ZJU-UIUC Institute
First Midterm Exam, ECE 220

Thursday 29 October 2020

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Correct Room</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Total 100 points
Problem 1 (22 points): Short Answer Questions and I/O

1. For MP2, your smart friend decided to write a subroutine called PRINT_WITH_VL to print a vertical line ‘|’ followed by the centered string passed in R1 (using PRINT_CENTERED in MP1). All registers for the subroutine are caller-saved.

While coding, he made a mistake. Fortunately, he wrote a test that exposed the bug. When he runs the test in lc3sim, he finds that the first call (at line 5) succeeds, printing “|  AAAAA  ”, but the second call (at line 9) fails, printing “  BBBBB  ” without the vertical line.

Assume that PRINT_CENTERED is correct, and that all registers except for R7 are callee-saved for PRINT_CENTERED.

A. (4 points) Using NO MORE THAN 15 WORDS, explain why the second call fails.

Executes data at line 19.

B. (4 points) Make one change to the code between lines 12 and line 36 (add a line, delete a line, or move a line/label) to fix the subroutine. You may not modify any code before line 12. NO CREDIT will be given for more than one change.

1. ORIG x3000

2. LEA R1, STR_1

3. JSR PRINT_WITH_VL ; SUCCESS

4. LEA R1, STR_2

5. JSR PRINT_WITH_VL ; FAIL

6. HALT

7. STR_1 .STRINGZ "AAAAA"

8. STR_2 .STRINGZ "BBBBB"

9. PRINT_WITH_VL

10. SAVE_R7 .FILL 0 ; part B: move this line outside of subroutine

11. ST R7, SAVE_R7 ; (ex: to 16 or 32 or 34)

12. LD R0, VLINE

13. OUT

14. JSR PRINT_CENTERED ; more detail: in first execution of

15. LD R7, SAVE_R7 ; PRINT_WITH_VL, .FILL 0 (no-op) is executed

16. RET ; then written with x3002 (ST R0,#2); in

17. VLINE .FILL x7C ; second execution, the ST instruction

18. .END ; writes x007C (no-op) over OUT, eliminating

19. ; the vertical line. Students need not

20. ; provide this detail, of course.
Problem 1, continued:

2. **(10 points)** The door of D-331 is always closed, which makes you very angry. You decide to add an *AI quantum magic button* to control the door with a piece of LC-3 code. When the button is pressed, it sends a message to a memory-mapped IO register (called AQMR) at address 0xFFD0 with the format shown here:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>reserved</td>
<td>magic number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- When the button is pressed, the “status” bit of AQMR becomes 1. Otherwise, the “Status” bit is 0.
- If and only if an ECE220 student presses the button, a 12-bit message 0x220 appears in the “magic number” part of AQMR. The “magic number” is something other than 0x220 (the exact bits are unspecified) when no ECE220 student is pressing the button.
- The “reserved” part of AQMR should not be used—do not assume 0s nor 1s in these bits.

Using **NO MORE LINES THAN PROVIDED BELOW** (you may leave some blank), complete the LC-3 code to control the door. The code should send a signal to unlock the door by writing x0220 to 0xFFD2 whenever an ECE220 student presses the button. You may **use any registers**.

```
.ORIG x3000
; An infinite loop to check AQMR and unlock the door if needed
INFINITE_LOOP       LDI R1, AQMR
                      BRzp INFINITE_LOOP
                      LDI R2, MASK       ; examine only the "magic number"
                      AND R1, R1, R2
                      LDI R2, NEG_VAL    ; load negated x0220 into R2
                      ADD R2, R2, R1
                      BRnp INFINITE_LOOP ; not the magic number!
                      STI R1, AQDR      ; R1 must hold x0220
                      BRnzp INFINITE_LOOP
                      ; LC-3 should never reach here
HALT
AQMR .FILL xFFD0
MASK  .FILL xOFFF   ; magic number bits only
NEG_VAL .FILL xFDE0  ; x0220, negated
AQDR  .FILL xFFD2
.END
```
Problem 1, continued:

3. **(4 points)** As part of an ECE408 MP, the TAs were asked to implement

\[
\text{ceildiv}(A,B) = \lceil A/B \rceil ,
\]

where \(A\) and \(B\) are positive integers and the ceiling function, \(\lceil X \rceil\), computes the smallest integer \(\geq X\). Note that the definition above is in math, not in C code. Sadly, the TAs need help. Please fill in the blank to complete the function. No control constructs nor function calls are allowed, and answers that do not fit in the blank will not be considered for credit.

```c
uint32_t ceildiv(uint32_t A, uint32_t B)
{
    return (A + B – 1) / B ;
}
```

There are many valid answers to this question. Technically, a conditional operator is likely to produce assembly more like an if statement (a control construct), but we accepted those answers as well since the conditional operator was introduced as an operator.
Problem 2 (22 points): Understanding LC-3 Code

The LC-3 subroutine MYSTERY appears below. The subroutine requires that R1 > 1 when it is called. Read the code, then answer the questions below.

MYSTERY       AND R5,R5,#0
                ADD R2,R1,#-1
OUTER_LOOP    ADD R4,R2,#-1
                BRz LABEL1
                ADD R3,R1,#0
INNER_LOOP    ADD R4,R2,#0
                NOT R4,R4
                ADD R4,R4,#1
                ADD R3,R3,R4
                BRp INNER_LOOP
                BRz LABEL2
                ADD R2,R2,#-1
                BRnzp OUTER_LOOP
LABEL1        ADD R5,R5,#1
LABEL2        RET

1. Assuming that R1=x0003, R2 contains bits, and R3=x0042 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.”

   R2: ______ 1 _______ R3: ______ -1 _______ R5: ______ 1 _______

2. Assuming that R1=x0004, R2=x0000, and R3 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.”

   R2: ______ 2 _______ R3: ______ 0 _______ R5: ______ 0 _______

3. Assuming that R1=x0005, R2=FFFF, and R3=x0110 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.”

   R2: ______ 1 _______ R3: ______ -1 _______ R5: ______ 1 _______

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

   Checks whether R1 is prime, returning R5=1 if yes and R5=0 if no.
   [ Also returns largest factor of R1 (other than R1 itself) in R2. We didn’t plan that fact, and didn’t require students to say it, but several did. ]
Problem 3 (24 points): Computing the Maximum Value on a Stack

Professor Lumetta needs your help! He knows that you implemented FACTORIAL during lecture (as a think-pair-share), which multiplied a stack of integers. Now, he needs you to write a subroutine to compute the maximum value among non-negative integers on a stack. The following subroutine is provided to you:

```
; MAX - compare two non-negative integers and return the larger one
; Input:  R1 - first non-negative integer
;        R2 - second non-negative integer
; Output: R0 - the larger one among R1 and R2
MAX     NOT R0,R1       ; store -R1 into R0
        ADD R0,R0,#1
        ADD R0,R0,R2    ; now R0 = R2 - R1
        BRp RETR2       ; if R2 - R1 > 0, goto RETR2
        ADD R0,R1,#0    ; return R1
        RET

RETR2   ADD R0,R2,#0    ; return R2
        RET
```

1. (10 points) First, write a subroutine called STACK_MAX that pops two integers from the stack, compares them using MAX, and pushes the larger one back onto the stack.
   - You must complete the pop operations before calling MAX.
   - The stack follows the same conventions used in lecture and the slides.
   - You may assume that there are at least two non-negative integers on top of the stack.

```
; STACK_MAX - pop two non-negative integers from the stack
;             and push back the larger one
; Input:  R6 - top of the stack
; Output: R6 - top of the stack after operation
; Registers: All registers are caller-saved.
ST R7,SM_R7  ; save R7
LDR R1,R6,#0 ; read first value into R1
LDR R2,R6,#1 ; read second value into R2
ADD R6,R6,#2 ; remove both values from stack
JSR MAX      ; find maximum of R1 and R2 in R0
ADD R6,R6,#-1 ; push R0 onto stack
STR R0,R6,#0
LD R7,SM_R7  ; restore R7
RET

SM_R7  .BLKW #1
```

Use NO MORE THAN 15 MEMORY LOCATIONS, including storage for any data needed. **Using more than 15 LOCATIONS will earn NO CREDIT.**

(Include comments for more partial credit.)
Problem 3, continued:

2. **(14 points)** Now it’s time to actually solve the problem! Write a subroutine called COMPUTE_MAX that processes the integers on the stack using the STACK_MAX subroutine that you wrote in part (1) and leaves the maximum integer as the only element on the stack.

; COMPUTE_MAX – process a stack of non-negative integers,;
;               leaving only the maximum value on the stack
; Input:  R6 – top of the stack
;        R5 – base of the stack
; Output: R6 – top of the stack (original base minus 1),
;         which points to the maximal integer
; All registers are caller-saved.

Use **NO MORE THAN 20 MEMORY LOCATIONS**, including storage for any data needed.
**Using more memory than 20 LOCATIONS will earn NO CREDIT.**
(Include comments for more partial credit.)

COMPUTE_MAX ST R7,CM_R7  ; save R7
    NOT R5,R5  ; compute –(R5 – 1) = (-R5) + 1
    ADD R5,R5,#2
LOOP  ADD R3,R6,R5 ; stack has one value? If so, all done.
    BRz DONE
    JSR STACK_MAX ; combine two values into one on stack
    BRnzp LOOP ; keep going until we have one value left
DONE  LD R7,CM_R7  ; restore R7
    RET

CM_R7  .BLKW #1
**Problem 4** (17 points): Basics of C Programming

1. Read the C program below, then answer the questions.

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

void func (int32_t p) {
    static int32_t x = 0;
    static int32_t y = 5;
    while (++x + y < p) {
        y += (x << 1);
        printf ("%d %d ", x, y);
    }
}

int main () {
    int x = 30;
    func (x);
    // func (x + 10);   // <-- this call added for part (B)
    return 0;
}
```

A. (6 points) Write the function’s output on the line below.

```
1 7 2 11 3 17 4 25
```

B. (3 points) If a second call to `func` is added (shown in the comment), what is the output from the second call to `func`? Write it on the line below.

```
6 37
```

2. (8 points) Read the C program below, then write the program’s output on the blank line below the code.

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

int main() {
    int32_t i = 0, j = 0;
    do {
        switch (i % 2) {
            case 0:
                j++;
                printf ("%d", j);
            case 1:
                printf ("%d", i);
                i++;
                break;
            default:
                printf("x");
                break;
        }
    } while (i++ + j < 6);
    return 0;
}
```

Output: 102234
Problem 5 (14 points): Understanding Compiled C Code

The LC-3 code below corresponds to the output of a non-optimizing compiler for the C function `funny`.

```
FUNNY
ADD R6, R6, #-5 ; space for linkage + 2 local variables
STR R5, R6, #2
ADD R5, R6, #1
STR R7, R5, #2 ; stack frame setup complete
AND R0, R0, #0 ; initialize one local variable (say B) to 0
STR R0, R5, #-1
LOOP
LDR R0, R5, #4 ; load X into R0
BRnz DONE ; if X <= 0, we’re "DONE"
LDR R0, R5, #5 ; add Y to B
LDR R1, R5, #-1
ADD R1, R1, R0
STR R1, R5, #-1 ; store sum back to B
LDR R0, R5, #4 ; decrement X
ADD R0, R0, #-1
STR R0, R5, #4 ; branch back to loop test
BRnzp LOOP
DONE
LDR R0, R5, #-1 ; return B (copy B to return value location)
STR R0, R5, #3
STR R7, R5, #2 ; stack frame teardown starts here
LDR R5, R5, #1
ADD R6, R6, #4
RET
```

Write C code below for the function `funny` 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 funny (int X, int Y)
{
    int A, B = 0;

    while (X > 0) {
        B += Y;
        X--;
    }
    return B;
}
```

// Note that the code changes both B and X, so this function is not
// the compiler implementing multiply (return X * Y when X >= 0),
// but rather part of the C code, as shown.