Homework05

1.

- Bob Computer just bought a fancy new graphics display for his LC-3. In order to test out how fast it is, he rewrote the OUT trap handler so it would not check the DSR before outputting. Sadly he discovered that his display was not fast enough to keep up with the speed at which the LC-3 was writing to the DDR. How was he able to tell?
  Some of the characters written to the DDR weren't being output to the screen.
- Bob also rewrote the handler for GETC, but when he typed ABCD into the keyboard, the following values were input:
  AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBCCCCCCCC
  CCCCCCCCCCCDDDDDDDDDDDDDDDDDDDD
  What did Bob do wrong?
  He didn’t check the KBSR before inputting the character.

2. (Adapted from 8.1)
   What are the defining characteristics of a stack? Give two implementations of a stack and describe their differences.
   Stack is a LAST IN FIRST OUT (LIFO) storing mechanism.

   Two Implementations and differences between them:
   1. Stack in hardware: Stack pointer points to the top of the stack and data entries move during push or pop operations. (ex. Coin holder)
   2. Stack in memory: Stack pointer points to the stack and moves during push or pop operations. Data entries do not move.

3. (Adapted from 8.9) The input stream of a stack is a list of all the elements we pushed onto the stack, in the order that we pushed them. The input stream from Exercise 8.8 on page 304 of the book for example is ABCDEFGHIJKLM
   The output stream is a list of all the elements that are popped off the stack in the order that they are popped off.
   a. If the input stream is ZYXWVUTSR, create a sequence of pushes and pops such that the output stream is YXVUWZSRT.

   Push Z, Push Y, Pop Y, Push X, Pop X, Push W, Push V, Pop V, Push U, Pop U , Pop W, Pop Z, Push T, Push S, Pop S, Push R, Pop R, Pop T。

   b. If the input stream is ZYXW, how many different output streams can be created? Note: only consider output streams that are 4 characters long.

   5+2x5+5=14

4. (Adapted from 8.6) Rewrite the PUSH and POP routines such that the stack on which they operate holds elements that take up two memory locations each. Assume we are writing a program to simulate a stack machine that manipulates 32-bit integers with the LC-3. We would need PUSH and POP routines that operate with a stack that holds elements which take up two memory locations each. Rewrite the PUSH and POP routines for this to be possible.

R0 stores [15:0]， R1 stores [31:16]

```
Push:   ADD R6,R6,#-2
        STR R0,R6,#0
        STR R1,R6,#1
Pop:    LDR R0,R6,#0
        LDR R1,R6,#1
        ADD R6,R6,#2
```

5. A zero-address machine is a stack-based machine where all operations are done by using values stored on the operand stack. For this problem, you may assume that the ISA allows the following operations:

PUSH M - pushes the value stored at memory location M onto the operand stack.
POP M - pops the operand stack and stores the value into memory location M.

OP - Pops two values off the operand stack and performs the binary operation OP on the two values. The result is pushed back onto the operand stack.

Note 1: *OP* can be *ADD*, *SUB*, *MUL*, or *DIV* for parts a and b of this problem. Note 2: To perform *DIV* and *SUB* operations, the top element of the stack is considered as the second operand. i.e. If we first push "A" and then push "B" followed by a "*SUB*" operation, "A" and "B" will be popped from stack and "A-B" will be pushed into stack.

a) Draw a picture of the stack after each of the instructions below are executed. What is the minimum number of memory locations that have to be used on the stack for the purposes of this program? Also write an arithmetic equation expressing u in terms of v, w, x, y, and z. The values u, v, w, x, y, and z are stored in memory locations U, V, W, X, Y, and Z.

PUSH V PUSH W PUSH X PUSH Y MUL ADD PUSH Z SUB DIV POP U

4, U={Z-(XY+W)}/V

b) Write the assembly language code for a zero-address machine (using the same type of instructions from part a) for calculating the expression below. The values a, b, c, d, and e are stored in memory locations A, B, C, D, and E. e = ((a * ((b - c) + d))/(a + c))

Push a, Push b, Push c, SUB, Push d, ADD, MUL, Push a, Push c, ADD, DIV ,Pop E

6. Assume that you have the following table in your program:

```
MASKS    .FILL x0001
         .FILL x0002
         .FILL x0004
         .FILL x0008
         .FILL x0010
         .FILL x0020
         .FILL x0040
         .FILL x0080
         .FILL x0100
         .FILL x0200
         .FILL x0400
         .FILL x0800
         .FILL x1000
         .FILL x2000
         .FILL x4000
         .FILL x8000
```

- a) Write a subroutine *CLEAR* in LC-3 assembly language that clears a bit in *R0* using the table above. The index of the bit to clear is specified in *R1*, *R0* and *R1* are inputs to the subroutine.

```
CLEAR    ST R2,SaveR2
         LEA R2,MASKS
         ADD R2,R1,R2
         LDR R2,R2,#0
         NOT R2,R2
         AND R0,R0,R2
         LD R2,SaveR2
         RET
         SaveR2 .BLKW 1
```

- b) Write a similar subroutine *SET* that sets the specified bit instead of clearing it. Hint: You should remember to save and restore any registers your subroutine uses (the "callee save" convention). Use the RET instruction as the last instruction in your subroutine (R7 contains the address of where in the caller to return to.)

```
CLEAR   ST R2,SaveR2
        LEA R2,MASKS
        ADD R2,R1,R2
        LDR R2,R2,#0
        NOT R2,R2
        NOT R0,R0
        AND R0,R0,R2
        NOT R0,R0
        LD R2,SaveR2
        RET
        SaveR2 .BLKW 1
```

7. Suppose we are writing an algorithm to multiply the elements of an array (unpacked, 16-bit 2's complement numbers), and we are told that a subroutine

"mult_all" exists which multiplies four values, and returns the product. The mult_all subroutine assumes the source operands are in R1, R2, R3, R4, and returns the product in R0. For purposes of this assignment, let us assume that the individual values are small enough that the result will always fit in a 16-bit 2's complement register.

Your job: Using this subroutine, write a program to multiply the set of values contained in consecutive locations starting at location x6001. The number of such values is contained in x6000. Store your result at location x7000. Assume there is at least one value in the array(i.e., M[x6000] is greater than 0).

Hint: Feel free to include in your program

PTR .FILL x6001

CNT .FILL x6000

```
        .ORIG x3000
        LD R5,PTR
        LDI R6,CNT
READ    LDR R1,R5,#0
        ADD R5,R5,#1
        ADD R6,R6,#-1
```

```
        BRz DONE1
        LDR R2,R5,#0
        ADD R5,R5,#1
        ADD R6,R6,#-1
        BRz DONE2
        LDR R3,R5,#0
        ADD R5,R5,#1
        ADD R6,R6,#-1
        BRz DONE3
        LDR R4,R5,#0
        ADD R5,R5,#1
        ADD R6,R6,#-1
        BRnzp DONE4
DONE 1  AND R2,R2,#0
        ADD R2,R2,#1
        ADD R3,R2,#0
        ADD R4,R2,#0
        BRnzp DONE4
DONE 2  AND R3,R3,#0
        ADD R3,R3,#1
        ADD R4,R3,#0
        BRnzp DONE4
DONE 3  AND R4,R4,#0
        ADD R4,R4,#1
        BRnzp DONE4
DONE 4  JSR mult_all
        ADD R6,R6,#0
        BRz EXIT
        ADD R5,R5,#-1
        STR R0,R5,#0
        ADD R6,R6,#1
```

```
            BRnzp READ
EXIT        STI R0,RESULT
            HALT
RESULT      .FILL x7000
PTR         .FILL x6001
CNT         .FILL x6000
```

8. (9.26) The following program is supposed to print the number 5 on the screen. It does not work. Why? Answer in no more than ten words, please.

```
        .ORIG   x3000
        JSR     A
        OUT                     ;TRAP   x21
        BRnzp   DONE
A       AND     R0,R0,#0
        ADD     R0,R0,#5
        JSR     B
        RET
DONE    HALT
ASCII   .FILL   x0030
B       LD      R1,ASCII
        ADD     R0,R0,R1
        RET
        .END
```

Second JSR B overwrites the first RET value.

9. (9.19) The following LC-3 program is assembled and then executed. There are no assemble time or run-time errors. What is the output of this program? Assume all registers are initialized to 0 before the program executes.

```
        .ORIG x3000
        ST R0, #6 ; x3007
        LEA R0, LABEL
        TRAP x22
        TRAP x25
LABEL   .STRINGZ "FUNKY"
LABEL2  .STRINGZ "HELLO WORLD"
        .END
```

FUN

10. The memory locations given below store students' exam scores in form of a linked list. Each node of the linked list uses three memory locations to store

1. Address of the next node
2. Starting address of the memory locations where name of the student is stored
3. Starting address of the memory locations where the his/her exam score is stored

in the given order. The first node is stored in locations x4000 ~ x4002. The ASCII code x0000 is used as a sentinel to indicate the end of the string. Both the name and exam score are stored as strings.
Write down the students' names and scores in the order that they appear in the list.

Address	*Contents*
x4000	*x4016*
x4001	*x4003*
x4002	*x4008*
x4003	*x004D*
x4004	*x0061*
x4005	*x0072*
x4006	*x0063*
x4007	*x0000*
x4008	*x0039*
x4009	*x0030*
x400A	*x0000*
x400B	*x0000*
x400C	*x4019*
x400D	*x401E*
x400E	*x004A*
x400F	*x0061*
x4010	*x0063*
x4011	*x006B*
x4012	*x0000*
x4013	*x0031*
x4014	*x0038*
x4015	*x0000*
x4016	*x400B*
x4017	*x400E*
X4018	*x4013*
x4019	*x004D*
x401A	*x0069*
x401B	*x006B*
x401C	*x0065*
x401D	*x0000*
x401E	*x0037*
x401F	*x0036*
x4020	*x0000*

Marc 90 Jack 18 Mike 76

11. The main program below calls a subroutine, F. The F subroutine uses R3 and R4 as input, and produces an output which is placed in R0. The subroutine modifies registers R0, R3, R4, R5, and R6 in order to complete its task. F calls two other subroutines, SaveRegisters and RestoreRegisters, that are intended handle the saving and restoring of the modified registers (although we will see in part b that this may not be the best idea!).

```
; Main Program
                .ORIG x3000
                ……
                ……
                JSR F
                ……
                ……
                HALT

; R3 and R4 are input.
; Modifies R0, R3, R4, R5, and R6
; R0 is the output


F               JSR SaveRegisters
                ……
                ……
                ……
                JSR RestoreRegisters
                RET
                .END
```

Part a) Write the two subroutines SaveRegisters and RestoreRegisters.

```
SAVEREGISTERS       ST R0, SAVER0
                    ST R3, SAVER3
                    ST R4, SAVER4
                    ST R5, SAVER5
                    ST R6, SAVER6
                    RET
RESTOREREGISTERS    LD R0, SAVER0
                    LD R3, SAVER3
                    LD R4, SAVER4
                    LD R5, SAVER5
                    LD R6, SAVER6

                    RET
```

```
SAVER0    .BLKW x1
SAVER1    .BLKW x1
SAVER2    .BLKW x1
SAVER3    .BLKW x1
SAVER4    .BLKW x1
SAVER5    .BLKW x1
SAVER6    .BLKW x1
```

Part b) When we run the code we notice there is an infinite loop. Why? What small change can we make to our program to correct this error. Please specify both the correction and the subroutine that is being corrected.

Should Save R7 before Call F

12. Suppose we want to make a 10 item queue starting from location x4000. In class, we discussed using a HEAD and a TAIL pointer to keep track of the beginning and end of the queue. In fact, we suggested that the HEAD pointer could point to the first element that we would remove from the queue and the TAIL pointer could point the last element that we have added the queue. It turns out that our suggestion does not work.

- a) What is wrong with our suggestion? (Hint: how do we check if the queue is full? How do we check if it is empty?)
  Our suggestion cannot distinguish between a full and empty queue.
- b) What simple change could be made to our queue to resolve this problem?
  We only allow n-1 items to be place in a queue with n memory spaces.
- c) Using your correction, write a few instructions that check if the queue is full. Use R3 for the HEAD pointer and R4 for the TAIL pointer.
  if Next（R4）=head,or (R3=START&&R4=END) then FULL

```
NOT R5,R4
ADD R5,R5,R3
BRz FULL
LD R5,NEGSTART
ADD R5,R5,R3
BRnp NOTFULL
LD R5,NEGEND
ADD R5,R5,R4
BRz FULL
```

- d) Using your correction, write a few instructions that check if the queue is empty. Again, using R3 for the HEAD pointer and R4 for the TAIL pointer.

if R3==R4 then FULL

NOT R5, R4
ADD R5, R5, #1
ADD R5, R5, R3

BRz EMPTY

13. The following nonsense program is assembled and executed.

```
            .ORIG x4000
            LD      R2,BOBO
            LD      R3,SAM
AGAIN       ADD     R3,R3,R2
            ADD     R2,R2,#-1
            BRnzp SAM
BOBO        .STRINGZ "Why are you asking me this?"
SAM         BRnp    AGAIN
            TRAP    x25
            .BLKW 5
JOE         .FILL x7777
            .END
```

How many times is the loop executed? When the program halts, what is the value in R3? (If you do not want to the arithmetic, it is okay to answer this with a mathematical expression.)

R2=ASCII(W)=x57,so the loop will execute 57 times.

R3=0000 101 #-32 =0000 101 1 1110 0000= x0BE0

R3= x0BE0+(57+1)x57/2= x0B10+0EF4 = x1AD4