Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 27 October 2006, 12:40–13:30 CDT

Problem 1		(50 pts)
Problem 2		(20 pts)
Problem 3		(30 pts)
Exam Total		(100 pts)

Alias 0x1010

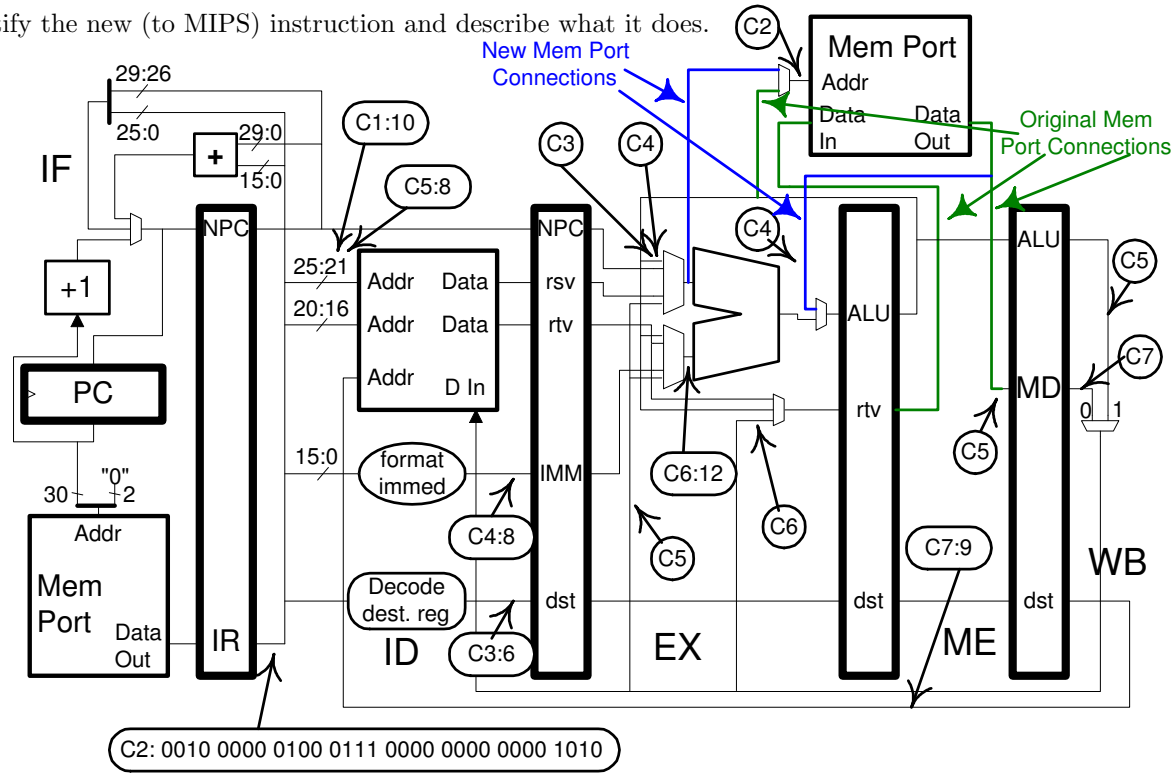
Good Luck!

Problem 1: In the MIPS implementation below the data memory port is in an unusual position that *avoids a stall* and allows the implementation of *a new (to MIPS but not CISC ISAs) instruction.* Some wires are labeled with cycle numbers and values that will then be present. For example, C1:10 indicates that at cycle 1 the wire will hold a 10. 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. [50 pts]

☑ Write a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Identify the new (to MIPS) instruction and describe what it does.




```
# SOLUTION.  See next page for discussion.
#
# Stage labels: eX: EX stage used for arithmetic / logical computation.
#               eA: EX stage used for effective address computation.
#               eM: EX stage used for memory access.
#               me: Memory stage not used.
#               mM: Memory stage used.
#
                    0  1  2  3  4  5  6  7  8
I1: lW   r2, 0(r10) IF ID eM me WB
I2: addI r7, r2, 4     IF ID eX me WB
I3: lW   r6, @(r7)        IF ID eM mM WB
I4: lW   r9, 8(r7)           IF ID eA mM WB
I5: sW   r6, 12(r8)             IF ID eA mM WB
                    0  1  2  3  4  5  6  7  8
```

The implementation diagram shows the data memory port moved from its home in ME to between the EX and ME stages. The connections highlighted in green are also present in the usual five-stage pipeline used in class, and so the memory port can be used by instructions in ME in the usual way. (The highlighting of differences and similarities is only present in the solution.) The new connections, highlighted in blue, allow an instruction in EX to use the memory port, with the address taken from the upper ALU input and the loaded data placed in the EX/ME.ALU latch. Each instruction is discussed below.

I1: The C2 at the memory port can be for an instruction in EX or ME, but in cycle 2 there is no listed instruction in ME and so it must be the instruction in EX, which is I1, and so I1 must be a memory instruction. The C1:10 indicates that its rs register is **r10** and because it's not using the ALU to compute its effective address the immediate must be zero (otherwise the control logic would have it use the memory port when in ME). The C3 at the ALU input indicates a dependence with the following instruction, I2, and so I1 must be some kind of load, the dependence also fixes the destination register to **r2** since that's what the rs register of I2 will turn out to be.

I2: The C2:0010 0000... provides the entire instruction. Those who happen to know that 8 is the opcode for **addi** could get the whole instruction here, for the rest of us there are hints. The C5 in WB indicates that I2 is writing back a value from the ALU, so it must be an arithmetic or logical instruction. If I2 were a type R instruction the rd register would be zero, but since only useful wires are labeled the destination register must be non-zero and so I2 is type I. From the IR value it's easy to find that **rs** is **r2**, **rt** (the destination) is **r7**, and the immediate is 10. It could be any arithmetic or logical instruction and so **addi** is possible but not certain.

I3: The C3:6 provides the destination register, **r6**. The C4 at the ALU output indicates that this instruction uses the memory port when it's in EX. The C5 in ME indicates that *it also uses the memory port when it's in ME*. How odd! This must be the new instruction since no existing instruction uses the data memory port twice. The first use of the memory port would use the **rs** register as an address, the second use would use the loaded value as the address, so I3 must be a memory indirect load.

I4: The C4:8 in ID limits I4 to a type-I instruction. The C5 in EX shows a bypass from I2 which means the **rs** register is **r7** (there is no way the bypass could be to the **rt** register in a type I ALU instruction). The C7 in WB reveals that this instruction used the memory port in ME for a load, so it must be an ordinary load instruction.

I5: C5:8 in ID gives the rs register, **r8**. It's a store since it uses the store value bypass path, C6 in EX, that connection also gives us the rt register: **r6**. The offset is provided by C6:12 in EX.

Instruction I1 is an ordinary load instruction but executes in EX instead of ME, and in doing so avoids stalling I2. The control logic presumably determined that the immediate was zero and so had it execute in EX since it didn't need the ALU. I1 could also be a new load instruction, one that does not have an offset. Unless the immediate field were used for something else there would be no benefit in that since, as shown above, an ordinary load instruction with a zero offset can use the memory port when in EX.

Instruction I3 is a memory indirect load, an instruction that MIPS does not have. And probably won't. In the original implementation the memory port gets the address in the beginning of the cycle, here it must wait for the signal to pass through two multiplexers; in the original implementation the output of the memory port goes straight to the pipeline latch, here it must go through a multiplexer, for a total of three added multiplexers for what is probably already the critical path. Also, if a load from the memory port required more than one cycle then several stages would have to be added to have two loads per instruction.

Problem 2: Answer each question below.

(*a*) Both the SPARC `subcc` and MIPS `slt` instructions below determine if the contents of r1 (g1) is less than r2 (g2).

```
! SPARC
subcc g1, g2, g3

# MIPS
slt r3, r1, r2
```

☑ [5 pts] Show where each writes the comparison result, what is written, and how the result is used for a branch.

SPARC: Comparison result is written in the CC register which consists of four bits: NZCV, N is set if result is negative, Z is set if result zero, C if a carry, and V if an overflow. Branch instructions check the CC register.

MIPS: Comparison result written to `r3`, 0 if false, 1 if true. Branch instructions check if register is zero.

☑ [5 pts] Describe two ways in which the SPARC instruction is more powerful.

Advantages:

The cc instructions do useful computation in addition to setting the condition-code register.

A cc instruction sets four condition bits so several different kinds of branches could use them, for example, one branch might be taken if result positive, another if an overflow, etc.

The use of a cc register gives the compiler (or programmer) one more general purpose register to use (between the time of the comparison and the last branch that uses it).

The control logic needed to test the four CC bits can be made faster than the logic needed in a MIPS implementation that must test a pair of 32- or 64-bit registers.

Grading Note: Several (incorrectly) gave the following advantage: instructions can be placed between the cc instruction and the branch. That's wrong because the same is true for MIPS, so it's not an advantage of SPARC over MIPS. (It is an advantage of SPARC and MIPS over those ISAs in which every arithmetic instruction sets condition code bits.)

(*b*) One advantage of variable-length ISAs is that programs are shorter (take up less memory). For each code fragment below show how CISC instructions can reduce code size using the MIPS code below as the starting point of an example: Make up CISC instruction(s) for the code below, show how they might be encoded, and indicate how much smaller the code fragments are with the CISC instructions.

☑ [5 pts] New CISC instruction(s). Encoding. Size Reduction.

```
lui r1, 0x1234
ori r1, r1, 0x5678
add r2, r2, r1

# Solution


# If r1's value not used again:
#
add.rri r2, r2, 0x12345678

Coding:  ! opcode ! operand types ! rd ! rs1 ! immed32 !
Size:      8        6               5    5     32        = 56 bits = 7 bytes


# If r1's value is used again:
#
move.ri r1, 0x12345678
add.rrr r2, r2, r1

Coding:  ! opcode ! operand types ! rd ! immed32 !
Size:      8        3               5    32        = 48 bits = 6 bytes

Coding:  ! opcode ! operand types ! rd ! rs1 ! rs2 !
Size:      8        6               5    5     5   = 29 bits = 4 bytes

Total : 10 bytes.


Original MIPS code: 3 * 4 = 12 bytes
Reduction: 2 bytes (r1 needed) or 5 bytes (r1 not needed)
```

The "operand types" field specifies the types of each operand. That field is set to 6 bits for a 3-operand instruction and 3 bits for a 2-operand instruction, based on a rough guess.

Grading note: The sizes above are rounded up to the nearest byte, that's something most did not do on their solutions.

Some solutions gave size reductions in "lines of code." That's wrong because the problem is asking for the reduction in the amount of memory used by the program. *Lines of code* is a relevant (albeit imperfect) measure for things like the amount of human effort needed to write or modify a program.

☑ [5 pts] New CISC instruction(s). Encoding. Size Reduction.

```
jr r31

# Solution


# Return instruction.
#
return

Coding: ! opcode !
Size:       8      = 1 byte.

Reduction: 3 bytes.


# Jump register instruction.
#
jr r31

Coding: ! opcode ! operand type ! rs !
            8           3         5     = 16 bits = 2 bytes

Reduction: 2 bytes
```

The instruction `jr r31` can be used for procedure returns. Since returns are frequent it would make sense to have a CISC instruction just for that, it would be just one byte. That's the first solution. The second is for a CISC instruction that could use any register for the address. Either solution received full credit.

Grading Notes: In several solutions the CISC instruction encoded the full return address. That won't work because a procedure can have more than one caller, and so a single return address could not be used.

Some solutions mentioned displacement. That's not an issue because the register holds the entire address, there is no need to add the register value to anything.

Problem 3: Answer each question below.

(*a*) The SPECcpu2006 benchmark contains two suites, CINT2006 and CFP2006.

☑ [8 pts] Why are there two suites? What would be the disadvantage of combining them into one suite?

There are two suites because the characteristics of integer and FP programs are very different. If they were combined then it would be harder to find systems that were, say, good at floating point.

(*b*) A company develops a new ISA and some implementations of it. It sells the implementations, but keeps the ISA secret.

☑ [8 pts] What's wrong with that?

Can't program the implementation without the ISA definition. You'd be limited to using the software that the manufacturer supplied, and if that didn't include a compiler then you could not write your own software. (Real Programmers don't use interpreted languages.)

(*c*) Operations on packed operand data types often use saturating arithmetic.

☑ [7 pts] What is it and why is it used? Explain using a typical application for packed-operand instructions.

It's arithmetic that uses the maximum representable value for overflow. (Or minimum, for negative overflow.) It is used because in many packed operand applications overflow is likely and a maximum value would be suitable. In graphics applications, for example, the instruction might be computing a pixel intensity, so using the maximum for an overflow will produce an acceptable result.

(*d*) One reason to not use optimization is to make debugging (using a debugger) easier.

☑ [7 pts] Describe two ways optimization makes debugging more difficult. *Hint: Consider single-stepping through code and printing variable values.*

Instructions can be re-arranged and so single-stepping will jump around code that the user would expect to execute sequentially. Common sub-expression elimination is one optimization that would cause code to be rearranged.

Through dead-code elimination variables may be optimized out so one could not print their values.