Name Solution

Computer Architecture
EE 4720
Midterm Examination

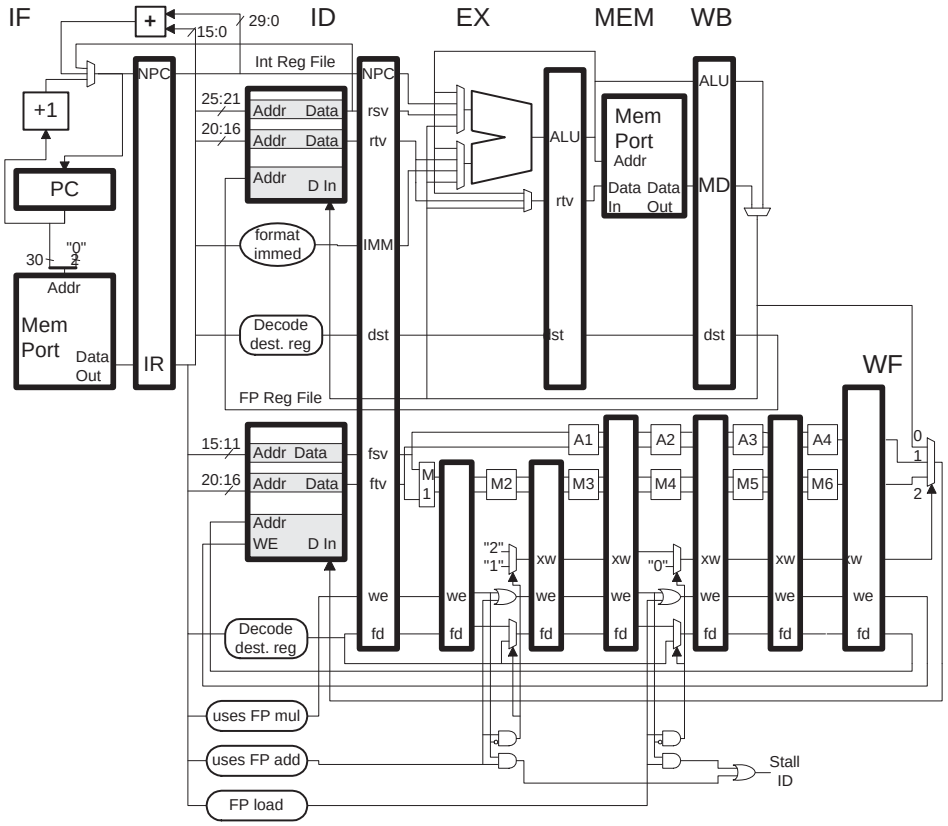
Friday, 27 March 2009, 11:40–12:30 CDT

Problem 1		(30 pts)
Problem 2		(30 pts)
Problem 3		(10 pts)
Problem 4		(20 pts)
Problem 5		(10 pts)
Exam Total		(100 pts)

Alias Blue sunset?

Good Luck!

Problem 1: [30 pts] The code below executes on the illustrated FP pipeline.




```
LOOP:
ldc1 f0, 0(r1)

addi r1, r1, 8

mul.d f2, f0, f0

bneq r1, r2, LOOP

add.d f6, f6, f2
```

(*a*) Add reasonable bypass paths needed by the code above, for both the integer and FP pipelines.

☑ Add reasonable bypass paths. Don't add unneeded bypass paths.

See discussion in the next part.

(*b*) Analyze the performance of the code using your bypasses:

☑ Show a PED for the code above using your bypasses. (Use this page or next page.)

The PED appears below. Three additional bypasses are needed for the code below. In cycle 4 `mul.d` uses two bypasses, from `WF` to `M1`, for the value loaded by `ldc1`. Also in cycle 4, `bneq` uses a bypass from `ME` to `ID`. In cycle 10 the `add.d` uses a bypass from `WF` to `A1` (this would be one of the bypasses to `mul.d`).

```
# SOLUTION
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
ldc1 f0, 0(r1)     IF ID EX ME WF
addi r1, r1, 8        IF ID EX ME WB
mul.d f2, f0, f0         IF ID M1 M2 M3 M4 M5 M6 WF
bneq r1, r2, LOOP           IF ID EX ME WB
add.d f6, f6, f2               IF ID ----------> A1 A2 A3 A4 WF
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
ldc1 f0, 0(r1)                    IF ----------> ID EX ME WF
addi r1, r1, 8                                   IF ID EX ME WB
mul.d f2, f0, f0                                    IF ID M1 M2 M3 M4 M5 M6 ..
bneq r1, r2, LOOP                                      IF ID EX ME WB
add.d f6, f6, f2                                          IF ID ----------> ..
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
ldc1 f0, 0(r1)                                               IF ----------> ..
```

☑ Compute the CPI of the code for a large number of iterations.

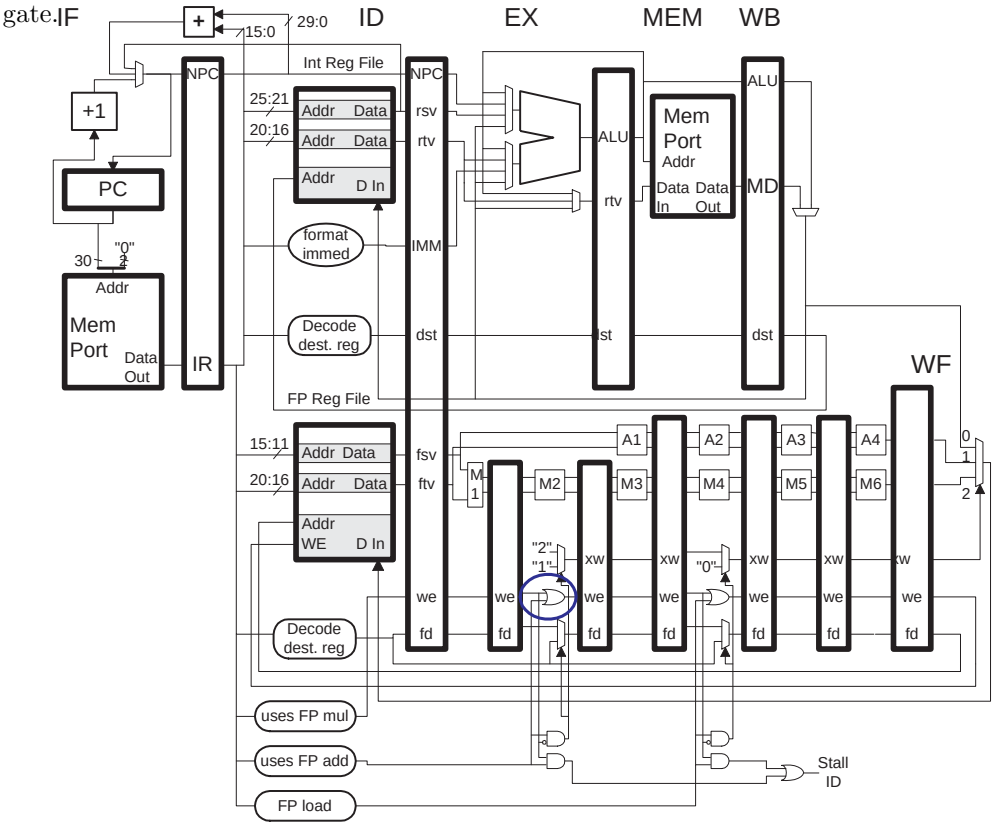
The iteration starting at cycle 5 and 14 start with the pipeline in the same state (`ldc1` in IF, `add.d` in ID, `bneq` in EX, etc.) and so the iteration starting at cycle 5 will match future ones. The iteration starting at cycle 5 takes $14 - 5 = 9$ cycles and uses five instructions, so the average instruction execution time is $\frac{14-5}{5=1.8\,\mathrm{CPI}}$.

Use this page for PED, if needed.

```
LOOP:
ldc1 f0, 0(r1)

addi r1, r1, 8

mul.d f2, f0, f0

bneq r1, r2, LOOP

add.d f6, f6, f2
```

Problem 1, continued:

(*c*) A component failure in the MIPS implementation below has changed the circled OR gate into an AND gate.



☑ Is it still possible to perform a FP multiply? If yes, show how with PED, if no explain.

```
# Original Code. (For answer show modified code with PED.)
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
mul.d f0, f2, f4  IF ID M1 M2 M3 M4 M5 M6 WF
nop                  IF ID EX ME WB
nop                     IF ID EX ME WB
```

Yes. First consider the original code with two `nop`s added, above. In cycle 3 the `mul.d` is in M2, the circled gate will have a `1` input from the `M1/M2.we` latch and a `0` input from the uses FP add logic, and so its output will be zero, snuffing out the `mul.d` instruction.

Now consider the code execution below in which the second `nop` is changed to an `add.d`. In cycle 3 the `mul.d` in M2 will be "helped" by the `add.d` in ID and so the output of the circled gate will be `1`; with this help the `mul.d` will complete normally.

The `add.d` stalls in ID in cycle 3 (as it should) but in cycle 4 it suffers the same problem as the `mul.d` in the original code: the `we` signal is `0` when it should be a `1`. As a result the `add.d` will not finish completely. All other logic is working correctly so there is no way to "fix" the `add.d`'s we signal in the later stages and so there is no way the `add.d` can finish.

```
# SOLUTION Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
mul.d f0, f2, f4  IF ID M1 M2 M3 M4 M5 M6 WF
nop                  IF ID EX ME WB                <- Other insns would work.
add.d f8, f2, f4        IF ID -> A1 A2 A3 A4 WF    <- Let mul go through.
```

☑ Is it still possible to perform a FP add? If yes, show how with PED, if no explain.

```
# Original Code. (For answer show modified code with PED.)
add.d f0, f2, f4
```

No. See the solution to the previous part.

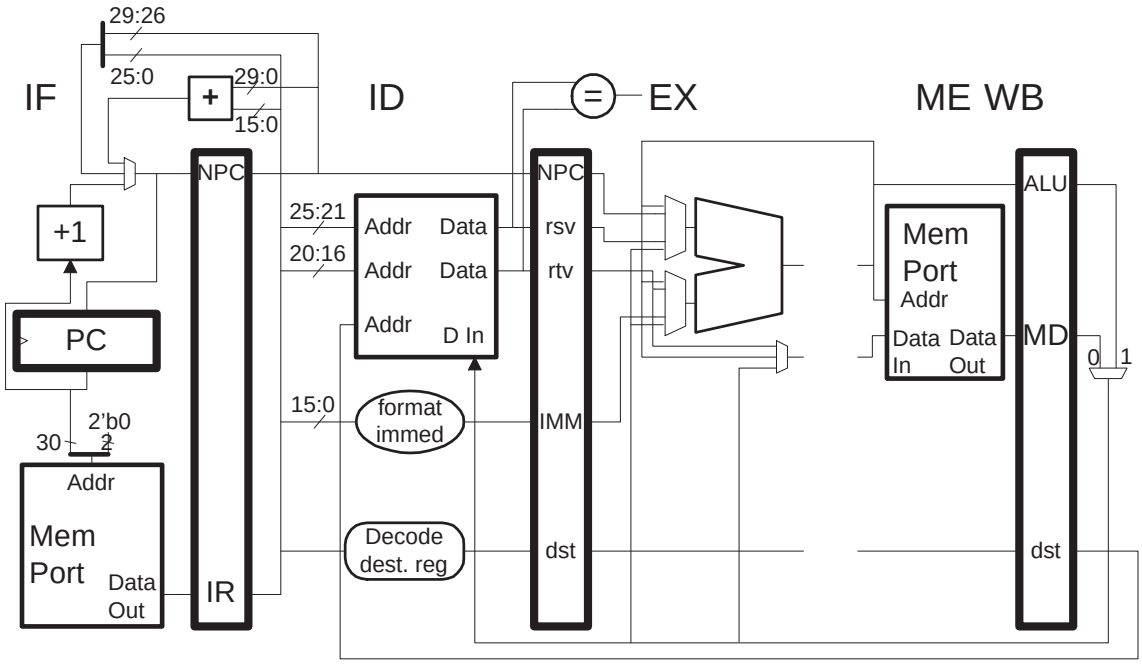
Problem 2: [30 pts] The MIPS-A ISA is like MIPS-I except that load and store instructions use only the `rs` register value for an address, they don't add an offset (or anything else). As a result MIPS-A can be implemented with a four-stage pipeline.

(*a*) Our familiar 5-stage MIPS-I implementation appears below with one of the pipeline latches missing. Make additional changes so that this is a reasonable four-stage implementation of MIPS-A.

☑ Show all connections to the memory port.

☑ Cross out wires and other items that are not needed, add what is needed.

The memory port address input should be connected to the output of the upper ALU mux. The bypass connections that used to be from ME should be removed. Ask for a diagram if needed!



(*b*) Describe two ways in which this MIPS-A implementation costs less than the five-stage MIPS-I.

☑ Two reasons for lower cost.

Reason 1: No **ME** to **EX** bypass connections (including the wires themselves, multiplexor inputs and control logic). Reason 2: No **EX/ME** pipeline latches.

Problem 2, continued:

(*c*) The four-stage MIPS-A implementation may be faster or slower than the five-stage MIPS-I.

☑ Provide a pair of equivalent code fragments, one for MIPS-I that runs on the five-stage implementation and one for MIPS-A that runs on the four-stage, in which the MIPS-A version is faster. *Hint: The MIPS-I code will have a familiar stall.*

Because the memory port and ALU are in the same stage the MIPS-A implementation can bypass a load value to the next instruction, avoiding a stall that occurs in MIPS-I. See the two code fragments. Note that the fact that WB occurs in four rather than five cycles of IF is not in itself a performance advantage, the advantage is in avoiding the stall.

```
# SOLUTION
# MIPS-I
lw r1, 0(r2)    IF ID EX ME WB
add r3, r1, r4     IF ID -> EX ME WB

# MIPS-A
lw r1, (r2)     IF ID EM WB
add r3, r1, r4     IF ID EM WB
```

☑ Provide a pair of equivalent code fragments, one for MIPS-I that runs on the five-stage implementation and one for MIPS-A that runs on the four-stage, in which the MIPS-A version is slower. *Hint: Think about the difference in the ISAs.*

MIPS-A will have to perform additions to retrieve data at an offset (small distance) from some base address, something that MIPS-I can avoid. See the examples below.

```
# SOLUTION
# MIPS-I
lw   r3, 0(r2)
lw   r4, 4(r2)

# MIPS-A
lw   r3, (r2)
addi r2, r2, 4
lw   r4, (r2)
```

(*d*) A company needs to decide whether to develop MIPS-I or MIPS-A. How does it decide? Assume that at this point software compatibility is not an issue.

☑ How should company decide which to develop?

Benchmarks based on customers' workloads should be analyzed to determine how often loads ands stores use non-zero offsets and how often a load/use stall cannot be avoided. If there are more unavoidable load/use stalls than non-zero offsets then MIPS-A might be better, otherwise MIPS-I might be better.

☑ Will having skilled compiler writers tilt the decision towards MIPS-A or MIPS-I? Explain.

MIPS-I, because the compiler can schedule to avoid load/use stalls, but there is little the compiler can do to avoid using offsets in most cases.

Problem 3: [10 pts] Answer the questions below.

(*a*) Why does MIPS have a `beq` but does not have a `blt` (branch less than), even though a `blt` instruction would be frequently used?

☑ Why `beq` but no `blt`?

Because the magnitude comparison needed for `blt` would take more time than the equality comparison needed for `beq`, long enough to lengthen the critical path.

(*b*) Explain why a branch delay slot can be though of as a short-sited feature in an ISA.

☑ Delay slots are good when...

The implementation has five stages and is scalar. Execution proceeds without a stall or squash whether or not a branch is taken (assuming no difficult dependencies).

☑ ...but delay slots are bad when...

The implementation is superscalar or deeper than five stages (roughly). In that case more than one instruction will be fetched by the time a branch is resolved and so instructions will have to be squashed. The benefit is smaller, but all of the complexity is still there.

Problem 4: [20 pts] The doing-it-the-hard-way MIPS code below loads the constant $\frac{1}{3}$ in IEEE 754 single-precision format, `0x3eaaaaab`, into register `f2`.

```
addi r10, r0, 1
addi r20, r0, 3
mtc1 f12, r10
mtc1 f22, r20
cvt.s.w f14, f12
cvt.s.w f24, f22
div.s f2, f14, f24
```

(*a*) Describe what the `mtc1` and `cvt.s.w` instructions do.

☑ Explain `mtc1`

Move to co-processor 1. This moves the value in `r10` to `f12`. It only moves it, it does not do format conversion.

☑ Explain `cvt.s.w`

Convert word (32-bit integer) to single (IEEE 754 single-precision floating point).

(*b*) The code above uses more instructions than are necessary and also uses a wastefully time-consuming instruction.

☑ What is the wastefully time-consuming instruction?

The divide, `div.s`. It's time-consuming because it's a divide, it's wasteful because it's computing something at run time that could have been computed at compile (or assembler-writing) time. See the next part.

☑ Re-write the code so it uses fewer instructions without using load instructions. *Hint: A correct answer uses three instructions and a piece of information slipped into the first sentence of the problem.*

Rather than computing it, just load the pre-computed FP value of $\frac{1}{3}$ into a register:

```
# Solution
lui r10, 0x3eaa
ori r10, r10, 0xaaab
mtc1 f2, r10
```

Problem 5: [10 pts] Answer the following SPECcpu questions.

(*a*) In the SPECcpu2000 suite profiling was allowed for both base and peak results but in SPECcpu2006 profiling is allowed for peak results but not for base results.

☑ Why isn't profiling allowed for base results in SPECcpu2006? Your answer should say something about the difference between base and peak.

The base results should reflect the performance attainable with normal effort. Profiling requires selecting a training set, compiling the program with instrumentation on, running the code on the training set, then re-compiling. Most programmers don't go to so much trouble.

(*b*) Company A has a reputation for reliable compilers, company B has a reputation for buggy compilers. Both companies and their customers are okay with this. (Think Italian sports cars.)

Optimization X results in a substantial improvement in SPECcpu base scores. Company B has it in their shipping compilers (those sold to customers) but company A only has optimization X in their experimental compilers (not available to customers), but they are working hard on it. The optimization achieves the same results for company A and B. Company B's compiler will not run on company A's system (as though they'd want to!). *Grading Note: The last line was not in the original exam, but a student did see the possibility of using B's compiler for A's spec run.*

☑ Can Company A use optimization X to prepare SPECcpu2006? Explain.

No, because they do not sell the compiler as a product.

☑ Can Company B use optimization X to prepare SPECcpu2006? Explain.

Yes, because the compiler is a product.

☑ Your answers above should reflect the letter of SPEC's rules. Do you think this achieves SPECcpu's goals or what you would like to see in benchmark results? Explain.

Yes, because SPECcpu does not claim to measure factors other than performance, such as cost, power consumption, or reliability. A person deciding between Company A and Company B's systems would balance the performance as measured by the SPECcpu numbers against the companies' reputations for quality. If A and B used the same compiler there would be no way to make this judgment.