CIS 240 Fall 2018: Final

Please put all answers in the exam booklet and remember to number them clearly.

Question 1 {10 pts}

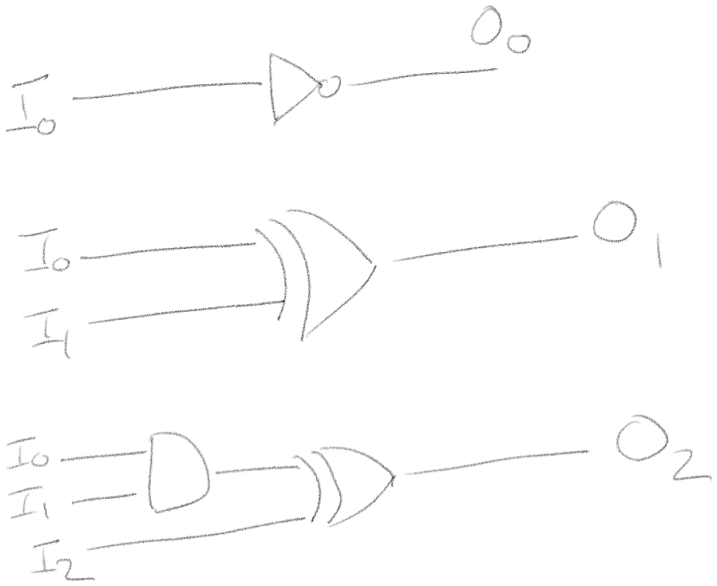
Your job is to design a gate level combinational circuit that takes as input a 3 bit unsigned number and produces as output that number plus 1 in a 3 bit unsigned output, so if the input number is 5 the output should be 6, if the input is 3 the output should be 4, if the input is 7 the output should be 0 because of wraparound. The three inputs should be labeled I_2, I_1 and I_0 where I_2 is the MSB and I_0 is the LSB. Similarly, the output bits must be labeled 0_2, O_1 and O_0.

Part 1 {4 pts}: Produce a table showing what the output should be for every possible input.
Part 2 {6 pts}: Design a circuit that performs the operation. Make sure to clearly label all inputs and outputs. You can use any gates you want, including xor gates. More points will be given for answers that use fewer gates so think carefully before implementing.

Answer:

I_2	I_1	I_0	0_2	0_1	0_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

From the table we notice the following patterns. The output O_0 is simply the inverse of the input I_0. The output O_1 is high whenever I_0 and I_1 differ so it can be computed using an xor gate on those two inputs. Lastly the MSB output O_2 is high if I_2 is 0 and (I_1 and I_0) are high or if I_2 is 1 and (I_1 and I_0) is low. This can be computed with another xor gate and an and gate. In a way this can be viewed as an expanded version of the simple bitwise half adder we studied in class where the carry input to each bit position is computed by explicitly anding all previous bits.



Question 2 (10 pts)
The following piece of C code was compiled with the lcc compiler.

```
int fact1 (int n) {
  if (n <= 0)
    return 1;
  else
    return n * fact1(n-1);
}

int fact2 (int n) {
  int i, output = 1;

  for (i=2; i<=n; ++i) {
    output *= i;
  }

  return output;
}
```

The resulting LC4 assembly code fragment is shown below. Five of the assembly instructions have been blacked out. Your job is to figure out what those 5 assembly instructions must have been.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;fact1;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .CODE
                .FALIGN
fact1
        ;; prologue
        STR R7, R6, #-2         ;; save return address
        STR R5, R6, #-3         ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0
        ADD R6, R6, #-1         ;; allocate stack space for local variables
        ;; function body
        LDR R7, R5, #3
        CONST R3, #0
        CMP R7, R3
        BRp L2_Final_2018_2
        CONST R7, #1
        JMP L1_Final_2018_2
L2_Final_2018_2
        LDR R7, R5, #3
        STR R7, R5, #-1
        ADD R3, R7, #-1
        ADD R6, R6, #-1
```

```
        MISSING_INSN_1; STR R3, R6, #0 ; place (n-1) on stack
        MISSING_INSN_2 : JSR fact1 ;; call the function
        LDR R7, R6, #-1     ;; grab return value
        ADD R6, R6, #1      ;; free space for arguments
        LDR R3, R5, #-1
        MUL R7, R3, R7
L1_Final_2018_2
        ;; epilogue
        ADD R6, R5, #0      ;; pop locals off stack
        ADD R6, R6, #3      ;; free space for return address, base pointer, and return
value
        STR R7, R6, #-1     ;; store return value
        LDR R5, R6, #-3     ;; restore base pointer
        LDR R7, R6, #-2     ;; restore return address
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;fact2;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .CODE
                .FALIGN
fact2
        ;; prologue
        STR R7, R6, #-2     ;; save return address
        STR R5, R6, #-3     ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0
        MISSING_INSN_3 : ADD R6, R6, #-2 ;; allocate space for 2 local variables
        ;; function body
        CONST R7, #1
        STR R7, R5, #-2
        CONST R7, #2
        STR R7, R5, #-1
        MISSING_INSN_4 : BRnzp L8_Final_2018_2 ;; branch to test
L5_Final_2018_2
        LDR R7, R5, #-2
        LDR R3, R5, #-1
        MUL R7, R7, R3
        STR R7, R5, #-2
L6_Final_2018_2
        LDR R7, R5, #-1
        ADD R7, R7, #1
        STR R7, R5, #-1
L8_Final_2018_2
        LDR R7, R5, #-1
        LDR R3, R5, #3
        CMP R7, R3
        MISSING_INSN_5 : BRnz L5_Final_2018_2 ;; branch back up to for body
```

```
        LDR R7, R5, #-2
L4_Final_2018_2
        ;; epilogue
        ADD R6, R5, #0      ;; pop locals off stack
        ADD R6, R6, #3      ;; free space for return address, base pointer, and return
value
        STR R7, R6, #-1     ;; store return value
        LDR R5, R6, #-3     ;; restore base pointer
        LDR R7, R6, #-2     ;; restore return address
        RET
```

Question 3 {10 pts}

In Question 2 the two functions, fact1 and fact2, compute the same quantity, the factorial of the input number. Which of the two compiled function contains more assembly instructions? On typical inputs, which version of the function would run faster (ie take fewer clock cycles) on an LC4 single cycle processor. Explain your answer briefly.

Answer:

The first part is a simple counting exercise, fact1 compiles to 27 instructions while fact2 compiles to 28 instructions so fact2 is longer. In practice however fact2 would run faster because fact1 executes by calling itself recursively, each recursive call involves a lot of extra work setting up and tearing down the stack just to perform a single subtraction and multiplication while fact2 uses only a single function call and computes the product iteratively.

Question 4 {5 pts}

Explain briefly why the lcc compiler inserts a .FALIGN assembly directive at the beginning of every compiled function.

Answer:

The key here is that functions are called by lcc using the JSR function. The JSR function takes the last 11 bits of the instruction left and left shifts them by 4 which is equivalent to multiplying by 16. LC4 does it this way to compensate for the fact that it cannot store an arbitrary address in a 16 bit instruction since some bits are needed for the opcode and other information. The end result is that JSR can only jump to addresses that are a multiple of 16. The .FALIGN directive tells the compiler to start the next code segment on such an address so that the JSR mechanism will work.

Question 5 {10 pts}

When the lcc compiler compiles an if statement like this:

```
if (some test) {
    code block inside if statement
}
```

it uses a BRANCH statement to implement the required control flow. What constraints, if any, does that place on the size of the code block inside the if statement?

Answer:

The Branch assignment uses the lower 9 bits of the instruction sign extended to compute the branch target. That means that the branch can reach any address that is within (PC + 1) + 255 to (PC + 1) -256. This means that the code block that we want to jump over should be no more than 255 LC4 instructions. If the clause you want to jump is longer than that this branch compilation strategy will not work.

Question 6 {10 pts}

Which of the following operations will cause PennSim to report an error?

1. Trying to execute code in the OS code section with privilege bit set to 0
    - Error : you need OS privilege to perform OS code
2. Trying to execute code in the user data section with privilege bit set to 1
    - Error : in LC4 you are not allowed to execute data as code
3. Trying to store into OS data section with privilege bit set to 1
    - No Problem : The OS can store into OS data section
4. Trying to execute code in USER code section with privilege bit set to 1
    - No Problem : The OS can execute both OS and USER code
5. Trying to store into USER data section with privilege bit set to 1
    - No Problem : The OS can access user data and does.

Question 7 {15 pts}

Your cousin, Crazy Eddie, has decided to add a new instruction to the LC4 instruction set with an opcode of `0011`. Here are the settings for the control signals for this operation.

	PCMux.CTL	rsMux.CTL	rtMux.CTL	rdMux.CTL	regFile.WE	regInputMux.CTL	NZP.WE	DATA.WE	Privilege.CTL	ALUInputMux.CTL	ALU.CTL
FOO	1	1	1	1	1	0	1	1	2	1	6

Your LC4 microprocessor is about to execute the following 16 bit instruction
`0011011000000001`

The table below shows the state of the LC4 processor right before this new instruction is executed. Your job is to fill in the table to indicate what the state will be after the instruction is executed. All values are given in hex, your answers should be as well. Please put the answer in your answer booklet.

	PC	PSR	R0	R1	R2	R3	R4	R5	R6	R7
Before	001F	0002	0003	0011	0022	0009	0017	0101	0013	004F
After	????	????	????	????	????	????	????	????	????	????

Are there any other relevant changes to the machine state?

Explain how this new instruction could be used to accelerate the implementation of the memset() function on an LC4 system. That is explain how would you use this new instruction to make a faster implementation of memset than you could before.

```
void *memset ( void * ptr, int value, size_t num );
```

Fill block of memory:
Sets the first *num* slots of the block of memory pointed by *ptr* to the specified *value*.

Answer:

The control settings will cause the following things to happen. The PC will be set to PC+1 on the next instruction. The value in R7 will be added to the sign extended immediate 5 field that value will be written back into R7 and the NZP bits will be

adjusted based on the sign of that result. Furthermore the computed value will be passed as an address to the Data Memory and the and that value will be passes as an address to the Data memory and the RT value will be written into memory at that location. The RT value will be pulled from bits 11-9 in the instruction.

The instruction that we will execute has an opcode of 0011, an rt field of 3 and an immediate field value of +1. The result of executing this function will be.

	PC	PSR	R0	R1	R2	R3	R4	R5	R6	R7
Before	001F	0002	0003	0011	0022	0009	0017	0101	0013	004F
After	0020	0001	0003	0011	0022	0009	0017	0101	0013	0050

Furthermore, the value in R3 which is 9 will be written into the data memory at address x0050. Note the PSR is updated to 0001 since the result in R7 is positive.

The interesting thing about this new instruction is that it uses R7 as a base address just like a STR instruction but it updates that address by the immediate field as it executes. This could be useful if you were implementing a loop that filled in a section of memory as in the memset operation. Instead of having one instruction to store the value and another to update the base pointer you could fuse this into a single instruction that did both jobs. This should speed up the operation. Here is the idea in pseudo code

```
Initialize R7 to ptr
Initialize counter to num
While (counter--) {
   Use Foo to write value into memory and update R7 to point to next entry
}
```

Question 8 {10 pts}

You are tasked with writing a program that will maintain a list of integers sorted in ascending order. We are providing some C code that does this but we have blanked out some of the lines. Your job is to tell us what these missing lines should be. HINT: The missing lines are all assignment statements.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct list_elt_tag {
  int elt;
  struct list_elt_tag *next;
} list_elt;

list_elt *LIST = NULL; // Initialize the list to empty

// Insert the number n on the list sorted in ascending order
void insert_entry_sorted (int n) {
  list_elt *entry, *new_entry;

  MISSING_LINE_1; new_entry = malloc(sizeof(list_elt)); // allocate space

  if (new_entry == NULL) exit(2);

  new_entry->elt = n;

  if ((LIST == NULL) || (LIST->elt >= n)) {
    MISSING_LINE_2; new_entry->next = LIST;
    MISSING_LINE_3; LIST = new_entry;
  } else {
    entry = LIST;

    while ((entry->next) && (((entry->next)->elt) < n)) {
      MISSING_LINE_4; entry = entry->next;  // get the next entry in the list
    }

    MISSING_LINE_5; new_entry->next = entry->next;
    MISSING_LINE_6; entry->next = new_entry;
  }
}
```

Note that for lines 2 and 3 and 5 and 6 the order of operations is critically important. If you do it in the wrong order you end up with memory leaks and segmentation faults.