Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 26 March 2010, 10:40–11:30 CDT

Problem 1		(40 pts)
Problem 2		(12 pts)
Problem 3		(14 pts)
Problem 4		(10 pts)
Problem 5		(24 pts)
Exam Total		(100 pts)

Alias Buffer Size + 1

Good Luck!

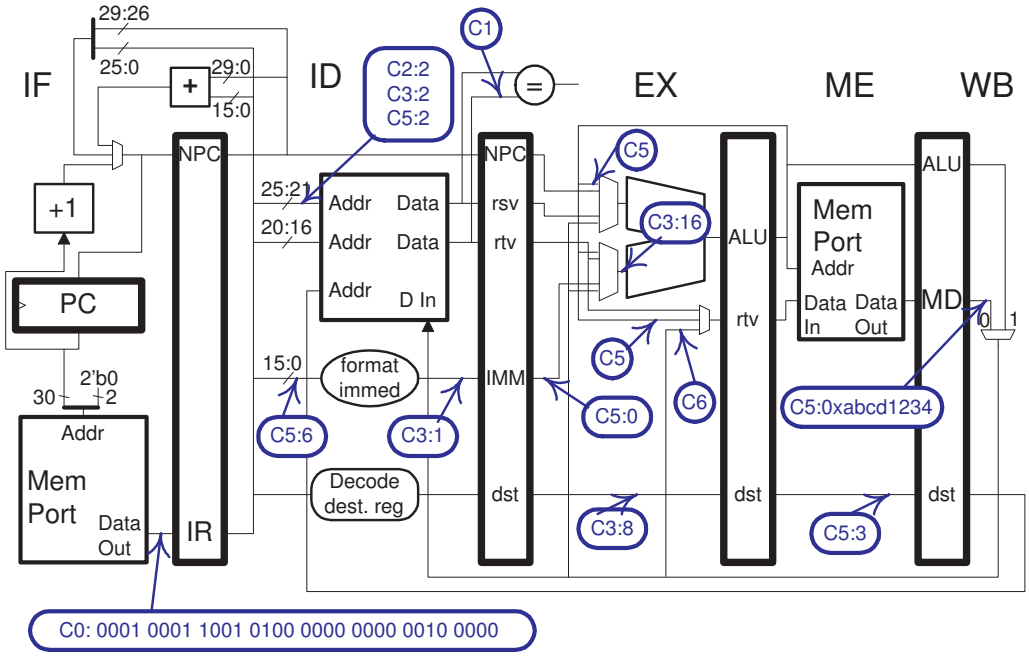
Problem 1: [40 pts] In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c5:6 indicates that at cycle 5 the wire will hold a 6. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Write a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ The first instruction address has been provided, **show the addresses of the remaining four instructions**.

☑ The third instruction is an `addi`, don't forget to show its registers and immediates.

☑ If an instruction is a load or store, show all possible size and sign possibilities. For example, (lw,lh)




```
# SOLUTION
#
# Cycle                 0  1  2  3  4  5  6  7  8
0x1000 beq r12, r20 TA  IF ID EX ME WB
0x1004 lw r8, 16(r2)       IF ID EX ME WB
TA:0x1084 addi r3, r2, 1      IF ID EX ME WB
0x1088 sb r3, 0(r3)              IF ID EX ME WB
0x108c sh (or sb) r3, 6(r2)         IF ID EX ME WB
# Cycle                 0  1  2  3  4  5  6  7  8
```

Easy Stuff: The destination for second instruction, `r8`, and the third instruction, `r3`, can be read off the `ID/EX.dst` and `EX/ME.dst` pipeline latches. Similarly, the first source register of several instructions can be read directly at the input to the register file in `ID`. Three immediate values are directly provided on both sides of the `ID/EX.IMM` pipeline latch. If this doesn't seem obvious within five minutes, please please please see the Statically Scheduled System Study Guide for tips and other sample problems, also consider asking for help. Don't bother reading the next paragraphs until everything above is easy.

Dependencies: The use of bypass paths shown by the two C5 bubbles and C6 bubble in **EX** reveals three dependencies. For example, the upper C5 means that the **rs** source of the fourth instruction is the same as the destination of the third (otherwise the bypass path would not be used). In this particular problem the result of the third instruction is bypassed to three places, two in the fourth and one in the fifth instruction. The destination register of the third instruction, **r3**, was directly determined; with the dependencies we just found we see that the three source registers must all be **r3**.

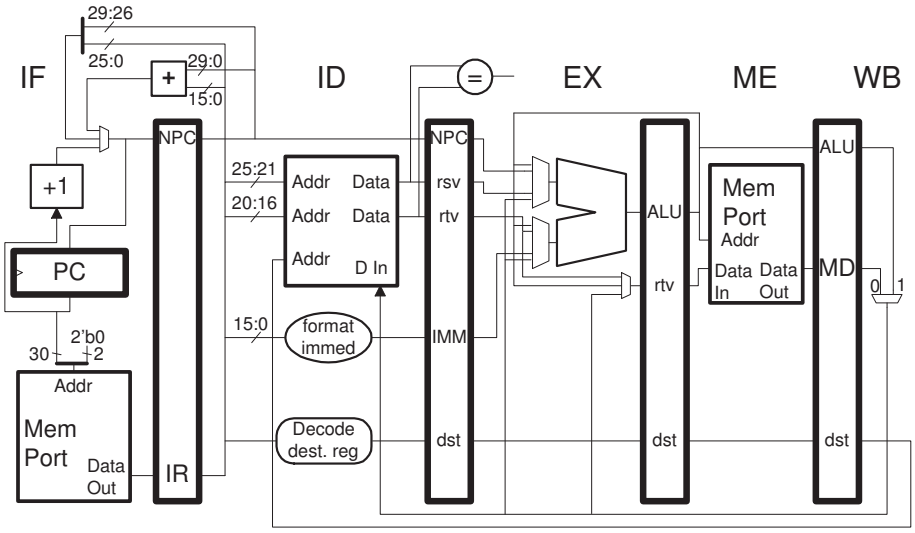
Rough Instruction Identities: Only store instructions can use the **EX/ME.rtv** latch, so we know that the fourth and fifth instructions must be stores. Only load instructions use the **ME/WB.MD** latch, and so the second instruction must be a load. Only branches use the = in the **ID** stage, and so the first instruction must be a branch.

The lower ALU input: At this point we know that the instruction in **EX** in cycle 3 is some kind of load, and so the C3:16 must be its immediate value.

Load and Store Data Sizes: Since the Mem Port sign-extends or pads the loaded value, the C5:0xabcd1234 could only be from a **lw**. (If it were a **lh**, which loads 16 bits, the upper 16 bits, instead of being **0xabcd**, would have to be either **0** or **0xffff**.) MIPS' load and store addresses must be aligned, meaning, they must be a multiple of the data size (in characters [popularly called bytes]). One does not need to be an expert on number theory to figure out that in that case the contents of register **r2** must be a multiple of 4 (given that **r2** $+ 16$ is a multiple of 4). We should also be able to figure out that **r3**, after the **addi**, cannot be a multiple of 4 or of a 2. Therefore the fourth instruction can only a **sb**. The fifth instruction can be a **sb** or a **sh**.

That long thing in **IF***:* The bubble in **IF** provides the entire first instruction. Test takers were not expected to memorize opcodes, but were expected to know the instruction formats. Since we already know it's a branch we should know it uses a type I format. Parsing that we find that **rs** is 12, **rt** is 20, and the immediate is **0x20**. The immediate is used to find the branch target: $1000_{16} + 4 + 4 \times 20_{16} = 1084_{16}$. The solution is written as though the branch was taken, but there is not enough information to tell whether it was taken or not and so a solution with the third instruction at the fall-through address, **0x1008**, would also be correct.

Problem 2: [12 pts] Consider the following cost-reducing design options for the MIPS implementation shown below. Performance is still important, and a compiler will be able to optimize for this lower-cost implementation, but even with skillful optimization there may be some performance loss.



(*a*) Suppose one had to choose between eliminating all upper-ALU-input bypass paths or all lower-ALU-input bypass paths. Which would you choose? *Note: The following sentence did not appear on the original exam.* (Only bypass paths are eliminated, other mux inputs remain.)

☑ Eliminate: upper or lower. (Circle one.)

☑ Briefly explain why.

Eliminate bypasses in lower mux. If upper mux bypasses were eliminated then we could not bypass any type I instructions. With the lower-mux inputs eliminated the compiler could accommodate at least some type R instructions that would have needed the lower mux by switching the `rs` and `rt` operands. (That would only work for commutative operations.)

(*b*) Suppose one had to choose between eliminating all upper-ALU-input bypass paths or all rtv (not to the ALU) bypass paths. Which would you choose (Note: Only bypass paths are eliminated, other mux inputs remain.)

☑ Eliminate: upper or rtv. (Circle one.)

☑ Briefly explain why.

This is an easy choice: Eliminate rtv bypasses. Stores are less frequent than type R instructions. *Grading Note: The opposite argument would get partial credit.*

(*c*) Suppose that the mux in the WB stage was moved to the ME stage.

☑ How would that reduce cost?

There would be one fewer ME/WB pipeline latch.

☑ How might that affect performance?

It would likely hurt performance because the memory port is most likely on the critical path. The clock frequency would have to be slowed to give time for the signal to propagate through the mux.

Problem 3: [14 pts] The branch delay slot ISA feature eliminates the need for the stall or squash following a branch that occurs in implementations such as our 5-stage pipeline.

(*a*) The two MIPS code fragments below do not exactly make a strong case for delay slots. In both cases there is no squash or stall, which sounds good. For each code fragment indicate whether performance is slower, equal to, or faster than corresponding non-delay-slot ISA code on a corresponding implementation (one that will suffer a squash after every branch). If faster indicate if the improvement is full (based on the eliminated squash) or partial. *Hint: it won't be full.*

```
# -------------------- Code Fragment 1 --------------------
beq r1, r2, TARG
nop
lw r3, 0(r1)
```

☑ Delay slot in code above results in: slower, equal, partial improvement, full improvement. (CIRCLE ONE).

☑ Explain.

```
# SOLUTION
# Execution with branch not taken.
# Delay Slot ISA  0  1  2  3  4  5  6      # Non Delay Slot ISA  0  1  2  3  4  5
beq r1, r2, TARG  IF ID EX ME WB           beq r1, r2, TARG      IF ID EX ME WB
nop                  IF ID EX ME WB        lw r3, 0(r1)             IF ID EX ME WB
lw r3, 0(r1)            IF ID EX ME WB

# Execution with branch taken.
# Delay Slot ISA  0  1  2  3  4  5  6      # Non Delay Slot ISA  0  1  2  3  4  5
beq r1, r2, TARG  IF ID EX ME WB           beq r1, r2, TARG      IF ID EX ME WB
nop                  IF ID EX ME WB        lw r3, 0(r1)             IFx
lw r3, 0(r1)
...
TARG: xor r4, r3, r5    IF ID EX ME WB     TARG: xor r4, r3, r5        IF ID EX ME WB
```

Slower performance.

Brief explanation that would get full credit: In the delay-slot ISA code the **nop** guarantees that nothing is useful is done after the branch, but in the non-delay-slot code the **lw** would do something useful if the branch were not taken.

Longer explanation: Two executions of the given code fragment are shown above on the left-hand side, in the first the branch is not taken, in the second it is taken (with an added target instruction). On the right-hand side a version of the code is shown for an ISA without delay slots; it too is shown executing both with the branch not taken, and taken.

In the not-taken case the **lw** is the first instruction after the branch doing something useful. That is executed one cycle earlier in the non-delay-slot code. In the taken case the **xor** instruction (added for this explanation) does useful work. That is executed at the same time (cycle 4) for both ISAs.

Therefore the delay-slot code is slower (with the reasonable assumption that the branch is sometimes not taken).

```
# -------------------- Code Fragment 2 --------------------
beq r1, r2 SKIP
add r9, r4, r5
or r6, r9, r8
SKIP:
sub r3, r6, r5
lw r9, 0(r3)
```

☑ Delay slot in code above results in: slower, equal, partial improvement, full improvement. (CIRCLE ONE).

☑ Explain.

```
# SOLUTION
# Execution with branch not taken.
#                 Delay Slot ISA            Non Delay Slot ISA
# Cycle           0  1  2  3  4  5  6       0  1  2  3  4  5  6
beq r1, r2 SKIP   IF ID EX ME WB            IF ID EX ME WB
add r9, r4, r5       IF ID EX ME WB            IF ID EX ME WB
or r6, r9, r8           IF ID EX ME WB            IF ID EX ME WB
SKIP:
sub r3, r6, r5
lw r9, 0(r3)

# Execution with branch taken.
#                 Delay Slot ISA            Non Delay Slot ISA
# Cycle           0  1  2  3  4  5  6       0  1  2  3  4
beq r1, r2 SKIP   IF ID EX ME WB            IF ID EX ME WB
add r9, r4, r5       IF ID EX ME WB            IFx
or r6, r9, r8
SKIP:
sub r3, r6, r5          IF ID EX ME WB            IF ID EX ME WB
lw r9, 0(r3)               IF ID EX ME WB            IF ID EX ME WB
```

Equal performance.

The `add` instruction in the delay slot is only useful if the branch is not taken. So when the branch is taken the delay-slot version of the code executes the `add` instruction but ignores the result (`r9` is overwritten by the `lw` before being read), while the non-delay-slot version fetches but then squashes the `add` instruction. Either way the first useful instruction after the branch is executed at the same time in both ISAs (cycle 3 if the branch is not taken, and cycle 4 if the branch is taken).

(*b*) How can Code Fragment 2 (above) be improved by profiling? Show the optimized fragment and how profiling led to the optimized fragment.

☑ Optimized version of Code Fragment 2

```
# SOLUTION
# Code identical for both ISAs.
# Execution shown for delay-slot ISA.
# Cycle            0  1  2  3  4  5  6  7  8  9
bne r1, r2  LINEX  IF ID EX ME WB
SKIP:
sub r3, r6, r5        IF ID EX ME WB
                             IF ID EX ME WB    (Second execution of sub).
lw r9, 0(r3)                    IF ID EX ME WB
...

LINEX:
J SKIP:                  IF ID EX ME WB
or r6, r9, r8               IF ID EX ME WB
```

☑ Specifically, how did profiling help?

Brief answer that would get full credit: Profiling told us which branch outcome is more likely, and so we could re-arrange the code so that the more-likely outcome is not taken, resulting in fewer wasted or squashed instructions.

Longer answer: The code above works best on both systems if the branch is not taken. Assume that with profiling we find that the branch is mostly taken. In the code above the **beq** was changed to a **bne** and the following code was re-arranged, so now the branch is mostly not taken. In the original code the **add** would be executed unnecessarily (or squashed) when the branch is taken. In the profile-optimized code above the **sub** is executed one extra time if the branch is taken.

Problem 4: [10 pts] The SPECcpu2006 integer suite has 12 benchmarks. Suppose instead it included only four benchmarks, but those benchmarks were chosen fairly, given that only four could be chosen.

(*a*) Considering just the base score (or ignoring the base/peak distinction altogether), what is the disadvantage of having just four benchmarks?

☑ Disadvantage of having four benchmarks.

Four is not enough to represent the broad range of characteristics found CPU and memory bound applications.

(*b*) Suppose that with this smaller set of benchmarks the difference between the base and peak scores was smaller than if 12 benchmarks were used. Assume that in both cases the testers were skilled and followed the rules. Also assume that the base score values were about the same with 4 or 12 benchmarks.

☑ Give a reason for the smaller difference that has to do with the different rules for base and peak tests.

In base tuning the same optimization switches must be used to prepare all benchmarks sharing a language. In peak, each benchmark can have its own set of optimization switches. Base rules are designed so that testers do not perform time-consuming per-benchmark tuning.

There are some optimizations that are good for a few benchmarks but bad for some others. Suppose such a switch helps 1% of the benchmarks and hurts 20%, and has no effect on the remaining 79%. With twelve benchmarks testers are more likely to leave out such switches because they might help just one benchmark but hurt several others. But with only four benchmarks there is a lower chance that a switch that helps one benchmark would hurt the other three. Therefore testers would spend lots of time tuning benchmarks under base tuning, which is not what the base tuning is supposed to measure.

Grading Note: Nobody got this right, nor did anyone mention the common-switch rule for base that was the key to answering this question.

☑ Does the smaller difference indicate that base and peak are not measuring what they should?

Yes, they are not measuring what they should because base is supposed to indicate normal compiling effort, but with so few benchmarks testers could still get away with per-benchmark tuning (see previous answer).

Problem 5: Answer the questions below.

(*a*) [14 pts] Write the following MIPS code fragments with the minimum number of instructions. If you don't know the exact mnemonic, such as `mtc1` or `cvt.w.f`, make up something that sounds right.

☑ Put constant `0x12345678` into register `r1`.

```
# Solution
lui r1, 0x1234
ori r1, r1, 0x5678
```

☑ Jump to address `0x72345678` from address `0x1000`.

```
# Solution
lui r1, 0x7234
ori r1, r1, 0x5678
jr r1
```

☑ Jump to address `0x72345678` from address `0x1000` using a SPARC-like `jmpl` (but in MIPS, like Homework 2).

```
# Solution
lui r1, 0x7234
jmpli r0, r1, 0x5678
```

☑ Registers `r1` and `r2` contain integers. Write register `f4` with the sum as a single-precision floating point number.

```
# Solution
add r3, r1, r2
mtc1 f2, r3
cvt.w.s f4, r2
```

(*b*) [10 pts] Typical CISC ISAs have a range of immediate sizes for use in ALU instructions, up to the data size limit. (Say, 8, 16, 24, and 32 bits). MIPS, and other RISC ISAs, have just one immediate size for ALU instructions.

☑ Why can't MIPS have a 32-bit immediate?

Because the entire instruction is 32 bits, there would be no place to put it.

☑ Why can a CISC ISA have a 32-bit immediate?

CISC ISAs have variable-size instructions, and so the instruction size can be adjusted to accommodate whatever immediate size is needed.

☑ Why would there be no benefit for MIPS to have both 8- and 16-bit immediate sizes for arithmetic instructions?

☑ What is the benefit for CISC ISAs in having 8-, 16-bit, (and more) immediate sizes for arithmetic instructions?

The instructions would be smaller, and then so would the programs. On modern general purpose systems the benefit would be a smaller instruction cache (not yet covered).