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

Spring 2003 John Kubiatowicz

Prerequisite Quiz

February 3, 2003
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:	
SID Number:	
Discussion Section:	

1	
2	
3	
4	
Total	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494 4

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

-13_{10} to base 2:

$13_{10}=1101_2$ so $-13_{10}= (1111\ 1111\ 1111\ 0011)_2$

$FF4E_{16}$ to base 10:

$FF4E_{16}=(1111\ 1111\ 0100\ 1110)_2= -(0000\ 0000\ 1011\ 0010)_2=-(11x16+2)=-178_{10}$

52_{10} to base 16:

$52_{10}=34_{16}$

$(1111111110111100)_2$ to base 10:

$(1111\ 1111\ 1011\ 1100)_2 =-(0000\ 0000\ \ 0100\ 0100)_2=-(4*16+4)=-68_{10}$

$8CB3_{16}$ to base 8:

$8CB3_{16}=(1000\ 1100\ 1011\ 0011)_2=(1\ 000\ 110\ 010\ 110\ 011)_2=(106263)_8$

[This page left for scratch]

2) Below is a recursive routine to add up values for every node in a tree.

```
typedef struct {
   int value;                ; 4 bytes (1 word)
   node *left;               ; 4 byte pointer (1 word)
   node *right;              ; 4 byte pointer (1 word)
} node;

int sum (node *root) {
   if (root == (node *)0) {      /* Null pointer => empty tree */
      return 0;
   } else {
      return (node->value) + sum(node->left) + sum(node->right);
   }
}
```

Each item in the tree is a structure with a value, a left pointer, and a right pointer. The `sumtree()` routine calls itself recursively on the left and right subtrees, adds in the value of the current node, then returns the result. An empty tree (or subtree) is indicated by a null (zero) pointer.

Note: If register `$t0` contains a pointer to a node, then the contents of `node->left` can be loaded into register `$t1` with:

M
```
getval:  lw    $t1, 4($t0)    ; Left pointer at offset 4
```

Here is a translation of the above program into MIPS assembly language (assume no branch delay slot). Assume that the stack convention is that $SP points at a *full* entry:

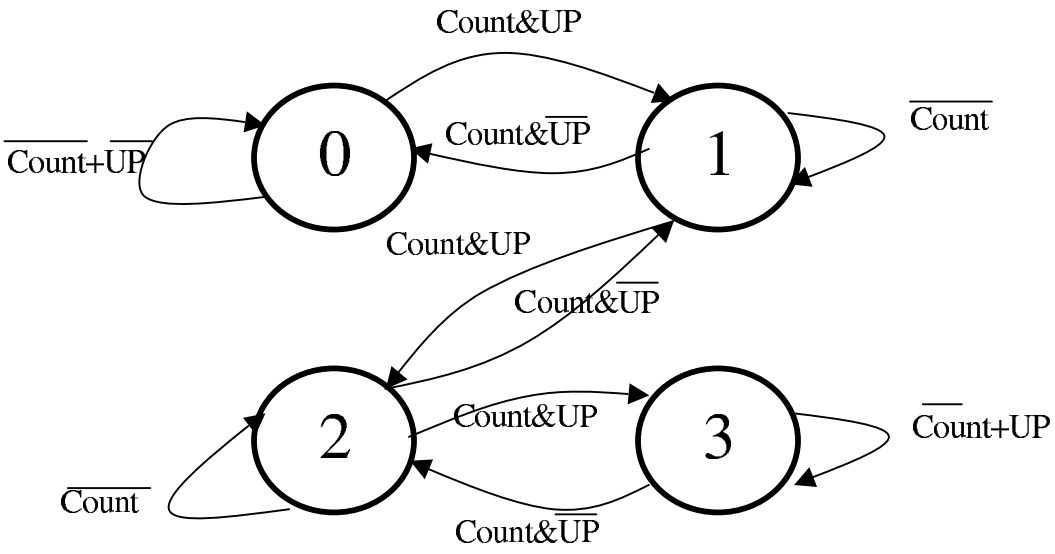
```
sumtree: beq   $a0, $zero, exit
         sub   $sp, $sp, -8

         sw    $ra, 0($sp)
         sw    $a0, 4($sp)
         sw    $s0, 8($sp)

         lw    $s0, 0($a0)          ; Value of node (sum in $s0)

         lw    $a0, 4($a0)          ; Traverse left tree
         jal   sumtree
         add   $s0, $v0, $s0        ; Add result to sum

         lw    $a0, 8($a0)          ; Traverse right tree
         jal   sumtree
         add   $s0, $v0, $s0        ; Add result to sum

         lw    $ra, 0($sp)

exit:    add   $sp, $sp, 8
         jr    $ra
```

There are a small number (< 8) of errors in this translation. Fix them in the above listing. *Assume that MIPS register conventions must be maintained throughout the execution.*

```
Sumtree :   add   $v0, $zero , $zero  ;initialization
            beq   $a0, $zero, exit
            addiu $sp, $sp, -12

            sw    $ra, 0($sp)
            sw    $a0, 4($sp)
            sw    $s0, 8($sp)


            lw    $s0, 0($a0)           ; Value of node (sum in $s0)

            lw    $a0, 4($a0)           ; Traverse left tree
            jal   sumtree
            add   $s0, $v0, $s0         ; Add result to sum

            lw    $a0, 4($sp)           ; $a0 has to be restored
            lw    $a0, 8($a0)           ; Traverse right tree
            jal   sumtree
            add   $v0, $v0, $s0         ; Add result to sum

            lw    $ra, 0($sp)

            lw    $s0, 8($sp)           ;pop up $s0
            addiu $sp, $sp, 12
exit:       jr    $ra
```

3) In this problem, you must design a four-state finite state machine (FSM) that implements a *saturating* counter that either counts up or down in normal binary. The counter only changes when the "COUNT" signal is asserted (i.e. = 1). When counting up (the "UP" signal asserted), the counter will count as follows: 0, 1, 2, 3, 3, 3…. When counting down (the "UP" not asserted), the counter will count as follows: 3, 2, 1, 0, 0, … The RESET signal will take the counter to the 0 state. We call this the SatCounter.

a) Complete the Following State Transition Diagram for the SatCounter count output. Include the "COUNT" and "UP" signals. Ignore RESET:



b) 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.

Count	Up	S1S0	S1'S0'
0	X	00	00
1	0	00	00
1	1	00	01
0	X	01	01
1	0	01	00
1	1	01	10
0	X	10	10
1	0	10	01
1	1	10	11
0	X	11	11
1	0	11	10
1	1	11	11

c) 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):

$$S_0' = \overline{Reset}(S0\overline{Count}+S1S0UP+S1\overline{S0}Count+\overline{S0}CountUP)$$

S1S0

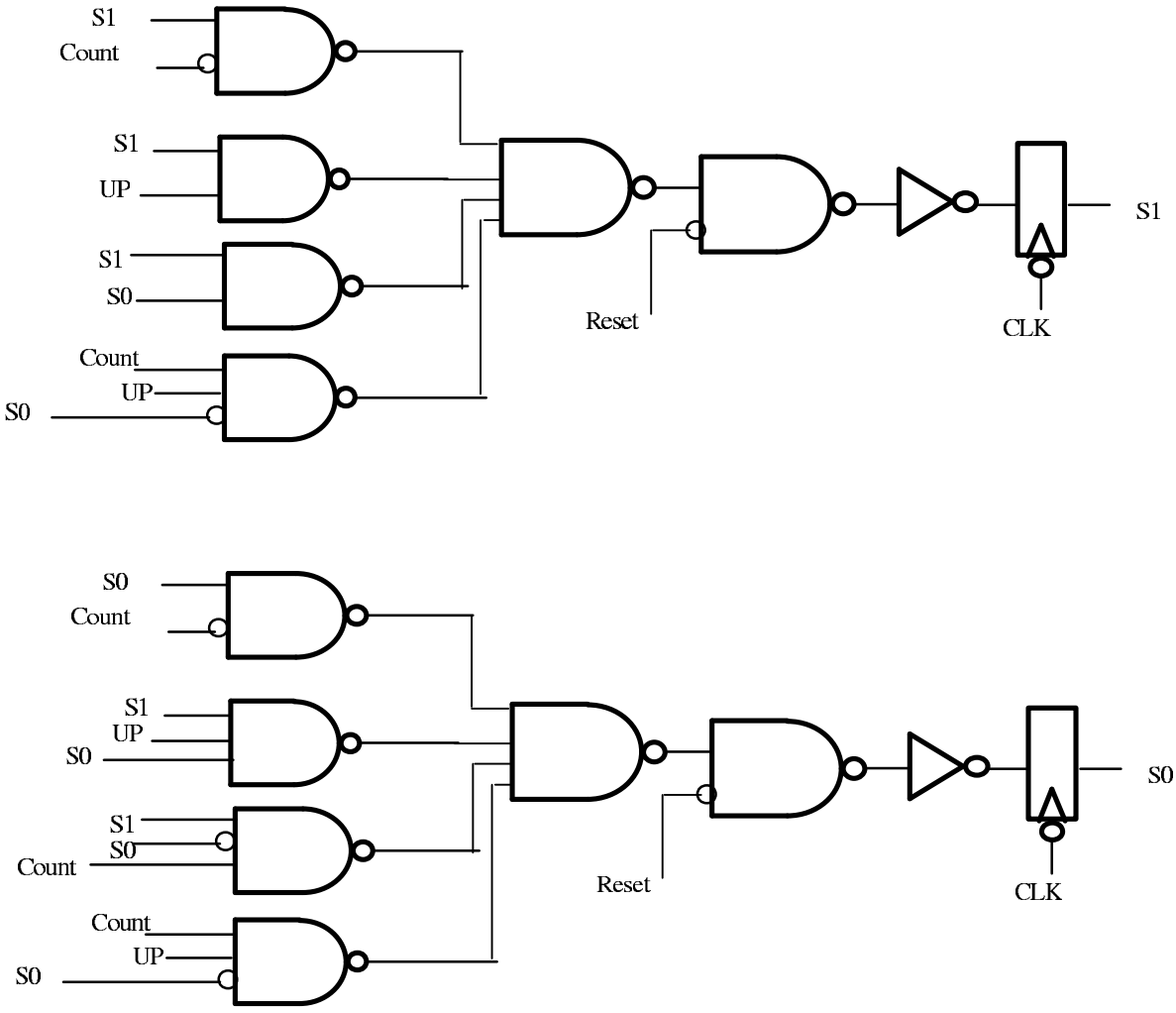
Count UP

S0'	00	01	11	10
00		1	1	
01		1	1	
11	1		1	1
10				1

$$S_1' = \overline{Reset}(S1\overline{Count}+S1UP+S1S0+S0CountUP)$$

	00	01	11	10
00			1	1
01			1	1
11		1	1	1
10			1	

d) Implement the complete SatCounter using falling-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, COUNT, UP, and RESET. It should have as output a 2-bit count value.



[This page left for scratch]

4) Assume that we have a 64-bit processor with 64-bit words, and that this processor is byte-addressed (i.e. addresses specify bytes). Now, suppose that this processor has a 32-word, *direct-mapped* cache with 2-word cache lines.

a) Split the 64-bit address into "tag", "index", and "cache-line offset" pieces. We start you off with one of these.

tag: bits 63-8
index: bits 7-4
cache-line offset: **bits 3 – 0**

b) Assume that the processor makes a series of byte accesses to memory (shown to the left below). Label each memory reference address as a Hit (H) or a Miss (M). Also, show the *final* contents of the cache in the table at the right. Fill in values for the TAG and words of the cache. Use the notation "M[i]" to mean the word in memory at address i (keep these word aligned!). Include no more than one value per word (words can be left blank).

Reference Address	Hit or Miss?
10_{10} ($00A_{16}$):	M
0_{10} (000_{16}):	H
32_{10} (020_{16}):	M
264_{10} (108_{16}):	M
291_{10} (123_{16}):	M
32_{10} (020_{16}):	M
256_{10} (100_{16}):	H
104_{10} (068_{16}):	M
511_{10} ($1FF_{16}$):	M
140_{10} ($08C_{16}$):	M
184_{10} ($0B8_{16}$):	M
615_{10} (267_{16}):	M
40_{10} (028_{16}):	H
496_{10} ($1F0_{16}$):	H
508_{10} ($1FC_{16}$):	H
96_{10} (060_{16}):	M

Index	TAG Value	Word 0	Word 1
0	0x1	M[100_{16}]	M[108_{16}]
1			
2	0x0	M[020_{16}]	M[028_{16}]
3			
4			
5			
6	0x0	M[060_{16}]	M[068_{16}]
7			
8	0x0	M[080_{16}]	M[088_{16}]
9			
A			
B	0x0	M[$0B0_{16}$]	M[$0B8_{16}$]
C			
D			
E			
F	0x1	M[$1F0_{16}$]	M[$1F8_{16}$]

c) Calculate the cache hit rate (you can leave as a fraction).

5/16

[This page left for scratch]

[The End]