ZJU-UIUC Institute
First Midterm Exam, ECE 220

Thursday 18 October 2018

Name (pinyin and Hanzi):

SOLUTION IS IN RED

Student ID:

- **Be sure that your exam booklet has TEN pages.**
- **Write your name and Student ID on the first page.**
- **Do not tear the exam apart other than to remove the reference sheet.**
- **This is a closed book exam. You may not use a calculator.**
- **Challenge problems are marked with ***.**
- **You are allowed one handwritten A4 sheet of notes (both sides).**
- **The last page of the exam gives RTL for LC-3 instructions (except JSRR). Copies of Patt & Patel's Appendix A are also available during the exam.**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**
- **Don't panic, and good luck!**

Problem 1	20 points	______
Problem 2	16 points	______
Problem 3	24 points	______
Problem 4	20 points	______
Problem 5	20 points	______
Total	100 points	______

(1) Use R_1, R_2 as counters with value #640 and #480 respectively.

7 (2) Use R_3 to hold value of pixel and R_4, R_5 as target two color

Because LC-3 has only 512 memory-mapped I/

Problem 1 (20 points): Short Answer Questions

1. **(12 points)** While working as an intern at a company developing self-driving vehicles, you are tasked with writing code for the anti-lock braking system (ABS) for 18-wheel trucks. Each truck has six brakes (four brakes control four wheels each, and two brakes control one wheel each).

The ABS code must check whether the human is pressing the brake pedal and whether the tires are spinning more slowly than the truck is moving (all of these values are provided to your code). If both conditions hold, the code must turn off all six brakes, pause for 100 milliseconds, and then turn on all six brakes again.

Using **NO MORE THAN 10 WORDS**, describe each of the following. Answering with code will earn no credit.

a. **(4 points)** One subtask for which you should use a sequential decomposition.

turn off ~~or~~ pause on all six brakes

b. **(4 points)** One subtask for which you should use a conditional decomposition.

check whether both conditions hold

c. **(4 points)** One subtask for which you should use an iterative decomposition.

pause for 100 milliseconds

turn on / turn ~~pause~~ off

2. **(4 points)** A friend wants to add a 640×480-pixel monochrome (two-color) graphics adapter to his LC-3-based computer. Using **NO MORE THAN 25 WORDS**, including any necessary calculations, explain how to accomplish this goal, or why the goal is impossible.

R1 R2

(640 × 480 pixels × 1 bit/pixel) / 16 bits/memory location = 19,200 memory locations

LC-3 has only 512 (xFE00 to xFFFF) usable for memory-mapped I/O, so …

(1) Cannot map individual pixels without changes to design (as students know it), or
(2) Can change board design (hardware for I/O) to expand memory-mapped I/O region, or
(3) Can use one or two ports with address / data I/O model [not something students have seen, but an acceptable answer].

LC-3 has only 512 usable for memory-mapped I-O, so we cannot map individual pixels to the display

0

R7

3. **(4 points)** A friend writes an LC-3 subroutine to calculate ⌊sqrt (**R7**)⌋, the largest integer that is not greater than the square root of `R7`.

R_1^2

Using **NO MORE THAN 15 WORDS**, explain why your friend's subroutine cannot work correctly.

Because to use the subroutine, he need to use JSR, which changes R7

Problem 2 (16 points): Understanding LC-3 Code

The LC-3 subroutine **MYSTERY** appears below. Read it, then answer the questions below.

```
MYSTERY  LD      R1,VALUE
         AND     R4,R4,R1
         AND     R3,R3,#0
LOOP1    ADD     R4,R4,#-16
         BRn     FINISH1
         ADD     R3,R3,#1
         BRnzp   LOOP1
FINISH1  LEA     R2,DATA
         ADD     R2,R2,R3
         LDR     R0,R2,#0
         AND     R6,R6,#0
         ADD     R6,R6,#1
LOOP2    ADD     R4,R4,#1
         BRzp    FINISH2
         ADD     R6,R6,R6
         BRnzp   LOOP2
FINISH2  AND     R5,R0,R6
         RET
VALUE    .FILL   x007F
DATA     .FILL   x0000
         .FILL   x0000
         .FILL   x0000
         .FILL   x0000
         .FILL   x7FFF
         .FILL   xFFE0
         .FILL   x7FFF
         .FILL   xFFE0
```

R1 = x007F R4 = x0040 R3 = 0 R4 = -16 R3 = 4 R0 = x7FFF R6 = x0001 R4 = -15 R5 = 0

R4 = x005E R4 = -2 R3 = 5 xFFE0

1. Assuming that **R1=x00F2, R2 contains bits, and R4=x0040** at the start of the **MYSTERY** subroutine, fill in the blanks below with final register values after the **RET** instruction executes. For any register for which you cannot know the value, write "bits."

R0: x7FFF R3: x0004 R6: ~~xF000~~ ~~x0000~~ x8000 R7: bits

2. Assuming that **R1 contains bits, R2=xABCD, and R4=xCFDE** at the start of the **MYSTERY** subroutine, fill in the blanks below with final register values after the **RET** instruction executes. For any register for which you cannot know the value, write "bits."

R0: xFFE0 R3: x0005 R6: ~~x0020~~ x0002 R7: bits

3. Assuming that **R1=x7301, R2=x1234, and R4 contains bits** at the start of the **MYSTERY** subroutine, fill in the blanks below with final register values after the **RET** instruction executes. For any register for which you cannot know the value, write "bits."

R0: bits R3: bits R6: bits R7: bits

4. *** Using NO MORE THAN 30 WORDS, explain what MYSTERY does.

~~0111 0011 0000 0001~~

0000 0000 0111 1111

MYSTERY count

check if R4[6:0] are in a set
return R5=0 for no or non-zero

15f 127 = ~~162~~ 71 / ~~16~~ 8   8 64   14   128

for yes

Problem 3 (24 points): Using a String as a Stack

1. **(10 points)** Given in **R4** a pointer to a NUL-terminated ASCII string consisting of hexadecimal digits (0-9 and A-F), write a sequence of LC-3 instructions to do the following:
   - point **R6** to the start of the given string,
   - change the NUL at the end of the string to an ASCII '0' (x0030), and
   - point **R2** to the memory location after the NUL.

You **may use all of the LC-3 registers**.

The **string may be empty**—in other words, the string may contain no hexadecimal digits.

The **string will not contain any ASCII characters other than 0 (x0030) through 9 (x0039) and A (x0041) through F (x0046)**.

Use **NO MORE THAN TEN MEMORY LOCATIONS**, including storage for any data needed.
**** Using more memory than TEN LOCATIONS will earn NO CREDIT. ****

Here's an example. Notice that, after the code executes, the string looks like a stack! You will use that fact in the next problem.

at start of code	address	contents	after code executes
R4 points here →	**x4123**	**x0032 '2'**	← **R6 points here**
	x4124	**x0041 'A'**	
	x4125	**x0000 NUL**	← **NUL replaced with x0030 '0'**
	x4126	**bits**	← **R2 points here**

(Include comments for more partial credit.)

Write your code here...

```
       ADD   R6, R4, #0     ; let R6 points to top of stack
loop   LDR   R0, R4, #0     ; check if the value pointed by R4 is 0
       BRz   Change         ; if yes, change it
       ADD   R4, R4, #1
       BRnzp loop
Change LDR   R0, data       ; change it with x0030
       STR   R0, R6, #0
       ADD   R2, R6, #1     ; let R2 points to base
data   .FILL x0030
```

Write any data that you need here…

Problem 3, continued:

2. **(14 points)** Now you must write a subroutine to make use of the "stack" produced by **part (1)**. Your subroutine, SUM_HEX, must use the CONVERT subroutine described below to convert the hex digits into 2's complement, and must use the STACK_ADD subroutine described below to add pairs of 2's complement values until only one remains on the stack. The subroutine should then return, leaving the 2's complement sum of the digits on the top of the stack (pointed to by **`R6`**). See the description below for more details on your subroutine.

These subroutines are provided to you:

```
CONVERT – convert a hexadecimal digit from ASCII to 2's complement
Input: R0 – ASCII character representing a hexadecimal digit
Output: R3 – value of R0 in 2's complement
All registers other than R3 and R7 are callee-saved.

STACK_ADD – add two 2's complement values on top of a stack (pops two values,
            adds them, and pushes the sum back onto the stack)
Input: R6 – pointer to top of stack
Output: R6 – pointer to top of stack after operation
All registers other than R6 and R7 are callee-saved.  R6 changes as described.
```

You must write the following subroutine:

```
SUM_HEX – convert and sum a stack of hexadecimal ASCII digits into a
          2's complement sum
Inputs: R2 – base of stack
        R6 – top of stack
Output: R6 – top of stack (must be one address less than original base), which
             points to the sum of the digits
All registers are caller-saved.
```

***** WRITE YOUR CODE ON THE NEXT PAGE *****

Your subroutine **may use all LC-3 registers** (all registers are caller-saved).

Use **NO MORE THAN TWENTY-FOUR MEMORY LOCATIONS**, including storage for any data needed. **** Using more memory than TWENTY-FOUR LOCATIONS will earn NO CREDIT. ****

(Include comments for more partial credit.)

Problem 3, continued: **(subroutine specifications duplicated for your convenience)**

These subroutines are provided to you: **(14 points)**

CONVERT – convert a hexadecimal digit from ASCII to 2's complement
Input: R0 – ASCII character representing a hexadecimal digit
Output: R3 – value of R0 in 2's complement
All registers other than R3 and R7 are callee-saved.

STACK_ADD – add two 2's complement values on top of a stack (pops two values, adds them, and pushes the sum back onto the stack)
Input: R6 – pointer to top of stack
Output: R6 – pointer to top of stack after operation
All registers other than R6 and R7 are callee-saved. R6 changes as described.

2 1

1

base

You must write the following subroutine:

SUM_HEX – convert and sum a stack of hexadecimal ASCII digits into a 2's complement sum
Inputs: R2 – base of stack
R6 – top of stack
Output: R6 – top of stack (must be one address less than original base), which points to the sum of the digits
All registers are caller-saved.

```
SUM_HEX  ST   R7,SR7
         ADD  R2,R6,#0
loop-1   LDR  R0,R6,#0
         JSR  CONVERT      < STR R3,R6,#0
         ADD  R6,R6,#1
         NOT  ~~R6,R6~~ R1,R6
         ADD  ~~R1,R6~~ R1,R1,R2
         ADD  R1,R1,#1
         BRp  loop-1
         ADD  R6,R2,#0
loop_2   JSR  STACK_ADD    < ADD R6,R6,#1
         NOT  R1,R6
         ADD  R1,R1,R2
         ~~ADD R1,R1,#-1~~
         BRp  loop_2
         ADD  R6,R6,#-1    < LD R7,SR7
         RET
```

SR7 .FILL x0000

1 3

5<=1 x=1

~~5<=2 x=2~~

5<=3 x=2 7 9

~~5<=4 x=4~~

Problem 4 (20 points): Basics of C Programming

1. **(8 points)** The two C programs shown below are identical except for the line marked by the comments, “DIFFERS!” Write the output of each program on the blank line below the corresponding code.

10

```
#include <stdio.h>
int main ()
{
    int32_t x = 0;
    int32_t i = 3;
    for (i = 0; 9 > i; i++) {
        if (5 <= ++i) {
            continue; // DIFFERS!
        }
        x++;
    }
    printf ("x: %d, i: %d\n",
            x, i);
    return 0;
}
```

6 8 10

5 7 9

x: 2, i: ~~9~~ 10

```
#include <stdio.h>
int main ()
{
    int32_t x = 0;
    int32_t i = 3;
    for (i = 0; 9 > i; i++) {
        if (5 <= ++i) {
            break; // DIFFERS!
        }
        x++;
    }
    printf ("x: %d, i: %d\n",
            x, i);
    return 0;
}
```

x: 2, i: 5

2. Read the C function below, then answer the questions.

```
void foo (int32_t x)
{
    switch ((x < 4) - ((x < 5) ? 0 : 1)) {
        case -1:
            printf ("A");
            break;
        case 0:
            printf ("B");
        case 1:
            printf ("C");
            break;
        default:
            printf ("D");
            break;
    }
    return;
}
```

0 − 0 = 0

1 − 0 = 1

0 − 1 = −1

a. **(4 points)** What is the function’s output when parameter **x** is equal to 4? BC

b. **(3 points)** For what values(s) of parameter **x**, if any, does the function output `D`? No

Problem 4, continued:

3. **(5 points)** Read the program below, then write the program's output on the blank line below the code.

```
#include <stdio.h>

int32_t
bar (int32_t x, int32_t y)
{
    if (y <= x) {
        x = x + y;
    }
    return x;
}

int
main ()
{
    int32_t y = 3;
    int32_t c = 6;

    {
        int32_t x = 2;

        c = bar (y, x);
        printf ("x: %d, y: %d, c: %d\n", x, y, c);
    }

    return 0;
}
```

c= bar(3,2)
= 5

2 <= 3 ✓
x= 3+2=5

Output: x: 2, y: 3, c: 5

Problem 5 (20 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       ; linkage + two local variables
        STR     R5,R6,#2
        ADD     R5,R6,#1
        STR     R7,R5,#2        ; end of stack frame setup
        LDR     R0,R5,#4        ; R0 ← X & Y & Z
        LDR     R1,R5,#5
        AND     R0,R0,R1
        LDR     R1,R5,#6
        AND     R0,R0,R1
        STR     R0,R5,#-1       ; A ← R0
        LDR     R0,R5,#-1       ; if (0 != A)
        BRz     LABEL
        LDR     R0,R5,#4        ; (then) push X - Y
        LDR     R1,R5,#5
        NOT     R1,R1
        ADD     R1,R1,#1
        ADD     R0,R0,R1
        ADD     R6,R6,#-1
        STR     R0,R6,#0
        LDR     R0,R5,#-1
        ADD     R6,R6,#-1
        STR     R0,R6,#0
        JSR     FUNC_ONE        ; call this subroutine "func_one" in C
        LDR     R0,R6,#0
        ADD     R6,R6,#3
        STR     R0,R5,#0
        BRnzp   DONE
LABEL   LDR     R0,R5,#4
        ADD     R6,R6,#-1
        STR     R0,R6,#0
        LDR     R0,R5,#6
        ADD     R6,R6,#-1
        STR     R0,R6,#0
        JSR     FUNC_TWO        ; call this subroutine "func_two" in C
        LDR     R0,R6,#0
        ADD     R6,R6,#3
        STR     R0,R5,#0
DONE    LDR     R0,R5,#0
        STR     R0,R5,#3
        LDR     R7,R5,#2
        LDR     R5,R5,#1
        ADD     R6,R6,#4
        RET
```

R0 func
R0 D
R0 →
R5 →
R5
R7
use

R0 ←
A
R5 → variable
R5 → previous stack
R7 → return addr
return value
para

~~base~~

R0=A=X
R1=B=Y
A=A&B=X&Y
R1=Z
A=A&B=X&Y&Z
if A=0
A=X Z
return FUNC_TWO(X,Z)
return (A-B, X&Y&Z)
X-Y

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

```
int foo (int X, int Y, int Z) {
    int A = (X & Y & Z), B;
    if (0 != A) {
        B = func_one(A, X - Y);
    } else {
        B = func_two(Z, X);
    }
    return B;
}
```

int foo (int X, int Y, int Z) {
int A, B, C;
if (X & Y & Z) == 0 {
A = FUNC_TWO(X, Z);
return A;
} else {

```
        B = X-Y;
        C = X&Y&Z;
        A = FUNC_ONE(B,C);
        return A;
    }
}
```