

University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2004

John Kubiawicz

Prerequisite Quiz
February 2, 2004
CS152 Computer Architecture and Engineering

This prerequisite quiz will be used in determining class admissions. The use of notes is not allowed during this quiz. Good Luck!

Your Name:	Peter Perfect
SID Number:	Answer Key
Discussion Section:	

1	
2	
3	
4	
5	
Total	

[This page left for π]

3.141592653589793238462643383279502884197169399375105820974944

- 1) Assume that we have a 16 bit system that uses signed, two's-complement integers. Perform the following conversions:

-21_{10} to base 2:

1111 1111 1110 1011

$FF2F_{16}$ to base 10:

-209

65_{10} to base 16:

41

$(1111111110110100)_2$ to base 10:

-76

$FEC3_{16}$ to base 8:

177303 or -475. Note that we accepted both answers since each digit in base 8 represents 3 bits, and thus we can only represent a 15 bit number, not 16. If we assumed the 16th and 17th bit to be both 00 or 11, we can get two different answers. 177303 gives the accurate bit representation of $FEC3_{16}$, but -475 gives the correct negative value.

257_{10} to base 2:

0000 0001 0000 0001

2) Suppose that we start with the following C program for fibonacci:

```
int fib (int n) {          /* n >= 0 */
    int temp;
    int sum;

    if (n <= 1) {
        return 1;
    } else {
        temp = fib (n-1);
        sum = temp + fib (n-2);
        return sum;
    }
}
```

Here is a translation of the above program into MIPS assembly language. Let's assume that the stack pointer points at a full entry (i.e. 0(\$sp) is ok). *Assume that we are using the RAW architecture (i.e. there are delay slots).*

```
fib:   slti   $s0, $a0, 2
       bne   $s0, $0, end

fib1:  addi   $sp, $sp, -8
       sw    $ra, 0($sp)
       sw    $a0, 4($sp)

       jal   fib
       addi  $a0, $a0, -1

       add   $s0, $v0, $0

       lw    $a0, 4($sp)
       jal   fib
       addi  $a0, $a0, -2

       add   $v0, $s0, $v0

       addi  $sp, $sp, 12
end:   jr    $ra
       nop
```

There are 8 mistakes in this translation. Fix them in the above listing. To fix an error, you can either (1) add a single line of code or (2) change an existing line of code. *Assume that MIPS register conventions must be maintained throughout the execution.*

```

fib:   addi $v0, $r0, 1 ; Missing
      slti $t0, $a0, 2 ; Use of $s0 incorrect
      bne $s0, $0, end; Use of $s0 incorrect

fib1:  addi $sp, $sp, -12 ; Need to save 3 words
      sw $ra, 0($sp)
      sw $a0, 4($sp)
      sw $s0, 8($sp) ; Missing

      jal fib
      addi $a0, $a0, -1

      add $s0, $v0, $0

      lw $a0, 4($sp)
      jal fib
      addi $a0, $a0, -2

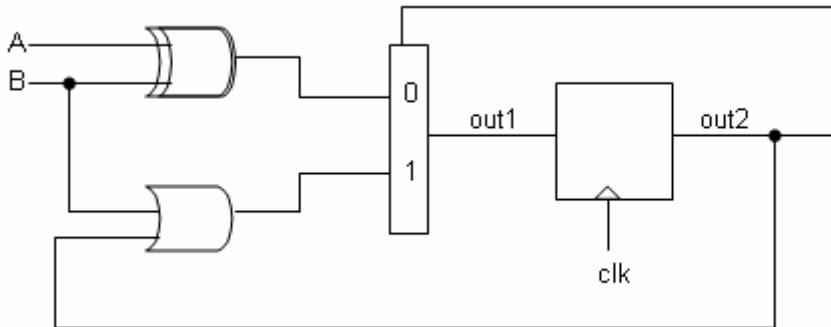
      add $v0, $s0, $v0

      lw $ra, 0($sp) ; Missing
      lw $s0, 8($sp) ; Missing
end:   addi $sp, $sp, 12 ; Need to move "end" up to counteract stack move
      jr $ra
      nop

```

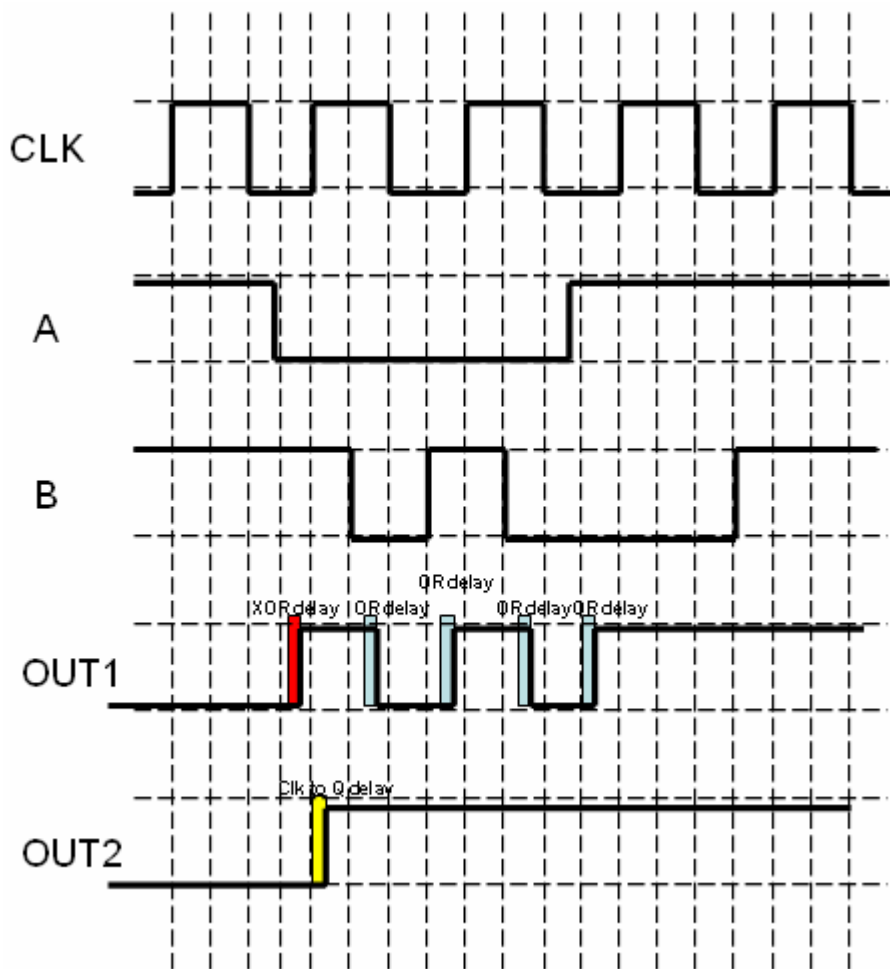
Note that, rather than moving the "end" label up one line, you could also add a "nop" after the first "bne" instruction above (problem is that the delay slot executes and decrements stack pointer, even if branch).

3) Please draw the waveform timing diagram for out1 and out2 below. Note that out1 and out2 start at 0. Clearly label any CLK-Q and combinational logic delays in your timing diagram!

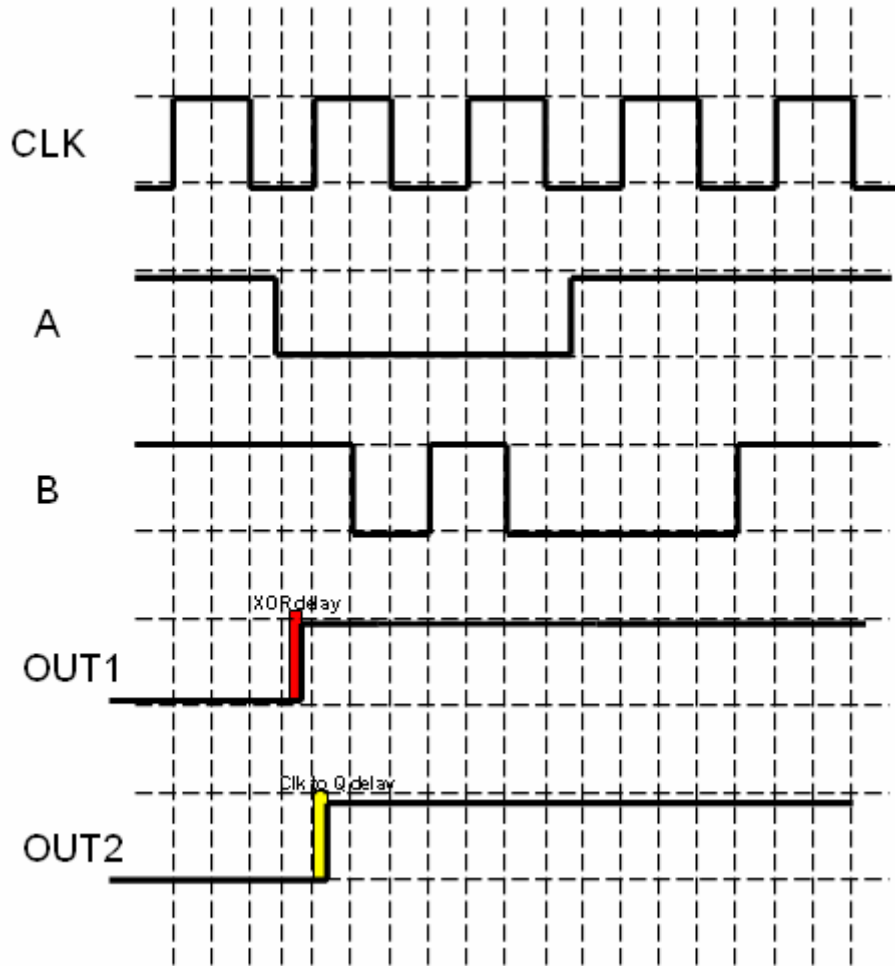


This problem was originally intended to have the OR gate be connected to A and B. However, the or gate is actually connected to b and out2. Both solutions were accepted and are shown below:

OR gate connected to A and B:



OR gate connected as shown in the problem:



4) Short Answer about MIPS:

- a) What is the distinction between $x = y$ and $*x = *y$ in C? Assume x is associated with register $\$s0$, y with $\$s1$. Here are 6 MIPS instructions, labeled L1 to L6:

```
L1: add $s0, $s1, zero
L2: add $s1, $s0, zero
L3: lw $s0, 0($s1)
L4: lw $t0, 0($s1)
L5: sw $t0, 0($s0)
L6: sw $s0, 0($s1)
```

Which (if any) is **true**? (“L4; L5” means L4 then L5)

- A: Line 2 is $x = y$; L4; L5 is $*x = *y$
- B: Line 1 is $x = y$; L4; L5 is $*x = *y$
- C: Line 1 is $x = y$; L6 is $*x = *y$
- D: Line 2 is $x = y$; L3 is $*x = *y$
- E: L4; L5 is $x = y$; L3 is $*x = *y$
- F: Line 2 is $x = y$; L4 is $*x = *y$

B. Note that L2 is $y = x$ as opposed to $x = y$.

- b) Suppose we have the following code sequence:

```
loadvals: lb $s0, 100($zero) #byte@100= 0x0F
          lb $s1, 200($zero) #byte@200= 0xFF
```

What are the 32-bit values loaded into $\$s0$ and $\$s1$? Explain!

- A: 15 255
- B: 15 -1
- C: 15 -255
- D: -15 255
- E: -15 -1
- F: -15 -255

B. load byte sign extends.

- c) Which of the codes below are pseudo-instructions in MIPS Assembly Language; that is, they are not found directly in the machine language? Explain!

- i. `addi $t0, $t1, 20000`
- ii. `beq $s0, $r0, Exit`
- iii. `sub $t0, $t1, 1`

- A: Only i.
- B: Only ii.
- C: Only iii.
- D: Both i. and ii.
- E: Both ii. and iii.
- F: Both i. and iii.
- G: All of the above

C. Only iii. Sub is a pseudo instruction, it gets turned into an add. The other two instructions are found directly in the machine language. Note that `addi $t0, $t1, 40000` would be a pseudo-instruction. Why?

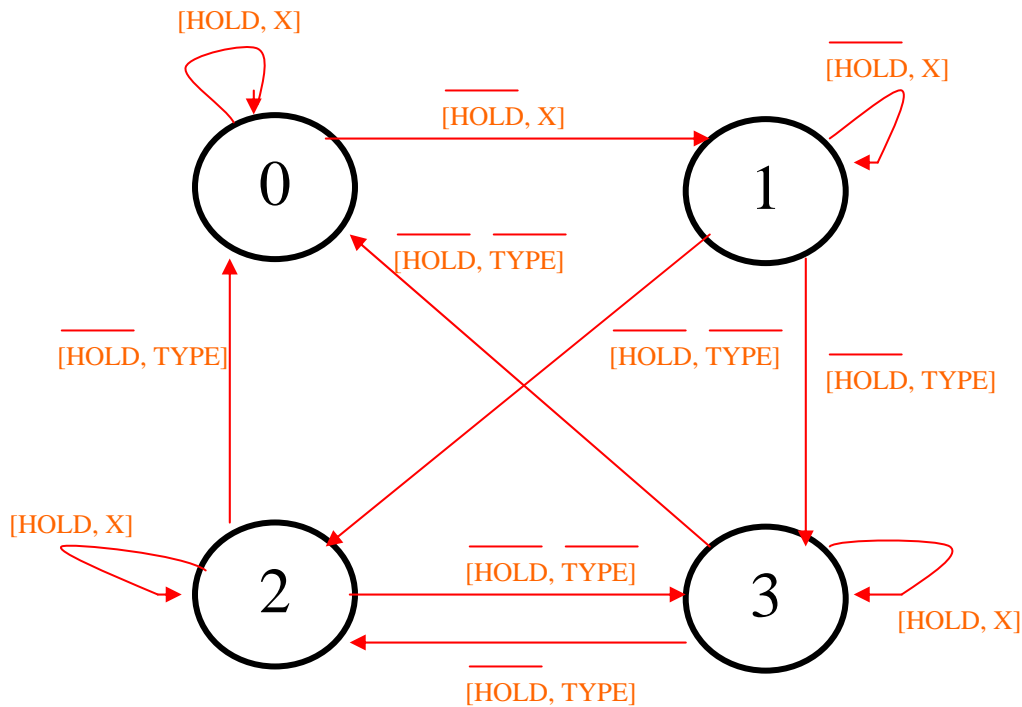
- d) Name two reasons that the HI/LO registers were added to MIPS instead of placing multiplication/division results into the normal register file:

Some answers were:

- Multiplication and Division both need 64 bits, so we need two register. (this counted as 1 response, not two).
- There is no way to specify 2 destination registers in a MIPS instruction, so we need special registers to place the value.
- 34 registers in your register file would require 6 bits to specify each register. This would be wasted space.
- Alternatively, you could use up 2 of your registers from your reg file for the HI/LO, but that does not make the common case fast.

5) In this problem, you must design a finite state machine (FSM) for a “versatile” counter that either counts in normal binary (*i.e.* 0, 1, 2, 3, 0, 1, ...) or as a Grey code (0, 1, 3, 2, 0, 1, 3,...) depending on the “TYPE” signal (TYPE=1 \Rightarrow Grey code). Further, when “HOLD” is asserted, it will stop counting (freeze the count). When RESET is asserted, the counter will return to the zero state.

- a) Complete the Following State Transition Diagram for the versatile counter. Include the “HOLD” and “TYPE” signals. Ignore RESET:



Almost everyone got this right. We took off a point or two if you used some sort of sloppy, ambiguous notation for the transitions.

Construct a State Transition Table for this FSM. Encode the state as 2 bits, S_1 and S_0 , where S_1 is the MSB (i.e. $S_1S_0=10$ for state 2). Ignore RESET.

S_1	S_0	Hold	Type	S_1'	S_0'
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	0	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	1	1
1	1	1	1	1	1

- b) Derive Next-State Logic Equations given the state transition table. Include the RESET signal in your equations. You will have 2 equations. Simplify these as much as possible (i.e. combine together terms as much as possible):

$$S_0' = \overline{\text{RESET}} * [(S_0 * H) + (S_1 * T * \overline{H}) + (S_0 * \overline{T} * \overline{H})]$$

$$S_1' = \overline{\text{RESET}} * [(S_1 * H) + (S_1 * S_0 * T) + (S_0 * T * H) + (S_1 * S_0 * H)]$$

There were, of course, many answers for these equations, most of which were correct. We took off points however, if your solution used more transistors than ours (i.e. was non-minimal). We also took off points for forgetting reset.

- c) Implement the complete counter using rising-edge triggered flip flops, inverters, and 2, 3, or 4-input NAND gates. Minimize the number of gates that you use. Clearly label input and output signals. It should have as input: CLOCK, HOLD, TYPE, and RESET. It should have as output a 2-bit count value.

I don't feel like drawing all this in word, so you get to practice your structural verilog. Really, it is just 3 levels of gates.

S0':

```
NAND (W1, S0, H)
NAND (W2, Not(S1), T, Not(H))
NAND (W3, Not(S0), Not(T), Not(H))
NAND (W4, W1, W2, W3)

NAND (W5, W4, Not(RESET))

DFLIP (.d(Not(W5), .q(S0'), .clk(clk))
```

S1':

```
NAND (W1, S1, H)
NAND (W2, S1, Not(S0), T)
NAND (W3, Not(S0), T, Not(H))
NAND (W4, Not(S1), S0, Not(H))
NAND (W5, W1, W2, W3, W4)

NAND (W6, W5, Not(RESET))

DFLIP (.d(Not(W6), .q(S1'), .clk(clk))
```


[This page left for scratch]

[Spare page for Scratch (tear off if desired)]

[This page left for scratch]

[The End]