F.1 Chapter 1 Solutions

1.1 Every computer can do the same thing as every other computer. A smaller or slower computer will just take longer.

1.2 No.

1.3 It is hard to increase the accuracy of analog machines.

1.4 Ambiguity.

1.5 (a) inputs to first (x) box are a and x
output of first (x) box is ax
inputs to second (+) box are ax and b
output of second (+) box is $ax + b$

(b) inputs to first (+) box are w and x
output of first (+) box is $w + x$
inputs to second (+) box are y and z
output of second (+) box is $y + z$
inputs to third (+) box are $(w + x)$ and $(y + z)$
output of third (+) box is $w + x + y + z$
inputs to fourth (x) box are $(w + x + y + z)$ and .25
output of fourth (x) box is $0.25(w + x + y + z)$, which is the average

(c) The key is to factor $a^2 + 2ab + b^2 = (a + b)^2$
inputs to first (+) box are a and b
output of first (+) box is $a + b$
inputs to second (x) box are $(a + b)$ and $(a + b)$
output of second (x) box is $(a + b)^2 = a^2 + 2ab + b^2$

1.6 Any ambiguous statement is fine. For example: I ate my sandwich on a bed of lettuce. The sandwich might have been sitting on a bed of lettuce on the plate, or I might have been sitting on a bed of lettuce eating a sandwich.

1.7 If the taxi driver is honorable, he/she asks you whether time or money is more important to you, and then gets you to the airport as quickly or as cheaply as possible. You are freed from knowing anything about the various ways one can get to the airport. If the taxi driver is dishonorable, you get to the airport late enough to miss your flight and/or at a taxi fare far in excess of what it should have been, as the taxi driver takes a very circuitous route.

1.8 He could mean a lot of things. This statement is ambiguous as it could mean different things.

Some reasonable interpretations are: a) John saw the man in “the park with a telescope” b) John saw the “man in the park” with a telescope.

As this statement is ambiguous, it is unacceptable as a statement in a program.

1.9 Yes, if phrased in a way that is definite and lacks ambiguity.

1.10 Definiteness: each step is precisely stated.

Effective Computability: each step can be carried out by a computer.

Finiteness: the procedure terminates.

1.11 (a) Lacks definiteness: Go south on Main St. for a mile or so.

(b) Lacks effective computability: Find the integer that is the square root of 14.

(c) Lacks finiteness: Do something. Repeat forever.

1.12 (a) Lacks definiteness, since it does not specify how two rows are to be added. Also, the 3rd or the 4th row could be added to the first row. So there are two posible answers.

(b) This is not effectively computable, because there is no end to the number line. Anything involving infinity must not be effectively computable. This is also not finite, for the same reason.

(c) This is an algorithm.

(d) This is not finite, so it is not an algorithm. If, as Calvin suspects, the coin is weighted, they will be flipping that coin forever.

(e) This is not finite, so it is not an algorithm. Steps 1 to 6 calculate, albeit in a long way, the number - 1. If the given number is negative or zero, then there will never be a time when you get 0 at the end of step 6.

1.13 Both computers, A and B, are capable of solving the same problems. Computer B can perform subtraction by taking the negative of the second number and adding it to the first one. As A and B are otherwise identical, they are capable of solving the same problems.

1.14 (a) 120 transformation processes.

(b) Any 3 of this form are fine: “Sort Algorithm 3, Fortran program, SPARC ISA, SPARC microarchitecture 1”.

(c) 120 again.

1.15 Advantages of a higher level language: Fewer instructions are required to do the same amount of work. This usually means it takes less time for a programmer to write a program to solve a problem. High level language programs are generally easier to read and therefore know what is going on. Disadvantages of a higher level language: Each instruction has less control over the underlying hardware that actually performs the computation that the program frequently executes less fficiently.
NOTE: this problem is beyond the scope of Chapter 1 or most students.

1.16 Possible operations, data types, addressing modes.

1.17 An ISA describes the interface to the computer from the perspective of the 0s and 1s of the program. For example, it describes the operations, data types, and addressing modes a programmer can use on that particular computer. It doesn’t specify the actual physical implementation. The microarchitecture does that. Using the car analogy, the ISA is what the driver sees, and the microarchitecture is what goes on under the hood.

1.18 A single microarchitecture typically implements only one ISA. However, many microarchitectures usually exist for the same ISA.

1.19 (a) Problem: For example, what is the sum of the ten smallest positive integers.

(b) Algorithm: Any procedure is fine as long as it has definiteness, effective computability, and finiteness.

(c) Language: For example, C, C++, Fortran, IA-32 Assembly Language.

(d) ISA: For example, IA-32, PowerPC, Alpha, SPARC.

(e) Microarchitecture: For example, Pentium III, Compaq 21064.

(f) Circuits: For example, a circuit to add two numbers together.

(g) Devices: For example, CMOS, NMOS, gallium arsenide.

1.20 Refering to the levels of transformation as the levels of abstraction is a reasonable characterization. Each level in Figure 1.6 is essentially a level of abstraction, abstracting the other levels. For example, if the problem statement said "Find the average of two numbers", you have abstracted the rest of the system away. Now, lets take the Language level. If you have a C language program, the lower levels are abstraced away. You dont have to worry about the exact ISA or microarchitecture you will run the programon. Similarily, you should be able to draw examples for all the other levels.

1.21 It is in the ISA of the computer that will run it. We know this because if the word procesing software were in a high- or low-level programming language, then the user would need to compile it or assemble it before using it. This never happens. The user just needs to copy the files to run the program, so it must already be in the correct machine language, or ISA.

1.22 The transformation from Problems to Algorithms is the most difficult step. There is ambiguity in a Problem statement which needs to be resolved in order to generate an algorithm. This requires the intelligence to actually understand the problem and make sense out of it. All the other transformations can be performed by a program written to perform that transformation.

1.23 ISA's don't change much between successive generations, because of the need for backward compatibility. You'd like your new computer to still run all your old software.

F.2 Chapter 2 Solutions

2.1 The answer is 2^n

2.2 For 26 characters, we need at least 5 bits. For 52 characters, we need at least 6 bits.

2.3 (a) For 400 students, we need at least 9 bits.

(b) $2^9 = 512$, so 112 more students could enter.

2.4 2^n integers can be represented. The range would be 0 to (2^n) - 1.

2.5 If each number is represented with 5 bits,

```
 7 = 00111 in all three systems
-7 = 11000 (1's complement)
   = 10111 (signed magnitude)
   = 11001 (2's complement)
```

2.6 100000.

2.7 Refer the following table:

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2.8 The answers are:

(a) 127 in decimal, 01111111 in binary.

(b) -128 in decimal, 10000000 in binary.

(c) (2^{n-1})-1

(d) -(2^{n-1})

2.9 Avogadro's number (6.02 x 10^{23}) requires 80 bits to be represented in two's complement binary representation.

2.10 The answers are:

(a) -6
(b) 90
(c) -2
(d) 14803

2.11 (a) 01100110
(b) 01000000
(c) 00100001
(d) 10000000
(e) 01111111

2.12 It is a multiple of 4.

2.13 (a) 11111010
(b) 00011001
(c) 11111000
(d) 00000001

2.14 (a) 1100
(b) 1010
(c) 1111
(d) 01011
(e) 10000

2.15 Dividing the number by two.

2.16 (a) 11111111 (binary) or -0 (decimal)
(b) 10001110 (binary) or -14(decimal)
(c) 00000000 (binary) or 0 (decimal)

2.17 (a) 1100 (binary) or -4 (decimal)
(b) 01010100 (binary) or 84 (decimal)
(c) 0011 (binary) or 3 (decimal)
(d) 11 (binary) or -1 (decimal)

2.18 The answers are:

(a) 1100 (binary) or 12 (decimal)

(b) 1011000 (binary) or 88 (decimal)
(c) 1011 (binary) or 11 (decimal)
(d) 11 (binary) or 3 (decimal)

2.19 11100101, 1111111111100101, 11111111111111111111111111100101. Sign extension does not affect the value represented.

2.20 (a) 1100 + 0011 = 1111
-4 + 3 = -1
(b) 1100 + 0100 = 0000
-4 + 4 = 0
(c) 0111 + 0001 = 1000 OVERFLOW!
7 + 1 = -8
(d) 1000 - 0001 = 1000 + 1111 = 0111 OVERFLOW!
-8 - 1 = -8 + (-1) = 7
(e) 0111 + 1001 = 0000
7 + -7 = 0

2.21 Overflow has occurred if both operands are positive and the result is negative, or if both operands are negative and the result is positive.

2.22 Any two 16-bit 2's complement numbers that add to more than +32767 or less than -32768 would be correct.

2.23 Overflow has occurred in an unsigned addition when you get a carry out of the leftmost bits.

2.24 Any two 16-bit unsigned numbers that add to more than 65535 would be correct.

2.25 Because their sum will be a number which if positive, will have a lower magnitude (less positive) than the original postive number (because a negative number is being added to it), and vice versa.

2.26 (a) 7 bits.
(b) 63 in decimal (0111111 in binary)
(c) 127 in decimal (1111111 in binary)

2.27 The problem here is that overflow has occurred as adding 2 positive numbers has resulted in a negative number.

2.28 When all of the inputs are 1.

2.29 Refer to the following table:

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

2.30 (a) 01010111
(b) 100
(c) 10100000
(d) 00010100
(e) 0000
(f) 0000

2.31 When atleast one of the inputs is 1.

2.32 Refer to the following table:

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

2.33 (a) 11010111
(b) 111
(c) 11110100
(d) 10111111
(e) 1101
(f) 1101

2.34 (a) 0111
(b) 0111
(c) 1101
(d) 0110

2.35 The masks are used to set bits (by ORing a 1) and to clear bits (by ANDing a 0).

2.36 (a) AND with 11111011
(b) OR with 01000100
(c) AND with 00000000
(d) OR with 11111111
(e) AND the BUSYNESS pattern with 00000100 to isolate b2. Then add that result to itself 5 times.

2.37 [(n AND m AND (NOT s)) OR ((NOT n) AND (NOT m) AND s)] AND 1000

2.38 Let N = n & 1000, M = m & 1000
Overflow = (N AND M) OR ((N OR M) AND ((NOT s) AND 1000))

2.39 (a) 0 10000000 11100000000000000000000

(b) 1 10000100 10111010111000000000000

(c) 0 10000000 10010010000111111011011

(d) 0 10001110 11110100000000000000000

2.40 (a) 2

(b) -17

(c) Positive infinity. NOTE: This was not explained in the text.

(d) -3.125

2.41 (a) 127

(b) -126

2.42 The ASCII values are being added, rather than the integer values. (ASCII ”5” is 53 in decimal, and ASCII ”8” is 56 in decimal, adding to 109, which is ASCII ”m”.)

2.43 (a) Hello!

(b) hELLO!

(c) Computers!

(d) LC-2

2.44 Add 0011 0000 (binary) or x30.

2.45 (a) xD1AF

(b) x1F

(c) x1

(d) xEDB2

2.46 (a) 0001 0000

(b) 1000 0000 0001

(c) 1111 0111 0011 0001

(d) 0000 1111 0001 1110 0010 1101

(e) 1011 1100 1010 1101

2.47 (a) -16

(b) 2047

(c) 22

(d) -32768

2.48 (a) x100

(b) x6F

(c) x75BCD15

(d) xD4

2.49 (a) x2939

(b) x6E36

(c) x46F4

(d) xF1A8

(e) The results must be wrong. In (3), the sum of two negative numbers produced a positive result. In (4), the sum of two positive numbers produced a negative result. We call such additions OVERFLOW.

2.50 (a) x5468

(b) xBBFD

(c) xFFFF

(d) x32A3

2.51 (a) x644B

(b) x4428E800

(c) x48656C6C6F

2.52 Refer to the table below.

	x434F4D50	x55544552
Unsigned Binary	1,129,270,608	1,431,586,130
1's Complement	1,129,270,608	1,431,586,130
2's Complement	1,129,270,608	1,431,586,130
IEEE 754 floating point	207.302001953125	14,587,137,097,728
ASCII String	COMP	UTER

2.53 Refer to the table below:

A	B	Q1	Q2
0	0	1	0
0	1	1	1
1	0	1	1
1	1	0	1

Q2 = A OR B

2.54 Refer to the table below:

X	Y	Z	Q1	Q2
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

2.55 (a) 63

(b) 4^n - 1

(c) 310

(d) 222

(e) 11011.11

(f) 0100 0001 1101 1110 0000 0000 0000 0000

(g) 4^{4m}

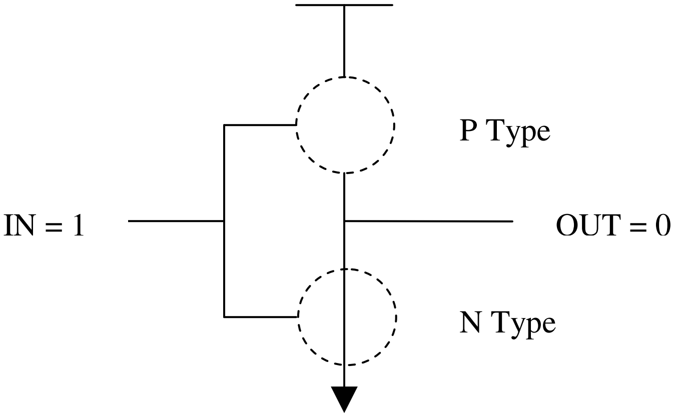
2.56 - 1.101 x $2^{(12-7)}$ = -52

F.3 Chapter 3 Solutions

3.1

	N-Type	P-Type
Gate=1	closed	open
Gate=0	open	closed

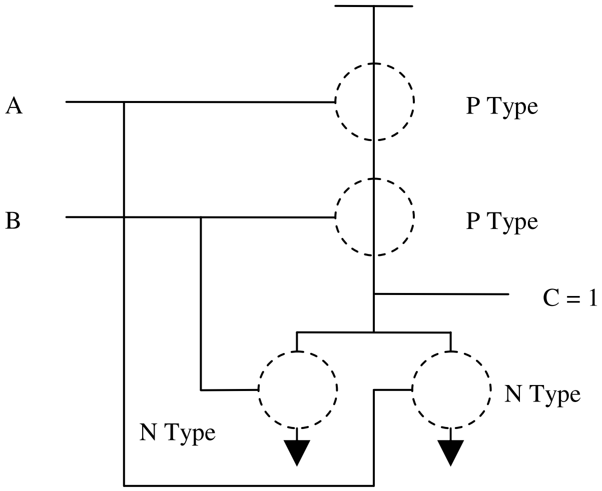
3.2

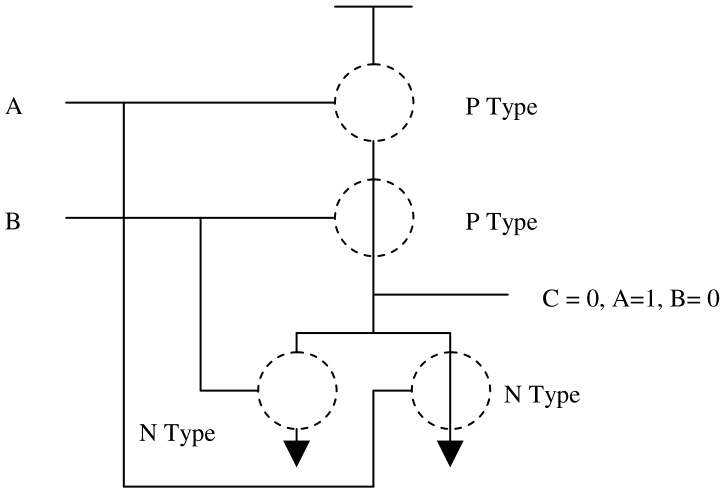


3.3 There can be 16 different two input logic functions.

3.4

A	B		C
0	0		1
0	1		0
1	0		0
1	1		0





3.5

A	B	C		OUT
0	0	0		1
0	0	1		0
0	1	0		1
0	1	1		0
1	0	0		1
1	0	1		0
1	1	0		0
1	1	1		0

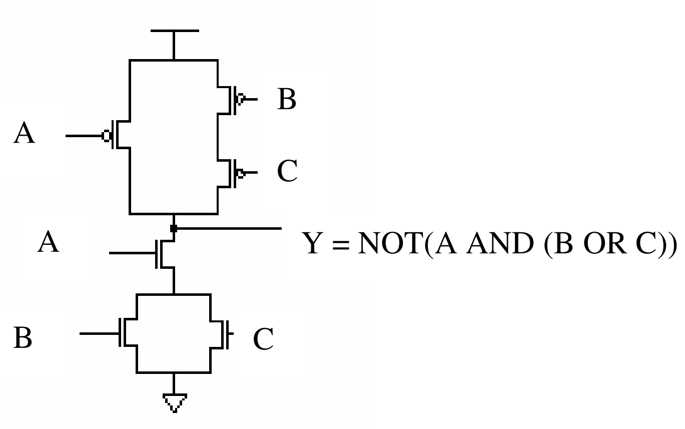
3.6 C = A’; D = B’; Z = (C+D)’ = (A’+B’)’ = A . B

A	B		C	D	Z
0	0		1	1	0
0	1		1	0	0
1	0		0	1	0
1	1		0	0	1

3.7 There is short circuit (path from Power to Ground) when either A = 1 and B = 0 or A = 0 and B = 1.

3.8 Correction: Please correct the logic equation to

Y = NOT (A AND (B OR C))



3.9

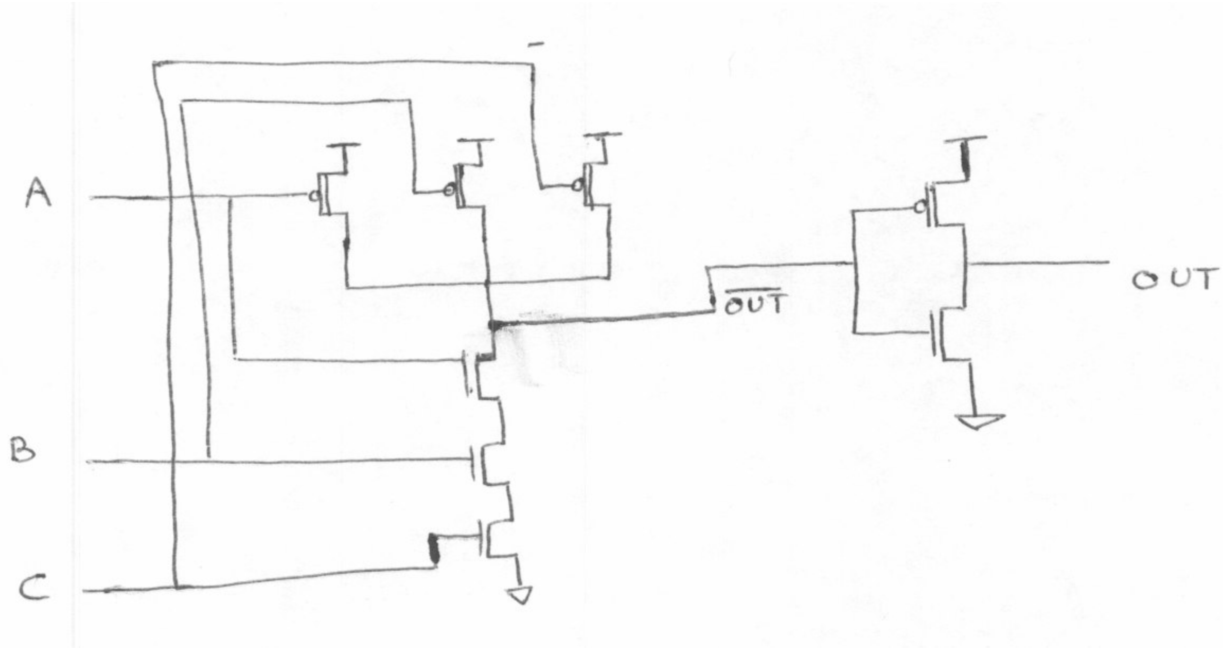
A	B		NOT(NOT(A) OR NOT(B))
0	0		0
0	1		0
1	0		0
1	1		1

AND gate has the same truth table.

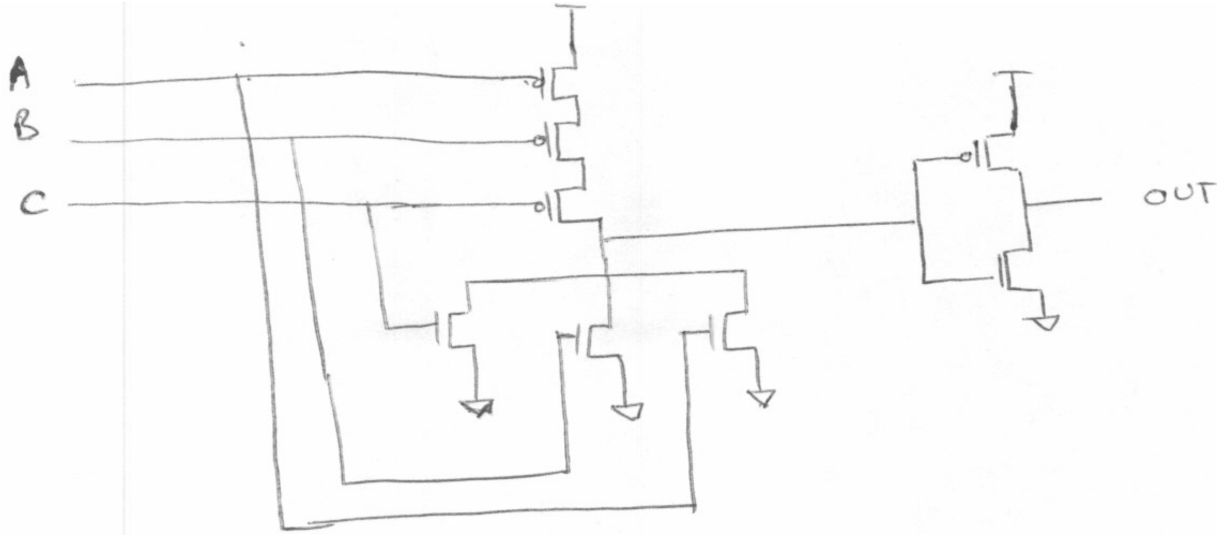
3.10

A	B		A NOR B
0	0		1
0	1		0
1	0		0
1	1		0

3.11 a. Three input And-Gate

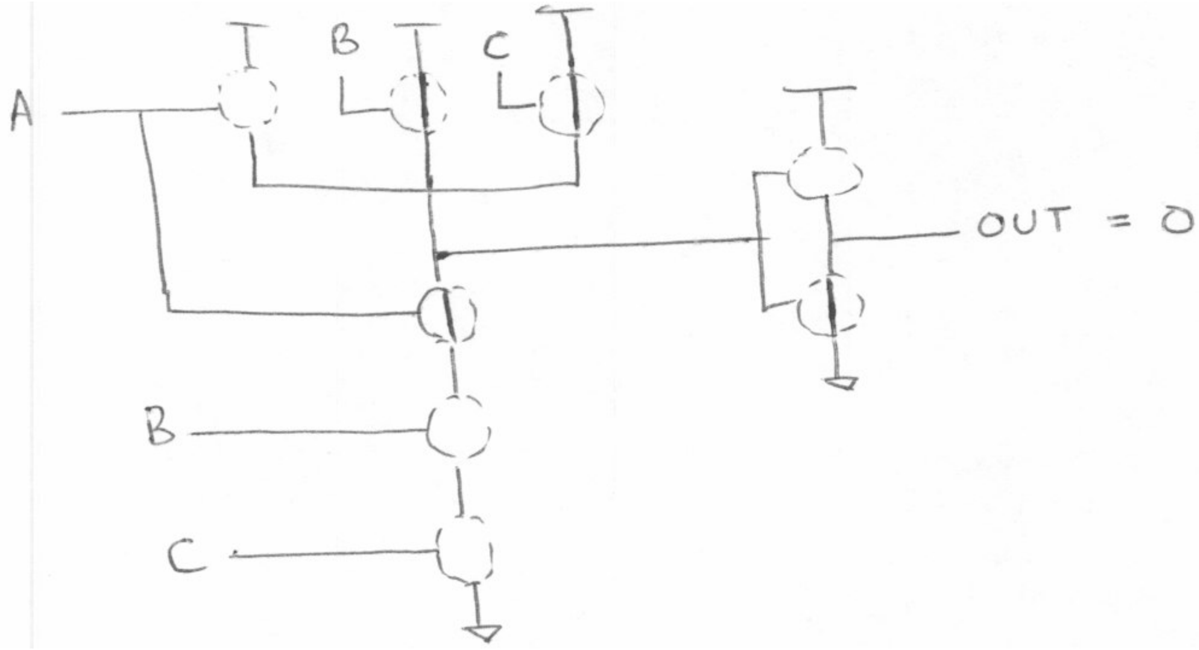


Three input OR-Gate

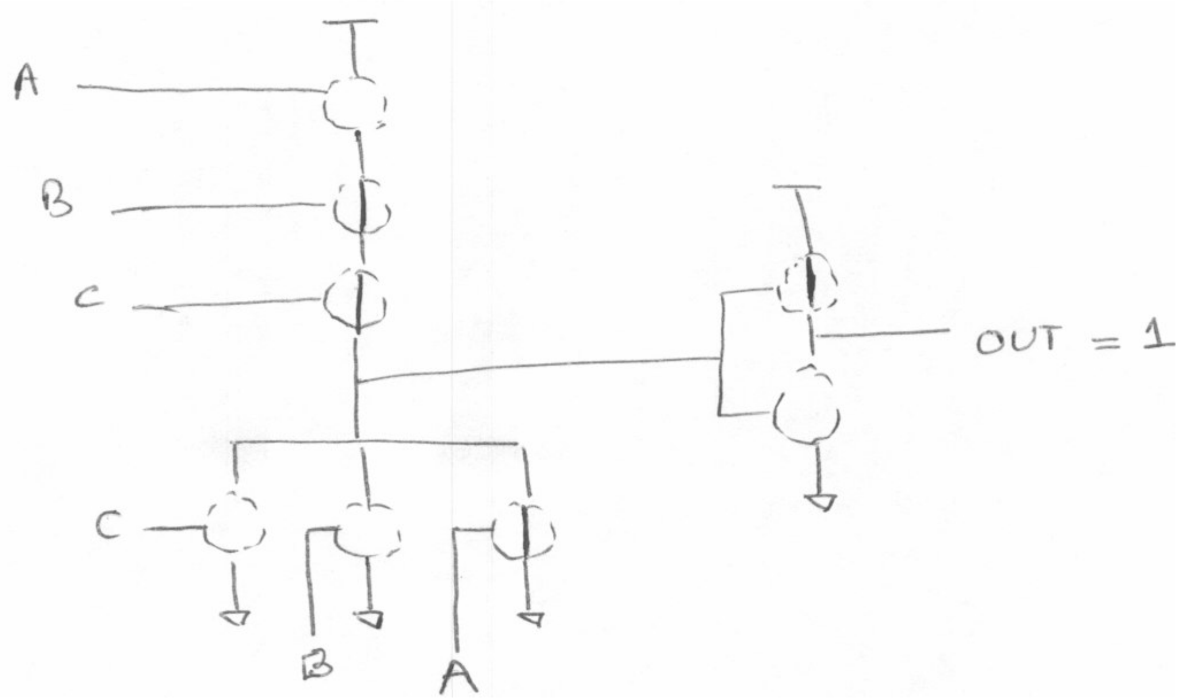


b. (1) A = 1, B = 0, C = 0.

AND Gate

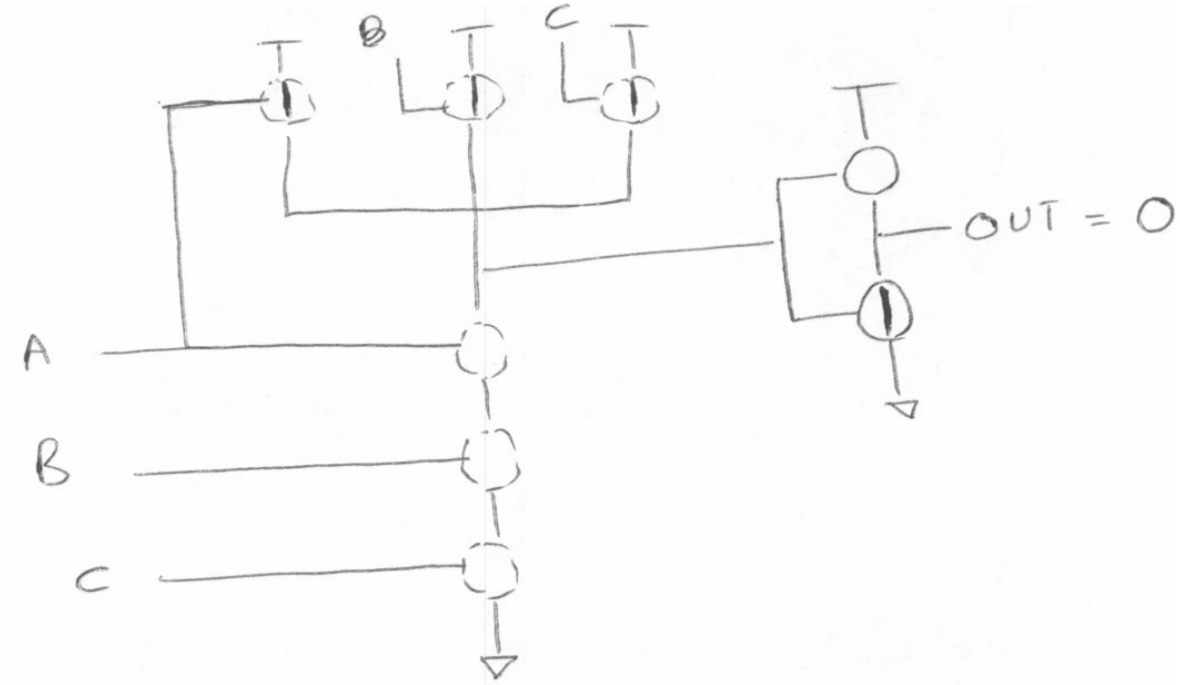


OR Gate

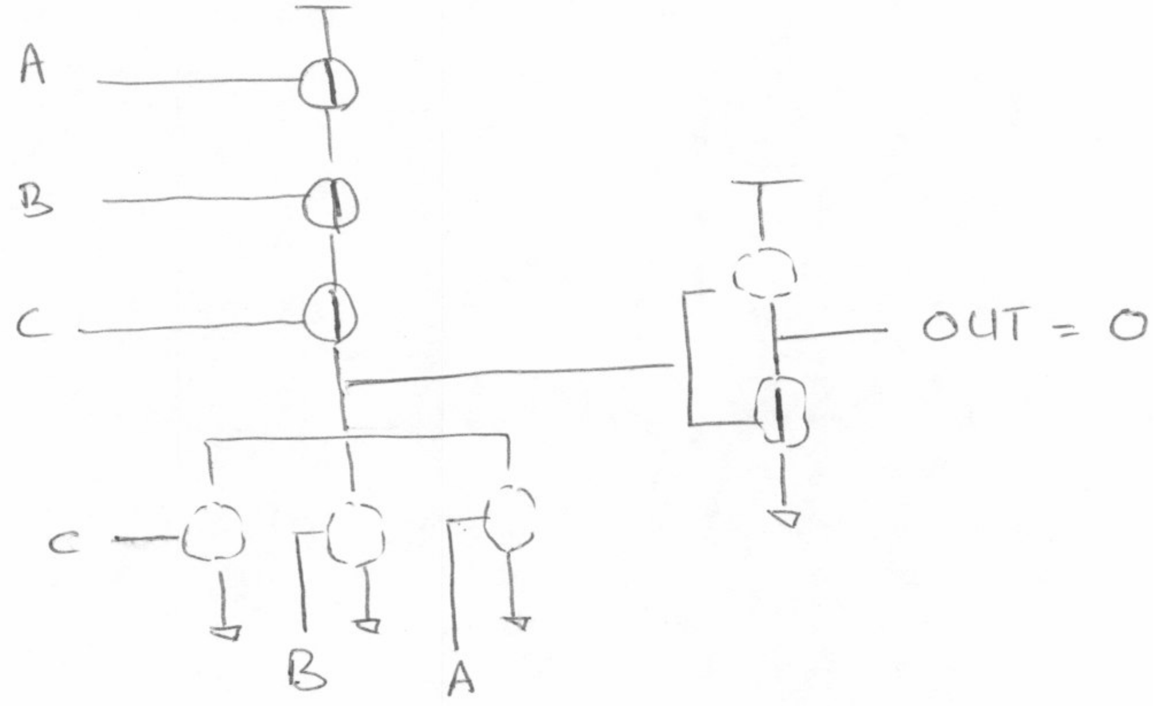


b. (2) A = 0, B = 0, C = 0

AND Gate

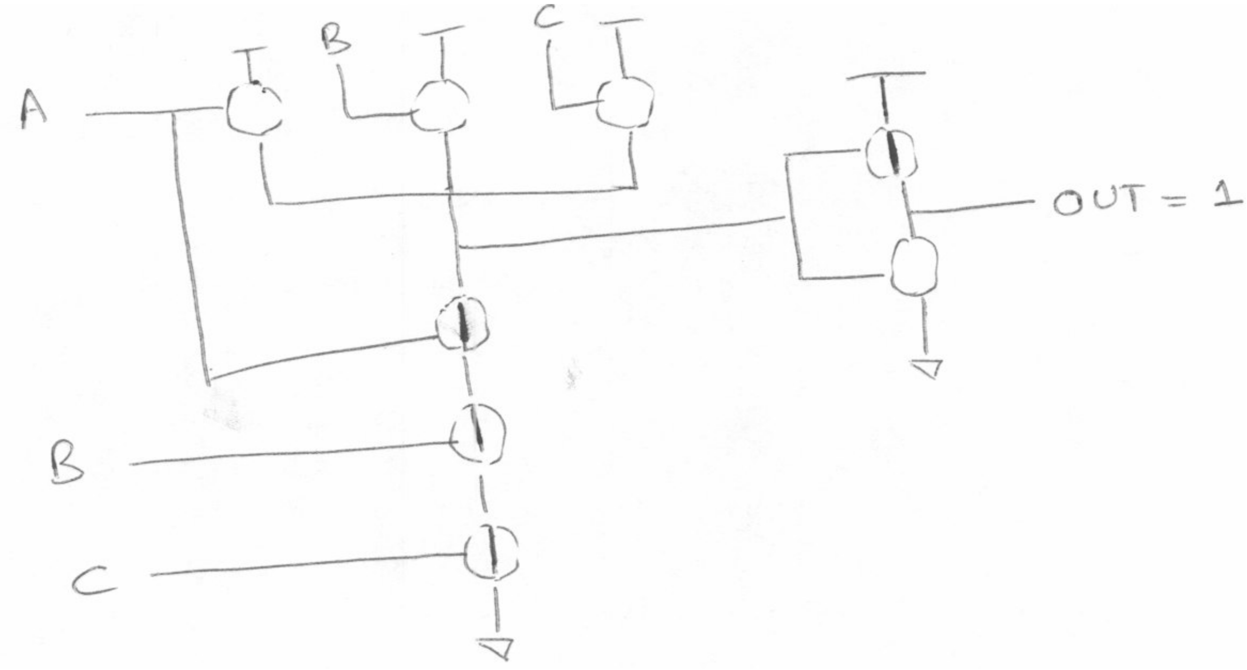


OR Gate

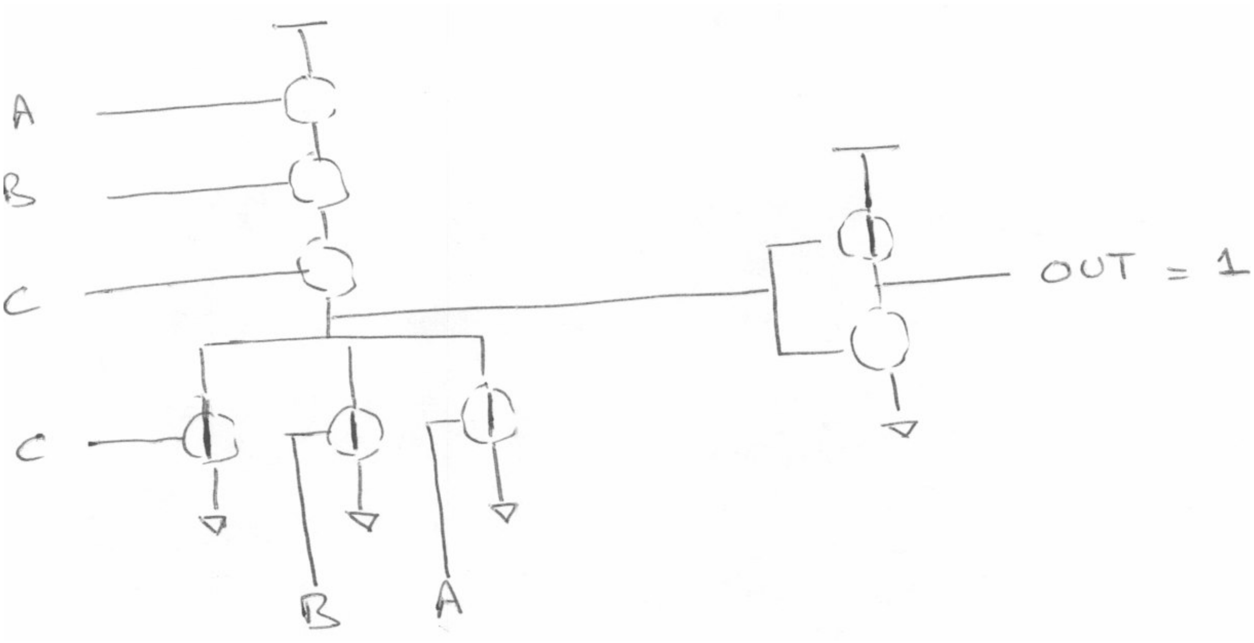


b. (3) A = 1, B = 1, C = 1

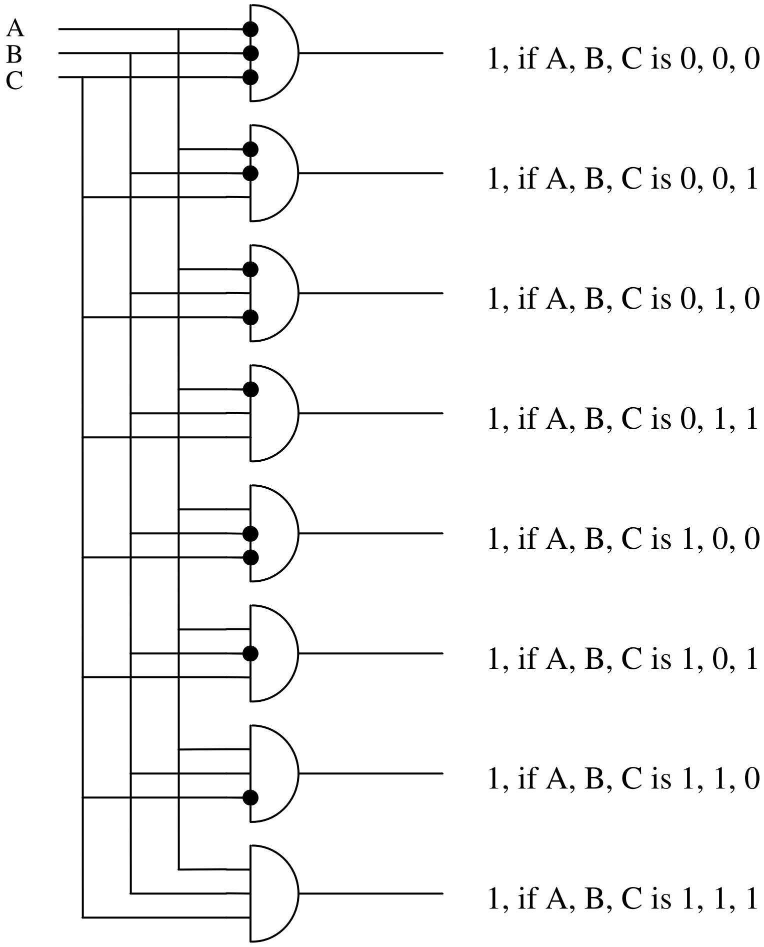
AND Gate



OR Gate



3.12



3.13 A five input decoder will have 32 output lines.

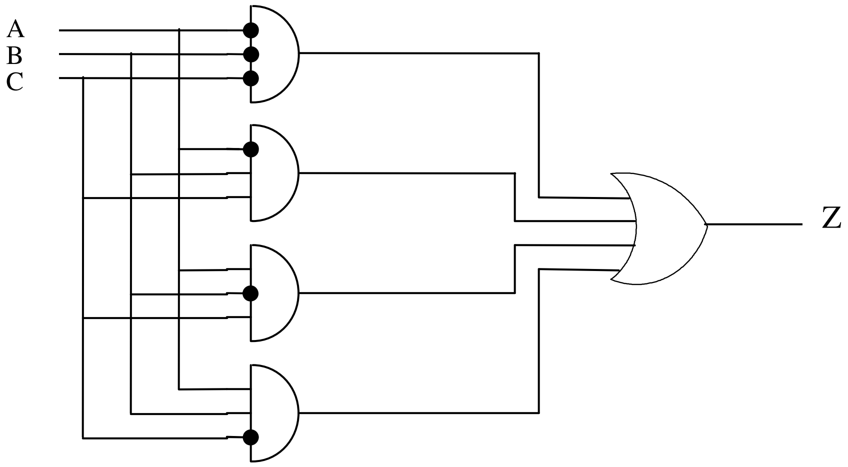
3.14 A 16 input multiplexer will have one output line (ofcourse!). It will have 4 select lines.

3.15

C_{in}	1	1	1	0
A	0	1	1	1
B	1	0	1	1
S	0	0	1	0
C_{out}	1	1	1	1

A = 7, B = 11, A + B = 18.
In the above calculation, the result (S) is 2 !! This is because 18 is too large a number to be represented in 4 bits. Hence there is an overflow - Cout[3] = 1.

3.16 Z = XNOR(A, B, C)

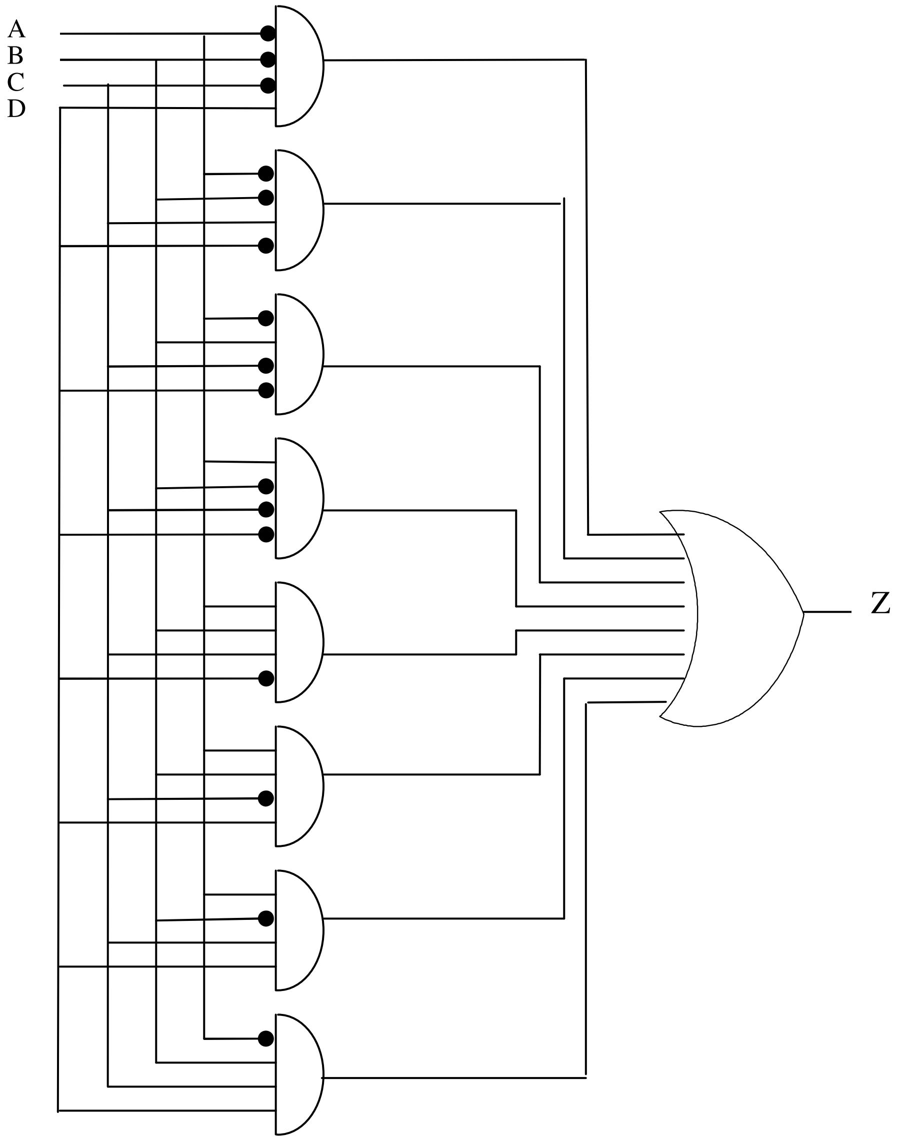


3.17 (a) The truth table will have 16 rows. Here is the truth table for Z = XOR (A, B, C, D). Any circuit with at least seven input combinations generating 1s at the output will work.

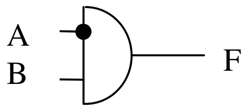
A	B	C	D		Z
0	0	0	0		0
0	0	0	1		1
0	0	1	0		1
0	0	1	1		0
0	1	0	0		1
0	1	0	1		0
0	1	1	0		0
0	1	1	1		1
1	0	0	0		1
1	0	0	1		0
1	0	1	0		0
1	0	1	1		1
1	1	0	0		0
1	1	0	1		1
1	1	1	0		1
1	1	1	1		0

Z = XOR (A,B,C,D)

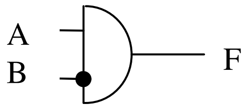
(b)



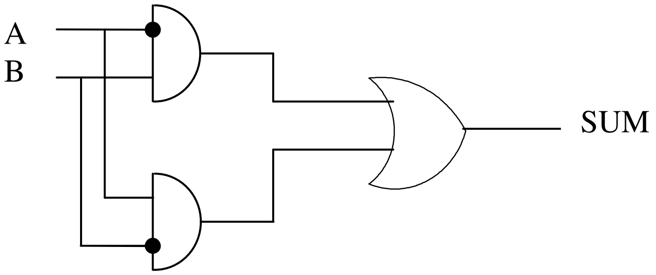
3.18. (a)



(b)



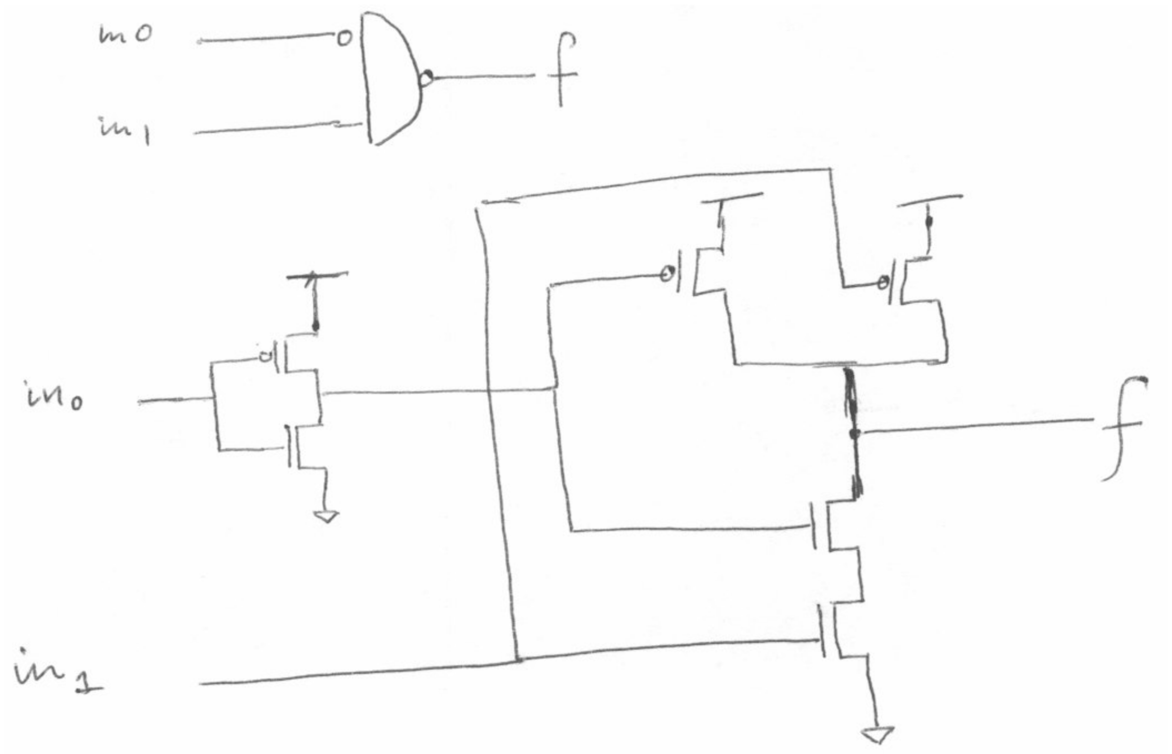
(c)



(d) No. The carry is not being generated/propagated.

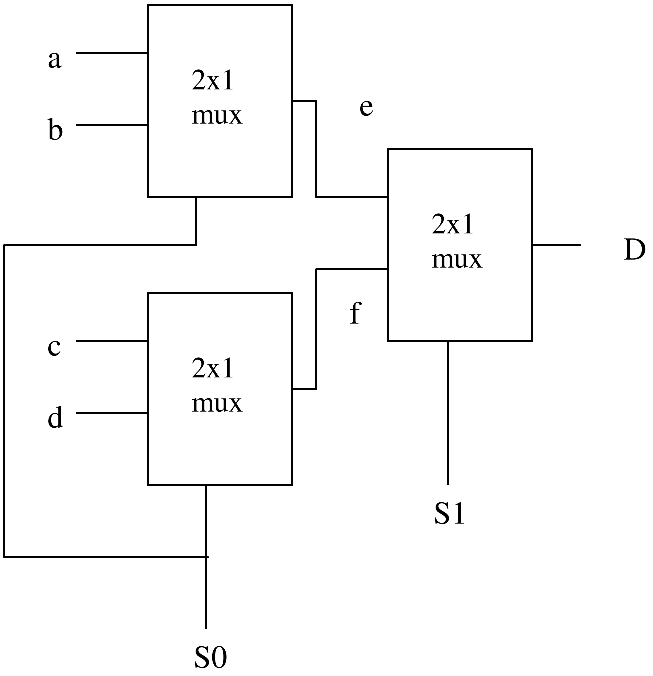
3.19 Figure 3.36 is a simple combinational circuit. The output value depends ONLY on the input values as they currently exist. Figure 3.37 is an R-S Latch. This is an example of a logic circuit that can store information. That is, if A, B are both 1, the value of D depends on which of the two (A or B) was 0 most recently.

3.20



3.21 $2 * 2^{14} = 2^{15} = 32768$ nibbles

3.22



S1	S0		e	f	D
0	0		a	c	a
0	1		b	d	b
1	0		a	c	c
1	1		b	d	d

3.23

A	B	C		Z
0	0	0		0
0	0	1		0
0	1	0		0
0	1	1		0
1	0	0		0
1	0	1		0
1	1	0		0
1	1	1		0

3.24 (a) X=0 => S = A+B, X=1 => S = A+C

3.24 (b) Circuit diagram is same as Figure 3.39 with the following modifications:
C = NOT (B), Carry-in = X

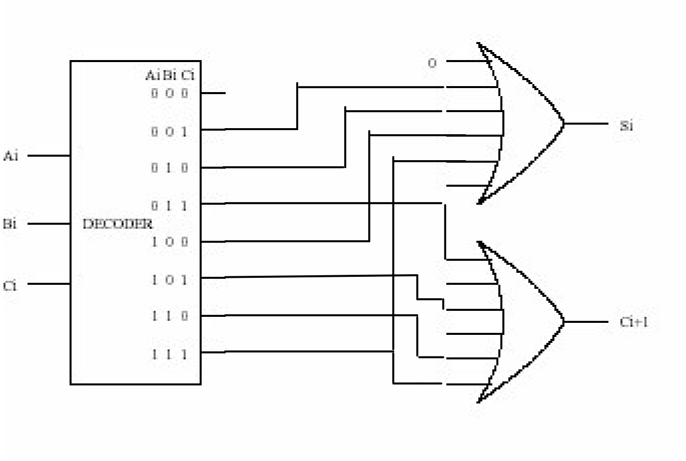
3.25 (a) 3 gate delays

3.25 (b) 3 gate delays

3.25 (c) 3*4 = 12 gate delays

3.25 (d) 3*32 = 96 gate delays

3.26



A	B	C		Si	Ci
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

3.27(a) When S=0, Z = A

3.27(b) When S=1, Z retains its previous value.

3.27(c) Yes; the circuit is a storage element.

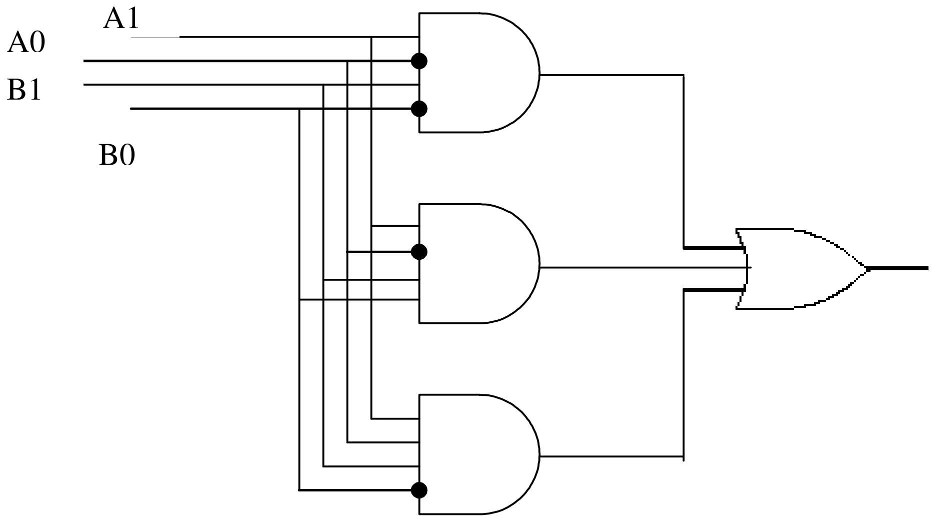
3.28

a) 3
b) 3
c) 9
d) 4

e)

A[1]	A[0]	B[1]	B[0]		Y[3]	Y[2]	Y[1]	Y[0]
0	1	0	1		0	0	0	1
0	1	1	0		0	0	1	0
0	1	1	1		0	0	1	1
1	0	0	1		0	0	1	0
1	0	1	0		0	1	0	0
1	0	1	1		0	1	1	0
1	1	0	1		0	0	1	1
1	1	1	0		0	1	1	0
1	1	1	1		1	0	0	1

f) $Y_2 = A_1.A_0'.B_1.B_0' + A_1.A_0'.B_1.B_0 + A_1.A_0.B_1.B_0'$



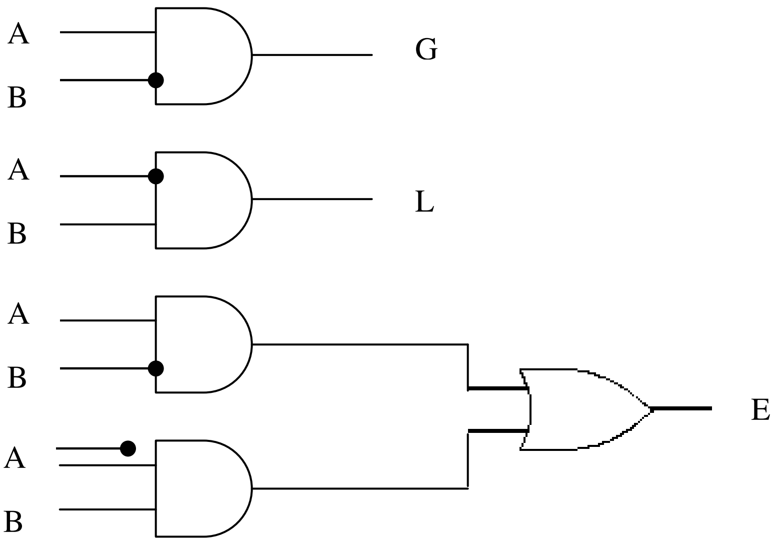
3.29 No. The original value cannot be recovered once a new value is written into a register.

3.30

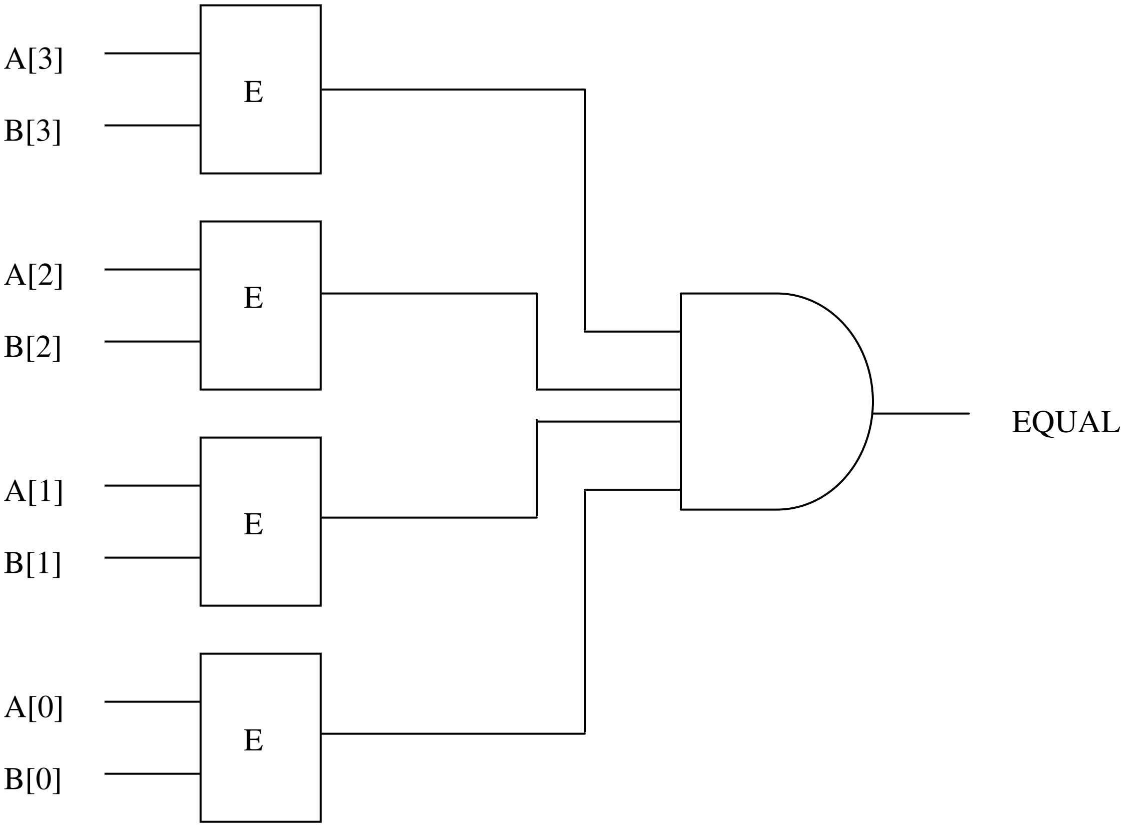
a)

A	B		G	E	L
0	0		0	1	0
0	1		0	0	1
1	0		1	0	0
1	1		0	1	0

b)



c)



3.31. 8 * (2^3) = 64 bytes

3.32 A memory address refers to a location in memory. Memory's addressability is the number of bits stored in each memory location.

3.33.(a) To read the 4th memory location, A[1,0] = 11, WE = 0

3.33.(b) A total of 6 address lines are required for a memory with 60 locations. The addressability of the memory will remain unchanged.

3.33.(c) A program counter of width 6 can address 2^6 = 64 locations. So without changing the width of the program counter, 64-60 = 4 more locations can be added to the memory.

3.34

a) 4 locations
b) 4 bits
c) 0001

3.35 Total bits of storage = 2^22 * 3 = 12582912

3.36 No effect, since it is a combinational logic circuit.

3.37 There are a total of four possible states in this lock. Any other state can be expressed as one of states A, B, C or D. For example, the state performed one correct followed by one incorrect operation is nothing but state A as the incorrect operation reset the lock.

3.38 Yes. We can have an arc from a state where the score is Texas 30, Oklahoma 28 to a state where the score is tied. This transition represents a Oklahoma player scoring a two-point shot.

3.39 No. An arc is needed between the two states.

(a) Game in Progress:

Texas *	Oklahoma
Fouls:4	Fouls: 4
73	68
First Half	
7:38	
Shot Clock : 14	

(b) Texas Win:

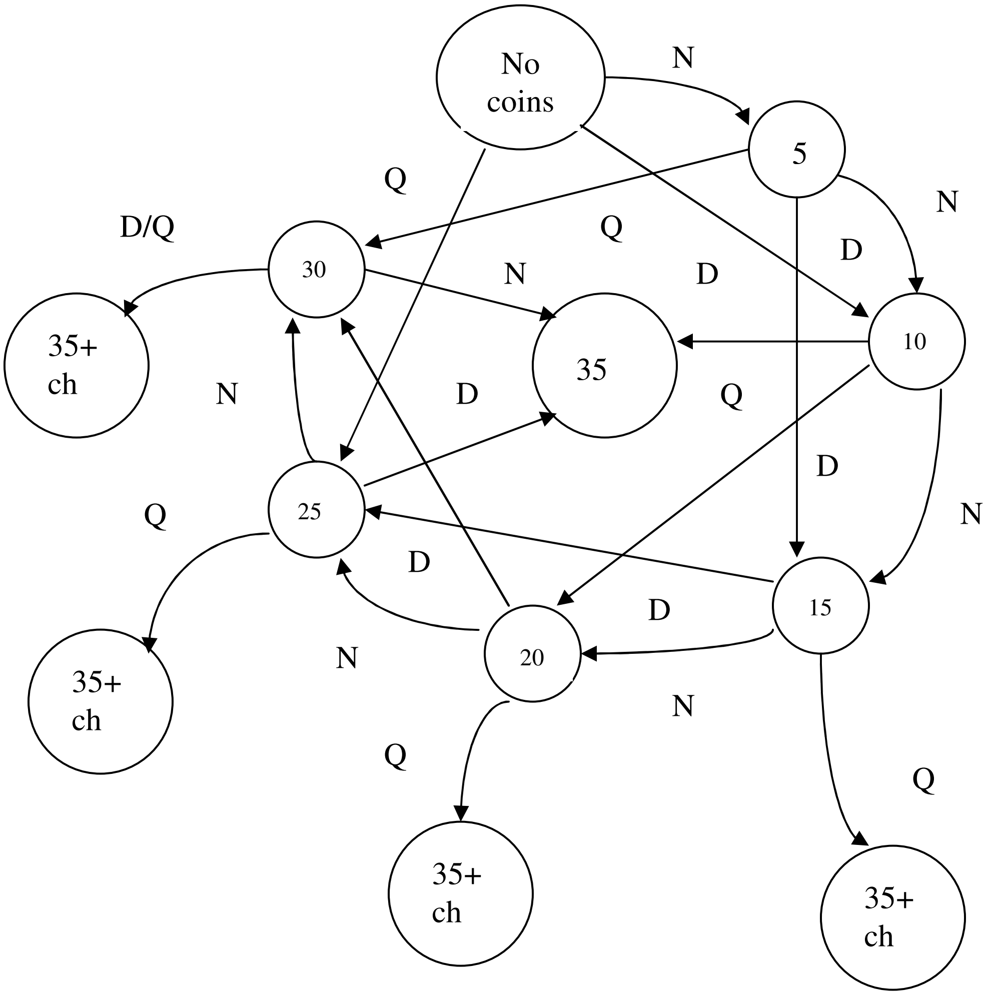
Texas *	Oklahoma
Fouls:10	Fouls: 10
85	70
Second Half	
0:00	
Shot Clock : 0	

(c) Oklahoma Win:

Texas *	Oklahoma
Fouls:10	Fouls: 10
81	90
First Half	
7:38	
Shot Clock : 0	

3.40 Left as an exercise. For each board state, come up with a transition to the best possible next state.

3.41



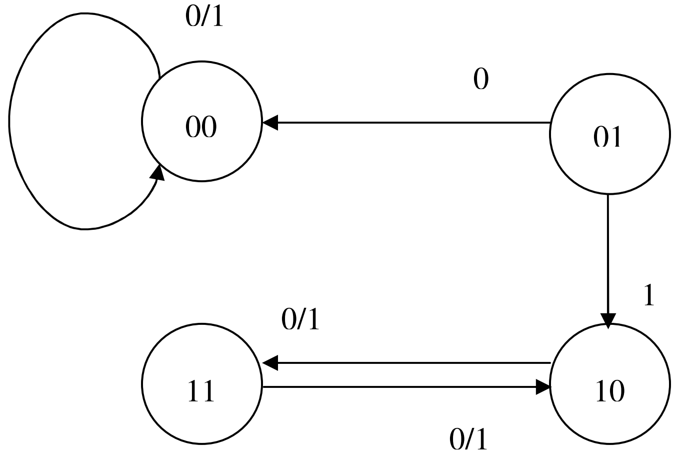
3.42 Since there are 3 states (states 01, 10 and 11) in which lights 1 and 2 are on, these lights are controlled by the output of the OR gate labeled U.

Storage element 2 should be set to 1 for the next clock cycle if the next state is 01 or 11.This is true when the current state is 00 or 10. So it is controlled by the output of the OR gate labeled U.

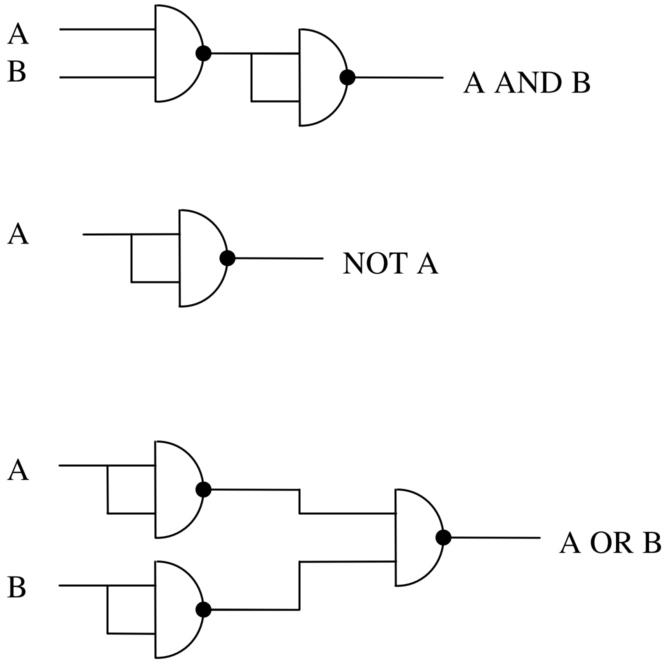
3.43
a)

S1	S0	X		D1	D0	Z
0	0	0		0	0	0
0	0	1		0	0	0
0	1	0		0	0	1
0	1	1		1	0	1
1	0	0		1	1	1
1	0	1		1	1	1
1	1	0		1	0	1
1	1	1		1	0	1

b)



3.44



F.4 Chapter 4 Solutions

4.1 Components of the Von Neumann Model:

(a) Memory: Storage of information (data/program)

(b) Processing Unit: Computation/Processing of Information

(c) Input: Means of getting information into the computer. e.g. keyboard, mouse

(d) Output: Means of getting information out of the computer. e.g. printer, monitor

(e) Control Unit: Makes sure that all the other parts perform their tasks correctly and at the correct time.

4.2 The communication between memory and processing unit consists of two registers: Memory Address Register (MAR) and Memory Data Register (MDR).

- To read, the address of the location is put in MAR and the memory is enabled for a read. The value is put in MDR by the memory.
- To write, the address of the location is put MAR, the data is put in MDR and the Write Enable signal is asserted. The value in MDR is written to the location specified.

4.3 The program counter does not maintain a count of any sort. The value stored in the program counter is the address of the next instruction to be processed. Hence the name 'Instruction Pointer'is more appropriate for it.

4.4 The size of the quantities normally processed by the ALU is referred to as the word length of the computer.

The word length does not affect what a computer can compute. A computer with a smaller word length can do the same computation as one with a larger word length; but it will take more time.

For example, to add two 64 bit numbers,

```
word length = 16 takes 4 adds.
word length = 32 takes 2 adds.
word length = 64 takes 1 add.
```

4.5 (a) Location 3 contains 0000 0000 0000 0000
Location 6 contains 1111 1110 1101 0011

(b) i. Two's Complement -
Location 0: 0001 1110 0100 0011 = 7747
Location 1: 1111 0000 0010 0101 = -4059

ii. ASCII - Location 4: 0000 0000 0110 0101 = 101 = 'e'

iii. Floating Point -
Locations 6 and 7: 0000 0110 1101 1001 1111 1110 1101 0011
Number represented is 1.10110011111111011010011 x 2^{-114}

iv. Unsigned -
Location 0: 0001 1110 0100 0011 = 7747
Location 1: 1111 0000 0010 0101 = 61477

(c) Instruction - Location 0: 0001 1110 0100 0011 = Add R7 R1 R3

(d) Memory Address - Location 5: 0000 0000 0000 0110 Refers to location 6. Value stored in location 6 is 1111 1110 1101 0011

4.6 The two components of and instruction are:

Opcode: Identifies what the instruction does.

Operands: Specifies the values on which the instruction operates.

4.7 60 opcodes = 6 bits
32 registers = 5 bits
So number of bits required for IMM = 32 - 6 - 5 - 5 = 16
Since IMM is a 2's complement value, its range is -2^{15} ... (2^{15} -1) = -32768 .. 32767.

4.8 a) 8-bits
b) 7-bits
c) Maximum number of unused bits = 3-bits

4.9 The second important operation performed during the FETCH phase is the loading of the address of the next instruction into the program counter.

4.10 Refer to the following table:

	Fetch		Evaluate	Fetch		Store
	Instruction	Decode	Address	Data	Execute	Result
PC	0001, 0110, 1100				1100	
IR	0001, 0110, 1100					
MAR	0001, 0110, 1100			0110		
MDR	0001, 0110, 1100			0110		

4.11 The phases of the instruction cycle are:

(a) Fetch: Get instruction from memory. Load address of next instruction in the Program Counter.

(b) Decode: Find out what the instruction does.

(c) Evaluate Address: Calculate address of the memory location that is needed to process the instruction.

(d) Fetch Operands: Get the source operands (either from memory or register file).

(e) Execute: Perform the execution of the instruction.

(f) Store Result: Store the result of the execution to the specified destination.

4.12 Considering the LC3 instruction formats

ADD

Fetch: Get instruction from memory. Load next address into PC. Decode: It is here that it is determined that the instruction is an add instruction. Evaluate Address: No memory operation so NOT REQUIRED. Fetch Operands: Get operands from register file. Execute: Perform the add operation. Store Result: Store result in the register file.

LDR

Fetch: Get instruction from memory. Load next address into PC. Decode: It is here that it is determined that the instruction is a Load Base+offset instruction. Evaluate Address: Calculate the memory address by adding the Base register with the sign extended offset.

Fetch Operands: Get value from the memory. Execute: No operation needed so NOT REQUIRED. Store Result: Store the value loaded into the register file.

JMP

Fetch: Get instruction from memory. Load next address into PC. Decode: It is here that it is determined that the instruction is a Jump instruction. Evaluate Address: No memory operation so NOT REQUIRED. Fetch Operands: Get the base register from register file. Execute: Store the value in PC. Store Result: NOT REQUIRED.

** Since we are considering a non pipelined implementation, the instruction phases where no operation is performed may not be present in its execution cycle.

4.13

		F	D	EA	FO	E	SR	
x86:	ADD [eax] edx	100	1	1	100	1	100	= 303
LC3:	ADD R6, R2, R6	100	1	-	1	1	1	= 104

4.14 JMP: 1100 0000 1100 0000

Fetch: Get instruction from memory. Load next address into PC.

Decode: It is here that it is determined that the instruction is JMP.

Evaluate Address: No memory operation, so NOT required.

Fetch Operands: Get the base register from the register file.

Execute: Load PC with the base register value, x369C.

4.15 Once the RUN latch is cleared, the clock stops, so no instructions can be processed. Thus, no instruction can be used to set the RUN latch. In order to re-initiate the instruction cycle, an external input must be applied. This can be in the form of an interrupt signal or a front panel switch, for example.

4.16 (a) $1/(2 * 10^{-9}) = 5 * 10^8$ machine cycles per second.

(b) $5 * 10^8/8 = 6.25 * 10^7$ instructions per second.

(c) It should be noted that once the first instruction reaches the last phase of the instruction, an instruction will be completed every cycle. So, except for this initial delay (known as latency), one instruction will be completed each machine cycle (assuming that there are no breaks in the sequential flow). If we ignore the latency, the number of instructions that will be executed each second is same as the number of machine cycles in a second = 5 * 10^8.

F.5 Chapter 5 Solutions

5.1 (a) ADD
- operate
- register addressing for destination and source 1
- register or immediate addressing for source 2

(b) JMP
- control
- register addressing

(c) LEA
- data movement
- immediate addressing

(d) NOT
- operate
- register addressing

5.2 The MDR is 64 bits. The statement does not tell anything about the size of the MAR.

5.3 Sentinel. It is a special element which is not part of the set of allowable inputs and indicates the end of data.

5.4 (a) 8 bits

(b) 6

(c) 6

5.5 (a) Addressing mode: mechanism for specifying where an operand is located.

(b) An instruction's operands are located as an immediate value, in a register, or in memory.

(c) The 5 are: immediate, register, direct memory address, indirect memory address, base + offset address. An immediate operand is located in the instruction. A register operand is located in a register (R0 - R7). A direct memory address, indirect memory address and base + offset address all refer to operands locate in memory.

(d) Add R2, R0, R1 => register addressing mode.

5.6 (a) 0101 011 010 1 00100
AND R3, R2, #4

(b) 0101 011 010 1 01100
AND R3, R2, #12

(c) 1001 011 010 111111
NOT R3,R2 if zero, no machine is busy.

(d) We cannot do this in only one instruction. We'd need to do an AND with 0000 0000 0100 0000, since the state of machine 6 is in bit [6:6]. This is impossible with the 5-bit immediate value. We could use a second instruction to load this value into a register, and then perform the AND.

5.7 01111 (decimal 15)

5.8 Increasing the number of registers to 32 will need 5 bits to denote the register number. Now, the minimum number of bits needed for the ADD instruction will be 4 (for the opcode) + 3 registers * 5 bits = 19 bits. This cannot fit in the 16-bits allocated for an lc-3 instruction.

5.9 (a) Add R1, R1, #0 => differs from a NOP in that it sets the CC's.

(b) BRnzp #1 => Unconditionally branches to one after the next address in the PC. Therefore no, this instruction is not the same as NOP.

(c) Branch that is never taken. Yes same as NOP.

5.10 A: BRnzp -171
B: JSR -171

Both A and B result in the PC being changed to (PC+1)-171.

However, B saves the linkage information in R7 and A does not affect R7.

5.11 No. We cannot do it in a single instruction as the smallest representable integer with the 5 bits available for the immediate field in the ADD instruction is -16. However this could be done in two instructions.

5.12 If R0[15] and R1[15] equal 0, but R2[15] equals 1, the result can be trusted. The other case causes an overflow, this one doesnt.

5.13 (a) 0001 011 010 1 00000 (ADD R3, R2, #0)

(b)
```
1001 011 011 111111  (NOT R3, R3 )
0001 011 011 1 00001 (ADD R3, R3, #1 )
0001 001 010 0 00011 (ADD R1, R2, R3 )
```

(c) 0001 001 001 1 00000 (ADD R1, R1, #0)
or
0101 001 001 1 11111 (AND R1, R1, #-1)

(d) Can't happen. The condition where N=1, Z=1 and P=0 would require the contents of a register to be both negative and zero.

(e) 0101 010 010 1 00000 (AND R2, R2, #0)

5.14 (2) : 1001 101 010 111111
(4) : 1001 011 110 111111

5.15
```
1110 001 000100000  ( LEA R1, 0x20    )  R1 <- 0x3121
0010 010 000100000  ( LD R2,  0x20    )  R2 <- Mem[0x3122] = 0x4566
1010 011 000100001  ( LDI R3, 0x20    )  R3 <- Mem[Mem[0x3123]] = 0xabcd
0110 100 010 000001 ( LDR R4, R2, 0x1 )  R4 <- Mem[R2 + 0x1] = 0xabcd
1111 0000 0010 0101 ( TRAp 0x25       )
```

5.16 (a) PC-relative mode

(b) Indirect Mode

(c) Base+offset mode

5.17 (a) LD: two, once to fetch the instruction, once to fetch the data.

(b) LDI: three, once to fetch the instruction, once to fetch the data address, and once to fetch the data.

(c) LEA: once, only to fetch the instruction.

5.18 LDR - 2 memory accesses
STI - 3 memory accesses
TRAP - 2 memory accesses

5.19 PC-64 to PC+63. The PC value used here is the incremented PC value.

5.20 7 bits will be required for the PC-relative offset.

5.21 The Trap instruction provides 8 bits for a trap vector. That means there could be 2^8 = 256 trap routines.

5.22 Please correct the LC-3 instructions to read:

```
x3010          1110 0110 0011 1111
x3011          0110 1000 1100 0000
x3012          0110 1101 0000 0000
```

These instructions cause the value x123B to be loaded into R6.

These three instructions could be replaced by the following instruction at location x3010:

```
x3010          1010 1100 0011 1111
```

5.23

```
x30ff 1110 0010 0000 0001  (LEA R1, #1) R1 <- 0x3101
x3100 0110 010 001 00 0010 (LDR R2, R1, #2) R2 <- 0x1482
x3101 1111 0000 0010 0101  (TRAP 0x25)
x3102 0001 0100 0100 0001
x3103 0001 0100 1000 0010
```

5.24 The largest address this instruction can load from is x4011 + x1F = x4030. This is specified by the following instruction: LDR R5, R4, x1F.

If the LDR offset were zero-extended, the largest address the LDR instruction can load from is x4011 + x3F = 0x4050 and the smallest address is x4011.

5.25

```
1001 100 011 111111  ;(NOT R4, R3)
0001 100 100 1 00001 ;(ADD R4, R4, #1)
0001 001 100 0 00 010 ;(ADD R1, R4, R2)
0000 010 000000101   ; (BRz Done)
0000 100 000000001   ; (BRn Reg3)
0000 001 000000010   ; (BRp Reg2)
0001 001 011 1 00000 ; (Reg3 ADD R1, R3, #0)
```

```
0000 111 000000001   ; (BRnzp Done)
0101 001 010 1 00000 ; (Reg2 ADD R1, R2, #0)
1111 0000 0010 0101  ; (Done TRAP 0x25)
```

5.26 NOTE: Please refer to the errata for the new problem statement.

(a) Yes, there is a problem. With 5 bits of opcode, and 4 bits each for DR and SR1 registers, and one steering bit to differentiate immediate vs. a second source register, there are only 2 bits left for the second source register. Consequently, ADD and AND can not be specified as in the original LC-3.

One possible way around this is to use the DR field to specify both the destination and one of the source registers, as are done in many ISAs, the x86, for example. If we adopted this approach (called a 2-address machine), we could specify the ADD and AND format as:

```
 ----------------------------------------------
|  ′  ′  ′  ′  |  ′  ′  ′  | |  ′  ′  ′  ′  ′  |
|    opcode        DR      I   imm or SR      |
|______________________________________________|
```

(b) 64K = 2^{16} means 16 bits per address

(c) Since we shift the trap vector left 5 bits before zero-extending, a minimum of 32 bytes are allocated to each trap routine. Since we wish to accommodate 2^7 trap routines, a minimum of 4KB of memory is required.

(d) Look at the Add instruction to discover the size of the immediate:

```
[opcode] [destreg] [srcreg] [steeringbit] [imm]
   4b        2b       2b          1b          ?b
```

This leaves 16 - 4 - 2 - 2 - 1 = 7 bits for the immediate. The largest immediate value then is 2^6 - 1 = 63.

5.27 Four different values: xAAAA, x30F4, x0000, x0005

5.28 Instructions at x3002 and x3003 below replace the store indirect instruction in the original code. Note that as a new instruction had to be inserted, the offsets increased by 1.

```
x3000 0010 0000 0000 0011
x3001 0010 0010 0000 0011
x3002 0111 0000 0100 0000
x3003 1111 0000 0010 0101
x3004 0000 0000 0100 1000
x3005 1111 0011 1111 1111
```

5.29 (a) LDR R2, R1, #0 ;load R2 with contents of location pointed to by R1
STR R2, R0, #0 ;store those contents into location pointed to by R0

(b) The constituent micro-ops are:
MAR $< -$ SR
MDR $< -$ Mem[MAR]
MAR $< -$ DR
Mem[MAR] $< -$ MDR

5.30 If the conditional branch is taken, it is known that R0 and R1 must contain the same value.

5.31 0x1000: 0001 101 000 1 11000

5.32 Please correct the LC-3 instructions to read:

```
x3050          0000 0010 0000 0010
x3051          0101 0000 0010 0000
x3052          0000 1110 0000 0010
x3053          0101 0000 0010 0000
x3054          0001 0000 0011 1111
```

The resulting condition codes are N=1, Z=0, P=0.

5.33 It can be inferred that R5 has exactly 5 of the lower 8 bits = 1.

5.34 The Reg File and the ALU in Fig 5.18 implement the NOT instruction, alongwith NZP and the logic which goes with it.

5.35 The IR, SEXT unit, SR2MUX, Reg File and ALU implement the ADD instruction, alongwith NZP and the logic which goes with it.

5.36 Memory, MDR, MAR, IR, PC, Reg File, the SEXT unit connected to IR[8:0], ADDR2MUX, ADDR1MUX set to PC, alongwith the ADDER they connect to, and MAXMUX and GateMARMUX implement the LD instruction, alongwith NZP and the logic which goes with it.

5.37 Memory, MDR, MAR, IR, PC, Reg File, the SEXT unit connected to IR[8:0], ADDR2MUX, ADDR1MUX set to PC, alongwith the ADDER they connect to, and MAXMUX and GateMARMUX implement the LDI instruction, alongwith NZP and the logic which goes with it.

5.38 Memory, MDR, MAR, IR, Reg File, the SEXT unit connected to IR[5:0], ADDR2MUX is set to 0, ADDR1MUX is set to SR1 out, alongwith the ADDER they connect to, and MAXMUX and GateMARMUX implement the LDR instruction, alongwith NZP and the logic which goes with it.

5.39 IR, PC, Reg File, the SEXT unit connected to IR[8:0], ADDR2MUX, ADDR1MUX set to PC, alongwith the ADDER they connect to, and MAXMUX and GateMARMUX implement the LEA instruction, alongwith NZP and the logic which goes with it.

5.40 The signal A indicates if the instruction in IR is a taken branch instruction.

5.41 (a) Y is the P Condition code.

(b) Yes. The error is that the logic should not have the logic gate A. X should be one whever the opcode field of the IR matches the opcodes which change the condition code registers. The problem is that X is 1 for the BR opcode (0000) in the given logic.

5.42 (d) MUL is the most useful instruction out of the choices given. All the others can be accomplished using it or existing instructions.

(a) MOVE can be accomplished using either the ADD or AND instructions.
(b) NAND can be accomplished by doing an AND followed by a NOT.
(c) SHFL can be accomplished by using MUL.

F.6 Chapter 6 Solutions

6.1 Yes, for example, an iterative block where the test condition remains true for each iteration. This procedure will never end and is therefore not finite and not an algorithm. The following is an example of a procedure that isn't an algorithm:

```
x3000 0101 000 000 1 00000 ( LOOP AND  R0, R0, #0 )
x3001 0000 010 111111110   (      BRz LOOP )
```

This is not an algorithm because the branch instruction is always taken and the program loops indefinitely.

6.2 Problem Statement: Subtract the value in R1 from the value in R2 and store the result in R0.

The following is the systematic decomposition of the problem concluding with an LC-3 program solving the problem.

(a)
```
Start -> Subtract R1 from R2; place result in R0
-> End
```

(b)
```
Start -> Compute the complement of the value in R1
-> Add this result to R2 and store the result in R0
-> End
```

(c)
```
Start -> Negate R1 -> Add 1 to R1 ->
Add R1 and R2; store result in R0 -> End
```

(d)
```
1001 001 001 111111 ( NOT R1, R1 )
0001 001 001 1 00001 ( ADD R1, R1, #1 )
0001 000 001 0 00 010 ( ADD R0, R1, R2 )
```

6.3 The following program uses DeMorgan's Law to set the appropriate bits of the machine busy register.

```
x3000 1010000000001110 (   LDI R0, S )
x3001 1010001000001110 (   LDI R1, I )
x3002 0101010010100000 (   AND R2, R2, #0 )
x3003 0001010010100001 (   ADD R2, R2, #1 )
x3004 0001001001111111 ( L ADD R1, R1, #-1 )
x3005 0000100000000010 (   BRn D )
x3006 0001010010000010 (   ADD R2, R2, R2 )
x3007 0000111111111100 (   BRnzp L )
x3008 0001001010100000 ( D ADD R1, R2, #0 )
x3009 1001000000111111 (   NOT R0, R0 )
x300a 1001001001111111 (   NOT R1, R1 )
x300b 0101000000000001 (   AND R0, R0, R1 )
x300c 1001000000111111 (   NOT R0, R0 )
x300d 1011000000000001 (   STI R0, S  )
```

```
x300e 1111000000100101 (   TRAP x25 )
x300e 0100000000000001 ( S .FILL x4001 )
x300f 0100000000000000 ( I .FILL x4000 )
```

```
6.4 x3000 0101000000100000 (     AND R0, R0, #0 )
    x3001 1001011001111111 (     NOT R3, R1     )
    x3002 0001011011100001 (     ADD R3, R3, #1 )
    x3003 0001011011000010 (     ADD R3, R3, R2 )
    x3004 0000010000000100 (     BRz done       )
    x3005 0000100000000010 (     BRn neg        )
    x3006 0001000000111111 (     ADD R0, R0, #-1)
    x3007 0000111000000001 (     BRnzp done     )
    x3008 0001000000100001 ( neg ADD R0, R0, #1 )
    x3009 1111000000100101 (done halt           )
```

6.5 The three additions of 88 + 88 + 88 requires fewer steps to complete than the eighty eight additions of 3 + 3 + ... + 3. Because 88 + 88 + 88 requires fewer instructions to complete, it is faster and therefore preferable.

6.6 Problem Statement: Multiply two numbers together using repeated addition. Make the procedure efficient by comparing the two numbers per the result of Problem 6.4.

```
Start -> Multiply two numbers; put result in R3 -> End
Start -> Compare two numbers -> Multiply two numbers
by repeated addition;
Put result in R3 -> End
Start -> Compare two numbers -> Put smaller number
in R2, larger number in R1 -> Add R1 to itself while
decrementing R2 -> Store result of repeated addition
to R0 -> End
```

The program compares the numbers located in memory locations x3010 and x3011, places the larger of the two in R1 and the smaller in R2, then repeatedly adds R1 to itself R2 times. The program stores the result in register R3.

```
x3000 0010110000010000 ( LD R6, A )
x3001 0010111000010000 ( LD R7, B )
x3002 1001101110111111 ( NOT R5, R6 )
x3003 0001101101100001 ( ADD R5, R5, #1 )
x3004 0001101111000101 ( ADD R5, R7, R5 )
x3005 0000110000000011 ( BRnz ABIG )
x3006 0001001111100000 ( ADD R1, R7, #0 )
x3007 0001010110100000 ( ADD R2, R6, #0 )
x3008 0000111000000010 ( BRnzp NEXT )
x3009 0001001110100000 ( ABIG ADD R1, R6, #0 )
x300a 0001010111100000 ( ADD R2, R7, #0 )
```

```
x300b 0101000000100000 ( NEXT AND R0, R0, #0 )
x300c 0001000000000001 ( LOOP ADD R0, R0, R1 )
x300d 0001010010111111 ( ADD R2, R2, #-1 )
x300e 0000001111111101 ( BRp LOOP )
x300f 0001011000100000 ( ADD R3, R0, #0)
x3010 1111000000100101 ( TRAP x25 )
x3011 0000000001011000 ( A .FILL #88 )
x3012 0000000000000011 ( B .FILL #3 )
```

6.7 This program adds together the corresponding elements of two lists of numbers (vector addition). One list starts at address x300e and the other starts at address x3013. The program finds the length of the two lists at memory location x3018. The first element of the first list is added to the first element of the second list and the result is stored back to the first element of the first list; the second element of the first list is added to the second element of the second list and the result is stored back to the second element of the first list; and so on.

6.8 Were R2 not initially cleared, a specious count would be computed by the program because R2 would initially contain garbage. After running, R2 would equal the difference between the number of occurances and whatever random initial value R2 happened to contain.

6.9

```
x3100 0010 000 0 0000 0101 (   LD R0, Z    )
x3101 0010 001 0 0000 0101 (   LD R1, C    )
x3102 1111 0000 0010 0001  ( L TRAP x21    )
x3103 0001 001 001 1 11111 (   ADD R1, R1, #-1 )
x3104 0000 001 1 1111 1101 (   BRp L       )
x3105 1111 0000 0010 0101  (   TRAP x25    )
x3106 0000 0000 0101 1010  ( Z .FILL x5A   )
x3107 0000 0000 0110 0100  ( C .FILL #100 )
```

6.10 This program tests whether R2 is even or odd. This program branches to x3110 if the number in R2 is even and branches to x3120 if the number in R2 is odd.

```
x3100 0101 000 010 1 00001 ( AND R0, R2, #1 )
x3101 0000 010 0 0000 1110 ( BRz Even )
x3102 0000 101 0 0001 1101 ( BRnp Odd )
```

6.11 This program increments each number in the list of numbers starting at address A and ending at address B. The program tells when it's done by comparing the last address it loaded data from with the address B. When the two addresses are equal, the program stops incrementing data values.

```
x3000 0010 000 011111111    (      LD R0, x3100 )
x3001 0010 001 011111111    (      LD R1, x3101 )
x3002 0001 001 001 1 00001  (      ADD R1, R1, #1 )
x3003 1001 001 001 111111   (      NOT R1, R1 )
x3004 0001 001 001 1 00001  (      ADD R1, R1 #1 )
```

```
x3005 0001 011 000 0 00 001 ( l    ADD R3, R0, R1 )
x3006 0000 010 000000101    (      BRz Done )
x3007 0110 010 000 000000   (      LDR R2, R0, #0 )
x3008 0001 010010100001     (      ADD R2, R2, #1 )
x3009 0111 010000000000     (      STR R2, R0, #0 )
x300a 0001 000000100001     (      ADD R0, R0, #1 )
x300b 0000 111111111001     (      BRnzp l )
x300c 1111 000000100101     ( Done HALT )
```

6.12 (a)

```
x3100 1111 0000 00100011 ( IN )
x3101 1111 0000 00100001 ( OUT )
x3102 0000 111 111111101 ( BRnzp x3100 )
```

(b)

```
x3000 0010 001 00001 0001  ( LD r1, nl )
x3001 1001 001 001 111111  ( NOT r1, r1 )
x3002 0001 001 001 1 00001 ( ADD r1, r1, #1 )
x3003 0010 010 0 0000 1111 ( LD r2, strg )
x3004 1111 0000 00100011   (iput IN )
x3005 0111 000 010 000000  ( STR r0, r2, #0 )
x3006 0001 011 001 0 00 000( ADD r3, r1, r0 )
x3007 0000 010 0 0000 0010 ( BRz oput )
x3008 0001 010 010 1 00001 ( ADD r2, r2, #1 )
x3009 0000 111 1 1111 1010 ( BRnzp iput )
x300a 0010 010 0 0000 1000 (oput LD r2, strg )
x300b 0110 000 010 000000  (oputl LDR r0, r2, #0 )
x300c 1111 0000 00100001   ( OUT )
x300d 0001 011 001 0 00 000  ( ADD r3, r1, r0 )
x300e 0000 010 0 0000 0010   ( BRz done )
x300f 0001 010 010 1 00001   ( ADD r2, r2, #1 )
x3010 0000 111 1 1111 1010   ( BRnzp oputl )
x3011 1111 0000 00100101     (done HALT )
x3012 0000 0000 0000 1010    (nl .FILL x0A )
x3013 0100 0000 0000 0000    (strg .FILL x4000 )
```

6.13 Memory location x3011 holds the number to be right shifted. The strategy here is to implement a one bit right shift by shifting to the left 15 bits. The most significant bit must be carried back to the least significant bit when it's shifted out (a circular left shift). The data to be shifted is stored at x3013. R1 is a counter to keep track of how many left shifts remain to be done.

```
x3000 0010000000010010 (      LD R0, NUM )
x3001 0101001001100000 (      AND R1, R1, #0 )
x3002 0001001001101111 (      ADD R1, R1, #15 )
x3003 0001000000100000 ( LOOP ADD R0, R0, #0 )
x3004 0000100000000001 (      BRn NEG )
x3005 0000001000000101 (      BRp POS )
```

```
x3006 0001000000000000 ( NEG  ADD R0, R0, R0 )
x3007 0001000000100001 (      ADD R0, R0, #1 )
x3008 0001001001111111 (      ADD R1, R1, #-1 )
x3009 0000110000000101 (      BRnz DONE )
x300a 0000111111111000 (      BRnzp LOOP )
x300b 0001000000000000 ( POS  ADD R0, R0, R0 )
x300c 0001001001111111 (      ADD R1, R1, #-1 )
x300d 0000110000000001 (      BRnz DONE )
x300e 0000111111110100 (      BRnzp LOOP )
x300f 0010001000000100 (DONE  LD R1, MASK )
x3010 0101000000000001 (      AND R0, R0, R1 )
x3011 0011000000000001 (      ST R0, NUM )
x3012 1111000000100101 (      HALT )
x3013 1000010000100001 ( NUM  .FILL x8421 )
x3014 0111111111111111 ( MASK .FILL x7FFF )
```

```
6.14 0x3000 0101 010 010 1 00000 ( AND R2, R2, #0 )
     0x3001 0001 001 001 1 11111 ( ADD R1, R1, #-1 )
     0x3002 0001 001 001 1 11111 ( ADD R1, R1, #-1 )
     0x3003 0001 001 001 1 11111 ( ADD R1, R1, #-1 )
     0x3004 0000 100 000000010 ( BRn x3007 )
     0x3005 0001 010 010 1 00001 ( ADD R2, R2, #1 )
     0x3006 0000 111 111111010 ( BRnzp x3001 )
     0x3007 1111 0000 00100101 ( TRAP 0x25 )
```

The possible values for R1 are: 9, 10, and 11.

```
6.15 0111 010 100 000111 ( STR R2, R4, #7 )
```

```
6.16 x3000 0010 000 0 0000 0100 ( LD R0, #4 )
     x3002 1011 000 0 0000 0011 ( STI R0, #3 )
     x3003 1111 0000 0010 0001  ( TRAP x21 )
     x3006 0100 0000 0000 0001
```

```
6.17 JSR #15
```

```
6.18 x3000 0101 0000 0010 0000 (       ADD R0, R0, #0 )
     x3001 1010 0010 0000 1011 (       LDI R1, PDIVIDEND )
     x3002 1010 0100 0000 1011 (       LDI r2, PDIVISOR  )
     x3003 1001 0110 1011 1111 (       NOT R3, R2 )
     x3004 0001 0110 1110 0001 (       ADD R3, R3, #1 )
     x3005 0001 0010 0100 0011 ( LOOP  ADD R1, R1, R3 )
     x3006 0000 1000 0000 0010 (       BRn REMN )
     x3007 0001 0000 0010 0001 (       ADD R0, R0, #1 )
     x3008 0000 1111 1111 1100 (   BRnzp LOOP )
     x3009 0001 0010 0100 0010 ( REMN  ADD R1, R1, R2
     x300a 1011 0000 0000 0100 (       STI R0, PQUO )
```

```
x300b 1011 0010 0000 0100 (       STI R1, PREM )
x300c 1111 0000 0010 0101 (       HALT )
x300d 0100 0000 0000 0000 ( PDIVIDEND .FILL x4000)
x300e 0100 0000 0000 0001 ( PDIVISOR  .FILL x4001)
x300f 0101 0000 0000 0000 ( PQUO  .FILL x5000)
x3010 0101 0000 0000 0001 ( PREM  .FILL x5001)
```

6.19 The bugs are:

1. The instruction at x3000 should be 0010 0000 0000 1010
2. The instruction at x3004 should be 0001 0100 1010 0100
3. The instruction at x3008 should be 0000 1111 1111 1001
4. The instruction at x3009 should be 0111 0100 0100 0000

6.20 Dividing signed numbers is more complicated than dividing positive numbers. The following rule is followed :
Dividend = Quotient x Divisor + Remainder

The sign of the quotient is set to MINUS if the divisor and the dividend are of different signs.The sign of the remainder is set according to the sign of the dividend.

```
x3000 0101 0000 0010 0000 (     and r0, r0, #0 ; quotient)
x3001 0101 1001 0010 0000 (     and r4, r4, #0 )
x3002 0101 1011 0110 0000 (     and r5, r5, #0 )
x3003 1010 0010 0001 1110 (     ldi r1, pnuma )
x3004 0000 1000 0000 0001 ( brn aneg )
x3005 0000 0110 0000 0100 (  brzp bld )
x3006 0001 1001 0010 0001 (aneg add r4, r4, #1 )
x3007 0001 1011 0110 0001 ( add r5, r5, #1 )
x3008 1001 0010 0111 1111 ( not r1, r1 )
x3009 0001 0010 0110 0001 (     add r1, r1, #1 )
x300a 1010 0100 0001 1000 ( bld ldi r2, pnumb )
x300b 0000 1000 0000 0001 ( brn bneg )
x300c 0000 0110 0000 0011 ( brzp cont )
x300d 0001 1001 0010 0001 (bneg add r4, r4, #1)
x300e 1001 0100 1011 1111 ( not r2, r2 )
x300f 0001 0100 1010 0001 ( add r2, r2, #1 )
x3010 1001 0110 1011 1111 ( cont not r3, r2 )
x3011 0001 0110 1110 0001 ( add r3, r3, #1 )
x3012 0001 0010 0100 0011 (loop add r1, r1, r3 )
x3013 0000 1000 0000 0010 ( brn remn )
x3014 0001 0000 0010 0001 (  add r0, r0, #1 )
x3015 0000 1111 1111 1100 (  brnzp loop )
x3016 0001 0010 0100 0010 (remn add r1, r1, r2 )
x3017 0101 1001 0010 0001 (done and r4, r4, #1 )
```

```
x3018 0000 0100 0000 0010 (  brz checkremsign )
x3019 1001 0000 0011 1111 (  not r0, r0 )
x301a 0001 0000 0010 0001 ( add r0, r0, #1 )
x301b 0101 1011 0110 0001 ( checkremsign and r5, r5, #1 )
x301c 0000 0100 0000 0010 (  brz storeres )
x301d 1001 0010 0111 1111 ( not r1, r1 )
x301e 0001 0010 0110 0001 ( add r1, r1, #1 )
x301f 1011 0000 0000 0100 (storeres sti r0, pquo )
x3020 1011 0010 0000 0100 (   sti r1, prem )
x3021 1111 0000 0010 0101 (  halt )
x3022 0100 0000 0000 0000 ( pnuma   .fill x4000 )
x3023 0100 0000 0000 0001 ( pnumb   .fill x4001 )
x3024 0101 0000 0000 0000 ( pquo    .fill x5000 )
x3025 0101 0000 0000 0001 ( prem    .fill x5001 )
```

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.

mem[RESULT] = mem[M0] * 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.

F.8 Chapter 8

8.1 (a) A device register is a register (or memory location) that is used for data transfer to/from an input/output device. It provides a means of communication between the processor and the input/output device. The processor can poll this register to find out whether it has received an input or it can send an output from/to the specific device that the device register belongs to. In memory mapped I/O device registers are dedicated memory locations for each I/O device. There may be more than one device register (dedicated memory location) for one device.

(b) A device data register is a device register (a dedicated memory location in memory-mapped I/O) that holds the data that is to be input/output.

(c) A device status register is a device register (a dedicated memory location in memory-mapped I/O) that indicates the status of the input/output. It allows for the processor to know whether or not input/output of the value in the device data register has occurred. Basically it is an important step to achieve synchronization in an asynchronous I/O system.

8.2 A ready bit is not needed if synchronous I/O is used because the processor will know exactly when the data will arrive and when it will be taken away (input and output). It will do input and/or output at regular intervals, and it will be guaranteed that during those intervals the input data is taken by the computer and the output data goes to the output device.

8.3 The processor can accept a character every clock cycle at its maximum rate. This means that a 300 MHz processor can accept a character each 1/(300M) seconds.That is this processor can accept a character every 3.333 nanoseconds which corresponds to a rate of 18 billion characters per one minute. This is the maximum rate it can accept input in one minute. If the typist would have to type 3 billion words per minute to synchronize with this maximum rate, then a word must be 18/3 = 6 characters long. (This, of course, counts the *space* between words as one of the characters in the word!)

8.4 (a) The interaction between a remote control and a television is synchronous. The television samples at specific intervals to see if a key on the remote control has been pressed. No synchronization is needed in this transaction.

(b) The interaction between the mail delivery person and you is asynchronous. Neither do you check your mail at regular intervals, nor does the mail delivery person come at the same time everyday. Instead you use the mailbox as a synchronization mechanism (much like the ”Ready bit”). Some mailboxes are located at the street, rather than at the door. They usually come equipped with a flag that the mail delivery person lifts when depositing mail, and you lower when removing mail. The flag is very much a ready bit.

(c) The interaction between a mouse and the PC is synchronous. The PC samples mouse movements at specific intervals. At each interval, the direction and speed of the mouse is read by the PC. No synchronization is needed in this interaction.

8.5 Bit [15] of the KBSR is the ready bit. This is used as the synchronization mechanism to let the processor know that input has occurred. If KBSR[15] is 0, no key has been struck and the value in KBDR is not valid. If KBSR[15] is 1, the value in KBDR is the ASCII code corresponding to the last key struck.

8.6 If KBSR[15] is 0, the data contained in the KBDR has already been read. The program would read the same character again.

8.7 Memory mapped and polling. The system is memory-mapped because KBSR and KBDR device registers have assigned addresses in the memory address space of the ISA. The system is polling because the Ready bit is tested to see if a key has been struck.

8.8 Check if the value in memory location x4000 is between 0 and 127.

```
START     LDI     R0,A
          BRn     DONE      ;Branch if value is less than 0
          LD      R1,B
          ADD     R1,R0,R1
          BRp     DONE      ;Branch if value is above 127
          OUT
DONE      HALT
A         .FILL   x4000
B         .FILL   #-127
```

8.9 If KBSR[15] is 1, the data contained in the KBDR has not been read by the processor. Thus, if the keyboard hardware does not check the KBSR before writing to the KBDR, user input could be lost.

8.10 The display device is an output device and can hence not write to the DDR.

8.11 Interrupt-driven I/O is more efficient than polling. Because, in polling, the processor needs to check a specific register (or memory location) regularly to see if anything is being input or output. This consumes unnecessary processing power because the processor checks the register periodically (stopping all other jobs) even when nothing is being input or output. (Most of the time the register will not be inputting or outputting anything unless it is a really I/O-intensive program). However, in interrupt-driven I/O, when something is input or output by a device, the device sends a signal to the processor. Only when the processor receives that signal, it stops all other jobs and does the I/O. Hence, processing power is used for I/O only when it is necessary to do so.

8.12 Assume that the KBDSR is assigned to address xF400.

```
START    LDI         R0, A
         BRz         START      ;Branch if KBDSR
                                ;contains zero
         AND         R1, R1, #0
         STI         R1, A      ;Clear the KBDSR
         BR          NEXT_TASK  ;Goto the next task
A        .FILL       xF400
```

8.13 Suppose the LC-3 datapath allows combining the two registers into one. Using separate registers, the test to see if the Ready bit is set simply involves checking bit 15 of the status register. This is performed using a branch instruction that tests if the value of the register is negative. If the KBSR and DSR are combined, the test to see if the Display device Ready bit is set involves masking out bit 14 and testing if the bit is set or cleared. Doing it this way requires more instructions than the first method.

8.14 The address control logic takes care of this. It accesses the KBDR if the address is xFE02. For the user, this access to KBDR looks like a normal load instruction.

8.15 NOTE: Please refer to the errata for the new problem statement.

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

(b) The character typed is echoed twice to the screen.

(c) The digit 2 some number of times, followed by the digit typed twice or three times, followed by the digit 2 continually thereafter.

(d) The digit typed will be displayed to the screen twice or three times, depending on when the typed character interrupted the program. If the program was interrupted immediately after LD R0, B , the character typed would appear on the screen three times.

8.16 This program outputs ABCDEFGHI.

F.9 Chapter 9 Solutions

9.1 The most important advantage of doing I/O through a trap routine is the fact that it is not necessary for the programmer to know the gory low-level details of the specific hardware's input/output mechanism. These details include:

- the hardware data registers for the input and output devices
- the hardware status registers for the input and output devices
- the asynchronous nature of the input relative to the executing program

Besides, these details may change from computer to computer. The programmer would have to know these details for the computer she's working on in order to be able to do input/output. Using a trap routine requires no hardware-specific knowledge on part of the programmer and saves time.

9.2 (a) The trap vector is 8 bits wide. 256 trap routines can be implemented in the LC-3.

(b) 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.

(c) One.

9.3 (a) Some external mechanism is the only way to start the clock (hence, the computer) after it is halted. The Halt service routine can never return after bit 15 of the machine control register is cleared because the clock has stopped, which means that instruction processing has stopped.

(b) STI R0, MCR This instruction clears the most significant bit of the machine control register, stopping the clock.

(c) LD R1, SaveR1

(d) The RET of the HALT routine will bring program control back to the program that executed the HALT instruction. The PC will point to the address following the HALT instruction.

9.4 (a) 1111000000100001 (xf021)

(b) x0430

(c) x0437

(d) HookemHorns

9.5 Note: This problem should be corrected to read as follows:

```
.ORIG  x3000
LEA    R0, LABEL
STR    R1, R0, #3
TRAP   x22
TRAP   x25
```

```
LABEL        .STRINGZ "FUNKY"
LABEL2       .STRINGZ "HELLO WORLD"
             .END
```

Answer: FUN

9.6

```
             .ORIG x3000
             LD  R2, LOWER    ; Load -A
             LD  R3, ASCII        ; Load ASCII difference
             LD  R4, UPPER     ; Load -Z
AGAIN   TRAP  x23             ; Request keyboard input
             ADD  R1, R2, R0
             BRn  EXIT
             ADD  R1, R4, R0
             BRp  EXIT
             ADD  R0, R0, R3    ; Change to lowercase
             TRAP  x21             ; Output to monitor
             BRnzp  AGAIN       ; ... and do it again!!

EXIT          TRAP  x25              ; Halt
LOWER      .FILL  xFFBF          ; FFBF = -A
UPPER      .FILL  xFFA6          ; FFA6 = -Z
ASCII      .FILL  x0020
```

9.7 Note: This problem belongs in chapter 10.

The three errors that arose in the first student's program are:

1. The stack is left unbalanced.

2. The privilege mode and condition codes are not restored.

3. Since the value in R7 is used for the return address instead of the value that was saved on the stack, the program will most likely not return to the correct place.

9.8 If the value in A is a prime number, 1 is stored in memory location RESULT; otherwise, 0 is stored in RESULT.

9.9 (a)

```
ST    R1,    SaveR1
ST    R2,    SaveR2
AND   R0,    R0,    #0  ;Zero out the
                        ;return value
LDI   R1,    MBUSY      ;Load the
                        ;contents of
                        ;machine busy bit
                        ;pattern into R1
LD    R2,    MASK       ;Load the mask, x00FF
AND   R1,    R1,    R2  ;Mask out bits <7:0>
LD    R2,    NMASK
```

```
         ADD   R1,    R1,    R2
         BRnp  Return            ;Branch if bit pattern
                                 ;is not x00FF (some
                                 ;machines busy)
         ADD   R0,    R0,    #1  ;No machines are busy,
                                 ;so return 1
Return   LD    R1,    SaveR1
         LD    R2,    SaveR2
         RET
SaveR1  .FILL  x0000
SaveR2  .FILL  x0000
MBUSY   .FILL  x4001
MASK    .FILL  x00FF
NMASK   .FILL  x-00FF
```

(b)

```
         ST    R1,    SaveR1
         ST    R2,    SaveR2
         AND   R0,    R0,    #0  ;Zero out the
                                 ;return value
         LDI   R1,    MBUSY      ;Load r1 with the
                                 ;contents of the machine
                                 ;busy bit
         LD    R2,    MASK       ;Load the mask, x00FF
         AND   R1,    R1,    R2  ;Mask out bits <7:0>
         BRNP  Return            ;Branch if bit
                                 ;pattern is not x0000
                                 ;(some machines not busy)
         ADD   R0,    R0,    #1  ;All are busy, so
                                 ;return 1
Return   LD    R1,    SaveR1
         LD    R2,    SaveR2
         RET
SaveR1  .FILL  x0000
SaveR2  .FILL  x0000
MBUSY   .FILL  x4001
MASK    .FILL  x00FF
```

(c)

```
        ST   R1,    SaveR1
        ST   R2,    SaveR2
        ST   R3,    SaveR3
        ST   R4,    SaveR4
        AND  R0,    R0,    #0    ;Zero out the
                                 ;return value
        LDI  R1,    MBUSY        ;Load R1 with the
                                 ;machine busy bit pattern
        LD   R2,    MASK         ;R2 will act as a mask
                              ;to mask out the bit needed
        LD   R3,    COUNT        ;R3 will act as the
                                 ;iteration counter
Loop    AND  R4,    R1,    R2  ;Mask off the bit to
                               ;check if machine is busy
        BRp  NotBusy             ;Branch if machine
                                 ;is not busy
        ADD  R0,    R0,    #1    ;Increment number
                                 ;of busy machines
NotBusy ADD  R2,    R2,    R2  ;Left shift mask to the
                                 ;next bit to be checked
        ADD  R3,    R3,    #-1   ;Decrement
                                 ;iteration counter
        BRp  Loop              ;Branch if counter is not zero
Return  LD   R1,    SaveR1
        LD   R2,    SaveR2
        LD   R3,    SaveR3
        LD   R4,    SaveR4
        RET
SaveR1  .FILL x0000
SaveR2  .FILL x0000
SaveR3  .FILL x0000
SaveR4  .FILL x0000
MBUSY   .FILL x4001
MASK    .FILL x0001
COUNT   .FILL #8
```

(d)

```
        ST   R1,    SaveR1
        ST   R2,    SaveR2
        ST   R3,    SaveR3
        ST   R4,    SaveR4
        AND  R0,    R0,    #0    ;Zero out the
                                 ;return value
        LDI  R1,    MBUSY        ;Load R1 with the machine
                                 ;busy bit pattern
        LD   R2,    MASK         ;R2 will act as a mask to
                                 ;mask out the bit needed
        LD   R3,    COUNT        ;R3 will act as the
                                 ;iteration counter
Loop    AND  R4,    R1,    R2    ;Mask off the bit to check
                                 ;if machine is busy
        BRz  Busy                ;Branch if machine
                                 ;is busy
        ADD  R0,    R0,    #1    ;Increment number
```

```
                                 ;of not
                                 ;busy machines
Busy    ADD  R2,    R2,    R2    ;Left shift mask to the
                                 ;next bit to be checked
        ADD  R3,    R3,    #-1   ;Decrement
                                 ;iteration counter
        BRp  Loop                ;Branch if counter is not zero
Return  LD   R1,    SaveR1
        LD   R2,    SaveR2
        LD   R3,    SaveR3
        LD   R4,    SaveR4
        RET
SaveR1  .FILL x0000
SaveR2  .FILL x0000
SaveR3  .FILL x0000
SaveR4  .FILL x0000
MBUSY   .FILL x4001
MASK    .FILL x0001
COUNT   .FILL #8
```

(e)

```
        ST   R1,    SaveR1
        ST   R2,    SaveR2
        ST   R3,    SaveR3

        AND  R0,    R0,    #0    ;Zero out the
                                 ;return value

ADD  R1,    R0,    #1
ADD  R3,    R5,    #0
BRz  Check
LP1 ADD  R1,    R1,    R1    ; Left-shift R1
ADD  R3,    R3,   #-1
BRnp LP1

        LDI  R2,    MBUSY        ;Load R2 with the machine
                                 ;busy bit pattern

Check   AND  R1,    R1,    R2

        BRz NotBusy                ;Branch if machine
                                 ;is busy
        ADD  R0,    R0,    #1
NotBusy LD   R1,    SaveR1
        LD   R2,    SaveR2
        LD   R3,    SaveR3
        RET
```

```
    SaveR1  .FILL x0000
    SaveR2  .FILL x0000
    SaveR3  .FILL x0000
    MBUSY   .FILL x4001
(f) ; This code assumes that at least one machine is free
            ST   R1,    SaveR1
            ST   R2,    SaveR2
            ST   R3,    SaveR3
            ST   R4,    SaveR4
            AND  R0,    R0,    #0    ;Zero out the
                                     ;return value
            LDI  R1,    MBUSY        ;Load R1 with the machine
                                     ;busy bit pattern
            LD   R2,    MASK         ;R2 will act as a mask to
                                     ;mask out the bit needed
            LD   R3,    COUNT        ;R3 will act as the
                                     ;iteration counter
    Loop    AND  R4,    R1,    R2    ;Mask off the bit to check
                                     ;if machine is busy
            BRz  Return              ;Branch if machine is free

    ADD  R2,    R2,    R2    ;Left shift mask to the
                                     ;next bit to be checked

    ADD  R0,    R0,    #1
            ADD  R3,    R3,    #-1
            BRp  Loop                ;Branch if counter is not zero

    Return  LD   R1,    SaveR1
            LD   R2,    SaveR2
            LD   R3,    SaveR3
            LD   R4,    SaveR4
            RET
    SaveR1  .FILL x0000
    SaveR2  .FILL x0000
    SaveR3  .FILL x0000
    SaveR4  .FILL x0000
    MBUSY   .FILL x4001
    MASK    .FILL x0001
    COUNT   .FILL #8
```

9.10 Since the LC-3 ISA allows for an 8-bit trap vector, 256 service routines can be created using the current semantics of the LC-3 ISA. However, if the address specified by the TRAP instruction contained the first instruction in the service routine, the number of possible service routines would be greatly reduced. If each service routine required 16 locations, then the number of possible service routines would only be 16 (256/16=16). The semantics of the TRAP instruction could be modified as follows: Change the trap vector to 4 bits (instead of 8); zero-extend the trap vector and shift it to the left by 4 to get the starting address of the service routine.

9.11 The label S_CHAR cannot be represented in 9-bit signed PC offset for the ST R0, S_CHAR and LEA R6, S_CHAR instructions. The range for a PCoffset9 instruction (such as LEA or ST) is only from -256 to 255 locations. Due to the number of locations that have been set aside for BUFFER, the location labeled S_CHAR falls oustide of this range for the ST and LEA instructions. This problem can be fixed by switching the lines BUFFER .BLKW 1001 and S_CHAR .FILL x0000.

9.12 The final values at DATA are the sorted version of the initial values at DATA in ascending order.

9.13 The linkage for JSR A is destroyed when JSR B is executed.

9.14 If the RUN latch is later set (manually), the service routine will restore the values in R0,R1, and R7 and return to the calling program. This use of the TRAP x25 instruction can be a useful tool in troubleshooting and debugging.

9.15 (a) TRAP x72

(b) Yes, this routine will work, but whatever value was in R0 before TRAP x72 is executed will be overwritten during the subroutine.

9.16 Error 1: The line VALUE .FILL X30000 will generate an assembly error because 0x30000 does not fit in one LC-3 memory location.

only one error in current problem statement

9.17 (a) LD R3, NEGENTER
(b) STR R0, R1, #0
(c) ADD R1, R1, #1
(d) STR R2, R2, #0

9.18 (a) ADD R1, R1, #1
(b) TRAP x25
(c) ADD R0, R0, #5
(d) BRzp K

9.19 (a) LD R2, MASK8
(b) JSR HARDDISK
(c) BR END
(d) LD R2, MASK4
(e) JSR ETHERNET
(f) BR END
(g) LD R2, MASK2
(h) JSR PRINTER
(i) BR END

(j) JSR CDROM
(k) HALT

F.10 Chapter 10 Solutions

10.1 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.

10.2 The entries in the model in Figure 10.2 actually move when other entries are pushed and popped, while they do not in the model of Figure 10.3. If the stack is implemented in memory, it makes more sense to access the one entry alone, plus the stack pointer, rather than access all entries on the stack. If the stack is implemented as a piece of tailored logic, it is faster to physically move the actual entries – provided that the power required to do so can handle it. But that is a subject for a later course.

10.3 (a) PUSH R1

(b) POP R0

(c) PUSH R3

(d) POP R7

10.4 This routine copies the value of the first element on stack into R0. If underflow occurs, R5 is set to 1 (failure) else R5 remains 0 (success). Overflow error checking is not necessary because we are not adding anything to the stack.

```
PEEK            AND    R5, R5, #0    ; initialize R5
                LEA    R0, StackBase
                NOT    R0, R0
                ADD    R0, R0, #-1  ;R0 = -(addr of
                                    ;stackbase + 1)
                ADD    R0, R0, R6   ;R6 - stack pointer
                BRZ    Underflow
                LDR    R0, R6, #0   ;put the first
                                    ;element in R0
                RET
Underflow       ADD    R5, R5, #1   ;failure
                RET
StackMax       .BLKW   10, x0000
StackBase      .FILL   x0000
```

10.5 One way to check for overflow and underflow conditions is to keep track of a pointer that tracks the bottom of the stack. This pointer can be compared with the address of the first and last addresses of the space allocated for the stack.

```
;
; Subroutines for carrying out the PUSH and POP functions. This
; program works with a stack consisting of memory locations x3FFF
; (BASE) through x3FFB (MAX). R6 is the bottom of the stack.
```

```
;

POP             ST  R1, Save1 ; are needed by POP.
                ST  R2, Save2
                ST  R3, Save3
                LD  R1, NBASE   ; BASE contains -x3FFF.
                ADD R1, R1, #-1 ; R1 contains -x4000.
                ADD R2, R6, R1  ; Compare bottom of stack to x4000

                BRz fail_exit   ; Branch if stack is empty.

                LD  R1, BASE    ;Iterate from the top of
                                ;the stack
                LDI R0, BASE    ;Load the value from the
                NOT R3, R6      ;top of stack
                ADD R3, R3, #1  ;Generate the
                                ;negative of the
                                ;bottom-of-stack pointer
                ADD R6, R6, #1  ;Increment the
                                ;bottom-of-stack
                                ;pointer

pop_loop        ADD   R2, R1, R3  ;Compare iterating
                                  ;pointer to
                                  ;bottom-of-stack pointer
                BRz   success_exit;Branch if no more
                                  ;entries to shift
                LDR   R2, R1, #-1  ;Load the entry to shift
                STR   R2, R1, #0  ;Shift the entry
                ADD   R1, R1, #-1  ;Increment the
                                  ;iterating pointer
                BRnzp pop_loop

PUSH            ST  R1, Save1 ; Save registers that
                ST  R2, Save2 ; are needed by PUSH.
                ST  R3, Save3
                LD  R1, MAX   ; MAX contains -x3FFB
                ADD R2, R6, R1 ; Compare stack pointer to -x3FFB
                BRz fail_exit ; Branch if stack is full.

                ADD R1, R6, #0 ;Iterate from the bottom
                                ;of stack
                LD  R3, NBASE    ;NBASE contains
                                ;-x3FFF
                ADD R3, R3, #-1   ; R3 = -x4000
```

```
push_loop       ADD   R2, R1, R3  ;Compare iterating
                                  ;pointer to
                                  ;bottom-of-stack pointer
                BRz   push_entry  ;Branch if no more
                                  ;entries to shift
                LDR   R2, R1, #0  ;Load the entry to shift
                STR   R2, R1, #-1  ;Shift the entry
                ADD   R1, R1, #1 ;Decrement the
                                  ;iterating pointer
                BRnzp push_loop

push_entry      ADD   R6, R6, #-1  ;Increment the
                                  ;bottom-of-stack pointer
                STI   R0, BASE    ;Push a value onto stack
                BRnzp success_exit

success_exit    LD    R1, Save1   ;Restore original
                LD    R2, Save2   ;register values
                LD    R3, Save3
                AND   R5, R5, #0  ;R5 <--- success
                RET

fail_exit       LD    R1, Save1   ;Restore original
                LD    R2, Save2   ;register values
                LD    R3, Save3
                AND   R5, R5, #0
                ADD   R5, R5, #1  ;R5 <--- failure
                RET

BASE            .FILL x3FFF
NBASE           .FILL xC001 ; NBASE contains -x3FFF.
MAX             .FILL xC005

Save1           .FILL x0000
Save2           .FILL x0000
Save3           .FILL x0000
```

10.6 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 <-- -x4000
                ADD     R2, R6, R2
                BRz     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   xC000
SaveR2          .FILL   x0000
```

10.7
```
; Subroutines for carrying out the PUSH and POP functions. This
; program works with a stack consisting of memory locations x3FFF
; (BASE) through x3FFB (MAX). R6 is the stack pointer. R3 contains
; the size of the stack element. R4 is a pointer specifying the
; location of the element to PUSH from or the space to POP to
;
```

```
POP             ST      R2, Save2 ; are needed by POP.
                ST      R1, Save1
                ST      R0, Save0
                LD      R1, BASE ; BASE contains -x3FFF.
                ADD     R1, R1, #-1 ; R1 contains -x4000.
                ADD     R2, R6, R1 ; Compare stack pointer to x4000
                BRz     fail_exit ; Branch if stack is empty.
                ADD     R0, R4, #0
                ADD     R1, R3, #0
                ADD     R5, R6, R3
                ADD     R5, R5, #-1
                ADD     R6, R6, R3

pop_loop        LDR     R2, R5, #0
                STR     R2, R0, #0
                ADD     R0, R0, #1
                ADD     R5, R5, #-1
                ADD     R1, R1, #-1
                BRp     pop_loop
                BRnzp   success_exit

PUSH            ST      R2, Save2 ; Save registers that
                ST      R1, Save1 ; are needed by PUSH.
                ST      R0, Save0
                LD      R1,MAX ; MAX contains -x3FFB
                ADD     R2,R6,R1 ; Compare stack pointer to -x3FFB
                BRz     fail_exit ; Branch if stack is full.
                ADD     R0, R4, #0
                ADD     R1, R3, #0
                ADD     R5, R6, #-1

                NOT     R2, R3
                ADD     R2, R2, #1
                ADD     R6, R6, R2

push_loop       LDR     R2, R0, #0
                STR     R2, R5, #0
                ADD     R0, R0, #1
                ADD     R5, R5, #-1
                ADD     R1, R1, #-1
                BRp     push_loop

success_exit    LD      R0, Save0
                LD      R1, Save1 ; Restore original
                LD      R2, Save2 ; register values.
                AND     R5, R5, #0 ; R5 <-- success.
                RET
fail_exit       LD      R0, Save0
                LD      R1, Save1 ; Restore original
                LD      R2, Save2 ; register values.

                AND     R5, R5, #0
                ADD     R5, R5, #1 ; R5 <-- failure.
                RET
BASE            .FILL   xC001 ; BASE contains -x3FFF.
MAX             .FILL   xC005
Save0           .FILL   x0000
Save1           .FILL   x0000
Save2           .FILL   x0000
```

10.8 (a) The stack looks like the following after each operation. Top of the stack is the rightmost element.

```
PUSH A : A
PUSH B : A B
POP    : A
PUSH C : A C
PUSH D : A C D
POP    : A C
PUSH E : A C E
POP    : A C
POP    : A
PUSH F : A F
```

So the stack contains A F after the PUSH F operation.

(b) The stack looks like the following after each operation. Top of the stack is the rightmost element.

```
PUSH G : A F G
PUSH H : A F G H
PUSH I : A F G H I
PUSH J : A F G H I J
POP    : A F G H I
PUSH K : A F G H I K
POP    : A F G H I
POP    : A F G H
POP    : A F G
PUSH L : A F G L
POP    : A F G
POP    : A F
PUSH M : A F M
```

The stack contains the most elements after PUSH J and PUSH K operations.

(c) Now the stack contains A F M (M is at the top of stack).

10.9 (a) BDECJKIHLG

(b)
```
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
```

(c) 14 different output streams.

10.10 For example, 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   R2, R2, #-1
BRp   LOOP
STI   R0, PRES
HALT
PTR .FILL  x4000
PRES .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 intruction at location x3005 and condition codes were not saved. Let R2 = 5 and hence the condition codes be P=1, N=0, Z=0 before servicing the interrupt. When control is returned to the instruction at location x3006, the BR instruction, the condition codes depend on the processing within the interrupt service routing. If they are P=0, N=0, Z=1, then the Br is not taken. This means that result stored is just the sum of the first five values and not all ten.

10.11 Correction, The question should have read:
In the example of Section 10.2.3, what are the contents of locations 0x01F1 and 0x01F2? They are part of a larger structure. Provide a name for that structure.

x01F1 - 0x6200
x01F2 - 0x6300

They are part of the Interrupt Vector Table.

10.12 (a) PC = x3006

Stack:
——
——

xxxxx - Saved SSP

(b) PC = x6200

Stack:
——
——

PSR of Program A - R6
x3007
xxxxx

(c) PC = x6300

Stack:
——
——

PSR for device B - R6
x6203
PSR of Program A
x3007
xxxxx

(d) PC = x6400

Stack:
——
——

PSR for device C - R6
x6311
PSR for device B
x6203
PSR of Program A
x3007
xxxxx

(e) PC = x6311

Stack:
——
——

PSR for device C
x6311
PSR for device B - R6
x6203
PSR of Program A
x3007
xxxxx

(f) PC = x6203

Stack:
——
——

PSR for device C
x6311
PSR for device B
x6203
PSR of Program A - R6
x3007
xxxxx

(g) PC = x3007

Stack:
——
——

PSR for device C
x6311
PSR for device B
x6203

PSR of Program A
x3007
xxxxx - Saved.SSP

10.13 (a) PC = x3006

Stack:
___
___

xxxxx - Saved SSP

(b) PC = x6200

Stack:
___
___

PSR of Program A - R6
x3007
xxxxx

(c) PC = x6300

Stack:
___
___

PSR for device B - R6
x6203
PSR of Program A
x3007
xxxxx

(d) PC = x6203

Stack:
___
___

PSR for device B
x6203
PSR of Program A - R6
x3007
xxxxx

(e) PC = x6400

Stack:
——
——

PSR for device B - R6
x6204
PSR of Program A
x3007
xxxxx

(f) PC = x6204

Stack:
——
——

PSR for device B
x6204
PSR of Program A - R6
x3007
xxxxx

(g) PC = x3007

Stack:
——
——

PSR for device B
x6204
PSR of Program A
x3007
xxxxx - Saved.SSP

10.14 Correction - If the buffer is full, a character has been stored in 0x40FE.

```
        LDI     R0, KBDR
        LDI     R1, PENDBF
        LD      R2, NEGEND
        ADD     R2, R1, R2
        BRz     ERR         ; Buffer is full
        STR     R0, R1, #0  ; Store the character
        ADD     R1, R1, #1
        STI     R1, PENDBF  ; Update next available empty
```

```
                                 ; buffer location pointer
        BRnzp   DONE
ERR     LEA     R0, MSG
PUTS
DONE    RTI
KBDR    .FILL   xFE02
PBUF    .FILL   x4000
PENDBF  .FILL   x40FF
NEGEND  .FILL   xBF01 ; xBF01 = -(x40FF)
MSG     .STRINGZ  "Character cannot be accepted; input buffer full."
```

10.15 Note: This problem introduces the concept of a data structure called a queue. A queue has a First-In-First-Out(FIFO) property - Data is removed in the order as it is inserted. By having the pointer to the next available empty location wrap around to the beginning of the buffer in this problem, the queue becomes a circular queue. A circular queue is space efficient as it makes use of entries which have been removed by the consuming program. These concepts will be covered in detail in a data structure or algorithms course.

```
        LDI     R0, KBDR
        LDI     R1, PNUMCH
        LD      R2, NMAXCH
        ADD     R2, R1, R2
        BRz     ERR         ; Buffer is full

        ADD     R1, R1, #1
        STI     R1, PNUMCH ; update char count

        LDI     R1, PENDBF
        STR     R0, R1, #0 ; Store the character

        LD      R2, NEGEND
        ADD     R2, R1, R2 ; Compare the next available empty location
                           ; pointer with x40FC.
        BRz     RESETPTR   ; If same, wrap around so that the next available
                           ; empty location is x4000
        ADD     R1, R1, #1
        BRnzp   STPTR
RESETPTR LD     R1, PBUF

STPTR   STI     R1, PENDBF ; Update next available empty buffer
                           ; location pointer
        BRnzp   DONE
ERR     LEA     R0, MSG
        PUTS
DONE    RTI
```

```
KBDR    .FILL   xFE02
PBUF    .FILL   x4000
PNUMCH  .FILL   x40FD
PENDBF  .FILL   x40FF
NEGEND  .FILL   xBF04 ; xBF04 = -(x40FC)
NMAXCH  .FILL   xFF03 ; xFF03 = -(xFD)
MSG     .STRINGZ  "Character cannot be accepted; input buffer full."
```

10.16 Correction - Consider the modified interrupt handler of Exercise 10.15.

The variable “number of characters in the buffer” is shared between both the interrupt handler which is adding numbers to the buffer and the program that is removing characters. So now if the program has just loaded the number of characters in the buffers value into a register when an interrupt occurs, the value in the register is going to be stale after the interrupt is serviced. Hence when the program writes this value back to x40FD, it is writing a wrong value.

10.17 The Multiply step works by adding the multiplicand a number of times to an accumulator. The number of times to add is determined by the multiplier. The number of instructions executed to perform the Multiply step = 3 + 3*n, where n is the value of the multiplier. We will in general do better if we replace the core of the Multiply routine (lines 17 through 19 of Figure 10.14) with the following, doing the Multiply as a series of shifts and adds:

```
                AND    R0, R0, #0
                ADD    R4, R0, #1    ;R4 contains the bit mask (x0001)

Again           AND    R5, R2, R4    ;Is corresponding
                BRz    BitZero       ;bit of multiplier=1
                ADD    R0, R0, R1    ;Multiplier bit=1
                                     ;--> add
                                     ;shifted multiplicand
                BRn    Restore2      ;Product has already
                                     ;exceeded range
BitZero         ADD    R1, R1, R1    ;Shift the
                                     ;multiplicand bits
                BRn    Check         ;Mcand too big
                                     ;--> check if any
                                     ;higher mpy bits = 1
                ADD    R4, R4, R4    ;Set multiplier bit to
                                     ;next bit position
                BRn    DoRangeCheck  ;We have shifted mpy
                BRnzp  Again         ;bit into bit 15
                                     ;-->done.

Check           AND    R5, R2, R4
                BRp    Restore2
                ADD    R4, R4, R4
                BRp    Check
DoRangeCheck
```

10.18 To add the two numbers, convert the two characters from their ASCII representations to 2's complement integers. After performing the addition, convert the result back to ASCII representation.

```
                LD       R2, NEGASCII
                TRAP     x23
                ADD      R1, R0, R2  ;Remove ASCII template
                TRAP     x23
                ADD      R0, R0, R2  ;Remove ASCII template
                ADD      R0, R1, R0
                LD       R2, ASCII
                ADD      R0, R0, R2  ;Add the ASCII template
                TRAP     x21
                TRAP     x25
NEGASCII        .FILL    x-0030
ASCII           .FILL    x0030
```

10.19 This program assumes that hex digits are all capitalized.

```
                LD       R3, NEGASCII
                LD       R5, NEGHEX
                TRAP     x23
                ADD      R1, R0, R3   ;Remove ASCII template
                LD       R4, HEXTEST  ;Check if digit is hex
                ADD      R0, R1, R4
                BRnz     NEXT1
                ADD      R1, R1, R5   ;Remove extra
                                      ;offset for hex
NEXT1           TRAP     x23
                ADD      R0, R0, R3   ;Remove ASCII template
                ADD      R2, R0, R4   ;Check if digit is hex
                BRnz     NEXT2
                ADD      R0, R0, R5   ;Remove extra
                                      ;offset for hex

NEXT2           ADD      R0, R1, R0   ;Add the numbers
                ADD      R1, R0, R4   ;Check if digit > 9
                BRnz     NEXT3
                LD       R2, HEX
                ADD      R0, R0, R2   ;Add offset for hex digits
```

```
NEXT3           LD       R2, ASCII
                ADD      R0, R0, R2   ;Add the ASCII template

DONE            TRAP     x21
                TRAP     x25

ASCII           .FILL    x0030
NEGASCII        .FILL    x-0030
HEXTEST         .FILL    #-9
HEX             .FILL    x0007
NEGHEX          .FILL    x-7
```

10.20

```
ASCIItoBinary   AND     R0, R0, #0         ; R0 will be used for our result
                LEA     R5, ASCIIBUFF
                LD      R4, NegASCIIOffset ; R4 gets xFFD0, i.e., -x0030
                ADD     R1, R1, #0         ; Test number of digits.

LOOP            BRz     DoneAtoB ;

                LD     R2, CNT
                ADD     R3, R0, #0
MUL10           ADD     R0, R0, R3
                ADD     R2, R2, #-1
                BRp     MUL10

                LDR     R3, R5, #0 ;
                ADD     R3, R4, R3 ; Strip off the ASCII template
                ADD     R0, R0, R3 ; Add digits contribution

                ADD     R5, R5, #1
                ADD     R1, R1, #-1
                BRnzp   LOOP

DoneAtoB        RET
NegASCIIOffset  .FILL   xFFD0
CNT             .FILL   #9
ASCIIBUFF       .BLKW   #10
```

10.21

```
;
; R1 contains the number of digits including ’x’. Hex
; digits must be in CAPS.

ASCIItoBinary   AND R0, R0, #0 ; R0 will be used for our result
                ADD  R1, R1, #0 ; Test number of digits.
                BRz  DoneAtoB   ; There are no digits
;
```

```
                LD  R3, NegASCIIOffset ; R3 gets xFFD0, i.e., -x0030
                LEA  R2, ASCIIBUFF
                LD R6, NegXCheck
                LDR R4, R2, #0
                ADD R6, R4, R6
                BRz DoHexToBin

                ADD  R2, R2,R1
                ADD   R2, R2, #-1 ; R2 now points to "ones" digit
;
                LDR  R4, R2, #0 ; R4 <-- "ones" digit
                ADD  R4, R4, R3 ; Strip off the ASCII template
                ADD  R0, R0, R4 ; Add ones contribution
;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had one digit
                ADD  R2, R2, #-1 ; R2 now points to "tens" digit
;
                LDR  R4, R2, #0 ; R4 <-- "tens" digit
                ADD  R4, R4, R3 ; Strip off ASCII template
                LEA  R5, LookUp10 ; LookUp10 is BASE of tens values
                ADD  R5, R5, R4 ; R5 points to the right tens value
                LDR  R4, R5, #0
                ADD  R0, R0, R4 ; Add tens contribution to total
;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had two digits
                ADD  R2, R2, #-1 ; R2 now points to "hundreds" digit
;
                LDR  R4, R2, #0 ; R4 <-- "hundreds" digit
                ADD  R4, R4, R3 ; Strip off ASCII template
                LEA R5, LookUp100 ; LookUp100 is hundreds BASE
                ADD  R5, R5, R4 ; R5 points to hundreds value
                LDR  R4, R5, #0
                ADD  R0, R0, R4 ; Add hundreds contribution to total
                RET

DoHexToBin      ; R3 = NegASCIIOffset
                ; R2 = Buffer Pointer
                ; R1 = Num of digits + x
                ;
                ST R7, SaveR7
                LD R6, NumCheck
                ADD R1, R1, #-1
                ADD  R2, R2,R1
;
                LDR  R4, R2, #0 ; R4 <-- "ones" digit
                ADD  R4, R4, R3 ; Strip off the ASCII template
                ADD R7, R4, R6
                BRnz Cont1
                LD R7, NHexDiff
                ADD R4, R4, R7
Cont1           ADD  R0, R0, R4 ; Add ones contribution

;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had one digit
                ADD  R2, R2, #-1 ; R2 now points to "tens" digit
;
                LDR  R4, R2, #0 ; R4 <-- "tens" digit
                ADD  R4, R4, R3 ; Strip off ASCII template
                ADD R7, R4, R6
                BRnz Cont2
                LD R7, NHexDiff
                ADD R4, R4, R7

Cont2           LEA  R5, LookUp16
                ADD  R5, R5, R4
                LDR  R4, R5, #0
                ADD  R0, R0, R4
;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had two digits
                ADD  R2, R2, #-1 ; R2 now points to "hundreds" digit
;
                LDR  R4, R2, #0
                ADD  R4, R4, R3 ; Strip off ASCII template
                ADD R7, R4, R6
                BRnz Cont3
                LD R7, NHexDiff
                ADD R4, R4, R7

Cont3           LEA R5, LookUp256
                ADD  R5, R5, R4
                LDR  R4, R5, #0
                ADD  R0, R0, R4

;
DoneAtoB        LD R7, SaveR7
```

```
                RET

NegASCIIOffset  .FILL  xFFD0
NumCheck        .FILL  #-9
NHexDiff        .FILL #-7
NegXCheck       .FILL  xFF88
SaveR7          .FILL  x0000

ASCIIBUFF       .BLKW 4
LookUp10        .FILL #0
                .FILL #10
                .FILL #20
                .FILL #30
                .FILL #40
                .FILL #50
                .FILL #60
                .FILL #70
                .FILL #80
                .FILL #90
;
LookUp100       .FILL #0
                .FILL #100
                .FILL #200
                .FILL #300
                .FILL #400
                .FILL #500
                .FILL #600
                .FILL #700
                .FILL #800
                .FILL #900
LookUp16        .FILL     #0
                .FILL     #16
                .FILL     #32
                .FILL     #48
                .FILL     #64
                .FILL     #80
                .FILL     #96
                .FILL     #112
                .FILL     #128
                .FILL     #144
                .FILL     #160
                .FILL     #176
                .FILL     #192
                .FILL     #208
                .FILL     #224
                .FILL    #240

LookUp256       .FILL    #0
                .FILL    #256
                .FILL    #512
                .FILL    #768
                .FILL    #1024
                .FILL    #1280
                .FILL    #1536
                .FILL    #1792
                .FILL    #2048
                .FILL    #2304
                .FILL    #2560
                .FILL    #2816
                .FILL    #3072
                .FILL    #3328
                .FILL    #3584
                .FILL    #3840
```

10.22

```
;
BinarytoASCII   AND     R4, R4, #0
                LD      R5, ASCIIoffset
                LEA     R1, ASCIIBUFF ; R1 points to string being generated
                ADD     R0, R0, #0 ; R0 contains the binary value
                BRn     NegSign ;
                BRnzp   Begin100
NegSign         LD      R2, ASCIIminus ; First store ASCII minus sign
                STR     R2, R1, #0
                NOT     R0, R0 ; Convert the number to absolute
                ADD     R0, R0, #1 ; value; it is easier to work with.
                ADD     R1, R1, #1
;
Begin100        AND     R2, R2, #0 ; Prepare for "hundreds" digit
;
                LD      R3, Neg100 ; Determine the hundreds digit
Loop100         ADD     R0, R0, R3
                BRn     End100
                ADD     R2, R2, #1
                BRnzp   Loop100
;
End100          ADD     R2, R2, #0
                BRz     Post100
                ADD     R2, R2, R5
                STR     R2, R1, #0 ; Store ASCII code for hundreds digit
                ADD     R1, R1, #1
                ADD     R4, R4, #1 ; Stored a digit
Post100         LD      R3, Pos100
                ADD     R0, R0, R3 ; Correct R0 for one-too-many subtracts
;
                AND     R2, R2, #0 ; Prepare for "tens" digit
;
Begin10         LD      R3, Neg10 ; Determine the tens digit
Loop10          ADD     R0, R0, R3
                BRn     End10
                ADD     R2, R2, #1
                BRnzp   Loop10
;
End10           ADD     R4, R4, #0
                BRp     Store10
                ADD     R2, R2, #0
                Brz     Post10
Store10         ADD     R2, R2, R5
                STR     R2, R1, #0 ; Store ASCII code for tens digit
                ADD     R1, R1, #1
Post10          ADD     R0, R0, #10 ; Correct R0 for one-too-many subtracts
Begin1          LD      R2, ASCIIoffset ; Prepare for "ones" digit
                ADD     R2, R2, R0
                STR     R2, R1, #0
                RET
;
ASCIIplus       .FILL x002B
ASCIIminus      .FILL x002D
ASCIIoffset     .FILL x0030
Neg100          .FILL xFF9C
Pos100          .FILL x0064
Neg10          .FILL xFFF6
ASCIIBUFF       .BLKW 4
```

10.23 This program reverses the input string. For example, given an input of “Howdy”, the output is “ydwoH”.

10.24 Please correct the problem to read:

Suppose the keyboard interrupt vector is x34 and the keyboard interrupt service routine starts at location x1000. What can you infer about the contents of any memory location from the above statement?

Solution:

Memory location x0134 contains the address x1000.

9.7 Note: This problem belongs in chapter 10.

The three errors that arose in the first student's program are:

1. The stack is left unbalanced.

2. The privilege mode and condition codes are not restored.

3. Since the value in R7 is used for the return address instead of the value that was saved on the stack, the program will most likely not return to the correct place.