Department of Electrical and Computer Engineering

The University of Texas at Austin

EE 306, Fall 2021
Problem Set 6
Due: Not to be turned in
Yale N. Patt, Instructor
TAs: Sabee Grewal, Ali Fakhrzadehgan, Ying-Wei Wu, Michael Chen, Jason Math, Adeel Rehman

Note: This problem set is unusually long, and is not to be turned in. We have put it together and handed it out to give you some challenging examples to help you prepare for the final exam.

1. 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. Is it ok to call TRAP x21 within this "new" Trap routine? Explain why or why not in 20 words or fewer.
   Yes, it's OK to call TRAP within a Trap service routine. Be sure to save/restore registers when implementing subroutines or service routines.

   ; 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.

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

```
        BRnzp START
DONE    LD R1, SAVER1
        RTI

SAVER0  .BLKW #1
SAVER1  .BLKW #1
        .END
```

2. (Adapted from 9.16)
    a. How many TRAP service routines can be implemented in the LC-3? Why?

        256 TRAP service routines can be implemented. x0000- x00FF

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

        RTI pops the PC and PSR from the system stack so that it can return to the original program after execution of the service routine. A BRnzp would not work because:

        - the TRAP routine may not be reached by a 9 bit offset.

        - if TRAP is called multiple times, the computer would not know which LABEL to go to (can change every time).

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

        4 memory accesses are made during TRAP instruction

        1st access:- instruction in fetch

        2nd access:- pushing the PSR to the system stack

        3rd access:- pushing the PC to the system stack

        4th access:- loading the contents of Table'Vector (see state 54 and 53).

3. (Adapted from 8.15)
    a. What does the following LC-3 program do?

```
.ORIG  x3000
LD R3 , A
STI R3, KBSR
```

```
AGAIN   LD R0,B
        TRAP X21
        BRnzp AGAIN
A       .FILL X4000
B       .FILL X0032
KBSR    .FILL XFE00
        .END
```

The keyboard interrupt is enabled, and the digit 2 is repeatedly written to the screen.

b. If someone strikes a key, the program will be interrupted and the keyboard interrupt service routine will be executed as shown below. What does the keyboard interrupt service routine do?

```
        .ORIG X1000
        LDI R0,KBDR
        TRAP X21
        TRAP X21
        RTI
KBDR    .FILL XFE02
        .END
```

c. Finally, suppose the program of part a started executing, and someone sitting at the keyboard struck a key. What would you see on the screen?

4. (9.34) What does the following LC-3 program do?

```
        .ORIG x3000
        LD  R0, ASCII
        LD  R1, NEG
AGAIN   LDI  R2, DSR
        BRzp AGAIN
        STI  R0, DDR
        ADD  R0, R0, #1
        ADD  R2, R0, R1
        BRnp AGAIN
        HALT
ASCII   .FILL x0041
```

```
NEG     .FILL xFFB6
DSR     .FILL xFE04
DDR     .FILL xFE06
        .END
```

Letter ABCDEFGHI will be displayed on console.

5. (Adapted from 10.1)
   What are the defining characteristics of a stack? Give two implementations of a stack and describe their differences.

   Stack is a storing mechanism. The concept of a stack is the specification of how it is to be accessed. That is, the defining ingredient of the stack is that the last thing you stored in it is the first things you remove from it. LAST IN FIRST OUT (LIFO)

   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.

6. Consider the following LC-3 assembly language program. Assuming that the memory locations DATA get filled before the program executes, what is the relationship between the final values at DATA and the initial values at DATA?

```
        .ORIG   x3000
LEA     R0, DATA
        AND     R1, R1, #0
        ADD     R1, R1, #9
LOOP1   ADD     R2, R0, #0
        ADD     R3, R1, #0
LOOP2   JSR     SUB1
        ADD     R4, R4, #0
        BRzp    LABEL
        JSR     SUB2
LABEL   ADD     R2, R2, #1
        ADD     R3, R3, #-1
        BRp     LOOP2
        ADD     R1, R1, #-1
```

```
                BRp     LOOP1
                HALT
DATA            .BLKW   #10
SUB1            LDR     R5, R2, #0
                NOT     R5, R5
                ADD     R5, R5, #1
                LDR     R6, R2, #1
                ADD     R4, R5, R6
                RET
SUB2            LDR     R4, R2, #0
                LDR     R5, R2, #1
                STR     R4, R2, #1
                STR     R5, R2, #0
                RET
                .END
```

The final values at DATA will be sorted in ascending order.

7. During the initiation of the interrupt service routine, the N, Z, and P condition codes are saved on the stack. By means of a simple example show how incorrect results would be generated if the condition codes were not saved. Also, clearly describe the steps required for properly handling an interrupt.

   Lets take the following program which adds 10 numbers starting at memory location x4000 and stores the result at x5000.

```
                .ORIG x3000
                LD R1, PTR
                AND R0, R0, #0
                LD R2, COUNT
LOOP            LDR R3, R1, #0
                ADD R0, R0, R3
                ADD R1, R1, #1
                ADD R2, R2, #-1
                BRp LOOP
                STI R0, RESULT
                HALT
PTR             .FILL x4000
RESULT          .FILL x5000
COUNT           .FILL #10
```

If the condition codes were not saved as part of initiation of the interrupt service routine, we could end up with incorrect results. In this program, take the case when an interrupt occurred during the processing of the instruction at location x3006 and the condition codes were not saved. Let R2 = 5 and hence the condition codes would be N=0, Z=0, P=1, before servicing the interrupt. When control is returned to the instruction at location x3007, the BRp instruction, the condition codes depend on the processing within the interrupt service routine. If they are N=0, Z=1, P=0, then the BRp is not taken. This means that the result stored is just the sum of the first five values and not all ten.

Steps for handling interrupts:

a. Saving the State of the machine
b. Loading the state of the interrupt
c. Service the Interrupt
d. RTI

Note: In-depth explanation of interrupt handling on pages 259-261 of the texbook.

8. The program below counts the number of zeros in a 16-bit word. Fill in the missing blanks below to make it work.

```
            .ORIG x3000
            AND   R0, R0, #0
            LD    R1, SIXTEEN
            LD    R2, WORD
A           BRn   B
            ADD R0, R0, #1
B           ADD R1, R1, #-1
            BRz   C
            ADD R2, R2, R2
            BR    A         ; note: BR = BRnzp
C           ST    R0, RESULT
            HALT

SIXTEEN     .FILL #16
WORD        .BLKW #1
RESULT      .BLKW #1
```

```
                .END
```

After you have the correct answer above, what one instruction can you change (without adding any instructions) that will make the program count the number of ones instead?

Replace the BRn instruction with a BRzp.

9. Fill in the missing blanks so that the subroutine below implements a stack multiply. That is it pops the top two elements off the stack, multiplies them, and pushes the result back on the stack. You can assume that the two numbers will be non-negative integers (greater than or equal to zero) and that their product will not produce an overflow. Also assume that the stack has been properly initialized, the PUSH and POP subroutines have been written for you and work just as described in class, and that the stack will not overflow or underflow.

   Note: All blanks must be filled for the program to operate correctly.

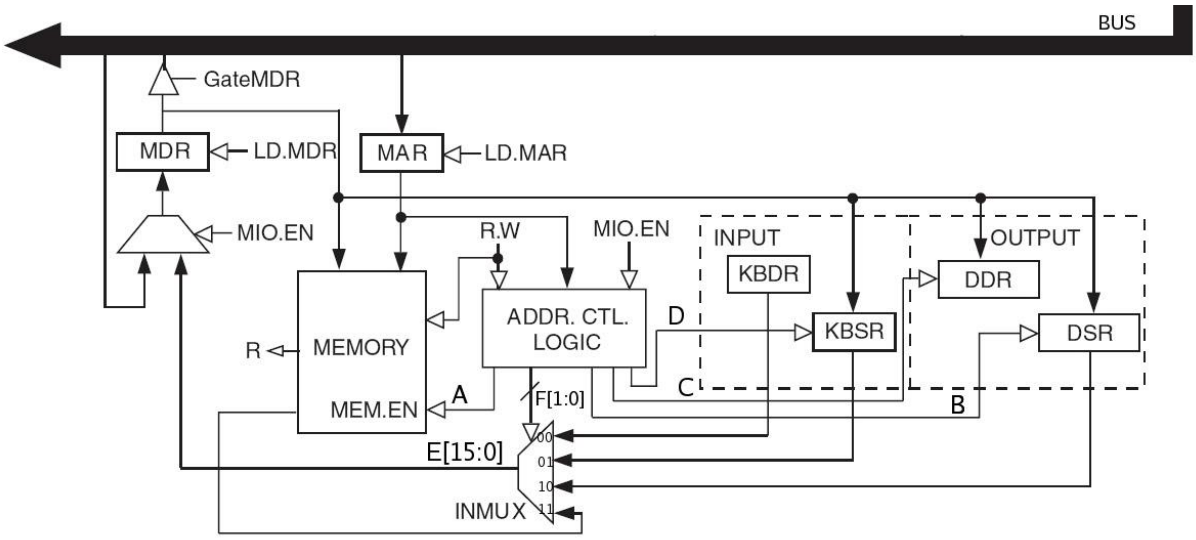
```
MUL     ST  R7, SAVER7
        ST  R0, SAVER0
        ST  R1, SAVER1
        ST  R2, SAVER2
        ST  R5, SAVER5
        AND R2, R2, #0
        JSR POP
        ADD R1, R0, #0
        JSR POP
        ADD R1, R1, #0
        BRz DONE
AGAIN   ADD R2, R2, R0
        ADD R1, R1, #-1
        BRp AGAIN
DONE    ADD R0, R2, #0
        JSR PUSH
        LD R7, SAVER7
        LD R0, SAVER0
        LD R1, SAVER1
        LD R2, SAVER2
        LD R5, SAVER5
        RET
```

10. The program below calculates the closest integer greater than or equal to the square root of the number stored in NUM, and prints it to the screen. That is, if the number stored in NUM is 25, "5" will be printed to the screen. If the number stored in NUM is 26, "6" will be printed to the screen. Fill in the blanks below to make the program work.

    Note: Assume that the value stored at NUM will be between 0 an 81.

```
            .ORIG x3000
            AND R2, R2, #0
            LD R3, NUM
            BRz OUTPUT
            NOT R3, R3
            ADD R3, R3, #1
OUTLOOP     ADD R2, R2, #1
            ADD R0, R2, #0
            AND R1, R1, #0
INLOOP      ADD R1, R1, R2
            ADD R0, R0, #-1
            BRp INLOOP
            ADD R1, R1, R3
            BRn OUTLOOP
OUTPUT      LD R0, ZERO
            ADD R0, R0, R2
            TRAP x21
            HALT
NUM         .BLKW 1
ZERO        .FILL x30
            .END
```

11. The figure below shows the part of the LC-3 data path that deals with memory and I/O. Note the signals labeled A through F. A is the memory enable signal, if it is 1 memory is enabled, if it is 0, memory is disabled. B, C, and D are the load enable signals for the Device Registers. If the load enable signal is 1, the register is loaded with a value, otherwise it is not. E is the 16-bit output of INMUX, and F is the 2-bit select line for INMUX.



The initial values of some of the processor registers and the I/O registers, and some memory locations are as follows:

R0 = x0000	KBSR = x8000	M[x3009] = xFE00
PC = x3000	KBDR = x0061	M[x300A] = xFE02
	DSR = x8000	M[x300B] = xFE04
	DDR = x0031	M[x300C] = xFE06

During the entire instruction cycle, memory is accessed between one and three times (why?). The following table lists two consecutive instructions to be executed on the LC-3. Complete the table with the values that each signal or register takes right after each of the memory accesses performed by the instruction. If an instruction does not require three memory accesses, draw a line accross the unused accesses. To help you get started, we have filled some of the values for you.

PC	Instruction	Access	MAR	A	B	C	D	E[15:0]	F[1]	F[0]	MDR
x3000	LD R0, x9	1	x3000	1	0	0	0	x2009	1	1	x2009

		2	x300A	1	0	0	0	xFE02	1	1	xFE02
		3	--------	---	---	---	---	--------	-----	-----	--------
x3001	LDR R0, R0, #0	1	x3001	1	0	0	0	x6000	1	1	x6000
		2	xFE02	0	0	0	0	x0061	0	0	x0061
		3	--------	---	---	---	---	--------	-----	-----	--------

12. Note: This problem is NOT easy. In fact, it took me a while to solve it, and I am supposed to be an expert on 306 material. So, if you are struggling to pass this course, I suggest you ignore it. On the other hand, if you are a hot shot and think no problem is beyond you, then by all means go for it. We put it on the problem set to keep some of the hot shots out of mischief. We would not put it on the final, because we think it is too difficult to put on the exam.

    A programmer wrote this program to do something useful. He, however, forgot to comment his code, and now can't remember what the program is supposed to do. Your job is to save him the trouble and figure it out for him. In 20 words or fewer tell us what valuable information the program below provides about the value stored in memory location INPUT. Assume that there is a non-zero value at location INPUT before the program is executed.

    HINT: When testing different values of INPUT pay attention to their bit patterns. How does the bit pattern correspond to the RESULT?

```
            .ORIG x3000
            LD R0, INPUT
             AND R3, R3, #0
            LEA R6, MASKS
            LD R1, COUNT
LOOP        LDR R2, R6, #0
            ADD R3, R3, R3
            AND R5, R0, R2
            BRz SKIP
            ADD R3, R3, #1
            ADD R0, R5, #0
SKIP        ADD R6, R6, #1
```

```
        ADD R1, R1, #-1
        BRp LOOP
        ST R3, RESULT
        HALT
COUNT   .FILL #4
MASKS   .FILL 0xFF00
        .FILL 0xF0F0
        .FILL 0xCCCC
        .FILL 0xAAAA
INPUT   .BLKW 1
RESULT  .BLKW 1
        .END
```

This program identifies the most significant bit position that is set in the value stored at INPUT and stores that bit position in RESULT. For example, if INPUT contained the value 0010 0100 0101 0110, RESULT would contain the value 13 since bit 13 is the most significant bit postition that is a 1.

13. Figure out what the following program does.

```
        .ORIG X3000
        LEA R2, C
        LDR R1, R2, #0
        LDI R6, C
        LDR R5, R1, #-3
        ST R5, C
        LDR R5, R1, #-4
        LDR R0, R2, #1
        JSRR R5
        AND R3, R3, #0
        ADD R3, R3, #7
        LEA R4, B
A       STR R4, R1, #0
        ADD R4, R4, #2
        ADD R1, R1, #1
        ADD R3, R3, #-1
        BRP A
        HALT
B       ADD R2, R2, #1
        LDR R0, R2, #0
```

```
        JSRR R5
        TRAP X29
        ADD R2, R2, #15
        ADD R0, R2, #3
        LD R5, C
        TRAP X2B
        ADD R2, R2, #5
        LDR R0, R2, #0
        JSRR R5
        TRAP X27
        JSRR R5
        JSRR R6
C       .FILL X25
        .STRINGZ "EE306 and tests are awesome"
        .END
```

The short answer is that the program outputs “EE some” this is because we over write the trap vector table. Below is a commented version of the program to help you see what is going on.

```
        .ORIG X3000
        LEA R2, C
        LDR R1, R2, #0 ; load x25 into R1
        LDI R6, C      ; loads the starting address of the HALT trap service routine into R6
        LDR R5, R1, #-3 ; loads the starting address of (x25 – 3) trap x22 (puts) into R5
        ST R5, C       ; stores the starting address or puts into C
        LDR R5, R1, #-4 ; loads the starting address of (x25 – 4) trap x21 (out) into r5
        LDR R0, R2, #1 ; loads R0 with the first charater of the stringz “E”
        JSRR R5        ; does the out routine (outputs “E” to the display)
        AND R3, R3, #0 ; clears r3
        ADD R3, R3, #7 ; makes r3 7
        LEA R4, B      ; loads the address of B into r4

;NOTE Loop A overwrites the trap vector table, x25 to x2b
; This makes trap x25 – trap x2b point to this program, see label B and below


A       STR R4, R1, #0 ; overwrites the trap vector with the address in R4
        ADD R4, R4, #2
        ADD R1, R1, #1
        ADD R3, R3, #-1
        BRP A
        HALT           ; What does this do? Trap x25, what is now at memory location x25?

;In the following section <- trap xY indicates what address is in memory location Y
B       ADD R2, R2, #1 ; <- trap x25 (makes R2 point to the first character in the stringz “E”)
        LDR R0, R2, #0 ; (loads r0 with the ascci code for “E“)
```

```
        JSRR R5         ; <- trap x26 (what is in r5? The starting address of out,outputs “E” on the screen)
        TRAP X29
        ADD R2, R2, #15; <- trap x27 (makes r2 points to the (6 + 15) 21th character of the .stringz
        ADD R0, R2, #3 ; (makes to point to the (21+3) 24th character of the stringz the s in awesome)
        LD R5, C        ; <- trap x28 (LD R5, C loads r5 with the starting address of puts)
        TRAP X2B
        ADD R2, R2, #5 ; <- trap x29 (“makes R2 point to the 6th character in the .stringz “ “)
        LDR R0, R2, #0 ; (loads r0 with the ascci code for “ “)
        JSRR R5         ; <- trap x2a (outputs a space on the screen)
        TRAP X27
        JSRR R5         ; <- trap x2b (jsrr to puts outputs “some” to the screen)
        JSRR R6         ; remember r6 contains the starting address of trap x25 (halt) so this halts

C       .FILL X25
        .STRINGZ "EE306 and tests are awesome"
        .END
```

14. (Adapted from 9.53)

Suppose we want to introduce two extra interrupts to the LC-3: INTA and INTB. INTA has priority 2 and an interrupt vector of x50. INTB has priority 4 and an interrupt vector of x60.

Recall that the priority is specified in bits [10:8] of the PSR. In fact, the full PSR specification is:

	15	14	13	12	11	10 9 8	7	6	5	4	3	2	1	0
PSR:	Pr	0	0	0	0	Priority	0	0	0	0	0	N	Z	P

where PSR[15] = 0 (Supervisor mode), 1 (User mode).
PSR[14:11] = 0000
PSR[10:8] = priority, 0 (lowest) to 7 (highest).
PSR[7:3] = 00000
PSR[2:0] = condition codes for N,Z,P

We want to provide some flexibility for developers to add their own INTA and INTB service routines, so we currently left them blank as shown below:

INTA service routine: INTB service routine:

```
.ORIG x1000
RTI
.END
```

```
.ORIG x2000
RTI
.END
```

a. In order to support INTA and INTB, the interrupt vector table must have entries. Show the addresses of these entries and the contents of those memory locations.

Memory Address	**Content**
x0150	x1000
x0160	x2000

b. Show how the content of PSR changes after the following user program starts executing at priority 0 and right before the HALT instruction.

```
        .ORIG x3000
        LD     R0, NUM
        INTA occurs
        AND  R1, R1, # 0
        ADD  R1, R1, # 5
        INTB occurs
        ADD  R2, R0, R1
        HALT
NUM     .FILL xFFF1
```

Changes	**PSR Content**
Initial	1 0000 000 00000 010
1	1 0000 000 00000 100
2	0 0000 010 00000 100
3	1 0000 000 00000 100
4	1 0000 000 00000 010
5	1 0000 000 00000 001
6	0 0000 100 00000 001

7	1 0000 000 00000 001
8	1 0000 000 00000 100