F.7 Chapter 7 Solutions

7.1 0xA7FE

7.2 0x23FF

7.3 Using an instruction as a label confuses the assembler because it treats the label as the opcode itself so the label AND will not be entered into the symbol table. Instead the assembler will give an error in the second pass.

7.4 The Symbol Table generated by the assembler is given below.

Symbol	Address
Test	x301F
Finish	x3027
Save3	x3029
Save2	x302A

7.5 (a) The program calculates the product of values at addresses M0 and M1. The product is stored at address RESULT.

$$\text{mem[RESULT]} = \text{mem[M0]} * \text{mem[M1]}$$

(b) x200C

7.6 Correction: Please correct the problem to read as follows. Also please look at the errata for a new problem 7.26.

Our assembler has crashed and we need your help! Create a symbol table for the program shown below, and assemble the instructions at labels A, B and D.

```
        .ORIG    x3000
        AND      R0, R0, #0
A       LD       R1, E
        AND      R2, R1, #1
        BRp      C
B       ADD      R1, R1, #-1
C       ADD      R0, R0, R1
        ADD      R1, R1, #-2
D       BRp      C
        ST       R0, F
        TRAP     x25
E       .BLKW    1
F       .BLKW    1
        .END
```

You may assume another module deposits a positive value into E before the module executes. In fifteen words or fewer, what does the above program do?

Solution:

The Symbol Table generated by the assembler is given below.

Symbol	Address
A	0x3001
B	0x3004
C	0x3005
D	0x3007
E	0x300A
F	0x300B

Assembly of instructions at A, B, and D:
A: 0010001000001000
B: 0001001001111111
D: 0000001111111101

It calculates the sum of the odd numbers between the value in E and zero and stores the result in F.

7.7 The assembly language program is:

```
           .ORIG   x3000
           AND     R5, R5, #0
           ADD     R5, R5, #1 ;R5 will act as a mask to
                              ;mask out the unneeded bit
           AND     R1, R1, #0 ;zero out the result register
           AND     R2, R2, #0 ;R2 will act as a counter
           LD      R3, NegSixt
MskLoop    AND     R4, R0, R5 ;mask off the bit
           BRz     NotOne     ;if bit is zero then don't
                              ;increment the result
           ADD     R1, R1, #1 ;if bit is one increment
                              ;the result
NotOne     ADD     R5, R5, R5 ;shift the mask one bit left
           ADD     R2, R2, #1 ;increment counter (tells us
                              ;where we are in bit pattern)
           ADD     R6, R2, R3
           BRn     MskLoop    ;not done yet go back and
                              ;check other bits
           HALT
NegSixt    .FILL   #-16
           .END
```

7.8 Register File:

Register	Value
R0	0xA400
R1	0x23FF
R2	0xE1FF
R3	0xA401
R4	0x0000
R5	0x0000
R6	0x0000
R7	0x0000

7.9 The .END pseudo-op tells the assembler where the program ends. Any string that occurs after that will be disregarded and not processed by the assembler. It is different from HALT instruction in very fundamental aspects:

1. It is not an instruction, it can never be executed.
2. Therefore it does not stop the machine.
3. It is just a marker that helps the assembler to know where to stop assembling.

7.10 Add R3, R3, #30 contains an immediate value that is too large to be stored in the Add instruction's immediate value. This instruction cannot be translated by the assembler, thus the error is detected when the program is assembled, not run on the LC-3.

7.11

```
; Prog 7.11
; This code does not perform error checking
; It accepts 3 characters as input
; The first one is either x or #
; The next two is the number.

.ORIG   x3000
IN                ; input the first char - either x or #
AND     R3, R3, #0
ADD     R3, R3, #9 ; R3 = 9 if we are working
           ; with a decimal or 16 if hex
LD      R4, NASCIID
LD      R5, NHEXDIF

LD      R1, NCONSD
ADD     R1, R1, R0
BRz     GETNUMS
LD      R1, NCONSX
ADD     R1, R1, R0
BRnp    FAIL
ADD     R3, R3, #6   ; R3 = 15
```

```
GETNUMS IN
        ST      R0, CHAR1
        IN
        ST      R0, CHAR2
        LEA     R6, CHAR1
        AND     R2, R2, #0
        ADD     R2, R2, #2   ; Loop twice
; Using R2, R3, R4, R5, R6 here
        AND     R0, R0, #0   ; Result

LOOP    ADD     R1, R3, #0
        ADD     R7, R0, #0
LPCUR   ADD     R0, R0, R7
        ADD     R1, R1, #-1
        BRp     LPCUR

        LDR     R1, R6, #0
        ADD     R1, R1, R4

        ADD     R0, R0, R1

        ADD     R1, R1, R5
        BRn     DONECUR
        ADD     R0, R0, #-7    ; for hex numbers
DONECUR
        ADD     R6, R6, #1
        ADD     R2, R2, #-1
        BRp     LOOP

        ; R0 has number at this point

        AND     R2, R2, #0
        ADD     R2, R2, #8

        LEA     R3, RESEND
        LD      R4, ASCNUM
        AND     R5, R5, #0
        ADD     R5, R5, #1

STLP    AND     R1, R0, R5
        BRp     ONENUM
        ADD     R1, R4, #0
        BRnzp   STORCH
ONENUM  ADD     R1, R4, #1
STORCH  ADD     R5, R5, R5
```

```
        STR     R1, R3, #-1
        ADD     R3, R3, #-1
        ADD     R2, R2, #-1
        BRp     STLP
        LEA     R0, RES
        PUTS
FAIL    HALT
CHAR1   .FILL   x0
CHAR2   .FILL   x0

ASCNUM  .FILL   x30
NHEXDIF .FILL   xFFEF   ; -x11
NASCIID .FILL   xFFD0   ; -x30
NCONSX  .FILL   xFF88   ; -x78
NCONSD  .FILL   xFFDD   ; -x23

RES     .BLKW 8
RESEND  .FILL x0
        .END
```

7.12 This program checks if the top 8 bits of the value in memory location x4000 are the same as the lower 8 bits of the same value. If they the same R5 is set to 1. If they are not the same R5 is set to 0.

7.13 Error 1:
Line 8: ST R1, SUM
SUM is an undefined label. This error will be detected at assembly time.

Error 2:
Line 3: ADD R1, R1, R0
R1 was not initialized before it was used; therefore, the result of this ADD instruction may not be correct. This error will be detected at run time.

7.14 (a)
```
1011 000 0 0000 0010 ( STI R0, x2 )
1111 0000 00100001   ( TRAP x21 )
1111 0000 00100101   ( TRAP x25 )
00000000 00100101    ( '\%' )
```

(b) STI should be replaced with LD.

(c) The STI instruction stores the value in R0 to the memory location stored at the addresss labeled LABEL. The value in R0 is 0x3000. The address stored at LABEL is the ASCII code for the '%' character. This ascii code is 0x25. The STI instruction therefore stores the value 0x3000 at address 0x0025.

The Out instruction outputs the NUL ASCII code.

The Halt instruction's trap vector is 0x25. The instruction jumps to the address located at 0x0025. This address is meant to point to a trap service routine. However, the first STI instruction stored the value 0x3000 to 0x0025. Therefore the HALT instruction causes control to jump back to the beginning of the program and the program is stuck in an infinite loop.

7.15 This program doubles all the positive numbers and leaves the negative numbers unchanged.

7.16 This program counts the number of even and the number of odd integers. It stores the number of even integers in R3 and stores the number of odd integers in R4.

7.17 There is not a problem in using the same label in separate modules assuming the programmer expected the label to refer to different addresses, one within each module. This is not a problem because each module has its own symbol table associated with it. It is an error on the otherhand if the programmer expected each label AGAIN to refer to the same address.

7.18 a) LDR R3,R1,#0

b) NOT R3,R3

c) ADD R3,R3,#1

or

a) LDR R3,R1,#0

b) NOT R4,R4

c) ADD R4,R4,#1

7.19 The instruction labeled LOOP executes 4 times.

7.20 Please correct Part (a) to read:

```
         .ORIG  x5000
         AND  R0, R0, #0
         ADD  R0, R0, #15
         ADD  R0, R0, #6
         STI  R0, PTR
         HALT
PTR .FILL x4000
.END
```

The difference in the approaches is when the value is actually stored in location x4000. In program (a), the value will be stored at run time. However, since program (b) only uses assembler directives, the value will be stored into x4000 when the object module is loaded into memory.

7.21 Correction: Please use the following LC-3 assembly language program for this problem:

```
        .ORIG x3000
        AND     R0, R0, #0
        ADD     R2, R0, #10
        LD      R1, MASK
        LD      R3, PTR1
LOOP    LDR     R4, R3, #0
        AND     R4, R4, R1
        BRz     NEXT
        ADD     R0, R0, #1
NEXT    ADD     R3, R3, #1
        ADD     R2, R2, #-1
        BRp     LOOP
        STI     R0, PTR2
        HALT
MASK    .FILL   x8000
PTR1    .FILL   x4000
PTR2    .FILL   x5000
```

Solution:
The assembled program:

```
0101 0000 0010 0000 ( AND R0, R0, #0 )
0001 0100 0010 1010 ( ADD R2, R0, #10 )
0010 0010 0000 1010 ( LD R1, MASK )
0010 0110 0000 1010 ( LD R3, PTR1 )
0110 1000 1100 0000 ( LDR R4, R3, #0 )
0101 1001 0000 0001 ( AND R4, R4, R1 )
0000 0100 0000 0001 ( BRz NEXT )
0001 0000 0010 0001 ( ADD R0, R0, #1 )
0001 0110 1110 0001 ( ADD R3, R3, #1 )
0001 0100 1011 1111 ( ADD R2, R2, #-1 )
0000 0011 1111 1001 ( BRp LOOP )
1011 0000 0000 0011 ( STI R0, PTR2 )
1111 0000 0010 0101 ( HALT )
1000 0000 0000 0000
0100 0000 0000 0000
0101 0000 0000 0000
```

This program counts the number of negative values in memory locations 0x4000 - 0x4009 and stores the result in memory location 0x5000.

7.22

```
                .ORIG x3000
                AND R5, R5, #0          ; R5 will contain resulting binary v
                AND R6, R6, #0          ; R6 will contain character count
                AND R4, R4, #0
                ADD R4, R4, #-4         ; to make sure only take 4 character
                LEA R1, BUFFER          ; R1 is pointer to BUFFER
                LD R2, NEGENTER
                LEA R0, PROMPT
                PUTS
;; get input opcode
AGAIN           GETC
                OUT
                ADD R3, R2, R0 ; check for enter
                BRz CONT
                ADD R6, R6, #1 ; increment character count
                ADD R3, R4, R6
                BRp INVALID ; don’t allow more than 4 characters
                STR R0, R1, #0
                ADD R1, R1, #1 ; increment pointer
                BR AGAIN
CONT            LEA R1, BUFFER
                ADD R4, R6, #-1
                BRnz INVALID ; means only 0 or 1 characters
                ADD R4, R6, #-2
                BRz TWO ; 2 characters
                ADD R4, R6, #-3
                BRz THREE ; 3 characters
;; 4 characters - could be JSRR or TRAP
                LDR R3, R1, #0
                LD R2, NEGJ
                ADD R4, R3, R2
                BRnp T_1 ; could be TRAP
                LDR R3, R1, #1
                LD R2, NEGS
                ADD R4, R3, R2
                BRnp INVALID ; starts with J, but isn’t JSRR
                LDR R3, R1, #2
                LD R2, NEGR
                ADD R4, R3, R2
                BRnp INVALID ; starts with JS, but isn’t JSRR
                LDR R3, R1, #3
                ADD R4, R3, R2
                BRnp INVALID ; starts with JSR, but isn’t JSRR
                ADD R5, R5, #4 ; is JSRR
                BR OUTPUT
T_1             LD R2, NEGT
                ADD R4, R3, R2
                BRnp INVALID ; isn’t an LC-3 opcode
                LDR R3, R1, #1
                LD R2, NEGR
                ADD R4, R3, R2
                BRnp INVALID ; starts with T, but isn’t TRAP
                LDR R3, R1, #2
                LD R2, NEGA
                ADD R4, R3, R2
                BRnp INVALID ; starts with TR, but isn’t TRAP
                LDR R3, R1, #3
                LD R2, NEGP
                ADD R4, R3, R2
                BRnp INVALID ; starts with TRA, but isn’t TRAP
                ADD R5, R5, #15 ; is TRAP
                BR OUTPUT
;; 2 characters - could be BR, LD, or ST
TWO             LDR R3, R1, #0
                LD R2, NEGB
                ADD R4, R3, R2
                BRnp L_1 ; could be LD (or ST)
                LDR R3, R1, #1
                LD R2, NEGR
                ADD R4, R3, R2
                BRnp INVALID ; starts with B, but isn’t BR
                BR OUTPUT ; is BR (R5 already contains 0)
L_1             LD R2, NEGL
                ADD R4, R3, R2
                BRnp S_1 ; could be ST
                LDR R3, R1, #1
                LD R2, NEGD
                ADD R4, R3, R2
                BRnp INVALID ; starts with L, but isn’t LD
                ADD R5, R5, #2 ; is LD
                BR OUTPUT
S_1             LD R2, NEGS
                ADD R4, R3, R2
                BRnp INVALID ; isn’t an LC-3 opcode
                LDR R3, R1, #1
                LD R2, NEGT
                ADD R4, R3, R2
                BRnp INVALID ; starts with S, but isn’t ST
                ADD R5, R5, #3 ; is ST
                BR OUTPUT
;; 3 characters - could be ADD, AND, JMP, JSR, LDI, LDR, LEA, NOT, RET, RTI,
THREE           LD R2, NEGA
                LDR R3, R1, #0
                ADD R4, R3, R2
                BRnp J_OP ; go check next opcode
                LDR R3, R1, #1
                LD R2, NEGD
                ADD R4, R3, R2
                BRnp N_1 ; could be AND
                LDR R3, R1, #2
                ADD R4, R3, R2
                BRnp INVALID ; starts with AD, but isn't ADD
                ADD R5, R5, #1 ; is ADD
                BR OUTPUT
N_1             LD R2, NEGN
                ADD R4, R3, R2
                BRnp INVALID ; starts with A, but isn't ADD or AND
                LDR R3, R1, #2
                LD R2, NEGD
                ADD R4, R3, R2
                BRnp INVALID ; starts with AN, but isn't AND
                ADD R5, R5, #5 ; is AND
                BR OUTPUT
J_OP            LD R2, NEGJ
                ADD R4, R3, R2
                BRnp L_OP ; go check next set of opcodes
                LDR R3, R1, #1
                LD R2, NEGM
                ADD R4, R3, R2
                BRnp S_2 ; could be JSR
                LDR R3, R1, #2
                LD R2, NEGP
                ADD R4, R3, R2
                BRnp INVALID ; starts with JM, but isn't JMP
                ADD R5, R5, #12 ; is JMP
                BR OUTPUT
S_2             LD R2, NEGS
                ADD R4, R3, R2
                BRnp INVALID ; starts with J, but isn't JMP or JSR
                LDR R3, R1, #2
                LD R2, NEGR
                ADD R4, R3, R2
                BRnp INVALID ; starts with JS, but isn't JSR
                ADD R5, R5, #4 ; is JSR
                BR OUTPUT
L_OP            LD R2, NEGL
                ADD R4, R3, R2
                BRnp N_OP ; check next opcode
                LDR R3, R1, #1
                LD R2, NEGD
                ADD R4, R3, R2
                BRnp E_1 ; could be LEA
                LDR R3, R1, #2
```

```
               LD R2, NEGI
               ADD R4, R3, R2
               BRnp R_1 ; could be LDR
               ADD R5, R5, #10 ; is LDI
               BR OUTPUT
R_1            LD R2, NEGR
               ADD R4, R3, R2
               BRnp INVALID ; starts with LD, but isn't LDI or LDR
               ADD R5, R5, #6 ; is LDR
               BR OUTPUT
E_1            LD R2, NEGE
               ADD R4, R3, R2
               BRnp INVALID ; starts with L, but isn't LDI, LDR, or LEA
               LDR R3, R1, #2
               LD R2, NEGA
               ADD R4, R3, R2
               BRnp INVALID ; starts with LE, but isn't LEA
               ADD R5, R5, #14 ; is LEA
               BR OUTPUT
N_OP           LD R2, NEGN
               ADD R4, R3, R2
               BRnp R_OP ; go check next set of opcodes
               LDR R3, R1, #1
               LD R2, NEGO
               ADD R4, R3, R2
               BRnp INVALID ; starts with N, but isn't NOT
               LDR R3, R1, #2
               LD R2, NEGT
               ADD R4, R3, R2
               BRnp INVALID ; starts with NO, but isn't NOT
               ADD R5, R5, #9 ; is NOT
               BR OUTPUT
R_OP           LD R2, NEGR
               ADD R4, R3, R2
               BRnp S_OP ; go check next set of opcodes
               LDR R3, R1, #1
               LD R2, NEGE
               ADD R4, R3, R2
               BRnp T_2 ; could be RTI
               LDR R3, R1, #2
               LD R2, NEGT
               ADD R4, R3, R2
               BRnp INVALID ; starts with RE, but isn't RET
               ADD R5, R5, #12 ; is RET
               BR OUTPUT
```

```
T_2             LD R2, NEGT
                ADD R4, R3, R2
                BRnp INVALID ; starts with R, but isn't RET or RTI
                LDR R3, R1, #2
                LD R2, NEGI
                ADD R4, R3, R2
                BRnp INVALID ; starts with RT, but isn't RTI
                ADD R5, R5, #8 ; is RTI
                BR OUTPUT
S_OP            LD R2, NEGS
                ADD R4, R3, R2
                BRnp INVALID ; isn't an LC-3 opcode
                LDR R3, R1, #1
                LD R2, NEGT
                ADD R4, R3, R2
                BRnp INVALID ; starts with S, but isn't STI or STR
                LDR R3, R1, #2
                LD R2, NEGI
                ADD R4, R3, R2
                BRnp R_2 ; could be STR
                ADD R5, R5, #11 ; is STI
                BR OUTPUT
R_2             LD R2, NEGR
                ADD R4, R3, R2
                BRnp INVALID ; starts with ST, but isn't STI or STR
                ADD R5, R5, #7 ; is STR
;; output binary for opcode
OUTPUT          LD R0, ENTER
                OUT
                AND R4, R4, #0
                ADD R4, R4, #12
; shift bits [3:0] into [15:12]
SHIFT           ADD R5, R5, R5
                ADD R4, R4, #-1
                BRp SHIFT
                ADD R4, R4, #4
; output 4 bits of opcode
OUT_LOOP        ADD R5, R5, #0          ; set CC based on R5
                BRn OUT_1
                LD R0, ZERO
                OUT
                BR SHIFT2
OUT_1           LD R0, ONE
                OUT
SHIFT2          ADD R5, R5, R5
```

```
                ADD R4, R4, #-1
                BRp OUT_LOOP
                BR DONE
; invalid opcode message
INVALID         LD R0, ENTER
                OUT
                LEA R0, ERROR
                PUTS
;
DONE            TRAP x25
NEGENTER        .FILL xFFF6
BUFFER          .BLKW 4
ENTER           .FILL x000A
NEGA            .FILL xFFBF
NEGB            .FILL xFFBE
NEGD            .FILL xFFBC
NEGE            .FILL xFFBB
NEGI            .FILL xFFB7
NEGJ            .FILL xFFB6
NEGL            .FILL xFFB4
NEGM            .FILL xFFB3
NEGN            .FILL xFFB2
NEGO            .FILL xFFB1
NEGP            .FILL xFFB0
NEGR            .FILL xFFAE
NEGS            .FILL xFFAD
NEGT            .FILL xFFAC
ONE             .FILL x0031
ZERO            .FILL x0030
PROMPT          .STRINGZ "Type an LC-3 opcode (in all caps): "
ERROR           .STRINGZ "Invalid opcode!"
                .END
```

7.23 (a) ADD R1, R1, #-1
(b) LDR R4, R1, #0
(c) ADD R0, R0, #1
(d) ADD R1, R1, #-1
(e) BR LOOP

7.24 When the BR LOOP instruction is executed, it checks the value of the condition codes, which have been set based on the value written into R3 as a result of the ADD R3, R3, R3 instruction. The condition codes are not changed by the branch instruction, so when the branch back to the label LOOP is taken, the next branch instruction (BRz DONE) will also use the condition codes as set by the value written into R3. However, the BRz DONE instruction should be branching based on the value in the register that is used to keep track of the loop counter, which is R2.

This problem can be fixed by switching the instructions ADD R2, R2, #-1 and ADD R3, R3, R3.

7.25 This is an assembler error. The number 0xFF004 does not fit in one LC-3 memory location and therefore this .FILL cannot be assembled.

7.26 Note: This is a new problem.

Problem Statement: Recall the assembly language program of problem 7.6. Consider the following program:

```
        .ORIG     x3000
        AND       R0, R0, #0
D       LD        R1, A
        AND       R2, R1, #1
        BRp       B
E       ADD       R1, R1, #-1
B       ADD       R0, R0, R1
        ADD       R1, R1, #-2
F       BRp       B
        ST        R0, C
        TRAP      x25
A       .BLKW     1
C       .BLKW     1
        .END
```

The assembler translates both assembly language programs into machine language programs. What can you say about the two resulting machine language programs?

Solution:

The two machine language programs are going to be exactly the same.