University of Illinois at Urbana-Champaign
First Midterm Exam, ECE 220 Honors Section

Thursday 15 February 2018

Name: SOLUTION IS IN RED

Net ID:

- **Be sure that your exam booklet has ELEVEN pages.**
- **Write your name and Net 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 <u>not</u> use a calculator.**
- **You are allowed one handwritten 8.5×11-inch sheet of notes (both sides).**
- **The last page of the exam gives RTL for LC-3 instructions (except JSRR; given JSRR BaseR, the RTL is PC ← BaseR, R7 ← PC). 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	24 points	_____
Problem 2	20 points	_____
Problem 3	20 points	_____
Problem 4	16 points	_____
Problem 5	20 points	_____
Total	100 points	_____

Problem 1 (24 points): Short Answer Questions

1. **(12 points)** A fellow ECE 220 student is attempting to write a program that calculates either the product or the sum of numbers from 1 to N. The user first types the value of N, which must be a positive integer. The student's code reads the number N from the keyboard using the READ_N subroutine, which handles any non-numeric input as well as overflow, returning only when the human user has entered a positive value for N. Next, the user presses either the asterisk key ("*") or the plus key ("+") to tell the program whether to calculate the product or the sum of the numbers from 1 to N, respectively. The result is then stored in memory.

Using **NO MORE THAN 15 WORDS**, describe each of the following. Answering with code will earn no credit.

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

calculate the product or the sum and store result

read N, read operation, compute, store

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

check if input overflow and check which of "*" and "+" is pressed

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

calculate the product or the sum of the numbers from 1 to N

2. **(4 points)** Fill in the blanks below so that the resulting LC-3 subroutine, MYOUTPUT, writes an ASCII character from R5 out to the display using memory-mapped I/O (in other words, do not use TRAPs nor other subroutine calls). You may change R0 and/or R7 if necessary. All other registers are callee-saved.

```
MYOUTPUT  ___ LDI R0, DDR ,#0
          _   BRzp  MYOUTPUT
          _   STI   R0, DDR, #0
          _____________________
          _____________________

          RET
DSR       .FILL xFE04
DDR       .FILL xFE06
```

LDI R0,DSR,
BRzp MYOUTPUT
STI R5, DDR

Problem 1, continued:

3. **(8 points)** Consider the multiplication subroutine shown below.

```
; Takes two positive integers and returns the product
MULT  AND R1, R1, #0
      ADD R3, R3, #0
      BRz DONE
LOOP  ADD R1, R1, R2
      ADD R3, R3, #-1
      BRp LOOP
DONE  RET
```

R1, R3

a. **(2 points)** Which registers hold the operands for multiplication by the subroutine? R2, R3

b. **(2 points)** In which register does the subroutine return the product? R1

c. **(2 points)** Which registers in the MULT subroutine are callee-saved? R0, R2, R4, R5, R6

d. **(2 points)** Which registers in the MULT subroutine are caller-saved? R1, R3, R7

R2-R3

Problem 2 (20 points): Writing LC-3 Code

In this problem, you must write a subroutine, SWAPBYTES, that swaps the two 8-bit bytes held in one 16-bit LC-3 register.

For example, given the initial value xCDAB, your subroutine must move xCD into the low 8 bits and xAB into the high 8 bits to produce the value xABCD. Similarly, given the initial value xEBEC, your subroutine must swap the xEB with the xEC to produce the 16-bit value xECEB.

The call interface for SWAPBYTES is as follows:

Input: **R1**, a 16-bit value (2 bytes)
Output: **R2**, a 16-bit value (2 bytes) with bytes swapped
All registers are caller-saved.

Requirements for your subroutine:

- Use at most 25 instructions (excludes labels, .ORIG, .END,. FILL, .BLKW).
  Any code after the 25th instruction will not be graded.
  It is possible to finish this problem with 12 instructions or less.
- You must use a loop(s) in your code. Manual repetition will receive **ZERO** credit.
- Briefly comment your code and describe how each register is used.
  You are not required to comment every line.

Write your subroutine on the next page.

(Use the back of this page if you need more space.)

Problem 2, continued: **(call interface and examples reproduced for your convenience)**

```
SWAPBYTES
Input: R1 - 2 bytes (16 bits)
Output: R2 - 2 bytes (16 bits) with bytes swapped
All registers are caller-saved.

Example 1:                              Example 2:
R1 - 0xCDAB → R2 - 0xABCD               R1 - 0xEBEC → R2 - 0xECEB
```

```
.ORIG x3100     ; R3 used to copy bit[15-8] of R
SWAPBYTES       AND R3, R3, #0  ; initialize R3 with value x0000

                LD   R5, counter ; let R5 be a counter with value x0008
copy            ADD  R1, R1, #0 ; check bit[15] of R1
                BRzp next_bit
                ADD  R3, R3, #1 ; if bit[15] is 1, add 1 to R3
next_bit        ADD  R5, R5, #-1 ; decreament R5
                BRz  next_part
                ADD  R3, R3, R3 ; shift R1, R3 to left 1 bit
                ADD  R1, R1, R1
                BRnzp copy
                ADD  R1, R1, R1
                ADD  R2, R1, R3
                RET
counter         x0008

.END
```

Problem 3 (20 points): Division with a Stack

Write subroutine STACKDIVIDE in the space below. Given a stack containing some number (≥ 2) of positive integers, with R6 pointing to the top of the stack, the routine STACKDIVIDE operates as follows:

1. Pops two numbers.
2. Uses the DIVIDE subroutine (interface specified below) to divide the first number popped by the second number popped, ignoring any remainder.
3. Jumps to **Step 6** if the stack is empty (stack pointer equal to value at BASEPTR).
4. Otherwise, pushes the result onto the stack
5. Returns to **Step 1** (repeats the process).
6. Stores the result of the final division into the memory location labeled ANS.

Step 2 must utilize the DIVIDE subroutine, whose address is stored at the label DIVIDE_ADDRESS. As input, the DIVIDE subroutine expects the dividend (numerator) in R1 and the divisor (denominator) in R2, then returns the quotient in R3. All registers except R3 and R7 are callee-saved with DIVIDE.

On return from STACKDIVIDE, R6 must point to the base of the stack. All other registers are caller-saved. Use the memory location labeled as SAVE to save any value that you find necessary to save. Briefly comment your code and describe how each register is used. Use the back of this sheet if you need more space, but we will only read up to 30 instructions.

```
STACKDIVIDE
                LDR   R1,R6,#0   ;let R1 store numerator
                ADD   R6,R6,#1
                LDR   R2,R6,#0   ;let R2 store denominator
                ADD   R6,R6,#1   ; Pop two numbers
                JSR   DIVIDE_ADDRESS
                LD    R4,BASEPTR ; check if R6 equals BASEPTR
                BRz   DONE
                ADD   R6,R6,#-1  ; Push the result
                STR   R3,R6,#0
                BRnzp STACKDIVIDE
DONE            ST    R3,ANS
                RET
BASEPTR         .FILL x4000 ; base of stack
SAVE            .BLKW #1
ANS             .BLKW #1
DIVIDE_ADDRESS .FILL x7000 ; the DIVIDE subroutine is at this address
```

Problem 4 (16 points): C Variables and Function Calls

1. **(12 points)** Read the program below.

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

int32_t mystery (int32_t x);

int main()
{
    int32_t a;
    int32_t b = 3;
    int32_t c = 5;
    int32_t d = 7;

    {
        int32_t c = 9;
        a = mystery (b);
        d = 11;
        printf ("a: %d, b: %d, c: %d, d: %d\n", a, b, c, d);
    }

    b = 7;
    a = mystery (b);
    printf ("a: %d, b: %d, c: %d, d: %d\n", a, b, c, d);
}

int32_t mystery (int32_t x)
{
    static int32_t y = 0;
    x = x + 1;
    y = y + x;
    return y;
}
```

Circle EXACTLY ONE ANSWER for each question.

a. **(6 points)** For the two calls to `printf()`, what are the expected printed values of **a** and **b**?

- **a: 4, b: 3** and **a: 8, b: 7**
- **a: 4, b: 3** and **a: 12, b: 7**
- **a: 4, b: 4** and **a: 8, b: 8**
- **a: 4, b: 4** and **a: 12, b: 8**
- The program does not compile.

b. **(6 points)** For the two calls to `printf()`, what are the expected printed values of **c** and **d**?

- **c: 5, d: 7** and **c: 5, d: 7**
- **c: 5, d: 11** and **c: 5, d: 11**
- **c: 9, d: 11** and **c: 5, d: 11**
- **c: 9, d: 11** and **c: 9, d: 7**
- The program does not compile.

Problem 4, continued:

2. **(4 points)** Consider a C function **`example`** called by another function. The left block of code below corresponds to the caller, and the right block corresponds to the callee (the function **`example`**).

   This question focuses on whether the LC-3 instructions **depend on the number of parameters** needed by the function **`example`**.

   For each of the four blanks (explained by the comments immediately above them), write "YES" if the LC-3 instructions depend on the number of parameters passed to **`example`**, or write "NO" if the LC-3 instructions do not depend on the number of parameters passed to **`example`**.

```
; prepare for call
        YES
JSR EXAMPLE
; clean up after call
        YES
```

```
EXAMPLE
; set up stack frame
        NO
; (execute C statements)
; tear down stack frame
        NO
RET
```

vari addr

RT → variable

previous frame

Problem 5 (20 points): Understanding Compiled C Code

1. **(15 points)** The LC-3 code below corresponds to the output of a compiler for the C function **foo**. Based on the LC-3 code, write C code for **foo** from which a non-optimizing compiler might have produced the LC-3 code.

```
FOO   ADD R6, R6, #-4
      STR R5, R6, #1
      ADD R5, R6, #0
      STR R7, R5, #2
      LDR R0, R5, #4
      STR R0, R5, #0
LOOP  LDR R0, R5, #4
      BRnz DONE
      LDR R1, R5, #5
      ADD R6, R6, #-1
      STR R1, R6, #0
      LDR R2, R5, #0
      ADD R6, R6, #-1
      STR R2, R6, #0
      JSR ANOTHER          ; call this subroutine "another" in C
      LDR R3, R6, #0
      ADD R6, R6, #3
      STR R3, R5, #0
      LDR R1, R5, #4
      ADD R1, R1, #-1
      STR R1, R5, #4
      BRnzp LOOP
DONE  LDR R0, R5, #0
      STR R0, R5, #3
      LDR R7, R5, #2
      LDR R5, R5, #1
      ADD R6, R6, #3
      RET
```

Write the C function **foo** below. 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**.

```
int foo (int X, int Y)
{
    int A = x;

    for ( /* blank */ ; x > 0; x--) {
        A = another (A, Y);
    }
    return A;
}
```

Problem 5, continued:

2. **(5 points)** Given the LC-3 implementation shown below of the C function **`another`**, write an expression for the value returned from function **`foo`** (from **Part 1**) in terms of the arguments passed (called **`A`** and **`B`** in the expression below):

   `foo (A, B)` evaluates to _____.

```
ANOTHER
      ADD R6, R6, #-3      ; no local variables
      STR R5, R6, #0
      ADD R5, R6, #-1
      STR R7, R5, #2       ; last instruction to set up stack frame
      LDR R1, R5, #4       ; R1 <- X
      LDR R2, R5, #5       ; R2 <- -Y
      NOT R2, R2
      ADD R2, R2, #1
      ADD R1, R1, R2       ; R1 <- X - Y
      STR R1, R5, #3       ; return X - Y
      LDR R7, R5, #2       ; tear down stack frame
      LDR R5, R5, #1
      ADD R6, R6, #2
      RET
```