Name Solution

Computer Architecture

EE 4720

Midterm Examination

Wednesday, 8 November 2007, 10:40–11:30 CST

Problem 1		(50 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Exam Total		(100 pts)

Alias 630 bugs to FF3

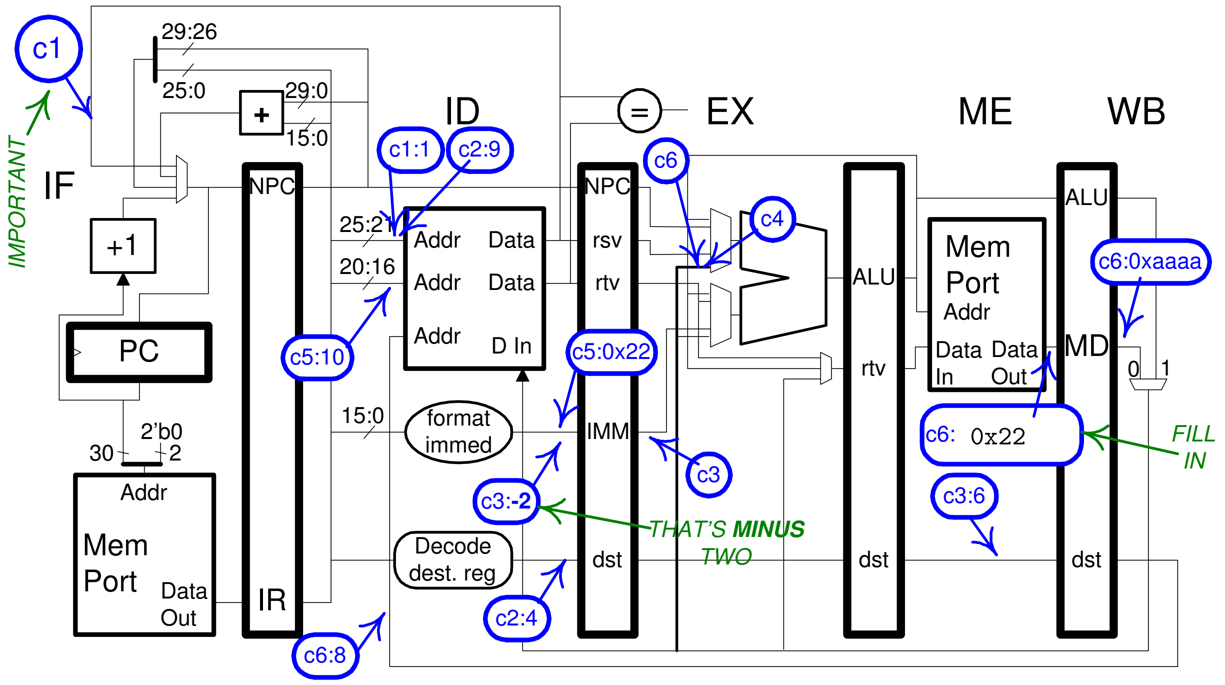
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, $\boxed{\text{c2:9}}$ indicates that at cycle 2 the wire will hold a 9. 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 the fourth instruction has 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 block marked *Fill In*.



```
# SOLUTION

#   Cycle                   0   1   2   3   4   5   6   7   8
0x1000 jalr r6, r1          IF  ID  EX  ME  WB
0x1004 ADDi r4, r9, 0xaaaa      IF  ID  EX  ME  WB
0x2000 lhu r8,-2(r6)                IF  ID  EX  ME  WB
0x2004 lb r3, 0xB(r1)                   IF  ID  EX  ME  WB
0x2008 sW r10, 0x22(r8)                     IF  ID  EX  ME  WB
#   Cycle                   0   1   2   3   4   5   6   7   8
```

Uppercase is used to indicate parts of an instruction that could vary. For example, the `ADDi` could have been a `SUBi` but it could not have been an `ADD`.

To determine the value for the fill-in block, `0x22`, one has to know that `r1` holds the target address of the `jalr`. To determine the immediate for the second instruction, `ADDi`, one has to know that `jalr` will write `r6` with the return address.

The last instruction has to be a store because no other instruction (extensively covered in class) uses an immediate, uses an rt register value, and uses the EX stage.

Problem 2: The VAX `locc` instruction (from Homework 3) appears below, along with its encoding (taken from the Homework 3 solution).[20 pts]

```
 locc #65, r2, (r3)

Instruction Encoding:

 -opcode-    -- 1st operand ----      -- 2nd op -    -- 3rd op -
  locc        imm   PC*    immed       reg   r2       reg-d r3
              mode         value       mode           mode
  0x3a        0x8   0xf    0x41        0x5   0x2      0x6   0x3   <- Encoded value.
 7     0     7   4 3   0   7   0      7   4 3   0    7   4 3   0  <- Bit position.
```

(*a*) A MIPS implementation can easily retrieve its two source register values in one clock cycle.

☑ What about the VAX instruction formats makes one-cycle source register value retrieval more difficult in a VAX implementation (without lowering clock frequency)? Consider instructions with at most two source register operands.

In MIPS formats the source register fields are always in the same place so the corresponding instruction register bits can be hard wired to the register file read ports. In VAX the position varies. In the example above the second source register field is in the fourth byte, but if the first operand used register addressing the second source register field would be in the third byte. Therefore one has to partially decode the instruction before one can determine where the register bits are, and so both the decode and the path through the multiplexer add to the cycle time.

(*b*) The constant 65 (`0x41`) is encoded in immediate mode, taking a total of two bytes, rather than literal mode, which would take only one byte if 65 were small enough for literal mode.

☑ Given the way VAX encodes operands one might expect the maximum literal size to be only four bits. Why?

Because the mode field is four bits, so there is only four bits remaining. (The four remaining bits are normally used for the register number.)

☑ In fact, the maximum literal size is six bits. How is that accomplished? (If you don't know or remember the details then make up something reasonable.)

Literal mode is specified by setting the first two mode bits to zero. This consumes four modes but provides for a six-bit literal.

Problem 3: Under SPECcpu2006 base rules at most four compiler optimization switches can be used per language. [15 pts]

☑ What is the rationale for that restriction?

The base scores are supposed to reflect normal effort. It's too vague to stipulate something like *valid compiler optimization switches shall be those that can be determined using normal effort.* With such a rule there would be endless arguments over whether the effort to determine a particular set of switches was normal or not. The four-switch rule, though arbitrary, is specific.

Suppose a tester would like to use five options:

```
cc -O4 --optimization-a --optimization-b --optimization-c --optimization-d
```

The tester modifies the compiler by combining options `a` and `b`, the modified compiler is made available as a product. Now compilation can be done consistent with the rules:

```
cc -O4 --optimization-ab --optimization-c --optimization-d
```

☑ Should that be considered cheating? Explain why or why not.

It should be considered cheating because it renders the four-switch rule irrelevant. The impetus for combining the switches was to get performance on SPECcpu benchmarks, so it's probably not the kind of switch combination that would make sense for other uses otherwise it would already be present.

Problem 4: Answer each question below.[15 pts]

(*a*) In MIPS the branch instruction uses a 16-bit immediate to specify a displacement target, and the `jal` instruction uses a 26-bit immediate. Why does it make sense to choose a coding for `jal` that has a larger immediate?

Branches are typically used for loops and if/else constructs. In both cases the target will be within a procedure. The `jal` instruction is usually used for a procedure call in which case the target will be in another procedure. Therefore `jal` jumps much further and so a coding with a larger immediate makes sense.

Grading Note: The question originally asked *Why does it make sense for* `jal` *to have a larger immediate?* The answer above would be considered correct for that question. The following answer is correct for the original question but not the one appearing further above: Branch instructions can specify as many as two registers for a condition, so there is less space for an immediate.

(*b*) An ISA should be designed to support decades of implementations but invariably ISA designers are biased towards a first implementation at the expense of later ones. A branch delay slot (as present in MIPS and SPARC, for example) is a good example of such a phenomenon.

☑ Explain why the branch delay slot is a good example of a shortsighted ISA feature.

In the classic five-stage MIPS implementation a branch target and direction can be determined (resolved) in the cycle before they are needed, this perfect timing is due to the delay slot. In superscalar implementations and those that are more deeply pipelined the branch would be resolved after the target was needed and so the delay slot provides less of a benefit, while complicating the design. That is, with a classic five-stage implementation there is no branch penalty because of a delay slot. In a superscalar or more deeply pipelined implementation there is a branch penalty (the squashed wrong-path instructions) despite the delay slot, and because of the delay slot control logic is more complex.