CS 270 Fall 2013 Mock Final

Name(s) of student(s) ______________________________

Please read these instructions completely before proceeding, and sign below. Your exam will not be graded without your signature.

- This is the mock final to help you prep for the final. The exam is a closed book exam, but you are allowed to bring in one page (one single side) of handwritten notes. It is designed so that the average score is about (67%).
- The exam would normally also have on the last pages, the LC-3 opcodes and the datapath of the LC3 (page 570). These are not in the mock version in order to save a few trees.
- You are strongly encouraged to collaborate in preparing the answers to this mock exam. Please also feel free to consult with the TAs and with Sanjay to solve the problems here.
- In order to reduce our effort on redundant grading, please write the names of all the students who collaborated on the first page. You can do this at the granularity of individual Problems but not finer. So if students A&B collaborated on Probs 1&2, and students B&C worked togther on problems 4&5, and all three students worked individually on the other problems, I would expect to see 5 submisions (one each for the individual work of A, B and C, one for the AB collaboration and one for the BC collaboration).

I have read the above instructions. I will do the exam honestly and fairly.

Signature ______________________________

Problem 0: Plan of Attack [15 pts] Quickly read through the exam and make a plan of attack. Think about what skills each question is testing for and your comfort level, and rate its difficulty (not how long it's going to take—some are long and some are short—but how hard it seems). Based on this, fill up the PoA columns on in the table below. Don't fill up the last two columns as yet.

Don't write in these columns				Plan of Attack			Revised PoA	
Prob	Topic	Max	Score	PoA	Start	End	PoA	Time
1	LC3 Architecture	20						
2	Assembly Programming I	20						
3	Assembly Programming II	15+5						
4	C (stack frames)	25						
5	C (memory, data)	20						
	Total	100+5						

Problem 1: LC-3 Architecture [20 pts]

For this problem, we will use the following names and conventions for the LC3 signals:

- LD.MAR, LD.MDR, LD.Reg, LD.CC, L.PC, LD.IR: signals that load the registers in the LC-3 datapath (one or more of them are on in any given cycle).
- GatePC, GateMARMUX, GateMDR, GateALU: signals that control access to the bus (no two of them can be simultaneously on)
- PCMUX.Sel, Addr1MUX.Sel, Addr2MUX.Sel, MARMUX.Sel, SR2MUX.Sel: select signals of the muxes. They have the following convention:
  - PCMUX.Sel = 00 to select the Bus, 01 for the output of the Adder (+), 10 for the output of the +1 box
  - Addr1MUX.Sel = 0 for the SR1 output from the Reg File, 1 to select the PC

- Addr2MUX.Sel = 00 for Sext(IR[10:0]), 01 for Sext(IR[8:0]), 10 for Sext(IR[4:0]), 11 for zero.
- MARMUX.Sel: 0 for Zext(IR[7:0]), and 1 for the output of the Adder
- SR2MUX.Sel: 0 for Sext(IR[5:0]), and 1 for the SR2 output from the Reg File.

- Memory control signals: Mem.EN enables a memory transaction (if it is 0, no interation with the memory); R.W controls whether the transaction is a read (R.W = 1) or a write (R.W = 0). R.W is ignored unless Mem.EN=1. If Mem.EN=1, R.W=1, then it is also necessary for LD.MDR to be 1, otherwise the value read from memory gets ignored.

Part a. The following, partially filled table gives the cycle by cycle details of what happens in the the LC-3 when an LD instruction is executed. Fill in all the missing entries. [7 pts]

Cycle	Active Signals	Register Transfer Notation
1	GatePC, LD_MAR	MAR $\leftarrow$ PC
2	LD_MDR, MEM_EN, R_W, LD_PC, PCMUX.Sel = 10	MDR $\leftarrow$ Mem[MAR], PC $\leftarrow$ PC + 1
3	LD_IR	IR $\leftarrow$ MDR
4		MAR $\leftarrow$ PC + Offset9[1]
5	Mem.EN, R.W, LD.MDR	
6		

[1] Or alternatively, MAR $\leftarrow$ PC + Sext(IR[8 : 0]), since Offset9 is just Sext(IR[8:0]).

We wish to modify the design of the LC-3, by getting the LDI/STI instructions to use base-register plus offset addressing, like the LDR and STR instructions, but still being indirect. In other words, the instruction, "`LDI DR, BaseR, Offset6`" should have the following effect: $\texttt{DR} \leftarrow \texttt{Mem[Mem[BaseR} + \texttt{Sext(Offset6)]]}$.

Part b. What would the effect of "`STI SR, BaseR, Offset6`" be? [3 pts]

Part c. Complete the following partially filled table the describes the cycle by cycle execution of the new `STI` instruction (the first three cycles of instruction fetch are omited). [10 pts]

Cycle	Active Signals	Register Transfer Notation
4		$\texttt{MAR} \leftarrow \texttt{RegFile[IR[8:6]]}$[2]
5		
6		
7		
8		

[2]This denotes the contents of the register identified in $\texttt{IR}[8:6]$. It would also be acceptable to simply say $\texttt{MAR} \leftarrow \text{BaseReg}$.

Problem 2: Assembly Programming I: [20 pts]

Part a. At a certain point in an LC3 program, you need to left shift register R2 by 1, and pad the least significant bit with a 1. Your code must work correctly whether the number in R2 is positive or negative. Write the sequence of LC3 instructions that will accomplish this. [4 pts]

We want to put into register R3, a mask which is 1 everywhere except the $k+1$ least significant bits, i.e., we want R3[k:0] to become 0 and R3[15:k+1] to be 1. The value of k is already in register R1, and is guaranteed to be an integer between 0 and 15. There are two strategies to achieve this.

Part b. Strategy 1: repeatedly (exactly k times) in a loop, left-shift an appropriate initial value, possibly with some update at each iteration, or at the end. Write the LC3 code to do this. [8 pts]

Part c. Strategy 2: since there are only 16 different answers possible, we declare a table of these values in memory and simply copy the correct answer into R3. Write the LC3 code to do this. [8 pts]

Problem 3: Assembly Programming II: [15+5 pts]

Part a. In order to find the bitwise OR of two source registers, say R0 and R1, and store it in a third one, say R2 (leaving the sources unchanged), you would write the following sequence of 6 instructions.

```
NOT R0,R0;    NOT R1,R1;    AND R2,R0,R1;    NOT R2,R2;    NOT R0,R0;    NOT R1,R1;
```

Modify this for the case when only two registers are involved, i.e., you want to compute, `R0` $\leftarrow$ `OR(R0, R1)` (don't write all your instructions on the same line, but use one line per instruction). [5 pts]

If you are computing such an OR operation inside a loop, the above code is inefficient—you negate some registers after the AND in one iteration, and then in the next iteration you negate them again before the AND. It's better if we could first negate registers outside the loop, and then not negate anything inside the loop.

The following LC3 assembly code implements a function `shift(input, k)`, the problem of PA4. We want to optimize it by deleting the four lines marked, to avoid repeated negation inside the loop. However, we need to be careful that everything else works perfectly—there can be other repercussions.

```
;;  This is the code for the function shift(input, k) that returns output
;;  First we complete the setting up the activation record (stack frame) so
;;  that R5 with an offset of 3 points to the return value (output) and with
;;  offset 4 points to input and with offset 5 points to k (the two parameters)
;;  Then comes some code to initialize R2 with mask1, and R3 with mask2
;;  All this is omited for brevity.  R2 has 2^k and R3 has 1 (i.e., 2^0)
;;  The code below also does not have a separate location in the stack frame
;;  for output, it just used the RV slot Mem[R5 + #3]
;;
                LDR R1, R5, #5  ; R1 is again a loop counter: it's
                ADD R1, R1, #-16; initialized to k-16 and incremented
                BRZ DONE        ; until it reaches 0

AGAIN           LDR R4, R5, #4  ; R4 <- INPUT
                AND R4, R4, R2  ; test mask1-th bit of R4 (INPUT)
                BRz SKIP1       ; if the bit is zero then don't modify output
                LDR R4, R5, #3  ; R4 <- OUTPUT
                NOT R4, R4      ; <-------- delete this line
                NOT R3, R3      ; <-------- delete this line
                AND R4, R4, R3  ;
                NOT R4, R4      ; <-------- delete this line
                NOT R3, R3      ; <-------- delete this line
                STR R4, R5, #3  ; store it back to OUTPUT
SKIP1           ADD R2, R2, R2  ; LSH mask1
                ADD R3, R3, R3  ; and also mask2
                ADD R1, R1, #1  ; increment the loop counter
                BRnp AGAIN      ; exit when it becomes zero
DONE    ;;  Epilog to pop the dynamic link, restore RA, and then
                RET
```

Part b. Read the above code carefully, cross out the four marked lines, and fix the rest of the code. [10 pts + 5 ec if you discover the subltle one]

Problem 4: Stack Frames & Activation Records [25 pts] Consider the following recursive C program, which actually computes the Fibonacci numbers in a slightly different way.

```
int foo(int n){
  int t1, t2, t3;
  if (n<2) return 1;
  if (n==2) return 2;
  t1 = foo(n-3);
  t2 = foo(n-2);
  t3 = t1+2*t2;
  return t3;
}
```

We are interested in a particular program state that we get to after the following execution sequence:

Top level `main` calls `foo(10)`
    which first calls `foo(7)`, that (eventually) returns 21 into `t1`
    and then the parent calls `foo(8)`.
        Now `foo(8)` calls `foo(5)` that eventually returns 8 into *its* `t1`
        and then calls `foo(6)`.

On the next page, draw a neat diagram of the activation record just before this call to `foo(6)` is about to return. We have provided a template for you and an indication of the kinds of comments you should write. Remember that all of these are going to be activation records for `foo` so it's important to disambiguate by telling which call. Whenever the value of any entry is known, please write the actual value there. Whenever a pointer is placed in any entry and its value is known (e.g., FPs), draw an arrow to where it points. Otherwise just describe what it should be in the comments. Draw a horizontal line at the start of each new activation record to highlight the start/end of each activation record. [25 pts]

RV	back to main (value unknown)
10 (xA)	parameter n of foo(10)

Problem 5: C pointers, memory and variables [20 pts]

```
typedef struct
{
int i;
float f;
} simple;
simple s;
simple *p=&s;
s.i = 1234;
s.f = 0.112233f;
printf("s.i = %d, s.f = %f\n", s.i, s.f);
p->i += 2345;
p->f *= 2.0f;
printf("s.i = %d, s.f = %f\n", s.i, s.f);
```

Part a. Pointers and structs/Arrays and static/dynamic allocation. What is the output of the code above. [5 pts]

Part b. How does the dot (.) operator differ from the arrow (->) with respect to accessing structs? [5 pts]

```
typedef struct
{    int iArray[64];
     float fArray[64];} large;
void f1(large l) {
printf("sizeof(l) = %d\n", (int)sizeof(l));
}
void f2(large *l){
printf("sizeof(l) = %d\n", (int)sizeof(l));
}
large s;
for (int i=0; i<64; ++i) {
s.iArray[i] = 2*i;
s.fArray[i] = (float) i;
}
f1(s);
f2(&s);
```

Part c. What is the output of the code above? [5 pts]

Part c. [**10 pts**] How many bytes are required on the stack for parameter storage for f1()? f2()? Which is more efficient and why? [5 pts]