Practice Questions for Final Exam
CS 64, Fall 2018, Dr. Ziad Matni

IMPORTANT NOTE: These questions are NOT representative of EVERYTHING you need to study for the final exam! You should also review your lecture slides/notes, lab assignments questions and also all the examples and demos done in class. The final exam is cumulative, but will be more heavily skewed towards the newer (post-midterm) material.

A. Data Representation and Assembly Language Code

1. Binary-to-decimal/hexadecimal conversion
    a. Convert **1001 0010 1100 0011** to 4-digit hexadecimal
    b. Convert the *signed* binary value **1001 1111** to decimal

2. Add the 2 following 8-bit numbers: **0110 0010** and **0011 0100** and indicate the status of the carry and overflow bits at the end of the addition. Interpret your findings.

3. Name one reason why **li** is a MIPS *pseudoinstruction*.

4. Translate this MIPS assembly code into C/C++ code.

```
.data
talk: .asciiz "blabla"
cs: .word 3
.text
main:
         li $t0, 5
         la $t1, cs
         lw $t2, 0($t1)
         blt $t0, $t2, gothere
         li $v0, 4
         la $a0, talk
         syscall
         j end
gothere:
         li $v0, 4
         la $a0, talk
         syscall
         syscall
end:
         li $v0, 10
         syscall
```

5. Write the following MIPS instructions in machine-language hexadecimals (show all work):
   `addiu $t0, $s0, 17` and **`sub $v0, $s4, $t5`**

6. Given a MIPS machine language instruction of **`0x02088024`**, and being told that it is an R-type, what is the assembly instruction?

7. What will the final value in register $s0 in this code be?

```
        li $s0, 20
        sll $s0, $s0, 2
        add $s0, $s0, $s0
        sra $s0, $s0, 4
```

8. Consider the C code below:

```
// arr is a globally accessible array of ints
// s0 already holds a value of type unsigned int
unsigned int s1 = arr[s0];
unsigned int s2 = arr[s0 - 1];
unsigned int s3 = arr[s0 + 1];
```

Using **no more than six instructions**, implement the above C code snippet in MIPS. You don't have to follow the MIPS Calling Convention.

9. Consider the C code below.

```
int sum( int arr[], int size )
{
   if ( size == 0 )
      return 0;
   else
      return sum( arr, size - 1 ) + arr[size – 1];
}
```

a. Knowing that you **have to follow the MIPS Calling Convention**, which variables should be preserved either directly (via the stack) or indirectly (in an s-register) in order to maintain the intended program behavior?

b. Implement the previously shown C/C++ code using MIPS assembly, taking care to preserve the values you identified previously. Ignore the .data part and just focus on the .text part of the program.

B. Digital Logic Design

10. Show how a NOR function can be used as an AND function?

11. Simplify this expression using Boolean algebra (i.e. not with K-Mpas):
F = NOT ((A NOR B) . (C + A.B)) and draw the resulting circuit.

12. Consider the following truth table, which includes don't cares:

A	**B**	**C**	**D**	**R**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	X
0	0	1	1	0
0	1	0	0	X
0	1	0	1	1
0	1	1	0	X
0	1	1	1	X
1	0	0	0	X
1	0	0	1	0
1	0	1	0	1
1	0	1	1	X
1	1	0	0	1
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

Simplify the output function **R** using a Karnaugh Map, and show the resulting sum-of-products representation. Show the map, along with the boxes you chose. For full credit, both the number of ORs (+) and the sizes of the products must be minimal.

13. Using one or more D-Latch, one or more 2-to-1 mux, and any number of basic logic blocks (i.e. AND, OR, NOT, etc.), construct a circuit that takes 2 bits as inputs (B0, B1) and writes them to 2 registers called REG0 and REG1, respectively, *but only* when another input, E = 1. The registers retain their values if E = 0. The output of this circuit is the value stored in one the registers as chosen by inputs S1 and S0. If S1=1 and S0=1, we should see the value in REG1 appear on the output (O) and if S1=0 and S0=0, we should see the value in REG0 appear at O. Any other combination of S1 and S0 should leave the output unchanged, that is the previous selection should remain the same.

14. Consider a device that consists of three buttons labeled "UP" and "RESET", along with a light. The device internally counts the number of times "UP" is pressed, and when it is pressed two times, the device causes the light to illuminate. Additional presses of "UP" do nothing. Pressing "RESET" at any point will reset the internal counter back to zero, and will cause the light to go out. (Note that the light may have already been off, as when the user presses "UP" once followed by "RESET".)

    For this question, you will implement this device as a finite state machine. The machine has the following two external inputs:
    R: set to 1 whenever "RESET" is pressed
    U: set to 1 whenever "UP" is pressed
    The machine also has one external output:
    L: set to 1 whenever the light should be illuminated
    If both "RESET" and "UP" are pressed at the same time, then the behavior should be as if only "RESET" was pressed.

    a. Draw the finite state machine diagram corresponding to this task. All transitions should be drawn as products of R and U. For example, if a particular transition should be taken only if R = 1 and U = 0, then this should be drawn as $R.\overline{U}$ or R.!U.
    b. Draw the current/next state table for this FSM and describe the outputs in the simplest terms of inputs in a sum-of-products format (you can use K-Maps to determine this). Don't forget to include L as one of the outputs you describe!
    c. Draw the digital logic circuit to implement this FSM.

****** GET THE MOST OUT OF THIS REVIEW!!!***

DO THE QUESTIONS FIRST BEFORE LOOKING AT THE ANSWERS!!!***

ANSWERS TO THE REVIEW QUESTIONS FOR FINAL EXAM

A. Data Representation and Assembly Language Code

1.
   a. 1001 0010 1100 0011 /bin = 0x92C3
   b. 1001 1111 /bin → 0110 0000 + 1 = 0110 0001 = –($2^6 + 2^5 + 1$) = -97

2.
```
  0110 0010
+ 0011 0100
  1001 0110
```
The carry out bit = 0, the overflow bit is 1.
So, if these were 2 unsigned numbers, there would be no carry out, but if these were 2 signed numbers, then we'd have overflow.

3. **li** takes as second argument a 32-bit signed number. MIPS instructions themselves are 32-bit long, so loading this number into a register should actually be done in pieces: first load the upper 16-bits of the number, then load the lower 16-bits. Therefore the instruction li is really a macro for (at least) 2 regular instructions – i.e. it's pseudocode.

4. In C/C++:
```
char talk[] = "blabla";  // or string talk = "blabla"
int cs = 3, t0 = 5;
if (t0 >= cs)
   { printf(talk); }
else
   { printf(talk);
     printf(talk); }
```

5.
```
addiu $t0, $s0, 17  =  0x26080011
sub $v0, $s4, $t5   =  0x028D1022
```

6. **`0x02088024`** = **`and $s0, $s0, $t0`**

7.
```
$s0 = 20                This is in decimal, so in binary, it's 0000 … 0001 0100 (in 32-bits)
sll $s0, $s0, 2            → $s0 becomes 0000 … 0101 0000
add $s0, $s0, $s0          → $s0 becomes 0000 … 1010 0000
sra $s0, $s0, 4            → $s0 becomes 0000 … 0000 1010 = 10 (decimal)
```

8. In 6 or under instructions:
```
la $t0, arr
sll $s0, $s0, 2
addu $t0, $t0, $s0
lw $s1, 0($t0)
lw $s2, -4($t0)
lw $s3, 4($t0)
```

9. We assume **arr is in $a0 and size is in $a1**. We'll preserve them inside $s0 and $s1, respectively. Since this program has a recursion, we need to also preserve $ra. All preservation is done in the stack.

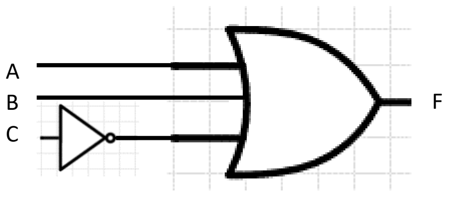
```
int sum( int arr[], int size ) {
  if ( size == 0 )
    return 0 ;
  else
    return sum( arr, size - 1 ) + arr[size-1]; }
```

```
.text
sum:
    addiu $sp, $sp, -12 # PUSH
    sw $ra, 8($sp)
    sw $s1, 4($sp)
    sw $s0, 0($sp)

    li $v0, 0
    beq $a1, $zero, return    # is size !=0?

    Addi $a1, $a1, -1   # size is now: size - 1
    move $s0, $a0       # preserve &a0
    move $s1, $a1       # preserve a1 (size)

    jal sum             # recursive call

    sll $s1, $s1, 2     # multiply size by 4
    add $s0, $s0, $s1   # s0 is now the address of a[size-1]

    lw $t0, 0($s0)      # Get that array element
    add $v0, $v0, $t0   # add it to $v0

return:
    lw $ra, 8($sp)      # POP
    lw $s1, 4($sp)
    lw $s0, 0($sp)
    addiu $sp, $sp, 12
    jr $ra

main:
    la $a0, arr     # a0 = &a[]
    li $a1, 4       # a1 = size

    jal sum

exit:
    li $v0, 10
    syscall
```

B. Digital Logic Design

10. Taking advantage of DeMorgan's theorm, you will note that if the inputs to the NOR are inverted, you get: **F = NOT($\bar{A}$ + $\bar{B}$) = A.B**

11. F = NOT ((A NOR B) . (C + A.B))
 = NOT (($\bar{A}$. $\bar{B}$) . (C + A.B))
 = NOT ($\bar{A}$. $\bar{B}$. C + $\bar{A}$. $\bar{B}$. A . B)
 = NOT ($\bar{A}$. $\bar{B}$. C)
 = A + B + $\bar{C}$



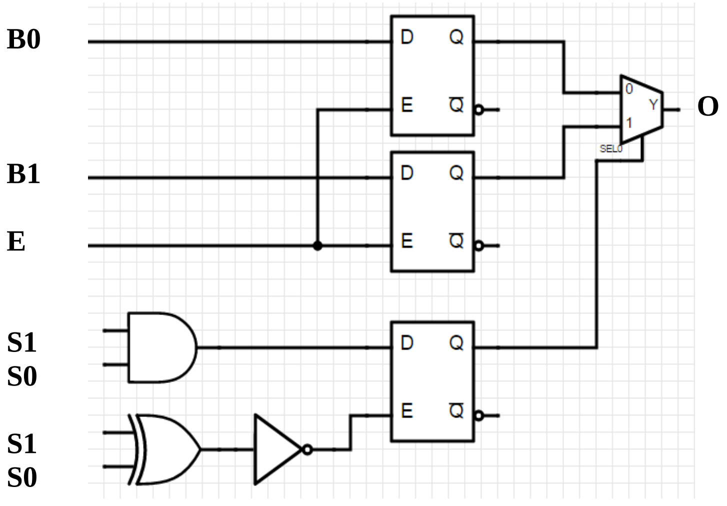
12. K-Map:

CD \ AB	00	01	11	10
00	1	X	1	X
01		1	X	
11		X	X	X
10	X	X	X	1

Since X's **can be either 0 or 1**, to maximize the size of our groupings and minimize the number of our groupings, we can transform the above to the following with 2 major groupings:

CD \ AB	00	01	11	10
00	1	1	1	1
01		1	1	
11		1	1	0
10	1	1	1	1

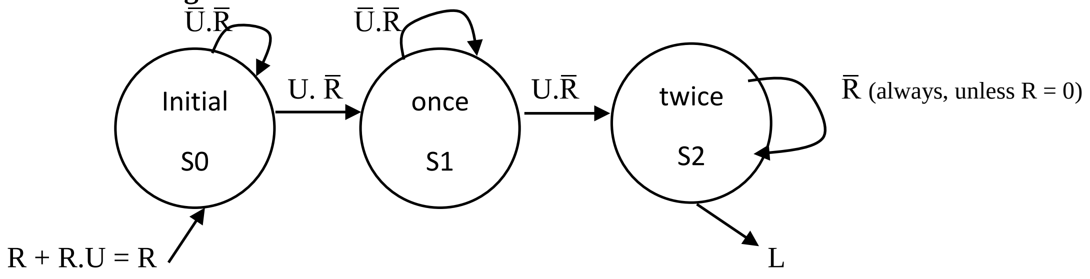
This gives us the formula: **F = $\bar{D}$ + B**
NOTE: I purposely made one of the Xs into a 0, so that I could minimize my groupings.

13.


The select line into the mux has to represent S1 = 1 && S0 = 1 ---or--- S1 = 0 && S0 = 0. **Any other combination of S1 and S0 should be ignored**. The best way to do that is to pass S1.S0 through a D-latch that is enabled by either case of S1 = 1 && S0 = 1 ---or--- S1 = 0 && S0 = 0 (that is, S1 XNOR S2).

14.

a. State diagram will show 3 states:


$\overline{U}.\overline{R}$
$\overline{U}.\overline{R}$
Initial
S0
$U.\ \overline{R}$
once
S1
$U.\overline{R}$
twice
S2
$\overline{R}$ (always, unless R = 0)
R + R.U = R
L

b. Current/Next table: There are 3 states, so we need at least 2 bits to describe them (B1, B0).

CURRENT STATE ***NEXT STATE***

State	**B1**	**B0**	**U**	**R**	**State***	**B1***	**B0***	**L**
Initial	0	0	0	0	Initial	0	0	0
	0	0	1	0	Once	0	1	0
Once	0	1	0	0	Once	0	1	0
	0	1	1	0	Twice	1	0	0
Twice	1	0	X	0	Twice	1	0	1
	X	X	X	1	Initial	0	0	0

Note the entry for R = 1 (regardless of all other inputs) making B1*=B0*=0.

K-Map for B1* (left) shows that **B1* = !B1.B0.U.!R + B1.!B0.!R**
K-Map for B0* (right) shows that **B0* = !B1.!B0.U.!R + !B1.B0.!U.!R**

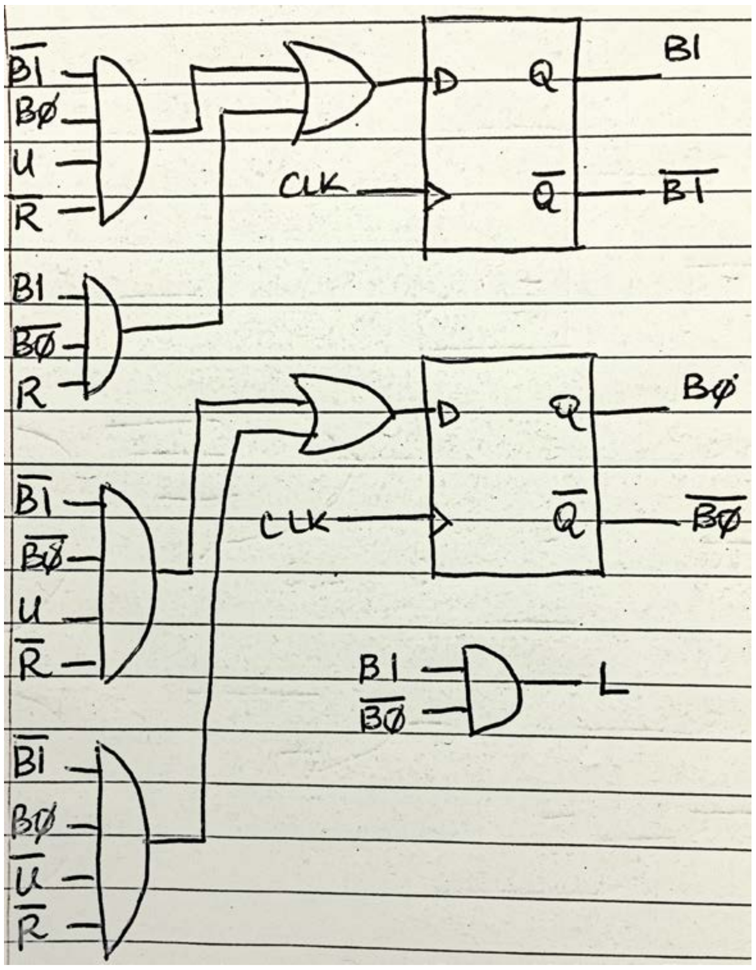
U.R \ B1.B0	**00**	**01**	**11**	**10**
00				1
01				
11				
10		1		1

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

Note that the one output, L, is true only when in the S2 state (i.e. B1=1, B0=0), so:
L = B1.!B0

Circuit diagram is on next page.

c. Logic circuit design:


$\overline{B1}$
$B\emptyset$
U
$\overline{R}$
$B1$
$\overline{B\emptyset}$
R
$\overline{B1}$
$\overline{B\emptyset}$
U
$\overline{R}$
$\overline{B1}$
$B\emptyset$
$\overline{U}$
$\overline{R}$
D
Q
$\overline{Q}$
CLK
$B1$
$\overline{B1}$
D
Q
$\overline{Q}$
CLK
$B\emptyset$
$\overline{B\emptyset}$
$B1$
$\overline{B\emptyset}$
L