INSTRUCTOR'S SOLUTIONS MANUAL

TO ACCOMPANY

Digital Systems Design Using Verilog

Charles H. Roth, Jr.
University of Texas, Austin

Lizy Kurian John
University of Texas, Austin

Contents

Chapter 1: Review of Logic Design Fundamentals 1

Chapter 2: Introduction to Verilog® 17

Chapter 3: Introduction to Programmable Logic Devices 47

Chapter 4: Design Examples 57

Chapter 5: SM Charts and Microprogramming 95

Chapter 6: Designing with Field Programmable Gate Arrays 123

Chapter 7: Floating-Point Arithmetic 147

Chapter 8: Additional Topics in Verilog 181

Chapter 9: Design of a Risc Microprocessor 203

Chapter 10: Hardware Testing and Design for Testability 223

Chapter 1: Review of Logic Design Fundamentals

1.1

A	B	C	$A \oplus B$	$B' \oplus C$	$(A \oplus B) \cdot C$	$A' \cdot (B' \oplus C)$	F
0	0	0	0	1	0	1	1
0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	0	0	0
1	0	1	1	0	1	0	1
1	1	0	0	0	0	0	0
1	1	1	0	1	0	0	0

1.2

X	Y	B_{in}	$Diff$	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Sum of Products: $Diff = XY'B_{in}' + X'Y'B_{in} + XYB_{in} + X'YB_{in}'$

$B_{out} = X'B_{in} + X'Y + YB_{in}$

Product of Sums: $Diff = (X + Y + B_{in})(X + Y' + B_{in}')(X' + Y + B_{in}')(X' + Y' + B_{in})$

$B_{out} = (Y + B_{in})(X' + B_{in})(X' + Y)$

1.3

CD \ AB	00	01	11	10
00	0	X	S	0
01	X	E	E	X
11	1	0	F′	1
10	0	0	F′	0

S = G + H′

Set all map-entered variables to 0 to get MS_0=B'D. Set E, F', and S to 1 one at a time and all 1's to X's to get MS_1=C'D(E), MS_2=ABC(F'), MS_3=BC'D'(G + H')

$Z = B'D + C'DE + ABCF' + BC'D'G + BC'D'H'$

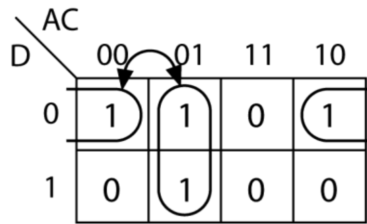
1.4 **(a)** $F = A'D' + AC'D + BCD' + A'B'C'E + BD'E$

(b) $Z = A'CD' + C'D + BC'E + B'DE + CD'F + A'C'G$

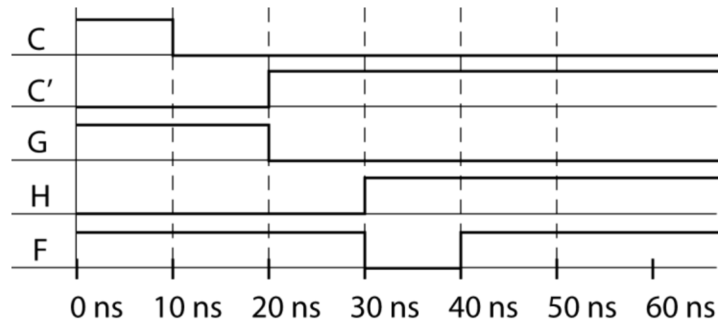
(c) $H = A'CD + A'B'CE + BCDF'$

(d) $G = C'E'F + DEF + AD'F'$

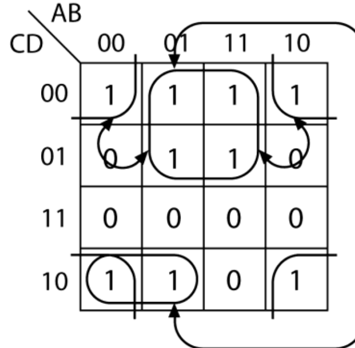
1.5



A static 1-hazard occurs when $A = 0$, $D = 0$, and C changes. When C changes from 1 to 0; $A'C$ also goes from 1 to 0. The hazard occurs because C' hasn't become 1 yet since it has to go through the inverter; therefore, F goes to 0 momentarily before going to 1. Gate delays are assumed to be 10ns in the timing diagram below.



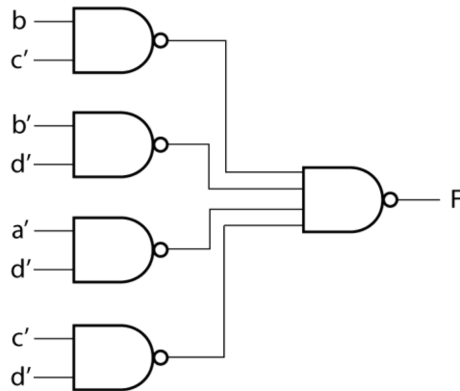
1.6



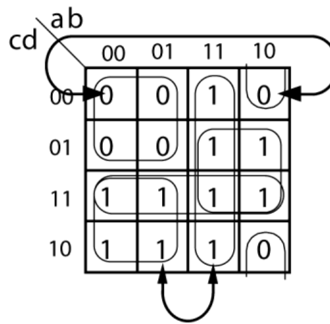
$F = bc' + b'd' + a'cd'$

3 hazards: A=0, C=0, D=0, B changes
A=1, C=0, D=0, B changes
A=0, B=1, D=0, C changes

To eliminate the hazards, add the term $c'd'$ (combining the four 1's in the top row) and replace $a'cd'$ with $a'd'$ (combining two 1's from the bottom left with two 1's from the top left.)



1.7 **(a)**



$$
\begin{aligned}
F &= ((ab)'.(a+c)'+(a'+d)')' \\
&= ab + ((a+c)'+(a'+d)')' \\
&= ab + (a + c)(a' + d) \\
&= ab + aa' + ad + a'c + cd;
\end{aligned}
$$

Circle all these terms on the K-map; arc shows nearby1's not in the same product term, indicating a 1 hazard.

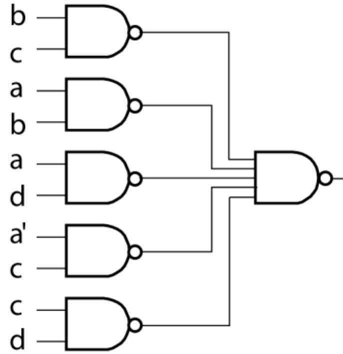
1-hazard $bcd = 110$, a changing 1 => 0 gates 1,5,3,4,5,
0 => 1 gates 3,4,5,1,5

$$
\begin{aligned}
F &= ab + aa' + ad + a'c + cd \\
&= ab + a(a' + d) + c(a' + d) \\
&= ab + (a+c)(a' + d) \\
&= (ab + a + c)(ab + a' + d) && \text{--see Table 1-1 for Boolean laws} \\
&= (a + c)(a + a' + d)(a' + b + d) && \text{-- a+ab=a; a'+d+ab=(a'+d+a)(a'+d+b)}
\end{aligned}
$$

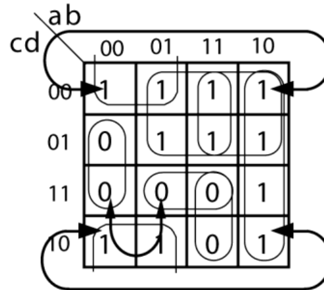
Circle all these terms of 0's in the K-map; arc shows 0's not in same term.

0-hazard $bcd = 000$, a changing 0 => 1 gates 3,4,5,3,5
1 => 0 gates 3,4,5,2,5

(b) We will design a 2-level sum of products circuit because a 2-level sum-of-products circuit has no 0-hazard as long as an input and its complement are not connected to the same AND gate. Avoid 1-hazard by adding product term bc. Use NAND gates as asked in the question.



1.8



(a) $Z = A'D' + (A + B)(B' + C')$
$= A'D' + AB' + AC' + BB' + BC'$

Static 1 hazard (see the arcs between nearby 1's not in the same product term.)
$ABCD = 0000$ to 1000
$ABCD = 0010$ to 1010

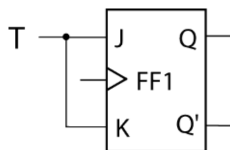
$Z = (A'D' + A + B)(A'D' + B' + C')$
$= (A' + A + B)(D' + A + B)(A' + B' + C')(D' + B' + C')$

Static 0 hazard (see the arcs between nearby 0's not in the same term)
$ABCD = 0111$ to 0011

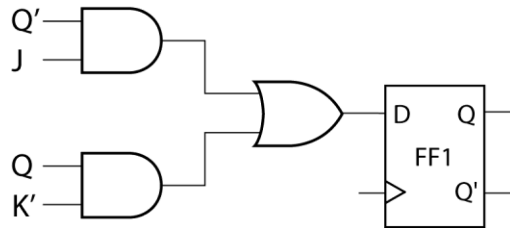
(b) One can design a hazard-free sum of products circuit as in the previous question. Or, one can design a product of sums (POS) circuit with no hazards. A properly designed 2-level POS circuit has no 1-hazards. Static 0-hazards can be avoided by including loops for all 0's that are adjacent. 4 terms here including the arc:

$Z = (A + B + D')(A' + B' + C')(D' + B' + C')(C' + D' + A)$
$= (D' + A + BC')(A' + B' + C')(D' + B' + C')$ combining 1st and 4th terms
$= (D' + A + BC')(B' + C' + A'D')$ combining 2nd and 3rd terms

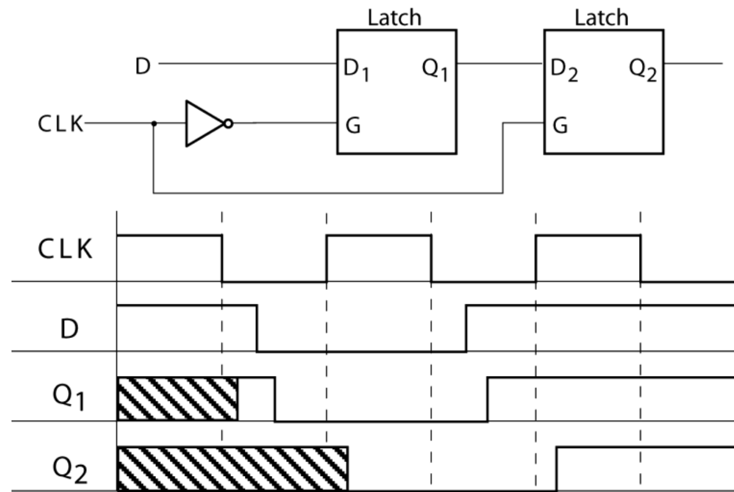
1.9 **(a)**



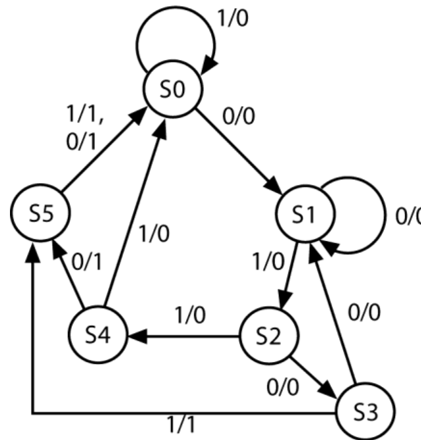
(b) From the characteristic equation for a J-K flip-flop ($Q^+ = JQ' + K'Q$):



1.10



1.11 (a)



Present State		Next State $X = 0$	Next State 1	Output $X = 0$	Output 1
Reset	S0	S1	S0	0	0
0	S1	S1	S2	0	0
01	S2	S3	S4	0	0
010	S3	S1	S5	0	1
011	S4	S5	S0	1	0
0101 or 0110	S5	S0	S0	1	1

(b) Guidelines:
I. (0,1,3),(0,4,5)
II. (0,1),(1,2),(3,4),(1,5),(0,5)
III. (0,1,2,3),(4,5),(3,5)

For state assignment:
S0 = 000 S1 = 001 S2 = 010 S3 = 011 S4 = 100 S5 = 101

Equations for NAND gate network:

$J_1 = XQ_2$ $K_1 = X + Q_3$

$J_2 = XQ_1'Q_3$ $K_2 = X + Q_3$

$J_3 = X'$ $K_3 = XQ_2' + Q_1$

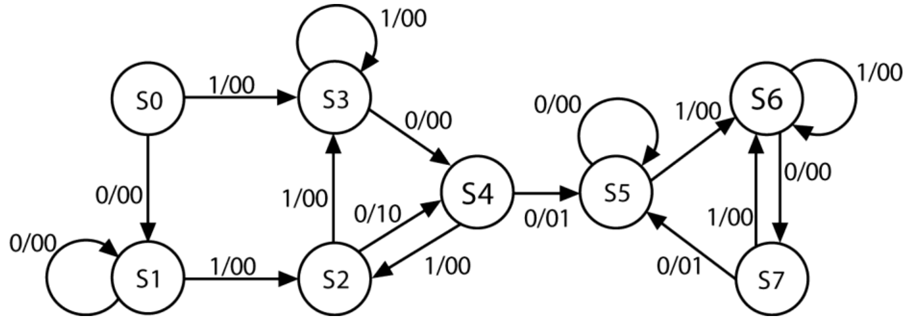
$Z = XQ_2Q_3 + X'Q_1 + Q_1Q_3$

For NOR gate network, use product of sums form:

$K_3 = (X + Q_1)(Q_2')$

$Z = (Q_1 + Q_2)(X' + Q_3)(X + Q_2')$

1.12 (a)



Present State	Next State $X = 0$	1	Output $X = 0$	1
S0	S1	S3	00	00
S1	S1	S2	00	00
S2	S4	S3	10	00
S3	S4	S3	00	00
S4	S5	S2	01	00
S5	S5	S6	00	00
S6	S7	S6	00	00
S7	S5	S6	01	00

(b) Guidelines:
I. (0,1),(2,3),(4,5,7),(0,2,3),(1,4),(5,6,7)
II. (1,3),(1,2),2x(3,4),(2,5),2x(5,6),(6,7)
III. (0,1,3,5,6),(4,7)

For state assignment:
S0 = 000 S1 = 100 S2 = 001 S3 = 101
S4 = 111 S5 = 011 S6 = 010 S7 = 110

Equations for NAND gate network:

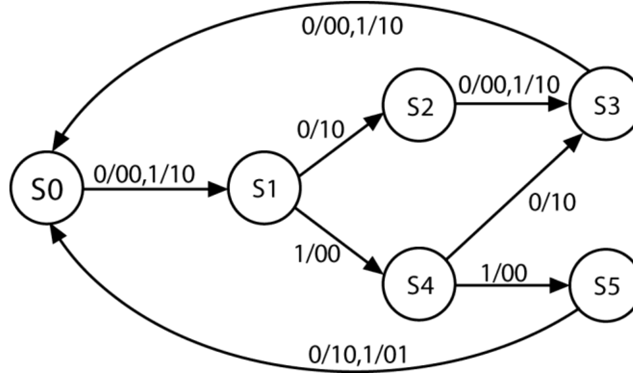
$J_1 = Q_2' + X'Q_3'$ $K_1 = XQ_3' + Q_2$

$J_2 = X'Q_3$ $K_2 = XQ_1Q_3$

$J_3 = XQ_2' + X'Q_1Q_2$ $\quad$ $K_3 = XQ_1'Q_2$
$Z_1 = X'Q_1'Q_2'Q_3$
$Z_2 = X'Q_1Q_2$

For NOR gate network , use product of sums form:
$J_1 = (Q_2' + Q_3')(X' + Q_2')$ $\quad$ $K_1 = (X + Q_2)(Q_2 + Q_3')$
$J_3 = (X + Q_2)(X' + Q_2')(X + Q_1)$

1.13 **(a)**



Present State	Next State X = 0	1	Output X = 0	1
S0	S1	S1	00	10
S1	S2	S4	10	00
S2	S3	S3	00	10
S3	S0	S0	00	10
S4	S3	S5	10	00
S5	S0	S0	10	01

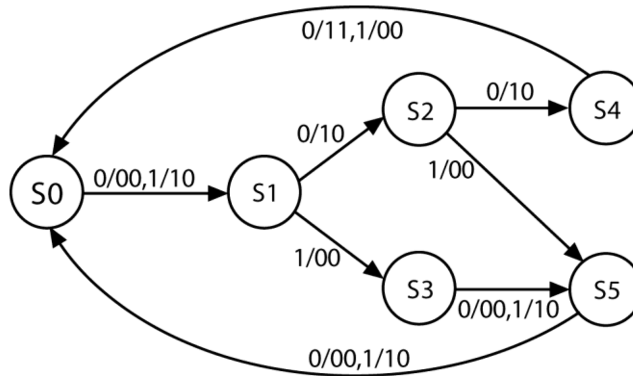
(b) Guidelines:
I. (2,4),2x(3,5)
II. (2,4),(3,5)
III. (0,2,3),(1,4,5)

For state assignment:
S0 = 000 S1 = 010 S2 = 001 S3 = 101 S4 = 011 S5 = 111

Equations for NAND gate network:
$D_1 = Q_1'Q_3$
$D_2 = Q_2'Q_3' + XQ_1'Q_2$
$D_3 = Q_1'Q_3 + Q_2Q_3'$ or $Q_1'Q_3 + Q_1'Q_2$
$S = XQ_2' + X'Q_2$
$V = XQ_1Q_2$

For NOR gate network , use product of sums form:
$S = (X + Q_2)(X' + Q_2')$
$D_2 = Q_1'(Q_2 + Q_3')(X + Q_2')$
$D_3 = Q_1'(Q_2 + Q_3)$

1.14 **(a)**



Present State	Next State X = 0	1	Output X = 0	1
S0	S1	S1	00	10
S1	S2	S3	10	00
S2	S4	S5	10	00
S3	S5	S5	00	10
S4	S0	S0	11	00
S5	S0	S0	00	10

(b) Guidelines:
I. (4,5),(2,3)
II. (2,3),(4,5)
III. (0,3,5),(1,2,4)

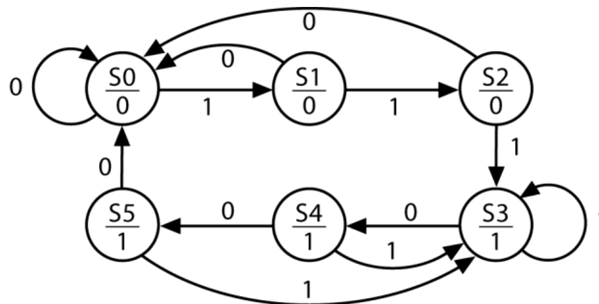
For state assignment:
S0 = 000 S1 = 100 S2 = 101 S3 = 001 S4 = 111 S5 = 011

Equations for NAND gate network:

$J_1 = Q_3'$ $K_1 = X + Q_2$
$J_2 = Q_3$ $K_2 = 1$
$J_3 = Q_1$ $K_3 = Q_2$
$D = X'Q_1 + XQ_1'Q_3$
$B = X'Q_1Q_2$

For NOR gate network, use product of sums form:
$D = (X' + Q_1')(X + Q_1)(Q_1 + Q_3)$

1.15



Present State	Next State X = 0	1	Output
S0	S0	S1	0
S1	S0	S2	0
S2	S0	S3	0
S3	S4	S3	1
S4	S5	S3	1
S5	S0	S3	1

1.16

Present State ($Q_2Q_1Q_0$)	Next State ($Q_2^+Q_1^+Q_0^+$)
000	001
001	010
010	011
011	100
100	101
101	000

$J_2 = Q_1Q_0$ $\quad K_2 = Q_0$

$J_1 = Q_2'Q_0$ $\quad K_1 = Q_0$

$J_0 = 1$ $\quad K_0 = 1$

1.17

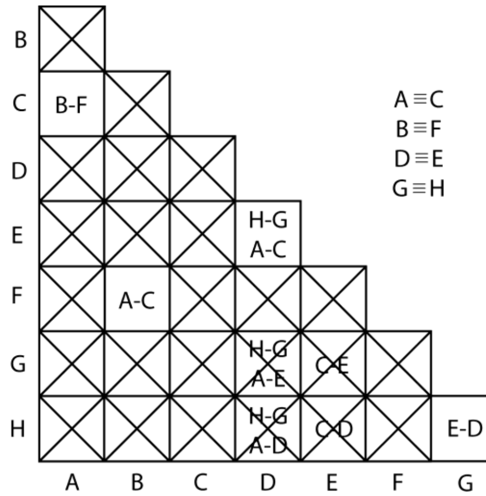
Present State ($Q_2Q_1Q_0$)	Next State ($Q_2^+Q_1^+Q_0^+$)
001	010
010	011
011	100
100	101
101	110
110	001

$D_2 = Q_1Q_0 + Q_2Q_1'$

$D_1 = Q_2'Q_0' + Q_1'Q_0$

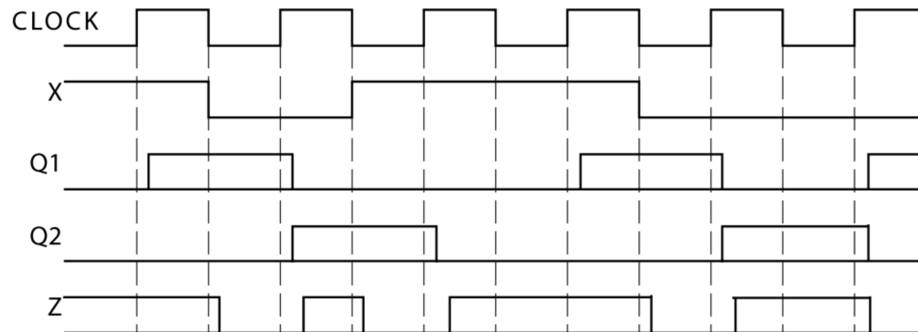
$D_0 = Q_0'$

1.18



present state	next state $X = 0$	next state $X = 1$	output $X = 0$	output $X = 1$
A	B	G	0	1
B	A	D	1	1
~~C~~	~~F~~	~~G~~	~~0~~	~~1~~
D	~~H~~ G	A	0	0
~~E~~	~~G~~	~~C~~	~~0~~	~~0~~
~~F~~	~~C~~	~~D~~	~~1~~	~~1~~
G	G	~~E~~ D	0	0
~~H~~	~~G~~	~~D~~	~~0~~	~~0~~

1.19 **(a)**



Z should be read just before the rising edge of the clock.

(b) Worst case $t_{xor} + t_p + t_{su} \le t_{clk}$

$$20\text{ns} + 10\text{ns} + 5\text{ns} \le t_{clk}$$

$$t_{clk} \ge 35 \text{ ns}$$

Clock Rate = 28.6 MHz

However, the input X changes at the same time as the falling edge of the clock. Data is clocked into D flip-flop at the rising edge of the clock. Therefore, the time t between the falling edge and the rising edge of the clock should satisfy the gate delay of XOR and also the setup time

$$t_{clk} \ge 20\text{ns} + 5\text{ns}$$

$$t_{clk} \ge 50\text{ns}$$

Clock Rate = 20 MHz

For proper synchronous operation, both condition 1 and condition 2 should be satisfied.

$t_{clk} \geq$ 50ns is the limiting factor

Therefore, Clock Rate = 20 MHz

(c) Q1 should remain unchanged for 2ns (t_h) after D2 is clocked

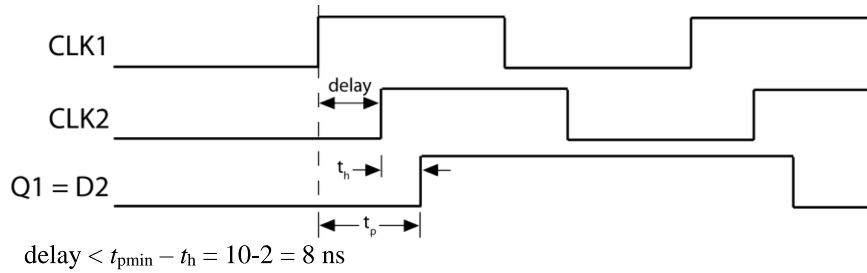
Q1 will be constant for at least 5ns (t_{pmin}) after the rising clock edge

$t_{constnat} = t_h$ + delay = 2ns + delay

$t_{constnat} = t_{pmin}$ = 5ns

delay = $t_{pmin} - t_h$ = 3 ns

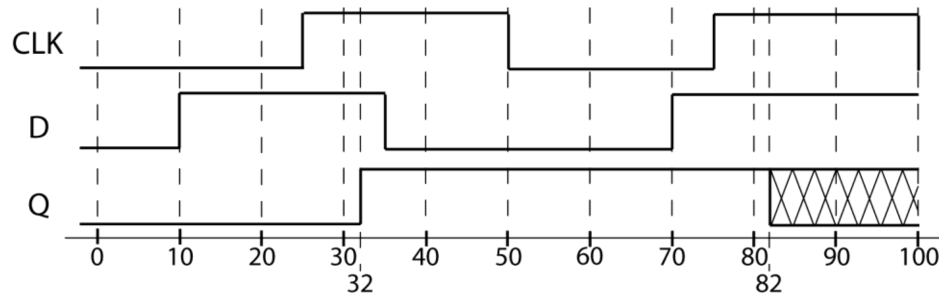
1.20 **(a)**



delay < $t_{pmin} - t_h$ = 10-2 = 8 ns

(b) $t_{clk} \geq t_{pmin} + t_{su}$ = 15+4 = 19 ns. (worst case occurs when delay = 0)

1.21 **(a)**



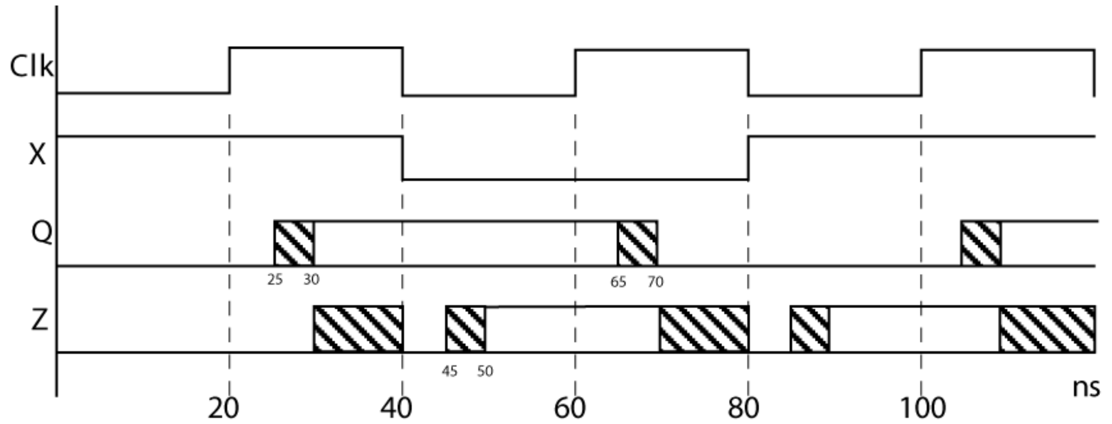
(b) By definition of set up and hold time, D should be constant 10 ns (t_{su}) before, and 5 ns (t_h) after the clock edge.

(c) External inputs should not change 18 ns before, and 1 ns after clock edge.

$t_y = t_{cmax} + t_{su}$ = (4+4) + 10 = 18

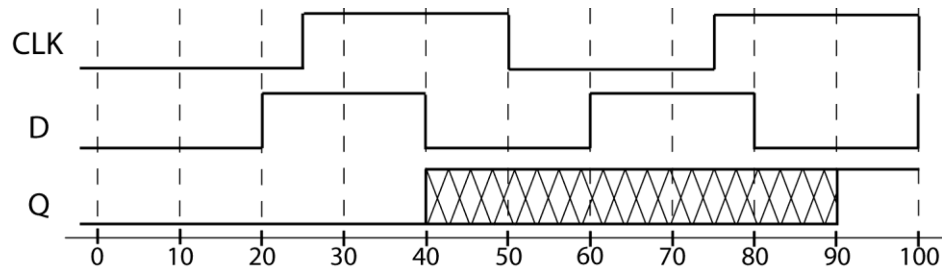
$t_x = t_h - t_{cmin}$ = 5 – (2+2) = 1

1.22 **(a)**



(b) Setup time = 20 ns -10 ns = 10 ns (due to change in X)
Hold time = $t_{pmin} + t_{pla}$ = 5 ns+ 5 ns= 10 ns (due to change in Q)

1.23



1.24 **(a)** $t_{clk} \geq t_{su} + t_{pmax} + t_{cmax}$
$t_{clk} \geq 5 + 12 + 4 = 21$ ns
$t_{clkmin} = 21$ ns

(b) $t_x = t_h - t_{cmin} = 3 - 1 = 2$ ns. X can change as early as 2 ns after the clock edge.

1.25 **(a)** 16 ns + 24 ns + 8 ns = <u>48 ns.</u> (to satisfy setup time) (hold time is not a problem because it takes at least 12 + 2 = 14 ns from rising clock edge until D changes)

(b) earliest time: to satisfy hold time, t_h - t_{cmin} = 4 ns – 2 ns = <u>2 ns</u>
latest time: to satisfy setup time, 8 ns + 16 ns = 24 ns before rising clock edge. 48 ns clock – 24 ns = <u>24 ns</u> after rising clock edge

1.26 **(a)** The maximum delay path of this circuit starts at flip-flop 2 and ends at flip-flop 1:

$$f_{max} = 1/(t_{pmax} + t_{cmax} + t_{su}) = 1/(24ns + 16ns + 8ns) \approx 20.83\ MHz$$

(b) $$f_{max} = 1/(t_{pmax} + t_{cmax} - t_{skew} + t_{su}) = 1/(24ns + 16ns - 5ns + 8ns) \approx 23.26\ MHz$$

(c) $$f_{max} = 1/(t_{pmax} + t_{cmax} + t_{skew} + t_{su}) = 1/(24ns + 16ns + 5ns + 8ns) \approx 18.87\ MHz$$

(d) $t_y \geq t_h - t_{cxmin} = 4ns - 2ns = 2ns$
$t_x \geq t_{cxmax} + t_{su} = 16ns + 8\ ns = 24\ ns$

X can change 2 ns after and 24 ns before the rising clock edge.

(e) From part (d), X can change 2 ns after and 24 ns before the rising clock edge of flip-flop 1. However, because the rising clock edge of flip-flop 2 is delayed 5 ns from the rising edge of flip-flop 1, then X can change -3 ns after and 29 ns before the rising clock edge of flip-flop 2. In other words, X cannot change between 29 and 3 ns before the rising clock edge of flip-flop 2.

(f) From part (d), X can change 2 ns after and 24 ns before the rising clock edge of flip-flop 1. However, because the rising clock edge of flip-flop 2 is advanced 5 ns from the rising edge of flip-flop 1, then X can change 7 ns after and 19 ns before the rising clock edge of flip-flop 2.

1.27 (a) Consider following delays:

input to first FF: $t_{su} = 20ns$
left FF to middle FF: $t_{pmax} + t_{c1max} + t_{skew1} + t_{su} = 10ns + 7ns + 0ns + 20ns = 37ns$
middle FF to right FF: $t_{pmax} + t_{c2max} - t_{skew2} + t_{su} = 10ns + 11ns - 0ns + 20ns = 41ns$
right FF to output: $t_{pmax} = 10ns$

The maximum of these is 41 ns. Therefore, the minimum clock period should be 41 ns.

(b) $t_{clk} \overset{?}{\geq} t_{pmax} + t_{c1max} + t_{su}$
$41\ ns \geq 10ns + 4\ ns + 20 = 34\ ns$

There is no setup time violation for the middle flip-flop. The setup time margin is 41 – 34 = 7 ns.

(c) $t_{pmin} + t_{c1min} \overset{?}{\geq} t_h$
$5ns + 1ns < 10ns$
There is a hold time violation for the middle flip-flop.

(d) For negative clock skew:

$$t_{skewmin} = t_h - t_{pmin} - t_{cmin} = 10ns - 5ns - 1ns = 4ns$$

To fix the hold time violation for the middle flip-flop, make $t_{skew1min}$ = 4 ns and keep $t_{skew2min}$ = 0 ns.

(e) The new worst-case delay of the path from the left flip-flop to the middle flip-flop is:

$$t_{pmax} + t_{c1max} + t_{skew1} + t_{su} = 10ns + 4ns + 4ns + 20ns = 38ns$$

However, this delay is still less than the 41 ns delay of the path from the middle flip-flop to the right flip-flop. Therefore, t_{clkmin} is still 41 ns.

1.28 **(a)** Consider following delays:

input to first FF: $t_{su} = 10ns$

left FF to middle FF: $t_{pmax} + t_{c1max} + t_{skew1} + t_{su} = 20ns + 7ns + 0ns + 10ns = 37ns$

middle FF to right FF: $t_{pmax} + t_{c2max} - t_{skew2} + t_{su} = 20ns + 11ns - 3ns + 10ns = 38ns$

right FF to output: $t_{pmax} = 20ns$

The maximum of these is 38 ns. Therefore, the minimum clock period should be 38 ns.

(b) $t_{clk} \overset{?}{\geq} t_{pmax} + t_{c1max} + + t_{skew1} + t_{su}$

$38\ ns \geq 20n + 4ns + 0ns + 10ns = 34\ ns$

There is no setup time violation for the middle flip-flop. The setup time margin is 38 – 34 = 4 ns.

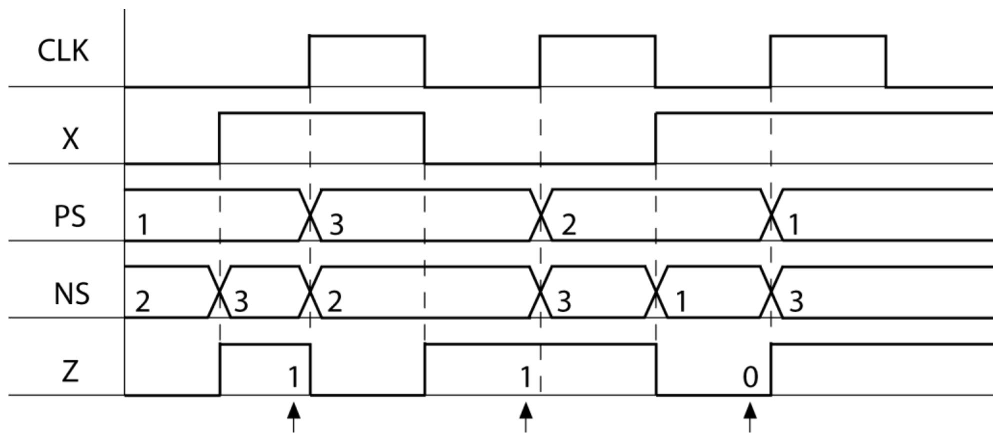
(c) $t_{pmin} + t_{c1min} \overset{?}{\geq} t_h - t_{skew1}$

There is no hold time violation for the middle flip-flop. The hold time margin is 13 – 2 = 11 ns.

(d) Because both setup and hold time requirements are met for the middle flip-flop, the clock skew delays in place are valid.

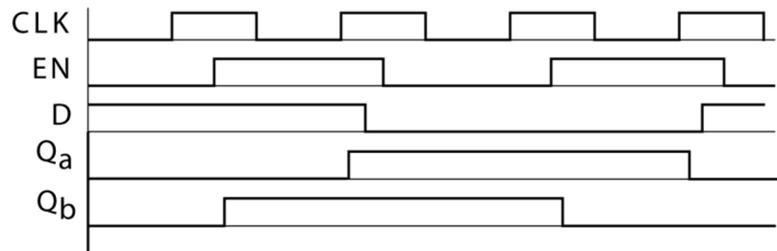
(e) The worst-delay path is still from the middle flip-flop to the right flip-flop. Therefore, t_{clkmin} is still 38 ns.

1.29

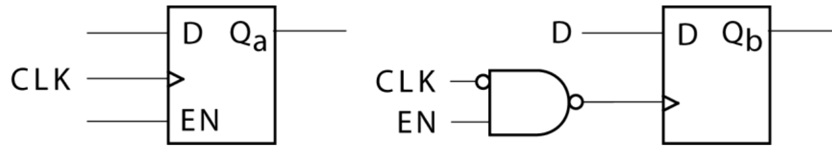


1.30 **(a)** No

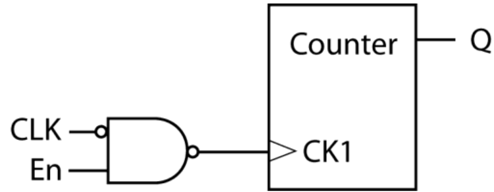
(b)



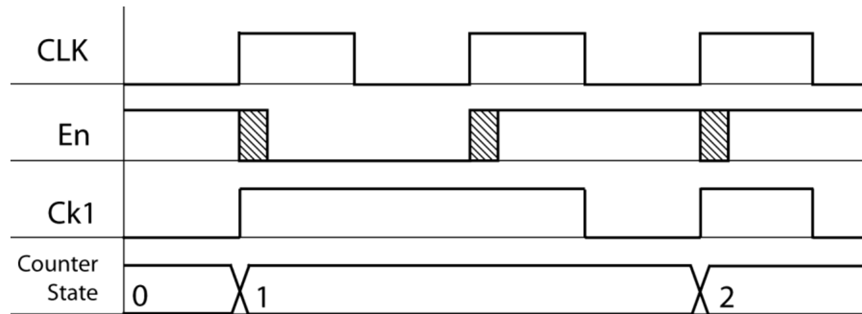
(c)



1.31 (a)



(b)



The clock input to the NAND gate in (a) is inverted. After the rising edge of the clock, this input to the NAND gate is a '0', so CK1 will remain a constant '1', regardless of any changes that may occur in the EN input of the gate due to transients.

1.32 $Eni = 0$ $Ena = 0$ $Enb = 0$ $Enc = 1$ $Lda = 1$ $Ldb = 1$ $Ldc = 0$

Chapter 2: Introduction to Verilog®

2.1 **(a)** HDL – Hardware Description Language
FPGA – Field Programmable Logic Array

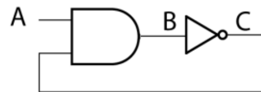
(b) Verilog has statements that execute concurrently since it must model real hardware in which the components are all in operation at the same time.

(c) A hardware description language allows a digital system to be designed and debugged at a higher level of abstraction than schematic capture with gates, flip-flops, and standard MSI building blocks. The details of the gates and flip-flops do not need to be handled during early phases of design. Designs are more portable when low-level library-specific details are not included in the model. HDLs allow the creation of such portable high-level behavioral models.

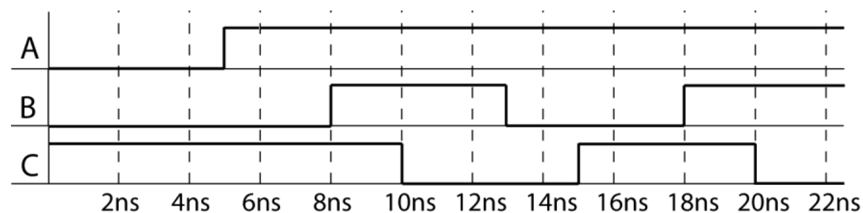
2.2 **(a)** Legal: `A_123`, `_A123`, `c1__c2`, `and1`; Illegal: `123A` (starts with number), `$A123_` (starts with $), `and` (reserved word).

(b) None of the Verilog identifiers are equivalent since Verilog is case sensitive.

2.3 **(a)**



(b)



2.4

```
module Gate(A, B, C, D, Z);
  input A, B, C, D;
  output Z;

  wire E, F, G, H, I;

  assign #5 H = A & B & C;
  assign #5 E = H | D;
  assign #5 G = ~(B | C);
  assign #5 F = ~(G & A);
  assign #2 I = ~F;
  assign #5 Z = E ^ I;
endmodule
```

2.5 **(a)**

```
module one_bit_sub(x, y, bin, diff, bout);
  input x, y, bin;
  output diff, bout;

  assign diff = x ^ y ^ bin;
  assign bout = (~x & bin) | (~x & y) | (bin & y);
endmodule
```

(b)

```
module four_bit_sub(a, b, bin, d, bout);
  input[3:0] a, b;
```

```
      input bin;
      output[3:0] d;
      output bout;

      wire[3:0] bo;

      one_bit_sub s1(a[0], b[0], bin, d[0], bo[1]);
      one_bit_sub s2(a[1], b[1], bo[1], d[1], bo[2]);
      one_bit_sub s3(a[2], b[2], bo[2], d[2], bo[3]);
      one_bit_sub s4(a[3], b[3], bo[3], d[3], bout);
    endmodule
```

2.6 **(a)**

```
module circuit(A, B, C, D, G);
  input A, B, C, D;
  output G;

  wire E, F;

  assign E = A & B;
  assign F = E | C;
  assign G = D & F;
endmodule
```

(b)

```
module circuit(A, B, C, D, G);
  input A, B, C, D;
  output reg G;

  reg E, F;

  initial begin
    E <= 0;
    F <= 0;
    G <= 0;
  end

  always @(*)
  begin
    E <= A & B;
    F <= E | C;
    G <= F & D;
  end
endmodule
```

2.7 **(a)** *A* changes to 1 at 25 ns, *B* changes to 1 at 25 ns, *C* change to 1 at 35 ns
(b) *A* changes to 1 at 25 ns, *B* changes to 1 at 20 + Δ ns, *C* does not change

2.8 **(a)** A falling-edge triggered D flip-flop with asynchronous active high clear and set
(b) Q = '0', because Clr = 1 has priority.

2.9

```
module SR_Latch(S, R, Q, Qn);
  input S, R;
  output reg Q;
  output Qn;

  initial begin
    Q <= 0;
  end
```

```
  always @(S, R)
  begin
    if(S == 1'b1)
      Q <= 1'b1;
    if(R == 1'b1)
      Q <= 1'b0;
  end

  assign Qn = ~Q;
endmodule
```

2.10
```
module MNFF(M, N, CLK, CLRn, Q, Qn);
  input M, N, CLK, CLRn;
  output reg Q;
  output Qn;

  initial begin
    Q <= 0;
  end

  always @(CLK, CLRn)
  begin
    if(CLRn == 1'b0)
    begin
      Q <= 0;
    end
    else if(CLK == 0)
    begin
      if(M == 0 && N == 0)
         Q <= ~Q;
      else if(M == 0 && N == 1)
         Q <= 1;
      else if(M == 1 && N == 0)
         Q <= 0;
      else if(M == 1 && N == 1)
         Q <= Q;
    end
  end

  assign Qn = ~Q;
endmodule
```

2.11
```
module DDFF(R, S, D, Clk, Q);
  input R, S, D, Clk;
  output reg Q;

  initial begin
    Q <= 0;
  end

  always @(Clk, R, S)
  begin
    if(R == 1'b0)
      Q <= 0;
    else if(S == 1'b0)
      Q <= 1;
    else
      Q <= D;
  end
endmodule
```

2.12 **(a)**

```
module ITFF(I0, I1, T, R, Q, QN);
  input I0, I1, T, R;
  output reg Q;
  output QN;

  initial begin
    Q <= 0;
  end

  always @(T, R)
  begin
    if(R == 1'b1)
      #5 Q <= 0;
    else begin
      if((I0 == 1'b1 && T == 1'b1) || (I1 == 1'b1 && T == 1'b0))
        #8 Q <= QN;
    end
  end

  assign QN = ~Q;
endmodule
```

(b)

```
add list *
add wave *
force T 0 0, 1 100 -repeat 200
force I1 0 0, 1 50, 0 450
force I0 0 0, 1 450
run 750 ns
```

2.13

ns	Δ	a	b	c	d	e
10	+0	0	0	0	0	0
20	+0	0	0	0	0	1
20	+1	0	7	0	0	1
25	+0	1	7	0	0	1
35	+0	5	7	0	0	1

2.14

ns	Δ	a	b	c	d	e
10	+0	0	0	0	0	0
20	+0	0	0	0	0	1
20	+1	0	7	0	0	1
25	+0	1	7	0	0	1
35	+0	5	7	0	0	1

2.15

i. 5’b10101
ii. 8’b11010101
iii. 3’b100

2.16

i. 4’b0000
ii. 8’b00001010
iii. 7’b0000101

2.17

i. 8’h0D
ii. 8’hFD
iii. 8’h50
iv. 8’h50
v. 8’h0D
vi. 8’hFD
vii. 8’h50
viii. 8’h50

2.18

(a)
A >> 4 == 2’h0C
A >>> 4 == 2’hFC
A << 4 == 8’h70
A <<< 4 == 8’h70

(b)
A >> 4 == 2’h0C
A >>> 4 == 2’h0C
A << 4 == 8’h70
A <<< 4 == 8’h70

(c)
A >> 4 == 2’h0C
A >>> 4 == 2’h0C
A << 4 == 8’h70
A <<< 4 == 8’h70

(d)
A >> 4 == 8’h0C
A >>> 4 == 8’h0C
A << 4 == 8’h70
A <<< 4 == 8’h70

(e)
A >> 4 == 8’h0C
A >>> 4 == 8’h0C
A << 4 == 8’h70
A <<< 4 == 8’h70

(f)
A >> 4 == 8’h0C
A >>> 4 == 8’hFC
A << 4 == 8’h70
A <<< 4 == 8’h70

(g)
A >> 4 == 8’h0C
A >>> 4 == 8’hFC
A << 4 == 8’h70
A <<< 4 == 8’h70

(h)
A >> 4 == 32’h0FFFFFFC
A >>> 4 == 32’hFFFFFFFC
A << 4 == 32’hFFFFFC70
A <<< 4 == 32’hFFFFFC70

2.19

i. 8’h0D
ii. 8’h0D
iii. 8’h50
iv. 8’h50
v. 8’h0D

vi. 8'h0D
vii. 8'h50
viii. 8'h50

2.20
i. 8'h0D
ii. 8'h0D
iii. 8'h50
iv. 8'h50
v. 8'h0D
vi. 8'h0D
vii. 8'h50
viii. 8'h50

2.21 The synthesized hardware is a 4-bit shift register.

2.22 The synthesized hardware is a single flip-flop.

2.23 Both modules are synthesized to 4-bit shift registers. There are no differences between the two shift registers.

2.24 **(a)** D1 = 5, D2 = 1. The values of D1 and D2 swap.

(b) D1 = 1, D2 = 1. The values of D1 and D2 do not swap.

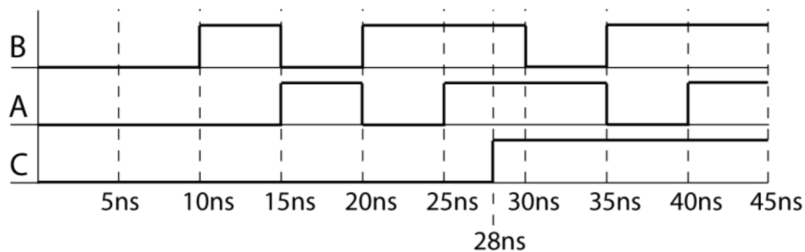
(c) iii

2.25 a; *y* must be in the sensitivity list, otherwise *sum* and *carry* will not update from changes to *y*

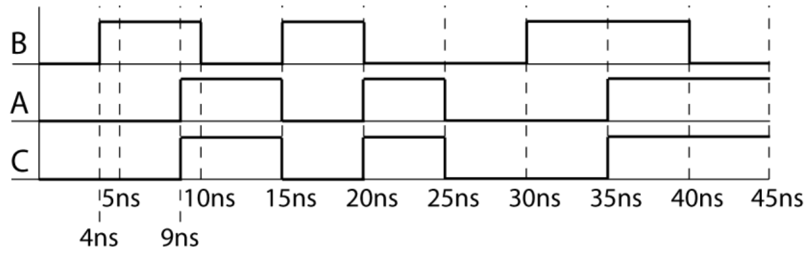
2.26

ns	Δ	a	b	c	d	e	f
0	+0	0	0	0	0	0	0
5	+0	0	0	0	1	0	0
5	+1	1	0	0	1	0	0
5	+2	1	1	1	1	0	0
6	+0	1	1	1	0	0	0
10	+0	1	1	1	0	1	0
10	+1	0	1	1	0	1	0
10	+2	0	0	0	0	1	0

2.27



2.28



2.29 **(a)**

ns	a	b	c
0	1	0	0
4	1	1	0
5	1	1	1
10	1	0	1
15	1	1	0
20	1	0	1
25	1	0	0
30	1	1	0
35	1	1	1
40	1	0	1
45	1	0	0

(b)

ns	a	b	c
0	1	0	0
4	1	1	0
5	1	1	1
10	1	0	1
15	1	1	0
20	1	0	1
25	1	0	0
30	1	1	0
35	1	1	1
40	1	0	1
45	1	0	0

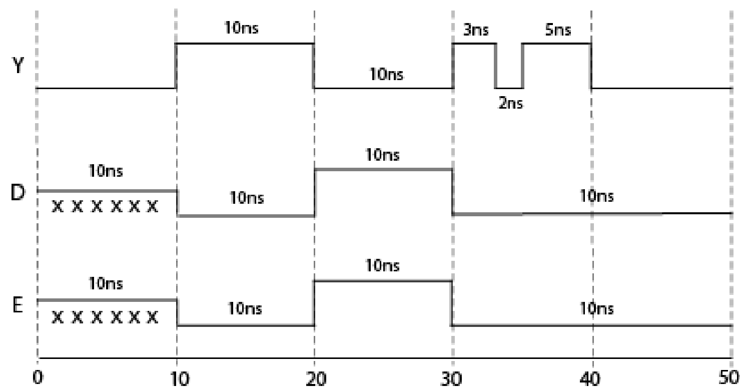
(c)

ns	a	b	c
0	1	0	0
4	1	1	0
9	1	1	1
10	1	0	1
15	1	1	0
20	1	0	1
25	1	0	0
30	1	1	0
35	1	1	1
40	1	0	1
45	1	0	0

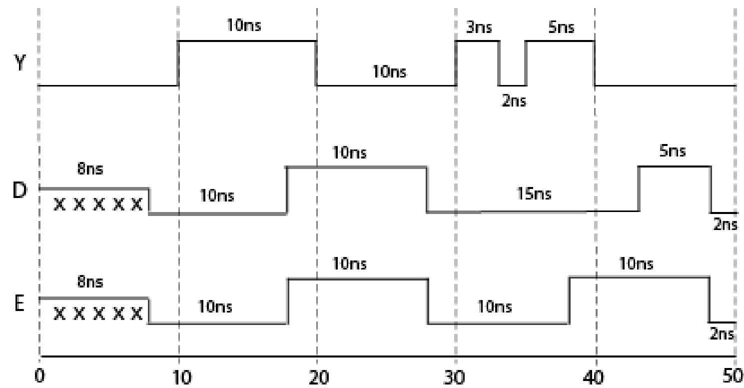
(d)

ns	a	b	c
0	1	0	0
4	1	1	0
10	1	0	0
15	1	1	0
20	1	0	1
25	1	0	0
30	1	1	0
35	1	1	1
40	1	0	1
45	1	0	0

2.30



2.31

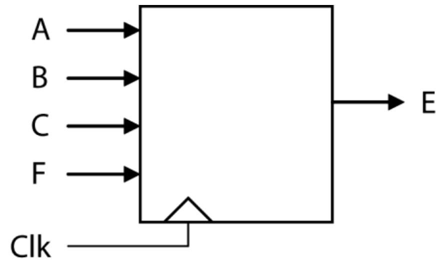


2.32

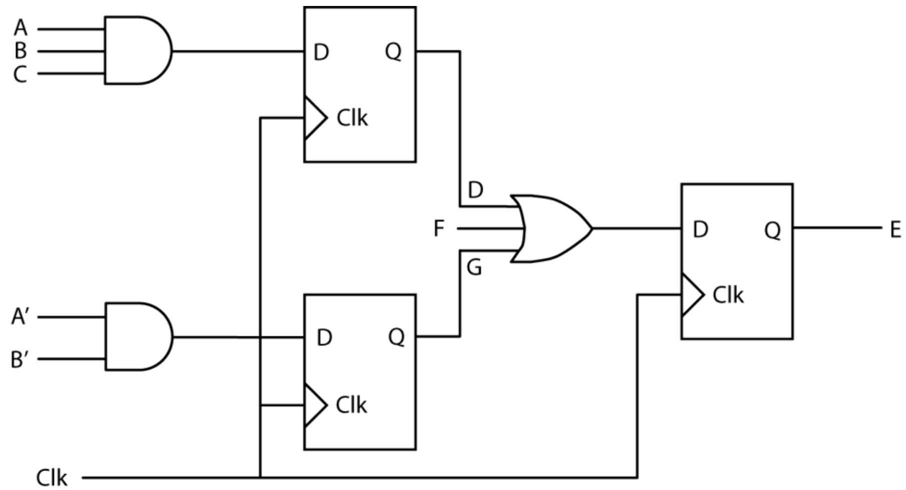
ns	Δ	a	b	c	d
4	+0	0	0	0	0
5	+0	1	0	0	0
10	+0	1	1	0	0
10	+1	0	1	0	0
11	+0	0	1	0	1
12	+0	0	1	1	1
15	+0	0	0	1	1

2.33 **(a)** 6’b111011
(b) 3’b001
(c) 3’b001
(d) 1’b0
(e) 3'b111

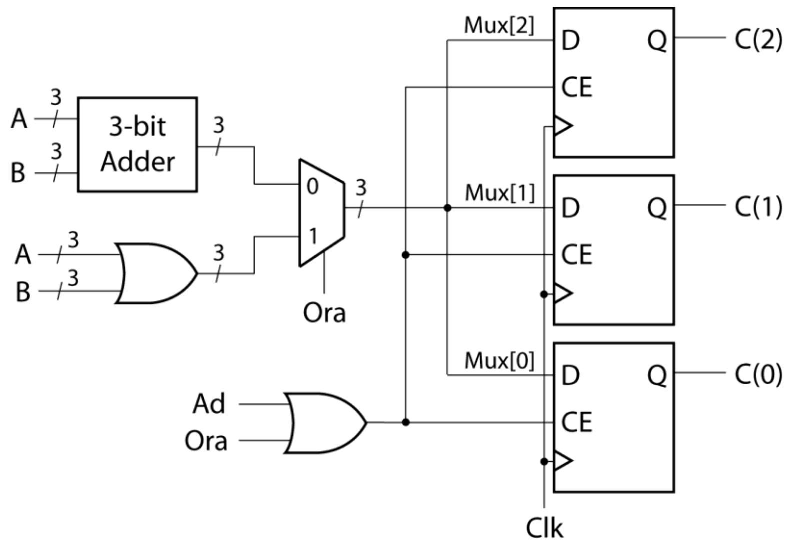
2.34 **(a)**



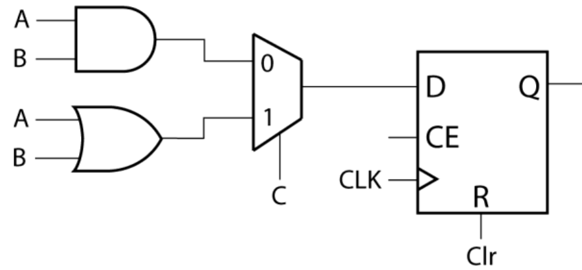
(b)



2.35



2.36



Clr is asynchronous, whereas C affects a synchronous input to the D flip-flop.

2.37 **(a)**

```
assign #10 F = (C == 0)? ((D==0)? ~A: B):((D==0)? ~B: 0);
```

(b)

```
always @(*)
begin
  if(C==0 && D==0)
    #10 F = ~A;
  else if(C==0 && D==1)
    #10 F = B;
  else if(C==1 && D==0)
    #10 F = ~B;
  else
    #10 F = 0;
end
```

(c)

```
always @(*)
begin
   case(sel)
     0: #10 F = ~A;
     1: #10 F = B;
     2: #10 F = ~B;
     3: #10 F = 0;
   endcase
end
```

2.38 **(a)**

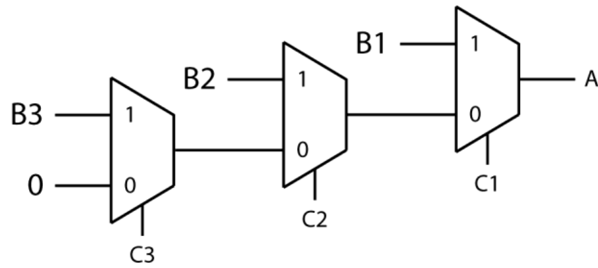
module 1:

```
always @(C, B1, B2, B3)
begin
  if (C == 1)
    A <= B1;
  else if (C == 2)
    A <= B2;
  else if (C == 3)
    A <= B3;
  else
    A <= 0;
 end
```

module 2:

```
assign A = (C==1)? B1 : ((C==2)? B2 : ((C==3)? B3 : 0 ));
```

(b)



2.39 **(a)**

```
module SR_Latch(S, R, P, Q);
  input S, R;
  output P, Q;

  assign Q = (S)? 1 : ((R)? 0 : Q);
  assign P = ~Q;

endmodule
```

(b)

```
assign Q = S | (~R & Q);
assign P = ~Q;
```

(c)

```
assign Q = ~(R | P);
assign P = ~(S | Q);
```

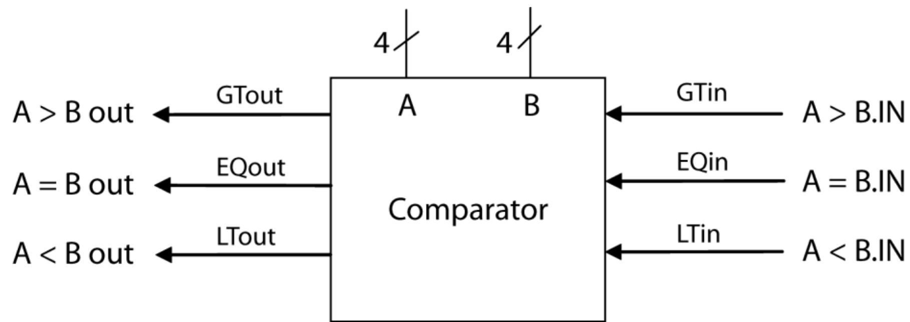
2.40

i. 8'b00010100
ii. 8'b00010100
iii. 8'b11111100
iv. 8'b11111100
v. 8'b00010100
vi. 8'b00010100

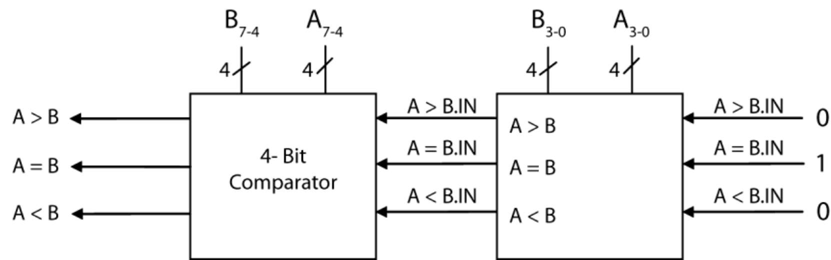
2.41

i. 32'hFFFFFF5B
ii. 32'hFFFFFF5B
iii. 32'h0000015B
iv. 32'h0000015B
v. 32'h0000005B
vi. 32'h0000005B

2.42 **(a)**



(b)



(c)

```
module comp4bit(A, B, EQin, GTin, LTin, EQout, GTout, LTout);
  input[3:0] A, B;
  input EQin, GTin, LTin;
  output reg EQout, GTout, LTout;

  initial begin
    EQout = 0;
    GTout = 0;
    LTout = 0;
  end

  always @(A, B, EQin, GTin, LTin)
  begin
    if(A > B) begin
      EQout <= 0;
      GTout <= 1;
      LTout <= 0;
    end
    else if(A < B) begin
       EQout <= 0;
       GTout <= 0;
       LTout <= 1;
    end
    else if(GTin == 1) begin
       EQout <= 0;
       GTout <= 1;
       LTout <= 0;
    end
    else if(LTin == 1) begin
       EQout <= 0;
       GTout <= 0;
       LTout <= 1;
    end
    else begin
       EQout <= 1;
       GTout <= 0;
       LTout <= 0;
    end
  end
endmodule
```

(d)

```
module comp8bit(A, B, EQi, GTi, LTi, EQ, GT, LT);
  input[7:0] A, B;
  input EQi, GTi, LTi;
  output EQ, GT, LT;

  wire LowEQ, LowGT, LowLT;

  comp4bit C1(A[3:0], B[3:0], EQi, GTi, LTi, LowEQ, LowGT, LowLT);
  comp4bit C2(A[7:4], B[7:4], LowEQ, LowGT, LowLT, EQ, GT, LT);
endmodule
```

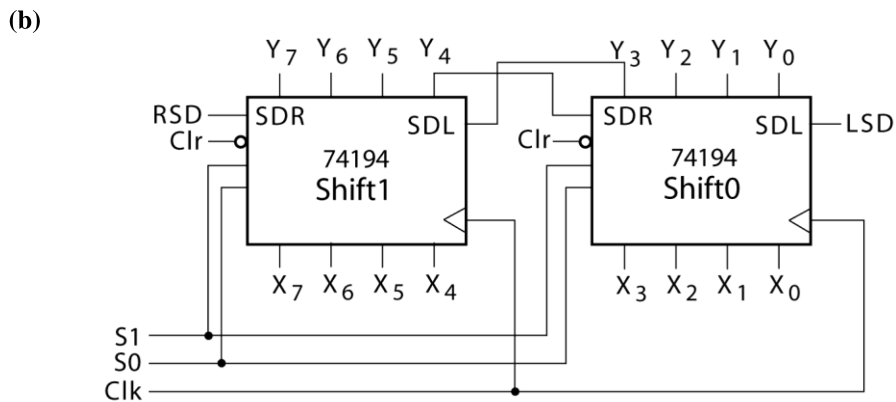
2.43

```
module shift_reg(SI, EN, CK, SO);
  input SI, EN, CK;
  output SO;
  reg[15:0] register;

  initial begin
    register <= 0;
  end

  always @(posedge CK)
  begin
    if(EN == 1)
      register <= {SI, register[15:1]};
  end

  assign SO = register[0];
endmodule
```

2.44 **(a)**

```
module shift74194(D, S, SDR, SDL, CLRb, CLK, Q);
  input[3:0] D;
  input[1:0] S;
  input SDR, SDL, CLRb, CLK;
  output reg [3:0] Q;

  initial begin
    Q <= 0;
  end

  always @(CLK, CLRb)
  begin
    if(CLRb == 0)
      Q <= 4'b0000;
    else if(CLK == 1)
    begin
      case(S)
      0: Q <= Q;
      1: Q <= {Q[2:0],SDL};
      2: Q <= {SDR, Q[3:1]};
      3: Q <= D;
      endcase
    end
  end
endmodule
```

(b)



```
module bit8shift(X, S, RSD, LSD, CLRb, CLK, Y);
  input[7:0] X;
  input[1:0] S;
  input RSD, LSD, CLRb, CLK;
  output[7:0] Y;

  shift74194 S1(X[3:0], S, Y[4], LSD, CLRb, CLK, Y[3:0]);
  shift74194 S2(X[7:4], S, RSD, Y[3], CLRb, CLK, Y[7:4]);
endmodule
```

2.45 **(a)**

```
module Counter(D, CLK, CLR, ENT, ENP, UP, LOAD, Q, CO);
  input[3:0] D;
  input CLK, CLR, ENT, ENP, UP, LOAD;
  output reg[3:0] Q;
  output CO;

  initial begin
    Q = 0;
  end

  assign CO = ((ENT == 1) && ((UP == 1 && (Q == 4'b1001)) ||
              (UP == 0 && (Q == 4'b0000))));

  always @(CLK, CLR)
  begin
    if(CLR == 0)
      Q <= 0;
    else if(CLK == 1)
    begin
      if(LOAD == 0)
        Q <= D;
      else if(ENT == 1 && ENP == 1 && UP == 0) begin
        if(Q == 0)
          Q <= 4'b1001;
        else
          Q <= Q - 1;
      end
      else if(ENT == 1 && ENP == 1 && UP == 1) begin
        if(Q == 4'b1001)
          Q <= 0;
        else
          Q <= Q + 1;
      end
    end
  end
endmodule
```

(b)

```
module Century_Counter(Din1, Din2, CLK, CLR, ENT, ENP, UP, LOAD,
Count, CO);
  input [3:0] Din1, Din2;
  input CLK, CLR, ENT, ENP, UP, LOAD;
  output [7:0] Count;
  output CO;

  wire [3:0] Qout1, Qout2;
  wire Carry1, Carry2;

  Counter ct1(Din1, CLK, CLR, ENT, ENP, UP, LOAD, Qout1, Carry1);
  Counter ct2(Din2, CLK, CLR, ENT, Carry1, UP, LOAD, Qout2,
Carry2);
```

```
  assign Count = {Qout2, Qout1};
  assign CO = Carry2;
endmodule
```

The block diagram is similar to Figure 2-45 with an "Up" input added to each counter.

(c)

```
add wave *
force Din2 4'b1001
force Din1 4'b1000
force CLK 0 0 ns, 1 50 ns -repeat 100 ns
force CLR 1 0 ns, 0 1000 ns
force LOAD 0 0 ns, 1 100 ns
force ENT 1 0 ns, 0 400 ns, 1 600 ns
force ENP 1
force UP 1 0 ns, 0 500 ns
run 1200 ns
```

2.46 Students should look on the web for 74HC192 data sheet. CLR is active high. LOADB is active low. Counting up happens when UP has a rising edge and DOWN=1. Counting down happens when DOWN has a rising edge and UP=1. CARRY indicates terminal count in the up direction , i.e. 9. BORROW indicates terminal count in the down direction, i.e. 0.

Operating Mode	CLR	LOADB	UP	DOWN	D	Q	Borrow	Carry
Clear	1	X	X	0	XXXX	0000	0	1
	1	X	X	1	XXXX	0000	1	1
Load	0	0	X	X	XXXX	Q = D	1*	1*
Count Up	0	1	↑	1	XXXX	Q = Q + 1	1	1**
Count Down	0	1	1	↑	XXXX	Q = Q – 1	1**	1

* when loading, if the input is 0 and down = 0, borrow will be 0. If the input is 9 and up = 0, carry will be 0

** Borrow = 0 when the counter is in state 0 and down = 0. Carry = 0 when the counter is in state 9 and up = 0.

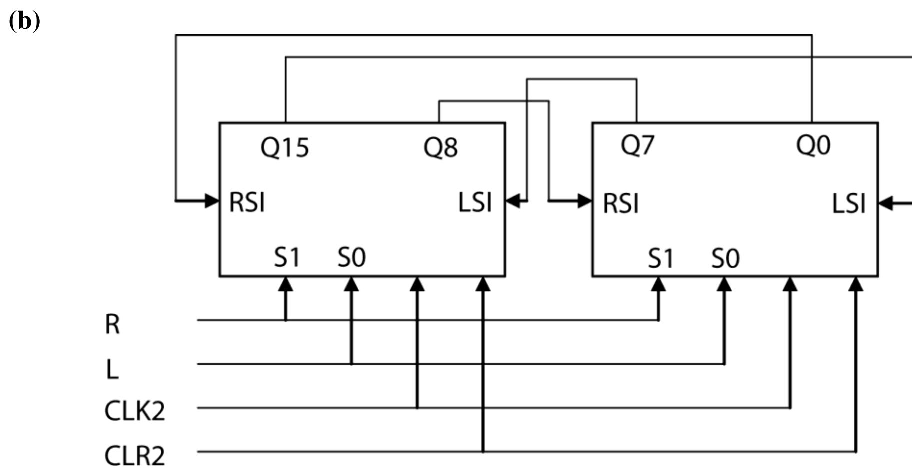
```
module count74HC192(DOWN, UP, CLR, LOADB, BORROW, CARRY, D, Q);
  input DOWN, UP, CLR, LOADB;
  input[3:0] D;
  output BORROW, CARRY;
  output reg[3:0] Q;

  initial begin
    Q = 0;
  end

  always @(DOWN, UP, CLR, LOADB)
  begin
    if(CLR == 1)
      Q <= 0;
    else if(LOADB == 0)
      Q <= D;
    else if(DOWN == 1) begin
      @(posedge UP)
      if(Q == 4'b1001)
        Q <= 0;
      else
        Q <= Q + 1;
    end
    else if(UP == 1) begin
      @(posedge DOWN)
```

```
      if(Q == 0)
        Q <= 4'b1001;
      else
        Q <= Q - 1;
    end
  end

  assign BORROW = (DOWN == 0 && Q == 0)? 0 : 1;
  assign CARRY = (UP == 0 && Q == 4'b1001)? 0 : 1;
endmodule
```

2.47 **(a)**

```
module shift8(Q, D, CLR, CLK, S0, S1, LSI, RSI);
  input[7:0] D;
  output reg[7:0] Q;
  input CLR, CLK, S0, S1, LSI, RSI;

  initial begin
    Q = 0;
  end

  always @(CLK, CLR)
  begin
    if(CLR == 1)
      Q <= 0;
    else if(CLK == 1) begin
      if(S0 == 1 && S1 ==1)
        Q <= D;
      else if(S0 == 0 && S1 == 1)
        Q <= {RSI, Q[7:1]};
      else if(S0 == 1 && S1 == 0)
        Q <= {Q[6:0], LSI};
      else
        Q <= Q;
    end
  end

endmodule
```

(b)



Note: D is not shown in the diagram.

(c)

```
module shiftreg(QQ, DD, CLK2, CLR2, L, R);
  input[15:0] DD;
  input CLK2, CLR2, L, R;
```

```
    output[15:0] QQ;

    shift8 SR1(QQ[15:8], DD[15:8], CLR2, CLK2, L, R, QQ[7], QQ[0]);
    shift8 SR2(QQ[7:0], DD[7:0], CLR2, CLK2, L, R, QQ[15], QQ[8]);
  endmodule
```

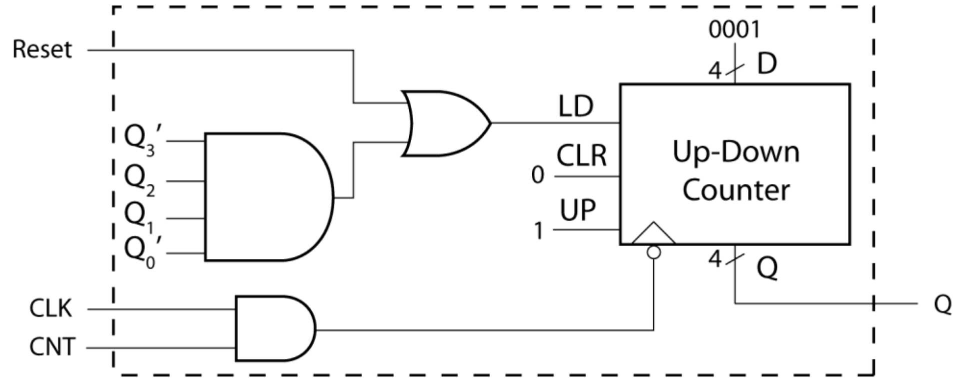
2.48

```
module countQ1(clk, Ld8, Enable, S5, Q);
  input clk, Ld8, Enable;
  output S5;
  output[3:0] Q;

  reg[3:0] Qint;

  initial begin
    Qint = 0;
  end

  always @(posedge clk)
  begin
    if(Ld8 == 1)
      Qint <= 4'b1000;
    else if(Enable == 1)
    begin
      if(Qint == 4'b0011)
        Qint <= 4'b1000;
      else
        Qint <= Qint - 1;
    end
  end

  assign S5 = (Qint == 4'b0101)? 1 : 0;
  assign Q = Qint;
endmodule
```

2.49 **(a)**

```
module up_down(CLK, CLR, LD, UP, D, Q);
  input CLK, CLR, LD, UP;
  input[3:0] D;
  output reg[3:0] Q;

  initial begin
    Q = 0;
  end

  always @(negedge CLK)
  begin
    if(CLR == 1)
      Q <= 4'b0000;
    else if(LD == 1)
      Q <= D;
    else if(UP == 1)
      Q <= Q + 1;
    else
      Q <= Q - 1;
  end
endmodule
```

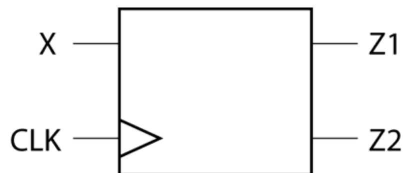
(b)



(c)

```
module modulo6(CLK, Reset, CNT, Q);
  input CLK, Reset, CNT;
  output[3:0] Q;
  wire load, clock;

  assign load = Reset | (~Q[0] & Q[1] & Q[2] & ~Q[3]);
  assign clock = CLK & CNT;

  up_down U1(clock, 1'b0, load, 1'b1, 4'b0001, Q);
endmodule
```

2.50 (a)



(b)

Present State	Next State X = 0	Next State X = 1	X = 0 Z1	X = 0 Z2	X = 1 Z1	X = 1 Z2
S0	S0	S1	1	0	0	0
S1	S1	S2	0	1	0	1
S2	S2	S3	0	1	0	1
S3	S0	S1	0	0	1	0

2.51 The following solutions utilize the solution for 1.13.

(a)

```
module P2_51a(X, CLK, S, V);
  input X, CLK;
  output S, V;

  reg [2:0] StateTable0 [5:0];
  reg [2:0] StateTable1 [5:0];
  reg [1:0] OutTable0 [5:0];
  reg [1:0] OutTable1 [5:0];

  reg [2:0] State;
  wire [2:0] NextState;
```

```
  initial begin
    StateTable0[0] <= 1; StateTable1[0] <= 1;
    StateTable0[1] <= 2; StateTable1[1] <= 4;
    StateTable0[2] <= 3; StateTable1[2] <= 3;
    StateTable0[3] <= 0; StateTable1[3] <= 0;
    StateTable0[4] <= 3; StateTable1[4] <= 5;
    StateTable0[5] <= 0; StateTable1[5] <= 0;
    OutTable0[0] <= 2'b00; OutTable1[0] <= 2'b10;
    OutTable0[1] <= 2'b10; OutTable1[1] <= 2'b00;
    OutTable0[2] <= 2'b00; OutTable1[2] <= 2'b10;
    OutTable0[3] <= 2'b00; OutTable1[3] <= 2'b10;
    OutTable0[4] <= 2'b10; OutTable1[4] <= 2'b00;
    OutTable0[5] <= 2'b10; OutTable1[5] <= 2'b01;
    State <= 0;
  end

  assign NextState = (X==0)? StateTable0[State] :
StateTable1[State];
  assign S = (X==0)? OutTable0[State][1] : OutTable1[State][1];
  assign V = (X==0)? OutTable0[State][0] : OutTable1[State][0];

  always @(negedge CLK)
  begin
    State <= NextState;
  end
endmodule
```

example simulation commands:

```
add wave *
force CLK 1 0 ns, 0 10 ns -repeat 20 ns
force X 1 0 ns, 0 15 ns, 1 35 ns, 0 75 ns, 1 95 ns, 0 175 ns
run 240 ns
```

(b)

```
module P2_51b(X, CLK, S, V);
  input X, CLK;
  output S, V;

  reg Q1, Q2, Q3;

  initial begin
    Q1 <= 0;
    Q2 <= 0;
    Q3 <= 0;
  end

  always @(negedge CLK)
  begin
    Q1 <= (~Q1 & Q3);
    Q2 <= (~Q2 & ~Q3) | (X & ~Q1 & Q2);
    Q3 <= (~Q1 & Q3) | (Q2 & ~Q3);
  end

  assign S = (X & ~Q2) | (~X & Q2);
  assign V = (X & Q1 & Q2);
endmodule
```

Read each set of outputs after 1/4 clock period before the falling edge of the clock but no later than the falling edge of the clock.

(c)

```
module P2_51c(X, CLK, S, V);
  input X, CLK;
  output S, V;

  wire Q1, Q2, Q3;
  wire XN, Q1N, Q2N, Q3N;
  wire D1, D2, D3;
  wire A1, A2, A3, A4, A5, A6;

  Inverter I1(X, XN);
  And2 G1(Q1N, Q3, D1);
  And2 G2(Q2N, Q3N, A1);
  And3 G3(X, Q1N, Q2, A2);
  Or2 G4(A1, A2, D2);
  And2 G5(Q1N, Q3, A3);
  And2 G6(Q2, Q3N, A4);
  Or2 G7(A3, A4, D3);
  And2 G8(X, Q2N, A5);
  And2 G9(XN, Q2, A6);
  Or2 G10(A5, A6, S);
  And3 G11(X, Q1, Q2, V);
  DFF DFF1(D1, CLK, Q1, Q1N);
  DFF DFF2(D2, CLK, Q2, Q2N);
  DFF DFF3(D3, CLK, Q3, Q3N);
endmodule
```

See Section 2.15 for the definition of the `DFF` component. The `And3`, `And2`, `Or2`, and `Inverter` components are all similar to the `Nand3` component given on pages 109-110 (section 2.15).

2.52 The following solutions utilize the solution for 1.14.

(a)

```
module P2_52a(X, CLK, D, B);
  input X, CLK;
  output reg D, B;
  reg[2:0] State, NextState;

  initial begin
    State <= 0;
    NextState <= 0;
  end

  always @(State, X)
  begin
    case(State)
    0:begin
      if(X == 0)
      begin
        D <= 0;
        B <= 0;
        NextState <= 1;
      end
      else begin
        D <= 1;
        B <= 0;
        NextState <= 1;
      end
    end
    1:begin
      if(X == 0)
      begin
        D <= 1;
```

```
        B <= 0;
        NextState <= 2;
      end
      else begin
        D <= 0;
        B <= 0;
        NextState <= 3;
      end
    end
    2: begin
      if(X == 0)
      begin
        D <= 1;
        B <= 0;
        NextState <= 4;
      end
      else begin
        D <= 0;
        B <= 0;
        NextState <= 5;
      end
    end
    3: begin
      if(X == 0)
      begin
        D <= 0;
        B <= 0;
        NextState <= 5;
      end
      else begin
        D <= 1;
        B <= 0;
        NextState <= 5;
      end
    end
    4: begin
      if(X == 0)
      begin
        D <= 1;
        B <= 1;
        NextState <= 0;
      end
      else begin
        D <= 0;
        B <= 0;
        NextState <= 0;
      end
    end
    5: begin
      if(X == 0)
      begin
        D <= 0;
        B <= 0;
        NextState <= 0;
      end
      else begin
        D <= 1;
        B <= 0;
        NextState <= 0;
      end
    end
    endcase
  end
```

```
  always @(negedge CLK)
  begin
    State <= NextState;
  end
endmodule
```

example simulation commands:

```
add wave *
force CLK 1 0 ns, 0 10 ns -repeat 20 ns
force X 1 0 ns, 0 15 ns, 1 35 ns, 0 75 ns, 1 95 ns, 0 175 ns
run 240 ns
```

(b)

```
module P2_52b(X, CLK, D, B);
  input X, CLK;
  output D, B;

  reg Q1, Q2, Q3;

  initial begin
    Q1 <= 0;
    Q2 <= 0;
    Q3 <= 0;
  end

  always @(negedge CLK)
  begin
    Q1 <= (~Q1 & ~Q3) | (~X & Q1 & ~Q2);
    Q2 <= (~Q2 & Q3);
    Q3 <= ~Q2 & (Q3 | Q1);
  end

  assign D = (~X & Q1) | (X & ~Q1 & Q3);
  assign B = ~X & Q1 & Q2;
endmodule
```

Read each set of outputs after 3/4 clock period before the falling edge of the clock but no later than the falling edge of the clock.

(c)

```
module PC_52c(X, CLK, D, B);
  input X, CLK;
  output D, B;
  wire A1, A2, A3;
  wire Q1, Q2, Q3;
  wire Q1N, Q2N, Q3N, XN, One;
  parameter I = 1;

  Inverter I1(X, XN);
  Nand2 G1(XN, Q2N, A1);
  JKFF FF1(I, I, Q3N, A1, CLK, Q1, Q1N);
  JKFF FF2(I, I, Q3, I, CLK, Q2, Q2N);
  JKFF FF3(I, I, Q1, Q2, CLK, Q3, Q3N);
  Nand2 G2(XN, Q1, A2);
  Nand3 G3(X, Q1N, Q3, A3);
  Nand2 G4(A2, A3, D);
  And3 G5(XN, Q1, Q2, B);
endmodule
```

The `Nand2`, `And3`, and `Inverter` components are similar to the `Nand3` component in Section 2.15. The `JKFF` component is similar to the `DFF` component in Section 2.15.

2.53

```
module moore_mach(X1, X2, Clk, Z);
  input X1, X2;
  input Clk;
  output Z;

  reg[1:0] state;

  initial begin
    state <= 1;
  end

  always @(negedge Clk)
  begin
    case(state)
    1: begin
      if((X1 ^ X2) == 1)begin
        #10 state <= 2;
      end
    end
    2: begin
      if(X2 == 1)begin
        #10 state <= 1;
      end
    end
    default: begin
    end
    endcase
  end

  assign #10 Z = state[1];
endmodule
```

2.54

```
module P_54(x1, x2, clk, z1, z2);
  input x1, x2, clk;
  output z1, z2;

  reg[1:0] state, next_state;

  initial begin
    state <= 1;
    next_state <= 1;
  end

  always @(state, x1, x2)
  begin
    case(state)
    1: begin
      if({x1,x2} == 2'b00)
        #10 next_state <= 3;
      else if({x1,x2} == 2'b01)
        #10 next_state <= 2;
      else
        #10 next_state <= 1;
    end
    2: begin
      if({x1,x2} == 2'b00)
        #10 next_state <= 2;
      else if({x1,x2} == 2'b01)
        #10 next_state <= 1;
      else
        #10 next_state <= 3;
```

```
    end
    3: begin
      if({x1,x2} == 2'b00)
        #10 next_state <= 1;
      else if({x1,x2} == 2'b01)
        #10 next_state <= 2;
      else
        #10 next_state <= 3;
    end
    endcase
  end

  always @(negedge clk)
  begin
    #5 state <= next_state;
  end

  assign #10 z1 = (state == 2)? 1: 0;
  assign #10 z2 = (state == 3)? 1: 0;
endmodule
```

2.55 **(a)** *nextstate* is not always assigned a new value in the conditional statements, i.e. else clauses are not specified. Therefore, a latch will be created to hold *nextstate* to its old value.

(b) The latch output would have the most recent value of *nextstate*.

(c)

```
always @(state, X)
begin
  case(state)
  0: begin
    if(X == 1)
      nextstate <= 1;
    else
      nextstate <= 0;
  end
  1: begin
    if(X == 0)
      nextstate <= 2;
    else
      nextstate <= 1;
  end
  2: begin
    if(X == 1)
      nextstate <= 0;
    else
      nextstate <= 2;
  end
  endcase
end
```

(d) Yes, unconditionally set *nextstate* to a default value at the beginning of the always block

2.56 The nonblocking assignments must be changed to blocking assignments. Otherwise *sel* will not update for current use. *sel* updates only at the end of the process so the case statement will get the wrong value.

2.57

ns	Δ	A	B	D
0	+0	0	0	0
5	+0	1	0	0
15	+0	1	0	1
15	+1	1	1	1
25	+0	1	1	0
25	+1	1	0	0
35	+0	1	0	1
35	+1	1	1	1
40	+0	0	1	1

2.58 Rising-edge triggered toggle flip-flop (T-flip-flop), with asynchronous active-high clear signal

2.59 **(a)**

```
module ROM4_3(ROMin, ROMout);
  input[3:0] ROMin;
  output[2:0] ROMout;

  reg[2:0] ROM16X3 [15:0];

  initial begin
    ROM16X3[0] <= 3'b000;
    ROM16X3[1] <= 3'b001;
    ROM16X3[2] <= 3'b001;
    ROM16X3[3] <= 3'b010;
    ROM16X3[4] <= 3'b001;
    ROM16X3[5] <= 3'b010;
    ROM16X3[6] <= 3'b010;
    ROM16X3[7] <= 3'b011;
    ROM16X3[8] <= 3'b001;
    ROM16X3[9] <= 3'b010;
    ROM16X3[10] <= 3'b010;
    ROM16X3[11] <= 3'b011;
    ROM16X3[12] <= 3'b010;
    ROM16X3[13] <= 3'b011;
    ROM16X3[14] <= 3'b011;
    ROM16X3[15] <= 3'b100;
  end

  assign ROMout = ROM16X3[ROMin];
endmodule
```

(b)

```
module P_59(A, count);
  input[11:0] A;
  output[3:0] count;

  wire[2:0] B,C,D;

  ROM4_3 R1(A[11:8], B);
  ROM4_3 R2(A[7:4], C);
  ROM4_3 R3(A[3:0], D);

  assign count = {1'b0, B} + C + D;
endmodule
```

(c)

A	Count	D	C	B
111111111111	1100	100	100	100
010110101101	0111	011	010	010
100001011100	0101	010	010	001

2.60

a	b	c	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

```
module decoder(A, B, C, Y);
  input A,B,C;
  output[7:0] Y;
  wire[2:0] index;
  reg[7:0] ROM[0:7];

  initial begin
    ROM[0] <= 8'b00000001;
    ROM[1] <= 8'b00000010;
    ROM[2] <= 8'b00000100;
    ROM[3] <= 8'b00001000;
    ROM[4] <= 8'b00010000;
    ROM[5] <= 8'b00100000;
    ROM[6] <= 8'b01000000;
    ROM[7] <= 8'b10000000;
  end

  assign index = {A,B,C};
  assign Y = ROM[index];
endmodule
```

2.61 **(a)**

```
always
begin
  MAX = A[0];
  for (i = 0; i < 20; i = i + 1) begin
    if(A[i] > MAX)
      MAX = A[i];
  end
end
```

(b)

```
always
begin
  MAX = A[0];
  i = 1;
  while (i < 20) begin
    if(A[i] > MAX)
      MAX = A[i];
    i = i + 1;
  end
end
```

2.62

```
module tester;
  reg CLK;
  reg [0:11] X;
  reg [0:11] Z;

  initial begin
    X = 12'b011011011100;
    Z = 12'b100110110110;
    CLK = 1;
  end

  reg Xin;
  wire Zout;
  integer i;

  always
    #50 CLK = ~CLK;

  Mealy M1(Xin, CLK, Zout);

  always
  begin
    for(i = 0; i < 12; i = i + 1) begin
      @(posedge CLK)
        #10 Xin = X[i];
        #80 // wait to read output
      if( Zout != Z[i]) begin
        $display("Error");
        $stop;
      end
    end
    $display("sequence correct");
    $stop;
  end
endmodule
```

2.63

```
module TestExcess3;
  reg[3:0] XA[1:10];
  reg[3:0] ZA[1:10];
  reg X, CLK;
  wire Z;
  integer i;
  integer j;

  initial begin
    X = 0;
    CLK = 0;
    XA[1] = 4'b0000;
    XA[2] = 4'b0001;
    XA[3] = 4'b0010;
    XA[4] = 4'b0011;
    XA[5] = 4'b0100;
    XA[6] = 4'b0101;
    XA[7] = 4'b0110;
    XA[8] = 4'b0111;
    XA[9] = 4'b1000;
    XA[10] = 4'b1001;
    ZA[1] = 4'b0011;
    ZA[2] = 4'b0100;
    ZA[3] = 4'b0101;
    ZA[4] = 4'b0110;
    ZA[5] = 4'b0111;
    ZA[6] = 4'b1000;
```

```
    ZA[7] = 4'b1001;
    ZA[8] = 4'b1010;
    ZA[9] = 4'b1011;
    ZA[10] = 4'b1100;
  end

  always
    #50 CLK <= ~CLK;

  Code_Converter C1(X, CLK, Z);

  always
  begin
    for(i = 1 ; i <= 10; i = i + 1)
    begin
      for(j= 0; j <= 3; j = j + 1)
      begin
        X = XA[i][j];
        @(posedge CLK);
        #(25);
        if(ZA[i][j] != Z) begin
          $display("sequence incorrect");
          $stop;
        end
      end
    end
    $display("all sequences correct");
    $stop;
  end
endmodule
```

2.64

```
module testbench;
  wire S5;
  reg clk, Ld8, Enable;
  wire[3:0] Q;

  initial begin
    clk = 1;
    Ld8 = 1;
    Enable = 0;
    #100
    Ld8 = 0;
    Enable = 1;
    #500 Enable = 0;
    #200 Enable = 1;
    #1000 Enable = 0;
  end

  always
    #50 clk <= ~clk;

  always @(posedge S5)
      $display($time);

  countQ1 C1(clk, Ld8, Enable, S5, Q);
endmodule
```

2.65

```
module testSMQ1(correct);
  output reg correct;
  reg clk, X;
  integer i;
  wire Z;
  reg[1:5] answer;
```

```
  initial begin
    answer[1] = 1;
    answer[2] = 1;
    answer[3] = 0;
    answer[4] = 1;
    answer[5] = 0;
    clk = 0;
    X = 1;
    #100 X = 0;
    #200 X = 1;
  end

  always
    #50 clk <= ~clk;

  SMQ1 S1(X, clk, Z);

  always
  begin
    for(i = 1; i <= 5; i = i + 1)
    begin
      @(posedge clk);
      correct = (answer[i] == Z);
      if(correct == 0)
        $display($time);
      #10;
    end
    $stop;
  end
endmodule
```

Chapter 3: Introduction to Programmable Logic Devices

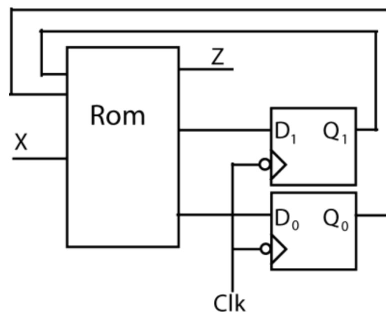
3.1 **(a)** $2^{17} \times 9$
(b) $2^{8} \times 7$
(c) $2^{6} \times 1$
(d) $2^{64} \times 33$
(e) $2^{3} \times 8$
(f) $2^{64} \times 32$
(g) $2^{32} \times 32$
(h) $2^{33} \times 17$
(i) $2^{8} \times 4$
(j) $2^{10} \times 5$
(k) $2^{11} \times 1$

Note: Cases i and j consider 1 output line to indicate output valid/invalid. Remember mux select lines for (k).

3.2

```
module FG(A, B, C, F, G);
  input A, B, C;
  output F, G;

  reg [0:1] ROM8_2 [0:7];

  initial begin
    ROM8_2[0] = 2'b11;
    ROM8_2[1] = 2'b11;
    ROM8_2[2] = 2'b10;
    ROM8_2[3] = 2'b00;
    ROM8_2[4] = 2'b01;
    ROM8_2[5] = 2'b01;
    ROM8_2[6] = 2'b10;
    ROM8_2[7] = 2'b01;
  end

  wire [0:1] romout;
  wire [0:1] index;

  assign index = {A, B, C};
  assign romout = ROM8_2[index];
  assign F = romout[0];
  assign G = romout[1];
endmodule
```

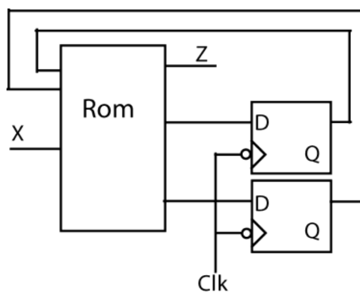
3.3 **(a)**



Q_1	Q_0	X	D_1	D_0	Z
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	1	1	0
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

(b)
```
module ROMQ3(X, Clk, Z);
  input X, Clk;
  output reg Z;

  reg [1:0] Q, Qplus;
  reg [2:0] FSM_ROM [0:7];
  reg [2:0] ROMValue;
  reg [2:0] index;

  initial begin
    Q = 1'b0;
    Qplus = 1'b0;
    FSM_ROM[0] = 3'b000;
    FSM_ROM[1] = 3'b011;
    FSM_ROM[2] = 3'b101;
    FSM_ROM[3] = 3'b110;
    FSM_ROM[4] = 3'b011;
    FSM_ROM[5] = 3'b110;
    FSM_ROM[6] = 3'b110;
    FSM_ROM[7] = 3'b101;
  end

  always @(Q, X)
  begin
    index = {Q, X};
    ROMValue = FSM_ROM[index];
    Qplus = ROMValue[2:1];
    Z = ROMValue[0];
  end

  always @(negedge Clk)
  begin
    Q <= Qplus;
  end
endmodule
```

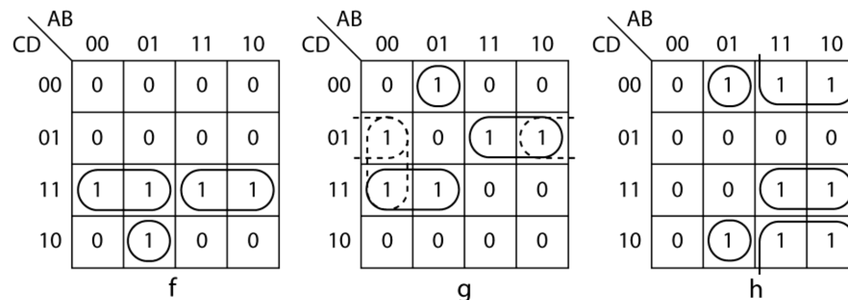
3.4 (a)



Q_1	Q_0	X	D_1	D_0	Z
0	0	0	0	1	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	0	0

(b)
```
module ROMQ4(X, Clk, Z);
   input X, Clk;
   output reg Z;

   reg [1:2] Q, Qplus;
   reg [2:0] FSM_ROM [0:7];
   reg [2:0] ROMValue;
   reg [1:3] index;

   initial begin
     Q = 2'b00;
     Qplus = 2'b00;
     FSM_ROM[0] = 3'b010;
     FSM_ROM[1] = 3'b101;
     FSM_ROM[2] = 3'b101;
     FSM_ROM[3] = 3'b001;
     FSM_ROM[4] = 3'b001;
     FSM_ROM[5] = 3'b010;
     FSM_ROM[6] = 3'b000;
     FSM_ROM[7] = 3'b000;
   end

   always @(Q, X)
   begin
     index = {Q, X};
     ROMValue = FSM_ROM[index];
     #10 Qplus = ROMValue[2:1];
     #10 Z = ROMValue[0];
   end

   always @(negedge Clk)
   begin
     #15 Q <= Qplus;
   end

endmodule
```

3.5 **(a)**

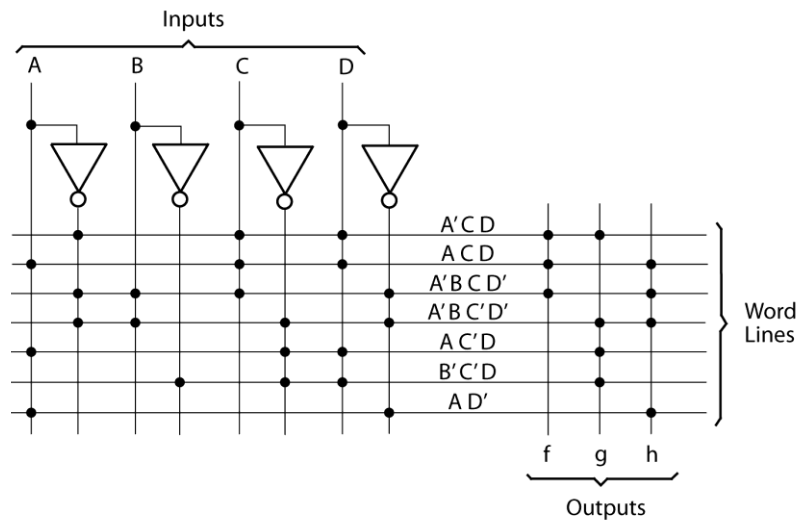


$f = \underline{A'CD} + \underline{ACD} + \underline{A'BCD'}$

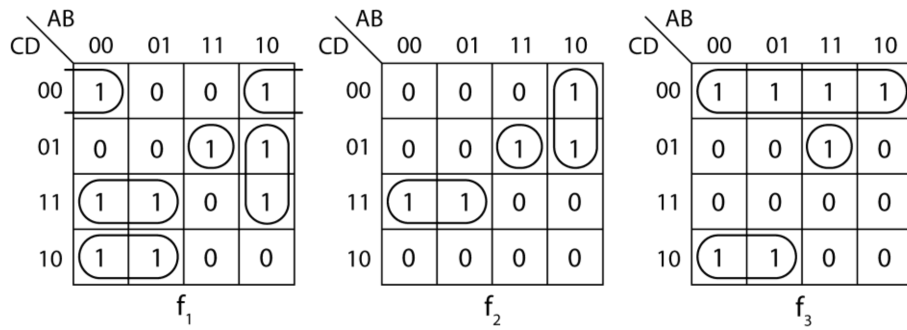
$g = \underline{A'BC'D'} + AC'D + \underline{A'CD} + (B'C'D \text{ or } A'B'D)$

$h = \underline{A'BC'D'} + \underline{A'BCD'} + AD' + \underline{ACD}$

Product Term	Inputs A	B	C	D	Outputs f	g	h
A'CD	0	-	1	1	1	1	0
ACD	1	-	1	1	1	0	1
A'BCD'	0	1	1	0	1	0	1
A'BC'D'	0	1	0	0	0	1	1
AC'D	1	-	0	1	0	1	0
B'C'D	-	0	0	1	0	1	0
AD'	1	-	-	0	0	0	1

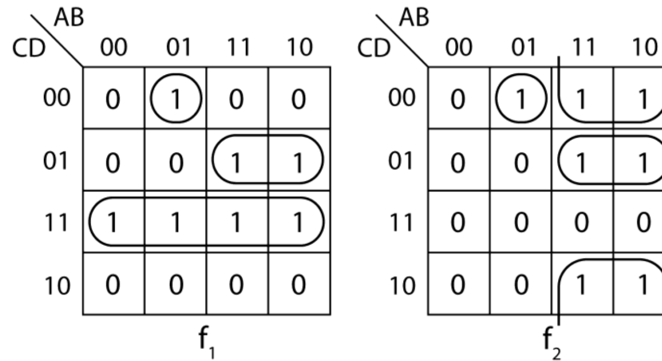


3.6 **(a)**



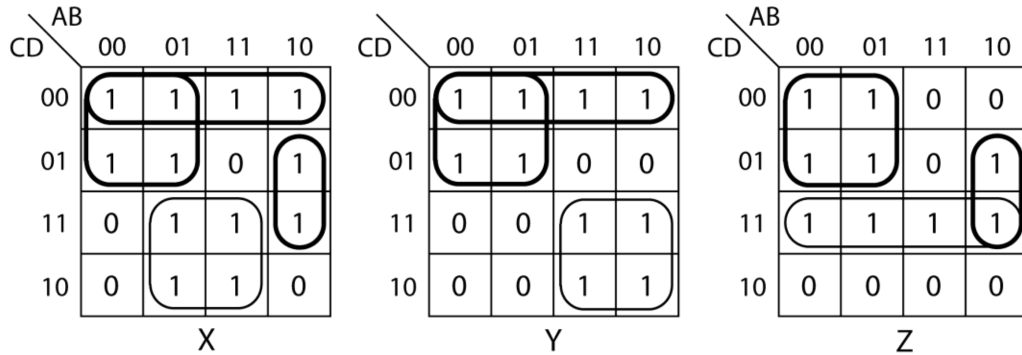
Product Term	Inputs A	B	C	D	Outputs f_1	f_2	f_3
B'C'D'	-	0	0	0	1	0	0
ABC'D	1	1	0	1	1	1	1
AB'D	1	0	-	1	1	0	0
A'CD	0	-	1	1	1	1	0
A'CD'	0	-	1	0	1	0	1
AB'C'	1	0	0	-	0	1	0
C'D'	-	-	0	0	0	0	1

(b)



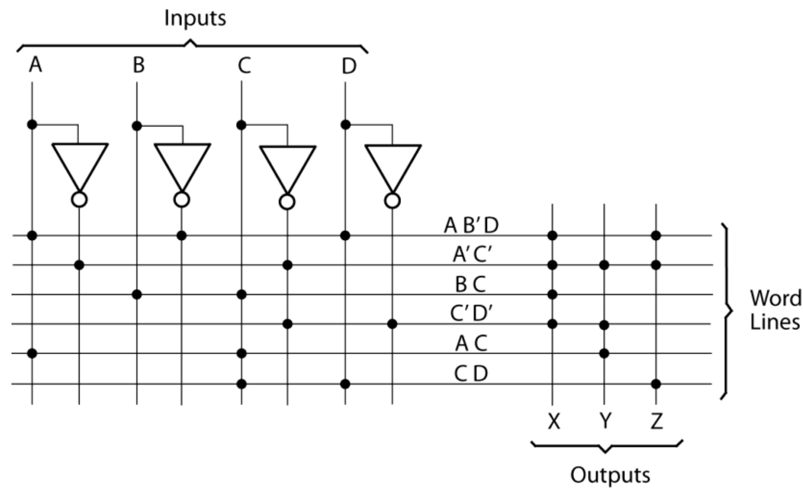
Product Term	Inputs A	B	C	D	Outputs f_1	f_2
A'BC'D'	0	1	0	0	1	1
AC'D	1	-	0	1	1	1
CD	-	-	1	1	1	0
AD'	1	-	-	0	0	1

3.7 (a)



Product Term	Inputs A	B	C	D	Outputs X	Y	Z
AB'D	1	0	-	1	1	0	1
A'C'	0	-	0	-	1	1	1
BC	-	1	1	-	1	0	0
C'D'	-	-	0	0	1	1	0
AC	1	-	1	-	0	1	0
CD	-	-	1	1	0	0	1

(b)



3.8

```
module output_macrocell(QR_Out, CK, SP, AR, Buffer_Out, S1, S0, Feedback,
Output);
    input QR_Out, CK, SP, AR, Buffer_Out, S1, S0;
    output Feedback, Output;

    reg Q;
    wire Qnot;

    always @(CK, AR)
    begin
      if(AR == 1)
        Q <= 1'b0;
      else if(CK == 1'b1) begin
        if(SP == 1'b1)
          Q <= 1'b1;
        else
          Q <= QR_Out;
      end
    end

    assign Qnot = !Q;
    assign Feedback = (S1 == 1'b0)? Qnot : Buffer_Out;
    assign Output = (S1 == 1'b0)? ((S0 == 1'b0)? Q : Qnot):
                                  ((S0 == 1'b0)? QR_Out : !QR_Out);
endmodule
```

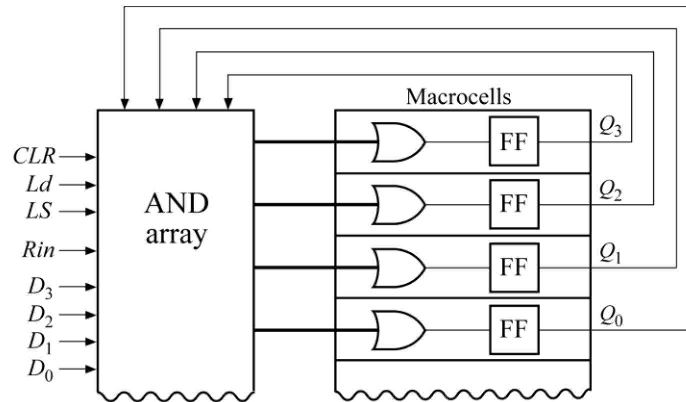
3.9 **(a)** N flip-flops + N data + 2 serial + 3 control + 1 clock = $2N + 6 \leq 22$

$N = 8$

(b) $A_0^+ = \text{Load}'\ \text{Rsh}\ A_1 + \text{Load}'\ \text{Lsh}\ \text{Rsi} + \text{Load}\ D_0 + \text{Rsh}'\ \text{Lsh}'\ \text{Load}'\ A_0$

$A_1^+ = \text{Load}'\ \text{Rsh}\ A_2 + \text{Load}'\ \text{Lsh}\ A_0 + \text{Load}\ D_1 + \text{Rsh}'\ \text{Lsh}'\ \text{Load}'\ A_1$

3.10



Q_3^+ = Clr' Ld' Ls' Q_3 + Clr' Ld' Ls Q_2 + Clr' Ld D_3
Q_2^+ = Clr' Ld' Ls' Q_2 + Clr' Ld' Ls Q_1 + Clr' Ld D_2
Q_1^+ = Clr' Ld' Ls' Q_1 + Clr' Ld' Ls Q_0 + Clr' Ld D_1
Q_0^+ = Clr' Ld' Ls' Q_0 + Clr' Ld' Ls Rin + Clr' Ld D_0

3.11 Conditions to satisfy - Number of flip-flops + outputs <=10; Number of flip-flops +outputs + inputs <=22. If four macrocells are used for outputs and one input is used for the clock, we have the following possibilities:
11 inputs and 64 states (6 flip-flops used)
12 inputs and 32 states (5 flip-flops used)
---
15 inputs and 4 states (2 flip-flops used)
16 inputs and 2 states (1 flip-flop used)
No, any Mealy circuit with the above number of inputs and states cannot be realized. A Mealy network with one of the above combinations can be realized only if the number of terms in the D flip-flop input equations and in the output equations fit in the 22V10. Maximum number of terms in any equation is 16.

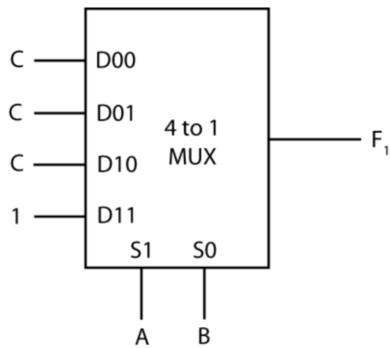
3.12 **(a)** FPGAs can be programmed outside of the factory
(b) symmetrical array, row based, hierarchical PLD, sea of gates
(c) antifuse, EEPROM, EPROM, SRAM
(d) flexibility/reprogramability
(e) fast, small, nonvolatile
(f) programmable logic blocks, programmable interconnect, programmable I/O blocks
(g) slow, , volatile, high area overhead
(h) no flexibility/reprogramability
(i) 6
(j) Mask Programmable Gate Array - custom gate array produced at a factory.
(k) FPGAs have a much more flexible general-purpose interconnect. Larger devices available in FPGA technology.
(l) predictability in timing, low-cost
(m) more flexibility
(n) Xilinx, Atmel, Cypress, Lattice Semiconductor, Altera
(o) Xilinx, Altera, Lattice Semiconductor, Actel, Quick Logic, Atmel

3.13 **(a)** Simple applications where the additional capabilities (and cost) of FPGAs are not needed. Applications where it is necessary to be able to accurately predict interconnect timing.

(b) High-volume applications that will not need field-programmable revisions.
(c) Prototyping, low-volume applications, applications that need field-programmable revisions.
- **(d)** SRAM
- **(e)** antifuse
- **(f)** FPGA
- **(g)** MPGA

3.14 **(a)**

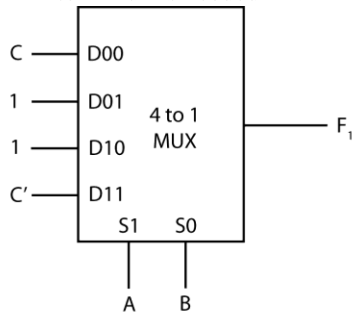
A	B	C	F_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



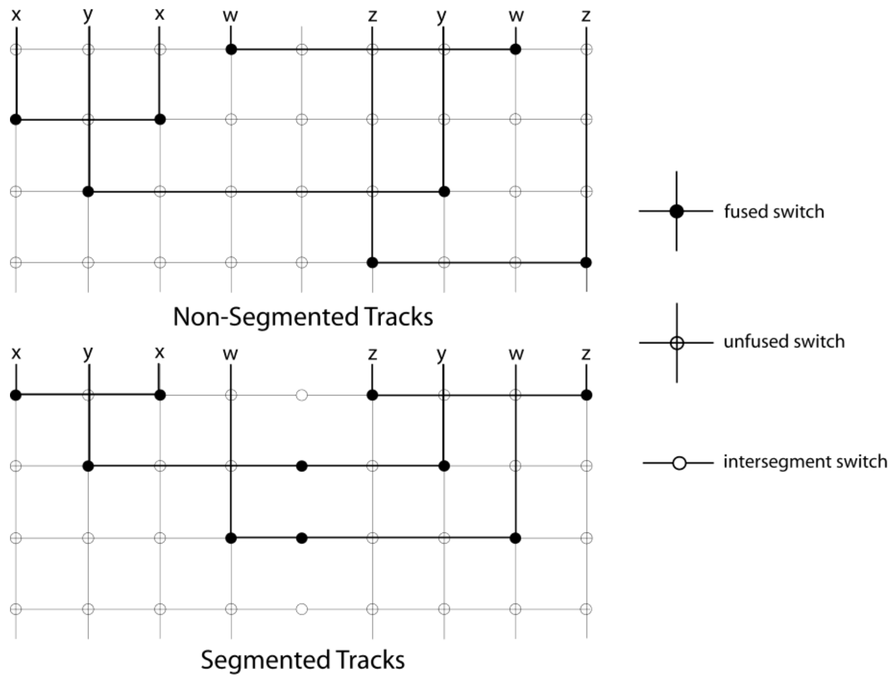
(b)

A	B	C	F_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

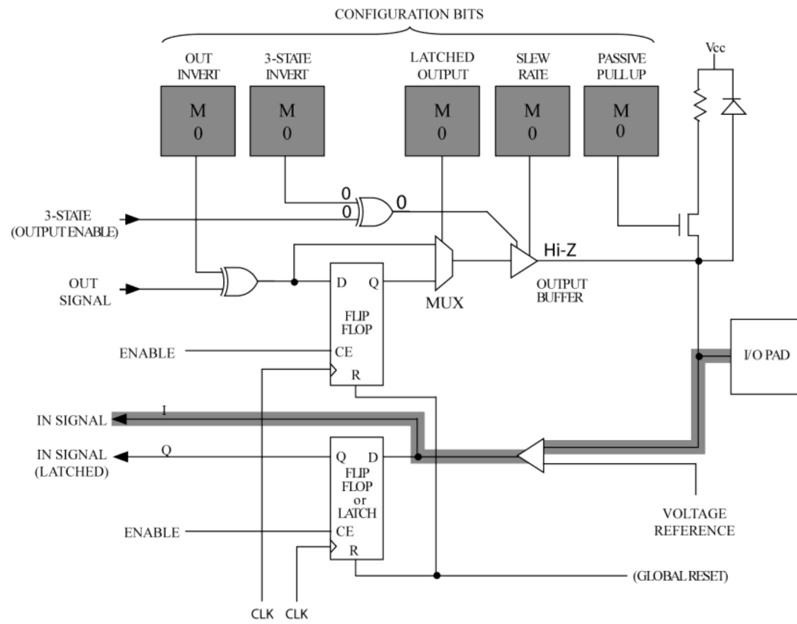
A 4-to-1 mux is needed



3.15

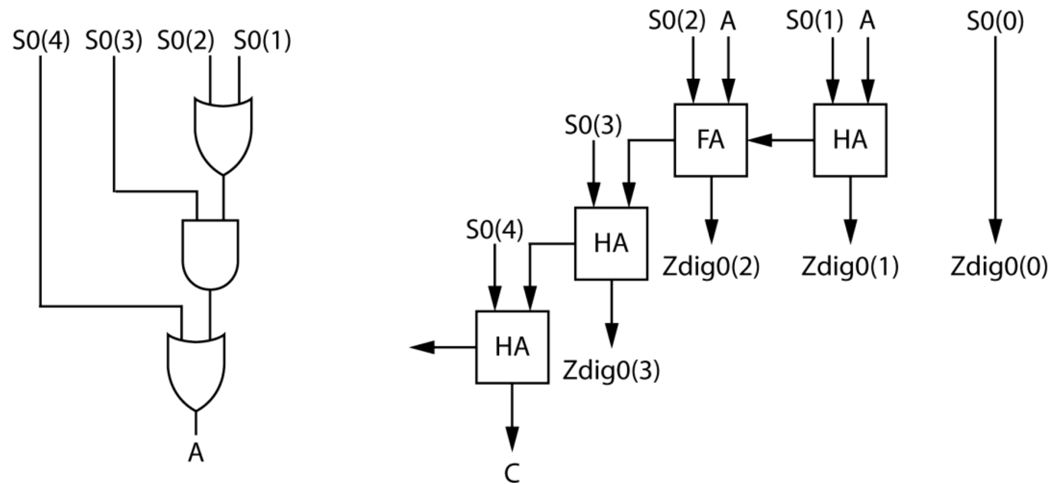


3.16



Chapter 4: Design Examples

4.1



4.2 **(a)** gate delay = 5 ns
1-bit adder delay = 2 * 5 = 10 ns
4-bit ripple carry adder delay = 4 * 10 = 40 ns

(b) gate delay = 5 ns
fastest 4-bit adder (using gates) = 2 * 5 = 10 ns
This is a 2-level sum-of-products adder that just uses combinational logic.

4.3

```
  `define A0 A[3:0]
  `define A1 A[7:4]
  `define A2 A[11:8]
  `define A3 A[15:12]
  `define B0 B[3:0]
  `define B1 B[7:4]
  `define B2 B[11:8]
  `define B3 B[15:12]
module CLA16(A, B, Ci, S, Co, PG, GG);
  input [15:0] A, B;
  input Ci;
  output [15:0] S;
  output Co, PG, GG;

  wire [3:0] S0, S1, S2, S3, G, P;
  wire [3:1] C;
  wire open;

  CLALogic CarryLogic(G, P, Ci, C, Co, PG, GG);
  CLA4 CLAa(`A0, `B0, Ci, S0, open, P[0], G[0]);
  CLA4 CLAb(`A1, `B1, C[1], S1, open, P[1], G[1]);
  CLA4 CLAc(`A2, `B2, C[2], S2, open, P[2], G[2]);
  CLA4 CLAd(`A3, `B3, C[3], S3, open, P[3], G[3]);

  assign S = {S3, S2, S1, S0};
endmodule
```

4.4

A_i	B_i	$P_i = A_i \oplus B_i$	$G_i = A_i B_i$	$C_{i+1} = G_i + P_i C_i$	$S_i = P_i \oplus C_i$
0	1	$P_0 = 1$	$G_0 = 0$	$C_0 = 0$	$S_0 = 1$
0	1	$P_1 = 1$	$G_1 = 0$	$C_1 = 0$	$S_1 = 1$
0	0	$P_2 = 0$	$G_2 = 0$	$C_2 = 0$	$S_2 = 0$
1	0	$P_3 = 1$	$G_3 = 0$	$C_3 = 0$	$S_3 = 1$
1	0	$P_4 = 1$	$G_4 = 0$	$\mathbf{C_4 = 0}$	$S_4 = 1$
1	0	$P_5 = 1$	$G_5 = 0$	$C_5 = 0$	$S_5 = 1$
1	1	$P_6 = 0$	$G_6 = 1$	$C_6 = 0$	$S_6 = 0$
1	1	$P_7 = 0$	$G_7 = 1$	$C_7 = 1$	$S_7 = 1$
0	0	$P_8 = 0$	$G_8 = 0$	$\mathbf{C_8 = 1}$	$S_8 = 1$
1	0	$P_9 = 1$	$G_9 = 0$	$C_9 = 0$	$S_9 = 1$
0	1	$P_{10} = 1$	$G_{10} = 0$	$C_{10} = 0$	$S_{10} = 1$
1	1	$P_{11} = 0$	$G_{11} = 1$	$C_{11} = 0$	$S_{11} = 0$
1	1	$P_{12} = 0$	$G_{12} = 1$	$\mathbf{C_{12} = 1}$	$S_{12} = 1$
0	1	$P_{13} = 1$	$G_{13} = 0$	$C_{13} = 1$	$S_{13} = 0$
1	0	$P_{14} = 1$	$G_{14} = 0$	$C_{14} = 1$	$S_{14} = 0$
0	0	$P_{15} = 0$	$G_{15} = 0$	$C_{15} = 1$	$S_{15} = 1$

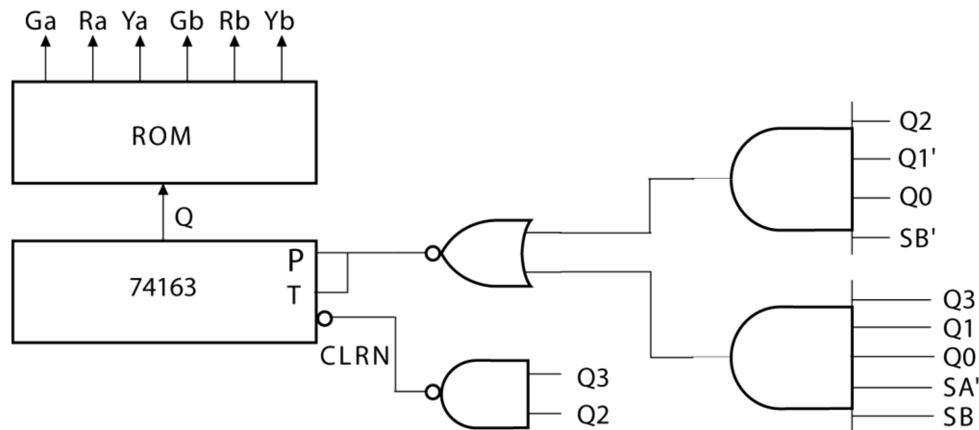
$P_{G0} = P_3 P_2 P_1 P_0 = 0$
$G_{G0} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 = 0$
$C_4 = G_{G0} + P_{G0} C_0 = 0$
$P_{G1} = P_7 P_6 P_5 P_4 = 0$
$G_{G1} = G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4 = 1$
$C_8 = G_{G1} + P_{G1} C_4 = 1$
$P_{G2} = P_{11} P_{10} P_9 P_8 = 0$
$G_{G2} = G_{11} + P_{11} G_{10} + P_{11} P_{10} G_9 + P_{11} P_{10} P_9 G_8 = 1$
$C_{12} = G_{G2} + P_{G2} C_8 = 1$
$P_{G3} = P_{15} P_{14} P_{13} P_{12} = 0$
$G_{G3} = G_{15} + P_{15} G_{14} + P_{15} P_{14} G_{13} + P_{15} P_{14} P_{13} G_{12} = 0$
$C_{out} = G_{G3} + P_{G3} C_{12} = 0$ $\quad$ S = 1001 0111 1011 1011

4.5 **(a)**

```
module FA_ACC(L, Ad, CLK, Y, CI, Acc, CO);
  input L, Ad, CLK, Y, CI;
  inout Acc;
  output CO;

  reg tempAcc;
  wire S;

  assign S = Acc ^ Y ^ CI;
  assign CO = (Acc & Y) | (Acc & CI) | (Y & CI);
  assign Acc = tempAcc;

  initial begin
    tempAcc <= 1'b0;
  end

  always @(posedge CLK)
  begin
    if(L == 1'b1)
      tempAcc <= Y;
    if(Ad == 1'b1)
      tempAcc <= S;
  end
endmodule
```

(b)
```
module SUB4(Ld, Su, CLK, B, ACC);
  input Ld, Su, CLK;
  input [3:0] B;
  output [3:0] ACC;

  wire [4:1] C;
  wire [3:0] Bin;
  assign Bin = (Su == 1'b1)? (~B) : B;

  FA_ACC S0(Ld, Su, CLK, Bin[0], C[4], ACC[0], C[1]);
  FA_ACC S1(Ld, Su, CLK, Bin[1], C[1], ACC[1], C[2]);
  FA_ACC S2(Ld, Su, CLK, Bin[2], C[2], ACC[2], C[3]);
  FA_ACC S3(Ld, Su, CLK, Bin[3], C[3], ACC[3], C[4]);
endmodule
```

4.6 **(a)** Clear when $Q_3 = Q_2 = 1$, CLRN = $(Q_3\ Q_2)'$, LOADN = 1
Count except in S_5, $S_b' = 1$ and in S_{11}, $S_a'\ S_b = 1$
So $P = T = (Q_2\ Q_1'\ Q_0\ S_b' + Q_3\ Q_1\ Q_0\ S_a'\ S_b)'$



Alternate Solution:
Clear as above. Set PT = 1 (always count except when loading or clearing). To prevent counting in S6, Sb' = 1. Load Q back into the counter: D = Q. LOADN = $(Q_2\ Q_1'\ Q_0\ S_b' + Q_3\ Q_1\ Q_0\ S_a'\ S_b)'$

(b)
```
module traffic_light(clk, Sa, Sb, Ra, Rb, Ga, Gb, Ya, Yb);
  input clk, Sa, Sb;
  output Ra, Rb, Ga, Gb, Ya, Yb;

  reg [5:0] ROMOut [0:15];

  wire [5:0] ROMValue;
  wire LdN, ClrN, P, T, Carry, A1, A2, A3;
  wire SbN, SaN, Q1N;
  wire [3:0] Q;

  initial begin
    ROMOut[0]  =  6'b100010;
    ROMOut[1]  =  6'b100010;
    ROMOut[2]  =  6'b100010;
    ROMOut[3]  =  6'b100010;
    ROMOut[4]  =  6'b100010;
    ROMOut[5]  =  6'b100010;
    ROMOut[6]  =  6'b001010;
    ROMOut[7]  =  6'b010100;
```

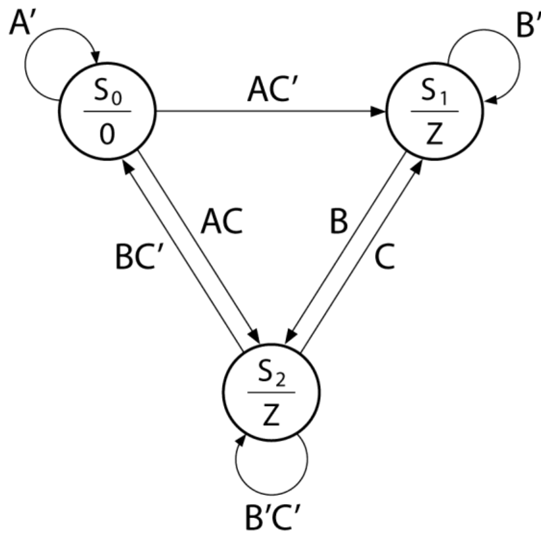
```
      ROMOut[8]  =  6'b010100;
      ROMOut[9]  =  6'b010100;
      ROMOut[10] =  6'b010100;
      ROMOut[11] =  6'b010100;
      ROMOut[12] =  6'b010001;
      ROMOut[13] =  6'b000000;
      ROMOut[14] =  6'b000000;
      ROMOut[15] =  6'b000000;
    end

    assign ROMValue = ROMOut[Q];
    assign Ga = ROMValue[5];
    assign Ra = ROMValue[4];
    assign Ya = ROMValue[3];
    assign Gb = ROMValue[2];
    assign Rb = ROMValue[1];
    assign Yb = ROMValue[0];

    // see code for 74163 in Figure 2-44
    // see Section 2.15 for the definition of the Nand3 component
    // the Inverter, Nand2, And4, And2, and Nor2 components are
    //   similar to the Nand3 component in Section 2.15

    Inverter I1(Sb, SbN);
    Inverter I2(Sa, SaN);
    Inverter I3(Q[1], Q1N);
    Nand2 G1(Q[3], Q[2], ClrN);
    And4 G2(Q[2], Q1N, Q[0], SbN, A1);
    And4 G3(Q[3], Q[1], Q[0], SaN, A2);
    And2 G4(A2, Sb, A3);
    Nor2 G5(A1, A3, P);
    C74163 CT1(1'b1, ClrN, P, P, clk, 4'b0000, Carry, Q);
  endmodule
```

(c)

```
    `define R 1
    `define Y 2
    `define G 3
  module testbench;
    reg [0:4] SaArray;
    time [0:4] SaDelay;
    reg [0:7] SbArray;
    time [0:7] SbDelay;

    reg clk;
    wire [0:1] lightA, lightB;
    reg Sa, Sb;
    wire Ra, Rb, Ga, Gb, Ya, Yb;

    integer i;

    initial begin
      clk = 0;
      Sa = 0;
      Sb = 0;
      i = 0;
      SaArray[0] = 1;
      SaArray[1] = 0;
      SaArray[2] = 1;
      SaArray[3] = 0;
      SaArray[4] = 1;
      SaDelay[0] = 0;
      SaDelay[1] = 40;
      SaDelay[2] = 130;
```

```
    SaDelay[3] = 60;
    SaDelay[4] = 20;
    SbArray[0] = 0;
    SbArray[1] = 1;
    SbArray[2] = 0;
    SbArray[3] = 1;
    SbArray[4] = 0;
    SbArray[5] = 1;
    SbArray[6] = 0;
    SbArray[7] = 1;
    SbDelay[0] = 0;
    SbDelay[1] = 70;
    SbDelay[2] = 30;
    SbDelay[3] = 20;
    SbDelay[4] = 30;
    SbDelay[5] = 60;
    SbDelay[6] = 40;
    SbDelay[7] = 20;
  end

  always #5 clk = !clk;

  always
  begin
    for (i = 0; i <= 4 ; i=i+1)
    begin
      wait(SaDelay[i]);
      Sa <= SaArray[i];
    end
  $stop;
  end

  always
  begin
    for (i = 0; i <=7 ; i=i+1)
    begin
      wait(SbDelay[i]);
      Sb <= SbArray[i];
    end
  $stop;
  end

  assign lightA =  (Ra == 1)? `R : ((Ya == 1)? `Y : ((Ga == 1)? `G :
0));
  assign lightB =  (Rb == 1)? `R : ((Yb == 1)? `Y : ((Gb == 1)? `G :
0));

  traffic_light traffic_light1(clk, Sa, Sb, Ra, Rb, Ga, Gb, Ya, Yb);
endmodule
```

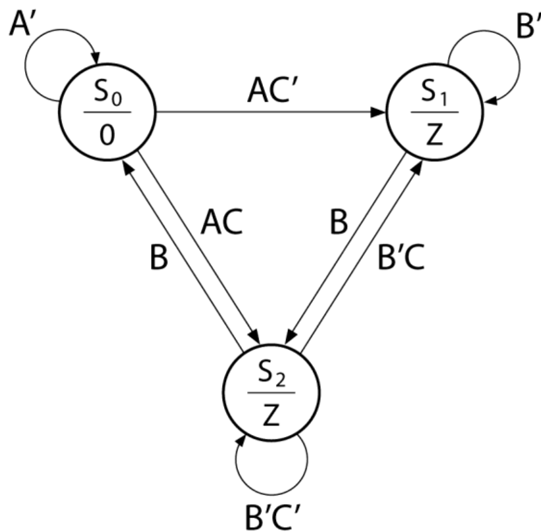
4.7 Solution 1:



S_0:	A' + A C' + A C = 1	(A') (A C') (A C) = 0
S_1:	B + B' = 1	(B) (B') = 0
S_2:	B' C' + B C' + C = 1	(B' C') (B C') (C) = 0

State	ABC 000	001	010	011	100	101	110	111	Z Output
S_0	S_0	S_0	S_0	S_0	S_1	S_2	S_1	S_2	0
S_1	S_1	S_1	S_2	S_2	S_1	S_1	S_2	S_2	1
S_2	S_2	S_1	S_0	S_1	S_2	S_1	S_0	S_1	1

Solution 2:



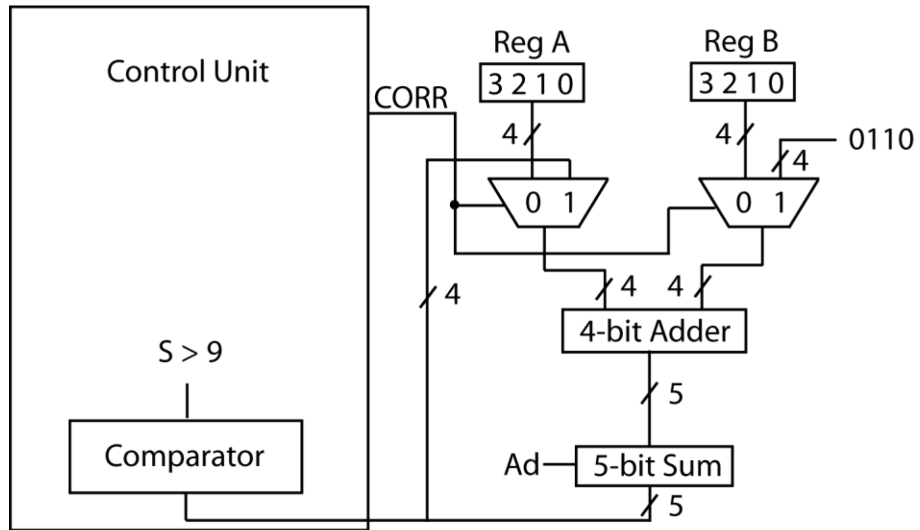
S_0:	A' + A C' + A C = 1	(A') (A C') (A C) = 0
S_1:	B + B' = 1	(B) (B') = 0
S_2:	B' C' + B' C + B = 1	(B' C') (B' C) (B) = 0

State	ABC 000	001	010	011	100	101	110	111	Z Output
S_0	S_0	S_0	S_0	S_0	S_1	S_2	S_1	S_2	0
S_1	S_1	S_1	S_2	S_2	S_1	S_1	S_2	S_2	1
S_2	S_2	S_1	S_0	S_0	S_2	S_1	S_0	S_0	1

4.8

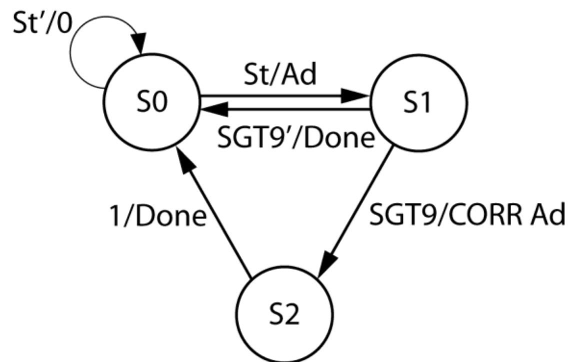
```
module wave(clk, W);
  input clk;
  output W;

  reg state;
  integer count;

  initial begin
    state = 0;
    count = 0;
  end

  assign W = state;

  always @(negedge clk)
  begin
    case(state)
    0: begin
      if(count == 29) begin
        count <= 1;
        state <= 1;
      end
      else begin
        count <= count + 1;
      end
    end
    1: begin
      if(count == 43) begin
        count <= 1;
        state <= 0;
      end
      else begin
        count <= count + 1;
      end
    end
    endcase
  end
endmodule
```

4.9 **(a)**

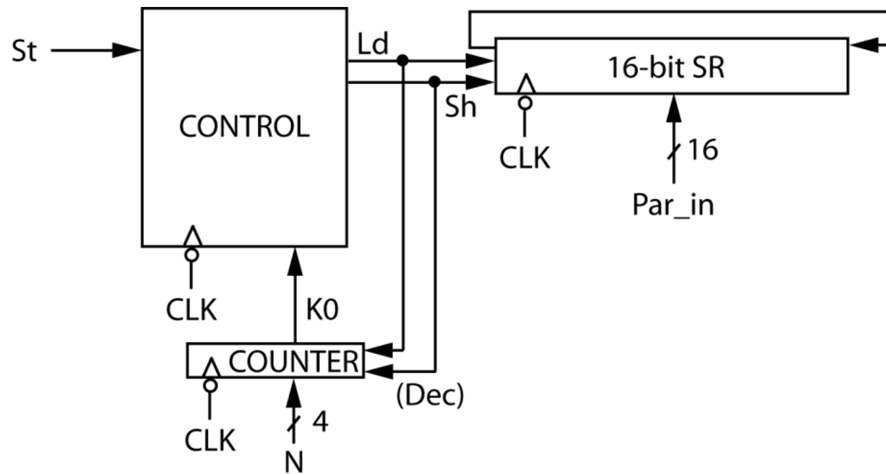


(b) 1. If St = 1, A + B → Sum; Ad = 1; CORR = 0
2. If S > 9, (add 6 to Sum; CORR = 1, Ad = 1), else go to 3
3. If Done = 1, go to 1 and wait for St

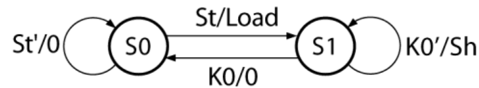
(c)



4.10 **(a)**



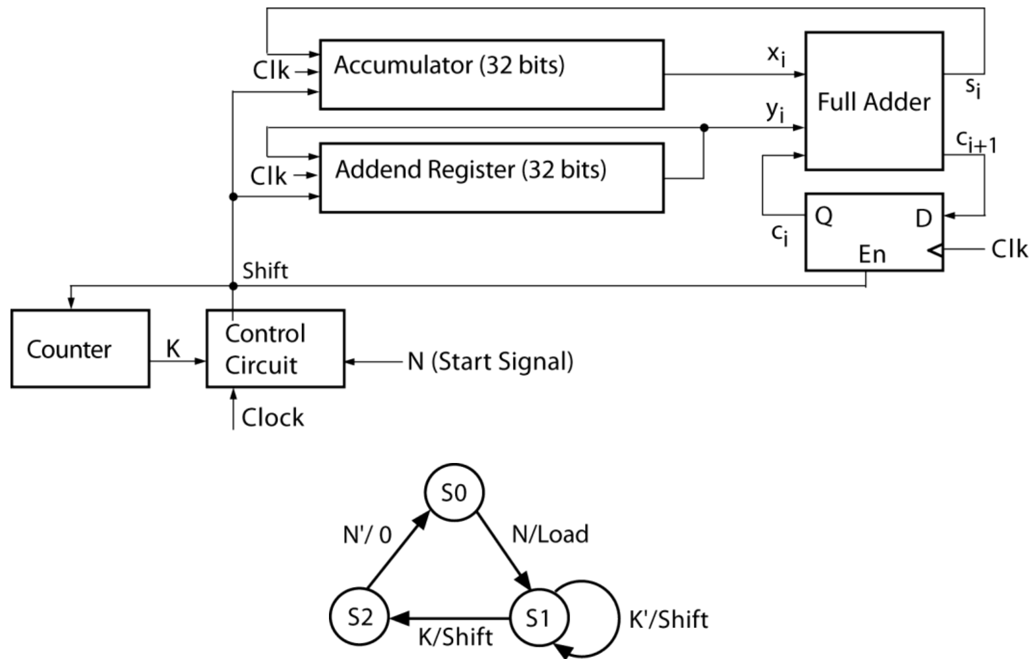
(b)



(c)
```
module shifter(St, CLK, Par_in, N, SR);
  input St, CLK;
  input [15:0] Par_in;
  input [3:0] N;
  output reg [15:0] SR;

  wire K0;
  reg Ld, Sh;
  integer PS, NS;
  integer CTR;

  initial begin
    SR = 0;
    Ld = 0;
    Sh = 0;
    PS = 0;
    NS = 0;
    CTR = 0;
  end

  assign K0 = (CTR == 0)? 1 : 0;

  always @(PS, St, K0)
  begin
    Ld = 0;
    Sh = 0;

    case(PS)
    0: begin
      if(St == 1) begin
        Ld = 1;
        NS = 1;
      end
      else begin
        NS = 0;
      end
    end
    1: begin
      if(K0 == 0) begin
        Sh = 1;
        NS = 1;
      end
      else begin
        NS = 0;
      end
    end
    endcase
  end

  always @(negedge CLK)
  begin
    PS <= NS;
    if(Ld == 1) begin
      SR  <= Par_in;
      CTR <= N;
    end
    if(Sh == 1) begin
```

```
      SR  <= {SR[14:0], SR[15]};
      CTR  <= CTR - 1;
    end
  end

endmodule
```

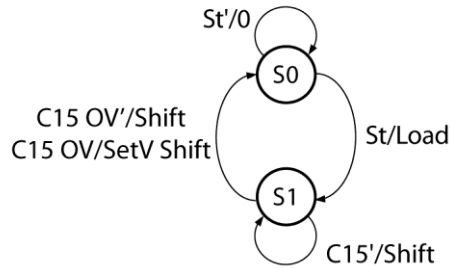
4.11 (a)



(b)
```
module SerialAdder(N, CLK, A, B, SUM);
  input N, CLK;
  input [31:0] A, B;
  output [32:0] SUM;

  wire K, SI, CIplus;
  reg CI, Load, Shift;
  reg [31:0] ACC, RegB;
  reg [1:0] state, nextstate;
  reg [4:0] Counter;

  initial begin
    CI = 0;
    Load = 0;
    Shift = 0;
    ACC = 0;
    RegB = 0;
    state = 0;
    nextstate = 0;
    Counter = 0;
  end

  assign K = (Counter == 31)? 1 : 0;
  assign SI = ACC[0] ^ RegB[0] ^ CI;
  assign CIplus = (ACC[0] & RegB[0] | (ACC[0] & CI) | (RegB[0] & CI));
  assign SUM = {CI, ACC};

  always @(state, N, K)
  begin
```

```
    Load = 0;
    Shift = 0;
    case(state)
    0: begin
      if(N == 1) begin
        Load = 1;
        nextstate = 1;
      end
      else begin
        nextstate = 0;
      end
    end
    1: begin
      if(K == 1) begin
        Shift = 1;
        nextstate = 2;
      end
      else begin
        Shift = 1;
        nextstate = 1;
      end
    end
    2: begin
      if(N == 0) begin
        nextstate = 0;
      end
      else begin
        nextstate = 2;
      end
    end
    default: begin
    end
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    if(Load == 1) begin
      ACC <= A;
      RegB <= B;
    end
    if(Shift == 1) begin
      ACC <= {SI, ACC[31:1]};
      RegB <= {RegB[0], RegB[31:1]};
      CI <= CIplus;
      if(K == 0)
        Counter <= Counter + 1;
      else
        Counter <= 0;
    end
  end

endmodule
```

4.12 **(a)**



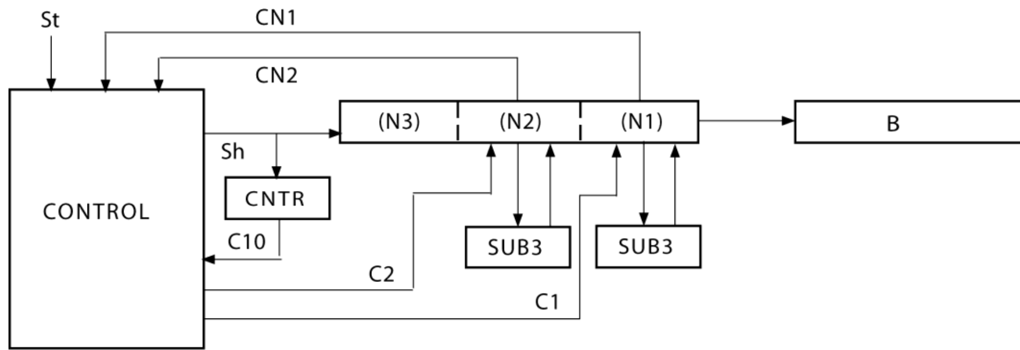
(b)
```
module subtracter(XIN, YIN, CLK, St, X, Y, V);
  input [15:0] XIN, YIN;
  input CLK, St;
  output [15:0] X, Y;
  output reg V;

  reg state, nextstate;
  integer C;
  wire S, CB, YP, C15, OV;
  reg CA, Load, Shift, SetV;
  reg [15:0] tempX, tempY;

  initial begin
    V = 0;
    state = 0;
    nextstate = 0;
    C = 0;
    CA = 0;
    Load = 0;
    Shift = 0;
    SetV = 0;
    tempX = 0;
    tempY = 0;
  end

  assign X = tempX;
  assign Y = tempY;
  assign C15 = (C == 15)? 1 : 0;
  assign OV = (!tempX[0] & !YP & S) | (tempX[0] & YP & !S);
  assign YP = !tempY[0];
  assign S = tempX[0] ^ YP ^ CA;
  assign CB = (tempX[0] & CA) | (YP & CA) | (tempX[0] & YP);

  always @(state, St, C15, OV)
  begin
    Load = 0;
    Shift = 0;
    SetV = 0;
    case(state)
    0: begin
      if(St == 1) begin
        Load = 1;
        nextstate = 1;
      end else
        nextstate = 0;
    end
    1: begin
      Shift = 1;
      if(C15 == 0) begin
        nextstate = 1;
      end
```

```
        else if(OV == 0) begin
          nextstate = 0;
        end
        else begin
          SetV = 1;
          nextstate = 0;
        end
      end
      endcase
    end

    always @(posedge CLK)
    begin
      state <= nextstate;
      if(Load == 1) begin
        tempX <= XIN;
        tempY <= YIN;
        CA <= 1;
        C <= 0;
        V <= 0;
      end
      if(Shift == 1) begin
        tempX <= {S, tempX[15:1]};
        tempY <= {1'b0, tempY[15:1]};
        CA <= CB;
        C <= C + 1;
      end
      if(SetV == 1) begin
        V <= 1;
      end
    end

  endmodule
```

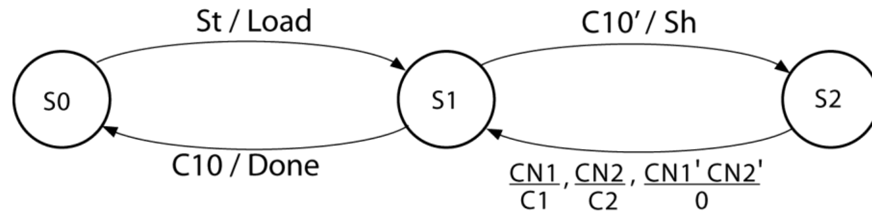
4.13 **(a)** 857 = 1000 0101 0111 (BCD number)

RegA	RegB	
1000 0101 0111	0000000000	shift
0100 0010 1011	1000000000	make correction
0100 0010 1000	1000000000	shift
0010 0001 0100	0100000000	shift (no correction needed)
0001 0000 1010	0010000000	make correction
0001 0000 0111	0010000000	shift
0000 1000 0011	1001000000	make correction
0000 0101 0011	1001000000	shift
0000 0010 1001	1100100000	make correction
0000 0010 0110	1100100000	shift
0000 0001 0011	0110010000	shift (no correction needed)
0000 0000 1001	1011001000	make correction
0000 0000 0110	1011001000	shift until finished (no corrections needed)
0000 0000 0000	1101011001	(binary result = 857)

(b)



(c)

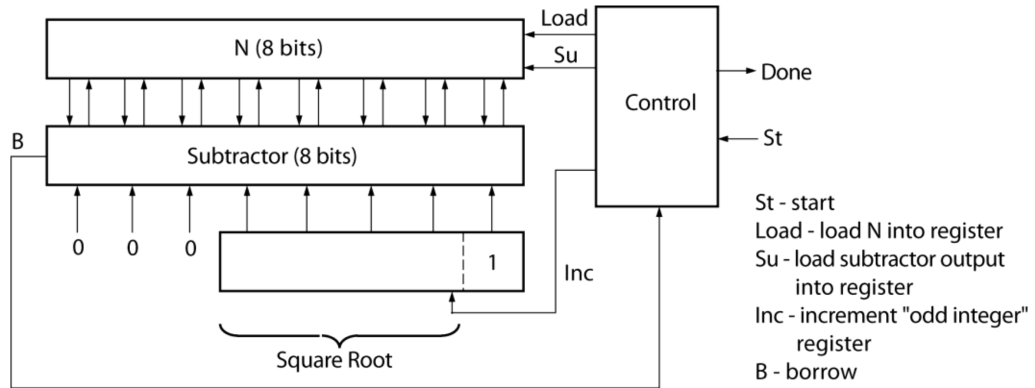


(d)

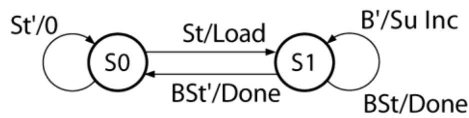
```
  `define CN1 N[3]
  `define CN2 N[7]
module BCD2binary(BCDin, St, clk, B, done);
  input [11:0] BCDin;
  input St, clk;
  output reg [9:0] B;
  output reg done;

  reg [11:0] N;
  reg load, C1, C2, Sh;
  reg [1:0] State, Nstate;
  integer ctr;
  wire [3:0] sub3_2, sub3_1;

  initial begin
    B = 0;
    done = 0;
    N = 0;
    load = 0;
    C1 = 0;
    C2 = 0;
    Sh = 0;
    State = 0;
    Nstate = 0;
    ctr = 0;
  end

  assign sub3_2 = N[7:4] + 4'b1101;
  assign sub3_1 = N[3:0] + 4'b1101;

  always @(State, St, `CN1, `CN2, ctr)
  begin
    load = 0;
    done = 0;
    C1 = 0;
    C2 = 0;
    Sh = 0;
```

```
    case(State)
    0: begin
      if(St == 1) begin
        load = 1;
        Nstate = 1;
      end
      else begin
        Nstate = 0;
      end
    end
    1: begin
      if(ctr == 10) begin
        done = 1;
        Nstate = 0;
      end
      else begin
        Sh = 1;
        Nstate = 2;
      end
    end
    2: begin
      Nstate = 1;
      if(`CN1 == 1)
        C1 = 1;
      if(`CN2 == 1)
        C2 = 1;
    end
    default: begin
    end
    endcase
  end

  always @(posedge clk)
  begin
    State <= Nstate;
    if(load == 1) begin
      N <= BCDin;
      B <= 10'b0000000000;
      ctr <= 0;
    end
    if(Sh == 1) begin
      N <= N >> 1;
      ctr <= ctr + 1;
      B <= {N[0], B[9:1]};
    end
    if(C2 == 1) begin
      N[7:4] <= sub3_2;
    end
    if(C1 == 1) begin
      N[3:0] <= sub3_1;
    end
  end

endmodule
```

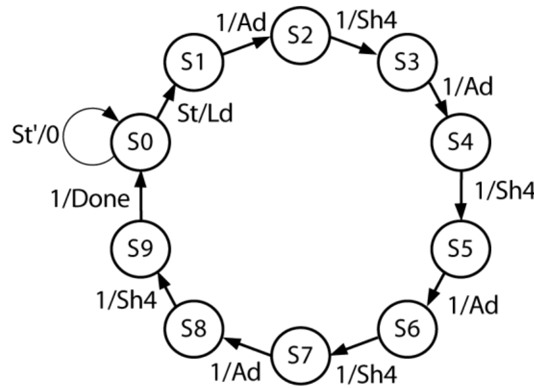
4.14 (a)



(b)



4.15 (a)



(b)
```
module mul16x4(Mult, Mcand, St, CLK, Done);
  input [15:0] Mult;
  input [3:0] Mcand;
  input St, CLK;
  output reg Done;

  reg [23:0] A;
  reg Ld, Ad, Sh4;
  reg [3:0] PS, NS;

  initial begin
    Done = 0;
    A = 0;
    Ld = 0;
    Ad = 0;
    Sh4 = 0;
    PS = 0;
    NS = 0;
  end

  always @(PS, St)
  begin
    Ld = 0;
```

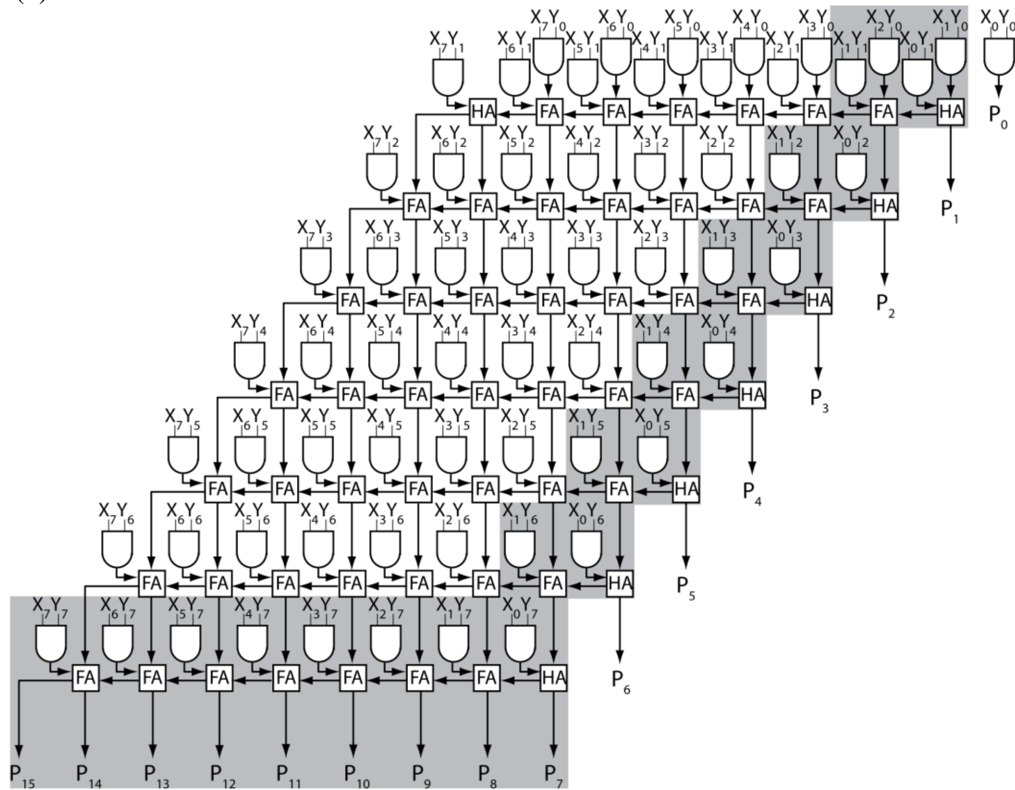
```
    Sh4 = 0;
    Ad = 0;
    Done = 0;
    case(PS)
    0: begin
      if(St == 1) begin
        Ld = 1;
        NS = 1;
      end
      else begin
        NS = 0;
      end
    end
    1,3,5,7: begin
      Ad = 1;
      NS = PS + 1;
    end
    2,4,6,8: begin
      Sh4 = 1;
      NS = PS + 1;
    end
    9: begin
      Done = 1;
      NS = 0;
    end
    default: begin
    end
    endcase
  end

  always @(posedge CLK)
  begin
    PS <= NS;
    if(Ld == 1)
      A <= {8'b00000000, Mult};
    if(Ad == 1)
      A[23:16] <= (Mcand * Mult[3:0]) + A[23:16];
    if(Sh4 == 1)
      A <= A >> 4;
  end

endmodule
```

4.16 **(a)** $16^2 = 256$ AND gates
16 half adders

$16 * (16 - 2) = 16 * 14 = 224$ full adders
$224 + 16 = 240$ adders

(b) longest delay in a 16 × 16 array multiplier = $44\ t_{ad} + t_g$

4.17 (a) & (b)

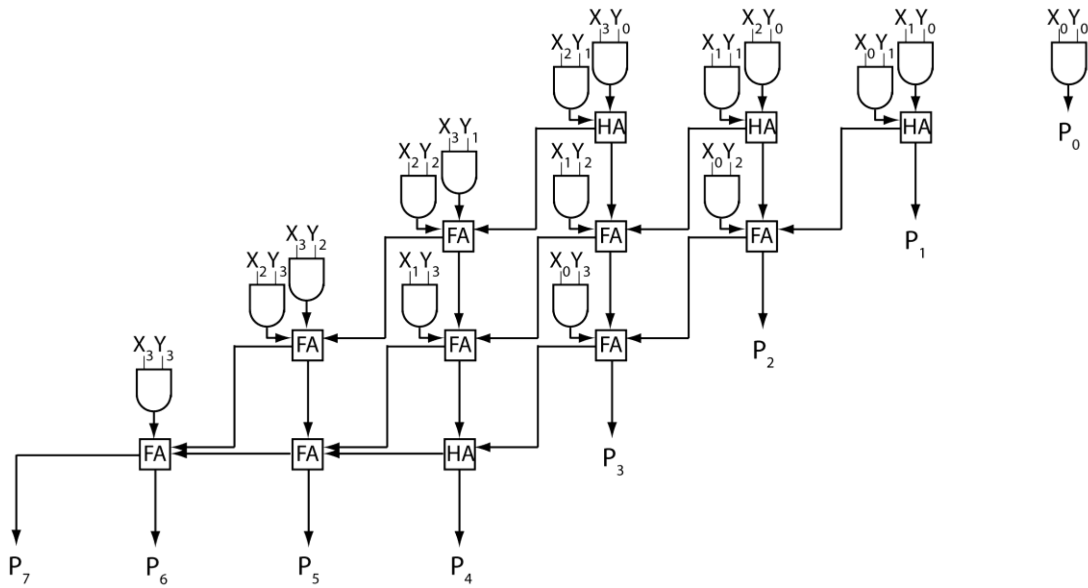


$n(n - 2) = 8 * (8 - 2) = 8 * 6 = 48$ full adders
$n = 8$ half adders
$n^2 = 8^2 = 64$ AND gates

(c) longest delay $= (3n - 4)t_{ad} + t_g = (3 * 8 - 4) * 2 + 1 = 20 * 2 + 1 = 41$ ns

(d) takes 16 clock cycles, frequency = 16 / 41 ns ≈ 390.244 MHz

4.18



4.19

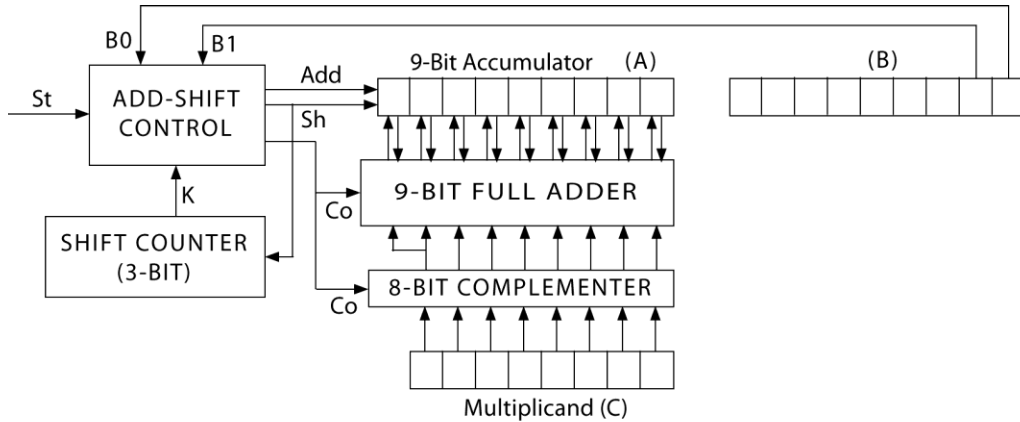
RegA	RegB
0000	1101
1111	1110
1111	1111
1111	0111
0000	0011

4.20 **(a)**

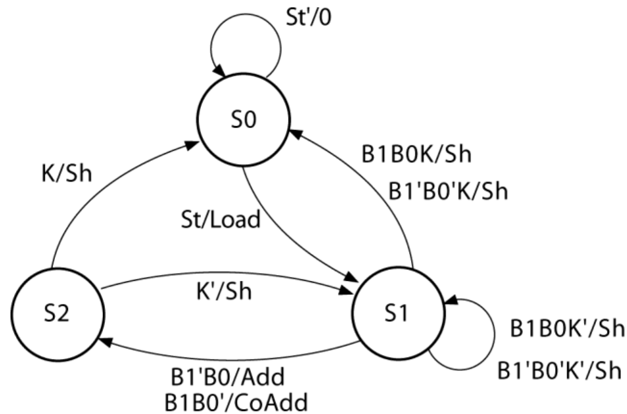
```
    1.0111   (-9/16) Multiplicand
     1.101   (-3/8) Multiplier
----------
 1.1110111   (-9/128)
 1.10111     (-9/32)
----------
 1.1010011
 0.1001      (+9/16) Add 2's Comp of Multiplicand
----------
 0.0011011   (+27/128)
```

(b) Block diagram is similar to Figure 4-31 with one bit added to the ACC, the Full Adder, the Complementer, and the multiplicand.

4.21 **(a)**



(b)



(c)

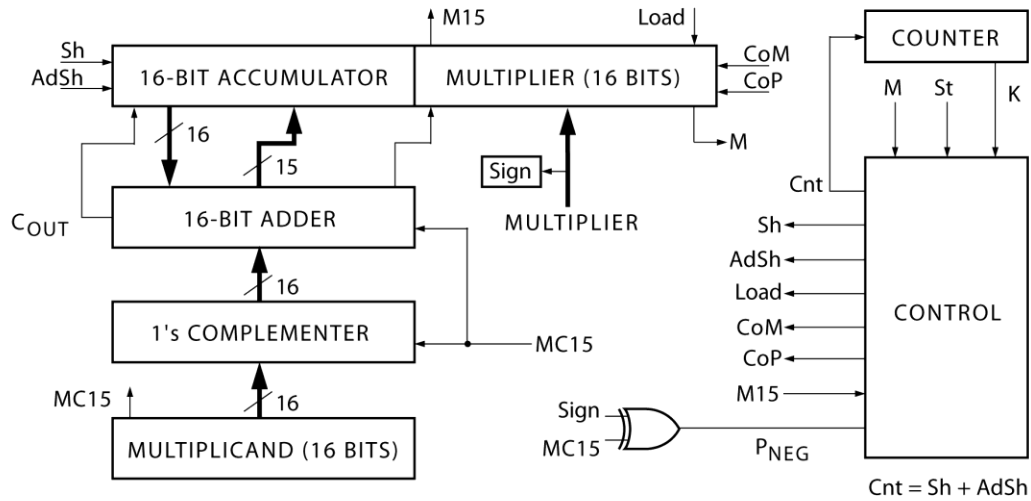
```
 `define B0 RegB[0]
 `define B1 RegB[1]
module BoothsMult(CLK, St, Mplier, Mcand, Product);
  input CLK, St;
  input [7:0] Mplier, Mcand;
  output [14:0] Product;

  reg [1:0] state;
  reg [2:0] Counter;
  reg [8:0] ACC, RegB;
  wire [8:0] Addout;
  reg [7:0] RegC;
  wire [7:0] Compout;
  wire Co;

  initial begin
    state = 0;
    Counter = 0;
    ACC = 0;
    RegB = 0;
    RegC = 0;
  end

  assign Product = {ACC[6:0], RegB[8:1]};
  assign Co = (`B1) & (!`B0);
  assign Compout = (Co == 1)? (!RegC) : (RegC);
  assign Addout = ACC + ({Compout[7], Compout}) + {7'b0000000, Co};
```

```
always @(posedge CLK)
begin
  case(state)
  0: begin
    if(St == 1) begin
      state <= 1;
      ACC <= 0;
      RegB <= {Mplier, 1'b0};
      RegC <= Mcand;
    end
    else begin
      state <= 0;
    end
  end
  1: begin
    if((`B0 ^ `B1) == 1) begin
      ACC <= Addout;
      state <= 2;
    end
    else begin
      ACC <= {ACC[8], ACC[8:1]};
      RegB <= {ACC[0], RegB[8:1]};
      if(Counter != 7) begin
        Counter <= Counter + 1;
        state <= 1;
      end
      else begin
        Counter <= 0;
        state <= 0;
      end
    end
  end
  2: begin
    if(Counter != 7) begin
      Counter <= Counter + 1;
      state <= 1;
    end
    else begin
      Counter <= 0;
      state <= 0;
    end
    ACC <= {ACC[8], ACC[8:1]};
    RegB <= {ACC[0], RegB[8:1]};
  end
  default: begin
  end
  endcase
end

endmodule
```

(d) Simulation should confirm the following results

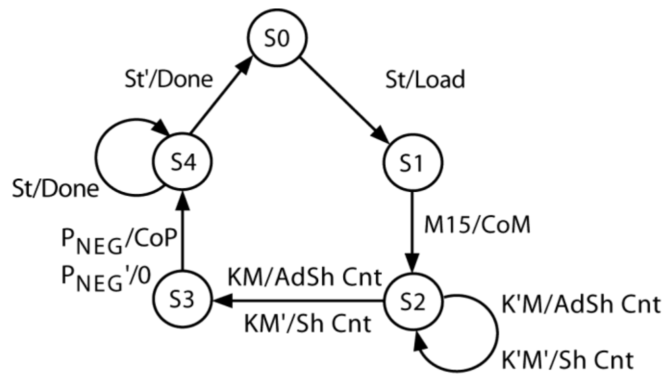
01100110 (102) × 00110011 (51)	= 001010001010010 (5202)
10100110 (-90) × 01100110 (102)	= 101110000100100 (-9180)
01101011 (107) × 10001110 (-114)	= 101000001011010 (-12198)
11001100 (-52) × 10011001 (-103)	= 001010011101100 (5356)

4.22 (a)



Cnt = Sh + AdSh Sh = M'(state = 2) AdSh = M(state = 2)
K = (Counter = 15) CoM = M15(state = 1) CoP = P_{NEG}(state = 3)

(b)



(c)

```
  `define ACC RegA[31:16]
  `define RegM RegA[15:0]
  `define M RegA[0]
  `define M15 RegA[15]
  `define MC15 RegB[15]
module Mult16_wo_Cntrl_Sigs(CLK, St, Mplier, Mcand, Done, Product);
  input CLK, St;
  input [15:0] Mplier, Mcand;
  output reg Done;
  output [30:0] Product;

  reg [2:0] state;
  reg [31:0] RegA;
  reg [15:0] RegB;
  reg Sign;
  wire [15:0] Compout;
  wire Pneg;
  wire [16:0] Addout;

  integer Cntr;

  initial begin
    Done = 0;
```

```
    state = 0;
    RegA = 0;
    RegB = 0;
    Sign = 0;
    Cntr = 0;
  end

  assign Pneg = Sign ^ `MC15;
  assign Product = RegA[30:0];
  assign Compout = (`MC15 == 0)? RegB : (!RegB);
  assign Addout = {RegA[31], `ACC} + {Compout[15], Compout} + {16'd0,
`MC15};

  always @(posedge CLK)
  begin
    case(state)
    0: begin
      if(St == 1) begin
        `ACC <= 0;
        `RegM <= Mplier;
        Sign <= Mplier[15];
        RegB <= Mcand;
        state <= 1;
      end
    end
    1: begin
      if(`M15 == 1) begin
        `RegM <= (!Mplier) + 1;
      end
      state <= 2;
    end
    2: begin
      if(Cntr != 15) begin
        state <= 2;
        Cntr <= Cntr + 1;
      end
      else begin
        state <= 3;
        Cntr <= 0;
      end
      if(`M == 0) begin
        RegA <= {1'b0, RegA[31:1]};
      end
      else begin
        `ACC <= Addout[16:1];
        `RegM <= {Addout[0], RegA[15:1]};
      end
    end
    3: begin
      if(Pneg == 1) begin
        RegA <= (!RegA) + 1;
      end
      state <= 4;
    end
    4: begin
      Done <= 1;
      if(St == 1) begin
        state <= 4;
      end
      else begin
        Done <= 0;
        state <= 0;
      end
```

```
      end
      default: begin
      end
      endcase
    end

  endmodule
```

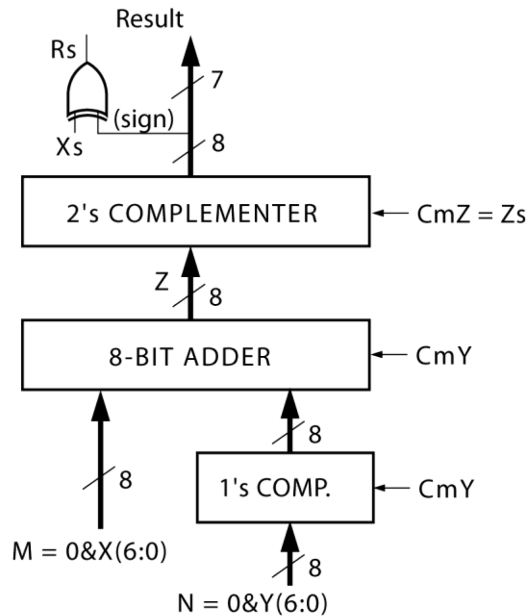
(d)

```
    `define ACC RegA[31:16]
    `define RegM RegA[15:0]
    `define M RegA[0]
    `define M15 RegA[15]
    `define MC15 RegB[15]
  module mult16_w_Cntrl_Sigs(CLK, St, Mplier, Mcand, Done, Product);
    input CLK, St;
    input [15:0] Mplier, Mcand;
    output reg Done;
    output [30:0] Product;

    reg [2:0] state, nextstate;
    reg [31:0] RegA;
    reg [15:0] RegB;
    reg Sign;
    reg Cnt, Sh, AdSh, Load, CoM, CoP;
    wire [15:0] Compout;
    wire Pneg;
    wire K;
    wire [16:0] Addout;

    integer Cntr;

    initial begin
      Done = 0;
      state = 0;
      nextstate = 0;
      RegA = 0;
      RegB = 0;
      Sign = 0;
      Cnt = 0;
      Sh = 0;
      AdSh = 0;
      Load = 0;
      CoM = 0;
      CoP = 0;
      Cntr = 0;
    end

    assign Compout = (`MC15 == 1)? (!RegB) : RegB;
    assign Addout = {RegA[31], `ACC} + {Compout[15], Compout} + {16'd0,
  `MC15};
    assign Pneg = Sign ^ `MC15;
    assign K = (Cntr == 15)? 1 : 0;
    assign Product = RegA[30:0];

    always @(state, St, `M15, K, `M, Pneg)
    begin
      Load = 0; CoM = 0; AdSh = 0; Sh = 0;
      Cnt = 0; CoP = 0; Done = 0;
      case(state)
      0: begin
        if(St == 1) begin
          Load = 1;
          nextstate = 1;
```

```
      end
      else begin
        nextstate = 0;
      end
    end
    1: begin
      if(`M15 == 1) begin
        CoM = 1;
      end
      nextstate = 2;
    end
    2: begin
      if(K == 1) begin
        nextstate = 3;
      end
      else begin
        nextstate = 2;
      end
      if(`M == 0) begin
        Sh = 1;
        Cnt = 1;
      end
      else begin
        AdSh = 1;
        Cnt = 1;
      end
    end
    3: begin
      if(Pneg == 1) begin
        CoP = 1;
      end
      nextstate = 4;
    end
    4: begin
      Done = 1;
      if(St == 1) begin
        nextstate = 4;
      end
      else begin
        nextstate = 0;
      end
    end
    default: begin
    end
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    if(Cnt == 1) begin
      if(Cntr == 15) begin
        Cntr <= 0;
      end
      else begin
        Cntr <= Cntr + 1;
      end
    end
    if(Sh == 1) begin
      `ACC <= {1'b0, RegA[31:17]};
      `RegM <= {RegA[16], RegA[15:1]};
    end
    if(AdSh == 1) begin
```

```
      `ACC <= Addout[16:1];
      `RegM <= {Addout[0], RegA[15:1]};
    end
    if(Load == 1) begin
      `ACC <= 0;
      `RegM <= Mplier;
      Sign <= Mplier[15];
      RegB <= Mcand;
    end
    if(CoM == 1) begin
      `RegM <= (!Mplier) + 1;
    end
    if(CoP == 1) begin
      RegA <= (!RegA) + 1;
    end
  end

endmodule
```

4.23 (a)



(b) In the following, Xs is the sign of X, M is the magnitude of X, Ys is the sign of Y, N is the magnitude of Y, N* is the 2's complement of N, and R is the final result. Consider the following 8 cases:

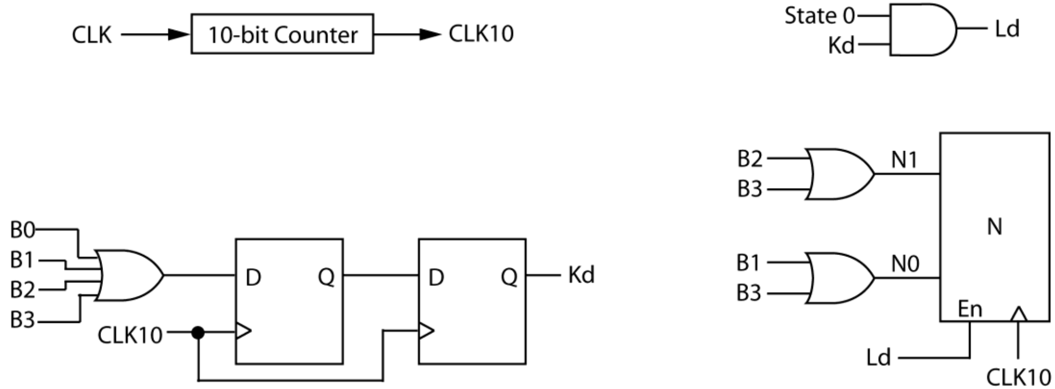
Sub	X_S	Y_S	CmY	formula for Z
0	0	0	0	M + N
0	0	1	1	M + (-N) → M + N*
0	1	0	1	-M + N = -(M - N) → M + N*, change sign of R
0	1	1	0	-M + (-N) = -(M + N) → M + N, change sign of R
1	0	0	1	M - N → M + N*
1	0	1	0	M - (-N) → M + N
1	1	0	0	-M + N = -(M - N) → M + N*, change sign of R
1	1	1	1	-M + (-N) = -(M + N) → M + N*, change sign of R

Method:
(1) If CmY = 1, take 2's complement of N. $CmY = X_S \oplus Y_S \oplus Sub$
(2) If sign of the adder output (Z_S) is 1, take 2's complement of Z. $CmZ = Z_S$.
(3) If $X_S = 1$, change sign of R.

(c) R_S = sign of result
$V = Sub'(X_S Y_S R_S' + X_S' Y_S' R_S) + Sub(X_S' Y_S R_S + X_S Y_S' R_S')$

4.24 **(a)**

(b)



4.25 **(a)** Truth table and resulting equations acquired used a CAD program (such as *LogicAid*).

R_0	R_1	R_2	R_3	C_0	C_1	C_2	C_3	N_3	N_2	N_1	N_0
1	0	0	0	1	0	0	0	0	0	0	1
1	0	0	0	0	1	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	1
1	0	0	0	0	0	0	1	1	0	1	0
0	1	0	0	1	0	0	0	0	1	0	0
0	1	0	0	0	1	0	0	0	1	0	1
0	1	0	0	0	0	1	0	0	1	1	0
0	1	0	0	0	0	0	1	1	0	1	1
0	0	1	0	1	0	0	0	0	1	1	1
0	0	1	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	0	1	0	0	1
0	0	1	0	0	0	0	1	1	1	0	0
0	0	0	1	1	0	0	0	1	1	1	0
0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	1	0	0	1	0	1	1	1	1
0	0	0	1	0	0	0	1	1	1	0	1

Logic equations for the decoder:

$N_3 = C_3 + R_2 C_0' + R_3 C_1'$

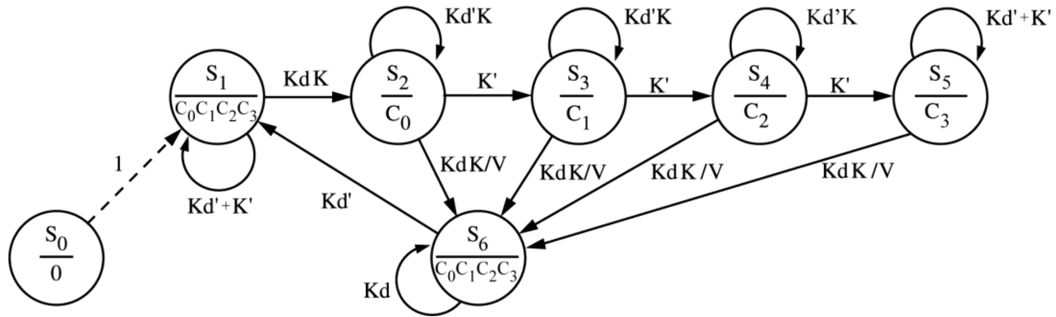
$N_2 = R_1 C_3' + R_3 C_1' + R_2 C_1' C_2'$

$N_1 = R_0 C_0' + R_2' C_2 + R_1 C_3 + R_0' R_1' C_0$

$N_0 = R_1' R_3' C_1' C_3' + R_1 C_0' C_2' + R_3 C_0' C_1'$

(b) Debouncing circuit is the same as pictured in Figure 4-44.

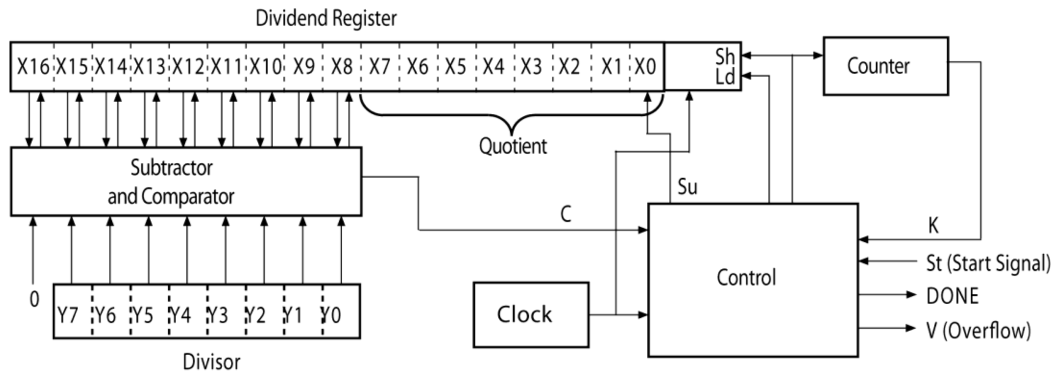
(c)



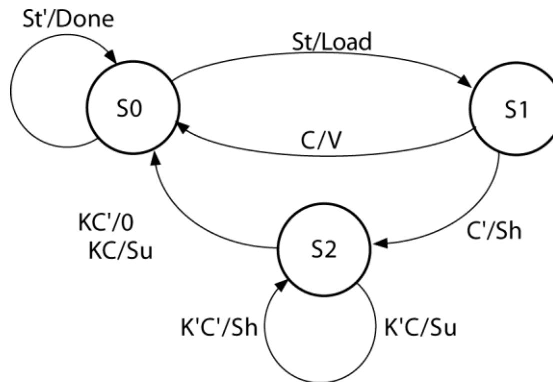
(d)
```
module scanner(R0, R1, R2, R3, CLK, C0, C1, C2, C3, N0, N1, N2, N3, V);
  input R0, R1, R2, R3, CLK;
  output reg C0, C1, C2, C3, V;
  output N0, N1, N2, N3;

  reg [2:0] state, nextstate;
  reg QA, Kd;
  wire K;

  initial begin
    C0 = 0;
    C1 = 0;
    C2 = 0;
    C3 = 0;
    V = 0;
    QA = 0;
    Kd = 0;
    state = 0;
    nextstate = 0;
  end

  assign K = R0 | R1 | R2 | R3;
  assign N3 = C3 | (R2 & !C0) | (R3 & !C1);
  assign N2 = (R1 & !C3) | (R3 & !C1) | (R2 & !C1 & !C2);
  assign N1 = (R0 & !C0) | (!R2 & C2) | (R1 & C3) | (!R0 & !R1 & C0);
  assign N0 = (!R1 & !R3 & !C1 & !C3) | (R1 & !C0 & !C2) | (R3 & !C0 &
!C1);

  always @(state, R0, R1, R2, R3, C0, C1, C2, C3, K, Kd, QA)
  begin
    C0 = 0; C1 = 0; C2 = 0; C3 = 0; V = 0;
    case(state)
    0: begin
      nextstate = 1;
    end
    1: begin
      C0 = 1; C1 = 1; C2 = 1; C3 = 1;
      if(Kd & K == 1)
        nextstate = 2;
      else
        nextstate = 1;
    end
    2: begin
      C0 = 1;
      if(Kd & K == 1) begin
        V = 1;
        nextstate = 6;
      end
      else if(K == 0)
```

```
        nextstate = 3;
      else
        nextstate = 2;
    end
    3: begin
      C1 = 1;
      if(Kd & K == 1) begin
        V = 1;
        nextstate = 6;
      end
      else if(K == 0)
        nextstate = 4;
      else
        nextstate = 3;
    end
    4: begin
      C2 = 1;
      if(Kd & K == 1) begin
        V = 1;
        nextstate = 6;
      end
      else if(K == 0)
        nextstate = 5;
      else
        nextstate = 4;
    end
    5: begin
      C3 = 1;
      if(Kd & K == 1) begin
        V = 1;
        nextstate = 6;
      end
      else
        nextstate = 5;
    end
    6: begin
      C0 = 1; C1 = 1; C2 = 1; C3 = 1;
      if(Kd == 0)
        nextstate = 1;
      else
        nextstate = 6;
    end
    default: begin
    end
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    QA <= K;
    Kd <= QA;
  end

endmodule
```

4.26 (a)



(b)



(c)

```
`define ACC X[16:8]
module Div16(CLK, St, Dvend, Dvsor, Quotient, V, Done);
  input CLK, St;
  input [15:0] Dvend;
  input [7:0] Dvsor;
  output [7:0] Quotient;
  output reg V, Done;

  reg [1:0] state, nextstate;
  reg [16:0] X;
  reg [8:0] Y;
  reg Sh, Su, Load;
  wire C, K;

  integer Count;

  initial begin
    V = 0;
    Done = 0;
    state = 0;
    nextstate = 0;
    X = 0;
    Y = 0;
    Sh = 0;
    Su = 0;
    Load = 0;
    Count = 0;
  end

  assign Quotient = X[7:0];
  assign K = (Count == 8)? 1 : 0;
  assign C = ((Y <= `ACC) && (state != 0)) ? 1 : 0;
```

```
  always @(state, St, C, K)
  begin
    Load = 0; Sh = 0; Su = 0; V = 0; Done = 0;
    case(state)
    0: begin
      if(St == 1) begin
        Load = 1;
        nextstate = 1;
      end
      else begin
        nextstate = 0;
        Done = 1;
      end
    end
    1: begin
      if(C == 1) begin
        V = 1;
        nextstate = 0;
      end
      else begin
        Sh = 1;
        nextstate = 2;
      end
    end
    2: begin
      if((C == 1) && (K == 0)) begin
        Su = 1;
        nextstate = 2;
      end
      else if((C == 0) && (K == 0)) begin
        Sh = 1;
        nextstate = 2;
      end
      else if((C == 0) && (K == 1)) begin
        nextstate = 0;
      end
      else begin
        Su = 1;
        nextstate = 0;
      end
    end
    default: begin
    end
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    if(Load == 1) begin
      Count <= 0; X <= {1'b0, Dvend}; Y <= {1'b0, Dvsor};
    end
    if(Sh == 1) begin
      X <= {X[15:0], 1'b0}; Count <= Count + 1;
    end
    if(Su == 1) begin
      `ACC <= `ACC - Y;
      X[0] <= 1'b1;
    end
  end

endmodule
```

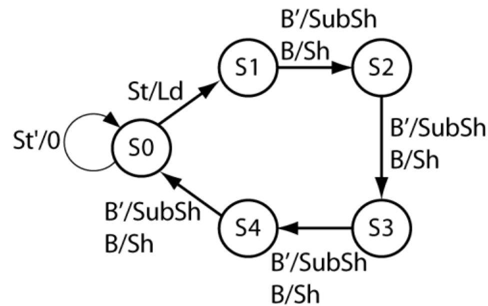
(d)
```
module div16test;
  reg [15:0] dividendarr [1:6];
  reg [7:0] divisorarr [1:6];

  reg CLK, St;
  reg [15:0] Dvend;
  reg [7:0] Dvsor;
  wire V, Done;

  integer Count;
  integer i;
  parameter N = 6;

  initial begin
    dividendarr[1] = 16'b0000000000000000;
    dividendarr[2] = 16'b1111111111111111;
    dividendarr[3] = 16'b0000001000000000;
    dividendarr[4] = 16'b0000000111111110;
    dividendarr[5] = 16'b0000000000000100;
    dividendarr[6] = 16'b0000000000000110;
    divisorarr[1] = 8'b00000001;
    divisorarr[2] = 8'b11111111;
    divisorarr[3] = 8'b00000010;
    divisorarr[4] = 8'b00000010;
    divisorarr[5] = 8'b00000010;
    divisorarr[6] = 8'b00000010;
    CLK = 0;
    St = 0;
    Dvend = 0;
    Dvsor = 0;
    Count = 0;
    i = 0;
  end

  always #10 CLK = !CLK;

  always
  begin
    for(i = 1; i <= N; i = i+1) begin
      St = 1;
      Dvend = dividendarr[i];
      Dvsor = divisorarr[i];
      @(posedge CLK)
      St = 0;
      wait((Done == 1) || (V == 1));
      @(posedge CLK)
      Count = i;
    end
  end

  Div16 Div(CLK, St, Dvend, Dvsor, Quotient, V, Done);
endmodule
```

4.27
```
module divider(St, clk, Input, Quotient);
  input St, clk;
  input [15:0] Input;
  output [15:0] Quotient;

  reg Ld1, Ld2, Su, Sh;
  reg [15:0] Divisor, Dividend, ACC;
  reg [4:0] Counter;
```

```
reg [1:0] state, nextstate;
wire K, B;

initial begin
  Ld1 = 0;
  Ld2 = 0;
  Su = 0;
  Sh = 0;
  Divisor = 0;
  Dividend = 0;
  Counter = 0;
  state = 0;
  nextstate = 0;
end

assign Quotient = Dividend;
assign K = (Counter == 16)? 1 : 0;
assign B = (ACC < Divisor)? 1 : 0;

always @(posedge clk)
begin
  state <= nextstate;
  if(Ld1 == 1)
    Divisor <= Input;
  if(Ld2 == 1) begin
    Dividend <= Input;
    ACC <= 16'b0000000000000000;
  end
  if(Sh == 1) begin
    ACC <= {ACC[14:0], Dividend[15]};
    Dividend <= Dividend << 1;
    Counter <= Counter + 1;
  end
  if(Su == 1) begin
    ACC <= ACC - Divisor;
    Dividend[0] <= 1;
  end
end

always @(state, St, B, K)
begin
  Ld1 = 0; Ld2 = 0; Su = 0; Sh = 0;
  case(state)
  0: begin
    if(St == 1) begin
      nextstate = 1;
      Ld1 = 1;
    end
    else
      nextstate = 0;
  end
  1: begin
    nextstate = 2;
    Ld2 = 1;
  end
  2: begin
    nextstate = 3;
    Sh = 1;
  end
  3: begin
    if((B == 1) && (K == 0)) begin
      nextstate = 3;
      Sh = 1;
```

```
          end
          else if((B == 0) && (K == 0)) begin
            nextstate = 3;
            Su = 1;
          end
          else if((B == 1) && (K == 1)) begin
            nextstate = 0;
          end
          else begin
            nextstate = 0;
            Su = 1;
          end
        end
        endcase
      end

    endmodule
```

4.28 (a)

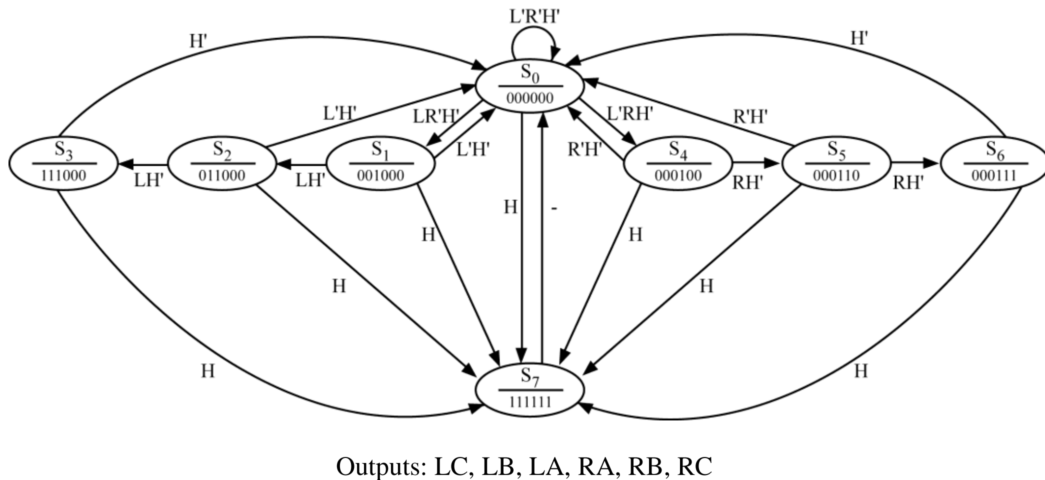


(b)

```
module divu(dividend, divisor, St, clk, quotient);
  input [7:0] dividend;
  input [3:0] divisor;
  input St, clk;
  output [3:0] quotient;

  reg [2:0] state, nextstate;
  reg [7:0] X;
  reg [3:0] Y;
  wire [4:0] Subout;
  wire sub;

  initial begin
    state = 0;
    nextstate = 0;
    X = 0;
    Y = 0;
  end

  assign quotient = X[3:0];
  assign sub = (X[7:3] >= Y)? 1 : 0;
  assign Subout = X[7:3] - Y;

  always @(posedge clk)
  begin
    case(state)
    0: begin
      if(St == 1) begin
        X <= dividend;
        Y <= divisor;
```

```
        state <= 1;
      end
      else
        state <= 0;
    end
    1,2,3,4: begin
      if(sub == 1) begin
        X[7:4] <= Subout[3:0];
        X[3:0] <= {X[2:0],1'b1};
      end
      else
        X <= {X[6:0], 1'b0};
      if(state < 4)
        state <= state + 1;
      else
        state <= 0;
    end
    default: begin
    end
    endcase
  end

endmodule
```

4.29 **(a)**



Outputs: LC, LB, LA, RA, RB, RC

(b) First, assign LC = Q_1, LB = Q_2, LA = Q_3, RA = Q_4, RB = Q_5, RC = Q_6.
So S_0 = 000000, S_1 = 001000, S_2 = 011000, etc.

This state machine has too many state variables to use Karnaugh maps. Instead, we will write down equations for each flip-flop by inspection.

First consider Q_1. $Q_1 = 1$ in states S_3 or S_7 only.

- S_7 is reached whenever H = 1 and we are not already in S_7: $H(Q_1Q_2Q_3Q_4Q_5Q_6)'$. But S_7 is the only state in which both $Q_3 = 1$ and $Q_4 = 1$, so assuming we are always in a valid state, we can use $H(Q_3Q_4)' = HQ_3' + HQ_4'$. Note: Any combination of one left light and one right light will also work, i.e. $HQ_1' + HQ_5'$.

- S_3 is reached whenever we are in S_2 and L = 1 while H = 0: $LH'Q_1'Q_2Q_3Q_4'Q_5'Q_6'$. But Q_3 = 1 whenever Q_2 = 1, and $Q_4 = Q_5 = Q_6 = 0$ whenever Q_1 = 0. So we can use $LH'Q_1'Q_2$.

- So $D_1 = LH'Q_1'Q_2 + HQ_3' + HQ_4' = LQ_1'Q_2 + HQ_3' + HQ_4'$ (using X + X'Y = X + Y). Similarly Q_2 = 1 in states S_3, S_2, and S_7 only.

- S_3 and S_2 are reached whenever we are in S_2 or S_1 and L = 1 while H = 0. $LH'Q_1'Q_2Q_3Q_4'Q_5'Q_6' + LH'Q_1'Q_2'Q_3Q_4'Q_5'Q_6' = LH'Q_1'Q_3Q_4'Q_5'Q_6'$. But again, $Q_4 = Q_5 = Q_6 = 0$ whenever $Q_1 = 0$, so $D_2 = LQ_1'Q_3 + HQ_3' + HQ_4'$ We can also get by inspection: $D_3 = LQ_1'Q_4' + HQ_3' + HQ_4'$; $D_4 = RQ_3'Q_6' + HQ_3' + HQ_4'$; $D_5 = RQ_4Q_6' + HQ_3' + HQ_4'$; $D_6 = RQ_5Q_6' + HQ_3' + HQ_4'$

(c)

State	LRH =	000	001	010	011	100	101	110	111	LC	LB	LA	RA	RB	RC
S_0		S_0	S_7	S_4	S_7	S_1	S_7	-	-	0	0	0	0	0	0
S_1		S_0	S_7	S_0	S_7	S_2	S_7	-	-	0	0	1	0	0	0
S_2		S_0	S_7	S_0	S_7	S_3	S_7	-	-	0	1	1	0	0	0
S_3		S_0	S_7	S_0	S_7	S_0	S_7	-	-	1	1	1	0	0	0
S_4		S_0	S_7	S_5	S_7	S_0	S_7	-	-	0	0	0	1	0	0
S_5		S_0	S_7	S_6	S_7	S_0	S_7	-	-	0	0	0	1	1	0
S_6		S_0	S_7	S_0	S_7	S_0	S_7	-	-	0	0	0	1	1	1
S_7		S_0	S_0	S_0	S_0	S_0	S_0	-	-	1	1	1	1	1	1

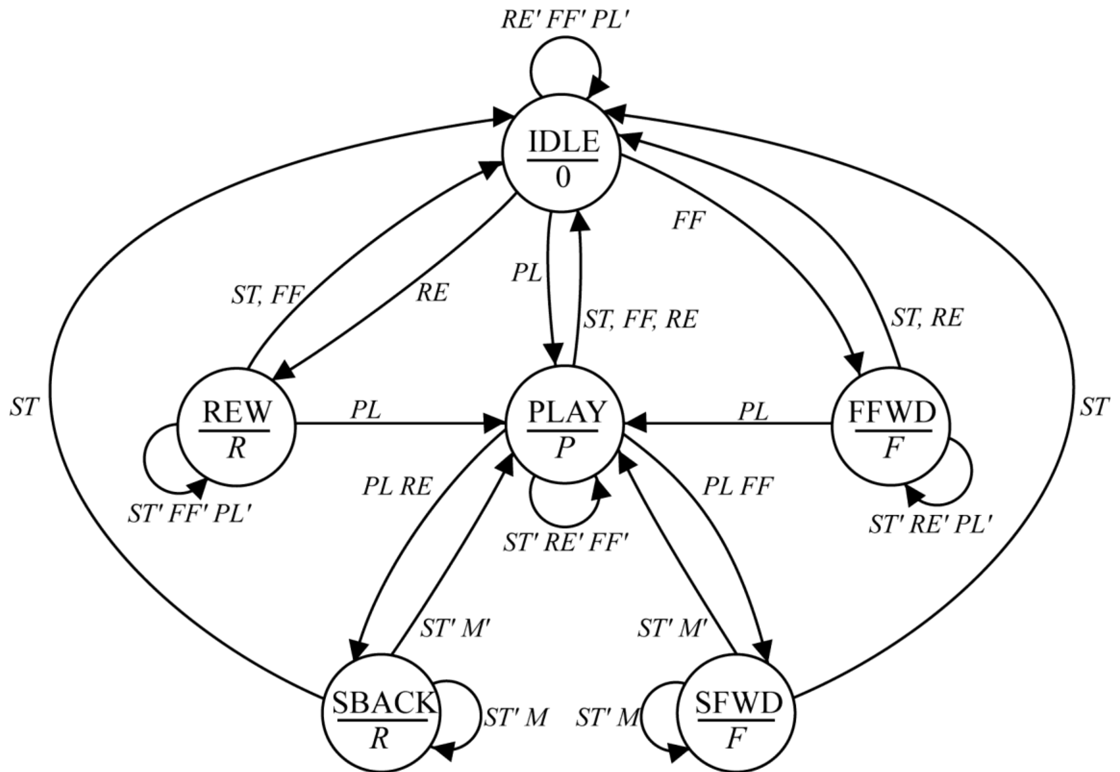
I. (S_0, S_1, S_2, S_3, S_4, S_5, S_6) for S_7 in LRH = 001, 011, 101
(S_1, S_2, S_3, S_6, S_7) for S_0 in LRH = 010
(S_3, S_4, S_5, S_6, S_7) for S_0 in LRH = 100

II. Every state matches S_0 and S_7. But S_0 and S_7 match the best, so
(S_0, S_7) × (many times)

III. (S_1, S_2, S_3, S_7) (S_4, S_5, S_6, S_7) etc.

From LogicAid:
So $D_1 = HQ_2 + RQ_1Q_2Q_3' + HQ_3 + LQ_1'Q_2'Q_3 + HQ_1' + RQ_1'Q_2'Q_3'$
$D_2 = RH'Q_1'Q_2'Q_3' + RH'Q_1Q_2 + LH'Q_1'Q_2'Q_3'$
$D_3 = LH'Q_1'Q_2Q_3' + LH'Q_1'Q_2'Q_3 + RH'Q_1Q_2$
$LC = Q_1Q_2'$; $LB = Q_1Q_2' + Q_2'Q_3$; $LA = Q_1Q_2' + Q_2'Q_3 + Q_1'Q_2Q_3'$
$RC = Q_1Q_2'Q_3' + Q_1'Q_2Q_3$; $RB = Q_1Q_2'Q_3' + Q_2Q_3$; $RA = Q_1Q_3' + Q_2Q_3$
Other minimum solutions can be found for D_2 and D_3 with this assignment.

4.30 (a)

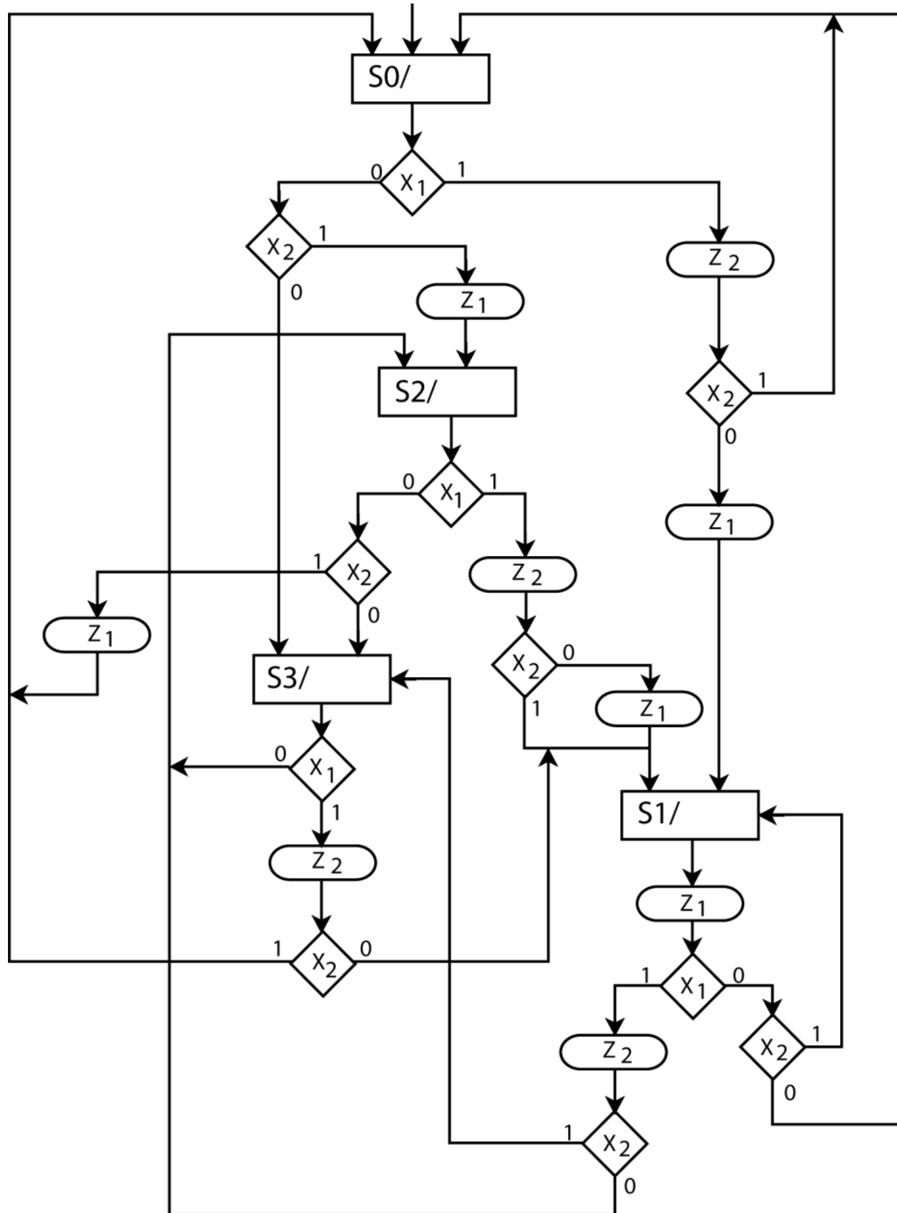


(b)
```
module tapeplayer(ST, RE, PL, FF, M, CLK, R, P, F);
  input ST, RE, PL, FF, M, CLK;
  output reg R, P, F;

  reg [2:0] state, nextstate;

  parameter idle = 0;
  parameter rew = 1;
  parameter play = 2;
  parameter ffwd = 3;
  parameter sback = 4;
  parameter sfwd = 5;

  initial begin
    R = 0;
    P = 0;
    F = 0;
    state = 0;
    nextstate = 0;
  end

  always @(state, ST, RE, PL, FF, M)
  begin
    R = 0; P = 0; F = 0;
    case(state)
    0: begin                    //idle
      if(RE == 1)
        nextstate = rew;
      else if(PL == 1)
        nextstate = play;
      else if(FF == 1)
        nextstate = ffwd;
```

```
      else
        nextstate = idle;
    end
    1: begin                    //rew
      R = 1;
      if((ST == 1) || (FF == 1))
        nextstate = idle;
      else if(PL == 1)
        nextstate = play;
      else
        nextstate = rew;
    end
    2: begin                    //play
      P = 1;
      if((ST == 1) || (RE == 1) || (FF == 1))
        nextstate = idle;
      else if((PL == 1) && (RE == 1))
        nextstate = sback;
      else if((PL == 1) && (FF == 1))
        nextstate = sfwd;
      else
        nextstate = play;
    end
    3: begin                    //ffwd
      F = 1;
      if((ST == 1) || (RE == 1))
        nextstate = idle;
      else if(PL == 1)
        nextstate = play;
      else
        nextstate = ffwd;
    end
    4: begin                    //sback
      R = 1;
      if(ST == 1)
        nextstate = idle;
      else if(M == 0)
        nextstate = play;
      else
        nextstate = sback;
    end
    5: begin                    //sfwd
      F = 1;
      if(ST == 1)
        nextstate = idle;
      else if(M == 0)
        nextstate = play;
      else
        nextstate = sfwd;
    end
    default: begin
    end
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
  end

endmodule
```

Chapter 5: SM Charts and Microprogramming

5.1 (a)



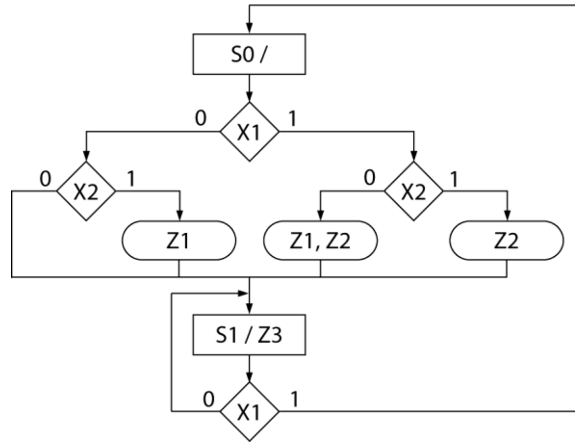
(b)

```
module SM_desc(X1, X2, CLK, Z1, Z2);
  input X1, X2, CLK;
  output reg Z1, Z2;

  reg [1:0] state, nextstate;

  initial begin
    state = 2'b00;
    nextstate = 2'b00;
    Z1 = 1'b0;
    Z2 = 1'b0;
  end
```

```
always @(state, X1, X2)
begin
  Z1 = 1'b0;
  Z2 = 1'b0;
  case(state)
    0: begin
      if(X1 == 1'b1) begin
        Z2 = 1'b1;
        if(X2 == 1'b1)
          nextstate = 2'b00;
        else begin
          Z1 = 1'b1;
          nextstate = 2'b01;
        end
      end
      else if(X2 == 1'b1) begin
        Z1 = 1'b1;
        nextstate = 2'b10;
      end
      else
        nextstate = 2'b11;
    end
    1: begin
      Z1 = 1'b1;
      if(X1 == 1'b1) begin
        Z2 = 1'b1;
        if(X2 == 1'b1)
          nextstate = 2'b11;
        else
          nextstate = 2'b10;
      end
      else if(X2 == 1'b1)
        nextstate = 2'b01;
      else
        nextstate = 2'b00;
    end
    2: begin
      if(X1 == 1'b1) begin
        Z2 = 1'b1;
        nextstate = 2'b01;
        if(X2 == 1'b0)
          Z1 = 1'b1;
      end
      else if(X2 == 1'b1) begin
        Z1 = 1'b1;
        nextstate = 2'b00;
      end
      else begin
        nextstate = 2'b11;
      end
    end
    3: begin
      if(X1 == 1'b1) begin
        Z2 = 1'b1;
        if(X2 == 1'b1)
          nextstate = 2'b00;
        else
          nextstate = 2'b01;
      end
      else
        nextstate = 2'b10;
    end
```

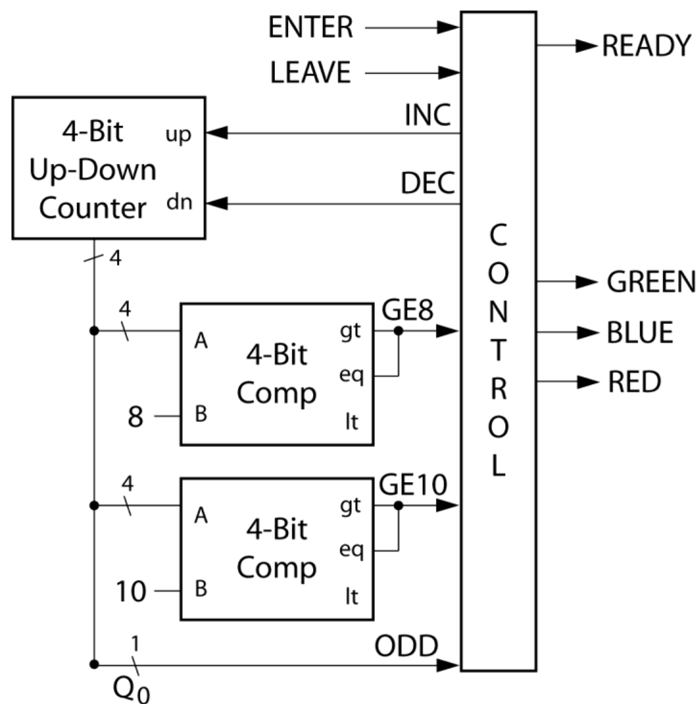
```
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
  end

endmodule
```

5.2

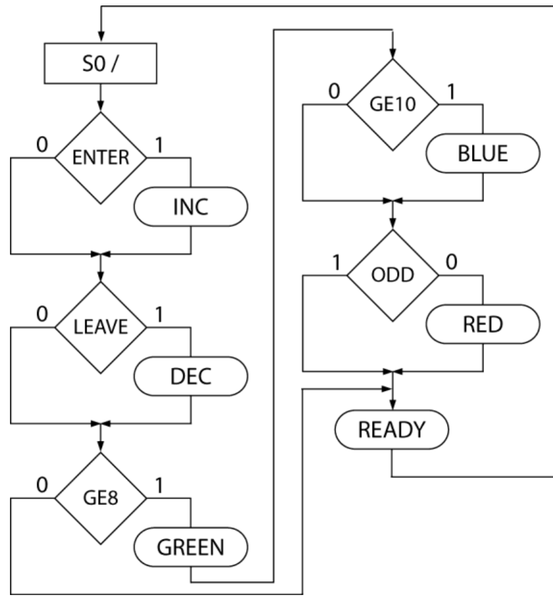


Z1 and Z2 are Mealy outputs; Z3 is a Moore output.

5.3 (a)

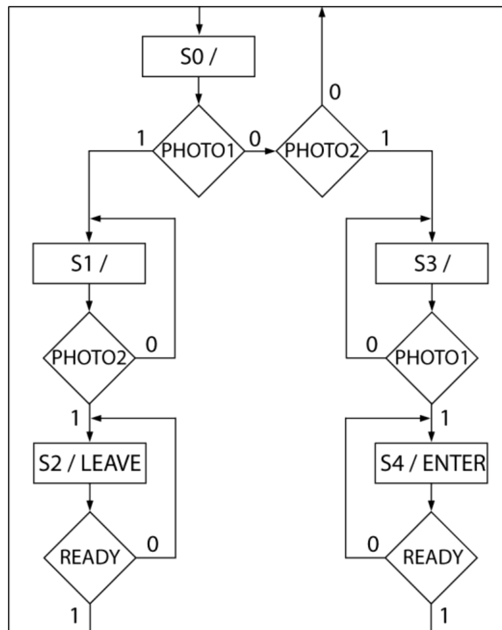


(b)

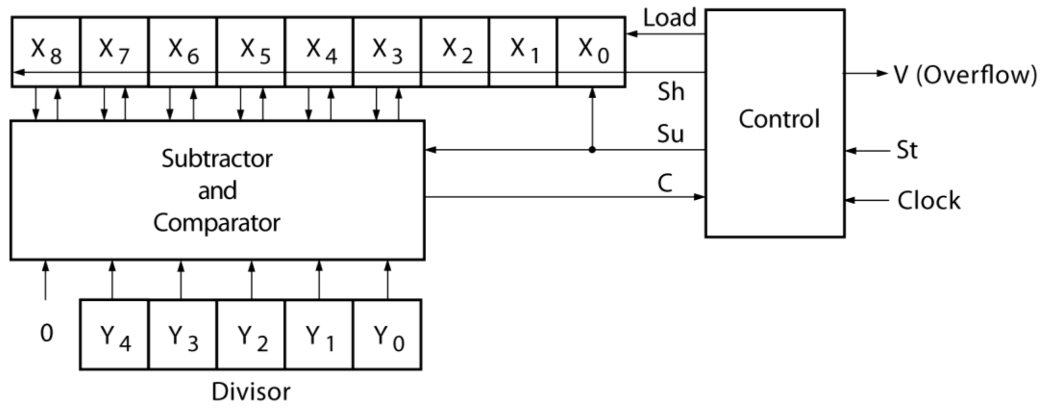


ENTER, *LEAVE*, *READY*, *RED*, *GREEN*, *BLUE* used as described in the problem
INC: increment counter by 1
DEC: decrement counter by 1
GE8: counter value is greater then or equal to 8
GE10: counter value is greater then or equal to 10
ODD: counter value is odd

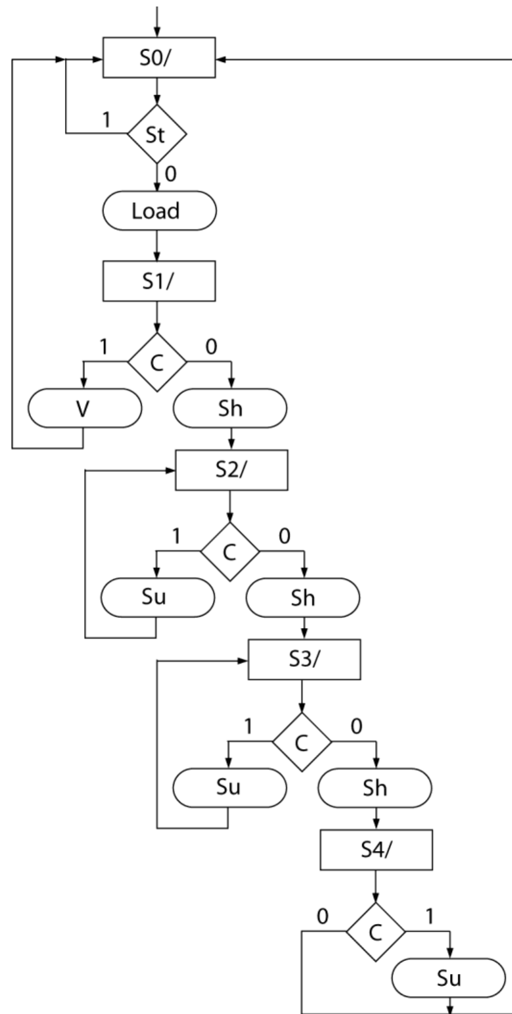
(c)



5.4 (a)



(b)



(c)

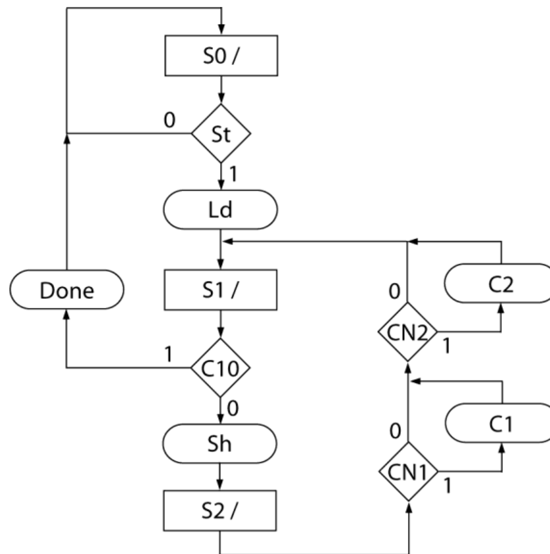
```
module DIV_SM_desc(CLK, St, Divisor, Dividend, V, Quotient);
  input CLK, St;
  input [4:0] Divisor;
  input [7:0] Dividend;
  output reg V;
```

```
output [2:0] Quotient;

reg [2:0] state, nextstate;
reg Load, Sh, Su;
reg [4:0] RegB;
reg [8:0] ACC;
wire [5:0] ACCin;
wire C;
wire [6:0] Diff;
wire [5:0] Subout;

initial begin
  state = 3'b000;
  nextstate = 3'b000;
  ACC = 9'b000000000;
  RegB = 5'b00000;
end

assign Quotient = ACC[2:0];
assign Diff = {1'b0, ACC[8:3]} - RegB;
assign Subout = Diff[5:0];
assign C = ~Diff[6];

always @(state, St, C)
begin
  Load = 1'b0;
  Sh = 1'b0;
  Su = 1'b0;
  V = 1'b0;
  case(state)
  0: begin
    if(St == 1'b1) begin
      Load = 1'b1;
      nextstate = 3'b001;
    end
    else
      nextstate = 3'b000;
  end
  1: begin
    if(C == 1'b1) begin
      V = 1'b1;
      nextstate = 3'b000;
    end
    else begin
      Sh = 1'b1;
      nextstate = 3'b010;
    end
  end
  2, 3: begin
    if(C == 1'b1) begin
      Su = 1'b1;
      nextstate = state;
    end
    else begin
      Sh = 1'b1;
      nextstate = state + 3'b001;
    end
  end
  4: begin
    nextstate = 3'b000;
    if(C == 1'b1)
      Su = 1'b1;
  end
```

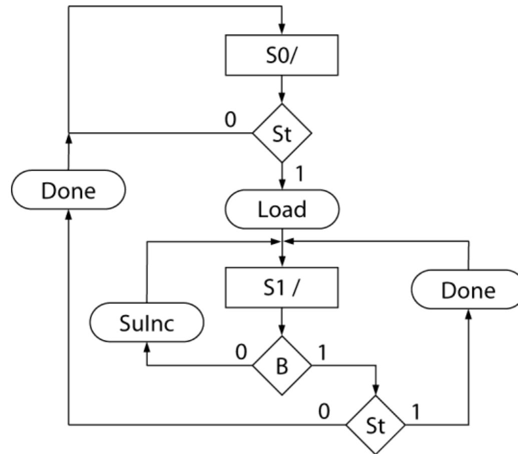
```
      default: begin
      end
      endcase
    end

    always @(posedge CLK)
    begin
      state <= nextstate;
      if(Load == 1'b1) begin
        ACC <= {1'b0, Dividend};
        RegB <= Divisor;
      end
      else if(Sh == 1'b1)
        ACC <= {ACC[7:0], 1'b0};
      else if(Su == 1'b1) begin
        ACC[8:3] <= Subout;
        ACC[0] <= 1'b0;
      end
      else begin
      end
    end

  endmodule
```

(d) add list *
add wave *
force Clk 0 0, 1 10 -repeat 20
force Dividend 01011101
force Divisor 10001
force St 1 5,0 15
run

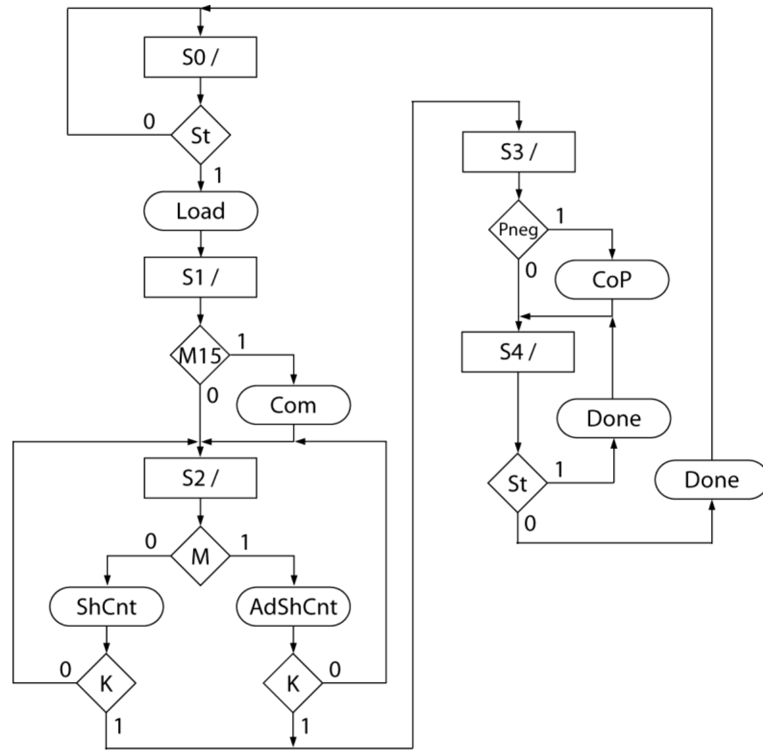
5.5



5.6



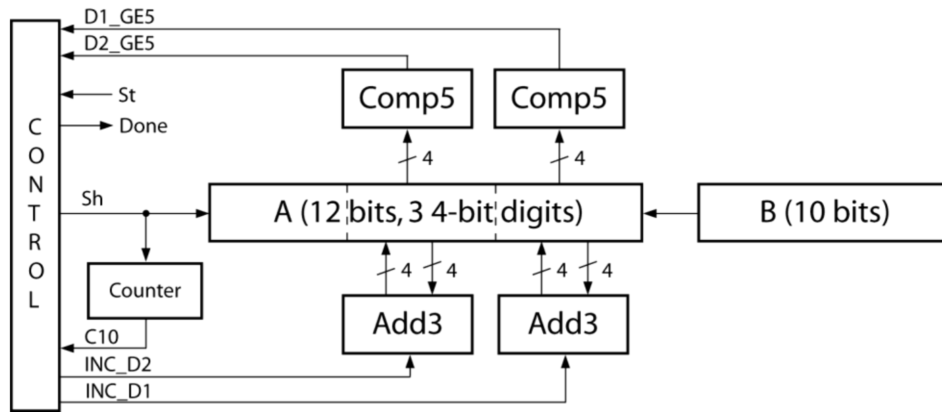
5.7



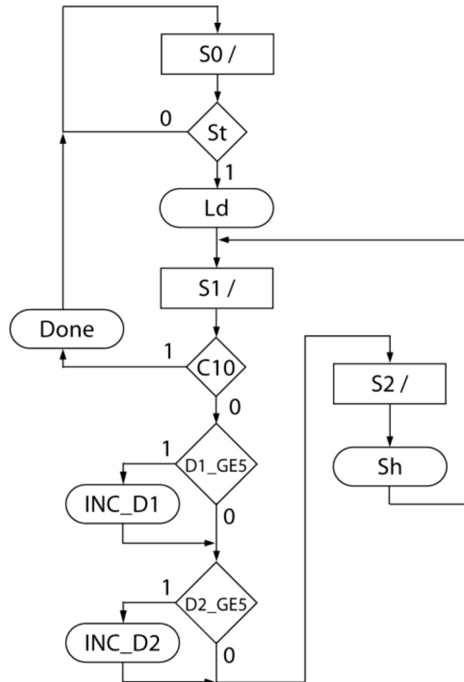
5.8 **(a)**

	A	B
Start		0100011101
Shift #1	0	100011101
Shift #2	01	00011101
Shift #3	010	0011101
Shift #4	0100	011101
Shift #5	0 1000	11101
Add	0 1011	11101
Shift #6	1 0111	1101
Add	1 1010	1101
Shift #7	11 0101	101
Add	11 1000	101
Shift #8	111 0001	01
Add	1010 0001	01
Shift #9	1 0100 0010	1
Shift #10	10 1000 0101	

(b)



(c)

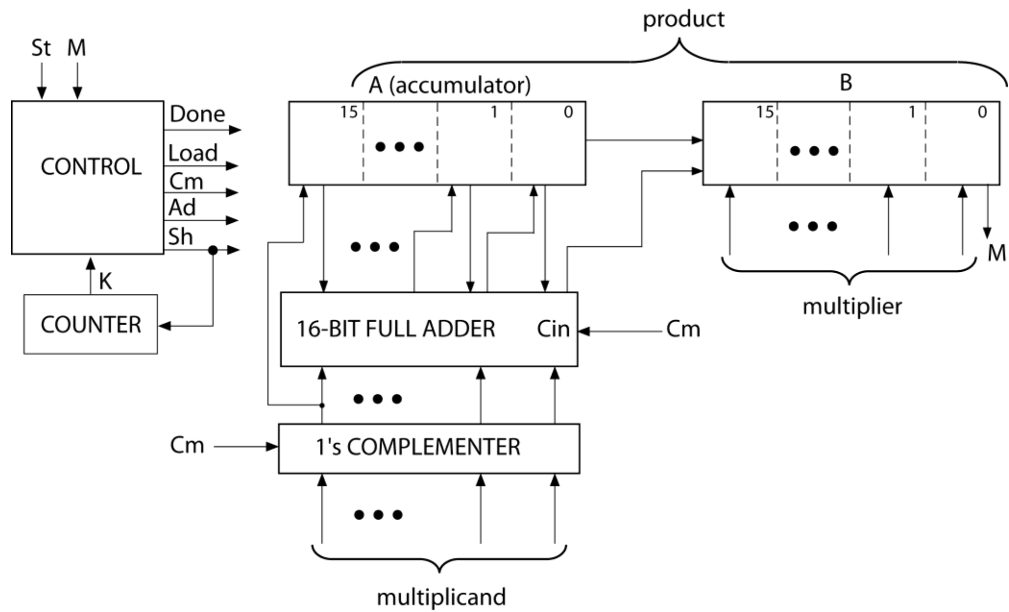


(d)

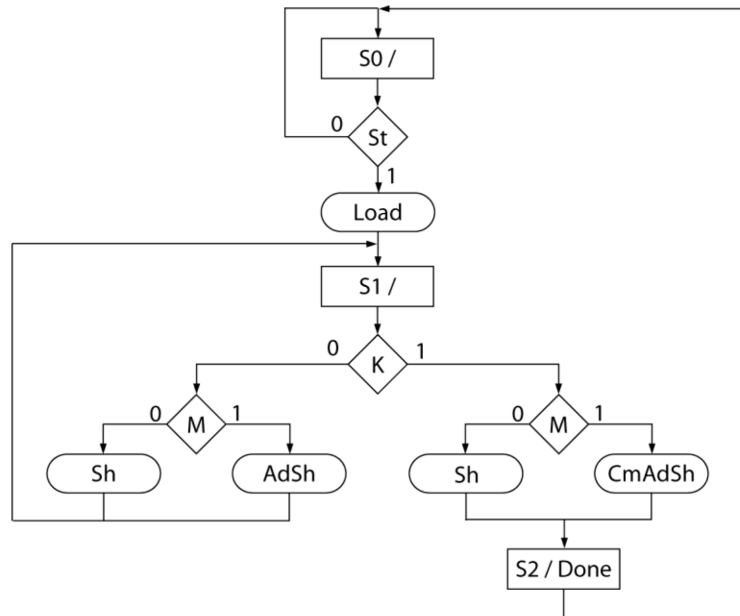
```
module P5_8(St, Clk, Number, A, Done);
  input St, Clk;
  input [9:0] Number;
  inout [11:0] A;
  output reg Done;

  reg [11:0] Atemp;
  reg [9:0] B;
  reg [1:0] state, nextstate;
  reg [3:0] cnt;
  reg Load, INC_D1, INC_D2, Sh;

  wire C10, D1_GE5, D2_GE5;

  initial begin
    Atemp = A;
    B = 10'b0000000000;
    state = 2'b00;
    nextstate = 2'b00;
    cnt = 4'b0000;
  end

  assign A = Atemp;
  assign C10 = (cnt == 4'b1010) ? 1'b1 : 1'b0;
  assign D1_GE5 = (A[3:0] >= 4'b0101)? 1'b1 : 1'b0;
  assign D2_GE5 = (A[7:4] >= 4'b0101)? 1'b1 : 1'b0;

  always @(state, St, C10, D1_GE5, D2_GE5)
  begin
    Load = 1'b0;
    INC_D1 = 1'b0;
    INC_D2 = 1'b0;
    Sh = 1'b0;
    Done = 1'b0;
    case(state)
    0: begin
      if(St == 1'b1) begin
        nextstate = 2'b01;
        Load = 1'b1;
      end
      else
        nextstate = 2'b00;
    end
    1: begin
      if(C10 == 1'b1) begin
        nextstate = 2'b00;
        Done = 1'b1;
      end
      else begin
        nextstate = 2'b10;
        if(D1_GE5 == 1'b1)
          INC_D1 = 1'b1;
        if(D2_GE5 == 1'b1)
          INC_D2 = 1'b1;
      end
    end
    2: begin
      Sh = 1'b1;
      nextstate = 2'b01;
    end
    default: begin
```

```
    end
    endcase
  end

  always @(posedge Clk)
  begin
    state <= nextstate;
    if(INC_D1 == 1'b1)
      Atemp[3:0] <= Atemp[3:0] + 4'b0011;
    if(INC_D2 == 1'b1)
      Atemp[7:4] <= Atemp[7:4] + 4'b0011;
    if(Sh == 1'b1) begin
      Atemp <= {Atemp[10:0], B[9]} ;
      cnt <= cnt + 4'b0001;
      B <= {B[8:0], 1'b0};
    end
    if(Load == 1'b1)
      B <= Number;
  end

endmodule
```

5.9 (a)



(b)

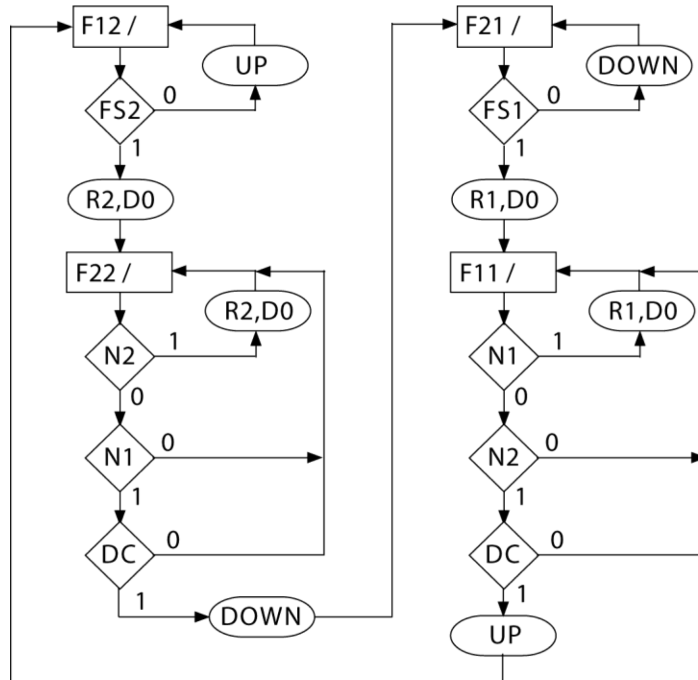


(c)

```
module P5_9(clk, St, Mplier, Mcand, Product, Done);
  input clk, St;
  input [15:0] Mplier, Mcand;
  output [30:0] Product;
  output Done;

  reg [1:0] State;
  reg [15:0] A, B;
  reg [4:0] Counter;
  wire K;
  wire M;
  wire [15:0] addout;

  initial begin
    State = 2'b00;
    A = 16'h0000;
    B = 16'h0000;
    Counter = 5'b00000;
  end

  assign M = B[0];
  assign Done = (State == 2'b10)? 1'b1: 1'b0;
  assign Product = {A[14:0], B};
  assign K = (Counter == 5'b01111)? 1'b1 : 1'b0;
  assign addout = (K == 1'b0)? (A + Mcand) : (A + (~Mcand) + 16'h0001);

  always @(posedge clk)
  begin
    case(State)
    0: begin
      if(St == 1'b1) begin
        A <= 16'h0000;
        B <= Mplier;
        State <= 2'b01;
        Counter <= 5'b00000;
      end
    end
```

```
    1: begin
      Counter <= Counter + 5'b00001;
      if(K == 1'b0) begin
        if(M == 1'b1) begin
          A <= {Mcand[15], addout[15:1]};
          B <= {addout[0], B[15:1]};
        end
        else begin
          A <= {A[15], A[15:1]};
          B <= {A[0], B[15:1]};
        end
      end
      else begin
        State <= 2'b10;
        if(M == 1'b1) begin
          A <= {(~Mcand[15]), addout[15:1]};
          B <= {addout[0], B[15:1]};
        end
        else begin
          A <= {A[15], A[15:1]};
          B <= {A[0], B[15:1]};
        end
      end
    end
    2: begin
      State <= 2'b00;
    end
    default: begin
    end
    endcase
  end

endmodule
```

5.10

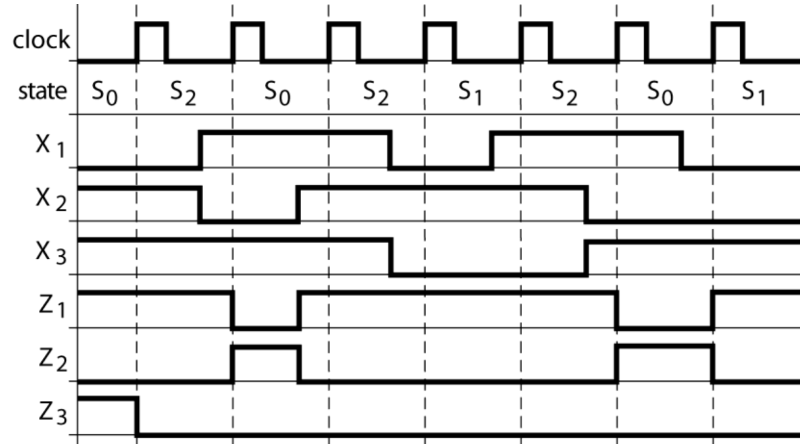


5.11

```
module test_el;
  reg CLK;
  reg CALL1, CALL2, FB1, FB2, FS1, FS2, DC;
  wire UP, DOWN, DO;

  initial begin
    CLK = 1'b0;
    CALL1 = 1'b0;
    CALL2 = 1'b0;
    FB1 = 1'b0;
    FB2 = 1'b0;
    FS1 = 1'b0;
    FS2 = 1'b0;
    DC = 1'b0;
  end

  elev_control eltest(CALL1, CALL2, FB1, FB2, FS1, FS2, DC, CLK, UP, DOWN,
DO);

  always
    #6000 CLK = ~CLK;

  always @(UP, DOWN)
  begin
    if(UP == 1'b1) begin
      if(FS1 == 1'b1) begin
        #3600 FS1 = 1'b0;
        #39600 FS2 = 1'b1;
      end
    end
    else if(DOWN == 1'b1) begin
      if(FS2 == 1'b1) begin
        #3600 FS2 = 1'b0;
        #39600 FS1 = 1'b1;
      end
    end
  end

  always @(posedge DO)
  begin
      DC = 1'b0;
      #18000;
      DC = 1'b1;
  end

  always
  begin
    CALL1 = 1'b1;
    #3600;
    CALL1 = 1'b0;
    #7200;
    FB2 = 1'b1;
    #3600;
    FB2 = 1'b0;
    #14400;
    FB1 = 1'b1;
    #3600;
    FB1 = 1'b0;
    #3600;
    CALL2 = 1'b1;
    #3600;
    CALL2 = 1'b0;
```

```
    #36000;
    FB2 = 1'b1;
    #3600;
    FB2 = 1'b0;
  end

endmodule
```

5.12 **(a)**



(b) $A^+ = A'BX_2 + A'B'X_2(X_1' + X_3) + \{AB\} = BX_2 + A'X_2(X_1' + X_3)$

$B^+ = A'B'(X_2' + X_1X_3') + AB'X_1' + A'BX_2' + \{AB\} = AX_1' + A'B'X_1X_3' + A'X_2'$

$Z_1 = A + B + X_2$

$Z_2 = A'B'X_2'$

$Z_3 = A'B'X_1'X_2$

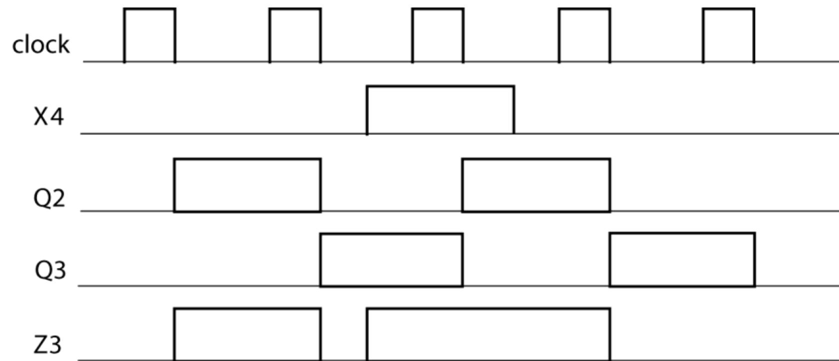
(c)

A	B	X_1	X_2	X_3	A^+	B^+	Z_1	Z_2	Z_3
-	1	-	1	-	1	0	0	0	0
0	-	0	1	-	1	0	0	0	0
0	-	-	1	1	1	0	0	0	0
1	-	0	-	-	0	1	0	0	0
0	0	1	-	0	0	1	0	0	0
0	-	-	0	-	0	1	0	0	0
1	-	-	-	-	0	0	1	0	0
-	1	-	-	-	0	0	1	0	0
-	-	-	1	-	0	0	1	0	0
0	0	-	0	-	0	0	0	1	0
0	0	0	1	-	0	0	0	0	1

(d) 32 words by 5 bits;

A	B	X_1	X_2	X_3	A^+	B^+	Z_1	Z_2	Z_3
0	0	0	0	0	0	1	0	1	0
0	0	0	0	1	0	1	0	1	0
0	0	0	1	0	1	0	0	0	1
0	0	0	1	1	1	0	1	0	1
0	0	1	0	0	0	1	0	1	0

5.13 **(a)**



(b) $Q_1^+ = Q_1X_1' + Q_3X_4'X_5$

$Q_2^+ = Q_2X_3 + Q_1X_1X_2' + Q_3X_4$

$Q_3^+ = Q_3X_4'X_5' + Q_1X_1X_2 + Q_2X_3'$

$Z_1 = Q_1X_1X_2'$

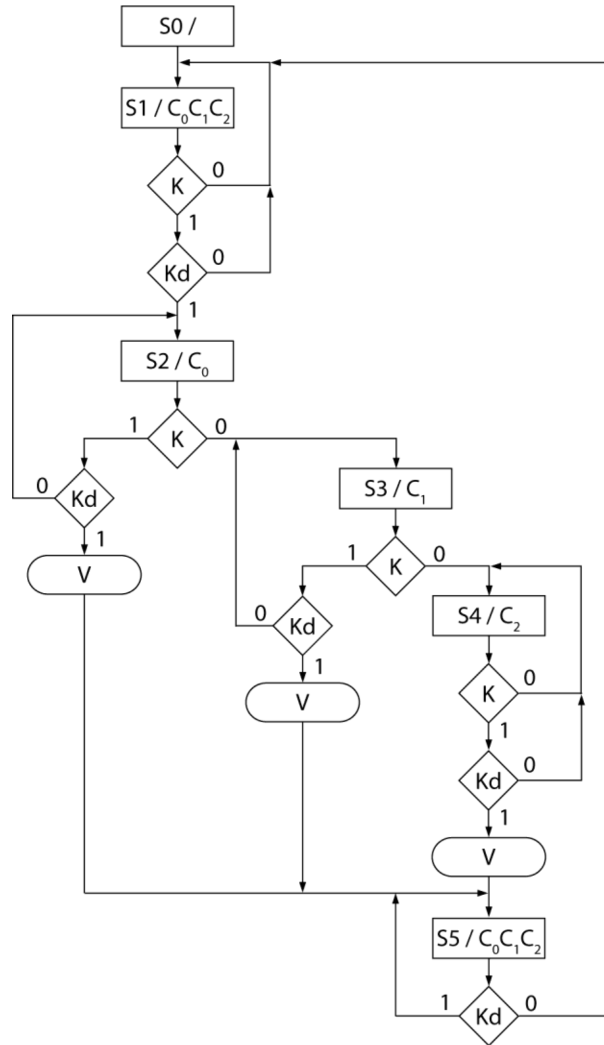
$Z_2 = Q_2X_3$

$Z_3 = Q_2 + Q_3X_4$

(c)

```
module Egns_Desc(X1, X2, X3, X4, X5, CLK, Z1, Z2, Z3);
  input X1, X2, X3, X4, X5, CLK;
  output Z1, Z2, Z3;

  reg Q1, Q2, Q3;

  initial begin
    Q1 = 1'b1;
    Q2 = 1'b0;
    Q3 = 1'b0;
  end

  assign Z1 = Q1 & X1 & (~X2);
  assign Z2 = Q2 & X3;
  assign Z3 = Q2 | (Q3 & X4);

  always @(negedge CLK)
  begin
    Q1 <= (Q1 & ~X1) | (Q3 & ~X4 & X5);
    Q2 <= (Q2 & X3) | (Q1 & X1 & ~X2) | (Q3 & X4);
    Q3 <= (Q3 & ~X4 & ~X5) | (Q1 & X1 & X2) | (Q2 & ~X3);
  end

endmodule
```

5.14 (a)



(b)

	A	B	C	K	Kd	A^+	B^+	C^+	C_0	C_1	C_2	V
S0	0	0	0	-	-	0	0	1	0	0	0	0
S1	0	0	1	1	1	0	1	0	1	1	1	0
	0	0	1	0	-	0	0	1	1	1	1	0
	0	0	1	-	0	0	0	1	1	1	1	0
S2		1	0	1	0	0	1	0	1	0	0	0
	0	1	0	1	1	1	0	1	1	0	0	1
	0	1	0	0	-	0	1	1	1	0	0	0
S3	0	1	1	1	0	0	1	1	0	1	0	0
	0	1	1	1	1	1	0	1	0	1	0	1
	0	1	1	0	-	1	0	0	0	1	0	0
S4	1	0	0	1	1	1	0	1	0	0	1	1
	1	0	0	0	-	1	0	0	0	0	1	0
	1	0	0	-	0	1	0	0	0	0	1	0
S5	1	0	1	-	0	0	0	1	1	1	1	0
	1	0	1	-	1	1	0	1	1	1	1	0

(c) $A^+ = A'BC'K\,Kd + A'BCK\,Kd + A'BCK' + AB'C' + AB'CKd$

(d) For state assignment S0 = 100000, S1 = 010000… and D flip-flops $Q_0Q_1Q_2Q_3Q_4Q_5$:

$Q_0^+ = 0$

$Q_1^+ = Q_0 + Q_1K' + Q1Kd' + Q_5Kd'$

$Q_2^+ = Q_1KKd + Q_2\ KKd'$

$Q_3^+ = Q_2K' + Q_3KKd'$

$Q_4^+ = Q_3K' + Q_4K' + Q_4Kd'$

$Q_5^+ = Q_2KKd + Q_3KKd + Q_4KKd + Q_5Kd$

$V = Q_2KKd + Q_3KKd + Q_4KKd$

$C_0 = Q_1 + Q_2 + Q_5$

$C_1 = Q_1 + Q_3 + Q_5$

$C_2 = Q_1 + Q_4 + Q_5$

5.15 **(a)**

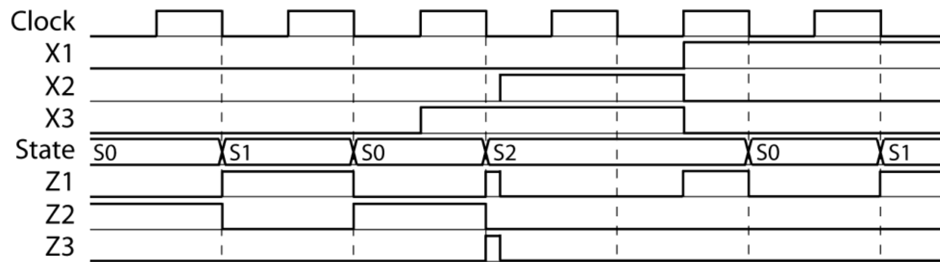
```
module P5_15(X1, X2, X3, Clk, Z1, Z2, Z3);
  input X1, X2, X3, Clk;
  output reg Z1, Z2, Z3;

  reg [1:0] state, nextstate;

  initial begin
    state = 2'b00;
    nextstate = 2'b00;
  end

  always @(state, X1, X2, X3)
  begin
    Z1 = 1'b0;
    Z2 = 1'b0;
    Z3 = 1'b0;
    case(state)
    0: begin
      if(X1 == 1'b1)
        nextstate = 2'b01;
      else begin
        Z2 = 1'b1;
        if(X2 == 1'b1)
          Z3 = 1'b1;
        if(X3 == 1'b1)
          nextstate = 2'b10;
        else
          nextstate = 2'b01;
      end
    end
    1: begin
      nextstate = 2'b00;
      Z1 = 1'b1;
    end
    2: begin
      if(X2 == 1'b1)
        nextstate = 2'b10;
      else begin
        Z1 = 1'b1;
        if(X1 == 1'b1)
          nextstate = 2'b00;
        else begin
          Z3 = 1'b1;
          nextstate = 2'b01;
        end
      end
```

```
        end
        default: begin
        end
        endcase
      end

      always @(negedge Clk)
      begin
        state <= nextstate;
      end

    endmodule
```

(b)

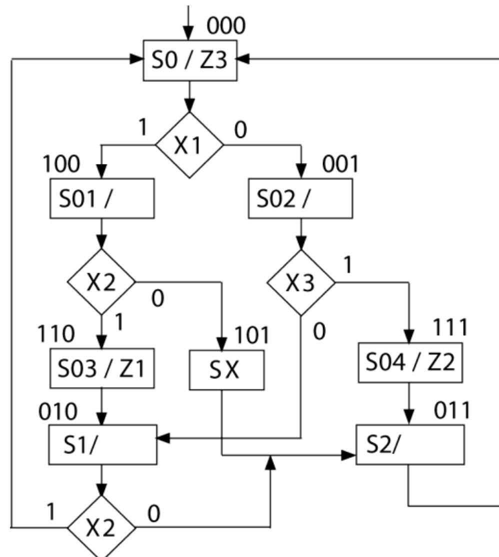
	A	B	X_1	X_2	X_3	A^+	B^+	Z_1	Z_2	Z_3
S0	0	0	1	-	-	0	1	0	0	0
	0	0	0	0	0	0	1	0	1	0
	0	0	0	0	1	1	0	0	1	0
	0	0	0	1	0	0	1	0	1	1
	0	0	0	1	1	1	0	0	1	1
S1	0	1	-	-	-	0	0	1	0	0
S2	1	0	-	1	-	1	0	0	0	0
	1	0	1	0	-	0	0	1	0	0
	1	0	0	0	-	0	1	1	0	1

(c)



5.16 **(a)** Block diagram is similar to Figure 5-33 with MUX inputs of 1, X1, X2, and X3; ROM outputs of Z1, Z2 and Z3; 2 bits for test, and 3 bits for the NST and counter.

(b)



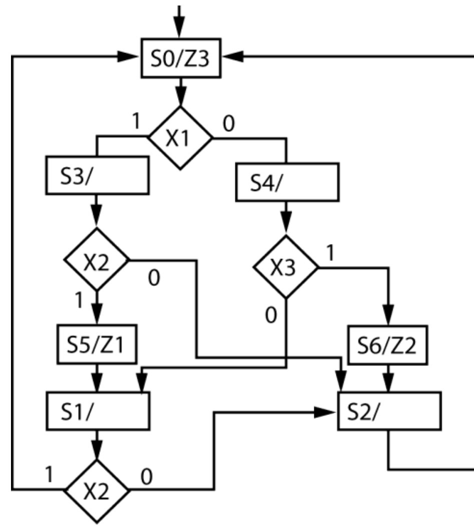
(c)

State	$Q_1Q_2Q_3$	Test	NST	Z1	Z2	Z3
S0	000	01	100	0	0	1
S02	001	11	111	0	0	0
S1	010	10	000	0	0	0
S2	011	00	000	0	0	0
S01	100	10	110	0	0	0
Sx	101	00	011	0	0	0
S03	110	00	010	1	0	0
S04	111	00	011	0	1	0

(d)

```
module P5_16(X1, X2, X3, CLK, Z1, Z2, Z3);
  input X1, X2, X3, CLK;
  output Z1, Z2, Z3;

  reg [2:0] PST;
  reg [7:0] ROM [0:7];
  wire [7:0] ROM_Out;
  wire Load;

  initial begin
    PST = 3'b000;
    ROM[0] = 8'b01100001;
    ROM[1] = 8'b11111000;
    ROM[2] = 8'b10000000;
    ROM[3] = 8'b00000000;
    ROM[4] = 8'b10110000;
    ROM[5] = 8'b00011000;
    ROM[6] = 8'b00010100;
    ROM[7] = 8'b00011010;
  end

  assign ROM_OUT = ROM[PST];
  assign Z3 = ROM_Out[2];
  assign Z2 = ROM_Out[1];
  assign Z1 = ROM_Out[0];
  assign Load = (ROM_Out[1] == 1'b0)? ((ROM_Out[0] == 1'b0)? 1'b1 : X1)
:
                                     ((ROM_Out[0] == 1'b0)? X2 : X3);

  always @(posedge CLK)
  begin
    if(Load == 1'b1)
      PST <= ROM_Out[5:3]; // NST
    else
      PST <= PST + 1;
  end
endmodule
```

5.17 **(a)**



T1 T2	
00	1
01	X1
10	X2
11	X3

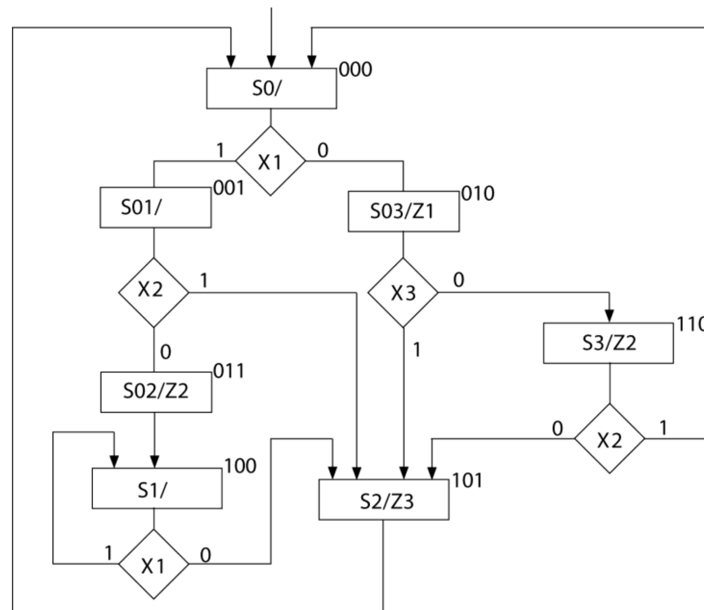
(b)

$Q_1Q_2Q_3$	TEST	NSF	NST	Z1	Z2	Z3
000	01	S4	S3	0	0	1
001	10	S2	S0	0	0	0
010	00	S0	S0	0	0	0
011	10	S2	S5	0	0	0
100	11	S1	S6	0	0	0
101	00	S1	S1	1	0	0
110	00	S2	S2	0	1	0

(c) 7×11 or 8×11

(d) $2^5 \times 5$

5.18 **(a)**



(b)

State	A B C	Test	NSF	NST	Z1	Z2	Z3
S0	000	01	010	001	0	0	0
S01	001	10	011	101	0	0	0
S03	010	11	110	101	1	0	0
S02	011	00	100	100	0	1	0
S1	100	01	101	100	0	0	0
S2	101	00	000	000	0	0	1
S3	110	10	101	000	0	1	0

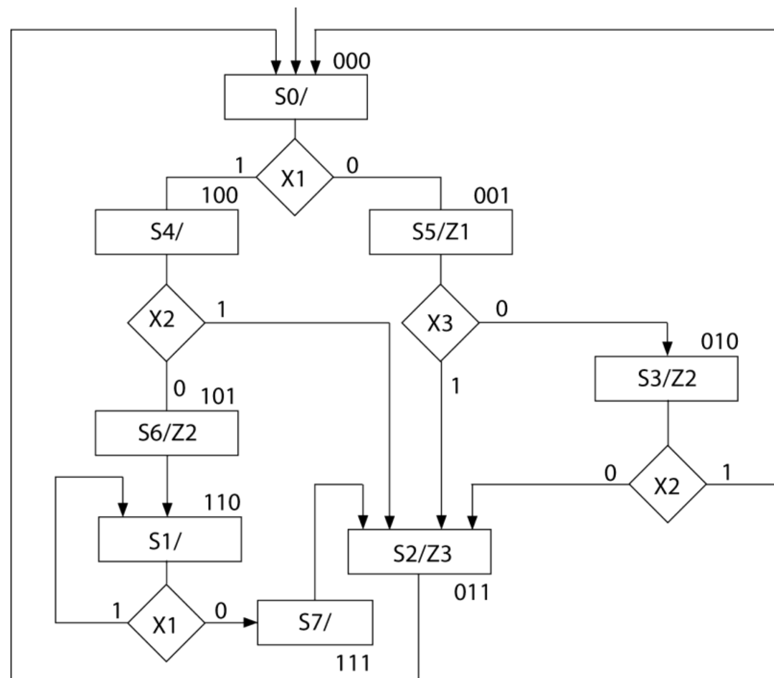
(c) Block diagram is similar to Figure 5-29 with inputs of X_1 and X_2; PLA outputs of Z_1, Z_2 and Z_3; and a 3-bit, 2-to-1 MUX to select NSF or NST.

5.19 **(a)** 1. Only Moore outputs; 2. Only 1 decision box per state; 3. NSF for each state should be state + 1

(b) Same as Solution 5.16 (b)

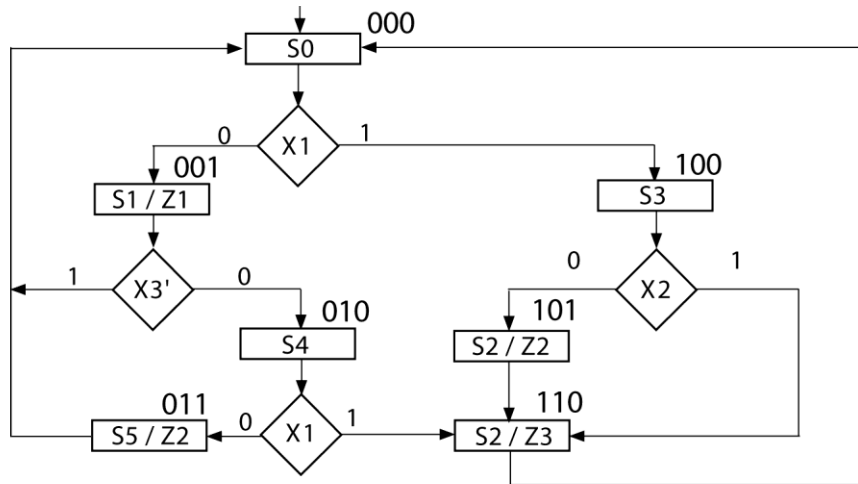
5.20 **(a)** 1. Only Moore outputs; 2. Only 1 decision box per state; 3. NSF for each state should be state + 1

(b)



5.21 **(a)** Block diagram is similar to Figure 5-29 with MUX inputs of 1, X1, X2, and X3'; ROM outputs of Z1, Z2 and Z3; and 3 bits for the NST and counter.

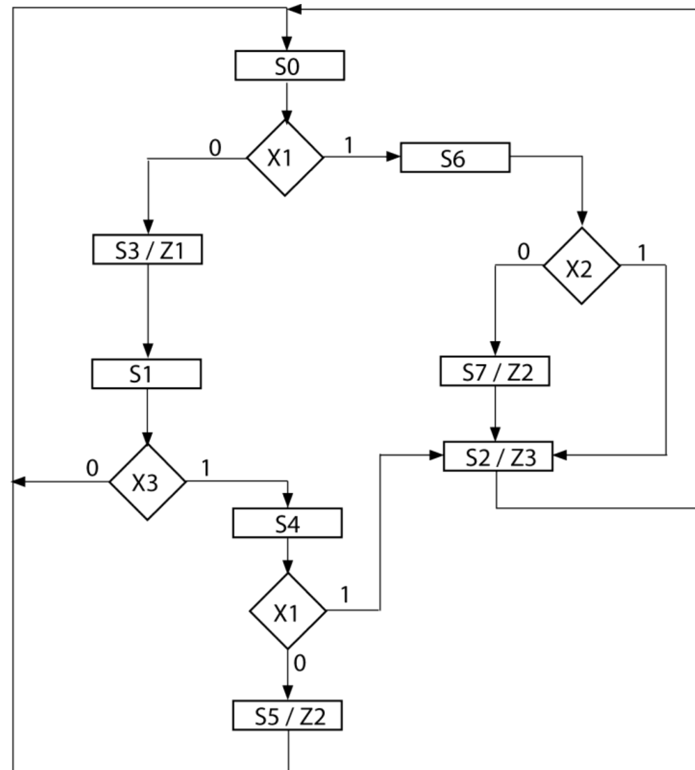
(b)



(c)

A B C	Test	NST	Z1	Z2	Z3
000	01	100	0	0	0
001	11	000	1	0	0
010	01	110	0	0	0
011	00	000	0	1	0
100	10	110	0	0	0
101	00	110	0	1	0
110	00	000	0	0	1

5.22 **(a)** 1. Convert Mealy outputs to Moore outputs; 2. Only one input tested per state



(b)

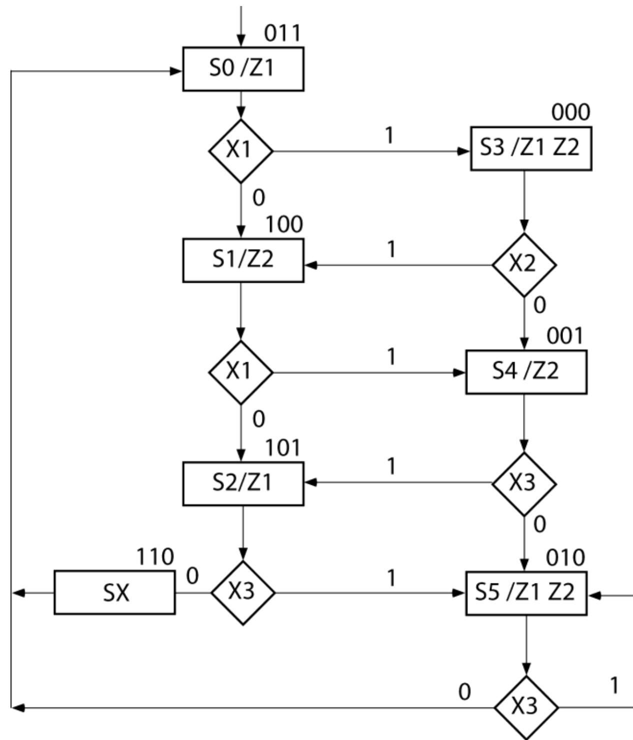
Addr	Test	NSF	NST	Z1	Z2	Z3
S0	00	S3	S6	0	0	0
S3	11	S1	S1	1	0	0
S1	10	S0	S4	0	0	0
S4	00	S5	S2	0	0	0
S5	11	S0	S0	0	1	0
S6	01	S7	S2	0	0	0
S7	11	S2	S2	0	1	0
S2	11	S0	S0	0	0	1

Test 00 – X1; Test 01 – X2; Test 10 – X3; Test 11 – 1

(c) There are 8 address locations and each have 11 bits (2 test, 3 NSF, 3 NST, 3 outputs) so there are 8×11 bits needed.

(d) 2^5 inputs are needed (2-bit FF state, 3 inputs) and 5 outputs are needed (2 for next state, 3 outputs) so $2^5 \times 5$ bits.

5.23 **(a)**



(b) Test 00: 1; Test 01: X1; Test 10: X2; Test 11: X3

Addr	Test	NST	Z1	Z2
000	10	100	1	1
001	11	101	0	1
010	11	010	1	1
011	01	000	1	0
100	01	001	0	1
101	11	010	1	0
110	00	011	0	0

(c) The circuit has 2^6 possible inputs (3-bit FF state, 3 inputs) and 5 output functions (3-bit next state, 2 outputs) so a $2^6 \times 5$ bit ROM is required.

5.24 **(a)** For flip-flops AB,

$A^+ = A'B'X_1X_3 + A'BX_2'$

$B^+ = A'B'X_1' + A'B'X_1X_3'$

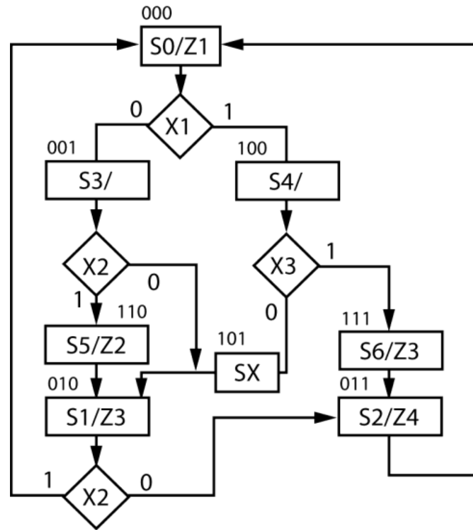
$Z_1 = A'B'$

$Z_2 = A'B'X_1'X_2$

$Z_3 = A'B'X_1X_3 + A'B$

$Z_4 = AB'$

(b)

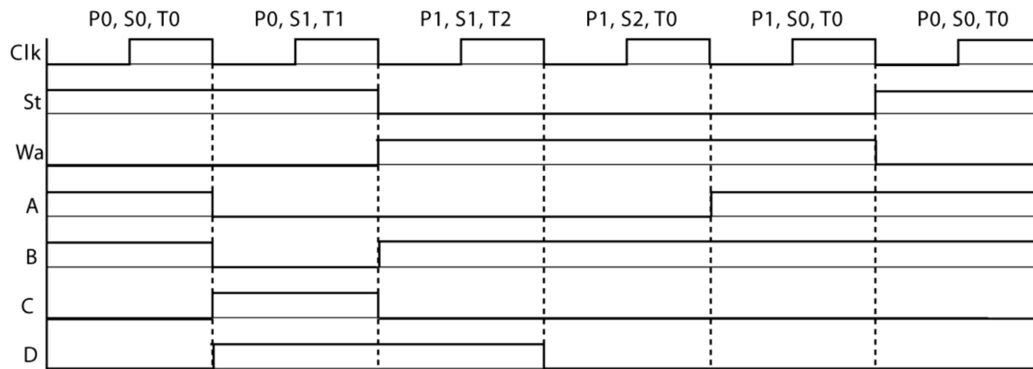


(c) Test 00: 1; Test 01: X1; Test 10: X2; Test 11: X3

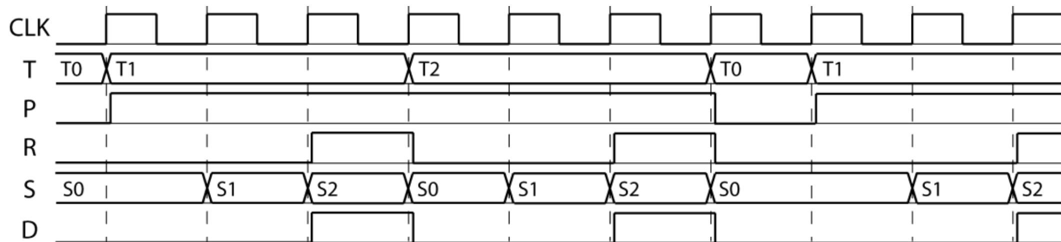
Addr	Test	NST	Z1	Z2	Z3	Z4
000	01	100	1	0	0	0
001	10	110	0	0	0	0
010	10	011	0	0	1	0
011	00	000	0	0	0	1
100	10	111	0	0	0	0
101	00	010	0	0	0	0
110	00	010	0	1	0	0
111	00	011	0	0	1	0

(d) 2^3 states, 2 test bits, 3 NST bits, 4 outputs: $2^3 \times 9$

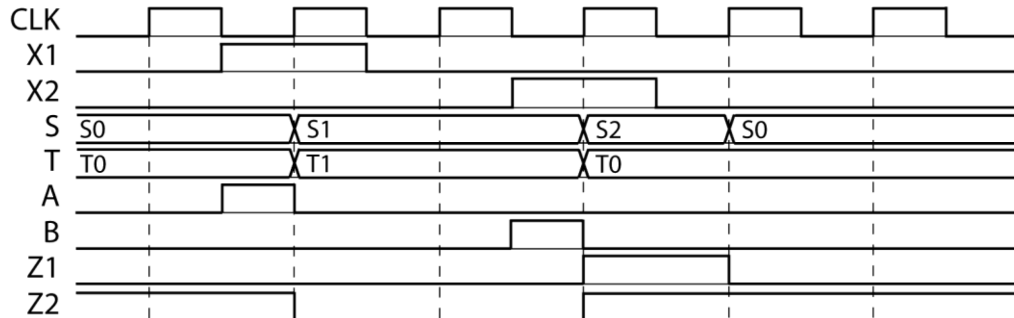
5.25



5.26



5.27 **(a)**



(b) State Assignment: S0: 100; S1: 010; S2: 001. For D flip-flops $Q_0Q_1Q_2$:

$Q_0^+ = Q_0X_1' + Q_2$

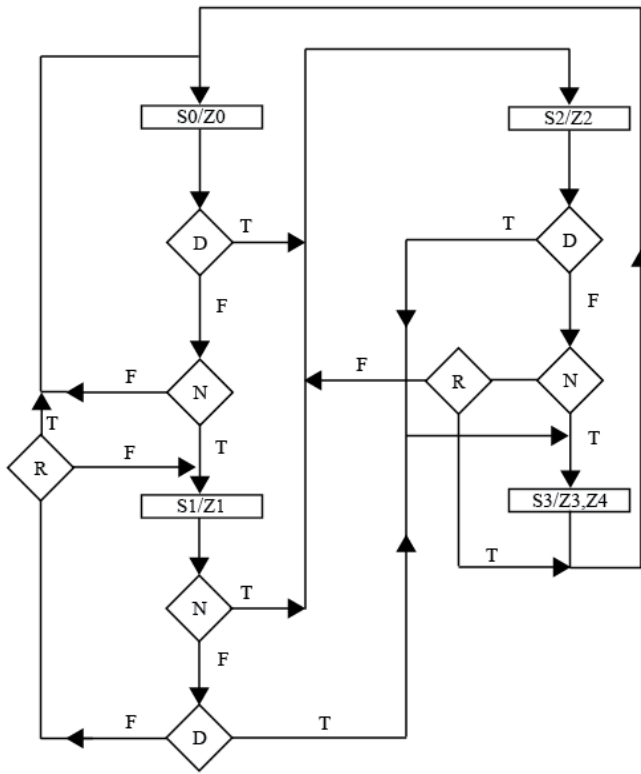
$Q_1^+ = Q_0 X_1 + Q_1 B'$

$Q_2^+ = Q_1 B$

$A = Q_0 X_1$

$Z_1 = Q_2$

5.28 **(a)**



Note: R = Reset

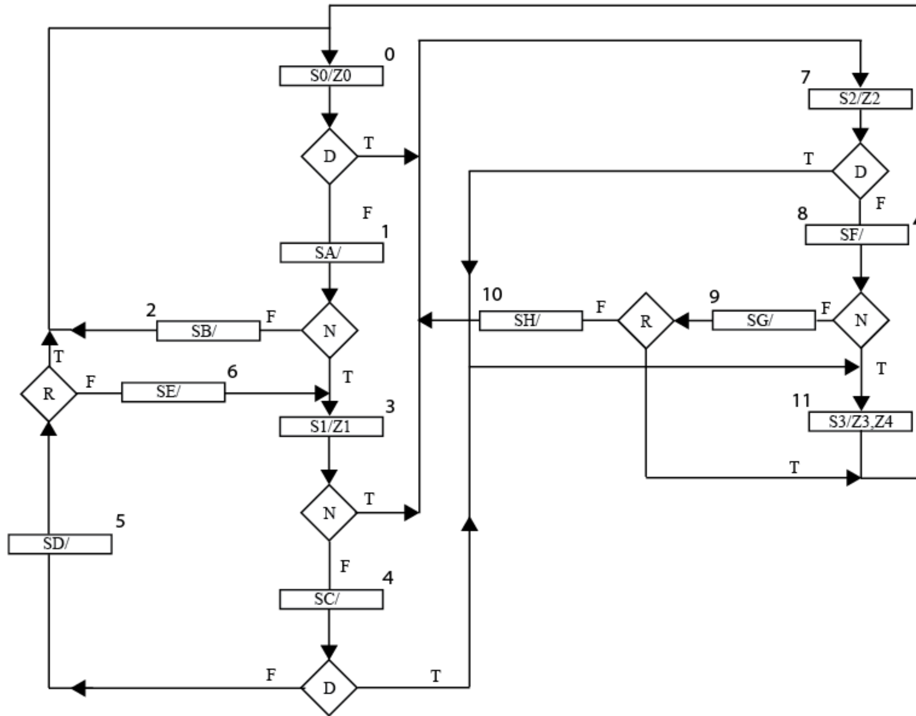
(b) State Assignment: S0: 00; S1: 01; S2: 10; S3: 11. For D flip-flops AB:

$A^+ = A'BN + A'D + AB'R' + AB'N + B'D$
$B^+ = B'D'N + A'BN'R' + A'BDN' + AB'D$

	A	B	D	N	R	A^+	B^+	Z_0	Z_1	Z_2	Z_3	Z_4
S0	0	0	0	0	-	0	0	1	0	0	0	0
	0	0	0	1	-	0	1	1	0	0	0	0
	0	0	1	-	-	1	0	1	0	0	0	0
S1	0	1	0	0	0	0	1	0	1	0	0	0
	0	1	0	0	1	0	0	1	1	0	0	0
	0	1	-	1	-	1	0	0	1	0	0	0
	0	1	1	0	-	1	1	0	1	0	0	0
S2	1	0	0	0	0	1	0	0	0	1	0	0
	1	0	0	0	1	0	0	1	0	1	0	0
	1	0	0	1	-	1	1	0	0	1	0	0
	1	0	1	-	-	1	1	0	0	1	0	0
S3	1	1	-	-	-	0	0	0	0	0	1	1

(c) A 32 x 7 ROM is required because there are 5 inputs and 7 outputs.

(d)



(e) Test 00: D; Test 01: N; Test 10: R; Test 11: 1

State	Addr	Test	NST	Z0	Z1	Z2	Z3	Z4
S0	0000	00	0111	1	0	0	0	0
SA	0001	01	0011	0	0	0	0	0
SB	0010	11	0000	0	0	0	0	0
S1	0011	01	0111	0	1	0	0	0
SC	0100	00	1011	0	0	0	0	0
SD	0101	10	0000	0	0	0	0	0
SE	0110	11	0011	0	0	0	0	0
S2	0111	00	1011	0	0	1	0	0
SF	1000	01	1011	0	0	0	0	0
SG	1001	10	1000	0	0	0	0	0
SH	1010	11	0111	0	0	0	0	0
S3	1011	11	0000	0	0	0	1	1

Chapter 6: Designing with Field Programmable Gate Arrays

6.1 **(a)** 4 Cells, if N is used as the clock enable. When N = 1 then

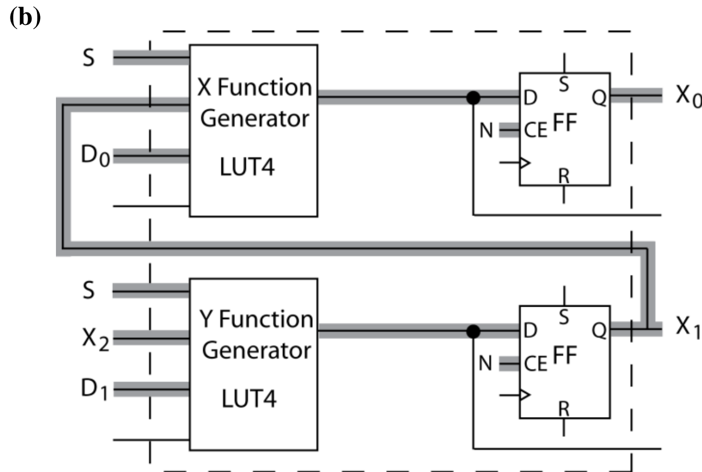
$X_0^+ = S' D_0 + S X_1$ (3 variable function) (two 3 variable functions

$X_1^+ = S' D_1 + S X_2$ (3 variable function) will fit into one cell)

If the clock enable is not used each bit requires a separate cell: 8 cells total.

$X_0^+ = N S' D_0 + N S X_1 + N' X_0$ (5 variable function)

(b)



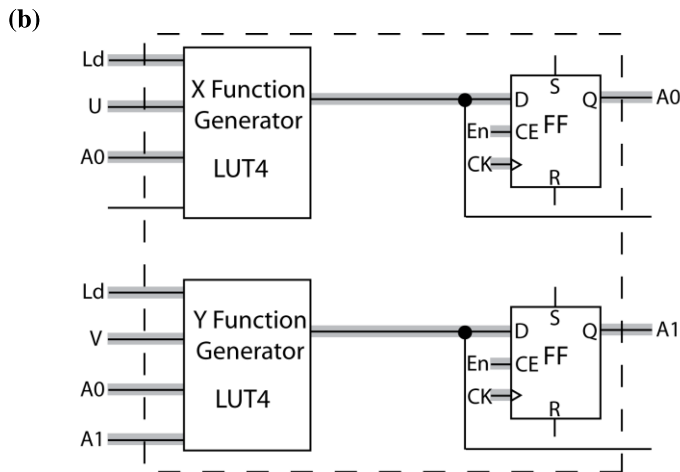
(c) X function generator output = $X_0^+ = S' D_0 + S X_1$

Y function generator output = $X_1^+ = S' D_1 + S X_2$

6.2 **(a)** $Q_{A0}^+ = En\ (Ld\ U + Ld'\ Q_{A0}') + En'\ Q_{A0} = En\ (X) + En'\ Q_{A0}$

$Q_{A1}^+ = En\ (Ld\ V + Ld'\ (Q_{A0} \oplus Q_{A1}) + En'\ Q_{A1} = En\ (Y) + En'\ Q_{A1}$

(b)



$X = Ld\ U + Ld'\ Q_{A0}'$

$Y = Ld\ V + Ld'\ (Q_{A0} \oplus Q_{A1})$

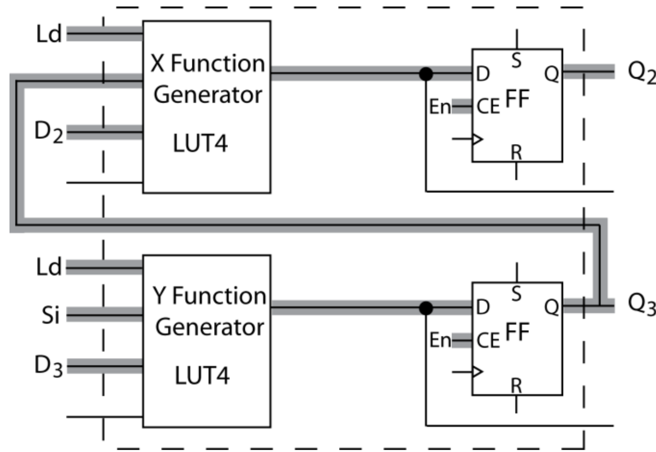
6.3 **(a)** $Q_2^+ = EN'\ Q_2 + EN\ (Ld\ D_2 + Ld'\ Q_3)$

$Q_1^+ = EN'\ Q_1 + EN\ (Ld\ D_1 + Ld'\ Q_2)$

$Q_0^+ = EN'\ Q_0 + EN\ (Ld\ D_0 + Ld'\ Q_1)$

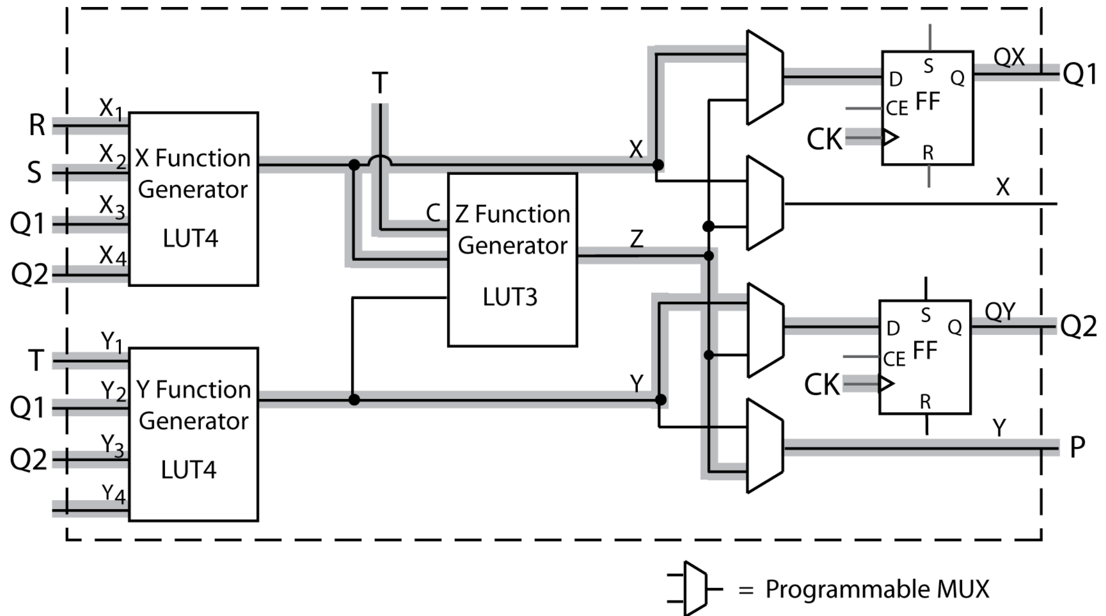
(b) Two cells

(c) $Y = Ld\ D_3 + Ld'\ Si$
$X = Ld\ D_2 + Ld'\ Q_3$



6.4 **(a)** The next state equation of Q_1 can be implemented using the X function generator with the inputs R, S, Q_1, and Q_2. The next state equation of Q_2 can be implemented using the Y function generator with the inputs T, Q_1, and Q_2. The output P can be implemented using the Z function generator with the inputs T (C input) and the X function generator.

(b)

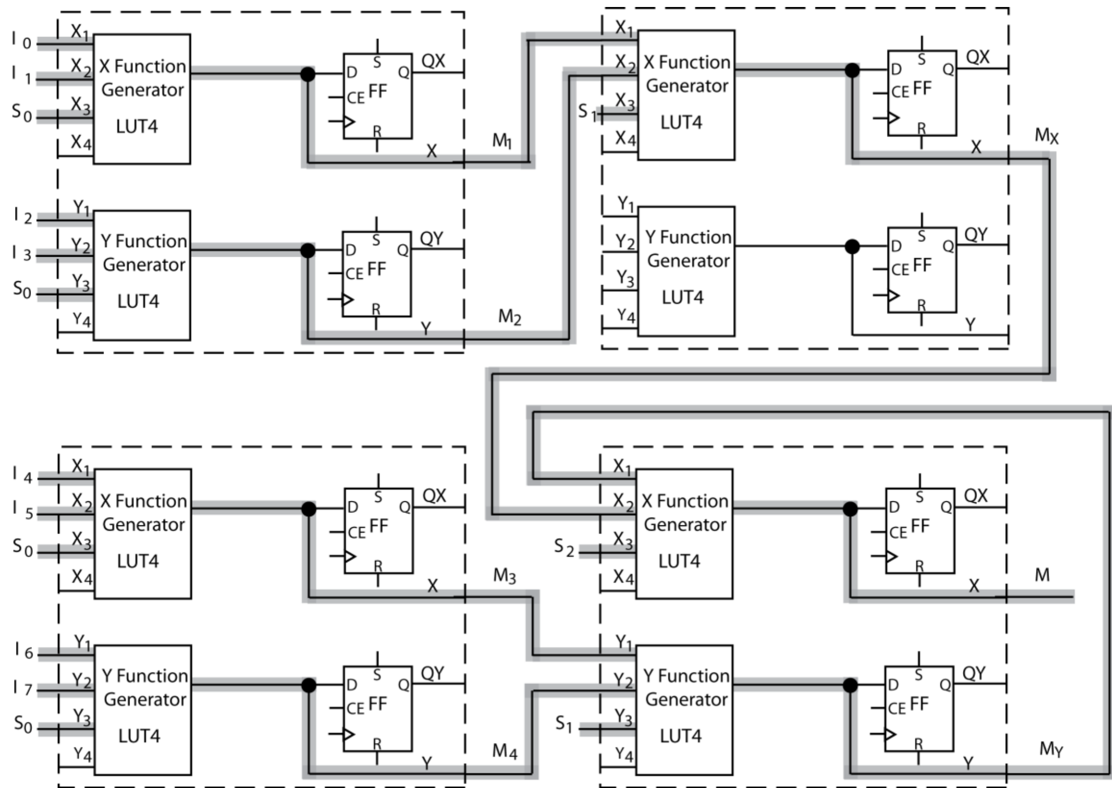


6.5 **(a)** $M = S_2'S_1'S_0'I_0 + S_2'S_1'S_0I_1 + S_2'S_1S_0'I_2 + S_2'S_1S_0I_3 + S_2S_1'S_0'I_4 + S_2S_1'S_0I_5 + S_2S_1S_0'I_6 + S_2S_1S_0I_7$

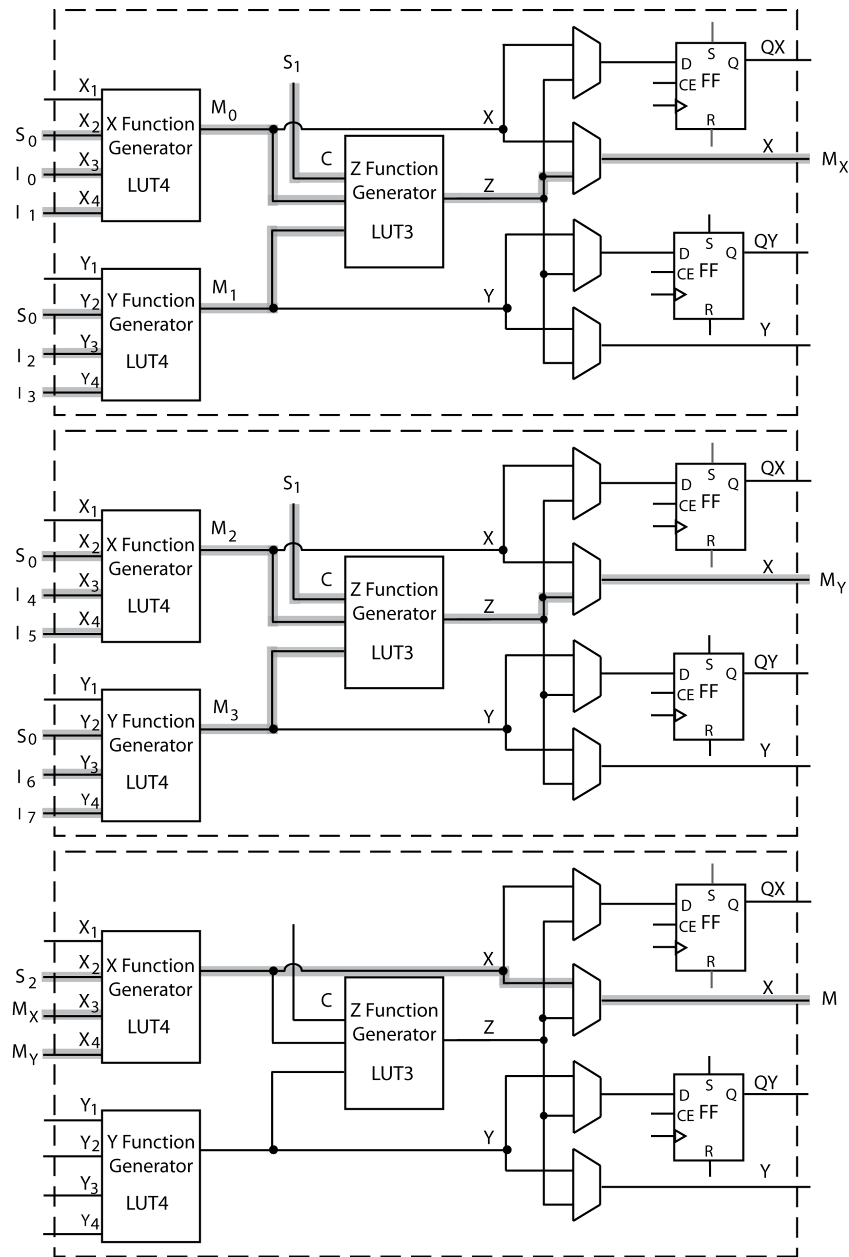
The 8-to-1 MUX can be decomposed into seven 2-to-1 MUXes, and implemented in four Figure 6-1(a) logic blocks.

$M = S_2'M_X + S_2M_Y$
$M_x = S_1'M_1 + S_1M_2$
$M_Y = S_1'M_3 + S_1M_4$
$M_1 = S_0'I_0 + S_0I_1$
$M_2 = S_0'I_2 + S_0I_3$
$M_3 = S_0'I_4 + S_0I_5$
$M_4 = S_0'I_6 + S_0I_7$

The X and Y functions for each block each implement one 2-to-1 mux as labeled:



(b) Three 2-to-1 MUXes (or a 4-to-1 mux) can be implemented in each Figure 6-3 logic block. In total, three blocks are required to implement seven 2-to-1 MUXes. The X, Y, and Z function generators for each block implement a 2-to-1 MUX as labeled:



(c) Each function generator used implements a 2-to-1 mux, and has the same LUT contents: 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1

(d) Each function generator used implements a 2-to-1 mux
X and Y LUT4s have 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1
Z LUT3s have 0, 0, 1, 1, 0, 1, 0, 1 (Consider C as MSB, Y as LSB)

6.6 **(a)**
```
module Figure6_1a(X_in, Y_in, clk, CE, Qx, Qy, X, Y, XLUT, YLUT);
  input [1:4] X_in, Y_in;
  input clk, CE;
  input [0:15] XLUT, YLUT;
  inout X, Y;
  output Qx, Qy;

  reg Qx, Qy;
  wire [1:4] X_Index, Y_Index;

  initial begin
    Qx = 1'b0;
    Qy = 1'b0;
  end

  assign X_Index = {X_in[4], X_in[3], X_in[2], X_in[1]};
  assign Y_Index = {Y_in[4], Y_in[3], Y_in[2], Y_in[1]};
  assign X = XLUT[X_Index];
  assign Y = YLUT[Y_Index];

  always @(posedge clk)
  begin
    if(CE == 1'b1) begin
      Qx <= X;
      Qy <= Y;
    end
  end
endmodule
```

(b)
```
module LUT_Mux(I0, I1, I2, I3, S0, S1, M);
  input I0, I1, I2, I3, S0, S1;
  output M;

  wire Qx1, Qy1, Qx2, Qy2, MM;
  wire [1:4] in1, in2, in3;
  wire M1, M2, Mout;

  assign in1 = {I0, I1, S0, 1'b0};
  assign in2 = {I2, I3, S0, 1'b0};
  assign in3 = {M1, M2, S1, 1'b0};
  assign M = Mout;

  Figure6_1a B0(in1, in2,1'b0, 1'b0, Qx1, Qy1, M1, M2,
16'b0101001101010011,
                16'b0101001101010011);
  Figure6_1a B1(in3, 4'b0000, 1'b0, 1'b0, Qx2, Qy2, Mout, MM,
16'b0101001101010011,
                16'b0000000000000000);
endmodule
```

6.7 **(a)**
```
module Figure6_3(X_in, Y_in, clk, CE, C, Qx, Qy, X, Y, XLUT, YLUT,
ZLUT,
                 SA, SB, SC, SD);
  input [1:4] X_in, Y_in;
  input clk, CE, C;
  input [0:15] XLUT, YLUT;
  input [0:7] ZLUT;
  input SA, SB, SC, SD;
  output X, Y;
  output reg Qx, Qy;
```

```
  initial begin
    Qx = 1'b0;
    Qy = 1'b0;
  end

  wire [1:4] X_Index, Y_Index;
  wire [1:3] Z_Index;
  wire X_int, Y_int, Z_int;
  wire MuxA, MuxB, MuxC, MuxD;

  assign X_Index = {X_in[4], X_in[3], X_in[2], X_in[1]};
  assign Y_Index = {Y_in[4], Y_in[3], Y_in[2], Y_in[1]};
  assign Z_Index = {Y_int, X_int, C};
  assign X_int = XLUT[X_Index];
  assign Y_int = YLUT[Y_Index];
  assign Z_int = ZLUT[Z_Index];

  assign MuxA = (SA == 1'b0)? X_int : Z_int;
  assign MuxB = (SB == 1'b0)? X_int : Z_int;
  assign MuxC = (SC == 1'b0)? Y_int : Z_int;
  assign MuxD = (SD == 1'b0)? Y_int : Z_int;

  assign X = MuxB;
  assign Y = MuxD;

  always @(posedge clk)
  begin
    if(CE == 1'b1) begin
      Qx <= MuxA;
      Qy <= MuxC;
    end
  end

endmodule
```

(b)

```
module Code_Converter(X, clk, Z);
  input X, clk;
  output Z;

  wire Q1, Q2, Q3, Zout;
  wire [3:0] D_in;
  wire T1, T2, T3, T4;

  assign in = {X, Q1, Q2, Q3};
  assign Z = Zout;

  Figure6_3 B0(D_in, D_in, clk, 1'b1, 1'b0, Q3, Q2, T1, T2,
16'b0001111111000000,
               16'b0110000001000000, 8'b00000000, 1'b0, 1'b0, 1'b0,
             1'b0);
  Figure6_3 B1(D_in, D_in, clk, 1'b1, 1'b0, Q1, T3, T4, Zout,
16'b1010001110000000,
               16'b1010010110011000, 8'b00000000, 1'b0, 1'b0, 1'b0,
             1'b0);
endmodule
```

6.8 **(a)** A 4-to-16 decoder requires 16 outputs, and each function needs no more than 4-variables. 8 Figure 6-1 (a) logic blocks are required.

(b) X-Function LUT: 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
Y-Function LUT: 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

6.9 **(a)** 4 logic blocks are required, 8 LUT4's (See Figure 3-6 for truth table).

$a = n_7 + n_6 + n_5 + n_4$
$b_1 = n_5' n_4' (n_3 + n_2)$
$b = n_7 + n_6 + b1$
$c_1 = n_5 + n_4' n_3 + n_4' n_2' n_1$
$c = n_7 + n_6' c1$
$d_1 = n_3 + n_2 + n_1 + n_0$
$d_2 = n_7 + n_6 + n_5 + n_4$
$d = d_2 + d_1$

(b) $F = a, X_3 = n_7, X_2 = n_6, X_1 = n_5, X_0 = n_4$

X_3	X_2	X_1	X_0	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

$G = b_1, Y_3 = n_5, Y_2 = n_4, Y_1 = n_3, Y_0 = n_2$

Y_3	Y_2	Y_1	Y_0	G
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

6.10 Expanding F around X_6 results in 4 variable functions which can be realized using one function generator each.

$$F = X_6 (X_1' X_2 X_3 + X_2 X_3' X_4' + X_2 X_3 X_4') + X_6' (X_2' X_3' X_4 + X_2 X_3' X_4' + X_3' X_4 X_5) + X_7$$
$$F = X_6 (F_1) + X_6' (F_2) + X_7$$

For block one: X LUT has inputs X_1, X_2, X_3, and X_4 and realizes $F_1 = X_1' X_2 X_3 + X_2 X_3' X_4' + X_2 X_3 X_4'$.
Y LUT has inputs X_2, X_3, X_4, and X_5 and realizes $F_2 = X_2' X_3' X_4 + X_2 X_3' X_4' + X_3' X_4 X_5$

For block two: X LUT has the outputs of block one's X LUT (F_1) and Y LUT (F_2), X_6, and X_7 as inputs. The X LUT realizes $F = X_6 (F_1) + X_6' (G_1) + X_7$. The Y LUT is unused.

6.11 Expanding Q^+ around U Q results in 4 variable equations which can be realized using one function generator each.

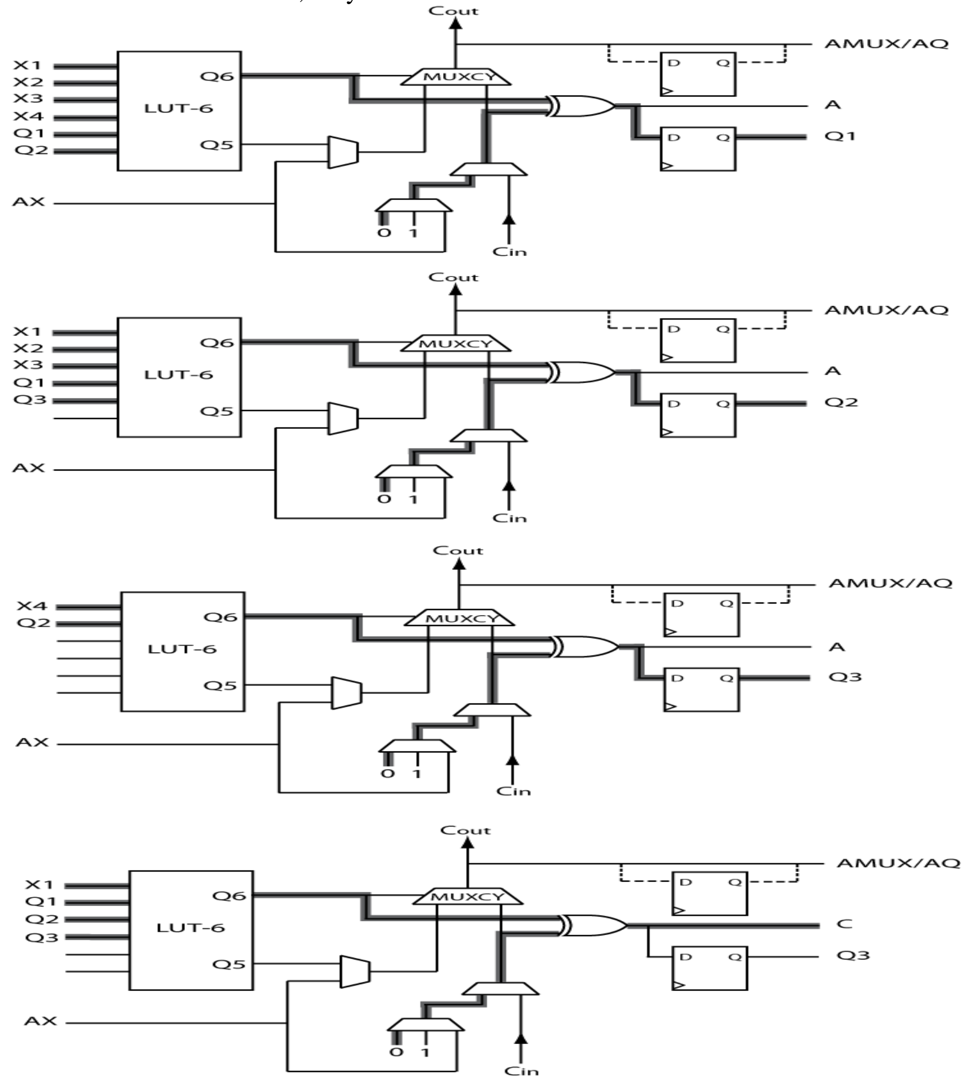
$$Q^+ = U\,Q\,(V' W + X' Y + V W') + U' Q' (V X' Y' + V' Y + X Y + V' X)$$
$$Q+ = U\,Q\,(X_{func}) + U' Q'(Yf_{unc})$$

Mark connections in a manner similar to Problem 6.1's solution.

For block one: X LUT has inputs V, W, X, and Y and realizes V' W + X' Y + V W'
Y LUT has inputs V, X, and Y and realizes V X' Y' + V' Y + X Y + V' X

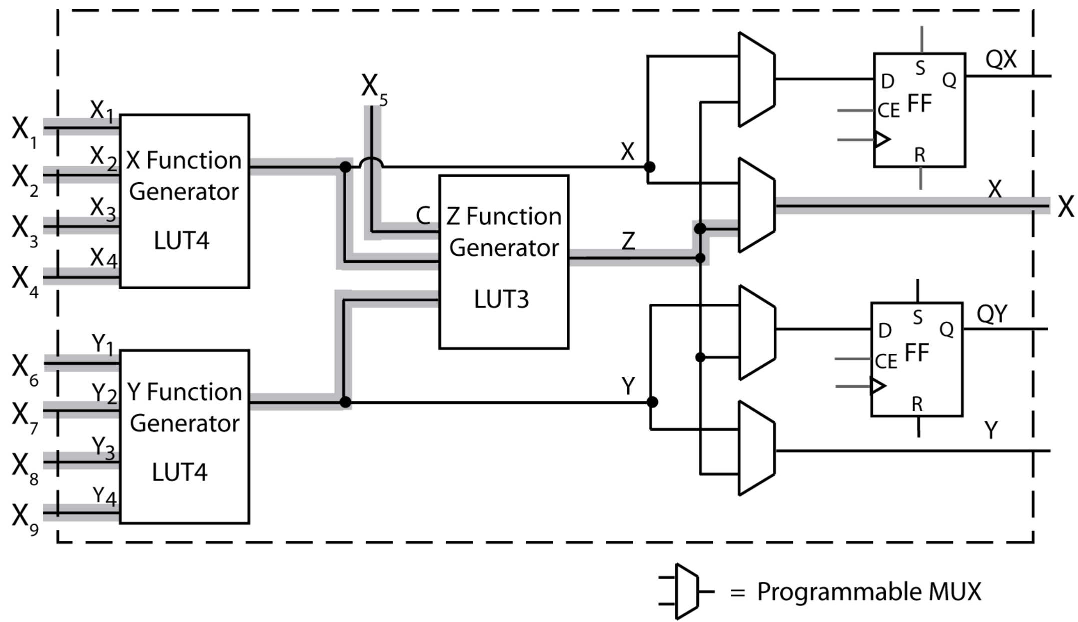
For block two: X LUT has U, Q, and block one's X_{func} and Y_{func} as inputs and realizes
$Q+ = U\, Q\, (X_{func}) + U'\, Q'(Y_{func})$

6.12 To realize the next-state equations, we need to use at least four Kintex logic slices (Figure 6-13). One Kintex logic slice is ¼ CLB. Therefore, only 1 CLB is needed.

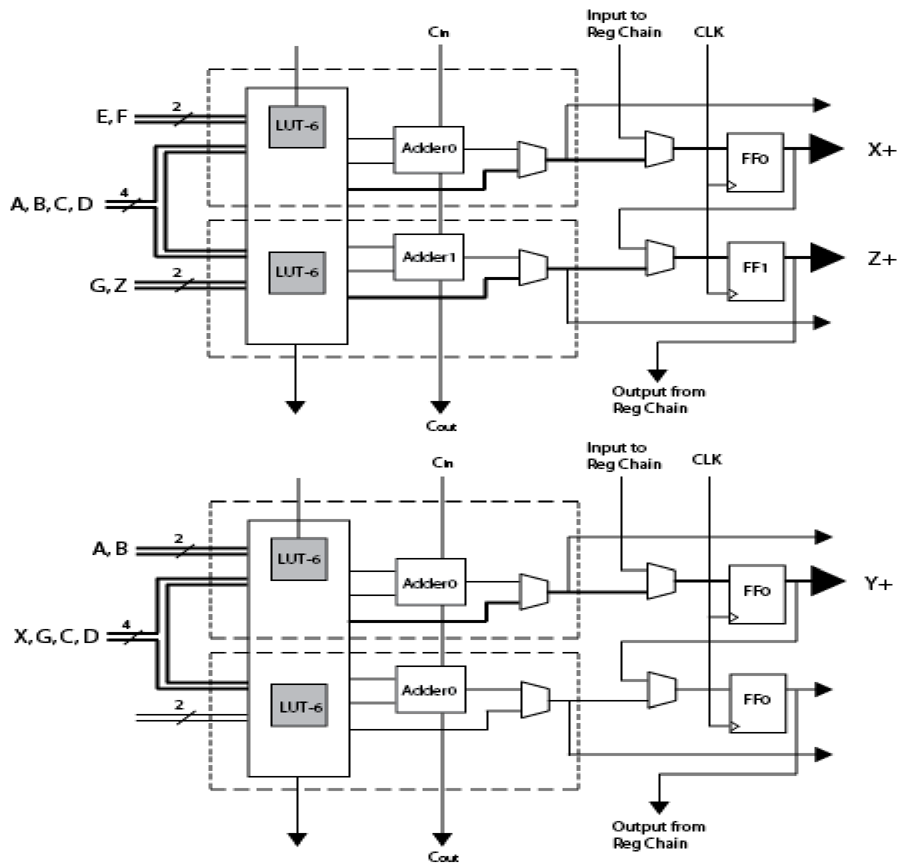


6.13 One cell. Expanding around X_5 results in 4 variable equations which can be realized using one function generator each and X_5 can be used as the C input.

$$X = X_5\, (X_1'\, X_2'\, X_3'\, X_4' + X_1\, X_2\, X_3\, X_4) + X_5'\, (X_6\, X_7'\, X_8'\, X_9 + X_6'\, X_7\, X_8\, X_9')$$
$$X_{func} = (X_1'\, X_2'\, X_3'\, X_4' + X_1\, X_2\, X_3\, X_4)$$
$$Y_{func} = (X_6\, X_7'\, X_8'\, X_9 + X_6'\, X_7\, X_8\, X_9')$$
$$Z_{func} = X_5\, (X_{func}) + X_5'\, (Y_{func})$$



6.14



6.15 **(a)** Expanding Z around Y results in 4 variable equations which can be realized using one function generator each.

$$Z = Y\ (V\ W'\ X + U'\ V'\ W) + Y'\ (V\ W'\ X + T\ V'\ W)$$

$$Z = Z_{func} = Y\ (X_{func}) + Y'\ (Y_{func})$$

Implement internal logic cell connections in a manner similar to Problem 6.12 Solution with U, V, W, and X as inputs to the X-function generator, T, V, W, and X as inputs to the Y-function generator and Y as the C input.

(b) The original equation can be implemented as follows:

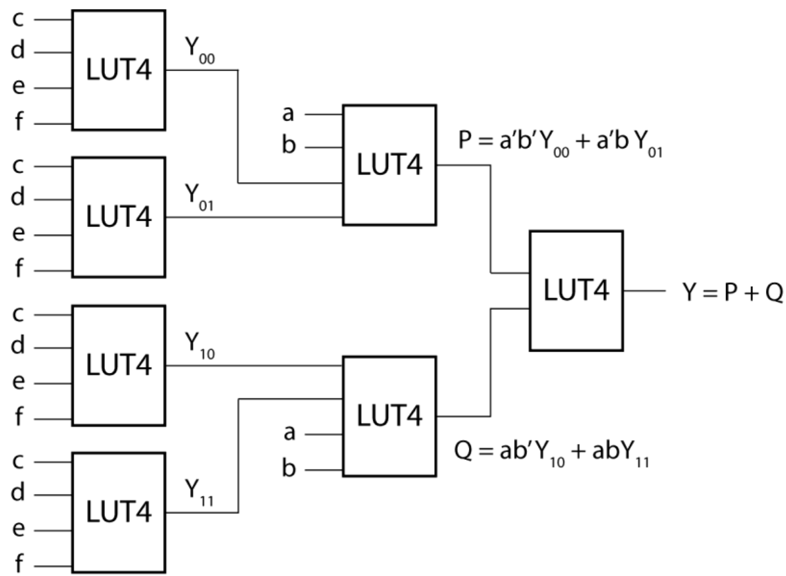
Block 1: X-LUT has inputs U, V, W, X and realizes V W' X + U' V' W
Y-LUT has inputs T, V, W, X and realizes V W' X + T V' W

Block 2: X-LUT has Y and Block 1's X_{func} and Y_{func} as inputs and realizes $Z = Y (X_{func}) + Y' (Y_{func})$
Y-LUT is unused

6.16 $F = X_6 (X_1' X_2 X_3' X_4 + X_2' X_4' + X_3 X_4 X_5 + X_1 X_3) + X_6' (X_2' X_3' X_4 + X_2 X_4 + X_3' X_4 + X_1 X_3)$

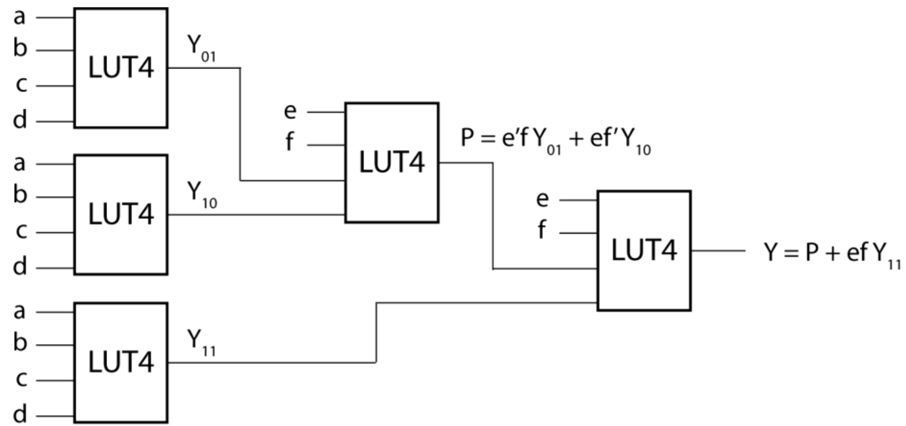
6.17 $Y = a'b'Y_{00} + a'bY_{01} + ab'Y_{10} + abY_{11}$

$Y_{00} = Y_{a=0,b=0} = cde'f + c'def$
$Y_{01} = Y_{a=0,b=1} = cde'f + cdef' + c'de'f$
$Y_{10} = Y_{a=1,b=0} = cde'f + cd'ef'$
$Y_{11} = Y_{a=1,b=1} = cde + cde'f + cdef' + cd'e'f$



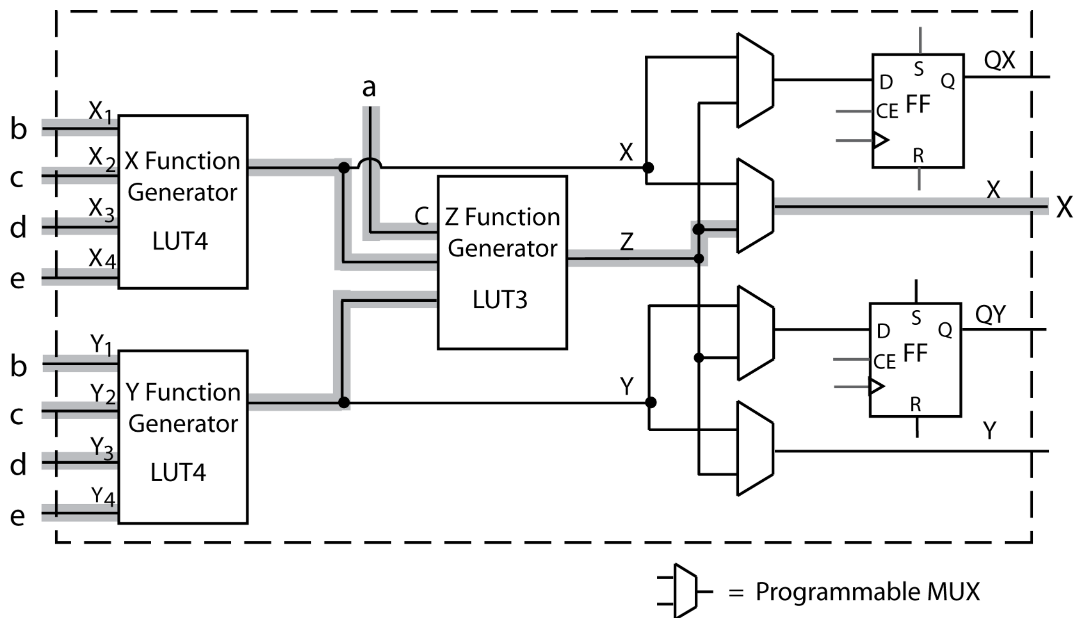
6.18 $Y = e'f'Y_{00} + e'fY_{01} + ef'Y_{10} + efY_{11}$

$Y_{00} = 0$
$Y_{01} = abcd$
$Y_{10} = a'bc'd' + b'c'$
$Y_{11} = ab'cd + a'bc'd'$



6.19 **(a)** $Y = a'(bc'd'e + b'c'e) + a(b'cd'e + b'c'e + bcde) = a'(Y1) + a(Y2)$
$Y1 = bc'd'e + b'c'e$
$Y2 = b'cd'e + b'c'e + bcde$

(b)



(c)

bcde	Y1 (X_{func})	Y2 (Y_{func})
0000	0	0
0001	1	1
0010	0	0
0011	1	1
0100	0	0
0101	0	1
0110	0	0
0111	0	0
1000	0	0
1001	1	0
1010	0	0
1011	0	0
1100	0	0
1101	0	0
1110	0	0
1111	0	1

a X_{func} Y_{func}	Z_{func}
000	0
001	0
010	1
011	1
100	0
101	1
110	0
111	1

6.20 **(a)** Eight LUTs are required. Each bit of the adder requires one LUT to generate the sum and one LUT to generate the carry-out.

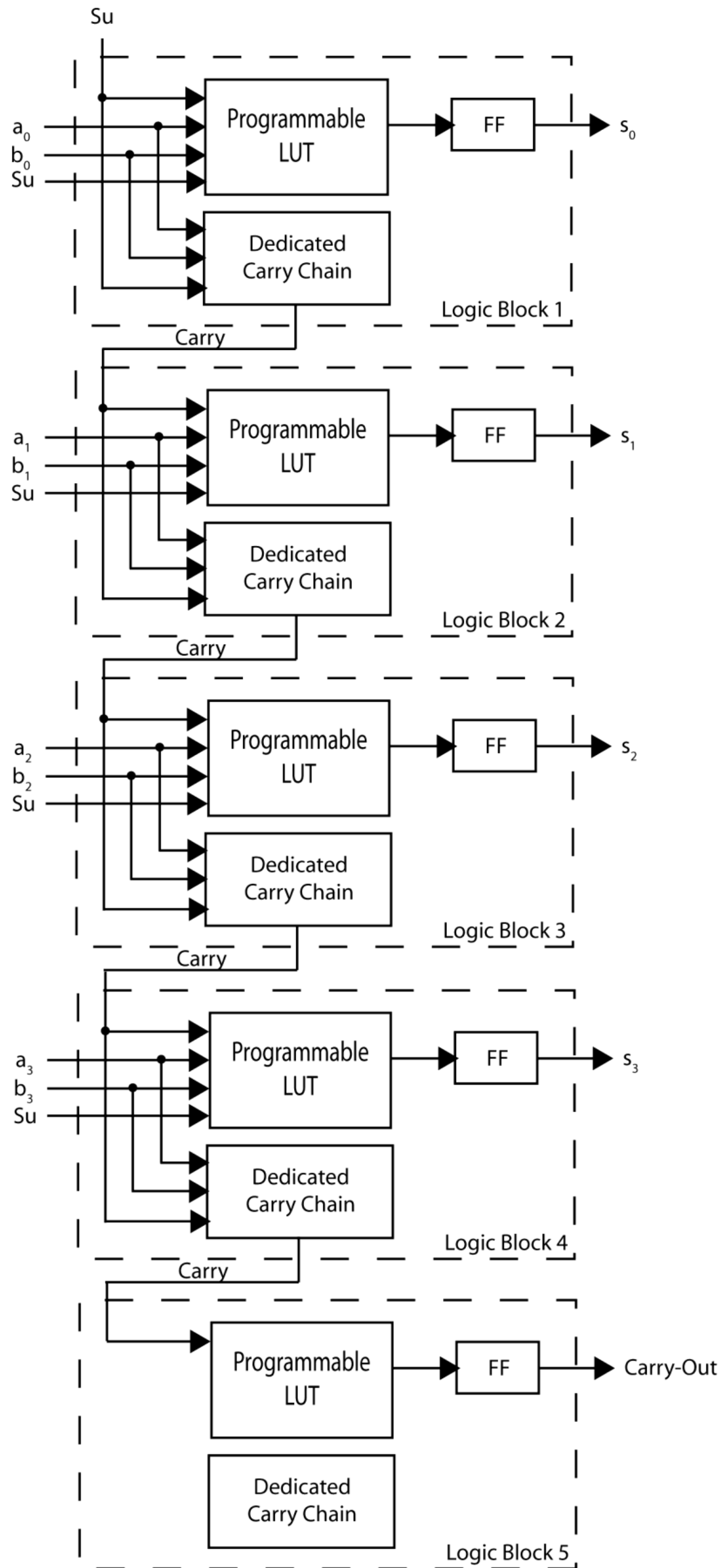
(b) Four LUT4s are required. Each bit of the adder requires one LUT4 to generate the sum. Dedicated carry chain logic generates the carry-out.

(c) When Su is 1, the circuit should add *a* to the 2's complement of *b* by inverting each bit of *b* and setting bit 0's C_{in} to.

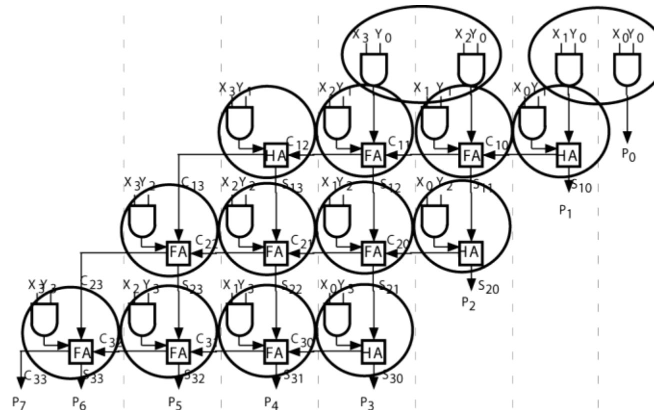
Each bit will have the same output function:

Su	a_i	b_i	C_{in}	Out_i
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

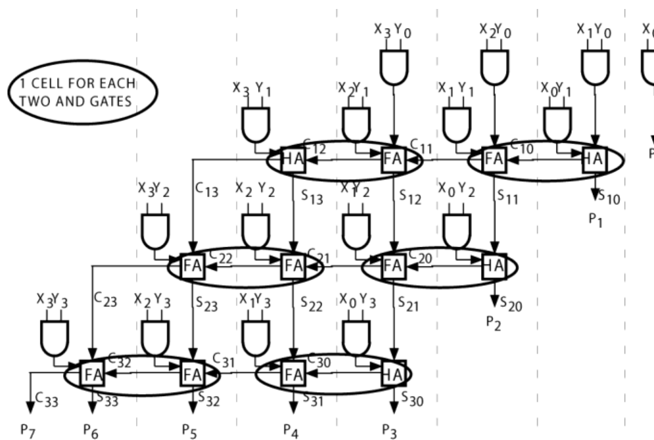
$Out_i = Su'a_i'b_i'C_{in} + Su'a_i'b_iC_{in}' + Su'a_ib_i'C_{in}' + Su'a_ib_iC_{in} + Sua_i'b_i'C_{in}' + Sua_i'b_iC_{in} + Sua_ib_i'C_{in} + Sua_ib_iC_{in}'$


Su
a0
b0
Su
Programmable LUT
FF
s0
Dedicated Carry Chain
Logic Block 1
Carry
a1
b1
Su
Programmable LUT
FF
s1
Dedicated Carry Chain
Logic Block 2
Carry
a2
b2
Su
Programmable LUT
FF
s2
Dedicated Carry Chain
Logic Block 3
Carry
a3
b3
Su
Programmable LUT
FF
s3
Dedicated Carry Chain
Logic Block 4
Carry
Programmable LUT
FF
Carry-Out
Dedicated Carry Chain
Logic Block 5

6.21 **(a)** 14 cells total.



(b) 14 cells total: 6 for adders and 8 for AND gates but propagation delay is less.



6.22 **(a)** $Z = A'(BC'D'EF' + B'C'E'F + BC'E'F') + A(B'CD'E'F + B'C'E'F + BCDE)$

$Z = A'(Z_0) + A(Z_1)$

$Z_0 = D'(Y_{00}) + D(Y_{01})$

$Y_{00} = BC'EF' + B'C'E'F + BC'E'F$

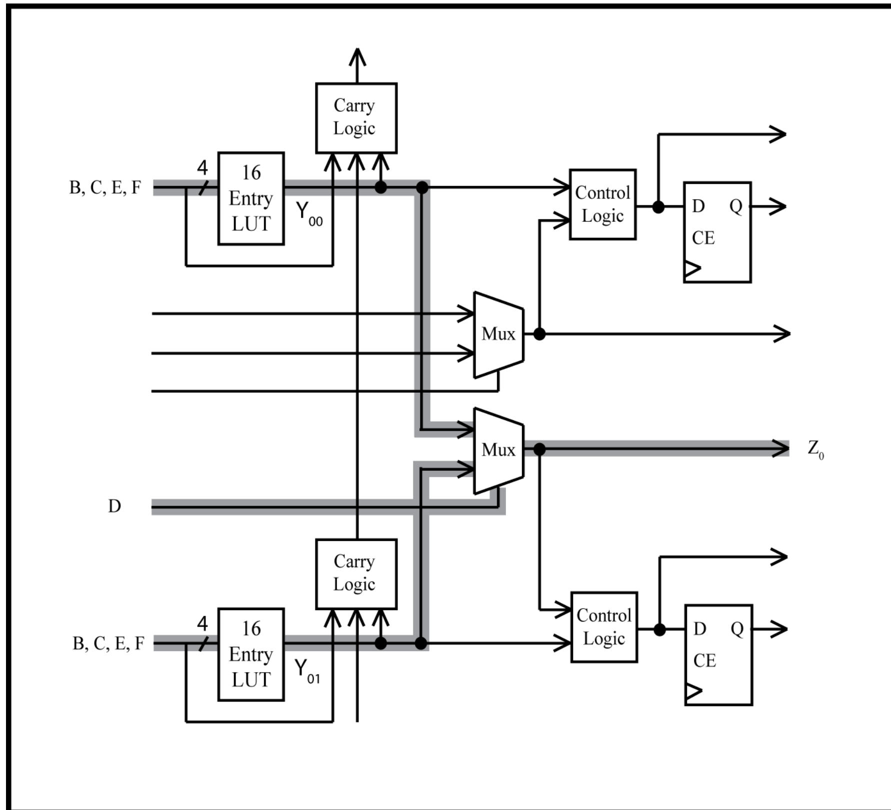
$Y_{01} = B'C'E'F + BC'E'F'$

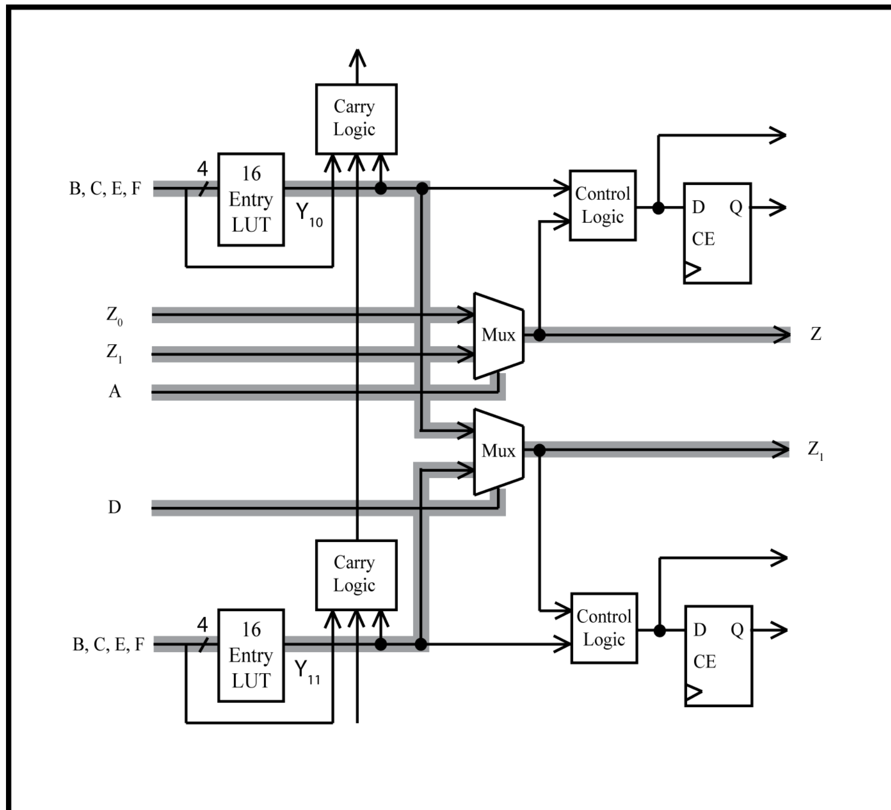
$Z_1 = D'(Y_{10}) + D(Y_{11})$

$Y_{10} = B'C'E'F + B'CE'F$

$Y_{11} = B'C'E'F + BCE$

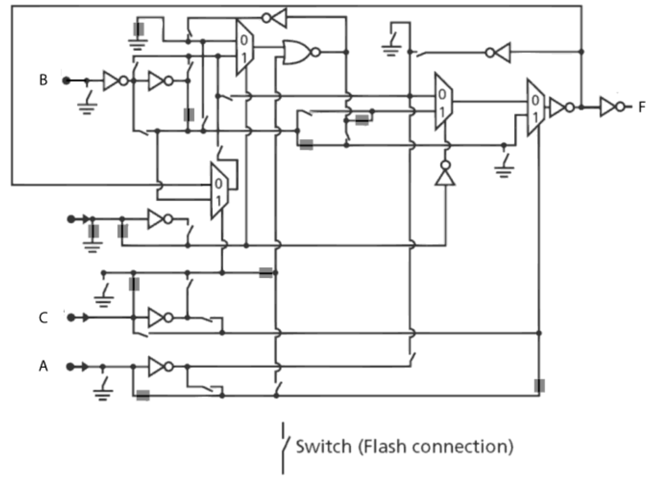
(b)



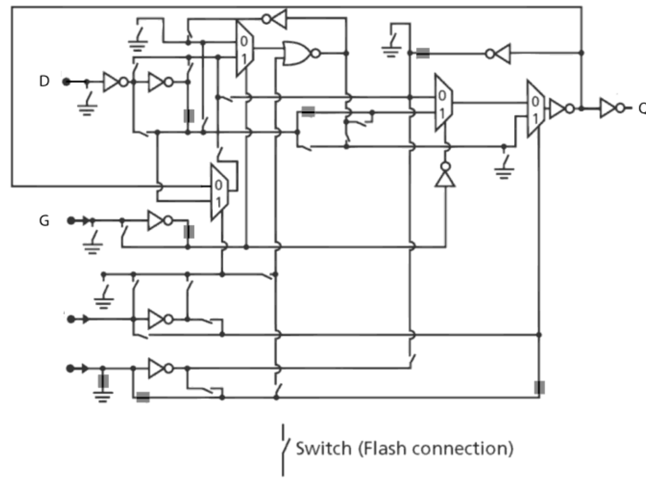


6.23 **(a)** No solution available

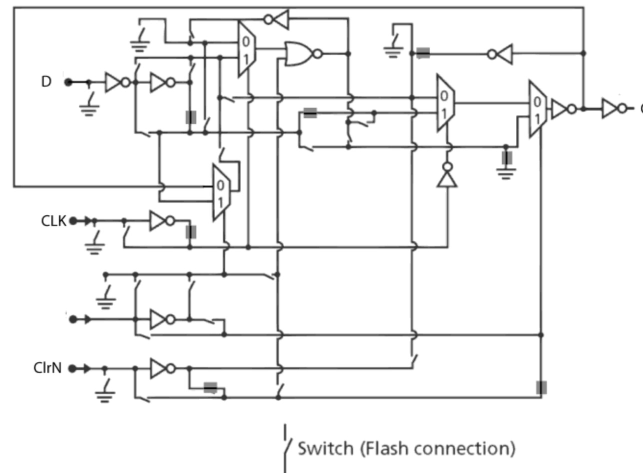
(b)



(c)



(d)



6.24 The possible functions are bolded below:

i) All 32-variable functions
ii) Some 32-variable functions
iii) All 8-variable functions
iv) Some 8-variable functions
v) All 7-variable functions
vi) Some 7-variable functions
vii) All 6-variable functions
viii) Some 6-variable functions
ix) All 36-variable functions
x) Some 36-variable functions
xi) All 39-variable functions
xii) Some 39-variable functions

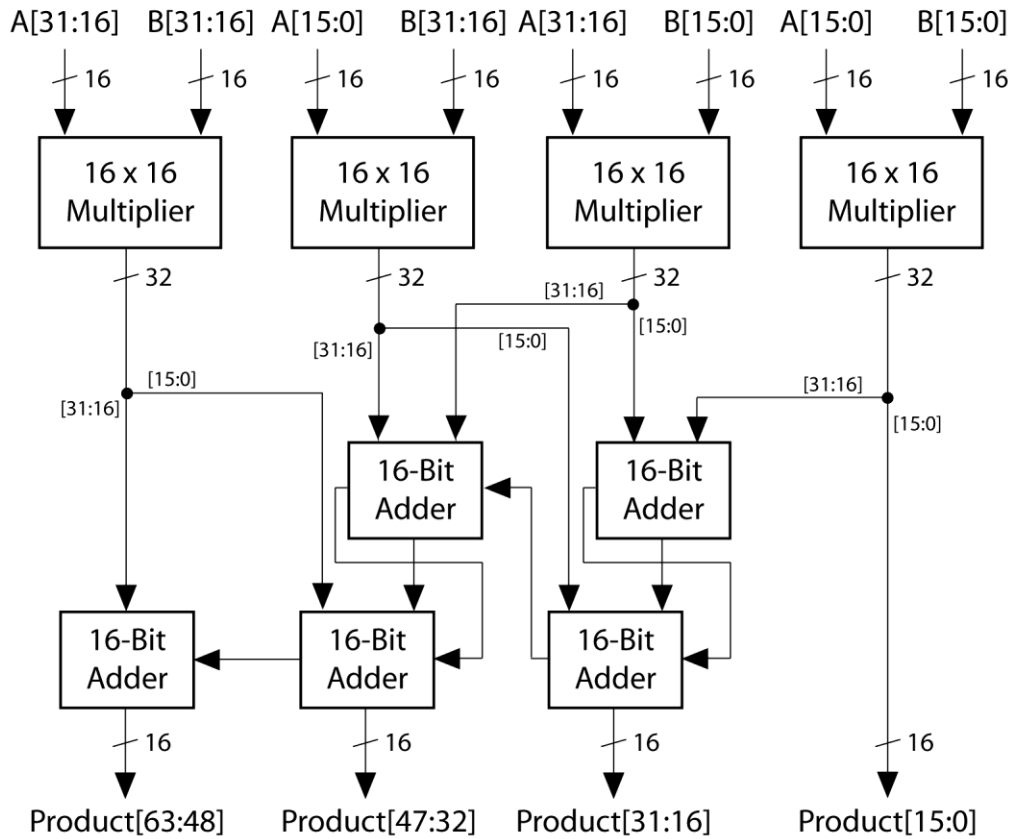
6.25 The sequential circuit requires 3 Virtex slices. For the first slice, the G inputs are Q_2, C, D, and E ($G = Q_2' C D E$). The F inputs are Q_2, A, B, and C ($F = Q_2 A B C$). The BX input is Q1. Then the X flip-flop implements the Q_1 flip-flop. Also, if the FXA input is 1, the FXB input is 0, and the BY input is Q_1, then the Y flip-flop implements Q_2. For the second slice, the G inputs are Q_2, A, and B ($G = Q_2' A B + Q_2' A' B'$). The F inputs are Q_2, A, B, and C ($F = Q_2' A B' + Q_2 (A' + B + C)$). The BX input is Q_1. Then the output to the F5 MUX implements Z_1. For the third slice, the G inputs are Q_1, Q_2, A, and B ($G = Q_1 A' + Q_1 B + Q_2'$). Then the Y combinational output implements Z_2.

6.26 **(a)** No solution provided

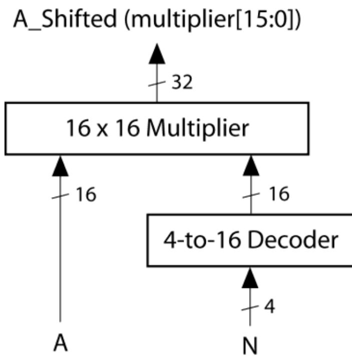
(b) No solution provided

6.27 Stratix V logic module is similar to Stratix IV logic module, except that there are four flip flops existing per logic module instead of the two in Stratix IV.

6.28

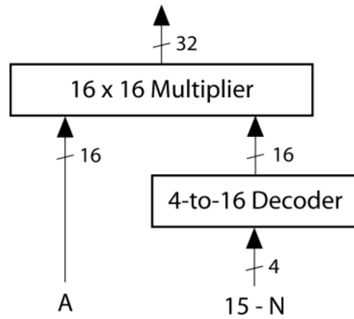


6.29 **(a)**



(b)
```
module P6_29b(A, N, A_Shft);
   input [15:0] A;
   input [0:15] N;
   output [15:0] A_Shft;

   wire [15:0] decoder_out;
   wire [31:0] mplier_out;

   assign decoder_out = (16'b0000000000000001) << N;
   assign mplier_out = A * decoder_out;
   assign A_Shft = mplier_out[15:0];
endmodule
```

(c)




```
module P6_29c(A, N, A_Shft);
  input [15:0] A;
  input [0:15] N;
  output [15:0] A_Shft;

  wire [15:0] decoder_out;
  wire [31:0] mplier_out;

  assign decoder_out = (16'b0000000000000001) << (15 - N);
  assign mplier_out = A * decoder_out;
  assign A_Shft = mplier_out[30:15];
endmodule
```

6.30 S_0: $Q_0Q_1Q_2Q_3$ = 1000, S_1: 0100, S_2: 0010, S_3: 0001

$Q_0^+ = St'Q_0 + Q_3$
$Q_1^+ = StQ_0 + K'M'Q_1 + K'Q_2$
$Q_2^+ = MQ_1$
$Q_3^+ = KM'Q_1 + KQ_2$
Load = StQ_0
Done = Q_3
Sh = $M'Q_1 + Q_2$
Ad = MQ_1

6.31 S_0: $Q_0Q_1Q_2Q_3Q_4Q_5Q_6$ = 1000000, S_1: 0100000, S_2: 0010000, S_3: 0001000, S_4: 0000100, S_5: 0000010,
S_6: 0000001

$Q_0^+ = St'Q_0 + CQ_4 + C'Q_6$
$Q_1^+ = StQ_0$
$Q_2^+ = Q_1$
$Q_3^+ = Q_2$
$Q_4^+ = Q_3$
$Q_5^+ = C'Q_4 + K'C'Q_5 + CQ_5$
$Q_6^+ = KC'Q_5 + CQ_6$
Rdy = Q_0
Ldu = StQ_0
Lds = StQ_0
Ldl = Q_1
Ldd = Q_2
Sh = $Q_3 + C'Q_4 + C'Q_5$

$Su = CQ_5 + CQ_6$
$V = CQ_4$
$Cm1 = C\text{ }'Q_{neg}Q_6$

For $S_0 = 0000000$, change all instances of Q_0 in above equations to Q_0':
$Q_0^+ = St'\ Q_0' + CQ_4 + C\ 'Q_6$
$Q_1^+ = StQ_0'$
$Rdy = Q_0'$
$Ldu = StQ_0'$
$Lds = StQ_0'$
All other equations unchanged

6.32 S_0: $Q_3Q_2Q_1Q_0 = 0000$, S_1: 1100, S_2: 1010, S_3: 1001

To create a one-hot encoding, if Q_3 is 0 in the reset state it must be 1 in all other states.

6.33 **(a)** $Q_0^+ = X_2\ Q_1 + X_4\ Q_3$
$Q_1^+ = X_1\ Q_0$
$Q_2^+ = X_1'\ Q_0 + X_2'\ Q_1 + X_3'\ Q_2 + X_4'\ Q_3$
$Q_3^+ = X_3\ Q_2$

$Z_1 = Q_0 + Q_2$
$Z_2 = Q_1 + Q_3$

(b) 5 Total:
1 Slice: Q_0 (one LUT4 and FF), Q_1 (one LUT4 and FF)
2.5 Slices: Q_2: (each AND term in one half-slice, one half-slice combines 4 product terms, one FF)
1 Slice: Q_3 (one LUT4 and FF), Z_1 (one LUT4)
½ Slice: Z_2 (one LUT4)

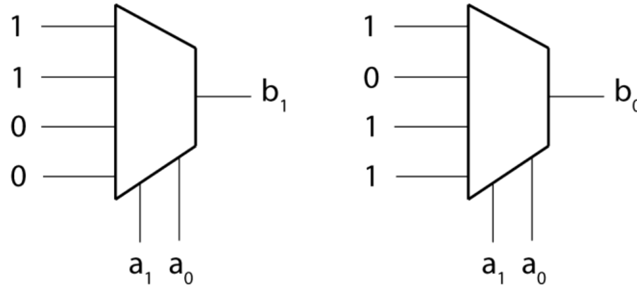
6.34 To ensure proper synthesis, amend the code for Figure 4-15 as follows:
- Within the first process, ensure that all If-Then statements include an Else portion.

6.35 Using the Xilinx ISE, targeted for a Spartan 3 FPGA:

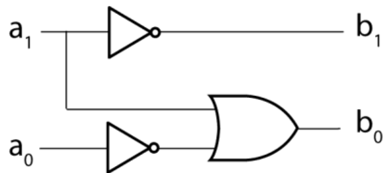
Settings	Figure 4-35	Figure 4-40
Goal: Speed FSM Encoding: Auto	25 Slices 11 Flip-Flops 47 LUT4s Max Speed: 188.656 MHz	13 Slices 14 Flip-Flops 24 LUT4s Max Speed: 194.714MHz
Goal: Area FSM Encoding: Auto	25 Slices 18 Flip-Flops 45 LUT4s Max Speed: 146.307MHz	13 Slices 21 Flip-Flops 23 LUT4s Max Speed: 110.252MHz
Goal: Area FSM Encoding: One-Hot	25 Slices 18 Flip-Flops 45 LUT4s Max Speed: 146.307MHz	13 Slices 21 Flip-Flops 23 LUT4s Max Speed: 110.252MHz
Goal: Area FSM Encoding: Compact	25 Slices 18 Flip-Flops 45 LUT5s Max Speed: 146.307MHz	13 Slices 21 Flip-Flops 23 LUT4s Max Speed: 110.252MHz

Figure 4-40 uses fewer resources then Figure 4-35, and each synthesis option uses about the same amount of resources. The solution to this problem may change depending on what synthesis tool and target device is used.

6.36 **(a)** A 4-to-1 mux for each bit of b:



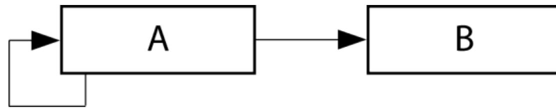
(b) Gate network:



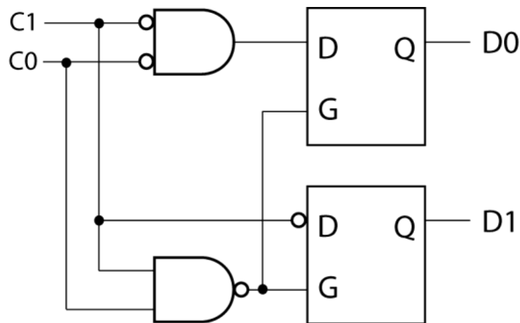
A_1	A_0	B_1	B_0
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	1

By inspection, $B_1 = A_1'$ and $B_0 = A_1 + A_0'$

6.37 **(a)** Arithmetic Right Shift register :



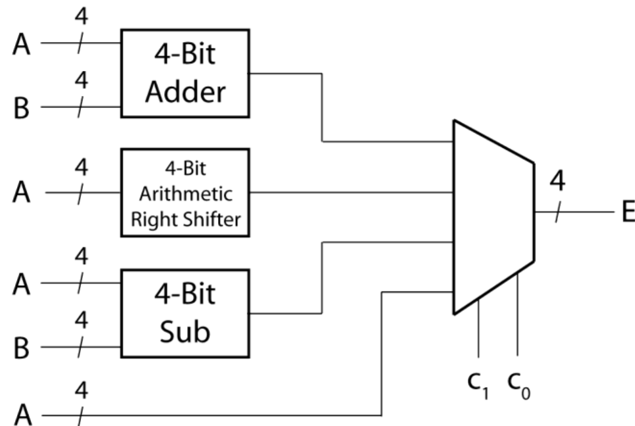
(b)



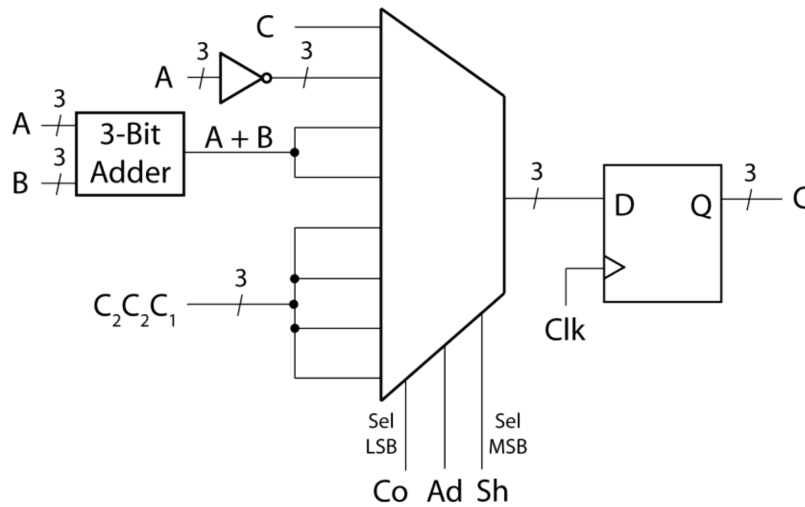
C1	C0	D1	D0
0	0	1	1
0	1	1	0
1	0	0	0
1	1	-	-

D0 = C1'C0'
D1 = C1'

(c)



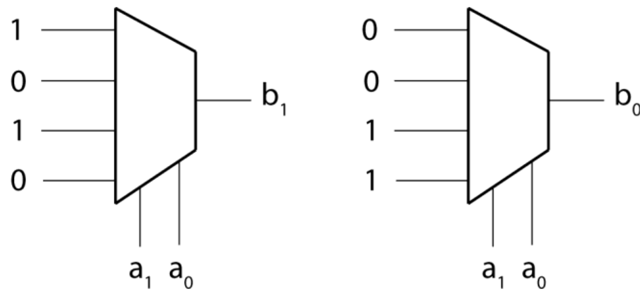
6.38 **(a)** Naïve implementation uses an 8-to-1 mux, 3 inverters (for not A), a 3-bit adder, and a 3-bit register. The arithmetic right shift can be accomplished by feeding in $C_2C_2C_1$.



An alternate implementation is possible if Co, Ad, and Sh will not become active at the same time: use 3 tri-state buffers with tri-state controls Co, Ad, and Sh instead of the mux.

(b) The circuit is a basic ALU, with register. If Co is true, A is complemented and loaded into register C. If Ad is true, A and B are added and loaded into C. If Sh is true, C is shifted right by 1. Sh has the highest priority, followed by Ad, and then by Co. Note that else clauses are not used.

6.39 **(a)** Unoptimized: Two 4-to-1 muxes

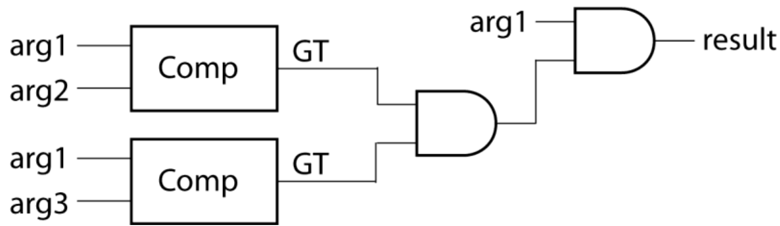


Optimized: 1 inverter. Write truth table and reduce as follows:

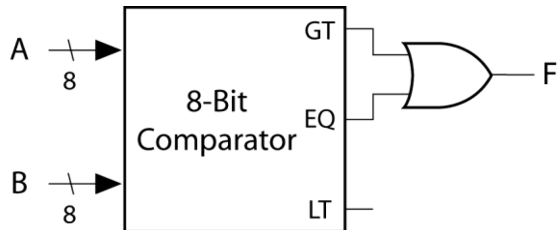
a_1	a_0	b_1	b_0
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	1

$b_1 = a_0'$
$b_0 = a_1$

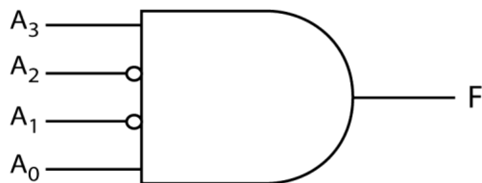
(b)



6.40



6.41



Chapter 7: Floating-Point Arithmetic

7.1 **(a)** $0.875 \times 2^7 = 112$
(b) $-1 \times 2^7 = -128$
(c) $(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$
(d) $-(2 - 2^{-23}) \times 2^{127} \approx -3.4 \times 10^{38}$
(e) $(2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}$
(f) $-(2 - 2^{-52}) \times 2^{1023} \approx -1.8 \times 10^{308}$

7.2 **(i)** 41CA0000
(ii) 44FA0800
(iii) 3F800000
(iv) 00000000
(v) 447A0000
(vi) 45FA0000
(vii) 49742400
(viii) C0ACCCCC
(ix) 7F800000
(x) 4EB2D05E

7.3 **(i)** 4039400000000000
(ii) 409F410000000000
(iii) 3FF0000000000000
(iv) 0000000000000000
(v) 408F400000000000
(vi) 40BF400000000000
(vii) 412E848000000000
(viii) C01599999999999A
(ix) 3730000000000000
(x) 41D65A0BC0000000

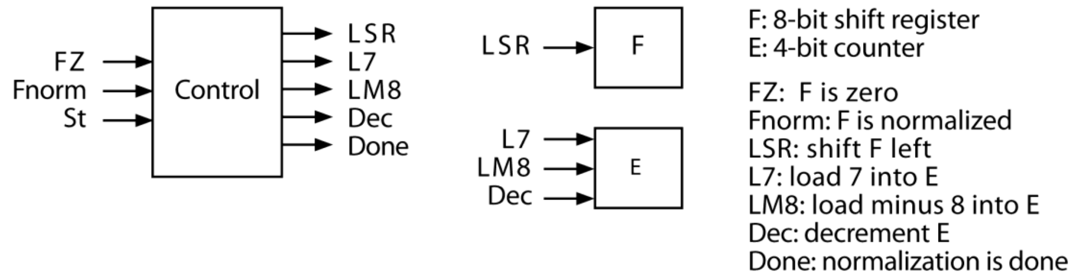
7.4 **(i)** $-1.3411764 \times 2^{-40}$
(ii) 1.5411764×2^{11}
(iii) NaN
(iv) 0
(v) 1.1333333×2^{-93}
(vi) $1.0078431 \times 2^{-125}$

7.5 **(i)** $-1.7294111251831055 \times 2^{-325}$
(ii) $1.3294115066528323 \times 2^{85}$
(iii) NaN
(iv) 0
(v) $1.0666666626939332 \times 2^{-750}$
(vi) $1.0627450980392157 \times 2^{-1007}$

7.6 **(a)** C20D0000
(b) $-1.6015625 \times 2^{-40}$

7.7 **(a)** 41CD0000
(b) C1799999

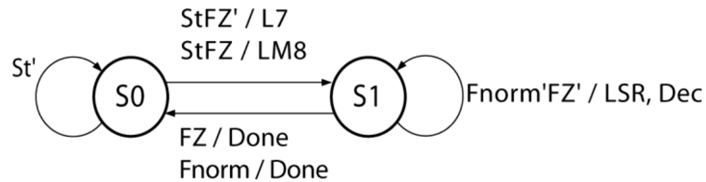
7.8 **(a)**



Algorithm: Step 1) Load integer and set E=7.
Step 2) If F=0, set E=-8 and Done.
Step 3) Repeatedly shift F left and decrement E until F is normalized.

Example:	F	E	
	1.1100101 (-27/128)	7	-27 = -27/128 × 27
	1.1001010 (-54/128)	6	1st left shift.
	1.0010100 (-108/128)	5	2nd left shift.

(b)



(c)

```
module Prob7_8(CLK, St, F, E, Done);
  input CLK, St;
  inout [7:0] F;
  inout [3:0] E;
  output reg Done;

  wire Fnorm, FZ;
  reg state, nextstate;
  reg L7, LM8, LSR, DEC;
  reg [7:0] Ftemp;
  reg [3:0] Etemp;

  initial begin
    state = 1'b0;
    nextstate = 1'b0;
    Ftemp = F;
    Etemp = E;
  end

  assign FZ = (F == 8'b00000000)? 1'b1 : 1'b0;
  assign Fnorm = (F[7] != F[6])? 1'b1 : 1'b0;
  assign F = Ftemp;
  assign E = Etemp;

  always @(state, Fnorm, FZ, St)
```

```
begin
  L7 = 1'b0;
  LM8 = 1'b0;
  LSR = 1'b0;
  DEC = 1'b0;
  Done = 1'b0;
  case(state)
  0: begin
    if(St == 1'b1) begin
      nextstate = 1'b1;
      if(FZ == 1'b1)
        LM8 = 1'b1;
      else
        L7 = 1'b1;
    end
    else
      nextstate = 1'b0;
  end
  1: begin
    if(Fnorm == 1'b1 || FZ == 1'b1) begin
      Done = 1'b1;
      nextstate = 1'b0;
    end
    else begin
      LSR = 1'b1;
      DEC = 1'b1;
      nextstate = 1'b1;
    end
  end
  endcase
end

always @(posedge CLK)
begin
  state <= nextstate;
  if(LSR == 1'b1)
    Ftemp <= {Ftemp[6:0], 1'b0};
  if(DEC == 1'b1)
    Etemp <= Etemp - 4'b0001;
  else if(LM8 == 1'b1)
    Etemp <= 4'b1000;
  else if(L7 == 1'b1)
    Etemp <= 4'b0111;
end

endmodule
```

7.9 **(a)**

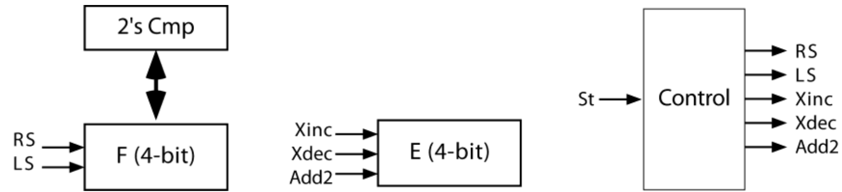
1.) Add Exponents	E = 1000
2.) Multiply Fractions	F = 0.100011
3.) Check for exponent overflow	Exponent overflow has occurred

(b)

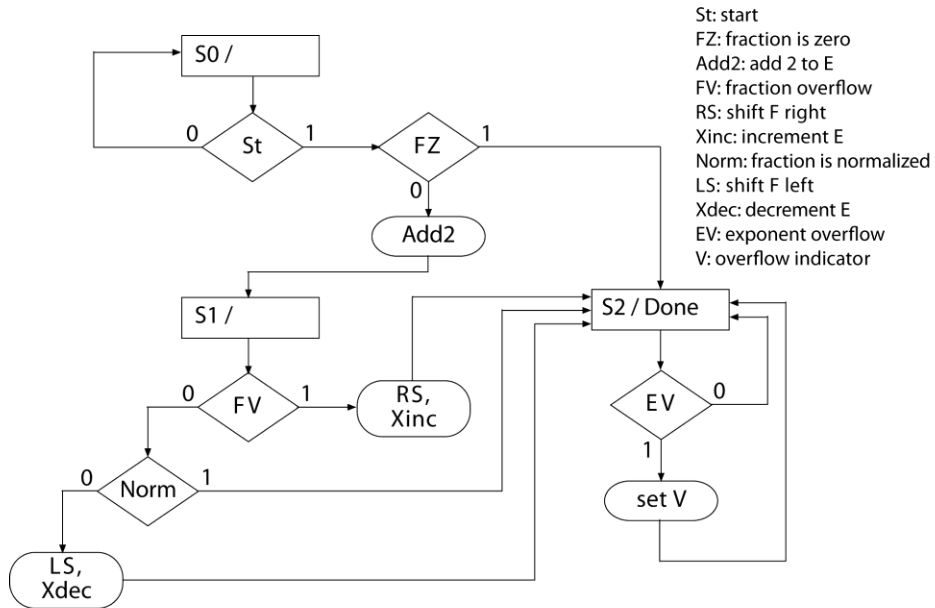
1.) Add Exponents	E = 1000
2.) Multiply Fractions	F = 1.100010
3.) Normalize	F = 1.000100, E = 0111
4.) Check for exponent overflow	Exponent overflow has occurred

7.10 **(a)** $(0.000 \times 2^{-8}) \times (-4)$ = (should result in no change)
$(0.1 \times 2^{x}) \times (-4) = 1.1 \times 2^{x+2} = 1.0 \times 2^{x+1}$ (could cause exponent overflow)
$(1.0 \times 2^{x}) \times (-4)$ $= 1.0 \times 2^{x+2}$ (should be +1, so fraction overflow)
$= 0.1 \times 2^{x+3}$ could cause exponent overflow)

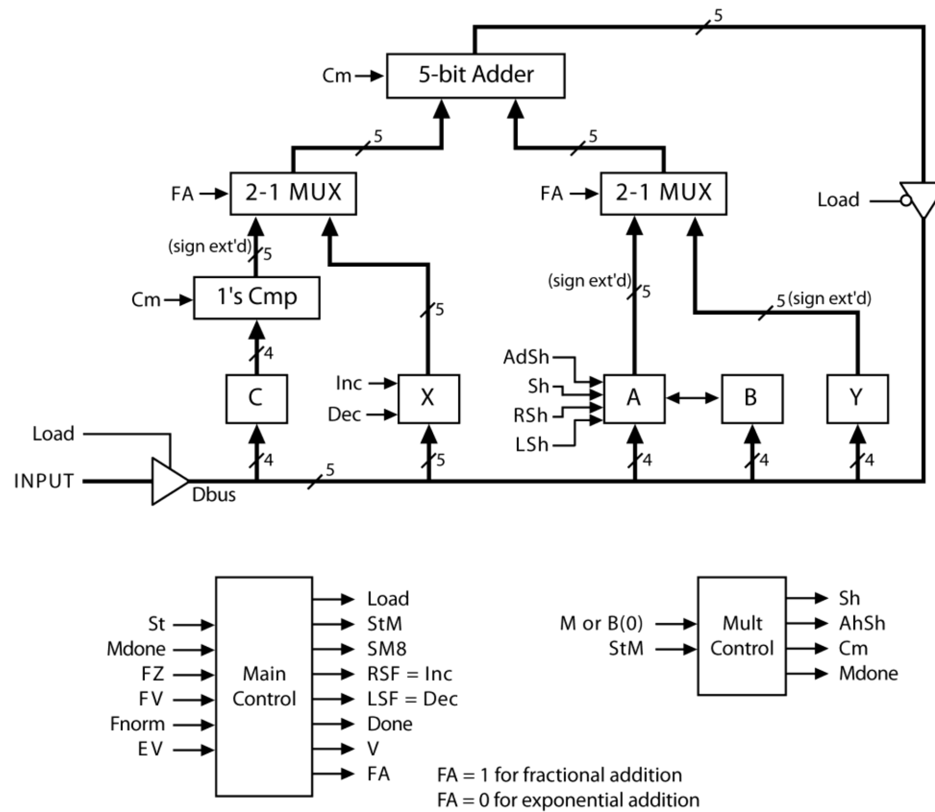
(b)



(c)



7.11 (a)



(b) SM chart is similar to Figure 7-8 with added states for loading.

(c)

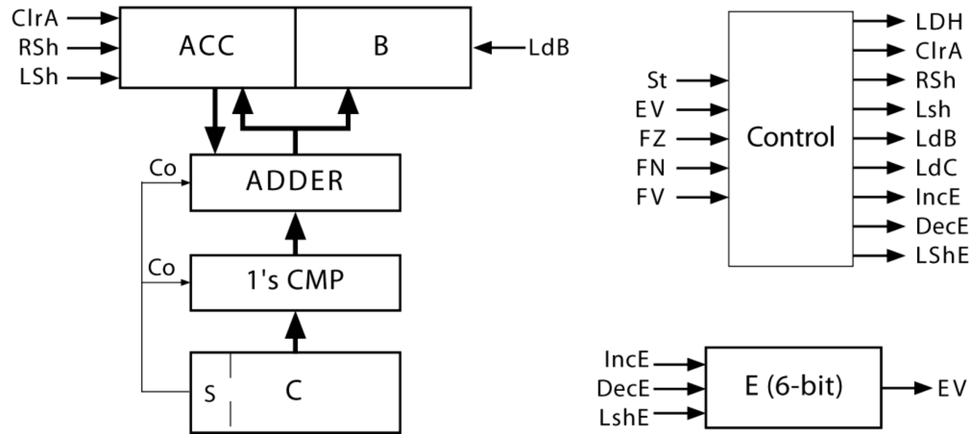
```
  `define M B[0]
module Prob7_11(CLK, St, Input, Fout, Eout, V, Done);
  input CLK, St;
  input [3:0] Input;
  output [6:0] Fout;
  output [3:0] Eout;
  output reg V, Done;

  wire [4:0] dbus, addout, Mux1, Mux2;
  wire [3:0] compNum;
  reg [3:0] A, B, C;
  reg [4:0] X;
  reg [3:0] Y;
  reg Load, CLRA, LdB, LdC, LdX, LdY, RSF, LSF, FA, StM, SM8;
  reg Mdone, Sh, AdSh, Cm;
  wire FZ, FV, Fnorm, EV;
  reg [2:0] state, nextstate;
  reg [2:0] mState, mNextState;

  initial begin
    state = 3'b000;
    nextstate = 3'b000;
    mState = 3'b000;
    mNextState = 3'b000;
    A = 4'b0000;
    B = 4'b0000;
    C = 4'b0000;
    X = 5'b00000;
```

```
  Y = 4'b0000;
end

assign dbus = (Load == 1'b1)? ({Input[3], Input}) : (addout);
assign compNum = (Cm == 1'b1)? ~C : C;
assign Mux1 = (FA == 1'b1)? ({compNum[3], compNum}) : X;
assign Mux2 = (FA == 1'b1)? ({A[3], A}) : ({Y[3], Y});
assign addout = Mux1 + Mux2 + ({3'b000, Cm});
assign Fout = {A[2:0], B};
assign Eout = X[3:0];
assign EV = (X[4] != X[3])? 1'b1 : 1'b0;    // exponent overflow
assign FZ = (A == 4'b0000)? 1'b1 : 1'b0;      // zero fraction
assign FV = (A[3] != A[2])? 1'b1 : 1'b0;    // fraction overflow
assign Fnorm = (A[2]!= A[1])? 1'b1 : 1'b0;    // fraction normalized

always @(mState, `M, StM)
begin
  AdSh = 1'b0;
  Sh = 1'b0;
  Cm = 1'b0;
  Mdone = 1'b0;
  mNextState = 3'b000;
  case(mState)
  0: begin
    if(StM == 1'b1) begin
      mNextState = 3'b001;
      if(`M == 1'b1)
        AdSh = 1'b1;
      else
        Sh = 1'b1;
    end
  end
  1, 2: begin
    mNextState = mState + 3'b001;
    if(`M == 1'b1)
      AdSh = 1'b1;
    else
      Sh = 1'b1;
  end
  3: begin
    mNextState = 3'b100;
    if(`M == 1'b1) begin
      Cm = 1'b1;
      AdSh = 1'b1;
    end
    else
      Sh = 1'b1;
  end
  4: begin
    Mdone = 1'b1;
  end
  default: begin
  end
  endcase
end

always @(state, St, FZ, FV, Fnorm, EV, Mdone)
begin
  Load = 1'b0;
  CLRA = 1'b0;
  LdB = 1'b0;
  LdC = 1'b0;
  LdX = 1'b0;
```

```
LdY = 1'b0;
RSF = 1'b0;
LSF = 1'b0;
V = 1'b0;
Done = 1'b0;
StM = 1'b0;
SM8 = 1'b0;
FA = 1'b0;
case(state)
0: begin
  if(St == 1'b1) begin
    Load = 1'b1;
    CLRA = 1'b1;
    nextstate = 3'b001;
  end
  else
    nextstate = 3'b000;
end
1: begin
  Load = 1'b1;
  LdC = 1'b1;
  nextstate = 3'b010;
end
2: begin
  Load = 1'b1;
  LdX = 1'b1;
  nextstate = 3'b011;
end
3: begin
  Load = 1'b1;
  LdY = 1'b1;
  nextstate = 3'b100;
end
4: begin
  LdX = 1'b1;
  nextstate = 3'b101;
end
5: begin
  FA = 1'b1;
  StM = 1'b1;
  nextstate = 3'b110;
end
6: begin
  if(Mdone == 1'b1) begin
    nextstate = 3'b111;
    FA = 1'b0;
    if(FZ == 1'b1)
      SM8 = 1'b1;
    else if(FV == 1'b1)
      RSF = 1'b1;
    else if(Fnorm == 1'b0)
      LSF = 1'b1;
  end
  else begin
    FA = 1'b1;
    nextstate = 3'b110;
  end
end
7: begin
  Done = 1'b1;
  if(EV == 1'b1)
    V = 1'b1;
  if(St == 1'b0)
```

```
        nextstate = 3'b000;
      else
        nextstate = 3'b111;
    end
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    if(CLRA == 1'b1)
      A <= 4'b0000;
    if(LdB == 1'b1)
      B <= dbus[3:0];
    if(LdC == 1'b1)
      C <= dbus[3:0];
    if(LdX == 1'b1)
      X <= dbus;
    if(LdY == 1'b1)
      Y <= dbus[3:0];
    if(SM8 == 1'b1)
      X <= 5'b11000;
    if(LSF == 1'b1) begin
      A <= {A[2:0], B[3]};
      B <= {B[2:0], 1'b0};
      X <= X - 1; //decrement X
    end
    else if(RSF == 1'b1) begin
      A <= {A[3], A[3:1]};
      B <= {A[0], B[3:1]};
      X <= X + 1; //increment X
    end
    // multiplier control signals
    mState <= mNextState;
    if(AdSh == 1'b1) begin
      A <= dbus[4:1];
      B <= {dbus[0], B[3:0]};
    end
    else if(Sh == 1'b1) begin
      A <= {A[3], A[3:1]};
      B <= {A[0], B[3:1]};
    end
  end

endmodule
```

7.12 (a)



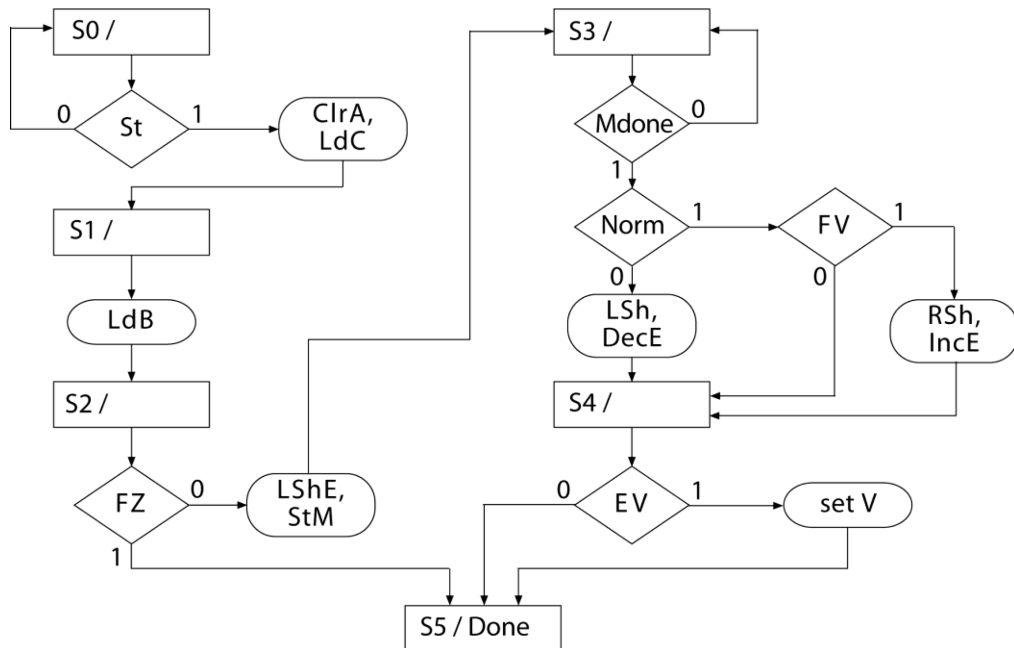
(b) Procedure:

Step 1) Clear Acc.
Step 2) Load |F| to multiplier. If "FZ", then Done.
Step 3) Shift E left once and start multiplication.
Step 4) If "FV", right shift product and increment E.
If not normalized, left shift product and decrement E.
Step 5) If "EV", set indicator.

Example:

	F	E
	1.0110	00100
2's Comp	0.1010	01000
Multiply	0.011001	01000
Normalize	0.11001	00111

(c)



(d)

```
    `define Cm C[4]
    `define M B[0]
  module Prob7_12(CLK, St, F, E, Fout, Eout, V, Done);
    input CLK, St;
    input [4:0] F, E;
    output [8:0] Fout;
    output [4:0] Eout;
    output reg V, Done;

    reg [4:0] A, B, C;
    reg [5:0] RegE;
    wire [4:0] posNum, addout;
    reg LoadB, LoadC, LoadE, CLRA, RSH, LSH, LSE, INCE, DECE, StM, Mdone;
    wire FZ, FV, Fnorm, EV;
    reg [2:0] state, nextstate, mState, mNextstate;
    reg AdSh, Sh;

    initial begin
      state = 3'b000;
      nextstate = 3'b000;
      mState = 3'b000;
      mNextstate = 3'b000;
      A = 5'b00000;
      B = 5'b00000;
      C = 5'b00000;
      RegE = 6'b000000;
    end

    assign posNum = (`Cm == 1'b1)? ~C : C;
    assign addout = A + posNum + {4'b0000, `Cm};
    assign Fout = {A[3:0], B};
    assign Eout = RegE[4:0];
    assign EV = (RegE[5] != RegE[4])? 1'b1 : 1'b0;
    assign FZ = (B == 5'b00000)? 1'b1 : 1'b0;
    assign FV = (A[4] != A[3])? 1'b1: 1'b0;
    assign Fnorm = (A[3] != A[2])? 1'b1 : 1'b0;

    always @(mState, `M, StM)
    begin
      AdSh = 1'b0;
      Sh = 1'b0;
      Mdone = 1'b0;
      case(mState)
      0: begin
        if(StM == 1'b1) begin
          mNextstate = 3'b001;
          if(`M == 1'b1)
            AdSh = 1'b1;
          else
            Sh = 1'b1;
        end
        else
          mNextstate = 3'b000;
      end
      1,2,3,4: begin
        if(`M == 1'b1)
          AdSh = 1'b1;
        else
          Sh = 1'b1;
        mNextstate = mState + 3'b001;
      end
      5: begin
```

```
    Mdone = 1'b1;
    mNextstate = 3'b000;
  end
  default: begin
  end
  endcase
end

always @(state, St, FZ, FV, Fnorm, EV, Mdone)
begin
  LoadB = 1'b0;
  LoadC = 1'b0;
  LoadE = 1'b0;
  CLRA = 1'b0;
  RSH = 1'b0;
  LSH = 1'b0;
  V = 1'b0;
  LSE = 1'b0;
  INCE = 1'b0;
  DECE = 1'b0;
  Done = 1'b0;
  StM = 1'b0;
  case(state)
  0: begin
    if(St == 1'b1) begin
      CLRA = 1'b1;
      LoadC = 1'b1;
      nextstate = 3'b001;
    end
    else
      nextstate = 3'b000;
  end
  1: begin
    LoadB = 1'b1;
    nextstate = 3'b010;
    LoadE = 1'b1;
  end
  2: begin
    if(FZ == 1'b1) begin
      Done = 1'b1;
      nextstate = 3'b100;
    end
    else begin
      LSE = 1'b1;
      StM = 1'b1;
      nextstate = 3'b011;
    end
  end
  3: begin
    if(Mdone == 1'b1) begin
      nextstate = 3'b100;
      if(Fnorm == 1'b1) begin
        if(FV == 1'b1) begin
          RSH = 1'b1;
          INCE = 1'b1;
        end
      end
      else begin
        LSH = 1'b1;
        DECE = 1'b1;
      end
    end
    else
```

```
          nextstate = 3'b011;
      end
      4: begin
        Done = 1'b1;
        if(EV == 1'b1)
          V = 1'b1;
        if(St == 1'b0)
          nextstate = 3'b000;
        else
          nextstate = 3'b100;
      end
      default: begin
      end
      endcase
    end

    always @(posedge CLK)
    begin
      state <= nextstate;
      if(CLRA == 1'b1)
        A <= 5'b00000;
      if(LoadC == 1'b1)
        C <= F;
      if(LoadB == 1'b1)
        B <= addout;
      if(LoadE == 1'b1)
        RegE <= {E[4], E};
      if(LSE == 1'b1)
        RegE <= {RegE[4:0], 1'b0};
      else if(INCE == 1'b1)
        RegE <= RegE + 6'b000001;
      else if(DECE == 1'b1)
        RegE <= RegE - 6'b000001;
      if(LSH == 1'b1) begin
        A <= {A[3:0], B[4]};
        B <= {B[3:0], 1'b0};
      end
      else if(RSH == 1'b1) begin
        A <= {A[4], A[4:1]};
        B <= {A[0], B[4:1]};
      end
      mState <= mNextstate;
      if(AdSh == 1'b1) begin
        A <= {1'd0, addout[4:1]};
        B <= {addout[0], B[4:1]};
      end
      else if(Sh == 1'b1) begin
        A <= {A[4], A[4:1]};
        B <= {A[0], B[4:1]};
      end
    end

  endmodule
```

7.13

```
module P7_13(clk, St, Mplier, Mcand, Product, V, Done);
  input clk, St;
  input [31:0] Mplier, Mcand;
  output reg [31:0] Product;
  output reg V, Done;

  wire [47:0] mult_out;
```

```
  wire [8:0] add_out;
  reg [22:0] F;
  reg [8:0] E;
  reg S;
  reg [8:0] E_Biased;
  reg [1:0] state;

  initial begin
    Product = 32'h00000000;
    E = 9'b000000000;
    S = 1'b0;
    E_Biased = 9'b000000000;
    state = 2'b00;
  end

  assign mult_out = ({1'b1, Mplier[22:0]}) * ({1'b1, Mcand[22:0]});
  assign add_out = (Mplier[30:23] - 8'd127) + (Mcand[30:23] - 8'd127);

  always @(posedge clk)
  begin
    Done <= 1'b0;
    V <= 1'b0;
    case(state)
    0: begin
      if(St == 1'b0)
        state <= 2'b00;
      else
        state <= 2'b01;
    end
    1: begin
      state <= 2'b10;
      S <= Mplier[31] ^ Mcand[31];
      if(Mplier == 32'h00000000 || Mcand == 32'h00000000) begin
        E <= 9'b000000000;
        E_Biased <= 9'b000000000;
        F <= mult_out[45:23];
      end
      else if(mult_out[47] == 1'b1) begin
        E <= add_out + 9'b000000001;
        E_Biased <= add_out + 9'd128;
        F <= mult_out[46:24];
      end
      else begin
        E <= add_out;
        E_Biased <= add_out + 9'd127;
        F <= mult_out[45:23];
      end
    end
    2: begin
      state <= 2'b00;
      Done <= 1'b1;
      if(E < 8'b10000010 && E > 8'b01111111)
        V <= 1'b1;
      Product <= {S, E_Biased[7:0], F};
    end
    default: begin
    end
    endcase
  end

endmodule
```

7.14

```
module fp_adder_test_bench(Test_Num);
  output reg Test_Num;

  reg CLK, ST;
  reg [31:0] INPUT;
  wire DONE, OVF,UNF;
  wire [31:0] SUM;
  parameter N  = 15;

  reg [31:0] addend_arr [1:N];
  reg [31:0] augend_arr [1:N];
  reg [31:0] result_arr [1:N];
  reg [1:N] ovf_arr;
  reg [1:N] unf_arr;
  integer i;

  initial begin
    CLK = 0;
    ST = 0;
    INPUT = 0;
    addend_arr[1]  = 32'b00000000000000000000000000000000;
    addend_arr[2]  = 32'b00111111100000000000000000000000;
    addend_arr[3]  = 32'b10111111100000000000000000000000;
    addend_arr[4]  = 32'b00111111100000000000000000000000;
    addend_arr[5]  = 32'b01111111011111111111111111111111;
    addend_arr[6]  = 32'b11111111011111111111111111111111;
    addend_arr[7]  = 32'b01111111011111111111111111111111;
    addend_arr[8]  = 32'b11111111011111111111111111111111;
    addend_arr[9]  = 32'b01000011111000000000000000000000;
    addend_arr[10] = 32'b11000011111000000000000000000000;
    addend_arr[11] = 32'b01111111011111111111111111111111;
    addend_arr[12] = 32'b11111111011111111111111111111111;
    addend_arr[13] = 32'b01111111011111111111111111111110;
    addend_arr[14] = 32'b11111111011111111111111111111110;
    addend_arr[15] = 32'b00000000110000000000000000000000;

    augend_arr[1]  = 32'b00000000000000000000000000000000;
    augend_arr[2]  = 32'b00111111100000000000000000000000;
    augend_arr[3]  = 32'b10111111100000000000000000000000;
    augend_arr[4]  = 32'b10111111100000000000000000000000;
    augend_arr[5]  = 32'b00111111100000000000000000000000;
    augend_arr[6]  = 32'b10111111100000000000000000000000;
    augend_arr[7]  = 32'b01111111011111111111111111111111;
    augend_arr[8]  = 32'b11111111011111111111111111111111;
    augend_arr[9]  = 32'b11000010111000000000000000000000;
    augend_arr[10] = 32'b01000010111000000000000000000000;
    augend_arr[11] = 32'b01110011100000000000000000000000;
    augend_arr[12] = 32'b11110011100000000000000000000000;
    augend_arr[13] = 32'b01110011100000000000000000000000;
    augend_arr[14] = 32'b11110011100000000000000000000000;
    augend_arr[15] = 32'b10000000100000000000000000000000;

    result_arr[1]  = 32'b00000000000000000000000000000000;
    result_arr[2]  = 32'b01000000000000000000000000000000;
    result_arr[3]  = 32'b11000000000000000000000000000000;
    result_arr[4]  = 32'b00000000000000000000000000000000;
    result_arr[5]  = 32'b01111111011111111111111111111111;
    result_arr[6]  = 32'b11111111011111111111111111111111;
    result_arr[7]  = 32'b00000000000000000000000000000000;
    result_arr[8]  = 32'b00000000000000000000000000000000;
    result_arr[9]  = 32'b01000011101010000000000000000000;
    result_arr[10] = 32'b11000011101010000000000000000000;
```

```
    result_arr[11] = 32'b00000000000000000000000000000000;
    result_arr[12] = 32'b00000000000000000000000000000000;
    result_arr[13] = 32'b01111111011111111111111111111111;
    result_arr[14] = 32'b11111111011111111111111111111111;
    result_arr[15] = 32'b00000000000000000000000000000000;

    ovf_arr = 15'b000000110011000;
    unf_arr = 15'b000000000000001;
  end

  always #10 CLK = ~CLK;

  FPADD FP_ADDER(CLK, ST, DONE, OVF, UNF, INPUT, SUM);

  always
  begin
    for(i = 1; i <= N; i = i + 1 ) begin
      Test_Num = i;
      @(negedge CLK)
        ST = 1;
        INPUT = addend_arr[i];
      @(negedge CLK)
        INPUT = augend_arr[i];
        ST = 0;
      wait (DONE == 1);
      if(OVF == 1 || UNF == 1) begin
        if(OVF != ovf_arr[i])
          $display("Overflow does not match");
        if(UNF != unf_arr[i])
          $display("Underflow does not match");
      end
    end
    $display("Tests Complete");
    $stop;
  end

endmodule
```

7.15

1) Make exponents equal.	F_1: 0.1011	E_1: 11111	$(11/16 \times 2^{-1})$
	F_2: 1.1101	E_2: 11111	$(-3/16 \times 2^{-1})$
2) Add the fractions	F: 10.1000	E: 11111	
Discard the carryout because 2's complement addition	F: 0.1000	E: 11111	$(8/16 \times 2^{-1})$

7.16 **(a)** The steps for floating point addition are listed in Section 7.3 of the text.

(b)

1) Make exponents equal.	F_1: 1.0101	E_1: 1001	$(-11/16 \times 2^{-6})$
	F_2: 0.0101	E_2: 1001	$(5/16 \times 2^{-6})$
2) Add the fractions.	F: 1.1010	E: 1001	$(-6/16 \times 2^{-6})$
3) Normalize the fraction.	F: 1.0100	E: 1000	$(-12/16 \times 2^{-7})$
4) Check for exponent overflow.	No overflow.		

(c)

```
module Prob7_16(CLK, St, F1, F2, E1, E2, V, Done);
  input CLK, St;
  input [4:0] F1, F2;
  input [3:0] E1, E2;
  output reg V, Done;

  reg [5:0] RegF1;
  reg [4:0] RegF2;
  reg [4:0] RegE1;
  reg [3:0] RegE2;
  wire [5:0] addout;
  wire [4:0] subout;
  reg Load, Ad, LM8, RSX1, LS, RSX2, INC1, INC2, DEC1, LdTwoToOne;
  wire LT, GT, E1G, E2G, EV, SubV;
  wire FZ, FV, Fnorm;
  reg [1:0] state, nextstate;

  initial begin
    V = 0;
    Done = 0;
    RegF1 = 0;
    RegF2 = 0;
    RegE1 = 0;
    RegE2 = 0;
    Load = 0;
    Ad = 0;
    LM8 = 0;
    RSX1 = 0;
    LS = 0;
    RSX2 = 0;
    INC1 = 0;
    INC2 = 0;
    DEC1 = 0;
    LdTwoToOne = 0;
    state = 0;
    nextstate = 0;
  end

  assign subout = RegE1 - {RegE2[3], RegE2};
  assign addout = RegF1 + {RegF2[4], RegF2};
  assign SubV = (subout[4] != subout[3])? 1 : 0;
  assign EV = (RegE1[4] != RegE1[3])? 1 : 0;
  assign LT = subout[3];
  assign GT = (subout[3] == 0 && subout[3:0] != 3'b000)? 1 : 0;
  assign E1G = ((SubV == 0 && subout[3] == 0 && subout[2] == 1 &&
                                                 (~(subout[1:0])==2'b00))
                                      ||
               (SubV == 1 && RegE1[3] == 0 && RegE2[3] == 1))? 1 : 0
          ;
  assign E2G = ((SubV == 0 && subout[3] == 1 && subout[2] == 0) ||
               (SubV == 1 && RegE1[3] == 1 && RegE2[3] == 0))? 1 : 0
          ;
  assign FZ = (RegF1 == 0)? 1 : 0;
  assign FV = (RegF1[5] != RegF1[4])? 1 : 0;
  assign Fnorm = (RegF1[4] != RegF1[3])? 1 : 0;

  always @(state, St, LT, GT, E1G, E2G, FZ, FV, Fnorm, EV)
  begin
  Load = 0; Ad = 0; LM8 = 0; RSX1 = 0; LS = 0; V = 0;
       RSX2 = 0; INC1 = 0; INC2 = 0; DEC1 = 0; Done = 0;
       LdTwoToOne = 0;
```

```
  case(state)
  0: begin
    if(St == 1) begin
      Load = 1;
      nextstate = 1;
    end
    else begin
      nextstate = 0;
    end
  end
  1: begin
    if(E1G == 1) begin
      nextstate = 3;
    end
    else if(E2G == 1) begin
      LdTwoToOne = 1;
      nextstate = 3;
    end
    else if(GT == 1) begin
      RSX2 = 1;
      INC2 = 1;
      nextstate = 1;
    end
    else if(LT == 1) begin
      RSX1 = 1;
      INC1 = 1;
      nextstate = 1;
    end
    else begin
      Ad = 1;
      nextstate = 2;
    end
  end
  2: begin
    if(FZ == 1) begin
      LM8 = 1;
      nextstate = 3;
    end
    else if(FV == 1) begin
      RSX1 = 1;
      INC1 = 1;
      nextstate = 3;
    end
    else if(Fnorm == 0) begin
      LS = 1;
      DEC1 = 1;
      nextstate = 2;
    end
    else begin
      nextstate = 3;
    end
  end
  3: begin
    Done =1 ;
    if(St == 0)
      nextstate = 0;
    else
      nextstate = 3;
    if(EV == 1)
      V = 1;
  end
  endcase
end
```

```
  always @(posedge CLK)
  begin
    state <= nextstate;
    if(Load == 1) begin
      RegF1 <= {F1[4],F1};
      RegF2 <= F2;
      RegE1 <= {E1[3],E1};
      RegE2 <= E2;
    end
    else begin
      //RegE1
      if(LM8 == 1)
        RegE1 <= 5'b11000;
      if(LdTwoToOne == 1)       begin
        RegE1 <= {RegE2[3], RegE2};
        RegF1 <= {RegF2[4], RegF2};
      end
      if(INC1 == 1)
        RegE1 <= RegE1 + 1;
      else if(DEC1 == 1)
        RegE1 <= RegE1 - 1;

      //RegE2
      if(INC2 == 1)
        RegE2 <= RegE2 + 1;

      //REGF1
      if(LS == 1)
        RegF1 <= {RegF1[4:0], 1'b0};
      else if(RSX1 == 1)
        RegF1 <= {RegF1[5], RegF1[5:1]};
      else if(Ad == 1)
        RegF1 <= addout;

      //RegF2
      if(RSX2 == 1)
        RegF2 <= {RegF2[4], RegF2[4:1]};
    end
  end

endmodule
```

7.17 (a)

```
module FPADD_P7_17a(CLK, St, done, ovf, unf, FPinput, FPsum);
  input CLK, St;
  output reg done, ovf, unf;
  input [31:0] FPinput;
  output [31:0] FPsum;

  reg [25:0] F1, F2;
  reg [7:0] E1, E2;
  reg S1,S2;

  wire FV, FU;
  wire [27:0] F1comp, F2comp, Addout, Fsum;
  reg [2:0] State;

  initial begin
    done = 0;
    ovf = 0;
    unf = 0;
    F1 = 0;
```

```
  F2 = 0;
  E1 = 0;
  E2 = 0;
  S1 = 0;
  S2 = 0;
  State = 0;
end

assign F1comp = (S1 == 1)?  ~{2'b00,F1} + 1 : {2'b00, F1};
assign F2comp = (S2 == 1)?  ~{2'b00,F2} + 1 : {2'b00, F2};
assign Addout = F1comp + F2comp;
assign Fsum = (Addout[27] == 0)?  Addout : ~Addout + 1;
assign FV = Fsum[27] ^ Fsum[26];
assign FU = !F1[25];
assign FPsum = {S1, E1, F1[24:2]};

always @(posedge CLK)
begin
  case(State)
  0: begin
    if(St == 1) begin
      if(FPinput[30:23] != 0)
        E1 <= FPinput[30:23];
      else
        E1 <= 8'b00000001;
      S1 <= FPinput[31];
      F1[24:0] <= {FPinput[22:0],2'b00};
      if(FPinput[30:23] == 0)
        F1[25] <= 0;
      else
        F1[25] <= 1;
      done <= 0;
      ovf <= 0;
      unf <= 0;
      State <= 1;
    end
  end
  1: begin
    if(FPinput[30:23] != 0)
      E2 <= FPinput[30:23];
    else
      E2 <= 8'b00000001;
    S2 <= FPinput[31];
    F2[24:0] <= {FPinput[2:0],2'b00};
    if(FPinput[30:23] == 0)
      F2[25] <= 0;
    else
      F2[25] <= 1;
    State <= 2;
  end
  2: begin
    if(F1 == 0 || F2 == 0) begin
      State <= 3;
    end
    else begin
      if(E1 == E2) begin
        State <= 3;
      end
      else if(E1 < E2) begin
        F1 <= {1'b0, F1[25:1]};
        E1 <= E1 + 1;
      end
      else begin
```

```
          F2 <= {1'b0, F2[25:1]};
          E2 <= E2 + 1;
        end
      end
    end
    3: begin
      S1 <= Addout[27];
      if(FV == 0) begin
        F1 <= Fsum[25:0];
      end
      else begin
        F1 <= Fsum[26:1];
        E1 <= E1 + 1;
      end
      State <= 4;
    end
    4: begin
      if(F1 == 0) begin
        E1 <= 0;
        State <= 6;
      end
      else if(E1 == 1 && F1[25] == 0) begin
        E1 <= 0;
        State <= 6;
      end
      else begin
        State <= 5;
      end
    end
    5: begin
      if(E1 == 1 && FU == 1) begin
        E1 <= 0;
        State <= 6;
      end
      else if(FU == 0) begin
        State <= 6;
      end
      else begin
        F1 <= {F1[24:0], 1'b0};
        E1 <= E1 - 1;
      end
    end
    6: begin
      if(E1 == 255)
        ovf <= 1;
      done <= 1;
      State <= 0;
    end
    endcase
  end

endmodule
```

(b)

```
module FPADD_P7_17b(CLK, St, FPinput, FPsum, done, ovf, unf);
  input CLK, St;
  input [31:0] FPinput;
  output [31:0] FPsum;
  output reg done, ovf, unf;

  reg [25:0] F1, F2;
  reg [7:0] E1,E2;
  reg S1, S2;
```

```
wire FV, FU;
wire [27:0] F1comp, F2comp, Addout, Fsum;
reg [2:0] State;

initial begin
  F1 = 0;
  F2 = 0;
  E1 = 0;
  E2 = 0;
  S1 = 0 ;
  S2 = 0;
  State = 0;
  done = 0;
  ovf = 0;
  unf = 0;
end

assign F1comp = (S1 == 1)? ~{2'b00, F1} + 1 : ~{2'b00, F1};
assign F2comp = (S2 == 1)? ~{2'b00, F2} + 1 : ~{2'b00, F2};
assign Addout = F1comp + F2comp;
assign Fsum = (Addout[27] == 0)? Addout : ~Addout + 1;
assign FV = Fsum[27] ^ Fsum[26];
assign FU = !F1[25];
assign FPsum = {S1, E1, F1[24:2]};

always @(posedge CLK)
begin
  case(State)
  0: begin
    if(St == 1) begin
      E1 <= FPinput[30:23];
      S1 <= FPinput[31];
      F1[24:0] <= {FPinput[22:0], 2'b00};
      if(FPinput == 0) begin
        F1[25] <= 0;
      end
      else begin
        F1[25] <= 1;
      end
      done <= 0;
      ovf <= 0;
      unf <= 0;
      State <= 1;
    end
  end
  1: begin
    E2 <= FPinput[30:23];
    S2 <= FPinput[31];
    F2[24:0] <= {FPinput[22:0], 2'b00};
    if(FPinput == 0)
      F2[25] <= 0;
    else
      F2[25] <= 1;
    State <= 2;
  end
  2: begin
    if(F1 == 0 || F2 == 0) begin
      State <= 3;
    end
    else if((E1 > E2) || ((E1 - E2) > 23)) begin
      State <= 6;
    end
    else if((E2 > E1) || ((E2 - E1) > 23)) begin
```

```
        State <= 6;
        F1 <= F2;
        E1 <= E2;
        S1 <= S2;
      end
      else begin
        if(E1 == E2) begin
          State <= 3;
        end
        else if(E1 < E2) begin
          F1 <= {1'b0, F1[25:1]};
          E1 <= E1 + 1;
        end
        else begin
          F2 <= {1'b0, F2[25:1]};
          E2 <= E2 + 1;
        end
      end
    end
    3: begin
      S1 <= Addout[27];
      if(FV == 0) begin
        F1 <= Fsum[25:0];
      end
      else begin
        F1 <= Fsum[26:1];
        E1 <= E1 + 1;
      end
      State <= 4;
    end
    4: begin
      if(F1 == 0) begin
        E1 <= 0;
        State <= 6;
      end
      else begin
        State <= 5;
      end
    end
    5: begin
      if(E1 == 0) begin
        unf <= 1;
        State <= 6;
      end
      else if(FU == 0) begin
        State <= 6;
      end
      else begin
        F1 <= {F1[24:0], 1'b0};
        E1 <= E1 - 1;
      end
    end
    6: begin
      if(E1 == 255) begin
        ovf <= 1;
      end
      done <= 1'b1;
      State <= 0;
    end
    endcase
  end

endmodule
```

(c)

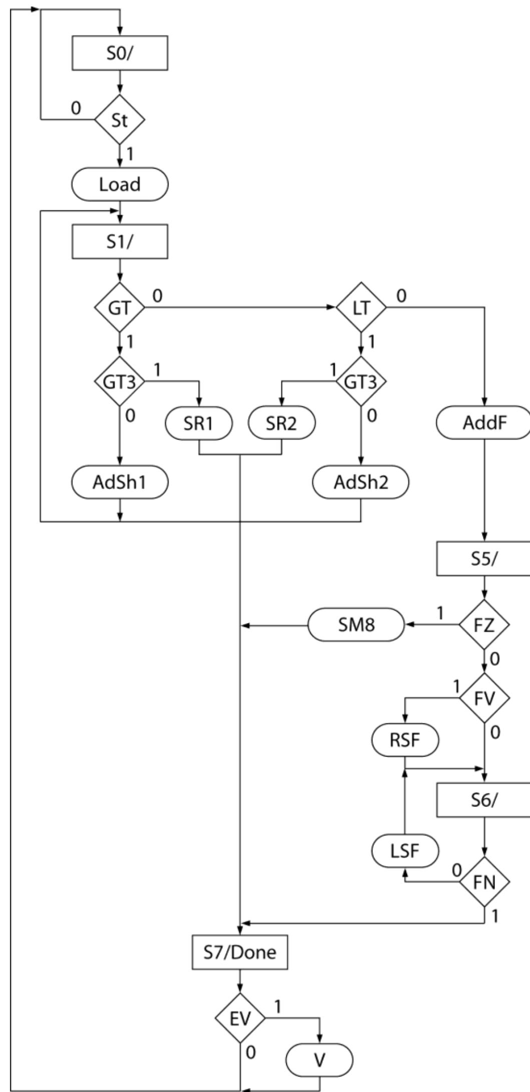
```
module FPADD_P7_17c(CLK, St, FPinput, FPsum, done, ovf, unf);
  input CLK, St;
  input [31:0] FPinput;
  output [31:0] FPsum;
  output reg done, ovf, unf;

  reg [25:0] F1, F2;
  reg [7:0] E1, E2;
  reg S1, S2;
  wire FV, FU;
  wire [27:0] F1comp, F2comp, Addout, Fsum;
  reg [2:0] State;

  initial begin
    F1 = 0;
    F2 = 0;
    E1 = 0;
    E2 = 0;
    S1 = 0;
    S2 = 0;
    State = 0;
    done = 0;
    ovf = 0;
    unf = 0;
  end

  assign Addout = F1comp + F2comp;
  assign Fsum = (Addout[27] == 0)? Addout : ~Addout + 1;
  assign FV = Fsum[27] ^ Fsum[26];
  assign FU = !F1[25];
  assign FPsum = {S1, E1, F1[24:2]};

  always @(posedge CLK)
  begin
    case(State)
    0: begin
      if(St == 1) begin
        E1 <= FPinput[30:23];
        S1 <= FPinput[31];
        F1[24:0] <= {FPinput[22:0], 2'b00};
        if(FPinput == 0) begin
          F1[25] <= 0;
        end
        else begin
          F1[25] <= 1;
        end
        done <= 0;
        ovf <= 0;
        unf <= 0;
        State <= 1;
      end
    end
    1: begin
      E2 <= FPinput[30:23];
      S2 <= FPinput[31];
      F2[24:0] <= {FPinput[22:0], 2'b00};
      if(FPinput == 0) begin
        F2[25] <= 0;
      end
      else begin
        F2[25] <= 1;
      end
```

```
  State <= 2;
end
2: begin
  if(F1 == 0 || F2 == 0) begin
    State <= 3;
  end
  else begin
    if(E1 == E2) begin
      State <= 3;
    end
    else if(E1 < E2) begin
      F1 <= {1'b0, F1[25:1]};
      E1 <= E1 + 1;
    end
    else begin
      F2 <= {1'b0, F2[25:1]};
      E2 <= E2 + 1;
    end
  end
end
3: begin
  S1 <= Addout[27];
  if(FV == 0) begin
    F1 <= Fsum[25:0];
  end
  else begin
    if(Fsum[2] == 0) begin
      F1 <= Fsum[26:1];
    end
    else begin
      F1 <= Fsum[26:1] + 26'b00000000000000000000000100;
    end
    E1 <= E1 + 1;
  end
  State <= 4;
end
4: begin
  if(F1 == 0) begin
    E1 <= 0;
    State <= 6;
  end
  else begin
    State <= 5;
  end
end
5: begin
  if(E1 == 0) begin
    unf <= 1;
    State <= 6;
  end
  else if(FU == 0) begin
    State <= 6;
  end
  else begin
    F1 <= {F1[24:0],1'b0};
    E1 <= E1 - 1;
  end
end
6: begin
  if(E1 == 255) begin
    ovf <= 1;
  end
  done <= 1;
```

```
      State <= 0;
    end
    endcase
  end

endmodule
```

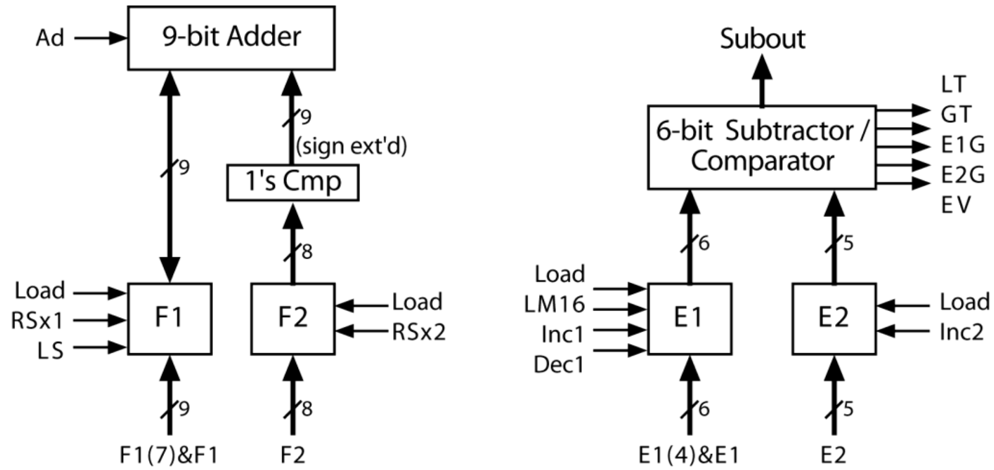
7.18 **(a)** 1.) Add 2 to E_2, shift F_2 right twice so exponents match: $E_2 = 011 + 010 = 101$; $F_2 = 0.00101$
2.) Add fractions: $0.11100 + 0.00101 = 1.00001$
3.) Normalize fraction and adjust exponent: $F = 0.100001$, $E = 110$

(b)



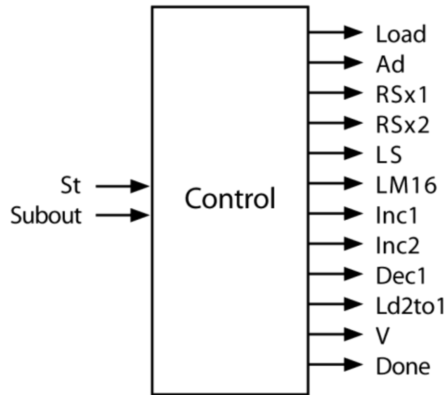
St - Start
GT = $E_1 > E_2$
LT = $E_1 < E_2$
Load = load all registers
GT3 = $|E_1 - E_2| > 3$
SR1 = Set result to $F = F_1$ and $E = E_1$
SR2 = Set result to $F = F_2$ and $E = E_2$
AdSh1 = Add 1 to E_1, right shift F_1
AdSh2 = Add 1 to E_2, right shift F_2
AddF = Add fractions
SM8 = Set exponent to -8
LSF = left shift fraction, dec. exponent
RSF = right shift fraction, inc. exponent
FZ = fraction zero
FV = fraction overflow
FN = fraction normalized
Done = done flag
EV = exponent overflow
V = overflow flag

7.19 (a)



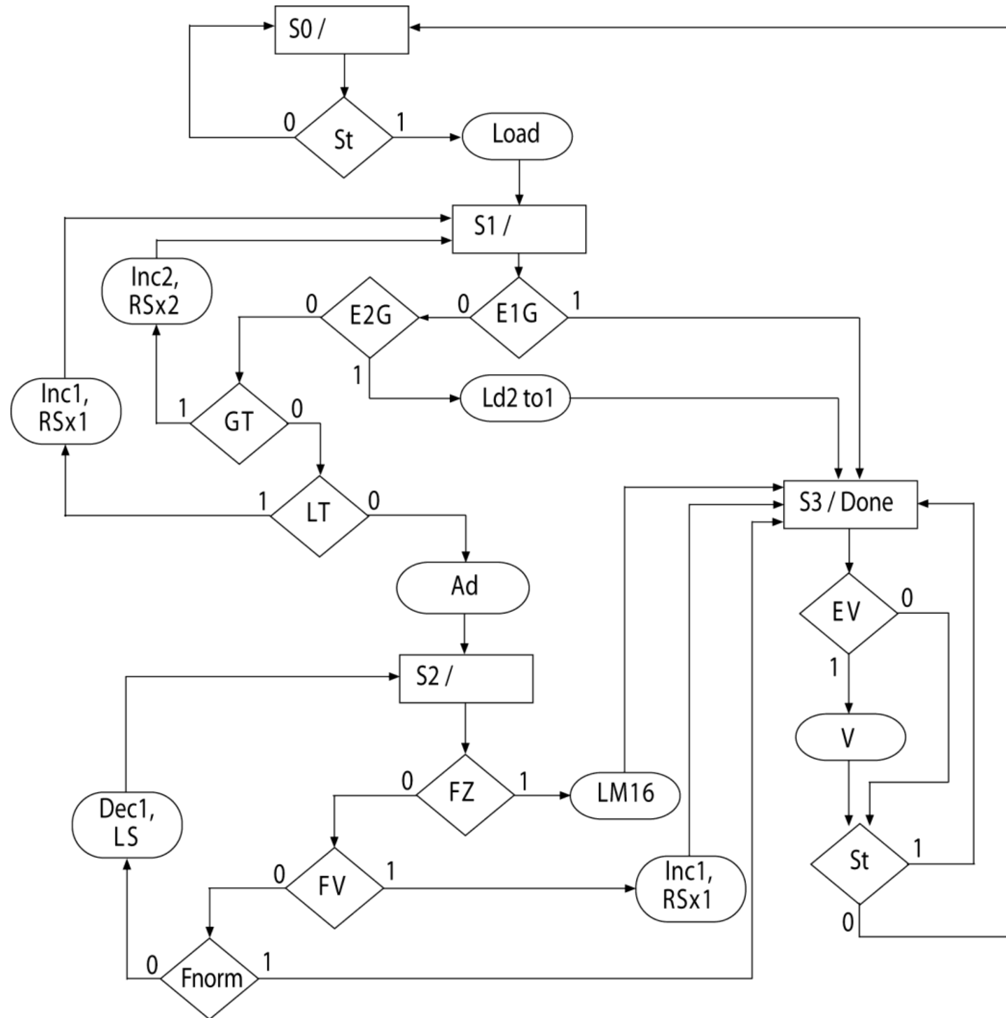
F1: 9-bit L-R shift register
F2: 8-bit L-R shift register
E1: 6-bit U-D counter
E2: 5-bit counter

LT: '1' when E1 < E2
GT: '1' when E1 > E2
EV: exponent overflow
E1G: '1' when E1–E2 > 7
E2G: '1' when E2–E1 > 7



St: start floating point addition
Load: load registers
Ad: enable adder output into F1
RSx1: shift F1 right, sign extended
RSx2: shift F2 right, sign extended
LS: shift F1 left
LM16: load minus 16 into E1
Inc1: increment E1
Inc2: increment E2
Dec1: decrement E1
Ld2to1: transfer F2 to F1, E2 to E1
Done: fraction addition is done
V: Overflow indicator

(b)



(c)

```
module Prob7_19(CLK, St, F1, F2, E1, E2, V, Done);
  input CLK, St;
  input [7:0] F1, F2;
  input [4:0] E1, E2;
  output reg V, Done;

  reg [8:0] RegF1;
  reg [7:0] RegF2;
  reg [5:0] RegE1;
  reg [4:0] RegE2;
  wire [8:0] addout;
  wire [5:0] Subout;
  reg Load, Ad, LM8, RSX1, LS, RSX2, INC1, INC2, DEC1, LdTwoToOne;
  wire LT, GT, E1G, E2G, EV, SubV;
  wire FZ, FV;
  reg [1:0] state, nextstate;

  initial begin
    V = 0;
    Done = 0;
    RegF1 = 0;
    RegF2 = 0;
```

```
  RegE1 = 0;
  RegE2 = 0;
  Load = 0;
  Ad = 0;
  LM8 = 0;
  RSX1 = 0;
  LS = 0;
  RSX2 = 0;
  INC1 = 0;
  INC2 = 0;
  DEC1 = 0;
  LdTwoToOne = 0;
  state = 0 ;
  nextstate = 0;
end

assign Subout = RegE1 - {RegE2[4], RegE2};
assign addout = RegF1 - {RegF2[7], RegF2};
assign SubV = (Subout[5] != Subout[4])? 1 : 0;
assign EV = (RegE1[5] != RegE1[4])? 1 : 0;
assign LT = Subout[4];
assign GT = (Subout[4] == 0 && Subout[3:0] != 0)? 1 : 0;
assign E1G = ((SubV == 0 && Subout[4] == 0 && Subout[3] == 1) ||
             (SubV == 1 && RegE1[4] == 0 && RegE2[4] == 1))? 1 : 0;
assign E2G = ((SubV == 0 && Subout[4] == 1 && Subout[3] == 0) ||
             (SubV == 0 && Subout[4:0] == 5'b11000) ||
             (SubV == 1 && RegE1[4] == 1 && RegE2[4] == 0))? 1 : 0;
assign FZ = (RegF1 == 0)? 1 : 0;
assign FV = (RegF1[8] != RegF1[7])? 1 : 0;
assign Fnorm = (RegF1[7] != RegF1[6])? 1 : 0;

always @(state, St, LT, GT, E1G, E2G, FZ, FV, Fnorm, EV)
begin
  Load <= 0; Ad <= 0; LM8 <= 0; RSX1 <= 0; LS <= 0; V <= 0;
  RSX2 <= 0; INC1 <= 0; INC2 <= 0; DEC1 <= 0; Done <= 0;
  LdTwoToOne <= 0;
  case(state)
  0: begin
    if(St == 1) begin
      Load = 1;
      nextstate = 1;
    end
    else begin
      nextstate = 0;
    end
  end
  1: begin
    if(E1G == 1) begin
     nextstate = 3;
    end
    else if(E2G == 1) begin
      LdTwoToOne = 1;
      nextstate = 3;
    end
    else if(GT == 1) begin
      RSX2 = 1;
      INC2 = 1;
      nextstate = 1;
    end
    else if(LT == 1) begin
      RSX1 = 1;
      INC1 = 1;
      nextstate = 1;
```

```
    end
    else begin
      Ad = 1;
      nextstate = 2;
    end
  end
  2: begin
    if(FZ == 1) begin
      LM8 = 1;
      nextstate = 3;
    end
    else if(FV == 1) begin
      RSX1 = 1;
      INC1 = 1;
      nextstate = 3;
    end
    else if(Fnorm == 0) begin
      LS = 1;
      DEC1 = 1;
      nextstate = 2;
    end
    else begin
      nextstate = 3;
    end
  end
  3: begin
    Done = 1;
    if(St == 0) begin
      nextstate = 0;
    end
    else begin
      nextstate = 3;
    end
    if(EV == 1) begin
      V = 1;
    end
  end
  endcase
end

always @(posedge CLK)
begin
  state <= nextstate;
  if(Load == 1) begin
    RegF1 <= {F1[7], F1};
    RegF2 <= F2;
    RegE1 <= {E1[4], E1};
    RegE2 <= E2;
  end
  else begin
    if(LM8 == 1) begin
      RegE1 <= 6'b110000;
    end
    if(LdTwoToOne == 1) begin
      RegE1 <= {RegE2[4], RegE2};
      RegF1 <= {RegF2[7], RegF2};
    end
    if(INC1 == 1) begin
      RegE1 <= RegE1 + 1;
    end
    else if(DEC1 == 1) begin
      RegE1 <= RegE1 - 1;
    end
```

```
      if(INC2 == 1) begin
        RegE2 <= RegE2 + 1;
      end
      if(LS == 1) begin
        RegF1 <= {RegF1[7:0],1'b0};
      end
      else if(RSX1 == 1) begin
        RegF1 <= {RegF1[8], RegF1[8:1]};
      end
      else if(Ad == 1) begin
        RegF1 <= addout;
      end
      if(RSX2 == 1) begin
        RegF2 <= {RegF2[7], RegF2[7:1]};
      end
    end
  end

endmodule
```

7.20 **(a)** The steps for floating point subtraction are the same those for addition listed in section 7.3 of the text with the exception of step 2, where the fractions are subtracted rather than added.

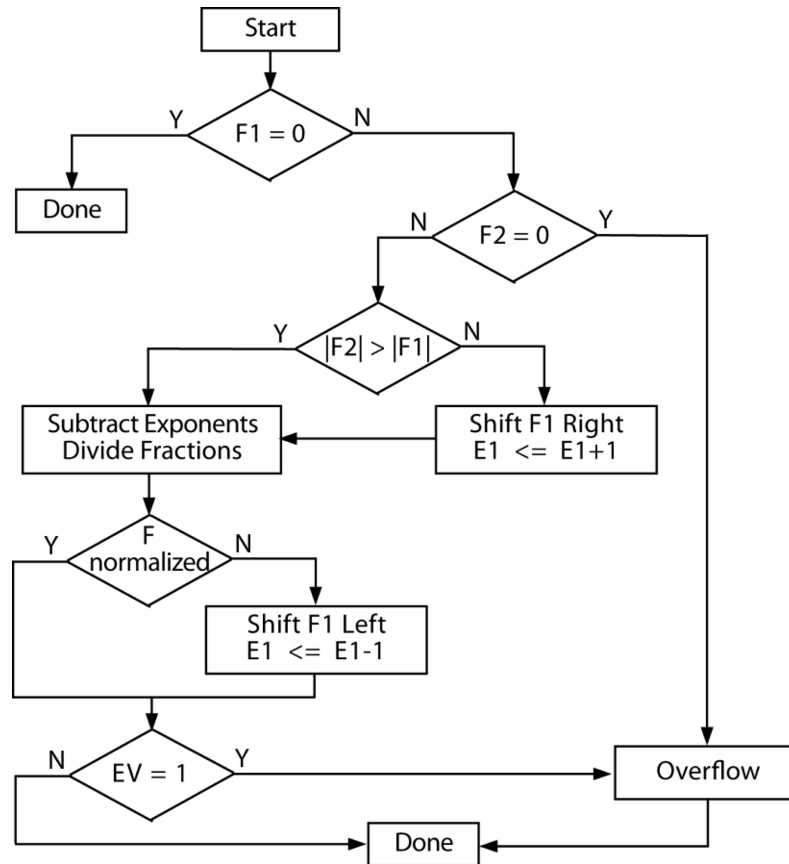
(b)

1) Make exponents equal.	F_1: 1.0111	E_1: 11101	$(-9/16 \times 2^{-3})$
	F_2: 1.110101	E_2: 11101	$(-11/64 \times 2^{-3})$
2) Subtract the fractions.	F: 1.100111	E: 11101	$(-25/64 \times 2^{-3})$
3) Normalize the fraction.	F: 1.00111	E: 11100	$(-25/32 \times 2^{-4})$
4) Check for exponent overflow.	No overflow.		

(c) The Verilog code is the same as listed in Problem 7.16 (c), with the 2nd line in the list of concurrent assignments
(`assign addout = RegF1 + {RegF2[4], RegF2};`) replaced with:

```
assign addout = RegF1 - {RegF2[4], RegF2};
```

7.21 (a)



(b) Step 1) $|F_2| \le |F_1|$, so shift F_1 right & increment E_1.

		F_1: 0.0111000	E_1: 00100 $(56/128 \times 2^4)$
		F_2: 1.0110000	E_2: 11110 $(-80/128 \times 2^{-2})$
Step 2)	Divide. Result is...	F: 1.0100111	E: 00110 $(-89/128 \times 2^6)$

(c)

```
  `define F2GTF1 CompareResult[7]
module Prob7_21(CLK, St, F1, F2, E1, E2, Fout, Eout, V, Done);
  input CLK, St;
  input [7:0] F1, F2;
  input [4:0] E1, E2;
  output [7:0] Fout;
  output [4:0] Eout;
  output reg V, Done;

  reg [7:0] RegF1;
  reg [7:0] RegF2;
  reg [5:0] RegE1;
  reg [4:0] RegE2;
  wire [5:0] Subout;
  wire [7:0] CompF1, ToCompF1, ToCompF2, CompareResult;
  reg Load, StD, RSF, LSF, Store;
  wire Cm, EV, F1Z, F2Z, Fnorm;
  reg [2:0] state, nextstate;
  wire [7:0] dividerResult;
  wire DivDone;
```

```
initial begin
  V = 0;
  Done = 0;
  RegF1 = 0;
  RegF2 = 0;
  RegE1 = 0;
  RegE2 = 0;
  Load = 0;
  StD = 0;
  RSF = 0;
  LSF = 0;
  Store = 0;
  state = 0;
  nextstate = 0;
end

Divider fdivider(CLK, StD, RegF1, RegF2, dividerResult, DivDone);

assign CompF1 = ~(RegF1) + 1;
assign ToCompF1 = (RegF1[7] == 0)? RegF1 : CompF1;
assign ToCompF2 = (RegF2[7] == 1)? RegF2 : ~RegF2;
assign Cm = !RegF2[7];
assign CompareResult = ToCompF1 + ToCompF2 + {7'b0000000, Cm};
assign Subout = RegE1 - {RegE2[4], RegE2};
assign Fout = RegF1;
assign Eout = RegE1[4:0];
assign EV = (RegE1[5] != RegE1[4])? 1 : 0;
assign F1Z = (RegF1[7:5] == 0)? 1 : 0;
assign F2Z = (RegF2[7:6] == 0)? 1 : 0;
assign Fnorm = (RegF1[7] != RegF1[6])? 1 : 0;

always @(state, St, F1Z, F2Z, `F2GTF1, Fnorm, EV, DivDone)
begin
  Load = 0; RSF = 0; LSF = 0; V = 0;
      Done = 0; StD <= 0; Store = 0;
  case(state)
  0: begin
    if(St == 1) begin
      Load = 1;
      nextstate = 1;
    end
    else begin
      nextstate = 0;
    end
  end
  1: begin
    if(F1Z == 1'b1 || F2Z == 1'b1) begin
      nextstate = 4;
    end
    else if(`F2GTF1 == 0) begin
      RSF = 1;
      nextstate = 1;
    end
    else begin
      StD = 1;
      nextstate = 2;
    end
  end
  2: begin
    if(DivDone == 1) begin
      Store = 1;
      nextstate = 3;
    end
```

```
      else begin
        nextstate = 2;
      end
    end
    3: begin
      if(Fnorm == 1) begin
        nextstate = 4;
      end
      else begin
        LSF = 1;
        nextstate = 3;
      end
    end
    4: begin
      Done = 1'b1;
      if(EV == 1 || F2Z == 1) begin
        V = 1;
      end
      if(St == 0) begin
        nextstate = 0;
      end
      else begin
        nextstate = 4;
      end
    end
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    if(Load == 1) begin
      RegF1 <= F1;
      RegF2 <= F2;
      RegE1 <= {E1[4], E1};
      RegE2 <= E2;
    end
    else if(LSF == 1) begin
      RegF1 <= {RegF1[6:0], 1'b0};
      RegE1 <= RegE1 - 1;
    end
    else if(RSF == 1) begin
      RegF1 <= {RegF1[7], RegF1[7:1]};
      RegE1 <= RegE1 + 1;
    end
    else if(StD == 1) begin
      RegE1 <= Subout;
    end
    else if(Store == 1) begin
      RegF1 <= dividerResult;
    end
  end

endmodule
```

7.22 $2^{40} + (-2^{40} + 1) = 2^{40} + -2^{40} = 0$
$(2^{40} + -2^{40}) + 1 = 0 + 1 = 1$

7.23 $2^{40} + (-2^{40} + 1) = 2^{40} + -2^{40} + 1$ (difference in exponents ≤ 52) $= 1$
$(2^{40} + -2^{40}) + 1 = 0 + 1 = 1$

7.24 $2^{65} + (-2^{65} + 1) = 2^{65} + -2^{65} = 0$
$(2^{65} + -2^{65}) + 1 = 0 + 1 = 1$

7.25 $2^{65} + (-2^{65} + 1) = 2^{65} + -2^{65} = 0$
$(2^{65} + -2^{65}) + 1 = 0 + 1 = 1$

Chapter 8: Additional Topics in Verilog

8.1

```
function [7:0] shift_count;
  input [7:0] word;
  reg [7:0] count;
  begin
      for (count = 8'd0; word[7] == 1'b0; word = word << 1)
      begin
        count = count + 1;
  end
  shift_count = count;
   end
endfunction
```

8.2 **(a)**

```
function [N-1:0] comp2;
  input [N-1:0] vect;
  input [31:0] size;c

  reg [N-1:0] comp;
  reg firstone;
  integer i;

  begin
    firstone = 1'b0;
    for (i=0; i<size; i=i+1)
    begin
      if(!firstone)
      begin
        if(vect[i] == 1'b1)
        begin
          firstone = 1'b1;
        end
        comp[i] = vect[i];
      end
      else
      begin
        comp[i] = ~vect[i];
      end
    end
    comp2 = comp;
  end
endfunction
```

(b)

```
module test_comp2(bit_vec, out_vec);
  parameter N = 8; // N is the length of the bit vectors

  input [N-1:0] bit_vec;
  output [N-1:0] out_vec;

  assign out_vec = comp2(bit_vec, N);
endmodule
```

8.3 Code to implement factorial:

```
if(num == 32'd0 || num == 32'd1)
  factorial = 1;
else
  factorial = num * factorial(num-1);
```

Code to compute the factorial of 9:

```
result = factorial(9);
```

8.4 **(a)**

```
function GT;
  input [N-1:0] num1, num2;
  input [31:0] size;

  integer i;

  begin
    GT = 0;
    for (i = size - 1; i >= 0; i = i - 1)
    begin
      if (num1[i] & ~num2[i])
      begin
        GT = 1;
      end
    end
  end
endfunction
```

(b)

```
module TEST_GT(A, B, C);
  parameter N = 8; // N is the length of the bit vectors

  input [N-1:0] A, B;
  output C;

  assign C = GT(A, B, N);

endmodule
```

8.5

- Verilog functions must have at least one input and can only return one value (no outputs or inouts);
- Verilog tasks can have any numbers of inputs, outputs, or inouts
- Verilog tasks can contain time-controlled statements; Verilog functions cannot
- Verilog tasks can call other tasks and functions; Verilog functions can only call other functions

8.6

```
module SortTen ();
  integer ARRAY [9:0];
  integer file, count, k;

  initial
  begin
    //read the numbers from the file and put them in ARRAY
    file = $fopen("sort.txt","r");
    for (k=0; k<10; k=k+1)
    begin
      count = $fscanf(file, "%d", ARRAY[k]);
    end
    $fclose(file);

    //sort the numbers
    sort(ARRAY[0], ARRAY[1], ARRAY[2], ARRAY[3], ARRAY[4],  //inputs
         ARRAY[5], ARRAY[6], ARRAY[7], ARRAY[8], ARRAY[9],  //inputs
         ARRAY[0], ARRAY[1], ARRAY[2], ARRAY[3], ARRAY[4],  //outputs
         ARRAY[5], ARRAY[6], ARRAY[7], ARRAY[8], ARRAY[9]); //outputs
  end

  task sort;
```

```
    input [31:0] in0, in1, in2, in3, in4, in5, in6, in7, in8, in9;
    output [31:0] out0, out1, out2, out3, out4, out5, out6, out7, out8,
out9;

    reg [31:0] t [9:0]; //temporary array used for indexing the elements
    reg [31:0] temp; //temporary register for swapping

    integer i, j;

    begin
      t[0] = in0; t[1] = in1; t[2] = in2; t[3] = in3; t[4] = in4;
      t[5] = in5; t[6] = in6; t[7] = in7; t[8] = in8; t[9] = in9;

      for (i=0; i<10; i=i+1)
      begin
        for (j=0; j<10; j=j+1)
        begin
          if (t[j] > t[j+1])
          begin
            temp = t[j];
            t[j] = t[j+1];
            t[j+1] = temp;
          end
        end
      end

    out0 = t[0]; out1 = t[1]; out2 = t[2]; out3 = t[3]; out4 = t[4];
    out5 = t[5]; out6 = t[6]; out7 = t[7]; out8 = t[8]; out9 = t[9];
    end
  endtask

endmodule
```

8.7 **(a)**

```
task count_ones;
  input [4:0] size;
  input [N-1:0] bitvec;
  output [4:0] numOnes;

  integer i;

  begin
    numOnes = 5'd0;
    for (i = 0; i < size; i = i + 1)
    begin
      if (bitvec[i])
        numOnes = numOnes + 5'd1;
    end
  end
endtask
```

(b)

```
module test_countOnes(A, B);
  parameter N = 8; // N is the length of the bit vector

  input [N-1:0] A;
  output reg [4:0] B;

  always @(A)
    count_ones(N, A, B);

endmodule
```

8.8

```
module BCD (CLK, LD, BCDin, Cin, Cout);
```

```
  input CLK, LD;
  input [15:0] BCDin;
  input Cin;
  output reg Cout;

  reg [15:0] BCDacc;
  reg [3:1] C; // internal carry signals

  initial
  begin
    BCDacc = 16'b0000000000000000;
    C = 3'b000;
  end

  always @(posedge CLK, posedge LD)
  begin
    if(LD == 1'b1)
    begin
      addBCD4(BCDin[3:0], BCDacc[3:0], Cin, BCDacc[3:0], C[1]);
      addBCD4(BCDin[7:4], BCDacc[7:4], C[1], BCDacc[7:4], C[2]);
      addBCD4(BCDin[11:8], BCDacc[11:8], C[2], BCDacc[11:8], C[3]);
      addBCD4(BCDin[15:12], BCDacc[15:12], C[3], BCDacc[15:12], Cout);
    end
  end

  task addBCD4;
    input [3:0] x, y;
    input c_in;
    output [3:0] sum;
    output c_out;

    begin
      if (x+y+c_in>4'd9)
        {c_out,sum} = x+y+c_in+4'd6;
      else
        {c_out,sum} = x+y+c_in;
    end
  endtask
endmodule
```

8.9

Time	B	C	
0	0000	0000	
5	0110	0000	call P1
5 + Δ	0110	1100	
5 + 2Δ	0110	0110	continuous assignment
6	0110	1100	call P1

When B changes, the first call to P1 is given priority over the continuous assignment. However, during the second call to P1 one second later at time = 6, the continuous assignment does not occur because the right hand side (B) doesn't change.

8.10 **(a)** Yes, the bit 3 (d1) from left is wrong. The original data is 0110, and the correct code word is 0110011.

(b) p4 = d2 xnor d3 xnor d4;
p2 = d1 xnor d3 xnor d4;
p1 = d1 xnor d2 xnor d4;

S3 = p4 xnor d2 xnor d3 xnor d4;
S2 = p2 xnor d1 xnor d3 xnor d4;
S1 = p1 xnor d1 xnor d2 xnor d4;

(c)

```
module error_detector(data, PARITY, Syndrome);
  input [6:0] data;
  input PARITY;
  output [2:0] Syndrome;

  assign Syndrome[2] = (PARITY)? (data[3] ^ data[4] ^ data[5] ^
data[6]) : (data[3] ~^ data[4] ~^ data[5] ~^ data[6]);
  assign Syndrome[1] = (PARITY)? (data[1] ^ data[2] ^ data[5] ^
data[6]) : (data[1] ~^ data[2] ~^ data[5] ~^ data[6]);
  assign Syndrome[0] = (PARITY)? (data[0] ^ data[2] ^ data[4] ^
data[6]) : (data[0] ~^ data[2] ~^ data[4] ~^ data[6]);
endmodule
```

(d)

```
task error_p_even;
  input [6:0] data;
  output [2:0] Syndrome;
  begin
    Syndrome[2] =  data[3] ^ data[4] ^ data[5] ^ data[6];
    Syndrome[1] =  data[1] ^ data[2] ^ data[5] ^ data[6];
    Syndrome[0] =  data[0] ^ data[2] ^ data[4] ^ data[6];
  end
endtask

task error_p_odd;
  input [6:0] data;
  output [2:0] Syndrome;
  begin
    Syndrome[2] =  data[3] ~^ data[4] ~^ data[5] ~^ data[6];
    Syndrome[1] =  data[1] ~^ data[2] ~^ data[5] ~^ data[6];
    Syndrome[0] =  data[0] ~^ data[2] ~^ data[4] ~^ data[6];
  end
endtask
```

8.11

```
module Circuit(A, B, C, D, E, Y);
  input A, B, C, D, E;
  output Y;

  wire X1, X2;

  and #(15) and1 (X1, A, B, C);
  xor #(14, 16) xor1 (X2, D, E);
  nor #(12, 14) nor1 (Y, X1, X2);

endmodule
```

8.12

```
primitive jkff(q, clk, j, k, c, p);
  output q;
  input clk, j, k;
  input c, p;

  reg q;
```

```
  table
  //clk  j   k   c   p     q   q+
     ?   ?   ?   1   0   : ? : 0 ; //clear
     ?   ?   ?   0   1   : ? : 1 ; //preset
    (01) 0   0   0   0   : ? : - ; //clock rising edge
    (01) 0   1   0   0   : ? : 0 ;
    (01) 1   0   0   0   : ? : 1 ;
    (01) 1   1   0   0   : 0 : 1 ;
    (01) 1   1   0   0   : 1 : 0 ;
    (10) ?   ?   0   0   : ? : - ; //clock falling edge
     ?   ?   ?   0   0   : ? : - ; //steady clock
  endtable
endprimitive
```

8.13

```
primitive odd_parity(p, x3, x2, x1, x0);
  output p;
  input x3, x2, x1, x0;

  table
 //x3 x2 x1 x0   p
    0  0  0  0 : 1 ;
    0  0  0  1 : 0 ;
    0  0  1  0 : 0 ;
    0  0  1  1 : 1 ;
    0  1  0  0 : 0 ;
    0  1  0  1 : 1 ;
    0  1  1  0 : 1 ;
    0  1  1  1 : 0 ;
    1  0  0  0 : 0 ;
    1  0  0  1 : 1 ;
    1  0  1  0 : 1 ;
    1  0  1  1 : 0 ;
    1  1  0  0 : 1 ;
    1  1  0  1 : 0 ;
    1  1  1  0 : 0 ;
    1  1  1  1 : 1 ;
  endtable
endprimitive
```

8.14 **(a)** and **(b)**

```
module P8_14(A, B, C, D);
  input A, B;
  output reg C, D;

  reg RegB;

  // partA
  always @(A)
  begin
    RegB = B;
    #3
    if (RegB == B)
      D = ~D;

    #2 C = ~A;
  end

  // partB
```

```
  specify
    $setup (B, posedge A, 2);
    $hold (posedge A, B, 1);
    $width (negedge B, 10);
  endspecify
endmodule
```

8.15

```
module addr_decoder(addr, check, Sel);
  input [7:0] addr;
  input [5:0] check;
  output Sel;

  assign Sel = isEqual(addr[7:2], check[5:0]);

  function isEqual;
    input [5:0] addr_vec, check_vect;

    reg result;
    integer i;

    begin
      result = 1'b1;
      for (i=0; i<6; i=i+1)
      begin
        if (addr_vec[i]!=check_vect[i] && check_vect[i]!==1'bX)
        begin
          result = 1'b0;
        end
      end
      isEqual = result;
    end
  endfunction

endmodule
```

8.16

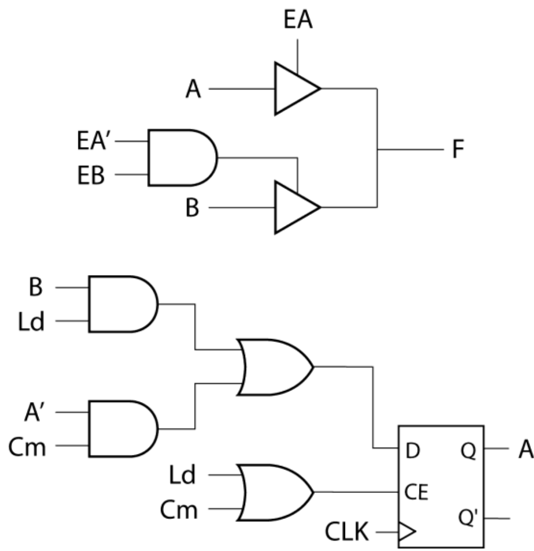
```
module OctDFF (D, OE_b, CLK, Q);
  input D, OE_b, CLK;
  output Q;

  reg store;

  specify
    $width (posedge CLK, 15);
    $setup (D, posedge CLK, 15);
    $hold (posedge CLK, D, 5);
  endspecify

  always @(posedge CLK)
  begin
    if (D === 1'bZ || D === 1'bX)
      store <= 1'bX;
    else
      store <= D;
  end

  assign Q = (OE_b == 1'b1) ? 1'bZ :
             (OE_b == 1'b0 && CLK !== 1'bX) ? store : 1'bX;
endmodule
```

8.17

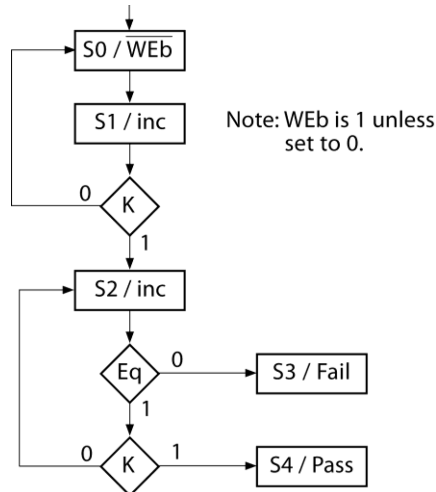
```
function cmpZ;
```

```
  input [7:0] a, b;

    reg result;
    integer i;

    begin
      result = 1'b1;
      for (i=0; i<8; i=i+1)
      begin
        if (a[i]!==b[i] || a[i]===1'bX || b[i]===1'bX)
        begin
          result = 1'b0;
        end
      end

      cmpZ = result;
    end
  endfunction
```

8.18



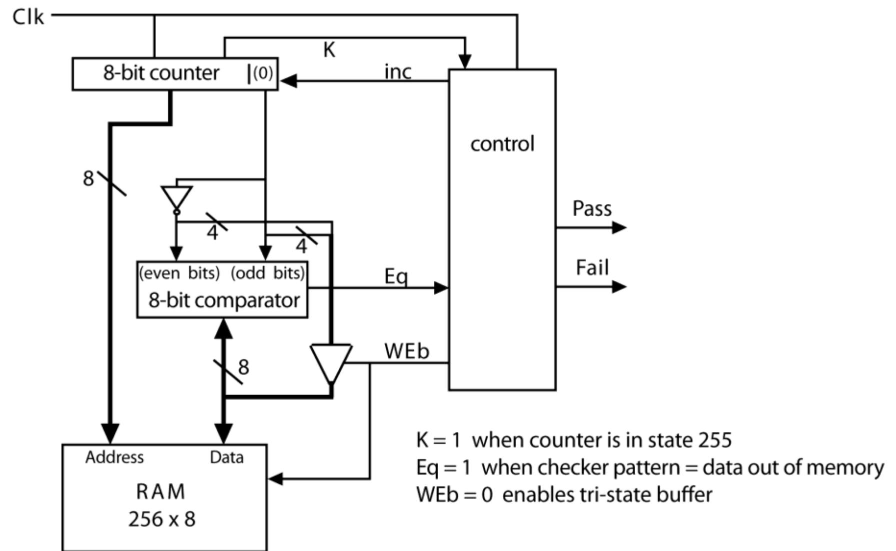
8.19 **(a)**

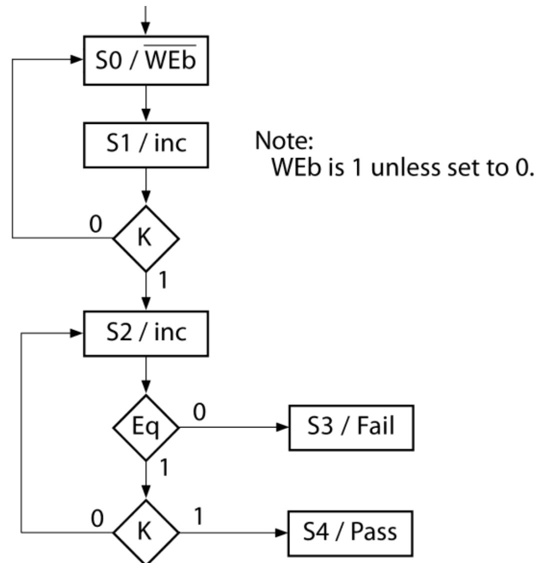


(b)
```
module top (CLK, Pass, Fail);
```

```
input CLK;
output reg Pass, Fail;

integer state, nextstate;
wire K, Eq;
reg inc, WEb;
reg [7:0] CNTR;
wire [7:0] RAMbus;

assign K = (CNTR == 8'hFF) ? 1 : 0;
assign Eq = (CNTR == RAMbus) ? 1 : 0;
assign RAMbus = (WEb == 1'b0) ? CNTR : 8'bZZZZZZZZ;

//module RAM_block (CS_b,WE_b,OE_b,Address,IO);
RAM_block RAM256(1'b0, WE_b, 1'b0, CNTR, RAMbus);

initial
begin
  inc = 0;
  WEb = 1;
  CNTR = 8'b00000000;
  state = 0;
  nextstate = 0;
end

always @(state, K, Eq)
begin
  WEb = 1'b1; inc = 0; Fail = 0; Pass = 0;
  case (state)
    0 : begin
      WEb = 0; nextstate = 1;
    end
    1 : begin
      inc = 1'b1;
      if (K == 0)
        nextstate = 0;
      else
        nextstate = 2;
    end
    2 : begin
      inc = 1'b1;
      if (Eq == 0)
        nextstate = 3;
      else if (K == 0)
        nextstate = 2;
      else
        nextstate = 4;
    end
    3 : Fail = 1'b1;
    4 : Pass = 1'b1;
  endcase
end

always@(posedge CLK)
begin
  state <= nextstate;
  if(CNTR == 8'hFF) CNTR <= 8'h00;
  else CNTR <= CNTR + 1'b1;
end

endmodule
```

8.20

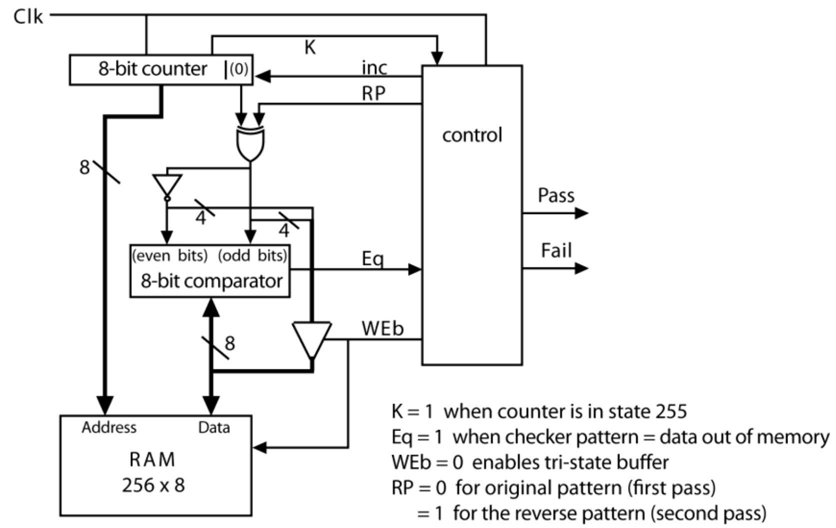




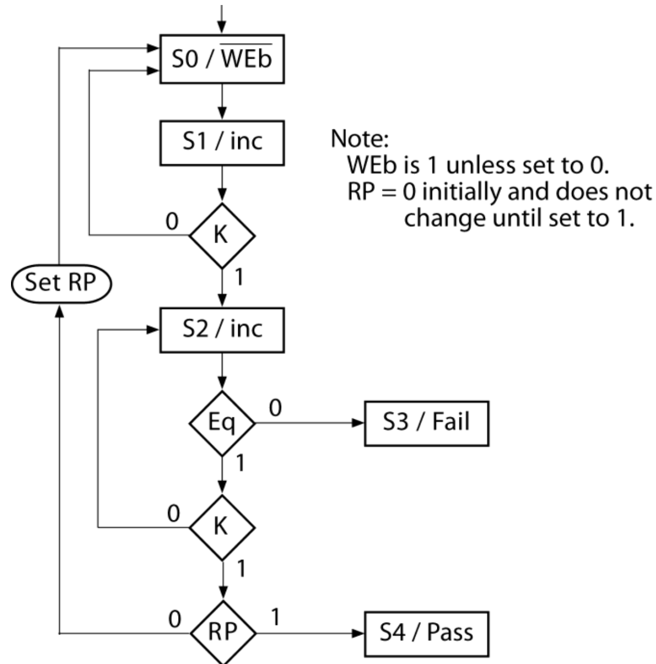

```
module Chex_test (CLK, Pass, Fail);
  input CLK;
  output reg Pass, Fail;

  integer state, nextstate;
  wire oddbit, evenbit;
  wire K, Eq;
  reg WE_b, inc;
  reg [7:0] CNTR;
  wire [7:0] DATA, RAMbus;

  initial
  begin
    WE_b = 1'b1;
    inc = 1'b0;
    CNTR = 8'h00;
    state = 0;
    nextstate = 0;
```

```
  end

  assign K = (CNTR == 8'hFF) ? 1'b1 : 1'b0;
  assign Eq = (DATA == RAMbus)? 1'b1 : 1'b0; //  comparator
  assign oddbit = CNTR[0]; //  odd bit generator
  assign evenbit = ~oddbit; //  even bit generator
  assign DATA = {oddbit, evenbit, oddbit, evenbit, oddbit, evenbit,
oddbit, evenbit};
  assign RAMbus = (WE_b == 1'b0)? DATA : 8'bZZZZZZZZ; //  buffer to RAM IO

  //module RAM_block (CS_b,WE_b,OE_b,Address,IO);
  RAM_block RAM256(1'b0, WE_b, 1'b0, CNTR, RAMbus);

  always @(state, K, Eq)
  begin
    WE_b = 1'b1; inc = 1'b0; Fail = 1'b0; Pass = 1'b0;
    case (state)
      0 : begin
        WE_b = 1'b0;
        nextstate = 1;
      end
      1 : begin
        inc = 1'b1;
        if (K == 1'b0)
          nextstate = 0;
        else
          nextstate = 2;
      end
      2 : begin
        inc = 1'b1;
        if (Eq == 1'b0)
          nextstate = 3;
        else if (K == 1'b0)
          nextstate = 2;
        else
          nextstate = 4;
      end
      3 : Fail = 1'b1;
      4 : Pass = 1'b1;
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    if (inc == 1'b1)
    begin
      if (CNTR == 8'hFF)
        CNTR <= 8'h00;
      else
        CNTR <= CNTR + 1'b1;
    end
  end

endmodule
```

8.21 (a)



(b)



(c)

```
module Chex_test2 (CLK, Pass, Fail);
  input CLK;
  output reg Pass, Fail;

  integer state, nextstate;
  wire oddbit, evenbit;
  wire K, Eq;
  reg WE_b, inc, RP, setRP;
  reg [7:0] CNTR;
  wire [7:0] DATA, RAMbus;

  initial
  begin
    WE_b = 1'b1;
    inc = 1'b0;
```

```
    RP = 1'b0;
    setRP = 1'b0;
    CNTR = 8'h00;
    state = 0;
    nextstate = 0;
  end

  assign K = (CNTR == 8'hFF) ? 1'b1 : 1'b0;
  assign Eq = (DATA == RAMbus)? 1'b1 : 1'b0; //  comparator
  assign oddbit = CNTR[0] ^ RP; //  odd bit generator
  assign evenbit = ~oddbit; //  even bit generator
  assign DATA = {oddbit, evenbit, oddbit, evenbit, oddbit, evenbit,
oddbit, evenbit};
  assign RAMbus = (WE_b == 1'b0)? DATA : 8'bZZZZZZZZ; //  buffer to RAM IO

  //module RAM6116 (CS_b,WE_b,OE_b,Address,IO);
  RAM6116 RAM(1'b0, WE_b, 1'b0, CNTR, RAMbus);

  always @(state, K, Eq, RP)
  begin
    WE_b = 1'b1; inc = 1'b0; Fail = 1'b0; Pass = 1'b0; setRP = 1'b0;
    case (state)
      0 : begin
        WE_b = 1'b0;
        nextstate = 1;
      end
      1 : begin
        inc = 1'b1;
        if (K == 1'b0)
          nextstate = 0;
        else
          nextstate = 2;
      end
      2 : begin
        inc = 1'b1;
        if (Eq == 1'b0)
          nextstate = 3;
        else if (K == 1'b0)
          nextstate = 2;
        else if (RP == 1'b0) begin
          nextstate = 0;
          setRP = 1'b1;
        end
        else
          nextstate = 4;
      end
      3 : Fail = 1'b1;
      4 : Pass = 1'b1;
    endcase
  end

  always @(posedge CLK)
  begin
    state <= nextstate;
    if (setRP == 1'b1)
    begin
      RP <= 1'b1;
    end
    if (inc == 1'b1)
    begin
      if (CNTR == 8'hFF)
        CNTR <= 8'h00;
      else
```

```
        CNTR <= CNTR + 1'b1;
      end
    end

  endmodule
```

8.22

```
  module TFF (CLK, T, Q, Qn);
    parameter Trise = 8;
    parameter Tfall = 10;
    parameter Tclkmin = 15;
    parameter Tsetup = 4;
    parameter Thold = 2;

    input CLK, T;
    output reg Q, Qn;

    reg Qint, Qintc; // internal signals

    initial
    begin
      Qint = 1'b0;
      Qintc = 1'b1;
    end

    always @(posedge CLK)
    begin
      if (T == 1'b1)
      begin
        Qint <= Qintc;
        Qintc <= Qint;
      end
    end

    always @(Qint)
    begin
      if (Qint == 1'b1)
        #(Trise) Q = Qint;
      else
        #(Tfall) Q = Qint;
    end

    always @(Qintc)
    begin
      if (Qintc == 1'b1)
        #(Trise) Qn = Qintc;
      else
        #(Tfall) Qn = Qintc;
    end

    specify
      $width (posedge CLK, Tclkmin);
      $setup (T, posedge CLK, Tsetup);
      $hold (posedge CLK, T, Thold);
    endspecify

  endmodule
```

8.23 (a)

```
module DFF(CLK, CLR, D, Q, Qn);
  parameter tplh = 10;
  parameter tphl = 10;
  parameter tsu = 5;
  parameter th = 3;
  parameter tcmin = 20;

  input CLK, CLR, D;
  output reg Q, Qn;

  reg Qint, Qintc; // internal signals

  initial
  begin
    Qint = 1'b0;
    Qintc = 1'b1;
  end

  always @(posedge CLK, negedge CLR)
  begin
    if (~CLR) // active low asynchronous clear
    begin
      Qint <= 1'b0;
      Qintc <= 1'b1;
    end
    else
    begin
      Qint <= D;
      Qintc <= ~D;
    end
  end

  always @(Qint)
  begin
    if (Qint == 1'b1)
      #(tplh) Q = Qint;
    else
      #(tphl) Q = Qint;
  end

  always @(Qintc)
  begin
    if (Qintc == 1'b1)
      #(tplh) Qn = Qintc;
    else
      #(tphl) Qn = Qintc;
  end

  specify
    $width (posedge CLK, tcmin);
    $setup (D, posedge CLK, tsu);
    $hold (posedge CLK, D, th);
  endspecify

endmodule
```

(b)

```
module DFF_Test();
  parameter N = 12;

  reg clk, reset, data;
  wire out, outc;

  reg [N-1:0] reset_vec, data_vec;
```

```
    integer delay_vec [N-1:0];
    reg [N-1:0] out_vec;
    reg [N-1:0] outc_vec;

    initial
    begin
      clk = 0;
      reset_vec = 12'b111110111110;
      data_vec  = 12'b010011010001;
      delay_vec[0]  = 25; //t=25
      delay_vec[1]  = 8;  //t=33
      delay_vec[2]  = 3;  //t=36
      delay_vec[3]  = 9;  //t=45
      delay_vec[4]  = 28; //t=73
      delay_vec[5]  = 22; //t=95
      delay_vec[6]  = 18; //t=113
      delay_vec[7]  = 12; //t=125
      delay_vec[8]  = 12; //t=137
      delay_vec[9]  = 8;  //t=145
      delay_vec[10] = 8;  //t=153
      delay_vec[11] = 20; //t=173
      out_vec  = 12'b100000110000;
      outc_vec = 12'b011111001111;
    end

    always
      #10 clk = ~clk; // min clock period 20 ns

    integer i;
    always
    begin
      for (i = 0; i < N; i = i + 1)
      begin
        reset = reset_vec[i];
        data = data_vec[i];

        #(delay_vec[i])

        if (out!=out_vec[i] || outc!=outc_vec[i])
        begin
          $display("Output Mismatch");
        end

      end
      $stop;
    end

    DFF dff1(clk, reset, data, out, outc);
  endmodule
```

8.24

```
module CMP(A, B, EQ, GT);
  parameter N = 8; // N is the length of the bit vectors

  input [N-1:0] A, B;
  output reg EQ, GT;

  reg LT;
  integer i;

  always @(A, B)
  begin
    EQ = 1; GT = 0; LT = 0;
```

```
    for (i=N-1; i>=0; i=i-1)
    begin
      if (A[i]>B[i] && LT == 1'b0 && GT == 1'b0)
      begin
        GT = 1'b1;
        EQ = 1'b0;
      end
      else if(A[i]<B[i] && LT == 1'b0 && GT == 1'b0)
      begin
        LT = 1'b1;
        EQ = 1'b0;
      end
    end
  end
endmodule
```

8.25

```
module P8_25 (abus, dbus, sel, wr);
  input [14:0] abus;
  inout [31:0] dbus;
  input sel;
  input wr;

  genvar i;
  generate
  for (i=0; i<4; i=i+1)
  begin: gen_loop
    SRAM MemX (.address(abus), .data(dbus[i*8+7:i*8]),
               .cs_b(sel), .we_b(wr), .oe_b(1'b 0));
  end
  endgenerate

endmodule
```

8.26

```
module Shift_Register(SI, Sh, CLK, SO);
  parameter N = 16; // N is the number of bits in the shift register

  input  SI, Sh, CLK;
  output SO;

  wire [N-1:1] Qint;

  // instantiate middle DFFs
  genvar i;
  generate
    for (i=1; i<N-1; i=i+1)
    begin
      // module DFF(clk, ce, d, q);
      DFF dffx(CLK, Sh, Qint[i+1], Qint[i]);
    end
  endgenerate

  // instantiate end DFFs
  DFF dffn_1(CLK, Sh, SI, Qint[N-1]);
  DFF dff0(CLK, Sh, Qint[1], SO);

endmodule
```

8.27

```
module and_N(A, B, C);
  parameter N = 4;
```

```
  input [N-1:0] A;
  input B;
  output [N-1:0] C;

  genvar i;
  generate
  for (i=0; i<N; i=i+1)
  begin: gen_loop
    and AND1 (C[i], A[i], B);
  end
  endgenerate

endmodule
```

8.28 **(a)**

```
NOR2 #(5, 4, 3) U2 (in3, in4, out2);
```

(b)

```
defparam U2.Trise = 4;
defparam U2.Tfall = 3;
defparam U2.load  = 2;
NOR2 U2 (in3, in4, out2);
```

(c) rise time: 3 ns
fall time: 2 ns
fan-out time: 3 ns

(d) If using the instantiation in part (a), rise delay = 14 ns and fall delay = 13 ns.
If using the instantiation in part (b), rise delay = 10 ns and fall delay = 9 ns.

8.29

```
module top(X,Y,P);
  input [3:0] X,Y;
  output [7:0] P;

  wire [3:0] and_gate_out [0:3];
  wire [3:0] adder_sum_out [0:3];
  wire [3:0] adder_carry_out [1:3];

  //16 and gates
  genvar i,j;
  generate
    for (i=0; i<=3; i=i+1)
    begin
      for (j=3; j>=0; j=j-1)
      begin
        and2 and2_x(X[j], Y[i], and_gate_out[i][j]);
      end
 end
  endgenerate

  assign adder_sum_out[0] = and_gate_out[0]; //row 0 has no adders

  //create half-adders at end of row
  generate
    for (i=1; i<=3; i=i+1)
    begin
        HA HA_x(adder_sum_out[i-1][1], and_gate_out[i][0],
adder_sum_out[i][0],
                adder_carry_out[i][0]);
    end
```

```
  endgenerate

  //create half-adder at far left of second row
  HA HA_1(and_gate_out[1][3], adder_carry_out[1][2], adder_sum_out[1][3],
adder_carry_out[1][3]);

  //create full-adders at far left of third and fourth row
  genvar r;
  generate
    for (r=2; r<=3; r=r+1)
    begin
      FA FA_x(and_gate_out[r][3], adder_carry_out[r-1][3],
adder_carry_out[r][2],
              adder_sum_out[r][3], adder_carry_out[r][3]);
    end
  endgenerate

  //generate 2 full-adders in middle of each row
  genvar k,l;
  generate
    for (k=1; k<=3; k=k+1)
    begin
      for (l=2; l>=1; l=l-1)
      begin
        FA FA_xx(and_gate_out[k][l], adder_sum_out[k-1][l+1],
adder_carry_out[k][l-1],
                adder_sum_out[k][l], adder_carry_out[k][l]);
      end
    end
  endgenerate

  assign P[7] = adder_carry_out[3][3];
  assign P[6:3] = adder_sum_out[3][3:0];
  assign P[2] = adder_sum_out[2][0];
  assign P[1] = adder_sum_out[1][0];
  assign P[0] = and_gate_out[0][0];

endmodule

module and2(A,B,C);
  input A, B;
  output C;
  assign C = A&B;
endmodule

module HA(A,B,S,Co);
  input A, B;
  output Co, S;
  assign S = A^B;
  assign Co = A&B;
endmodule

module FA(A,B,Cin,S,Co);
  input A, B, Cin;
  output Co, S;
  assign S = A^B^Cin;
  assign Co = (A&B)|(A&Cin)|(B&Cin);
endmodule
```

8.30

```
file = $fopen("FILE2", "r");
for (i=0; i<5; i=i+1)
begin
```

```
  count = $fscanf(file, "%d", B[i]);
end
$fclose(file);
```

8.31
```
task fileAssign;
  input [31:0] file;
  // signal is a global integer.
  // Otherwise, it is updated only once when the task completes.
  integer delay, value, count;

  begin
    while (!$feof(file))
    begin
      count = $fscanf(file, "%d", delay);
      count = $fscanf(file, "%d", value);
      #(delay) signal = value;
    end
  end
endtask
```

8.32
```
module top(sig);
  `define N 3 // N is the size of the bit vector
  input [`N-1:0] sig;

  always@(sig)
  begin
    logger(sig);
  end

  task logger;
    input [`N-1:0] sig;
    integer file;
    begin
      file = $fopen("filename","a"); //open for appending at end of file
      $fwrite(file, "%0dns : %d \n", $time, sig);
      $fclose(file);
    end
    endtask

endmodule
```

8.33 regA: 503
regB: 256

Because RegB is not declared as a signed reg, $signed() will have no effect.

8.34 **(a)** Add the following lines inside the module:
```
`define Sum(x,y) x+y
parameter A = 9;
parameter B = 11;
```

(b) The sum of 9 and 11 is 20

8.35 **$stop** will temporary suspend simulation, but **$finish** will exit the simulator.

8.36

Time	Enc	Ena	Enb	Bus
0	Z	Z	Z	Z
2	1	0	0	8'd15
4	0	1	0	8'd5
6	0	0	1	8'd10
8	1	1	1	X
10	1	1	1	X

Chapter 9: Design of a Risc Microprocessor

9.1 ISA means Instruction Set Architecture. Yes, the Pentium 3 and 4 both use the x86 ISA.

9.2 No. *X* could have many addressing modes for each instruction, non uniform instruction widths, many instruction formats, a non load/store architecture, or many implied operands with each instruction.

9.3 Uniform instruction length, few instruction formats, few addressing modes, large number of registers, load/store architecture, no implied operands or side-effects.

9.4 A RISC ISA uses a small and simple set of instructions, unlike the CISC ISA, which uses a variety of complex instructions and versatile addressing modes. The benefits of a RISC ISA are simpler instruction decoding, larger number of high-speed registers, and no implied operands or side effects. The benefits of a CISC ISA are more compact instruction sizes, often resulting in a smaller instruction memory size. Because RISC instructions are simpler and can be pipelined easily, each RISC instruction require fewer processor clock cycles to go through an instruction cycle. Thus, RISC machines have higher instructions per cycle (IPC).

9.5 `addiu` is an unsigned add-immediate. Unlike the signed add-immediate instruction `addi`, `addiu` never causes an overflow exception.

9.6 Branch instructions will either cause the processor to start execution of an instruction sequence starting at a specified memory location or continue execution of the current instruction sequence, depending on whether the branch condition is taken or not taken. However, jump instructions unconditionally cause the processor to start execution of an instruction sequence starting at a specified memory location.

9.7 **(i)** 00E83020
(ii) 8CC50004
(iii) 2443F830
(iv) 00071B00
(v) 10C5FFF0
(vi) 080003E8

9.8 **(i)** 20850FA0
(ii) AC650014
(iii) 00A32021
(iv) 14430020
(v) 00A00008
(vi) 0C0007D0

9.9 **(i)** `andi $19, $25, 13056`
(ii) `lw $13, -29299($12)`
(iii) `bne $27, $23, -256`
(iv) illegal opcode
(v) illegal opcode

9.10 **(i)** `addi $0, $1, 8224`
(ii) `mult $7, $7`
(iii) `beq $30, $19, 200`
(iv) `slt $5, $12, $18`
(v) `jal -9956700`

9.11

```
      andi $3, $3, 0     ;initialize loop counter $3 to 0. See Note.
      addi $2, $3, 400   ;loop bound. Since $3 has 0, it is used.
loop: lw   $15, 4000($3) ;load x(i) to R15
      lw   $14, 8000($3) ;load y(i) to R14
      mult $14, $15      ;hi, lo = $14 * $15
      mflo $14           ;$14 = low bits
      sw   $14, 4000($3) ;save new x(i)
      addi $3, $3, 4     ;update address register, address= address + 4
      bne  $3, $2, loop   ;check if loop counter=loop bound
```

Note: In an actual MIPS, register $0 is always 0. So clearing register $3 can be done by add $3, $0, $0.

9.12

```
      andi $3, $3, 0     ;clear loop counter $3. See Note.
      addi $2, $3, 400   ;loop bound
      lw   $14, 4000($3) ;load x(i-1) to R14
loop: lw   $15, 4004($3) ;load x(i) to R15
      add  $14, $14, $15 ;$14 = x(i) + x(i-1)
      sw   $14, 4004($3) ;save new x(i)
      addi $3, $3, 4     ;update address register, address= address + 4
      bne  $3, $2, loop    ;check if loop counter=loop bound
```

Note: In an actual MIPS, register $0 is always 0. So clearing register $3 can be done by add $3, $0, $0.

9.13

```
      andi $3, $3, 0      ;initialize loop counter $3 to 0. See Note.
      addi $2, $3, 400    ;loop bound
      lw   $16, 12000($3) ;$16 = a
loop: lw   $15, 4000($3)  ;load x(i) to R15
      lw   $14, 8000($3)  ;load y(i) to R14
      mult $16, $15       ;hi, lo = a * x(i)
      mflo $15            ;$15 = low bits of a * x(i)
      add  $14, $14, $15  ;$14 = a * x(i) + y(i)
      sw   $14, 8000($3)  ;save new y(i)
      addi $3, $3, 4      ;update address register, address= address + 4
      bne  $3, $2, loop     ;check if loop counter=loop bound
```

Note: In actual an MIPS, register $0 is always 0. So clearing register $3 can be done by add $3, $0, $0.

9.14

```
         addi $1, $0, 0 ; i = 0
         addi $2, $0, 40 ; outer for loop bound
$for:    lw $3, 2000($1) ; load x(i) into R3
         lw $4, 4000($1) ; load y(i) into R4
         mult $3, $4
         mfhi $3 ; higher part of x(i) * y(i)
         sw $3, 2000($1) ; save new x(i)
         addi $5, $0, 0 ; z = 0
$while:  lw $6, 4004($1) ; load y(i+1) into R6
```

```
add $4, $6, $4 ; y(i) + y(i+1)
sw $4, 4000($1) ; save new y(i)
addi $5, $5, 1 ; z = z + 1
slti $7, $5, 20 ; z <? 20
bne $7, $0, $while
addi $1, $1, 4 ; i++
bne $1, $2, $for
```

Note: In actual an MIPS, register $0 is always 0. So comparing with 0 can be done by bne $7, $0, $label.

9.15 For a Spartan 3 FPGA prototyping board: 605 slices, 189 flip-flops, 1104 LUT4s, and no block memories (the register file was implemented as distributed RAM).

9.16 **(a)** This solution is for a prototyping board based on the Spartan 3 FPGA. The board has a 50 MHz clock. This solution uses 1 switch on the prototyping board as the HALT switch and 8 LEDs on the prototyping board to display the lower 8 bits of Register 1.

Modifications to Figure 9-6 (Register File)

- In the port interface list, add `Reg1`:
```
module REG (CLK, RegW, DR, SR1, SR2, Reg_In, ReadReg1, ReadReg2, Reg1);
```
- In the list of interface ports, add:
```
output reg [7:0] Reg1;
```
- In the `always` block, after the assignments to `ReadReg1` and `ReadReg2`, add the statement:
```
Reg1 <= REG[1][7:0];               //low 8 bits of $1
```

Modifications to Figure 9-8 (Processor)

- In the port interface list, add `Reg1`:
```
module MIPS (CLK, RST, CS, WE, ADDR, Mem_Bus, Reg1);
```
- In the list of interface ports, add:
```
output [7:0] Reg1;
```
- Add `Reg1` to the `REG` module instantiation statement, for example:
```
REG Register(CLK, regw, dr, `sr1, `sr2, reg_in, readreg1, readreg2, Reg1);
```

Modified Figure 9-9 (Complete MIPS)

```
module Complete_MIPS (Clk50Mhz, RST, Halt_Switch, A_Out, D_Out, LEDs);

  input Clk50Mhz;
  input RST;
  input Halt_Switch;
  output [31:0] A_Out;
  output [31:0] D_Out;
  output [7:0] LEDs;

  wire CS, WE;
  wire [31:0] ADDR, Mem_Bus;
  wire CLK;
  reg Clk8Hz;
  reg [31:0] count;

  initial
  begin
    Clk8Hz = 1'b0;
    count = 0;
```

```
  end

  assign CLK = (Halt_Switch == 1'b0) ? Clk8Hz : 1'b0;
  assign A_Out = ADDR;
  assign D_Out = Mem_Bus;

  // 50MHz to 8Hz clock divider
  always @(posedge Clk50Mhz)
  begin
    if (count == 6250000)
    begin
      count <= 1;
      if (Clk8Hz == 1'b1)
      begin
         Clk8Hz <= 1'b0;
      end
      else
      begin
         Clk8Hz <= 1'b1;
      end
    end
    else
    begin
      count <= count + 1;
    end
  end

  MIPS CPU(CLK, RST, CS, WE, ADDR, Mem_Bus, LEDs);
  Memory MEM(CS, WE, CLK, ADDR, Mem_Bus);

endmodule
```

(b) This solution assumes a 100 Hz clock, and all offsets and jumps are based on the word-addressed memory used by Figure 9-8. This solution uses patterns loaded from memory. An alternate implementation could use 1 pattern and shift instructions. However, we have not implemented a rotate instruction; hence, restarting after the 8th pattern must be implemented with an extra jump instruction (j 2 in our case)

Line	Machine Code	MIPS Assembly Code
0	x"30000000"	andi $0, $0, 0 ;clear $0.See Note 1.
1	x"20040008"	addi $4, $0, 8 ;$4 = number of patterns = 8
2	x"30420000"	andi $2, $2, 0 ;$2 = pattern offset counter = 0
3	x"8C41000A"	lw $1, 10($2) ;load new light pattern
4	x"2003000C"	addi $3, $0, 12 ;load delay counter with 12 (see Note 2)
5	x"2063FFFF"	addi $3, $3, -1 ;decrement delay counter
6	x"1460FFFE"	bne $3, $0, -2 ;if counter is not 0, repeat loop (branch to 5)
7	x"20420001"	addi $2, $2, 1 ;increment offset counter
8	x"1444FFFA"	bne $2, $4, -6 ;when counter /= 8, load next pattern (go to 3)
9	x"08000002"	j 2 ;when counter = 8, restart sequence
10	x"00000080"	(Data Memory) ;pattern 0 – "10000000"
11	x"00000040"	(Data Memory) ;pattern 1 – "01000000"
12	x"00000020"	(Data Memory) ;pattern 2 – "00100000"
13	x"00000010"	(Data Memory) ;pattern 3 – "00010000"
14	x"00000008"	(Data Memory) ;pattern 4 – "00001000"
15	x"00000004"	(Data Memory) ;pattern 5 – "00000100"
16	x"00000002"	(Data Memory) ;pattern 6 – "00000010"
17	x"00000001"	(Data Memory) ;pattern 7 – "00000001" See Note 3

Note 1: This will not be required in an actual MIPS. $0 is always 0 in an actual MIPS; however, we have not implemented such a $0 in Figure 9-8.

Note 2: Delay calculation: Each light pattern should be active one second, or 100 cycles with a 100 Hz clock. Each light pattern begins when an `lw` instruction (line 3) is executed. The `lw` instruction on line 3 and `addi` instruction on line 4 require 9 cycles to execute. Each repetition of the *delay* counter loop (lines 5 and 6) requires 7 cycles. Finally, when the light pattern finishes, the `addi` and `bne` instructions on lines 7 and 8 are executed, taking 7 cycles. Thus, the required delay can be calculated as 100 = 9 + 7**delay* + 7, so *delay* = 12.

Note 3: When pattern 7 is displayed, it will execute an extra jump (line 9) and `andi` (line 2) to restart the light sequence. Restarting causes the last pattern to last 5 cycles, or 0.05 seconds, longer then the other patterns. This delay is not compensated for in the solution to simplify the code.

(c) This solution assumes a 100 Hz clock, and all offsets and jumps are based on the word-addressed memory used by Figure 9-8.

```
Line  Machine Code  | MIPS Assembly Code
  0   x"30000000"   | andi $0, $0, 0  ;clear $0
  1   x"20040006"   | addi $4, $0, 6  ;$4 = number of patterns = 6
  2   x"30420000"   | andi $2, $2, 0  ;$2 = pattern offset counter = 0
  3   x"8C41000A"   | lw   $1, 10($2) ;load new light pattern
  4   x"8C430010"   | lw   $3, 16($2) ;load delay counter
  5   x"2063FFFF"   | addi $3, $3, -1 ;decrement delay counter
  6   x"1460FFFE"   | bne  $3, $0, -2 ;if counter is not 0, repeat loop (branch to 5)
  7   x"20420001"   | addi $2, $2, 1  ;increment offset counter
  8   x"1444FFFA"   | bne  $2, $4, -6 ;when counter /= 6, load next pattern (go to 3)
  9   x"08000002"   | j    2          ;when counter = 6, restart sequence
 10   x"0000000C"   | (Data Memory)   ;pattern 0 - "001100". Order is RYG RYG
 11   x"00000014"   | (Data Memory)   ;pattern 1 - "010100"
 12   x"00000024"   | (Data Memory)   ;pattern 2 - "100100"
 13   x"00000021"   | (Data Memory)   ;pattern 3 - "100001"
 14   x"00000022"   | (Data Memory)   ;pattern 4 - "100010"
 15   x"00000024"   | (Data Memory)   ;pattern 5 - "100100"
 16   x"00000045"   | (Data Memory)   ;delay 0 = 69 = 5 seconds
 17   x"0000001A"   | (Data Memory)   ;delay 1 = 26 = Approx 2 seconds (1.99 seconds)
 18   x"0000000C"   | (Data Memory)   ;delay 2 = 12 = Approx 1 second (1.01 seconds)
 19   x"00000045"   | (Data Memory)   ;delay 3 = 69 = 5 seconds
 20   x"0000001A"   | (Data Memory)   ;delay 4 = 26 = Approx 2 seconds (1.99 seconds)
 21   x"0000000C"   | (Data Memory)   ;delay 5 = 12 = Approx 1 second (1.01 seconds)
```

Delay calculation: Same as part (b), except each new pattern begins with two loads instead of one load and one add, so each pattern will run for 10 + 7**delay* + 7 cycles. Each pattern should run 100 cycles per second the pattern is active. Thus, *delay* for each pattern can be calculated as (100 * *seconds* – 17) / 7, rounded to the nearest integer.

9.17

```
Line  Machine Code  | MIPS Assembly Code
  0   x"30000000"   | andi $0, $0, 0  ;clear $0. See Note 1.
  1   x"20010001"   | addi $1, $0, 1  ;$1 = 1
  2   x"8C020007"   | lw   $2, 7($0)  ;$2 = F0002F2F. See Note 2.
  3   x"AC400000"   | sw   $0, 0($2)  ;F0002F2F = 0
  4   x"08000005"   | j    5          ;no-op (to making timing same, creating sq wave)
  5   x"AC410000"   | sw   $1, 0($2)  ;F0002F2F = 1
  6   x"08000003"   | j    3          ;restart
  7   x"F0002F2F"   | (Data Memory)   ;memory location
```

Note 1: This will not be required in an actual MIPS. $0 is always 0 in an actual MIPS; however, we have not implemented such a register 0 in Figure 9-8.

Note 2: This lw can be avoided if lui instruction was implemented. A lui + add instruction will do the job.

9.18 **(a)** Modifications to Figure 9-8:

1. In the port interface list, add a new output `V`, for example:
```
module MIPS (CLK, RST, CS, WE, ADDR, Mem_Bus, V);
```
2. In the list of interface ports, add the output:
```
output reg V;
```
3. In the list of interface ports, modify the declarations for `alu_in_A` and `alu_in_B` as follows, and add the new signal below:
```
wire [32:0] alu_in_A, alu_in_B;
reg [32:0] alu_calc;
```
4. Modify the assignments to `alu_in_A` and `alu_in_B` as follows:
```
assign alu_in_A = {readreg1[31], readreg1};
assign alu_in_B = (reg_or_imm_save == 1)? {imm_ext[31], imm_ext} :
       {readreg2[31], readreg2}; //ALU MUX (MUX2)
```
5. Add the following line inside the initial block:
```
alu_calc = 0;
```
6. Replace the code for state 2 (i.e. `2:`) with the following code:
```
nstate = 3'd3;
if (opsave == and1) alu_calc = alu_in_A & alu_in_B;
else if (opsave == or1) alu_calc = alu_in_A | alu_in_B;
else if (opsave == add) alu_calc = alu_in_A + alu_in_B;
else if (opsave == sub) alu_calc = alu_in_A - alu_in_B;
else if (opsave == srl) alu_calc = alu_in_B >> `numshift;
else if (opsave == sll) alu_calc = alu_in_B << `numshift;
else if (opsave == slt) alu_calc = (alu_in_A < alu_in_B)? 33'd1
: 33'd0;
else if (opsave == xor1) alu_calc = alu_in_A ^ alu_in_B;
  if (((alu_in_A == alu_in_B)&&(`opcode == beq)) ||
     ((alu_in_A != alu_in_B)&&(`opcode == bne))) begin
     npc = pc + imm_ext;
     nstate = 3'd0;
  end
  else if ((`opcode == bne)||(`opcode == beq)) nstate = 3'd0;
  else if (opsave == jr) begin
  npc = alu_in_A;
  nstate = 3'd0;
end
alu_result = alu_calc;
```
7. In the clock always block, right below the begin for the always block, add:
```
V <= 1'd0;
```
8. In the clock always block, modify the state 2 if conditional:
```
else if (state == 3'd2) begin
alu_result_save <= alu_result;
   if(`opcode == addi || (`opcode == 0 && `f_code == 32))
      V <= alu_calc[32] ^ alu_calc[31];
end
```

(b)
```
module MIPS_Testbench ();
  reg CLK;
  wire CS, WE, V;

  parameter N = 12;
  parameter W = 46;

  reg [31:0] Instr_List [W-1:0];
  reg [31:0] exp_output [N-1:0];
  wire [N-1:0] exp_V = 12'b010110100000;
  reg RST_VCNT;
  reg [31:0] Mem_Bus;
```

```
  wire [31:0] Address;
  reg [31:0] AddressTB;
  wire [31:0] Address_Mux;
  reg RST;
  reg init;
  wire WE_Mux;
  wire CS_Mux;
  reg WE_TB;
  reg CS_TB;
  reg VCNT;

  wire [31:0] Mem_Bus_wire = Mem_Bus;

  integer i;

  initial
  begin
    Instr_List[0]  = 32'h30000000; // 0.  andi $0, $0, 0  ;$0=0
    Instr_List[1]  = 32'h8C020029; // 1.  lw   $2, 41($0)
    Instr_List[2]  = 32'h20410001; // 2.  addi $1, $2, 1  ;$1=32'h7FFFFFFE" + 1
(no overflow)
    Instr_List[3]  = 32'hAC010040; // 3.  sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[4]  = 32'h8C02002C; // 4.  lw   $2, 44($0)
    Instr_List[5]  = 32'h8C030028; // 5.  lw   $3, 40($0)
    Instr_List[6]  = 32'h00430822; // 6.  sub  $1, $2, $3 ;$1="x80000001"-
32'h00000001" (no V)
    Instr_List[7]  = 32'hAC010040; // 7.  sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[8]  = 32'h8C020028; // 8.  lw   $2, 40($0)
    Instr_List[9]  = 32'h20410001; // 9.  addi $1, $2, 1  ;$1=32'h00000001" + 1
(no overflow)
    Instr_List[10] = 32'hAC010040; // 10. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[11] = 32'h8C02002D; // 11. lw   $2, 45($0)
    Instr_List[12] = 32'h8C03002D; // 12. lw   $3, 45($0)
    Instr_List[13] = 32'h00430820; // 13. add  $1, $2, $3
;$3=32'hFFFFFFFF"+32'hFFFFFFFF"(no V)
    Instr_List[14] = 32'hAC010040; // 14. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[15] = 32'h00430822; // 15. sub  $1, $2, $3 ;$1=32'hFFFFFFFF"-
32'hFFFFFFFF"(no V)
    Instr_List[16] = 32'hAC010040; // 16. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[17] = 32'h8C02002A; // 17. lw   $2, 42($0)
    Instr_List[18] = 32'h20410001; // 18. addi $1, $2, 1  ;$1=32'h7FFFFFFF" + 1
(overflow)
    Instr_List[19] = 32'hAC010040; // 19. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[20] = 32'h8C02002B; // 20. lw   $2, 43($0)
    Instr_List[21] = 32'h8C030028; // 21. lw   $3, 40($0)
    Instr_List[22] = 32'h00430822; // 22. sub  $1, $2, $3 ;$1=$2 - $3 (overflow
- 9.19 only)
    Instr_List[23] = 32'hAC010040; // 23. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[24] = 32'h2041FFFF; // 24. addi $1, $2, -1 ;$1=32'h80000000" - 1
(overflow)
    Instr_List[25] = 32'hAC010040; // 25. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[26] = 32'h8C02002A; // 26. lw   $2, 42($0)
    Instr_List[27] = 32'h8C03002A; // 27. lw   $3, 42($0)
    Instr_List[28] = 32'h00430820; // 28. add  $1, $2, $3
;$1=32'h7FFFFFFF"+32'h7FFFFFFF" (V)
    Instr_List[29] = 32'hAC010040; // 29. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[30] = 32'h00430822; // 30. sub  $1, $2, $3 ;$1=32'h7FFFFFFF"-
32'h7FFFFFFF"(no V)
    Instr_List[31] = 32'hAC010040; // 31. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[32] = 32'h8C02002B; // 32. lw   $2, 43($0)
    Instr_List[33] = 32'h8C03002B; // 33. lw   $3, 43($0)
```

```
    Instr_List[34] = 32'h00430820; // 34. add  $1, $2, $3 ;$1=32'h80000000" +
32'h80000000" (V)
    Instr_List[35] = 32'hAC010040; // 35. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[36] = 32'h8C02002B; // 36. lw   $2, 43($0)
    Instr_List[37] = 32'h8C03002A; // 37. lw   $3, 42($0)
    Instr_List[38] = 32'h00430822; // 38. sub  $1, $2, $3 ;$1=$2 - $3 (overflow
- 9.19 only)
    Instr_List[39] = 32'hAC010040; // 39. sw   $1, 64($0) ;store to trigger test
bench check
    Instr_List[40] = 32'h00000001; // 40. Data memory 28
    Instr_List[41] = 32'h7FFFFFFE; // 41. Data memory 29
    Instr_List[42] = 32'h7FFFFFFF; // 42. Data memory 2A
    Instr_List[43] = 32'h80000000; // 43. Data memory 2B
    Instr_List[44] = 32'h80000001; // 44. Data memory 2C
    Instr_List[45] = 32'hFFFFFFFF; // 45. Data memory 2D

    // This test bench performs a series of test adds and subtracts. Each test
    // is followed by a store. When the test bench observes a store instruction,
    // it checks the expected output, and the expected number of overflows, then
    // resets the 'overflow counter'

    exp_output[0]  = 32'h7FFFFFFF;
    exp_output[1]  = 32'h80000000;
    exp_output[2]  = 32'h00000002;
    exp_output[3]  = 32'hFFFFFFFE;
    exp_output[4]  = 32'h00000000;
    exp_output[5]  = 32'h80000000;
    exp_output[6]  = 32'h7FFFFFFF;
    exp_output[7]  = 32'h7FFFFFFF;
    exp_output[8]  = 32'hFFFFFFFE;
    exp_output[9]  = 32'h00000000;
    exp_output[10] = 32'h00000000;
    exp_output[11] = 32'h00000001;

    CLK = 1'b0;
    RST_VCNT = 1'b0;
  end

  MIPS CPU (CLK, RST, CS, WE, Address, Mem_Bus_wire, V);
  Memory MEM (CS_Mux, WE_Mux, CLK, Address_Mux, Mem_Bus_wire);

  always
    #10 CLK = ~CLK;

  assign Address_Mux = (init == 1'b1) ? AddressTB : Address;
  assign WE_Mux = (init == 1'b1) ? WE_TB : WE;
  assign CS_Mux = (init == 1'b1) ? CS_TB : CS;

  always @(posedge V, posedge RST_VCNT)
  begin
    if (RST_VCNT == 1'b1)
    begin
      VCNT = 1'b0;
    end
    else if (V == 1'b1)
    begin
      VCNT = VCNT + 1'b1;
    end
  end

  always
  begin
    RST <= 1'b1;
    @ (posedge CLK);

    // Initialize the instructions from the test bench
    init <= 1'b1;
    CS_TB <= 1'b1; WE_TB <= 1'b1;

    for (i = 1; i <= W; i = i + 1)
```

```
        begin
            @ (posedge CLK);
            AddressTB <= i - 1;
            Mem_Bus <= Instr_List[i-1];
        end
        @ (posedge CLK);
        Mem_Bus <= 32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
        CS_TB <= 1'b0;
        WE_TB <= 1'b0;
        init <= 1'b0;
        @ (posedge CLK);
        RST <= 1'b0;

        for (i = 1; i <= N; i = i + 1)
        begin
          @ (posedge WE); //  When a store word is executed
          if(!(VCNT == exp_V[i-1]))
          begin
              $display("Overflow mismatch");
          end
          RST_VCNT <= 1'b1;
          @ (negedge CLK);
          if (!(Mem_Bus_wire == exp_output[i-1]))
          begin
            $display("Output mismatch:");
          end
          RST_VCNT <= 1'b0;
        end
        $display("Testing Finished");
        $stop;
    end

endmodule
```

9.19 **(a)** The affected instructions are `add`, `addi`, and `sub`. Modify the condition check for assigning `V` in the updated Figure 9-8 of Solution 9.18(a) as follows:

```
  if (`opcode == addi || (`opcode == 0 && (`f_code == 32 || `f_code ==
34)))
    V <= alu_calc[32] ^ alu_calc[31];
```

(b) Modify the bitvector of expected *V* outputs in the test bench of Solution 9.18 (b) as follows:

```
  wire [N-1:0] exp_V = 12'b110111100000;
```

9.20 **(a)** Make the following changes to the updated Figure 9-8 of Solution 9.18(a) as follows:

1. In the list of internal signals, add the signal:
   ```
     reg savepc;
   ```
2. In the list of non-special instructions, add the constant:
   ```
     parameter jal = 6'b000011;
   ```
3. Replace the logic for the *DR* mux with:
   ```
   assign dr = (savepc == 1)? 5'b11111 :
   ((format == R)? instr[15:11] : instr[20:16]); //Destination
   Register MUX (MUX1)
   ```
4. Replace the logic for the register input data mux with:
   ```
   assign reg_in =
     (savepc == 1)? pc : ((alu_or_mem_save)? Mem_Bus :
   alu_result_save); //Data MUX
   ```
5. Replace the logic for the *format* field with:
   ```
   assign format =
     (`opcode == 6'd0)? R : ((`opcode == 6'd2 || `opcode == 6'd3)? J
   : I);
   ```
6. Add the following assignment inside the initial block:
   ```
     savepc = 0;
   ```
7. In state one (`1:`), modify the contents of the `if (format == J)` conditional as follows:

```
if (format == J) begin //jump, and finish
  if (`opcode == j) begin
    npc = {6'b000000, instr[25:0]}; nstate = 3'd0;
  end
  else begin //jal
    nstate = 3'd4;
  end
end
```

8. Add a conditional statement at the in state 4 (`4:`), after the if conditional

```
else if(`opcode == jal) begin
  regw = 1; savepc = 1;
  npc = {6'b000000, instr[25:0]};
end
```

(b) Make the following changes to the updated Figure 9-10 of Solution 9.18(a) as follows:

1. Change the values of the constants *N* and *W*:

```
parameter N = 2;
parameter W = 6;
```

2. In the initial block, change the initialization of the the *Instr_List* array to the following:

```
    Instr_List[0]  = 32'h30000000; // 0. andi $0, $0, 0  ;$0 = 0
    Instr_List[1]  = 32'h30210000; // 1. andi $1, $1, 0  ;$1 = 0
    Instr_List[2]  = 32'h0C000004; // 2. jal 4
    Instr_List[3]  = 32'h20010006; // 3. addi $1, $0, 1  ; should be
skipped
    Instr_List[4]  = 32'hAC010040; // 4. sw $1, 64($0)   ; Mem(64) =
$1
    Instr_List[5]  = 32'hAC1F0041; // 5. sw $31, 65($0)  ; Mem(65) =
$31
```

3. In the initial block, change the initialization of the *exp_output* array to the following:

```
exp_output[0]  = 32'h00000000;
exp_output[1]  = 32'h00000003;
```

4. In the main always block, remove the `if (!(VCNT == exp_V[i-1]))` conditional.

9.21 (a) This solution uses R-format encoding for the instruction, with an *`f_code* of 63. Any R-format encoding not already used in the MIPS subset of Figure 9-8 is acceptable.

Make the following changes to the updated Figure 9-8 of Solution 9.18(a) as follows::

1. In the list of special instructions, add the constant:

```
parameter mult = 6'b111111;
```

2. Add an additional *opsave* check after the first `if` conditional in state two:

```
  else if (opsave == mult) alu_calc = alu_in_A[15:0] *
alu_in_B[15:0];
```

(b) Make the following changes to the updated Figure 9-10 of Solution 9.18(a) as follows:

1. Change the values of the constants *N* and *W*:

```
parameter N = 4;
parameter W = 18;
```

2. In the initial block, change the initialization of the the *Instr_List* array to the following:

```
Instr_List[0]  = 32'h30000000;  // andi $0, $0, 0
Instr_List[1]  = 32'h20010000;  // addi $1, $0, 0
Instr_List[2]  = 32'h20020000;  // addi $2, $0, 0
Instr_List[3]  = 32'h0022183F;  // mul $3, $1, $2 ;$3 = 0 * 0 = 0
Instr_List[4]  = 32'h20210001;  // addi $1, $1, 1
Instr_List[5]  = 32'h20420001;  // addi $2, $2, 1
Instr_List[6]  = 32'h0022203F;  // mul $4, $1, $2 ;$4 = 1 * 1 = 1
Instr_List[7]  = 32'h20217FFF;  // addi $1, $1, x7FFF
Instr_List[8]  = 32'h20217FFF;  // addi $1, $1, x7FFF
Instr_List[9]  = 32'h20427FFF;  // addi $2, $2, x7FFF
Instr_List[10] = 32'h20427FFF;  // addi $2, $2, x7FFF
```

```
        Instr_List[11] = 32'h0022283F;  // mul $5, $1, $2 ;$5 = xFFFF *
xFFFF = xFFFE0001
        Instr_List[12] = 32'h20210001;  // addi $1, $1, 1
        Instr_List[13] = 32'h0022303F;  // mul $6, $1, $2 ;$6 = x10000 *
xFFFF = 0
        Instr_List[14] = 32'hAC030040;  // sw $3, 64($0)  ;Mem(64) = $3
        Instr_List[15] = 32'hAC040041;  // sw $4, 65($0)  ;Mem(65) = $4
        Instr_List[16] = 32'hAC050042;  // sw $5, 66($0)  ;Mem(66) = $5
        Instr_List[17] = 32'hAC060043;  // sw $6, 67($0)  ;Mem(67) = $6
```

3. In the initial block, change the initialization of the *exp_output* array to the following:

```
exp_output[0]  = 32'h00000000;
exp_output[1]  = 32'h00000001;
exp_output[2]  = 32'hFFFE0001;
exp_output[3]  = 32'h00000000;
```

4. In the main always block, remove the `if (!(VCNT == exp_V[i-1]))` conditional.

9.22 **(a)** This solution uses R-format encoding for the instruction, with an `` ` ``*f_code* of 33. Any R-format encoding not

already used in the MIPS subset of Figure 9-8 is acceptable.

Make the following changes to the updated Figure 9-8 of Solution 9.18(a) as follows::

1. In the list of special instructions, add the constant:

```
parameter addb = 6'b100001;
```

2. Add an additional *opsave* check after the `if (opsave == add)` conditional in state two:

```
else if (opsave == addb) alu_calc = {
  alu_in_A[31:24] + alu_in_B[31:24],
  alu_in_A[23:16] + alu_in_B[23:16],
  alu_in_A[15:8] + alu_in_B[15:8],
  alu_in_A[7:0] + alu_in_B[7:0]};
```

(b) Make the following changes to the updated Figure 9-10 of Solution 9.18(a) as follows:

1. Change the values of the constants *N* and *W*:

```
parameter N = 2;
parameter W = 11;
```

2. In the initial block, change the initialization of the the *Instr_List* array to the following:

```
Instr_List[0]  = 32'h30000000; // 0. andi $0, $0, 0  ;$0 = 0
Instr_List[1]  = 32'h8C010008; // 1. lw $1, 8($0)
Instr_List[2]  = 32'h8C020009; // 2. lw $2, 9($0)
Instr_List[3]  = 32'h8C03000A; // 3. lw $3, 10($0)
Instr_List[4]  = 32'h00220821; // 4. addb $1, $1, $2
Instr_List[5]  = 32'h00431021; // 5. addb $2, $2, $3
Instr_List[6]  = 32'hAC010040; // 6. sw $1, 64($0)  ; Mem(64) =
$1
Instr_List[7]  = 32'hAC020041; // 7. sw $2, 65($0)  ; Mem(65) =
$2
Instr_List[8]  = 32'h01244567; // 8. Data memory 8
Instr_List[9]  = 32'hFEDCBA99; // 9. Data memory 9
Instr_List[10] = 32'hFFFFFFFE; // 10. Data memory A
```

3. In the initial block, change the initialization of the *exp_output* array to the following:

```
exp_output[0]  = 32'hFF00FF00;
exp_output[1]  = 32'hFDDBB997;
```

4. In the main always block, remove the `if (!(VCNT == exp_V[i-1]))` conditional.

9.23 **(a)** This solution uses R-format encoding for the instruction, with an `` ` ``*f_code* of 39. Any R-format encoding not

already used in the MIPS subset of Figure 9-8 is acceptable.

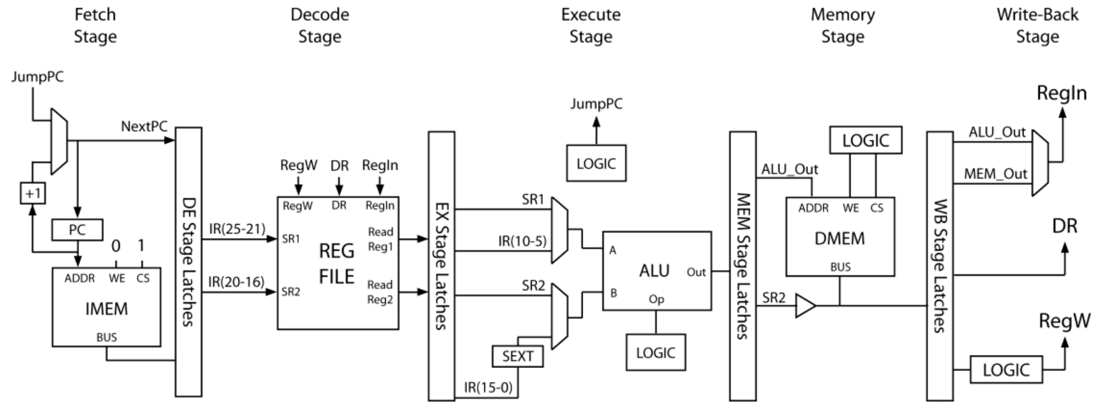
Make the following changes to the updated Figure 9-8 of Solution 9.18(a) as follows:

1. In the list of special instructions, add the constant:
```
parameter rbit = 6'b100111;
```
In the list of internal signals, add the new signal:
```
wire [4:0] src1;
```
2. Add the following assignment:
```
assign src1 = (format == R && `f_code == rbit)? dr : `sr1; //SR1
MUX
```
3. In the register instantiation, replace `` `sr1 `` with `src1`:
```
REG Register(CLK, regw, dr, src1, `sr2, reg_in, readreg1,
readreg2);
```
4. Right before the initial block, add:
```
integer i;
```
5. Add an additional *opsave* check after the `if (opsave == sub)` conditional in state two:
```
else if (opsave == rbit) begin
  for (i = 0; i < 32; i = i + 1) begin
    alu_calc[i] = alu_in_A[31-i];
  end
end
```

(b) Make the following changes to the updated Figure 9-10 of Solution 9.18(a) as follows:

1. Change the values of the constants *N* and *W*:
```
parameter N = 3;
parameter W = 13;
```
2. In the initial block, change the initialization of the the *Instr_List* array to the following:
```
Instr_List[0]  = 32'h30000000; // 0. andi $0, $0, 0  ;$0 = 0
Instr_List[1]  = 32'h8C01000A; // 1. lw $1, 10($0)
Instr_List[2]  = 32'h8C02000B; // 2. lw $2, 11($0)
Instr_List[3]  = 32'h8C03000C; // 3. lw $3, 12($0)
Instr_List[4]  = 32'h00000827; // 4. rbit $1
Instr_List[5]  = 32'h00001027; // 5. rbit $2
Instr_List[6]  = 32'h00001827; // 6. rbit $3
Instr_List[7]  = 32'hAC010040; // 7. sw $1, 64($0)  ; Mem(64) =
$1
Instr_List[8]  = 32'hAC020041; // 8. sw $2, 65($0)  ; Mem(65) =
$2
Instr_List[9]  = 32'hAC030042; // 9. sw $3, 66($0)  ; Mem(66) =
$3
Instr_List[10] = 32'h1E6A2C48; // 10. Data memory A
Instr_List[11] = 32'hA5A5A5A5; // 11. Data memory B
Instr_List[12] = 32'hDEADBEEF; // 12. Data memory C
```
3. In the initial block, change the initialization of the *exp_output* array to the following:
```
exp_output[0]  = 32'h12345678;
exp_output[1]  = 32'hA5A5A5A5;
exp_output[2]  = 32'hF77DB57B;
```
4. In the main always block, remove the `if (!(VCNT == exp_V[i-1]))` conditional.

9.24 (a)




```
`define DE_Opcode DE_IR[31:26]
`define DE_Fcode DE_IR[5:0]
`define EX_Opcode EX_IR[31:26]
`define EX_Fcode EX_IR[5:0]
`define MEM_Opcode MEM_IR[31:26]
`define SR1 DE_IR[25:21]
`define SR2 DE_IR[20:16]

module Pipelined_MIPS (CLK, RST, DMem_CS, DMem_WE, DMem_ADDR, IMEM_ADDR,
DMEM_Bus, IMEM_Out);
  input  CLK;
  input  RST;
  output DMem_CS;
  inout  DMem_WE;
  output [31:0] DMem_ADDR, IMEM_ADDR;
  inout  [31:0] DMEM_Bus;
  input  [31:0] IMEM_Out;

  //non-special instructions, values of opcodes:
  parameter addi = 6'b001000;
  parameter andi = 6'b001100;
  parameter ori = 6'b001101;
  parameter lw = 6'b100011;
  parameter sw = 6'b101011;
  parameter beq = 6'b000100;
  parameter bne = 6'b000101;
  parameter j = 6'b000010;

  //instruction format
  parameter R = 2'd0;
  parameter I = 2'd1;
  parameter J = 2'd2;

  // Combinational Logic
  wire unresolved_branch, DE_isBranch, DE_modsDR, DE_needsSR1,
      DE_needsSR2, dep_stall, EX_Imm, RegW, EX_modsSR1, EX_modsSR2,
      MEM_modsSR1, MEM_modsSR2;
  wire [31:0] PC_Mux, Jump_PC, Imm_Ext, ALU_Out, A, B,
             RegIn, ReadReg1, ReadReg2;
  wire [4:0] DE_DR;
  wire [1:0] DE_Format;
  wire [2:0] Op;

  // LATCHES
  reg [31:0] PC, DE_nPC, EX_nPC, DE_IR, EX_IR, MEM_IR,
             EX_SR1, EX_SR2, MEM_SR2, MEM_ALU_Result,
             WB_ALU_Result, WB_MEM_Output;
  reg [4:0] EX_DR, MEM_DR, WB_DR;
  reg [1:0] EX_Format;
  reg DE_V, EX_V, MEM_V, WB_V, EX_isBranch, EX_modsDR,
            MEM_modsDR, WB_modsDR, WB_Store_Sel;
```

```
  initial begin
    PC <= 32'h00000000;
    DE_V <= 1'b0; EX_V <= 1'b0; MEM_V <= 1'b0; WB_V <= 1'b0;
  end

  //===============//
  //  Fetch Stage  //
  //===============//
  assign IMEM_ADDR = PC;
  assign unresolved_branch = (DE_isBranch && DE_V) || (EX_isBranch && EX_V);
  assign PC_Mux = (EX_isBranch == 1'b1 && EX_V == 1'b1)? Jump_PC : PC + 1;

  //================//
  //  Decode Stage  //
  //================//
  REG Register(CLK, RegW, WB_DR, `SR1, `SR2, RegIn, ReadReg1, ReadReg2);
  assign DE_Format = (`DE_Opcode == 6'd0)? R : ((`DE_Opcode == 6'd2)? J : I);
  assign DE_DR = (DE_Format == R)? DE_IR[15:11] : DE_IR[20:16];
  assign DE_isBranch = ((DE_Format == J) || (`DE_Opcode == bne) || (`DE_Opcode
== beq) ||
                        (`DE_Opcode == 6'd0 && `DE_Fcode == 6'd8))? 1'b1 : 1'b0;
  assign DE_modsDR = ((DE_Format != J) &&
                      (!(`DE_Opcode == 6'd0 && `DE_Fcode == 6'd8)) &&
(`DE_Opcode != sw) &&
                      (`DE_Opcode != beq) && (`DE_Opcode != bne))? 1'b1 : 1'b0;
  assign DE_needsSR1 = ((DE_V == 1'b1) && (DE_Format != J) &&
                        !(`DE_Opcode == 6'd0 && (`DE_Fcode == 6'd2 || `DE_Fcode
== 6'd0)))?
                                                                           1'b1
: 1'b0;
  assign DE_needsSR2 = ((DE_V == 1'b1) && (DE_Format != J) &&
                        !(`DE_Opcode == 6'd0 && `DE_Fcode == 6'd8) &&
                        (`DE_Opcode != lw) && (`DE_Opcode != ori) &&
                        (`DE_Opcode != andi) && (`DE_Opcode != addi))? 1'b1 :
1'b0;
  assign EX_modsSR1 = ((EX_V == 1'b1) && (EX_DR == `SR1) && (EX_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign EX_modsSR2 = ((EX_V == 1'b1) && (EX_DR == `SR2) && (EX_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign MEM_modsSR1 = ((MEM_V == 1'b1) && (MEM_DR == `SR1) && (MEM_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign MEM_modsSR2 = ((MEM_V == 1'b1) && (MEM_DR == `SR2) && (MEM_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign dep_stall = (DE_needsSR1 && (EX_modsSR1 || MEM_modsSR1)) ||
                     (DE_needsSR2 && (EX_modsSR2 || MEM_modsSR2));

  //=================//
  //  Execute Stage  //
  //=================//
  ALU Alu1(A, B, Op, ALU_Out);
  assign Imm_Ext = (EX_IR[15] == 1'b1)? {16'hFFFF, EX_IR[15:0]} : {16'h0000,
EX_IR[15:0]};
  assign EX_Imm = ((`EX_Opcode == addi) || (`EX_Opcode == lw) || (`EX_Opcode ==
sw) ||
                  (`EX_Opcode == andi) || (`EX_Opcode == ori))? 1'b1 : 1'b0;
  assign A = ((`EX_Opcode == 6'd0) && (`EX_Fcode == 6'd2 || `EX_Fcode == 6'd0))?
                                 {24'h000000, 3'b000, EX_IR[10:6]} : EX_SR1;
  assign B = (EX_Imm == 1'b1)? Imm_Ext : EX_SR2;
  assign Op = ((`EX_Opcode == 6'd0 && (`EX_Fcode == 6'd32)) ||
              (`EX_Opcode == addi) || (`EX_Fcode == lw) || (`EX_Opcode == sw))?
3'b000 :
            ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd34) ||
               (`EX_Opcode == beq) || (`EX_Opcode == bne))? 3'b001 :
            ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd36) || (`EX_Opcode ==
andi))? 3'b010 :
            ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd37) || (`EX_Opcode ==
ori))? 3'b011 :
            ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd42))? 3'b100 : //slt
            ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd2))? 3'b101 : //shr
            ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd0))? 3'b110 : //shl
                                                        3'b000 ))))));
```

```
  assign Jump_PC = (EX_Format == J)? {6'b000000, EX_IR[25:0]} :
                 ( ((ALU_Out == 32'h00000000 && `EX_Opcode == beq) ||
                    (ALU_Out != 32'h00000000 && `EX_Opcode == bne))? EX_nPC +
Imm_Ext :
                 ( (`EX_Opcode == 6'd0 && `EX_Fcode == 6'd8)? EX_SR1 : EX_nPC
));

  //================//
  //  Memory Stage  //
  //================//
  assign DMem_ADDR = MEM_ALU_Result;
  assign DMem_Bus = (DMem_WE == 1'b0)? 32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ :
MEM_SR2;
  assign DMem_WE = ((MEM_V == 1'b1) && (`MEM_Opcode == sw))? 1'b1 : 1'b0;
  assign DMem_CS = ((MEM_V == 1'b1) && ((`MEM_Opcode == sw) || (`MEM_Opcode ==
lw)))? 1'b1: 1'b0;

  //====================//
  //  Write-Back Stage  //
  //====================//
  assign RegIn = (WB_Store_Sel == 1'b0)? WB_ALU_Result : WB_MEM_Output;
  assign RegW = WB_modsDR && WB_V;

  //====================//
  //  Pipeline Latches  //
  //--------------------//
  always @(posedge CLK, posedge RST) begin
    if(RST == 1'b1) begin
      PC <= 32'h00000000;
      DE_V <= 1'b0; EX_V <= 1'b0; MEM_V <= 1'b0; WB_V <= 1'b0;
    end
    else begin
      //===============//
      //  Fetch Stage  //
      //===============//
      //do not overwrite DE's latches if stalled on a dependency.
      if (dep_stall == 1'b0) begin
        DE_nPC <= PC_Mux;
        DE_IR <= IMEM_Out;
        PC <= PC_Mux;
        //insert a bubble into the pipeline if there is an unresolved branch.
        if (unresolved_branch == 1'b1) DE_V <= 1'b0;
        else DE_V <= 1'b1;
      end
      else begin
        DE_nPC <= DE_nPC;
        DE_IR <= DE_IR;
        PC <= PC;
        DE_V <= DE_V;
      end

      //================//
      //  Decode Stage  //
      //================//
      EX_nPC <= DE_nPC;
      EX_IR <= DE_IR;
      EX_DR <= DE_DR;
      EX_Format <= DE_Format;
      EX_isBranch <= DE_isBranch;
      EX_modsDR <= DE_modsDR;
      EX_SR1 <= ReadReg1;
      EX_SR2 <= ReadReg2;
      EX_V <= DE_V && !(dep_stall);

      //=================//
      //  Execute Stage  //
      //=================//
      MEM_IR <= EX_IR;
      MEM_DR <= EX_DR;
      MEM_modsDR <= EX_modsDR;
      MEM_SR2 <= EX_SR2;
```

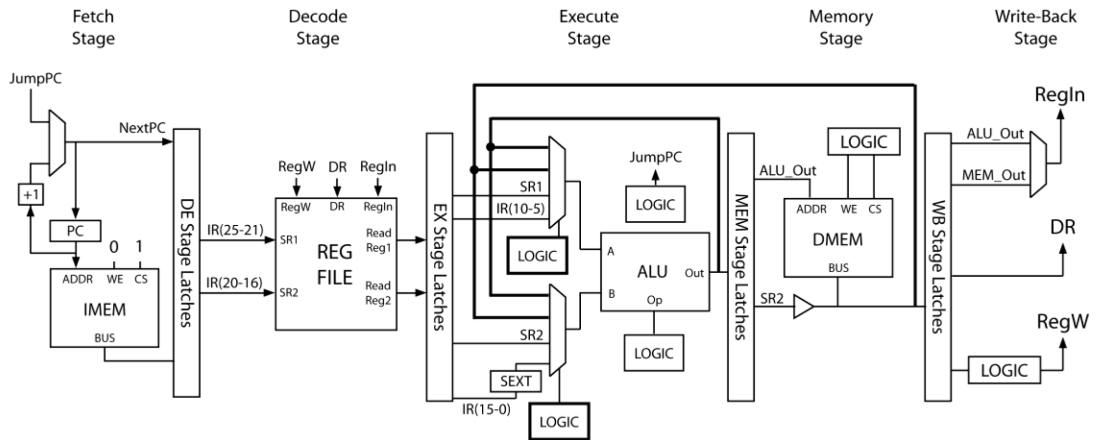
```
        MEM_ALU_Result <= ALU_Out;
        MEM_V <= EX_V;

        //================//
        //  Memory Stage  //
        //================//
        WB_DR <= MEM_DR;
        WB_modsDR <= MEM_modsDR;
        WB_ALU_Result <= MEM_ALU_Result;
        WB_MEM_Output <= DMEM_Bus;
        if (`MEM_Opcode == lw) WB_Store_Sel <= 1'b1;
        else WB_Store_Sel <= 1'b0;
        WB_V <= MEM_V;
      end
    end
endmodule
```

(b) $4 + n$

(c) 11 cycles

Cycle	1	2	3	4	5	6	7	8	9	10	11
`add $5, $4, $3`	F	D	Ex	M	Wb						
`add $6, $5, $4`		F	D	D	Ex	M	Wb				
`add $7, $6, $5`			F	F	D	D	Ex	M	Wb		
`add $8,$7,$6`					F	F	D	D	Ex	M	Wb

9.25 **(a)**




```
  `define DE_Opcode DE_IR[31:26]
  `define DE_Fcode DE_IR[5:0]
  `define EX_Opcode EX_IR[31:26]
  `define EX_Fcode EX_IR[5:0]
  `define MEM_Opcode MEM_IR[31:26]
  `define SR1 DE_IR[25:21]
  `define SR2 DE_IR[20:16]

module Pipelined_MIPS (CLK, RST, DMem_CS, DMem_WE, DMem_ADDR, IMEM_ADDR,
DMEM_Bus, IMEM_Out);
  input  CLK;
  input  RST;
  output DMem_CS;
  inout  DMem_WE;
  output [31:0] DMem_ADDR, IMEM_ADDR;
  inout  [31:0] DMEM_Bus;
  input  [31:0] IMEM_Out;

  //non-special instructions, values of opcodes:
```

```
  parameter addi = 6'b001000;
  parameter andi = 6'b001100;
  parameter ori = 6'b001101;
  parameter lw = 6'b100011;
  parameter sw = 6'b101011;
  parameter beq = 6'b000100;
  parameter bne = 6'b000101;
  parameter j = 6'b000010;

  //instruction format
  parameter R = 2'd0;
  parameter I = 2'd1;
  parameter J = 2'd2;

  // Combinational Logic
  wire unresolved_branch, DE_isBranch, DE_modsDR, DE_needsSR1,
      DE_needsSR2, dep_stall, EX_Imm, RegW, EX_modsSR1, EX_modsSR2,
      MEM_modsSR1, MEM_modsSR2, fwd_MEM_Out_to_SR1, fwd_ALU_Out_to_SR1,
      fwd_MEM_ALU_to_SR1, fwd_MEM_Out_to_SR2, fwd_ALU_Out_to_SR2,
      fwd_MEM_ALU_to_SR2;
  wire [31:0] PC_Mux, Jump_PC, Imm_Ext, ALU_Out, A, B,
             RegIn, ReadReg1, ReadReg2, SR1_Mux, SR2_Mux;
  wire [4:0] DE_DR;
  wire [1:0] DE_Format;
  wire [2:0] Op;

  // LATCHES
  reg [31:0] PC, DE_nPC, EX_nPC, DE_IR, EX_IR, MEM_IR,
             EX_SR1, EX_SR2, MEM_SR2, MEM_ALU_Result,
             WB_ALU_Result, WB_MEM_Output;
  reg [4:0] EX_DR, MEM_DR, WB_DR;
  reg [1:0] EX_Format;
  reg DE_V, EX_V, MEM_V, WB_V, EX_isBranch, EX_modsDR,
            MEM_modsDR, WB_modsDR, WB_Store_Sel;

  initial begin
    PC <= 32'h00000000;
    DE_V <= 1'b0; EX_V <= 1'b0; MEM_V <= 1'b0; WB_V <= 1'b0;
  end

  //===============//
  //  Fetch Stage  //
  //===============//
  assign IMEM_ADDR = PC;
  assign unresolved_branch = (DE_isBranch && DE_V) || (EX_isBranch && EX_V);
  assign PC_Mux = (EX_isBranch == 1'b1 && EX_V == 1'b1)? Jump_PC : PC + 1;

  //================//
  //  Decode Stage  //
  //================//
  REG Register(CLK, RegW, WB_DR, `SR1, `SR2, RegIn, ReadReg1, ReadReg2);
  assign DE_Format = (`DE_Opcode == 6'd0)? R : ((`DE_Opcode == 6'd2)? J : I);
  assign DE_DR = (DE_Format == R)? DE_IR[15:11] : DE_IR[20:16];
  assign DE_isBranch = ((DE_Format == J) || (`DE_Opcode == bne) || (`DE_Opcode
== beq) ||
                        (`DE_Opcode == 6'd0 && `DE_Fcode == 6'd8))? 1'b1 : 1'b0;
  assign DE_modsDR = ((DE_Format != J) &&
                      (!(`DE_Opcode == 6'd0 && `DE_Fcode == 6'd8)) &&
(`DE_Opcode != sw) &&
                      (`DE_Opcode != beq) && (`DE_Opcode != bne))? 1'b1 : 1'b0;
  assign DE_needsSR1 = ((DE_V == 1'b1) && (DE_Format != J) &&
                        !(`DE_Opcode == 6'd0 && (`DE_Fcode == 6'd2 || `DE_Fcode
== 6'd0)))?
                                                                          1'b1
: 1'b0;
  assign DE_needsSR2 = ((DE_V == 1'b1) && (DE_Format != J) &&
                        !(`DE_Opcode == 6'd0 && `DE_Fcode == 6'd8) &&
                        (`DE_Opcode != lw) && (`DE_Opcode != ori) &&
                        (`DE_Opcode != andi) && (`DE_Opcode != addi))? 1'b1 :
1'b0;
```

```
  assign EX_modsSR1 = ((EX_V == 1'b1) && (EX_DR == `SR1) && (EX_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign EX_modsSR2 = ((EX_V == 1'b1) && (EX_DR == `SR2) && (EX_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign MEM_modsSR1 = ((MEM_V == 1'b1) && (MEM_DR == `SR1) && (MEM_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign MEM_modsSR2 = ((MEM_V == 1'b1) && (MEM_DR == `SR2) && (MEM_modsDR ==
1'b1))? 1'b1 : 1'b0;
  assign dep_stall = (((DE_needsSR1 == 1'b1) && (EX_modsSR1 == 1'b1) &&
(`EX_Opcode == lw)) ||
                      ((DE_needsSR2 == 1'b1) && (EX_modsSR2 == 1'b1) &&
(`EX_Opcode == lw)))? 1'b1 : 1'b0;
  assign fwd_ALU_Out_to_SR1 = ((DE_needsSR1 == 1'b1) && (EX_modsSR1 == 1'b1) &&
                               (`EX_Opcode != lw))? 1'b1 : 1'b0;
  assign fwd_MEM_Out_to_SR1 = ((DE_needsSR1 == 1'b1) && (EX_modsSR1 == 1'b0) &&
                               (MEM_modsSR1 == 1'b1) && (`MEM_Opcode == lw))?
1'b1 : 1'b0;
  assign fwd_MEM_ALU_to_SR1 = ((DE_needsSR1 == 1'b1) && (EX_modsSR1 == 1'b0) &&
                               (MEM_modsSR1 == 1'b1) && (`MEM_Opcode != lw))?
1'b1 : 1'b0;
  assign fwd_ALU_Out_to_SR2 = ((DE_needsSR2 == 1'b1) && (EX_modsSR2 == 1'b1) &&
                               (`EX_Opcode != lw))? 1'b1 : 1'b0;
  assign fwd_MEM_Out_to_SR2 = ((DE_needsSR2 == 1'b1) && (EX_modsSR2 == 1'b0) &&
                               (MEM_modsSR2 == 1'b1) && (`MEM_Opcode == lw))?
1'b1 : 1'b0;
  assign fwd_MEM_ALU_to_SR2 = ((DE_needsSR2 == 1'b1) && (EX_modsSR2 == 1'b0) &&
                               (MEM_modsSR2 == 1'b1) && (`MEM_Opcode != lw))?
1'b1 : 1'b0;
  assign SR1_Mux = (fwd_ALU_Out_to_SR1 == 1'b1)? ALU_Out :
                 ( (fwd_MEM_Out_to_SR1 == 1'b1)? DMEM_Bus :
                 ( (fwd_MEM_ALU_to_SR1 == 1'b1)? MEM_ALU_Result : ReadReg1 ));
  assign SR2_Mux = (fwd_ALU_Out_to_SR2 == 1'b1)? ALU_Out :
                 ( (fwd_MEM_Out_to_SR2 == 1'b1)? DMEM_Bus :
                 ( (fwd_MEM_ALU_to_SR2 == 1'b1)? MEM_ALU_Result : ReadReg2 ));

  //=================//
  //  Execute Stage  //
  //=================//
  ALU Alu1(A, B, Op, ALU_Out);
  assign Imm_Ext = (EX_IR[15] == 1'b1)? {16'hFFFF, EX_IR[15:0]} : {16'h0000,
EX_IR[15:0]};
  assign EX_Imm = ((`EX_Opcode == addi) || (`EX_Opcode == lw) || (`EX_Opcode ==
sw) ||
                   (`EX_Opcode == andi) || (`EX_Opcode == ori))? 1'b1 : 1'b0;
  assign A = ((`EX_Opcode == 6'd0) && (`EX_Fcode == 6'd2 || `EX_Fcode == 6'd0))?
                              {24'h000000, 3'b000, EX_IR[10:6]} : EX_SR1;
  assign B = (EX_Imm == 1'b1)? Imm_Ext : EX_SR2;
  assign Op = ((`EX_Opcode == 6'd0 && (`EX_Fcode == 6'd32)) ||
              (`EX_Opcode == addi) || (`EX_Fcode == lw) || (`EX_Opcode == sw))?
3'b000 :
           ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd34) ||
              (`EX_Opcode == beq) || (`EX_Opcode == bne))? 3'b001 :
           ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd36) || (`EX_Opcode ==
andi))? 3'b010 :
           ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd37) || (`EX_Opcode ==
ori))? 3'b011 :
           ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd42))? 3'b100 : //slt
           ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd2))? 3'b101 : //shr
           ( ((`EX_Opcode == 6'd0 && `EX_Fcode == 6'd0))? 3'b110 : //shl
                                                       3'b000 ))))));
  assign Jump_PC = (EX_Format == J)? {6'b000000, EX_IR[25:0]} :
                 ( ((ALU_Out == 32'h00000000 && `EX_Opcode == beq) ||
                    (ALU_Out != 32'h00000000 && `EX_Opcode == bne))? EX_nPC +
Imm_Ext :
                 ( (`EX_Opcode == 6'd0 && `EX_Fcode == 6'd8)? EX_SR1 : EX_nPC
));

  //================//
  //  Memory Stage  //
  //================//
  assign DMem_ADDR = MEM_ALU_Result;
```

```
  assign DMem_Bus = (DMem_WE == 1'b0)? 32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ :
MEM_SR2;
  assign DMem_WE = ((MEM_V == 1'b1) && (`MEM_Opcode == sw))? 1'b1 : 1'b0;
  assign DMem_CS = ((MEM_V == 1'b1) && ((`MEM_Opcode == sw) || (`MEM_Opcode ==
lw)))? 1'b1: 1'b0;

  //====================//
  //  Write-Back Stage  //
  //====================//
  assign RegIn = (WB_Store_Sel == 1'b0)? WB_ALU_Result : WB_MEM_Output;
  assign RegW = WB_modsDR && WB_V;

  //====================//
  //  Pipeline Latches  //
  //--------------------//
  always @(posedge CLK, posedge RST) begin
    if(RST == 1'b1) begin
      PC <= 32'h00000000;
      DE_V <= 1'b0; EX_V <= 1'b0; MEM_V <= 1'b0; WB_V <= 1'b0;
    end
    else begin
      //===============//
      //  Fetch Stage  //
      //===============//
      //do not overwrite DE's latches if stalled on a dependency.
      if (dep_stall == 1'b0) begin
        DE_nPC <= PC_Mux;
        DE_IR <= IMEM_Out;
        PC <= PC_Mux;
        //insert a bubble into the pipeline if there is an unresolved branch.
        if (unresolved_branch == 1'b1) DE_V <= 1'b0;
        else DE_V <= 1'b1;
      end
      else begin
        DE_nPC <= DE_nPC;
        DE_IR <= DE_IR;
        PC <= PC;
        DE_V <= DE_V;
      end

      //================//
      //  Decode Stage  //
      //================//
      EX_nPC <= DE_nPC;
      EX_IR <= DE_IR;
      EX_DR <= DE_DR;
      EX_Format <= DE_Format;
      EX_isBranch <= DE_isBranch;
      EX_modsDR <= DE_modsDR;
      EX_SR1 <= SR1_Mux;
      EX_SR2 <= SR2_Mux;
      EX_V <= DE_V && !(dep_stall);

      //=================//
      //  Execute Stage  //
      //=================//
      MEM_IR <= EX_IR;
      MEM_DR <= EX_DR;
      MEM_modsDR <= EX_modsDR;
      MEM_SR2 <= EX_SR2;
      MEM_ALU_Result <= ALU_Out;
      MEM_V <= EX_V;

      //================//
      //  Memory Stage  //
      //================//
      WB_DR <= MEM_DR;
      WB_modsDR <= MEM_modsDR;
      WB_ALU_Result <= MEM_ALU_Result;
      WB_MEM_Output <= DMEM_Bus;
      if (`MEM_Opcode == lw) WB_Store_Sel <= 1'b1;
```

```
      else WB_Store_Sel <= 1'b0;
      WB_V <= MEM_V;
    end
  end
endmodule
```

(b) Assuming solution 9.10 is used as a test program:
Figure 9-8 (non-pipelined) – 2302 cycles
Solution 9.24 (pipelined) – 1395 cycles
Solution 9.25 (pipelined with data forwarding) – 896 cycles

Chapter 10: Hardware Testing and Design for Testability

10.1 **(a) & (b)**

A	B	C	D	Faults Tested
X	1	0	1	u0, b0, d0, w0
X	1	1	0	u0, b0, c0, w0

(c)

A	B	C	D	Faults Tested
0	1	0	0	a1, c1, d1, q1, r1, u1, v1, w1, p0

10.2 **(a)**

A	B	C	D	Faults Tested
0	0	1	X	a1, b1, e1, g1, h1, i1, j1

OR

A	B	C	D	Faults Tested
0	0	1	1	a1, b1, e1, g1, h1, i1, j1, f0

(b)

A	B	C	D	Faults Tested
X	X	0	1	d0, g0, i0, j0, f1

10.3

a	b	c	d	e	f	i	Faults Tested
1	0	0	1	0	0	1	a0, d0, g0, h0, i0, Z0
0	1	0	0	1	0	1	b0, e0, g0, h0, i0, Z0
0	0	1	0	0	1	1	c0, f0, g0, h0, i0, Z0
0	0	0	1	X	X	1	a1, b1, c1, g1, Z1
1	X	X	0	0	0	1	d1, e1, f1, h1, Z1
1	X	X	1	X	X	0	i1, Z1

10.4

A	B	C	D	Faults Tested
1	1	0	0	A0, B0, P0, E0
0	X	1	0	C0, Q0, E0
0	X	0	1	D0, Q0, E0
0	1	0	0	A1, C1, D1, P1, Q1, E1
1	0	0	0	B1, C1, D1, P1, Q1, E1

10.5

a	b	c	d	e	Faults Tested
1	0	1	0	0	a0, c0, f0, g0, h0, b1
0	X	X	1	0	d0, h0
0	X	X	0	1	e0, h0
0	0	1	0	0	a1, d1, e1, g1, h1
1	1	1	0	0	d1, e1, f1, g1, h1, b0
1	0	0	0	0	c1, d1, e1, g1, h1

10.6

A	B	C	D	Faults Tested
0	1	X	1	k0, l0, m0, r0, z0
1	1	0	X	e0, f0, g0, p0, z0
1	0	1	X	h0, i0, j0, q0, z0
0	0	1	1	h1, l1, p1, q1, r1, z1
0	1	0	0	e1, m1, p1, q1, r1, z1
1	0	0	X	f1, j1, p1, q1, r1, z1
1	1	1	1	g1, i1, k1, p1, q1, r1, z1

10.7 Test sequence: '1', '1', '-'.

time	state	input
t0	00	1
t1	10	1
t2	01	-
t3	11 (correct state), 10 (incorrect state)	

10.8

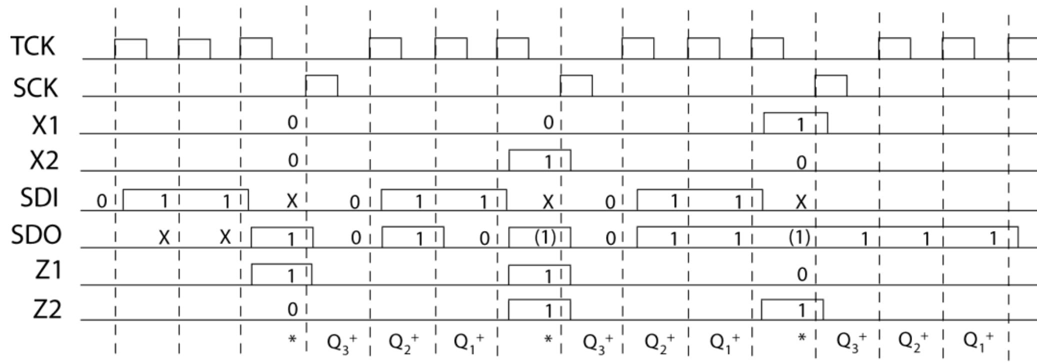
State	Distinguishing Sequence	Output
S1	11	01
S2	11	10
S3	11	00

Testing State	**Input Sequence**	Correct Output	Correct State
S1	**R011**	000	S3
S1	**R111**	010	S2
S3	**R0011**	0101	S1
S3	**R0111**	0000	S3
S2	**R1011**	0101	S1
S2	**R1111**	0100	S3

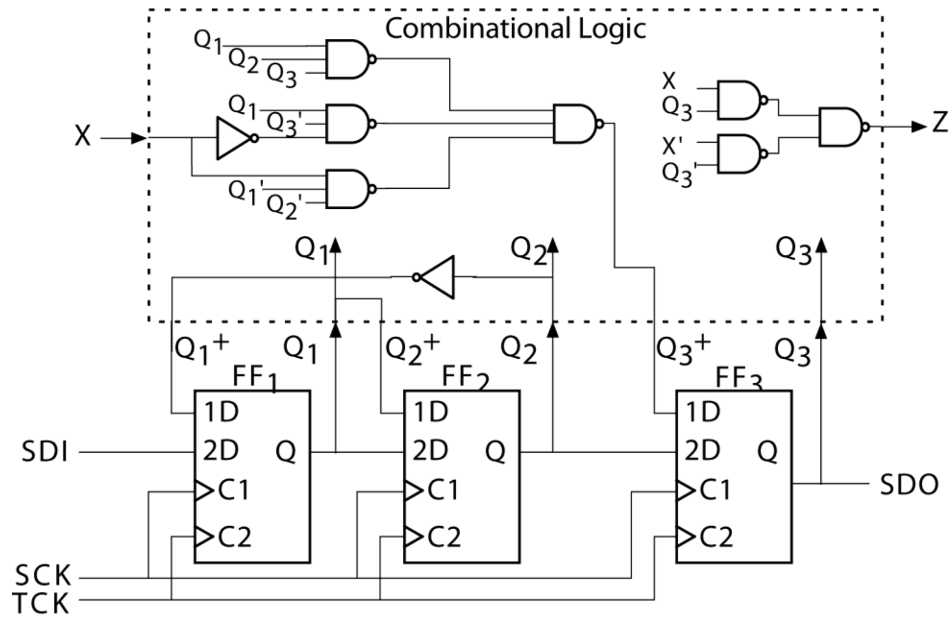
10.9 Input: 0, 0, 0, 1, 1, 1, 1 Correct Output: 1, 1, 1, 0, 0, 0, 1
Incorrect Output: 1, 1, 1, 0, 0, 0, 0

10.10 Some major advantages of scan-path testing over applying input sequences and observing output sequences are that you save time and save effort. Instead of applying an input sequence to get to the state that needs to be tested, with scan-path testing, just serially shift in the state. After testing the state, there is no need to use a distinguishing sequence and interpret the output to determine the state of the machine. Scan-path testing allows the flip-flop values to be serially shifted out and easily read. This method reduces testing sequential circuits to testing combinational circuits.

10.11



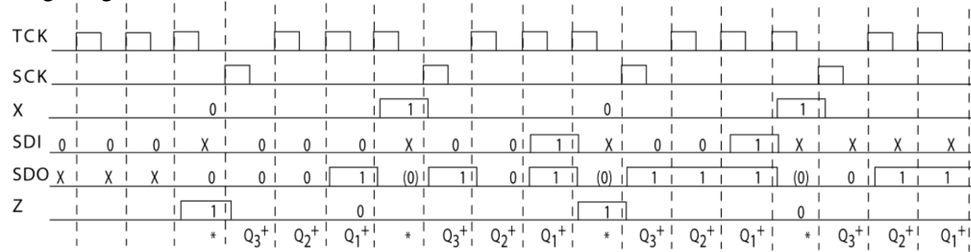
10.12 (a)



(b) First 2 rows of the transition table:

	$Q_1^+ Q_2^+ Q_3^+$		Z	
$Q_1 Q_2 Q_3$	X = 0	X = 1	X = 0	X = 1
000	100	101	1	0
100	111	110	1	0

Timing Diagram:



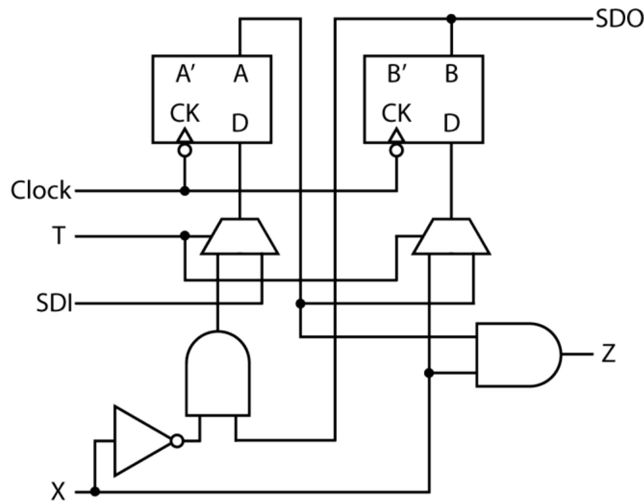
10.13 **(a)**
```
module DPFF(D1, D2, C1, C2, Q);
  input D1, D2, C1, C2;
  output Q;

  reg Q;

  initial begin
    Q = 0;
  end

  always @(C1, C2)
  begin
    if(C1 == 1'b1)
      Q <= D1;
    else if(C2 == 1'b1)
      Q <= D2;
    else begin
    end
  end

endmodule
```

(b)
```
module convert(X, SDI, SCK, TCK, Z, SDO);
  input X, SDI, SCK, TCK;
  output Z, SDO;

  wire Q1, Q2, Q3, Q2N, Q3N, D3;

  assign Q2N = ~Q2;
  assign Q3N = ~Q3;
  assign SDO = Q3;

  DPFF DPFF1(Q2N, SDI, SCK, TCK, Q1);
  DPFF DPFF2(Q1, Q1, SCK, TCK, Q2);
  DPFF DPFF3(D3, Q2, SCK, TCK, Q3);

  assign Z = (X & Q3) | (~X & ~Q3);
  assign D3 = (Q1 & Q2 & Q3) | (~X & Q1 & ~Q3) | (X & ~Q1 & ~Q2);
endmodule
```

(c)
```
module testconverter;
  reg CLK;
  reg [3:0] testvector [3:0];
  reg X, SDI, SCK, TCK;
  wire Z, SDO;
  integer i;
  integer j;

  initial begin
    CLK = 0;
    X = 0;
    SDI = 0;
    SCK = 0;
    TCK = 0;
    testvector[0] = 4'b1001;
    testvector[1] = 4'b1000;
    testvector[2] = 4'b0001;
    testvector[3] = 4'b0000;
  end

  always #50 CLK = ~CLK;
```

```
    convert conv(X, SDI, SCK, TCK, Z, SDO);

    always
    begin
      for(i  = 3; i >= 0; i = i - 1) begin
        for(j = 1; j <= 3; j = j + 1) begin
          SDI <= testvector[i][j];
          @(posedge CLK)
          TCK <= 1;
          @(negedge CLK)
          TCK <= 0;
        end
        X <= testvector[i][0];
        @(posedge CLK)
        SCK <= 1;
        @(negedge CLK)
        SCK <= 0;
      end
    end

  endmodule
```

10.14



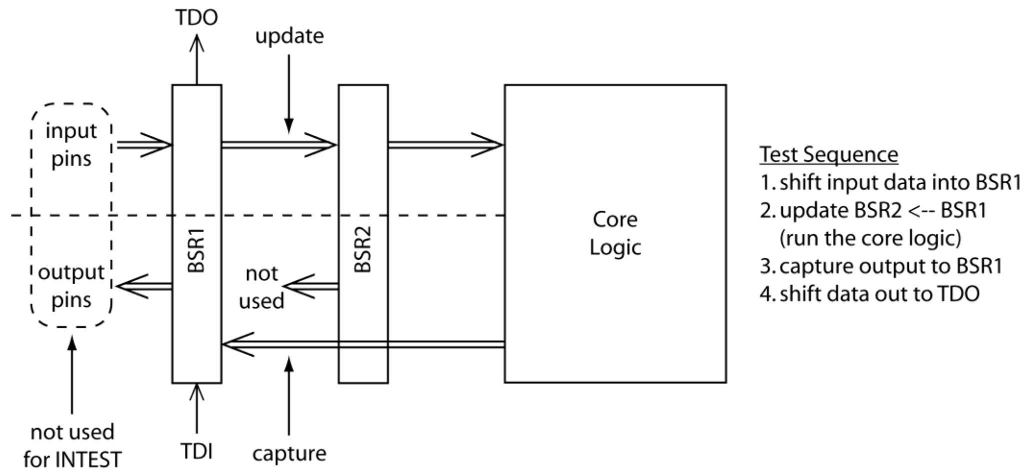
10.15

State	0	1	2	9	10	11	11	11	12	15	2	3	4	4	4	4	5	8	1
TMS	0	1	1	0	0	0	0	1	1	1	0	0	0	0	0	1	1	0	
TDI	-	-	-	-	-	1	1	0	-	-	-	-	1	0	1	1	-	-	

10.16 (a)

State	0	1	2	9	10	11	11	11	12	15	2	3	4	4	4	5	8	1
TMS	0	1	1	0	0	0	0	1	1	1	0	0	0	0	1	1	0	
TDI	-	-	-	-	-	0	1	0	-	-	-	-	1	1	0	-	-	

(b)



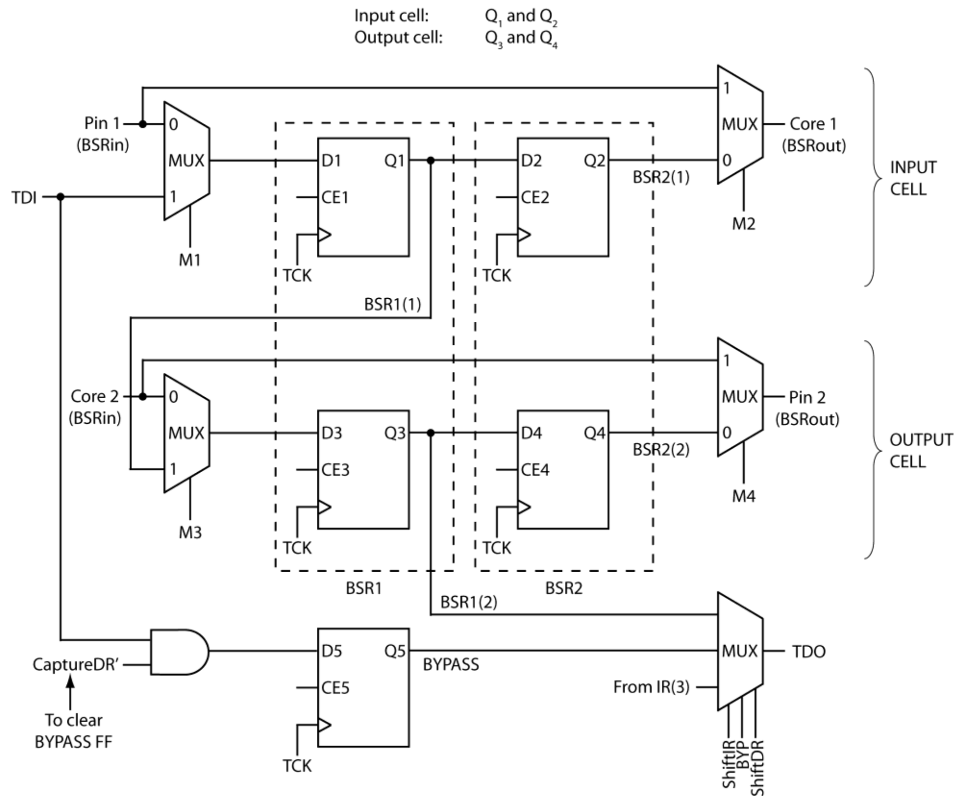
Changes to code:

CaptureDR state, insert after IDR = "001": `or IDR = "010"`

UpdateDR state, insert after IDR = "001": `or IDR = "010"`

BSRout statement: no change since BSR2 already goes to core logic for input cells when IDR = "010"

10.17



10.18 In Figure 10-22, in the list of concurrent assignments, change:
`assign BSR1in[1] = BSR2out[4];` to `assign BSR1in[1] = 1'b0;`

10.19 Verilog code for Figure 10-14 (b) is as follows:

```
module BS_Cell(TDI, BSRin, SLmode, TNmode, TCK, Capture_Shift, Update,
TDO, BSRout);
  input TDI, BSRin, SLmode, TNmode, TCK, Capture_Shift, Update;
  output TDO, BSRout;

  reg Q1, Q2;
  wire D1;

  initial begin
    Q1 = 0;
    Q2 = 0;
  end

  assign BSRout = (TNmode == 1'b1)? BSRin : Q2;
  assign D1 = (SLmode == 1'b1)? BSRin : TDI;
  assign TDO = Q1;

  always @(posedge TCK)
  begin
    if(Capture_Shift == 1'b1)
      Q1 <= D1;
    if(Update == 1'b1)
      Q2 <= Q1;
  end

endmodule
```

To use the above boundary scan-cell, amend the code of Figure 10-21 as follows:

- To the list of internal signals, **add**:

```
reg [1:NCELLS] Capture_Shift, Update;
wire [0:NCELLS] TD;
wire SLmode, TNmode;
wire F1;
```

- **Add** the following generate statement:

```
genvar i;
generate
  for(i = 1; i <= NCELLS; i= i + 1) begin
    BS_Cell celli(TD[i-1], BSRin[i], SLmode, TNmode, TCK,
                        Capture_Shift[i], Update[i], TD[i], BSRout[i]);
  end
 endgenerate
```

- **Change** case "`CaptureDR:`" to:

```
CaptureDR: begin
  if(IDR == 3'b111)
    BYPASS <= 0;
  if(TMS == 0)
    St <= ShiftDR;
  else
    St <= Exit1DR;
end
```

- **Change** case "`ShiftDR:`" to:

```
ShiftDR: begin
  if(IDR == 3'b111)
```

```
        BYPASS <= TDI;
      if(TMS == 1'b0)
        St <= ShiftDR;
      else
        St <= Exit1DR;
    end
```

- **Change** case " `UpdateDR:`" to:

```
    UpdateDR: begin
      if(TMS == 1'b0)
        St <= RunTest_Idle;
      else
        St <= SelectDRScan;
       end
```

- **Add** the following concurrent statements:

```
   assign SLmode = (St == ShiftDR)? 1'b0 : 1'b1;
   assign TNmode = (St == TestLogicReset | (~(IDR == 3'b000))) ?
 1'b0 : 1'b1;
   assign TD[0] = TDI;
   assign F1 = ((St == ShiftDR) & (IDR != 3'b000));
```

- **Add** the following always block:

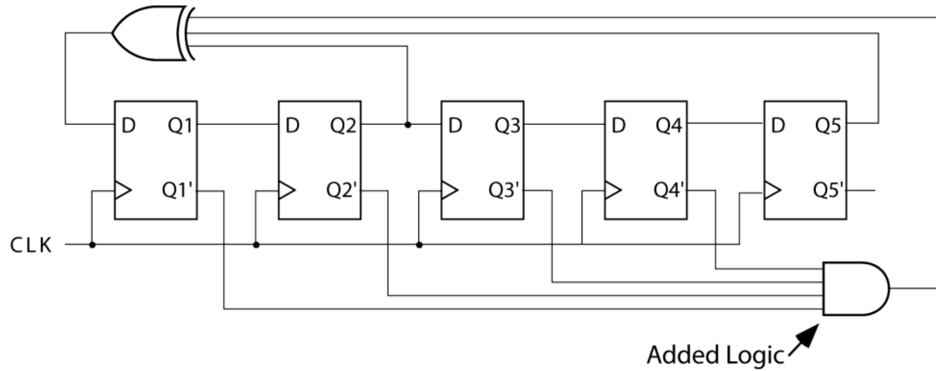
```
 integer j;
 always
 begin
   for(j = 1; j <= NCELLS; j = j + 1) begin
     if(F1 == 1'b1 || (St == CaptureDR && (IDR == 3'b001 ||
                      (IDR == 3'b000 && CellType[j] == 1'b0))))
       Capture_Shift[j] = 1;
     else
       Capture_Shift[j] = 0;
     if(St == UpdateDR &&
         (IDR == 3'b001 || (IDR == 3'b000 && CellType[j] == 1'b1)))
       Update[j] = 1;
     else
       Update[j] = 0;
   end
 end
```

- **Change** concurrent statement "`assign TDO =`" statement (2nd to last statement of the code) to:

```
  assign TDO = (St == ShiftDR && IDR==3'b111)? BYPASS :
                ((St == ShiftDR)? TD[NCELLS] : ((St == ShiftIR)?
            IR[3] : TDO));
```

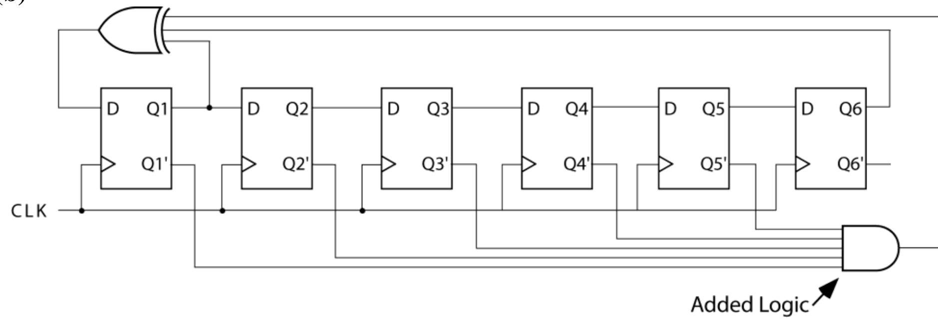
- **Delete** concurrent statement "`assign BSRout =`" statement (last statement of the code).

10.20 (a) & (b)



(c) State Sequence: 00000, 10000, 01000, 10100, 01010, 10101, 11010, 11101, 01110, 10111, 11011, 01101, 00110, 00011, 10001, 11000, 11100, 11110, 11111, 01111, 00111, 10011, 11001, 01100, 10110, 01011, 00101, 10010, 01001, 00100, 00010, 00001, 00000

10.21 (a) & (b)

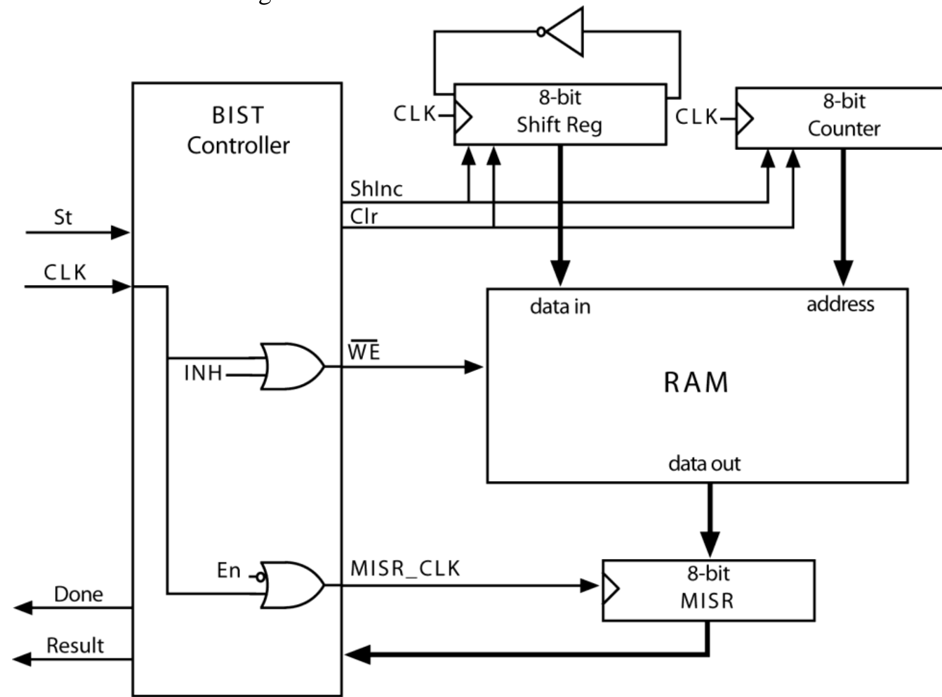


(c) State Sequence: 000000, 100000, 110000, 111000, 111100, 111110, 111111, 011111, 101111, 010111, 101011, 010101, 101010, 110101, 011010, 001101, 100110, 110011, 011001, 101100, 110110, 111011, 011101, 101110, 110111, 011011, 101101, 010110, 001011, 100101, 010010, 001001, 100100, 110010, 111001, 011100, 001110, 000111, 100011, 010001, 101000, 110100, 111010, 111101, 011110, 001111, 100111, 010011, 101001, 010100, 001010, 000101, 100010, 110001, 011000, 001100, 000110, 000011, 100001, 010000, 001000, 000100, 000010, 000001, 000000

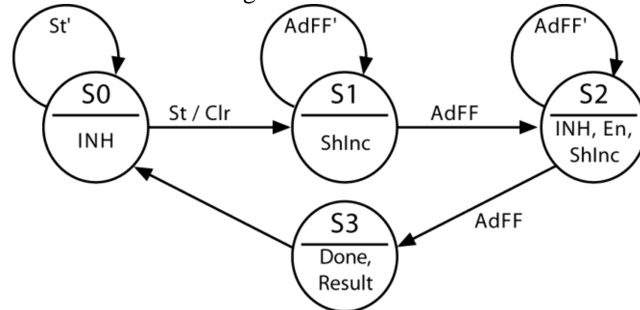
10.22 (a)

```
module MISR_8bit(CLK, Z, D, Q);
  input CLK;
  input [1:8] Z, D;
  output [1:8] Q;
  reg [1:8] Q;

  initial begin
    Q = D;
  end

  always @(posedge CLK)
  begin
    Q <= {(Q[1] ^ Q[8]), Q[1:7]} ^ Z;
  end

endmodule
```

(b) Self-Test Circuit Block Diagram:



BIST Controller State Diagram:



AdFF=1 when adress=FF
Read when INH=1
Write when INH=0

(c)
```
module RAM_Selftest(CLK, St, Done, Result);
  input CLK, St;
  output reg Done, Result;

  reg CLR, INH, ShInc, En;
  wire MISR_CLK, AdFF, We_b;
  wire [7:0] Addr, MISR_In, MISR_Out, RAMbus;
  reg [7:0] Counter;
  reg [7:0] ShREG;
  reg [1:0] State, Nextstate;
  reg Cs_b;

  initial begin
    Done = 0;
    Result = 0;
    CLR = 0;
    INH = 0;
    ShInc = 0;
    En = 0;
    Counter = 0;
    ShREG = 0;
```

```
  State = 0;
  Nextstate = 0;
  Cs_b = 0;
end

assign AdFF = (Counter == 255)? 1 : 0;
assign We_b = CLK | INH;
assign Addr = $unsigned(Counter);
assign RAMbus = (INH == 0)? ShREG : 8'bzzzzzzzz;
assign MISR_CLK = CLK | (!En);
assign MISR_In = (En == 1'b1)? RAMbus : 8'b00000000;

RAM6116 RAM(Cs_b, We_b, 1'b0, Addr, RAMbus);
MISR_8bit MISR(MISR_CLK, MISR_In, MISR_Out);

always @(State, St, AdFF)
begin
  CLR = 0; INH = 0; ShInc = 0;
  En = 0; Done = 0; Result = 0;
  case(State)
  0: begin
    INH = 1;
    if(St == 1) begin
      CLR = 1;
      Nextstate = 1;
    end
    else
      Nextstate = 0;
  end
  1: begin
    ShInc = 1;
    if(AdFF == 1)
      Nextstate = 2;
    else
      Nextstate = 1;
  end
  2: begin
    INH = 1;
    ShInc = 1;
    En = 1;
    if(AdFF == 1)
      Nextstate = 3;
    else
      Nextstate = 2;
  end
  3: begin
    Done = 1;
    Nextstate = 0;
    if(MISR_Out == 8'b10100001)
      Result = 1;
    else
      Result = 0;
  end
  endcase
end

always @(posedge CLK)
begin
  State <= Nextstate;
  if(CLR == 1) begin
    ShREG <= 8'b00000000;
    Counter <= 0;
  end
```

```
    if(ShInc == 1) begin
      ShREG <= {!ShREG[0], ShREG[7:1]};
      if(Counter == 255)
        Counter <= 0;
      else
        Counter <= Counter + 1;
    end
  end

endmodule
```

10.23 (1) Load A with 1011 and B with 1110, clear C.

B_1 B_2 S_i	Action
000	shift in C(0)
000	shift in C(1)
000	shift in C(2)
000	shift in C(3)
000	shift in B(0)
001	shift in B(1)
001	shift in B(2)
001	shift in B(3)
001	shift in A(0)
001	shift in A(1)
000	shift in A(2)
001	shift in A(3)

(2) Test the system by using A and B as pattern generators and C as a signature register for four clock times.

B_1 B_2 S_i	Action
01X	PRPG mode
01X	PRPG mode
01X	PRPG mode
01X	PRPG mode

(3) Shift the C register output into the tester.

B_1 B_2 S_i	Action
00X	shift out C(3)
00X	shift out C(2)
00X	shift out C(1)
00X	shift out C(0)

(4) Return to the normal system mode.

B_1 B_2 S_i	Action
10X	normal mode

10.24

Operating Mode	B_1 B_0
normal	10
shift register	00
PRPG (LFSR)	01
MISR	11

State Sequence: 001, 100, 110, 111, 011, 101, 010, 001