Department of Electrical and Computer Engineering

The University of Texas at Austin

EE 306, Fall 2021
Problem Set 5
Due: **November 15th**, before class
Yale N. Patt, Instructor
TAs: Sabee Grewal, Ali Fakhrzadehgan, Ying-Wei Wu, Michael Chen, Jason Math, Adeel Rehman

Instructions: You are encouraged to work on the problem set in groups and turn in one problem set for the entire group. **The problem sets are to be submitted on Gradescope**. Only one student should submit the problem set on behalf of the group, but everyone should create a gradescope account and be tagged on the homework

1. a) 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.

   b) Bob also rewrote the handler for GETC, but when he typed ABCD into the keyboard, the following values were input:

   AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCC
   CCCCDDDDDDDDDDDDDDDDDDD

   What did Bob do wrong?

   The handler didn't check the KBSR before inputting the character.

2. Shown below are the partial contents of memory locations x3000 to x3006.

	15															0
x3000	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
x3001	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1
x3002	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1
x3003	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1
x3004	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3005	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
x3006	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The PC contains the value x3000, and the RUN button is pushed. As the program executes, we keep track of all values loaded into the MAR. Such a record is often referred to as an address trace. It is shown below.

MAR Trace

x3000
x3005
x3001
x3002
x3006
x4001
x3003
x0021

Your job: Fill in the missing bits in memory locations x3000 to x3006.

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? Only consider output streams that are 4 characters long

14 different output streams.

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 that take up two memory locations each. Rewrite the PUSH and POP routines for this to be possible.

The problem assumes that each element of the value being pushed on the stack is 32-bits.

For the PUSH, assume bits [15:0] of that value to be pushed are in R0 and bits [31:16] are in R1.

For the POP, bits [15:0] will be popped into R0 and bits [31:16] will be popped into R1.

Also assume the lower order bits of the number being pushed or popped are located in the smaller address in memory. For example if the two memory locations to be used to store the number are x2FFF and x2FFE, bits [15:0] will be stored in x2FFE and [31:16] will be stored in x2FFF.

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 the stack and "A-B" will be pushed onto the 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

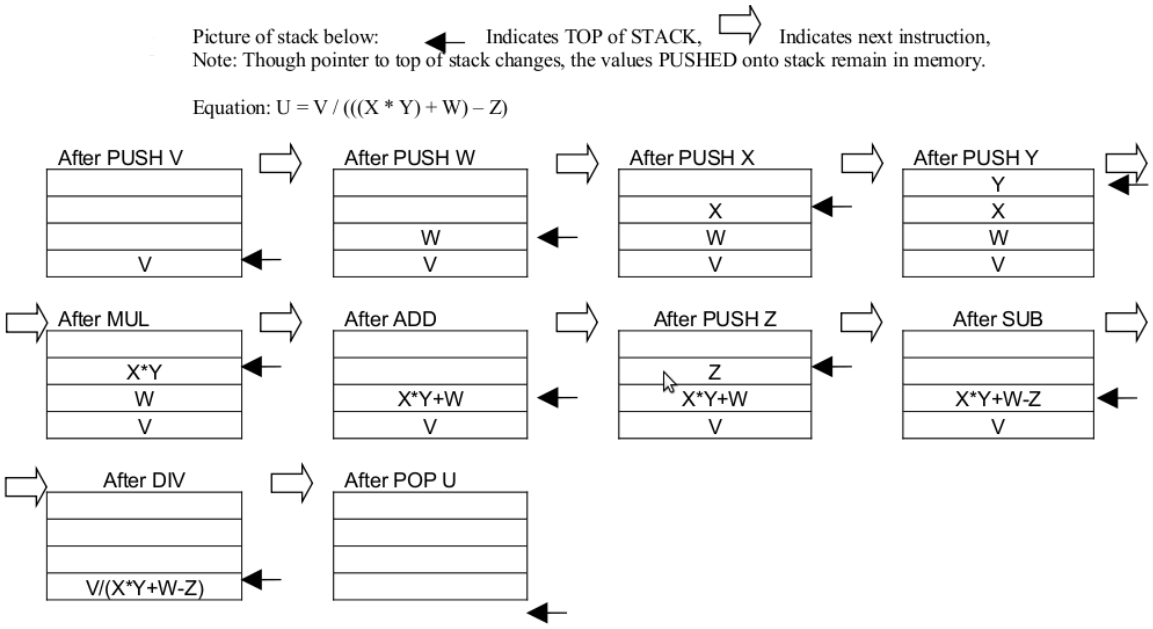
ADD

PUSH Z

SUB

DIV

POP U



Minimum number of memory locations required: 4

Note: If we assume top of stack as first operand in SUB and DIV instructions, we would have Z-((X*Y)+W) on top of stack after SUB and (Z-((X*Y)+W))/V after DIV.

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.

$$e = ((a * ((b - c) + d))/(a + c))$$

(Note: There are multiple solutions to part b.)

PUSH A

PUSH B

PUSH C

SUB

PUSH D

ADD

MUL

PUSH A

PUSH C

ADD

DIV

POP E

Note: Possible answer in case of assuming top of stack as first operand in SUB and DIV instructions:

PUSH A

PUSH C

ADD

PUSH A

PUSH D

PUSH C

PUSH B

SUB

ADD

MUL

DIV

POP E

7. 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,TEMP
        LEA R2,MASKS
        ADD R2,R1,R2
        LDR R2,R2,#0
        NOT R2,R2
        AND R0,R2,R0
        LD  R2,TEMP
        RET
```

```
TEMP:          .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 to return to.)

```
SET:    ST  R2,TEMP
        LEA R2,MASKS
        ADD R2,R1,R2
        LDR R2,R2,#0
        NOT R2,R2
        NOT R0,R0
        AND R0,R2,R0
        NOT R0,R0
        LD  R2, TEMP
        RET
TEMP: .BLKW #1
```

8. 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
    BRz DONEz    ;checks if more numbers to multiply(CNT=0)
MORE   LDR R1,R5,#0
    ADD R5,R5,#1
    ADD R6,R6,#-1
    BRz DONE1    ;continues if more numbers to multiply
    LDR R2,R5,#0
    ADD R5,R5,#1
    ADD R6,R6,#-1
    BRz DONE2    ;continues if more numbers to multiply
    LDR R3,R5,#0
    ADD R5,R5,#1
    ADD R6,R6,#-1
    BRz DONE3    ;continues if more numbers to multiply
    LDR R4,R5,#0
    ADD R5,R5,#1
    ADD R6,R6,#-1
```

```
    BRnzp READY  ;CNT is multiple of 4
DONEz  AND R0,R0,#0
    ADD R0,R0,#1
    BRnzp END

;(CNT = 4x+1) multiplies R1 by three 1's
DONE1  AND R2,R2,#0
    ADD R2,R2,#1      ;R2 = 1
    ADD R3,R2,#0      ;R3 = 1
    ADD R4,R2,#0      ;R4 = 1
    BRnzp READY

;(CNT = 4x+2) multiplies R1,R2 by two 1's
DONE2  AND R3,R3,#0
    ADD R3,R3,#1      ;R3 = 1
    ADD R4,R4,#0      ;R4 = 1
    BRnzp READY
;
;(CNT = 4x+3) multiplies R1,R2,R3 by 1
DONE3  AND R4,R4,#0
    ADD R4,R4,#1
READY  JSR mult_all
    ADD R6,R6,#0
    BRz END          ;checks CNT
;
;if CNT is not zero takes R0 from subroutine and puts back into memory to multiply more
numbers
    ADD R5,R5,#-1
    STR R0,R5,#0
;add one back to CNT because R0 is back into memory
    ADD R6,R6,#1
    BRnzp MORE
```

```
;
;store result of multiplication in memory location RESULT
END    ST R0,RESULT
       HALT
RESULT       .BLKW 1
mult_all ... ;multiples R1,R2,R3,R4 and stores result in R0
            ...
            ...
            RET
PTR    .FILL x6001
CNT    .FILL x6000
.END
```

9. (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
```

Need to save R7 so 1st service routine can return. Second RET overwrites the first RET value.

10. (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

11. 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 the name of the student is stored.
    3. Starting address of the memory locations where 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

12. 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.

Calling program forgot to save R7, the program will keep going back. We can save R7 to avoid this.

13. 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 to the last element that we have added to the queue. It turns out that our suggestion does not work.

    Part 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. (Using some other metadata to keep track of full or empty is not efficient.)

Part 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.

Part 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.

We need to check if next(R4) = head. This can be either the next address or having the tail before the wrap around and the head before the wrap around. Any variation of the code below will work. A 10 item queue requires 11 addresses.

```
; Store all registers that may be clobbered.
...
...
...
NOT R5, R4; We don't add 1 because we want to subtract 1 right afterwards.
ADD R5, R5, R3; R4+1 == R3?
BRz FULL
LD R5, NEGSTART
ADD R5, R5, R3
BRnp NOTFULL
LD R5, NEGEND
ADD R5, R5, R4
BRz FULL
NOTFULL ... ; Do something at label NOTFULL
...
...
...
FULL ...; Do something at label FULL
...
```

...

...

; Restore all registers used

RET

Part 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.

NOT R5, R4

ADD R5, R5, #1

ADD R5, R5, R3; R4 == R3?

14. The following nonsense program is assembled and executed.

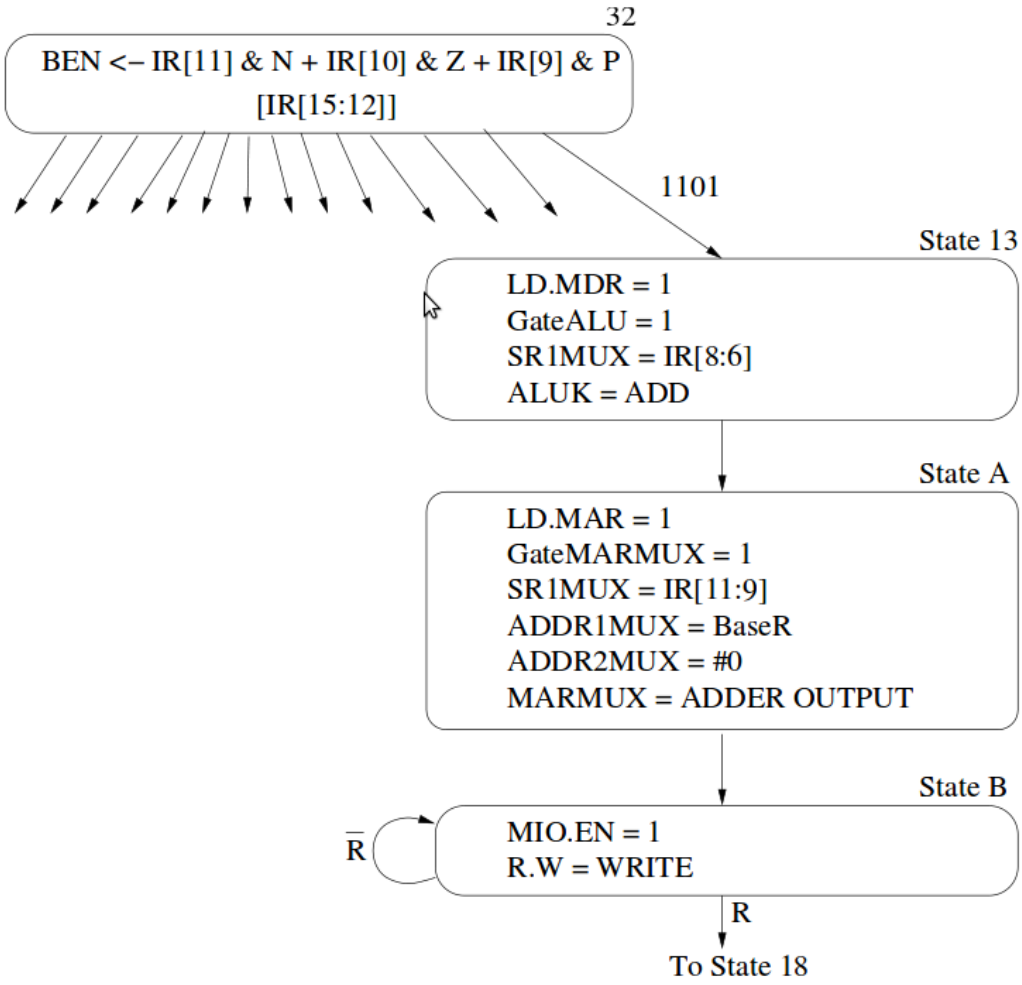
```
        .ORIG x4000

        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.)

Work: BOBO is length 28 (27 + 1 for null). BRnp AGAIN in binary is 0000 101 #-32 = 0000 101 1 1110 0000 = x0BE0. R3 holds x0BE0. R2 starts with the value of W

which is x57. R. The loop executes 57 times. The final value of R3 is x0BE0 + (x57 + x1) * x57 / x2 = x0BE0 + x0EF4 = x1AD4 or #6868. Note that x0BE0 is #3040.

15. Suppose we use the unused opcode 1101 to specify a new instruction. We will require 3 states after decode (state 32) to complete the job. The control signals required to carry out the work of the new instruction are shown below. All control signals not shown below are 0.



a. What does this new instruction do?

Take SR1 ([8:6]), add it to OP2 (SR2 or imm5), and then store it in M[DR] ([11:9]).

b. There are two different formats for specifying the operands of this instruction. Fill them out below.

16. The following program iloads a number N from a label 'N' (not shown), and stores the result in a label RESULT (which is shown).

```
                .ORIG x3000
                LEA R6, BASE
                ADD R6, R6, #-1
                LD R0, N
                STR R0, R6, #0
                AND R0, R0, #0
GOGOGO          LDR R1, R6, #0
                ADD R6, R6, #1
                ADD R5, R1, #1
                BRz DONE
                ADD R5, R1, #0
                BRz GOGOGO
NOTZERO         ADD R5, R1, #-1
                 BRnp NOTONE
                ADD R0, R0, #1
                BR GOGOGO
```

```
NOTONE      ADD R6, R6, #-1
            ADD R1, R1, #-1
            STR R1, R6, #0
            ADD R6, R6, #-1
            ADD R1, R1, #-1
            STR R1, R6, #0
            BR GOGOGO
DONE        ST R0, RESULT
            HALT
DATA        .BLKW #5
BASE        .FILL xFFFF
RESULT      .BLKW #1
            .END
```

a. What does this program do?

Calculates the Nth fibonacci number and stores it in result

b. What is the largest number N for which the program will still work properly? Why won't it work for larger numbers?

N = 9. The Data array will overflow and overwrite code.

c. What would be a better (more memory efficient) way to write this program? (Explain how it works, no need to write the code for it)

Complete iteratively (bottom-up) rather than recursively (top-down)