hw5

T1

> P245 7.3

What is the problem with using the string AND as a label?

T2

> p247 7.9

What is the purpose of the .END pseudo-op? How does it differ from the HALT instruction?

T3

Whether multiple .END pseudo-ops can exist in one file? And please explain your answer.

> Hint: if ok, how? if not, why.

T4

> Adapted from 7.10

The following program fragment has an error in it.

```
     ADD  R3, R3, #30
     ST   R3, A
     HALT
A    .BLKW 1
```

(a) Identify the error and explain how to fix it.

b) Will this error be detected when this code is assembled or when this code is run on the LC-3?

T5

> P250 7.17

Suppose you write two separate assembly language modules that you expect to be combined by the linker. Each module uses the label `AGAIN`, and neither module contains the pseudo-op `.EXTERNAL AGAIN`.

Is there a problem using the label AGAIN in both modules? Why or why not?

T6

> Adapted from 7.19

When the following LC-3 program is executed, a) how many times will the instruction at the memory address labeled LOOP execute?

```
     .ORIG x3005
     LEA R2, DATA
     LDR R4, R2, #0
LOOP ADD R4, R4, #-3
     BRzp LOOP
     TRAP x25
DATA .FILL x8002
     .END
```

b) Now if we replace the instruction in x300A with `DATA .FILL x8003`, will your answer be the same?

T7

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  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
                 _______________

AGAIN           ADD R2, R2, R0
                 _______________
                 BRp AGAIN
DONE            ADD R0, R2, #0
                 JSR PUSH
                 _______________
                 LD R0, SAVER0
                 LD R1, SAVER1
                 LD R2, SAVER2
                 LD R5, SAVER5
                 RET
```

T8

Figure 8.18 in P286 describes a FIB subroutine as shown below:

```
;FIB subroutine
; + FIB(0) = 0
; + FIB(1) = 1
; + FIB(n) = FIB(n-1) + FIB(n-1)
;
; Input is in R0
; Return answer in R1
;
FIB  ADD R6, R6, #-1
     STR R7, R6, #0   ; Push R7, the return linkage
     ADD R6, R6, #-1
     STR R0, R6, #0   ; Push R0, the value of n
     ADD R6, R6, #-1
     STR R2, R6, #0   ; Push R2, which is needed in the subroutine

; Check for base case
     AND R2, R0, #-2
     BRnp SKIP        ; Z=0 if R0=0,1
     ADD R1, R0, #0   ; R0 is the answer
     BRnzp DONE

; Not a base case, do the recursion
SKIP ADD R0, R0, #-1
     JSR FIB          ; R1 = FIB(n-1)
     ADD R2, R1, #0   ; Move result before calling FIB again
     ADD R0, R0, #-1
     JSR FIB          ; R1 = FIB(n-2)
     ADD R1, R2, R1   ; R1 = FIB(n-1) + FIB(n-2)

; Restore registers and return
DONE LDR R2, R6, #0
     ADD R6, R6, #1
     LDR R0, R6, #0
     ADD R6, R6, #1
     LDR R7, R6, #0
     ADD R6, R6, #1
     RET
```

a) Is `R0` *caller save* or *callee save*? What about `R2` and `R7`?

b) The following table shows the instruction cycles used to execute this subroutine for corresponding input n:

n	2	3	4	5	6	7	8	9	10
$f(n)$ (cycles)	55	93	169	283	473	777	1271	2069	3361

Can you define $f(n)$ recursively(递归地) and explain your definition?

c) How can you improve the efficiency of this **recursive** subroutine? Just describe your idea briefly.

T9

> 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.

> Hint: 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        ____________________
            ____________________
            ____________________

POP         ____________________
            ____________________
            ____________________
```

T10

> 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 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 ABCD, create a sequence of pushes and pops such that the output stream is BDCA.

b) If the input stream is ABCD, is it posible to create a sequence of pushes and pops such that the output stream is DBCA? Why?

c) If the input stream is ABCDE, how many different output streams can be created? Only consider output streams that are 5 characters long?

T11

> adapted from P258 7.33

We have a program with some missing instructions, and we have a table consisting of some information and some missing information associated with five specific clock cycles of the program's execution. Your job is to complete both!

Insert the missing instructions in the program and the missing information in the table. Cycle numbering starts at 1. That is, cycle 1 is the first clock cycle of the processing of LD R0,A. Note that we have not said anything about the number of clock cycles a memory access takes.

You do have enough information to figure that out for yourself. Note that we are asking for the value of the registers DURING each clock cycle.

```
     .ORIG x3000
     LD R0, A
     LD R1, B
     NOT R1, R1
     ADD R1, R1, #1
     AND R2, R2, #0
AGAIN -------------- (a)
     -------------- (b)
     BRnzp AGAIN
DONE ST R2, C
     HALT
A    .FILL #5
B    .FILL -------- (c)
C    .BLKW #1
     .END
```

cycle number	state number	information
		LD.REG:1 DR: GateMDR: LD.CC: GateALU: GatePC:
16	30	LD.MDR: MDR: IR: LD.IR:
50		LD.REG:1 MDR: x_4A_ BUS: x0001 DR： GateMDR:
57	1	PC: IR: x_040 BUS: x0003 GateALU: GatePC:
	22	ADDR1MUX: ADDR2MUX: LD.PC: 1 PC: x3008 PCMUX: ADDER

What is stored in C at the end of execution for the specific operands given in memory locations A and B?

How many cycles does it take to access memory?

Actually, the program was written by a student, so as expected, he did not get it quite right. Almost, but not quite! Your final task on this problem is to examine the code, figure out what the student was trying todo, and point out where he messed up and how you would fix it. It is not necessary to write any code, just explain briefly how you would fix it.

What was the student trying to do?

How did the student mess up?

How would you fix his program?

> Hint: you may need to refer to Figure C.2 and Figure C.3.