Homework 06

1.

- How many TRAP service routines can be implemented in the LC-3? Why?

  Since the trap vector is 8 bits wide, 256 trap routines can be implemented in the LC-3.

- Why must a RET instruction be used to return from a TRAP routine? Why won't a BRnzp (unconditional BR) instruction work instead?

  After the TRAP routine is executed, program control must be passed back to the code that called the TRAP instruction. This is done by copying the value in R7 into the PC. The RET instruction provides this functionality. BRnzp does not restore the PC.

- How many accesses to memory are made during the processing of a TRAP instruction?

  Twice.(Memory -> IR, Memory->PC)

2. What are the defining characteristics of a stack? Give two implementations of a stack and describe their differences.

The defining characteristic of a stack is the unique specification of how it is to be accessed. Stack is a LIFO (Last in First Out) structure. This means that the last thing that is put in the stack will be the first one to get out from the stack.

(Any two reasonable implementations are OK.) We can implement a stack by a sequence or a list. The difference between is how their elements store in memory. The fisrt one is sequential, while the other is discrete.

3. 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 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.
14 different output streams.

4. 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.

This routine pushes the values in R0 and R1 onto the stack. R5 is set to 0 if operation is successful. If overflow occurs R5 is set to 1. This routine works with a stack consisting of memory locations x3FFF(BASE) to x3FFA(MAX).

```
PUSH ST R2, SaveR2
        LD R2, MAX
        NOT R2, R2
        ADD R2, R2, #1 ; R2 = -addr of stackmax
        ADD R2, R6, R2
        BRz Failure
        STR R0, R6, #-1
        STR R1, R6, #-2
        ADD R6, R6, #-2
        AND R5, R5, #0
        LD R2, SaveR2
        RET
Failure AND R5, R5, #0
        ADD R5, R5, #1
        LD R2, SaveR2
        RET
MAX .FILL x3FFA
SaveR2 .FILL x0000
```

This routine pops the top two elements of the stack into R0 and R1. R5 is set to 0 if the operation is successful. If underflow occurs R5 is set to 1. This routine works with a stack consisting of memory locations x3FFF(BASE) to x3FFA(MAX).

```
POP ST R2, SaveR2
        LD R2, EMPTY ; EMPTY <-- -x3FFF
        ADD R2, R6, R2
        BRzp Failure ; Underflow
        LDR R1, R6, #0 ; Pop the first value
        LDR R0, R6, #1 ; Pop the second value
```

```
            ADD R6, R6, #2
            AND R5, R5, #0
            RET
Failure AND R5, R5, #0
            ADD R5, R5, #1
            RET
EMPTY .FILL x3FFF
SaveR2 .FILL x0000
```

5.Figure out what the following program does.

```
        .ORIG X3000
        LEA R2, C           ; R2=&C
        LDR R1, R2, #0      ;R1=x25
        LDI R6, C           ;R6=[x25]
        LDR R5, R1, #-3     ;R5=[x22]
        ST R5, C            ;C=[x22]
        LDR R5, R1, #-4     ;R5=[x21]
        LDR R0, R2, #1      ;R0=&C+1
        JSRR R5 ;           ;trap x21 ,Output 'E'
        AND R3, R3, #0      ;R3=0
        ADD R3, R3, #7      ;R3=7
        LEA R4, B           ;R4=&B
A       STR R4, R1, #0      ;[x25]=&B, [x26]=&B+2, [x27]=&B+4, [x28]=&B+6,
                            ;[x29]=&B+8, [x2A]=&B+10,[x2B]=&B+12
        ADD R4, R4, #2      ;R4=R4+2
        ADD R1, R1, #1      ;
        ADD R3, R3, #-1
        BRP A
        HALT
B       ADD R2, R2, #1      ;R2=&C+1 ;
        LDR R0, R2, #0      ;R0=[&C1+1]
        JSRR R5             ;trap x21, OUTPUT 'E'
        TRAP X29            ; PC=B+8
        ADD R2, R2, #15     ;
        ADD R0, R2, #3
        LD R5, C            ;R5=x25
        TRAP X2B            ;PC=&B+12
        ADD R2, R2, #5      ;R2=&C+6
        LDR R0, R2, #0      ;R0=[&C+6]
        JSRR R5             ;trap x21, OUTPUT ' '
        TRAP X27            ;PC=[&B+6]
        JSRR R5             ;PC=&B ;.....
        JSRR R6
C       .FILL X25
```

```
.STRINGZ "EE306 and tests are awesome"
.END
```

Output "EE some"(You could refer to annotation I appended above)

6. Jane Computer (Bob's adoring wife), not to be outdone by her husband, decided to rewrite the TRAP x22 handler at a different place in memory. Consider her implementation below. If a user writes a program that uses this TRAP handler to output an array of characters, how many times is the ADD instruction at the location with label A executed? Assume that the user only calls this "new" TRAP x22 once. What is wrong with this TRAP handler? Now add the necessary instructions so the TRAP handler executes properly.

Hint: RET uses R7 as linkage back to the caller (RET is equivalent to JMP R7).

```
; TRAP handler
; Outputs ASCII characters stored in consecutive memory locations.
; R0 points to the first ASCII character before the new TRAP x22 is called.
; The null character (x00) provides a sentinel that terminates the output sequence.
```

The execution time of ADD instruction at the location with label A is equal to the length of the array.

```
        .ORIG x020F
START   ST R1, SAVER1
        LDR R1, R0, #0
        BRz DONE
        ST R0, SAVER0
        AND R0, R0, #0
        ADD R0, R1, #0
        ST R7, SAVER7
        TRAP x21
        LD R0, SAVER0
        LD R1, SAVER1
        LD R7, SAVER7
A       ADD R0, R0, #1
        BRnzp START
DONE    RET

SAVER0  .BLKW #1
SAVER1  .BLKW #1
SAVER7  .BLKW #1

        .END
```

7. 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: OP can be ADD, SUB, MUL, or DIV for parts a and b of this problem.

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
```

			Y						
		x	X	X*y		z			
	w	w	W	W	X*y+w	X*y+w	z-(X*y+w)		
v	v	v	v	V	v	v	v	(z-(x*y+w))/v	

The minimum number of memory location is 4.
U=(z-(x*y+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 C
```

```
ADD
PUSH C
PUSH B
SUB
PUSH D
ADD
PUSH A
MUL
DIV
POP E
```