briefly describe your strategy.

- The cache is empty when the program starts.
- Assume that any address can be read or written.

```
char *a = 0x1000;
int bad_bit = -1;  // At end should be set to position of bad tag bit.
```

Problem 4: Answer each question below.

(*a*) (6 pts) A trap instruction is sort of like a procedure call (*e.g.*, `jal`) to the operating system.

☐ Describe a difference between a trap and `jal` in how the target address is specified.

☐ Describe another difference between a trap and a `jal`.

(*b*) (6 pts) A `log` (logarithm) instruction is to be added to an ISA. Group E wants to define the `log` instruction as producing the IEEE 754 double representation that is closest to the exact result. Group A would define `log` as producing any result within a certain number of bits of the exact result. Group A argues that the precision of an exact result is not needed and their approximate result is sufficient. *Group E agrees with this*, they want an exact result for other reasons. *Hint: Think about the reasons for separating ISA and implementation.*

☐ Why might group A want an approximate result?

☐ Why might group E want an exact result?

(*c*) (6 pts) Consider two scalar MIPS implementations, implementation A is similar to the one covered in class with the familiar stages `IF ID EX ME WB` while implementation B has stages `IF I1 I2 I3 I4 EX ME WB`. The two implementations run at the same clock frequency and are similar in other ways.

☐ Explain why implementation A does not need a branch predictor, or at best would only gain a small amount of performance.

☐ Explain why a branch predictor would help implementation B much more than implementation A.

☐ Consider the performance of implementation A and implementation B when branch prediction is perfect for both. Which (if any) is faster, and by how much? Explain, state any assumptions made.

(*d*) (6 pts) The code below executes on a two-way superscalar dynamically scheduled machine similar to the one presented in class. The `sub` instruction reads the value of `r1` from the register file in cycle 10, `xori` writes a value for `r1` in cycle 6 and `lw` writes a value for `r1` in cycle 9.

```
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12
lw r1, 0(r2)     IF ID Q  RR EA          ME WB C
add r3, r9, r1   IF ID Q                 RR EX WB C
or  r6, r3, r8      IF ID Q                  RR EX WB C
xori r1, r4, 5      IF ID Q  RR EX WB                C
sub  r5, r1, r3        IF ID Q                  RR EX WB C
```

☐ Briefly explain why the code runs correctly despite the fact that `lw` writes *after* `xori`.

(*e*) (6 pts) Modern VLIW ISAs are designed with modern implementations in mind, unlike decades old RISC ISAs.

☐ Show the contents of a typical VLIW bundle.

☐ Provide an example of a VLIW feature that's designed to make implementation easier. Explain how it does so.