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 not 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.

main task: read N, read operation, compute, store result

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

compute: sum or product?

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

sum / multiply 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,DSR
            BRzp MYOUTPUT
            STI R5,DDR
            ____________
            ____________

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

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
```

- **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___

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
SWAPBYTES
      AND R2,R2,#0     ; R2 will hold the low byte of R1
      AND R3,R3,#0     ; R3 is a loop counter (8 bits)
      ADD R3,R3,#8
LOOP  ADD R2,R2,R2     ; shift R2 left to make space for a bit
      ADD R1,R1,#0     ; copy bit 15 from R1
      BRzp ZERO
      ADD R2,R2,#1
ZERO  ADD R1,R1,R1     ; shift copied bit out of R1
      ADD R3,R3,#-1    ; count down
      BRp LOOP         ; keep copying until 8 bits shifted
      ; R2 now has the high 8 bits of original value in low byte
      ; R1 now has the low 8 bits of original value in high byte
      ADD R2,R2,R1     ; merge bytes into R2
      RET
.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 <u>this</u> sheet if you need more space, but we will only read up to 30 instructions.

```
STACKDIVIDE
      ST R7,SAVE          ; save R7 - need to use JSRR
      LD R4,BASEPTR       ; set R4 to -<base of stack>
      NOT R4,R4
      ADD R4,R4,#1
LOOP  LDR R1,R6,#0        ; pop numerator into R1
      LDR R2,R6,#1        ; pop denominator into R2
      ADD R6,R6,#2
      LD R7,DIVIDE_ADDRESS  ; call DIVIDE
      JSRR R7
      ADD R5,R4,R6        ; is stack empty?
      BRz DONE            ; if so, go to DONE
      ADD R6,R6,#-1       ; push R3 (quotient)
      STR R3,R6,#0
      BRnzp LOOP          ; go back to step 1
DONE  ST R3,ANS           ; store answer
      LD R7,SAVE          ; restore R7



               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`**?
   1) **`a: 4, b: 3`** and **`a: 8, b: 7`**
   2) **`a: 4, b: 3`** and **`a: 12, b: 7`**
   3) **`a: 4, b: 4`** and **`a: 8, b: 8`**
   4) **`a: 4, b: 4`** and **`a: 12, b: 8`**
   5) The program does not compile.

b. **(6 points)** For the two calls to `printf()`, what are the expected printed values of **`c`** and **`d`**?
   1) **`c: 5, d: 7`** and **`c: 5, d: 7`**
   2) **`c: 5, d: 11`** and **`c: 5, d: 11`**
   3) **`c: 9, d: 11`** and **`c: 5, d: 11`**
   4) **`c: 9, d: 11`** and **`c: 9, d: 7`**
   5) 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
```

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       ; one local variable
      STR R5, R6, #1
      ADD R5, R6, #0
      STR R7, R5, #2        ; last instruction to set up stack frame
      LDR R0, R5, #4        ; A = x
      STR R0, R5, #0
LOOP  LDR R0, R5, #4        ; if x <= 0, branch to DONE (loop ends)
      BRnz DONE
      LDR R1, R5, #5        ; push Y
      ADD R6, R6, #-1
      STR R1, R6, #0
      LDR R2, R5, #0        ; push A
      ADD R6, R6, #-1
      STR R2, R6, #0
      JSR ANOTHER           ; call this subroutine "another" in C
      LDR R3, R6, #0        ; R3 <- return value
      ADD R6, R6, #3        ; pop return value and parameters
      STR R3, R5, #0        ; A = return value
      LDR R1, R5, #4        ; x--
      ADD R1, R1, #-1
      STR R1, R5, #4
      BRnzp LOOP
DONE  LDR R0, R5, #0        ; return A
      STR R0, R5, #3
      LDR R7, R5, #2        ; tear down stack frame
      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 ________(A > 0 ? A * (1 – B) : A)____________ .

```
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
```

not needed for solution, but added here for clarity:

```
int another (int X, int Y)
{
    return X - Y;
}
```