ZJU-UIUC Institute
First Midterm Exam, ECE 220

Monday 1 November 2020

Name (pinyin and Hanzi):

Student ID: Lab TA Name:

- **Be sure that your exam booklet has THIRTEEN pages. ALL THIRTEEN pages must be returned when you hand in the exam.**
- **Write your name, student ID, and lab section TA name on this page above.**
- **Do not tear the exam apart, but you can remove the last THREE pages which are two scratch pages for drafts, and one page with RTL for LC-3 instructions (except JSRR).**
- **Copies of Patt & Patel's Appendix A are also available during the exam.**
- **This is a closed book exam. You may not use a calculator.**
- **Challenge problems are marked with ***.**
- **You are allowed one handwritten or printed A4 sheet of notes (both sides).**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**

Problem 1	20 points	________________
Problem 2	20 points	________________
Problem 3	20 points	________________
Problem 4	20 points	________________
Problem 5	19 points	________________
Correct Room	1 point	________________
Total	100 points	________________

Problem 1 (20 points): Short Answer

A. **(4 points)** An LC-3 assembly program may contain both RET and JMP R7 instructions. Does an LC-3 processor do anything differently when processing either of these two instructions? Explain your answer **USING 20 OR FEWER WORDS**.

NO, they both set PC to the value contained in R7

B. **(4 points)** The C function below contains EXACTLY ONE BUG. Mark the bug directly **ON THE CODE** and indicate how it can be corrected.

```
void hail()
{
    int start, step;
    step = 0;
    printf ("Enter a start number: ");
    if (1 != scanf ("%d", start)) {
        start = 42;
    }
    step = (start >> 2);
    while (0 < --step) {
        if (1 == (start % 2)) {
            start = 3 * start + 1;
        } else {
            start = start / 2;
        }
        printf ("%d,", start);
    }
}
```

it should be &start

start=21

C. **(4 points)** Assuming that the bug in the function **hail** in **Part B** has been corrected, decide which of the following possible inputs to the function **hail** generates the output string “64, 32, 16, 8, 4,” (without quotes). **CIRCLE EXACTLY ONE ANSWER.**

>>2 step

i. 128 32

ii. 64 16

iii. 21 5

[✓ iv.] None of the above.

0 < 4
start= 64
64
0 <3
start=32
32
0 < 2
start:16
16
0 <1
start=8
8

Problem 1, continued:

D. **(4 points)** Read the following C code. Then written down the program output on the line below the code.

```
int i = 0;  // PART E ASKS ABOUT THIS DECLARATION

void foo (int x)
{
    int a = 0;
    static int b = 0;

    a += 5;
    b += 5;
    i += 5;

    if (x == 6) { printf ("a = %d, b = %d, i = %d", a, b, i); }
}

int main ()
{
    // PART E SUGGESTS MOVING THE DECLARATION TO THIS LINE

    for (int x = 0; x < 7; x++) { foo (x); }

    return 0;
}
```

x=0 b=5 i=5
x=1 b=10 i=10
x=6 b=35 i=35

Output: a= 5 , b= 35, i = 35

E. **(4 points)** What happens if the declaration of the variable **`i`** at the start of the code in **Part D** is moved into **`main`**? (See the comments in the code for exact lines.) Will the code with this change still compile? If so, will the printed value of variable **`i`** change, and if yes, to what value?

~~Yes, no~~ Won't

(Do not write below this line.)

Problem 2 (20 points): Elevator Design with LC-3

You must implement a control system for a high-rise building elevator using an LC-3 processor. The building contains 16 floors numbered 0 to 15. On each floor, buttons in the corridor allow a person to call the elevator: floors 0 to 14 provide an Up button, and floors 1 to 15 provide a Down button. Inside the elevator there are other 16 buttons to choose the target floor.

The status of all 2×15+16=46 buttons have been memory-mapped into the addresses xFE10 to xFE3D. The first 15 addresses (starting at xFE10) are used for the Up buttons on each floor from 0 to 14 (outside the elevator) — in that order, with floor 0 first, floor 1 second, and so forth. The next 15 addresses are used for the Down buttons on each floor from 1 to 15 — in that order. The remaining 16 addresses are used for the buttons inside the elevator, from floor 0 to 15 — in that order.

For each button, the corresponding memory address may be read to determine whether or not the button has been pressed (do not concern yourself with how the pressed buttons are then reset in this problem). When a button's memory address is read, bit 15 (the left-most bit) of the memory address is set to a 1 if the corresponding button has been pressed. Bits 0 to 14 of the I/O addresses are undefined and must not be used to control the elevator.

Using NO MORE LINES THAN PROVIDED ON THE NEXT PAGE (you need not even use all these lines)**,** write the subroutine NEXT_UP, which finds the elevator's next stop when it is moving upward. Details are provided in the subroutine header on the next page. For full credit, **be sure to include comments** explaining your code (trivial comments such as "Add 1 to R2" will be ignored). Your subroutine must not read the status of any Down button; they are not relevant to this subroutine's result.

(Do not write below this line. For credit, write your code on the next page!)

Problem 2, continued:

```
; NEXT_UP - find the next floor at which the elevator must stop,
;           assuming that the elevator is currently moving upward
; INPUT: R1 - the floor that the elevator HAS JUST DEPARTED UPWARD
;              (assume that 0 <= R1 < 15)
; OUTPUT: R1 - the floor at which the elevator must stop next
; REGISTERS: all registers are caller-saved
; SIDE_EFFECTS: none

NEXT_UP     ; write the subroutine's code and data below
```

Problem 3 (20 points): Checking a Fibonacci-like Sequence with a Stack

We define a sequence **A[n]** to be a **Fibonacci-like mod 2^{16} (abbreviated FLM)** iff for every integer **n > 2, (A[n] = A[n – 1] + A[n – 2]) mod 2^{16}.** In this problem, the sequence **A[n]**, n=1,2,3,…,N has been pushed in order onto a stack (i.e. A[0] is pushed first, then A[1] and so on). You must write the code to determine whether or not **A[n]** on the stack is **FLM**. The following subroutine has been provided to help you.

```
; CHECK_ITEM - checks whether the three integers on top of the stack satisfy
;              the FLM property, and pops the top integer from the stack
; Input:  R6 - top of the stack
; Output: R0=1 if the top element is the sum of the next two, and 0 otherwise
;         R6 - top of the stack (after the top integer on the stack is removed)
; Registers: R1, R2, R3 and R7 are caller-saved; R4 and R5 are callee-saved

CHECK_ITEM  AND R0,R0,#0
            LDR R1,R6,#0      ; first integer
            LDR R2,R6,#1      ; second integer
            LDR R3,R6,#2      ; third integer
            ADD R6,R6,#1      ; pop first integer from stack
            NOT R1,R1         ; negate first integer
            ADD R1,R1,#1
            ADD R1,R1,R2      ; add second and third integers
            ADD R1,R1,R3
            BRnp NOTFLM       ; not zero?  If so, not FLM.
            ADD R0,R0,#1      ; FLM, so R0=1 on return
NOTFLM      RET
```

USING NO MORE LINES THAN PROVIDED ON THE NEXT PAGE (you need not use all), write the subroutine CHECK_STACK to determine whether a sequence **A[n]** pushed onto the stack (as specified earlier, with **A[0]** pushed first, **A[1]** second, and so forth) is **FLM**. Your subroutine must use the CHECK_ITEM subroutine**.** You may assume that the sequence **A[n]** has length at least **N=1**. Sequences of length **N < 3** are trivially FLM. The length **N** of the sequence A[n] corresponds to the number of values on the stack at the start of the subroutine. Your subroutine must return the correct answer in register R4. **The stack follows the conventions as used in the lectures.** Finally, while there are no requirements on the final value of the stack pointer (register R6), your code must not read past the base of the stack, and therefore your subroutine must not call CHECK_ITEM if fewer than three values remain on the stack. For full credit, **be sure to include comments** explaining your code (trivial comments such as "Add 1 to R2" will be ignored).

(Do not write below this line. For credit, write your code on the next page!)

Problem 3, continued:

```
; CHECK_STACK - process a stack of non-negative integers,
;               check if the sequence on the stack is FLM
; Input:  R6 - top of the stack
;         R5 - base of the stack
; Output: R4=1 if the sequence on the stack is FLM, or 0 otherwise
; Registers: all registers are caller-saved

CHECK_STACK ; write the subroutine's code and data below
```

Problem 4 (20 points): Fun with Arrays and Lists in LC-3

Consider the following LC-3 program.

```
 1    .ORIG x3000
 2    LEA R1,SOURCE_NUMBERS
 3    AND R2,R2,#0
 4    LD R0,SOURCE_LENGTH
 5 START_LOOP
 6    BRz DONE
 7    ADD R2,R2,R2
 8    ADD R3,R2,R2
 9    ADD R3,R3,R3
10    ADD R3,R3,R2
11    LDR R4,R1,#0
12    ADD R2,R4,R3
13    ADD R1,R1,#1
14    ADD R0,R0,#-1
15    BRp START_LOOP
16 DONE
17    ST R2,TARGET_NUMBER
18    HALT
19 TARGET_NUMBER  .FILL x0000
20 SOURCE_LENGTH  .FILL x0003
21 SOURCE_NUMBERS .FILL x0004
22                .FILL x0003
23                .FILL x0002
24                .FILL x0001
25                .FILL x0000
26 .END
```

A. **(8 points)** What is the value stored at address TARGET_NUMBER after executing the program? Write your answer in **either hexadecimal or decimal** form.

Stored value = ______________________________

B. **(4 points)** Lines 20 to 26 of the program in **Part A** are replaced with these lines:

```
20 SOURCE_LENGTH  .FILL x0007
21 SOURCE_NUMBERS .FILL x0001
22                .FILL x0001
23                .FILL x0001
24                .FILL x0001
25                .FILL x0001
26                .FILL x0001
27                .FILL x0001
28 .END
```

The modified program is then executed, but now the value stored at address TARGET_NUMBER does not match the programmer's expectations. **USING 20 OR FEWER WORDS**, explain why.

Problem 4, continued:

C. *****(8 points)** Later this semester, you will learn about linked lists. A linked list is a data structure consisting of an ordered sequence of nodes. Each node simply contains some data and also an address to the next node in the list. Each node is stored in the contiguous memory locations, but different nodes may be stored in unrelated parts of memory, and in any order. For this problem, a node in the list is made up of two adjacent memory locations. For example:

```
NODE1    .FILL x0004 ; data in node 1
         .FILL NODE2 ; address of node 2 (next node in the list)
```

The first memory locations of nodes in the linked list can be used to store the consecutive values at SOURCE_NUMBERS in the original code. The second memory locations of nodes in the linked list are used to find the memory location of the next nodes in the list, which allows to read the sequence of data stored in the linked list.

Using NO MORE LINES THAN PROVIDED BELOW (you need not use all), complete the new version of the program to traverse the list starting at the first node in order to calculate TARGET_NUMBER.

```
         .ORIG x3000

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________

START_LOOP
         BRz DONE
         ADD R2,R2,R2
         ADD R3,R2,R2
         ADD R3,R3,R3
         ADD R3,R3,R2

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________

         BRp START_LOOP
DONE
         ST R2,TARGET_NUMBER
         HALT
TARGET_NUMBER .FILL x0000
SOURCE_LENGTH .FILL x0003
SOURCE_NUMBER .FILL x3300 ; address of the first node
         .END
```

C

B

R5 -> A

2nd stack

pre stack
rt addr
rt val
X
Y
Z

Problem 5 (19 points): Understanding Compiled C Code

The LC-3 code below corresponds to the output of a compiler for the C function **foo**.

```
FOO     ADD R6,R6,#-5               A, B
        STR R5,R6,#2
        ADD R5,R6,#1
        STR R7,R5,#2
        LDR R0,R5,#6
        ADD R6,R6,#-1               C= Z
        STR R0,R6,#0
        JSR ONEFUNC       ; call of the subroutine "onefunc" in C
        LDR R0,R6,#0
        ADD R6,R6,#2               B = ONEFUN(Z)
        STR R0,R5,#-1
        LDR R0,R5,#6
        AND R0,R0,#7               R0= (Z & #7)-4
        ADD R0,R0,#-4
        BRnp TESTF                 if R0 = 0
        LDR R0,R5,#4 X
        LDR R1,R5,#-1 B            A= B+X
        ADD R0,R0,R1
        STR R0,R5,#0
        BRnzp DONE
TESTF   LDR R0,R5,#-1              B→R0
        ADD R6,R6,#-1  R6→B
        STR R0,R6,#0
        LDR R0,R5,#6   Z
        ADD R6,R6,#-1              C= Z
        STR R0,R6,#0
        LDR R0,R5,#5
        ADD R6,R6,#-1
        STR R0,R6,#0
        JSR TWOFUNC       ; call of the subroutine "twofunc" in C
        LDR R0,R6,#0
        ADD R6,R6,#4
        STR R0,R5,#0
DONE    LDR R0,R5,#0               return A
        STR R0,R5,#3
        LDR R7,R5,#2
        LDR R5,R5,#1
        ADD R6,R6,#4
        RET
```

Write the C code corresponding to the function **foo** from which a non-optimizing compiler might have produced the LC-3 code above. For function parameters, choose names from X, Y, and Z. For local variables, choose names from A, B, and C. (There are no more than three variables of either type.) All variable types are **int** (16-bit 2's complement).

Scratch page 2: Please write your name and student ID at the top of this page if you tear it off.

Scratch page 1: Please write your name and student ID at the top of this page if you tear it off.