Name Solution

Computer Architecture
EE 4720
Midterm Examination

Monday, 10 March 2008, 12:40–13:30 CDT

Problem 1		(50 pts)
Problem 2		(20 pts)
Problem 3		(10 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias Well ARMed

Good Luck!

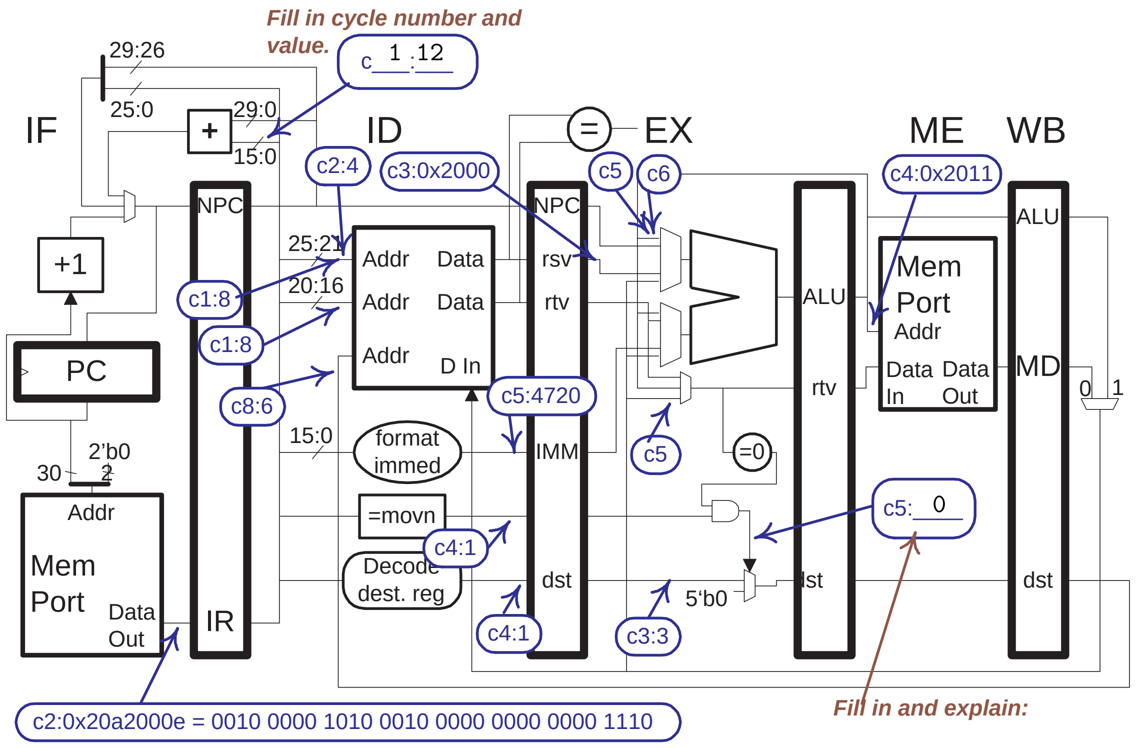
Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, $\boxed{\text{c2:4}}$ indicates that at cycle 2 the wire will hold a 4. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Note that instruction addresses have been provided. [50 pts]

☑ Finish a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Be sure to fill the two blocks marked *Fill In*.

☑ Provide an explanation for the EX-stage fill-in block.



```
# SOLUTION

#  Cycle                  0  1  2  3  4  5  6  7  8
0x1000 beq r8, r8, 0x1034 IF ID EX ME WB
0x1004 lb r3, 0x11(r4)       IF ID EX ME WB
0x1034 ADDi r2, r5, 14          IF ID EX ME WB
0x1038 movn r1, r2, r3             IF ID EX ME WB
0x103c ORi  r6, r1, 4720              IF ID EX ME WB
#  Cycle                  0  1  2  3  4  5  6  7  8
```

Solution discussion on next page.

Solution Discussion

`0x1000`: This has to be some kind of control transfer because of the break in consecutive addresses after the next instruction (the address of the third instruction is `0x1034` instead of `0x1008`). The ID-stage fill-in points at the dedicated adder, which is used only for branches (not jumps or traps). Therefore this instruction must be a branch, the cycle number on the fill in must be 1 (when this instruction is in ID), and the value must be the displacement which is $\frac{0x1034-0x1004}{4} = 12$. (The displacement is the number of instructions to skip starting at the delay-slot instruction which in this case is at address `0x1004`.) The $\boxed{\text{c1:8}}$ bubbles tell us the registers to use for comparison and also that it must be a `beq`. It can't be a `bne` because the branch is taken and it can't be a branch such as `bgtz` because those branches only read one register value.

`0x1004`: The $\boxed{\text{c4:0x2011}}$ tells us this is a memory operation that operates on byte-sized data (since the address is not a multiple of 4 or 2). The $\boxed{\text{c3:3}}$ tells us that it writes a destination register so it must be a load. The $\boxed{\text{c2:4}}$ reveals the source register and $\boxed{\text{c3:3}}$ in EX provides the destination. The displacement, `0x11`, is determined using $\boxed{\text{c3:0x2000}}$ in EX and $\boxed{\text{c4:0x2011}}$ in ME.

`0x1034`: The bubble in IF provides the encoded form of this instruction. It can't be a type-J instruction because the last instruction follows the penultimate one, and so it is either type-R or type-I. Either way, the rs register is `r5` and the `rt` register is `r2`. If it were a type-R the rd register would be `r0` but since, as the $\boxed{\text{c5}}$ at the upper ALU mux indicates, the result is bypassed to another instruction and so it can't be type-R because `r0` would not be bypassed. Therefore it's a type-I. Unless one memorized MIPS opcodes there is no way to determine exactly which type I instruction it is so any arithmetic instruction would be considered correct. The opcode is for an `addi`.

`0x1038`: The lower $\boxed{\text{c4:1}}$ in ID provides the destination register, `r1`, and the one above that shows that this is a `movn` instruction. The ALU $\boxed{\text{c5}}$ provides the rs register number, `r2` from `0x1034`, the rtv mux $\boxed{\text{c5}}$ provides the rt register number, `r3` from the `0x1004 lb`.

If the $\boxed{\text{c5:\ \ }}$ EX-stage fill-in is 0 then the `movn` instruction completes the move, otherwise the destination register remains unchanged. The instruction at `0x103c` bypasses the `movn` destination, `r1`. This would only be correct if the `movn` was to write `r1` and therefore the fill-in value must be 0. (If the value were 1 the `movn` would not change `r1` but the instruction at `0x103c` would read the value of `r2` rather than `r1`.)

`0x103c`: The $\boxed{\text{c5:4720}}$ in ID provides the immediate, indicating that this must be a type-I instruction. The $\boxed{\text{c6}}$ in EX gives the rs register, `r1` from the `movn`. The $\boxed{\text{c8:6}}$ gives the destination register, `r6`. This can be any arithmetic type I instruction, `ori` is assumed.

Problem 2: Answer the following questions about, or inspired by, ARM.

(*a*) [10 pts] MIPS lacks a counterpart to the ARM instruction shown below. *Hint: This has nothing to do with MIPS'* `movn`.

```
mov r1, #5   // Move the constant 5 to register r1.
```

☑ Explain how `r0` makes such a MIPS instruction unnecessary.

You can use an `addi` instruction with `r0` as the first source register.

☑ Show how to perform the same operation using MIPS instruction(s).

```
addi r1, r0, 5
```

(*b*) [5 pts] The ARM ISA states that the result of executing an instruction like `str r15, [r1]` is that either `PC+8` or `PC+12` is stored in memory, depending on the implementation. (Remember that ARM `r15` is an alias for `PC`.)

☑ What is the benefit of making the result of the store implementation dependent?

Some implementations would be lower cost because they wouldn't need to send a "correct" value of `PC` through the pipeline, instead they would use the `PC` value of whatever instruction was being fetched when the store reached `ME`.

(*c*) [5 pts] Consider an ISA which stated that the number of branch delay slots could be either zero or one, depending on the implementation.

☑ As an ISA feature, how does this delay-slot implementation dependence compare in practicality to ARM's store `PC` implementation dependent behavior?

The delay slot implementation dependence is a really bad idea because portable code (code that runs on any implementation of the ISA) could not put anything other than a NOP in delay slots. Since branches are frequent the impact on performance would be large (unless the code were compiled for a specific implementation). In contrast, the stored `PC` could always be added to a constant (-8 or -12) to obtain the correct `PC` value. That would take a little extra time but storing the `PC` in memory using a store instruction is not something programs need to do very often.

Problem 3: In RISC ISAs instructions are of fixed size, that is, all instructions are the same size. For example, in MIPS, all instructions are 32 bits. The character size in MIPS (and all ISAs mentioned in this test) is 8 bits.

(*a*) [5 pts] Describe a benefit of having fixed-size instructions. (This answer can be mentioned in the next answer's explanation.)

☑ Fixed-size instruction benefit for RISC.

Benefits: Instruction fetch is simple because the PC is incremented by a constant amount and instructions are fetched already properly aligned (because instruction addresses are required to be a multiple of four).

(*b*) [5 pts] In the Itanium VLIW ISA all instructions are 41 bits. Why would 41-bit instructions be difficult or wasteful to implement in a RISC ISA, such as MIPS, but cause no difficulties in VLIW ISA implementations, including Itanium implementations.

☑ Forty-one bit RISC instructions difficult or wasteful because:

An instruction would use six bytes of memory, that would either waste 7 bits or else require having the end of one instruction and the beginning of the next instruction together in one memory location. Branch targets would be harder to compute since one would have to multiply by 6 (and that's the space-wasting alternative). A multiply by six is just an add of two shifted values, but that's still one more trouble than just shifting a displacement.

☑ Forty-one bit VLIW instructions not wasteful and make sense because:

Fetch operates on 3-instruction bundles, which are 128 bits (16 bytes), and there is no way to jump to a middle of a bundle. There is no wasted space because $128 - 3 \times 41 = 5$ bits are used to store dependency information. It is easy to compute displacement, just shift left by 4 bits.

Problem 4: Answer the following compiler questions.

(*a*) [10 pts] A company is considering removing a bypass connection in a design of an implementation. Analysis on good test programs shows that performance drops by 5% with the bypass removed (and no other changes).

☑ What can compiler writers do about the performance drop?

They can re-do code scheduling so that the bypass is rarely used. For example, suppose the bypass is used when an `sll` uses the result of a preceding `lw`. The compiler might place two (or whatever) instructions between the `lw` and `sll`.

(*b*) [10 pts] Dead-code elimination is a commonly used compiler optimization, while profiling is used less often because it is a multi-step process.

☑ Using an example briefly describe dead-code elimination.

```
# Solution:
 x = a + b;   // The compiler would produce no code for this line ...
 x = c + d;   // ... because this line overwrites the first x before it's used.
```

☑ Briefly describe the steps used in profiling.

First, compile the program using a profile switch. Second, run the program on what is expected to be typical input data. The program will write profile information. Third, compile again, this time telling the compiler to read the profile information.

☑ Which is more likely to result in disappointing results that surprise the programmer? Explain.

Profiling is more likely to disappoint because the "typical input data" chosen for the profiling run may not match what most end-users use closely enough to get good profiling results. This might happen because there are many different inputs in common use and the compiler would optimize differently for different inputs so no profile run would result in good performance on all, or even most, inputs.