Name Solution

Computer Architecture
EE 4720
Midterm Examination

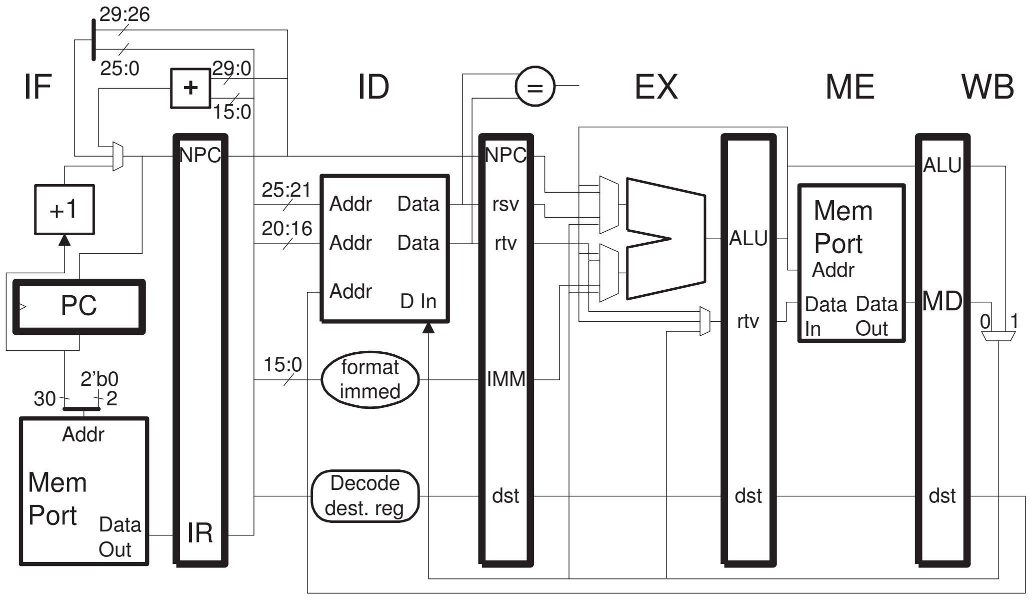
Wednesday, 30 March 2011, 9:40–10:30 CDT

Problem 1		(15 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(10 pts)
Problem 5		(10 pts)
Problem 6		(15 pts)
Problem 7		(15 pts)
Exam Total		(100 pts)

Alias The end of history?

Good Luck!

Problem 1: [15 pts] The MIPS code below executes on the illustrated hopefully familiar implementation.



n

```
      # SOLUTION
LOOP: # Cycle        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
lh r1, 0(r2)         IF ID EX ME WB                            First Iteration
addi r4, r4, 4          IF ID EX ME WB
andi r3, r1, 0xf0f0        IF ID EX ME WB
bne r3, r0, LOOP              IF ID ----> EX ME WB
lw r2, 4(r2)                     IF ----> ID EX ME WB
#       Cycle        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
lh r1, 0(r2)         Second Iteration     IF ID -> EX ME WB
addi r4, r4, 4                               IF -> ID EX ME WB
andi r3, r1, 0xf0f0                                IF ID EX ME WB
bne r3, r0, LOOP                                      IF ID ----> EX ME WB
lw r2, 4(r2)                                             IF ----> ID EX ME WB
#       Cycle        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
lh r1, 0(r2)         Third Iteration                                IF ID -> EX
```

(*a*) Show a pipeline execution diagram for the code below running on the illustrated MIPS implementation for enough iterations to determine CPI.

☑ Show a pipeline execution diagram.

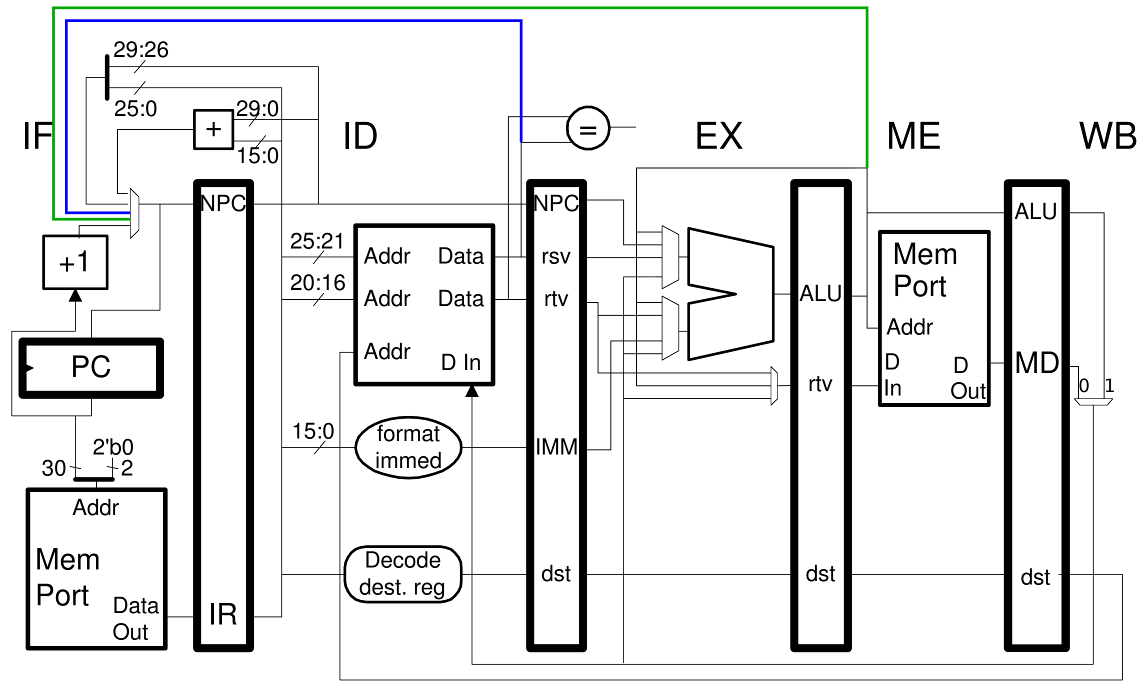
Diagram appears above.

(*b*) Determine the CPI.

☑ Find the CPI.

The state of the second and third iterations are identical when they start, in cycles 7 and 15, respectively. (At both cycles 7 and 15 there is an `lh` in `IF`, `lw` in `ID`, and a `bne` in `EX`, the other stages are empty. Therefore the third iteration will be just like the second, and so on. The time for the second iteration is $15 - 7 = 8$ cycles (using the `IF` of the first instruction as a reference point), and so the CPI is $\frac{15-7}{5} = 1.6$ CPI.

Problem 2: [20 pts] In the implementation below the datapath for the `jalr` and `jr` instructions is not shown.



(*a*) Add datapath (but not control logic) needed for the `jr` instruction to the diagram above.

☑ Add datapath (but not control logic) for `jr`.

Change appear in blue. All that's needed is a path from the register file output for `rsv` to the `IF`-stage mux. Note that the register values are available early enough to avoid the risk of lengthening the critical path.

(*b*) Show the execution of the code below based upon the answer above. Show execution starting from the first instruction (as usual) and continue until `ori` executes or eight instructions have executed whichever is longer. (In the correct answer `ori` is the eighth instruction to execute.) Note that the `jalr` will jump to `ROUTINE` the first time it executes.

☑ Show execution consistent with previous answer.

Solution appears below. Notice that the `jalr` must stall two cycles since it can only get the source register from the register file.

```
# SOLUTION
# Cycles              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 addi r1, r1, 16      IF ID EX ME WB
 jalr r31, r1            IF ID ----> EX ME WB
 addi r4, r31, -8           IF ----> ID EX ME WB
 ori r6, r6, 0xff

# Cycles              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
ROUTINE:
 addi r4, r4, 4                      IF ID EX ME WB
 jr r4                                  IF ID ----> EX ME WB
 lw r31, 0(r7)                             IF ----> ID EX ME WB
```

```
# Cycles                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 # NOTE: These are the instructions that follow the jalr.
 addi r4, r31, -8                                          IF ID -> EX ME WB
 ori r6, r6, 0xff                                             IF -> ID EX ME WB
```

(*c*) If your answer to the previous part was correct the code above would suffer stall(s). Add bypasses to avoid as many of the stalls as possible without significant critical path impact.

☑ Bypass paths to eliminate stalls.

The solution has to balance "as many as possible" with "significant impact." The value at the output of the ALU would not be available until close to the end of the clock cycle, so connecting it to the `IF`-stage mux would likely increase the critical path by a small amount. Instead, an `ME`-stage bypass is used. There will still be one stall cycle but the clock frequency will be untouched. That stall cycle is only suffered when a `jr` or `jalr` instruction immediately follows the instruction writing the target register, and that won't happen very often.

The changes appear in green on the diagram.

Problem 3: [15 pts] In the code fragment below the `lw` raises an exception due to a TLB miss. Remember that a TLB miss is something that happens all the time to almost any load.

Solution omitted.

```
 # Cycle          0   1   2   3   4   5   6   7
 lw r1, 0(r2)     IF
 add r3, r5, r6
 xor r4, r3, r9
...
HANDLER:
 sw r10, ..
```

(*a*) Based on the exception hardware presented in class, at which cycle will the first instruction of the handler be in `IF`?

☑ Handler in `IF` in cycle:

For the next parts of this problem don't consider the exception hardware presented in class, instead the `lw` will trigger a deferred exception. That is, the `lw` will raise a TLB miss exception but `add` will successfully complete `WB` before the handler is called. Register `r1` will have some garbage value but the `add` will execute correctly.

(*b*) This is **not** a good example for why precise exceptions are needed for loads. Why not?

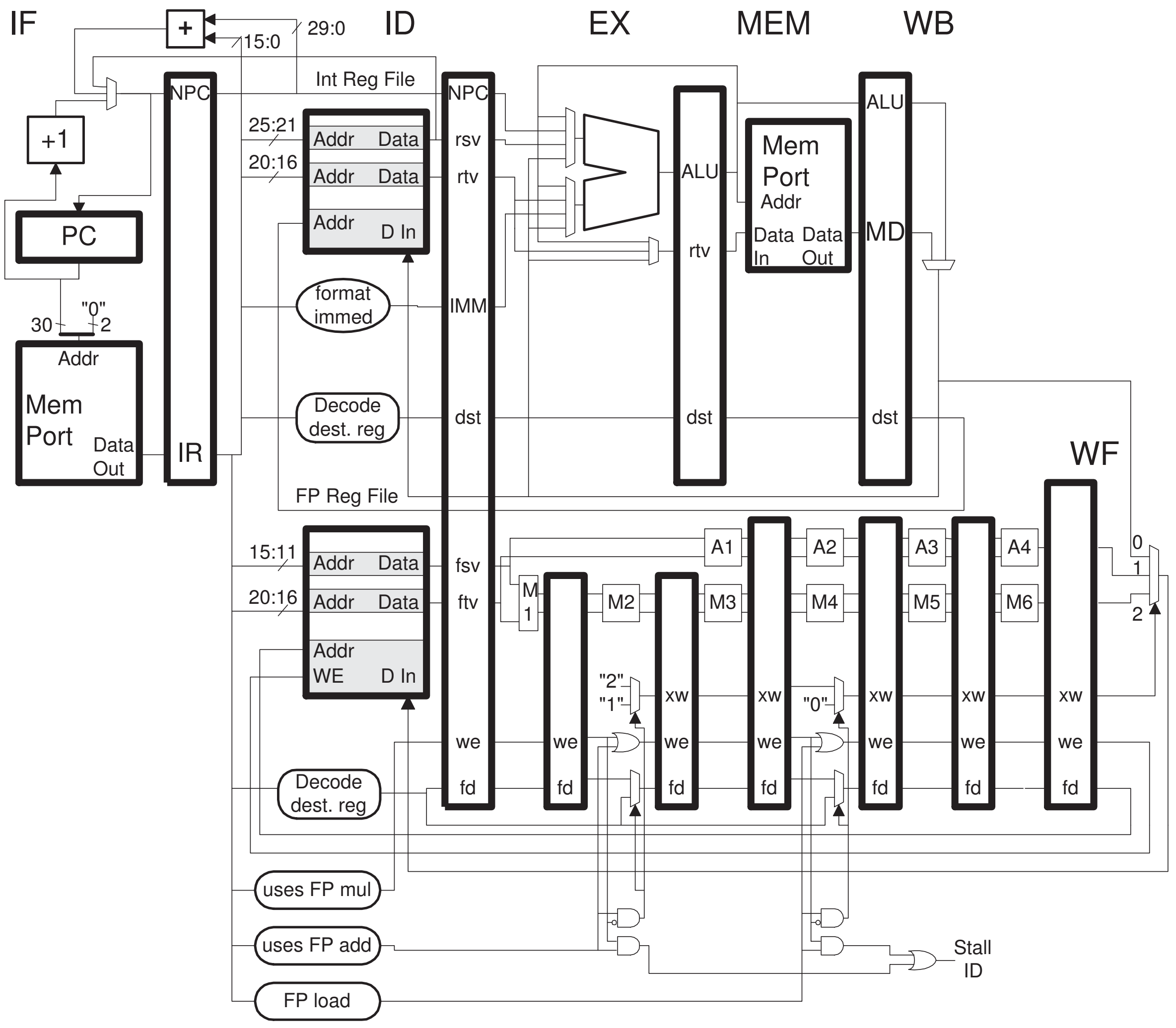
☑ Not a good example because:

(*c*) Modify the code after the `lw` so that this *is* a good example of why precise exceptions are needed. Briefly explain why.

☑ Modify code.

☑ Explain why it is a better example.

Problem 4: [10 pts] Show the execution of the MIPS code below on the illustrated implementation. Assume that all needed bypass paths are available.



☑ Execution of code.

Solution omitted.

```
add.d f2, f4, f6

sub.d f8, f4, f10

mul.d f12, f2, f14

addi r1, r1, 8
```

Problem 5: [10 pts] Answer each question about the SPECcpu suite.

(*a*) With SPECcpu the tester is responsible for providing compilation tools and choosing optimization switches. Which feature or goal of the SPEC benchmarks does that support?

☑ Compilation tools and optimization choices supports:

Test the full potential of the newest implementations and ISAs.

(*b*) Part of the SPEC rules requires that compilation tools be actively marketed as products, not just available to someone who knows exactly how to ask for them. What sort of abuse does that prevent?

☑ Actively marketed compilation tools prevents:

It prevents compiling the suite using a compiler with optimizations that are really fast but which in many cases will generate incorrect code. Specific compiler bugs encountered when building the SPECcpu code might be fixed, but other compiler bugs might remain. A company would not want to hurt its reputation by marketing a buggy compiler, and so the rule helps avoid this abuse.

Problem 6: [15 pts] Answer each question below.

(*a*) ISAs are frequently updated, examples mentioned in class include MIPS I, MIPS II, SPARC v8, SPARC v9, and the multimedia extensions such as MMX, SSE, SSE 2, etc. for IA-32. With all these updates it sounds like an ISA is not really frozen for All Time. Does that mean the main advantage in separating ISA design from implementation design has been lost?

☑ Has main advantage of ISA/implementation separation been lost? Explain.

No. The main advantage is that code written for an ISA will run correctly on any of the ISA's implementations. ISA updates and extensions are done in such a way that code written for an earlier version will run on an implementation of a later version. For example, code written for the SPARC v8 ISA will run correctly on an implementation of the SPARC v9 ISA.

(*b*) Why is the typical RISC program larger than an equivalent CISC program?

☑ RISC larger than CISC because:

Because RISC instruction sizes are all the same, meaning that some instructions take up more space than the equivalent CISC instruction.

(*c*) Explain how the approaches to encoding immediate arithmetic instructions differ in the SPARC and MIPS ISAs.

☑ How approaches differ.

In MIPS integer instructions that require an immediate, such as `addi`, use a different opcode and encoding than their two-source-register counterparts, such as `add`. In contrast, many instructions that use an immediate, such as `add`, use the same opcode and opcode extension, `op3`, for their immediate and two-source-register forms; one bit in the encoding, `i`, distinguishes the two.

Problem 7: [15 pts] Answer each question below.

(*a*) Provide the following optimization examples:

☑ An optimization that can be performed well without knowing the particular implementation being targeted.

Dead code elimination.

☑ An optimization that can be performed well only if the compiler knows the particular implementation being targeted.

Scheduling to avoid stalls.

(*b*) In an alternate universe, when designing the ALT-MIPS ISA, computer engineers insisted, perhaps for cost reasons, that the data memory port and ALU had to be in the same stage (that is, the `EX` and `ME` stages would be combined), resulting in a four-stage pipeline. How would the matching ALT-MIPS be different from our MIPS ISA? Note that elegance and performance are important for both ALT-MIPS and MIPS. Provide two code samples, one for which the four-stage ALT-MIPS is better and one for which the five-stage MIPS is better.

☑ ALT-MIPS differences with MIPS.

Because there is no time to add an immediate to a base register, load and store instructions in ALT-MIPS would not have an offset.

☑ Code sample better for ALT-MIPS.

☑ Explain.

Because the memory port and ALU are in the same stage it is possible to bypass a value from a load instruction to an arithmetic instruction. The example below has such a pair of instructions. Note that the load instruction does not have an offset, making it appropriate for ALT-MIPS. The combined `EX` and `ME` stages are called `EM`. In cycle 3 the value loaded by the **`lw`** is being bypassed to the **`add`**.

```
# SOLUTION         0  1  2  3  4  5
lw r1, (r2)        IF ID EM WB
add r3, r1, r3        IF ID EM WB
```

☑ Code sample better for MIPS.

☑ Explain.

In the first code fragment below, for MIPS, the **`lh`** has an offset, and so it could not execute on ALT-MIPS. The ALT-MIPS version follows, an **`addi`** had to be inserted to compute the load address.

```
# SOLUTION  MIPS Code (could not run on ALT-MIPS)
lw r1, 0(r2)
lh r3, 4(r2)

# The corresponding ALT-MIPS code.
lw r1, (r2)
addi r2, r2, 4
lh r3, (r2)
```