Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 28 March 2007, 11:40–12:30 CDT

Problem 1		(40 pts)
Problem 2		(30 pts)
Problem 3		(30 pts)
Exam Total		(100 pts)

Alias 0x501f

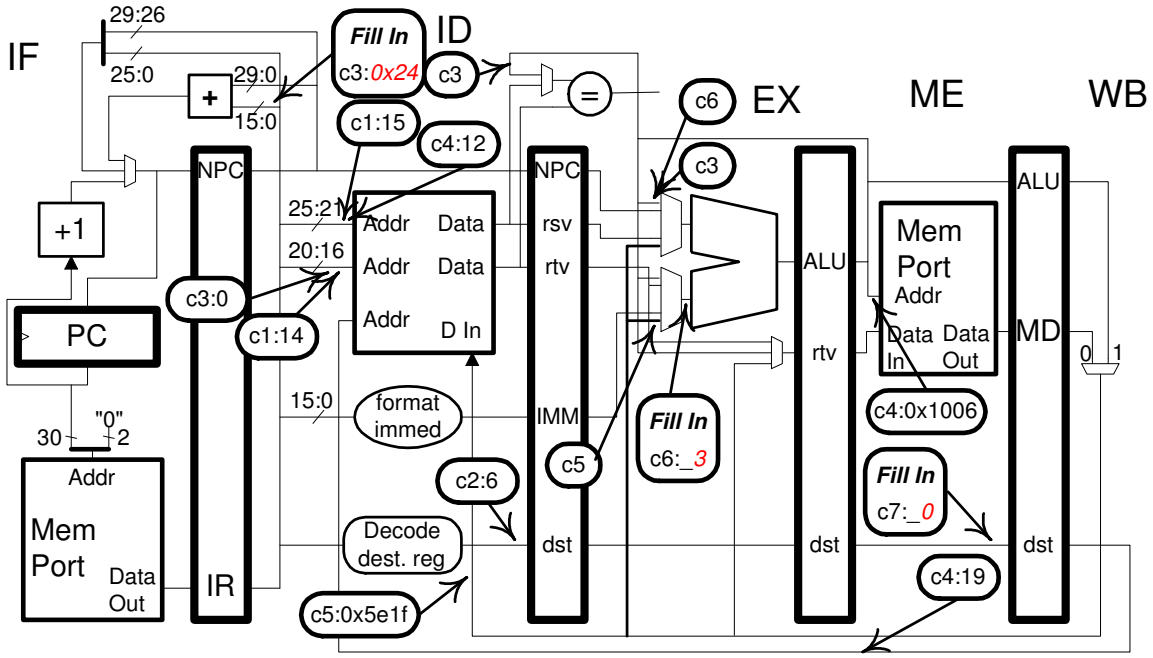
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c1:15 indicates that at cycle 1 the wire will hold a 15. 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 last instruction and the address of two instructions have been provided. [40 pts]

☑ Finish a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

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




```
# Solution Below
#
     #  Cycle               0  1  2  3  4  5  6  7  8
0x1000  xor r19, r15, r14   IF ID EX ME WB
0x1004  lhu r6, 0x5e1f(r19)    IF ID EX ME WB
0x1008  bne r19, r0, TARG         IF ID EX ME WB
0x100c  or r4, r12, r6               IF ID EX ME WB
0x1030  sb  r17, 3(r4)                  IF ID EX ME WB
     #  Cycle               0  1  2  3  4  5  6  7  8
```

Instruction at 0x1000: The c1:15 and c1:14 directly provide the two source registers and c4:19 directly provides the destination register. Because it uses an rt register and writes back it must be a format R instruction, though it could be almost any arithmetic or logical format R instruction (such as `add`, `sub`, `slt`, `or`). Of these `xor` was chosen arbitrarily.

Instruction at 0x1004: The destination register, **r6**, is determined directly by the c2:6. The c3 in EX indicates that this instruction bypasses its rs register value from the previous instruction and so the rs register of this instruction must be the same as the destination of the previous instructions, **r19**. The c4:0x1006 indicates that this is a memory instruction of size either byte or half, and since it writes back it must be some kind of a load. The size of loaded value, c5:0x5elf, rules out a **lb** or **lbh**, so the instruction must be a **lh** or a **lhu**. Peculiarly, the address being loaded from is actually the two least significant bytes of the instruction itself and so the loaded value provides the instruction's immediate field value: **0x5elf**. (Determining this instruction's immediate value was probably the subtlest part of this problem.)

Instruction at 0x1008: The ID-stage c3 indicates that this is a branch, and that the rs register is **r19**. The c3:0 indicates an rt register of 0. The last instruction has address **0x1030** and so the branch must have been taken. Because the previous instruction loaded from address **0x1006** the value in register **r19** could not have been zero, and so the branch must be **bne**. The value for the Fill In: c3: is the number of instructions to skip, the target address minus the delay slot address: **0x1030 - 0x100c = 0x24** or for those proud of their ten fingers, 36.

Instruction at 0x100c: The c4:12 directly indicates the rs register. The bypass indicated by the EX-stage c6 indicates that this instruction's destination register value is bypassed to the following one and so this instruction must write register **r4**. The c5 in the EX stage indicates that this instruction reads an rt register, since it also writes a register it must be format R. As with the first instruction there is no way to tell which one it is, **or** is an arbitrary choice.

Instruction at 0x1030: This instruction itself is given, but there are the fill-in blocks in EX and ME. The c6: fill-in block points to the lower input of the ALU when this instruction is in EX. For stores (and also loads), the lower input is the immediate value, which in this case is 3. The c7: fill-in block points to the destination register number. This instruction is a store, since stores don't write registers the destination register number is 0. (Register **r17** specifies the value to write to memory.)

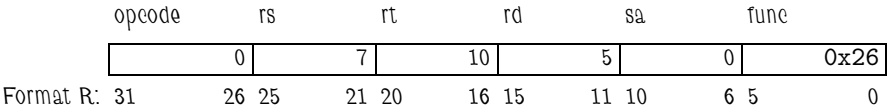
Problem 2: Answer each question below.

(*a*) Show the encoding of the MIPS `xor` instruction below. In particular, show the name of each field in the encoded instruction, the fields' bit positions, and the fields' values. If you don't know the value of a field (there's one or two you're not expected to know) make up a value and label it "made up." *Hint: It is not cheating to look at the diagram for Problem 1.*

```
xor r5, r7, r10
```

☑ [10 pts] Encoding showing field names, bit positions, and values.

Fields shown below. Only the `func` field can have a value marked "made up." One was expected to know that for common format R instructions the `opcode` field value would be zero.



(*b*) Unlike some other benchmark suites, in SPECcpu the tester compiles the benchmarks (rather than having SPEC provide the benchmarks already compiled). [10 pts]

☑ How does this difference make SPECcpu valuable to those in the area of computer architecture?

Those in the area of computer architecture are interested in the performance potential of new designs. Because they are usually testing products they would like to sell, it is in the testers' interests to get results reflecting the full potential of their systems so they will select compilers and other build tools appropriate for the new design and make the best use of these tools.

If SPEC were to provide compiled (built) benchmarks then they might be compiled for some average system, and so would not make full use of a new designs' special features. If a new design implemented a new ISA then SPEC binaries would not run at all, which of course would disappoint the computer architecture research community.

☑ Why might this difference make SPECcpu less valuable to those looking for the fastest computer to run their favorite game?

SPEC shows the performance of a program compiled specially for a particular machine. Games are compiled for some typical machine. If you have a system with some uncommon special feature, the SPECcpu benchmarks might show performance when that feature is used, but games, intended for a broad audience, might not be compiled for that uncommon feature. So one might get lower performance on a game than would be led to expect by looking at similar SPECcpu benchmark numbers.

(*c*) A company is considering adding a BCD data type to their new ISA. An analysis of a suite of Cobol programs, widely used by their customers, shows that the ISA's BCD data type would be extensively used. [10 pts]

☑ What more does the company need to know to make the decision?

The company needs to know how much faster a system would be with the new data type.

If the hardware didn't have the data type then BCD arithmetic would have to be done in software. That would be slower, but would only have a significant impact if a large amount of BCD arithmetic were performed.

Note that issues such as precision or the ability to represent decimal fractions in binary are irrelevant because BCD would be used whether or not the data type were present. If not present, numbers would still be represented in BCD but rather than using, say, a BCD add instruction they'd use a whole bunch of shift, and, and add instructions.

Problem 3: Answer each question below.

(*a*) The MIPS code below loads a character from memory and places it in bit positions 15:8 of register `r3`. [10 pts]

```
lbu r1, 0(r2)   # Note: r2 can be any address.
sll r1, r1, 8
and r3, r3, r4  # r4 = 0xffff00ff
or  r3, r3, r1
```

☑ Explain how the operation performed by this code is similar to, and differs from, the operation performed by `lwl` and `lwr` (from Homework 1). *Note: the phrase "the operation performed by this code" was not present on the original exam.*

Similar: loaded value shifted and combined with existing contents. Difference: Can place any byte in a specific position.

☑ Suppose the operation performed by the code above is something that important benchmarks do often. **Given that MIPS already has `lwl` and `lwr`** is it worthwhile adding an instruction that performs the same operation as the code above? Explain, stating any necessary assumptions or made-up data.

The problem indicates that the operation performed by the code is important and done often. It should also be clear that a machine instruction could perform the operation faster since three of the four instructions it would replace just drop a value in a particular bit range.

Whether it is worthwhile to add this instruction then depends upon how much it would cost. The problem clearly states that `lwl` and `lwr` are already present. Since the hardware to implement these instructions can load something from memory and place it in parts of the destination register without changing the register's other contents, the additional hardware needed for the new instruction would be minimal. And for that reason it would be worthwhile.

Grading Note: Very few saw the point of this question.

(*b*) The first code fragment below, standard MIPS, uses a `slt` to perform a comparison for a branch. The second uses a proposed MIPS branch instruction that does the comparison itself and as a result the target is fetched one cycle sooner. The first execution is on *sImp*, an implementation of standard MIPS, the second execution is on *bImp*, an implementation of the proposed MIPS; bImp includes `blt` but is otherwise identical to sImp. [10 pts]

```
 # Standard MIPS  Cyc: 0  1  2  3  4  5  6  7
 slt r1, r2, r3        IF ID EX ME WB
 bne r1, r0,  TARG        IF ID EX ME WB
 nop                         IF ID EX ME WB
TARG: xor r4, r5, r6            IF ID EX ME WB

 # Proposed MIPS  Cyc: 0  1  2  3  4  5  6  7
 blt r2, r3, TARG      IF ID EX ME WB
 nop                      IF ID EX ME WB
TARG: xor r4, r5, r6         IF ID EX ME WB
```

☑ Why might the clock frequency of bImp be lower than sImp?

The hardware needs to know whether the branch is taken by the end of its ID stage. In sImp the comparison used for the branch starts at the beginning of the branch's ID stage (when `slt` is in EX). In bImp the comparison is also done when the branch is in ID, but it starts only after the registers are retrieved from the register file so there is less time to do so. If comparison takes long enough then bImp's clock frequency would be forced to be lower.

☑ How does knowing the percentage of branches in a program help determine if bImp can run the program faster than sImp (when compiled for the respective implementation)?

With bImp there would be fewer instructions, and if the clock frequency were identical it would run faster. But if bImp had a lower clock frequency then the only way for it to be faster than sImp would be bImp programs to be small enough to overcome the clock frequency disadvantage. The reduction in the number of instructions is based on the number of `slt` / branch pairs, so knowing that can help one estimate performance.

(*c*) All a compiler needs to know about the target (the implementation being compiled for) is its ISA. However, if the compiler also knows the implementation it can produce faster code. [10 pts]

☑ What's wrong with the following statement: *If the compiler also knows the target implementation it can make better register assignment decisions since it knows the exact number of registers available.*

The exact number of registers is something specified in the ISA, not the implementation.

☑ How can the compiler produce better code knowing operation latencies, such as the time needed for a `mul` instruction?

With a knowledge of operation latencies the compiler could choose the best code for an operation that can be performed several ways. For example, to multiply by a small constant, say 6, one might use an integer multiply or one could use a series of shifts and adds. Without knowing operation latencies one can't tell which method is faster.

Scheduling (rearranging instructions) is another thing the compiler can do better knowing operation latencies. (Scheduling is not something covered before the exam and so that was not expected as an answer.)