# CS 250: Computer Architecture

**Midterm Exam** - October 22, 2012
Closed-book/notes/discussion

TIME: 90 minutes (8:00 PM - 9:30 PM)
LOCATION: PHYS 112

Sample Questions

**1. 10 "True or False" statements. Sample statements are:**

(1) [ F ] 5 bits are sufficient to represent every state of the United States.
Need 6 bits for 50 states

(2) [ T ] A demultiplexor has more output lines than input lines.

(3) [ T ] In both one's complement and sign-and-magnitude schemes, there are two representations for zero.

(4) [ F ] In MIPS, *some* R-type instructions also access the data memory.
No R-type instruction accesses data memory.

(5) [ F ] In MIPS, the 16-bit "address" field of the **beq** instruction is treated as an unsigned integer. It's a signed integer (2's complement)

(6) [ T ] In the simple MIPS processor, the subset of datapath involved in the execution of an instruction can be predicted except the branching instructions.

(7) [ F ] A stack grows from low address to high address.
from high addr. to low addr.

**2. 10 short "Q&A"s. Your answer to each question should have no more than three sentences. Sample questions are:**

(1) Name at least three hardware components inside the processor.

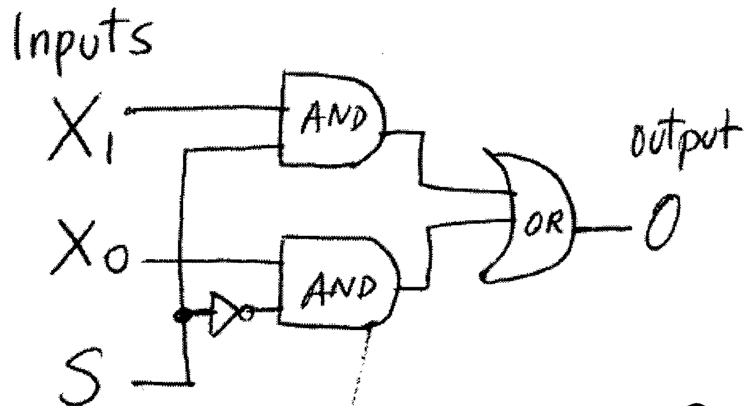
- ALU
- "PC+4" adder
- Register file
- Multiplexors

⋮

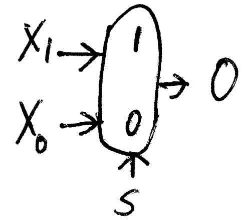
(2) Under two's complement, what is the decimal value of binary number 1111 0000? What if this is a binary number under one's complement?

- Under 2's complement: −16
- Under 1's complement: −15

(3) What does the digital circuit below do?




This is a "2-to-1" multiplexor




(4) Translate the following C statement into MIPS assembly.

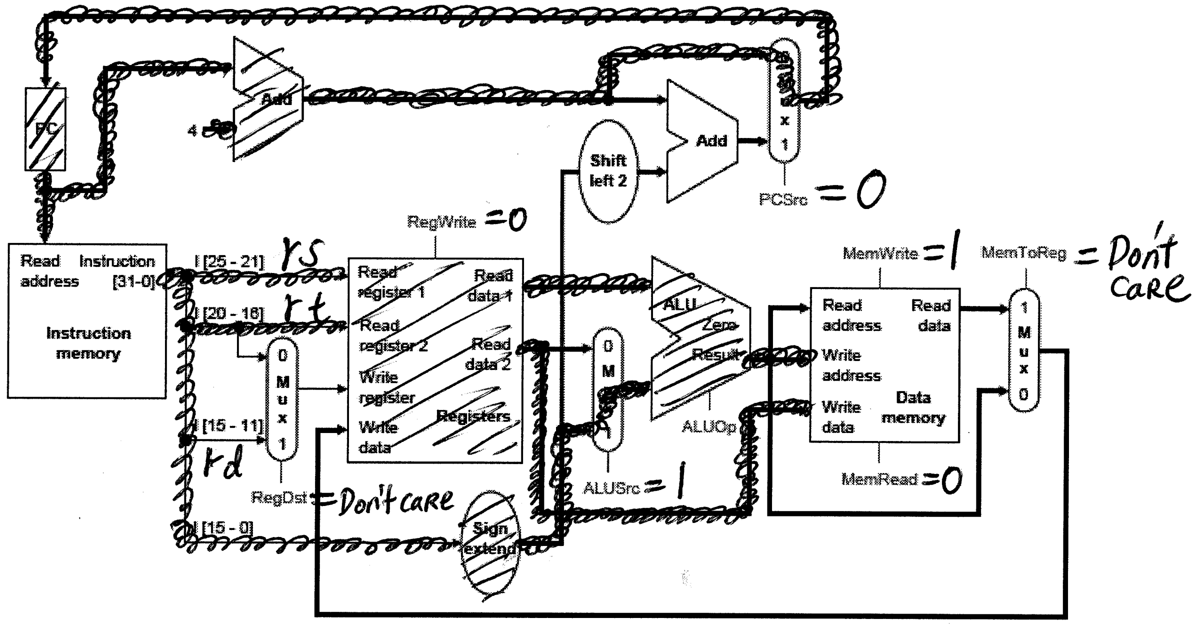
```
If (x == y) {
        i++;
}
j++;
```

You can assume that the values of variables x, y, i, and j are already in registers $1, $2, $10, $20, respectively.

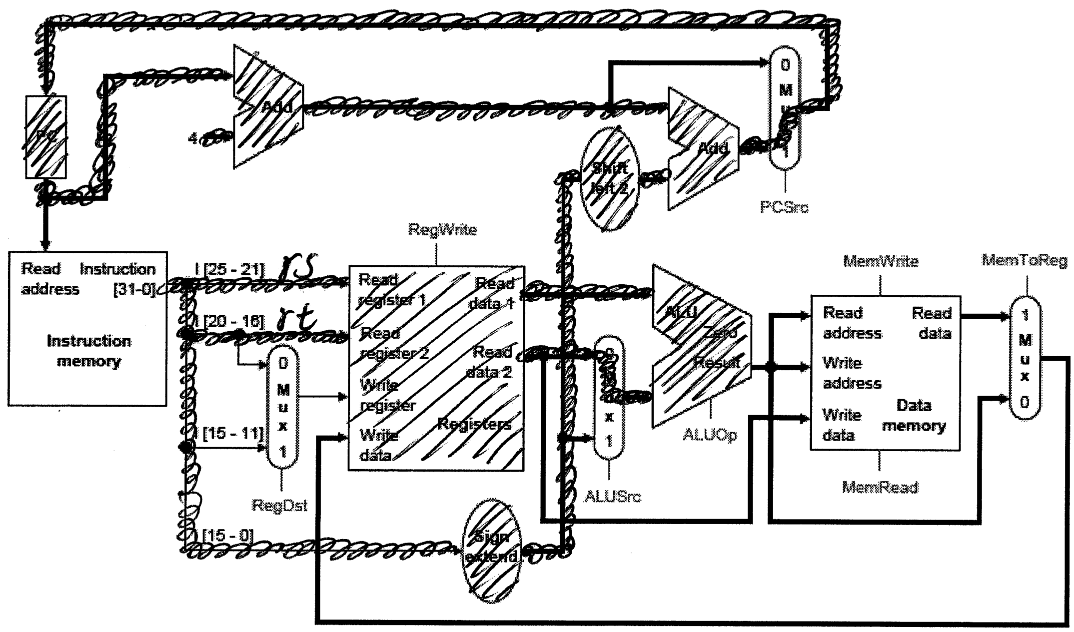
```
       bne $1, $2, label
       addi $10, $10, 1
       sw $10, i($0)
label: addi $20, $20, 1
       sw $20, j($0)
```

memory addresses of variables i and j, respectively.

(5) When the simple, single-cycle MIPS processor executes instruction **`sw $rt, offset($rs)`** (rs: instruction bits 25-21, rt: bits 20-16), what are the values of control signals 'RegDst', 'RegWrite', 'ALUSrc', 'MemRead', 'MemWrite', and 'MemToReg'? Indicate the values in the figure.

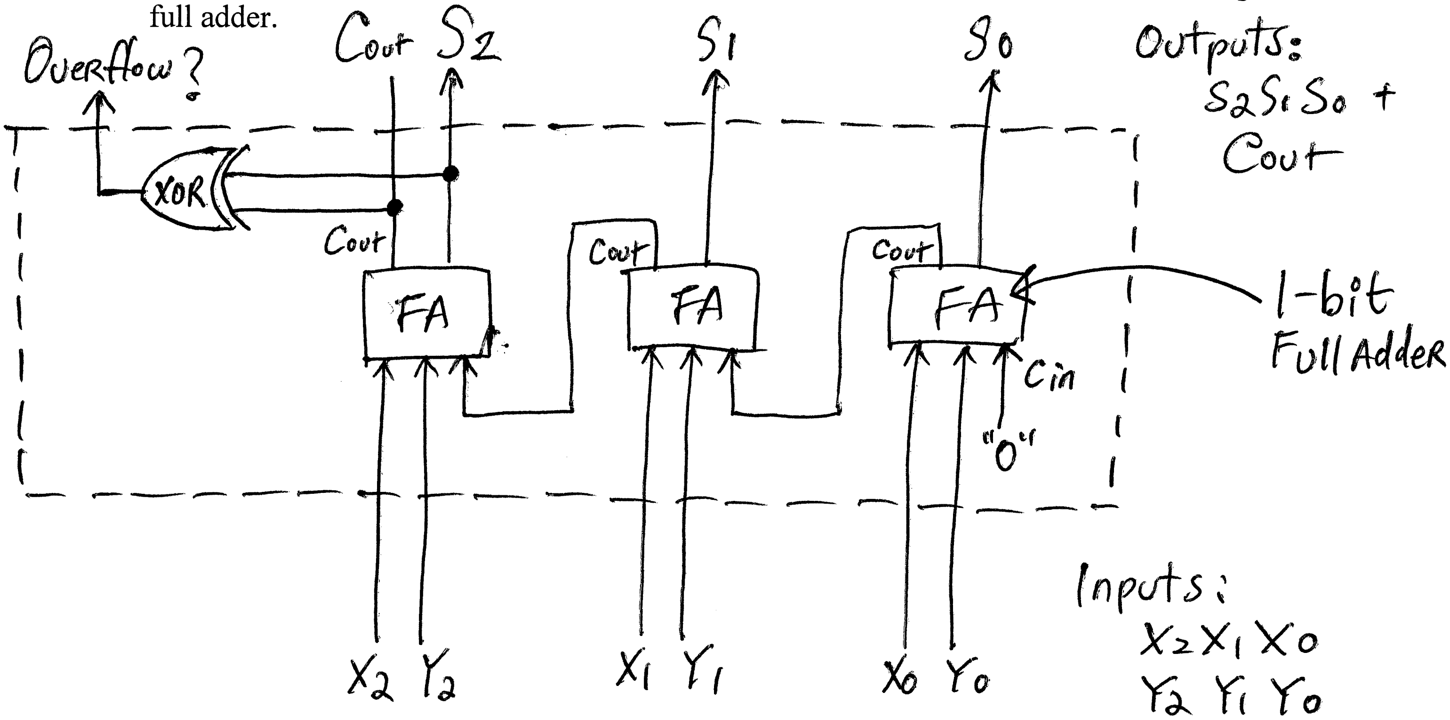



(6) Highlight the subset of the MIPS datapath when executing instruction **`beq $rs, $rt, label`**, assuming that the branch is taken.



**3. One digital circuit design problem** Consider a single-digit *full adder* like the one on pp. 28 of lecture notes Part 1. The full adder takes three inputs: bit X, bit Y, and the *carry-in* bit; and generates two outputs: *sum* of X and Y and the *carry-out* bit.

**(1)** Show how to use three (3) of these single-digit full adders to construct a three-digit full adder.




**(2)** We now use the three-digit full adder constructed in **(1)** to perform *addition* operation between two three-digit signed integers, represented under 2's complement scheme. What is the output of the three-digit full adder when computing $3_{10} + 3_{10}$ ? We call this an "overflow" situation because the sum ($6_{10}$) cannot be represented by three binary digits. Please enhance the three-digit full adder to detect such an overflow. You can show your changes in your diagram for **(1)**.

Results: $S_2S_1S_0 = 110$
$C_{out} = 0$

An overflow occurs iff. Cout and $S_2$ have different values. namely Cout XOR $S_2 = 1$. To detect overflow, Simple add an XOR gate as shown above.

**(3)** In **(2)**, if the two operands of the addition operation are of *opposite* signs (namely one positive and the other one negative), overflow will *not* happen. Briefly explain why.

When two operands are of opposite signs, the value of their sum will always be >>BETWEEN<< the two operands. Hence no overflow will happen.

**4. One MIPS assembly programming (recursive) problem** The Greatest Common Divisor (GCD) of integers x and y is the largest positive integer that divides x and y without a remainder. GCD(x,y) can be computed using the following recursion (assuming that x, y > 0 for simplicity):

$$
\begin{aligned}
GCD(x, y) &= x, && \text{if } x = y;\\
GCD(x, y) &= GCD(x-y, y), && \text{if } x > y;\\
GCD(x, y) &= GCD(x, y-x), && \text{if } x < y.
\end{aligned}
$$

Here is the C code that implements the above recursive function:

```
GCD(int x, int y)   // assume x > 0 and y > 0
{
   if (x = = y) return x;
   else if (x > y)  // case x > y
               return GCD((x-y), y);
         else       // case x < y
               return GCD(x, (y-x));
}
```

A skeleton of the equivalent MIPS assembly code is given on the next page. Your mission is to understand the assembly code skeleton and complete it.

**(1)** Fill in each of the first three (smaller) boxes with *one* missing instruction.

**(2)** Fill in the last (large) box with *a sequence of* instructions to complete the code. And *Comment* each instruction.

See next page.

```
# x is the first argument and has been stored in $a0
# y is the second argument and has been stored in $a1
GCD:   subi $sp, $sp, 12          # create stack frame
       sw   $a0, 0($sp)           # save x
       sw   $a1, 4($sp)           # save y
       sw   $ra, 8($sp)           # save return address

# if x != y, jump to 'rec'
       bne  $a0, $a1, rec

# if x == y, return x
       move $v0, $a0              # $v0 ← x
       addi $sp, $sp, 12          # destroy stack frame
       jr $ra                     # return

# The recursion begins
rec:   bgt  $a0, $a1, xgty        # if x > y, jump to xgty

xlty:  sub  $a1, $a1, $a0         # $a1 ← y-x
       jal GCD                    # call GCD(x, (y-x))
     # after returning from GCD(x, (y-x))
       lw   $a0, 0($sp)           # restore x
       lw   $a1, 4($sp)           # restore y
       lw   $ra, 8($sp)           # restore return address
       addi $sp, $sp, 12          # destroy stack frame
       jr $ra                     # return

xgty:  sub $a0, $a0, $a1          # $a0 ← $a0 - $a1
       jal GCD                    # call GCD (x-y, y)

       lw $a0, 0($sp)             # restore x
       lw $a1, 4($sp)             # restore y
       lw $ra, 8($sp)             # restore return addR.
       addi $sp, $sp, 12          # destroy stack frame
       jr $ra                     # return
```

**5. One problem on simple, single-cycle MIPS processor** Sample problem: The "jump-and-link" instruction (**`jal`**) is used for making function calls in MIPS:

```
jal Label
```

| op | Label |
|---|---|
| op[31-26] | label[25-0] |

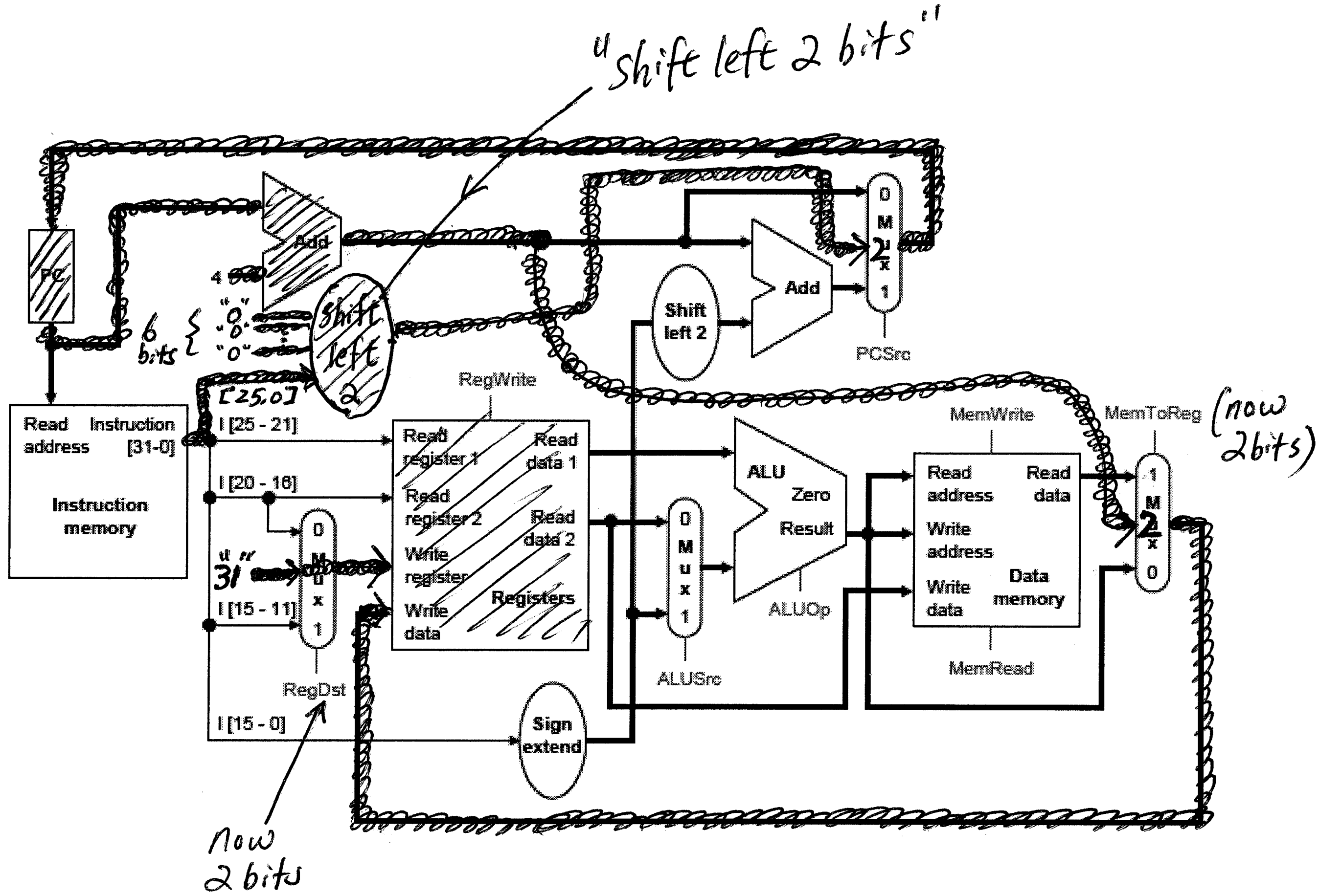
Before the function is called, the return address (PC+4) will be written to \$ra (namely \$31).

**(1)** Modify the MIPS processor diagram on the next page to support the execution of **`jal`**.

**(2)** Highlight the subset of the modified datapath involved in the execution of **`jal`**.

**(3)** In the same situation as in **(2)**, indicate (in the same figure) the values of control signals 'RegDst', 'RegWrite', 'PCSrc', and 'MemToReg'.

See next page.




Q (1), (2): see diagram above

Q (3): RegDst = 2; (decimal)

RegWrite = 1;

PCSrc = 2; (decimal)

MemtoReg = 2; (decimal)

| Name | Fields | | | | | | Comments | Example |
|---|---|---|---|---|---|---|---|---|
| Field size | 31 6 bits 26 | 25 5 bits 21 | 20 5 bits 16 | 15 5 bits 11 | 10 5 bits 6 | 5 6 bits 0 | All MIPS instructions 32 bits | |
| R-format | op | rs | rt | rd | shamt | funct | Arithmetic instruction format | add $rd,$rs,$rt |
| I-format | op | rs | rt | address/immediate | | | Transfer, branch, imm. format | bne $rs,$rt, addR |
| J-format | op | target address | | | | | Jump instruction format | J addR |

**FIGURE 2.26 MIPS instruction formats in this chapter.** Highlighted portions show instruction formats introduced

| Category | Example Instruction | Meaning |
|---|---|---|
| Arithmetic | addi $t0,$t1,100<br>add $t0, $t1, $t2<br>sub $t0, $t1, $t2<br>rem $t0, $t1, $t2<br>div $t0, $t1, $t2 | $t0=$t1+100<br>$t0 = $t1 + $t2<br>$t0 = $t1 – $t2<br>$t0 = $t1 % $t2<br>$t0 = $t1 / $t2 |
| Logical | and $t0, $t1, $t2<br>or $t0, $t1, $t2<br>sll $t0, $t1, $t2<br>srl $t0, $t1, $t2<br>sra $t0, $t1, $t2 | $t0 = $t1 & $t2 (Logical AND)<br>$t0 = $t1 \| $t2 (Logical OR)<br>$t0 = $t1 << $t2 (Shift Left Logical)<br>$t0 = $t1 >> $t2 (Shift Right Logical)<br>$t0 = $t1 >> $t2 (Shift Right Arithmetic) |
| Register Setting | move $t0, $t1<br>li $t0, 100 | $t0 = $t1<br>$t0 = 100 |
| Data Transfer | lw $t0, 100($t1)<br>lb $t0, 100($t1)<br>sw $t0, 100($t1)<br>sb $t0, 100($t1) | $t0 = Mem[100 + $t1] 4 bytes<br>$t0 = Mem[100 + $t1] 1 byte<br>Mem[100 + $t1] = $t0 4 bytes<br>Mem[100 + $t1] = $t0 1 byte |
| Branch | beq $t0, $t1, Label<br>bne $t0, $t1, Label<br>bge $t0, $t1, Label<br>bgt $t0, $t1, Label<br>ble $t0, $t1, Label<br>blt $t0, $t1, Label | if ($t0 = $t1) go to Label<br>if ($t0 ≠ $t1) go to Label<br>if ($t0 ≥ $t1) go to Label<br>if ($t0 > $t1) go to Label<br>if ($t0 ≤ $t1) go to Label<br>if ($t0 < $t1) go to Label |
| Set | slt $t0, $t1, $t2<br>slti $t0, $t1, 100 | if ($t1 < $t2) then $t0 = 1 else $t0 = 0<br>if ($t1 < 100) then $t0 = 1 else $t0 = 0 |
| Jump | j Label<br>jr $ra<br>jal Label | go to Label<br>go to address in $ra<br>$ra = PC + 4; go to Label |

The second source operand of the arithmetic, logical, and branch instructions may be a constant.

**Register Conventions**

The *caller* is responsible for saving any of the following registers that it needs, before invoking a function.

$t0-$t9 $a0-$a3 $v0-$v1

The *callee* is responsible for saving and restoring any of the following registers that it uses.

$s0-$s7 $s8/$fp $sp $ra

**Pointers in C:**

Declarartion: either *char *char_ptr* -or- *char char_array[]* for *char c*

Dereference: *c = c_array[i]* -or- *c =*c_pointer*

Take address of: *c_pointer = &c*