Name Solution

Computer Architecture
EE 4720
Midterm Examination

Monday, 29 March 2004, 13:40–14:30 CST

Problem 1		(40 pts)
Problem 2		(25 pts)
Problem 3		(35 pts)
Exam Total		(100 pts)

Alias Hazardous Data Dependency

Good Luck!

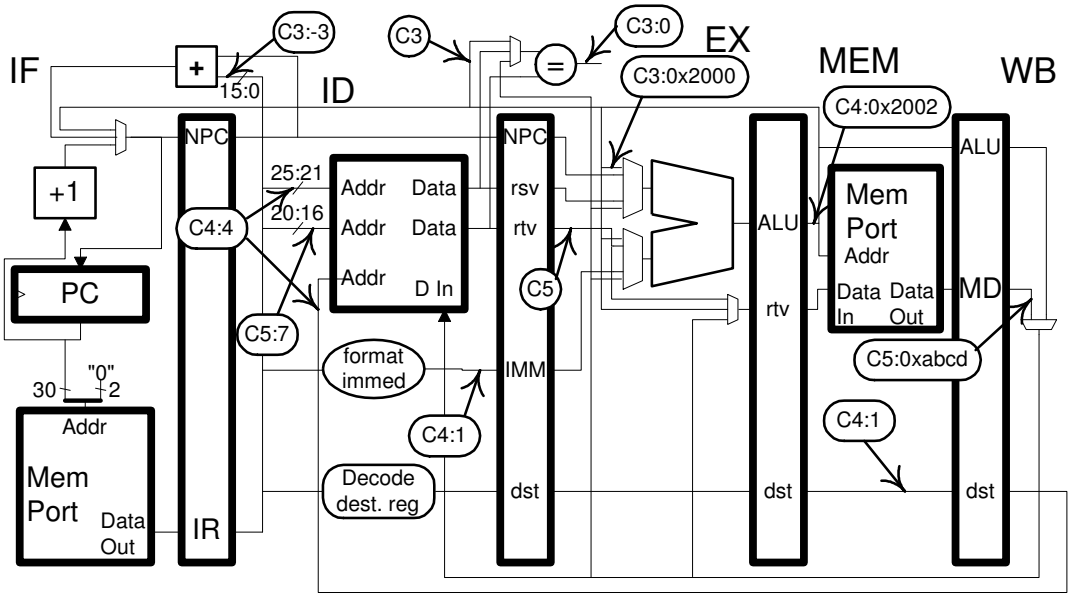
Problem 1: In the diagram below some wires are labeled with cycle numbers and values that will then be present. For example, C3:0 indicates that at cycle 3 the pointed-to wire will hold a 0. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. There are no stalls during the execution of the code. The first instruction (`or`) is shown (but don't forget to add the registers). [40 pts]

☑ Write a program consistent with these labels.

☑ Show the address of every instruction.

☑ Show every register number that can be determined and use `r10`, `r11`, etc. for other register numbers.

☑ Show the exact instruction (they can all be determined). For example, not just a load, but a load ____.




```
# Solution:

#                Cycle:  0   1   2   3   4   5   6   7   8
0x1000: or  r4, r10, r7   IF  ID  EX  ME  WB
0x1004: lhu r1, 2(r4)         IF  ID  EX  ME  WB
0x1008: bne r4, r5 TARG           IF  ID  EX  ME  WB
0x100c: sb  r11, 1(r4)                IF  ID  EX  ME  WB
0x1000: or  r4, r10, r7                   IF  ID  EX  ME  WB
#                Cycle:  0   1   2   3   4   5   6   7   8
```

Detailed discussion on next page.

Register numbers can be determined from boxes such as C4:4 and by the use of bypass paths (which tell you that the two registers are the same) such as C3:0x2000 (this indicates that **rs** register number of the second instruction is the same as the destination of the first; the **0x2000** provides additional information).

Each instruction can be determined from a variety of clues. The first instruction is given (**or**), the others are determined as follows:

lhu: The use of MD in cycle 5 tells us it's some kind of load. The effective address, C5:0x2002, is a multiple of two but not a multiple of 4, so it cannot be a load word. The loaded data, **0xabcd**, spans 16 bits so it can't be a load byte. It can't be an **lh** because the bit at position 15 is 1 and would have been sign extended (the data would be **0xffffabcd**) if it were. Therefore it must be a **lhu** (load half unsigned).

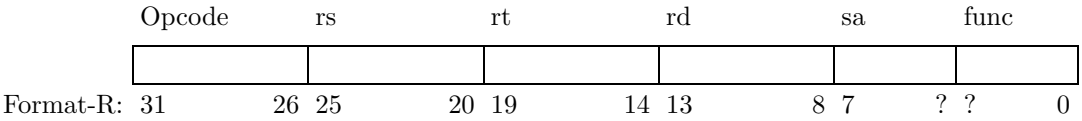
bne: The C3:-3 tells us it is a branch and that the branch is taken. (Because the problem description states that labeled wires are being used.) The C3:0 tells us it is either **bne** or **beq** (it can't be **bgtz**, etc. because those branches do not test equality or use the **rt** register value). The C3:0 also tells us that the two operands are not equal, since the branch is taken it must be a **bne** (branch not equal).

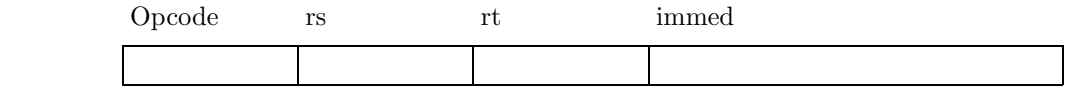
sb: This instruction uses both an immediate (C4:1) and the **rt** register value (C5) beyond the ID stage, so it must be a store. (Branches use the **rt** register value and an immediate in ID, not beyond it.) The store address is **1(r4)**, while the **lhu** is **2(r4)**, since **r4** does not change the store address is not a multiple of 2 and so it can only be a **sb** (store byte).

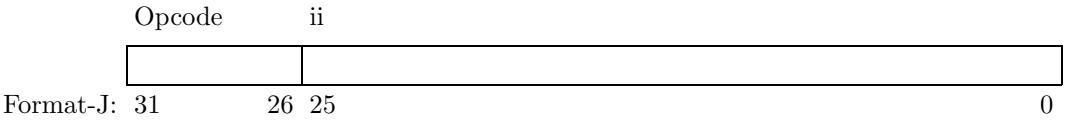
or: The C3:-3 tells us that the branch is branching up three instructions (the delay slot instruction, **sb**, is zero), so we know the branch target is the first instruction, an **or**. So the last instruction is the same as the first.

Problem 2: Consider a new ISA, F-MIPS, similar to MIPS-I except that it has 64 rather than 32 general purpose registers. F-MIPS has R, I, and J instruction formats like MIPS-I but with modifications to handle the larger number of registers. A goal of F-MIPS is to have all of MIPS-I instructions.

Possible instruction formats are shown below. Some details of Format R are omitted and are the subject of the first question.







☑ [6 pts] Describe a problem with Format-R F-MIPS instructions using the Format R shown above.

Field sa or func (or both) would have to be shrunk. If sa were shrunk there would be no way to specify all shifts. If func were shrunk it might not be possible to code all instructions.

☑ [7 pts] Fix the problem in Format R (show the changes in the illustration above). Explain the impact (this is important) on the coding of F-MIPS instructions.

Keep the func field six bits and change the sa field to a reserved-for-future-use field. The immediate shift instructions can be encoded in Format I or the rs field can be used for the shift amount.

☑ [6 pts] Explain an impact on typical F-MIPS Format-I instructions (compared to their MIPS-I counterparts) that would not apply to Format-R instructions.

The immediate is smaller so branches can't branch as far and code that would require one immediate instruction in MIPS-I might require two instructions in F-MIPS.

☑ [6 pts] Why is something going to have to be done about `lui`? Describe a new version of `lui`, possibly using a new format, that would fix the problem. *Hint: That's load upper immediate.*

There should be some way to load a 32-bit immediate in two instructions, with the immediate now at 14-bits that can't be done with two Format I instructions. Code frequently needs to load a 32-bit immediate and so a solution that requires three or more instructions would have a significant impact on performance and code size. A two-instruction solution would be to have a new format, say Format Ib in which the immediate field extends into rt and rs is used for the destination.

Problem 3: Answer each question below.

(*a*) A company has to choose between developing two new implementations of their ISA. Implementation A would have a peak (result) score of 2200 and a base (baseline) score of 2000 on the SPEC CINT2000 benchmarks. Implementation B would have a peak (result) score of 2150 and a base score of 2100 on the benchmarks.

☑ [0 pts] Which implementation should the company choose? *Hint: Either answer is correct.*

Implementation A. *Implementation B.* (Both answers are discussed below.)

☑ [9 pts] Why? Your reason should say something about the difference between the peak and base scores and about the company's customers.

The base numbers are obtained by compiling the benchmarks with ordinary optimizations, the peak numbers are obtained by compiling the benchmarks with great skill and effort to get the best results.

Implementation A: This implementation gives the better peak performance, which is appropriate for our customers since they compile the code for each implementation and their conscientious programmers are experts at optimization.

Implementation B: This implementation gives the better base performance. Our customers do want high performance but not enough to choose programmers skilled in performance tuning over those who can get code out the door quickly.

(*b*) Should a BCD data type be added to a modern general-purpose ISA?

☑ [8 pts] Explain why or why not, using the criteria discussed in class for adding data types to an ISA. (Discuss specific features of BCD, don't give an answer that could apply to any data type.)

BCD represents numbers as a string of decimal digits, encoding each digit using four bits. Any fixed point decimal number (within range) can be represented exactly in BCD, such as 0.3. Such numbers cannot be represented exactly in fixed- or floating-point binary. For example, 0.3 is $0.0100110011001100\overline{1100}_2$.

Due in part to the inexact representation, certain computations on certain arithmetic hardware would produce results that are off by a small amount. The problem might occur with financial calculations and of course is considered unacceptable. An early solution to that problem was to include a BCD data type and have BCD arithmetic instructions.

In the IEEE 754 floating point standard rounding is precisely specified and the default rounding mode was chosen so rounding results in the expected answer. Most modern systems implement this standard and so financial computations can use floating-point arithmetic without fear of rounding errors.

Note: Both answers below were marked correct.

No, because few modern computer languages use BCD and so there is no need to support it in hardware. Compilers for languages that still use BCD can use software to perform BCD computations, so it will still be possible to run old code.

Yes, because there is still alot of COBOL code (smoking as killed off or incapacitated most, but not all, COBOL programmers) and there are other languages that still use BCD and their code must be supported.

(*c*) Re-write the SPARC code fragment below in MIPS-I. Use as few instructions as possible. [9 pts]

The `subcc` instructions performs a subtraction and sets the condition-code (CC) register bits based on the result. It also writes an ordinary destination register, in this case `r0`. The branch (and a few other) instructions read the CC register to determine if the branch (or other action) should be taken. The `be` instruction means branch if the result of the last cc instruction is equal to zero (assuming a subcc that can be interpreted as meaning branch if the operands are equal). Since the `subcc` writes the zero register there is no need for the MIPS code to include a subtract instruction, it can just compare the two registers.

```
! Notes: r0 is the zero register;  destination is rightmost register;
!        be means branch if equal.  Use the same register names.

! Note: Comments in SPARC code are part of solution.

subcc %r1, %r2, %r0   ! r0 = r1 - r2, set CC register.
add   %r3, %r4, %r5
be    TARG            ! Branch if result (from last cc insn) is zero.
xor   %r6, 10, %r6    ! Delay slot instruction.
```

```
# Solution shown below.

add r5, r3, r4
beq r1, r2, TARG
xori r6, r6, 10
```

(*d*) The code below includes a hypothetical MIPS predicated instruction. Re-write the code using real MIPS instructions. [9 pts]

The `add` is the predicated instruction. It is executed if `r1` is nonzero. In the equivalent code a branch instruction checks if `r1` is nonzero, if not the `add` instruction is skipped.

```
      sub r3, r6, r7
 (r1) add r2, r3, r4
      xor r5, r8, r7
```

```
# Solution below.

      beq r1, r0 SKIP
      sub r3, r6, r7
      add r2, r3, r4
SKIP:
      xor r5, r8, r7
```