SOLUTIONS MANUAL

DIGITAL DESIGN

FOURTH EDITION

M. MORRIS MANO
California State University, Los Angeles

MICHAEL D. CILETTI
University of Colorado, Colorado Springs

rev 01/21/2007

CHAPTER 1

1.1

Base-10:	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Octal:	20	21	22	23	24	25	26	27	30	31	32	33	34	35	36	37	40
Hex:	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20
Base-13	A	B	C	10	11	12	13	14	15	16	17	18	19	23	24	25	26

1.2 **(a)** 32,768 **(b)** 67,108,864 **(c)** 6,871,947,674

1.3 $(4310)_5 = 4 * 5^3 + 3 * 5^2 + 1 * 5^1 = 580_{10}$

$(198)_{12} = 1 * 12^2 + 9 * 12^1 + 8 * 12^0 = 260_{10}$

$(735)_8 = 7 * 8^2 + 3 * 8^1 + 5 * 8^0 = 477_{10}$

$(525)_6 = 5 * 6^2 + 2 * 6^1 + 5 * 6^0 = 197_{10}$

1.4 14-bit binary: 11_1111_1111_1111
Decimal: $2^{14}-1 = 16{,}383_{10}$
Hexadecimal: $3FFF_{16}$

1.5 Let b = base

(a) 14/2 = (b + 4)/2 = 5, so b = 6

(b) 54/4 = (5*b + 4)/4 = b + 3, so 5 * b = 52 – 4, and b = 8

(c) (2 *b + 4) + (b + 7) = 4b, so b = 11

1.6 $(x – 3)(x – 6) = x^2 –(6 + 3)x + 6*3 = x^2 -11x + 22$

Therefore: 6 + 3 = b + 1m so b = 8
Also, $6*3 = (18)_{10} = (22)_8$

1.7 $68BE = 0110\_1000\_1011\_1110 = 110\_100\_010\_111\_110 = (64276)_8$

1.8 **(a)** Results of repeated division by 2 (quotients are followed by remainders):

431_{10} = 215(1); 107(1); 53(1); 26(1); 13(0); 6(1) 3(0) 1(1)
Answer: $1111\_1010_2 = FA_{16}$

(b) Results of repeated division by 16:

431_{10} = 26(15); 1(10) (Faster)
Answer: FA = 1111_1010

1.9 **(a)** $10110.0101_2 = 16 + 4 + 2 + .25 + .0625 = 22.3125$

(b) $16.5_{16} = 16 + 6 + 5*(.0615) = 22.3125$

(c) $26.24_8 = 2 * 8 + 6 + 2/8 + 4/64 = 22.3125$

(d) $FAFA.B_{16} = 15*16^3 + 10*16^2 + 15*16 + 10 + 11/16 = 64{,}250.6875$

(e) $1010.1010_2 = 8 + 2 + .5 + .125 = 10.625$

1.10 **(a)** $1.10010_2 = 0001.1001_2 = 1.9_{16} = 1 + 9/16 = 1.563_{10}$

(b) $110.010_2 = 0110.0100_2 = 6.4_{16} = 6 + 4/16 = 6.25_{10}$

Reason: 110.010_2 is the same as 1.10010_2 shifted to the left by two places.

1.11

```
     1011.11
101 | 111011.0000
       101
       01001
        101
         1001
          101
          1000
           101
            0110
```

The quotient is carried to two decimal places, giving 1011.11
Checking: $111011_2 / 101_2 = 59_{10} / 5_{10} \cong 1011.11_2 = 58.75_{10}$

1.12 **(a)** 10000 and 110111

```
  1011          1011
 +101          x101
 -----         ----
 10000         1011
               1011
               ----
               110111
```

$= 16_{10}$ $= 55_{10}$

(b) 62_h and 958_h

```
 2E_h   0010_1110          2E_h
+34_h   0011_0100         x34_h
-----   ---------         -----
 62_h   0110_0010          B^3 8
                           8^2 A
                           -----
                           9 5 8_h
```

$= 98_{10}$ $= 2392_{10}$

1.13 **(a)** Convert 27.315 to binary:

	Integer Quotient		Remainder	Coefficient
27/2 =	13	+	½	$a_0 = 1$
13/2	6	+	½	$a_1 = 1$
6/2	3	+	0	$a_2 = 0$
3/2	1	+	½	$a_3 = 1$
½	0	+	½	$a_4 = 1$

$27_{10} = 11011_2$

		Integer		Fraction	Coefficient
.315 x 2	=	0	+	.630	$a_{-1} = 0$
.630 x 2	=	1	+	.26	$a_{-2} = 1$
.26 x 2	=	0	+	.52	$a_{-3} = 0$
.52 x 2	=	1	+	.04	$a_{-4} = 1$

$.315_{10} \cong .0101_2 = .25 + .0625 = .3125$

$27.315 \cong 11011.0101_2$

(b) 2/3 ≅ .6666666667

		Integer		Fraction	Coefficient
.6666_6666_67 x 2	=	1	+	.3333_3333_34	$a_{-1} = 1$
.3333333334 x 2	=	0	+	.6666666668	$a_{-2} = 0$
.6666666668 x 2	=	1	+	.3333333336	$a_{-3} = 1$
.3333333336 x 2	=	0	+	.6666666672	$a_{-4} = 0$
.6666666672 x 2	=	1	+	.3333333344	$a_{-5} = 1$
.3333333344 x 2	=	0	+	.6666666688	$a_{-6} = 0$
.6666666688 x 2	=	1	+	.3333333376	$a_{-7} = 1$
.3333333376 x 2	=	0	+	.6666666752	$a_{-8} = 0$

$.6666666667_{10} \cong .10101010_2 = .5 + .125 + .0313 + ..0078 = .6641_{10}$

$.101010102 = .1010\_1010_2 = .AA_{16} = 10/16 + 10/256 = .6641_{10}$ (Same as (b)).

1.14

(a) 1000_0000
1s comp: 0111_1111
2s comp: 1000_0000

(b) 0000_0000
1s comp: 1111_1111
2s comp: 0000_0000

(c) 1101_1010
1s comp: 0010_0101
2s comp: 0010_0110

(d) 0111_0110
1s comp: 1000_1001
2s comp: 1000_1010

(e) 1000_0101
1s comp: 0111_1010
2s comp: 0111_1011

(f) 1111_1111
1s comp: 0000_0000
2s comp: 0000_0001

1.15

(a) 52,784,630
9s comp: 47,215,369
10s comp: 47,215,370

(b) 63,325,600
9s comp: 36,674,399
10s comp: 36,674,400

(c) 25,000,000
9s comp: 74,999,999
10s comp: 75,000,000

(d) 00,000,000
9s comp: 99,999,999
10s comp: 00,000,000

1.16

B2FA
15s comp: 4D05
16s comp: 4D06

B2FA: 1011_0010_1111_1010
1s comp: 0100_1101_0000_0101
2s comp: 0100_1101_0000_0110 = 4D06

1.17 **(a)** 3409 → 03409 →96590 (9s comp) → 96591 (10s comp)
06428 – 03409 = 06428 + 96591 = 03019

(b) 1800 → 01800 → 98199 (9s comp) → 98200 (10 comp)
125 – 1800 = 00125 + 98200 = 98325 (negative)
Magnitude: 1675
Result: 125 – 1800 = 1675

(c) 6152 → 06152 → 93847 (9s comp) → 93848 (10s comp)
2043 – 6152 = 02043 + 93848 = 95891 (Negative)
Magnitude: 4109
Result: 2043 – 6152 = -4109

(d) 745 → 00745 → 99254 (9s comp) → 99255 (10s comp)
1631 -745 = 01631 + 99255 = 0886 (Positive)
Result: 1631 – 745 = 886

1.18 Note: Consider sign extension with 2s complement arithmetic.

(a)

	10001
1s comp:	01110
2s comp:	01111
	10011
Diff:	00010

(b)

	100011	
1s comp:	1011100	with sign extension
2s comp:	1011101	
	0100010	
	1111111	sign bit indicates that the result is negative
	0000001	2s complement
	-000001	result

(c)

	101000	
1s comp:	1010111	
2s comp:	1011000	
	001001	
Diff:	1100001	(negative)
	0011111	(2s comp)
	-011111	(diff is -31)

(d)

	10101	
1s comp:	1101010	with sign extension
2s comp:	1101011	
	110000	
	0011011	sign bit indicates that the result is positive
	Check: 48 -21 = 27	

1.19 +9286 → 009286; +801 → 000801; -9286 → 990714; -801 → 999199

(a) (+9286) + (_801) = 009286 + 000801 = 010087

(b) (+9286) + (-801) = 009286 + 999199 = 008485

(c) (-9286) + (+801) = 990714 + 000801 = 991515

(d) (-9286) + (-801) = 990714 + 999199 = 989913

1.20 +49 → 0_110001 (Needs leading zero indicate + value); +29 → 0_011101 (Leading 0 indicates + value)
-49 → 1_001111; -29 → 1_100011

(a) (+29) + (-49) = 0_011101 + 1_001111 = 1_101100 (1 indicates negative value.)
Magnitude = 0_010100; Result (+29) + (-49) = -20

(b) (-29) + (+49) = 1_100011 + 0_110001 = 0_010100 (0 indicates positive value)
(-29) + (+49) = +20

(c) Must increase word size by 1 (sign extension) to accomodate overflow of values:
(-29) + (-49) = 11_100011 + 11_001111 = 10_110010 (1 indicates negative result)
Magnitude: 1_001110 = 78_{10}
Result: (-29) + (-49) = -78

1.21 +9742 → 009742 → 990257 (9's comp) → 990258 (10s) comp
+641 → 000641 → 999358 (9's comp) → 999359 (10s) comp

(a) (+9742) + (+641) → 010383

(b) (+9742) + (-641) →009742 + 999359 = 009102
Result: (+9742) + (-641) = 9102

(c) -9742) + (+641) = 990258 + 000641 = 990899 (negative)
Magnitude: 009101
Result: (-9742) + (641) = -9101

(d) (-9742) + (-641) = 990258 + 999359 = 989617 (Negative)
Magnitude: 10383
Result: (-9742) + (-641) = -10383

1.22 8,723
BCD**:** 1000_0111_0010_0011
ASCII: 0_011_1000_011_0111_011_0010_011_0001

1.23

```
      1000  0100  0010 (  842)
      0101  0011  0111 (+537)
      1101  0111  1001
      0110
 0001 0011  0111  0101 (1,379)
```

1.24 **(a)**

6	3	1	1	Decimal
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	1	0	0	3
0	1	1	0	4 (*or* 0101)
0	1	1	1	5
1	0	0	0	6
1	0	1	0	7 (*or* 1001)
1	0	1	1	8
1	1	0	0	9

(b)

6	4	2	1	Decimal
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
1	0	0	0	6 (*or* 0110)
1	0	0	1	7
1	0	1	0	8
1	0	1	1	9

1.25

(a) $5{,}137_{10}$	BCD:	0101_0011_0111
(b)	Excess-3:	1000_0100_0110_1010
(c)	2421:	1011_0001_0011_0111
(d)	6311:	0111_0001_0100_1001

1.26 5,137 9s Comp: 4,862
2421 code: 0100_1110_1100_1000
1s comp: 1011_0001_0011_0111 same as (c) in 1.25

1.27 For a deck with 52 cards, we need 6 bits (32 < 52 < 64). Let the msb's select the suit (e.g., diamonds, hearts, clubs, spades are encoded respectively as 00, 01, 10, and 11. The remaining four bits select the "number" of the card. Example: 0001 (ace) through 1011 (9), plus 101 through 1100 (jack, queen, king). This a jack of spades might be coded as 11_1010. (Note: only 52 out of 64 patterns are used.)

1.28 G (dot) (space) B o o l e
01000111_11101111_01101000_01101110_00100000_11000100_11101111_11100101

1.29 Bill Gates

1.30 73 F4 E5 76 E5 4A EF 62 73

73: 0_111_0011 s
F4: 1_111_0100 t
E5: 1_110_0101 e
76: 0_111_0110 v
E5: 1_110_0101 e
4A: 0_100_1010 j
EF: 1_110_1111 o
62: 0_110_0010 b
73: 0_111_0011 s

1.31 62 + 32 = 94 printing characters

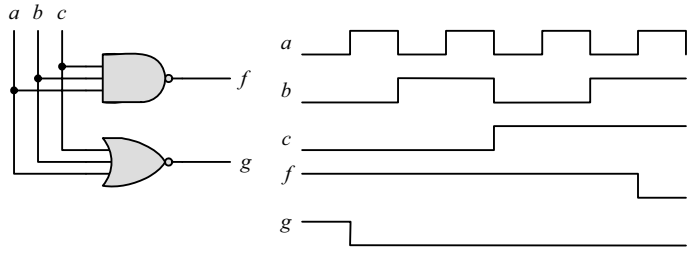
1.32 bit 6 from the right

1.33 **(a)** 897 **(b)** 564 **(c)** 871 **(d)** 2,199

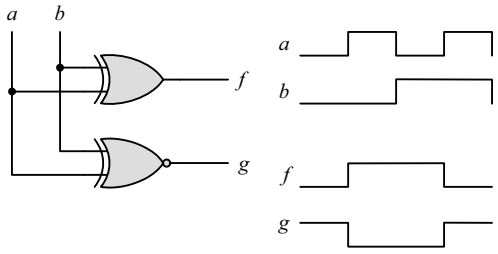
1.34 ASCII for decimal digits with odd parity:

(0):	10110000	(1):	00110001	(2):	00110010	(3):	10110011
(4):	00110100	(5):	10110101	(6):	10110110	(7):	00110111
(8):	00111000	(9):	10111001				

1.35 **(a)**



1.36



CHAPTER 2

2.1 **(a)**

x y z	$x+y+z$	$(x+y+z)'$	x'	y'	z'	$x'y'z'$
0 0 0	0	1	1	1	1	1
0 0 1	1	0	1	1	0	0
0 1 0	1	0	1	0	1	0
0 1 1	1	0	1	0	0	0
1 0 0	1	0	0	1	1	0
1 0 1	1	0	0	1	0	0
1 1 0	1	0	0	0	1	0
1 1 1	1	0	0	0	0	0

$x\ y\ z$	(xyz)	$(xyz)'$	x'	y'	z'	$x'+y'+z'$
0 0 0	0	1	1	1	1	1
0 0 1	0	1	1	1	0	1
0 1 0	0	1	1	0	1	1
0 1 1	0	1	1	0	0	1
1 0 0	0	1	0	1	1	1
1 0 1	0	1	0	1	0	1
1 1 0	0	1	0	0	1	1
1 1 1	1	0	0	0	0	0

(b)

$x\ y\ z$	$x+yz$	$(x+y)$	$(x+z)$	$(x+y)(x+z)$
0 0 0	0	0	0	0
0 0 1	0	0	1	0
0 1 0	0	1	0	0
0 1 1	1	1	1	1
1 0 0	1	1	1	1
1 0 1	1	1	1	1
1 1 0	1	1	1	1
1 1 1	1	1	1	1

(c)

$x\ y\ z$	$x(y+z)$	xy	xz	$xy+xz$
0 0 0	0	0	0	0
0 0 1	0	0	0	0
0 1 0	0	0	0	0
0 1 1	0	0	0	0
1 0 0	0	0	0	0
1 0 1	1	0	1	1
1 1 0	1	1	0	1
1 1 1	1	1	1	1

(c)

$x\ y\ z$	x	$y+z$	$x+(y+z)$	$(x+y)$	$(x+y)+z$
0 0 0	0	0	0	0	0
0 0 1	0	1	1	0	1
0 1 0	0	1	1	1	1
0 1 1	0	1	1	1	1
1 0 0	1	0	1	1	1
1 0 1	1	1	1	1	1
1 1 0	1	1	1	1	1
1 1 1	1	1	1	1	1

(d)

$x\ y\ z$	yz	$x(yz)$	xy	$(xy)z$
0 0 0	0	0	0	0
0 0 1	0	0	0	0
0 1 0	0	0	0	0
0 1 1	1	0	0	0
1 0 0	0	0	0	0
1 0 1	0	0	0	0
1 1 0	0	0	1	0
1 1 1	1	1	1	1

2.2 **(a)** $xy + xy' = x(y + y') = x$

(b) $(x + y)(x + y') = x + yy' = x(x + y') + y(x + y') = xx + xy' + xy + yy' = x$

(c) $xyz + x'y + xyz' = xy(z + z') + x'y = xy + x'y = y$

(d) $(A + B)'(A' + B') = (A'B')(A\ B) = (A'B')(BA) = A'(B'BA) = 0$

(e) $xyz' + x'yz + xyz + x'yz' = xy(z + z') + x'y(z + z') = xy + x'y = y$

(f) $(x + y + z')(x' + y' + z) = xx' + xy' + xz + x'y + yy' + yz + x'z' + y'z' + zz' =$
$= xy' + xz + x'y + yz + x'z' + y'z' = x \oplus y + (x \oplus z)' + (y \oplus z)'$

2.3 **(a)** $ABC + A'B + ABC' = AB + A'B = B$

(b) x'yz + xz = (x'y + x)z = z(x + x')(x + y) = z(x + y)

(c) $(x + y)'(x' + y') = x'y'(x' + y') = x'y'$

(d) $xy + x(wz + wz') = x(y + wz + wz') = x(w + y)$

(e) $(BC' + A'D)(AB' + CD') = BC'AB' + BC'CD' + A'DAB' + A'DCD' = 0$

(f) $(x + y' + z')(x' + z') = xx' + xz' + x'y' + y'z' + x'z' + z'z' = z' + y'(x' + z') = z' + x'y'$

2.4 **(a)** $A'C' + ABC + AC' = C' + ABC = (C + C')(C' + AB) = AB + C'$

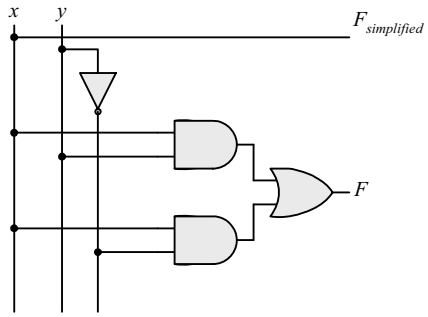
(b) $(x'y' + z)' + z + xy + wz = (x'y')'z' + z + xy + wz = [(x + y)z' + z] + xy + wz =$
$= (z + z')(z + x + y) + xy + wz = z + wz + x + xy + y = z(1 + w) + x(1 + y) + y = x + y + z$

(c) $A'B(D' + C'D) + B(A + A'CD) = B(A'D' + A'C'D + A + A'CD)$
$= B(A'D' + A + A'D(C + C') = B(A + A'(D' + D)) = B(A + A') = B$

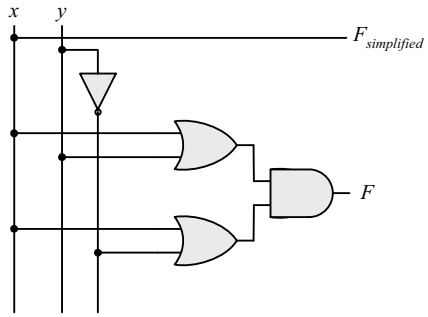
(d) $(A' + C)(A' + C')(A + B + C'D) = (A' + CC')(A + B + C'D) = A'(A + B + C'D)$
$= AA' + A'B + A'C'D = A'(B + C'D)$

(e) $ABCD + A'BD + ABC'D = ABD + A'BD = BD$

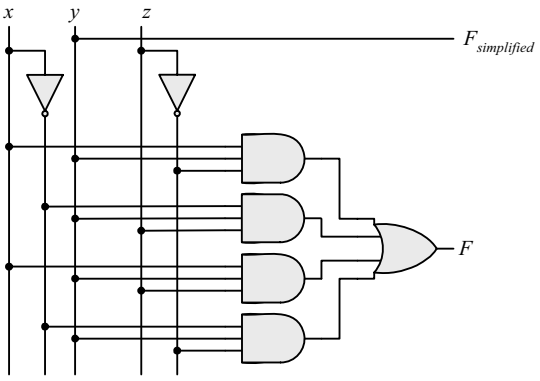
2.5 **(a)**



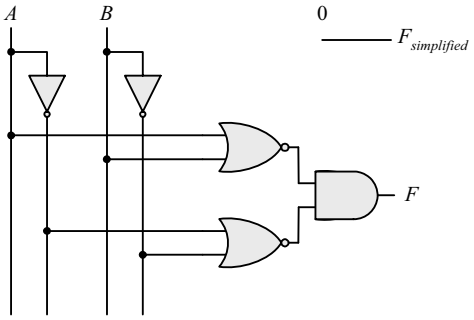
(b)



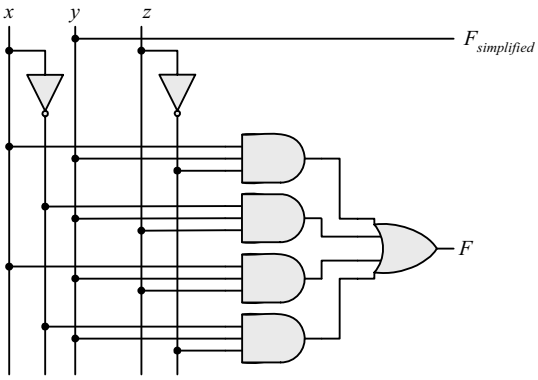
(c)



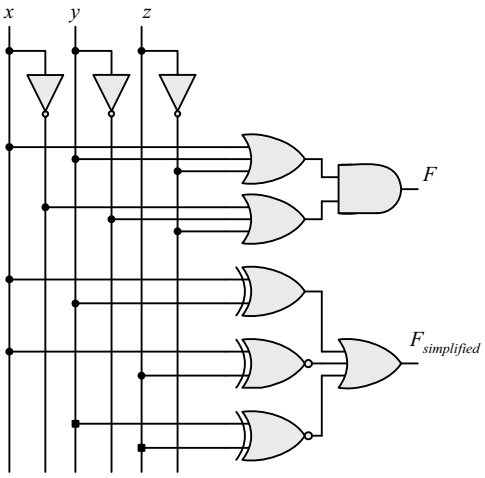
(d)



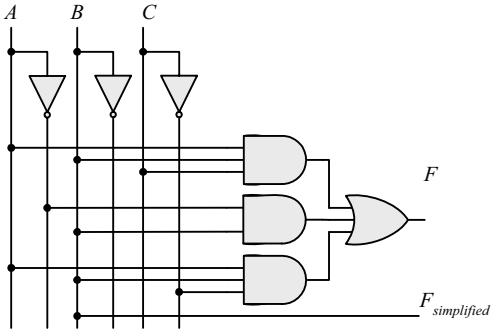
(e)



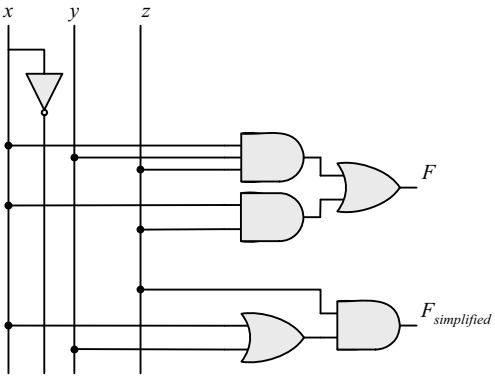
(f)



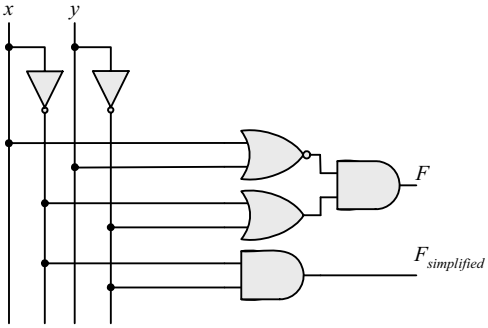
2.6 **(a)**



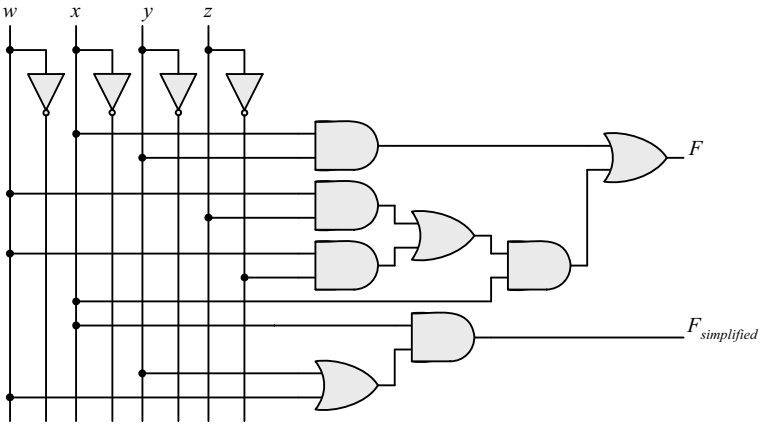
(b)



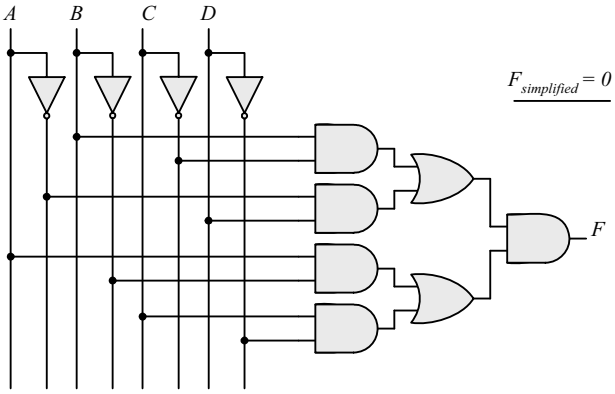
(c)



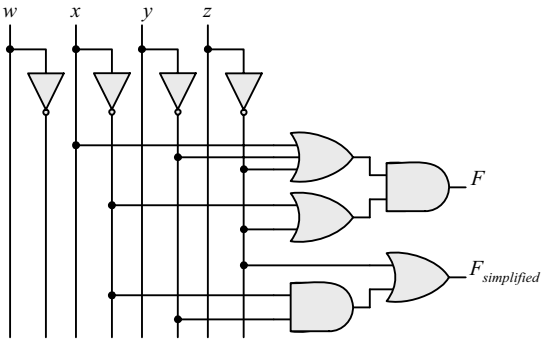
(d)



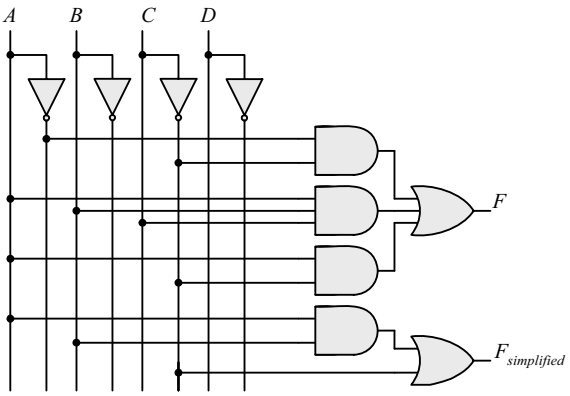
(e)



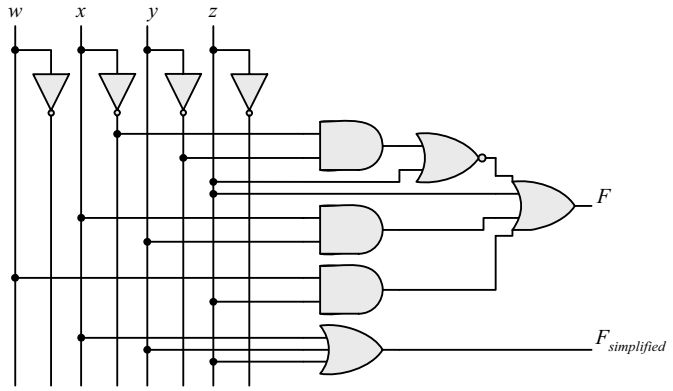
(f)



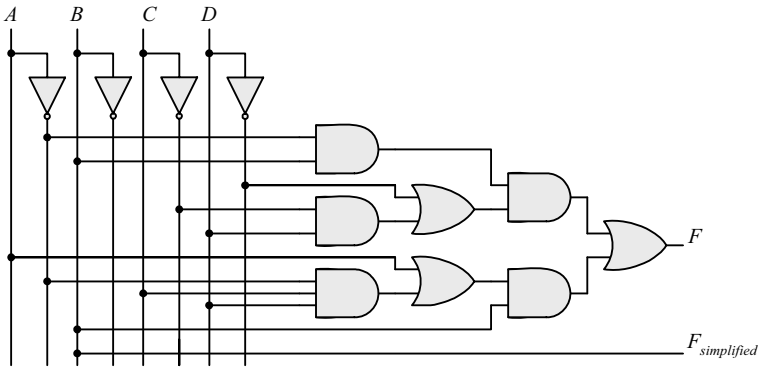
2.7 **(a)**



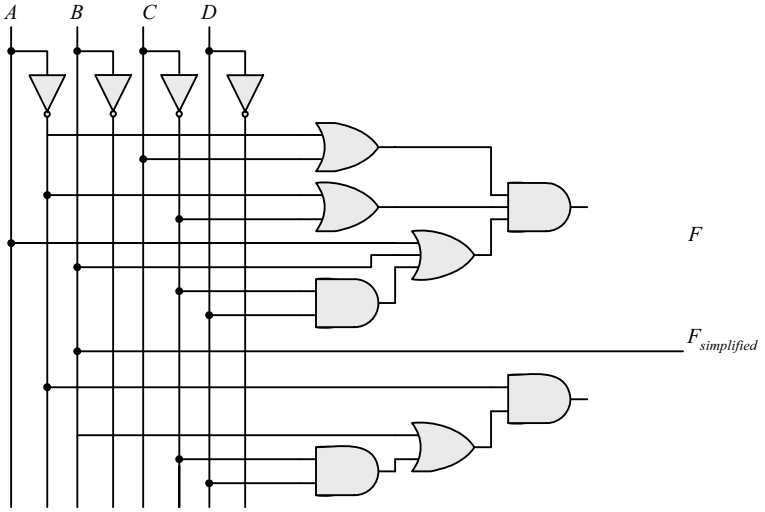
(b)



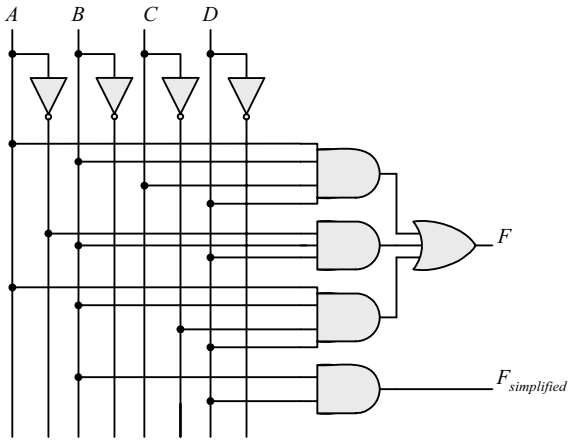
(c)



(d)



(e)



2.8 $F' = (wx + yz)' = (wx)'(yz)' = (w' + x')(y' + z')$

$FF' = wx(w' + x')(y' + z') + yz(w' + x')(y' + z') = 0$
$F + F' = wx + yz + (wx + yz)' = A + A' = 1$ *with* $A = wx + yz$

2.9 **(a)** $F' = (xy' + x'y)' = (xy')'(x'y)' = (x' + y)(x + y') = xy + x'y'$

(b) $F' = [(A'B + CD)E' + E]' = [(A'B + CD) + E]' = (A'B + CD)'E' = (A'B)'(CD)'E'$
$F' = (A + B')(C' + D')E' = AC'E' + A\,D'E' + B'C'E' + B'D'E'$

(c) $F' = [(x' + y + z')(x + y')(x + z)]' = (x' + y + z')' + (x + y')' + (x + z)' =$
$F' = xy'z + x'y + x'z'$

2.10 **(a)** $F_1 + F_2 = \Sigma\, m_{1i} + \Sigma m_{2i} = \Sigma\, (m_{1i} + m_{2i})$

(b) $F1\ F2 = \Sigma\, m_i\, \Sigma m_j$ where $m_i\, m_j = 0$ if $i \neq j$ and $m_i\, m_j = 1$ if $i = j$

2.11 **(a)** $F(x, y, z) = \Sigma(1, 4, 5, 6, 7)$

(b) $F(x, y, z) = \Sigma(0, 2, 3, 7)$

$F = xy + xy' + y'z$

x y z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

$F = x'z' + yz$

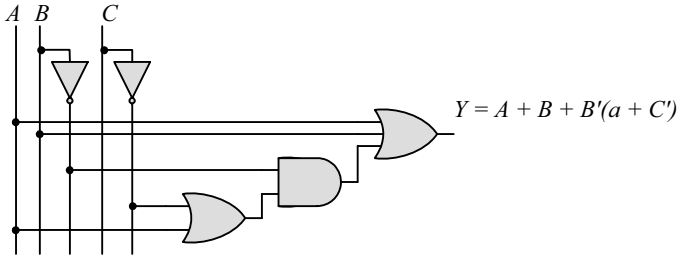
x y z	F
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	0
1 1 1	1

2.12 *A = 1011_0001*
B = 1010_1100

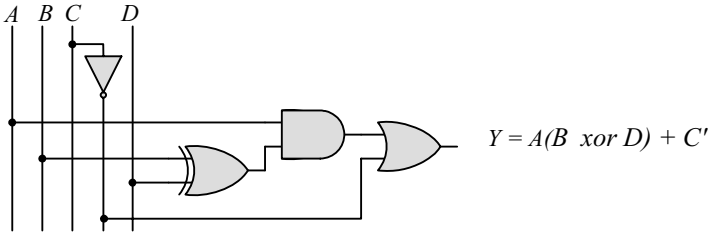
(a) *A AND B = 1010_0000*
(b) *A OR B = 1011_1101*
(c) *A XOR B = 0001_1101*

(d) *NOT A = 0100_1110*
(e) *NOT B = 0101_0011*

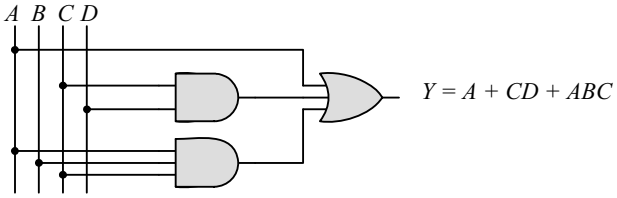
2.13 **(a)**



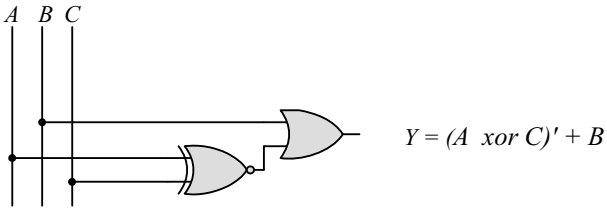
(b)



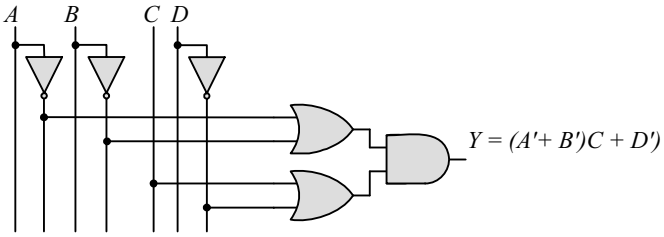
(c)



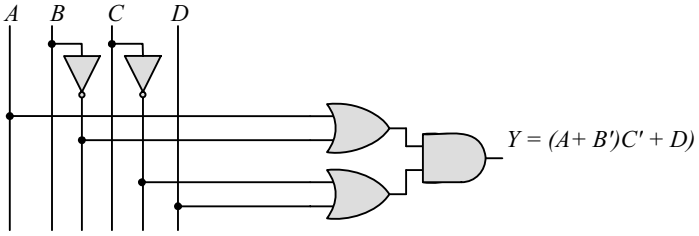
(d)



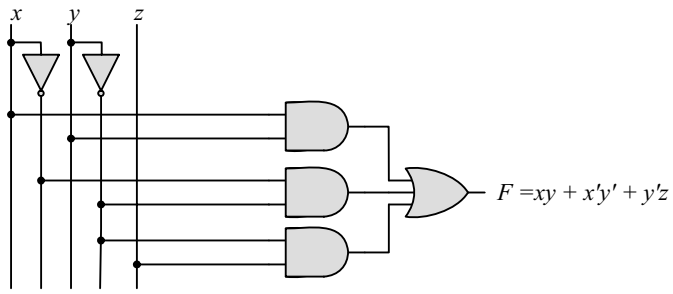
(e)



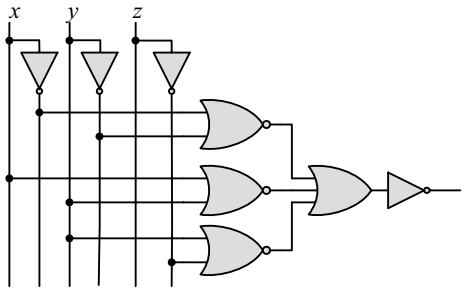
(f)



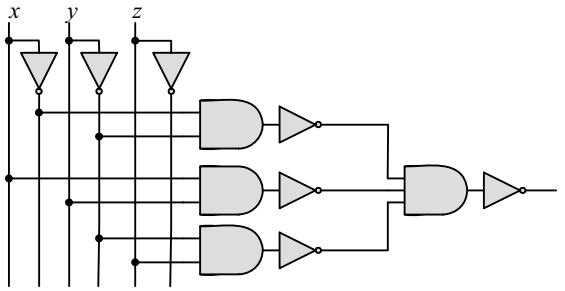
2.14 **(a)**



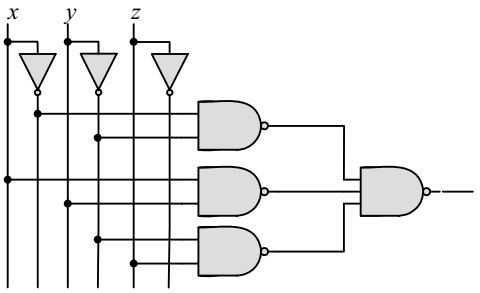
(b)



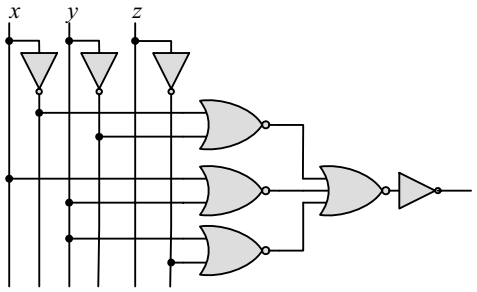
(c)



(d)



(e)



F = xy + x'y' + y'z

= (x′ + y′)' + (x + y)' + (y + z′)'

2.15 **(a)** $T_1 = A'B'C' + A'B'C + A'BC' = A'B'(C' + C) + A'C'(B' + B) = A'B' + A'C' = A'(B' + C')$

(b) $T_2 = T_1' = A'BC + AB'C' + AB'C + ABC' + ABC$
$= BC(A' + A) + AB'(C' + C) + AB(C' + C)$
$= BC + AB' + AB = BC + A(B' + B) = A + BC$

$\sum(3, 5, 6, 7) = \Pi(0, 1, 2, 4)$

$T_1 = A'B'C' + A'B'C + A'BC'$

$A'B'$ $A'C'$

$T_1 = A'B'\ A'C' = A'(B' + C')$

$T_2 = A'BC + AB'C' + AB'C + ABC' + ABC$

AC' AC

BC

$T_2 = AC' + BC + AC = A + BC$

2.16 **(a)** $F(A, B, C) = A'B'C' + A'B'C + A'BC' + A'BC + AB'C' + AB'C + ABC' + ABC$
$= A'(B'C' + B'C + BC' + BC) + A((B'C' + B'C + BC' + BC)$
$= (A' + A)(B'C' + B'C + BC' + BC) = B'C' + B'C + BC' + BC$
$= B'(C' + C) + B(C' + C) = B' + B = 1$

(b) $F(x_1, x_2, x_3, ..., x_n) = \Sigma m_i$ has $2^n/2$ minterms with x_1 and $2^n/2$ minterms with x'_1, which can be factored and removed as in (a). The remaining 2^{n-1} product terms will have $2^{n-1}/2$ minterms with x_2 and $2^{n-1}/2$ minterms with x'_2, which and be factored to remove x_2 and x'_2. continue this process until the last term is left and $x_n + x'_n = 1$. Alternatively, by induction, F can be written as $F = x_nG + x'_nG$ with $G = 1$. So $F = (x_n + x'_n)G = 1$.

2.17 **(a)** $(xy + z)(y + xz) = xy + yz + xyz + xz = \Sigma(3, 5, 6, 7) = \Pi(0, 1, 2, 4)$

(b) $(A' + B)(B' + C) = A'B' + A'C + BC = \Sigma(0, 1, 3, 7) = \Pi(2, 4, 5, 6)$

(c) $y'z + wxy' + wxz' + w'x'z = \Sigma(1, 3, 5, 9, 12, 13, 14) = \Pi(0, 2, 4, 6, 7, 8, 10, 11, 15)$

(d) $(xy + yz' + x'z)(x + z) = xy + xyz' + xyz + x'z$
$= \Sigma(1, 3, 9, 11, 14, 15) = \Pi(0, 2, 4, 5, 6, 7, 8, 10, 12, 13)$

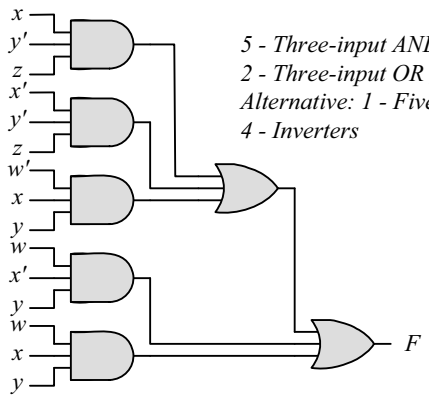
2.18 (a)

wx y z	F
00 0 0	0
00 0 1	1
00 1 0	0
00 1 1	0
01 0 0	0
01 0 1	1
01 1 0	1
01 1 1	1
10 0 0	0
10 0 1	1
10 1 0	1
10 1 1	1
11 0 0	0
11 0 1	1
11 1 0	1
11 1 1	1

$F = xy'z + x'y'z + w'xy + wx'y + wxy$

$F = \Sigma(1, 5, 6, 7, 9, 10\ 11, 13, 14, 15)$

(b)

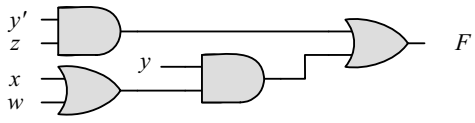


5 - Three-input AND gates
2 - Three-input OR gates
Alternative: 1 - Five-input OR gate
4 - Inverters

(c) $F = xy'z + x'y'z + w'xy + wx'y + wxy = y'z + xy + wy = y'z + y(w + x)$

(d) $F = y'z + yw + yx) = \Sigma($1, 5, 9, 13 , 10, 11, 13, 15, 6, 7, 14, 15)
$= \Sigma(1, 5, 6, 7, 9, 10, 11, 13, 14, 15)$

(e)



1 – Inverter, 2 – Two-input AND gates, 2 – Two-input OR gates

2.19 $F = B'D + A'D + BD$

ABCD	*ABCD*	*ABCD*
-B'-D	*A'--D*	*-B-D*
0001 = 1	*0001 = 1*	*0101 = 5*
0011 = 3	*0011 = 3*	*0111 = 7*
1001 = 9	*0101 = 5*	*1101 = 13*
1011 = 11	*0111 = 7*	*1111 = 15*

$F = \Sigma(1, 3, 5, 7, 9, 11,13, 15) = \Pi(0, 2, 4, 6, 8, 10, 12, 14)$

2.20 **(a)** $F(A, B, C, D) = \Sigma(3, 5, 9, 11, 15)$
$F'(A, B, C, D) = \Sigma(0, 1, 2, 4, 6, 7, 8, 10, 12, 13, 14)$

(b) $F(x, y, z) = \Pi(2, 4, 5, 7)$
$F' = \Sigma(2, 4, 5, 7)$

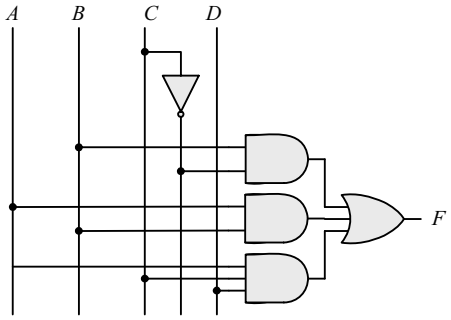
2.21 ***(a)*** $F(x, y, z) = \Sigma(2, 5, 6) = \Pi(0, 1, 3, 4, 7)$

(b) $F(A, B, C, D) = \Pi(0, 1, 2, 4, 7, 9, 12) = \Sigma(3, 5, 6, 8, 10, 11, 13, 14, 15)$

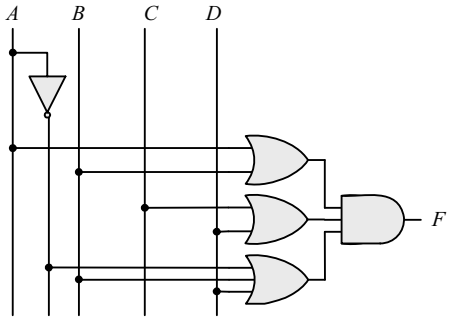
2.22 **(a)** $(AB + C)(B + C'D) = AB + BC + ABC'D + CC'D = AB(1 + C'D) + BC$
$= AB + BC$ (SOP form)
$= B(A + C)$ (POS form)

(b) $x' + x(x + y')(y + z') = (x' + x)[x' + (x + y')(y + z')] =$
$= (x' + x + y')(x' + y + z')$
$= x' + y + z'$

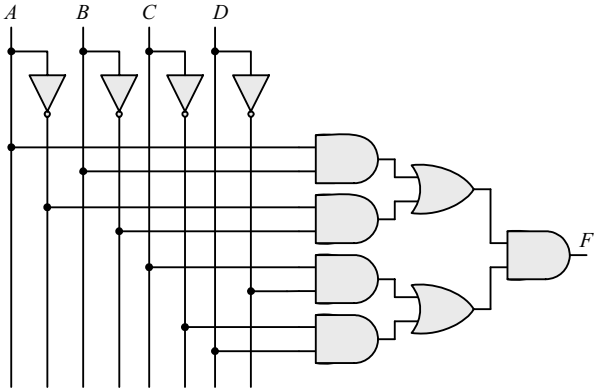
2.23 **(a)** $B'C + AB + ACD$



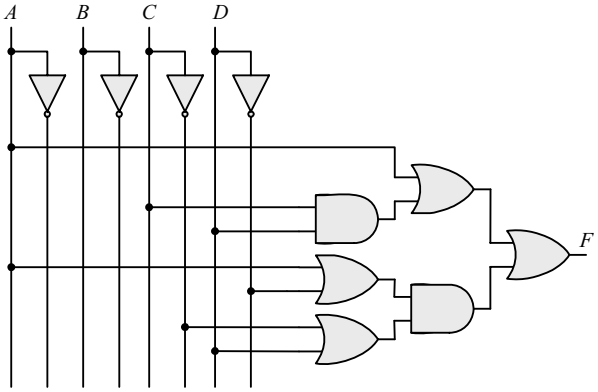
(b) $(A + B)(C + D)(A' + B + D)$



(c) $(AB + A'B')(CD' + C'D)$



(d) $A + CD + (A + D')(C' + D)$



2.24 $x \oplus y = x'y + xy'$ *and* $(x \oplus y)' = (x + y')(x' + y)$

Dual of $x'y + xy' = (x' + y)(x + y') = (x \oplus y)'$

2.25 **(a)** $x \mid y = xy' \neq y \mid x = x'y$ Not commutative

$(x \mid y) \mid z = xy'z' \neq x \mid (y \mid z) = x(yz')' = xy' + xz$ Not associative

(b) $(x \oplus y) = xy' + x'y = y \oplus x = yx' + y'x$ Commutative

$(x \oplus y) \oplus z = \sum(1, 2, 4, 7) = x \oplus (y \oplus z)$ Associative

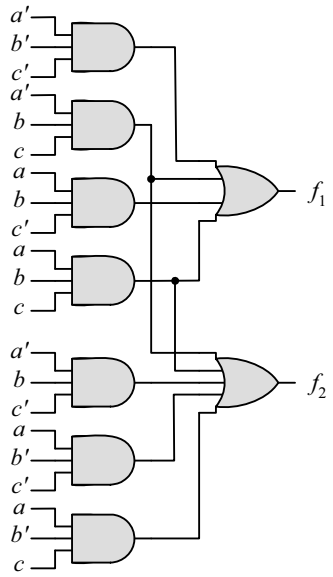
2.26

Gate		NAND (Positive logic)		NOR (Negative logic)	
x y	z	x y	z	x y	z
L L	H	0 0	1	1 1	0
L H	H	0 1	1	1 0	0
H L	H	1 0	1	0 1	0
H H	L	1 1	0	0 0	1

Gate		NOR (Positive logic)		NAND (Negative logic)	
x y	z	x y	z	x y	z
L L	H	0 0	1	1 1	0
L H	L	0 1	0	1 0	1
H L	L	1 0	0	0 1	1
H H	L	1 1	0	0 0	1

2.27 $f_1 = a'b'c + a'bc + abc' + abc$

$f_2 = a'bc' + a'bc + ab'c' + ab'c + abc'$



2.28 **(a)** $y = a(bcd)'e = a(b' + c' + d')e$

$y = a(b' + c' + d')e = ab'e + ac'e + ad'e$
$= \Sigma(17, 19, 21, 23, 25, 27, 29)$

a bcde	y	a bcde	y
0 0000	0	1 0000	0
0 0001	0	**1 0001**	1
0 0010	0	1 0010	0
0 0011	0	**1 0011**	1
0 0100	0	1 0100	0
0 0101	0	**1 0101**	1
0 0110	0	1 0110	0
0 0111	0	**1 0111**	1
	0		0
0 1000	0	1 1000	0
0 1001	0	**1 1001**	1
0 1010	0	1 1010	0
0 1011	0	**1 1011**	1
0 1100	0	1 1100	0
0 1101	0	**1 1101**	1
0 1110	0	1 1110	0
0 1111	0	1 1111	0

(b) $y_1 = a \oplus (c + d + e) = a'(c + d + e) + a(c'd'e') = a'c + a'd + a'e + ac'd'e'$

$y_2 = b'(c + d + e)f = b'cf + b'df + b'ef$

$y_1 = a\ (c + d + e) = a'(c + d + e) + a(c'd'e') = a'c + a'd + a'e + ac'd'e'$

$y_2 = b'(c + d + e)f = b'cf + b'df + b'ef$

a'-c---	*a'--d--*	*a'---e-*	*a-c'd'e'-*
001000 = 8	000100 = 8	000010 = 2	100000 = 32
001001 = 9	000101 = 9	000011 = 3	100001 = 33
001010 = 10	000110 = 10	000110 = 6	110000 = 34
001011 = 11	000111 = 11	000111 = 7	110001 = 35
001100 = 12	001100 = 12	001010 = 10	
001101 = 13	001101 = 13	001011 = 11	
001110 = 14	001110 = 14	001110 = 14	
001111 = 15	001111 = 15	001111 = 15	
011000 = 24	010100 = 20	010010 = 18	
011001 = 25	010101 = 21	010011 = 19	
011010 = 26	010110 = 22	010110 = 22	
011011 = 27	010111 = 23	010111 = 23	
011100 = 28	011100 = 28	011010 = 26	
011101 = 29	011101 = 29	011001 = 27	
011110 = 30	011110 = 30	011110 = 30	
011111 = 31	011111 = 31	011111 = 31	

-b' c--f	-b' -d-f	-b' --ef
001001 = 9	001001 = 9	000011 = 3
001011 = 11	001011 = 11	000111 = 7
001101 = 13	001101 = 13	001011 = 11
001111 = 15	001111 = 15	001111 = 15
101001 = 41	101001 = 41	100011 = 35
101011 = 43	101011 = 43	100111 = 39
101101 = 45	101101 = 45	101011 = 51
101111 = 47	101111 = 47	101111 = 55

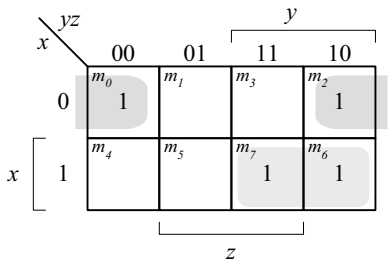
$y_1 = \Sigma$ (2, 3, 6, 7, 8, 9, 10 ,11, 12, 13, 14, 15, 18, 19, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35)

$y_2 = \Sigma$ (3, 7, 9, 13, 15, 35, 39, 41, 43, 45, 47, 51, 55)

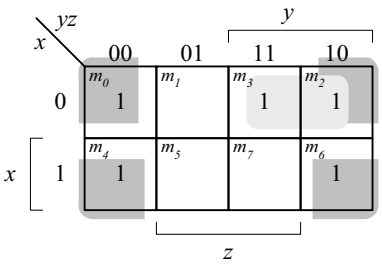
ab cdef	y_1 y_2	*ab cdef*	y_1 y_2	*ab cdef*	y_1 y_2	*ab cdef*	y_1 y_2
00 0000	0 0	01 0000	0 0	10 0000	1 0	11 0000	0 0
00 0001	0 0	01 0001	0 0	10 0001	1 0	11 0001	0 0
00 0010	1 0	01 0010	1 0	10 0010	1 0	11 0010	0 0
00 0011	1 1	01 0011	1 0	10 0011	1 1	11 0011	0 1
00 0100	0 0	01 0100	0 0	10 0100	0 0	11 0100	0 0
00 0101	0 0	01 0101	0 0	10 0101	0 0	11 0101	0 0
00 0110	1 0	01 0110	1 0	10 0110	0 0	11 0110	0 0
00 0111	1 1	01 0111	1 0	10 0111	0 1	11 0111	0 1
00 1000	1 0	01 1000	1 0	10 1000	0 0	11 1000	0 0
00 1001	1 1	01 1001	1 0	10 1001	0 1	11 1001	0 0
00 1010	1 0	01 1010	1 0	10 1010	0 0	11 1010	0 0
00 1011	1 0	01 1011	1 0	10 1011	0 1	11 1011	0 0
00 1100	1 0	01 1100	1 0	10 1100	0 0	11 1100	0 0
00 1101	1 1	01 1101	1 0	10 1101	0 1	11 1101	0 0
00 1110	1 0	01 1110	1 0	10 1110	0 0	11 1110	0 0
00 1111	1 1	01 1111	1 0	10 1111	0 1	11 1111	0 0

Chapter 3

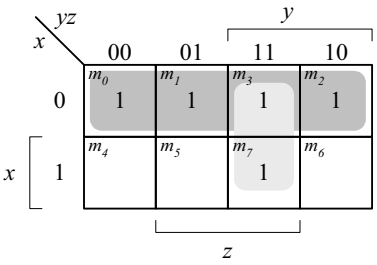
3.1



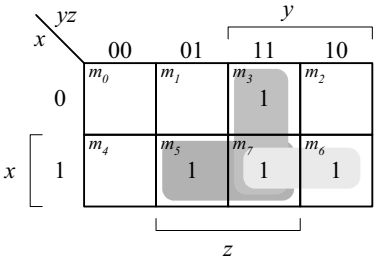
(a) $F = xy + x'z'$



(b) $F = z' + x'y$

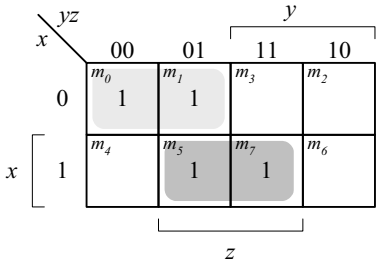


(c) $F = x' + yz$

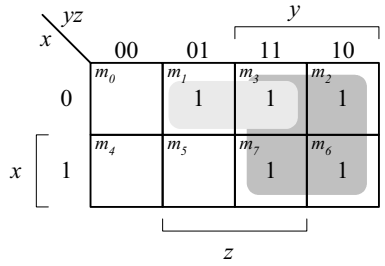


(d) $F = xy + xz + yz$

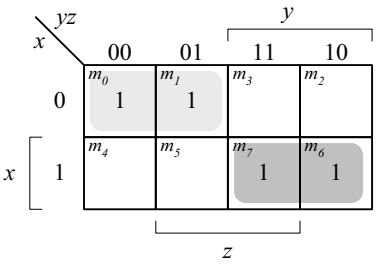
3.2



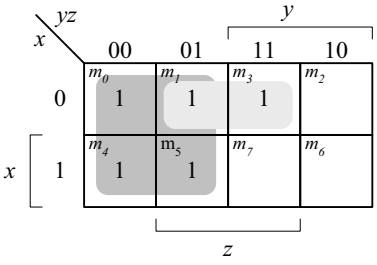
(a) $F = x'y' + xz$



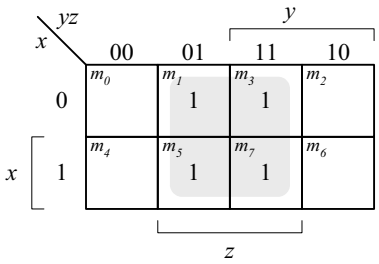
(b) $F = y + x'z$



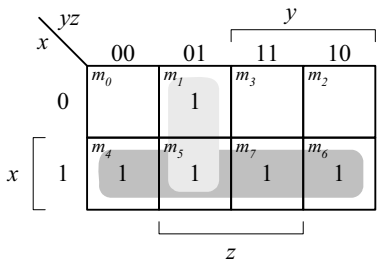
(c) $F = x'y' + xy$



(d) $F = y' + x'z$

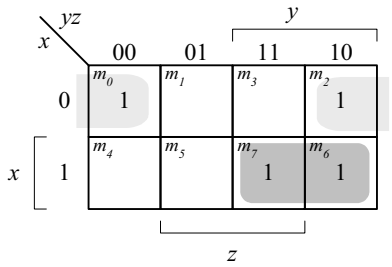


(e) $F = z$

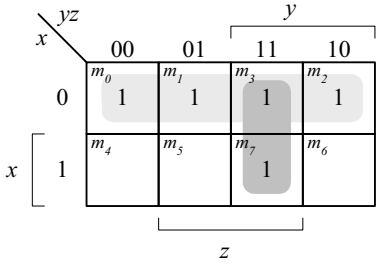


(f) $F = x + y'z$

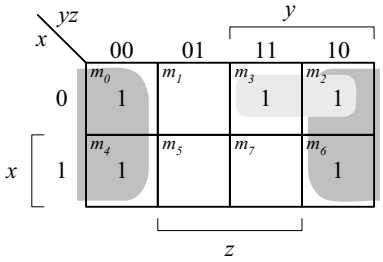
3.3



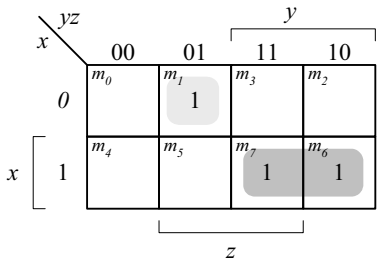
(a) $F = xy + x'y'z' + x'yz'$
$F = xy + x'z'$



(b) $F = x'y' + yz + x'yz'$
$F = x' + yz$

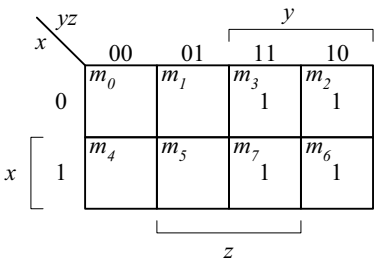


(c) $F = x'y + yz' + y'z'$
$F = = x'y + z'$

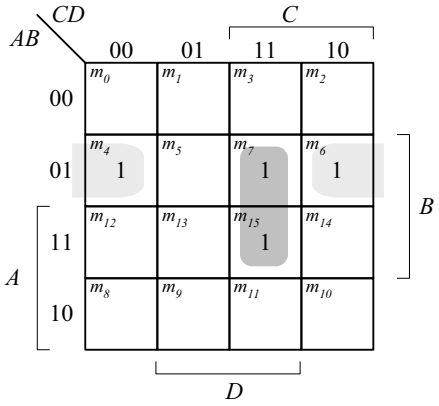


(d) $F = xyz + x'y'z + xyz'$
$F = x'y'z + xy$

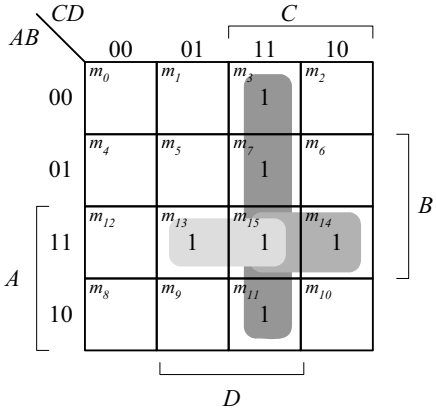
3.4



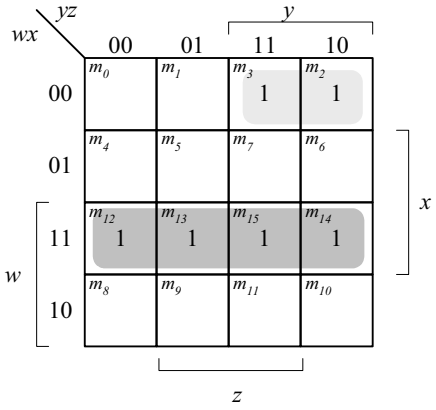
(a) $F = y$



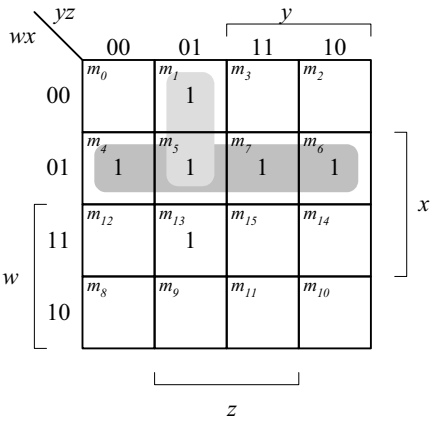
(b) $F = BCD + A'BD'$



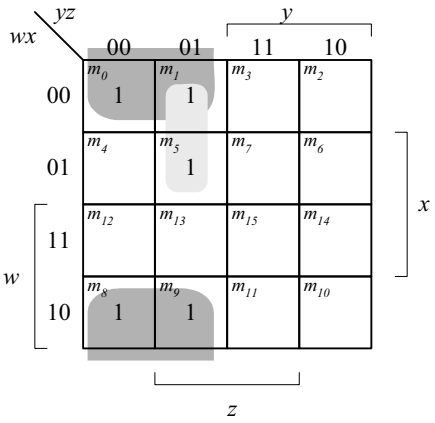
(c) $F = CD + ABD + ABC$



(d) $F = w'x'y + wx$

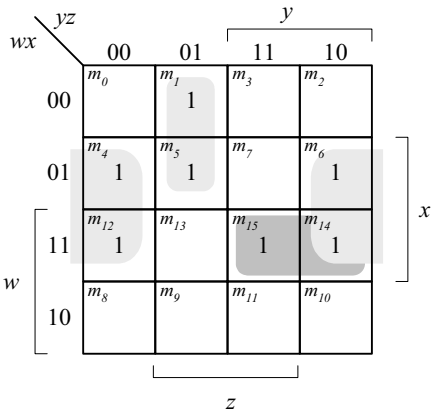


(e) $F = w'x + w'y'z$

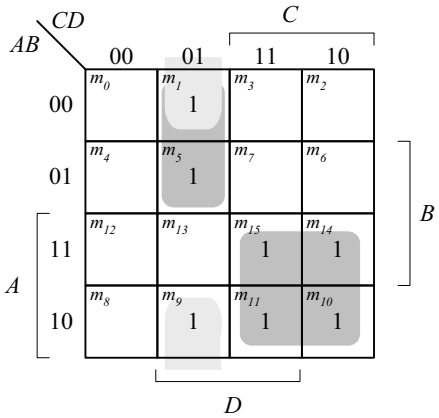


(f) $F = x'y' + w'y'z$

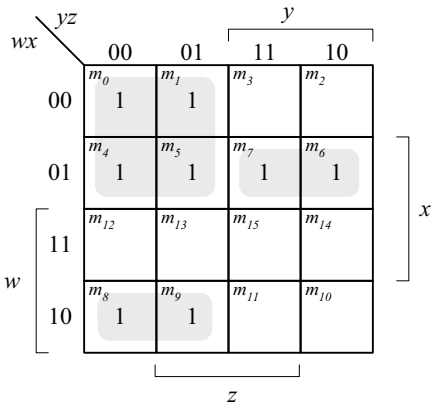
3.5



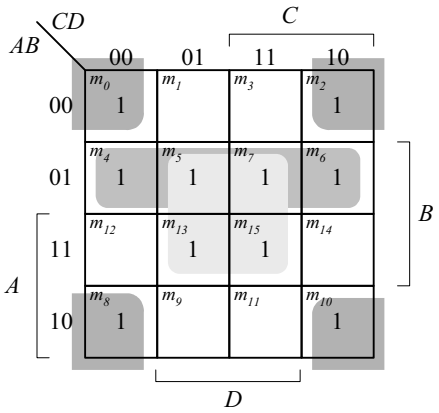
(a) $F = xz' + w'y'z + wxy$



(b) $F = A'C + A'C'D + B'C'D$

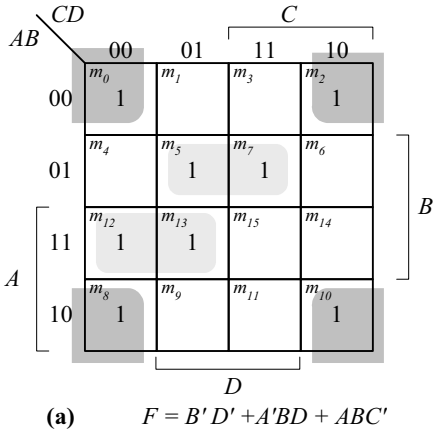


(c) $F = w'y' + wx'y' + w'xy$

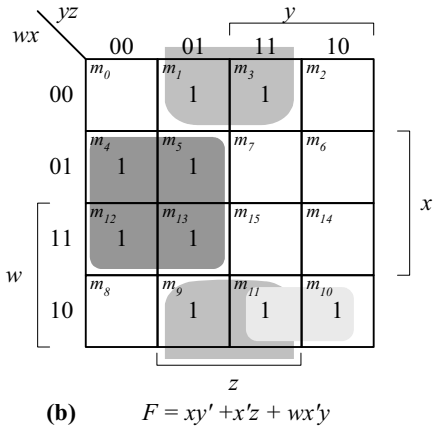


(d) $F = BD + A'B + B'D'$
$or = BD + B'D' + A'D'$

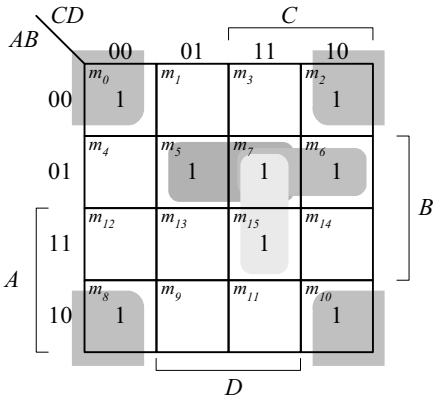
3.6



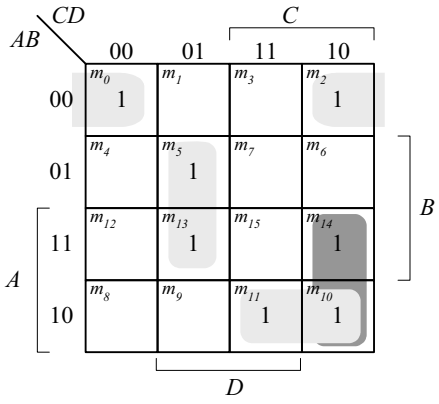
(a) $F = B'D' + A'BD + ABC'$



(b) $F = xy' + x'z + wx'y$

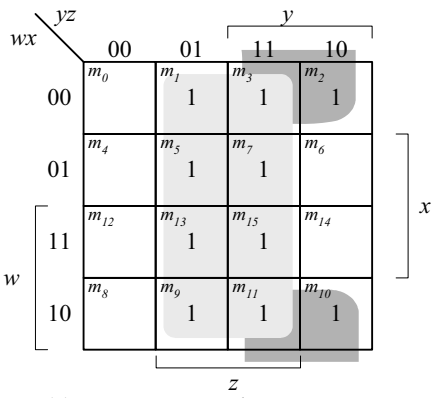


(c) $F = B'D' + BCD + A'BD + A'BC$

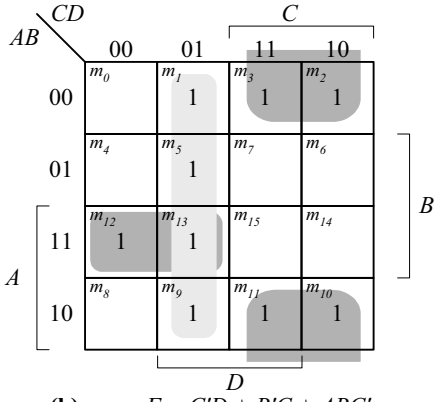


(d) $F = A'B'D' + BC'D + ACD' + AB'C$

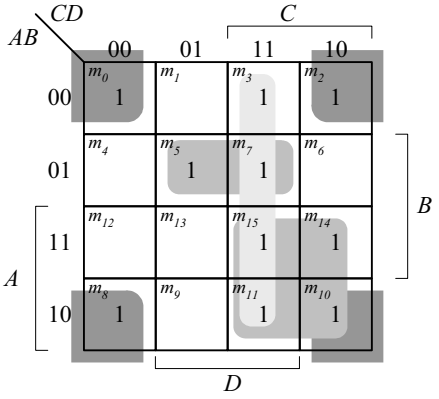
3.7



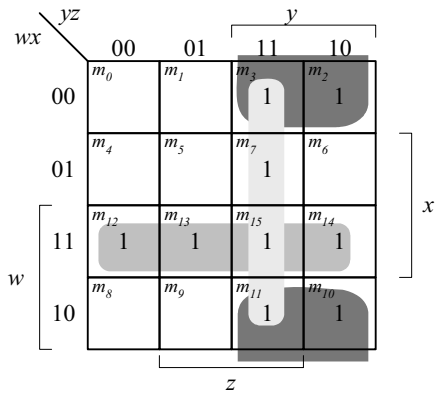
(a) $F = z + x'y$



(b) $F = C'D + B'C + ABC'$



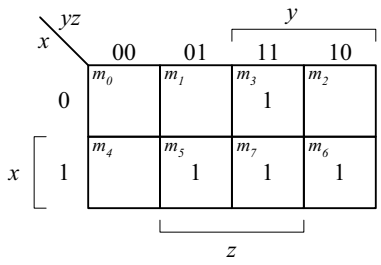
(c) $F = B'D' + AC + A'BD + CD$ (or $B'C$)



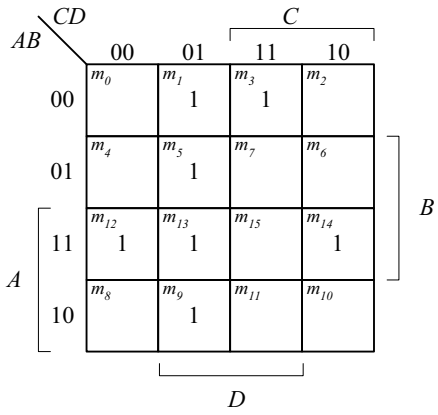
(d) $F = wx + x'y + yz$

3.8

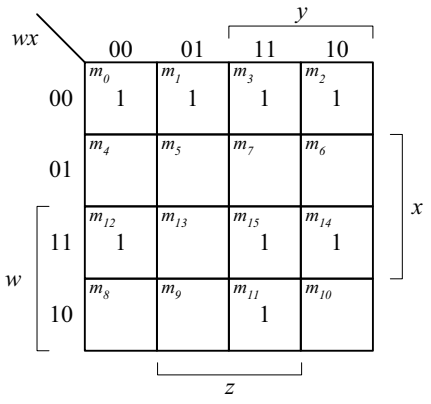
(a) $F(x, y, z) = \Sigma(3, 5, 6, 7)$



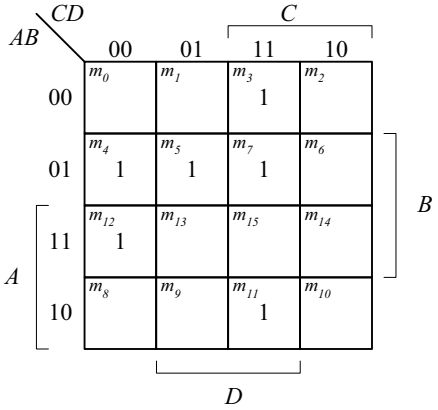
(b) $F = \Sigma(1, 3, 5, 9, 12, 13, 14)$



(c) $F = \Sigma(0, 1, 2, 3, 11, 12, 14, 15)$

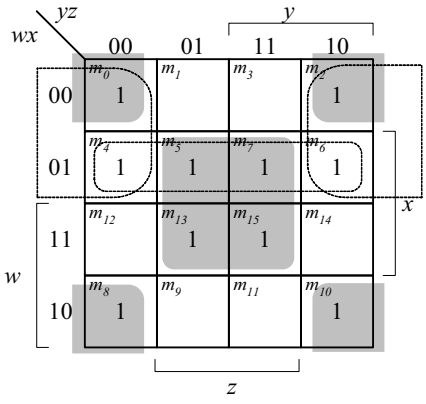


(d) $F = \Sigma(3, 4, 5, 7, 11, 12)$



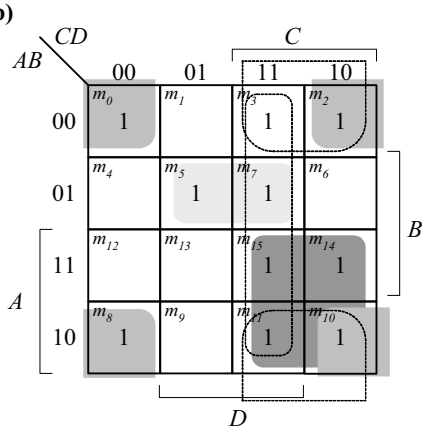
3.9

(a)



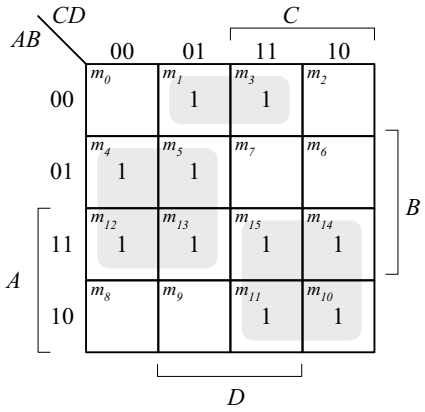
Essential: *xz, x′z′*
Non-essential: *w′x, w′z′*
F = xz + x′z′ + (w′x or w′z′)

(b)



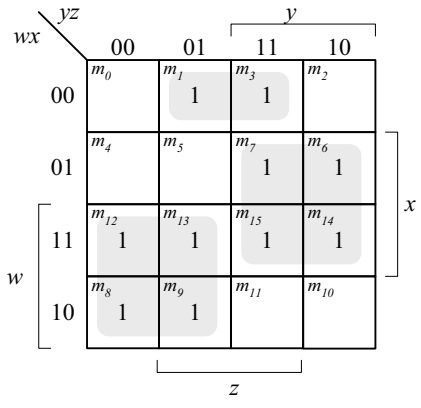
Essential: *B′D′, AC, A′BD*
Non-essential: *CD, B′C*
F = B′D′ + AC + A′BD + (CD OR B′C)

(c)



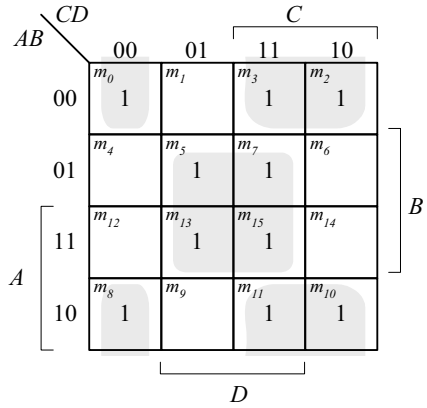
Essential: *BC′, AC, A′B′D*
F = BC′ + AC + A′B′D

(d)



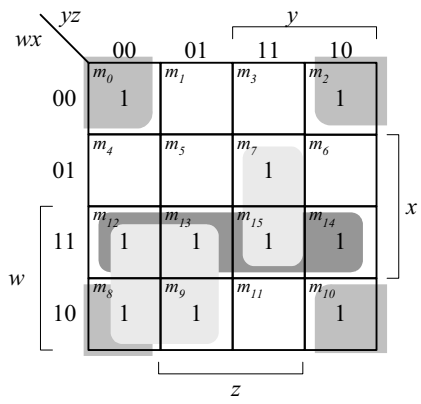
Essential: *wy′, xy, w′x′z*
F = wy′ + xy + w′x′z

(e)



***Essential**: BD, B′C, B′C′D′*
F = BD + B′C + B′C′D′

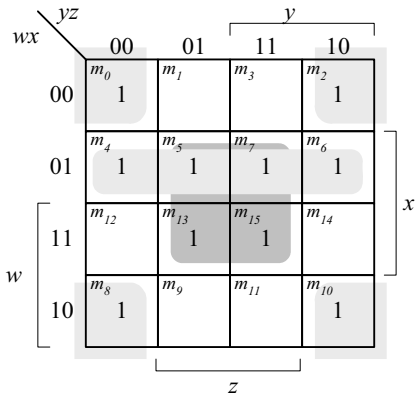
(f)



***Essential**: wy′, wx, x′z′, xyz*
F = wy′ + wx + x′z′ + xyz

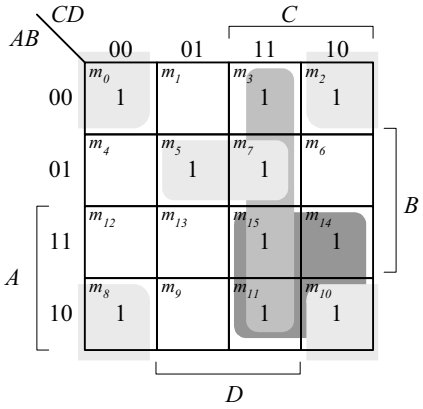
3.10

(a)



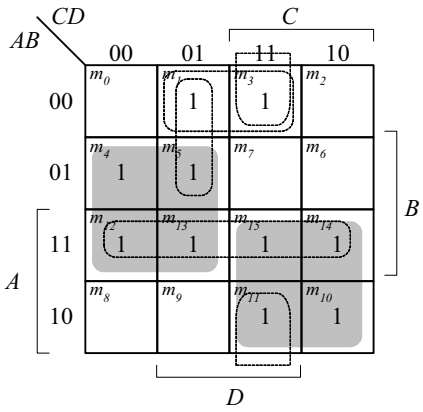
Essential: *xz, w′x, x′z′*
F = xz + w′x + x′z′

(b)



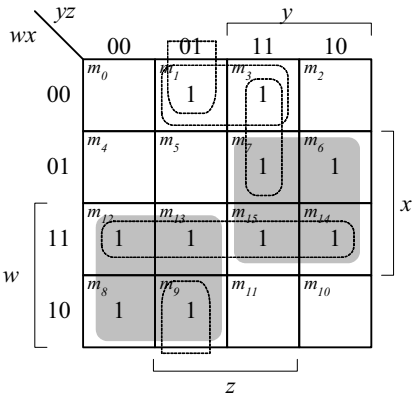
Essential: *AC, B′D′, CD, A′BD*
F = AC + B′D′ + CD + A′BD

(c)



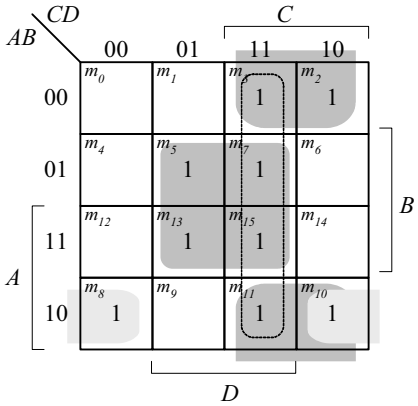
Essential: BC', AC
Non-essential: *AB, A′B′D, B′CD, A′C′D*
F = BC′ + AC + A′B′D

(d)



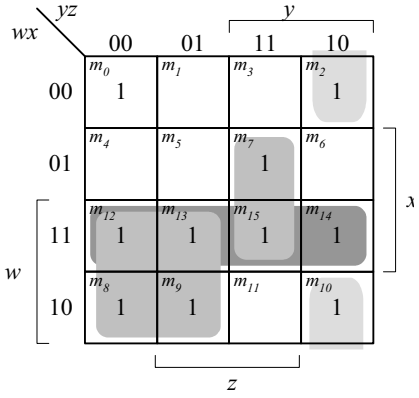
Essential: *wy′, xy*
Non-essential: *wx, x′y′z, w′wz, w′x′z*
F = wy′ + xy + w′x′z

(e)



Essential: *BD, B′C, AB′C*
Non-essential: *CD*
F = BD + B′C + AB′C

(f)

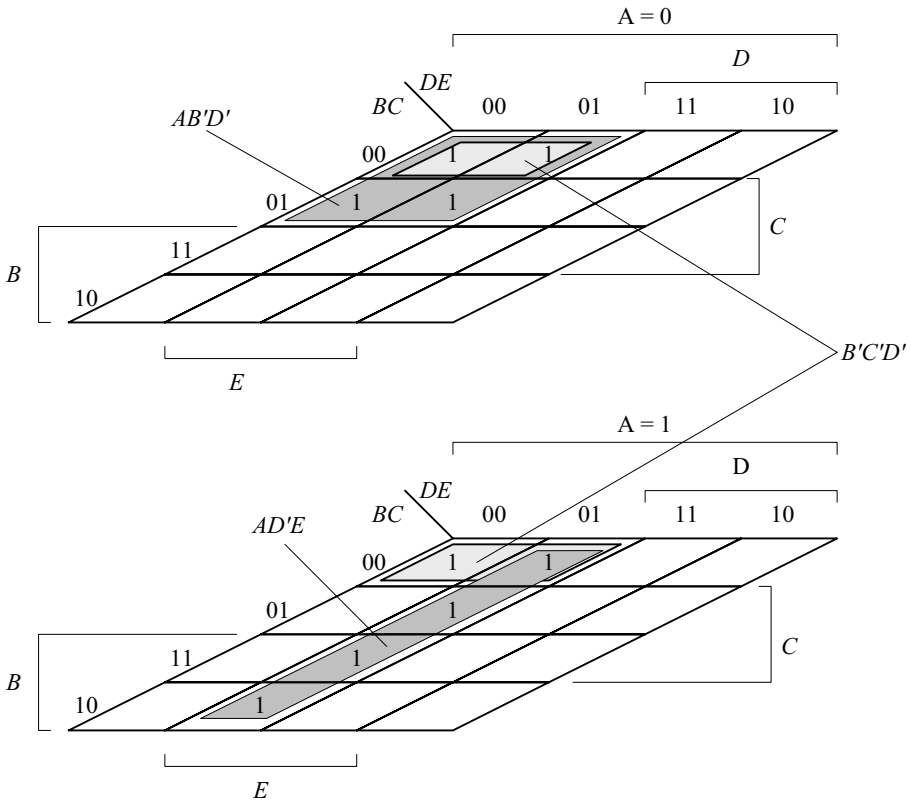


Essential: *wy′, wx, xyz, x′yz′*
F = wy′ + wx + xyz + x=yz′

3.11 **(a)** *F(A, B, C, D, E)* = ∑ (0, 1, 4, 5, 16, 17, 21, 25, 29

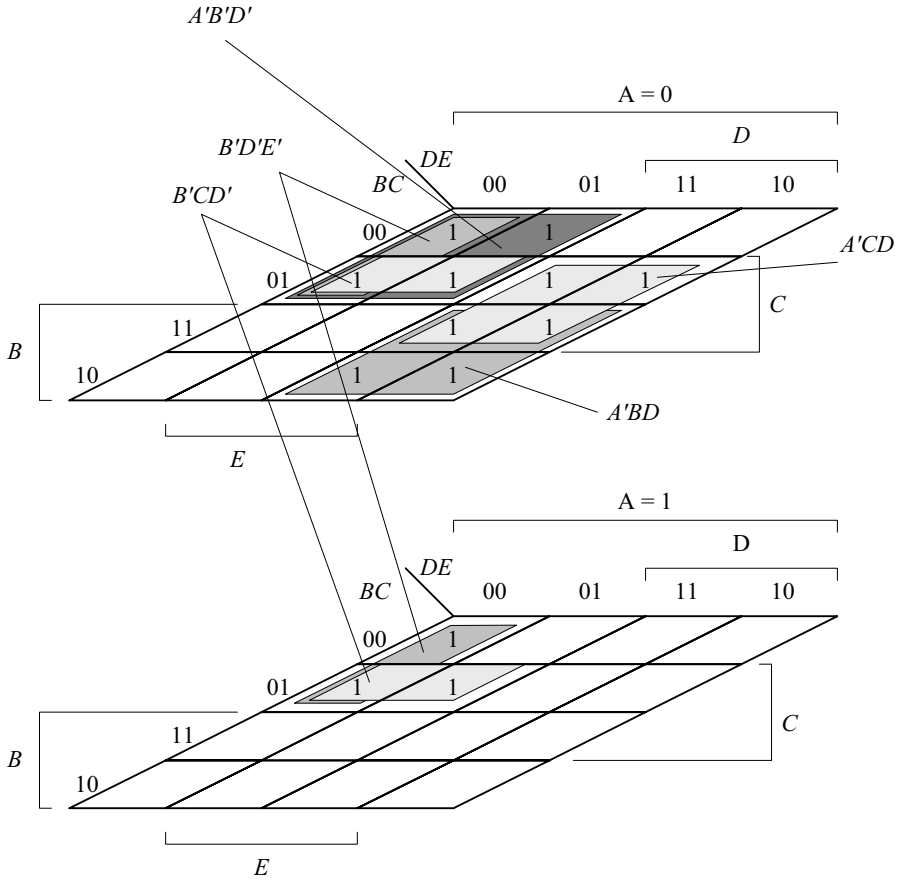
$F = A'B'D' + AD'E + B'C'D'$

m_0: $A'B'C'D'E'$ = 00000
m_1: $A'B'C'D'E$ = 00001
m_4: $A'B'CD'E'$ = 00100
m_5: $A'B'CD'E$ = 00101
m_{16}: $AB'C'D'E'$ = 10000
m_{17}: $AB'C'D'E$ = 10001
m_{21}: $AB'CD'E$ = 10101
m_{25}: $ABC'D'E$ = 11001
m_{29}: $ABCD'E$ = 11101



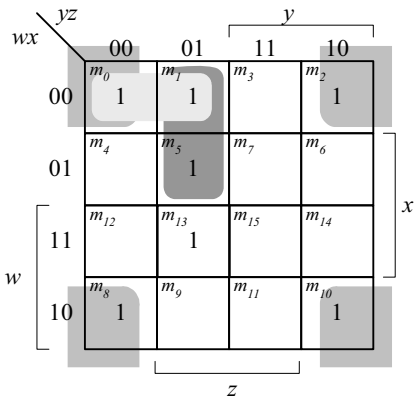
(b) *F(A, B, C, D, E) = A'B'CE' + B'C'D'E' + A'B'D' + B'CD' + A'CD + A'BD*
F(A, B, C, D, E) = A'B'D' + B'D'E' + B'CD' + A'CD + A'BD

A'B'CE': *AB'CDE' + A'B'CD'E'*
B'C'D'E': *AB'C'D'E' + A'B'C'D'E'*
A'B'D': *A'B'CD'E + A'B'CD'E' + A'B'C'D'E + A'B'C'D'E'*
B'CD': *AB'CD'E + AB'CD'E' + A'B'CD'E + A'B'CD'E'*
A'CD: *A'BCDE + A'BCDE' + A'B'CDE + A'B'CDE'*
A'BD: *A'BCDE + A'BCDE' + A'BC'DE + A'BC'DE'*

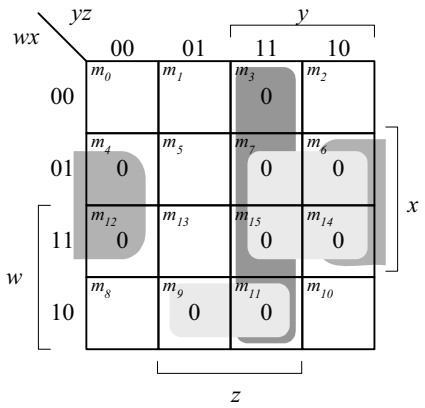


3.12

(a)

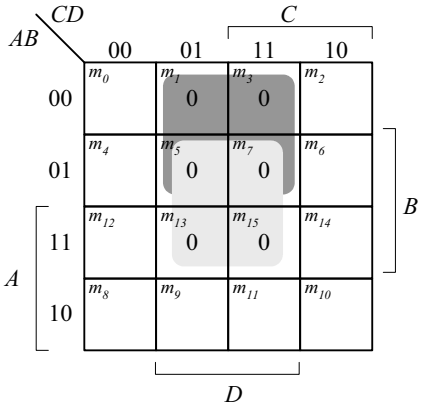


$F = \Sigma(0, 1, 2, 5, 8, 10, 13)$
$F = x'z' + w'x'y' + w'y'z$



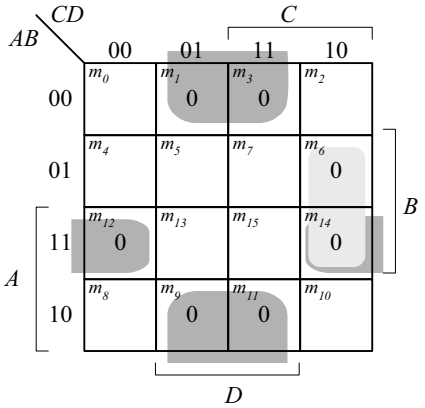
$F' = yz + xz' + xy + wx'z$
$F = (y' + z')(x' + z)(x' + y')(w' + x + z')$

(b)



$F = \Pi(1, 3, 5, 7, 13, 15)$
$F' = A'D + B'D$
$F = (A + D')(B' + D')$
$F = C'D' + AB' + CD'$

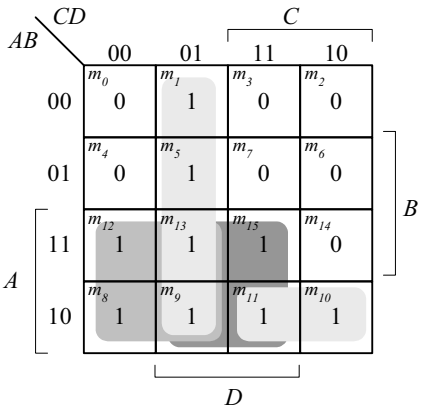
(c)



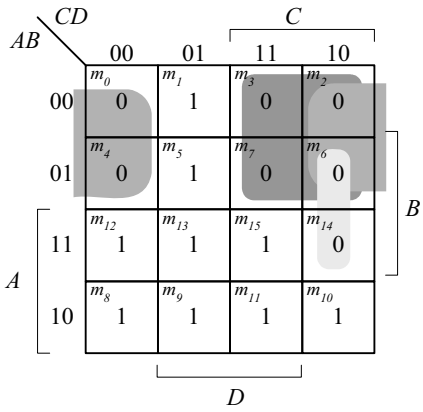
$F = \Pi(1, 3, 6, 9, 11, 12, 14)$
$F' = B'D + BCD' + ABD'$
$F = (B + D')(B' + C' + D)(A' + B' + D)$
$F = BD + B'D' + A'C'D'$

3.13 **(a)** $F = xy + z' = (x + z')(y + z')$

(b)



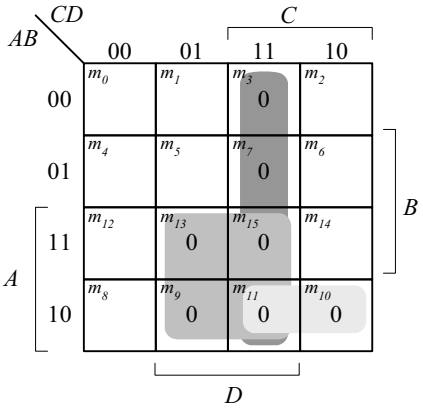
$F = AC' + AD + C'D + AB'C$

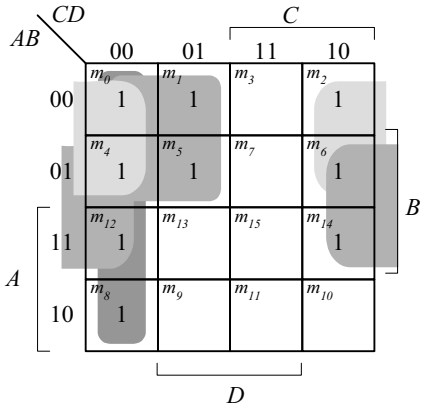


$F' A'D' + A'C + BCD'$

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

(c)





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

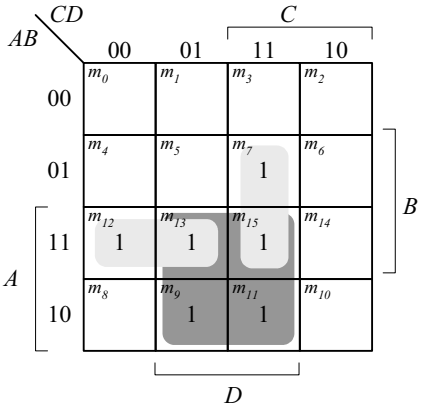
$F' = A'CD + ABD + AB'D + AB'C$

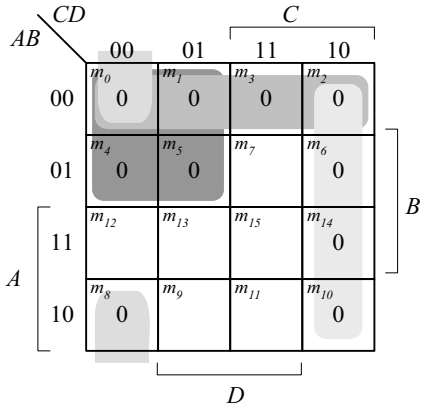
$F = A'C + A'D' + BD' + C'D'$

$F' = AD + CD + AB'C$

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

(d)

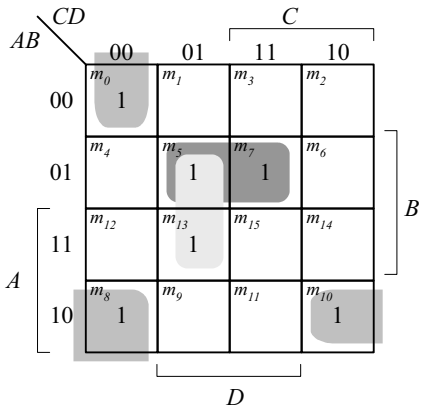


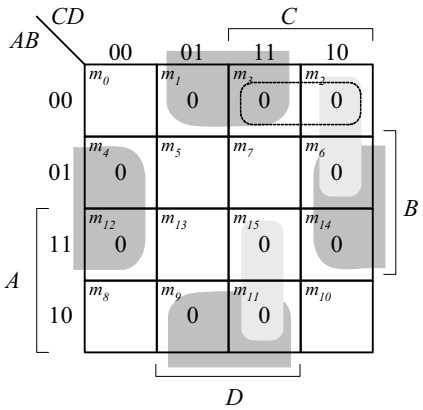


$F = ABC' + AB'D + BCD$
$F = AD + ABC' + BCD$

$F' = A'C' + A'B' + CD' + B'C'D'$
$F = (A + C)(A + B)(C' + D)(B + C + D)$

3.14





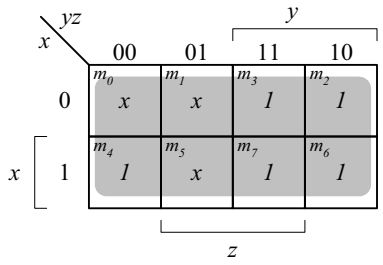
SOP form (using 1s): $F = B'C'D' + AB'D' + BC'D + A'BD$
$F = B'D'(A + C') + BD(A' + C')$

POS form (using 0s): $F' = BD' + B'D + A'CD' + ACD$
$F = [(B' + D)(B + D')][(A + C' + D)(A' + C' + D')]$

Alternative POS: $F' = BD' + B'D + A'CD' + A'B'C$
$F = [(B' + D)(B + D')][(A + C' + D)(A' + B + C)]$

3.15

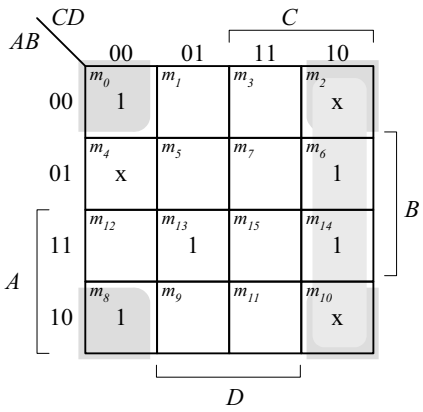
(a)



$F = 1$

$F = \Sigma(0,1, 2, 3, 4, 5, 6, 7)$

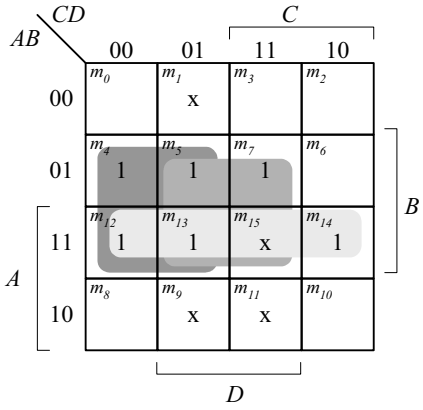
(b)



$F = B'D' + ABC'D$

$F = \Sigma(0, 2, 6, 8, 10, 13, 14)$

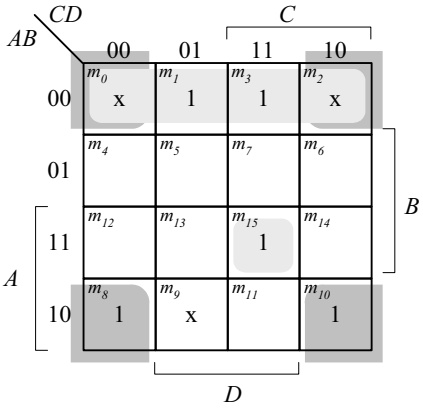
(c)



$F = BC' + BD + AB$

$F = \Sigma(4, 5, 7, 12, 13, 14, 15)$

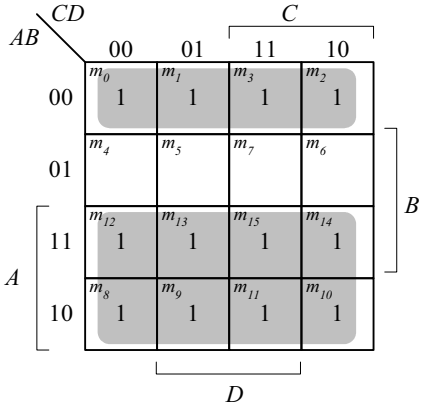
(d)



$F = B'D' + A'B' + ABCD$

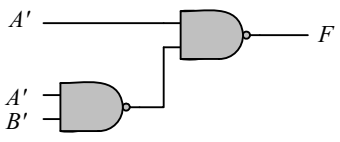
$F =$ $F = \Sigma(0, 1, 2, 3, 8, 10, 15)$

3.16 **(a)**

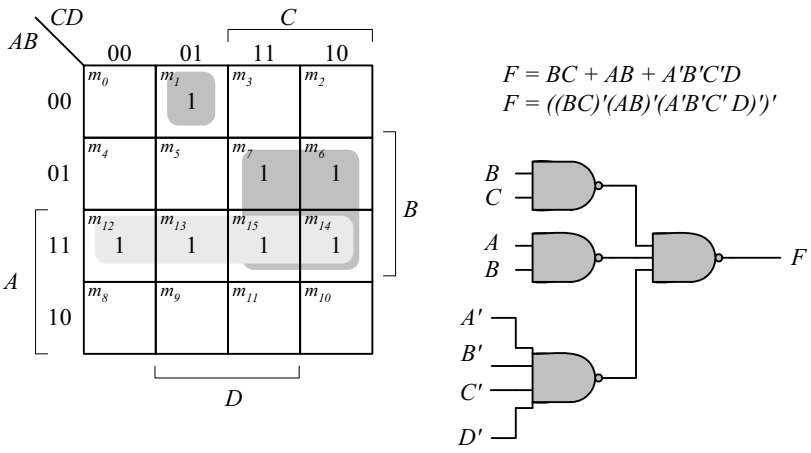


$F = A + A'B'$

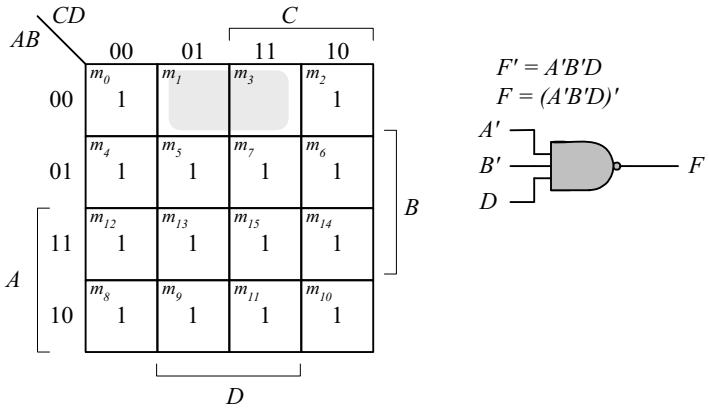
$F = (A'(A'B')')'$



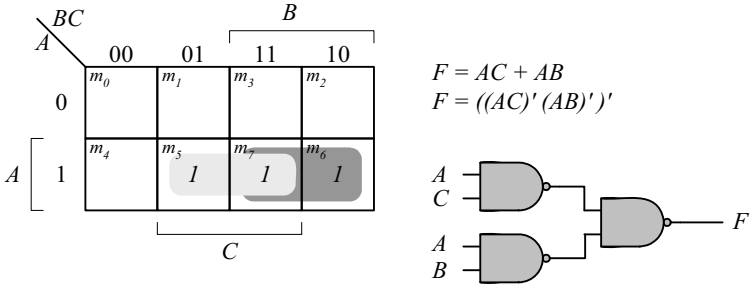
(b)



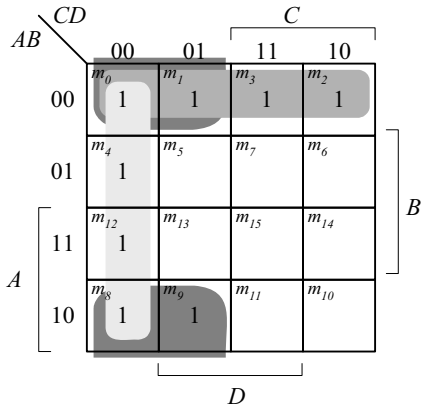
(c)

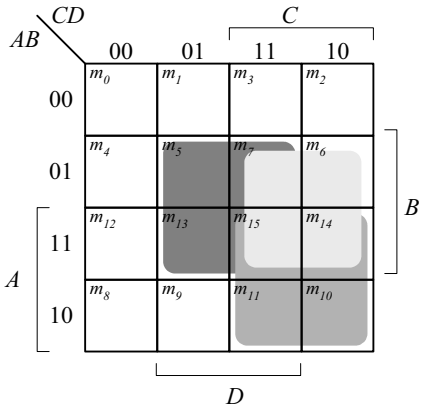


(d)



3.17

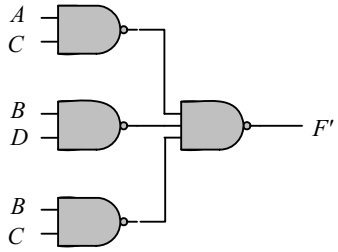




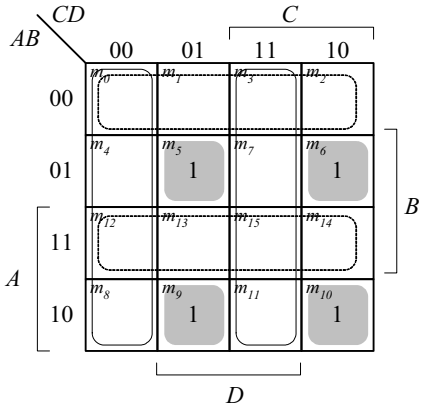
$F = A'B' + C'D' + B'C'$

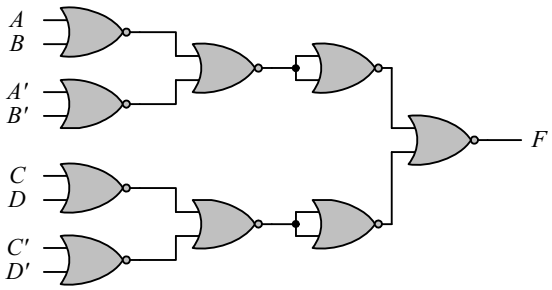
$F' = BC + AC + BD$

$F = (BC)'(AC)'(BD)'$



3.18 $F = (A \oplus)B'(C \oplus D) = (AB' + A'B)(CD' + C'D) = AB'CD' + AB'C'D + A'BCD' + A'BC'D$





$F = AB'CD' + AB'C'D + A'BCD' + A'BC'D$ and $F' = A'B' + AB + C'D' + CD$

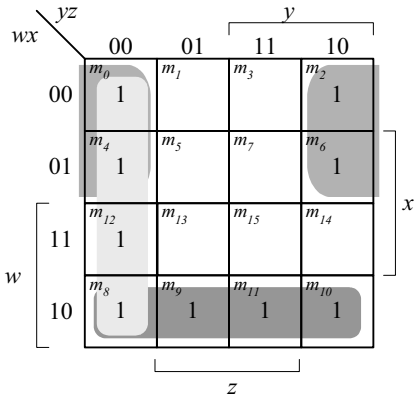
$F = (A'B')'(AB)'(C'D')'(CD)' = (A + B)(A' + B')\ (C' + D')(C + D)$

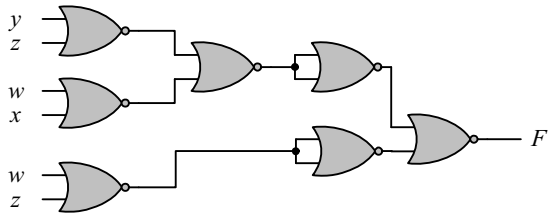
$F' = [(A + B)(A' + B')]' + [(C' + D')(C + D)]'$

$F = ([(A + B)(A' + B')]' + [(C' + D')(C + D)]')'$

$F = ([(A + B)' + (A' + B')'] + [(C' + D')' + (C + D)'])'$

3.19 **(a)** $F = (w + z')(x' + z')(w' + x' + y')$



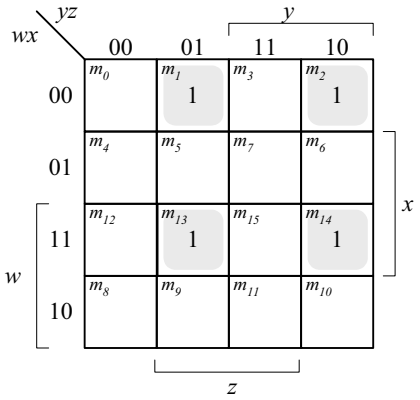


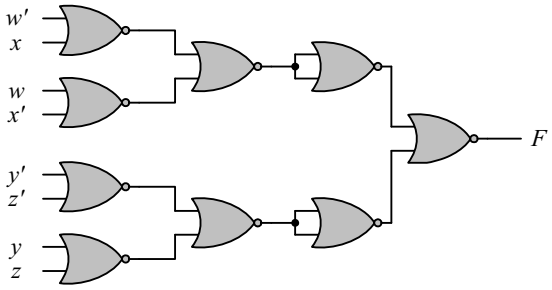
$F = y'z' + wx' + w'z'$

$F = [(y + z)' + (w' + x)' + (w + z)']$

$F' = [(y + z)' + (w' + x)' + (w + z)']'$

(b)





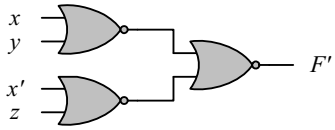
$F = \Sigma(1, 2, 13, 14)$

$F' = w'x + wx' + y'z' + yz = [(w + x')(w' + x)(y + z)(y' + z')]'$

$F = (w + x')' + (w' + x)' + (y + z)' + (y' + z')$

(c) $F = [(x + y)(x' + z)]' = (x + y)' + (x' + z)'$

$F' = [(x + y)' + (x' + z)']'$



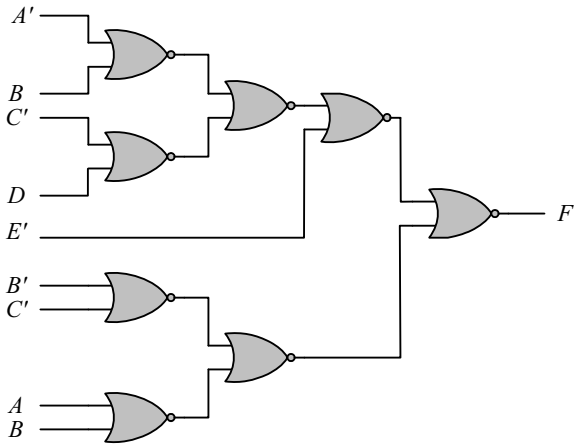
3.20 Multi-level NOR:

$F = (AB' + CD')E + BC(A + B)$

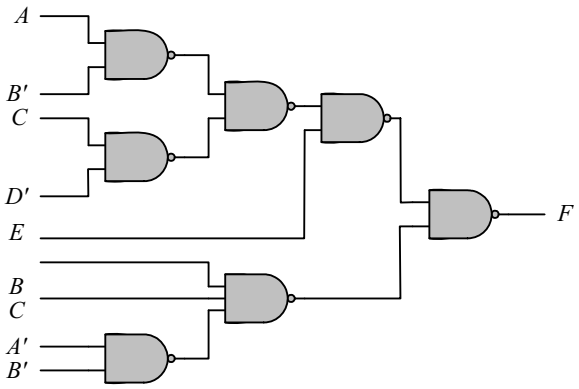
$F' = [(AB' + CD')E + BC(A + B)]'$

$F' = [\ [(AB' + CD')' + E']' + [\ (BC)' + (A + B)']'\ ]'$

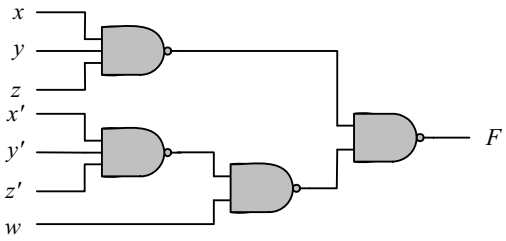
$F' = [\ [((A' + B)' + (C' + D)')' + E']' + [\ (B' + C')' + (A + B)']'\ ]'$



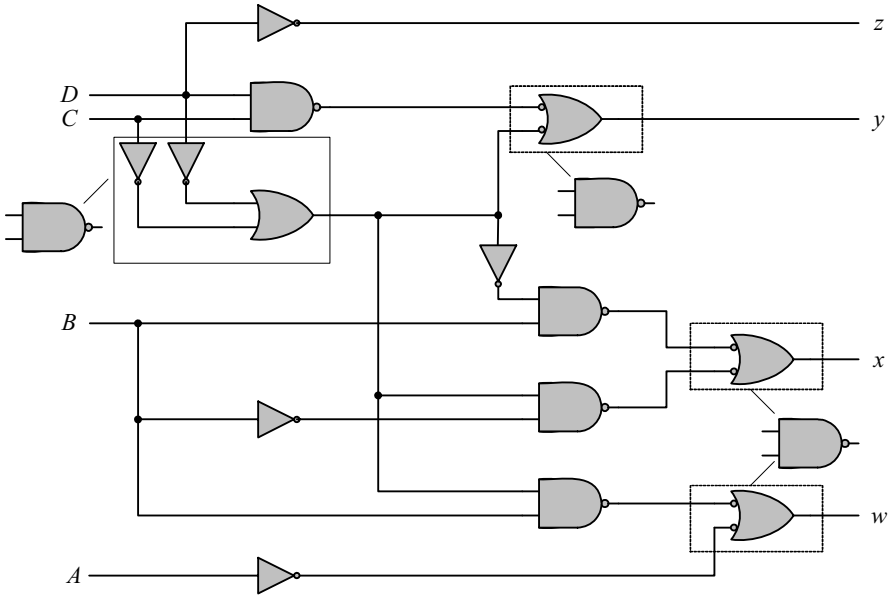
Multi-level NAND:
F = (AB′ + CD′)E + BC(A + B)
F′ = [(AB′ + CD′)E]′ [BC(A + B)]′
F′ = [((AB′)′(CD′)′)′E]′ [BC(A′B′)′]′



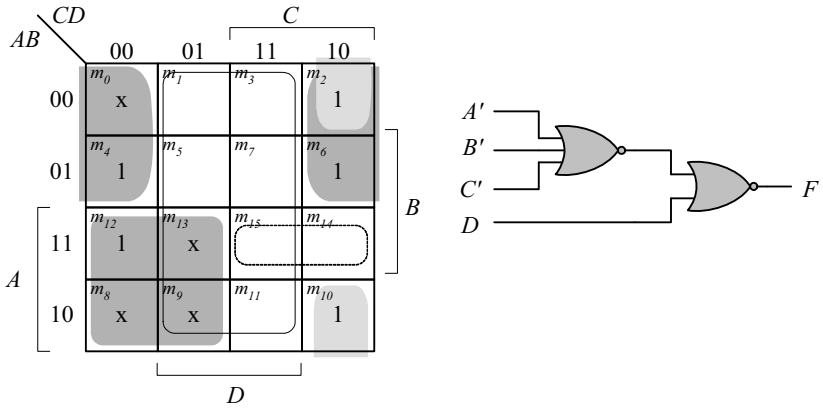
3.21 *F = w(x + y + z) + xyz*
F′ = [w(x + y + z)]′[xyz]′ = [w(x′y′z′)′]′(xyz)′



3.22

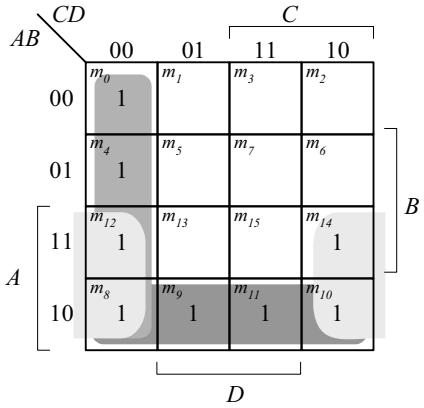


3.23



$F = AC' + A'D' + B'CD'$
$F' = D + ABC$
$F = [D + ABC]' = [D + (A' + B' + C']')]'$

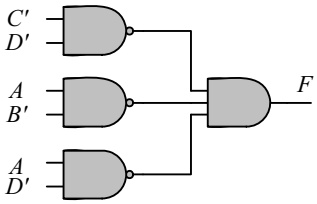
3.24



(a) $F = C'D' + AB' + AD'$

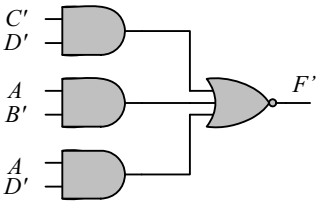
$F' = (C'D')'(AB')'(AD')'$

AND-NAND:



(b) $F' = [C'D' + AB' + AD']'$

AND-NOR:

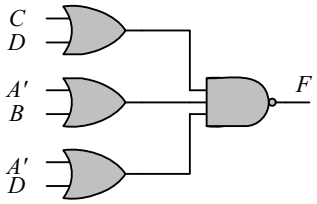


(c) $F = C'D' + AB' + AD' = (C + D)' + (A' + B)' + (A' + D)'$

$F' = (C'D')'(AB')'(AD')' = (C + D)(A' + B)(A' + D)$

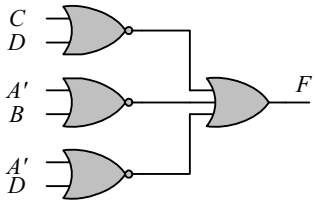
$F = [\,(C + D)(A' + B)(A' + D)\,]'$

OR-NAND:

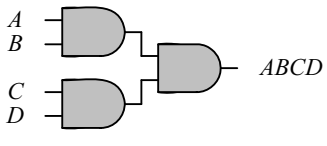


(d) $F = C'D' + AB' + AD' = (C + D)' + (A' + B)' + (A' + D)'$

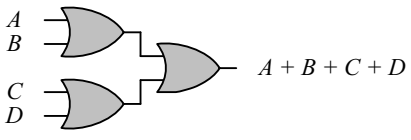
NOR-OR:



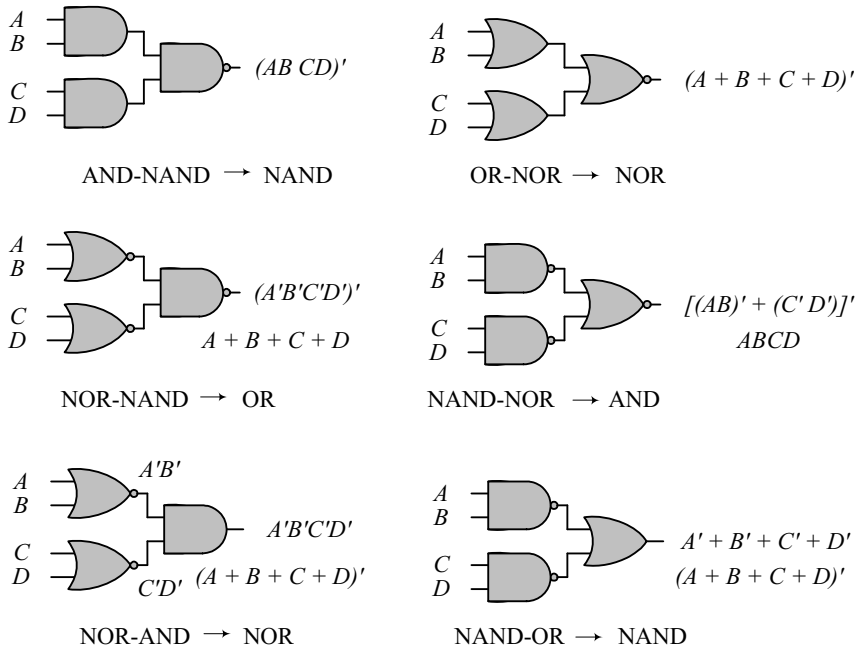
3.25



AND-AND → AND

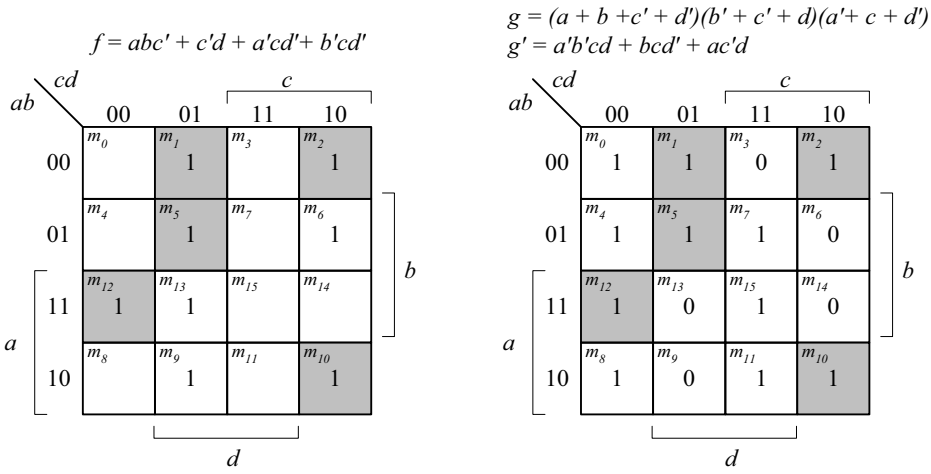


OR-OR → OR



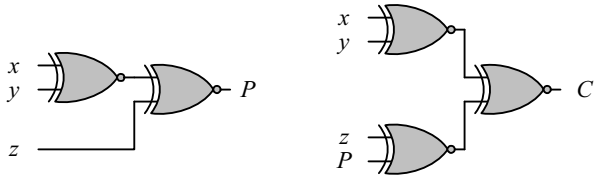
The degenerate forms use 2-input gates to implement the functionality of 4-input gates.

3.26



3.27 $x \oplus y = x'y + xy'$; Dual $= (x' + y)(x + y') = (x \oplus y)'$

3.28



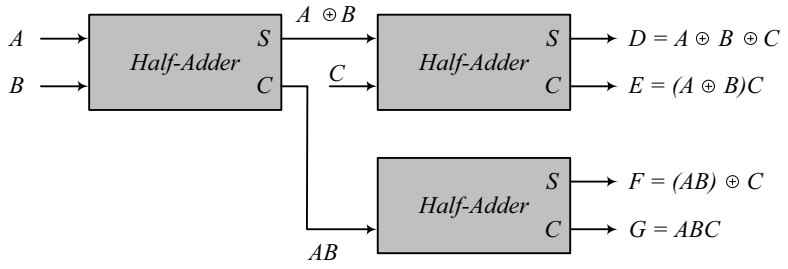
(a) 3-bit odd parity generator *(b) 4-bit odd parity generator*

3.29 $D = A \oplus B \oplus C$

$E = A'BC + AB'C = (A \oplus B)C$

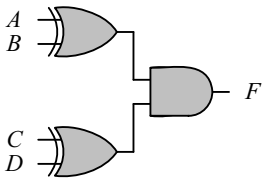
$F = ABC' + (A' + B')C = ABC' + (AB)'C = (AB) \oplus C$

$G = ABC$



3.30 $F = AB'CD' + A'BCD' + AB'C'D + A'BC'D$

$F = (A \oplus B)CD' + (A \oplus B)\,C'D = (A \oplus B)(C \oplus D)$



3.31 Note: It is assumed that a complemented input is generated by another circuit that is not part of the circuit that is to be described.

(a)
```
module Fig_3_22a_gates (F, A, B, C, C_bar, D);
  output   F;
  input    A, B, C, C_bar, D;
  wire     w1, w2, w3, w4;
  and      (w1, C, D);
  or       (w2, w1, B);
  and      (w3, w2, A);
  and      (w4, B, C_bar);
  or       (F, w3, w4);
endmodule
```

(b)
```
module Fig_3_22b_gates (F, A, B, C, C_bar, D);
  output   F;
  input    A, B, C, C_bar, D;
  wire     w1, w2, w3, w4;
  not      (w1_bar, w1);
  not      (B_bar, B);
  not      (w3_bar, w3);
  not      (w4_bar, w4);
  nand     (w1, C, D);
  or       (w2, w1_bar, B_bar);
  nand     (w3, w2, A);
  nand     (w4, B, C_bar);
  or       (F, w3_bar, w4_bar);
endmodule
```

(c)

```
module Fig_3_23a_gates (F, A, A_bar, B, B_bar, C, D_bar);
 output  F;
 input   A, A_bar, B, B_bar, C, D_bar;
 wire    w1, w2, w3, w4;
 and     (w1, A, B_bar);
 and     (w2, A_bar, B);
 or      (w3, w1, w2);
 or      (w4, C, D_bar);
 or      (F, w3, w4);
endmodule
```

(d)

```
module Fig_3_23b_gates (F, A, A_bar, B, B_bar, C_bar, D);
 output  F;
 input   A, A_bar, B, B_bar, C_bar, D;
 wire    w1, w2, w3, w4;
 nand    (w1, A, B_bar);
 nand    (w2, A_bar, B);
 not     (w1_bar, w1);
 not     (w2_bar, w2);
 or      (w3, w1_bar, w2_bar);
 or      (w4, C, D_bar);
 not     (w5, C_bar);
 not     (w6, D);
 nand    (F_bar, w5, w6);
 not     (F, F_bar);
endmodule
```

(e)

```
module Fig_3_26_gates (F, A, B, C, D, E_bar);
 output  F;
 input   A, B, C, D, E_bar;
 wire    w1, w2, w1_bar, w2_bar, w3_bar;
 not     (w1_bar, w1);
 not     (w2_bar, w2);
 not     (w3_bar, E_bar);
 nor     (w1, A, B);
 nor     (w2, C, D);
 nand    (F, w1_bar, w2_bar, w3_bar);
endmodule
```

(f)

```
module Fig_3_27_gates (F, A, A_bar, B, B_bar, C, D_bar);
 output  F;
 input   A, A_bar, B, B_bar, C, D_bar
 wire    w1, w2, w3, w4, w5, w6, w7, w8, w7_bar, w8_bar;
 not     (w1, A_bar);
 not     (w2, B_bar);
 not     (w3, A);
 not     (w4, B_bar);
 not     (w7_bar, w7);
 not     (w8_bar, w8);
 and     (w5 w1, w2);
 and     (w6, w3, w4);
 nor     (w7, w5, w6);
 nor     (w8, C, D_bar);
 and     (F, w7_bar, w8_bar);
endmodule
```

3.32 Note: It is assumed that a complemented input is generated by another circuit that is not part of the circuit that is to be described.

(a)
```
module Fig_3_22a_CA (F, A, B, C, C_bar, D);
  output   F;
  input    A, B, C, C_bar, D;
  wire     w1, w2, w3, w4;
  assign   w1 = C & D;
  assign   w2 = w1| B;
  assign   w3 = w2 & A);
  assign   w4 = B & C_bar);
  assign   F = w3 | w4);
endmodule
```

(b)
```
module Fig_3_22b_CA (F, A, B, C, C_bar, D);
  output   F;
  input    A, B, C, C_bar, D;
  wire     w1, w2, w3, w4;
  assign   w1_bar = ~w1;
  assign   B_bar = ~B;
  assign   w3_bar = ~w3;
  assign   w4_bar = ~w4;
  assign   w1 = ~(C & D);
  assign   w2 = w1_bar | B_bar;
  assign   w3 = ~(w2 & A);
  assign   w4 = ~(B & C_bar);
  assign   F = w3_bar | w4_bar;
endmodule
```

(c)
```
module Fig_3_23a_CA (F, A, A_bar, B, B_bar, C, D_bar);
  output   F;
  input A, A_bar, B, B_bar, C, D_bar;
  wire     w1, w2, w3, w4;
  assign   w1 = A & B_bar;
  assign   w2 = A_bar & B;
  assign   w3 = w1 | w2);
  assign   w4 = C | D_bar;
  assign   F = w3 | w4;
endmodule
```

(d)
```
module Fig_3_23b_CA (F, A, A_bar, B, B_bar, C_bar, D);
  output   F;
  input A, A_bar, B, B_bar, C_bar, D;
  wire     w1, w2, w3, w4;
  assign   w1 = ~(A & B_bar);
  assign   w2 = ~(A_bar & B);
  assign   w1_bar = ~w1;
  assign   w2_bar = ~w2;
  assign   w3 = w1_bar | w2_bar;
  assign   w4, C | D_bar;
  assign   w5 = ~C_bar;
  assign   w6 = ~D;
  assign   F_bar = ~(w5 & w6);
  assign   F = ~F_bar;
endmodule
```

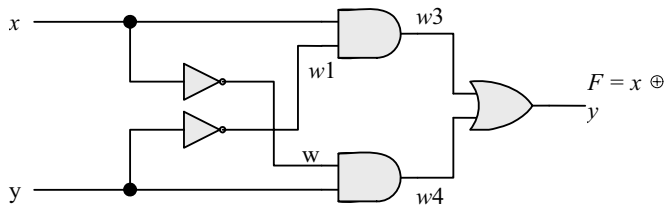
(e)

```
module Fig_3_26_CA (F, A, B, C, D, E_bar);
 output   F;
 input    A, B, C, D, E_bar;
 wire     w1, w2, w1_bar, w2_bar, w3_bar;
 not      w1_bar = ~w1;
 not      w2_bar = ~w2;
 not      w3_bar = ~E_bar;
 nor      w1 = (A | B;
 nor      w2 = (C | D;
 nand     F = ~( w1_bar & w2_bar & w3_bar);
endmodule
```

(f)

```
module Fig_3_27_CA (F, A, A_bar, B, B_bar, C, D_bar);
 output   F;
 input    A, A_bar, B, B_bar, C, D_bar
 wire     w1, w2, w3, w4, w5, w6, w7, w8, w7_bar, w8_bar;
 not      w1 = ~A_bar;
 not      w2 = ~B_bar;
 not      w3 = ~A;
 not      w4 = ~B_bar;
 not      w7_bar = ~w7;
 not      w8_bar = ~w8;
 assign   w5 = w1 & w2;
 assign   w6 = w3 & w4;
 assign   w7 = ~(w5 | w6);
 assign   w8 =  ~(C | D_bar);
 assign   F = w7_bar & w8_bar;
endmodule
```

3.32 **(a)**



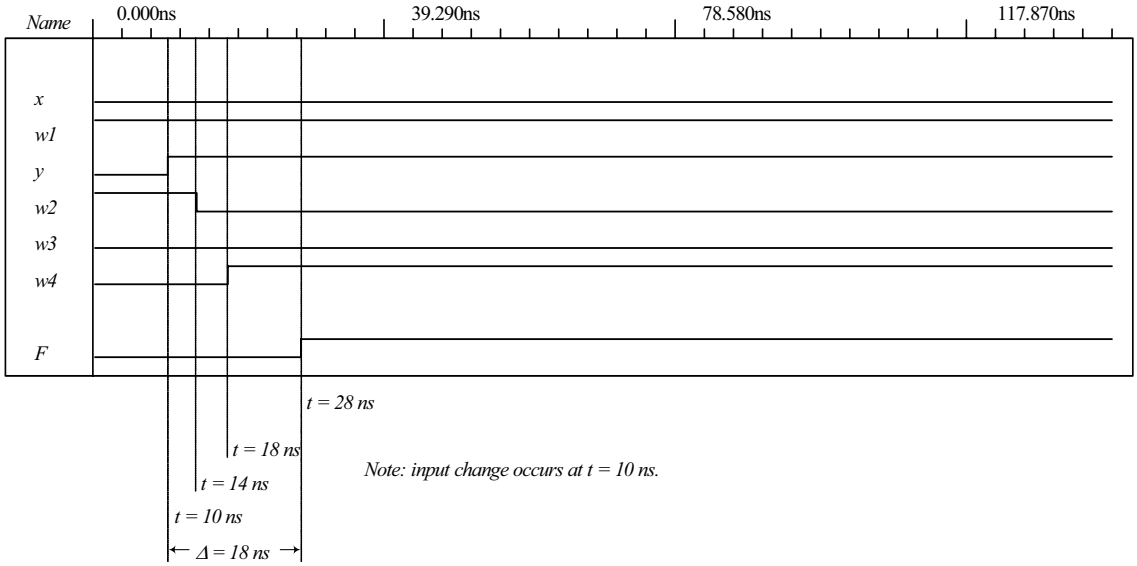
Initially, with $xy = 00$, $w1 = w2 = 1$, $w3 = w4 = 0$ and $F = 0$. $w1$ should change to 0 4ns after xy changes to 01. $w4$ should change to 1 8 ns after xy changes to 01. F should change from 0 to 1 10 ns after $w4$ changes from 0 to 1, i.e., 18 ns after xy changes from 00 to 01.

(b)

```
`timescale 1ns/1ps

module Prob_3_33 (output F, input x, y);
wire w1, w2, w3, w4;

and #8 (w3, x, w1);
not #4 (w1, x);
and #8 (w4, y, w1);
not #4 (w2, y);
or  #10 (F, w3, w4);

endmodule

module t_Prob_3_33 ();
 reg x, y;
 wire F;
```

```
Prob_3_33 M0 (F, x, y);

initial #200 $finish;
initial fork
  x = 0;
  y = 0;
  #20 y = 1;
 join
endmodule
```

(c) To simulate the circuit, it is assumed that the inputs xy = 00 have been applied sufficiently long for the circuit to be stable before xy = 01 is applied. The testbench sets xy = 00 at t = 0 ns, and xy = 1 at t = 10 ns. The simulator assumes that xy = 00 has been applied long enough for the circuit to be in a stable state at t = 0 ns, and shows F = 0 as the value of the output at t = 0. The waveforms show the response to xy = 01 applied at t = 10 ns.



3.34

```
module Prob_3_34 (Out_1, Out_2, Out_3, A, B, C, D);
  output   Out_1, Out_2, Out_3;
  input    A, B, C, D;
  wire     A_bar, B_bar, C_bar, D_bar;
  assign   A_bar = ~A;
  assign   B_Bar = ~B;
  assign   C_bar = ~C;
  assign   D_bar = ~D;
  assign   Out_1 = ~( (C | B) & (A_bar | D) & B );
  assign   Out_2 = ((C * B_bar) | (A & B & C) | (C_bar & B)  ) & (A | D_bar);
  assign   Out_3 = C & ( (A & D) | B ) | (C & A_bar);
endmodule
```

3.35

```
module Exmpl-3(A, B, C, D, F)      // Line 1
 inputs A, B, C, Output D, F,      // Line 2
 output B                          // Line 3
 and g1(A, B, B);                  // Line 4
 not (D, B, A),                    // Line 5
 OR (F, B; C);                     // Line 6
```

endofmodule; // Line 7

Line 1: Dash not allowed, use underscore: Exmpl_3. Terminate line with semicolon (;).

Line 2: **inputs** should be **input** (no s at the end). Change last comma (,) to semicolon (;). *Output* is declared but does not appear in the port list, and should be followed by a comma if it is intended to be in the list of inputs. If *Output* is a mispelling of **output** and is to declare output ports, *C* should be followed by a semicolon (;) and *F* should be followed by a semicolon (;).

Line 3: *B* cannot be declared as input (Line 2) and output (Line 3). Terminate the line with a semicolon (;).

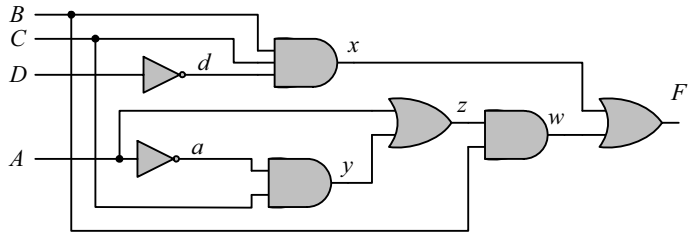
Line 4: *A* cannot be an output of the primitive if it is an input to the module

Line 5: Too many entries for the not gate (only two allowed).

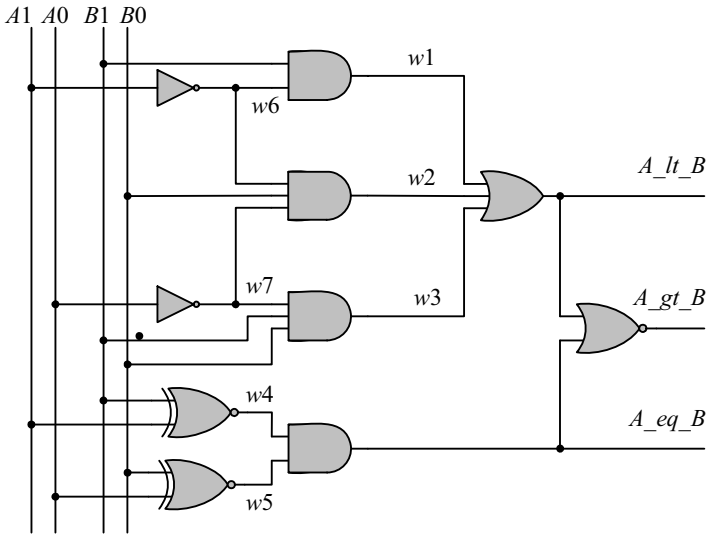
Line 6: OR must be in lowercase: change to "or".

Line 7: **endmodule** is mispelled. Remove semicolon (no semicolon after endmodule).

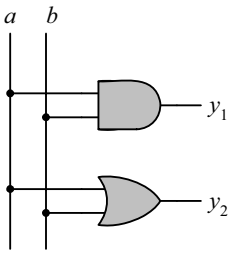
3.36 **(a)**



(b)



(c)



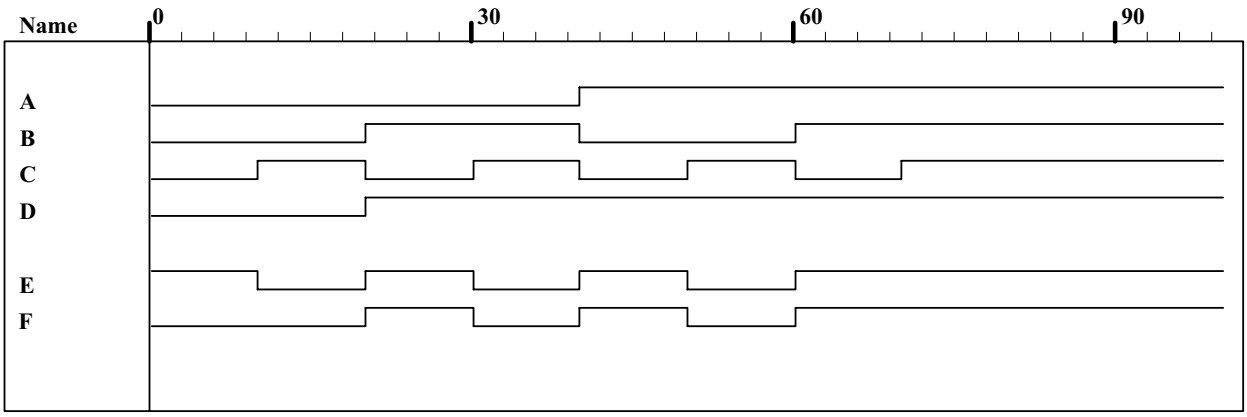
3.37

```
UDP_Majority_4 (y, a, b, c, d);
 output y;
 input      a, b, c, d;
 table
 // a   b   c   d   :   y
    0   0   0   0   :   0;
    0   0   0   1   :   0;
    0   0   1   0   :   0;
    0   0   1   1   :   0;
    0   1   0   0   :   0;
    0   1   0   1   :   0;
    0   1   1   0   :   0;
    0   1   1   1   :   1;

    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   :   1;
    1   1   1   1   :   1;
  endtable
endprimitive
```

3.38

```
module t_Circuit_with_UDP_02467;
  wire  E, F;
  reg   A, B, C, D;
  Circuit_with_UDP_02467 m0 (E, F, A, B, C, D);

  initial #100 $finish;
  initial fork
    A = 0; B = 0; C = 0; D = 0;
    #40 A = 1;
    #20 B = 1;
    #40 B = 0;
    #60 B = 1;
    #10 C = 1; #20 C = 0; #30 C = 1; #40 C = 0; #50 C = 1; #60 C = 0; #70 C = 1;
    #20 D = 1;
  join
endmodule

// Verilog model: User-defined Primitive
primitive UDP_02467 (D, A, B, C);
  output D;
  input A, B, C;
// Truth table for D = f (A, B, C) = Σ (0, 2, 4, 6, 7);
```

```
  table
//   A   B   C   :   D   // Column header comment
     0   0   0   :   1;
     0   0   1   :   0;
     0   1   0   :   1;
     0   1   1   :   0;
     1   0   0   :   1;
     1   0   1   :   0;
     1   1   0   :   1;
     1   1   1   :   1;
  endtable
endprimitive
// Verilog model: Circuit instantiation of Circuit_UDP_02467
module Circuit_with_UDP_02467 (e, f, a, b, c, d);
  output   e, f;
  input    a, b, c, d;

 UDP_02467  M0 (e, a, b, c);
  and          (f, e, d);        //Option gate instance name omitted
endmodule
```

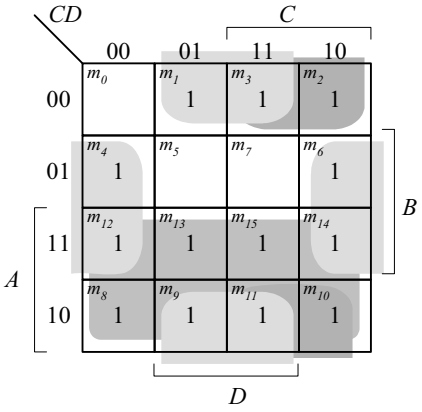


CHAPTER 4

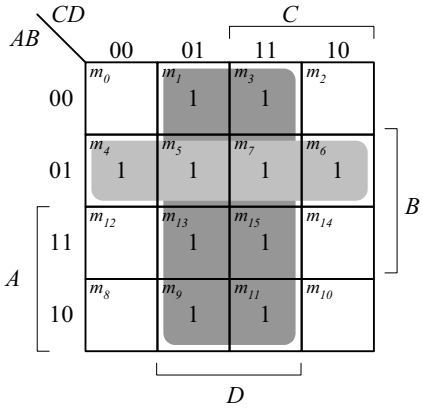
4.1 **(a)** $T_1 = B'C$, $T_2 = A'B$, $T_3 = A + T_1 = A + B'C$,
$T_4 = D \oplus T_2 = D \oplus (A'B) = A'BD' + D(A + B') = A'BD' + AD + B'D$
$F_1 = T_3 + T_4 = A + B'C + A'BD' + AD + B'D$
With $A + AD = A$ and $A + A'BD' = A + BD'$:
$F_1 = A + B'C + BD' + B'D$
Alternative cover: $F_1 = A + CD' + BD' + B'D$

$F_2 = T_2 + D = A'B + D$

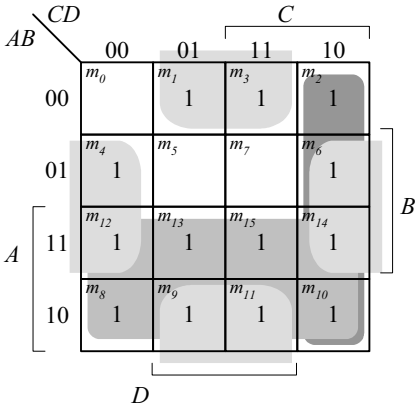
ABCD	T_1	T_2	T_3	T_4	F_1	F_2
0000	0	0	0	0	0	0
0001	0	0	0	1	1	1
0010	1	0	1	0	1	0
0011	1	0	1	1	1	1
0100	0	1	0	1	1	1
0101	0	1	0	0	0	1
0110	0	1	0	1	1	1
0111	0	1	0	0	0	1
1000	0	0	1	0	1	0
1001	0	0	1	1	1	1
1010	1	0	1	0	1	0
1011	1	0	1	1	1	1
1100	0	0	1	0	1	0
1101	0	0	1	1	1	1
1110	0	0	1	0	1	0
1111	0	0	1	1	1	1



$F_1 = A + B'C + B'D + BD'$

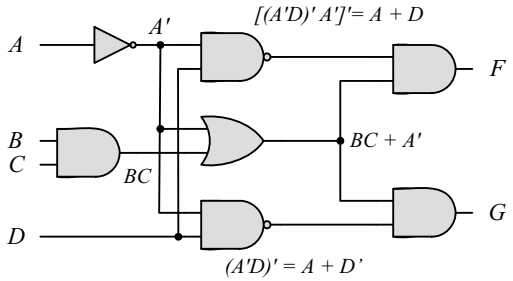


$F_2 = A'B + D$



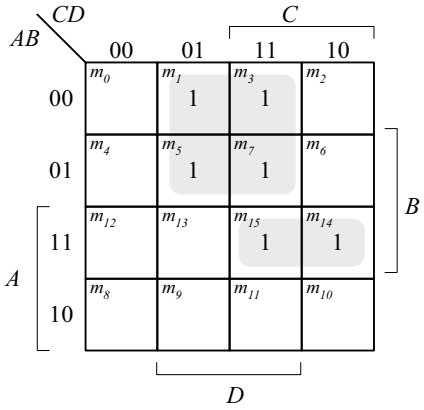
$F_1 = A + CD' + B'D + BD'$

4.2

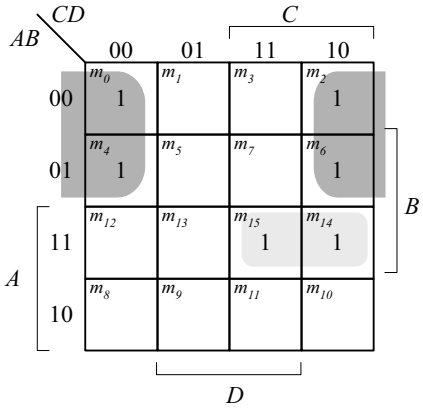


$F = (A + D)(A' + BC) = A'D + ABC + BCD + = A'D + ABC$

$F = (A + D')(A' + BC) = A'D' + ABC + BCD' = A'D' + ABC$



$F = A'D + ABC + BCD = A'D + ABC$



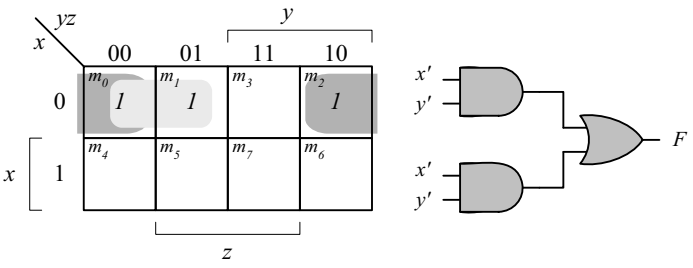
$G = A'D' + ABC + BCD' = A'D' + ABC$

4.3 **(a)** $Y_i = (A_iS' + B_iS)E'$ *for* $i = 0, 1, 2, 3$

(b) 1024 rows and 14 columns

4.4 **(a)**

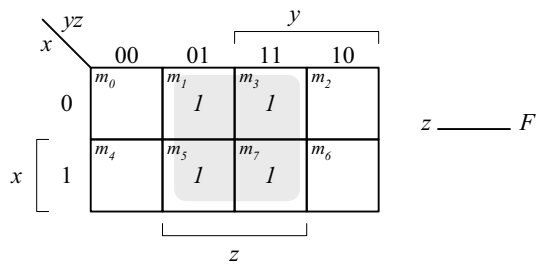
xyz	F
000	1
001	1
010	1
011	0
100	0
101	0
110	0
111	0



$F = x'y' + x'z'$

(b)

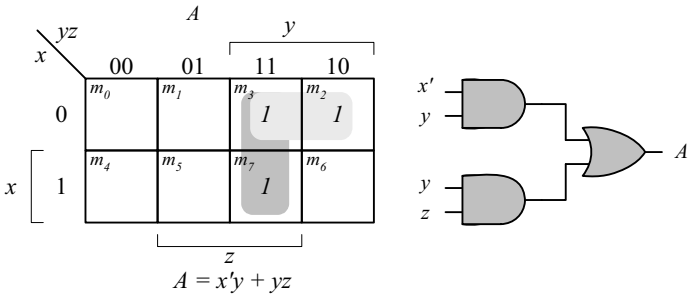
xyz	F
000	0
001	1
010	0
011	0
100	0
101	0
110	0
111	0



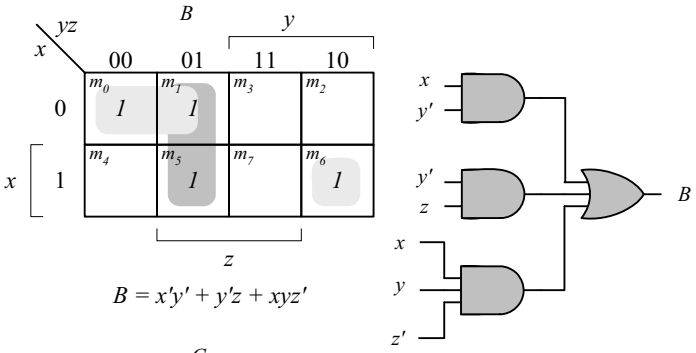
$F = z$

4.5

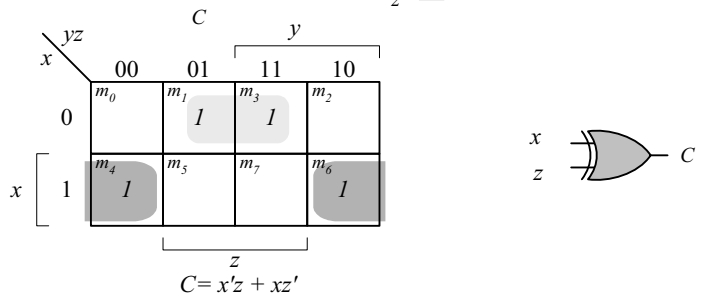
xyz	ABC
000	010
001	011
010	100
011	101
100	001
101	010
110	011
111	100



$A = x'y + yz$



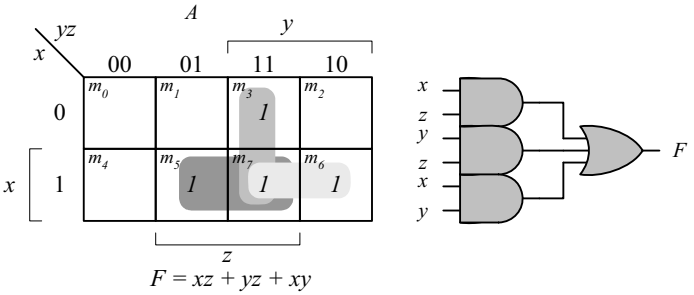
$B = x'y' + y'z + xyz'$



$C = x'z + xz'$

4.6

xyz	F
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

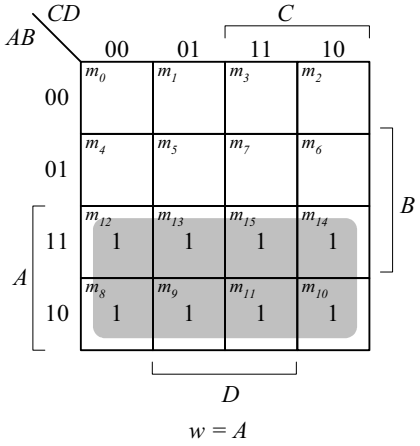


$F = xz + yz + xy$

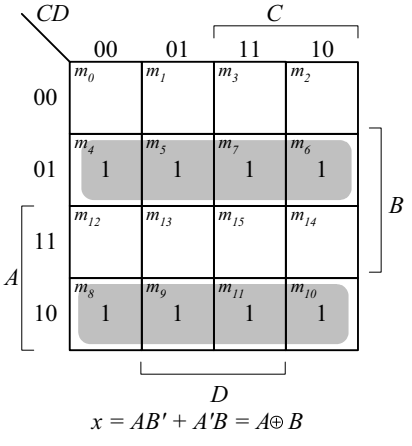
```
module Prob_4_6 (output F, input x, y, z);
  assign F = (x & z) | (y & z) | (x & y);
endmodule
```

4.7 **(a)**

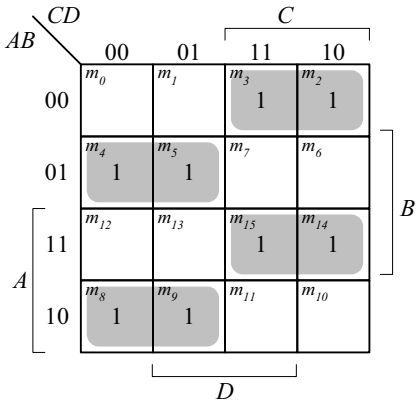
ABCD	wxyz
0000	0000
0001	0001
0011	0010
0010	0011
0110	0100
0111	0101
0101	0110
0100	0111
1100	1000
1101	1001
1111	1010
1110	1011
1010	1100
1011	1101
1001	1110
1000	1111



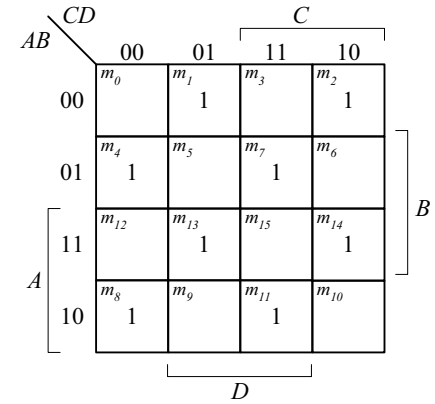
$w = A$



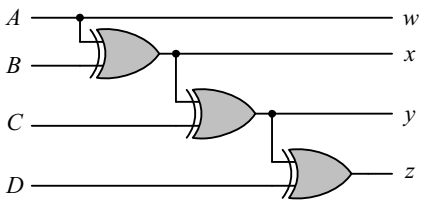
$x = AB' + A'B = A \oplus B$



$y = A'B'C\ \ A'BC' + ABC + AB'C'$
$= A'(A \oplus B) + A(B \oplus C)'$
$= A \oplus B \oplus C$
$= X \oplus C$



$z = A \oplus B \oplus C \oplus D$
$= y \oplus D$



(b)

```
module Prob_4_7(output w, x, y, z, input A, B, C, D);
  always @ (A, B, C, D)
    case ({A, B, C, D})
      4'b0000:      {w, x, y, z} = 4'b0000;
```

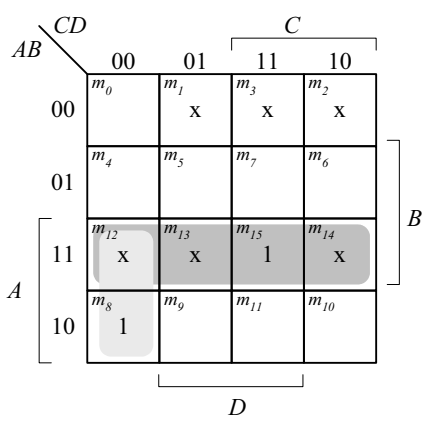
```
        4'b0001:    {w, x, y, z} = 4'b1111;
        4'b0010:    {w, x, y, z} = 4'b1110;
        4'b0011:    {w, x, y, z} = 4'b1101;
        4'b0100:    {w, x, y, z} = 4'b1100;
        4'b0101:    {w, x, y, z} = 4'b1011;
        4'b0110:    {w, x, y, z} = 4'b1010;
        4'b0111:    {w, x, y, z} = 4'b1001;

        4'b1000:    {w, x, y, z} = 4'b1000;
        4'b1001:    {w, x, y, z} = 4'b0111;
        4'b1010:    {w, x, y, z} = 4'b0110;
        4'b1011:    {w, x, y, z} = 4'b0101;
        4'b1100:    {w, x, y, z} = 4'b0100;
        4'b1101:    {w, x, y, z} = 4'b0011;
        4'b1110:    {w, x, y, z} = 4'b0010;
        4'b1111:    {w, x, y, z} = 4'b0001;
    endcase
endmodule
```

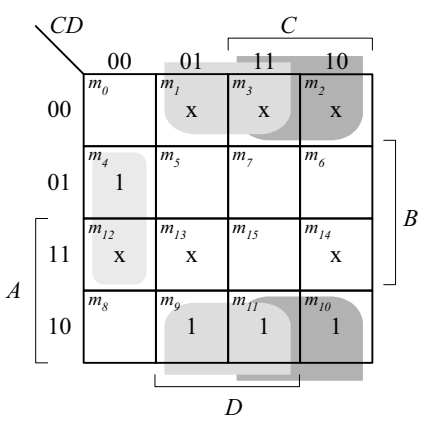
Alternative model:

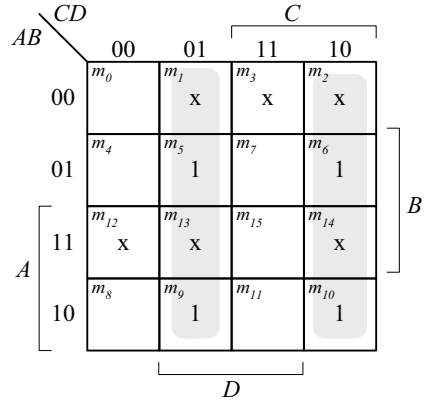
```
module Prob_4_7(output w, x, y, z, input A, B, C, D);
  assign w = A;
  assign x = A ^ B);
  assign y = x ^ C;
  assign z = y ^ D;
endmodule
```

4.8

ABCD	*wxyz*
0000	0000
0001	0001
0011	0010
0010	0011
0110	0100
0111	0101
0101	0110
0100	0111
1100	1000
1101	1001
1111	1010
1110	1011
1010	1100
1011	1101
1001	1110
1000	1111







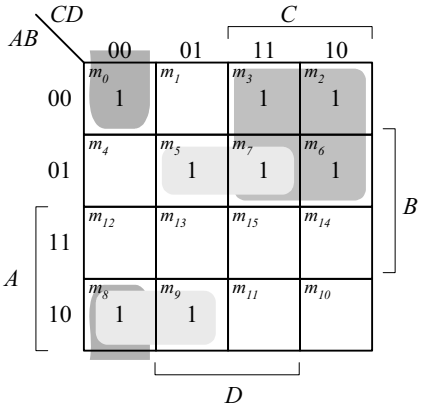
$w = AB+AC'D'$
$x = B'C + B'D + BC'D'$
$y = CD' + C'D$
$z = D$

Alternative model:

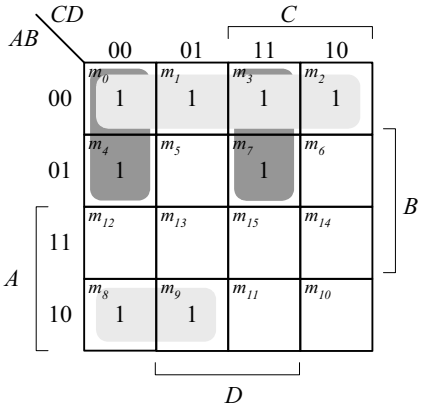
```
module Prob_4_8(output w, x, y, z, input A, B, C, D);
  assign w = (A&B) | (A & (~C)) & (~D) ;
  assign x = ( (~B) & C)  | ((~B) & D) | (B & (~C)) & (~D);
  assign y = C ^ D;
  assign z = D;
endmodule
```

4.9

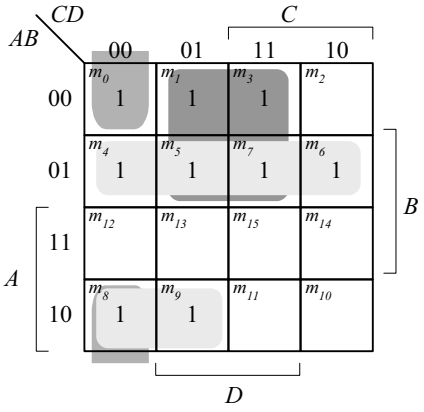
ABCD	a	b	c	d	e	f	g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1



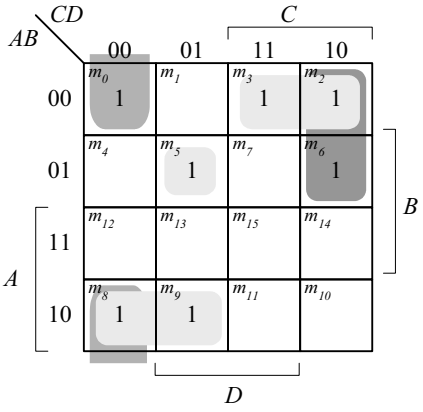
$a = A'C + A'BD + B'C'D' + AB'C'$



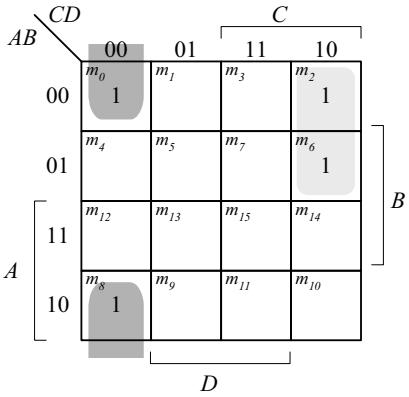
$b = A'B' + A'C'D' + A'CD + AB'C'$



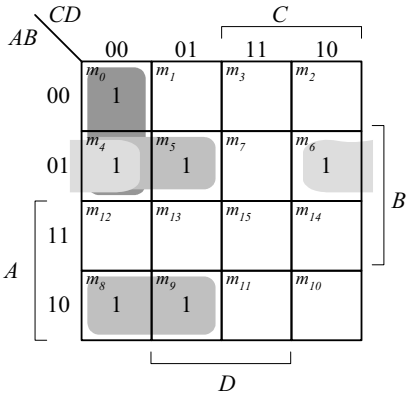
$c = A'B + A'D + B'C'D' + AB'C'$



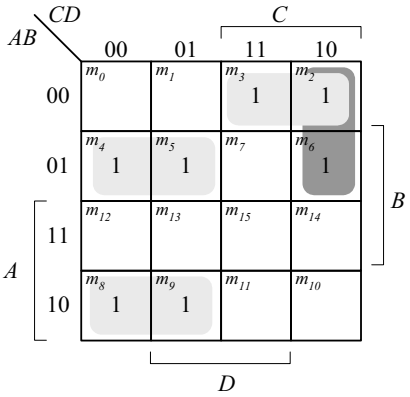
$d = A'CD' + A'B'C + B'C'D' + AB'C' + A'BC'D$



$e = A'CD' + B'C'D'$



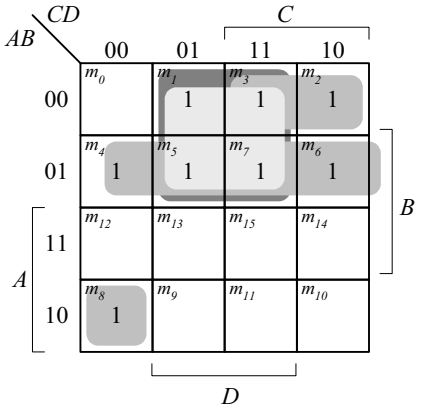
$f = A'BC' + A'C'D' + A'BD + AB'C'$



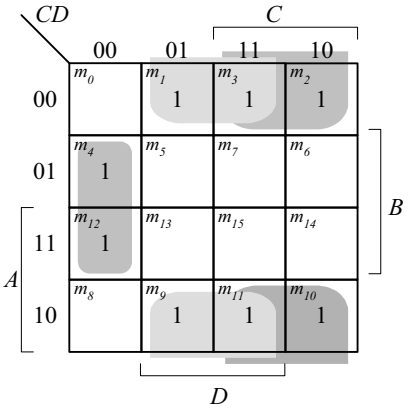
$g = A'CD' + A'B'C + A'BC' + AB'C'$

4.10

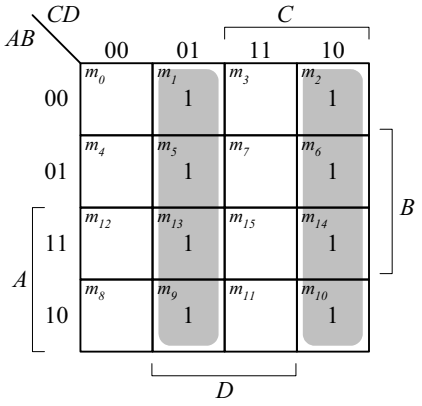
ABCD	wxyz
0000	0000
0001	1111
0010	1110
0011	1101
0100	1100
0101	1011
0110	1001
0111	1000
1000	1000
1001	0111
1010	0110
1011	0101
1100	0100
1101	0011
1110	0010
1111	0001



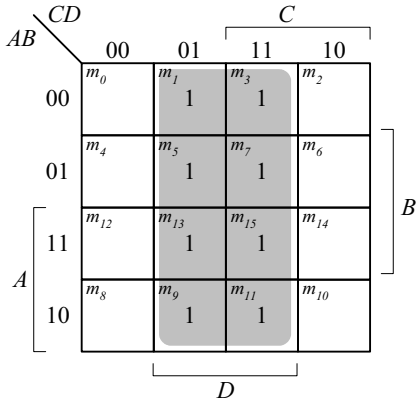
$w = A'(B + C + D) + AB'C'D'$
$= A \oplus (B + C + D)$



$x = B'(C + D) + CB'D'$
$= B \oplus (C + D)$



$y = CD' + C'D = C \oplus D$

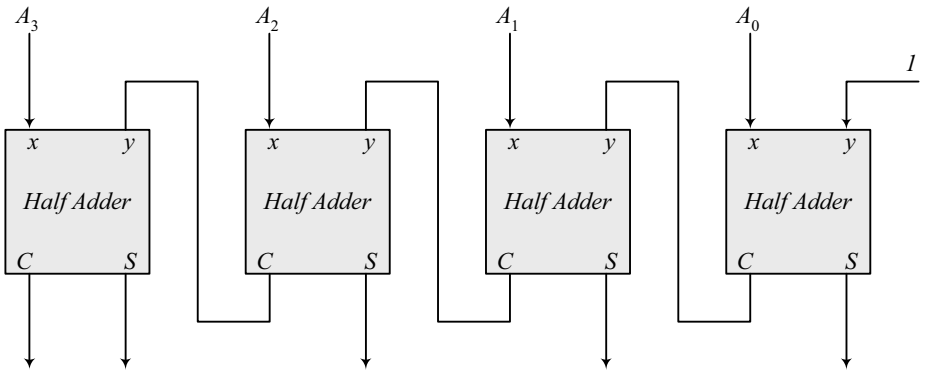


$z = D$

For a 5-bit 2's complementer with input E and output v:

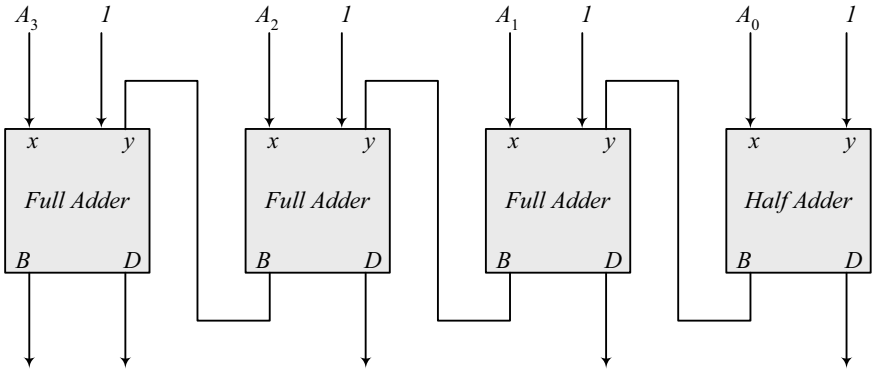
$$v = E \oplus (A + B + C + D)$$

4.11 **(a)**



Note: 5-bit output

(b)



Note: To decrement the 4-bit number, add -1 to the number. In 2's complement format (add F_h) to the number. An attempt to decrement 0 will assert the borrow bit. For waveforms, see solution to Problem 4.52.

4.12

(a)

x y	B D
0 0	0 0
0 1	1 1
1 0	0 1
1 1	0 0

$D = x'y + xy'$
$B = x'y$

(b)

x y B_{in}	B D
0 0 0	0 0
0 0 1	1 1
0 1 0	1 1
0 1 1	1 0
1 0 0	0 1
1 0 1	0 0
1 1 0	0 0
1 1 1	1 1

$Diff = x \oplus y \oplus z$
$B_{out} = x'y + x'z + yz$

4.13

	Sum	C	V
(a)	1101	0	1
(b)	0001	1	1
(c)	0100	1	0
(d)	1011	0	1
(e)	1111	0	0

4.14 xor AND OR XOR

10 + 5 + 5 + 10 = 30 ns

4.15 $C_4 = G_3 + P_3C_3 = G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0)$

$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

4.16 **(a)**

$(C'G'_i + p'_i)' = (C_i + G_i)P_i = G_iP_i + P_iC_i$
$= A_iB_i(A_i + B_i) + P_iC_i$
$= A_iB_i + P_iC_i = G_i + P_iC_i$
$= A_iB_i + (A_i + B_i)C_i = A_iB_i + A_iC_i + B_iC_i = C_{i+1}$
$(P_iG'_i) \oplus C_i = (A_i + B_i)(A_iB_i)' \oplus C_i = (A_i + Bi)(A'_i + B'_i) \oplus C_i$
$= (A'_iB_i + A_iB'_i) \oplus C_i = A_i \oplus B_i \oplus C_i = S_i$

(b)

Output of NOR gate $= (A_0 + B_0)' = P'_0$
Output of NAND gate $= (A_0B_0)' = G'_0$
$S_1 = (P_0G'_0) \oplus C_0$
$C_1 = (C'_0G'_0 + P'_0)'$ as defined in part (a)

4.17 **(a)**

$(C'_iG'_i + P'_i)' = (C_i + G_i)P_i = G_iP_i + P_iC_i = A_iB_i(A_i + B_i) + P_iC_i$
$= A_iB_i + P_iC_i = G_i + P_iC_i$
$= A_iB_i + (A_i + B_i)C_i = A_iB_i + A_iC_i + B_iC_i = C_{i+1}$

$(P_iG'_i) \oplus C_i = (A_i + B_i)(A_iB_i)' \oplus C_i = (A_i + B_i)(A'_i + B'_i) \oplus C_i$
$= (A'_iB_i + A_iB'_i) \oplus C_i = A_i \oplus B_i \oplus C_i = S_i$

(b)

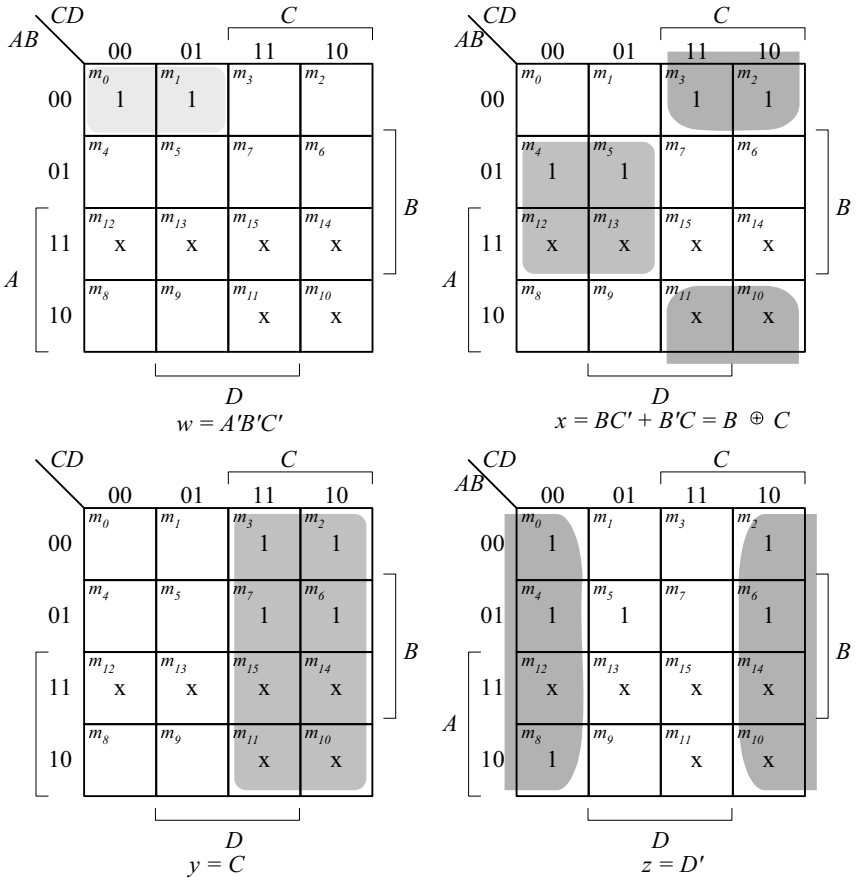
Output of NOR gate $= (A_0 + B_0)' = P'_0$
Output of NAND gate $= (A_0B_0)' = G'_0$

$S_0 = (P_0G'_0) \oplus C_0$
$C_1 = (C'_0G'_0 + P'_0)'$ as defined in part (a)

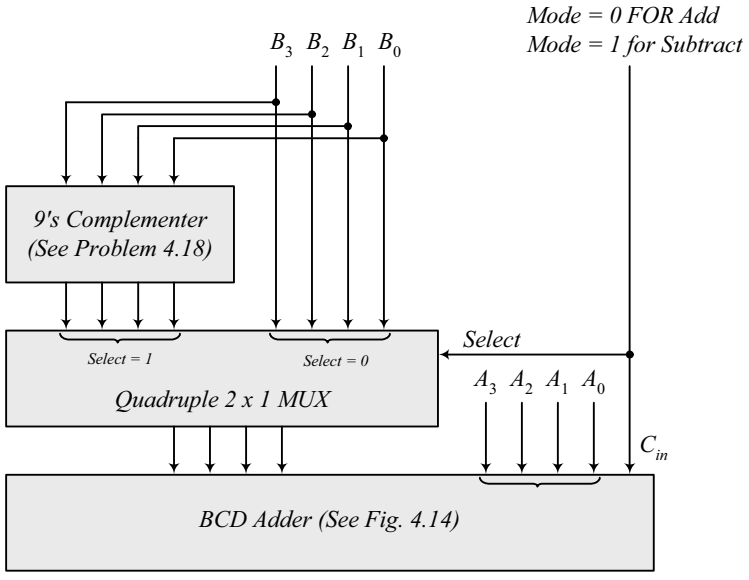
4.18

Inputs *ABCD*	Outputs *wxyz*
0000	1001
0001	1000
0010	0111
0011	0110
0100	0101
0101	0100
0110	0011
0111	0010
1000	0001
1001	0000

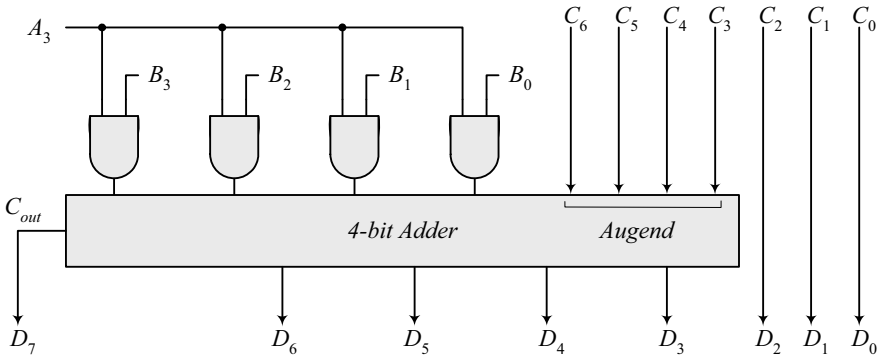
$d(A, b, c, d) = \Sigma(10, 11, 12, 13, 14, 15)$



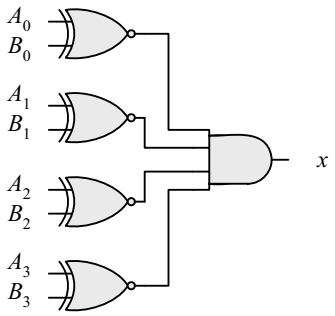
4.19



4.20 Combine the following circuit with the 4-bit binary multiplier circuit of Fig. 4.16.



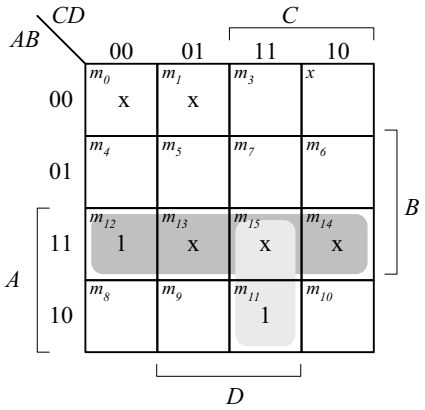
4.21



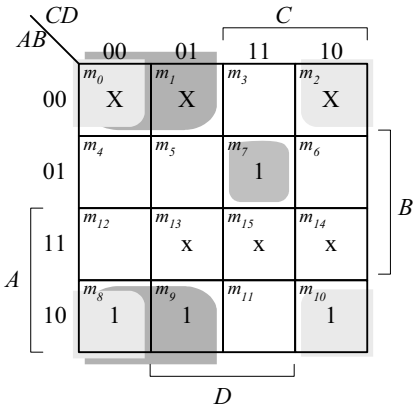
$$x = (A_0 \oplus B_0)'(A_1 \oplus B_1)'(A_2 \oplus B_2)'(A_3 \oplus B_3)'$$

4.22

XS-3 *ABCD*	Binary *wxyz*
0011	0000
0100	0001
0101	0010
0110	0011
0111	0100
1000	0101
1001	0110
1010	0111
1011	1000
1100	1001



$w = AB + ACD$

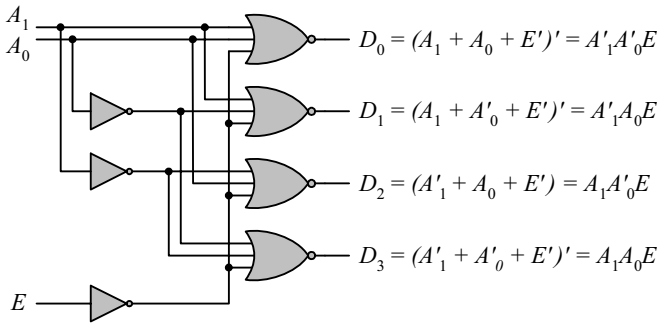


$x = B'C' + B'D' + BCD$

$y = C'D + CD'$

$z = D'$

4.23

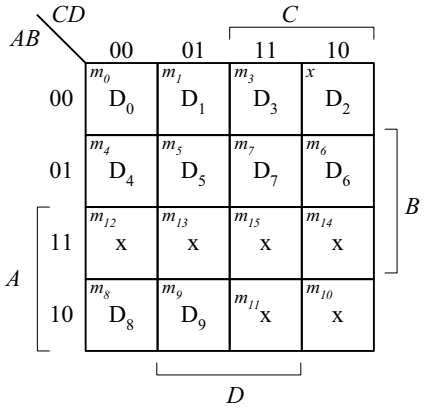


4.24

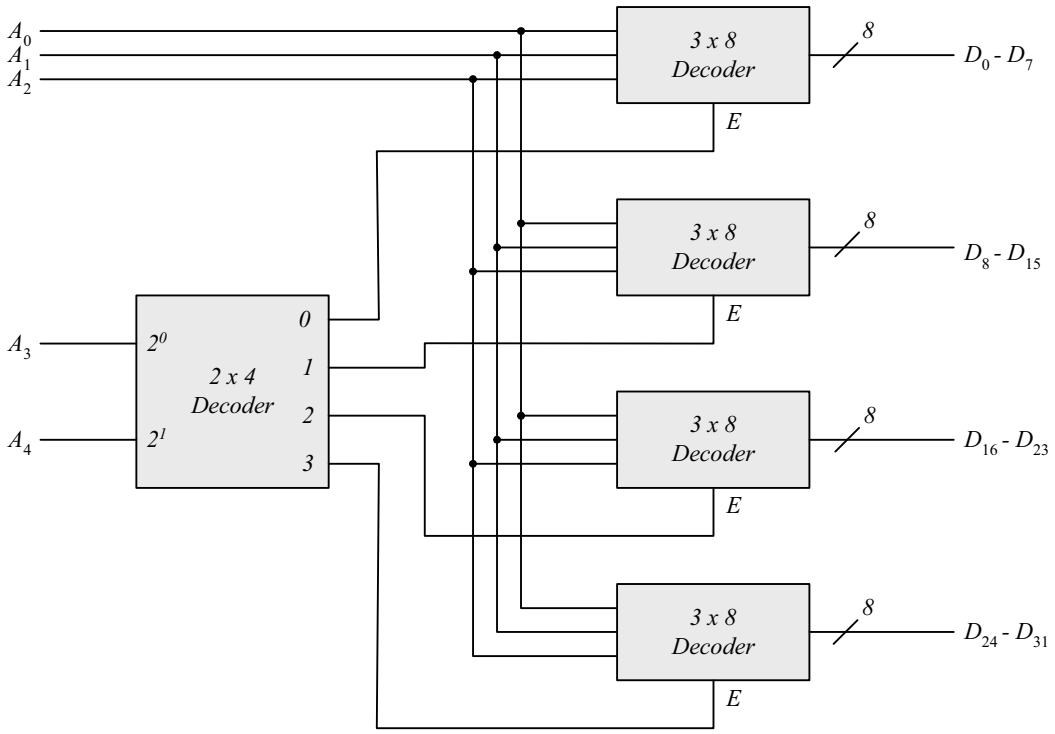
Inputs: *A, B, C, D*

$D_0 = A'B'C'D'$
$D_1 = A'B'C'D$
$D_2 = B'CD'$
$D_3 = B'CD$
$D_4 = BC'D'$

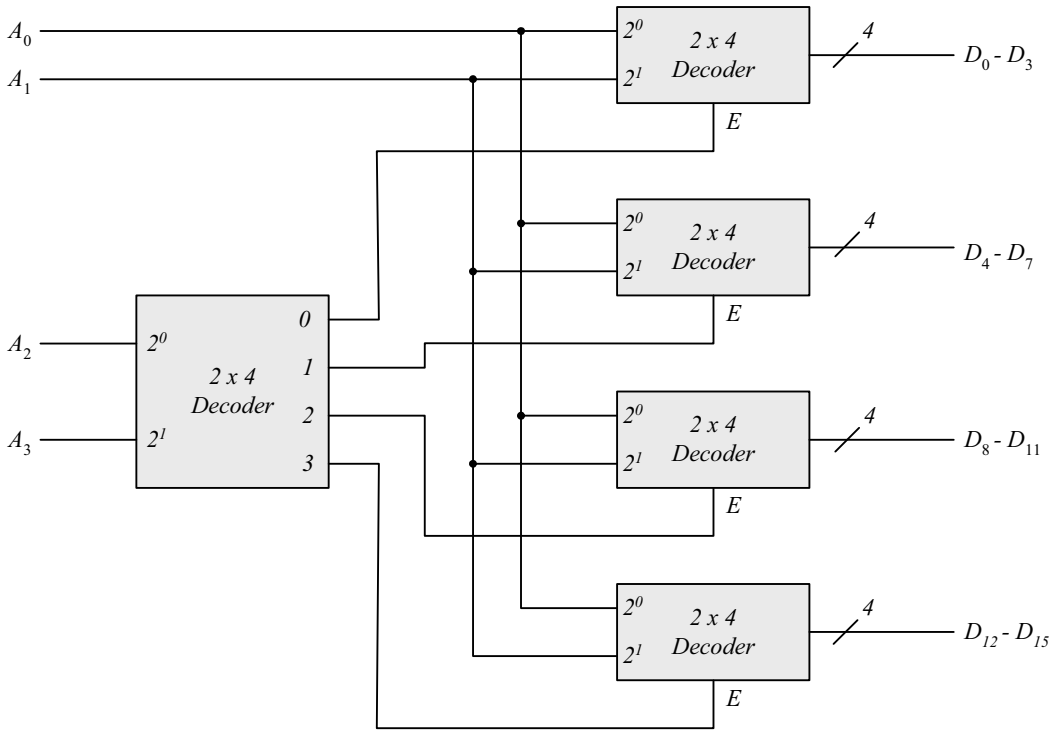
Outputs: $D_0, D_1, \ldots D_9$
$D_5 = BC'D$
$D_6 = BCD'$
$D_7 = BCD$
$D_8 = AD'$
$D_9 = AD$



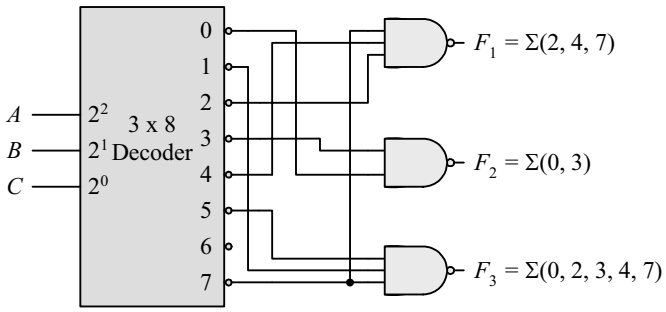
4.25



4.26



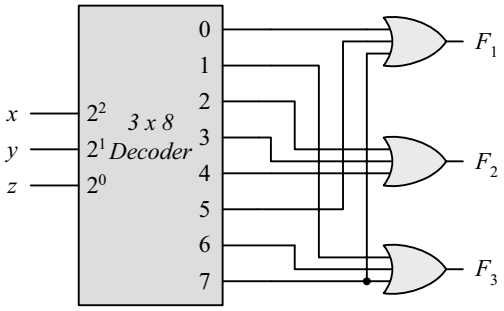
4.27



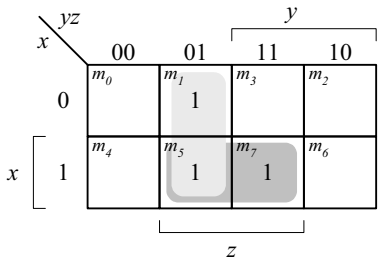
$(F'_3 = \Sigma(1, 5, 6))$

4.28 (a)

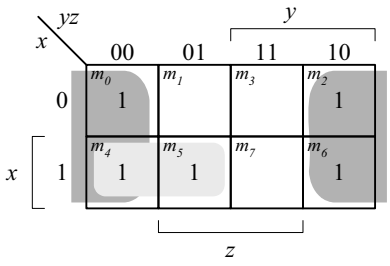
$F_1 = x(y + y')z = x'y'z' = \Sigma(0, 5, 7)$
$F_2 = xy'z' + x'y + x'y(z + z') = \Sigma(2, 3, 4)$
$F_3 = x'y'z + xy(z + z') = \Sigma(1, 6, 7)$



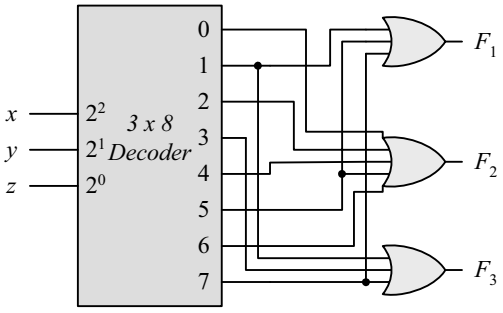
(b)

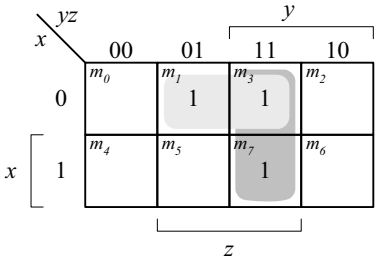


$F_1 = y'z + xz = \Sigma(1, 5, 7)$



$F_2 = y'z' + xy' + yz' = \Sigma(0, 2, 4, 5, 6)$

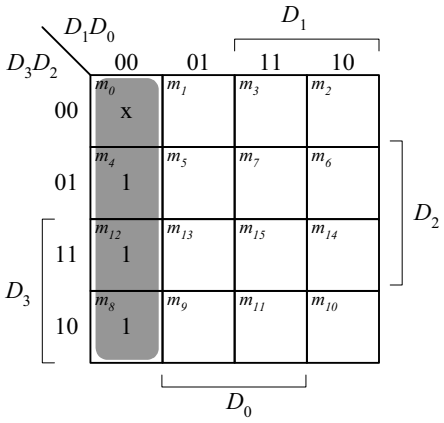




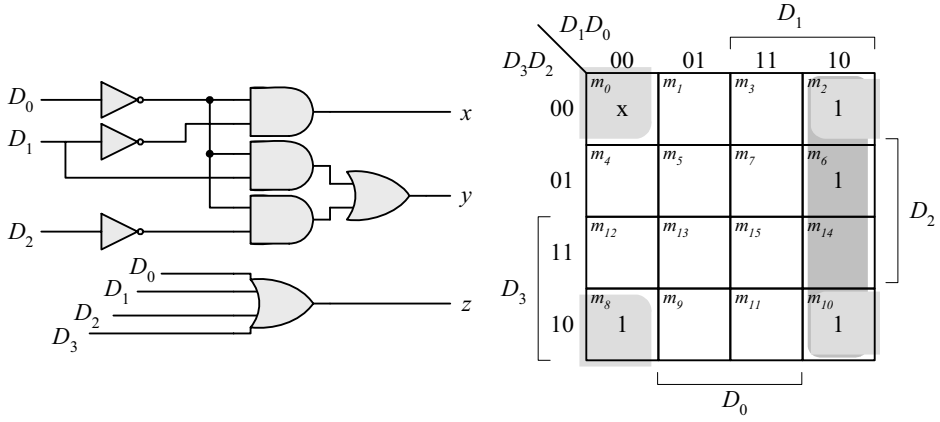
$F_3 = x'z + yz = \Sigma(1, 3, 7)$

4.29

Inputs $D_3\ D_2\ D_1\ D_0$	*Outputs* $X\ Y\ Z$
0 0 0 0	x x 0
x x x 1	0 0 1
x x 1 0	0 1 1
x 1 0 0	1 0 1
1 0 0 0	1 1 1



$$v = D_0 + D_1 + D_2 + D_3$$



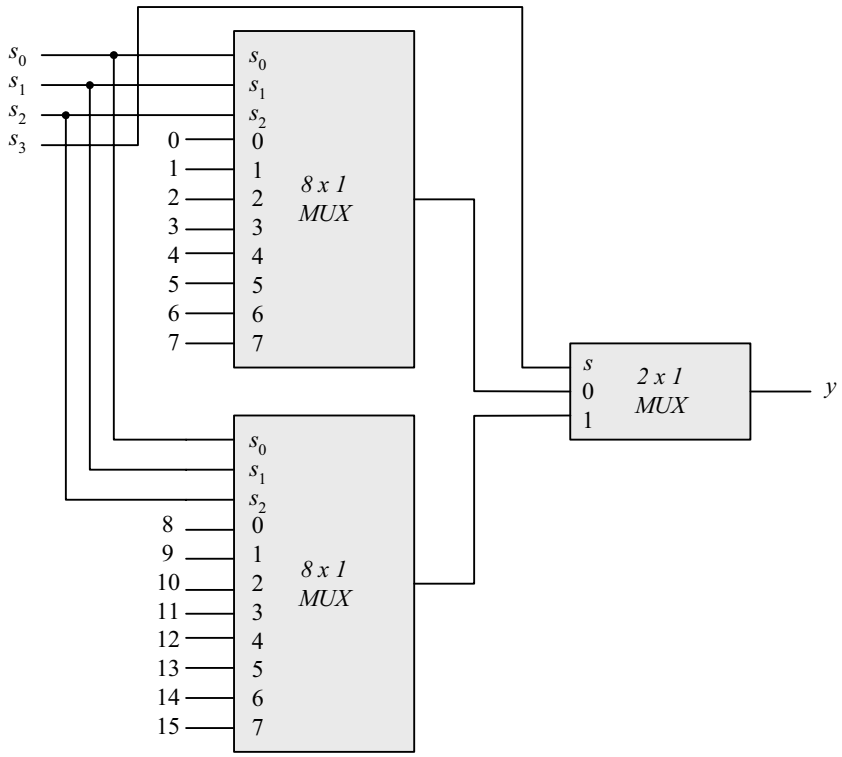
$$y = D'_0 D_1 + D'_0 D'_2$$

4.30

Inputs D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	*Outputs* $x\ y\ z$	V
0	0	0	0	0	0	0	0	x x x	0
1	0	0	0	0	0	0	0	0 0 0	1
x	1	0	0	0	0	0	0	0 0 1	1
x	x	1	0	0	0	0	0	0 1 0	1
x	x	x	1	0	0	0	0	0 1 1	1
x	x	x	x	1	0	0	0	1 0 0	1
x	x	x	x	x	1	0	0	1 0 1	1
x	x	x	x	x	x	1	0	1 0 0	1
x	x	x	x	x	x	x	1	1 1 1	1

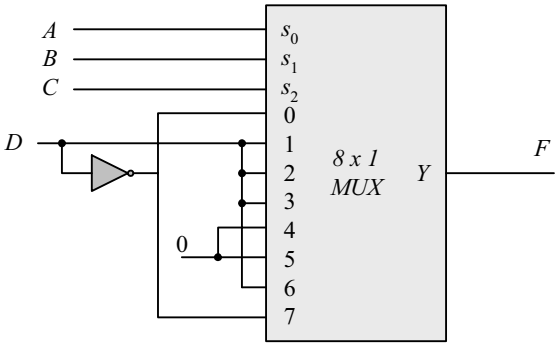
If $D_2 = 1$, $D_6 = 1$, all others = 0
Output xyz = 100 and V = 1

4.31



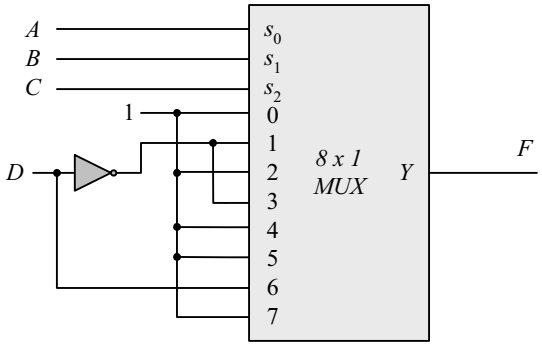
4.32 **(a)** $F = \Sigma\,(0, 2, 5, 7, 11, 14)$

Inputs ABCD	*F*	
0000	1	$F = D'$
0001	0	
0010	1	$F = D$
0011	0	
0100	0	$F = D$
0101	1	
0110	0	$F = D$
0111	1	
1000	0	$F = 0$
1001	0	
1010	0	$F = 0$
1011	0	
1100	0	$F = D$
1101	1	
1110	1	$F = D'$
1111	0	



(b) $F = \Pi\,(3, 8, 12) = (A' + B' + C + D)(A + B' + C' + D')(A + B + C' + D')$
$F' = ABC'D' + A'BCD + A'B'CD = \Sigma(12, 7, 3)$
$F = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 10, 11, 13, 14, 15)$

Inputs ABCD	F	
0000	1	F = 1
0001	1	
0010	1	F = D'
0011	0	
0100	1	F = 1
0101	1	
0110	1	F = D'
0111	0	
1000	1	F = 1
1001	1	
1010	1	F = 1
1011	1	
1100	0	F = D
1101	1	
1110	1	F = 1
1111	1	

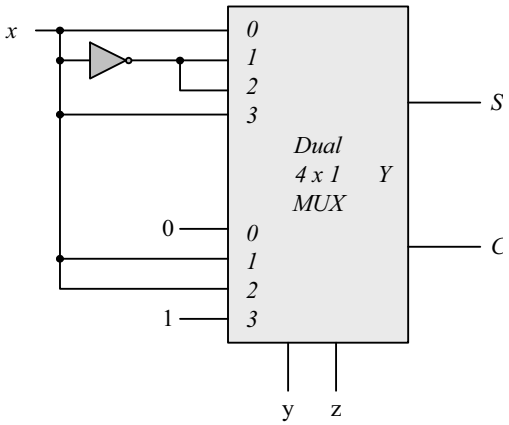


4.33

$S(x, y, z) = \Sigma(1, 2, 4, 7)$
$C(x, y, z) = \Sigma(3, 5, 6, 7)$

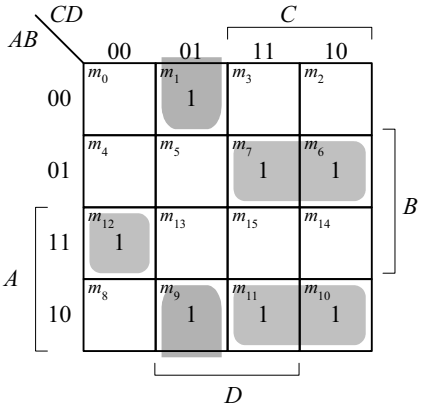
S	I_0	I_1	I_2	I_3
x′	0	1	2	3
x	4	5	6	7
	x	x′	x′	x

C	I_0	I_1	I_2	I_3
x′	0	1	2	3
x	4	5	6	7
	0	x′	x′	1



4.34 **(a)**

	A	B	C	D	F
$I_3 = 1$	0	1	1	0	1
	0	1	1	1	1
$I_5 = 1$	1	0	1	0	1
	1	0	1	1	1
$I_0 = D$	0	0	0	0	0
	0	0	0	1	1
$I_4 = D$	1	0	0	0	0
	1	0	0	1	1
$I_6 = D'$	1	1	0	0	1
	1	1	0	1	0



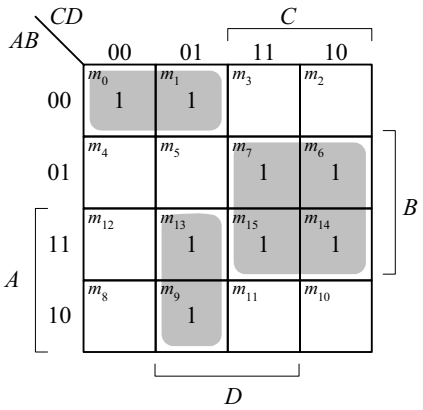
Other minterms = 0
since $I_1 = I_2 = I_7 = 0$

$F(A, B, C, D) = \Sigma(1, 6, 7, 9, 10, 11, 12)$

(b)

	A	B	C	D	F
$I_1 = 0$	0	0	1	0	0
	0	0	1	1	0
$I_2 = 0$	0	1	0	0	0
	0	1	0	1	0
$I_3 = 1$	0	1	1	0	1
	0	1	1	1	1
$I_7 = 1$	1	1	1	0	1
	1	1	1	1	1
$I_4 = D$	1	0	0	0	0
	1	0	0	1	1
$I_0 = D'$	0	0	0	0	1
	0	0	0	1	0
$I_6 = D'$	1	1	0	0	1
	1	1	0	1	0

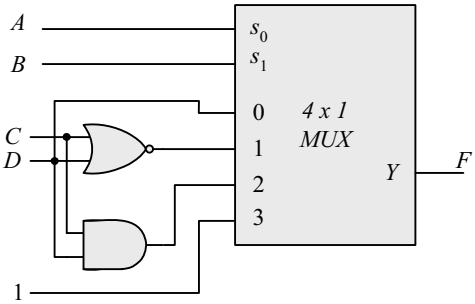
Other minterms = 0
since $I_1 = I_2 = 0$



$F(A, B, C, D) = \Sigma(0, 1, 6, 7, 9, 13, 14, 15)$

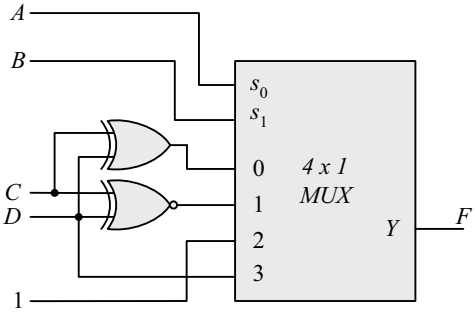
4.35 **(a)**

Inputs ABCD	F	
0000	0	
0001	1	$AB = 00$
0010	0	$F = D$
0011	1	
0100	1	
0101	0	$AB = 01$
0110	0	$F = C'D'$
0111	0	$= (C + D)'$
1000	0	
1001	0	$AB = 10$
1010	0	$F = CD$
1011	1	
1100	1	
1101	1	$AB = 11$
1110	1	$F = 1$
1111	1	



(b)

Inputs ABCD	F	
0000	0	
0001	1	$AB = 00$
0010	1	$F = C'D + CD'$
0011	0	
0100	1	
0101	0	$AB = 01$
0110	0	$F = C'D' + CD$
0111	1	
1000	1	
1001	1	$AB = 10$
1010	1	$F = 1$
1011	1	
1100	0	
1101	1	$AB = 11$
1110	0	$F = D$
1111	1	



4.36

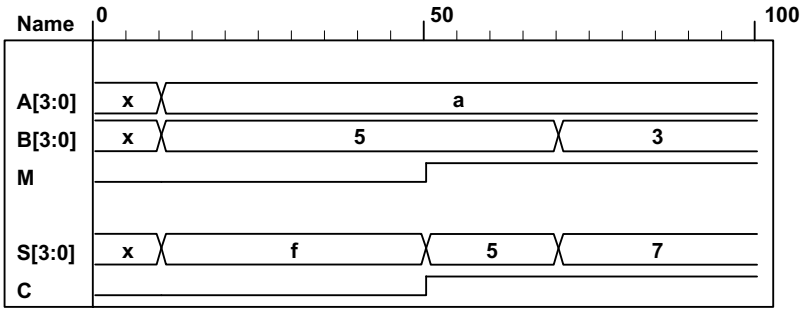
```
module priority_encoder_gates (output x, y, V, input D0, D1, D2, D3);  // V2001
  wire  w1, D2_not;
  not   (D2_not, D2);
  or    (x, D2, D3);
  or    (V, D0, D1, x);
  and   (w1, D2_not, D1);
  or    (y, D3, w1);
endmodule
```

Note: See Problem 4.45 for testbench)

4.37

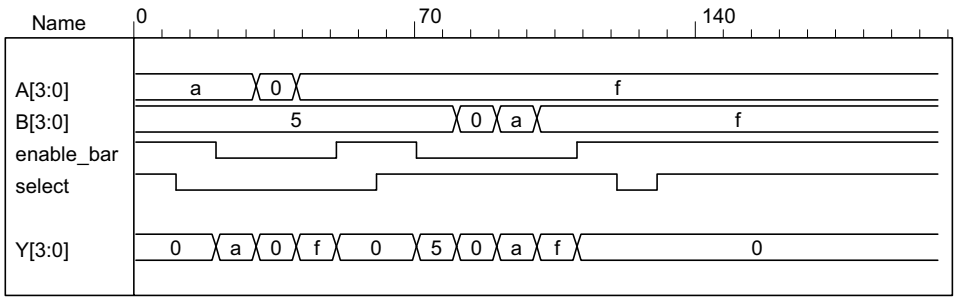
```
module Add_Sub_4_bit (
  output [3: 0] S,
  output C,
  input [3: 0] A, B,
  input M
);
  wire  [3: 0] B_xor_M;
  wire  C1, C2, C3, C4;
  assign C = C4;       // output carry
  xor (B_xor_M[0], B[0], M);
  xor (B_xor_M[1], B[1], M);
  xor (B_xor_M[2], B[2], M);
  xor (B_xor_M[3], B[3], M);
  // Instantiate full adders
  full_adder FA0 (S[0], C1, A[0], B_xor_M[0], M);
  full_adder FA1 (S[1], C2, A[1], B_xor_M[1], C1);
  full_adder FA2 (S[2], C3, A[2], B_xor_M[2], C2);
  full_adder FA3 (S[3], C4, A[3], B_xor_M[3], C3);
endmodule

module full_adder (output S, C, input x, y, z); // See HDL Example 4.2
  wire S1, C1, C2;
  // instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule

module half_adder (output S, C, input x, y);        // See HDL Example 4.2
  xor   (S, x, y);
  and   (C, x, y);
endmodule

module t_Add_Sub_4_bit ();
  wire [3: 0] S;
  wire C;
  reg [3: 0] A, B;
  reg M;

  Add_Sub_4_bit M0 (S, C, A, B, M);

  initial #100 $finish;
  initial fork
    #10 M = 0;
    #10 A = 4'hA;
    #10 B = 4'h5;
```

```
   #50 M = 1;
   #70 B = 4'h3;
  join
endmodule
```

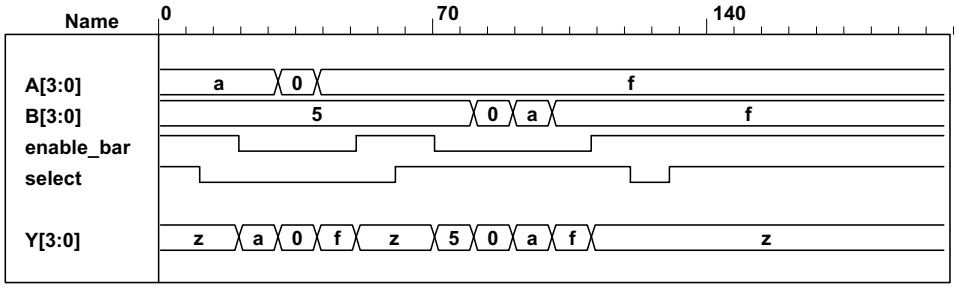


4.38

```
module quad_2x1_mux (                  // V2001
  input     [3: 0]   A, B,             // 4-bit data channels
  input              enable_bar, select,  // enable_bar is active-low)
  output    [3: 0]   Y                 // 4-bit mux output
);
  //assign Y = enable_bar ? 0 : (select ? B : A);          // Grounds output
  assign Y = enable_bar ? 4'bzzzz : (select ? B : A);    // Three-state output
endmodule
// Note that this mux grounds the output when the mux is not active.

module t_quad_2x1_mux ();
  reg      [3: 0] A, B, C;                 // 4-bit data channels
  reg             enable_bar, select;      // enable_bar is active-low)
  wire  [3: 0]   Y;                        // 4-bit mux

  quad_2x1_mux M0 (A, B, enable_bar, select, Y);

  initial #200 $finish;
  initial fork
    enable_bar = 1;
    select = 1;
    A = 4'hA;
    B = 4'h5;
    #10 select = 0;       // channel A
    #20 enable_bar = 0;
    #30 A = 4'h0;
    #40 A = 4'hF;
    #50 enable_bar = 1;
    #60 select = 1;       // channel B
    #70 enable_bar = 0;
    #80 B = 4'h00;
    #90 B = 4'hA;
    #100 B = 4'hF;
    #110 enable_bar = 1;
    #120 select = 0;
    #130 select = 1;
    #140 enable_bar = 1;
  join
endmodule
```



With three-state output:



4.39

```
// Verilog 1995
module Compare (A, B, Y);
  input     [3: 0]   A, B;   // 4-bit data inputs.
  output    [5: 0]   Y;      // 6-bit comparator output.
  reg       [5: 0]   Y;      // EQ, NE, GT, LT, GE, LE

  always @ (A or B)
    if (A==B)           Y = 6'b10_0011;        // EQ, GE, LE
    else if (A < B)     Y = 6'b01_0101;        // NE, LT, LE
    else                Y = 6'b01_1010;        // NE, GT, GE
endmodule


// Verilog 2001, 2005

module Compare (input [3: 0] A, B, output reg [5:0] Y);
  always @ (A, B)
    if (A==B)           Y = 6'b10_0011;        // EQ, GE, LE
    else if (A < B)     Y = 6'b01_0101;        // NE, LT, LE
    else                Y = 6'b01_1010;        // NE, GT, GE
endmodule
```

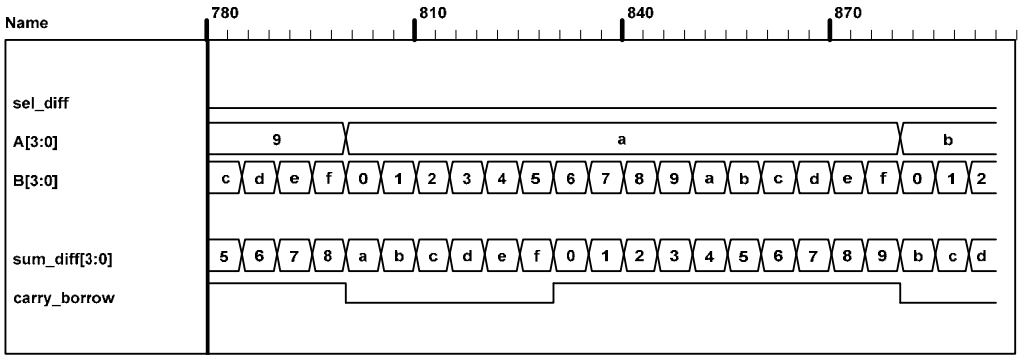
4.40

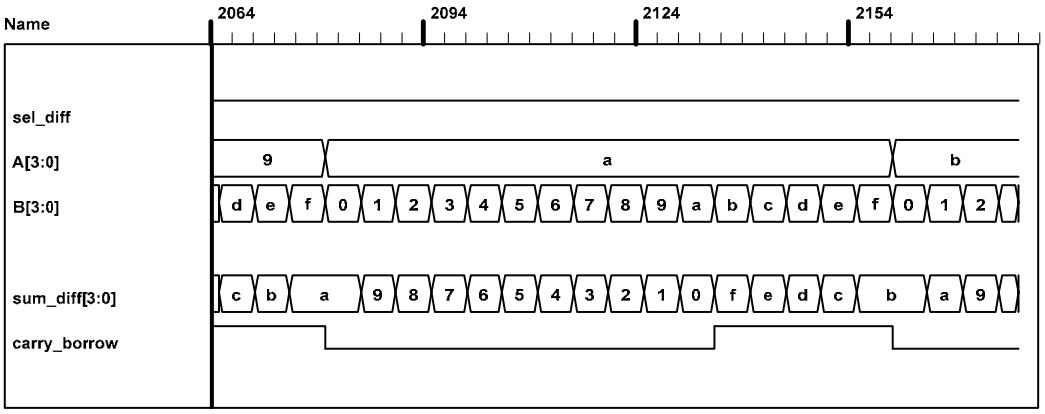
```
  module Prob_4_40 (
  output [3: 0]  sum_diff, output carry_borrow,
  input [3: 0] A, B, input sel_diff
  );

    assign   {carry_borrow, sum_diff} = sel_diff ? A - B : A + B;
endmodule

module t_Prob_4_40;
  wire  [3: 0] sum_diff;
  wire carry_borrow;
  reg [3:0] A, B;
```

```
  reg sel_diff;

  integer I, J, K;
  Prob_4_40 M0 (  sum_diff, carry_borrow,  A, B, sel_diff);
  initial #4000 $finish;
  initial begin
   for (I = 0; I < 2; I = I + 1) begin
     sel_diff = I;
     for (J = 0; J < 16; J = J + 1) begin
       A = J;
       for (K = 0; K < 16; K = K + 1) begin B = K; #5 ; end
     end
   end
  end
endmodule
```

4.41

```
  module Prob_4_41 (
  output reg [3: 0]  sum_diff, output reg carry_borrow,
  input [3: 0] A, B, input sel_diff
  );

    always @ (A, B, sel_diff)
    {carry_borrow, sum_diff} = sel_diff ? A - B : A + B;

endmodule


module t_Prob_4_41;
  wire  [3: 0] sum_diff;
  wire carry_borrow;
  reg [3:0] A, B;
  reg sel_diff;

  integer I, J, K;
  Prob_4_46 M0 (  sum_diff, carry_borrow,  A, B, sel_diff);
  initial #4000 $finish;
  initial begin
   for (I = 0; I < 2; I = I + 1) begin
     sel_diff = I;
     for (J = 0; J < 16; J = J + 1) begin
       A = J;
       for (K = 0; K < 16; K = K + 1) begin B = K; #5 ; end
     end
   end
  end
endmodule
```





4.42 **(a)**

```
module Xs3_Gates (input A, B, C, D, output w, x, y, z);
  wire  B_bar, C_or_D_bar;
  wire  CD, C_or_D;
  or    (C_or_D, C, D);
  not   (C_or_D_bar, C_or_D);
  not   (B_bar, B);
  and   (CD, C, D);
  not   (z, D);
  or    (y, CD,  C_or_D_bar);
  and   (w1, C_or_D_bar, B);
  and   (w2, B_bar, C_or_D);
  and   (w3, C_or_D, B);
  or    (x, w1, w2);
  or    (w, w3, A);
endmodule
```

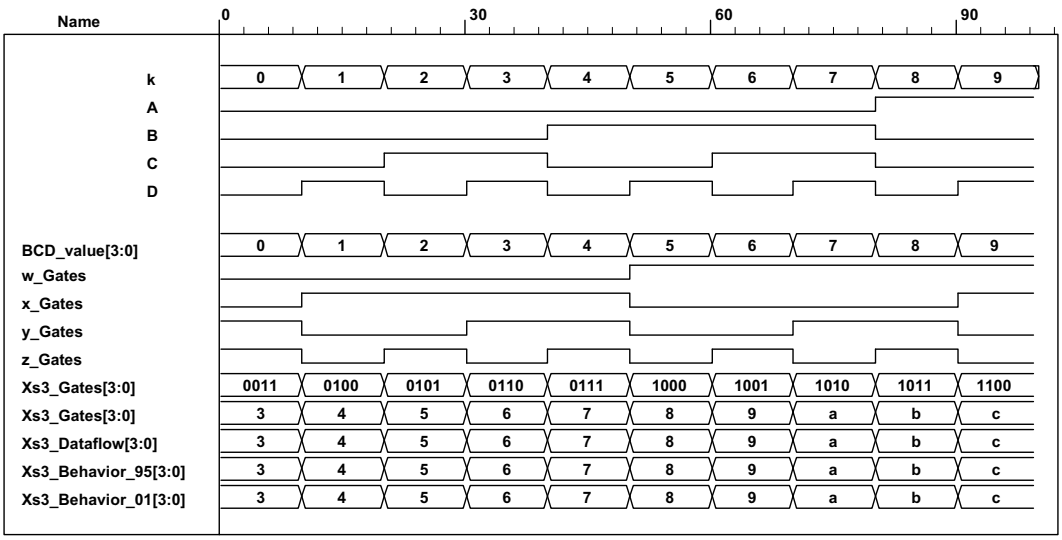
(b)

```
module Xs3_Dataflow (input A, B, C, D, output w, x, y, z);
assign {w, x, y, z} = {A, B, C, D} + 4'b0011;
endmodule
```

(c)

```
module Xs3_Behavior_95 (A, B, C, D, w, x, y, z);
  input     A, B, C, D;
  output    w, x, y, z;
  reg   w, x, y, z;

 always @ (A or B or C or D) begin {w, x, y, z} = {A, B, C, D} + 4'b0011; end
endmodule

module Xs3_Behavior_01 (input A, B, C, D, output reg w, x, y, z);
```

```
  always @ (A, B, C, D) begin {w, x, y, z} = {A, B,C, D} + 4'b0011; end
endmodule

module t_Xs3_Converters ();
  reg A, B, C, D;
  wire w_Gates, x_Gates, y_Gates, z_Gates;
  wire w_Dataflow, x_Dataflow, y_Dataflow, z_Dataflow;
  wire w_Behavior_95, x_Behavior_95, y_Behavior_95, z_Behavior_95;
  wire w_Behavior_01, x_Behavior_01, y_Behavior_01, z_Behavior_01;
  integer k;
  wire [3: 0] BCD_value;
  wire [3: 0] Xs3_Gates = {w_Gates, x_Gates, y_Gates, z_Gates};
  wire [3: 0] Xs3_Dataflow = {w_Dataflow, x_Dataflow, y_Dataflow, z_Dataflow};
  wire [3: 0] Xs3_Behavior_95 = {w_Behavior_95, x_Behavior_95, y_Behavior_95, z_Behavior_95};
  wire [3: 0] Xs3_Behavior_01 = {w_Behavior_01, x_Behavior_01, y_Behavior_01, z_Behavior_01};

  assign BCD_value = {A, B, C, D};
  Xs3_Gates   M0 (A, B, C, D, w_Gates, x_Gates, y_Gates, z_Gates);
  Xs3_Dataflow    M1 (A, B, C, D, w_Dataflow, x_Dataflow, y_Dataflow, z_Dataflow);
  Xs3_Behavior_95 M2 (A, B, C, D, w_Behavior_95, x_Behavior_95, y_Behavior_95, z_Behavior_95);
  Xs3_Behavior_01 M3 (A, B, C, D, w_Behavior_01, x_Behavior_01, y_Behavior_01, z_Behavior_01);

  initial #200 $finish;
  initial begin
    k = 0;
    repeat (10) begin {A, B, C, D} = k; #10 k = k + 1; end
  end
endmodule
```

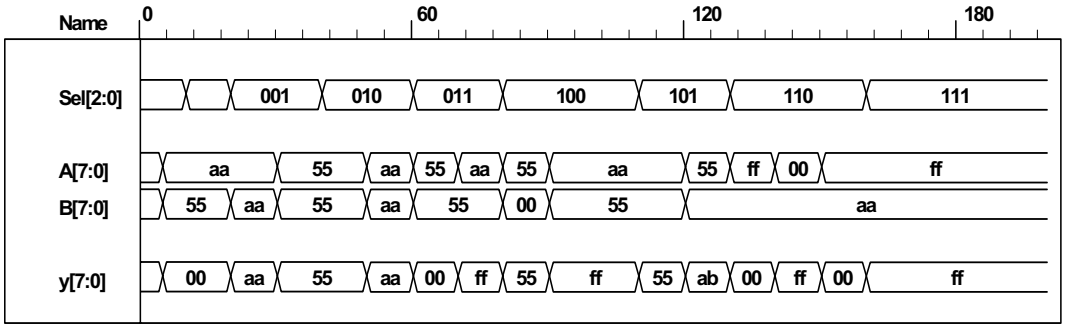


4.43 Two-channel mux with 2-bit data paths, enable, and three-state output.

4.44

```
module ALU (output reg [7: 0] y, input [7: 0] A, B, input [2: 0] Sel);
  always @ (A, B, Sel) begin
    y = 0;
    case (Sel)
      3'b000:    y = 8'b0;
      3'b001:    y = A & B;
      3'b010:    y = A | B;
      3'b011:    y = A ^ B;
```

```
    3'b100:    y = A + B;
    3'b101:    y = A - B;
    3'b110:    y = ~A;
    3'b111:    y = 8'hFF;
   endcase
  end

endmodule

module t_ALU ();
  wire[7: 0]y;
  reg [7: 0] A, B;
  reg [2: 0] Sel;

  ALU M0 (y, A, B, Sel);

initial #200 $finish;
  initial fork
     #5 begin A = 8'hAA; B = 8'h55; end      // Expect y = 8'd0
    #10 begin Sel = 3'b000; A = 8'hAA; B = 8'h55; end // y = 8'b000   Expect y = 8'd0
    #20 begin Sel = 3'b001; A = 8'hAA; B = 8'hAA; end // y = A & B   Expect y = 8'hAA = 8'1010_1010
    #30 begin Sel = 3'b001; A = 8'h55; B = 8'h55; end  // y = A & B   Expect y = 8'h55 = 8'b0101_0101
    #40 begin Sel = 3'b010; A = 8'h55; B = 8'h55; end  // y = A | B   Expect y = 8'h55 = 8'b0101_0101
    #50 begin Sel = 3'b010; A = 8'hAA; B = 8'hAA; end // y = A | B   Expect y = 8'hAA = 8'b1010_1010
    #60 begin Sel = 3'b011; A = 8'h55; B = 8'h55; end  // y = A ^ B   Expect y = 8'd0
    #70 begin Sel = 3'b011; A = 8'hAA; B = 8'h55; end // y = A ^ B   Expect y = 8'hFF = 8'b1111_1111
    #80 begin Sel = 3'b100; A = 8'h55; B = 8'h00; end  // y = A + B   Expect y = 8'h55 = 8'b0101_0101
    #90 begin Sel = 3'b100; A = 8'hAA; B = 8'h55; end // y = A + B   Expect y = 8'hFF = 8'b1111_1111
  #110 begin Sel = 3'b101; A = 8'hAA; B = 8'h55; end // y = A – B   Expect y = 8'h55 = 8'b0101_0101
  #120 begin Sel = 3'b101; A = 8'h55; B = 8'hAA; end // y = A – B   Expect y = 8'hab = 8'b1010_1011
  #130 begin Sel = 3'b110; A = 8'hFF; end            // y = ~A      Expect y = 8'd0
  #140 begin Sel = 3'b110; A = 8'd0; end             // y = ~A      Expect y = 8'hFF = 8'b1111_1111
  #150 begin Sel = 3'b110; A = 8'hFF; end            // y = ~A      Expect y = 8'd0
  #160 begin Sel = 3'b111;   end                     // y = 8'hFF   Expect y = 8'hFF = 8'b1111_1111
  join
endmodule
```

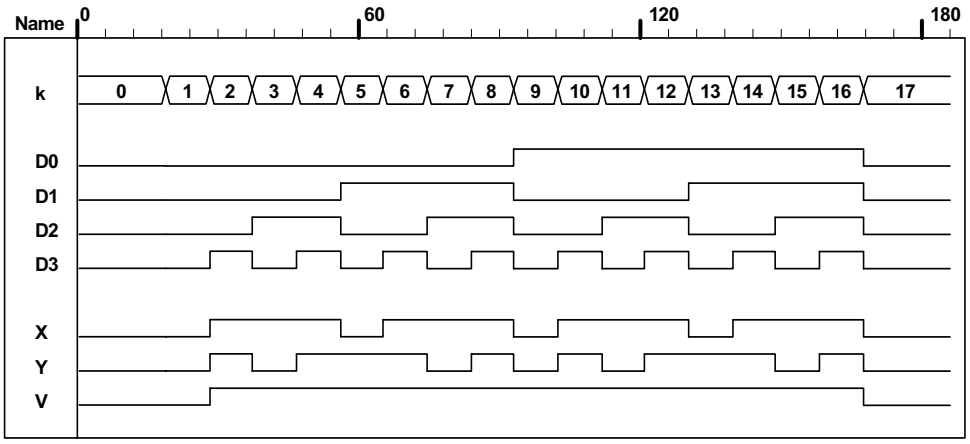


Note that the subtraction operator performs 2's complement subtraction. So 8'h55 – 8'hAA adds the 2's complement of 8'hAA to 8'h55 and gets 8'hAB. The sign bit is not included in the model, but hand calculation shows that the 9th bit is 1, indicating that the result of the operation is negative. The magnitude of the result can be obtained by taking the 2's complement of 8'hAB.

4.45

```
module priority_encoder_beh (output reg X, Y, V, input D0, D1, D2, D3);   // V2001
  always @ (D0, D1, D2, D3) begin
    X = 0;
    Y = 0;
```

```
    V = 0;
    casex ({D0, D1, D2, D3})
      4'b0000:   {X, Y, V} = 3'bxx0;
      4'b1000:   {X, Y, V} = 3'b001;
      4'bx100:   {X, Y, V} = 3'b011;
      4'bxx10:   {X, Y, V} = 3'b101;
      4'bxxx1:   {X, Y, V} = 3'b111;
      default:   {X, Y, V} = 3'b000;
    endcase
  end
endmodule

module t_priority_encoder_beh ();   // V2001
  wire X, Y, V;
  reg D0, D1, D2, D3;
  integer k;

  priority_encoder_beh M0 (X, Y, V, D0, D1, D2, D3);

  initial #200 $finish;
  initial begin
    k = 32'bx;
    #10 for (k = 0; k <= 16; k = k + 1)  #10 {D0, D1, D2, D3} = k;
  end
endmodule
```



4.46 **(a)**

$F = \Sigma(0, 2, 5, 7, 11, 14)$

See code below.

(b) From prob 4.32:

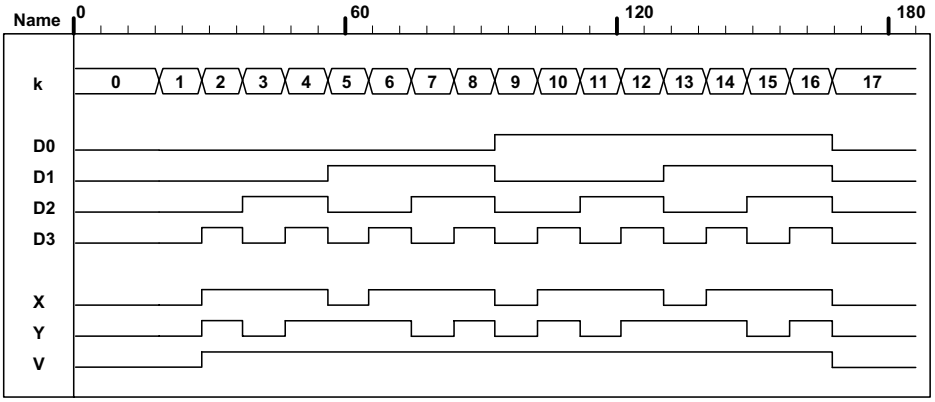
$F = \Pi(3, 8, 12) = (A' + B' + C + D)(A + B' + C' + D')(A + B + C' + D')$

$F' = ABC'D' + A'BCD + A'B'CD = \Sigma(12, 7, 3)$

$F = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 10, 11, 13, 14, 15)$

```
module Prob_4_46a (output F, input A, B, C, D);
assign F = (~A&~B&~C&~D) | (~A&~B&C&~D) | (~A&B&~C&D) | (~A&B&C&D) | (A&~B&C&D) |
(A&B&C&~D);
endmodule


module Prob_4_46b (output F, input A, B, C, D);
assign F = (~A&~B&~C&~D) | (~A&~B&~C&D) | (~A&~B&C&~D) | (~A&B&~C&~D) | (~A&B&~C&D) |
```

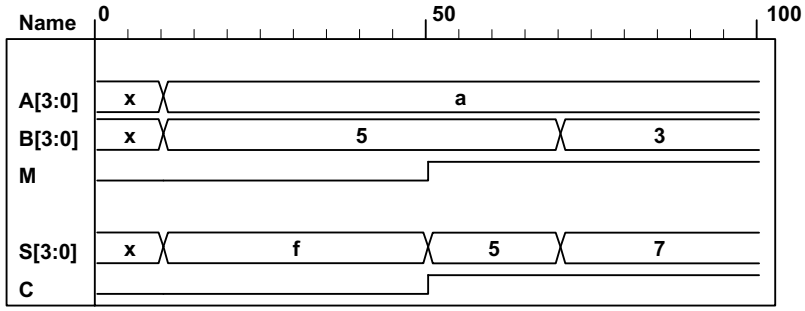
```
(~A&B&C&~D) | (A&~B&~C&~D) | (A&~B&~C&D) | (A&~B&C&~D) | (A&~B&C&D) | (A&B&~C&D) |
(A&B&C&~D) | (A&B&C&D);
endmodule

module t_Prob_4_46a ();
wire F_a, F_b;
reg A, B, C, D;
integer k;
  Prob_4_46a M0 (F_a, A, B, C, D);
  Prob_4_46b M1 (F_b, A, B, C, D);

  initial #200 $finish;
  initial begin
    k = 0;
    #10 repeat (15) begin {A, B, C, D} = k;  #10  k = k + 1; end
  end
endmodule
```



4.47

```
module Add_Sub_4_bit_Dataflow (
  output   [3: 0]       S,
  output        C, V,
  input    [3: 0]       A, B,
  input         M
);
  wire          C3;

  assign {C3, S[2: 0]} = A[2: 0] + ({M, M, M} ^ B[2: 0]) + M;
  assign {C, S[3]} = A[3] + M ^ B[3] + C3;
  assign V = C ^ C3;
endmodule

module t_Add_Sub_4_bit_Dataflow ();
  wire [3: 0] S;
  wire C, V;
  reg [3: 0] A, B;
  reg M;

  Add_Sub_4_bit_Dataflow M0 (S, C, V, A, B, M);

  initial #100 $finish;
  initial fork
    #10 M = 0;
    #10 A = 4'hA;
    #10 B = 4'h5;
```

```
    #50 M = 1;
    #70 B = 4'h3;
  join
endmodule
```



4.48

```
module ALU_3state (output [7: 0] y_tri, input [7: 0] A, B, input [2: 0] Sel, input En);
 reg [7: 0] y;
 assign y_tri = En ? y: 8'bz;
 always @ (A, B, Sel) begin
  y = 0;
  case (Sel)
    3'b000:    y = 8'b0;
    3'b001:    y = A & B;
    3'b010:    y = A | B;
    3'b011:    y = A ^ B;
    3'b100:    y = A + B;
    3'b101:    y = A - B;
    3'b110:    y = ~A;
    3'b111:    y = 8'hFF;
  endcase
 end

endmodule

module t_ALU_3state ();
 wire[7: 0] y;
 reg [7: 0] A, B;
 reg [2: 0] Sel;
 reg En;

 ALU_3state M0 (y, A, B, Sel, En);

initial #200 $finish;
 initial fork
    #5 En = 1;

    #5 begin A = 8'hAA; B = 8'h55; end      // Expect y = 8'd0
   #10 begin Sel = 3'b000; A = 8'hAA; B = 8'h55; end // y = 8'b000   Expect y = 8'd0
   #20 begin Sel = 3'b001; A = 8'hAA; B = 8'hAA; end // y = A & B    Expect y = 8'hAA = 8'1010_1010
   #30 begin Sel = 3'b001; A = 8'h55; B = 8'h55; end  // y = A & B    Expect y = 8'h55 = 8'b0101_0101
   #40 begin Sel = 3'b010; A = 8'h55; B = 8'h55; end  // y = A | B Expect y = 8'h55 = 8'b0101_0101
   #50 begin Sel = 3'b010; A = 8'hAA; B = 8'hAA; end // y = A | B     Expect y = 8'hAA = 8'b1010_1010
   #60 begin Sel = 3'b011; A = 8'h55; B = 8'h55; end  // y = A ^ B     Expect y = 8'd0
   #70 begin Sel = 3'b011; A = 8'hAA; B = 8'h55; end // y = A ^ B     Expect y = 8'hFF = 8'b1111_1111
   #80 begin Sel = 3'b100; A = 8'h55; B = 8'h00; end  // y = A + B    Expect y = 8'h55 = 8'b0101_0101
   #90 begin Sel = 3'b100; A = 8'hAA; B = 8'h55; end // y = A + B    Expect y = 8'hFF = 8'b1111_1111
   #100 En = 0;
   #115 En = 1;
```

```
  #110 begin Sel = 3'b101; A = 8'hAA; B = 8'h55; end  // y = A – B    Expect y = 8'h55 = 8'b0101_0101
   #120 begin Sel = 3'b101; A = 8'h55; B = 8'hAA; end // y = A – B    Expect y = 8'hab = 8'b1010_1011
   #130 begin Sel = 3'b110; A = 8'hFF; end            // y = ~A       Expect y = 8'd0
   #140 begin Sel = 3'b110; A = 8'd0; end             // y = ~A       Expect y = 8'hFF = 8'b1111_1111
   #150 begin Sel = 3'b110; A = 8'hFF; end            // y = ~A       Expect y = 8'd0
   #160 begin Sel = 3'b111;   end                     // y = 8'hFF    Expect y = 8'hFF = 8'b1111_1111
   join
  endmodule
```

4.49

```
// See Problem 4.1
module Problem_4_49_Gates (output F1, F2, input A, B, C, D);
  wire A_bar = !A;
  wire B_bar = !B;

  and (T1, B_bar, C);
  and (T2, A_bar, B);
  or (T3, A, T1);
  xor (T4, T2, D);
  or (F1, T3, T4);
  or (F2, T2, D);
endmodule

module Problem_4_49_Boolean_1 (output F1, F2, input A, B, C, D);
  wire A_bar = !A;
  wire B_bar = !B;
  wire T1 = B_bar && C;
  wire T2 = A_bar && B;
  wire T3 = A || T1;
  wire T4 = T2 ^ D;
  assign F1 = T3 || T4;
  assign F2 = T2 || D;
endmodule

module Problem_4_49_Boolean_2(output F1, F2, input A, B, C, D);
  assign F1 = A || (!B && C) || (B && (!D)) || (!B && D);
  assign F2 = ((!A) && B) || D;
endmodule

module t_Problem_4_49;
  reg A, B, C, D;
  wire F1_Gates, F2_Gates;
  wire F1_Boolean_1, F2_Boolean_1;
  wire F1_Boolean_2, F2_Boolean_2;

  Problem_4_48_Gates       M0 (F1_Gates, F2_Gates, A, B, C, D);
  Problem_4_48_Boolean_1   M1 (F1_Boolean_1, F2_Boolean_1, A, B, C, D);
  Problem_4_48_Boolean_2   M2 (F1_Boolean_2, F2_Boolean_2, A, B, C, D);

  initial #100 $finish;
  integer K;
  initial begin
    for (K = 0; K < 16; K = K + 1) begin {A, B, C, D} = K; #5; end
  end
endmodule
```

4.50

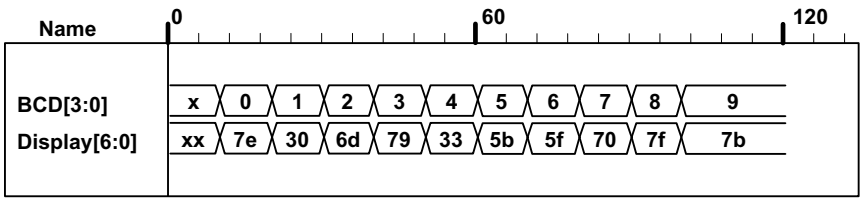
```
// See Problem 4.8 and Table 1.5.
// Verilog 1995

module Prob_4_50 (Code_8_4_m2_m1,  A, B, C, D);
output [3: 0]    Code_8_4_m2_m1;
input          A, B, C, D;
reg [3: 0]     Code_8_4_m2_m1;
...

// Verilog 2001, 2005
```

```
module Prob_4_50 (output reg [3: 0] Code_8_4_m2_m1,  input A, B, C, D);

 always @ (A, B, C, D)              // always @ (A or B or C or D)
  case ({A, B, C, D})
    4'b0000: Code_8_4_m2_m1 = 4'b0000;      // 0     0
    4'b0001: Code_8_4_m2_m1 = 4'b0111;      // 1     7
    4'b0010: Code_8_4_m2_m1 = 4'b0110;      // 2     6
    4'b0011: Code_8_4_m2_m1 = 4'b0101;      // 3     5
    4'b0100: Code_8_4_m2_m1 = 4'b0100;      // 4     4
    4'b0101: Code_8_4_m2_m1 = 4'b1011;      // 5     11
    4'b0110: Code_8_4_m2_m1 = 4'b1010;      // 6     10
    4'b0111: Code_8_4_m2_m1 = 4'b1001;      // 7     9
    4'b1000: Code_8_4_m2_m1 = 4'b1000;      // 8     8
    4'b1001: Code_8_4_m2_m1 = 4'b1111;      // 9     15

    4'b1010: Code_8_4_m2_m1 = 4'b0001;      // 10    1
    4'b1011: Code_8_4_m2_m1 = 4'b0010;      // 11    2
    4'b1100: Code_8_4_2_1   = 4'b0011;      // 12    3
    4'b1101: Code_8_4_2_1   = 4'b1100;      // 13    12
    4'b1110: Code_8_4_2_1   = 4'b1101;      // 14    13
    4'b1111: Code_8_4_2_1   = 4'b1110;      // 15    14
  endcase
endmodule

module t_Prob_4_50;
 wire [3: 0]  BCD;
 reg          A, B, C, D;
 integer      K;

 Prob_4_50 M0 ( BCD,  A, B, C, D);   // Unit under test (UUT)

 initial #100 $finish;
 initial begin
  for (K = 0; K < 16; K = K + 1) begin {A, B, C, D} = K; #5 ; end
 end
endmodule
```

4.51 Assume that that the LEDs are asserted when the output is high.

```
module Seven_Seg_Display_V2001 (
 output reg  [6: 0]   Display,
 input       [3: 0]   BCD
);

 //                      abc_defg
 parameter   BLANK       = 7'b000_0000;
 parameter   ZERO        = 7'b111_1110;      // h7e
 parameter   ONE         = 7'b011_0000;      // h30
 parameter   TWO         = 7'b110_1101;      // h6d
 parameter   THREE       = 7'b111_1001;      // h79
 parameter   FOUR        = 7'b011_0011;      // h33
 parameter   FIVE        = 7'b101_1011;      // h5b
 parameter   SIX         = 7'b101_1111;      // h5f
 parameter   SEVEN       = 7'b111_0000;      // h70
 parameter   EIGHT       = 7'b111_1111;      // h7f
 parameter   NINE        = 7'b111_1011;      // h7b

 always @ (BCD)
  case (BCD)
    0:      Display = ZERO;
```

```
    1:      Display = ONE;
    2:      Display = TWO;
    3:      Display = THREE;
    4:      Display = FOUR;
    5:      Display = FIVE;
    6:      Display = SIX;
    7:      Display = SEVEN;
    8:      Display = EIGHT;
    9:      Display = NINE;
   default:  Display = BLANK;
  endcase
endmodule

module t_Seven_Seg_Display_V2001 ();
  wire  [6: 0]  Display;
  reg   [3: 0]  BCD;

  parameter  BLANK      = 7'b000_0000;
  parameter  ZERO       = 7'b111_1110;      // h7e
  parameter  ONE        = 7'b011_0000;      // h30
  parameter  TWO        = 7'b110_1101;      // h6d
  parameter  THREE      = 7'b111_1001;      // h79
  parameter  FOUR       = 7'b011_0011;      // h33
  parameter  FIVE       = 7'b101_1011;      // h5b
  parameter  SIX        = 7'b001_1111;      // h1f
  parameter  SEVEN      = 7'b111_0000;      // h70
  parameter  EIGHT      = 7'b111_1111;      // h7f
  parameter  NINE       = 7'b111_1011;      // h7b

  initial #120 $finish;
  initial fork
    #10 BCD = 0;
    #20 BCD = 1;
    #30 BCD = 2;
    #40 BCD = 3;
    #50 BCD = 4;
    #60 BCD = 5;
    #70 BCD = 6;
    #80 BCD = 7;
    #90 BCD = 8;
    #100 BCD = 9;

  join

  Seven_Seg_Display_V2001 M0 (Display, BCD);
endmodule
```



Alternative with continuous assignments (dataflow):

```
module Seven_Seg_Display_V2001_CA (
  output        [6: 0]  Display,
  input         [3: 0]  BCD
);
```

```
  //                          abc_defg
  parameter BLANK     = 7'b000_0000;
  parameter ZERO      = 7'b111_1110;      // h7e
  parameter ONE       = 7'b011_0000;      // h30
  parameter TWO       = 7'b110_1101;      // h6d
  parameter THREE     = 7'b111_1001;      // h79
  parameter FOUR      = 7'b011_0011;      // h33
  parameter FIVE      = 7'b101_1011;      // h5b
  parameter SIX       = 7'b101_1111;      // h5f
  parameter SEVEN     = 7'b111_0000;      // h70
  parameter EIGHT     = 7'b111_1111;      // h7f
  parameter NINE      = 7'b111_1011;      // h7b
  wire      A, B, C, D, a, b, c, d, e, f, g;

  assign A = BCD[3];
  assign B = BCD[2];
  assign C = BCD[1];
  assign D = BCD[0];
  assign Display = {a,b,c,d,e,f,g};
  assign a  = (~A)&C | (~A)&B&D | (~B)&(~C)&(~D) | A & (~B)&(~C);
  assign b = (~A)&(~B) | (~A)&(~C)&(~D) | (~A)&C&D | A&(~B)&(~C);
  assign c = (~A)&B | (~A)&D | (~B)&(~C)&(~D) | A&(~B)&(~C);
  assign d = (~A)&C&(~D) | (~A)&(~B)&C | (~B)&(~C)&(~D) | A&(~B)&(~C) | (~A)&B&(~C)&D;
  assign e = (~A)&C&(~D) | (~B)&(~C)&(~D);
  assign f = (~A)&B&(~C) | (~A)&(~C)&(~D) | (~A)&B&(~D) | A&(~B)&(~C);
  assign g = (~A)&C&(~D) | (~A)&(~B)&C | (~A)&B&(~C) | A&(~B)&(~C);
endmodule

module t_Seven_Seg_Display_V2001_CA ();
  wire    [6: 0]   Display;
  reg     [3: 0]   BCD;

  parameter     BLANK       = 7'b000_0000;
  parameter ZERO      = 7'b111_1110;      // h7e
  parameter ONE       = 7'b011_0000;      // h30
  parameter TWO       = 7'b110_1101;      // h6d
  parameter THREE     = 7'b111_1001;      // h79
  parameter FOUR      = 7'b011_0011;      // h33
  parameter FIVE      = 7'b101_1011;      // h5b
  parameter SIX       = 7'b001_1111;      // h1f
  parameter SEVEN     = 7'b111_0000;      // h70
  parameter EIGHT     = 7'b111_1111;      // h7f
  parameter NINE      = 7'b111_1011;      // h7b

  initial #120 $finish;
  initial fork
    #10 BCD = 0;
    #20 BCD = 1;
    #30 BCD = 2;
    #40 BCD = 3;
    #50 BCD = 4;
    #60 BCD = 5;
    #70 BCD = 6;
    #80 BCD = 7;
    #90 BCD = 8;
    #100 BCD = 9;

  join

  Seven_Seg_Display_V2001_CA M0 (Display,   BCD);
```

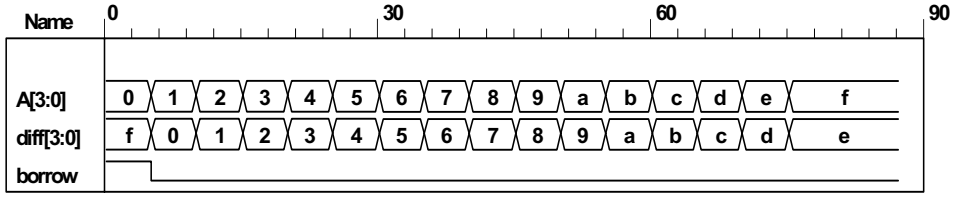
```
endmodule
```

4.52 **(a)** Incrementer for unsigned 4-bit numbers

```
module Problem_4_52a_Data_Flow  (output [3: 0] sum, output carry, input [3: 0] A);
  assign {carry, sum} = A + 1;
endmodule

module t_Problem_4_52a_Data_Flow;
  wire [3: 0]  sum;
  wire         carry;
  reg [3: 0]   A;

  Problem_4_52a_Data_Flow  M0 (sum, carry, A);

  initial # 100 $finish;
  integer K;
  initial begin
    for (K = 0; K < 16; K = K + 1) begin A = K; #5; end
  end
endmodule
```

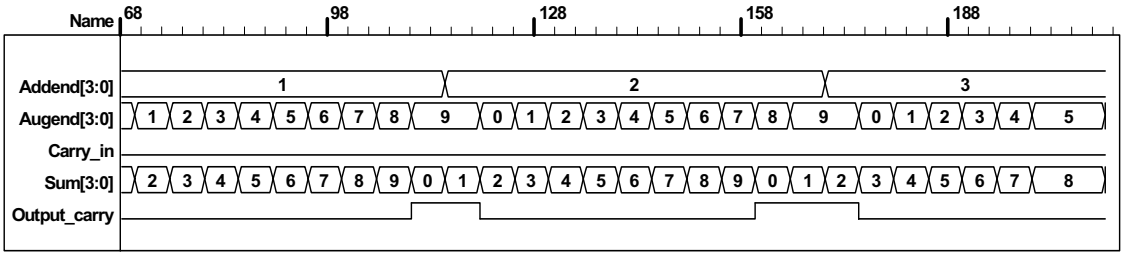
(b) Decrementer for unsigned 4-bit numbers

```
module Problem_4_52b_Data_Flow  (output [3: 0] diff, output borrow, input [3: 0] A);
  assign {borrow, diff} = A - 1;
endmodule

module t_Problem_4_52b_Data_Flow;
  wire [3: 0]      diff;
  wire             borrow;
  reg [3: 0]       A;

  Problem_4_52b_Data_Flow  M0 (diff, borrow, A);

  initial # 100 $finish;
  integer K;
  initial begin
    for (K = 0; K < 16; K = K + 1) begin A = K; #5; end
  end
endmodule
```



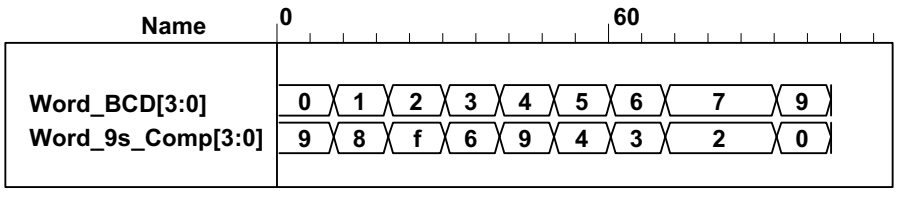
4.53 **// BCD Adder**

```
module Problem_4_53_BCD_Adder (
  output        Output_carry,
  output [3: 0] Sum,
   input [3: 0]  Addend, Augend,
   input         Carry_in);
  supply0       gnd;
  wire [3: 0]   Z_Addend;
  wire          Carry_out;
  wire          C_out;
  assign  Z_Addend = {1'b0, Output_carry, Output_carry, 1'b0};
  wire [3: 0] Z_sum;

  and (w1, Z_sum[3], Z_sum[2]);
  and (w2, Z_sum[3], Z_sum[1]);
  or (Output_carry, Carry_out, w1, w2);

  Adder_4_bit M0 (Carry_out, Z_sum, Addend, Augend, Carry_in);
  Adder_4_bit M1 (C_out, Sum, Z_Addend, Z_sum, gnd);
endmodule


module Adder_4_bit (output carry, output [3:0] sum, input [3: 0] a, b, input c_in);
  assign {carry, sum} = a + b + c_in;
endmodule

module t_Problem_4_53_Data_Flow;
  wire [3: 0]   Sum;
  wire          Output_carry;
  reg [3: 0]    Addend, Augend;
  reg           Carry_in;

  Problem_4_53_BCD_Adder M0 (Output_carry, Sum, Addend, Augend, Carry_in);

  initial # 1500 $finish;
  integer i, j, k;
  initial begin
    for (i = 0; i <= 1; i = i + 1) begin Carry_in = i; #5;
      for (j = 0; j <= 9; j = j +1) begin Addend = j; #5;
        for (k = 0; k <= 9; k = k + 1) begin Augend = k; #5;
       end
     end
    end
  end
endmodule
```

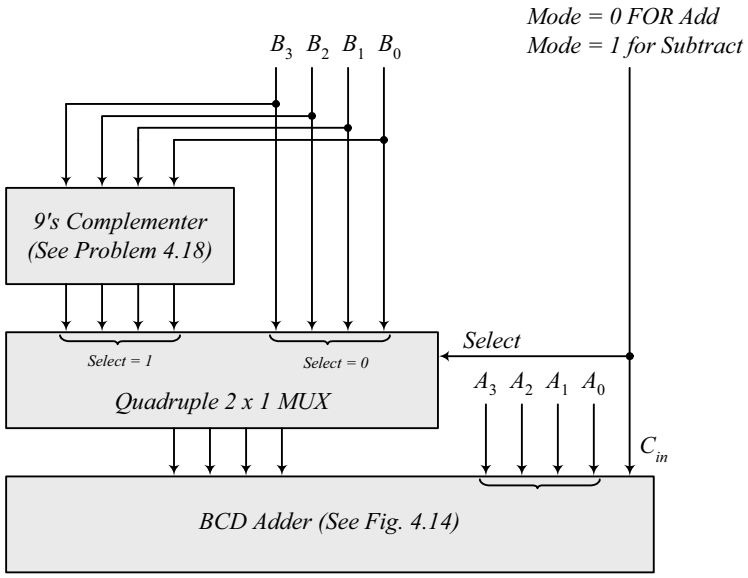


4.54

```
module Nines_Complementer (                 // V2001
  output reg   [3: 0]   Word_9s_Comp,
  input        [3: 0]   Word_BCD
);
  always @ (Word_BCD) begin
    Word_9s_Comp = 4'b0;
    case (Word_BCD)
      4'b0000:   Word_9s_Comp = 4'b1001;      // 0 to 9
      4'b0001:   Word_9s_Comp = 4'b1000;      // 1 to 8
      4'b0010:   Word_9s_Comp = 4'b1111;      // 2 to 7
      4'b0011:   Word_9s_Comp = 4'b0110;      // 3 to 6
      4'b0100:   Word_9s_Comp = 4'b1001;      // 4 to 5
      4'b0101:   Word_9s_Comp = 4'b0100;      // 5 to 4
      4'b0110:   Word_9s_Comp = 4'b0011;      // 6 to 3
      4'b0111:   Word_9s_Comp = 4'b0010;      // 7 to 2
      4'b1000:   Word_9s_Comp = 4'b0001;      // 8 to 1
      4'b1001:   Word_9s_Comp = 4'b0000;      // 9 to 0
      default:   Word_9s_Comp = 4'b1111;      // Error detection
    endcase
  end
endmodule

module t_Nines_Complementer ();
  wire  [3: 0]   Word_9s_Comp;
  reg   [3: 0]   Word_BCD;

  Nines_Complementer M0 (Word_9s_Comp, Word_BCD);

  initial #11$finish;
  initial fork
          Word_BCD = 0;
    #10 Word_BCD = 1;
    #20 Word_BCD = 2;
    #30 Word_BCD = 3;
    #40 Word_BCD = 4;
    #50 Word_BCD = 5;
    #60 Word_BCD = 6;
    #70 Word_BCD = 7;
    #20 Word_BCD = 8;
    #90 Word_BCD = 9;
    #100 Word_BCD = 4'b1100;
  join
endmodule
```

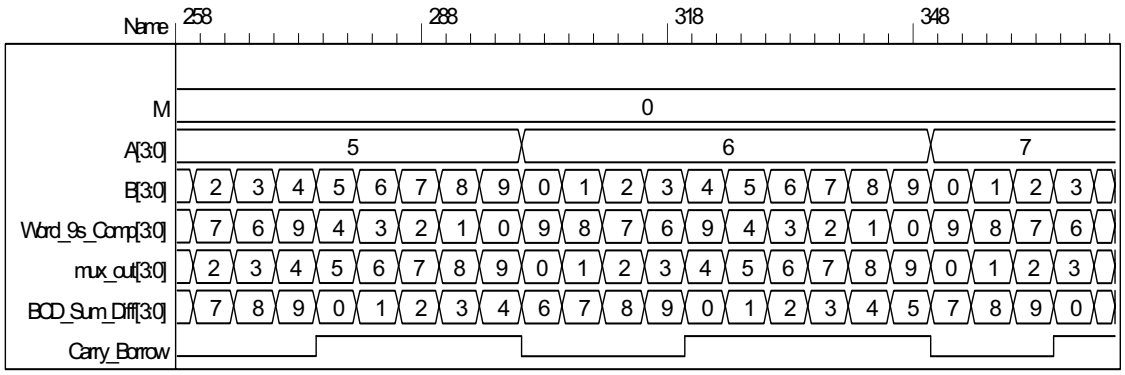


4.55 From Problem 4.19:

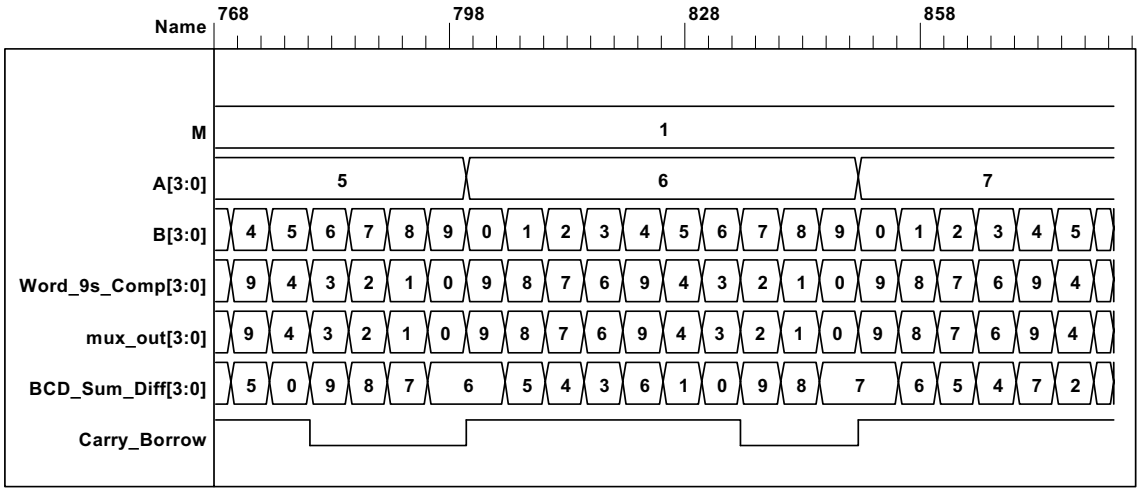



```
// BCD Adder – Subtractor
module Problem_4_55_BCD_Adder_Subtractor (
  output [3: 0]     BCD_Sum_Diff,
  output            Carry_Borrow,
  input [3: 0]      B, A,
  input             Mode

);
  wire [3: 0] Word_9s_Comp, mux_out;
  Nines_Complementer   M0 (Word_9s_Comp, B);
  Quad_2_x_1_mux       M2 (mux_out, Word_9s_Comp, B, Mode);
  BCD_Adder            M1 (Carry_Borrow,  BCD_Sum_Diff, mux_out, A, Mode);
endmodule

module Nines_Complementer (              // V2001
  output reg  [3: 0]  Word_9s_Comp,
  input       [3: 0]  Word_BCD
);
  always @ (Word_BCD) begin
    Word_9s_Comp = 4'b0;
    case (Word_BCD)
      4'b0000:   Word_9s_Comp = 4'b1001;      // 0 to 9
      4'b0001:   Word_9s_Comp = 4'b1000;      // 1 to 8
      4'b0010:   Word_9s_Comp = 4'b0111;      // 2 to 7
      4'b0011:   Word_9s_Comp = 4'b0110;      // 3 to 6
      4'b0100:   Word_9s_Comp = 4'b1001;      // 4 to 5
      4'b0101:   Word_9s_Comp = 4'b0100;      // 5 to 4
      4'b0110:   Word_9s_Comp = 4'b0011;      // 6 to 3
      4'b0111:   Word_9s_Comp = 4'b0010;      // 7 to 2
      4'b1000:   Word_9s_Comp = 4'b0001;      // 8 to 1
      4'b1001:   Word_9s_Comp = 4'b0000;      // 9 to 0
      default:   Word_9s_Comp = 4'b1111;          // Error detection
    endcase
  end
endmodule
```

```
module Quad_2_x_1_mux (output reg [3: 0] mux_out, input [3: 0] b, a,  input select);
  always @ (a, b,  select)
    case (select)
      0:  mux_out = a;
      1:  mux_out = b;
    endcase
endmodule

module BCD_Adder (
  output            Output_carry,
  output   [3: 0]   Sum,
  input    [3: 0]   Addend, Augend,
  input             Carry_in);
  supply0           gnd;
  wire     [3: 0]   Z_Addend;
  wire              Carry_out;
  wire              C_out;
  assign  Z_Addend = {1'b0, Output_carry, Output_carry, 1'b0};
  wire [3: 0] Z_sum;

  and (w1, Z_sum[3], Z_sum[2]);
  and (w2, Z_sum[3], Z_sum[1]);
  or (Output_carry, Carry_out, w1, w2);

  Adder_4_bit M0 (Carry_out, Z_sum, Addend, Augend, Carry_in);
  Adder_4_bit M1 (C_out, Sum, Z_Addend, Z_sum, gnd);
endmodule

module Adder_4_bit (output carry, output [3:0] sum, input [3: 0] a, b, input c_in);
  assign {carry, sum} = a + b + c_in;
endmodule

module t_Problem_4_55_BCD_Adder_Subtractor();
  wire  [3: 0]   BCD_Sum_Diff;
  wire           Carry_Borrow;
  reg   [3: 0]   B, A;
  reg            Mode;

  Problem_4_55_BCD_Adder_Subtractor M0 (BCD_Sum_Diff, Carry_Borrow, B, A, Mode);

  initial #1000 $finish;

  integer J, K, M;
  initial begin
    for (M = 0; M < 2; M = M + 1) begin
      for (J = 0; J < 10; J = J + 1) begin
        for (K = 0; K < 10; K = K + 1) begin
          A = J; B = K; Mode = M; #5 ;
       end
      end
    end
  end
endmodule
```



Note: For subtraction, Carry_Borrow = 1 indicates a positive result; Carry_Borrow = 0 indicates a negative result.



4.56

```
assign match = (A == B);  // Assumes reg [3: 0] A, B;
```

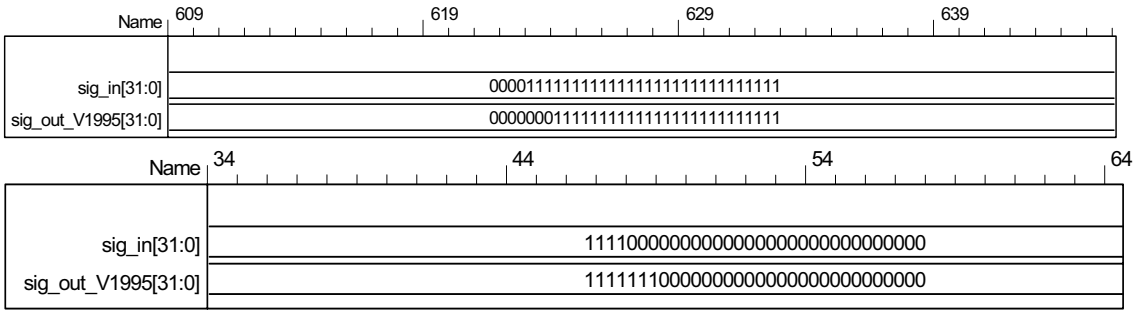
4.57

```
// Priority encoder (See Problem 4.29)
// Caution: do not confuse logic value x with identifier x.
// Verilog 1995

module Prob_4_57 (x, y, v, D3, D2, D1, D0);
output  x, y, v;
input   D3, D2, D1, D0;
reg     x, y, v;
...

// Verilog 2001, 2005

module Prob_4_57 (output reg x, y, v, input D3, D2, D1, D0);
 always @ (D3, D2, D1, D0) begin    // always @ (D3 or D2 or D1 or D0)
   x = 0;
   y = 0;
   v = 0;
   casex ({D3, D2, D1, D0})
    4'b0000: {x, y, v} = 3'bxx0;
    4'bxxx1: {x, y, v} = 3'b001;
```

```
    4'bxx10: {x, y, v} = 3'b011;
    4'bx100: {x, y, v} = 3'b101;
    4'b1000: {x, y, v} = 3'b110;
   endcase
  end
endmodule

module t_Prob_4_57;
  wire      x, y, v;
  reg       D3, D2, D1, D0;
  integer   K;
  Prob_4_57 M0 (x, y, v, D3, D2, D1, D0);
  initial #100 $finish;
  initial begin
    for (K = 0; K < 16; K = K + 1) begin {D3, D2, D1, D0} = K; #5 ; end
  end
endmodule
```

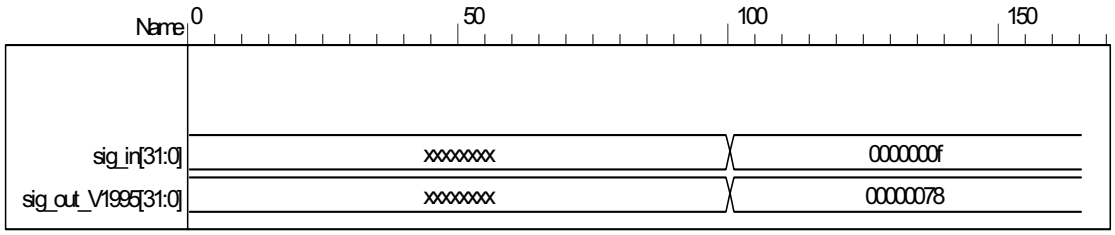
4.58

```
//module shift_right_by_3_V2001 (output [31: 0]  sig_out, input [31: 0] sig_in);
 // assign sig_out = sig_in >>> 3;
//endmodule

module shift_right_by_3_V1995 (output reg [31: 0]  sig_out, input [31: 0] sig_in);
  always @ (sig_in)
    sig_out = {sig_in[31], sig_in[31], sig_in[31],  sig_in[31: 3]};
endmodule


module t_shift_right_by_3 ();
  wire [31: 0]  sig_out_V1995;
  wire [31: 0]  sig_out_V2001;

  reg [31: 0] sig_in;

  //shift_right_by_3_V2001 M0 (sig_out_V2001, sig_in);

  shift_right_by_3_V1995 M1 (sig_out_V1995, sig_in);
  integer k;
  initial #1000 $finish;
  initial begin
    sig_in = 32'hf000_0000;
    #100 sig_in = 32'h8fff_ffff;
    #500 sig_in = 32'h0fff_ffff;
  end
endmodule
```



4.59

```
//module shift_left_by_3_V2001 (output [31: 0]  sig_out, input [31: 0] sig_in);
 // assign sig_out = sig_in <<< 3;
//endmodule

module shift_left_by_3_V1995 (output reg [31: 0]  sig_out, input [31: 0] sig_in);
  always @ (sig_in)
    sig_out = {sig_in[28: 0], 3'b0};
endmodule


module t_shift_left_by_3 ();
  wire [31: 0]  sig_out_V1995;
  //wire [31: 0]  sig_out_V2001;

  reg [31: 0] sig_in;

  //shift_left_by_3_V2001 M0 (sig_out_V2001, sig_in);

  shift_left_by_3_V1995 M1 (sig_out_V1995, sig_in);
  integer k;
  initial #500 $finish;
  initial begin
    #100 sig_in = 32'h0000_000f;
  end
endmodule
```



4.60

```
module BCD_to_Decimal (output reg [3: 0] Decimal_out, input [3: 0] BCD_in);
  always @ (BCD_in) begin
    Decimal_out = 0;
    case (BCD_in)
      4'b0000:   Decimal_out = 0;
      4'b0001:   Decimal_out = 1;
      4'b0010:   Decimal_out = 2;
      4'b0011:   Decimal_out = 3;
      4'b0100:   Decimal_out = 4;
      4'b0101:   Decimal_out = 5;
      4'b0110:   Decimal_out = 6;
      4'b0111:   Decimal_out = 7;
      4'b1000:   Decimal_out = 8;
      4'b1001:   Decimal_out = 9;
      default:   Decimal_out = 4'bxxxx;
    endcase
  end
endmodule
```

4.61

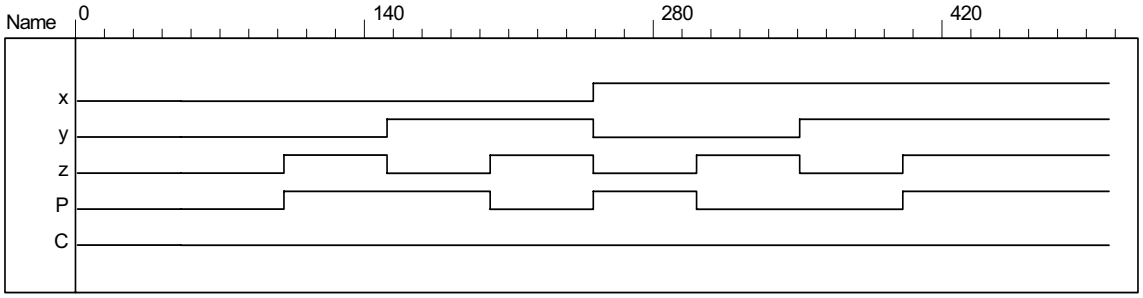
```
module Even_Parity_Checker_4 (output  P, C, input x, y, z);
  xor (w1, x,  y);
  xor (P, w1, z);
  xor (C, w1, w2);
```

```
  xor (w2, z, P);
endmodule
```

See Problem 4.62 for testbench and waveforms.

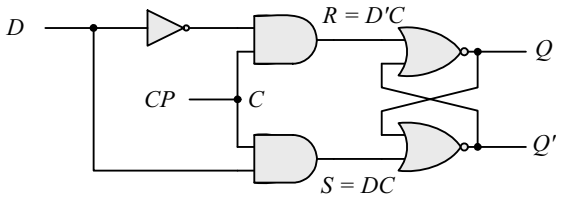
4.62

```
module Even_Parity_Checker_4 (output  P, C, input x, y, z);
  assign w1 = x ^ y;
  assign P = w1 ^ z;
  assign C = w1 ^ w2;
  assign w2 = z ^ P;
endmodule
```

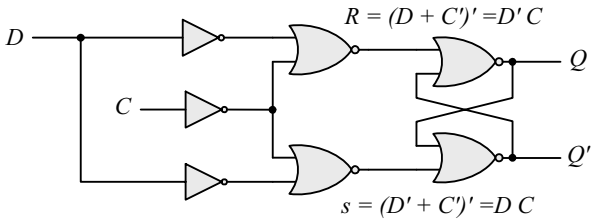


CHAPTER 5

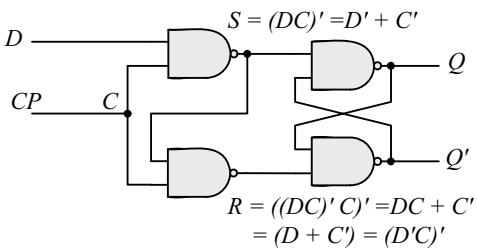
5.1 **(a)**



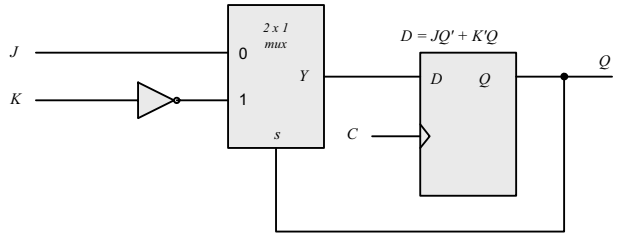
(b)



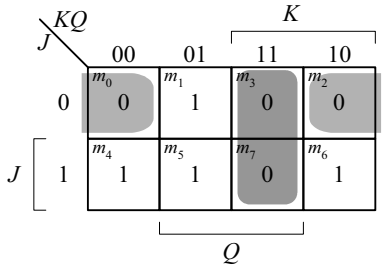
(c)



5.2



5.3 $Q'(t+1) = (JQ' + K'Q)' = (J' + Q)(K + Q') = J'Q' + KQ$



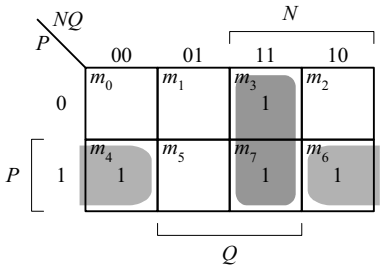
5.4

(a)

P	N	Q(t + 1)
0	0	0
0	1	Q(t)
1	0	Q'(t)
1	1	1

(b)

P	N	Q(t)	Q(t + 1)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



$Q(t+1) = PQ' + NQ$

(c)

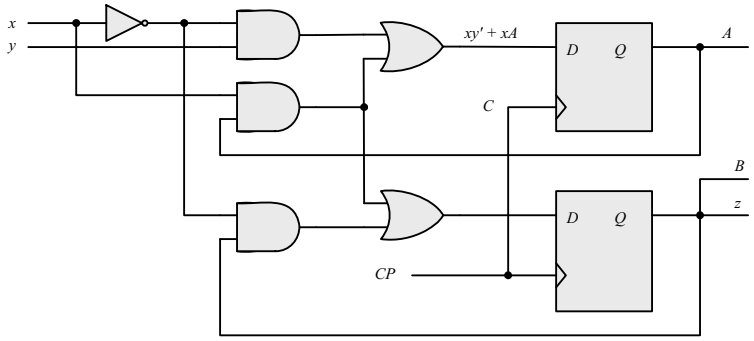
Q(t)	Q(t+1)	P	N
0	0	0	x
0	1	1	x
1	0	x	0
1	1	x	1

(d) Connect P and N together.

5.5

The truth table describes a combinational circuit.
The state table describes a sequential circuit.
The characteristic table describes the operation of a flip-flop.
The excitation table gives the values of flip-flop inputs for a given state transition.
The four equations correspond to the algebraic expression of the four tables.

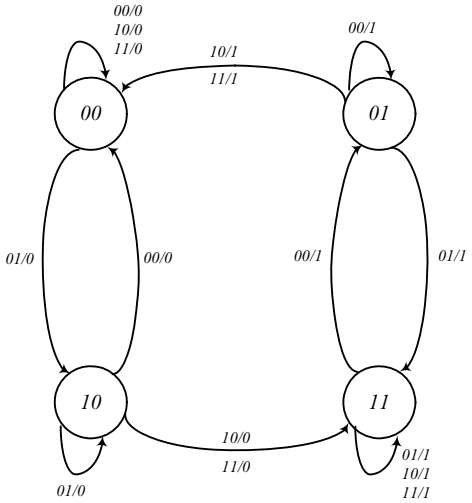
5.6 **(a)**



(b)

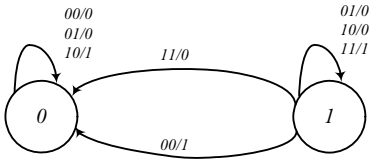
Present state		Inputs		Next state		Output
A	B	x	y	A	B	z
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	0	0	0
1	0	0	1	1	0	0
1	0	1	0	1	1	0
1	0	1	1	1	1	0
1	1	0	0	0	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1

(c)



5.7

Present state	Inputs		Next state	Output
Q	x	y	Q	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$S = x \oplus y \oplus Q$

$Q(t + 1) = xy + xQ + yQ$

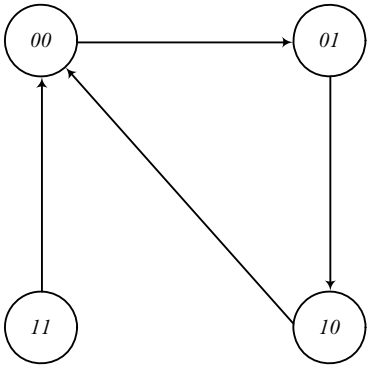
5.8 A counter with a repeated sequence of 00, 01, 10.

Present state		Next state		FF Inputs	
A	B	A	B	T_A	T_B
0	0	0	0	0	1
0	1	1	0	1	1
1	0	0	0	1	0
1	1	0	0	1	1

$$T_A = A + B$$
$$T_B = A' + B$$

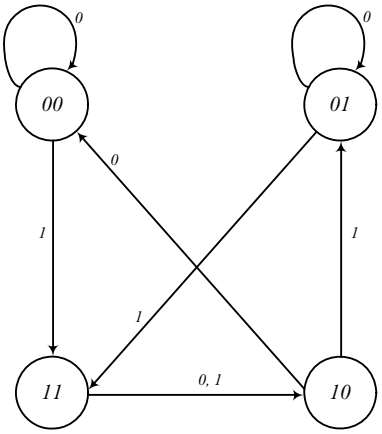
Repeated sequence:

$00 \rightarrow 01 \rightarrow 10 \rightarrow$



5.9

$$A(t+1) = J_A A' + K'A = xA' + BA$$
$$B(t+1) = J_B B' + K'_B B = xB' + A'B$$



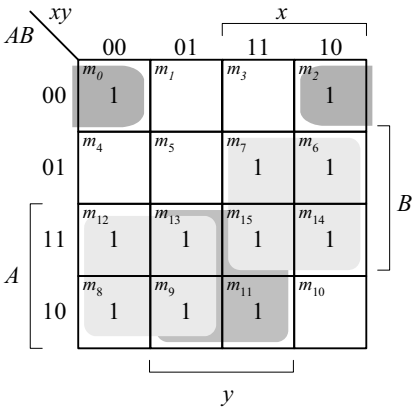
5.10 **(a)** $J_A = Bx + B'y'$ $\quad J_B = A'x$

$K_A = B'xy'$ $\quad K_B = A + xy'$ $\quad z = Axy + Bx'y'$

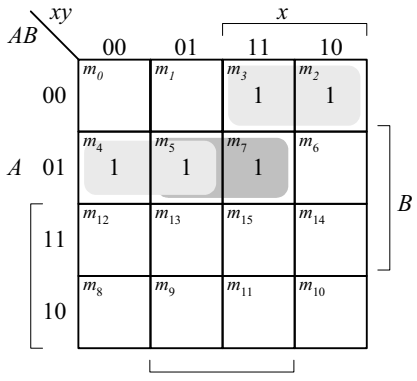
(b)

Present state		Inputs		Next state		Output	FF Outputs			
A	B	x	y	A	B	z	J_A	K_A	J_A	J_B
0	0	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	1	0	1	1	1	1
0	0	1	1	0	1	0	0	0	1	0
0	1	0	0	0	1	1	0	0	0	0
0	1	0	1	0	1	0	0	0	0	0
0	1	1	0	1	0	0	1	0	1	0
0	1	1	1	1	1	0	1	0	1	0
1	0	0	0	1	0	0	1	0	0	1
1	0	0	1	1	0	0	0	0	0	1
1	0	1	0	0	0	0	1	1	0	1
1	0	1	1	1	0	0	0	0	0	1
1	1	0	0	1	0	1	0	0	0	1
1	1	0	1	1	0	0	0	0	0	1
1	1	1	0	1	0	0	1	0	0	1
1	1	1	1	1	0	1	1	0	0	1

(c)



$A(t+1) = Ax' + Bx + Ay + A'B'y'$



$B(t+1) = A'B'x + A'B'(x' + y)$

5.11

Present state:	00	00	01	00	01	11	00	01	11	10	00	01	11	10	10
Input:	0	1	0	1	1	0	1	1	1	0	1	1	1	1	0
Output:	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1
Next state:	00	01	00	01	11	00	01	11	10	00	01	11	10	10	00

5.12

Present state	Next state		Output	
	0	**1**	**0**	**1**
a	f	b	0	0
b	d	a	0	0
d	g	a	1	0
f	f	b	1	1
g	g	d	0	1

5.13 **(a)**

State:	*a*	*f*	*b*	*c*	*e*	*d*	*g*	*h*	*g*	*g*	*h*	*a*
Input:	0	1	1	1	0	0	1	0	0	1	1	
Output:	0	1	0	0	0	1	1	1	0	1	0	

(b)

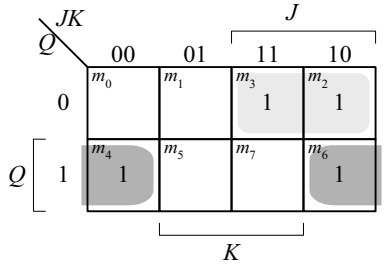
State:	*a*	*f*	*b*	*a*	*b*	*d*	*g*	*d*	*g*	*g*	*d*	*a*
Input:	0	1	1	1	0	0	1	0	0	1	1	
Output:	0	1	0	0	0	1	1	1	0	1	0	

5.14

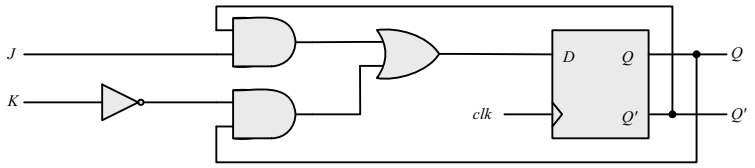
	Present state A B C	Next state x=0	Next state x=1	Output x=1	Output x=0
a	0 0 0	000	001	0	0
b	0 0 1	011	010	0	0
c	0 1 1	000	010	0	0
d	0 1 0	110	010	0	1
e	1 1 0	000	010	0	1

5.15 $D_Q = Q'J + QK'$

Present state Q	*Inputs* J	K	*Next state* Q	
0	0	0	0	No change
0	0	1	0	Reset to 0
0	1	0	1	Set to 1
0	1	1	1	Complement
1	0	0	1	No change
1	0	1	0	Reset to 0
1	1	0	1	Set to 1
1	1	1	0	Complement



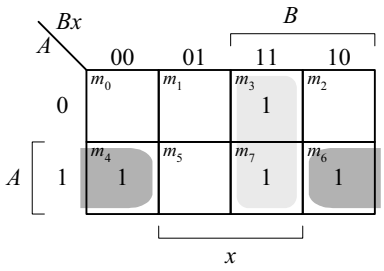
$Q(t+1) = D_Q + Q'J + QK'$



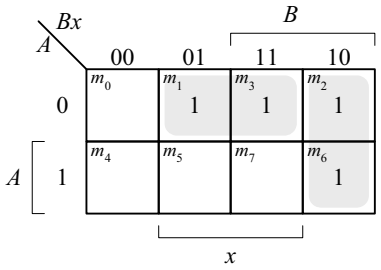
5.16 **(a)** $D_A = Ax' + Bx$

$D_B = A'x + Bx'$

Present state		*Input*	*Next state*	
A	B	x	A	B
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0



$D_A = Ax' + Bx$

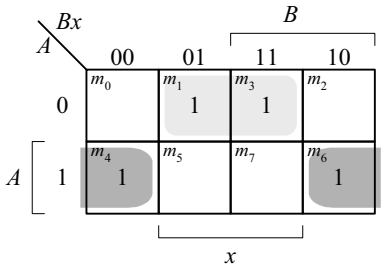


$D_B = A'x + Bx'$

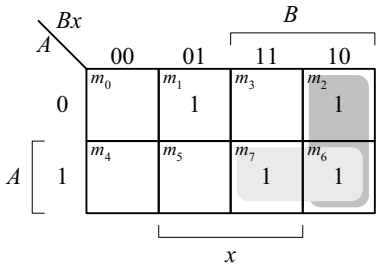
(b) $D_A = A'x + Ax'$

$D_B = AB + Bx'$

Present state		*Input*	*Next state*	
A	B	x	A	B
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	1

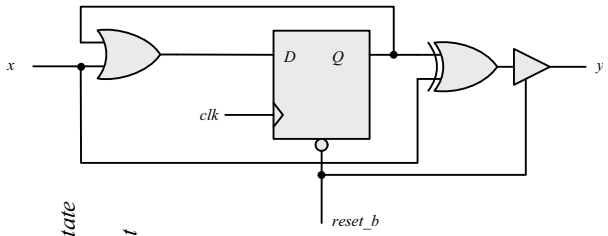


$D_A = A'x + Ax'$

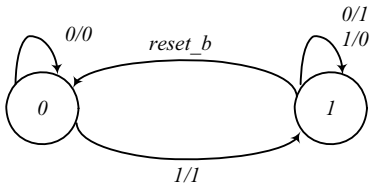


$D_B = AB + Bx'$

5.17 The output is 0 for all 0 inputs until the first 1 occurs, at which time the output is 1. Thereafter, the output is the complement of the input. The state diagram has two states. In state 0: output = input; in state 1: output = input'.



Present state	*Input*	*Next state*	*Output*
A	*x*	*A*	*y*
0	*0*	*0*	*0*
0	*1*	*1*	*1*
1	*0*	*1*	*1*
1	*1*	*1*	*0*

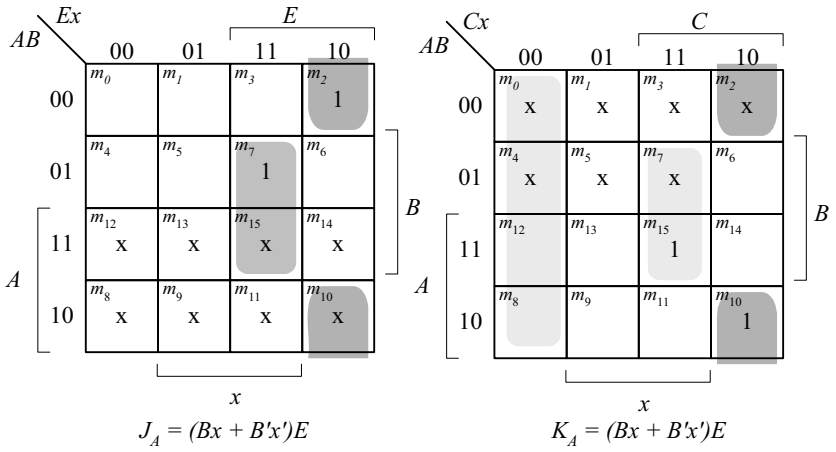


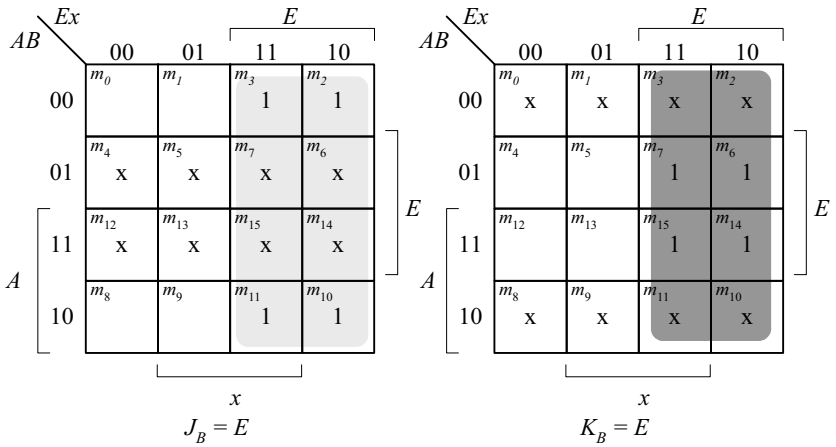
$D_A = A + x$

$y = Ax' + A'x$

5.18 Binary up-down counter with enable *E*.

Present state A B	*Input* x	*Next state* A B	*Flip-flop inputs* J_A K_A	J_B K_B
0 0	0 1	0 0	0 x	0 x
0 0	0 1	0 0	0 x	0 x
0 0	1 0	1 1	1 x	1 x
0 0	1 1	0 1	0 x	1 x
0 1	0 0	0 1	0 x	x 0
0 1	0 1	0 1	0 x	x 0
0 1	1 0	0 1	0 x	x 1
0 1	1 1	1 0	1 x	x 1
1 0	0 0	1 0	x 0	1 0
1 0	0 1	1 0	x 0	1 0
1 0	1 0	0 1	x 1	x 1
1 0	1 1	1 1	x 0	x 1
1 1	0 0	1 1	x 0	x 0
1 1	0 1	1 1	x 0	x 0
1 1	1 0	1 1	1 0	x 1
1 1	1 1	1 1	x 1	x 1

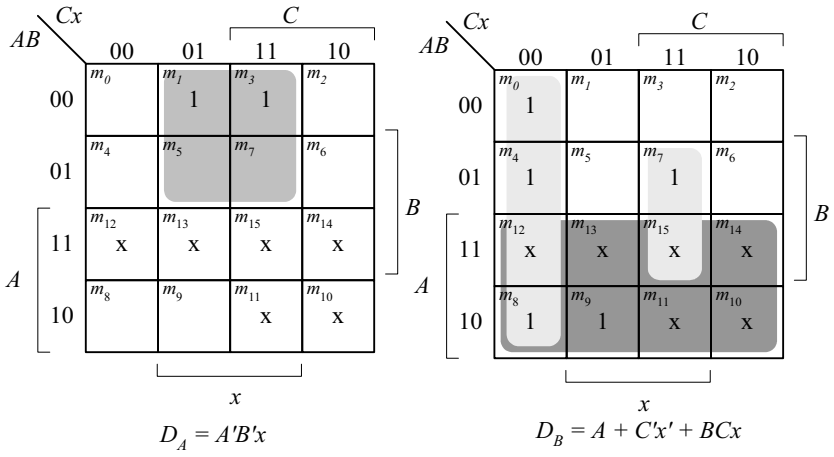




5.19 **(a)** Unused states (see Fig. P5.19): 101, 110, 111.

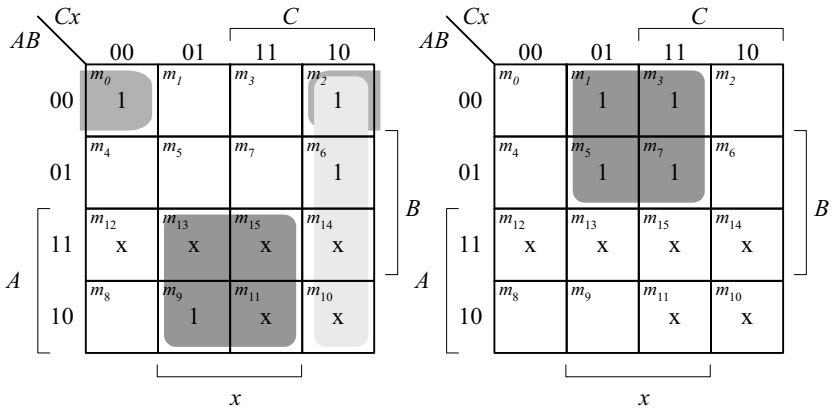
Present state ABC	*Input x*	*Next state ABC*	*Output y*
000	0	011	0
000	1	100	1
001	0	001	0
001	1	100	1
010	0	010	0
010	1	000	1
011	0	001	0
011	1	010	1
100	0	010	0
100	1	011	1

$d(A, B, C, x) = \Sigma\,(10, 11, 12, 13, 14, 15)$



$D_A = A'B'x$

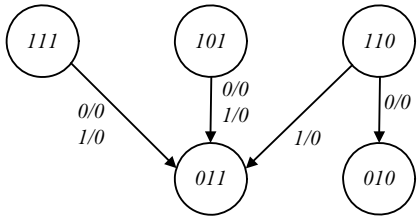
$D_B = A + C'x' + BCx$



$D_C = Cx' + Ax + A'B'x'$

$y = A'x$

The machine is self-correcting, i.e., the unused states transition to known states.



(b) With JK flip=flops, the state table is the same as in (a).

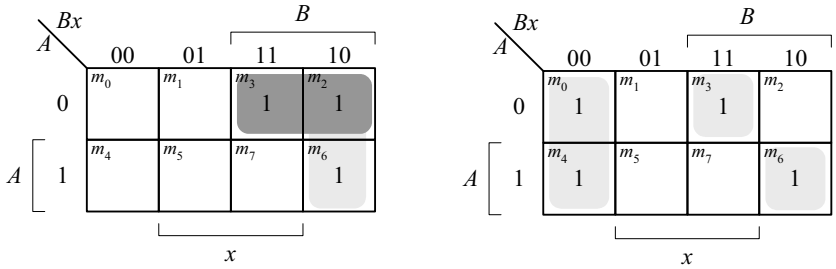
Flip-flop inputs

J_A	K_A	J_B	K_B	J_C	K_C
0	x	1	x	1	x
1	x	0	x	0	x
0	x	0	x	x	0
1	x	0	x	x	1
0	x	x	0	0	x
0	x	x	1	0	x
0	x	x	1	x	0
0	x	x	0	x	1
x	1	1	x	0	x
x	1	1	x	1	x

$J_A = B'x$ $K_A = 1$
$J_B = A + C'x'$ $K_B = C' x + Cx'$
$J_C = Ax + A'B'x'$ $K_C = x$
$y = A'x$

The machine is self-correcting because $K_A = 1$.

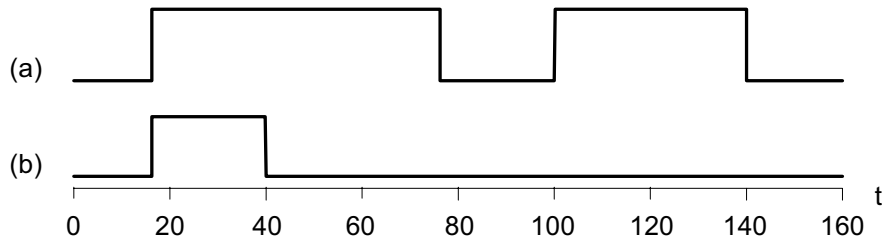
5.20 From state table 5.4: $T_A(A, B, x) = \Sigma(2, 3, 6)$, $T_B(A, B, x) = \Sigma(0, 3, 4, 6)$.



$T_A = A'B + Bx'$ $T_B = B'x' + A'x + A'Bx$

5.21 The statements associated with an **initial** keyword execute once, in sequence, with the activity expiring after the last statment competes execution; the statements assocated with the **always** keyword execute repeatedly, subject to timing control (e.g, #10).

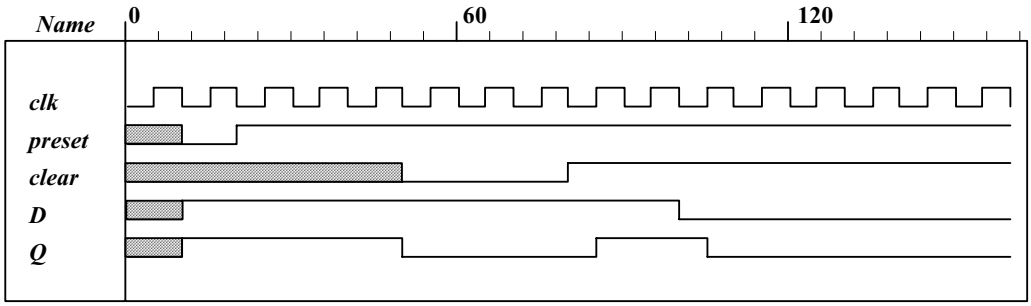
5.22



5.23 **(a)** *RegA* = 125, *RegB* = 125
(b) *RegA* = 125, *RegB* = 30

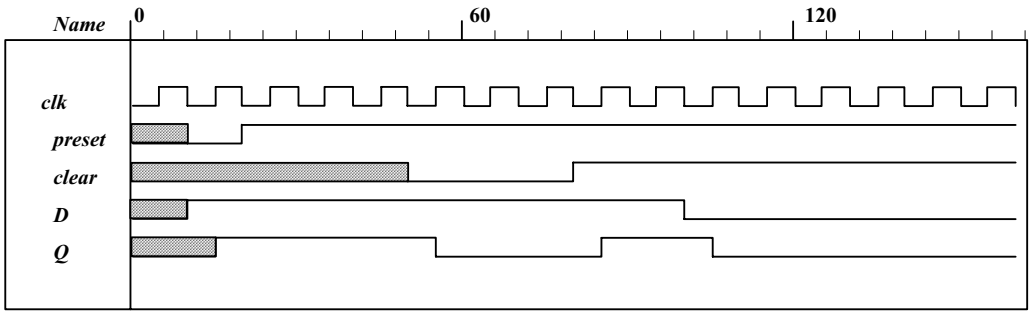
5.24 (a)

```
module DFF (output reg Q, input D, clk, preset, clear);
  always @ (posedge clk, negedge preset, negedge clear )
    if (preset  == 0) Q <= 1'b1;
    else if (clear == 0) Q <= 1'b0;
    else Q <= D;
endmodule

module t_DFF ();
  wire  Q;
  reg   clk, preset, clear;
  reg   D;

  DFF M0 (Q, D, clk, preset, clear);

  initial #160 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #10 preset = 0;
    #20 preset = 1;
    #50 clear = 0;
    #80 clear = 1;
    #10 D = 1;
    #100 D = 0;
    #200 D = 1;
  join
endmodule
```



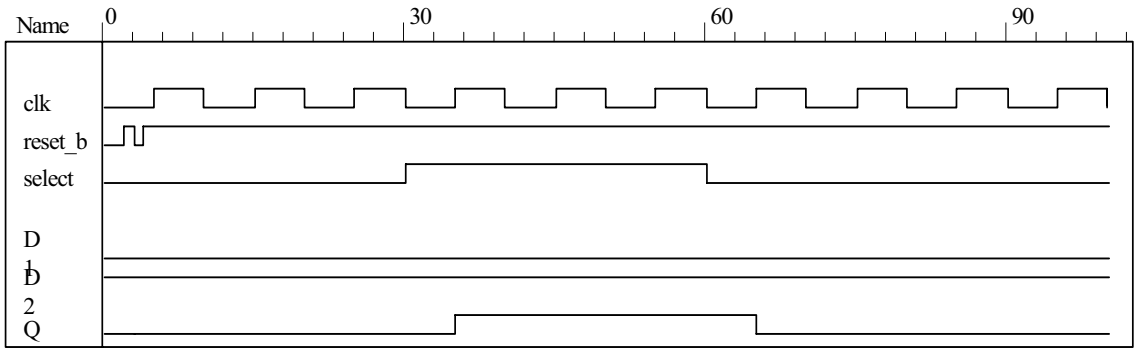
(b)

```
module DFF (output reg Q, input D, clk, preset, clear);
  always @ (posedge clk)
    if (preset  == 0) Q <= 1'b1;
    else if (clear == 0) Q <= 1'b0;
    else Q <= D;
endmodule
```



5.25

```
module Dual_Input_DFF (output reg Q, input D1, D2, select, clk, reset_b);
  always @ (posedge clk, negedge reset_b)
    if (reset_b == 0) Q <= 0;
    else Q <= select ? D2 : D1;
endmodule

module t_Dual_Input_DFF ();
  wire Q;
  reg D1, D2, select, clk, reset_b;
   Dual_Input_DFF M0 (Q, D1, D2, select, clk, reset_b);
  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    select = 0;
    #30 select = 1;
    #60 select = 0;
  join
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;
    #4 reset_b = 1;
       D1 = 0;
       D2 = 1;
  join
endmodule
```



5.26 (a)

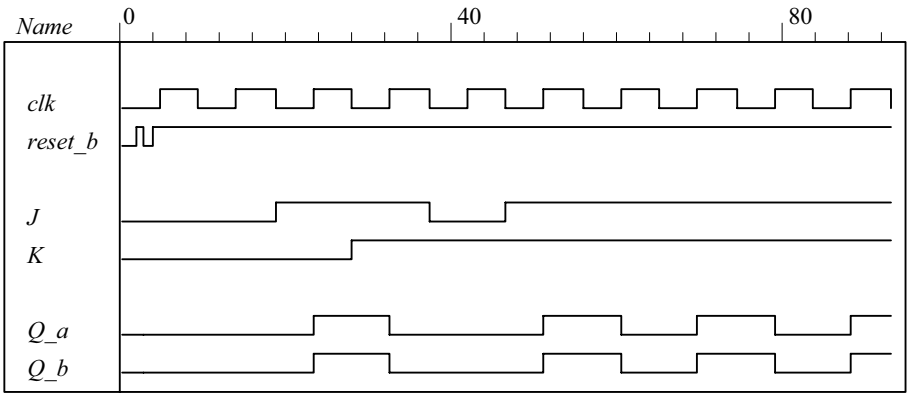
$Q(t + 1) = JQ' + K'Q$

When $Q = 0$, $Q(t + 1) = J$
When $Q = 1$, $Q(t + 1) = K'$

```
module JK_Behavior_a (output reg Q, input J, K, CLK, reset_b);
  always @ (posedge CLK, negedge reset_b)
    if (reset_b == 0) Q <= 0; else
      if (Q == 0)      Q <= J;
      else         Q <= ~K;
endmodule
```

(b)

```
module JK_Behavior_b (output reg Q, input J, K, CLK, reset_b);
  always @ (posedge CLK, negedge reset_b)
    if (reset_b == 0) Q <= 0;
    else
    case ({J, K})
      2'b00:  Q <= Q;
      2'b01:  Q <= 0;
      2'b10:  Q <= 1;
      2'b11:  Q <= ~Q;
    endcase
endmodule

module t_Prob_5_26  ();
  wire Q_a, Q_b;
  reg J, K, clk, reset_b;
   JK_Behavior_a M0 (Q_a, J, K, clk, reset_b);
   JK_Behavior_b M1 (Q_b, J, K, clk, reset_b);

   initial #100 $finish;
   initial begin clk = 0; forever #5 clk = ~clk; end
   initial fork
     #2 reset_b = 1;
     #3 reset_b = 0;          // Initialize to s0
     #4 reset_b = 1;
    J =0; K = 0;
    #20 begin J= 1; K = 0;  end
    #30 begin J = 1; K = 1; end
    #40 begin J = 0; K = 1; end
    #50 begin J = 1; K = 1; end
  join
 endmodule
```

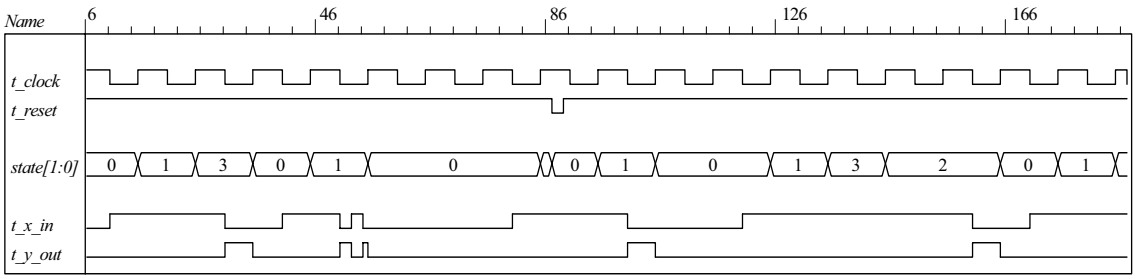


5.27

```
// Mealy FSM zero detector (See Fig. 5.16)
module Mealy_Zero_Detector (
  output reg y_out,
  input x_in, clock, reset
);
  reg [1: 0] state, next_state;
  parameter   S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @ (posedge clock, negedge reset)   // state transition
    if (reset == 0) state <= S0;
    else state <= next_state;
```

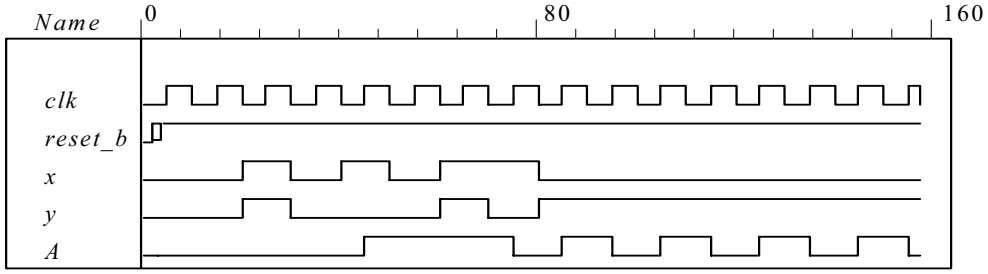
```
  always @ (state, x_in) // Form the next state
    case (state)
      S0:begin y_out = 0;   if (x_in)   next_state = S1; else next_state = S0; end
      S1:     begin y_out = ~x_in; if (x_in)       next_state = S3; else next_state = S0; end
      S2:begin y_out = ~x_in; if (~x_in) next_state = S0; else next_state = S2; end
      S3:     begin y_out = ~x_in; if (x_in)   next_state = S2; else next_state = S0; end
    endcase

endmodule

module t_Mealy_Zero_Detector;
  wire  t_y_out;
  reg    t_x_in, t_clock, t_reset;

Mealy_Zero_Detector M0 (t_y_out, t_x_in, t_clock, t_reset);


initial #200 $finish;
initial begin t_clock = 0; forever #5 t_clock = ~t_clock; end

initial fork
      t_reset = 0;
  #2 t_reset = 1;
  #87 t_reset = 0;
  #89 t_reset = 1;
  #10 t_x_in = 1;
  #30 t_x_in = 0;
  #40 t_x_in = 1;
  #50 t_x_in = 0;
  #52 t_x_in = 1;
  #54 t_x_in = 0;
  #70 t_x_in = 1;
  #80 t_x_in = 1;
  #70 t_x_in = 0;
  #90 t_x_in = 1;
  #100 t_x_in = 0;
  #120 t_x_in = 1;
  #160 t_x_in = 0;
  #170 t_x_in = 1;

  join
endmodule
```

Note: Simulation results match Fig. 5.22.



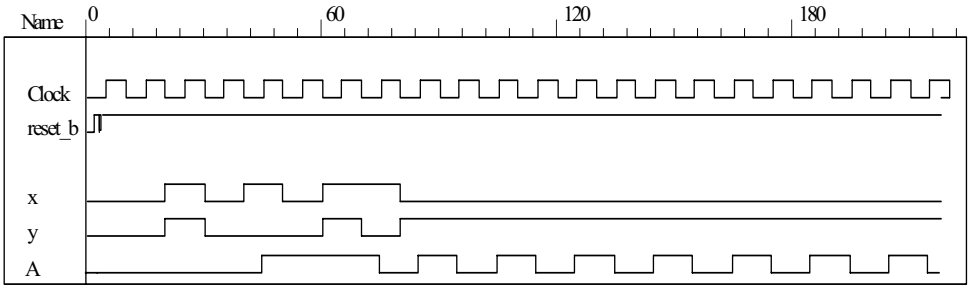
5.28 (a)

```
module Prob_5_28a  (output A, input x, y, clk, reset_b);
 parameter s0 = 0, s1 = 1;
 reg state, next_state;
 assign A = state;

 always @ (posedge clk, negedge reset_b)
   if (reset_b == 0) state <= s0; else state <= next_state;

 always @ (state, x, y) begin
   next_state = s0;
   case (state)
     s0:      case ({x, y})
                2'b00, 2'b11: next_state = s0;
                2'b01, 2'b10: next_state = s1;
              endcase
     s1:      case ({x, y})
                2'b00, 2'b11: next_state = s1;
                2'b01, 2'b10: next_state = s0;
              endcase
    endcase
  end
endmodule
module t_Prob_5_28a  ();
 wire A;
 reg x, y, clk, reset_b;
  Prob_5_28a M0 (A, x, y, clk, reset_b);
  initial #350 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;      // Initialize to s0
    #4 reset_b = 1;
   x =0; y = 0;
   #20 begin x= 1; y = 1;  end
   #30 begin x = 0; y = 0; end
   #40 begin x = 1; y = 0; end
   #50 begin x = 0; y = 0; end
   #60 begin x = 1; y = 1; end
   #70 begin x = 1; y = 0; end
   #80 begin x = 0; y = 1; end
 join
endmodule
```



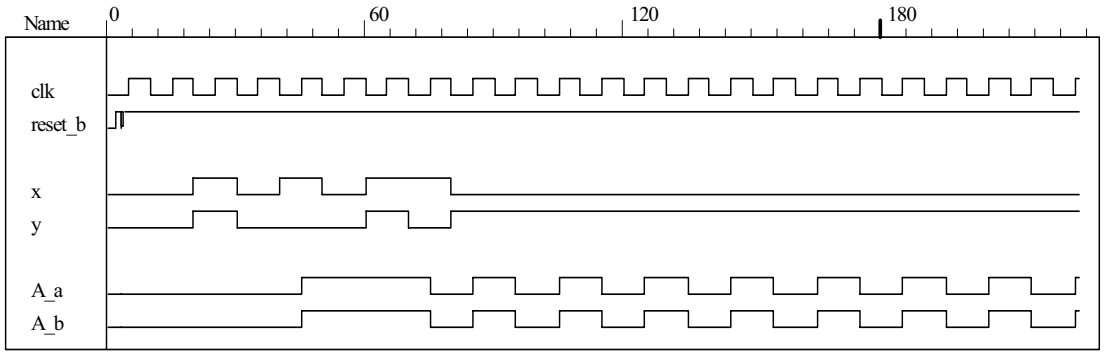
(b)

```
module Prob_5_28b  (output A, input x, y, Clock, reset_b);
  xor (w1, x, y);
  xor (w2, w1, A);
  DFF M0 (A, w2, Clock, reset_b);
endmodule

module DFF (output reg Q, input D, Clock, reset_b);
  always @ (posedge Clock, negedge reset_b)
    if (reset_b == 0) Q <= 0;
    else Q <= D;
endmodule

module t_Prob_5_28b  ();
  wire A;
  reg x, y, clk, reset_b;
   Prob_5_28b M0 (A, x, y, clk, reset_b);
   initial #350 $finish;
   initial begin clk = 0; forever #5 clk = ~clk; end
   initial fork
     #2 reset_b = 1;
     #3 reset_b = 0;          // Initialize to s0
     #4 reset_b = 1;
    x =0; y = 0;
    #20 begin x= 1; y = 1;  end
    #30 begin x = 0; y = 0; end
    #40 begin x = 1; y = 0; end
    #50 begin x = 0; y = 0; end
    #60 begin x = 1; y = 1; end
    #70 begin x = 1; y = 0; end
    #80 begin x = 0; y = 1; end
  join
endmodule
```



(c) See results of (b) and (c).

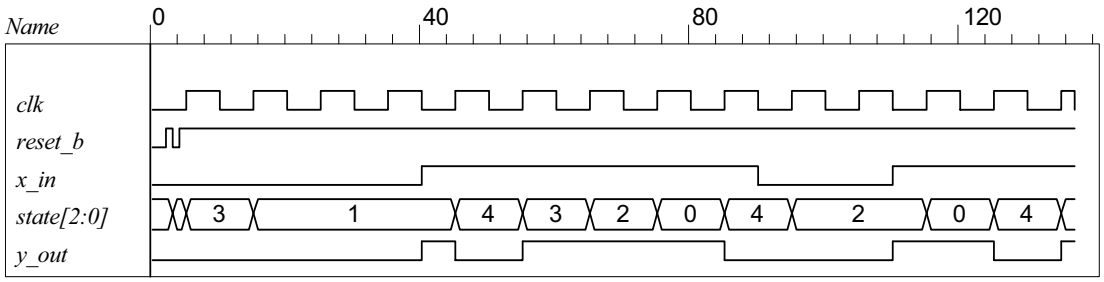
```
module t_Prob_5_28c  ();
 wire A_a, A_b;
 reg x, y, clk, reset_b;
  Prob_5_28a M0 (A_a, x, y, clk, reset_b);
  Prob_5_28b M1 (A_b, x, y, clk, reset_b);

  initial #350 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;          // Initialize to s0
    #4 reset_b = 1;
   x =0; y = 0;
    #20 begin x= 1; y = 1;  end
    #30 begin x = 0; y = 0; end
    #40 begin x = 1; y = 0; end
    #50 begin x = 0; y = 0; end
    #60 begin x = 1; y = 1; end
    #70 begin x = 1; y = 0; end
    #80 begin x = 0; y = 1; end
  join
 endmodule
```



5.29

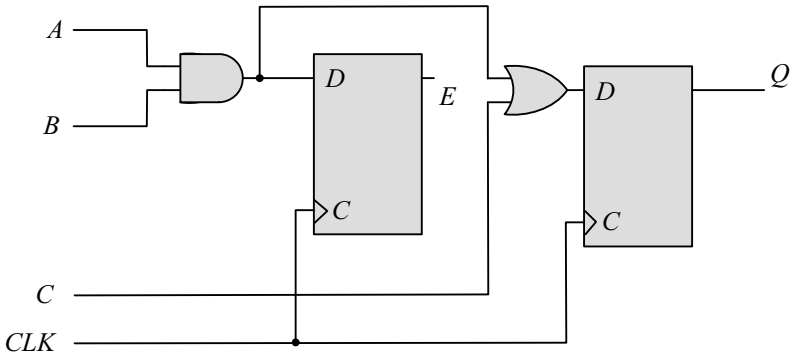
```
module Prob_5_29 (output reg y_out, input x_in, clock, reset_b);
 parameter s0 = 3'b000, s1 = 3'b001, s2 = 3'b010, s3 = 3'b011, s4 = 3'b100;
 reg [2: 0] state, next_state;

 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) state <= s0;
  else state <= next_state;

 always @ (state, x_in) begin
  y_out = 0;
  next_state = s0;
  case (state)
    s0:     if (x_in) begin next_state = s4; y_out = 1; end else begin next_state = s3; y_out = 0; end
    s1:     if (x_in) begin next_state = s4; y_out = 1; end else begin next_state = s1; y_out = 0; end
    s2:     if (x_in) begin next_state = s0; y_out = 1; end else begin next_state = s2; y_out = 0; end
    s3:     if (x_in) begin next_state = s2; y_out = 1; end else begin next_state = s1; y_out = 0; end
    s4:     if (x_in) begin next_state = s3; y_out = 0; end else begin next_state = s2; y_out = 0; end
    default:   next_state = 3'bxxx;
  endcase
 end
endmodule
```

```
module t_Prob_5_29 ();
  wire y_out;
  reg x_in, clk, reset_b;

  Prob_5_29 M0 (y_out, x_in, clk, reset_b);

  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;      // Initialize to s0
    #4 reset_b = 1;
                         // Trace the state diagram and monitor y_out
     x_in = 0;           // Drive from s0 to s3 to  S1 and park
    #40 x_in = 1;        // Drive to s4 to s3 to s2 to s0 to s4 and loop
    #90 x_in = 0;        // Drive from s0 to s3 to s2 and part
    #110  x_in = 1;      // Drive s0 to s4  etc
  join
endmodule
```

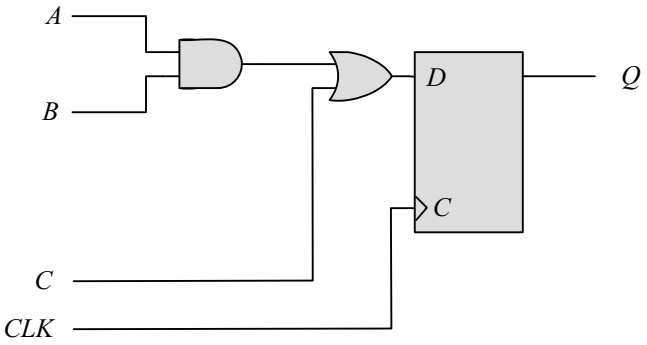


5.30

With non-blocking (<=) assignment operator:



With blocking (=) assignment operator:



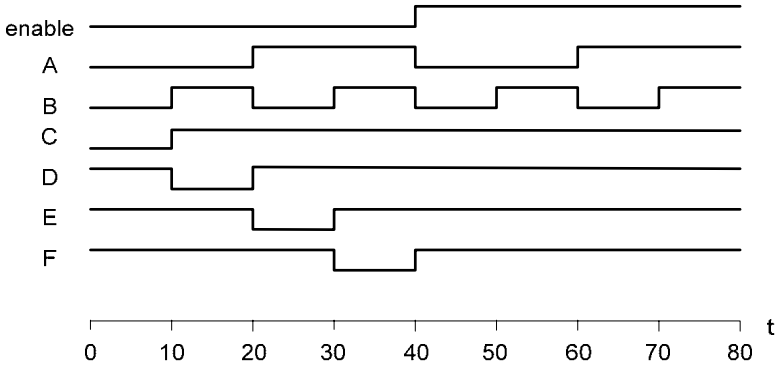
Note: The expression substitution implied by the sequential ordering with the blocking assignment operator results. in the elimination of E by a synthesis tool. To retain E, it is necessary to declare E to be an output port of the module.

5.31

```
module Seq_Ckt (input A, B, C, CLK, output reg Q);
  reg E;
  always @ (posedge CLK)
  begin
    Q = E | C;
    E = A & B;
  end
endmodule
```

Note: The statements must be written in an order than produces the effect of concurrent assignments.

5.32

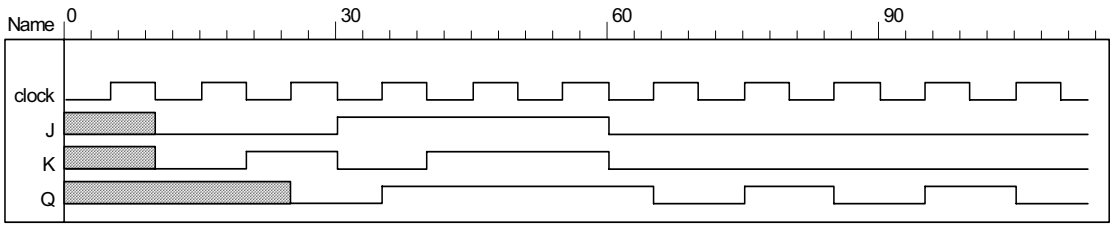



```
initial begin
  enable = 0; A = 0; B = 0; C = 0; D = 1; E = 1; F = 1;
  # 10  B = 1;
     C = 1;
     D = 0;
#10     A = 1;
     B = 0;
     D = 1;
     E = 0;
#10     B = 1;
     E = 1;
     F = 0;
#10     enable = 1;
     A = 0;
     B = 0;
     F = 0;
#10     B = 1;
#10     A = 1;
     B = 0;
#10     B = 1;
end

initial fork
  enable = 0; A = 0; B = 0; C = 0; D = 1; E = 1; F = 1;
  #40 enable = 1;
  #20 A = 1;
  #40 A = 0;
  #60 A = 1;
  #10 B = 1;
  #20 B = 0;
  #30 B = 1;
  #40 B = 0;
  #50 B = 1;
  #60 B = 0;
  #70 B = 1;
  #10 C = 1;
  #10 D = 0;
  #20 D = 1;
  #20 E = 0;
  #30 E = 1;
  #30 F = 0;
  #40 F = 1;
join
```

5.33 Signal transitions that are caused by input signals that change on the active edge of the clock race with the clock itself to reach the affected flip-flops, and the outcome is indeterminate (unpredictable). Conversely, changes caused by inputs that are synchronized to the inactive edge of the clock reach stability before the active edge, with predictable outputs of the flip-flops that are affected by the inputs.

5.34

```
module JK_flop_Prob_5_34 (output Q, input J, K, clk);
  wire K_bar;
  D_flop M0 (Q, D, clk);
  Mux M1 (D, J, K_bar, Q);
  Inverter M2 (K_bar, K);
endmodule

module D_flop (output reg Q, input D, clk);
  always @ (posedge clk) Q <= D;
endmodule

module Inverter (output y_bar, input y);
  assign y_bar = ~y;
endmodule

module Mux (output y, input a, b, select);
  assign y = select ? a: b;
endmodule

module t_JK_flop_Prob_5_34 ();
  wire  Q;
  reg   J, K, clock;
  JK_flop_Prob_5_34 M0 (Q, J, K, clock);
  initial #500 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
  #10 begin J = 0; K = 0; end      // toggle Q unknown
  #20 begin J = 0; K = 1; end      // set Q to 0
  #30 begin J = 1; K = 0; end      // set q to 1
  #40 begin J = 1; K = 1; end      // no change
  #60 begin J = 0; K = 0; end      // toggle Q
  join
endmodule
```



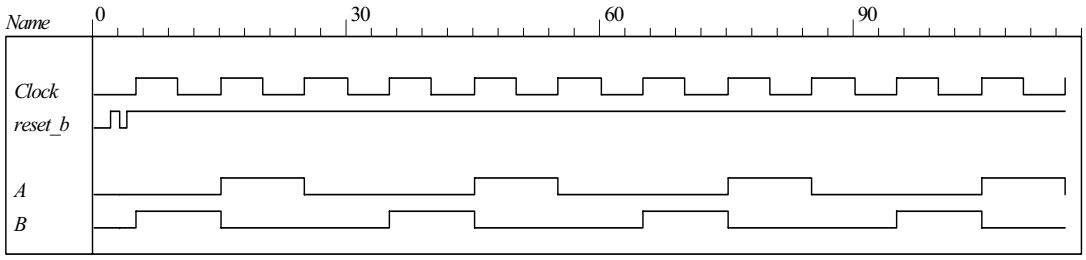
5.35

```
initial begin
  enable = 0; A = 0; B = 0; C = 0; D = 1; E = 1; F = 1;
  #10 begin B = 1; C = 1; D = 0; end
  #10 begin A = 1; B = 0; D = 1; E = 0; end
  #10 begin A = 1; B = 0; E = 1; F = 0; end
  #10 begin enable = 1; A = 0; B = 0; F = 1; end
  #10 begin B = 1; end
  #10 begin A = 1; B = 0; end
  #10 B = 1;
end
```

```
initial fork
  enable = 0;
  #40 enable = 1;
  #20 A = 1;
  #40 A =0;
  #60 A = 1;

  #10 B = 1;
  #20 B = 0;
  #30 B = 1;
  #40 B = 0;
  #50 B = 1;
  #60 B = 0;
  #70 B = 1;

  #10 C = 1;

  #10 D = 0;
  #20 D = 1;

  #20 E = 0;
  #30 E = 1;

  #30 F = 0;
  #40 F = 1;
join
```

5.36 Note: See Problem 5.8 (counter with repeated sequence: (A, B) = 00, 01, 10, 00 ....

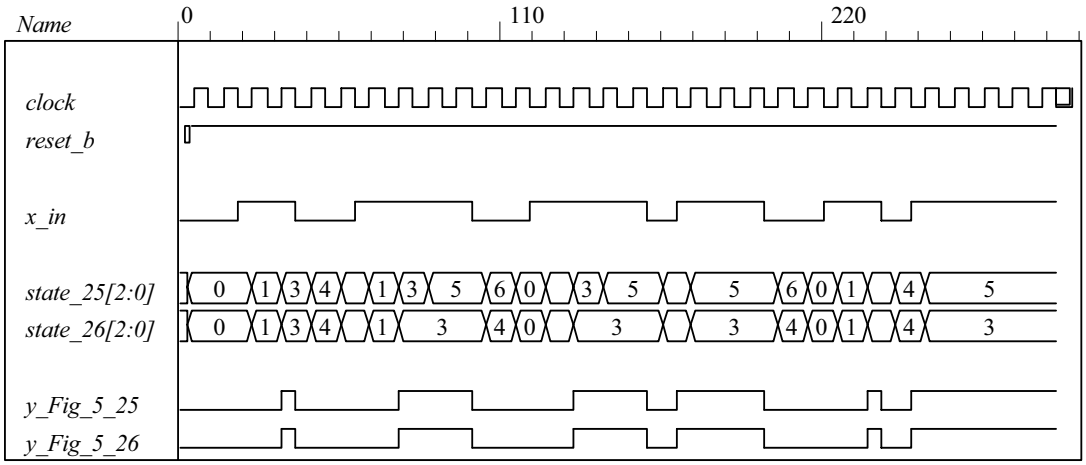
```
// See Fig. P5.8
module Problem_5_36 (output A, B, input Clock, reset_b);
  or (T_A, A, B);
  or (T_B, A_b, B);
  T_flop M0 (A, A_b, T_A, Clock, reset_b);
  T_flop M1 (B, B_b, T_B, Clock, reset_b);
endmodule


module T_flop (output reg Q, output QB, input T, Clock, reset_b);
  assign QB = ~ Q;
  always @ (posedge Clock, negedge reset_b)
    if (reset_b == 0) Q <= 0;
    else if (T) Q <= ~Q;
endmodule

module t_Problem_5_36 ();
  wire A, B;
  reg  Clock, reset_b;

  Problem_5_36 M0 (A, B, Clock, reset_b);

  initial #350$finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;
    #4 reset_b = 1;
  join
endmodule
```



5.37

```
module Problem_5_37_Fig_5_25 (output reg y, input x_in, clock, reset_b);

 parameter a = 3'b000, b = 3'b001, c = 3'b010, d = 3'b011, e = 3'b100, f = 3'b101, g = 3'b110;
 reg [2: 0] state, next_state;

 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) state <= a;
  else state <= next_state;

 always @ (state, x_in) begin
  y = 0;
  next_state = a;
  case (state)
   a:      begin y = 0; if (x_in == 0) next_state = a; else next_state = b; end

   b:      begin y = 0; if (x_in == 0) next_state = c; else next_state = d; end

   c:      begin y = 0; if (x_in == 0) next_state = a; else next_state = d; end

   d:      if (x_in == 0) begin y = 0; next_state = e; end
           else begin y = 1; next_state = f; end

   e:      if (x_in == 0) begin y = 0;  next_state = a; end
           else begin y = 1; next_state = f; end

   f:      if (x_in == 0) begin y = 0;  next_state = g; end
           else begin y = 1; next_state = f; end

   g:      if (x_in == 0) begin y = 0; next_state = a; end
           else begin y = 1; next_state = f; end

    default:   next_state = a;
  endcase
 end
endmodule
module Problem_5_37_Fig_5_26 (output reg y, input x_in, clock, reset_b);
 parameter a = 3'b000, b = 3'b001, c = 3'b010, d = 3'b011, e = 3'b100;
 reg [2: 0] state, next_state;

 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) state <= a;
  else state <= next_state;
```

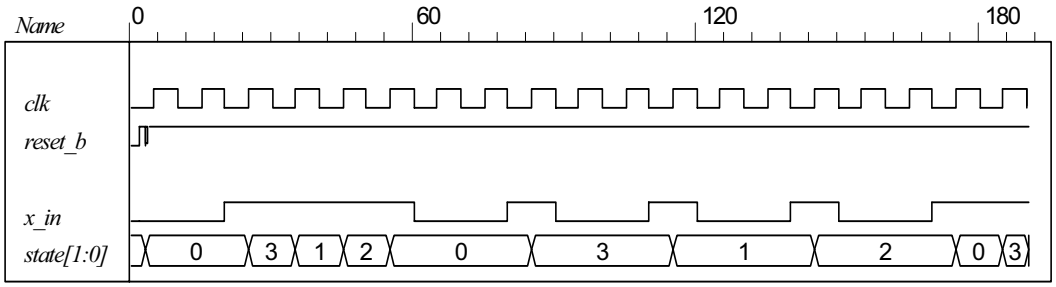
```
  always @ (state, x_in) begin
    y = 0;
    next_state = a;
    case (state)
      a:      begin y = 0; if (x_in == 0) next_state = a; else next_state = b; end

      b:      begin y = 0; if (x_in == 0) next_state = c; else next_state = d; end

      c:      begin y = 0; if (x_in == 0) next_state = a; else next_state = d; end

      d:      if (x_in == 0) begin y = 0; next_state = e; end
              else begin y = 1; next_state = d; end

      e:      if (x_in == 0) begin y = 0;  next_state = a; end
              else begin y = 1; next_state = d; end

      default:   next_state = a;
    endcase
  end
endmodule

module t_Problem_5_37 ();
  wire y_Fig_5_25, y_Fig_5_26;
  reg x_in, clock, reset_b;

  Problem_5_37_Fig_5_25 M0 (y_Fig_5_25, x_in, clock, reset_b);
  Problem_5_37_Fig_5_26 M1 (y_Fig_5_26, x_in, clock, reset_b);

  wire [2: 0] state_25 = M0.state;
  wire [2: 0] state_26 = M1.state;

  initial #350 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
    x_in = 0;
    #2 reset_b = 1;
    #3 reset_b = 0;
    #4 reset_b = 1;
    #20 x_in = 1;
    #40 x_in = 0;    // abdea, abdea

    #60 x_in = 1;
    #100 x_in = 0;  // abdf....fga, abd ... dea

    #120 x_in = 1;
    #160 x_in = 0;
    #170 x_in = 1;
    #200 x_in = 0; // abdf....fgf...fga, abd ...ded...ea

    #220 x_in = 1;
    #240 x_in = 0;
    #250 x_in = 1;  // abdef...  // abded...
  join
endmodule
```



5.38 (a)

```
module Prob_5_38a (input x_in, clock, reset_b);
 parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
 reg [1: 0] state, next_state;

 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) state <= s0;
  else state <= next_state;

 always @ (state, x_in) begin
   next_state = s0;
  case (state)
   s0:     if (x_in == 0) next_state = s0;
           else if (x_in == 1) next_state = s3;

   s1:     if (x_in == 0) next_state = s1;
           else if (x_in == 1) next_state = s2;

   s2:     if (x_in == 0) next_state = s2;
           else if (x_in == 1) next_state = s0;

   s3:     if (x_in == 0) next_state = s3;
           else if (x_in == 1) next_state = s1;
    default:       next_state = s0;
  endcase
 end
endmodule
```

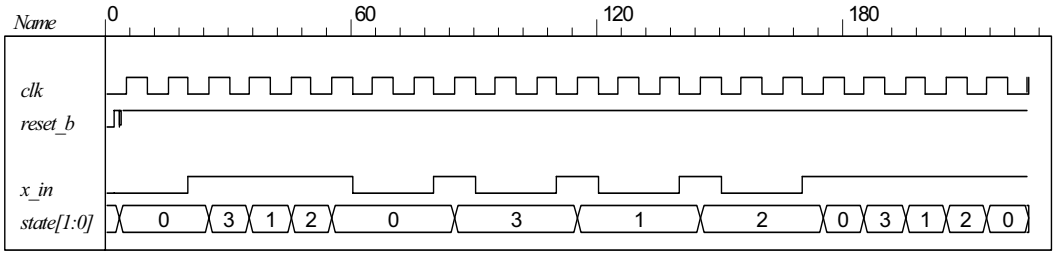
```
module t_Prob_5_38a ();
  reg x_in, clk, reset_b;

  Prob_5_38a M0 ( x_in, clk, reset_b);

  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;      // Initialize to s0
    #4 reset_b = 1;
    #2 x_in = 0;
    #20 x_in = 1;
    #60 x_in = 0;
    #80 x_in = 1;
    #90 x_in = 0;
    #110 x_in = 1;
    #120 x_in = 0;
    #140 x_in = 1;
    #150 x_in = 0;
    #170 x_in= 1;
  join
endmodule
```



(b)

```
module Prob_5_38b (input x_in, clock, reset_b);
  parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
  reg [1: 0] state, next_state;

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= s0;
    else state <= next_state;

  always @ (state, x_in) begin
     next_state = s0;
    case (state)
      s0:     if (x_in == 0) next_state = s0;
              else if (x_in == 1) next_state = s3;

      s1:     if (x_in == 0) next_state = s1;
              else if (x_in == 1) next_state = s2;

      s2:     if (x_in == 0) next_state = s2;
              else if (x_in == 1) next_state = s0;

      s3:     if (x_in == 0) next_state = s3;
              else if (x_in == 1) next_state = s1;
      default:        next_state = s0;
    endcase
  end
```

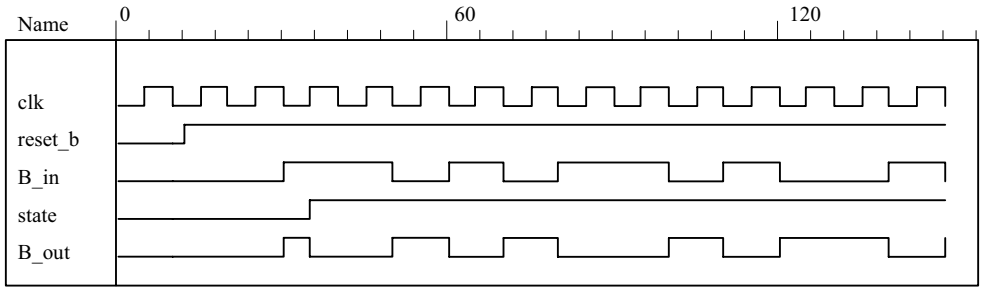
```
endmodule

module t_Prob_5_38b ();

  reg x_in, clk, reset_b;

  Prob_5_38b M0 ( x_in, clk, reset_b);

  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;      // Initialize to s0
    #4 reset_b = 1;
    #2 x_in = 0;
    #20 x_in = 1;
    #60 x_in = 0;
    #80 x_in = 1;
    #90 x_in = 0;
    #110 x_in = 1;
    #120 x_in = 0;
    #140 x_in = 1;
    #150 x_in = 0;
    #170 x_in= 1;
   join
endmodule
```



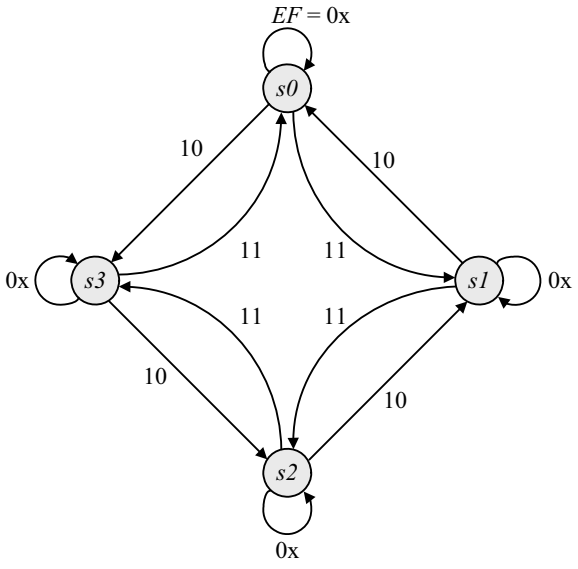
5.39

```
module Serial_2s_Comp (output reg B_out, input B_in, clk, reset_b);
// See problem 5.17
  parameter S_0 = 1'b0, S_1 = 1'b1;
  reg state, next_state;
  always @ (posedge clk, negedge reset_b) begin
    if (reset_b == 0) state <= S_0;
    else state <= next_state;
  end

  always @ (state, B_in) begin
    B_out = 0;
    case (state)
      S_0:    if (B_in == 0) begin next_state = S_0; B_out = 0; end
           else if (B_in == 1) begin next_state = S_1;  B_out = 1; end

      S_1:    begin next_state = S_1; B_out = ~B_in; end
     default:     next_state = S_0;

    endcase
  end
endmodule
```

```
module t_Serial_2s_Comp ();
  wire B_in, B_out;
  reg clk, reset_b;
  reg [15: 0] data;
  assign B_in = data[0];

  always @ (negedge clk, negedge reset_b)
    if (reset_b == 0) data <= 16'ha5ac; else data <= data >> 1;  // Sample bit stream

  Serial_2s_Comp M0 (B_out, B_in, clk, reset_b);

  initial #150 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #10 reset_b = 0;
    #12 reset_b = 1;
  join
endmodule
```

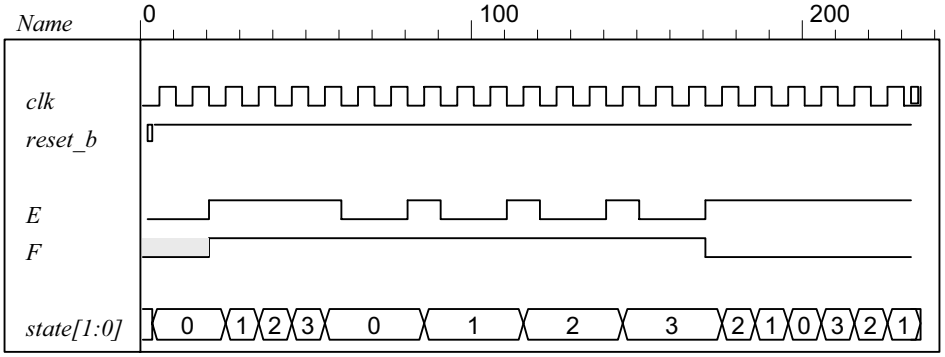


5.40



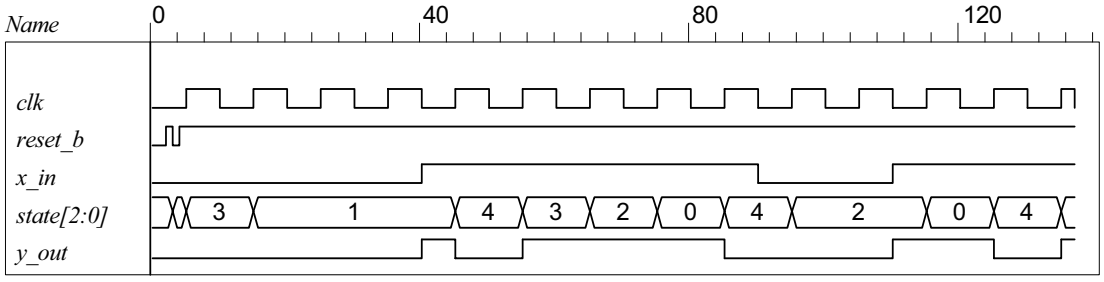

```
module Prob_5_40 (input E, F, clock, reset_b);
  parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
  reg [1: 0] state, next_state;

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= s0;
    else state <= next_state;
```

```
  always @ (state, E, F) begin
    next_state = s0;
   case (state)
     s0:      if (E == 0) next_state = s0;
              else if (F == 1) next_state = s1; else next_state = s3;

     s1:      if (E == 0) next_state = s1;
              else if (F == 1) next_state = s2; else next_state = s0;

     s2:      if (E == 0) next_state = s2;
              else if (F == 1) next_state = s3; else next_state = s1;

     s3:      if (E == 0) next_state = s3;
              else if (F == 1) next_state = s0; else next_state = s2;
     default:    next_state = s0;
   endcase
  end
endmodule

module t_Prob_5_40 ();

  reg E, F, clk, reset_b;

  Prob_5_40 M0 ( E, F, clk, reset_b);

  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
   #2 reset_b = 1;
   #3 reset_b = 0;      // Initialize to s0
   #4 reset_b = 1;
   #2 E = 0;
   #20 begin E = 1; F = 1; end
   #60 E = 0;
   #80 E = 1;
   #90 E = 0;
   #110 E = 1;
   #120 E = 0;
   #140 E = 1;
   #150 E = 0;
   #170 E= 1;
   #170 F = 0;
  join
endmodule
```



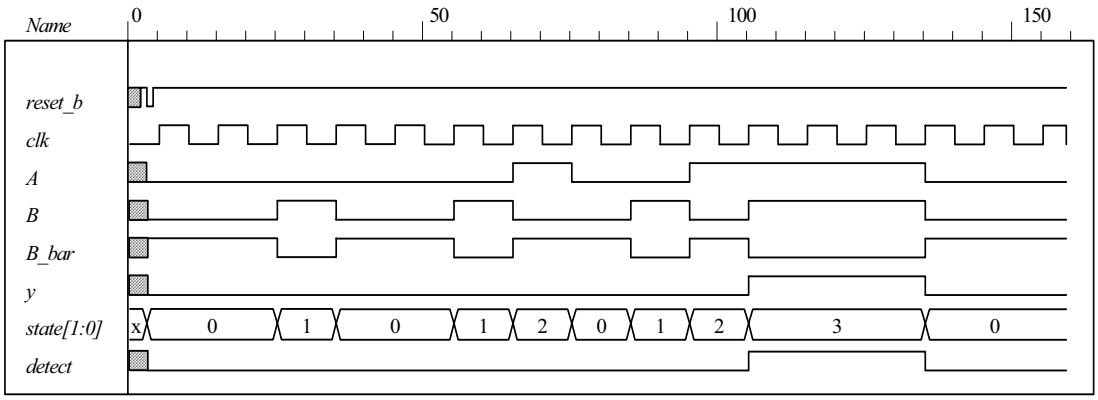
5.41

```
module Prob_5_41 (output reg y_out, input x_in, clock, reset_b);
  parameter s0 = 3'b000, s1 = 3'b001, s2 = 3'b010, s3 = 3'b011, s4 = 3'b100;
  reg [2: 0] state, next_state;

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= s0;
    else state <= next_state;

  always @ (state, x_in) begin
    y_out = 0;
    next_state = s0;
    case (state)
      s0:      if (x_in) begin next_state = s4; y_out = 1; end else begin next_state = s3; y_out = 0; end
      s1:      if (x_in) begin next_state = s4; y_out = 1; end else begin next_state = s1; y_out = 0; end
      s2:      if (x_in) begin next_state = s0; y_out = 1; end else begin next_state = s2; y_out = 0; end
      s3:      if (x_in) begin next_state = s2; y_out = 1; end else begin next_state = s1; y_out = 0; end
      s4:      if (x_in) begin next_state = s3; y_out = 0; end else begin next_state = s2; y_out = 0; end
      default:    next_state = 3'bxxx;
    endcase
  end
endmodule

module t_Prob_5_41 ();
  wire y_out;
  reg x_in, clk, reset_b;

  Prob_5_41 M0 (y_out, x_in, clk, reset_b);

  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;      // Initialize to s0
    #4 reset_b = 1;
              // Trace the state diagram and monitor y_out
     x_in = 0;          // Drive from s0 to s3 to  S1 and park
    #40 x_in = 1;         // Drive to s4 to s3 to s2 to s0 to s4 and loop
    #90 x_in = 0;         // Drive from s0 to s3 to s2 and part
    #110  x_in = 1;       // Drive s0 to s4  etc
  join
endmodule
```



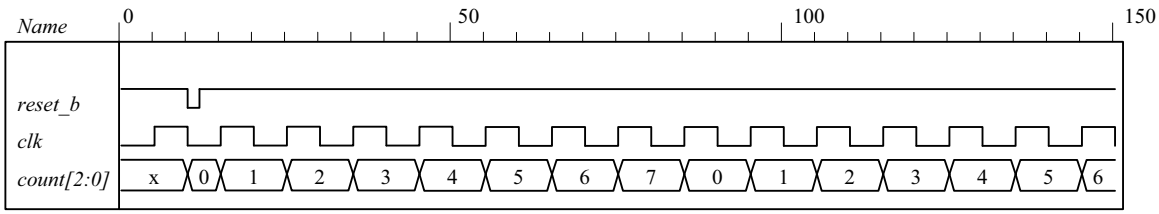
5.42

```
module Prob_5_42 (output A, B, B_bar, y, input x, clk, reset_b);
// See Fig. 5.29
  wire w1, w2, w3, D1, D2;
  and (w1, A, x);
  and (w2, B, x);
  or (D_A, w1, w2);

  and (w3, B_bar, x);
  and (y, A, B);
  or (D_B, w1, w3);
  DFF M0_A (A, D_A, clk, reset_b);
  DFF M0_B (B, D_B, clk, reset_b);
  not (B_bar, B);
endmodule

module DFF (output reg Q, input data, clk, reset_b);
  always @ (posedge clk, negedge reset_b)
  if (reset_b == 0) Q <= 0; else Q <= data;
endmodule

module t_Prob_5_42 ();
  wire A, B, B_bar, y;
  reg bit_in, clk, reset_b;
  wire [1:0] state;
  assign state = {A, B};
  wire detect = y;

  Prob_5_42 M0 (A, B, B_bar, y, bit_in, clk, reset_b);

  // Patterns from Problem 5.45.

  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;
    #4reset_b = 1;
                          // Trace the state diagram and monitor detect (assert in S3)
        bit_in = 0;       // Park in S0
    #20 bit_in = 1;       // Drive to S0
    #30 bit_in = 0;       // Drive to S1 and back to S0 (2 clocks)
    #50 bit_in = 1;
    #70 bit_in = 0;       // Drive to S2 and back to S0 (3 clocks)
    #80 bit_in = 1;
    #130 bit_in = 0;// Drive to S3, park, then  and back to S0
   join
endmodule
```



5.43

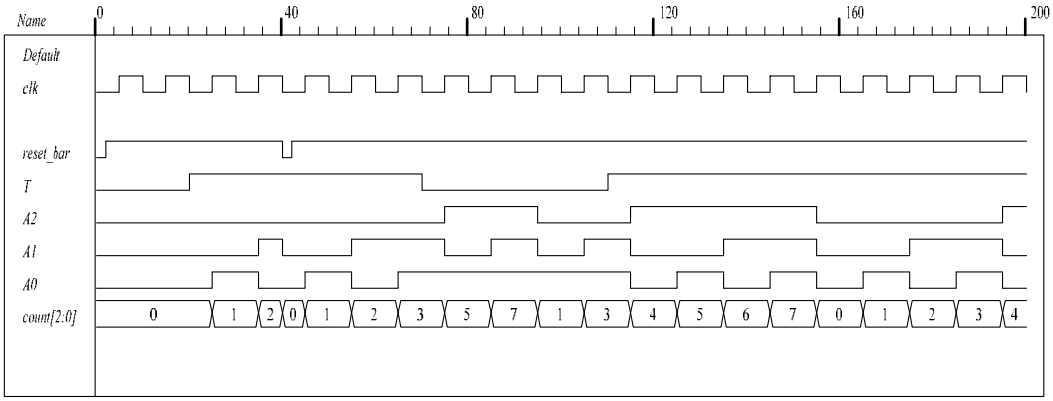
```
module Binary_Counter_3_bit (output [2: 0] count, input clk, reset_b)
  always @ (posedge clk) if (reset_b == 0) count <= 0; else count <= next_count;
  always @ (count) begin
    case (state)
      3'b000:    count = 3'b001;
      3'b001:    count = 3'b010;
      3'b010:    count = 3'b011;
      3'b011:    count = 3'b100;
      3'b100:    count = 3'b001;
      3'b101:    count = 3'b010;
      3'b110:    count = 3'b011;
      3'b111:    count = 3'b100;
     default:    count = 3'b000;
    endcase
  end
endmodule

module t_Binary_Counter_3_bit ()
  wire [2: 0] count;
  reg clk, reset_b;
  Binary_Counter_3_bit M0 ( count, clk, reset_b)

  initial #150 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    reset = 1;
    #10 reset = 0;
    #12 reset = 1;
endmodule
```



Alternative: structural model.

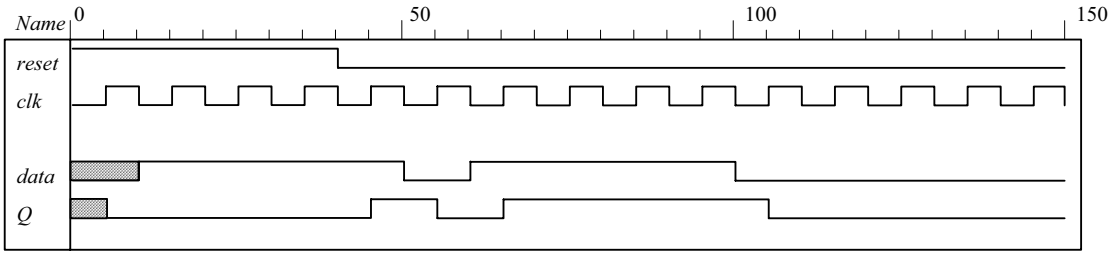
```
module Prob_5_41 (output A2, A1, A0, input T, clk, reset_bar);
  wire   toggle_A2;

  T_flop M0 (A0, T, clk, reset_bar);
  T_flop M1 (A1, A0, clk, reset_bar);
  T_flop M2 (A2, toggle_A2, clk, reset_bar);
  and (toggle_A2, A0, A1);
endmodule

module T_flop (output reg Q, input T, clk, reset_bar);
  always @ (posedge clk, negedge reset_bar)
    if (!reset_bar) Q <= 0; else if (T) Q <= ~Q; else Q <= Q;
endmodule

module t_Prob_5_41;
  wire A2, A1, A0;
  wire [2: 0] count = {A2, A1, A0};
  reg T, clk, reset_bar;
  Prob_5_41 M0 (A2, A1, A0, T, clk, reset_bar);

  initial #200 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork reset_bar = 0; #2 reset_bar = 1; #40 reset_bar = 0; #42 reset_bar = 1; join
  initial fork T = 0; #20 T = 1; #70 T = 0; #110 T = 1; join
endmodule
```

If the input to $A0$ is changed to 0 the counter counts incorrectly. It resumes a correct counting sequence when T is changed back to 1.



5.44

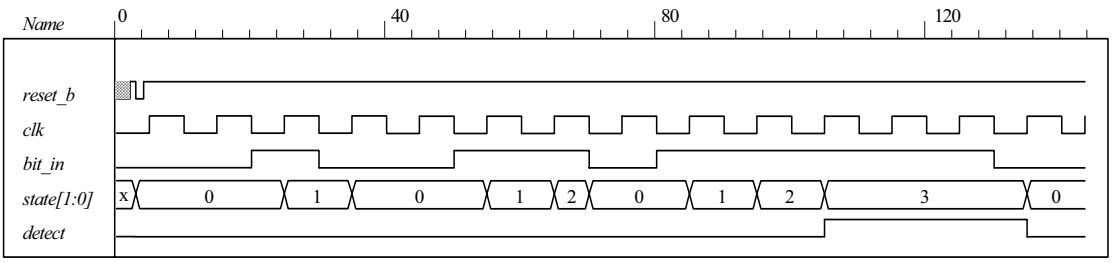
```
module DFF_synch_reset (output reg Q, input data, clk, reset);
 always @ (posedge clk)
 if (reset) Q <= 0; else Q <= data;
endmodule

module t_DFF_synch_reset ();
 reg data, clk, reset;
 wire Q;

 DFF_synch_reset M0 (Q, data, clk, reset);

 initial #150 $finish;
 initial begin clk = 0; forever #5 clk = ~clk; end
 initial fork
  reset = 1;
  #20 reset = 1;
  #40 reset = 0;

  #10 data = 1;
  #50 data = 0;
  #60 data = 1;
  #100 data = 0;
 join
endmodule
```



5.45

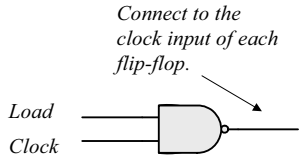
```
module Seq_Detector_Prob_5_45 (output detect, input bit_in, clk, reset_b);
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
reg [1: 0] state, next_state;

assign detect = (state == S3);
always @ (posedge clk, negedge reset_b)
if (reset_b == 0) state <= S0; else state <= next_state;

 always @ (state, bit_in) begin
  next_state = S0;
  case (state)
   0:      if (bit_in) next_state = S1; else state = S0;
   1:      if (bit_in) next_state = S2; else next_state = S0;
   2:      if (bit_in) next_state = S3; else state = S0;
   3:      if (bit_in) next_state = S3; else next_state = S0;
   default:   next_state = S0;
  endcase
 end
endmodule
```

```
module t_Seq_Detector_Prob_5_45 ();
 wire detect;
 reg bit_in, clk, reset_b;

 Seq_Detector_Prob_5_45 M0 (detect, bit_in, clk, reset_b);

 initial #350$finish;
 initial begin clk = 0; forever #5 clk = ~clk; end
 initial fork
  #2 reset_b = 1;
  #3 reset_b = 0;
  #4reset_b = 1;
                        // Trace the state diagram and monitor detect (assert in S3)
       bit_in = 0;      // Park in S0
  #20 bit_in = 1;       // Drive to S0
  #30 bit_in = 0;       // Drive to S1 and back to S0 (2 clocks)
  #50 bit_in = 1;
  #70 bit_in = 0;       // Drive to S2 and back to S0 (3 clocks)
  #80 bit_in = 1;
  #130 bit_in = 0;// Drive to S3, park, then  and back to S0
 join
endmodule
```

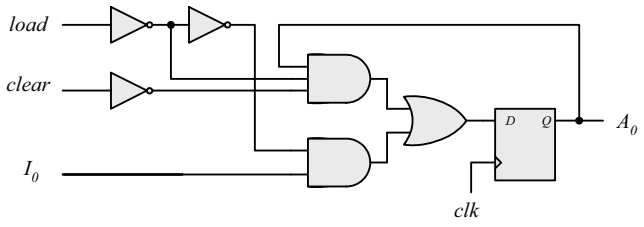


CHAPTER 6

6.1 The structure shown below gates the clock through a nand gate. In practice, the circuit can exhibit two problems if the load signal is asynchronous: (1) the gated clock arrives in the setup interval of the clock of the flip-flop, causing metastability, and (2) the load signal truncates the width of the clock pulse. Additionally, the propagation delay through the nand gate might compromise the synchronicity of the overall circuit.



6.2 Modify Fig. 6.2, with each stage replicating the first stage shown below:



Load	*Clear*	*D*	*Operation*
0	*0*	A_0	*No change*
0	*1*	*0*	*Clear to 0*
1	*x*	I_0	*Load input*

Note: In this design, *load* has priority over *clear*.

6.3 Serial data is transferred one bit at a time. Parallel data is transferred n bits at a time (n > 1).

A shift register can convert serial data into parallel data by first shifting one bit a time into the register and then taking the parallel data from the register outputs.

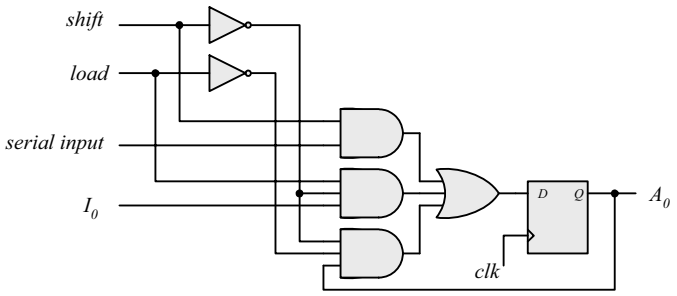
A shift register with parallel load can convert parallel data to a serial format by first loading the data in parallel and then shifting the bits one at a time.

6.4 101101 ⇒ 1101; 0110; 1011; 1101; 0110; 1011

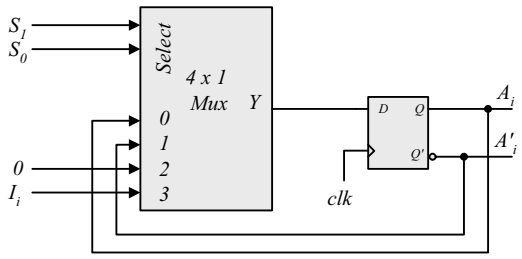
6.5 **(a)** See Fig. 11.19: IC 74194

(b) See Fig. 11.20. Connect two 74194 ICs to form an 8-bit register.

6.6 First stage of register:



6.7 First stage of register:

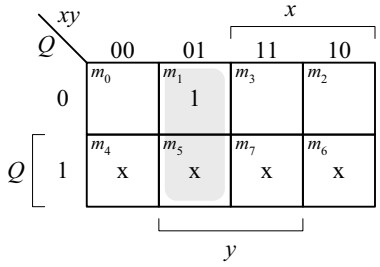


6.8 A = 0010, 0001, 1000, 1100. Carry = 1, 1, 1, 0

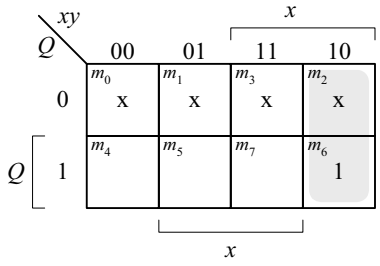
6.9 **(a)** In Fig. 6.5, complement the serial output of shift register B (with an inverter), and set the initial value of the carry to 1.

(b)

Present state Q	Inputs x	y	Next state Q	Output D	FF inputs J_Q	K_Q
0	0	0	0	0	0	x
0	0	0	1	1	1	x
0	0	1	0	1	0	x
0	0	1	0	0	0	x
1	1	0	1	1	x	0
1	1	0	1	0	x	0
1	1	1	0	0	x	1
1	1	1	1	1	x	0



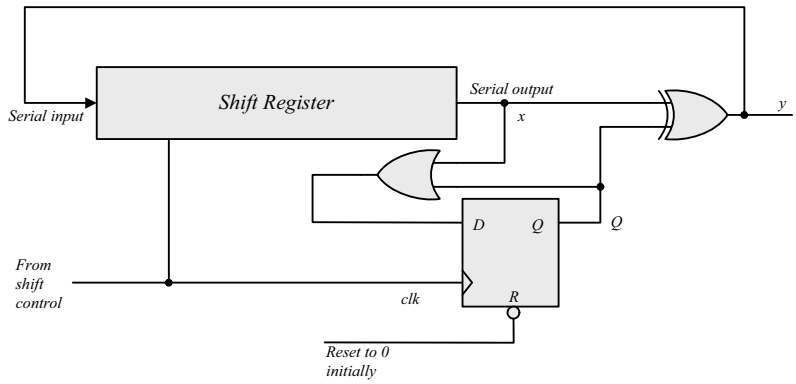
$J_Q = x'y$



$K_Q = xy' \oplus \quad \oplus$

$D = Q \oplus x \oplus y$

6.10 See solution to Problem 5.7.
Note that y = x if Q = 0, and y = x' if Q = 1. Q is set on the first 1 from x.
Note that x ⊕ 0 = x, and x ⊕ 1 = x'.



6.11 **(a)** A count down counter.

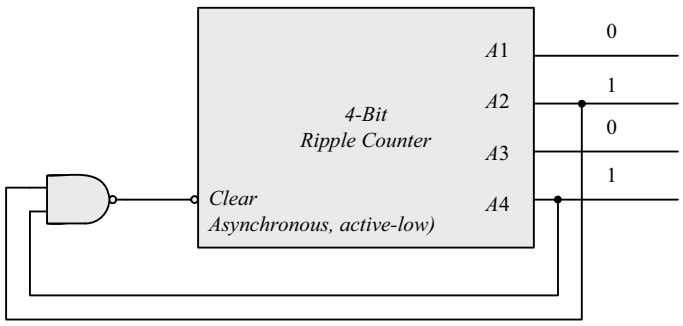
(b) A count up counter.

6.12 Similar to diagram of Fig. 6.8.

(a) With the bubbles in *C* removed (positive-edge).

(b) With complemented flip-flops connected to *C*.

6.13



6.14 **(a)** 4; **(b)** 9; **(c)** 10

6.15 The worst case is when all 10 flip-flops are complemented. The maximum delay is 10 x 3ns = 30 ns.

The maximum frequency is $10^9/30 = 33.3$ MHz

6.16

Q8 Q4 Q2 Q1 :	1010	1100	1110	Self correcting
Next state:	1011	1101	1111	
Next state:	0100	0100	0000	

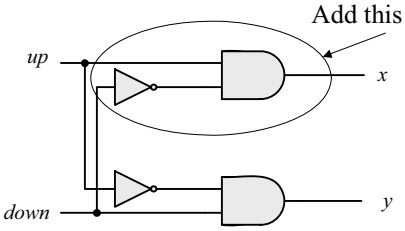
$1010 \rightarrow 1011 \rightarrow 0100$
$1100 \rightarrow 1101 \rightarrow 0100$
$1110 \rightarrow 1111 \rightarrow 0000$

6.17 With E denoting the count enable in Fig. 6.12 and D-flip-flops replacing the J-K flip-flops, the toggling action of the bits of the counter is determined by: $T_0 = E$, $T_1 = A_0E$, $T_2 = A_0A_1E$, $T_3 = A_0A_1A_2E$. Since $D_A = A \oplus T_A$ the inputs of the flip-flops of the counter are determined by: $D_{A0} = A_0 \oplus E$; $D_{A1} = A_1 \oplus (A_0E)$; $D_{A2} = A_2 \oplus (A_0A_1E)$; $D_{A3} = A_3 \oplus (A_0A_1A_2E)$.

6.18 When *up* = *down* = 1 the circuit counts up.

up	*down*	*x*	*y*	Operation
0	0	0	0	No change
0	1	0	0	Count down
1	0	1	0	Count up
1	1	0	0	No change

Add this to Fig. 6.13



x = up (down)'
y = (up)'down

6.19 **(b)** From the state table in Table 6.5:

$D_{Q1} = Q'_1$
$D_{Q2} = \sum (1, 2, 5, 6)$
$D_{Q4} = \sum (3, 4, 5, 6)$
$D_{Q8} = \sum (7, 8)$
Don't care: $d = \sum (10, 11, 12, 13, 14, 15)$

Simplifying with maps:
$D_{Q2} = Q_2Q'_1 + Q'_8Q'_2Q_1$
$D_{Q4} = Q_4Q'_1 + Q4Q'2 + Q'_4Q_2Q_1$
$D_{Q8} = Q_8Q'_1 + Q_4Q_2Q_1$

(a)

Present state	*Next state*	*Flip-flop inputs*			
$A_8\,A_4\,A_2\,A_1$	$A_8\,A_4\,A_2\,A_1$	$J_{A8}\,K_{A8}$	$J_{A4}\,K_{A4}$	$J_{A2}\,K_{A2}$	$J_{A1}\,K_{A1}$
0 0 0 0	0 0 0 1	0 x	0 x	0 x	1 x
0 0 0 1	0 0 1 0	0 x	0 x	1 x	x 1
0 0 1 0	0 0 1 1	0 x	0 x	x 0	1 x
0 0 1 1	0 1 0 0	0 x	1 x	x 1	x 1
0 1 0 0	0 1 0 1	0 x	x 0	0 x	1 x
0 1 0 1	0 1 1 0	0 x	x 0	1 x	x 1
0 1 1 0	0 1 1 1	0 x	x 0	x 0	1 x
0 1 1 1	1 0 0 0	1 x	x 1	x 1	x 1
1 0 0 0	1 0 0 1	x 0	0 x	0 x	1 x
1 0 0 1	0 0 0 0	x 1	0 x	0 x	x 1

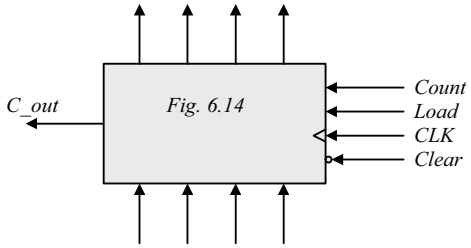
$J_{A1} = 1$
$K_{A1} = 1$
$J_{A2} = A_1A'_8$
$K_{A2} = A_1$
$J_{A4} = A_1A_2$
$K_{A4} = A_1A_2$
$J_{A8} = A_1A_2A_4$
$K_{A8} = A_1$

$d(A_8, A_4, A_2, A_1) = \Sigma\,(10, 11, 12, 13, 14, 15)$

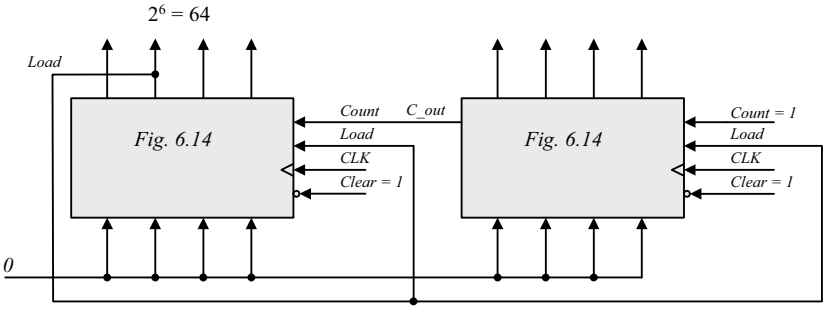
6.20 **(a)**

Block diagram of 4-bit circuit:

16-bit counter needs 4 circuits with output carry connected to the count input of the next stage.



(b)



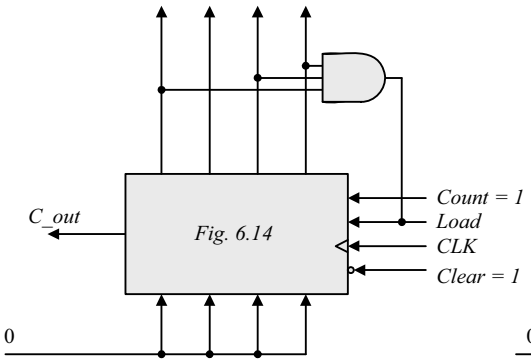
6.21 **(a)**

$J_{A0} = LI_0 + L'C \qquad KA_0 = LI'_0 + L'C$

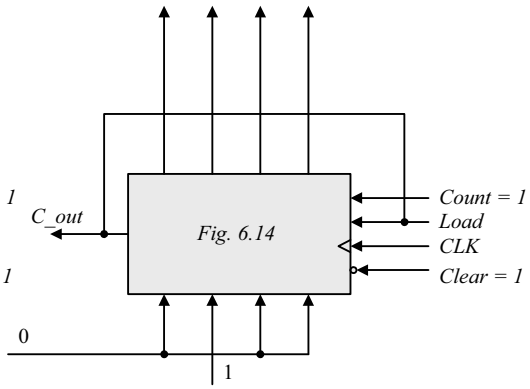
(b)

$J = [L(LI)']'(L + C) = (L' + LI)(L + C)$
$LI + L'C + LIC = LI + L'C$ *(use a map)*
$K = (LI)'\,(L + C) = (L' + I')(L + C) = LI' + L'C$

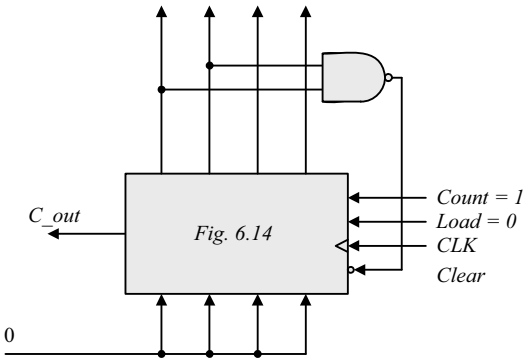
6.22



Count sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11



Count sequence: 4, 5, 6, 7, 8, 9, 10, 11, 1,2 13, 14, 15

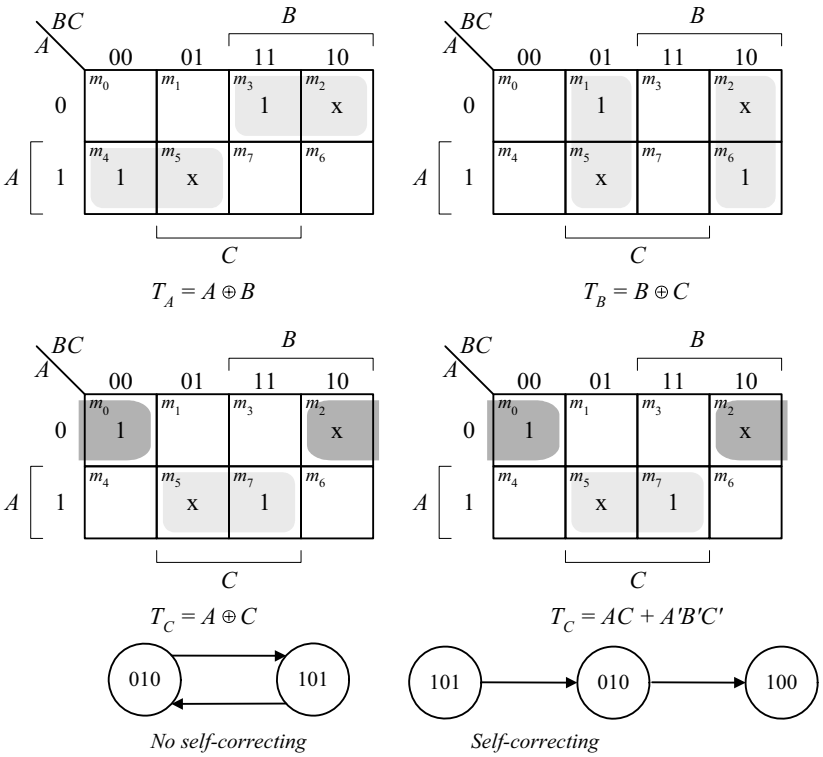


Count sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

6.23 Use a 3-bit counter and a flip-flop (initially at 0). A start signal sets the flip-flop, which in turn enables the counter. On the count of 7 (binary 111) reset the flip-flop to 0 to disable the count (with the value of 00 0).

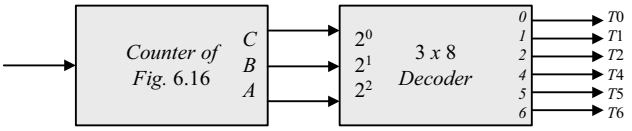
6.24

Present state *ABC*	*Next state* *ABC*	*Flip-flop inputs* T_A	T_B	T_C
000	001	0	0	1
001	011	0	1	0
010	xxx	x	x	x
011	111	1	1	0
100	000	1	1	0
101	xxx	x	x	x
110	100	0	1	0
111	110	0	0	1



6.25 **(a)** Use a 6-bit ring counter.

(b)

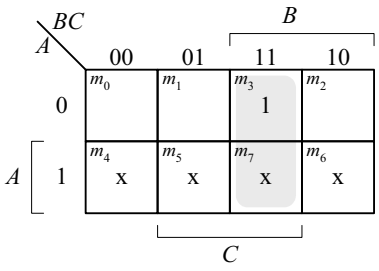


6.26 The clock generator has a period of 12.5 ns. Use a 2-bit counter to count four pulses.

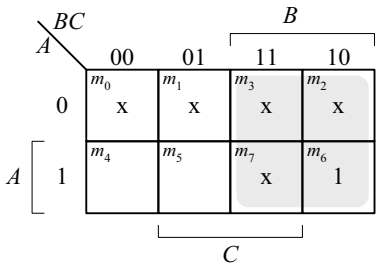
80/4 = 20 MHz; cycle time = 1000 x 10^{-9} /20 = 50 ns.

6.27

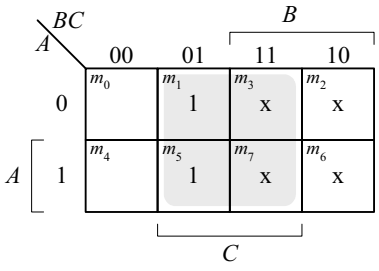
Present state *ABC*	*Next state* *ABC*	*Flip-flop inputs* J_A	K_A	J_B	K_B	J_C	K_C
000	001	0	x	0	x	1	x
001	010	0	x	1	x	x	1
010	011	0	x	x	0	1	x
011	100	1	x	x	1	x	1
100	100	x	x	0	0	1	x
101	110	x	x	1	x	x	1
110	000	x	x	x	1	0	x
111	xxx	x	x	x	x	x	x



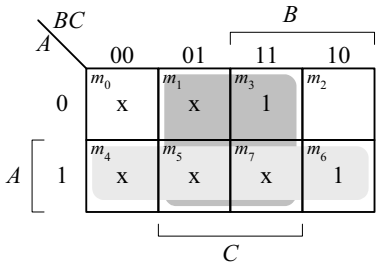
$J_A = BC$



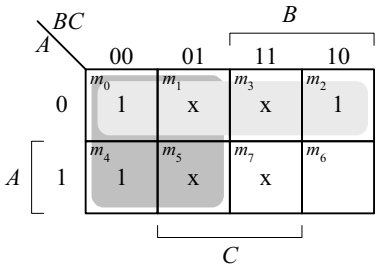
$K_A = B$



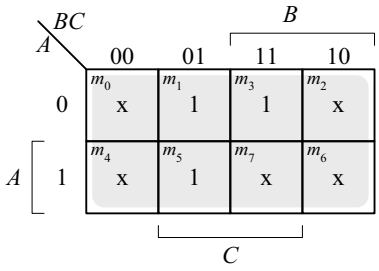
$J_B = C$



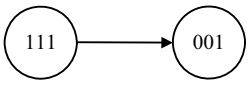
$K_B = A + C$



$J_C = A' + B'$



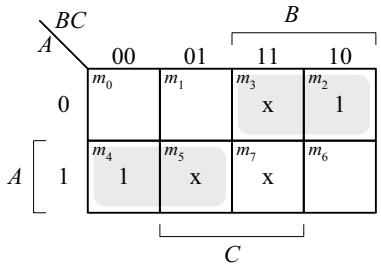
$K_C = 1$



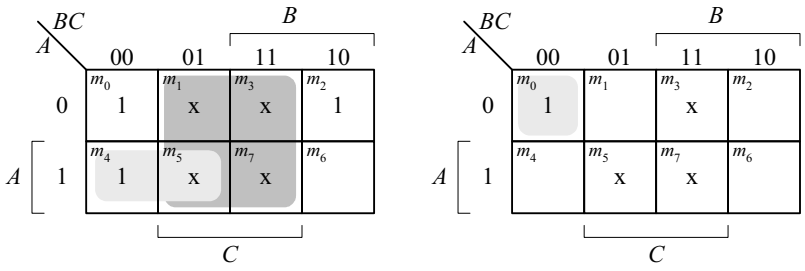
Self-correcting

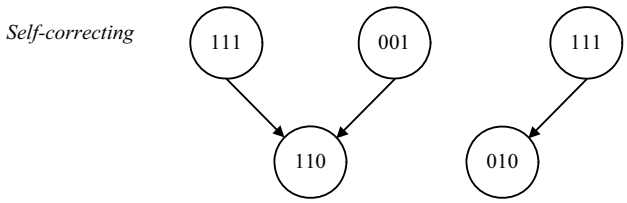
6.28

Present state ABC	*Next state ABC*
000	001
001	010
010	100
011	xxx
100	110
101	xxx
110	000
111	xxx



$D_A = A \oplus B$



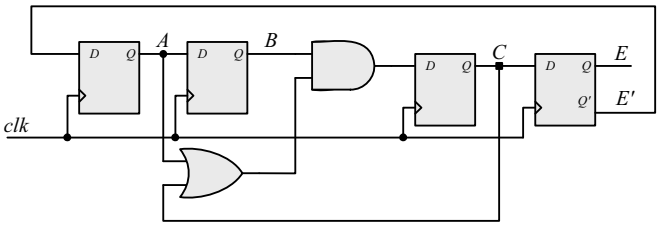


6.29 **(a)** The 8 valid states are listed in Fig. 8.18(b), with the sequence: 0, 8, 12, 14, 15, 7, 3, 1, 0, ...

The 8 unused states and their next states are shown below:

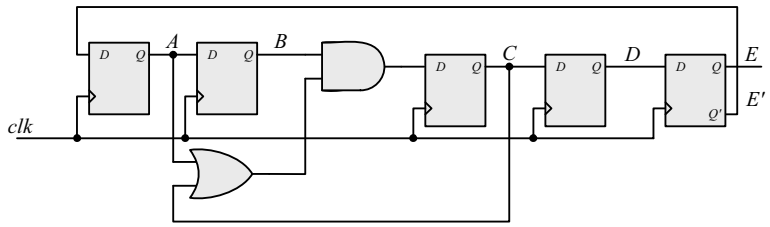
State *ABCE*	*Next state* *ABCE*	*All invalid states*
0000	1001	9
0100	1010	10
0101	0010	2
0110	1011	11
1001	0100	4
1010	1101	13
1011	0101	5
1101	0110	6

(b) Modification: $D_C = (A + C)B$.

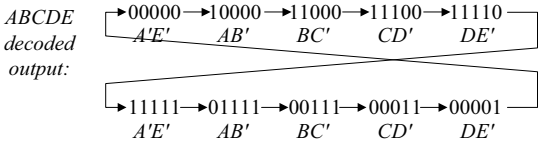


The valid states are the same as in (a). The unused states have the following sequences: 2→ 9→ 4→ 8 and 10→ 13→ 6→11→ 5→ 0. The final states, 0 and 8, are valid.

6.30

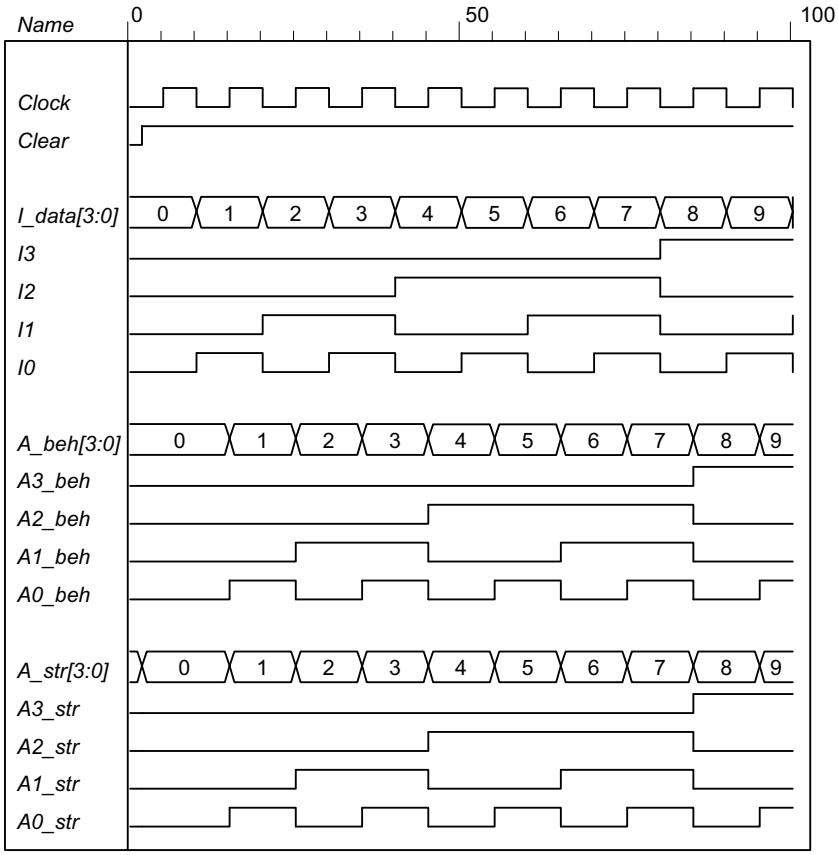


The 5-bit Johnson counter has the following state sequence:



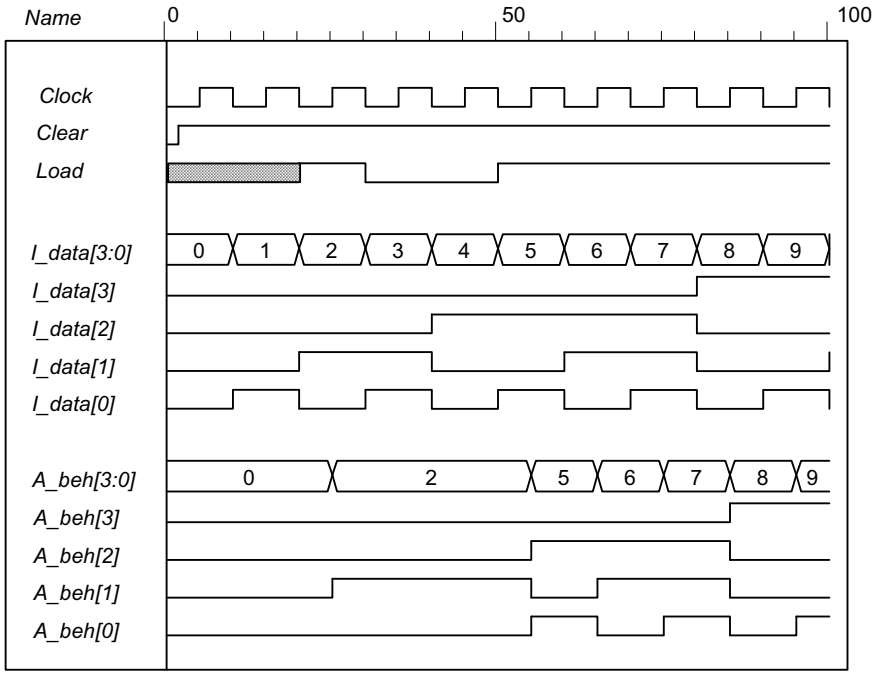
6.31

```
module Reg_4_bit_beh (output reg A3, A2, A1, A0, input I3, I2, I1, I0, Clock, Clear);
  always @ (posedge Clock, negedge Clear)
    if (Clear == 0) {A3, A2, A1, A0} <= 4'b0;
    else {A3, A2, A1, A0} <= {I3, I2, I1, I0};
endmodule

module Reg_4_bit_Str (output A3, A2, A1, A0, input I3, I2, I1, I0, Clock, Clear);
  DFF M3DFF (A3, I3, Clock, Clear);
  DFF M2DFF (A2, I2, Clock, Clear);
  DFF M1DFF (A1, I1, Clock, Clear);
  DFF M0DFF (A0, I0, Clock, Clear);
endmodule

module DFF(output reg Q, input D, clk, clear);
  always @ (posedge clk, posedge clear)
    if (clear == 0) Q <= 0; else Q <= D;
endmodule

module t_Reg_4_bit ();
  wire A3_beh, A2_beh, A1_beh, A0_beh;
  wire A3_str, A2_str, A1_str, A0_str;
  reg I3, I2, I1, I0, Clock, Clear;
  wire [3: 0] I_data = {I3, I2, I1, I0};
  wire [3: 0] A_beh = {A3_beh, A2_beh, A1_beh, A0_beh};
  wire [3: 0] A_str = {A3_str, A2_str, A1_str, A0_str};

  Reg_4_bit_beh M_beh (A3_beh, A2_beh, A1_beh, A0_beh, I3, I2, I1, I0, Clock, Clear);
  Reg_4_bit_Str M_str (A3_str, A2_str, A1_str, A0_str, I3, I2, I1, I0, Clock, Clear);

  initial #100 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial begin Clear = 0; #2 Clear = 1; end
  integer K;
  initial begin
    for (K = 0; K < 16; K = K + 1) begin {I3, I2, I1, I0} = K; #10 ; end
  end
endmodule
```



6.32 (a)

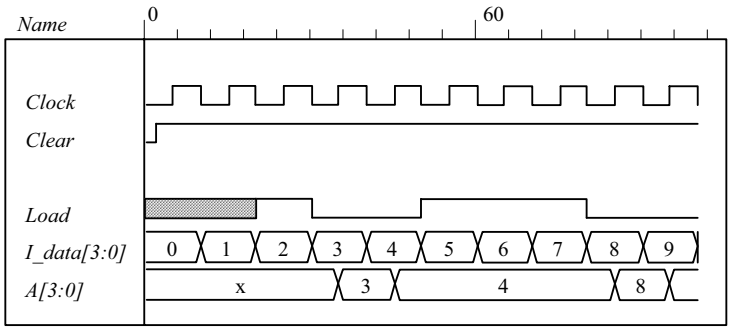
```
module Reg_4_bit_Load (output reg A3, A2, A1, A0, input I3, I2, I1, I0, Load, Clock, Clear);
  always @ (posedge Clock, negedge Clear)
    if (Clear == 0) {A3, A2, A1, A0} <= 4'b0;
    else if (Load) {A3, A2, A1, A0} <= {I3, I2, I1, I0};
endmodule

module t_Reg_4_Load ();
  wire A3_beh, A2_beh, A1_beh, A0_beh;
   reg I3, I2, I1, I0, Load, Clock, Clear;
  wire [3: 0] I_data = {I3, I2, I1, I0};
  wire [3: 0] A_beh = {A3_beh, A2_beh, A1_beh, A0_beh};

  Reg_4_bit_Load M0 (A3_beh, A2_beh, A1_beh, A0_beh, I3, I2, I1, I0, Load, Clock, Clear);

  initial #100 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial begin Clear = 0; #2 Clear = 1; end
  integer K;
  initial fork
    #20 Load = 1;
    #30 Load = 0;
    #50 Load = 1;
  join
  initial begin
    for (K = 0; K < 16; K = K + 1) begin {I3, I2, I1, I0} = K; #10 ; end
  end
endmodule
```



(b)

```
module Reg_4_bit_Load_str (output A3, A2, A1, A0, input I3, I2, I1, I0, Load, Clock, Clear);
  wire y3, y2, y1, y0;
  mux_2 M3 (y3, A3, I3, Load);
  mux_2 M2 (y2, A2, I2, Load);
  mux_2 M1 (y1, A1, I1, Load);
  mux_2 M0 (y0, A0, I0, Load);

  DFF M3DFF (A3, y3, Clock, Clear);
  DFF M2DFF (A2, y2, Clock, Clear);
  DFF M1DFF (A1, y1, Clock, Clear);
  DFF M0DFF (A0, y0, Clock, Clear);
endmodule

module DFF(output reg Q, input D, clk, clear);
  always @ (posedge clk, posedge clear)
    if (clear == 0) Q <= 0; else Q <= D;
endmodule

module mux_2 (output y, input a, b, sel);
  assign y = sel ? a: b;
endmodule

module t_Reg_4_Load_str ();
  wire A3, A2, A1, A0;
  reg I3, I2, I1, I0, Load, Clock, Clear;
  wire [3: 0] I_data = {I3, I2, I1, I0};
  wire [3: 0] A = {A3, A2, A1, A0};

  Reg_4_bit_Load_str M0 (A3, A2, A1, A0, I3, I2, I1, I0, Load, Clock, Clear);
```

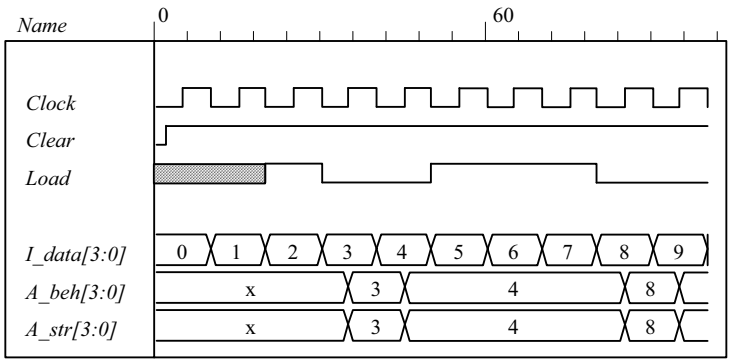
```
  initial #100 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial begin Clear = 0; #2 Clear = 1; end
  integer K;
  initial fork
    #20 Load = 1;
    #30 Load = 0;
    #50 Load = 1;
    #80 Load = 0;
  join
  initial begin
    for (K = 0; K < 16; K = K + 1) begin {I3, I2, I1, I0} = K; #10 ; end
  end
endmodule
```



(c)

```
module Reg_4_bit_Load_beh (output reg A3, A2, A1, A0, input I3, I2, I1, I0, Load, Clock, Clear);
  always @ (posedge Clock, negedge Clear)
    if (Clear == 0) {A3, A2, A1, A0} <= 4'b0;
    else if (Load) {A3, A2, A1, A0} <= {I3, I2, I1, I0};
endmodule

module Reg_4_bit_Load_str (output A3, A2, A1, A0, input I3, I2, I1, I0, Load, Clock, Clear);
  wire y3, y2, y1, y0;
  mux_2 M3 (y3, A3, I3, Load);
  mux_2 M2 (y2, A2, I2, Load);
  mux_2 M1 (y1, A1, I1, Load);
  mux_2 M0 (y0, A0, I0, Load);

  DFF M3DFF (A3, y3, Clock, Clear);
  DFF M2DFF (A2, y2, Clock, Clear);
  DFF M1DFF (A1, y1, Clock, Clear);
  DFF M0DFF (A0, y0, Clock, Clear);
endmodule

module DFF(output reg Q, input D, clk, clear);
  always @ (posedge clk, posedge clear)
    if (clear == 0) Q <= 0; else Q <= D;
endmodule

module mux_2 (output y, input a, b, sel);
  assign y = sel ? a: b;
endmodule
```

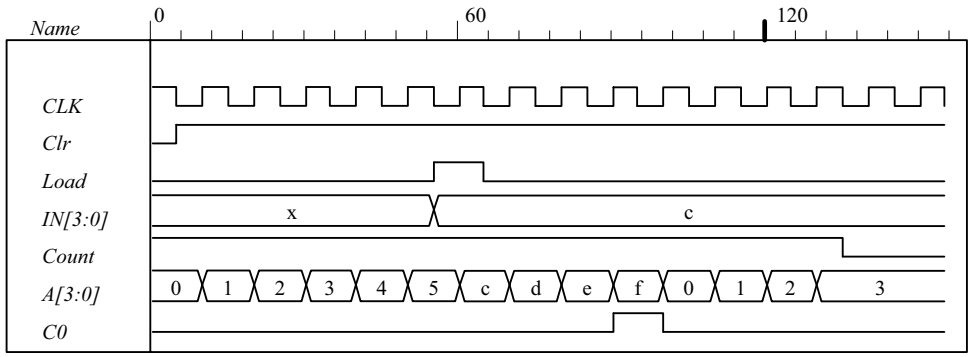
```
module t_Reg_4_Load_str ();
  wire A3_beh, A2_beh, A1_beh, A0_beh;
  wire A3_str, A2_str, A1_str, A0_str;
  reg I3, I2, I1, I0, Load, Clock, Clear;
  wire [3: 0] I_data, A_beh, A_str;
  assign  I_data = {I3, I2, I1, I0};
  assign  A_beh = {A3_beh, A2_beh, A1_beh, A0_beh};
  assign A_str = {A3_str, A2_str, A1_str, A0_str};

  Reg_4_bit_Load_str M0 (A3_beh, A2_beh, A1_beh, A0_beh, I3, I2, I1, I0, Load, Clock, Clear);
  Reg_4_bit_Load_str M1 (A3_str, A2_str, A1_str, A0_str, I3, I2, I1, I0, Load, Clock, Clear);

  initial #100 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial begin Clear = 0; #2 Clear = 1; end
  integer K;
  initial fork
    #20 Load = 1;
    #30 Load = 0;
    #50 Load = 1;
    #80 Load = 0;
  join
  initial begin
    for (K = 0; K < 16; K = K + 1) begin {I3, I2, I1, I0} = K; #10 ; end
  end
endmodule
```



6.33

```
// Stimulus for testing the binary counter of Example 6-3

module testcounter;
  reg Count, Load, CLK, Clr;
  reg [3: 0] IN;
  wire C0;
  wire [3: 0] A;
  Binary_Counter_4_Par_Load M0 (
  A,     // Data output
  C0,        // Output carry
  IN,        // Data input
  Count,         // Active high to count
  Load,          // Active high to load
  CLK,       // Positive edge sensitive
  Clr        // Active low
);
```

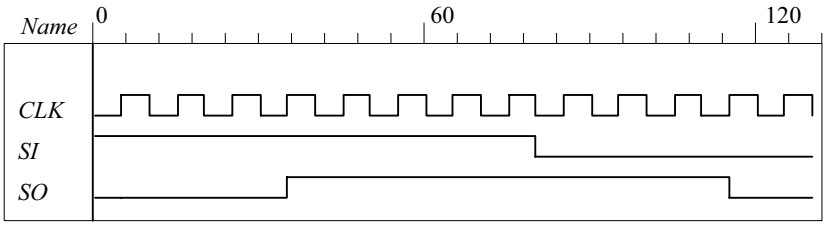
```
 always
   #5 CLK = ~CLK;
 initial
   begin
     Clr = 0;                  // Clear de-asserted
     CLK = 1;                  // Clock initialized high
     Load = 0; Count = 1;              // Enable count
     #5 Clr = 1;                       // Clears count, then counts for five cycles
     #50 Load = 1; IN = 4'b1100;       // Count is set to 4'b1100 (12.0)
     #10 Load = 0;
     #70 Count = 0;              // Count is deasserted at t = 135
     #20 $finish;                // Terminate simulation
   end
endmodule
// Four-bit binary counter with parallel load
// See Figure 6-14 and Table 6-6
module Binary_Counter_4_Par_Load (
  output reg [3:0] A_count,   // Data output
  output           C_out, // Output carry
  input [3:0]      Data_in,   // Data input
  input            Count,     // Active high to count
                   Load,  // Active high to load
                   CLK,   // Positive edge sensitive
                   Clear  // Active low
);


  assign C_out = Count & (~Load) & (A_count == 4'b1111);
  always @ (posedge CLK, negedge Clear)
  if (~Clear)        A_count <= 4'b0000;
  else if (Load)     A_count <= Data_in;
  else if (Count)    A_count <= A_count + 1'b1;
  else               A_count <= A_count; // redundant statement
endmodule

// Note: a preferred description if the clock is given by:
// initial begin CLK = 0; forever #5 CLK = ~CLK; end
```



6.34

```
module Shiftreg (SI, SO, CLK);
  input     SI, CLK;
  output    SO;
  reg    [3: 0]   Q;
  assign    SO = Q[0];
  always @ (posedge CLK)
    Q = {SI, Q[3: 1]};
endmodule
```

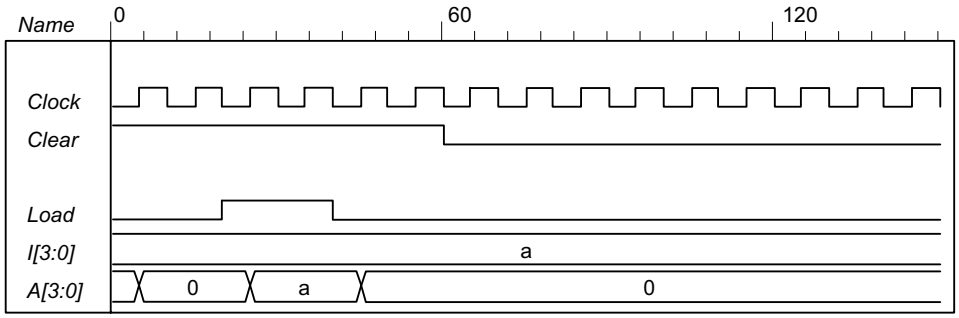
```
// Test plan
//
// Verify that data shift through the register
// Set SI =1 for 4 clock cycles
// Hold SI =1 for 4 clock cycles
// Set SI = 0 for 4 clock cycles
// Verify that data shifts out of the register correctly

module t_Shiftreg;
 reg   SI, CLK;
 wire  SO;

 Shiftreg M0 (SI, SO, CLK);

 initial #130 $finish;
 initial begin CLK = 0; forever #5 CLK = ~CLK; end
 initial fork
  SI = 1'b1;
  #80 SI = 0;
 join
endmodule
```



6.35 **(a)** Note that *Load* has priority over *Clear*.

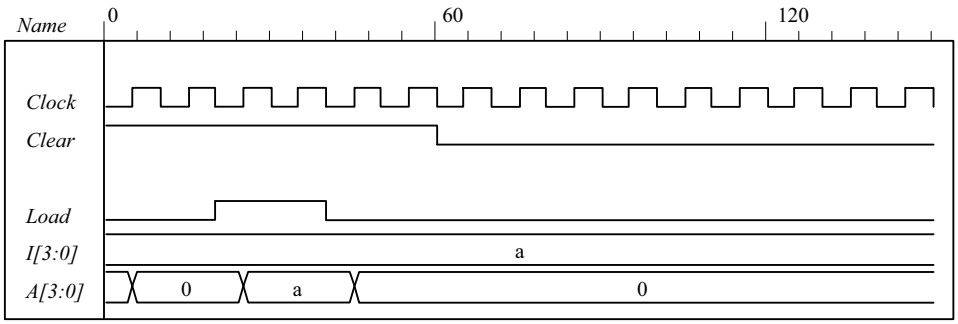
```
module Prob_6_35a (output [3: 0] A, input [3:0] I, input Load,  Clock, Clear);
  Register_Cell R0 (A[0], I[0], Load, Clock, Clear);
  Register_Cell R1 (A[1], I[1], Load, Clock, Clear);
  Register_Cell R2 (A[2], I[2], Load, Clock, Clear);
  Register_Cell R3 (A[3], I[3], Load, Clock, Clear);
endmodule

module Register_Cell (output A, input I, Load, Clock, Clear);
  DFF M0 (A, D, Clock);
  not (Load_b, Load);
  not (w1, Load_b);
  not (Clear_b, Clear);
  and (w2, I, w1);
  and (w3, A, Load_b, Clear_b);
  or (D, w2, w3);
endmodule

module DFF (output reg Q, input D, clk);
always @ (posedge clk) Q <= D;
endmodule

module t_Prob_6_35a ( );

  wire [3: 0] A;
  reg [3: 0] I;
  reg Clock, Clear, Load;
```

```
  Prob_6_35a M0 ( A, I, Load, Clock, Clear);
  initial #150 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial fork
  I = 4'b1010;Clear = 1;
  #40 Clear = 0;
  Load = 0;
  #20 Load = 1;
  #40 Load = 0;
  join
endmodule
```



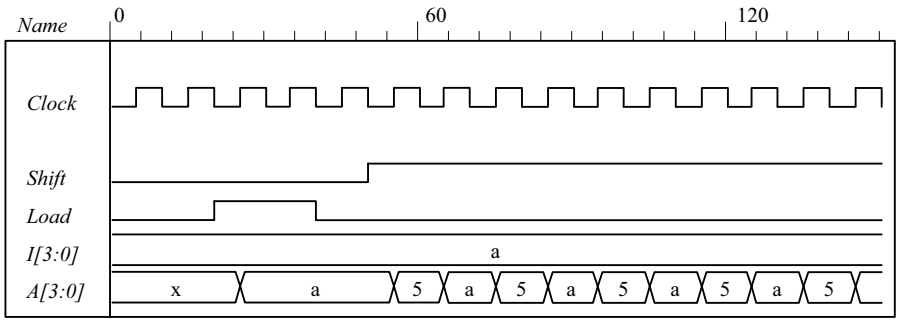
(b) Note: The solution below replaces the solution given on the CD.

```
module Prob_6_35b (output reg [3: 0] A, input [3:0] I, input Load, Clock, Clear);
  always @ (posedge Clock)
   if (Load) A <= I;
  else if (Clear) A <= 4'b0;
  //else A <= A;              // redundant statement
endmodule

module t_Prob_6_35b ( );

  wire [3: 0] A;
  reg [3: 0] I;
  reg Clock, Clear, Load;

  Prob_6_35b M0 ( A, I, Load, Clock, Clear);
  initial #150 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial fork
  I = 4'b1010; Clear = 1;
  #60 Clear = 0;
  Load = 0;
  #20 Load = 1;
  #40 Load = 0;
   join
endmodule
```



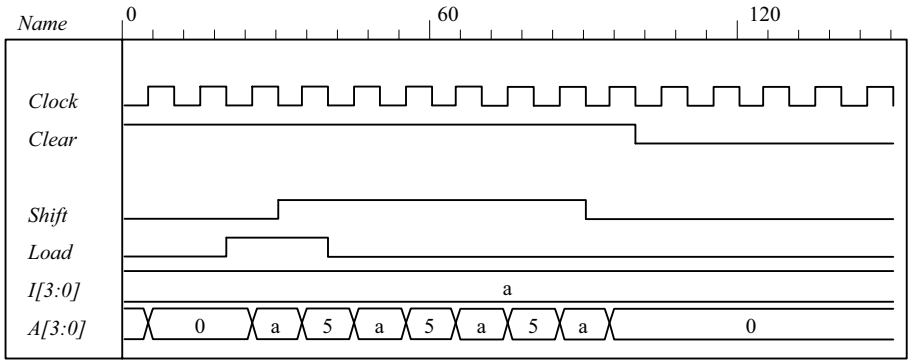
(c)

```
module Prob_6_35c (output [3: 0] A, input [3:0] I, input Shift, Load,  Clock);
  Register_Cell R0 (A[0], I[0], A[1], Shift, Load, Clock);
  Register_Cell R1 (A[1], I[1], A[2], Shift, Load, Clock);
  Register_Cell R2 (A[2], I[2], A[3], Shift, Load, Clock);
  Register_Cell R3 (A[3], I[3], A[0], Shift, Load, Clock);
endmodule

module Register_Cell (output A, input I, Serial_in, Shift, Load, Clock);
  DFF M0 (A, D, Clock);
  not (Shift_b, Shift);
  not (Load_b, Load);
  and (w1, Shift, Serial_in);
  and (w2, Shift_b, Load, I);

  and (w3, A, Shift_b, Load_b);
  or (D, w1, w2, w3);
endmodule

module DFF (output reg Q, input D, clk);
always @ (posedge clk) Q <= D;
endmodule

module t_Prob_6_35c ( );

  wire [3: 0] A;
  reg [3: 0] I;
  reg Clock, Shift, Load;

  Prob_6_35c M0 (A, I, Shift, Load, Clock);
  initial #150 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial fork
  I = 4'b1010;
  Load = 0; Shift = 0;
  #20 Load = 1;
  #40 Load = 0;
  #50 Shift = 1;
   join
endmodule
```



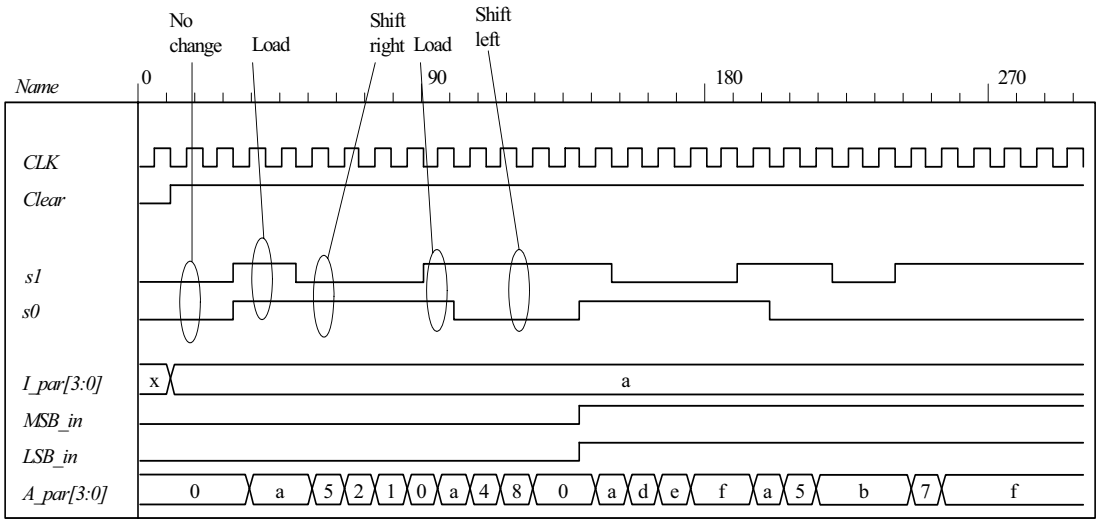
(d)

```
module Prob_6_35d (output reg [3: 0] A, input [3:0] I, input Shift, Load, Clock, Clear);
  always @ (posedge Clock)
    if (Shift) A <= {A[0], A[3:1]};
    else if (Load) A <= I;
    else if (Clear) A <= 4'b0;
    //else A <= A;              // redundant statement
endmodule

module t_Prob_6_35d ( );

  wire [3: 0] A;
  reg [3: 0] I;
  reg Clock, Clear, Shift, Load;

    Prob_6_35d M0 ( A, I, Shift, Load, Clock, Clear);
  initial #150 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial fork
    I = 4'b1010; Clear = 1;
    #100 Clear = 0;
    Load = 0;
    #20 Load = 1;
    #40 Load = 0;
    #30 Shift = 1;
    #90 Shift = 0;
  join
endmodule
```



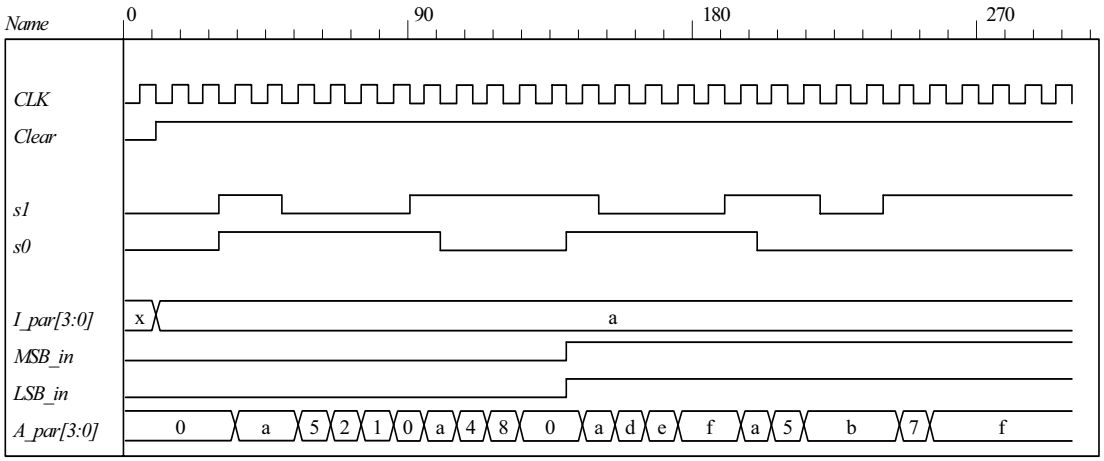
(e)

```
module Shift_Register
 (output [3: 0] A_par, input [3: 0] I_par, input MSB_in, LSB_in, s1, s0, CLK, Clear);
 wire y3, y2, y1, y0;
 DFF D3 (A_par[3],  y3, CLK, Clear);
 DFF D2 (A_par[2],  y2, CLK, Clear);
 DFF D1 (A_par[1],  y1, CLK, Clear);
 DFF D0 (A_par[0],  y0, CLK, Clear);

 MUX_4x1 M3 (y3, I_par[3], A_par[2], MSB_in, A_par[3], s1, s0);
 MUX_4x1 M2 (y2, I_par[2], A_par[1], A_par[3], A_par[2], s1, s0);
 MUX_4x1 M1 (y1, I_par[1], A_par[0], A_par[2], A_par[1], s1, s0);
 MUX_4x1 M0 (y0, I_par[0], LSB_in, A_par[1], A_par[0], s1, s0);
endmodule

module MUX_4x1 (output reg y, input I3, I2, I1, I0, s1, s0);
 always @ (I3, I2, I1, I0, s1, s0)
  case ({s1, s0})
   2'b11:  y = I3;
   2'b10:  y = I2;
   2'b01:  y = I1;
   2'b00:  y = I0;
  endcase
endmodule

module DFF (output reg Q,  input D, clk, reset_b);
 always @ (posedge clk, negedge reset_b) if (reset_b == 0) Q <= 0; else Q <= D;
endmodule

module t_Shift_Register ( );
 wire  [3: 0] A_par;
 reg   [3: 0] I_par;
 reg MSB_in, LSB_in, s1, s0, CLK, Clear;

 Shift_Register M_SR( A_par, I_par, MSB_in, LSB_in, s1, s0, CLK, Clear);
 initial #300 $finish;
 initial begin CLK = 0; forever #5 CLK = ~CLK; end
  initial fork
   MSB_in = 0; LSB_in = 0;
   Clear = 0;              // Active-low reset
   s1  = 0; s0 = 0;            // No change
   #10 Clear = 1;
   #10 I_par = 4'hA;
   #30 begin s1 = 1; s0 = 1; end // 00: load I_par into A_par
   #50 s1 = 0;            // 01: shift right (1010  to 0101 to 0010 to 0001 to 0000)
   #90 begin s1 = 1; s0 = 1; end // 11: reload A with 1010
   #100 s0 = 0;               // 10: shift left (1010 to 0100 to 1000 to 000)
   #140 begin s1 = 1; s0 = 1; MSB_in = 1;  LSB_in = 1; end    // Repeat with MSB and LSB
   #150 s1 = 0;
   #190 begin s1 = 1; s0 = 1;  end     // reload with A = 1010
   #200 s0 = 0;               // Shift left
   #220 s1 = 0;               // Pause
     #240  s1 = 1;            // Shift left
  join
endmodule
```



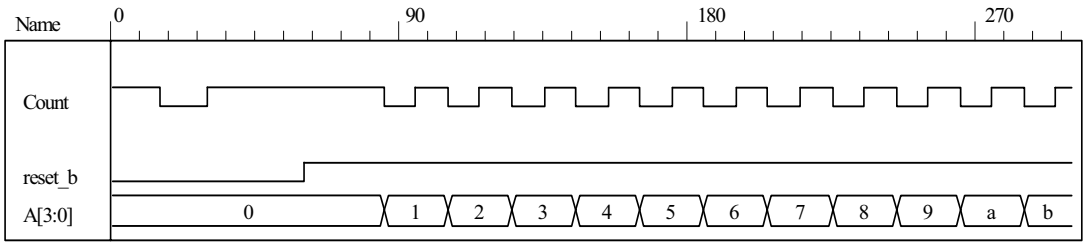
(f)

```
module Shift_Register_BEH
 (output [3: 0] A_par, input [3: 0] I_par, input MSB_in, LSB_in, s1, s0, CLK, Clear);

 always @ (posedge CLK, negedge Clear) if (Clear == 0) A_par <= 4'b0;
  else case ({s1, s0})
    2'b11:    A_par <= I_par;
    2'b01:    A_par <= {MSB_in, A_par[3: 1]};
    2'b10:    A_par <= {A_par[2: 0], LSB_in};
    2'b00:    A_par <=A_par;
  endcase
endmodule

module t_Shift_Register ( );
 wire [3: 0] A_par;
 reg  [3: 0] I_par;
 reg MSB_in, LSB_in, s1, s0, CLK, Clear;

 Shift_Register_BEH M_SR( A_par, I_par, MSB_in, LSB_in, s1, s0, CLK, Clear);
 initial #300 $finish;
 initial begin CLK = 0; forever #5 CLK = ~CLK; end
  initial fork
   MSB_in = 0; LSB_in = 0;
   Clear = 0;              // Active-low reset
   s1  = 0; s0 = 0;            // No change
   #10 Clear = 1;
   #10 I_par = 4'hA;
   #30 begin s1 = 1; s0 = 1; end // 00: load I_par into A_par
   #50 s1 = 0;                       // 01: shift right (1010  to 0101 to 0010 to 0001 to 0000)
   #90 begin s1 = 1; s0 = 1; end // 11: reload A with 1010
   #100 s0 = 0;                      // 10: shift left (1010 to 0100 to 1000 to 000)
   #140 begin s1 = 1; s0 = 1; MSB_in = 1;  LSB_in = 1; end    // Repeat with MSB and LSB
   #150 s1 = 0;
   #190 begin s1 = 1; s0 = 1;  end     // reload with A = 1010
   #200 s0 = 0;                // Shift left
   #220 s1 = 0;                // Pause
     #240  s1 = 1;             // Shift left
  join
endmodule
```



(g)

```
module Ripple_Counter_4bit (output [3: 0] A,  input Count, reset_b);
  reg A0, A1, A2, A3;
  assign A = {A3, A2, A1, A0};
  always @ (negedge Count, negedge reset_b)
    if (reset_b == 0) A0 <= 0; else A0 <= ~A0;

  always @ (negedge A0, negedge reset_b)
    if (reset_b == 0) A1 <= 0; else A1 <= ~A1;

  always @ (negedge A1, negedge reset_b)
    if (reset_b == 0) A2 <= 0; else A2 <= ~A2;

  always @ (negedge A2, negedge reset_b)
    if (reset_b == 0) A3 <= 0; else A3 <= ~A3;
endmodule

module t_Ripple_Counter_4bit ();
  wire [3: 0] A;
  reg Count, reset_b;

  Ripple_Counter_4bit M0 (A,  Count, reset_b);

  initial #300 $finish;
   initial fork
       reset_b  = 0;          // Active-low reset
    #60 reset_b = 1;

     Count = 1;
    #15 Count = 0;
    #30 Count = 1;
    #85 begin  Count = 0;  forever #10 Count = ~Count; end
  join
endmodule
```

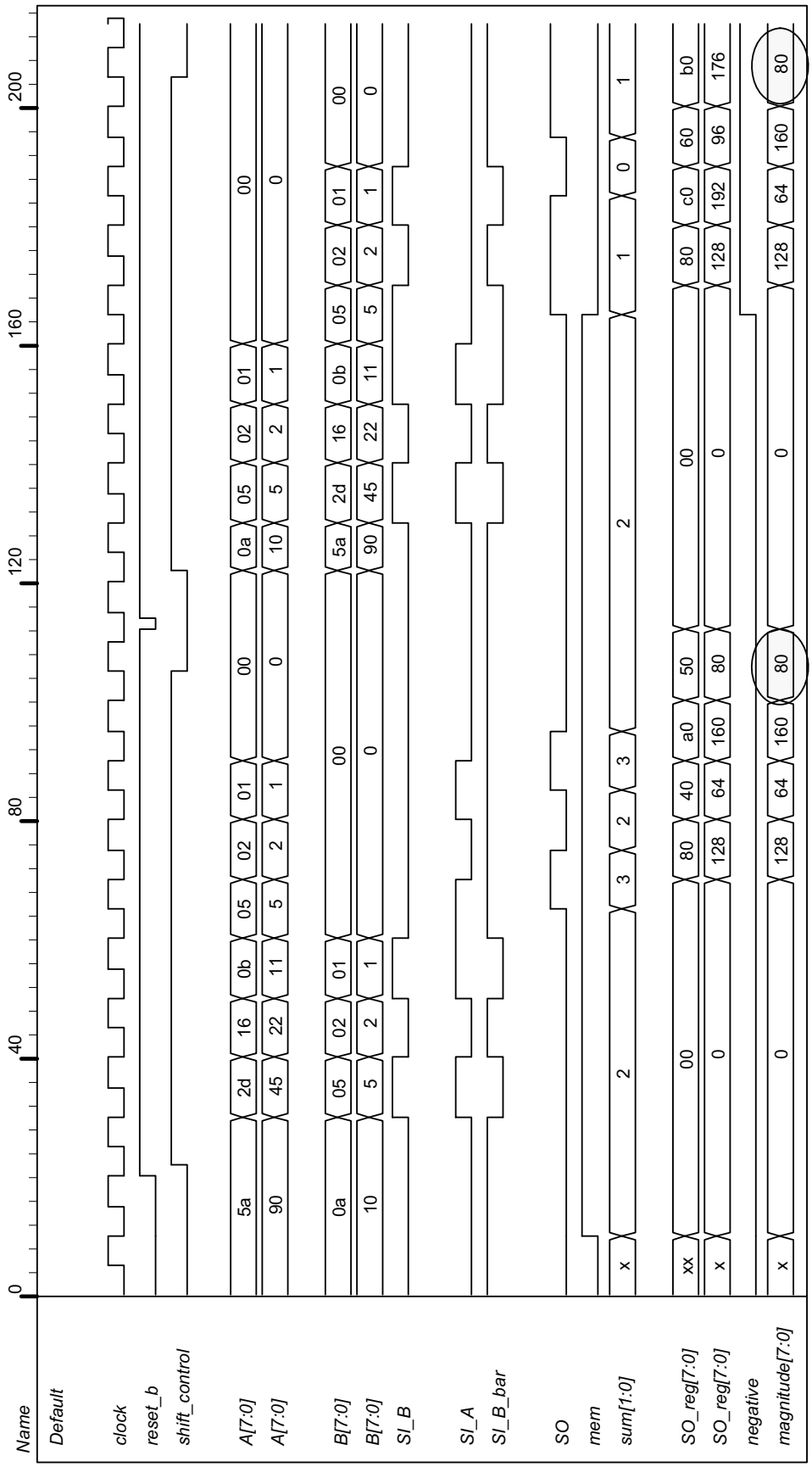


(h) Note: This version of the solution situates the data shift registers in the test bench.

```
module Serial_Subtractor (output SO, input SI_A, SI_B, shift_control, clock, reset_b);
// See Fig. 6.5 and Problem 6.9a (2s complement serial subtractor)
  reg [1: 0] sum;
  wire mem = sum[1];
  assign SO = sum[0];

   always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) begin
      sum <= 2'b10;
    end
    else if (shift_control) begin
      sum <= SI_A + (!SI_B) + sum[1];
    end
endmodule

module t_Serial_Subtractor ();
  wire SI_A, SI_B;
  reg shift_control, clock, reset_b;

  Serial_Subtractor M0 (SO, SI_A, SI_B, shift_control, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
    shift_control = 0;
    #10 reset_b = 0;
    #20 reset_b = 1;
    #22 shift_control = 1;
    #105 shift_control = 0;
    #112 reset_b = 0;
    #114 reset_b = 1;
    #122 shift_control = 1;
    #205 shift_control = 0;
  join
  reg [7: 0] A, B, SO_reg;
  wire s7;
  assign s7 = SO_reg[7];
  assign SI_A = A[0];
  assign SI_B = B[0];
  wire SI_B_bar = ~SI_B;
  initial fork
    A = 8'h5A;
    B = 8'h0A;
    #122 A = 8'h0A;
    #122 B = 8'h5A;
  join
```

```
  always @ (negedge clock, negedge reset_b)
    if (reset_b == 0) SO_reg <= 0;
    else if (shift_control == 1) begin
      SO_reg <= {SO, SO_reg[7: 1]};
      A <= A >> 1;
      B <= B >> 1;
    end
  wire negative = !M0.sum[1];
  wire [7: 0] magnitude = (!negative)? SO_reg: 1'b1 + ~SO_reg;
endmodule
```

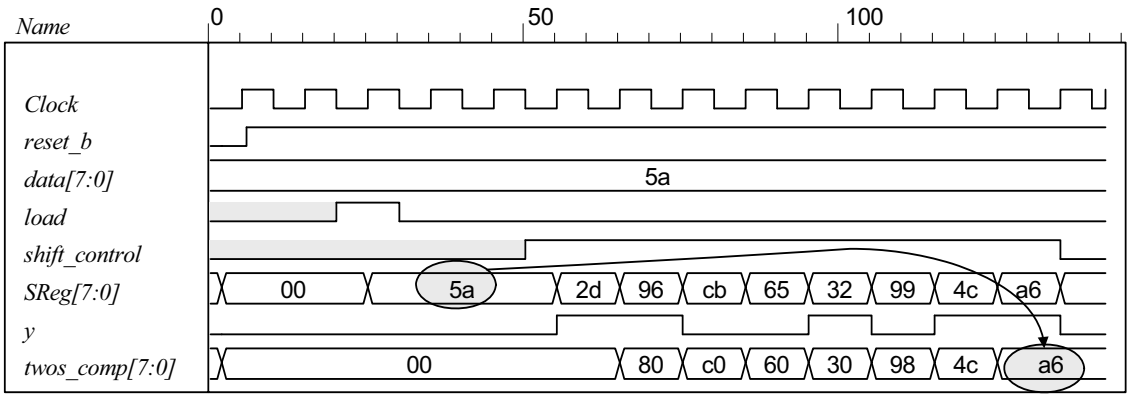
Simulation results are shown for 5Ah – 0Ah = 50h = 80 d and 0Ah – 5Ah = -80. The magnitude of the result is also shown.



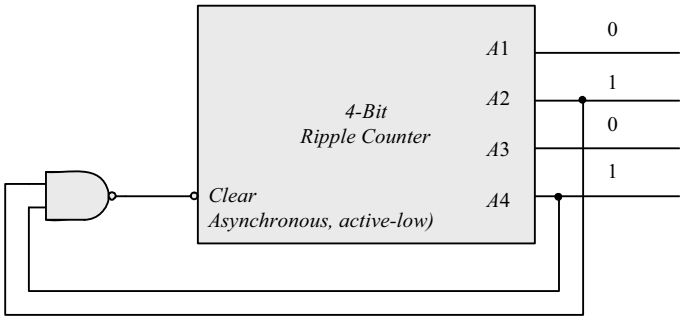
(i) See Prob. 6.35h.

(j)

```
module Serial_Twos_Comp (output y, input [7: 0] data, input load, shift_control, Clock, reset_b);
 reg [7: 0] SReg;
 reg Q;
 wire SO = SReg [0];
 assign y = SO ^ Q;
 always @ (posedge Clock, negedge reset_b)
 if (reset_b == 0) begin
  SReg <= 0;
  Q <= 0;
 end
 else begin
  if (load) SReg = data;
  else if (shift_control) begin
   Q <= Q | SO;
   SReg <= {y, SReg[7: 1]};
  end
 end
endmodule

module t_Serial_Twos_Comp ();
 wire y;
 reg [7: 0] data;
 reg load, shift_control, Clock, reset_b;

 Serial_Twos_Comp M0 (y, data, load, shift_control, Clock, reset_b);

 reg [7: 0] twos_comp;

 always @ (posedge Clock, negedge reset_b)
 if (reset_b == 0) twos_comp <= 0;
 else if (shift_control && !load) twos_comp <= {y, twos_comp[7: 1]};

 initial #200 $finish;
 initial begin Clock = 0; forever #5 Clock = ~Clock; end
 initial begin #2 reset_b = 0; #4 reset_b = 1; end

 initial fork
  data = 8'h5A;
  #20 load = 1;
  #30 load = 0;
  #50 shift_control = 1;
  #50 begin repeat (9) @ (posedge Clock) ;
   shift_control = 0;
  end
 join
endmodule
```



(k) From the solution to Problem 6.13:

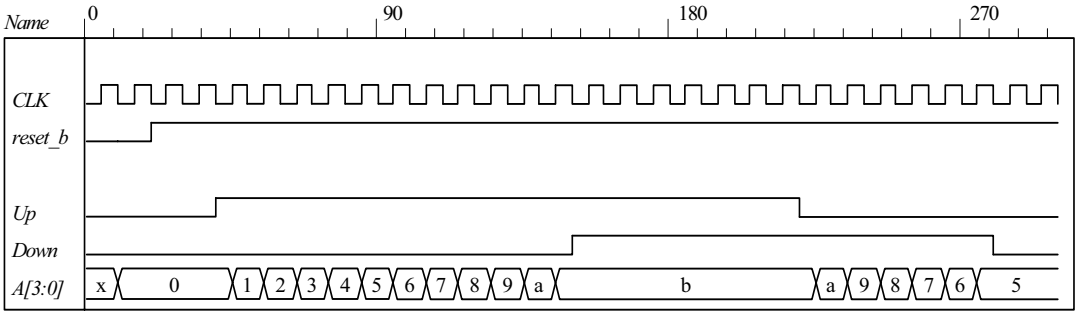



```
module Prob_6_35k_BCD_Counter (output A1, A2, A3, A4, input clk, reset_b);
  wire {A1, A2, A3, A4} = A;
  nand (Clear, A2, A4);
   Ripple_Counter_4bit M0 (A, Clear, reset_b);
endmodule

module Ripple_Counter_4bit (output [3: 0] A,  input Count, reset_b);
  reg A0, A1, A2, A3;
  assign A = {A3, A2, A1, A0};
  always @ (negedge Count, negedge reset_b)
    if (reset_b == 0) A0 <= 0; else A0 <= ~A0;
   always @ (negedge A0, negedge reset_b)
    if (reset_b == 0) A1 <= 0; else A1 <= ~A1;
   always @ (negedge A1, negedge reset_b)
    if (reset_b == 0) A2 <= 0; else A2 <= ~A2;
  always @ (negedge A2, negedge reset_b)
    if (reset_b == 0) A3 <= 0; else A3 <= ~A3;
endmodule
module t_ Prob_6_35k_BCD_Counter ();
  wire [3: 0] A;
  reg Count, reset_b;

  Prob_6_35k_BCD_Counter M0 (A1, A2, A3, A4, reset_b);
  initial #300 $finish;
   initial fork
       reset_b  = 0;          // Active-low reset
    #60 reset_b = 1;
/*
    Count = 1;
    #15 Count = 0;
    #30 Count = 1;
```

```
   #85 begin  Count = 0;  forever #10 Count = ~Count; end*/
  join
endmodule
```

(l)

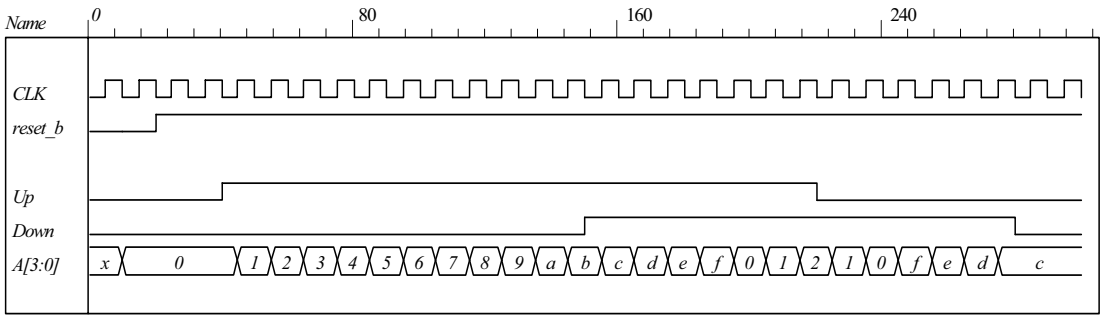
```
module Prob_6_35l_Up_Dwn_Beh (output reg [3: 0] A, input CLK, Up, Down, reset_b);

 always @ (posedge CLK, negedge reset_b)
  if (reset_b ==0) A <= 4'b0000;
  else case ({Up, Down})
    2'b10:  A <= A + 4'b0001;    // Up
    2'b01:  A <= A - 4'b0001;    // Down
    default:   A <= A;    // Suspend (Redundant statement)
  endcase
endmodule

module t_Prob_6_35l_Up_Dwn_Beh ();
 wire [3: 0] A;
 reg CLK, Up, Down, reset_b;

 Prob_6_35l_Up_Dwn_Beh M0 (A, CLK, Up, Down, reset_b);

 initial #300 $finish;
 initial begin CLK = 0; forever #5 CLK = ~CLK; end
 initial fork
     Down = 0; Up= 0;
  #10 reset_b = 0;
  #20 reset_b = 1;
  #40 Up = 1;
  #150 Down = 1;
  #220 Up = 0;
  #280 Down = 0;
 join
endmodule
```



6.36 (a)

```
// See Fig. 6.13., 4-bit Up-Down Binary Counter
module Prob_6_36_Up_Dwn_Beh (output reg [3: 0] A, input CLK, Up, Down, reset_b);

 always @ (posedge CLK, negedge reset_b)
  if (reset_b ==0) A <= 4'b0000;
  else if (Up) A <= A + 4'b0001;
  else if (Down) A <= A - 4'b0001;
endmodule

module t_Prob_6_36_Up_Dwn_Beh ();
 wire [3: 0] A;
 reg CLK, Up, Down, reset_b;
```

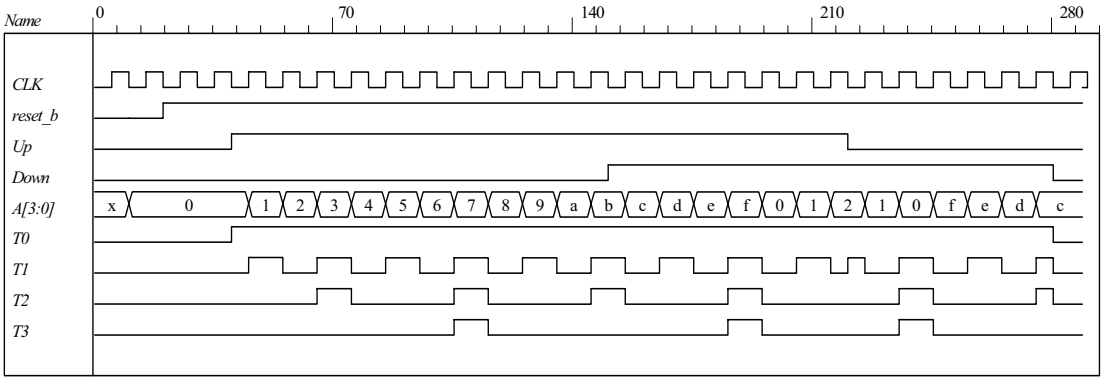
```
  Prob_6_36_Up_Dwn_Beh M0 (A, CLK, Up, Down, reset_b);

  initial #300 $finish;
  initial begin CLK = 0; forever #5 CLK = ~CLK; end
  initial fork
      Down = 0; Up= 0;
   #10 reset_b = 0;
   #20 reset_b = 1;
   #40 Up = 1;
   #150 Down = 1;
   #220 Up = 0;
   #280 Down = 0;
  join
endmodule
```



(b)

```
module Prob_6_36_Up_Dwn_Str (output [3: 0] A, input CLK, Up, Down, reset_b);
  wire Down_3, Up_3, Down_2, Up_2, Down_1, Up_1;
  wire A_0b, A_1b, A_2b, A_3b;


  stage_register SR3 (A[3], A_3b, Down_3, Up_3, Down_2, Up_2, A[2], A_2b, CLK, reset_b);
  stage_register SR2 (A[2], A_2b, Down_2, Up_2, Down_1, Up_1, A[1], A_1b, CLK, reset_b);
  stage_register SR1 (A[1], A_1b, Down_1, Up_1, Down_not_Up, Up, A[0], A_0b, CLK, reset_b);
  not (Up_b, Up);
  and (Down_not_Up, Down, Up_b);
  or (T, Up, Down_not_Up);
  Toggle_flop TF0 (A[0], A_0b, T, CLK, reset_b);
endmodule

module stage_register (output A, A_b, Down_not_Up_out, Up_out, input Down_not_Up, Up, A_in,
A_in_b, CLK, reset_b);

  Toggle_flop T0 (A, A_b, T, CLK, reset_b);
  or (T, Down_not_Up_out, Up_out);
  and (Down_not_Up_out, Down_not_Up, A_in_b);
  and (Up_out, Up, A_in);
endmodule

module Toggle_flop (output reg Q, output Q_b, input T, CLK, reset_b);
  always @ (posedge CLK, negedge reset_b) if (reset_b == 0) Q <= 0; else Q <= Q ^ T;
  assign Q_b = ~Q;

endmodule

module t_Prob_6_36_Up_Dwn_Str ();
  wire [3: 0] A;
  reg CLK, Up, Down, reset_b;
```

```
  wire T3 = M0.SR3.T;
  wire T2 = M0.SR2.T;
  wire T1 = M0.SR1.T;
  wire T0 = M0.T;

  Prob_6_36_Up_Dwn_Str M0 (A, CLK, Up, Down, reset_b);

  initial #150 $finish;
  initial begin CLK = 0; forever #5 CLK = ~CLK; end
  initial fork
    Down = 0; Up= 0;
    #10 reset_b = 0;
    #20 reset_b = 1;
    #50 Up = 1;
    #140 Down = 1;
    #120 Up = 0;
    #140 Down = 0;
  join
endmodule
```



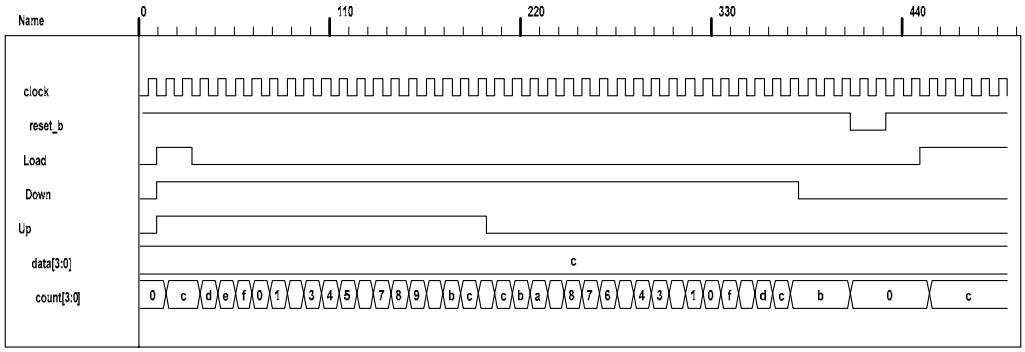
6.37

```
module Counter_if (output reg [3: 0] Count, input clock, reset);
  always @ (posedge clock , posedge reset)
    if (reset)Count <= 0;
    else if (Count == 0) Count <= 1;
    else if (Count == 1) Count <= 3;     // Default interpretation is decimal
    else if (Count == 3) Count <= 7;
    else if (Count == 4) Count <= 0;
    else if (Count == 6) Count <= 4;
    else if (Count == 7) Count <= 6;
    else Count <= 0;
endmodule

module Counter_case (output reg [3: 0] Count, input clock, reset);
  always @ (posedge clock , posedge reset)
    if (reset)Count <= 0;
    else begin
      Count <= 0;
      case (Count)
        0:       Count <= 1;
        1:       Count <= 3;
        3:       Count <= 7;
        4:       Count <= 0;
        6:       Count <= 4;
        7:       Count <= 6;
        default:    Count <= 0;
      endcase
```

```
  end
endmodule

module Counter_FSM (output reg [3: 0] Count, input clock, reset);
  reg [2: 0] state, next_state;
  parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4, s5 = 5, s6 = 6, s7 = 7;

  always @ (posedge clock ,  posedge reset)
    if (reset) state <= s0; else state <= next_state;

  always @ (state) begin
    Count = 0;
    case (state)
       s0:      begin next_state = s1; Count = 0; end
       s1:      begin next_state = s2; Count = 1; end
       s2:      begin next_state = s3; Count = 3; end
       s3:      begin next_state = s4; Count = 7; end
       s4:      begin next_state = s5; Count = 6; end
       s5:      begin next_state = s6; Count = 4; end
       default:    begin next_state = s0; Count = 0; end
     endcase
  end
endmodule
```

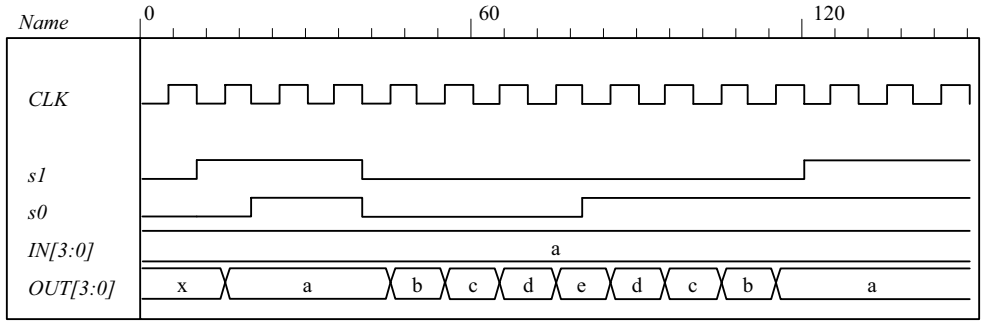
6.38 (a)

```
module Prob_6_38a_Updown (OUT, Up, Down, Load, IN, CLK);    // Verilog 1995
  output [3: 0]   OUT;
  input  [3: 0]   IN;
  input           Up, Down, Load, CLK;
  reg    [3:0]    OUT;

  always @ (posedge CLK)
  if (Load) OUT <= IN;
  else if (Up)     OUT <= OUT + 4'b0001;
  else if (Down) OUT <= OUT -  4'b0001;
  else         OUT <= OUT;
  endmodule

module updown (                    // Verilog 2001, 2005
  output reg[3: 0]   OUT,
  input      [3: 0]   IN,
  input           Up, Down, Load, CLK
);
```



(b)

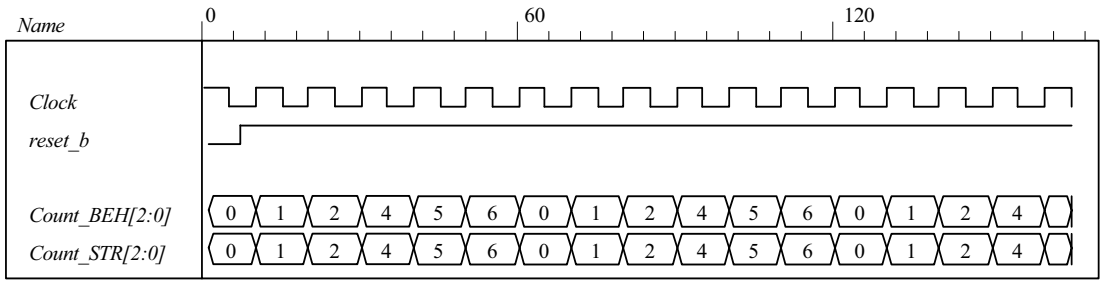
```
module Prob_6_38b_Updown (output reg [3: 0] OUT, input [3: 0] IN, input s1, s0, CLK);

 always @ (posedge CLK)
 case ({s1, s0})
   2'b00:  OUT <= OUT + 4'b0001;
   2'b01:  OUT <= OUT - 4'b0001;
   2'b10:  OUT <= IN;
   2'b11:  OUT <= OUT;
 endcase
 endmodule

 module t_Prob_6_38b_Updown ();
  wire [3: 0] OUT;
  reg [3: 0] IN;
  reg s1, s0, CLK;
 Prob_6_38b_Updown M0 (OUT, IN, s1, s0, CLK);

 initial #150 $finish;
 initial begin CLK = 0; forever #5 CLK = ~CLK; end
  initial fork
 IN = 4'b1010;

 #10  begin s1 = 1; s0 = 0; end     // Load IN
 #20  begin s1 = 1; s0 = 1; end     // no change
 #40  begin s1 = 0; s0 = 0; end     // UP;
 #80  begin s1 = 0; s0 = 1; end     // DOWN
 #120 begin s1 = 1; s0 = 1; end
 join
endmodule
```



6.39

```
module Prob_6_39_Counter_BEH (output reg [2: 0] Count, input Clock, reset_b);
 always @ (posedge Clock, negedge reset_b)  if (reset_b == 0) Count <= 0;
  else case (Count)
    0:  Count <= 1;
    1:  Count <= 2;
    2:  Count <= 4;
    4:  Count <= 5;
    5:  Count <= 6;
    6:  Count <= 0;
  endcase
endmodule

module Prob_6_39_Counter_STR (output [2: 0] Count, input Clock, reset_b);
 supply1 PWR;
 wire Count_1_b = ~Count[1];
```

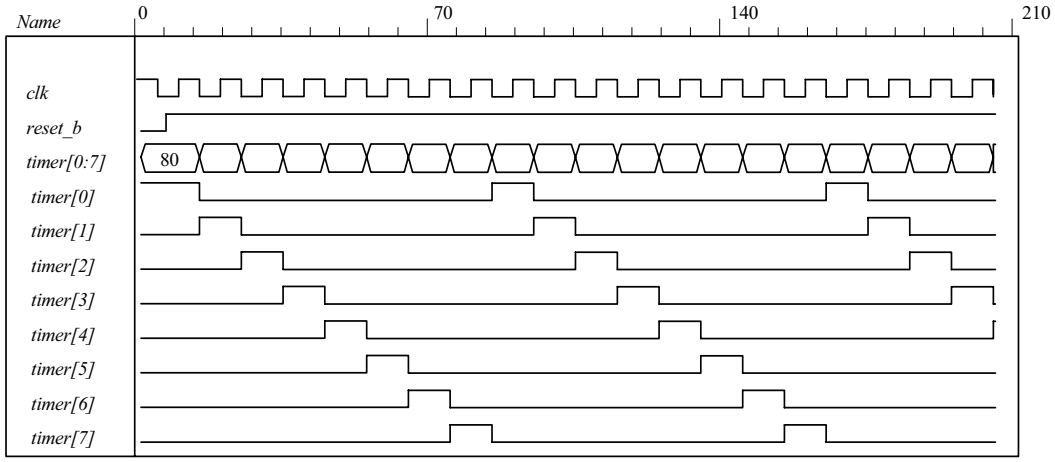
```
  JK_FF M2 (Count[2],    Count[1],   Count[1],   Clock, reset_b);
  JK_FF M1 (Count[1],    Count[0],   PWR,        Clock, reset_b);
  JK_FF M0 (Count[0],    Count_1_b,          PWR,        Clock, reset_b);
endmodule

module JK_FF (output reg Q, input J, K, clk, reset_b);
  always @ (posedge clk, negedge reset_b) if (reset_b == 0) Q <= 0; else
    case ({J,K})
      2'b00:  Q <= Q;
      2'b01:  Q <= 0;
      2'b10:  Q <= 1;
      2'b11:  Q <= ~Q;
    endcase
endmodule

module t_Prob_6_39_Counter ();
  wire [2: 0] Count_BEH, Count_STR;
  reg Clock, reset_b;

  Prob_6_39_Counter_BEH M0_BEH (Count_STR, Clock, reset_b);
  Prob_6_39_Counter_STR M0_STR (Count_BEH, Clock, reset_b);

  initial #250 $finish;
  initial fork #1 reset_b = 0; #7 reset_b = 1; join
  initial begin  Clock = 1; forever #5 Clock = ~Clock; end
endmodule
```



6.40

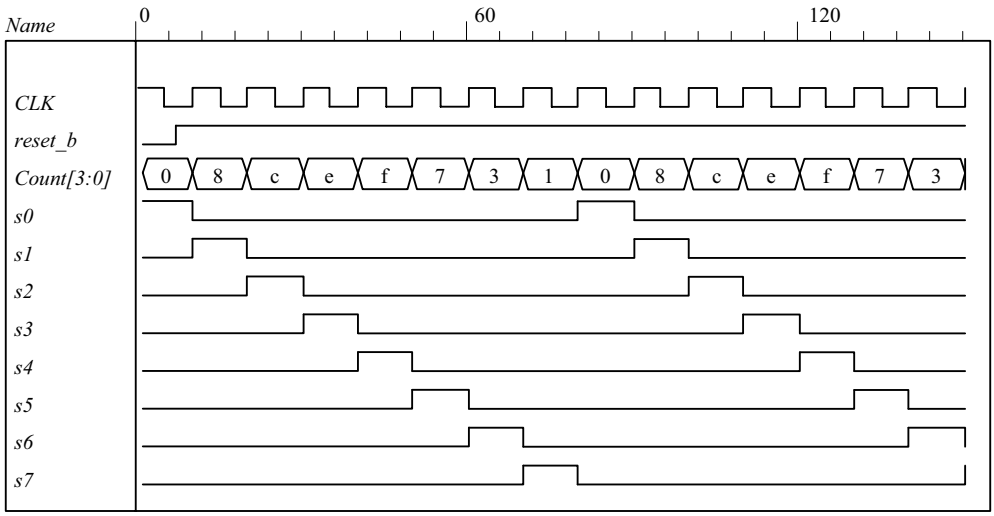
```
module Prob_6_40 (output reg [0: 7] timer, input clk, reset_b);

  always @ (negedge clk, negedge reset_b)
    if (reset_b  == 0) timer <= 8'b1000_0000; else
    case (timer)

      8'b1000_0000:    timer <= 8'b0100_0000;
      8'b0100_0000:    timer <= 8'b0010_0000;
      8'b0010_0000:    timer <= 8'b0001_0000;
      8'b0001_0000:    timer <= 8'b0000_1000;
      8'b0000_1000:    timer <= 8'b0000_0100;
      8'b0000_0100:    timer <= 8'b0000_0010;
      8'b0000_0010:    timer <= 8'b0000_0001;
      8'b0000_0001:    timer <= 8'b1000_0000;
      default:         timer <= 8'b1000_0000;
    endcase
endmodule

module t_Prob_6_40 ();
  wire [0: 7] timer;
  reg clk, reset_b;
```

```
  Prob_6_40 M0 (timer, clk, reset_b);

  initial #250 $finish;
  initial fork #1 reset_b = 0; #7 reset_b = 1; join
  initial begin  clk = 1; forever #5 clk = ~clk; end
endmodule
```



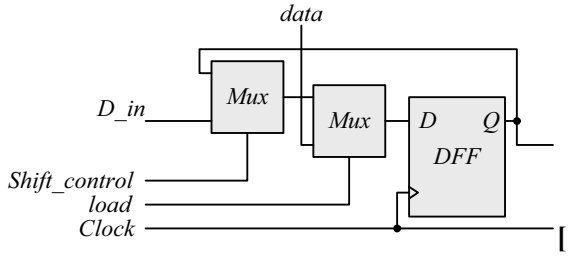
6.41

```
module Prob_6_41_Switched_Tail_Johnson_Counter (output [0: 3] Count, input CLK, reset_b);
  wire Q_0b, Q_1b, Q_2b, Q_3b;

  DFF M3 (Count[3], Q_3b, Count[2], CLK, reset_b);
  DFF M2 (Count[2], Q_2b, Count[1], CLK, reset_b);
  DFF M1 (Count[1], Q_1b, Count[0], CLK, reset_b);
  DFF M0 (Count[0], Q_0b, Q_3b, CLK, reset_b);
endmodule

module DFF (output reg Q, output Q_b,  input D, clk, reset_b);
  assign Q_b = ~Q;
  always @ (posedge clk, negedge reset_b) if (reset_b ==0) Q <= 0; else Q <= D;
endmodule

module t_Prob_6_41_Switched_Tail_Johnson_Counter ();
  wire [3: 0] Count;
  reg CLK, reset_b;
  wire s0 = ~ M0.Count[0] && ~M0.Count[3];
  wire s1 =   M0.Count[0] && ~M0.Count[1];
  wire s2 =   M0.Count[1] && ~M0.Count[2];
  wire s3 =   M0.Count[2] && ~M0.Count[3];
  wire s4 =   M0.Count[0] &&  M0.Count[3];
  wire s5 = ~ M0.Count[0] &&  M0.Count[1];
  wire s6 = ~ M0.Count[1] &&  M0.Count[2];
  wire s7 = ~ M0.Count[2] &&  M0.Count[3];

  Prob_6_41_Switched_Tail_Johnson_Counter M0 (Count, CLK, reset_b);
  initial #150 $finish;
  initial fork #1 reset_b = 0; #7 reset_b = 1; join
  initial begin  CLK = 1; forever #5 CLK = ~CLK; end
endmodule
```



6.42 Because A is a register variable, it retains whatever value has been assigned to it until a new value is assigned. Therefore, the statement $A <= A$ has the same effect as if the statement was omitted.

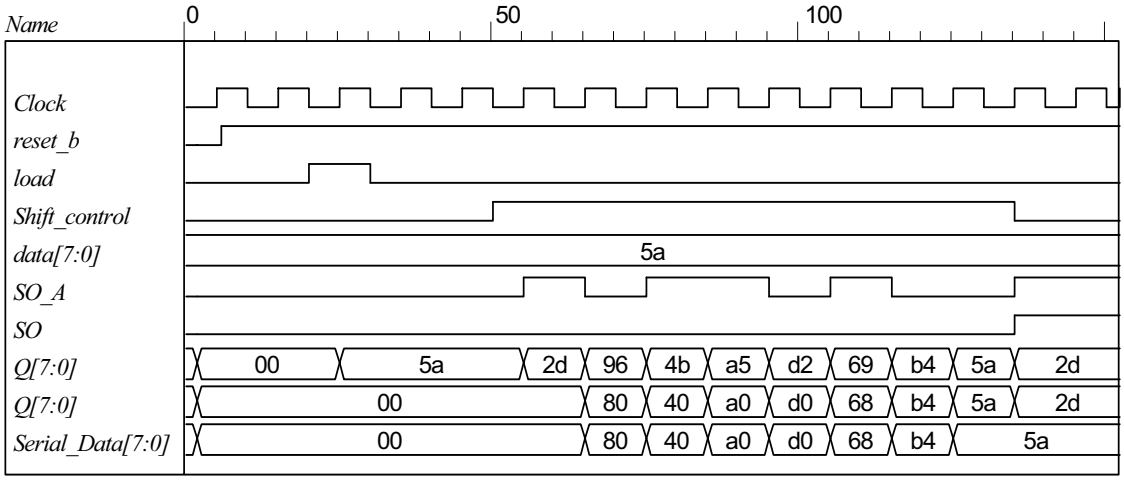
6.43




```
module Prob_6_43_Str (output SO, input [7: 0] data, input load, Shift_control, Clock, reset_b);
  supply0 gnd;
  wire SO_A;

  Shift_with_Load M_A (SO_A, SO_A, data, load, Shift_control, Clock, reset_b);
  Shift_with_Load M_B (SO,  SO_A, data, gnd, Shift_control, Clock, reset_b);

endmodule
```

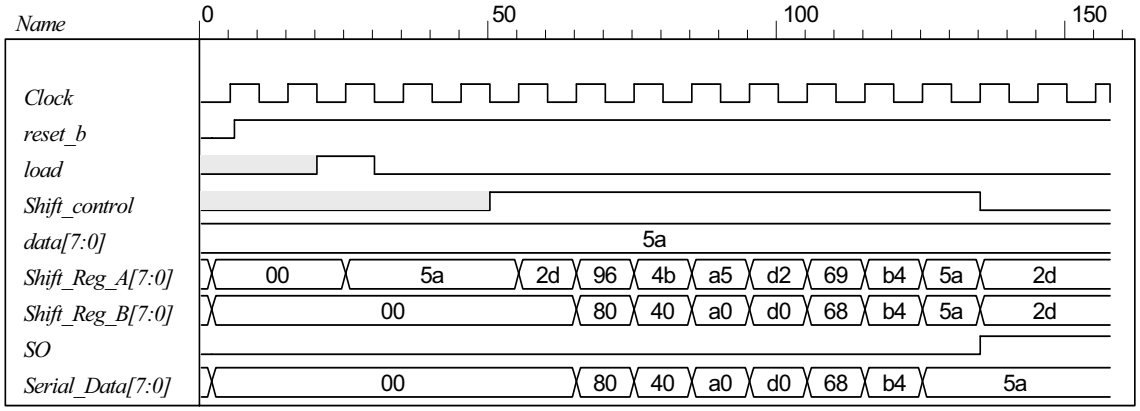
```
module Shift_with_Load (output SO, input D_in, input [7: 0] data, input load, select, Clock, reset_b);
  wire [7: 0] Q;
  assign SO = Q[0];
  SR_cell M7 (Q[7], D_in, data[7], load, select, Clock, reset_b);
  SR_cell M6 (Q[6], Q[7], data[6], load, select, Clock, reset_b);
  SR_cell M5 (Q[5], Q[6], data[5], load, select, Clock, reset_b);
  SR_cell M4 (Q[4], Q[5], data[4], load, select, Clock, reset_b);
  SR_cell M3 (Q[3], Q[4], data[3], load, select, Clock, reset_b);
  SR_cell M2 (Q[2], Q[3], data[2], load, select, Clock, reset_b);
  SR_cell M1 (Q[1], Q[2], data[1], load, select, Clock, reset_b);
  SR_cell M0 (Q[0], Q[1], data[0], load, select, Clock, reset_b);
```

```
endmodule

module SR_cell (output Q, input D, data, load, select, Clock, reset_b);
  wire y;
  DFF_with_load M0 (Q, y, data, load, Clock, reset_b);
  Mux_2 M1 (y, Q,  D, select);
endmodule
module DFF_with_load (output reg Q, input D, data, load, Clock, reset_b);
  always @ (posedge Clock, negedge reset_b)
    if (reset_b == 0) Q <= 0; else if (load) Q <= data; else Q <= D;
endmodule

module Mux_2 (output reg y, input a, b, sel);
  always @ (a, b, sel) if (sel ==1) y = b; else y = a;
endmodule

module t_Fig_6_4_Str ();
  wire SO;
  reg load, Shift_control, Clock, reset_b;
  reg [7: 0] data, Serial_Data;

  Prob_6_43_Str M0 (SO, data, load, Shift_control, Clock, reset_b);

  always @ (posedge Clock, negedge reset_b)
  if (reset_b == 0) Serial_Data <= 0;
  else if (Shift_control ) Serial_Data <= {M0.SO_A, Serial_Data [7: 1]};

  initial #200 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial begin #2 reset_b = 0; #4 reset_b = 1; end

  initial fork
    data = 8'h5A;
    #20 load = 1;
    #30 load = 0;
    #50 Shift_control = 1;
    #50 begin repeat (9) @ (posedge Clock) ;
      Shift_control = 0;
    end
  join
endmodule
```



Alternative: a behavioral model for synthesis is given below. The behavioral description implies the need for a mux at the input to a D-type flip-flop.

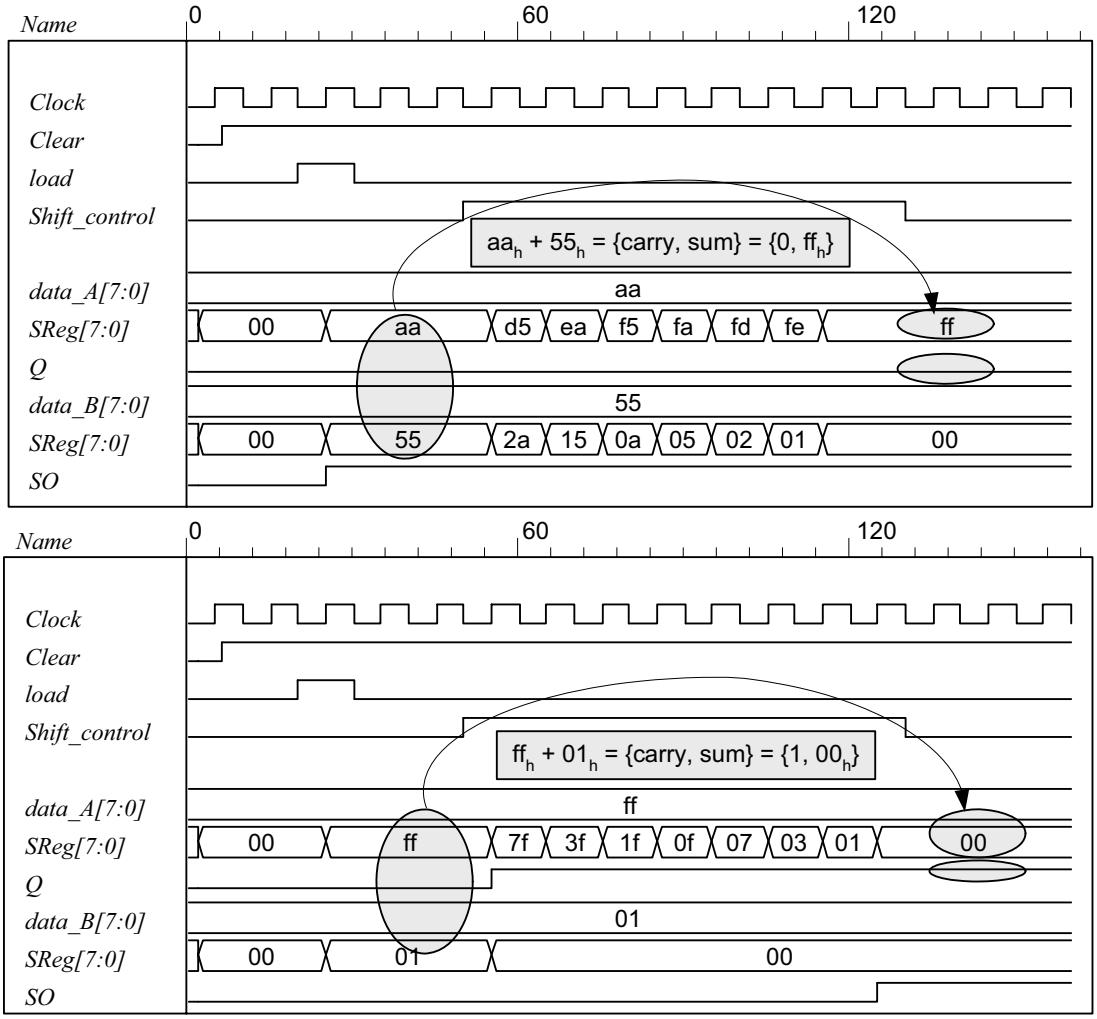
```
module Fig_6_4_Beh (output SO, input [7: 0] data, input load, Shift_control, Clock, reset_b);
  reg [7: 0] Shift_Reg_A, Shift_Reg_B;
  assign SO = Shift_Reg_B[0];
  always @ (posedge Clock, negedge reset_b)
    if (reset_b == 0) begin
      Shift_Reg_A <= 0;
      Shift_Reg_B <= 0;
    end
    else if (load) Shift_Reg_A <= data;
    else if (Shift_control) begin
      Shift_Reg_A <= { Shift_Reg_A[0], Shift_Reg_A[7: 1]};
      Shift_Reg_B <= {Shift_Reg_A[0], Shift_Reg_B[7: 1]};
    end

endmodule
module t_Fig_6_4_Beh ();
  wire SO;
  reg load, Shift_control, Clock, reset_b;
  reg [7: 0] data, Serial_Data;

  Fig_6_4_Beh M0 (SO, data, load, Shift_control, Clock, reset_b);

  always @ (posedge Clock, negedge reset_b)
  if (reset_b == 0) Serial_Data <= 0;
  else if (Shift_control ) Serial_Data <= {M0.Shift_Reg_A[0], Serial_Data [7: 1]};

  initial #200 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial begin #2 reset_b = 0; #4 reset_b = 1; end

  initial fork
    data = 8'h5A;
    #20 load = 1;
    #30 load = 0;
    #50 Shift_control = 1;
    #50 begin repeat (9) @ (posedge Clock) ;
      Shift_control = 0;
    end
  join
endmodule
```



6.44

```
// See Figure 6.5
// Note: Sum is stored in shift register A; carry is stored in Q
// Note: Clear is active-low.

module Prob_6_44_Str (output SO, input [7: 0] data_A, data_B, input S_in, load, Shift_control, CLK,
        Clear);
  supply0 gnd;
  wire sum, carry;
  assign SO = sum;
  wire SO_A, SO_B;

  Shift_Reg_gated_clock M_A (SO_A, sum, data_A, load, Shift_control, CLK, Clear);
  Shift_Reg_gated_clock M_B (SO_B,  S_in, data_B, load, Shift_control, CLK, Clear);
  FA M_FA (carry, sum, SO_A, SO_B, Q);
  DFF_gated M_FF (Q, carry, Shift_control, CLK, Clear);

endmodule

module Shift_Reg_gated_clock (output SO, input S_in, input [7: 0] data, input load, Shift_control,
        Clock, reset_b);
  reg [7: 0] SReg;
  assign SO = SReg[0];

  always @ (posedge Clock, negedge reset_b)
    if (reset_b == 0) SReg <= 0;
    else if (load) SReg <= data;
    else if (Shift_control) SReg <= {S_in, SReg[7: 1]};
endmodule

module DFF_gated (output Q, input D, Shift_control, Clock, reset_b);
  DFF M_DFF (Q,   D_internal, Clock, reset_b);
  Mux_2 M_Mux (D_internal, Q, D, Shift_control);
endmodule

module DFF (output reg Q, input D, Clock, reset_b);
  always @ (posedge Clock, negedge reset_b)
    if (reset_b == 0) Q <= 0; else Q <= D;
endmodule

module Mux_2 (output reg y, input a, b, sel);
  always @ (a, b, sel) if (sel ==1) y = b; else y = a;
 endmodule

module FA (output reg carry, sum, input a, b, C_in);
  always @ (a, b, C_in) {carry, sum} = a + b + C_in;
endmodule

module t_Prob_6_44_Str ();
  wire SO;
  reg SI, load, Shift_control, Clock, Clear;
  reg [7: 0] data_A, data_B;

  Prob_6_44_Str M0 (SO, data_A, data_B, SI, load, Shift_control, Clock, Clear);

  initial #200 $finish;
  initial begin Clock = 0; forever #5 Clock = ~Clock; end
  initial begin #2 Clear = 0; #4 Clear = 1; end
```

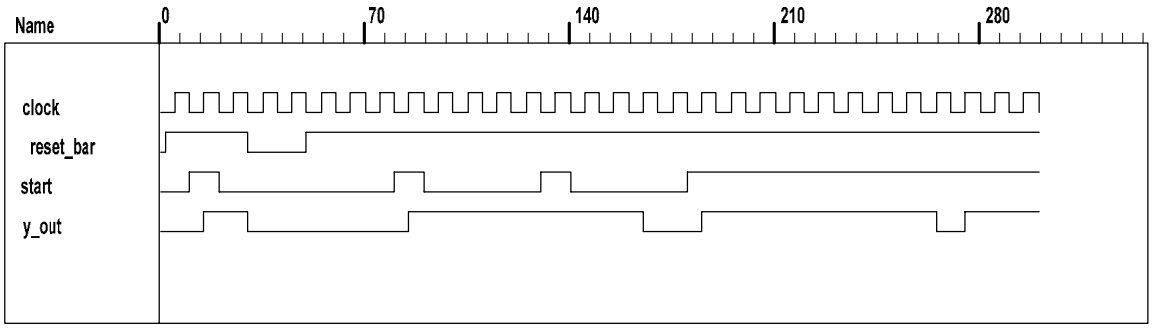
```
 initial fork
  data_A =  8'hAA;   //8'h ff;
  data_B = 8'h55;    //8'h01;
  SI = 0;
  #20 load = 1;
  #30 load = 0;
  #50 Shift_control = 1;
  #50 begin repeat (8) @ (posedge Clock) ;
    #5 Shift_control = 0;
  end
 join
endmodule
```



6.45

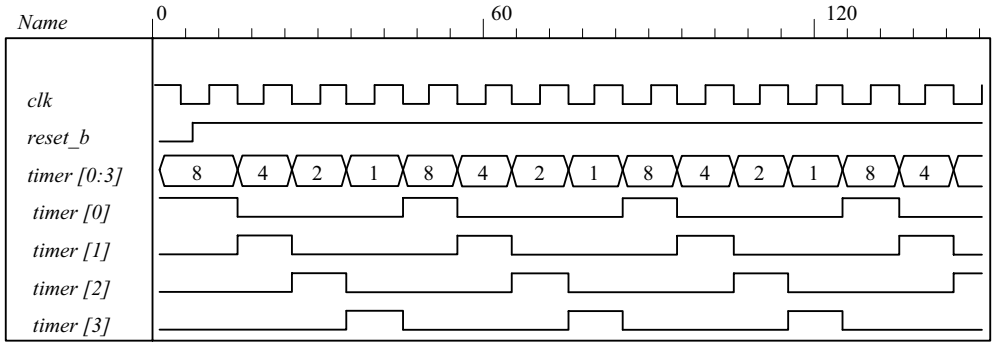
```
module Prob_6_45 (output reg y_out, input start, clock, reset_bar);
 parameter      s0 = 4'b0000,
      s1 = 4'b0001,
      s2 = 4'b0010,
      s3 = 4'b0011,
      s4 = 4'b0100,
      s5 = 4'b0101,
```

```
      s6 = 4'b0110,
      s7 = 4'b0111,
      s8 = 4'b1000;
  reg [3: 0]   state, next_state;

  always @ (posedge clock, negedge reset_bar)
    if (!reset_bar) state <= s0; else state <= next_state;

  always @ (state, start) begin
    y_out = 1'b0;
    case (state)
      s0:   if (start) next_state = s1; else next_state = s0;
      s1:   begin next_state = s2; y_out = 1; end
      s2:   begin next_state = s3; y_out = 1; end
      s3:   begin next_state = s4; y_out = 1; end
      s4:   begin next_state = s5; y_out = 1; end
      s5:   begin next_state = s6; y_out = 1; end
      s6:   begin next_state = s7; y_out = 1; end
      s7:   begin next_state = s8; y_out = 1; end
      s8:   begin next_state = s0; y_out = 1; end
      default: next_state = s0;
    endcase
  end
endmodule


// Test plan

// Verify the following:
// Power-up reset
// Response to start in initial state
// Reset on-the-fly
// Response to re-assertion of start after reset on-the-fly
// 8-cycle counting sequence
// Ignore start during counting sequence
// Return to initial state after 8 cycles and await start
// Remain in initial state for one clock if start is asserted when the state is entered

module t_Prob_6_45;
  wire    y_out;
  reg     start, clock, reset_bar;

  Prob_6_45 M0 (y_out, start, clock, reset_bar);

  initial #300 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
    reset_bar = 0;
    #2 reset_bar = 1;
    #10 start = 1;
    #20 start = 0;
    #30 reset_bar = 0;
    #50 reset_bar = 1;
    #80 start = 1;
    #90 start = 0;
    #130 start = 1;
    #140 start = 0;
    #180 start = 1;
  join
endmodule
```



6.46

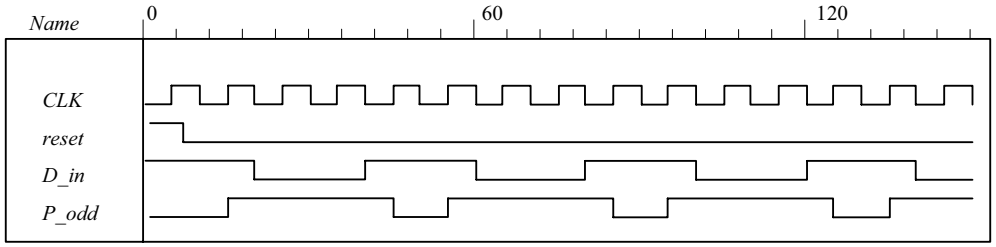
```
module Prob_6_46 (output reg [0: 3] timer, input clk, reset_b);

  always @ (negedge clk, negedge reset_b)
    if (reset_b  == 0) timer <= 4'b1000; else
    case (timer)
      4'b1000:   timer <= 4'b0100;
      4'b0100:   timer <= 4'b0010;
      4'b0010:   timer <= 4'b0001;
      4'b0001:   timer <= 4'b1000;
      default:   timer <= 4'b1000;
    endcase
endmodule

module t_Prob_6_46 ();
  wire [0: 3] timer;
  reg clk, reset_b;

  Prob_6_46 M0 (timer, clk, reset_b);

  initial #150 $finish;
  initial fork #1 reset_b = 0; #7 reset_b = 1; join
  initial begin  clk = 1; forever #5 clk = ~clk; end
endmodule
```



6.47

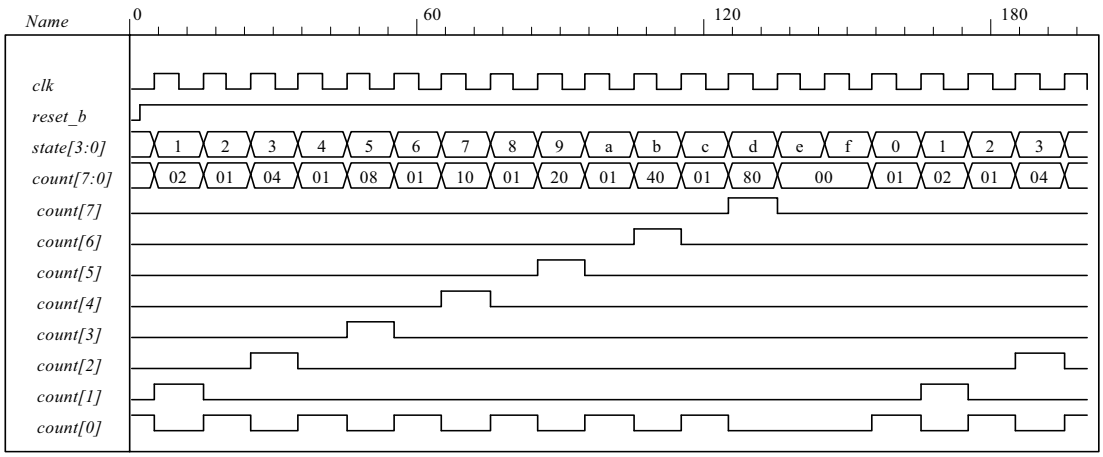
```
module Prob_6_47 (
  output reg P_odd,
  input D_in, CLK, reset
);
  wire D;

  assign D = D_in ^ P_odd;
  always @ (posedge CLK, posedge reset)
```

```
   if (reset)     P_odd <= 0;
   else      P_odd <= D;
endmodule

 module t_Prob_6_47 ();
  wire P_odd;
  reg D_in, CLK, reset;

  Prob_6_47 M0 (P_odd,  D_in, CLK, reset);

  initial #150 $finish;
  initial fork #1 reset = 1; #7 reset = 0; join
  initial begin  CLK = 0; forever #5 CLK = ~CLK; end
  initial begin D_in = 1; forever #20 D_in = ~D_in; end

endmodule
```



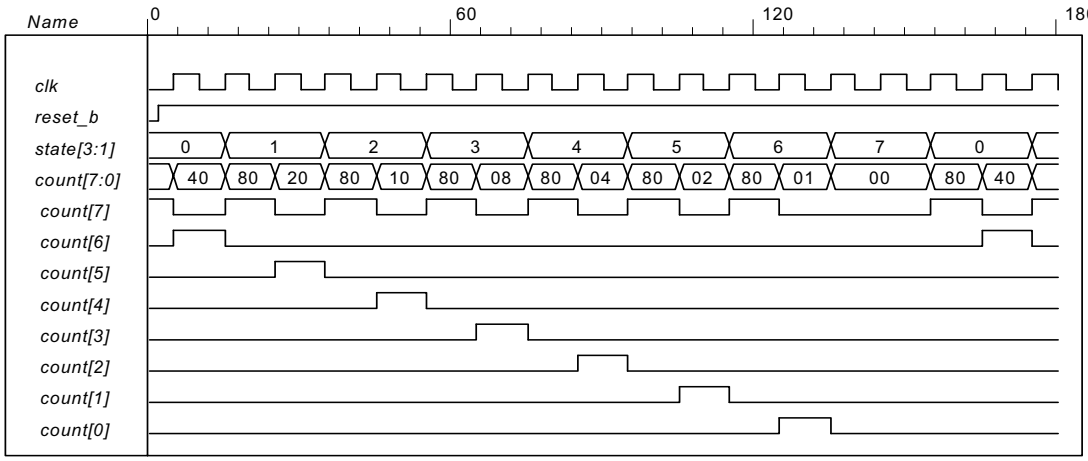
6.48 (a)

```
module Prob_6_48a (output reg [7: 0] count, input clk, reset_b);
  reg [3: 0] state;
  always @ (posedge clk, negedge reset_b)
    if (reset_b == 0) state <= 0; else state <= state + 1;
  always @ (state)
    case (state)
      0, 2, 4, 6, 8, 10, 12:   count = 8'b0000_0001;
      1:            count = 8'b0000_0010;
      3:            count = 8'b0000_0100;
      5:            count = 8'b0000_1000;
      7:            count = 8'b0001_0000;
      9:            count = 8'b0010_0000;
      11:           count = 8'b0100_0000;
      13:           count = 8'b1000_0000;
      default:        count = 8'b0000_0000;
    endcase
endmodule

module t_Prob_6_48a ();
  wire [7: 0] count;
  reg clk, reset_b;

  Prob_6_48a M0 (count, clk, reset_b);

  initial #200 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial begin reset_b = 0; #2 reset_b = 1; end
endmodule
```



(b)

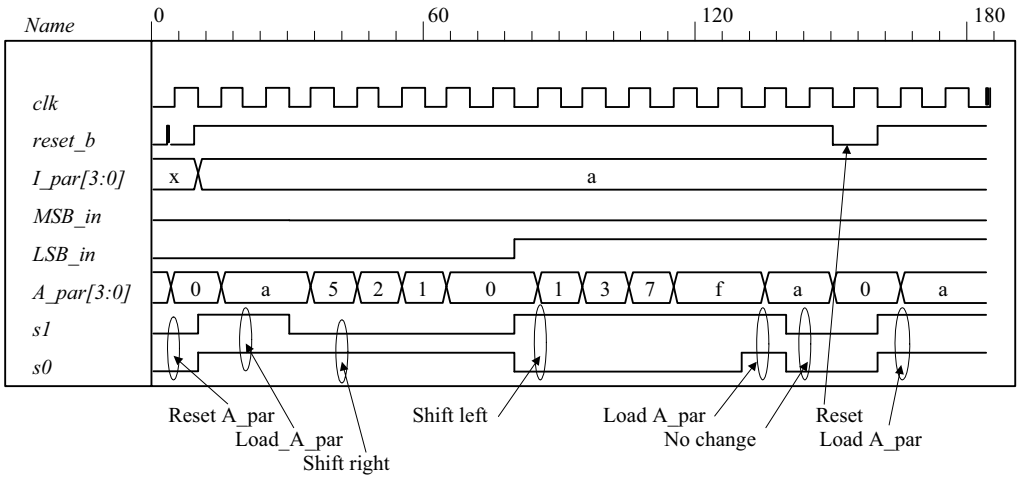
```
module Prob_6_48b (output reg [7: 0] count, input clk, reset_b);
  reg [3: 0] state;
  always @ (posedge clk, negedge reset_b)
    if (reset_b == 0) state <= 0; else state <= state + 1;
  always @ (state)
    case (state)
      0, 2, 4, 6, 8, 10, 12:   count = 8'b1000_0000;
      1:              count = 8'b0100_0000;
      3:              count = 8'b0010_0000;
      5:              count = 8'b0001_0000;
      7:              count = 8'b0000_1000;
      9:              count = 8'b0000_0100;
      11:             count = 8'b0000_0010;
      13:             count = 8'b0000_0001;
      default:        count = 8'b0000_0000;
    endcase
endmodule

module t_Prob_6_48b ();
  wire [7: 0] count;
  reg clk, reset_b;

  Prob_6_48b M0 (count, clk, reset_b);

  initial #180 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial begin reset_b = 0; #2 reset_b = 1; end
endmodule
```



6.49

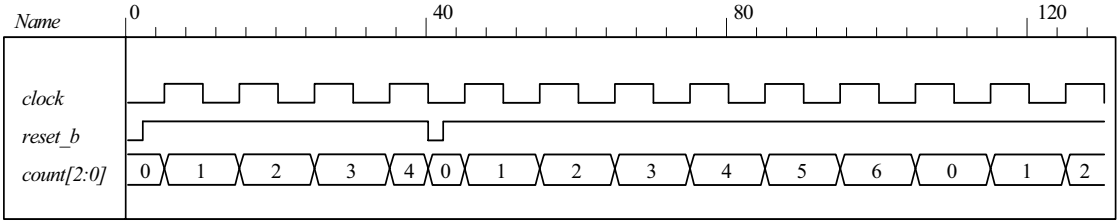
```
// Behavioral description of a 4-bit universal shift register
// Fig. 6.7 and Table 6.3
module Shift_Register_4_beh (            // V2001, 2005
  output reg  [3: 0]   A_par,            // Register output
  input       [3: 0]   I_par,            // Parallel input
  input                s1, s0,           // Select inputs
              MSB_in, LSB_in,  // Serial inputs
              CLK, Clear           // Clock and Clear
);
  always @ (posedge CLK, negedge Clear)    // V2001, 2005
    if (~Clear) A_par <= 4'b0000;
    else
      case ({s1, s0})
        2'b00: A_par <= A_par;           // No change
        2'b01: A_par <= {MSB_in, A_par[3: 1]};          // Shift right
        2'b10: A_par <= {A_par[2: 0], LSB_in};          // Shift left
        2'b11: A_par <= I_par;           // Parallel load of input
      endcase
endmodule
```

```
// Test plan:
// test reset action load
// test parallel load
// test shift right
// test shift left
// test circulation of data
// test reset on the fly

module t_Shift_Register_4_beh ();
  reg    s1, s0,           // Select inputs
     MSB_in, LSB_in,       // Serial inputs
     clk, reset_b;         // Clock and Clear
  reg    [3: 0]   I_par;       // Parallel input
  wire   [3: 0]   A_par;       // Register output

  Shift_Register_4_beh M0 (A_par, I_par,s1, s0, MSB_in, LSB_in, clk, reset_b);

  initial #200 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end

  initial fork
    // test reset action load
    #3 reset_b = 1;
    #4 reset_b = 0;
    #9 reset_b = 1;

    // test parallel load
    #10 I_par = 4'hA;
    #10 {s1, s0} = 2'b11;

    // test shift right
    #30 MSB_in = 1'b0;
    #30 {s1, s0} = 2'b01;

    // test shift left
    #80 LSB_in = 1'b1;
    #80 {s1, s0} = 2'b10;

    // test circulation of data
   #130 {s1, s0} = 2'b11;
    #140 {s1, s0} = 2'b00;

   // test reset on the fly

    #150 reset_b = 1'b0;
    #160 reset_b = 1'b1;
    #160 {s1, s0} = 2'b11;

  join
endmodule
```



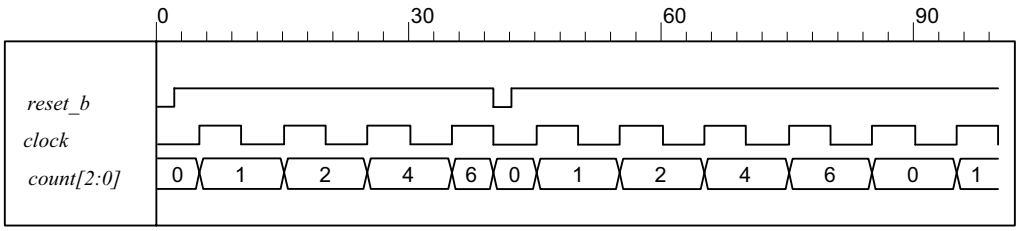
6.50 **(a)** See problem 6.27.

```
module Prob_8_50a (output reg [2: 0] count, input clk, reset_b);
  always @ (posedge clk, negedge reset_b)
    if (!reset_b) count <= 0;
    else case (count)
      3'd0:    count <= 3'd1;
      3'd1:    count <= 3'd2;
      3'd2:    count <= 3'd3;
      3'd3:    count <= 3'd4;
      3'd4:    count <= 3'd5;
      3'd5:    count <= 3'd6;
      3'd4:    count <= 3'd6;
      3'd6:    count <= 3'd0;
      default:    count <= 3'd0;
    endcase
endmodule

module t_Prob_8_50a;
  wire [2: 0] count;
  reg clock, reset_b  ;

  Prob_8_50a M0 (count, clock, reset_b);

  initial #130 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
        reset_b = 0;
    #2 reset_b = 1;
    #40 reset_b = 0;
    #42 reset_b = 1;
  join
endmodule
```



(b) See problem 6.28.

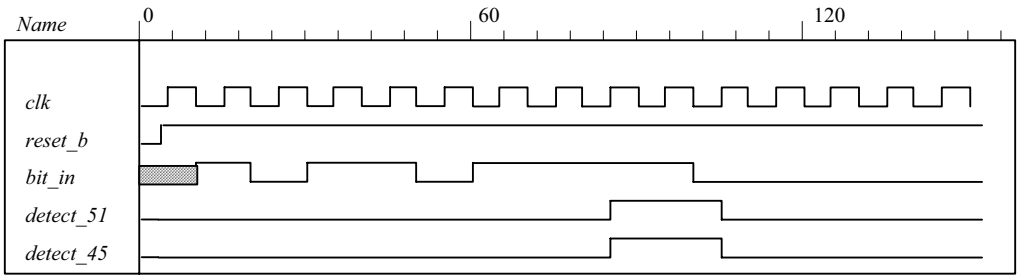
```
module Prob_8_50b (output reg [2: 0] count, input clk, reset_b);
 always @ (posedge clk, negedge reset_b)
   if (!reset_b) count <= 0;
   else case (count)
     3'd0:    count <= 3'd1;
     3'd1:    count <= 3'd2;
     3'd2:    count <= 3'd4;
     3'd4:    count <= 3'd6;
     3'd6:    count <= 3'd0;
     default:    count <= 3'd0;
   endcase
endmodule

module t_Prob_8_50b;
 wire [2: 0] count;
 reg clock, reset_b  ;

 Prob_8_50b M0 (count, clock, reset_b);

 initial #100 $finish;
 initial begin clock = 0; forever #5 clock = ~clock; end
 initial fork
       reset_b = 0;
   #2 reset_b = 1;
   #40 reset_b = 0;
   #42 reset_b = 1;
 join
endmodule
```



6.51

```
module Seq_Detector_Prob_5_51 (output detect, input bit_in, clk, reset_b);
 reg [2: 0] sample_reg;
 assign detect = (sample_reg == 3'b111);
 always @ (posedge clk, negedge reset_b) if (reset_b ==0) sample_reg <= 0;
   else sample_reg <= {bit_in, sample_reg [2: 1]};
endmodule

module Seq_Detector_Prob_5_45 (output detect, input bit_in, clk, reset_b);
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
reg [1: 0] state, next_state;

assign detect = (state == S3);
always @ (posedge clk, negedge reset_b)
if (reset_b == 0) state <= S0; else state <= next_state;

 always @ (state, bit_in) begin
   next_state = S0;
   case (state)
```

```
    0:      if (bit_in) next_state = S1; else state = S0;
    1:      if (bit_in) next_state = S2; else next_state = S0;
    2:      if (bit_in) next_state = S3; else state = S0;
    3:      if (bit_in) next_state = S3; else next_state = S0;
    default:   next_state = S0;
   endcase
  end
endmodule

module t_Seq_Detector_Prob_6_51 ();
  wire detect_45, detect_51;
  reg bit_in, clk, reset_b;

  Seq_Detector_Prob_5_51 M0 (detect_51, bit_in, clk, reset_b);
  Seq_Detector_Prob_5_45 M1 (detect_45, bit_in, clk, reset_b);

  initial #350$finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork

        reset_b = 0;
   #4   reset_b = 1;
   #10 bit_in = 1;
   #20 bit_in = 0;
   #30 bit_in = 1;
   #50 bit_in = 0;
   #60 bit_in = 1;
   #100 bit_in = 0;
  join
endmodule
```



The circuit using a shift register uses less hardware.

Chapter 7

7.1 **(a)** 8 K x 32 = 2^{13} x 16 A = 13 D = 16

(b) 2 G x 8 = 2^{31} x 8 A = 31 D = 8

(c) 16 M x 32 = 2^{24} x 32 A = 24 D = 32

(d) 256 K x 64 = 2^{18} x 64 A = 18 D = 64

(e)

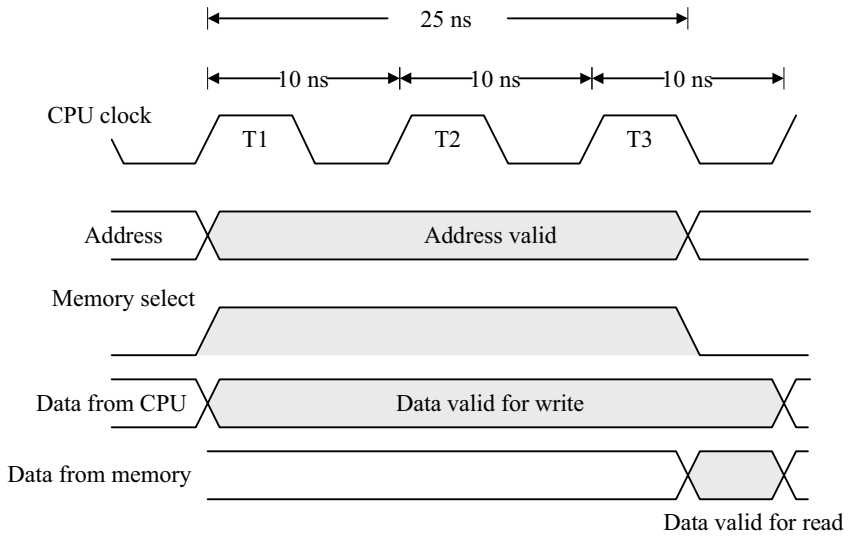
7.2 **(a)** 2^{13} **(b)** 2^{31} **(c)** 2^{26} **(d)** 2^{21}

7.3 723 = 512 + 128 + 64 + 16 + 2 + 1

3451 = 2048 + 1024 + 256 + 64 + 32 + 16 + 8 + 2 + 1

Address: 10 1101 0011 = $2D3_{16}$
Data: 0000 1101 0111 1011 = $0D7B_{16}$

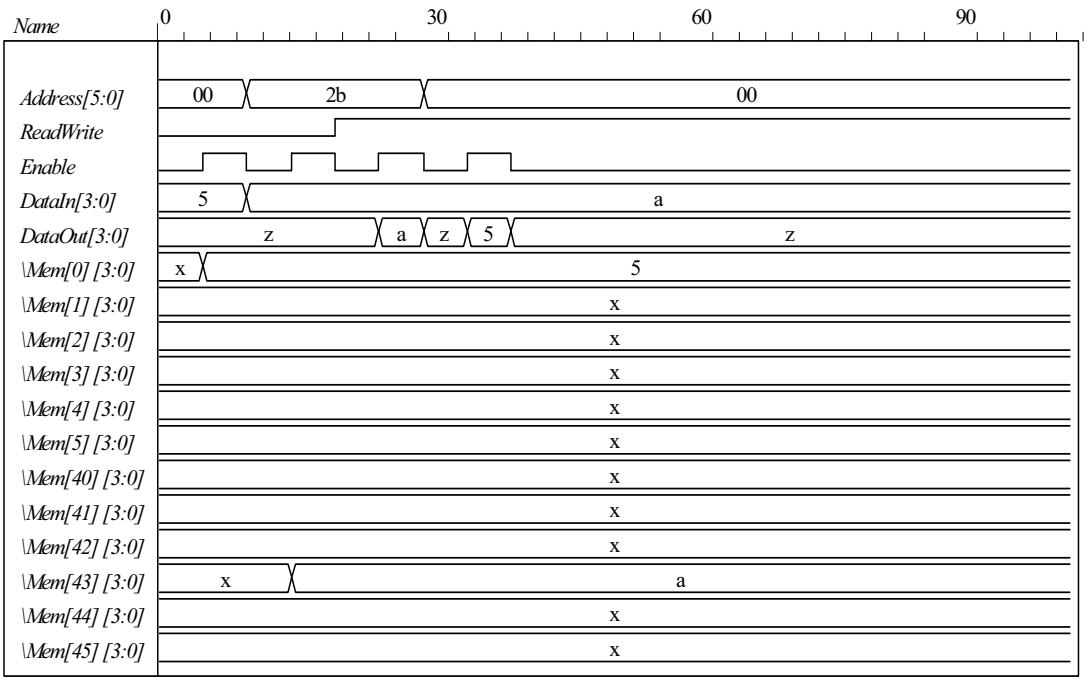
7.4 f_{CPU} = 100 MHz, $T_{CPU} = 1/f_{CPU} = 10^{-8}$ $Hz^{-1} = 10 \times 10^{-9}$ Hz^{-1} = 10 ns



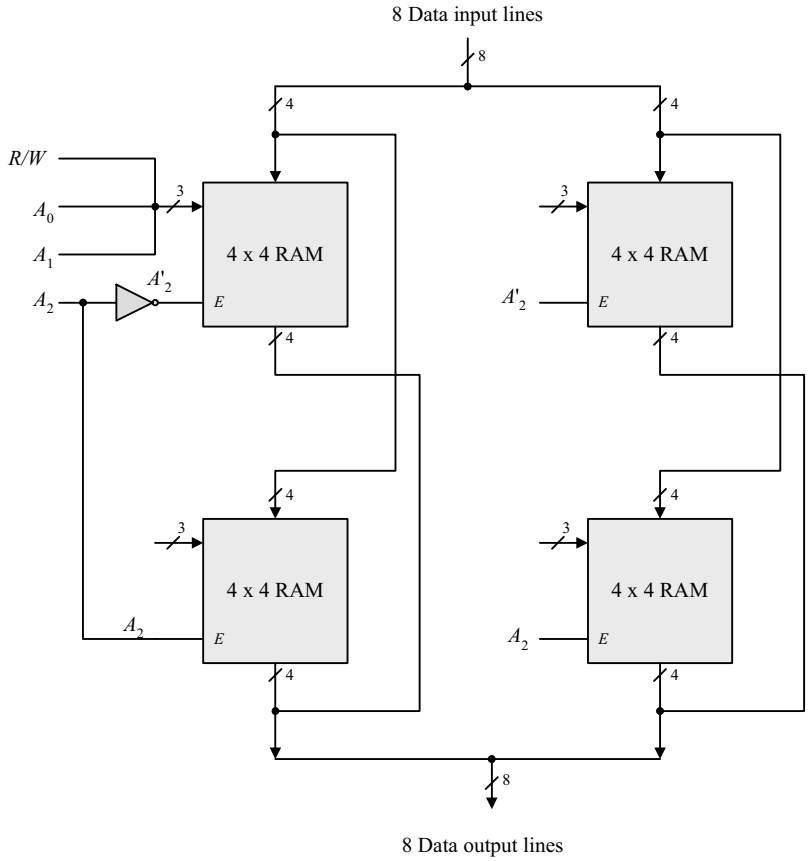
7.5

```
// Testing the memory of HDL Example 7.1.
module t_memory ();
 reg          Enable, ReadWrite;
 reg   [3: 0] DataIn;
 reg   [5: 0] Address;
 wire  [3: 0] DataOut;

 memory M0 (Enable, ReadWrite, Address, DataIn, DataOut);
 initial #200 $finish;
 initial begin
  Enable = 0; ReadWrite = 0; Address = 3; DataIn = 5;
  repeat (8) #5 Enable = ~Enable;
 end
 initial begin
```

```
    #10 Address = 43; DataIn = 10;
    #10 ReadWrite = 1;
    #10 Address = 0;
  end
  initial
    $monitor ("E = %b RW = %b Add = %b D_in = %b D_out = %b T = %d",
      Enable, ReadWrite, Address, DataIn, DataOut, $time);

  wire mem0 = M0.Mem[0];
  wire mem1 =M0.Mem[1];
  wire mem2 =M0.Mem[2];
  wire mem3 =M0.Mem[3];
  wire mem4 =M0.Mem[4];
  wire mem5 =M0.Mem[5];
  wire mem40 =M0.Mem[40];
  wire mem41 =M0.Mem[41];
  wire mem42 =M0.Mem[42];
  wire mem43 =M0.Mem[43];
  wire mem44 =M0.Mem[44];
  wire mem45 =M0.Mem[45];
endmodule

//Read and write operations of Mem
//Mem size is 64 words of 4 bits each.
module memory (Enable, ReadWrite, Address, DataIn, DataOut);
  input Enable, ReadWrite;
  input  [3: 0]  DataIn;
  input      [5: 0]   Address;
  output    [3:0]    DataOut;
  reg    [3: 0]   DataOut;
  reg    [3: 0]   Mem [0: 63];            //64 x 4 Mem
  always @ (Enable or ReadWrite)
    if (Enable)
      if (ReadWrite) DataOut = Mem[Address];        //Read
      else Mem[Address] = DataIn;              //Write
    else DataOut = 4'bz;                  //High impedance state
endmodule
```



7.6



7.7 **(a)** 16 K = $2^{14} = 2^7 \times 2^7 = 128 \times 128$
Each decoder is 7×128
Decoders require 256 AND gates, each with 7 inputs

(b) 6,000 = 0101110_1110000
$x = 46$ $y = 112$

7.8 **(a)** 256 K / 32 K = 8 chips

(b) 256 K = 2^{18} (18 address lines for memory); 32 K = 215 (15 address pins / chip)

(c) 18 – 15 = 3 lines ; must decode with 3×8 decoder

7.9 13 + 12 = 25 address lines. Memory capacity = 2^{25} words.

7.10 01011011 = 1 2 3 4 5 6 7 8 9 10 11 12 13
P_1 P_2 0 P_4 1 0 1 P_8 1 0 1 1 P_{13}

P_1 = Xor of bits (3, 5, 7, 9, 11) = 0, 1, 1, 1, 1 = 0 (Note: even # of 0s)
P_2 = Xor of bits (3, 6, 7, 10, 11) = 0, 0, 1, 0, 1 = 0
P_4 = Xor of bits (5, 6, 7, 12) = 1, 0, 1, 1 = 1 (Note: odd # of 0s)
P_8= Xor of bits (9, 10, 11, 12) = 1, 0, 1, 1, = 1

Composite 13-bit code word: 0001 1011 1011 1

7.11 11001001010 = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
P_1 P_2 1 P_4 1 0 0 P_8 1 0 0 1 0 1 0

P_1 = Xor of bits (3, 5, 7, 9, 11, 13, 15) = 1, 1, 0, 1, 0, 0, 0 = 1 (Note: odd # of 0s)
P_2 = Xor of bits (3, 6, 7, 10, 11, 14, 15) = 1, 0, 0, 0, 0, 1, 0 = 0 (Note: even # of 0s)
P_4 = Xor of bits (5, 6, 7, 12, 13, 14, 15) = 1, 0, 0, 1, 0, 1, 0 = 1
P_8= Xor of bits (9, 10, 11, 12, 13, 14, 15) = 1, 0, 0, 1, 0, 1, 0 = 1

Composite 15-bit code word: 101 110 011 001 010

7.12 **(a)** 1 2 3 4 5 6 7 8 9 10 11 12
0 0 0 0 1 1 1 0 1 0 1 0

C_1 (1, 3, 5, 7, 9, 11) = 0, 0, 1, 1, 1, 1 = 0
C_2 (2, 3, 6, 7, 10, 11) = 0, 0, 1, 1, 0, 1 = 1
C_4 (4, 5, 6, 7, 12) = 0, 1, 1, 1, 0 = 1
C_8 (8, 9, 10, 11, 12) = 0, 1, 0, 1, 0 = 0

C = 0110
Error in bit 6.
Correct data: 0101 1010

(b) 1 2 3 4 5 6 7 8 9 10 11 12
1 0 1 1 1 0 0 0 0 1 1 0

C_1 (1, 3, 5, 7, 9, 11) = 1, 1, 1, 0, 0, 1 = 0
C_2 (2, 3, 6, 7, 10, 11) = 0, 1, 0, 0, 1, 1 = 1
C_4 (4, 5, 6, 7, 12) = 1, 1, 0, 0, 0 = 0
C_8 (8, 9, 10, 11, 12) = 0, 0, 1, 1, 0 = 0

C = 0010
Error in bit 2 = Parity bit P_2.

	3	5	6	7	9	10	11	12
Correct 8-bit data:	1	1	0	0	0	1	1	0

(c)

1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	1	1	1	1	1	0	1	0	0

C = 0000)No errors)
C_1 (1, 3, 5, 7, 9, 11) = 1, 1, 1, 0, 0, 1 = 0
C_2 (2, 3, 6, 7, 10, 11) = 0, 1, 0, 0, 1, 1 = 1
C_4 (4, 5, 6, 7, 12) = 1, 1, 0, 0, 0 = 0
C_8 (8, 9, 10, 11, 12) = 0, 0, 1, 1, 0 = 0

	3	5	6	7	9	10	11	12
Correct 8-bit data:	1	1	1	1	0	1	0	0

7.13 (**a**) 16-bit data (From Table 7.2): 5 Check bits
1 bit
---
6 parity bits

(**b**) 32-bit data (From Table 7.2): 6 Check bits
1 bit
---
7 parity bits

(6) 16-bit data (From Table 7.2): 5 Check bits
1 bit
---
6 parity bits

7.14 **(a)**

1	2	3	4	5	6	7
P_1	P_2	0	P4	0	1	0

P_1 = Xor (3, 5, 7) = 0, 0, 0 = 1
P_2 = Xor (3, 6, 7) = 0, 1, 0 = 0
P_4 = Xor (5, 6, 7) = 0, 1, 0 = 1

7-bit word: 0101010

(b) No error:

C_1 = Xor (1, 3, 5, 7) = 0, 0, 0, 0 = 0
C_2 = Xor (2, 3, 6, 7) = 1, 0, 1, 0 = 0
C_4 = Xor (4, 5, 6, 7) = 1, 0, 1, 0 = 0

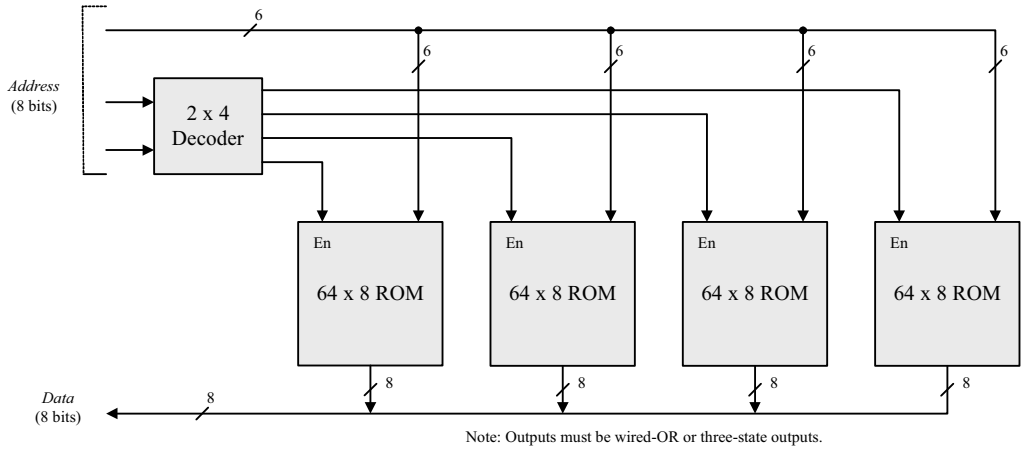
(c) Error in bit 5:

1	2	3	4	5	6	7
0	1	0	1	1	1	0

C_1 = Xor (0, 0, 1, 0) = 1
C_2 = Xor (1, 0, 1, 0) = 0
C_4 = Xor (1, 1, 1, 0) = 1
Error in bit 5: C = 101

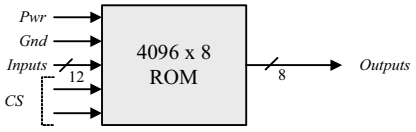
(d) 8-bit word 1 2 3 4 5 6 7 8
0 1 0 1 0 1 0 1
Error in bits 2 and 5: 0 0 0 1 1 1 0 1
C_1 = Xor (0, 0, 1, 0) = 1
C_2 = Xor (0, 0, 1, 0) = 1
C4 = Xor (1, 1, 1, 0) = 1
P = 0
C =(1, 1, 1) ≠ 0 and P = 0 indicates double error.

7.15



7.16

Note: $4096 = 2^{12}$



16 inputs + 8 outputs requires a 24-pin IC.

7.18 **(a)** 256 × 8 **(b)** 512 × 5 **(c)** 1024 × 4 **(d)** 32 × 7

7.17

Input Address	Output of ROM			
$I_5 I_4 I_3 I_2 I_1$	$D_6 D_5 D_4$	$D_3 D_2 D_1$	D_0 (2^0)	Decimal
0 0 0 0 0	0 0 0	0 0 0	0, 1	0, 1
0 0 0 0 1	0 0 0	0 0 1	0, 1	2, 3
…	…	…	…	…
…	…	…	…	…
0 1 0 0 0	0 0 1	0 1 1	0, 1	16, 17
0 1 0 0 1	0 0 1	1 0 0	0, 1	18, 19
…	…	…	…	…
…	…	…	…	…
1 1 1 1 0	1 1 0	0 0 0	0, 1	60, 61
1 1 1 1 1	1 1 0	0 0 1	0, 1	62, 63

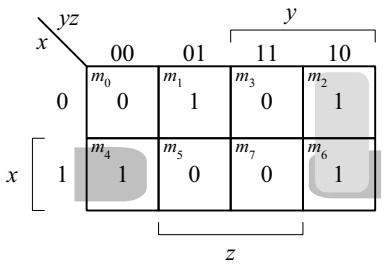
7.18 **(a)** 8 inputs 8 outputs 2^8 x 8 256 x 8 ROM

(b) 9 inputs 5 outputs 2^9 x 5 512 x 5 ROM

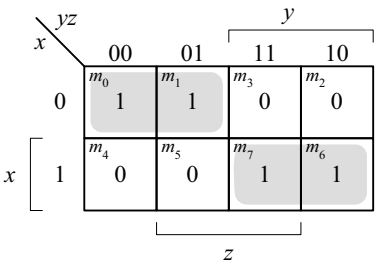
(c) 10 inputs 4 outputs 2^{10} x 4 1024 x 4 ROM

(d) 5 inputs 7 outputs 2^5 x 7 32 x 7 ROM

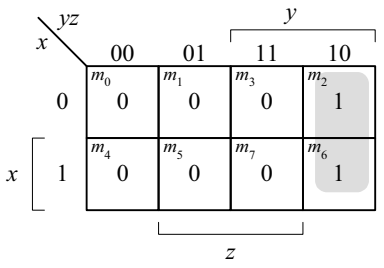
7.19



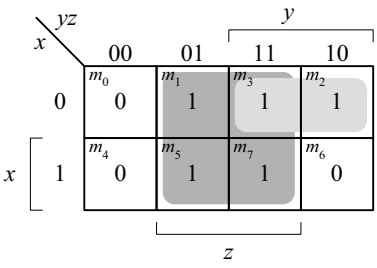
$A = yz' + xz' + x'y'z$
$A' = yz + xz + x'y'z'$



$B = xy + x'y'$
$B' = x'y' + x'y$



$C = yz'$
$C' = y' + z$



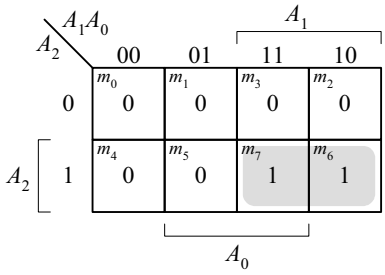
$D = z + x'y$
$D' = y'z' + xz'$

	Product term	Inputs $x\ y\ z$	Outputs $A\ B\ C\ D$
yz'	1	- 1 0	1 - 1 -
xz'	2	1 - 0	1 - - -
$x'y'z$	3	0 0 1	1 - - -
xy'	4	1 0 -	- 1 - -
$x'y$	5	0 1 -	- 1 - 1
z	6	- - 1	- - - 1
			T C T T

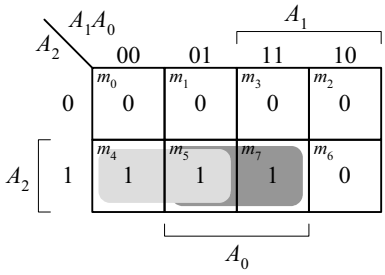
7.20

Inputs $x\ y\ z$	Outputs A, B, C, D	
0 0 0	1 1 0 1	
0 0 1	0 1 1 1	$M[001] = 0111$
0 1 0	0 0 0 0	
0 1 1	1 0 0 0	
1 0 0	1 0 0 1	$M[100] = 1001$
1 0 1	0 0 1 1	
1 1 0	1 1 0 0	
1 1 1	0 1 0 1	

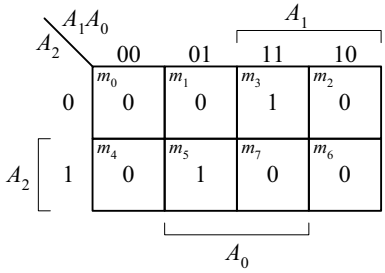
7.21 Note: See truth table in Fig. 7.12(b).



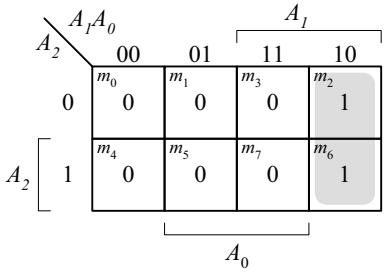
$F_1 = A_2A_1$
$F'_1 = A'_2 + A'_1$



$F_2 = A_2A'_1 + A_2A_0$
$F_2' = A'_2 + A_1A'_0$



$F_3 = A'_2A_1A_0 + A_2A'_1A_0$
$F_3' = A'_0 + A'_2A'_1 + A_2A_1$



$F_4 = A_1A'_0$
$F'_4 = A'_1 + A_0$

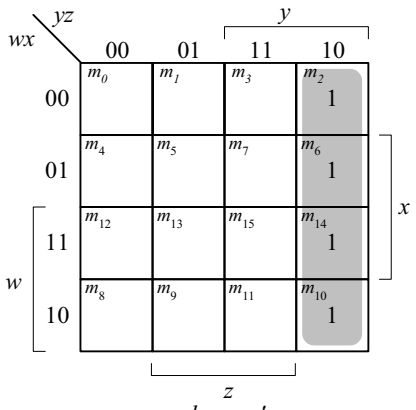
	Product term	Inputs $A_2A_1A_0$	Outputs F_1	F_2	F_3	F_4
A_2A_1	1	1 1 -	1	-	-	-
A'_2	2	0 - -	-	1	-	-
$A_1A'_0$	3	- 1 0	-	1	-	1
$A'_2A_1A_0$	4	- 1 1	-	-	1	-
$A_2A'_1$	5	1 0 1	-	-	1	-
			T	*C*	*T*	*T*

Alternative: F'_1, F'_2, F_3, F_4 (5 terms)

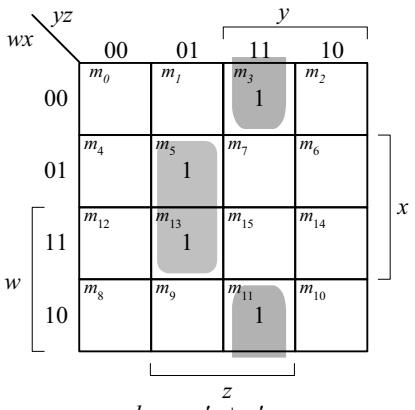
7.22

Decimal		w	x	y	z	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	1
2	4	0	0	1	0	0	0	0	0	0	1	0	0
3	9	0	0	1	1	0	0	0	0	1	0	0	1
4	16	0	1	0	0	0	0	0	1	0	0	0	0
5	25	0	1	0	1	0	0	0	1	1	0	0	1
6	36	0	1	1	0	0	0	1	0	0	1	0	0
7	49	0	1	1	1	0	0	1	1	0	0	0	1
8	64	1	0	0	0	0	1	0	0	0	0	0	0
9	81	1	0	0	1	0	1	0	1	0	0	0	1
10	100	1	0	1	0	0	1	1	0	0	1	0	0
11	121	1	0	1	1	0	1	1	1	1	0	0	1
12	144	1	1	0	0	1	0	0	1	0	0	0	0
13	169	1	1	0	1	1	0	1	0	1	0	0	1
14	196	1	1	1	0	1	1	0	0	0	1	0	0
15	225	1	1	1	1	1	1	1	0	0	0	0	1

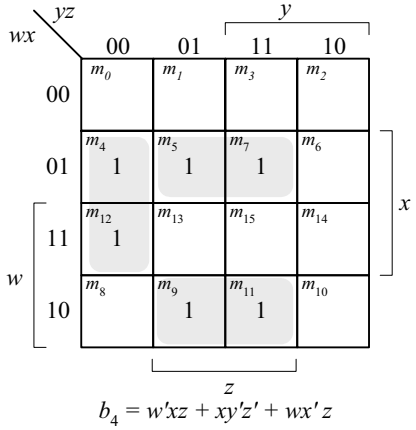
Note: $b_0 = z$, and $b_1 = 0$. ROM would have 4 inputs and 6 outputs. A 4 x 8 ROM would waste two outputs.



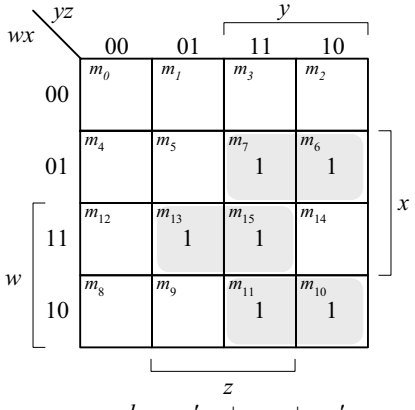
$b_2 = yx'$



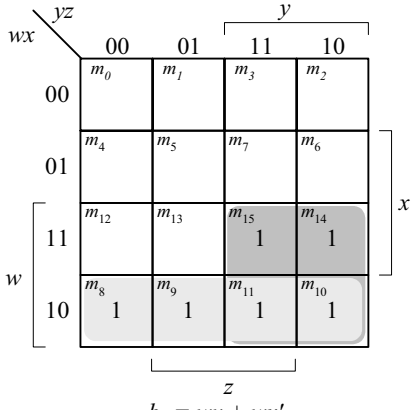
$b_3 = xy'z + x'yz$



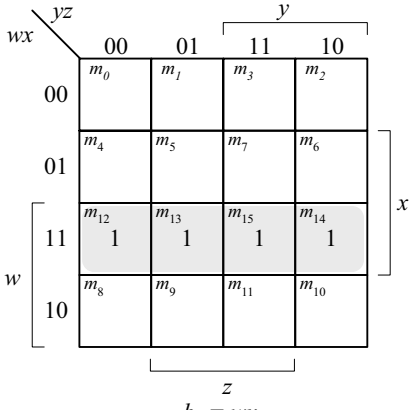
$b_4 = w'xz + xy'z' + wx'z$



$b_5 = w'xy + wxz + wx'y$



$b_6 = wy + wx'$



$b_7 = wx$

7.23

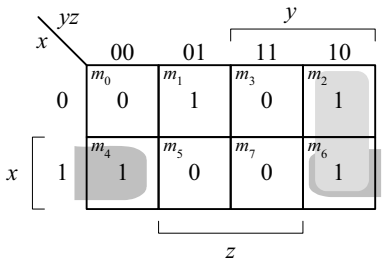
From Fig. 4-3:
$w = A + BC + BD$
$w' = A'B' + A'C'D'$
$x = B'C + B'D + BC'D'$
$x' = B'C'D' + BC\ \ BD$
$y = CD + C'D'$
$y' = C'D + CD'$
$z = D'$
$z' = D$
Use w, x', y, z (7 terms)

	Product term	Inputs $A\ B\ C\ D$	Outputs $F_1\ F_2\ F_3\ F_4$
A	1	1 - - -	1 - - -
BC	2	- 1 1 -	1 1 - -
BD	3	- 1 - 1	1 1 - -
$B'C'D'$	4	- 0 0 0	- 1 - -
CD	5	- - 1 1	- - 1 -
$C'D'$	6	- - 0 0	- - 1 -
D'	7	- - - 0	- - - 1
			$T\ C\ T\ T$

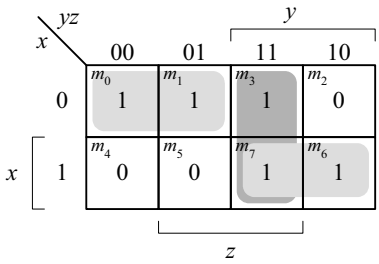
7.24

Product term	AND Inputs $A\ B\ C\ D$	Outputs
1	1 - - -	$w = A + BC + BD$
2	- 1 1 -	
3	- 1 - 1	
4	- 0 1 -	$x = B'C + B'D + BC'D'$
5	- 0 - 1	
6	- 1 0 0	
7	- - 1 1	$y = CD + C'D'$
8	- - 0 0	
9	- - - -	
10	- - - 0	$z = D'$
11	- - - -	
12	- - - -	

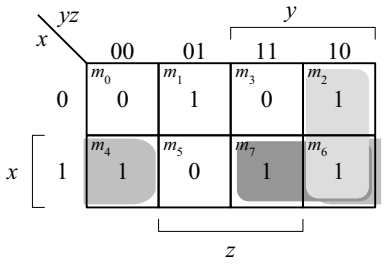
7.25



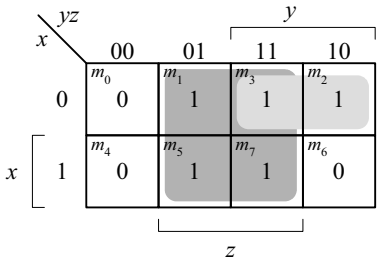
$A = yz' + xz' + x'y'z$



$B = x'y' + xy + yz$



$C = A + xyz$

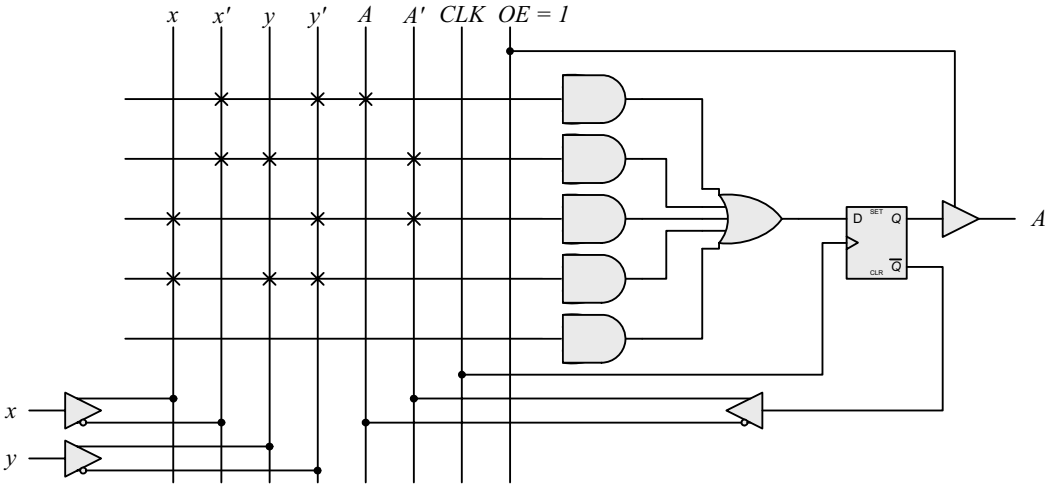


$D = z + x'y$

Product term	AND Inputs x y z A	Outputs
1	- 1 0 -	A = yz' + xz' + x'y'z
2	1 - 0 -	
3	0 0 1 -	
4	0 0 - -	B = x'y' + xy + yz
5	1 1 - -	
6	0 1 1 -	
7	0 - - 1	C = A + xyz
8	1 1 1 -	
9	0 - - -	
10	0 - 1 -	D = z + x'y
11	0 1 - -	
12	- - - -	

$A = yz' + xz' + x'y'z$
$B = x'y' + xy + yz$
$C = A + xyz$
$D = z + x'y$

7.26

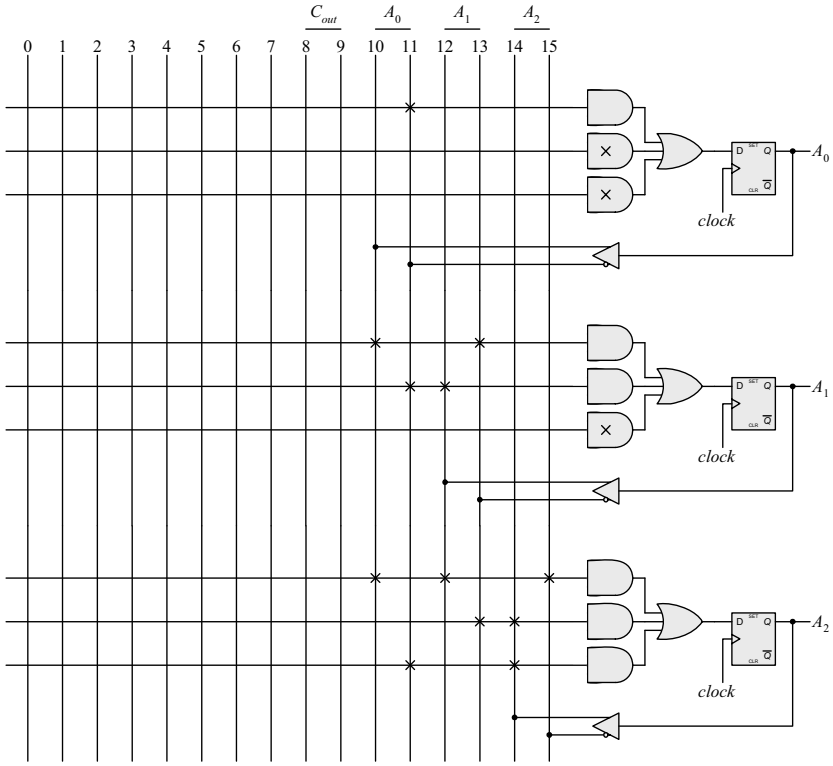


7.27

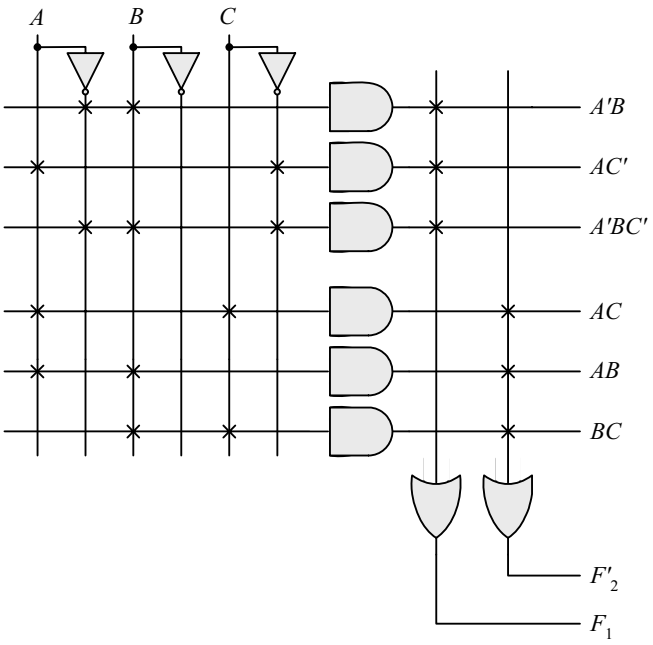
The results of Prob. 6.17 can be used to develop the equations for a three-bit binary counter with D-type flip-flops.

$DA_0 = A'_0$
$DA_1 = A'_1A_0 + A_1A'_0$
$DA_2 = A'_2\,A_1A_0 + A_2A'_1 + A_2A'_0$

$C_{out} = A_2A_1A_0$



7.28



7.29

	Product term	Inputs $x\ y\ A$	Output D_A
$x'y'A$	1	0 0 1	1
$x'yA'$	2	0 1 0	1
$xy'A'$	3	1 0 0	1
xyA	4	1 1 1	1

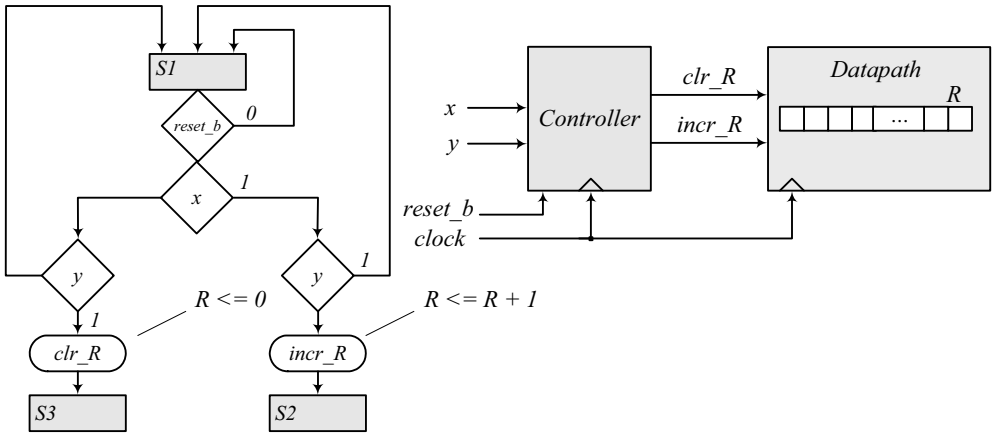
CHAPTER 8

8.1 **(a)** The transfer and increment occur concurrently, i.e., at the same clock edge. After the transfer, *R2* holds the contents that were in *R1* before the clock edge, and *R2* holds its previous value incremented by 1.

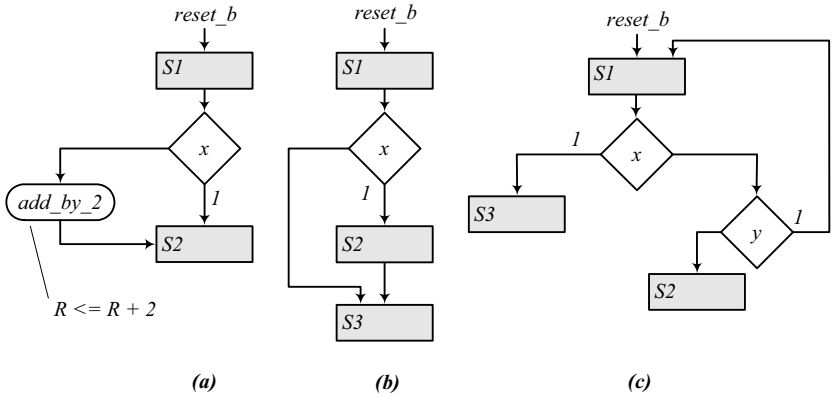
(b) Decrement the content of *R3* by one.

(c) If (S_1 = 1), transfer content of *R1* to *R0*. If (S_1 = 0 and S_2 = 1), transfer content of *R2* to *R0*.

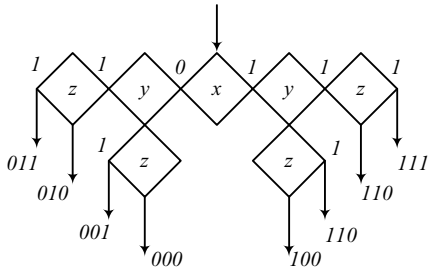
8.2



8.3



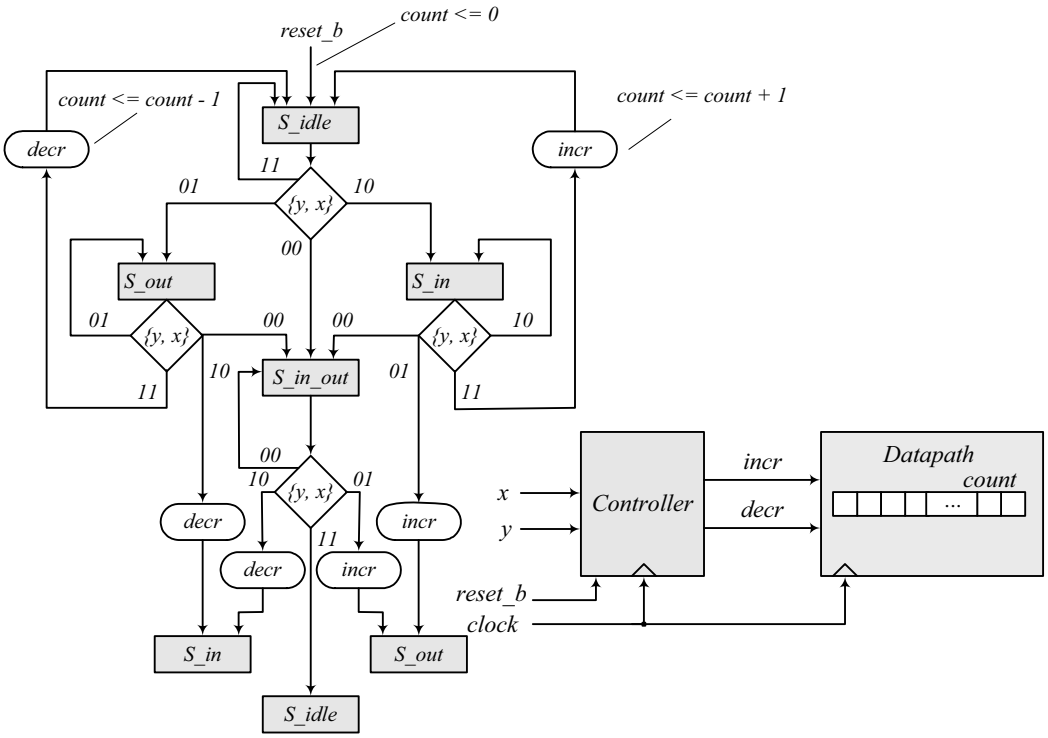
8.4



8.5 The operations specified in a flowchart are executed sequentially, one at a time. The operations specified in an ASM chart are executed concurrently for each ASM block. Thus, the operations listed within a state box, the operations specified by a conditional box, and the transfer to the next state in each ASM block are executed at the same clock edge. For example, in Fig. 8.5 with *Start* = 1 and Flag = 1, signal *Flush_R* is asserted. At the clock edge the state moves to *S_2*, and register *R* is flushed.

8.6

Note: In practice, the asynchronous inputs x and y should be synchronized to the clock to avoid metastable conditons in the flip-flops..



Note: To avoid counting a person more than once, the machine waits until x or y is de-asserted before incrementing or decrementing the counter. The machine also accounts for persons entering and leaving simultaneously.

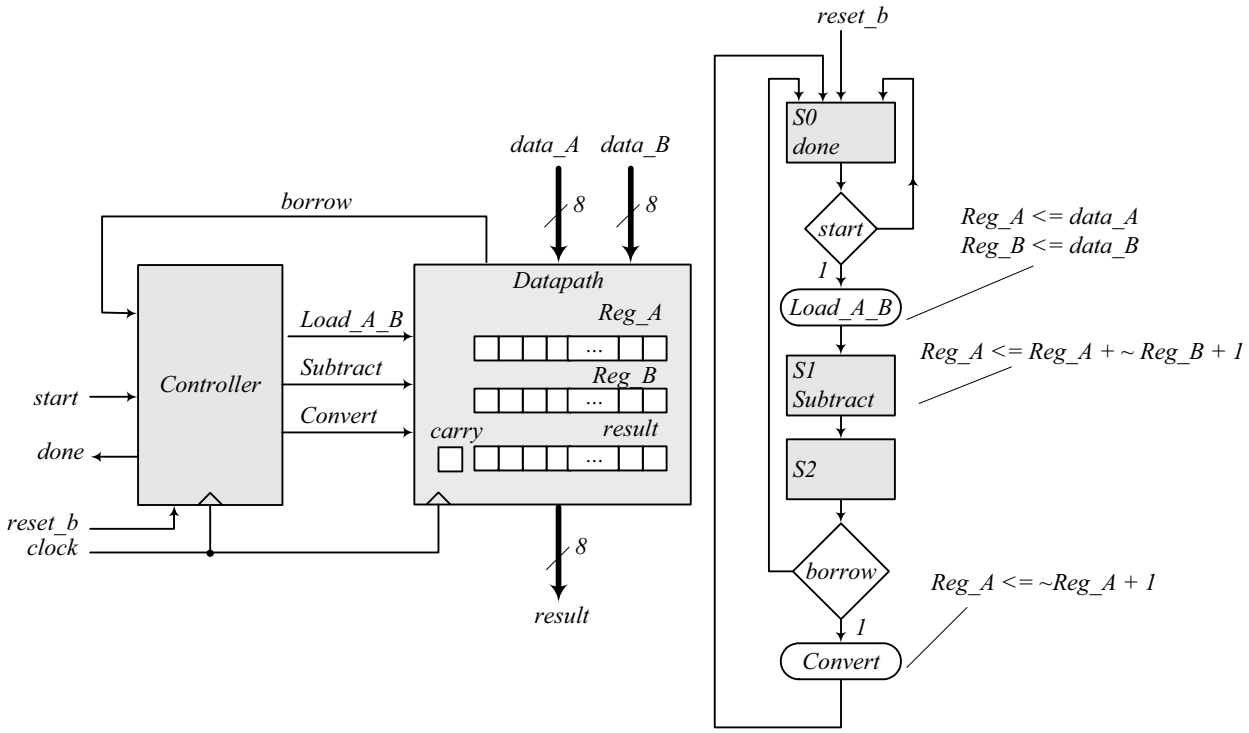
8.7 RTL notation:

S0: Initial state: if (start = 1) then (*RA* ← data_*A*, *RB* ← data_*B*, go to *S1*).

S1: {Carry, *RA*} ← *RA* + (2's complement of *RB*), go to *S2*.

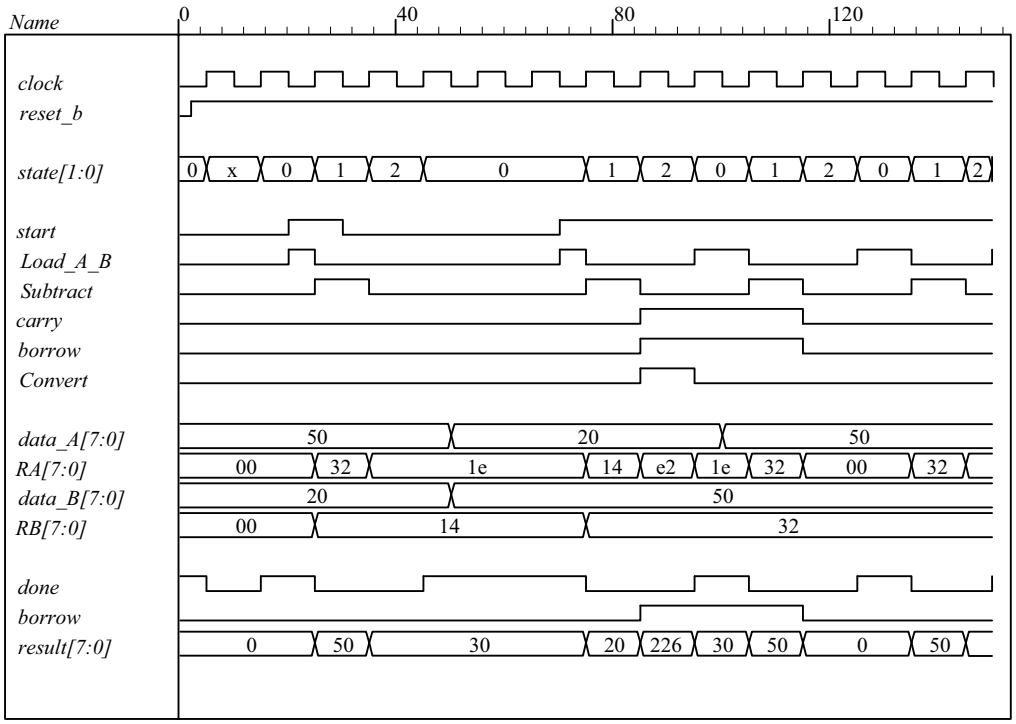
S2: If (borrow = 0) go to *S0*. If (borrow = 1) then *RA* ← (2's complement of *RA*), go to *S0*.

Block diagram and ASMD chart:

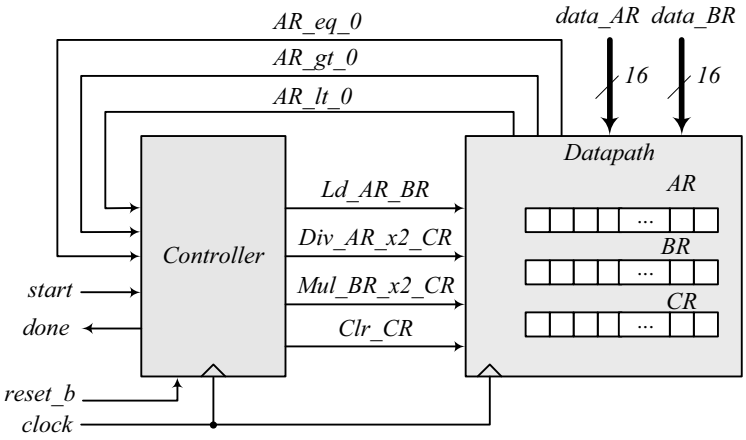


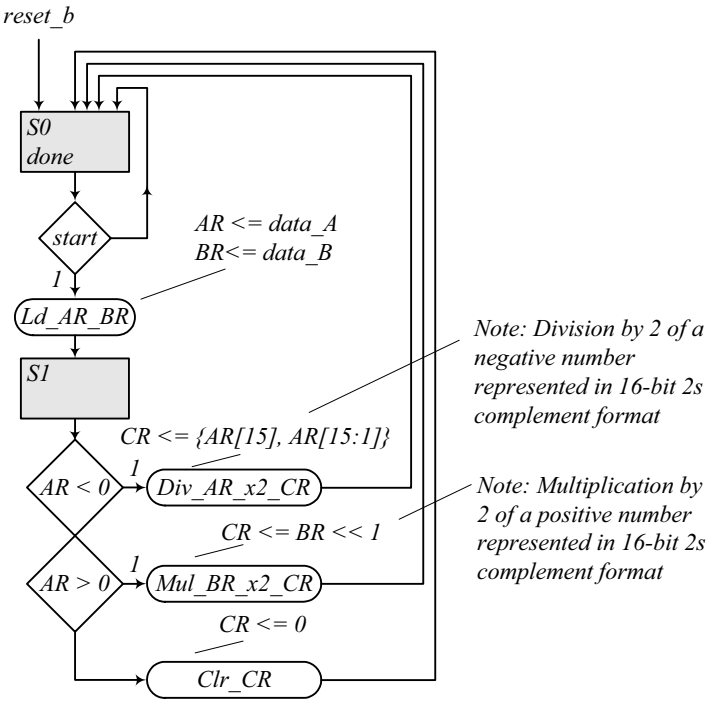

```
module Subtractor_P8_7
 (output done, output [7:0] result, input [7: 0] data_A, data_B, input  start, clock, reset_b);

 Controller_P8_7 M0 (Load_A_B, Subtract, Convert, done, start, borrow, clock, reset_b);
 Datapath_P8_7 M1 (result, borrow, data_A, data_B, Load_A_B, Subtract, Convert, clock, reset_b);
endmodule

module Controller_P8_7 (output reg Load_A_B, Subtract, output reg Convert, output done,
 input start, borrow, clock, reset_b);
 parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10;
 reg [1: 0] state, next_state;
 assign done = (state == S0);

 always @ (posedge clock, negedge reset_b)
  if (!reset_b) state <= S0; else state <= next_state;

 always @ (state, start, borrow) begin
  Load_A_B = 0;
  Subtract = 0;
  Convert = 0;

  case (state)
   S0:       if (start) begin Load_A_B = 1; next_state = S1; end
   S1:       begin  Subtract = 1; next_state = S2; end
   S2:        begin next_state = S0; if (borrow) Convert = 1; end
   default: next_state = S0;
  endcase
 end
endmodule

module Datapath_P8_7 (output [7: 0] result, output borrow, input [7: 0] data_A, data_B,
 input Load_A_B, Subtract, Convert, clock, reset_b);
 reg      carry;
reg [8:0] diff;
 reg [7: 0]   RA, RB;
```

```
  assign      borrow = carry;
  assign result = RA;

  always @ (posedge clock, negedge reset_b)
    if (!reset_b) begin carry <= 1'b0; RA <= 8'b0000_0000; RB <= 8'b0000_0000; end
    else begin
      if (Load_A_B) begin RA <= data_A; RB <= data_B; end
      else if (Subtract) {carry, RA} <= RA +  ~RB + 1;

      // In the statement above, the math of the LHS is done to the wordlength of the LHS
      // The statement below is more explicit about how the math for subtraction is done:
      // else if (Subtract) {carry, RA} <= {1'b0, RA} + {1'b1, ~RB } + 9'b0000_0001;
      // If the 9-th bit is not considered, the 2s complement operation will generate a carry bit,
      //  and borrow must be formed as borrow = ~carry.

      else if (Convert) RA <= ~RA + 8'b0000_0001;
    end
endmodule

// Test plan – Verify;
// Power-up reset
// Subtraction with data_A > data_B
// Subtraction with data_A < data_B
// Subtraction with data_A = data_B
// Reset on-the-fly: left as an exercise

module t_Subtractor_P8_7;
  wire          done;
  wire   [7:0]  result;
  reg    [7: 0] data_A, data_B;
  reg           start, clock, reset_b;

  Subtractor_P8_7 M0 (done, result, data_A, data_B, start, clock, reset_b);

  initial #200 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
     reset_b = 0;
      #2 reset_b = 1;
    #90 reset_b = 1;
    #92 reset_b = 1;
  join

  initial fork
    #20 start = 1;
    #30 start = 0;
    #70 start = 1;
    #110 start = 1;
  join

  initial fork
    data_A = 8'd50;
    data_B = 8'd20;

    #50 data_A = 8'd20;
    #50 data_B = 8'd50;

    #100 data_A = 8'd50;
    #100 data_B = 8'd50;
  join
endmodule
```



8.8 RTL notation:
S0: if (start = 1) *AR* ← input data, *BR* ← input data, go to *S1*.
S1: if (AR [15]) = 1 (sign bit negative) then *CR* ← *AR*(shifted right, sign extension).
else if (positive non-zero) then (Overflow ← *BR*([15] ⊕ [14]), *CR* ← *BR*(shifted left)
else if (*AR* = 0) then (CR ← 0).

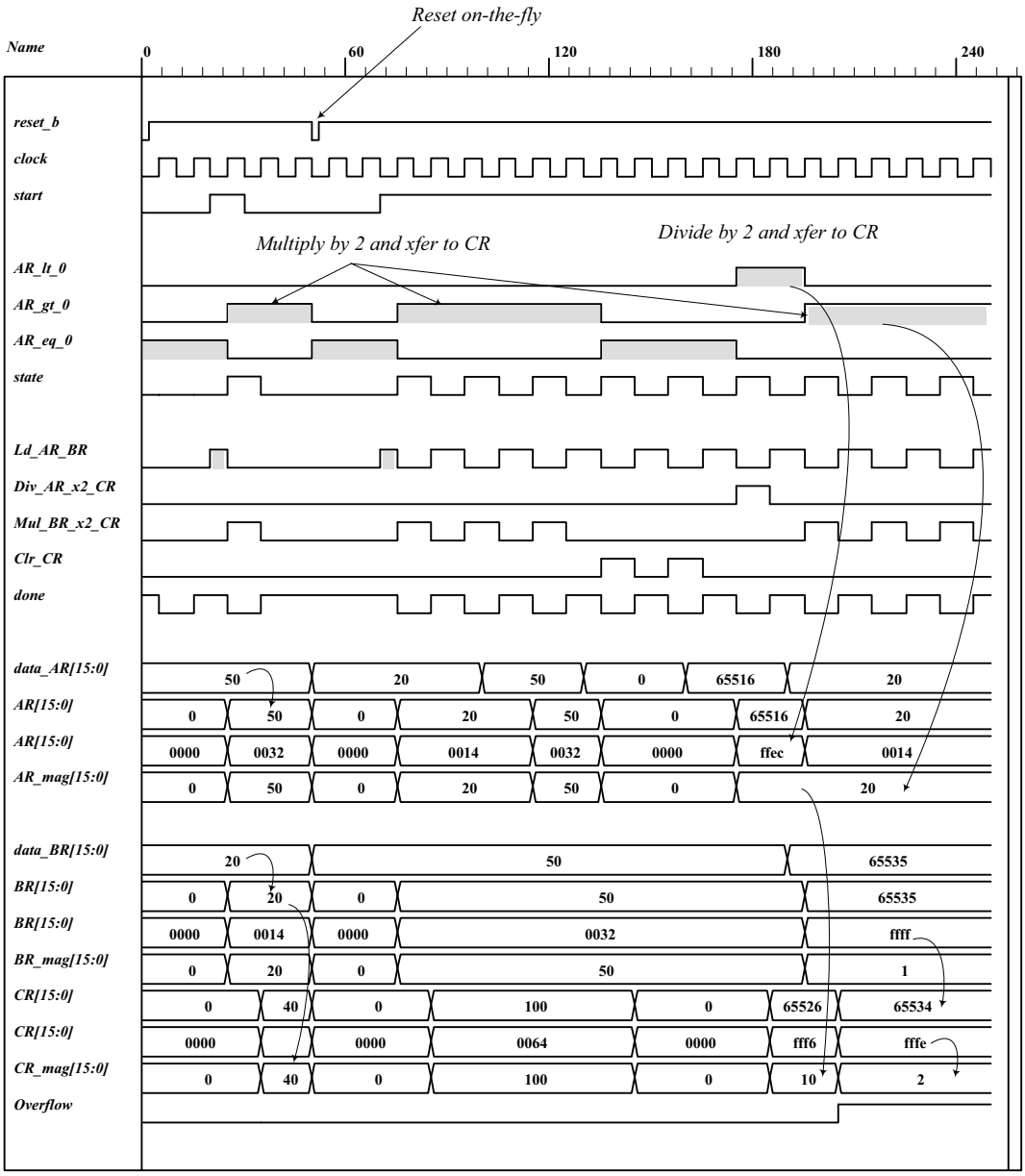





```
module Prob_8_8 (output done, input [15: 0] data_AR, data_BR, input  start, clock, reset_b);

  Controller_P8_8 M0 (
    Ld_AR_BR, Div_AR_x2_CR, Mul_BR_x2_CR, Clr_CR, done,
    start, AR_lt_0, AR_gt_0, AR_eq_0, clock, reset_b
  );


  Datapath_P8_8 M1 (
    Overflow, AR_lt_0, AR_gt_0, AR_eq_0, data_AR, data_BR,
    Ld_AR_BR, Div_AR_x2_CR, Mul_BR_x2_CR, Clr_CR, clock, reset_b
  );
endmodule
```

```
module Controller_P8_8 (
  output reg Ld_AR_BR, Div_AR_x2_CR, Mul_BR_x2_CR, Clr_CR,
  output done, input start, AR_lt_0, AR_gt_0, AR_eq_0, clock, reset_b
);
  parameter  S0 = 1'b0, S1 = 1'b1;
  reg state, next_state;
  assign done = (state == S0);

  always @ (posedge clock, negedge reset_b)
    if (!reset_b) state <= S0; else state <= next_state;

  always @ (state, start, AR_lt_0, AR_gt_0, AR_eq_0) begin
    Ld_AR_BR = 0;
    Div_AR_x2_CR = 0;
    Mul_BR_x2_CR = 0;
    Clr_CR = 0;

    case (state)
      S0:       if (start) begin Ld_AR_BR = 1; next_state = S1; end
      S1:       begin
                  next_state = S0;
                   if (AR_lt_0) Div_AR_x2_CR = 1;
                  else if (AR_gt_0) Mul_BR_x2_CR = 1;
                  else if (AR_eq_0) Clr_CR = 1;
                end
      default: next_state = S0;
    endcase
  end
endmodule

module Datapath_P8_8 (
  output reg Overflow, output AR_lt_0, AR_gt_0, AR_eq_0, input [15: 0] data_AR, data_BR,
  input Ld_AR_BR, Div_AR_x2_CR, Mul_BR_x2_CR, Clr_CR, clock, reset_b
);
  reg [15: 0]  AR, BR, CR;
  assign       AR_lt_0 = AR[15];
  assign       AR_gt_0 = (!AR[15]) && (| AR[14:0]);          // Reduction-OR
  assign       AR_eq_0 = (AR == 16'b0);


  always @ (posedge clock, negedge reset_b)
    if (!reset_b) begin AR <= 8'b0; BR <= 8'b0; CR <= 16'b0; end
    else begin
      if (Ld_AR_BR) begin AR <= data_AR; BR <= data_BR; end
      else if (Div_AR_x2_CR) CR <= {AR[15], AR[15:1]};   // For compiler without arithmetic right shift
      else if (Mul_BR_x2_CR) {Overflow, CR} <= (BR << 1);
      else if (Clr_CR) CR <= 16'b0;
    end
 endmodule

// Test plan – Verify;
// Power-up reset
// If AR < 0 divide AR by 2 and transfer to CR
// If AR > 0 multiply AR by 2 and transfer to CR
// If AR = 0 clear CR
// Reset on-the-fly
```

```
module t_Prob_P8_8;
  wire        done;
  reg [15: 0] data_AR, data_BR;
  reg         start, clock, reset_b;
  reg [15: 0] AR_mag, BR_mag, CR_mag;    // To illustrate 2s complement math

// Probes for displaying magnitude of numbers
  always @ (M0.M1.AR)              // Hierarchical dereferencing
    if (M0.M1.AR[15]) AR_mag = ~M0.M1.AR+ 16'd1; else AR_mag = M0.M1.AR;
  always @ (M0.M1.BR )
    if (M0.M1.BR[15]) BR_mag = ~M0.M1.BR+ 16'd1; else BR_mag = M0.M1.BR;
  always @ (M0.M1.CR)
    if (M0.M1.CR[15]) CR_mag = ~M0.M1.CR + 16'd1; else CR_mag = M0.M1.CR;

  Prob_8_8 M0 (done, data_AR, data_BR, start, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
         reset_b = 0;       // Power-up reset
     #2 reset_b = 1;
    #50 reset_b = 0;  // Reset on-the-fly
    #52 reset_b = 1;
    #90 reset_b = 1;
    #92 reset_b = 1;
  join

  initial fork
    #20 start = 1;
    #30 start = 0;
    #70 start = 1;
    #110 start = 1;
  join

  initial fork
    data_AR = 16'd50;         // AR > 0
    data_BR = 16'd20;         // Result should be 40

    #50 data_AR = 16'd20;
    #50 data_BR = 16'd50;   // Result should be 100

    #100 data_AR = 16'd50;
    #100 data_BR = 16'd50;

    #130 data_AR = 16'd0;   // AR = 0, result should clear CR

    #160 data_AR = -16'd20;     // AR < 0, Verilog stores 16-bit 2s complement
    #160 data_BR = 16'd50;// Result should have magnitude10

    #190 data_AR = 16'd20;// AR < 0, Verilog stores 16-bit 2s complement
    #190 data_BR = 16'hffff;// Result should have overflow
  join
endmodule
```



8.9

Design equations:

$D_{S\_idle} = S\_2 + S\_idle\ Start'$

$D_{S\_1} = S\_idle\ Start + S\_1\ (A2\ A3)'$

$D_{S\_2} = A2\ A3\ S\_1$

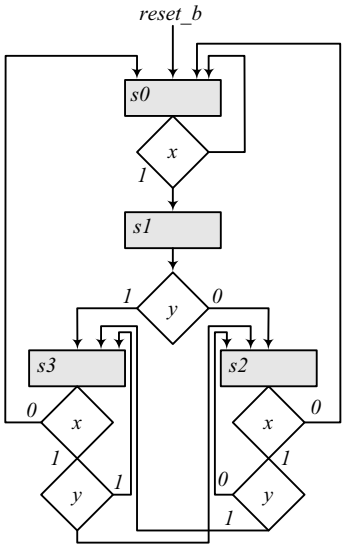
HDL description:

```
module Prob_8_9 (output E, F, output [3: 0] A, output A2, A3,  input  Start, clock, reset_b);

 Controller_Prob_8_9 M0 (set_E, clr_E, set_F, clr_A_F, incr_A, Start, A2, A3, clock, reset_b);
 Datapath_Prob_8_9 M1 (E, F,  A,  A2, A3, set_E, clr_E, set_F, clr_A_F, incr_A, clock, reset_b);

endmodule

// Structural version of the controller (one-hot)
// Note that the flip-flop for S_idle must have a set input and reset_b is wire to the set
// Simulation results match Fig. 8-13

module Controller_Prob_8_9 (
  output   set_E, clr_E, set_F, clr_A_F, incr_A,
  input    Start, A2, A3, clock, reset_b
);

  wire     D_S_idle, D_S_1, D_S_2;
  wire     q_S_idle, q_S_1, q_S_2;
  wire     w0, w1, w2, w3;
  wire [2:0]    state = {q_S_2, q_S_1, q_S_idle};

  // Next-State Logic
  or (D_S_idle, q_S_2, w0);       // input to D-type flip-flop for q_S_idle
  and (w0, q_S_idle, Start_b);
  not (Start_b, Start);

  or (D_S_1, w1, w2, w3);         // input to D-type flip-flop for q_S_1
  and (w1, q_S_idle, Start);
  and (w2, q_S_1, A2_b);
  not (A2_b, A2);
  and (w3, q_S_1, A2, A3_b);
  not (A3_b, A3);

  and (D_S_2, A2, A3, q_S_1);       // input to D-type flip-flop for q_S_2

  D_flop_S M0 (q_S_idle, D_S_idle, clock, reset_b);
  D_flop M1 (q_S_1, D_S_1, clock, reset_b);
  D_flop M2 (q_S_2, D_S_2, clock, reset_b);

  // Output Logic
  and (set_E, q_S_1, A2);
  and (clr_E,  q_S_1, A2_b);
  buf (set_F, q_S_2);
  and (clr_A_F, q_S_idle, Start);
  buf (incr_A, q_S_1);
endmodule

module D_flop (output reg q, input data, clock, reset_b);
  always @ (posedge clock, negedge reset_b)
    if (!reset_b) q <= 1'b0; else q <= data;
endmodule
```

```
module D_flop_S (output reg q, input data, clock, set_b);
  always @ (posedge clock, negedge set_b)
    if (!set_b) q <= 1'b1; else q <= data;
endmodule

/*
// RTL Version of the controller
// Simulation results match Fig. 8-13

module Controller_Prob_8_9 (
  output reg   set_E, clr_E, set_F, clr_A_F, incr_A,
  input        Start, A2, A3, clock, reset_b
);
  parameter   S_idle = 3'b001, S_1 = 3'b010, S_2 = 3'b100;       // One-hot
  reg [2: 0] state, next_state;

  always @ (posedge clock, negedge reset_b)
    if (!reset_b) state <= S_idle; else state <= next_state;

  always @ (state, Start, A2, A3) begin
    set_E    = 1'b0;
    clr_E    = 1'b0;
    set_F    = 1'b0;
    clr_A_F      = 1'b0;
    incr_A   = 1'b0;
    case (state)
      S_idle:     if (Start) begin next_state = S_1; clr_A_F = 1; end
                    else next_state = S_idle;

      S_1:    begin
                      incr_A = 1;
                      if (!A2) begin next_state = S_1; clr_E = 1; end
                      else begin
                        set_E = 1;
                        if (A3) next_state = S_2; else next_state = S_1;
                      end
                    end
      S_2:    begin next_state = S_idle; set_F = 1; end
      default:    next_state = S_idle;
    endcase
  end
endmodule
*/
module Datapath_Prob_8_9 (
  output reg E, F, output reg [3: 0] A, output A2, A3,
  input set_E, clr_E, set_F, clr_A_F, incr_A, clock, reset_b
);
  assign A2 = A[2];
  assign A3 = A[3];

  always @ (posedge clock, negedge reset_b) begin
    if (!reset_b) begin E <= 0; F <= 0; A <= 0; end
    else begin
      if (set_E)  E <= 1;
      if (clr_E)   E <= 0;
      if (set_F)  F <= 1;
      if (clr_A_F)     begin A <= 0; F <= 0; end
      if (incr_A) A <= A + 1;
    end
  end
endmodule
```

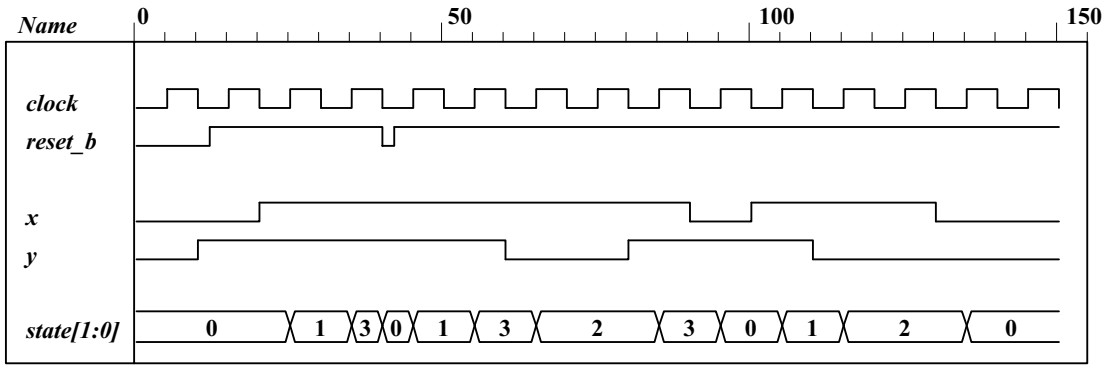
```
// Test Plan - Verify: (1) Power-up reset, (2) match ASMD chart in Fig. 8-9 (d),
// (3) recover from reset on-the-fly

module t_Prob_8_9;
  wire E, F;
  wire [3: 0] A;
  wire A2, A3;
  reg Start, clock, reset_b;

  Prob_8_9 M0 (E, F, A, A2, A3, Start, clock, reset_b);

  initial #500 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    #20 Start = 1;
    #40 reset_b = 0;
    #62 reset_b = 1;
  join
endmodule
```

8.10




```
module Prob_8_10 (input x, y, clock, reset_b);
  reg [ 1: 0]     state, next_state;
  parameter    s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= s0; else state <= next_state;

  always @ (state, x, y) begin
    next_state = s0;
    case (state)
      s0: if (x == 0) next_state = s0; else next_state = s1;
      s1: if (y == 0) next_state = s2; else next_state = s3;
      s2: if (x == 0) next_state = s0; else if (y == 0) next_state = s2; else next_state = s3;
      s3: if (x == 0) next_state = s0; else if (y == 0) next_state = s2; else next_state = s3;
    endcase
  end
endmodule
```

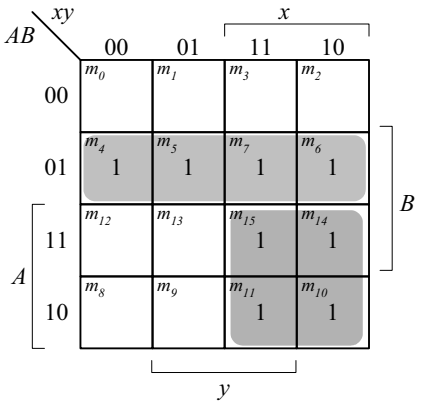
```
module t_Prob_8_10 ();
  reg x, y, clock, reset_b;

  Prob_8_10 M0 (x, y, clock, reset_b);

  initial #150 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
    reset_b = 0;
    #12 reset_b = 1;
     x = 0; y = 0;          // Remain in s0
    #10 y = 1;              // Remain in s0
    #20 x = 1;              // Go to s1 to s3
    #40 reset_b = 0;        // Go to s0
    #42 reset_b = 1;        // Go to s2 to s3
    #60 y = 0;              // Go to s2
    #80 y = 1;              // Go to s3
    #90  x = 0;             // Go to s0
    #100 x = 1;             // Go to s1
    #110 y = 0;             // Go to s2
    #130     x = 0;         // Go to s0
  join
endmodule
```



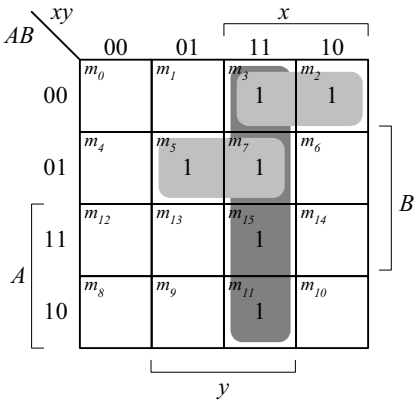
8.11 $D_A = A'B + Ax$

$D_B = A'B'x + A'By + xy$

state		inputs		next state	
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	1	1

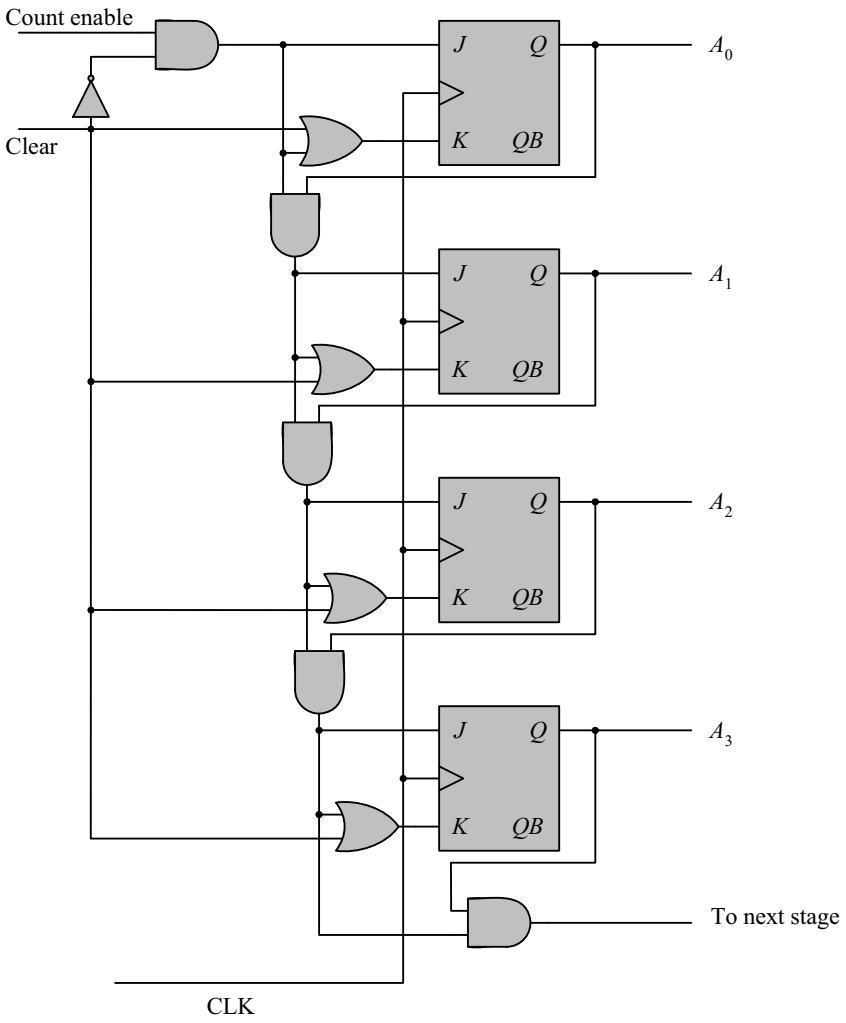


$D_A = A'B + Ax$



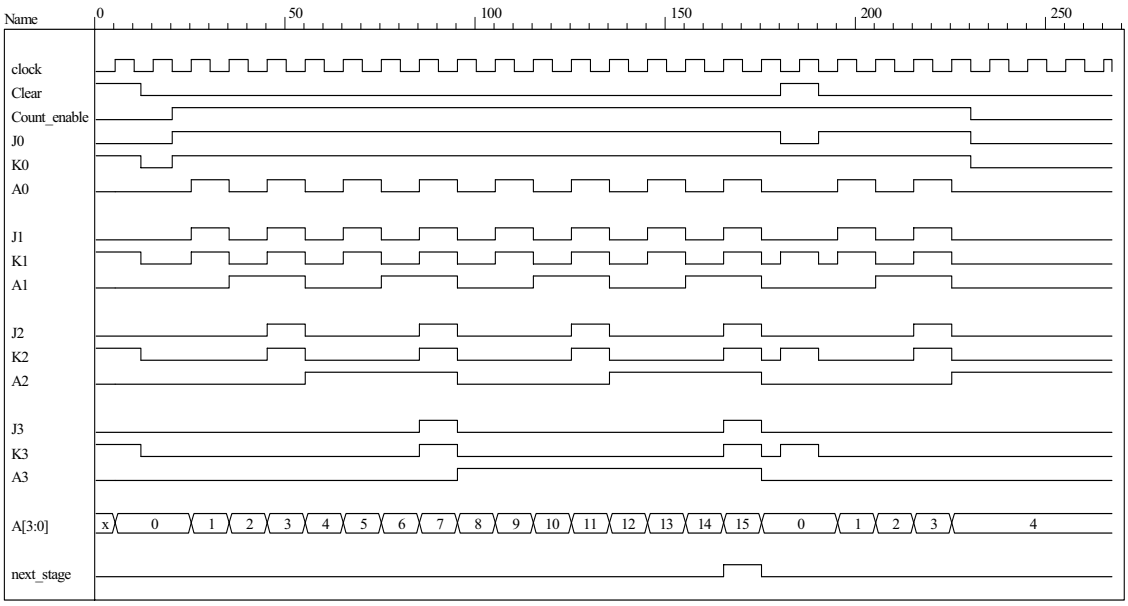
$D_B = A'B'x + A'By + xy$

8.12 Modify the counter in Fig. 6.12 to add a signal, Clear, to clear the counter synchronously, as shown in the circuit diagram below.




```
module Counter_4bit_Synch_Clr (output [3: 0]  A, output next_stage, input Count_enable, Clear, CLK);
  wire A0, A1, A2, A3;
  assign A[3: 0] = {A3, A2, A1, A0};
  JK_FF M0 (A0, J0, K0, CLK);
  JK_FF M1 (A1, J1, K1, CLK);
  JK_FF M2 (A2, J2, K2, CLK);
  JK_FF M3 (A3, J3, K3, CLK);

  not (Clear_b, Clear);
  and (J0, Count_enable, Clear_b);
  and (J1, J0, A0);
  and (J2, J1, A1);
  and (J3, J2, A2);

  or (K0, Clear, J0);
  or (K1, Clear, J1);
  or (K2, Clear, J2);
  or (K3, Clear, J3);

  and (next_stage, A3, J3);

endmodule
```

```
module JK_FF (output reg Q, input J, K, clock);
  always @ (posedge clock)
    case ({J,K})
      2'b00:  Q <= Q;
      2'b01:  Q <= 0;
      2'b10:  Q <= 1;
      2'b11:  Q <= ~Q;
    endcase
endmodule

module t_Counter_4bit_Synch_Clr ();
  wire [3: 0]  A;
  wire next_stage;
   reg Count_enable, Clear, clock;

  Counter_4bit_Synch_Clr M0 (A, next_stage, Count_enable, Clear, clock);

  initial #300 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
    Clear = 1;
    Count_enable = 0;
    #12 Clear = 0;
    #20 Count_enable = 1;
    #180 Clear = 1;
    #190 Clear = 0;
    #230 Count_enable = 0;
  join
endmodule
```



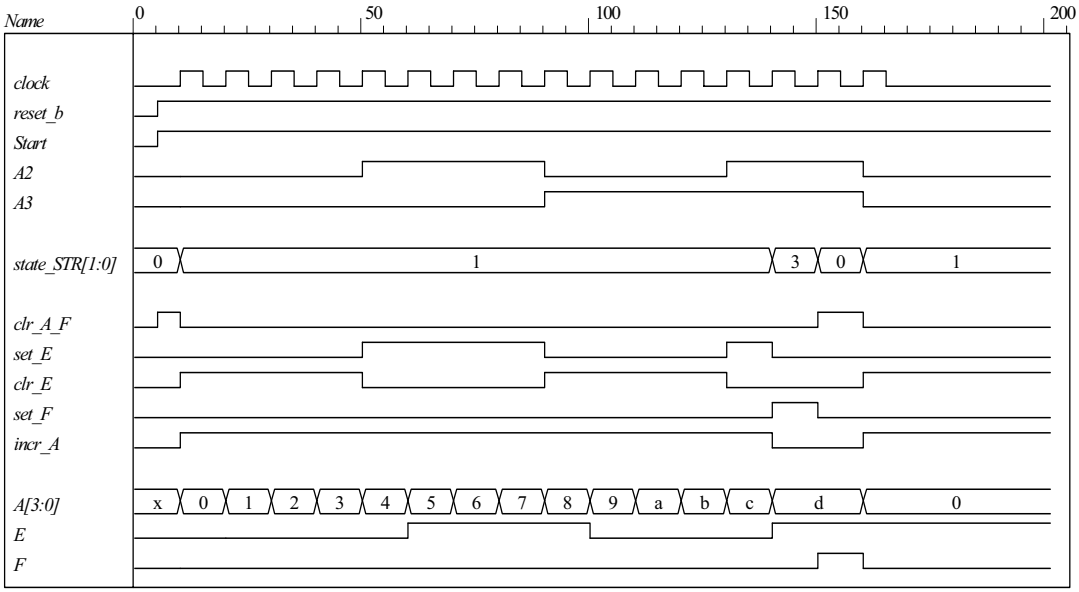
8.13

```
// Structural description of design example (Fig. 8-10, 8-12)
module Design_Example_STR

( output  [3:0]   A,
  output          E, F,
  input           Start, clock, reset_b
);
```

```
  Controller_STR M0 (clr_A_F, set_E, clr_E, set_F, incr_A, Start, A[2], A[3], clock, reset_b );
  Datapath_STR M1 (A, E, F, clr_A_F, set_E, clr_E, set_F, incr_A, clock);
endmodule

module Controller_STR
( output clr_A_F, set_E, clr_E, set_F, incr_A,
   input Start, A2, A3, clock, reset_b
);

  wire       G0, G1;
  parameter   S_idle = 2'b00, S_1 = 2'b01, S_2 = 2'b11;
  wire       w1, w2, w3;

  not (G0_b, G0);
  not (G1_b, G1);
  buf (incr_A, w2);
  buf (set_F, G1);
  not (A2_b, A2);
  or (D_G0, w1, w2);
  and (w1, Start, G0_b);
  and (clr_A_F, G0_b, Start);
  and (w2, G0, G1_b);
  and (set_E, w2, A2);
  and (clr_E, w2, A2_b);
  and (D_G1, w3, w2);
  and (w3, A2, A3);
  D_flip_flop_AR M0 (G0, D_G0, clock, reset_b);
  D_flip_flop_AR M1 (G1, D_G1, clock, reset_b);
endmodule

// datapath unit

module Datapath_STR
( output [3: 0] A,
  output    E, F,
  input      clr_A_F, set_E, clr_E, set_F, incr_A, clock
);

  JK_flip_flop_2 M0 (E, E_b, set_E, clr_E, clock);
  JK_flip_flop_2 M1 (F, F_b, set_F, clr_A_F, clock);
  Counter_4 M2 (A, incr_A, clr_A_F, clock);

endmodule

module Counter_4 (output reg [3: 0] A, input incr, clear, clock);
  always @ (posedge clock)
    if (clear)      A <= 0; else if (incr) A <= A + 1;
endmodule

module D_flip_flop_AR (Q, D, CLK, RST);
  output    Q;
  input D, CLK, RST;
  reg    Q;

  always @ (posedge CLK, negedge RST)
    if (RST == 0) Q <= 1'b0;
    else Q <= D;
endmodule

module JK_flip_flop_2 (Q, Q_not, J, K, CLK);
```

```
  output    Q, Q_not;
  input J, K, CLK;
  reg    Q;


  assign    Q_not = ~Q
;
  always @ (posedge CLK)
    case ({J, K})
      2'b00:  Q <= Q;
      2'b01:  Q <= 1'b0;
      2'b10:  Q <= 1'b1;
      2'b11:  Q <= ~Q;
    endcase
 endmodule

module t_Design_Example_STR;
  reg         Start, clock, reset_b;
  wire [3: 0]     A;
  wire        E, F;
  wire [1:0] state_STR = {M0.M0.G1, M0.M0.G0};


  Design_Example_STR M0 (A, E, F, Start, clock, reset_b);

  initial #500 $finish;
  initial
    begin
      reset_b = 0;
      Start = 0;
      clock = 0;
      #5 reset_b = 1; Start = 1;
      repeat (32)
        begin
          #5 clock = ~ clock;
        end
    end
  initial
  $monitor ("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule
```

The simulation results shown below match Fig. 8.13.



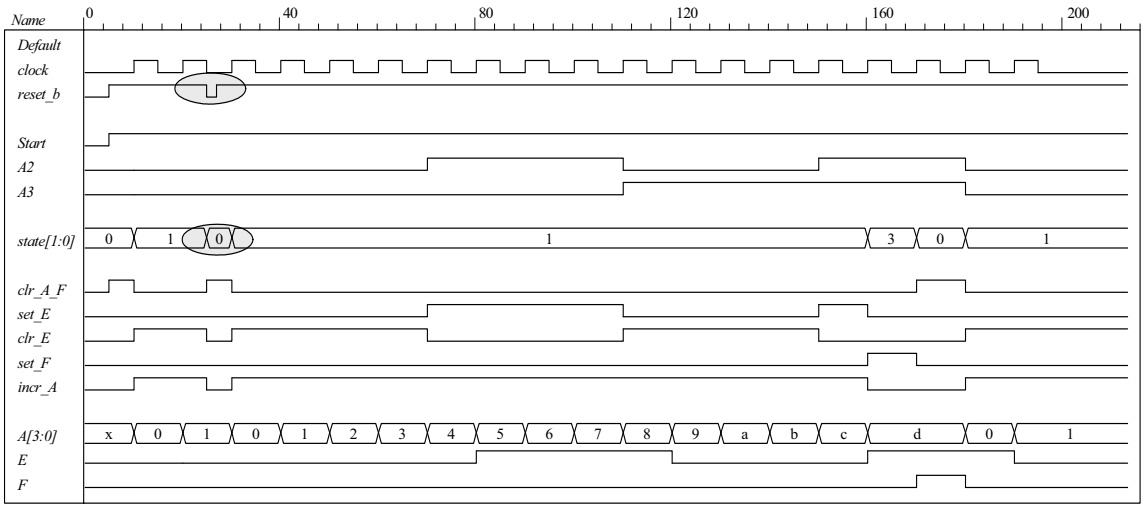
8.14 The state code 2'b10 is unused. If the machine enters an unused state, the controller is written with default assignment to *next_state*. The default assignment forces the state to *S_idle*, so the machine recovers from the condition.

8.15 Modify the test bench to insert a reset event and extend the clock.

```
// RTL description of design example (see Fig.8-11)

module Design_Example_RTL (A, E, F, Start, clock, reset_b);

  // Specify ports of the top-level module of the design
  // See block diagram Fig. 8-10

  output [3: 0]     A;
  output            E, F;
  input             Start, clock, reset_b;

  // Instantiate controller and datapath units

  Controller_RTL M0 (set_E, clr_E, set_F, clr_A_F, incr_A, A[2], A[3], Start, clock, reset_b );
  Datapath_RTL M1 (A, E, F, set_E, clr_E, set_F, clr_A_F, incr_A, clock);

endmodule

module Controller_RTL (set_E, clr_E, set_F, clr_A_F, incr_A, A2, A3, Start, clock, reset_b);
  output reg   set_E, clr_E, set_F, clr_A_F, incr_A;
  input      Start, A2, A3, clock, reset_b;
  reg [1:0]  state, next_state;
  parameter    S_idle = 2'b00, S_1 = 2'b01, S_2 = 2'b11;   // State codes

  always @ (posedge clock or negedge reset_b)       // State transitions (edge-sensitive)
    if (reset_b == 0) state <= S_idle;
    else state <= next_state;

  // Code next state logic directly from ASMD chart (Fig. 8-9d)

  always @ (state, Start, A2, A3 ) begin          // Next state logic (level-sensitive)
    next_state = S_idle;
```

```
  case (state)
     S_idle:   if (Start) next_state = S_1; else next_state = S_idle;
     S_1:      if (A2 & A3) next_state = S_2; else next_state = S_1;
     S_2:      next_state = S_idle;
     default:  next_state = S_idle;
  endcase
 end

 // Code output logic directly from ASMD chart (Fig. 8-9d)

 always @ (state, Start, A2) begin
  set_E = 0;       // default assignments; assign by exception
  clr_E = 0;
  set_F = 0;
  clr_A_F = 0;
  incr_A = 0;
  case (state)
   S_idle:         if (Start) clr_A_F = 1;
   S_1:       begin incr_A = 1; if (A2) set_E = 1; else clr_E = 1; end
   S_2:       set_F = 1;
  endcase
 end
endmodule

module  Datapath_RTL  (A, E, F, set_E, clr_E, set_F, clr_A_F, incr_A, clock);
 output reg [3: 0]     A;          // register for counter
 output reg        E, F;           // flags
 input             set_E, clr_E, set_F, clr_A_F, incr_A, clock;

 // Code register transfer operations directly from ASMD chart (Fig. 8-9d)

 always @ (posedge clock) begin
  if (set_E)               E <= 1;
  if (clr_E)               E <= 0;
  if (set_F)               F <= 1;
  if (clr_A_F)             begin A <= 0; F <= 0; end
  if (incr_A)              A <= A + 1;
 end
endmodule

module t_Design_Example_RTL;
 reg          Start, clock, reset_b;
 wire [3: 0]  A;
 wire         E, F;

 // Instantiate design example

 Design_Example_RTL M0 (A, E, F, Start, clock, reset_b);

 // Describe stimulus waveforms

 initial #500 $finish;     // Stopwatch
 initial fork
    #25 reset_b = 0;       // Test for recovery from reset on-the-fly.
    #27 reset_b = 1;
 join
 initial
  begin
    reset_b = 0;
    Start = 0;
    clock = 0;
```

```
    #5 reset_b = 1; Start = 1;
    //repeat (32)
    repeat (38)                  // Modify for test of reset_b on-the-fly
      begin
        #5 clock = ~ clock;      // Clock generator
      end
  end
 initial
   $monitor ("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule
```

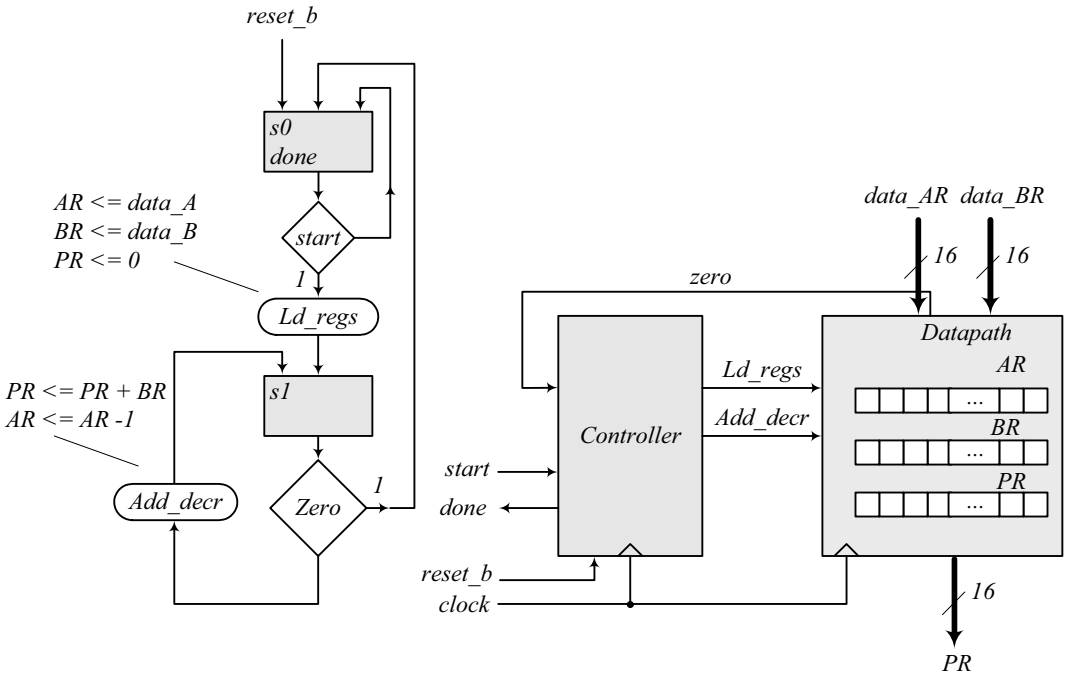


8.16 RTL notation:

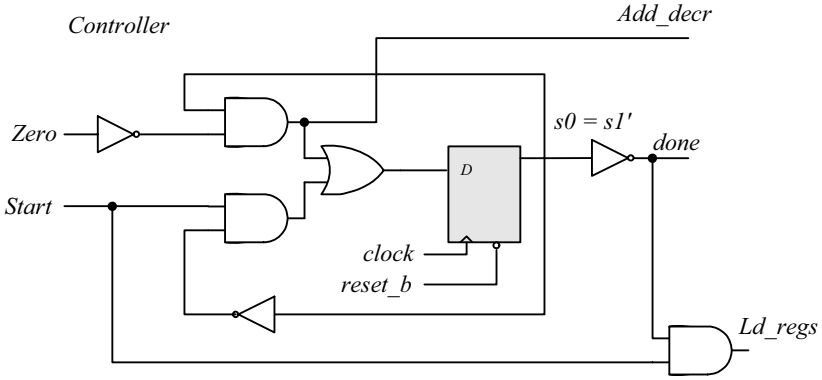
s0: (initial state) If *start* = 0 go back to state *s0*, If (*start* = 1) then *BR* ← *multiplicand*, *AR* ← *multiplier*, *PR* ← 0, go to *s1*.

s1: (check *AR* for Zero) *Zero* = 1 if *AR* = 0, if (*Zero* = 1) then go back to *s0* (*done*) If (*Zero* = 0) then go to *s1*, *PR* ← *PR* + *BR*, *AR* ← *AR* – 1.

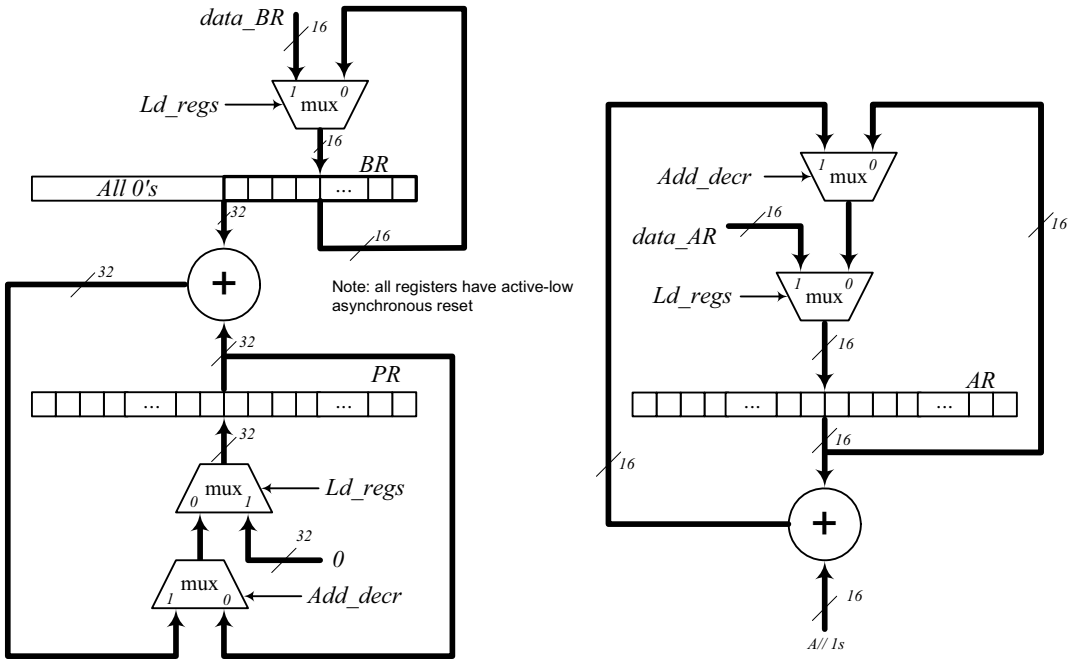
The internal architecture of the datapath consists of a double-width register to hold the product (*PR*), a register to hold the multiplier (*AR*), a register to hold the multiplicand (*BR*), a double-width parallel adder, and single-width parallel adder. The single-width adder is used to implement the operation of decrementing the multiplier unit. Adding a word consisting entirely of 1s to the multiplier accomplishes the 2's complement subtraction of 1 from the multiplier. Figure 8.16 (a) below shows the ASMD chart, block diagram, and controller of the circuit. Figure 8.16 (b) shows the internal architecture of the datapath. Figure 8.16 (c) shows the results of simulating the circuit.



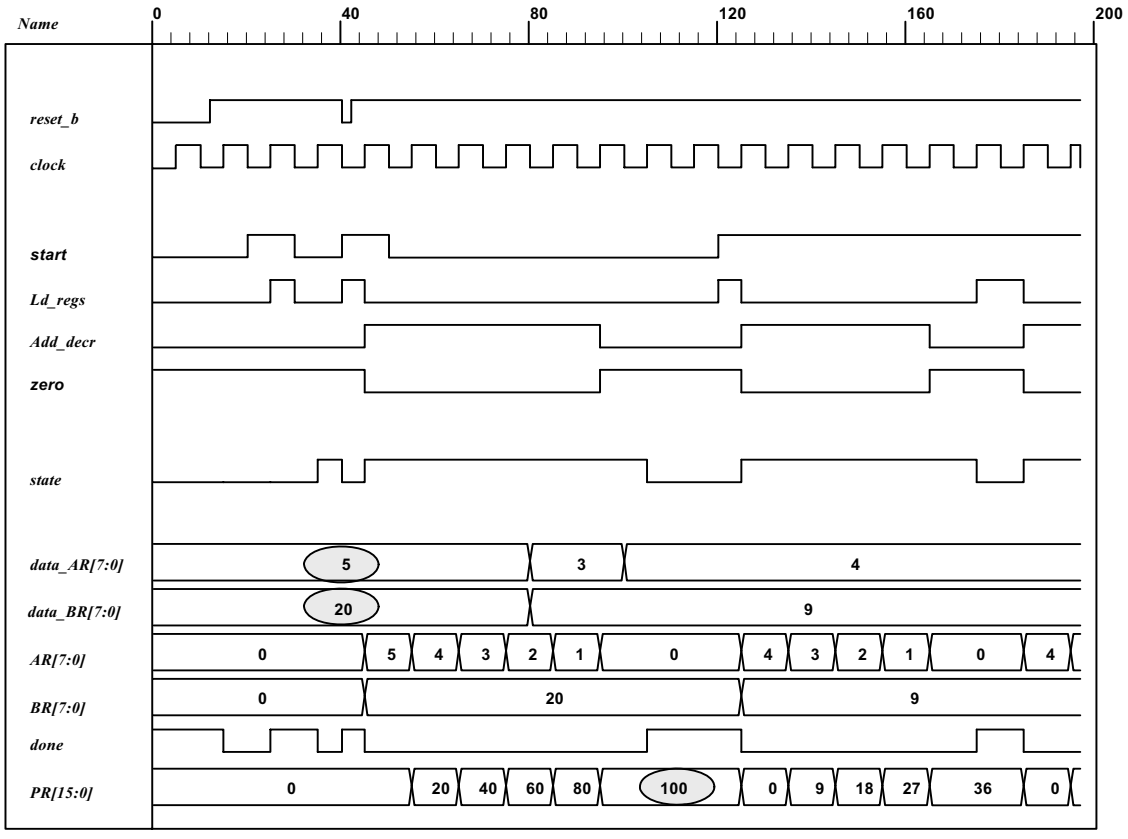
Note: Form Zero as the output of an OR gate whose inputs are the bits of the register AR.



(a) ASMD chart, block diagram, and controller



(b) Datapath



(c) Simulation results

```
module Prob_8_16_STR (
output [15: 0] PR, output done,
input [7: 0] data_AR, data_BR, input  start, clock, reset_b
);

Controller_P8_16 M0 (done, Ld_regs, Add_decr, start, zero, clock, reset_b);

 Datapath_P8_16 M1 (PR, zero, data_AR, data_BR, Ld_regs, Add_decr, clock, reset_b);
endmodule

module Controller_P8_16 (output done, output reg Ld_regs, Add_decr, input start, zero, clock, reset_b);
parameter  s0 = 1'b0, s1 = 1'b1;
reg state, next_state;
assign done = (state == s0);

always @ (posedge clock, negedge reset_b)
if (!reset_b) state <= s0; else state <= next_state;

always @ (state, start, zero) begin
Ld_regs = 0;
Add_decr = 0;
case (state)
s0:      if (start) begin Ld_regs = 1; next_state = s1; end
s1:      if (zero) next_state = s0; else  begin next_state = s1; Add_decr = 1; end
default:    next_state = s0;
endcase
end
endmodule

module Register_32 (output [31: 0] data_out, input [31: 0] data_in, input clock, reset_b);
Register_8 M3 (data_out [31: 24] , data_in  [31: 24], clock, reset_b);
Register_8 M2 (data_out [23: 16] , data_in  [23: 16], clock, reset_b);
Register_8 M1 (data_out [15: 8] , data_in  [15: 8], clock, reset_b);
Register_8 M0 (data_out [7: 0] , data_in  [7: 0], clock, reset_b);
endmodule

module Register_16 (output [15: 0] data_out, input [15: 0] data_in, input clock, reset_b);
Register_8 M1 (data_out [15: 8] , data_in  [15: 8], clock, reset_b);
Register_8 M0 (data_out [7: 0] , data_in  [7: 0], clock, reset_b);
endmodule

module Register_8 (output [7: 0] data_out, input [7: 0] data_in, input clock, reset_b);
D_flop M7 (data_out[7] data_in[7], clock, reset_b);
D_flop M6 (data_out[6] data_in[6], clock, reset_b);
D_flop M5 (data_out[5] data_in[5], clock, reset_b);
D_flop M4 (data_out[4] data_in[4], clock, reset_b);
D_flop M3 (data_out[3] data_in[3], clock, reset_b);
D_flop M2 (data_out[2] data_in[2], clock, reset_b);
D_flop M1 (data_out[1] data_in[1], clock, reset_b);
D_flop M0 (data_out[0] data_in[0], clock, reset_b);
endmodule

module Adder_32 (output c_out, output [31: 0]  sum, input [31: 0] a, b);
assign {c_out, sum} = a + b;
endmodule

module Adder_16 (output c_out, output [15: 0] sum, input [15: 0] a, b);
assign {c_out, sum} = a + b;
endmodule
```

```
module D_flop (output q, input data, clock, reset_b);
always @ (posedge clock, negedge reset_b)
if (!reset_b) q <= 0; else q <= data;
endmodule

module  Datapath_P8_16 (
output reg [15: 0] PR, output zero,
input [7: 0] data_AR, data_BR, input Ld_regs, Add_decr, clock, reset_b
);

reg [7: 0]   AR, BR;
assign      zero = ~( | AR);

always @ (posedge clock, negedge reset_b)
if (!reset_b) begin AR <= 8'b0; BR <= 8'b0; PR <= 16'b0; end
else begin
if (Ld_regs) begin AR <= data_AR; BR <= data_BR; PR <= 0; end
else if (Add_decr) begin PR <= PR + BR; AR <= AR -1; end
end
endmodule


// Test plan – Verify;
// Power-up reset
// Data is loaded correctly
// Control signals assert correctly
// Status signals assert correctly
// start is ignored while multiplying
// Multiplication is correct
// Recovery from reset on-the-fly

module t_Prob_P8_16;
wire         done;
wire [15: 0]     PR;
reg [7: 0]   data_AR, data_BR;
reg          start, clock, reset_b;

Prob_8_16_STR M0 (PR, done, data_AR, data_BR, start, clock, reset_b);

initial #500 $finish;
initial begin clock = 0; forever #5 clock = ~clock; end
initial fork
reset_b = 0;
#12 reset_b = 1;
#40 reset_b = 0;
#42 reset_b = 1;
#90 reset_b = 1;
#92 reset_b = 1;
join

initial fork
#20 start = 1;
#30 start = 0;
#40 start = 1;
#50 start = 0;
#120 start = 1;
#120 start = 0;
join
```

```
initial fork
data_AR = 8'd5;        // AR > 0
data_BR = 8'd20;

#80 data_AR = 8'd3;
#80 data_BR = 8'd9;

#100 data_AR = 8'd4;
#100 data_BR = 8'd9;
join
endmodule
```

8.17 $(2^n - 1)(2^n - 1) < (2^{2n} - 1)$ for $n \geq 1$

8.18 **(a)** The maximum product size is 32 bits available in registers A and Q.
(b) P counter must have 5 bits to load 16 (binary 10000) initially.
(c) Z (zero) detection is generated with a 5-input NOR gate.

8.19

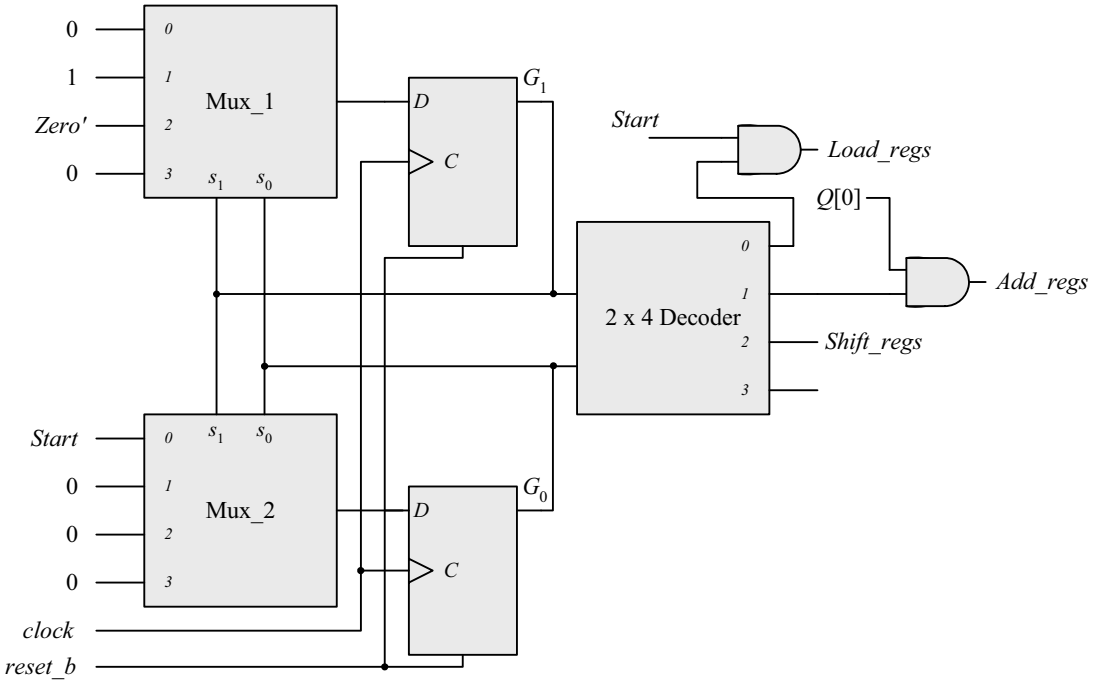
Multiplicand $B = 11011_2 = 27_{10}$
Multiplier $Q = 10111_2 = 23_{10}$
Product: $CAQ = 621_{10}$

	C	***A***	***Q***	***P***
Multiplier in Q	0	00000	10111	101
$Q0 = 1$; add B		11011		
First partial product	0	11011	10111	100
Shift right CAQ	0	01101	11011	
$Q0 = 1$; add B		11011		
Second partial product	1	01000	11011	011
Shift right CAQ	0	10100	01101	
$Q0 = 1$; add B		11011		
Third partial product	1	01111	01101	010
Shift right CAQ	0	10111	10110	
Shift right CAQ	0	01011	11011	
Fourth partial product	0	01011	11011	001
$Q0 = 1$; add B		11011		
Fifth partial product	1	00110	11011	000
Shift right CAQ	0	10011	01101	
Final product in AQ:				
$AQ = 10011\_01101 = 621_{10}$				

8.20 S_idle = 1t ns
The loop between S_add and S_shift takes 2nt ns)
Total time to multiply: $(2n + 1)$t

8.21

State codes:	G_1	G_0
S_idle	0	0
S_add	0	1
*S_shift*1	0	
unused	0	0

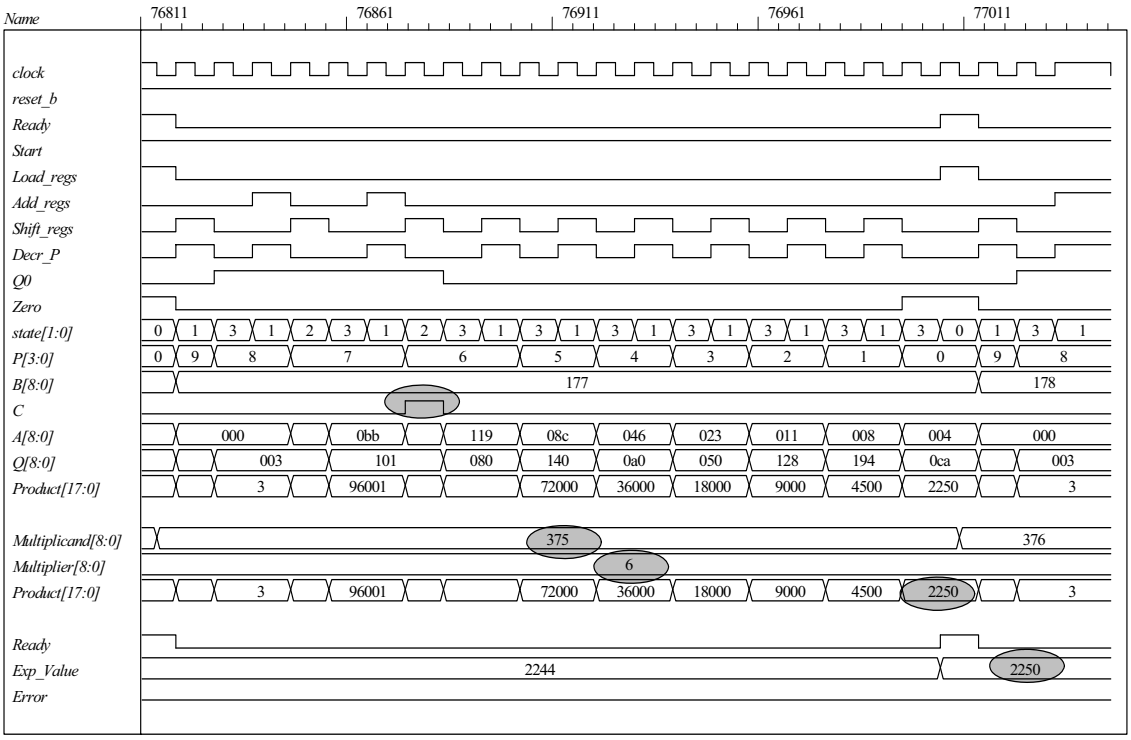


8.22 Note that the machine described by Fig. P8.22 requires four states, but the machine described byFig. 8.15 (b) requires only three. Also, observe that the sample simulation results show a case where the carry bit regsiter, C, is needed to support the addition operation. The datapath is 8 bits wide.

```
module Prob_8_22 # (parameter m_size = 9)
(
  output [2*m_size -1: 0] Product,
  output Ready,
  input [m_size -1: 0] Multiplicand, Multiplier,
  input Start, clock, reset_b
);
  wire [m_size -1: 0] A, Q;

  assign Product = {A, Q};
  wire Q0, Zero, Load_regs, Decr_P, Add_regs, Shift_regs;

Datapath_Unit M0 (A, Q, Q0, Zero, Multiplicand, Multiplier, Load_regs, Decr_P, Add_regs, Shift_regs,
clock, reset_b);
Control_Unit M1 (Ready, Decr_P, Load_regs, Add_regs, Shift_regs, Start, Q0, Zero, clock, reset_b);
endmodule
```

```
module Datapath_Unit # (parameter m_size = 9,  BC_size = 4)
(
  output reg [m_size -1: 0] A, Q,
  output Q0, Zero,
  input [m_size -1: 0] Multiplicand, Multiplier,
  input Load_regs, Decr_P,  Add_regs, Shift_regs, clock, reset_b
);
  reg C;
  reg [BC_size -1: 0] P;
  reg [m_size -1: 0] B;

  assign Q0 = Q[0];
  assign Zero = (P == 0);

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) begin
    B <= 0;C <= 0;
    A <= 0;
    Q <= 0;
    P <= m_size;
  end
  else begin
    if (Load_regs) begin
      A <= 0;
      C <= 0;
      Q <= Multiplier;
      B <= Multiplicand;
      P <= m_size;
    end
    if (Decr_P) P <= P -1;
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
  end
endmodule

module Control_Unit (
  output Ready, Decr_P, output reg Load_regs, Add_regs, Shift_regs, input Start, Q0, Zero, clock,
reset_b
);
  reg [ 1: 0]     state, next_state;
  parameter    S_idle = 2'b00, S_loaded = 2'b01, S_sum = 2'b10, S_shifted = 2'b11;
  assign Ready = (state == S_idle);
  assign Decr_P = (state == S_loaded);

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= S_idle; else state <= next_state;

  always @ (state, Start, Q0, Zero) begin
    next_state = S_idle;
     Load_regs = 0;
    Add_regs = 0;
    Shift_regs = 0;
    case (state)
      S_idle: if (Start == 0) next_state = S_idle; else begin next_state = S_loaded; Load_regs = 1; end
      S_loaded: if (Q0) begin next_state = S_sum; Add_regs = 1; end
      else begin next_state = S_shifted; Shift_regs = 1; end
      S_sum:     begin next_state = S_shifted; Shift_regs = 1; end
      S_shifted: if (Zero) next_state = S_idle; else next_state = S_loaded;
    endcase
  end
endmodule
```

```
module t_Prob_8_22 ();
  parameter                 m_size = 9;             // Width of datapath
  wire  [2 * m_size - 1: 0] Product;
  wire                      Ready;
  reg    [m_size - 1: 0]    Multiplicand, Multiplier;
  reg                       Start, clock, reset_b;
  integer                   Exp_Value;
  reg                       Error;

   Prob_8_22 M0 (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);


  initial #140000 $finish;
  initial begin clock = 0; #5 forever #5 clock = ~clock; end
  initial fork
    reset_b = 1;

    #2 reset_b = 0;
    #3 reset_b = 1;
  join
  initial begin #5 Start = 1; end
    always @ (posedge Ready) begin
    Exp_Value = Multiplier * Multiplicand;
    //Exp_Value = Multiplier * Multiplicand +1;     // Inject error to confirm detection
  end
  always @ (negedge Ready) begin
    Error = (Exp_Value ^ Product) ;
  end

  initial begin
    #5 Multiplicand = 0;
    Multiplier = 0;
    repeat (64) #10 begin Multiplier = Multiplier + 1;
      repeat (64) @ (posedge M0.Ready) #5 Multiplicand = Multiplicand + 1;
    end
  end
endmodule
```

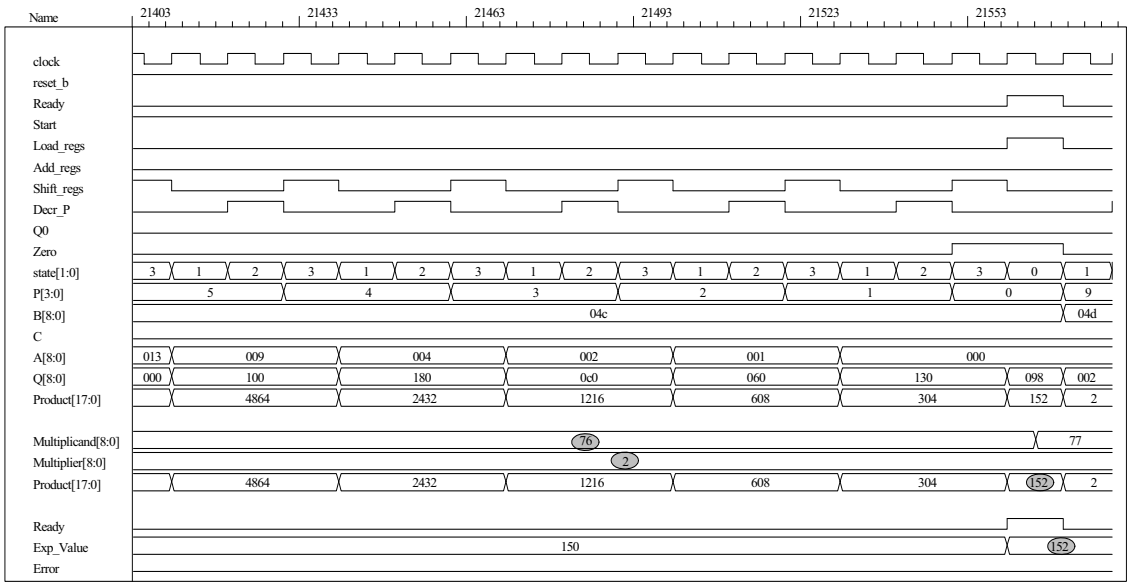


8.23 As shown in Fig. P8.23 the machine asserts *Load_regs* in state *S_load*. This will cause the machine to operate incorrectly. Once *Load_regs* is removed from *S_load* the machine operates correctly. The state *S_load* is a wasted state. Its removal leads to the same machine as dhown in Fig. P8.15b.

```
module Prob_8_23 # (parameter m_size = 9)
(
  output [2*m_size -1: 0] Product,
  output Ready,
  input [m_size -1: 0] Multiplicand, Multiplier,
  input Start, clock, reset_b
);
  wire [m_size -1: 0] A, Q;

  assign Product = {A, Q};
  wire Q0, Zero, Load_regs, Decr_P, Add_regs, Shift_regs;

Datapath_Unit M0 (A, Q, Q0, Zero, Multiplicand, Multiplier, Load_regs, Decr_P, Add_regs, Shift_regs,
clock, reset_b);
Control_Unit M1 (Ready, Decr_P, Shift_regs, Add_regs, Load_regs, Start, Q0, Zero, clock, reset_b);
endmodule

module Datapath_Unit # (parameter m_size = 9,  BC_size = 4)
(
  output reg [m_size -1: 0] A, Q,
  output Q0, Zero,
  input [m_size -1: 0] Multiplicand, Multiplier,
  input Load_regs, Decr_P,  Add_regs, Shift_regs, clock, reset_b
);
  reg C;
  reg [BC_size -1: 0] P;
  reg [m_size -1: 0] B;
```

```
  assign Q0 = Q[0];
  assign Zero = (P == 0);

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) begin
    A <= 0;
    C <= 0;
    Q <= 0;
    B <= 0;
    P <= m_size;
  end
  else begin
    if (Load_regs) begin
      A <= 0;
      C <= 0;
      Q <= Multiplier;
      B <= Multiplicand;
      P <= m_size;
    end
    if (Decr_P) P <= P -1;
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
  end
endmodule

module Control_Unit (
  output Ready, Decr_P, Shift_regs, output reg Add_regs, Load_regs, input Start, Q0, Zero, clock,
reset_b
);
  reg [ 1: 0]     state, next_state;
  parameter    S_idle = 2'b00, S_load = 2'b01, S_decr = 2'b10, S_shift = 2'b11;

  assign Ready = (state == S_idle);
  assign Shift_regs = (state == S_shift);
  assign Decr_P = (state == S_decr);

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= S_idle; else state <= next_state;

  always @ (state, Start, Q0, Zero) begin
    next_state = S_idle;
    Load_regs = 0;
    Add_regs = 0;
    case (state)
      S_idle: if (Start == 0) next_state = S_idle; else begin next_state = S_load; Load_regs = 1; end
      S_load:     begin next_state = S_decr; end
      S_decr:     begin next_state = S_shift; if (Q0) Add_regs = 1; end
      S_shift:    if (Zero) next_state = S_idle; else next_state = S_load;
    endcase
  end
endmodule

module t_Prob_8_23 ();
  parameter            m_size = 9;             // Width of datapath
  wire  [2 * m_size - 1: 0]      Product;
  wire                Ready;
  reg    [m_size - 1: 0]          Multiplicand, Multiplier;
  reg                 Start, clock, reset_b;
  integer                 Exp_Value;
  reg                 Error;
```

```
 Prob_8_23 M0 (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);


initial #140000 $finish;
initial begin clock = 0; #5 forever #5 clock = ~clock; end
initial fork
  reset_b = 1;

  #2 reset_b = 0;
  #3 reset_b = 1;
join
initial begin #5 Start = 1; end
always @ (posedge Ready) begin
  Exp_Value = Multiplier * Multiplicand;
  //Exp_Value = Multiplier * Multiplicand +1;     // Inject error to confirm detection
end
always @ (negedge Ready) begin
  Error = (Exp_Value ^ Product) ;
end

initial begin
  #5 Multiplicand = 0;
  Multiplier = 0;
  repeat (64) #10 begin Multiplier = Multiplier + 1;
    repeat (64) @ (posedge M0.Ready) #5 Multiplicand = Multiplicand + 1;
  end
end
endmodule
```

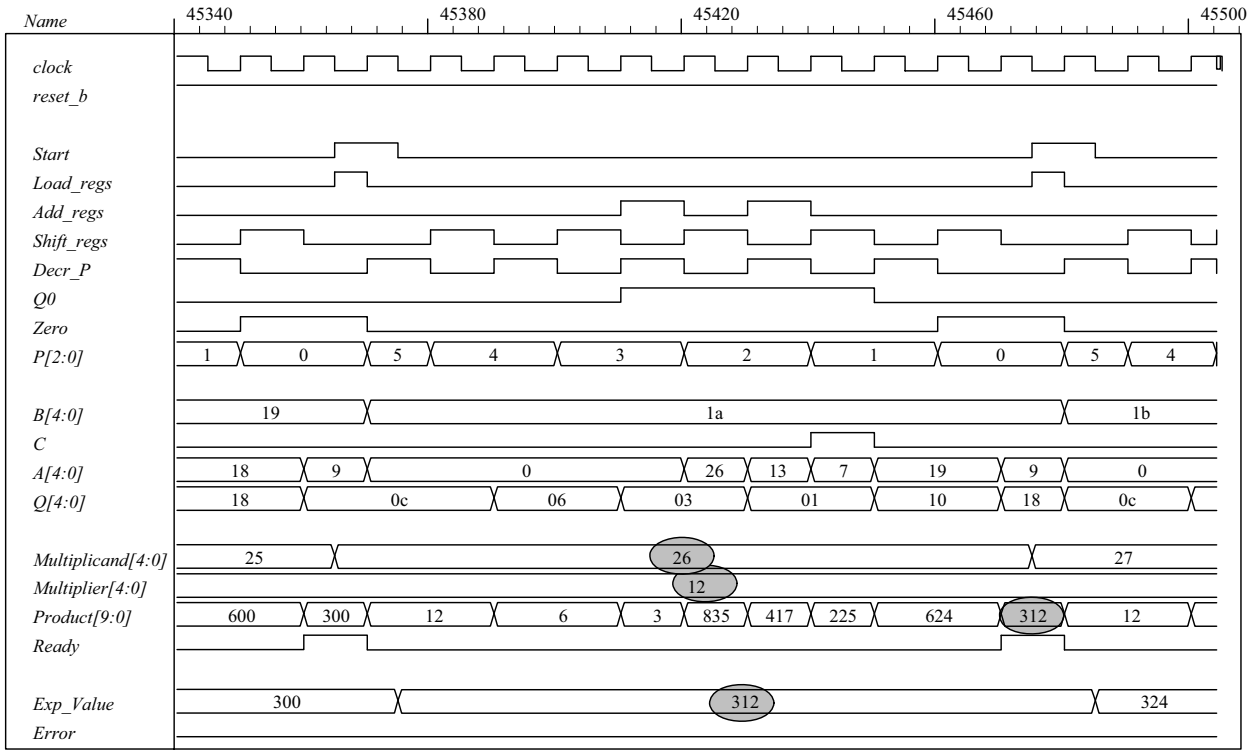


8.24

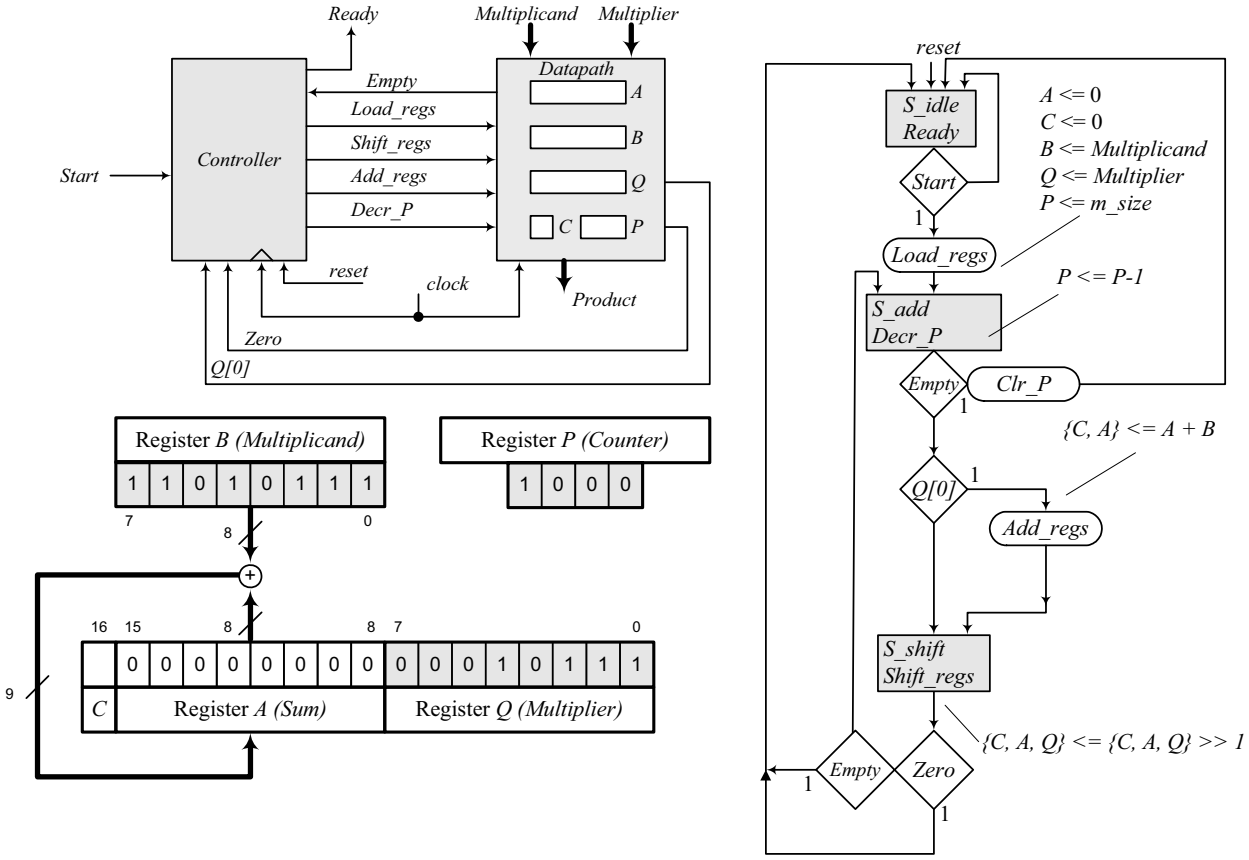
```
module Prob_8_24 # (parameter dp_width = 5)
(
 output   [2*dp_width - 1: 0]   Product,
 output                         Ready,
 input    [dp_width - 1: 0]     Multiplicand, Multiplier,
 input                          Start, clock, reset_b
);
 wire Load_regs, Decr_P, Add_regs, Shift_regs, Zero, Q0;

 Controller M0 (
  Ready, Load_regs, Decr_P, Add_regs, Shift_regs, Start, Zero, Q0,
  clock, reset_b
);

Datapath M1(Product, Q0, Zero,Multiplicand, Multiplier,
 Start, Load_regs, Decr_P, Add_regs, Shift_regs, clock, reset_b);
endmodule

module Controller (
 output Ready,
 output reg Load_regs, Decr_P, Add_regs, Shift_regs,
 input Start, Zero, Q0, clock, reset_b
);

 parameter      S_idle =   3'b001,        // one-hot code
                S_add =    3'b010,
                S_shift =  3'b100;
 reg   [2: 0]   state, next_state;      // sized for one-hot
 assign         Ready = (state == S_idle);

 always @ (posedge clock, negedge reset_b)
  if (~reset_b) state <= S_idle; else state <= next_state;

 always @ (state, Start, Q0, Zero) begin
  next_state = S_idle;
  Load_regs = 0;
  Decr_P = 0;
  Add_regs = 0;
  Shift_regs = 0;
  case (state)
   S_idle: if (Start) begin next_state = S_add; Load_regs = 1; end
   S_add: begin next_state = S_shift; Decr_P = 1; if (Q0) Add_regs = 1; end
   S_shift:    begin
    Shift_regs = 1;
    if (Zero) next_state = S_idle;
    else next_state = S_add;
                end
   default:   next_state = S_idle;
  endcase
 end
endmodule
```

```
module Datapath #(parameter dp_width = 5, BC_size = 3) (
  output [2*dp_width - 1: 0] Product, output Q0, output Zero,
  input [dp_width - 1: 0] Multiplicand, Multiplier,
  input Start, Load_regs, Decr_P, Add_regs, Shift_regs, clock, reset_b
);
// Default configuration: 5-bit datapath
  reg   [dp_width - 1: 0]       A, B, Q;                  // Sized for datapath
  reg                 C;
  reg   [BC_size - 1: 0]        P;                // Bit counter

  assign Q0 = Q[0];
  assign   Zero = (P == 0);             // Counter is zero
  assign Product = {C, A, Q};
  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) begin          // Added to this solution, but
      P <= dp_width;                      // not really necessary since Load_regs
      B <= 0;                             // initializes the datapath
      C <= 0;
      A <= 0;
      Q <= 0;
    end
    else begin
    if (Load_regs) begin
      P <= dp_width;
      A <= 0;
      C <= 0;
      B <= Multiplicand;
      Q <= Multiplier;
    end
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
    if (Decr_P) P <= P -1;
  end
endmodule

module t_Prob_8_24;
  parameter                     dp_width = 5;           // Width of datapath
  wire  [2 * dp_width - 1: 0]   Product;
  wire                          Ready;
  reg   [dp_width - 1: 0]       Multiplicand, Multiplier;
  reg                           Start, clock, reset_b;
  integer                       Exp_Value;
  reg                           Error;

  Prob_8_24 M0(Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);

  initial #115000 $finish;
  initial begin clock = 0; #5 forever #5 clock = ~clock; end
  initial fork
    reset_b = 1;
    #2 reset_b = 0;
    #3 reset_b = 1;
  join
  always @ (negedge Start) begin
    Exp_Value = Multiplier * Multiplicand;
    //Exp_Value = Multiplier * Multiplicand +1;     // Inject error to confirm detection
  end
  always @ (posedge Ready) begin
    # 1 Error <= (Exp_Value ^ Product) ;
  end
```

```
 initial begin
  #5 Multiplicand = 0;
  Multiplier = 0;

  repeat (32) #10 begin
    Start = 1;
    #10 Start = 0;
    repeat (32) begin
      Start = 1;
      #10 Start = 0;
      #100 Multiplicand = Multiplicand + 1;
    end
   Multiplier = Multiplier + 1;
  end
 end
endmodule
```



8.25 (a)



(b)

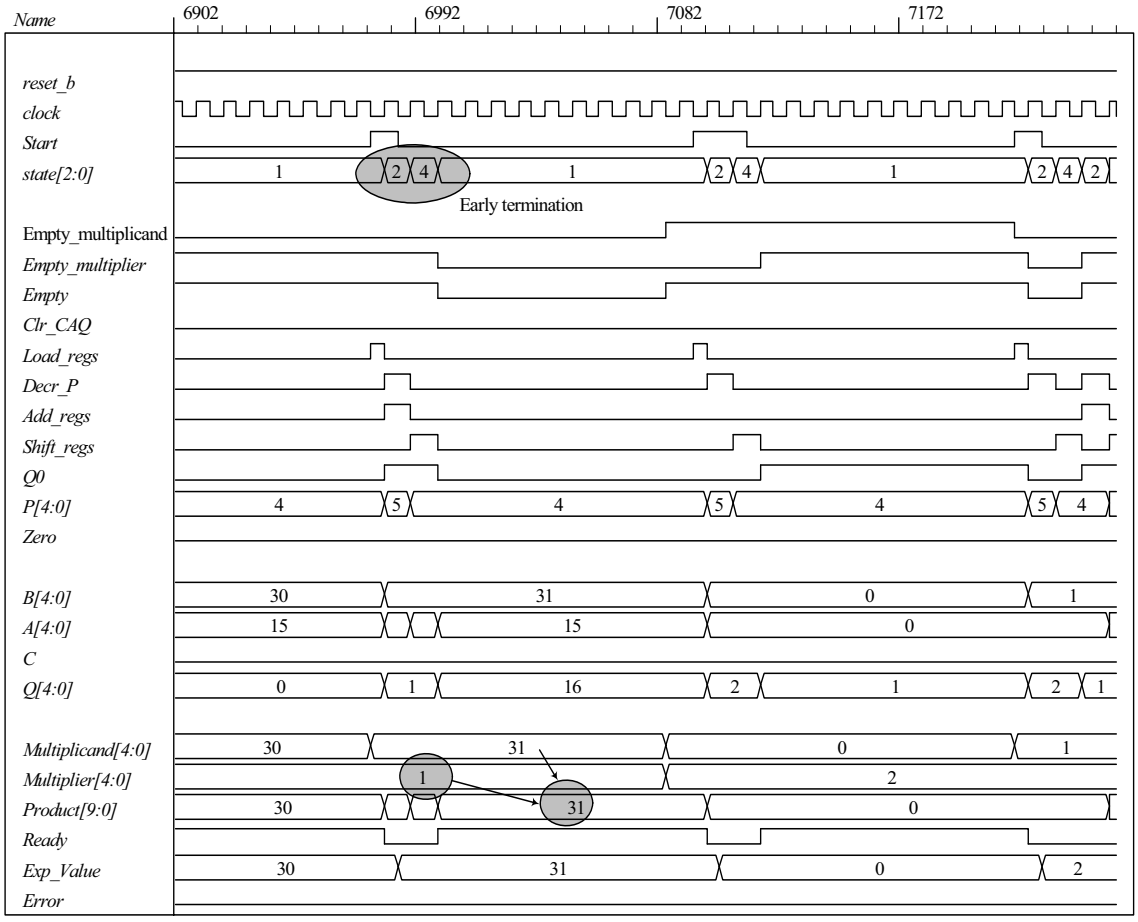
```
// The multiplier of Fig. 8.15 is modified to detect whether the multiplier or multiplicand are initially zero,
// and to detect whether the multiplier becomes zero before the entire multiplier has been applied
// to the multiplicand. Signal empty is generated by the datapath unit and used by the
// controller. Note that the bits of the product must be selected according to the stage at which
// termination occurs.  The test for the condition of an empty multiplier is hardwired here for
// dp_width = 5 because the range bounds of a vector must be defined by integer constants.
// This prevents development of a fully parameterized model.
// Note: the test bench has been modified.

module Prob_8_25 #(parameter dp_width = 5)
(
  output   [2*dp_width - 1: 0]     Product,
  output                           Ready,
  input    [dp_width - 1: 0]       Multiplicand, Multiplier,
  input                            Start, clock, reset_b
);
  wire Load_regs, Decr_P, Add_regs, Shift_regs, Empty, Zero, Q0;
  Controller M0 (
    Ready, Load_regs, Decr_P, Add_regs, Shift_regs, Start, Empty, Zero, Q0,
    clock, reset_b
);

Datapath M1(Product, Q0, Empty, Zero,Multiplicand, Multiplier,
  Start, Load_regs, Decr_P, Add_regs, Shift_regs, clock, reset_b);

endmodule
```

```
module Controller (
  output Ready,
  output reg Load_regs, Decr_P, Add_regs, Shift_regs,
  input Start, Empty, Zero, Q0, clock, reset_b
);

  parameter       BC_size =      3;       // Size of bit counter
  parameter       S_idle =    3'b001,         // one-hot code
                  S_add =     3'b010,
                  S_shift =   3'b100;
  reg    [2: 0]   state, next_state;      // sized for one-hot
  assign          Ready = (state == S_idle);

  always @ (posedge clock, negedge reset_b)
    if (~reset_b) state <= S_idle; else state <= next_state;

  always @ (state, Start, Q0, Empty, Zero) begin
    next_state = S_idle;
    Load_regs = 0;
    Decr_P = 0;
    Add_regs = 0;
    Shift_regs = 0;
    case (state)
      S_idle:    if (Start) begin next_state = S_add; Load_regs = 1; end
      S_add:     begin next_state = S_shift; Decr_P = 1; if (Q0) Add_regs = 1; end
      S_shift:   begin
                   Shift_regs = 1;
                   if (Zero) next_state = S_idle;
                   else if (Empty) next_state = S_idle;
                   else next_state = S_add;
                 end
      default:   next_state = S_idle;
    endcase
  end
endmodule

module Datapath #(parameter dp_width = 5, BC_size = 3) (
  output reg [2*dp_width - 1: 0] Product, output Q0, output Empty, output Zero,
  input [dp_width - 1: 0] Multiplicand, Multiplier,
  input Start, Load_regs, Decr_P, Add_regs, Shift_regs, clock, reset_b
);
// Default configuration: 5-bit datapath
  parameter       S_idle =    3'b001,         // one-hot code
                  S_add =     3'b010,
                  S_shift =   3'b100;
  reg    [dp_width - 1: 0]    A, B, Q;              // Sized for datapath
  reg                         C;
  reg    [BC_size - 1: 0]     P;              // Bit counter
  wire   [2*dp_width -1: 0]   Internal_Product = {C, A, Q};

  assign          Q0 = Q[0];
  assign          Zero = (P == 0);              // Bit counter is zero

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) begin             // Added to this solution, but
      P <= dp_width;                    // not really necessary since Load_regs
      B <= 0;                           // initializes the datapath
      C <= 0;
      A <= 0;
      Q <= 0;
    end
```

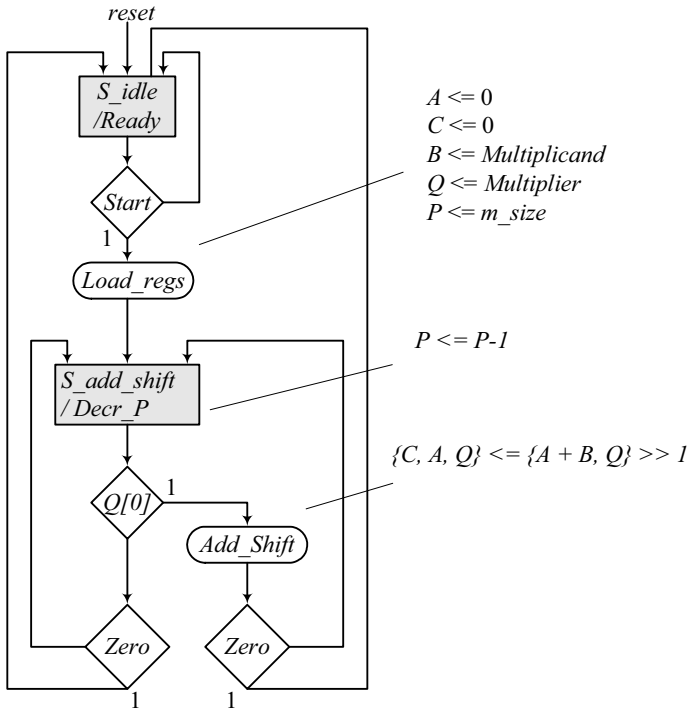
```
    else begin
    if (Load_regs) begin
      P <= dp_width;
      A <= 0;
      C <= 0;
      B <= Multiplicand;
      Q <= Multiplier;
    end
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
    if (Decr_P) P <= P -1;
  end
  // Status signals
  reg    Empty_multiplier;
  wire  Empty_multiplicand = (Multiplicand == 0);
  assign Empty = Empty_multiplicand || Empty_multiplier;

  always @ (P, Internal_Product) begin// Note: hardwired for dp_width  5
    Product = 0;
    case (P)          // Examine multiplier bits
      0:  Product  = Internal_Product;
      1:  Product = Internal_Product [2*dp_width -1: 1];
      2:  Product = Internal_Product [2*dp_width -1: 2];
      3:  Product = Internal_Product [2*dp_width -1: 3];
      4:  Product = Internal_Product [2*dp_width -1: 4];
      5:  Product = 0;
    endcase
  end

  always @ (P, Q) begin              // Note: hardwired for dp_width  5
    Empty_multiplier = 0;
    case (P)
      0:  Empty_multiplier = 1;
      1:  if (Q[1] == 0) Empty_multiplier = 1;
      2:  if (Q[2: 1] == 0) Empty_multiplier = 1;
      3:    if (Q[3: 1] == 0) Empty_multiplier = 1;
      4:  if (Q[4: 1] == 0) Empty_multiplier = 1;
      5:    if (Q[5: 1] == 0) Empty_multiplier = 1;
     default: Empty_multiplier = 1'bx;
    endcase
  end
endmodule

module t_Prob_8_25;
  parameter            dp_width = 5;            // Width of datapath
  wire  [2 * dp_width - 1: 0]    Product;
  wire                Ready;
  reg    [dp_width - 1: 0]        Multiplicand, Multiplier;
  reg                 Start, clock, reset_b;
  integer                 Exp_Value;
  reg                 Error;

  Prob_8_25 M0(Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);

  initial #115000 $finish;
  initial begin clock = 0; #5 forever #5 clock = ~clock; end
  initial fork
    reset_b = 1;
    #2 reset_b = 0;
    #3 reset_b = 1;
  join
```

```
always @ (negedge Start) begin
  Exp_Value = Multiplier * Multiplicand;
  //Exp_Value = Multiplier * Multiplicand +1;    // Inject error to confirm detection
end
always @ (posedge Ready) begin
  # 1 Error <= (Exp_Value ^ Product) ;
end

initial begin
  #5 Multiplicand = 0;
  Multiplier = 0;

  repeat (32) #10 begin
     Start = 1;
     #10 Start = 0;
     repeat (32) begin
       Start = 1;
       #10 Start = 0;
       #100 Multiplicand = Multiplicand + 1;
     end
    Multiplier = Multiplier + 1;
  end
 end
endmodule
```

(c) Test plan: Exhaustively test all combinations of multiplier and multiplicand, using automatic error checking. Verify that early termination is implemented. Sample of simulation results is shown below.



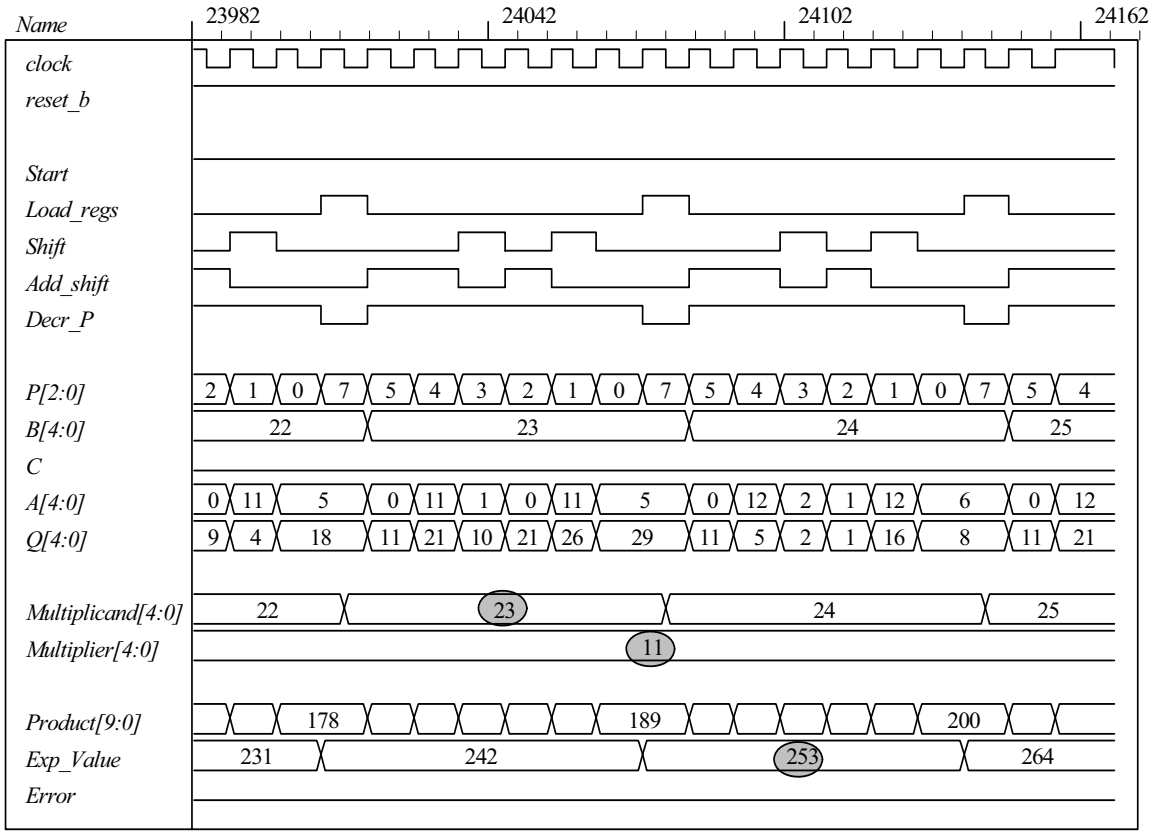
8.26




```
module Prob_8_26 (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);
// Default configuration: 5-bit datapath
  parameter              dp_width = 5;      // Set to width of datapath
  output   [2*dp_width - 1: 0]   Product;
  output                         Ready;
  input        [dp_width - 1: 0] Multiplicand, Multiplier;
  input                          Start, clock, reset_b;
  parameter                      BC_size =     3;      // Size of bit counter
  parameter                      S_idle =    2'b01,    // one-hot code
                                 S_add_shift =     2'b10;

  reg   [2: 0]                   state, next_state;
  reg   [dp_width - 1: 0]        A, B, Q;              // Sized for datapath
  reg                            C;
  reg   [BC_size -1: 0]          P;
  reg             Load_regs, Decr_P, Add_shift, Shift;
  assign          Product = {C, A, Q};
  wire            Zero = (P == 0);          // counter is zero
  wire            Ready = (state == S_idle);   // controller status

// control unit
  always @ (posedge clock, negedge reset_b)
    if (~reset_b) state <= S_idle; else state <= next_state;

  always @ (state, Start, Q[0], Zero) begin
    next_state = S_idle;
    Load_regs = 0;
    Decr_P = 0;
    Add_shift = 0;
    Shift = 0;
    case (state)
      S_idle:                 begin if (Start) next_state = S_add_shift; Load_regs = 1; end
```

```
    S_add_shift:        begin
                          Decr_P = 1;
                          if (Zero) next_state = S_idle;
                          else begin
                            next_state = S_add_shift;
                            if (Q[0]) Add_shift = 1; else Shift = 1;
                           end
                        end
     default:           next_state = S_idle;
   endcase
  end

// datapath unit
  always @ (posedge clock) begin
   if (Load_regs) begin
    P <= dp_width;
    A <= 0;
    C <= 0;
    B <= Multiplicand;
   Q <= Multiplier;
   end
   if (Decr_P) P <= P -1;
   if (Add_shift) {C, A, Q} <= {C, A+B, Q} >> 1;
   if (Shift) {C, A, Q} <= {C, A, Q} >> 1;
  end
endmodule

module t_Prob_8_26;
  parameter          dp_width = 5;          // Width of datapath
  wire  [2 * dp_width - 1: 0]   Product;
  wire                          Ready;
  reg   [dp_width - 1: 0]       Multiplicand, Multiplier;
  reg                           Start, clock, reset_b;
  integer                       Exp_Value;
  wire                          Error;

   Prob_8_26 M0 (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);

  initial #70000 $finish;
  initial begin clock = 0; #5 forever #5 clock = ~clock; end
  initial fork
    reset_b = 1;
    #2 reset_b = 0;
    #3 reset_b = 1;
  join
  initial begin #5 Start = 1; end
  always @ (posedge Ready) begin
    Exp_Value = Multiplier * Multiplicand;
  end
  assign Error = Ready & (Exp_Value ^ Product);
  initial begin
    #5 Multiplicand = 0;
    Multiplier = 0;
    repeat (32) #10 begin Multiplier = Multiplier + 1;
      repeat (32) @ (posedge M0.Ready) #5 Multiplicand = Multiplicand + 1;
    end
  end
endmodule
```

Sample of simulation results.



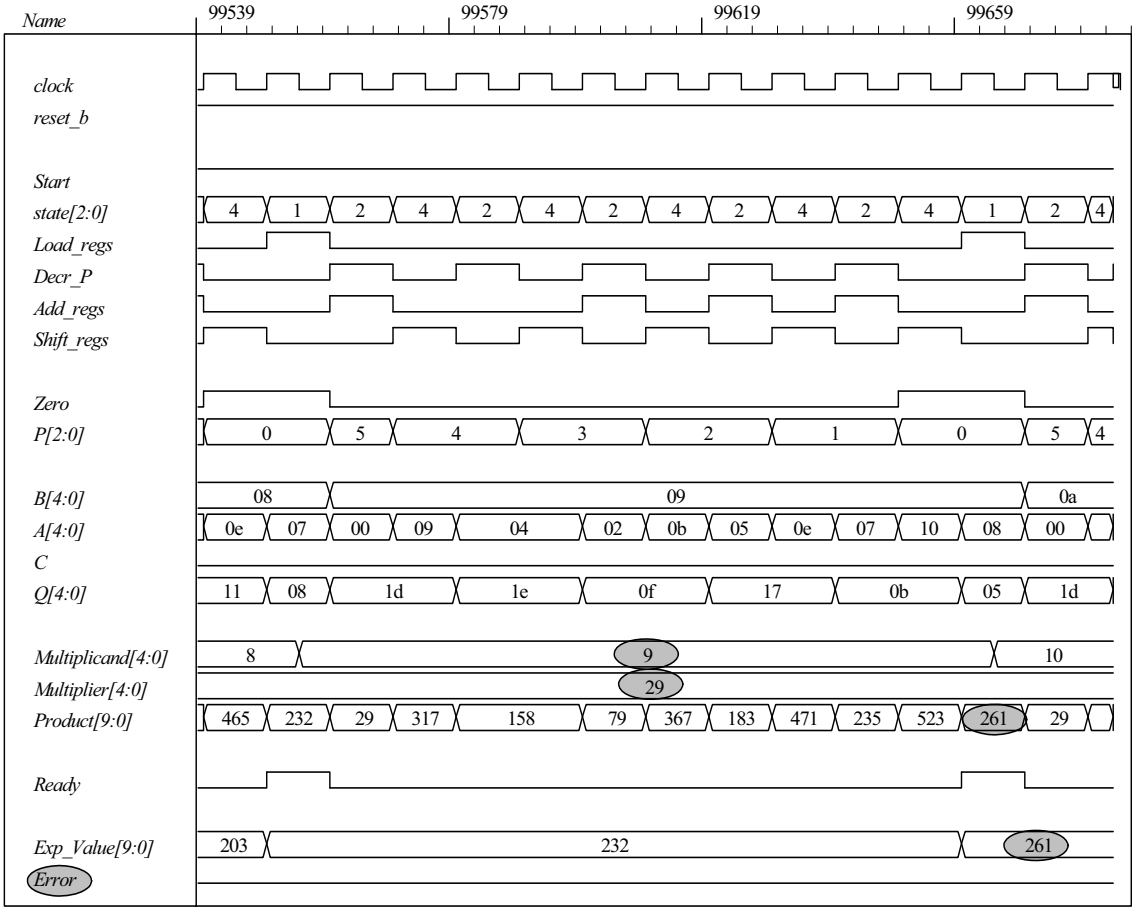
8.27 **(a)**

```
// Test bench for exhaustive simulation
module t_Sequential_Binary_Multiplier;
 parameter                  dp_width = 5;          // Width of datapath
 wire  [2 * dp_width - 1: 0] Product;
 wire                       Ready;
 reg   [dp_width - 1: 0]    Multiplicand, Multiplier;
 reg                        Start, clock, reset_b;

  Sequential_Binary_Multiplier M0 (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);

 initial #109200 $finish;
 initial begin clock = 0; #5 forever #5 clock = ~clock; end
 initial fork
   reset_b = 1;
   #2 reset_b = 0;
   #3 reset_b = 1;
 join
 initial begin #5 Start = 1; end
 initial begin
   #5 Multiplicand = 0;
   Multiplier = 0;
     repeat (31) #10 begin Multiplier = Multiplier + 1;
       repeat (32) @ (posedge M0.Ready) #5 Multiplicand = Multiplicand + 1;
     end
   Start = 0;
 end

// Error Checker
```

```
 reg Error;
 reg [2*dp_width -1: 0] Exp_Value;
 always @ (posedge Ready) begin
  Exp_Value = Multiplier * Multiplicand;
  //Exp_Value = Multiplier * Multiplicand + 1;        // Inject error to verify detection
  Error = (Exp_Value ^ Product);
 end
endmodule

 module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);
// Default configuration: 5-bit datapath
 parameter                      dp_width = 5;       // Set to width of datapath
 output   [2*dp_width - 1: 0]   Product;
 output                         Ready;
 input    [dp_width - 1: 0]     Multiplicand, Multiplier;
 input                          Start, clock, reset_b;

 parameter          BC_size =    3;   // Size of bit counter
 parameter          S_idle =     3'b001,     // one-hot code
                    S_add =      3'b010,
                    S_shift =    3'b100;

 reg   [2: 0]                   state, next_state;
 reg   [dp_width - 1: 0]        A, B, Q;                 // Sized for datapath
 reg                            C;
 reg   [BC_size - 1: 0]         P;
 reg                            Load_regs, Decr_P, Add_regs, Shift_regs;

// Miscellaneous combinational logic

 assign          Product = {C, A, Q};
 wire            Zero = (P == 0);            // counter is zero
 wire            Ready = (state == S_idle);        // controller status

// control unit

 always @ (posedge clock, negedge reset_b)
  if (~reset_b) state <= S_idle; else state <= next_state;

 always @ (state, Start, Q[0], Zero) begin
  next_state = S_idle;
  Load_regs = 0;
  Decr_P = 0;
  Add_regs = 0;
  Shift_regs = 0;
  case (state)
   S_idle: begin if (Start) next_state = S_add; Load_regs = 1; end
   S_add: begin next_state = S_shift; Decr_P = 1; if (Q[0]) Add_regs = 1; end
   S_shift:    begin Shift_regs = 1; if (Zero) next_state = S_idle;
   else next_state = S_add; end
   default:    next_state = S_idle;
  endcase
 end
```

```
// datapath unit

  always @ (posedge clock) begin
    if (Load_regs) begin
      P <= dp_width;
      A <= 0;
      C <= 0;
      B <= Multiplicand;
      Q <= Multiplier;
    end
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
    if (Decr_P) P <= P -1;
  end
endmodule
```

Sample of simulation results:

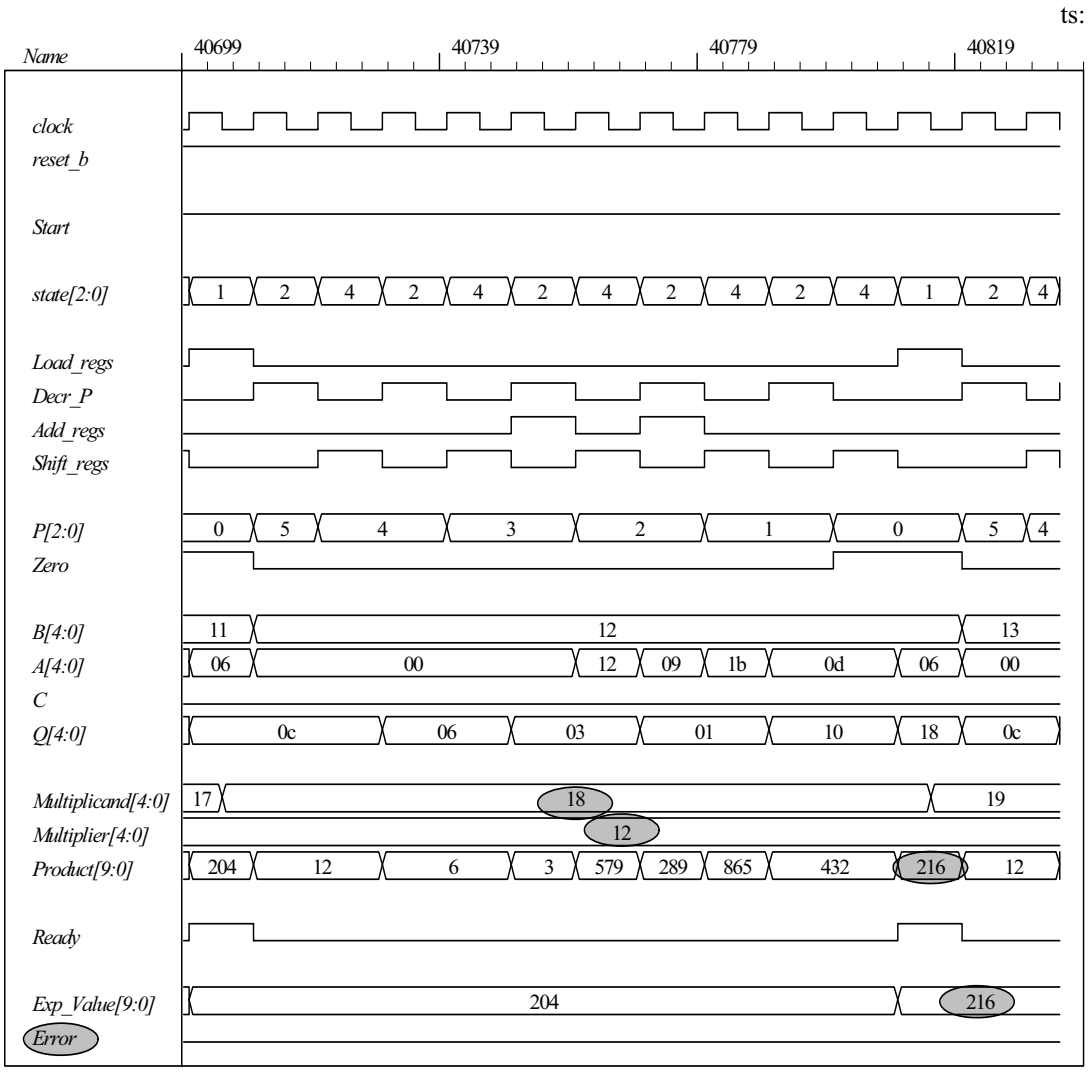


(b) In this part the controller is described by Fig. 8.18. The test bench includes probes to display the state of the controller.

```
// Test bench for exhaustive simulation
module t_Sequential_Binary_Multiplier;
  parameter                 dp_width = 5;          // Width of datapath
  wire  [2 * dp_width - 1: 0] Product;
  wire                      Ready;
  reg   [dp_width - 1: 0]   Multiplicand, Multiplier;
  reg                       Start, clock, reset_b;

   Sequential_Binary_Multiplier M0 (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);

  initial #109200 $finish;
  initial begin clock = 0; #5 forever #5 clock = ~clock; end
  initial fork
    reset_b = 1;
    #2 reset_b = 0;
    #3 reset_b = 1;
  join
  initial begin #5 Start = 1; end
  initial begin
    #5 Multiplicand = 0;
    Multiplier = 0;
    repeat (31) #10 begin Multiplier = Multiplier + 1;
      repeat (32) @ (posedge M0.Ready) #5 Multiplicand = Multiplicand + 1;
    end
    Start = 0;
  end

// Error Checker
  reg Error;
  reg [2*dp_width -1: 0] Exp_Value;
  always @ (posedge Ready) begin
    Exp_Value = Multiplier * Multiplicand;
    //Exp_Value = Multiplier * Multiplicand + 1;       // Inject error to verify detection
    Error = (Exp_Value ^ Product);
  end

  wire [2: 0] state = {M0.G2, M0.G1, M0.G0};
endmodule

 module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);
// Default configuration: 5-bit datapath
  parameter                 dp_width =    5;       // Set to width of datapath
  output  [2*dp_width - 1: 0] Product;
  output                    Ready;
  input   [dp_width - 1: 0] Multiplicand, Multiplier;
  input                     Start, clock, reset_b;

  parameter                 BC_size =     3;  // Size of bit counter
  reg   [dp_width - 1: 0]   A, B, Q;             // Sized for datapath
  reg                       C;
  reg   [BC_size - 1: 0]    P;
  wire                      Load_regs, Decr_P, Add_regs, Shift_regs;
```

```
// Status signals

  assign        Product = {C, A, Q};
  wire          Zero = (P == 0);        // counter is zero
  wire          Q0 = Q[0];

// One-Hot Control unit (See Fig. 8.18)
  DFF_S M0 (G0, D0, clock, Set);
  DFF M1 (G1, D1, clock, reset_b);
  DFF M2 (G2, G1, clock, reset_b);
  or (D0, w1, w2);
  and (w1, G0, Start_b);
  and (w2, Zero, G2);
  not (Start_b, Start);
  not (Zero_b, Zero);
  or (D1, w3, w4);
  and (w3, Start, G0);
  and (w4, Zero_b, G2);

  and (Load_regs, G0, Start);
  and (Add_regs, Q0, G1);
  assign Ready = G0;
  assign Decr_P = G1;
  assign Shift_regs = G2;
  not (Set, reset_b);

// datapath unit

  always @ (posedge clock) begin
    if (Load_regs) begin
      P <= dp_width;
      A <= 0;
      C <= 0;
      B <= Multiplicand;
      Q <= Multiplier;
    end
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
    if (Decr_P) P <= P -1;
  end
endmodule

module DFF_S (output reg Q, input data, clock, Set);
  always @ ( posedge clock, posedge Set)
    if (Set) Q <= 1'b1; else Q<= data;
endmodule
module DFF (output reg Q, input data, clock, reset_b);
  always @ ( posedge clock, negedge reset_b)
    if (reset_b == 0) Q <= 1'b0; else Q<= data;
endmodule
```

Sample of simulation results:

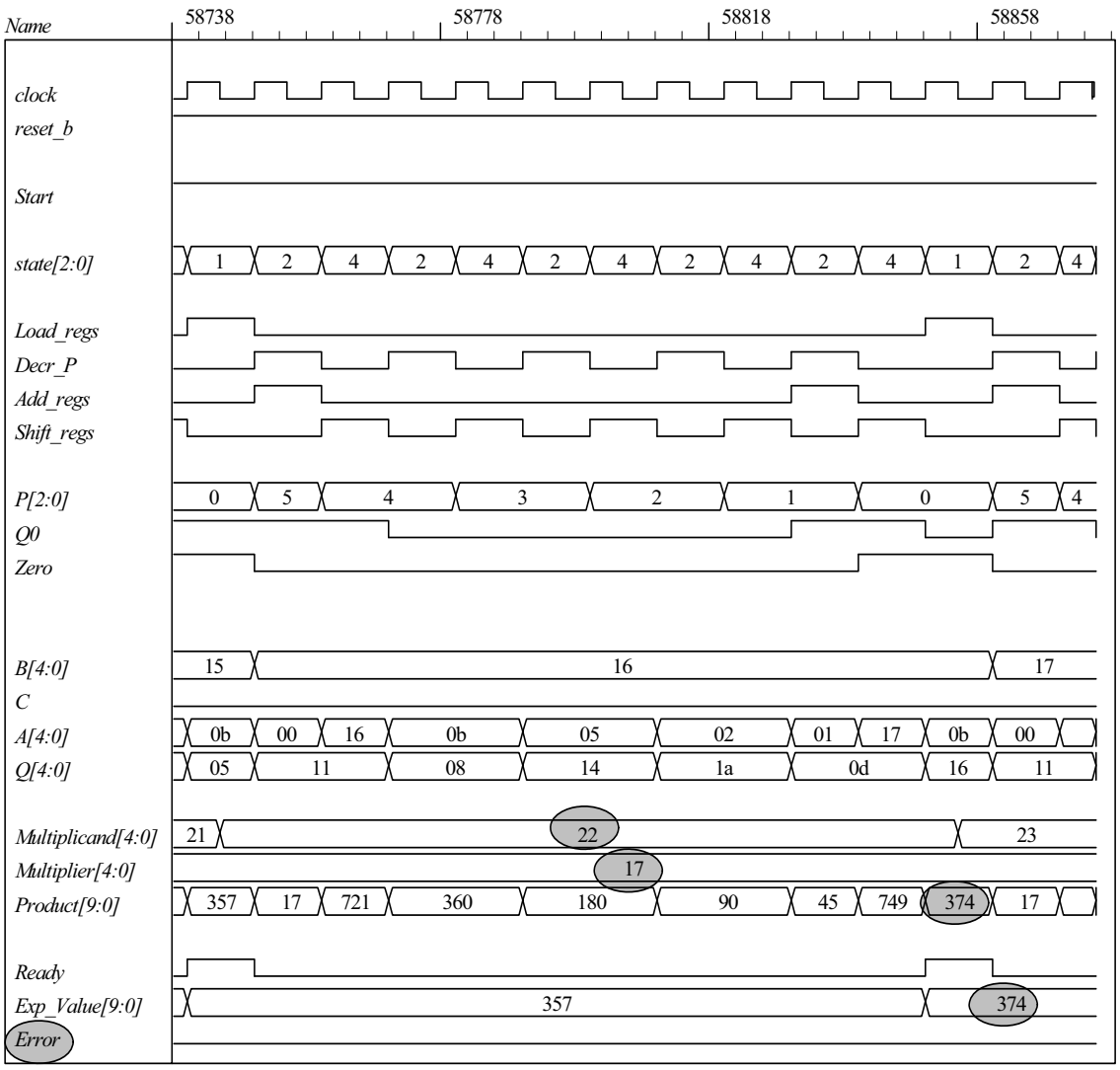


8.28

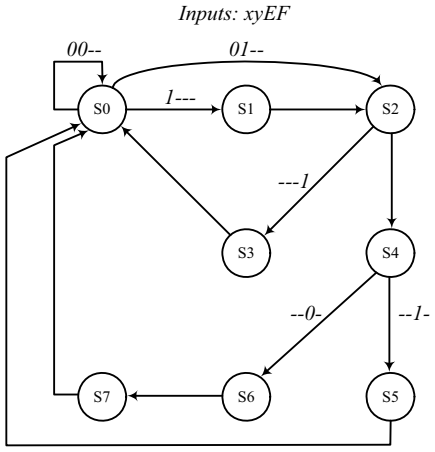
```
// Test bench for exhaustive simulation
module t_Sequential_Binary_Multiplier;
 parameter                  dp_width = 5;          // Width of datapath
 wire  [2 * dp_width - 1: 0] Product;
 wire                        Ready;
 reg   [dp_width - 1: 0]     Multiplicand, Multiplier;
 reg                         Start, clock, reset_b;

  Sequential_Binary_Multiplier M0 (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);

 initial #109200 $finish;
 initial begin clock = 0; #5 forever #5 clock = ~clock; end
 initial fork
   reset_b = 1;
   #2 reset_b = 0;
   #3 reset_b = 1;
 join
```

```
  initial begin #5 Start = 1; end
  initial begin
    #5 Multiplicand = 0;
    Multiplier = 0;
    repeat (31) #10 begin Multiplier = Multiplier + 1;
      repeat (32) @ (posedge M0.Ready) #5 Multiplicand = Multiplicand + 1;
    end
    Start = 0;
  end

// Error Checker
  reg Error;
  reg [2*dp_width -1: 0] Exp_Value;
  always @ (posedge Ready) begin
    Exp_Value = Multiplier * Multiplicand;
    //Exp_Value = Multiplier * Multiplicand + 1;        // Inject error to verify detection
    Error = (Exp_Value ^ Product);
  end
  wire [2: 0] state = {M0.M0.G2, M0.M0.G1, M0.M0.G0};   // Watch state
endmodule

 module Sequential_Binary_Multiplier
   #(parameter dp_width =    5)
(
  output [2*dp_width -1: 0]    Product,
  output                       Ready,
  input [dp_width -1: 0]       Multiplicand, Multiplier,
  input                        Start, clock, reset_b
);
   wire Load_regs, Decr_P, Add_regs, Shift_regs, Zero, Q0;

  Controller M0 (Ready, Load_regs, Decr_P, Add_regs, Shift_regs, Start, Zero, Q0, clock, reset_b);
  Datapath M1(Product, Q0, Zero,Multiplicand, Multiplier, Start, Load_regs, Decr_P, Add_regs,
    Shift_regs, clock, reset_b);
endmodule

module Controller (
  output Ready,
  output Load_regs, Decr_P, Add_regs, Shift_regs,
  input Start, Zero, Q0, clock, reset_b
);
// One-Hot Control unit (See Fig. 8.18)
  DFF_S M0 (G0, D0, clock, Set);
  DFF M1 (G1, D1, clock, reset_b);
  DFF M2 (G2, G1, clock, reset_b);
  or (D0, w1, w2);
  and (w1, G0, Start_b);
  and (w2, Zero, G2);
  not (Start_b, Start);
  not (Zero_b, Zero);
  or (D1, w3, w4);
  and (w3, Start, G0);
  and (w4, Zero_b, G2);

  and (Load_regs, G0, Start);
  and (Add_regs, Q0, G1);
  assign Ready = G0;
  assign Decr_P = G1;
  assign Shift_regs = G2;
  not (Set, reset_b);
endmodule
```

```
module Datapath #(parameter dp_width = 5, BC_size = 3) (
  output [2*dp_width - 1: 0] Product, output Q0, output Zero,
  input [dp_width - 1: 0] Multiplicand, Multiplier,
  input Start, Load_regs, Decr_P, Add_regs, Shift_regs, clock, reset_b
);
  reg   [dp_width - 1: 0]       A, B, Q;              // Sized for datapath
  reg                 C;
  reg   [BC_size - 1: 0]        P;
  assign                Product = {C, A, Q};
  // Status signals
  assign                Zero = (P == 0);        // counter is zero
  assign                Q0 = Q[0];

  always @ (posedge clock) begin
    if (Load_regs) begin
      P <= dp_width;
      A <= 0;
      C <= 0;
      B <= Multiplicand;
      Q <= Multiplier;
    end
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
    if (Decr_P) P <= P -1;
  end
endmodule

module DFF_S (output reg Q, input data, clock, Set);
  always @ ( posedge clock, posedge Set)
    if (Set) Q <= 1'b1; else Q<= data;
endmodule
module DFF (output reg Q, input data, clock, reset_b);
  always @ ( posedge clock, negedge reset_b)
    if (reset_b == 0) Q <= 1'b0; else Q<= data;
endmodule
```



8.29 **(a)**



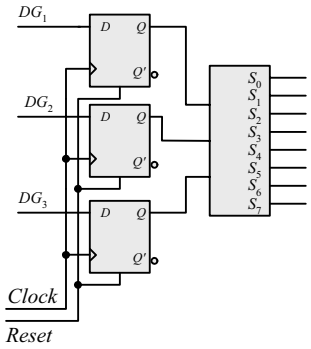
(b)

$DS_0 = x'y'S_0 + S_3 + S_5 + S_7$
$DS_1 = xS_0$
$DS_2 = x'yS_0 + S_1$
$DS_3 = FS_2$
$DS_4 = F'S_2$
$DS_5 = E'S_5$
$DS_6 = E'S_4$
$DS_7 = S_6$

(c)

Output	Present state $G_1\,G_2\,G_3$	Inputs $x\ y\ E\ F$	Next state $G_1\,G_2\,G_3$
*S*0	0 0 0	0 0 x x	0 0 0
*S*0	0 0 0	1 x x x	0 0 1
*S*0	0 0 0	0 1 x x	0 1 0
*S*1	0 0 1	x x x x	0 1 0
*S*2	0 1 0	x x 0 x	1 0 0
*S*2	0 1 0	x x 1 x	0 1 1
*S*3	0 1 1	x x x x	0 0 0
*S*4	1 0 0	x x x 0	1 1 0
S4	1 0 0	x x x 1	1 0 1
*S*5	1 0 1	x x x x	0 0 0
*S*6	1 1 0	x x x x	1 1 0
*S*7	1 1 1	x x x x	0 0 0

(d)

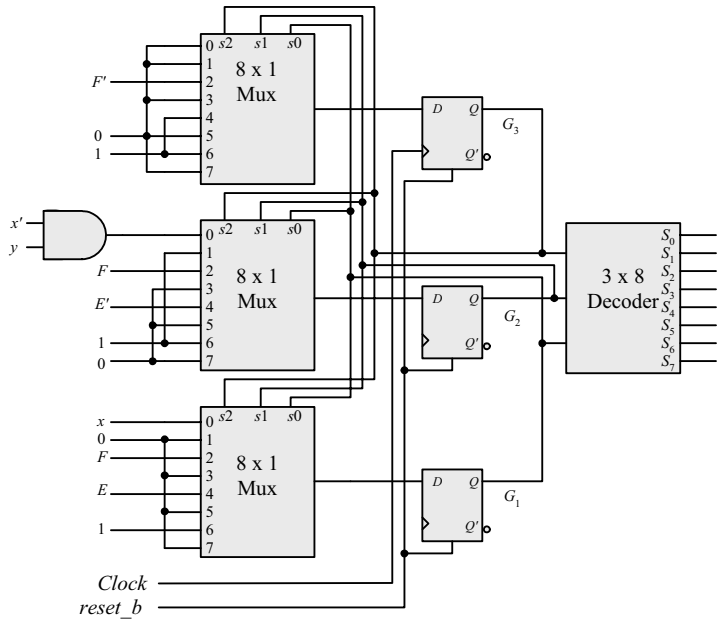


$DG_1 = F'S_2 + S_4 + S_6$
$DG_2 = x'yS_0 + S_1 + FS_2 + E'S_4 + S_6$
$DG_3 = xS_0 + FS_2 + ES_4 + S_6$

(e)

Present state $G_1\,G_2\,G_3$	Next state $G_1\,G_2\,G_3$	Input conditions	Mux1	Mux2	Mux3
0 0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1 0	$x'y'$ x $x'y$	0	$x'y$	x
0 0 1	0 1 0	None	0	1	0
0 1 0 0 1 0	1 0 0 0 1 1	F' F'	F'	F	F
0 1 1	0 0 0	None	0	0	0
1 0 0 1 0 0	1 1 0 1 0 1	E' E'	1	E'	E
1 0 1	0 0 0	None	0	0	0
1 1 0	1 1 0	None	1	1	1
1 1 1	0 0 0	None	0	0	0

(f)



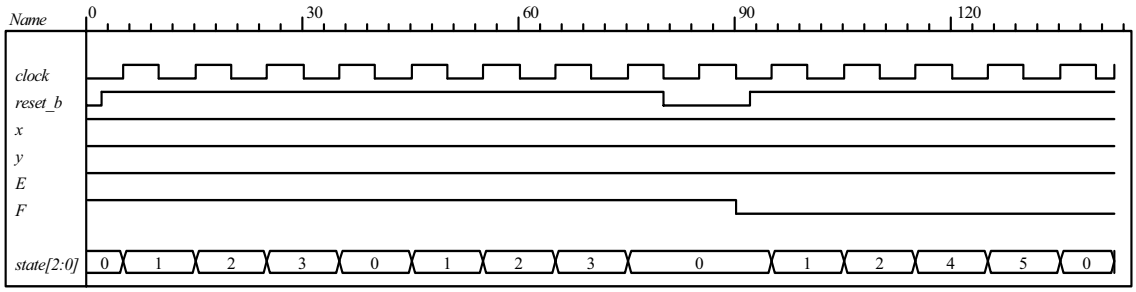
(g)

```
module Controller_8_29g (input  x, y, E, F, clock, reset_b);
  supply0 GND;
  supply1 VCC;

  mux_8x1 M3 (m3, GND, GND, F_bar, GND, VCC, GND, VCC, GND, G3, G2, G1);
  mux_8x1 M2 (m2, w1, VCC, F, GND, E_bar, GND, VCC, GND, G3, G2, G1);
  mux_8x1 M1 (m1, x, GND, F, GND, E, GND, VCC, GND, G3, G2, G1);
  DFF_8_28g DM3 (G3, m3, clock, reset_b);
  DFF_8_28g DM2 (G2, m2, clock, reset_b);
  DFF_8_28g DM1 (G1, m1, clock, reset_b);
  decoder_3x8 M0_D (y0, y1, y2, y3, y4, y5, y6, y7, G3, G2, G1);
```

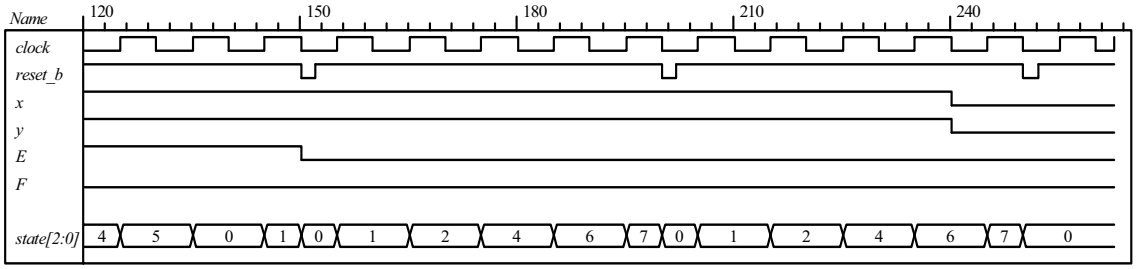
```
  and (w1, x_bar, y);
  not (F_bar, F);
  not (E_bar, E);
  not (x_bar, x);
endmodule

// Test plan: Exercise all paths of the ASM chart

module t_Controller_8_29g ();
  reg  x, y, E, F, clock, reset_b;
  Controller_8_29g M0 (x, y, E, F, clock, reset_b);
  wire [2: 0] state = {M0.G3, M0.G2, M0.G1};

  initial #500 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin end
  initial fork
    reset_b = 0; #2 reset_b = 1;
    #0 begin x = 1; y = 1; E = 1; F = 1; end  // Path: S_0, S_1, S_2, S_34
    #80 reset_b = 0; #92 reset_b = 1;
    #90 begin x = 1; y = 1; E = 1; F = 0; end
    #150 reset_b = 0;
    #152 reset_b = 1;
    #150 begin x = 1; y = 1; E = 0; F = 0; end    // Path: S_0, S_1, S_2, S_4, S_5
    #200 reset_b = 0;
    #202 reset_b = 1;
    #190 begin x = 1; y = 1; E = 0; F = 0; end    // Path: S_0, S_1, S_2, S_4, S_6, S_7
    #250 reset_b = 0;
    #252 reset_b = 1;
    #240 begin x = 0; y = 0; E = 0; F = 0; end    // Path: S_0
    #290 reset_b = 0;
    #292 reset_b = 1;
    #280 begin x = 0; y = 1; E = 0; F = 0; end    // Path: S_0, S_2, S_4, S_6, S_7
    #360 reset_b = 0;
    #362 reset_b = 1;
    #350 begin x = 0; y = 1; E = 1; F = 0; end    // Path: S_0, S_2, S_4, S_5
    #420 reset_b = 0;
    #422 reset_b = 1;
    #410 begin x = 0; y = 1; E = 0; F = 1; end    // Path: S_0, S_2, S_3
  join
endmodule

module  mux_8x1 (output reg y, input x0, x1, x2, x3, x4, x5, x6, x7, s2, s1, s0);
  always @ (x0, x1, x2, x3, x4, x5, x6, x7, s0, s1, s2)
    case ({s2, s1, s0})
      3'b000: y = x0;
      3'b001: y = x1;
      3'b010: y = x2;
      3'b011: y = x3;
      3'b100: y = x4;
      3'b101: y = x5;
      3'b110: y = x6;
      3'b111: y = x7;
    endcase
endmodule

module DFF_8_28g (output reg q, input data, clock, reset_b);
  always @ (posedge clock, negedge reset_b)
    if (!reset_b) q <= 1'b0; else q <= data;
endmodule
```

```
module  decoder_3x8 (output reg y0, y1, y2, y3, y4, y5, y6, y7, input x2, x1, x0);
  always @ (x0, x1, x2) begin
    {y7, y6, y5, y4, y3, y2, y1, y0} = 8'b0;
    case ({x2, x1, x0})
      3'b000: y0= 1'b1;
      3'b001: y1= 1'b1;
      3'b010: y2= 1'b1;
      3'b011: y3= 1'b1;
      3'b100: y4= 1'b1;
      3'b101: y5= 1'b1;
      3'b110: y6= 1'b1;
      3'b111: y7= 1'b1;
    endcase
  end
endmodule
```

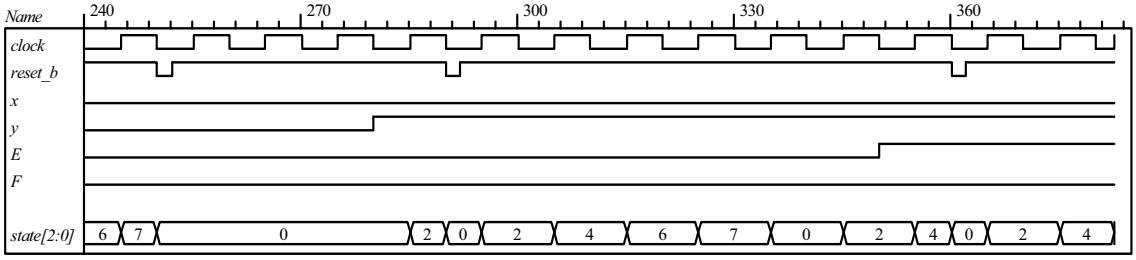
Path: S_0, S_1, S_2, S_3 and Path: S_0, S_1, S_2, S_4, S_5



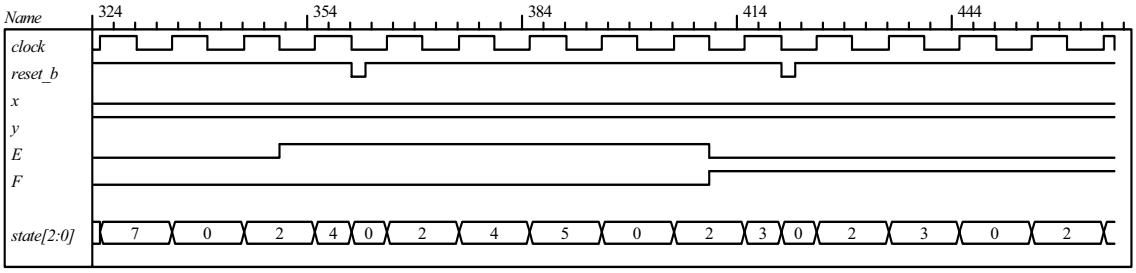
Path: S_0, S_1, S_2, S_4, S_6, S_7



Path: S_0 and Path , S_0, S_2, S_4, S_6, S_7



Path: S_0, S_2, S_4, S_5 and path S_0, S_2, S_3



(h)

```
module Controller_8_29h (input  x, y, E, F, clock, reset_b);
  parameter  S_0 = 3'b000, S_1 = 3'b001, S_2 = 3'b010,
     S_3 = 3'b011, S_4 = 3'b100, S_5 = 3'b101, S_6 = 3'b110, S_7 = 3'b111;
  reg [2: 0 ] state, next_state;

  always @ (posedge clock, negedge reset_b)
    if (!reset_b) state <= S_0; else state <= next_state;
```

```
 always @ (state, x, y, E, F) begin
   case (state)
     S_0:              if (x) next_state = S_1;
                       else next_state =  y ? S_2: S_0;
     S_1:              next_state = S_2;
     S_2:              if (F) next_state = S_3; else next_state = S_4;
     S_3, S_5, S_7:    next_state = S_0;
     S_4:              if (E) next_state = S_5; else next_state = S_6;
     S_6:              next_state = S_7;
     default:          next_state = S_0;
   endcase
 end
endmodule

// Test plan: Exercise all paths of the ASM chart

module t_Controller_8_29h ();
 reg  x, y, E, F, clock, reset_b;

 Controller_8_29h M0 (x, y, E, F, clock, reset_b);

 initial #500 $finish;
 initial begin clock = 0; forever #5 clock = ~clock; end
 initial begin end
 initial fork
  reset_b = 0; #2 reset_b = 1;
  #20 begin x = 1; y = 1; E = 1; F = 1; end// Path: S_0, S_1, S_2, S_34
  #80 reset_b = 0; #92 reset_b = 1;
  #90 begin x = 1; y = 1; E = 1; F = 0; end
  #150 reset_b = 0;
  #152 reset_b = 1;
  #150 begin x = 1; y = 1; E = 0; F = 0; end    // Path: S_0, S_1, S_2, S_4, S_5
  #200 reset_b = 0;
  #202 reset_b = 1;
  #190 begin x = 1; y = 1; E = 0; F = 0; end    // Path: S_0, S_1, S_2, S_4, S_6, S_7
  #250 reset_b = 0;
  #252 reset_b = 1;
  #240 begin x = 0; y = 0; E = 0; F = 0; end    // Path: S_0
  #290 reset_b = 0;
  #292 reset_b = 1;
  #280 begin x = 0; y = 1; E = 0; F = 0; end    // Path: S_0, S_2, S_4, S_6, S_7
  #360 reset_b = 0;
  #362 reset_b = 1;
  #350 begin x = 0; y = 1; E = 1; F = 0; end    // Path: S_0, S_2, S_4, S_5
  #420 reset_b = 0;
  #422 reset_b = 1;
  #410 begin x = 0; y = 1; E = 0; F = 1; end    // Path: S_0, S_2, S_3
 join
endmodule
```

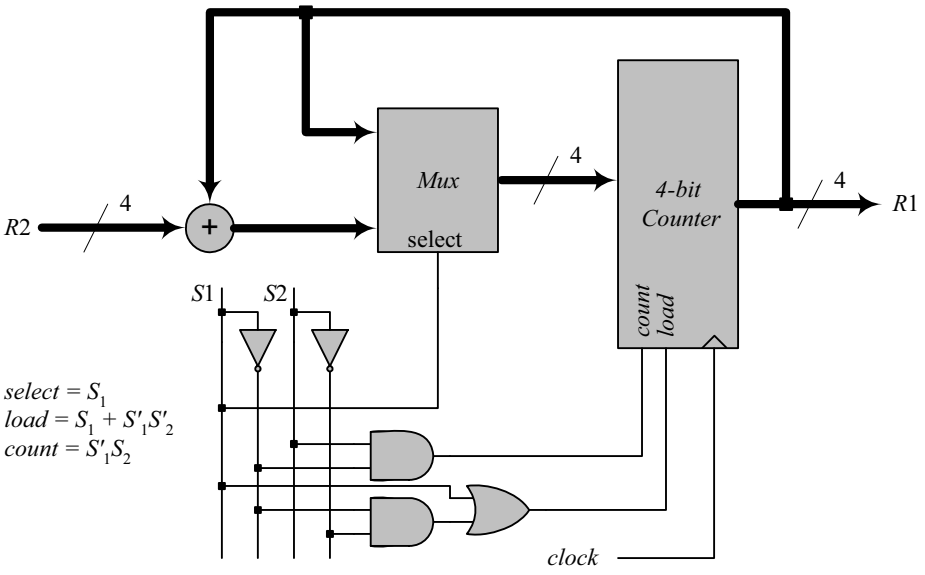
Note: Simulation results match those for 8.39g.

8.30 **(a)** $E = 1$ **(b)** $E = 0$

8.31 A = 0110, B = 0010, C = 0000.

A * B = 1100	A | B = 0110	A && C = 0
A + B = 1000	A ∧ B = 0100	| A = 1
A – B = 0100	&A = 0	A < B = 0
~ C = 1111	~|C = 1	A > B = 1
A & B = 0010	A || B = 1	A ! B = 1

8.32



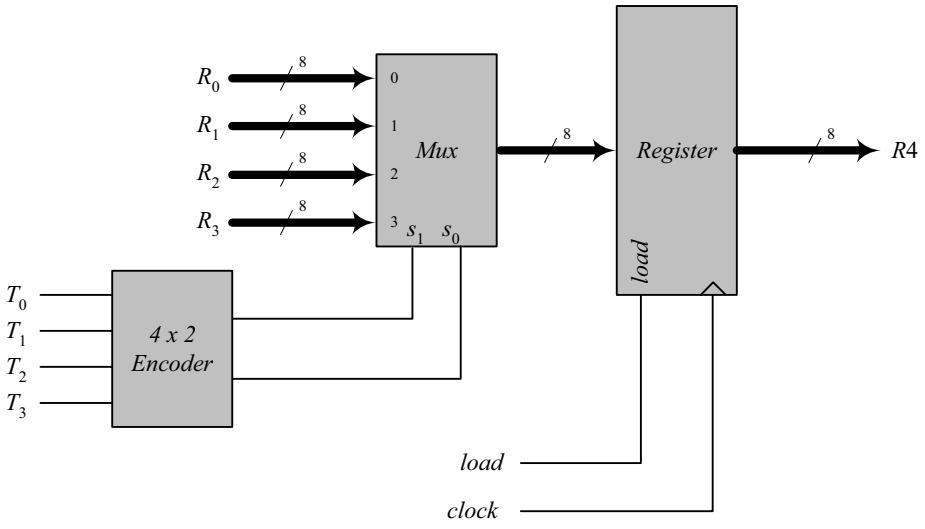
8.33

Assume that the states are encoded one-hot as T_0, T_1, T_2, T_3. The select lines of the mux are generated as:

$s_1 = T_2 + T_3$
$s_0 = T_1 + T_3$

The signal to load R_4 can be generated by the host processor or by:

$load = T_0 + T_1 + T_2 + T_3$.

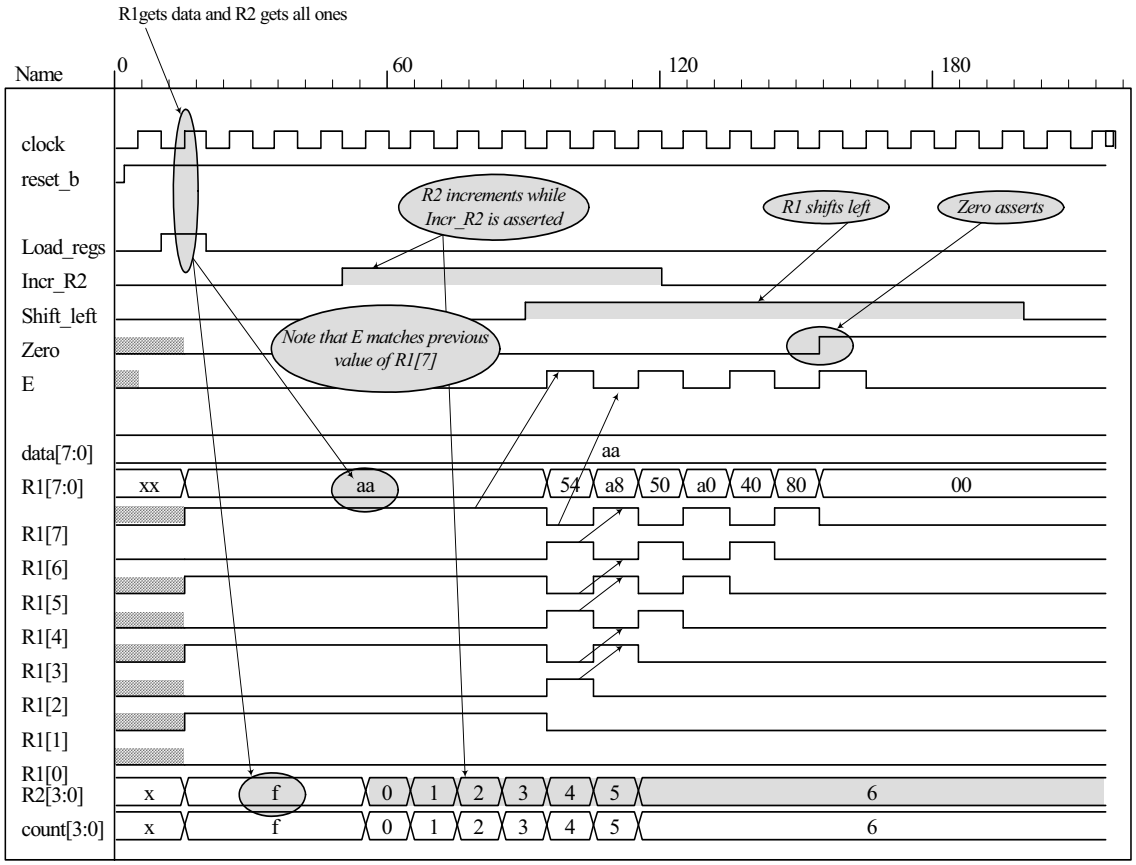


8.34 (a)

```
module Datapath_BEH
#(parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0] count, output reg E, output Zero, input [dp_width -1: 0] data,
input Load_regs, Shift_left, Incr_R2, clock, reset_b);
```

```
  reg [dp_width -1: 0] R1;
  reg [R2_width -1: 0] R2;

  assign count = R2;
  assign Zero = ~(| R1);
  always @ (posedge clock) begin
    E <= R1[dp_width -1] & Shift_left;
    if (Load_regs) begin R1 <= data; R2 <= {R2_width{1'b1}}; end
    if (Shift_left) {E, R1} <= {E, R1} << 1;
    if (Incr_R2) R2 <= R2 + 1;
  end
endmodule

// Test Plan for Datapath Unit:
// Demonstrate action of Load_regs
//      R1 gets data, R2 gets all ones
// Demonstrate action of Incr_R2
// Demonstrate action of Shift_left and detect E

// Test bench for datapath

module t_Datapath_Unit
#(parameter dp_width = 8, R2_width = 4)
( );
  wire [R2_width -1: 0]   count;
  wire                    E, Zero;
  reg [dp_width -1: 0]    data;
  reg                     Load_regs, Shift_left, Incr_R2, clock, reset_b;

  Datapath_BEH M0 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock, reset_b);


  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    data = 8'haa;
    Load_regs = 0;
    Incr_R2 = 0;
    Shift_left = 0;
    #10 Load_regs = 1;
    #20 Load_regs = 0;
    #50 Incr_R2 = 1;
    #120 Incr_R2 = 0;
    #90 Shift_left = 1;
    #200 Shift_left = 0;
  join
endmodule
```

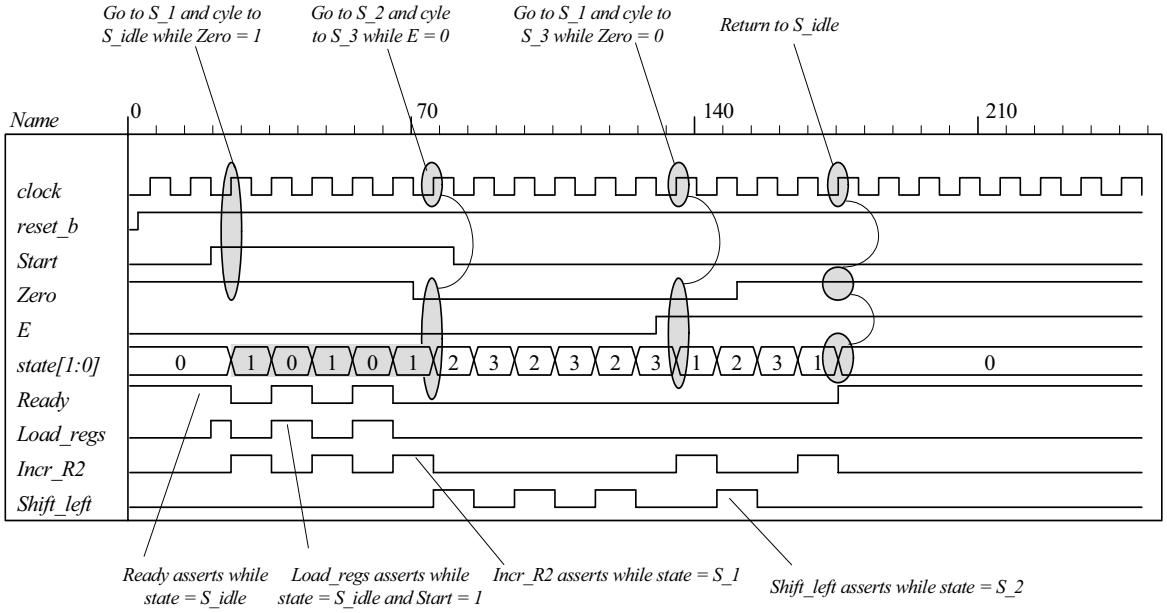
Note: The simulation results show tests of the operations of the datapath independent of the control unit, so *count* does not represent the number of ones in the *data*.



(b)

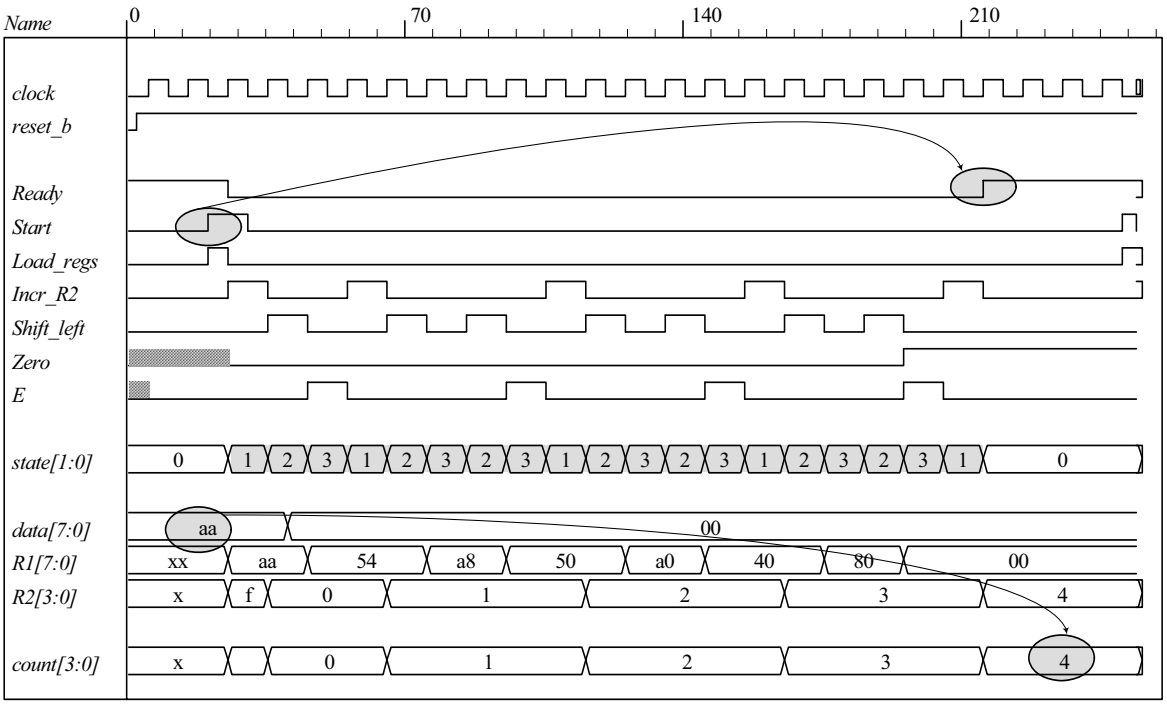
```
// Control Unit
module Controller_BEH (
  output      Ready,
  output reg  Load_regs,
  output      Incr_R2, Shift_left,
  input       Start, Zero, E, clock, reset_b
);
  parameter S_idle = 0, S_1 = 1, S_2 = 2, S_3 = 3;
  reg [1:0] state, next_state;

  assign Ready = (state == S_idle);
  assign Incr_R2 = (state == S_1);
  assign Shift_left = (state == S_2);

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= S_idle;
    else state <= next_state;

  always @ (state, Start, Zero, E) begin
    Load_regs = 0;
    case (state)
      S_idle:   if (Start) begin Load_regs = 1; next_state = S_1; end
                else next_state = S_idle;
      S_1:      if (Zero) next_state = S_idle; else next_state = S_2;

      S_2:      next_state = S_3;
      S_3:      if (E) next_state = S_1; else next_state = S_2;
    endcase
  end
```

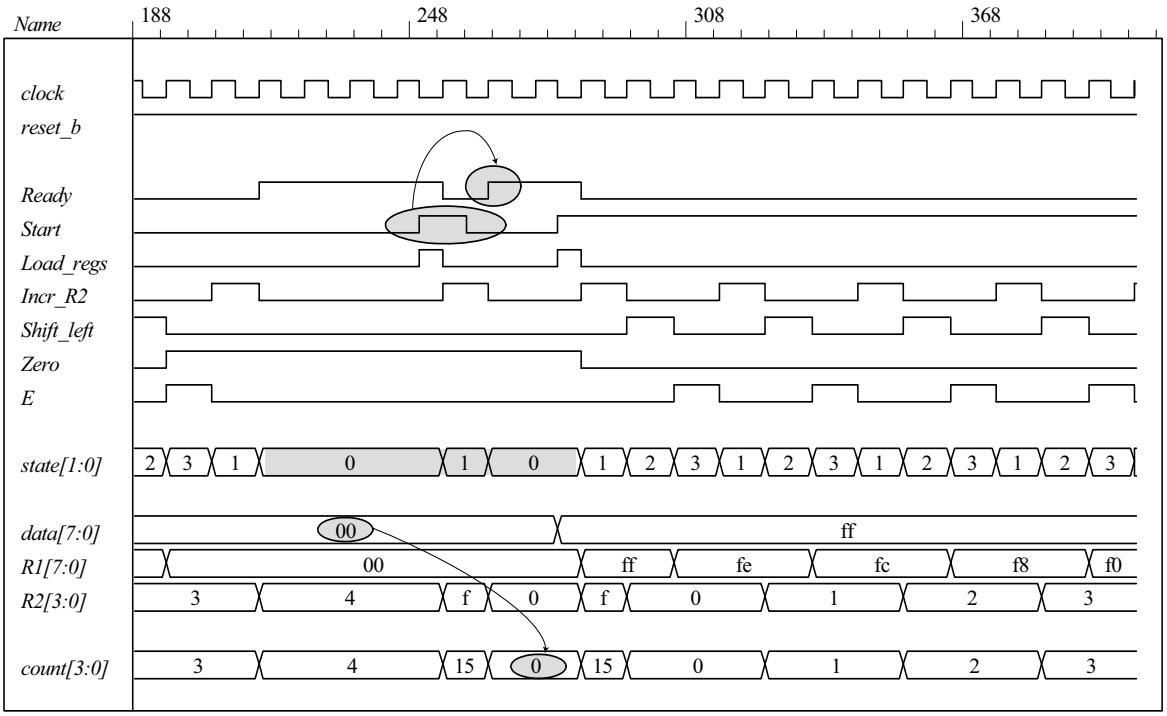
```
endmodule

// Test plan for Control Unit
// Verify that state enters S_idle with reset_b asserted.
// With reset_b de-asserted, verify that state enters S_1 and asserts Load_Regs when
// Start is asserted.
// Verify that Incr_R2 is asserted in S_1.
// Verify that state returns to S_idle from S_1 if Zero is asserted.
// Verify that state goes to S_2 if Zero is not asserted.
// Verify that Shift_left is asserted in S_2.
// Verify that state goes to S_3 from S_2 unconditionally.
//  Verify that state returns to S_2 from S_3 id E is not asserted.
// Verify that state goes to S_1 from S_3 if E is asserted.

// Test bench for Control Unit

module t_Control_Unit ();
  wire Ready, Load_regs, Incr_R2, Shift_left;
  reg Start, Zero, E, clock, reset_b;

  Controller_BEH M0 (Ready, Load_regs, Incr_R2, Shift_left, Start, Zero, E, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    Zero = 1;
    E = 0;
    Start = 0;
    #20 Start = 1;   // Cycle from S_idle to S_1
    #80 Start = 0;
    #70 Zero = 0;   // S_idle to S_1 to S_2 to S_3 and cycle to S_2.
    #130 E = 1; // Cycle to S_3 to S_1 to S_2 to S_3
    #150 Zero = 1; // Return to S_idle
  join
endmodule
```

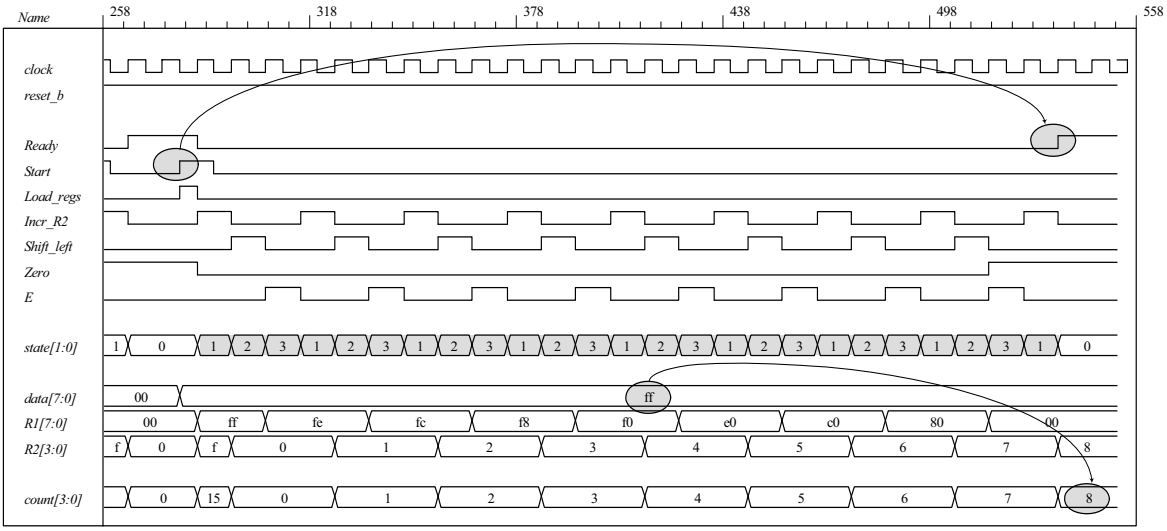


(c)

```
// Integrated system
module Count_Ones_BEH_BEH
# (parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0]      count,
  input [dp_width -1: 0]    data,
  input                Start, clock, reset_b
);
  wire  Load_regs, Incr_R2, Shift_left, Zero, E;
  Controller_BEH M0 (Ready, Load_regs, Incr_R2, Shift_left, Start, Zero, E, clock, reset_b);
  Datapath_BEH M1 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock, reset_b);
endmodule

// Test plan for integrated system
// Test for data values of 8'haa, 8'h00, 8'hff.

// Test bench for integrated system

module t_count_Ones_BEH_BEH ();
  parameter dp_width = 8, R2_width = 4;
  wire [R2_width -1: 0] count;
  reg [dp_width -1: 0] data;
  reg Start, clock, reset_b;

  Count_Ones_BEH_BEH M0 (count, data, Start, clock, reset_b);
  initial #700 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    data = 8'haa;         // Expect count = 4
    Start = 0;
    #20 Start = 1;
    #30 Start = 0;
    #40 data = 8'b00;   // Expect count = 0
    #250 Start = 1;
    #260 Start = 0;
    #280 data = 8'hff;
    #280 Start = 1;
    #290 Start = 0;
  join
endmodule
```







(d)

```
// One-Hot Control unit

module Controller_BEH_1Hot
(
  output       Ready,
  output reg   Load_regs,
  output       Incr_R2, Shift_left,
  input        Start, Zero, E, clock, reset_b
);
  parameter S_idle = 4'b001, S_1 = 4'b0010, S_2 = 4'b0100, S_3 = 4'b1000;
  reg [3:0] state, next_state;

  assign Ready = (state == S_idle);
  assign Incr_R2 = (state == S_1);
  assign Shift_left = (state == S_2);

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= S_idle;
    else state <= next_state;

  always @ (state, Start, Zero, E) begin
    Load_regs = 0;
    case (state)
      S_idle: if (Start) begin Load_regs = 1; next_state = S_1; end
              else next_state = S_idle;
      S_1:    if (Zero) next_state = S_idle; else next_state = S_2;

      S_2:    next_state = S_3;
      S_3:    if (E) next_state = S_1; else next_state = S_2;
    endcase
  end
endmodule
```

Note: Test plan, test bench and simulation results are same as (b), but with states numbered with one-hot codes.

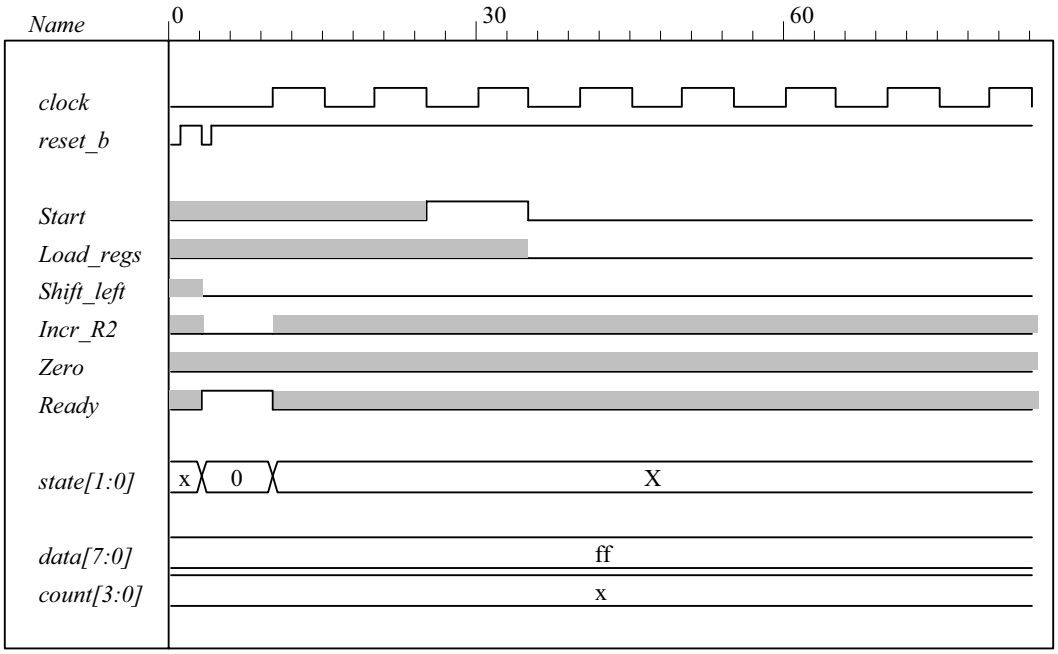
(e)

```
// Integrated system with one-hot controller
```

```
module Count_Ones_BEH_1Hot
# (parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0]     count,
  input [dp_width -1: 0]      data,
  input                       Start, clock, reset_b
);
  wire  Load_regs, Incr_R2, Shift_left, Zero, E;
  Controller_BEH_1Hot M0 (Ready, Load_regs, Incr_R2, Shift_left, Start, Zero, E, clock, reset_b);
  Datapath_BEH M1 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock, reset_b);
endmodule
```

Note: Test plan, test bench and simulation results are same as (c), but with states numbered with one-hot codes.

8.35 Note: Signal *Start* is initialized to 0 when the simulation begins. Otherwise, the state of the structural model will become X at the first clock after the reset condition is deasserted, with Start and Load_Regs having unknown values. In this condition the structural model cannot operate correctly.



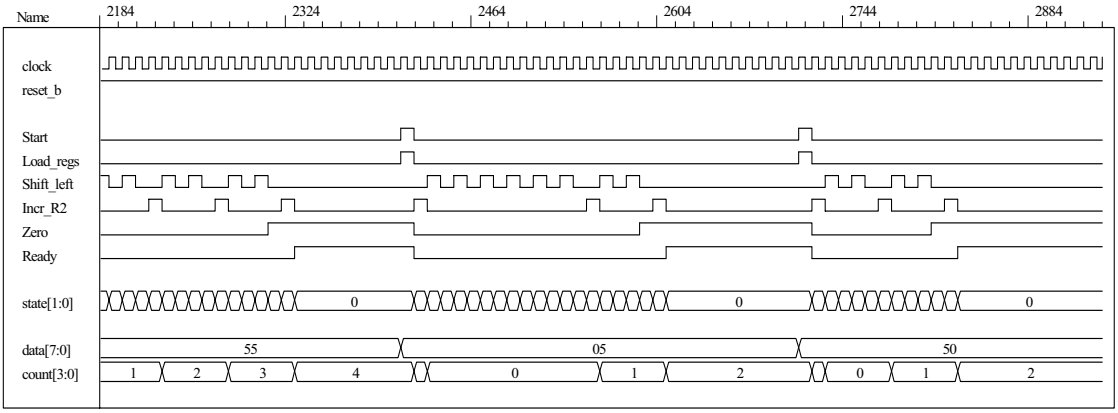

```
module Count_Ones_STR_STR (count, Ready, data, Start, clock, reset_b);
// Mux – decoder implementation of control logic
// controller is structural
// datapath is structural

  parameter   R1_size = 8, R2_size = 4;
  output      [R2_size -1: 0]   count;
  output                        Ready;
  input       [R1_size -1: 0]   data;
  input                         Start, clock, reset_b;
  wire                          Load_regs, Shift_left, Incr_R2, Zero, E;

 Controller_STR   M0 (Ready, Load_regs, Shift_left, Incr_R2, Start, E, Zero, clock, reset_b);
 Datapath_STR   M1 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock);

endmodule
```

```
module Controller_STR (Ready, Load_regs, Shift_left, Incr_R2, Start, E, Zero, clock, reset_b);
  output       Ready;
  output       Load_regs, Shift_left, Incr_R2;
  input        Start;
  input        E, Zero;
  input        clock, reset_b;
  supply0      GND;
  supply1      PWR;
  parameter    S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;  // Binary code
  wire         Load_regs, Shift_left, Incr_R2;
  wire         G0, G0_b, D_in0, D_in1, G1, G1_b;
  wire         Zero_b = ~Zero;

  wire         E_b = ~E;
  wire [1:0]   select = {G1, G0};
  wire [0:3]   Decoder_out;

  assign           Ready = ~Decoder_out[0];
  assign           Incr_R2 = ~Decoder_out[1];
  assign           Shift_left = ~Decoder_out[2];
  and              (Load_regs, Ready, Start);
  mux_4x1_beh      Mux_1 (D_in1, GND, Zero_b, PWR, E_b, select);
  mux_4x1_beh      Mux_0     (D_in0, Start, GND, PWR, E, select);
  D_flip_flop_AR_b M1     (G1, G1_b, D_in1, clock, reset_b);
  D_flip_flop_AR_b M0     (G0, G0_b, D_in0, clock, reset_b);
  decoder_2x4_df M2     (Decoder_out, G1, G0, GND);

endmodule

module Datapath_STR (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock);
  parameter      R1_size = 8, R2_size = 4;
  output  [R2_size -1: 0]  count;
  output                   E, Zero;
  input [R1_size -1: 0]    data;
  input                    Load_regs, Shift_left, Incr_R2, clock;
  wire [R1_size -1: 0] R1;
  supply0                  Gnd;
  supply1                  Pwr;
  assign                   Zero = (R1 == 0);

 Shift_Reg            M1     (R1, data, Gnd, Shift_left, Load_regs, clock, Pwr);
 Counter              M2     (count, Load_regs, Incr_R2, clock, Pwr);
  D_flip_flop_AR      M3     (E, w1, clock, Pwr);
  and              (          w1, R1[R1_size -1], Shift_left);
endmodule

module Shift_Reg (R1, data, SI_0, Shift_left, Load_regs, clock, reset_b);
  parameter                R1_size = 8;
  output [R1_size -1: 0]   R1;
  input [R1_size -1: 0]    data;
  input                    SI_0, Shift_left, Load_regs;
  input                    clock, reset_b;
  reg [R1_size -1: 0]      R1;

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) R1 <= 0;
    else begin
      if (Load_regs) R1 <= data; else
        if (Shift_left) R1 <= {R1[R1_size -2:0], SI_0}; end
endmodule
```

```
module Counter (R2, Load_regs, Incr_R2, clock, reset_b);
  parameter               R2_size = 4;
  output [R2_size -1: 0]  R2;
  input                   Load_regs, Incr_R2;
  input                   clock, reset_b;
  reg [R2_size -1: 0]     R2;

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) R2 <= 0;
    else if (Load_regs) R2 <= {R2_size {1'b1}};  // Fill with 1
      else if (Incr_R2 == 1) R2 <= R2 + 1;
endmodule

module D_flip_flop_AR (Q, D, CLK, RST);
  output   Q;
  input    D, CLK, RST;
  reg      Q;

  always @ (posedge CLK, negedge RST)
    if (RST == 0) Q <= 1'b0;
    else Q <= D;
endmodule

module D_flip_flop_AR_b (Q, Q_b, D, CLK, RST);
  output   Q, Q_b;
  input    D, CLK, RST;
  reg      Q;
  assign       Q_b = ~Q;
  always @ (posedge CLK, negedge RST)
    if (RST == 0) Q <= 1'b0;
    else Q <= D;
endmodule

// Behavioral description of 4-to-1 line multiplexer
// Verilog 2005 port syntax

module mux_4x1_beh
( output reg   m_out,
  input        in_0, in_1, in_2, in_3,
  input [1: 0] select
);

  always @ (in_0, in_1, in_2, in_3, select)   // Verilog 2005 syntax
    case (select)
      2'b00: m_out = in_0;
      2'b01: m_out = in_1;
      2'b10: m_out = in_2;
      2'b11: m_out = in_3;
    endcase
  endmodule
// Dataflow description of 2-to-4-line decoder
// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (D, A, B, enable);
  output   [0: 3]   D;

  input             A, B;
  input             enable;
```

```
  assign    D[0] = ~(~A & ~B & ~enable),
            D[1] = ~(~A & B & ~enable),
            D[2] = ~(A & ~B & ~enable),
            D[3] = ~(A & B & ~enable);
endmodule

module t_Count_Ones;
  parameter R1_size = 8, R2_size = 4;
  wire  [R2_size -1: 0]     R2;

  wire  [R2_size -1: 0]     count;
  wire                      Ready;
  reg   [R1_size -1: 0]     data;
  reg                       Start, clock, reset_b;
  wire [1: 0] state;      // Use only for debug
  assign state = {M0.M0.G1, M0.M0.G0};

  Count_Ones_STR_STR M0 (count, Ready, data, Start, clock, reset_b);

  initial #4000 $finish;
  initial begin clock = 0; #5 forever #5 clock = ~clock; end
  initial fork
    Start = 0;

    #1 reset_b = 1;
    #3 reset_b = 0;
    #4 reset_b = 1;
     data = 8'Hff;
    # 25 Start = 1;
    # 35  Start = 0;
    #310 data = 8'h0f;
    #310 Start = 1;
    #320 Start = 0;
    #610 data = 8'hf0;
    #610 Start = 1;
    #620 Start = 0;
    #910 data = 8'h00;
    #910 Start = 1;
    #920 Start = 0;
    #1210 data = 8'haa;
    #1210 Start = 1;
    #1220 Start = 0;
    #1510 data = 8'h0a;
    #1510 Start = 1;
    #1520 Start = 0;
    #1810 data = 8'ha0;
    #1810 Start = 1;
    #1820 Start = 0;
    #2110 data = 8'h55;
    #2110 Start = 1;
    #2120 Start = 0;
    #2410 data = 8'h05;
    #2410 Start = 1;
    #2420 Start = 0;
    #2710 data = 8'h50;
    #2710 Start = 1;
    #2720 Start = 0;
    #3010 data = 8'ha5;
    #3010 Start = 1;
    #3020 Start = 0;
    #3310 data = 8'h5a;
    #3310 Start = 1;
```

```
  #3320 Start = 0;
 join
endmodule
```



8.36 Note: See Prob. 8.35 for a behavioral model of the datapath unit, Prob. 8.36d for a one-hot control unit.

(a) T_0, T_1, T_2, T_3 be asserted when the state is in *S_idle*, *S*_1, *S*_2, and *S*_3, respectively. Let *D*0, *D*1, *D*2, and *D*3 denote the inputs to the one-hot flip-flops.

$D_0 = T_0\, Start' + T_1\, Zero$
$D_1 = T_0\, Start + T_3\, E$
$D_2 = T_1\, Zero' + T_3\, E'$
$D_3 = T_2$

(b) Gate-level one-hot controller

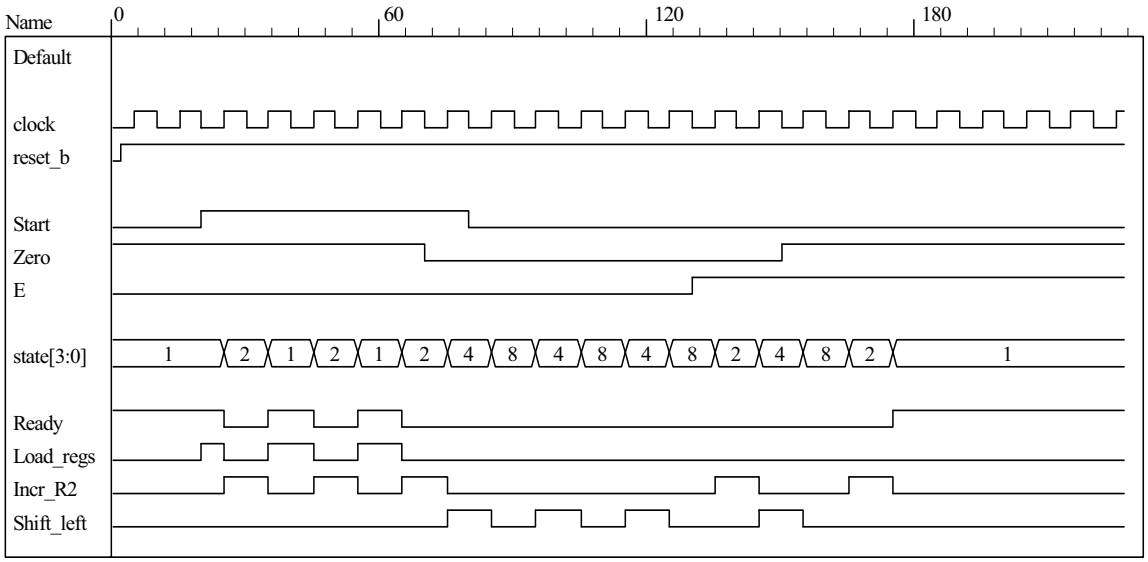
```
module Controller_Gates_1Hot
(
  output     Ready,
  output     Load_regs, Incr_R2, Shift_left,
  input      Start, Zero, E, clock, reset_b
);
  wire w1, w2, w3, w4, w5, w6;
  wire T0, T1, T2, T3;
  wire set;
  assign Ready = T0;
  assign Incr_R2 = T1;
  assign Shift_left = T2;
  and (Load_regs, T0, Start);
  not (set, reset_b);
  DFF_S M0 (T0, D0, clock, set);        // Note: reset action must initialize S_idle = 4'b0001
  DFF M1 (T1, D1, clock, reset_b);
  DFF M2 (T2, D2, clock, reset_b);
  DFF M3 (T3, D3, clock, reset_b);

  not (Start_b, Start);
  and (w1, T0, Start_b);
  and (w2, T1, Zero);
  or (D0, w1, w2);
```

```
  and (w3, T0, Start);
  and (w4, T3, E);
  or (D1, w3, w4);

  not (Zero_b, Zero);
  not (E_b, E);
  and (w5, T1, Zero_b);
  and (w6, T3, E_b);
  or (D2, w5, w6);

  buf (D3, T2);
endmodule

module DFF (output reg Q, input D, clock, reset_b);
  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) Q <= 0;
    else Q <= D;
endmodule
module DFF_S (output reg Q, input D, clock, set);
  always @ (posedge clock, posedge set)
    if (set == 1) Q <= 1;
    else Q <= D;
endmodule
```

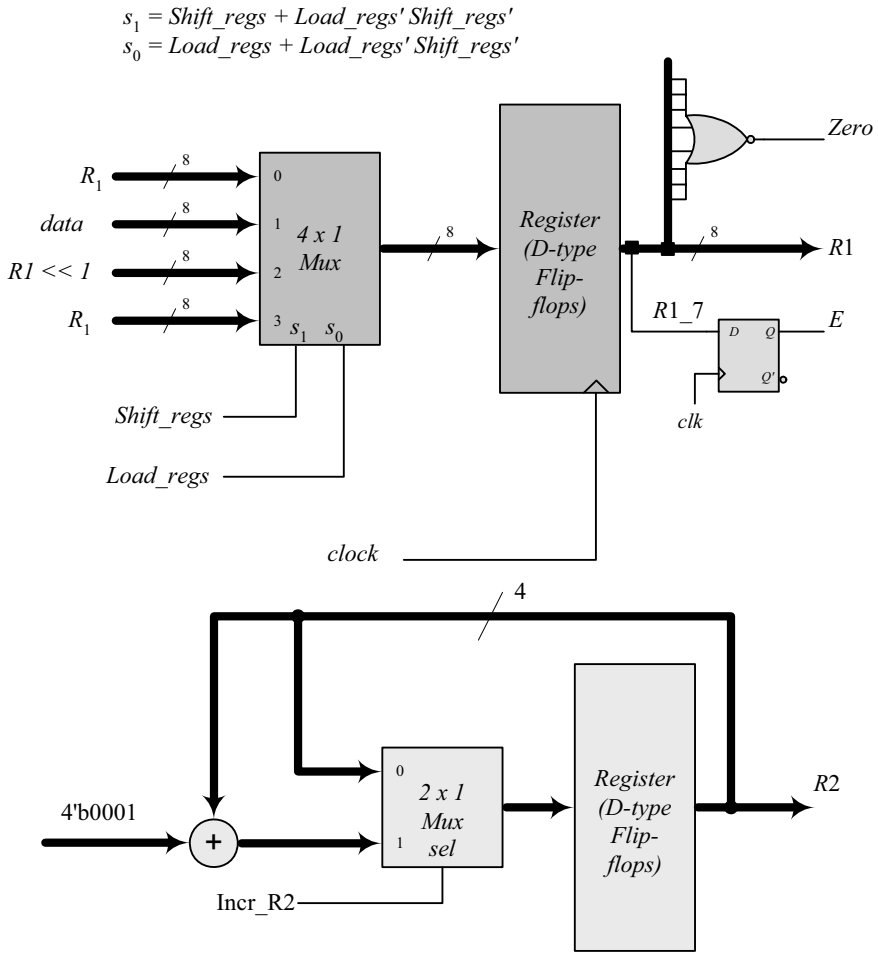
(c)

```
// Test plan for Control Unit
// Verify that state enters S_idle with reset_b asserted.
// With reset_b de-asserted, verify that state enters S_1 and asserts Load_Regs when
// Start is asserted.
// Verify that Incr_R2 is asserted in S_1.
// Verify that state returns to S_idle from S_1 if Zero is asserted.
// Verify that state goes to S_2 if Zero is not asserted.
// Verify that Shift_left is asserted in S_2.
// Verify that state goes to S_3 from S_2 unconditionally.
//  Verify that state returns to S_2 from S_3 id E is not asserted.
// Verify that state goes to S_1 from S_3 if E is asserted.

// Test bench for One-Hot Control Unit

module t_Control_Unit ();
  wire Ready, Load_regs, Incr_R2, Shift_left;
  reg Start, Zero, E, clock, reset_b;
  wire [3: 0] state = {M0.T3, M0.T2, M0.T1, M0.T0};       // Observe one-hot state bits
  Controller_Gates_1Hot M0 (Ready, Load_regs, Incr_R2, Shift_left, Start, Zero, E, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    Zero = 1;
    E = 0;
    Start = 0;
    #20 Start = 1;// Cycle from S_idle to S_1
    #80 Start = 0;
    #70 Zero = 0;      // S_idle to S_1 to S_2 to S_3 and cycle to S_2.
    #130 E = 1;        // Cycle to S_3 to S_1 to S_2 to S_3
    #150 Zero = 1;     // Return to S_idle
  join
endmodule
```

Note: simulation results match those for Prob. 8.34(d). See Prob. 8.34(c) for annotations.



(d) Datapath unit detail:

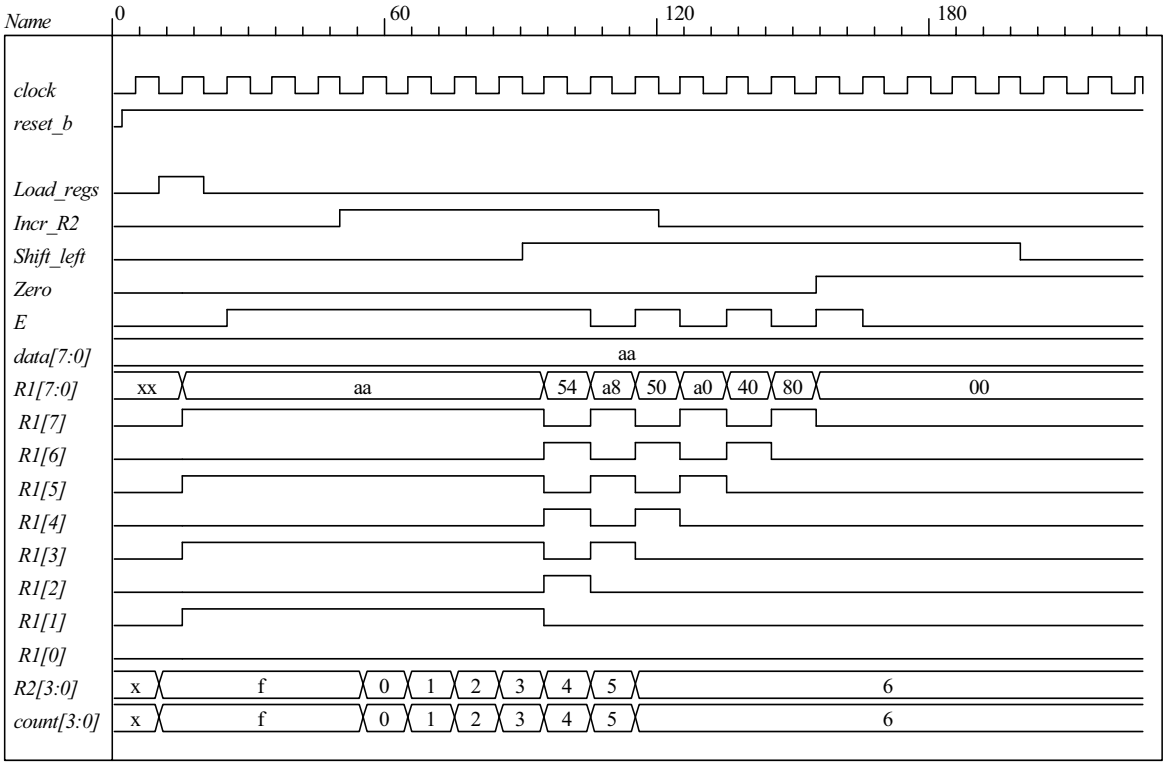


```
//  Datapath unit – structural model
module Datapath_STR
#(parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0] count, output E, output Zero, input [dp_width -1: 0] data,
  input Load_regs, Shift_left, Incr_R2, clock, reset_b);
  supply1 pwr;
  supply0 gnd;
  wire [dp_width -1: 0] R1_Dbus, R1;
  wire [R2_width -1: 0] R2_Dbus;
  wire DR1_0, DR1_1, DR1_2, DR1_3, DR1_4, DR1_5, DR1_6, DR1_7;
  wire R1_0, R1_1, R1_2, R1_3, R1_4, R1_5, R1_6, R1_7;
  wire R2_0, R2_1, R2_2, R2_3;
  wire [R2_width -1: 0] R2 = {R2_3, R2_2, R2_1, R2_0};
  assign count = {R2_3, R2_2, R2_1, R2_0};
  assign R1 = { R1_7, R1_6, R1_5, R1_4, R1_3, R1_2, R1_1, R1_0};
  assign DR1_0 = R1_Dbus[0];
  assign DR1_1 = R1_Dbus[1];
  assign DR1_2 = R1_Dbus[2];
  assign DR1_3 = R1_Dbus[3];
  assign DR1_4 = R1_Dbus[4];
  assign DR1_5 = R1_Dbus[5];
  assign DR1_6 = R1_Dbus[6];
  assign DR1_7 = R1_Dbus[7];

  nor (Zero, R1_0, R1_1, R1_2, R1_3, R1_4, R1_5, R1_6, R1_7);
  DFF D_E (E, R1_7, clock, pwr);

  DFF DF_0 (R1_0, DR1_0, clock, pwr);      // Disable reset
  DFF DF_1 (R1_1, DR1_1, clock, pwr);
  DFF DF_2 (R1_2, DR1_2, clock, pwr);
  DFF DF_3 (R1_3, DR1_3, clock, pwr);
  DFF DF_4 (R1_4, DR1_4, clock, pwr);
  DFF DF_5 (R1_5, DR1_5, clock, pwr);
  DFF DF_6 (R1_6, DR1_6, clock, pwr);
  DFF DF_7 (R1_7, DR1_7, clock, pwr);

  DFF_S DR_0 (R2_0, DR2_0, clock, Load_regs);  // Load_regs (set) drives R2 to all ones
  DFF_S DR_1 (R2_1, DR2_1, clock, Load_regs);
  DFF_S DR_2 (R2_2, DR2_2, clock, Load_regs);
  DFF_S DR_3 (R2_3, DR2_3, clock, Load_regs);

  assign DR2_0 = R2_Dbus[0];
  assign DR2_1 = R2_Dbus[1];
  assign DR2_2 = R2_Dbus[2];
  assign DR2_3 = R2_Dbus[3];

  wire [1: 0] sel = {Shift_left, Load_regs};
  wire [dp_width -1: 0] R1_shifted = {R1_6, R1_5, R1_4, R1_3, R1_2, R1_1, R1_0, 1'b0};
  wire [R2_width -1: 0] sum = R2 + 4'b0001;

  Mux8_4_x_1 M0 (R1_Dbus, R1, data, R1_shifted, R1, sel);
  Mux4_2_x_1 M1 (R2_Dbus, R2, sum, Incr_R2);
endmodule
```

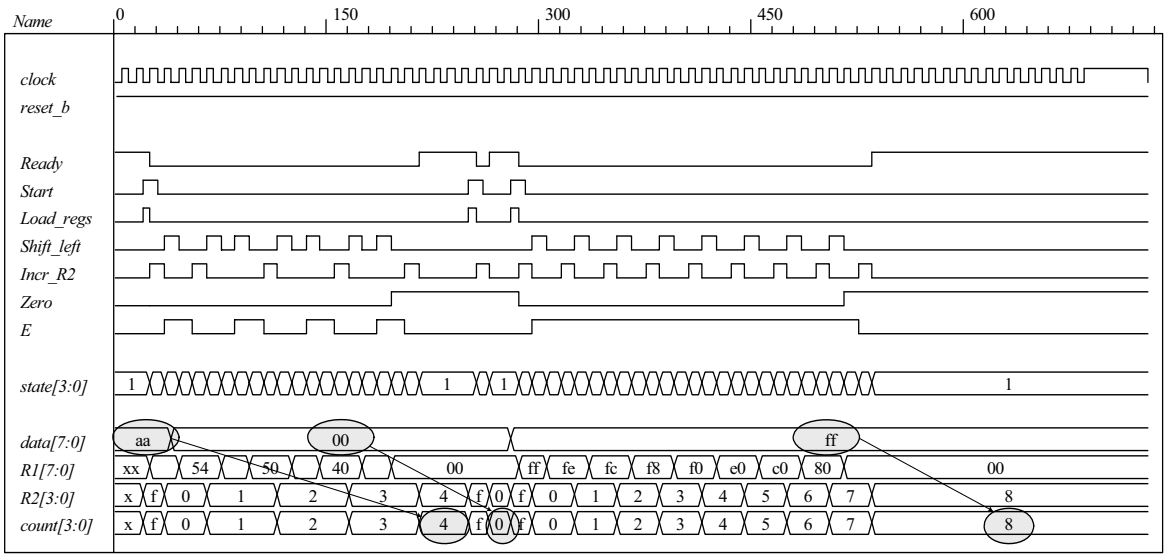
```
  module Mux8_4_x_1 #(parameter dp_width = 8) (output reg [dp_width -1: 0] mux_out,
    input [dp_width -1: 0] in0, in1, in2, in3, input [1: 0] sel);
    always @ (in0, in1, in2, in3, sel)
      case (sel)
        2'b00:  mux_out = in0;
        2'b01:  mux_out = in1;
        2'b10:  mux_out = in2;
        2'b11:  mux_out = in3;
      endcase
  endmodule

  module Mux4_2_x_1 #(parameter dp_width = 4) (output [dp_width -1: 0] mux_out,
    input [dp_width -1: 0] in0, in1, input  sel);
    assign mux_out = sel ? in1: in0;
endmodule

// Test Plan for Datapath Unit:
// Demonstrate action of Load_regs
//       R1 gets data, R2 gets all ones
// Demonstrate action of Incr_R2
// Demonstrate action of Shift_left and detect E

// Test bench for datapath
module t_Datapath_Unit
#(parameter dp_width = 8, R2_width = 4)
( );
  wire [R2_width -1: 0]  count;
  wire                   E, Zero;
  reg [dp_width -1: 0]    data;
  reg                    Load_regs, Shift_left, Incr_R2, clock, reset_b;
  Datapath_STR M0 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    data = 8'haa;
    Load_regs = 0;
    Incr_R2 = 0;
    Shift_left = 0;
    #10 Load_regs = 1;
    #20 Load_regs = 0;
    #50 Incr_R2 = 1;
    #120 Incr_R2 = 0;
    #90 Shift_left = 1;
    #200 Shift_left = 0;
  join
endmodule

// Integrated system
module Count_Ones_Gates_1_Hot_STR
# (parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0]    count,
  input [dp_width -1: 0] data,
  input             Start, clock, reset_b
);
  wire     Load_regs, Incr_R2, Shift_left, Zero, E;
  Controller_Gates_1Hot M0 (Ready, Load_regs, Incr_R2, Shift_left, Start, Zero, E, clock, reset_b);
  Datapath_STR M1 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock, reset_b);
endmodule
```

```
// Test plan for integrated system
// Test for data values of 8'haa, 8'h00, 8'hff.

// Test bench for integrated system

module t_count_Ones_Gates_1_Hot_STR ();
  parameter dp_width = 8, R2_width = 4;
  wire [R2_width -1: 0] count;
  reg [dp_width -1: 0] data;
  reg Start, clock, reset_b;
  wire [3: 0] state = {M0.M0.T3, M0.M0.T2, M0.M0.T1, M0.M0.T0};

  Count_Ones_Gates_1_Hot_STR M0 (count, data, Start, clock, reset_b);
  initial #700 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    data = 8'haa;       // Expect count = 4
    Start = 0;
    #20 Start = 1;
    #30 Start = 0;
    #40 data = 8'b00;      // Expect count = 0
    #250 Start = 1;
    #260 Start = 0;
    #280 data = 8'hff;
    #280 Start = 1;
    #290 Start = 0;
  join
endmodule
```

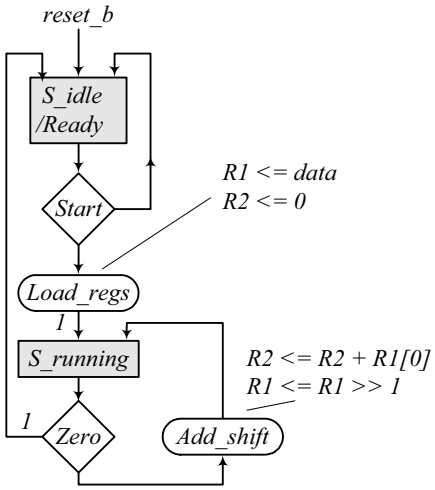
Note: The simulation results show tests of the operations of the datapath independent of the control unit, so count does not represent the number of ones in the data.



Simulations results for the integrated system match those shown in Prob. 8.34(e). See those results for additional annotation.



8.37 **(a)** ASMD chart:

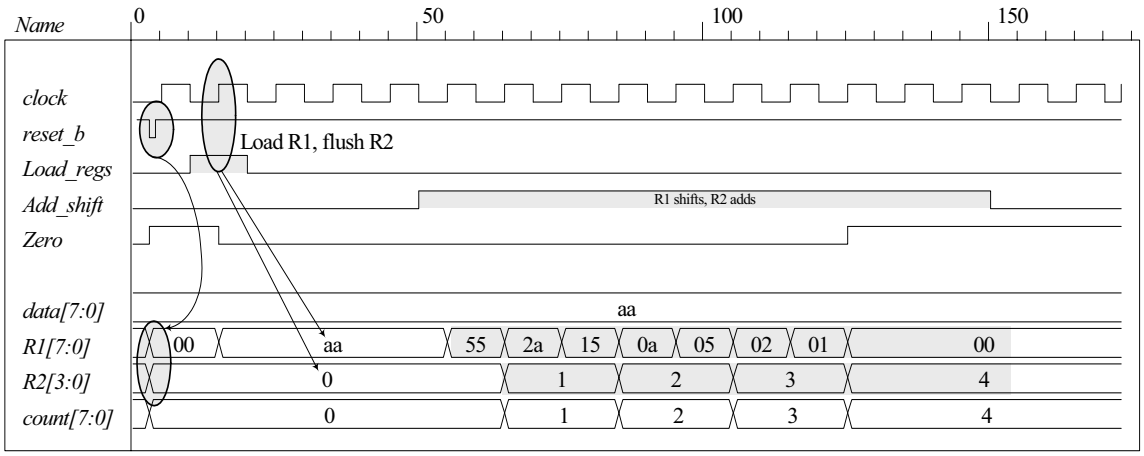


(b) RTL model:

```
module Datapath_Unit_2_Beh #(parameter dp_width = 8, R2_width = 4)
(
 output [R2_width -1: 0]  count,
 output                   Zero,
 input [dp_width -1: 0]   data,
 input                    Load_regs, Add_shift, clock, reset_b
);
 reg [dp_width -1: 0]     R1;
 reg [R2_width -1: 0]     R2;
 assign count = R2;
```

```
  assign Zero = ~|R1;
  always @ (posedge clock, negedge reset_b)
    begin
      if (reset_b == 0) begin R1 <= 0; R2 <= 0; end else begin
        if (Load_regs) begin R1 <= data; R2 <= 0; end
        if (Add_shift) begin R1 <= R1 >> 1; R2 <= R2 + R1[0]; end // concurrent operations
      end
    end
endmodule

// Test plan for datapath unit
// Verify active-low reset action
// Test for action of Add_shift
// Test for action of Load_regs

module t_Datapath_Unit_2_Beh();
  parameter R1_size = 8, R2_size = 4;
  wire [R2_size -1: 0]  count;
  wire                  Zero;
  reg  [R1_size -1: 0]  data;
  reg                   Load_regs, Add_shift, clock, reset_b;

  Datapath_Unit_2_Beh M0 (count, Zero, data, Load_regs, Add_shift, clock, reset_b);

  initial #1000 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
    #1 reset_b = 1;
    #3 reset_b = 0;
    #4 reset_b = 1;
  join
  initial fork
    data = 8'haa;
    Load_regs = 0;
    Add_shift = 0;
    #10 Load_regs = 1;
    #20 Load_regs = 0;
    #50 Add_shift = 1;
    #150 Add_shift  = 0;
  join
endmodule
```

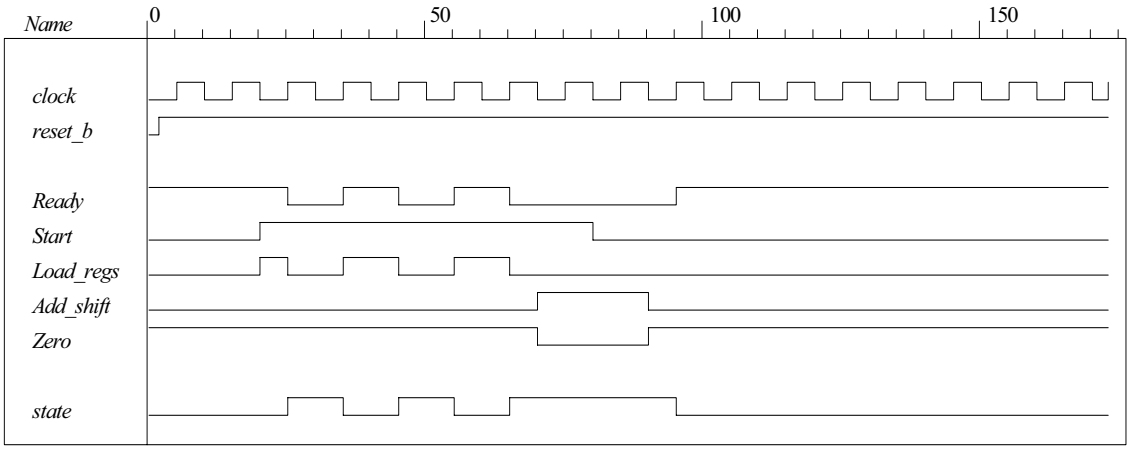
Note that the operations of the datapath unit are tested independent of the controller, so the actions of *Load_regs* and *add_shift* and the value of *count* do not correspond to *data*.




```
module Controller_2_Beh (
  output       Ready,
  output reg   Load_regs,
               Add_shift,
  input        Start, Zero, clock, reset_b
);
  parameter    S_idle = 0, S_running = 1;
  reg          state, next_state;
  assign       Ready = (state == S_idle);

  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) state <= S_idle;
    else state <= next_state;

  always @ (state, Start, Zero) begin
    next_state = S_idle;
    Load_regs = 0;
    Add_shift = 0;

    case (state)
      S_idle:      if (Start) begin Load_regs = 1; next_state = S_running; end
      S_running:   if (Zero) next_state = S_idle;
                   else begin Add_shift = 1; next_state = S_running; end
    endcase
  end
endmodule

module t_Controller_2_Beh ();
  wire Ready, Load_regs, Add_shift;
  reg Start, Zero, clock, reset_b;

  Controller_2_Beh M0 (Ready, Load_regs, Add_shift, Start, Zero, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    Zero = 1;
    Start = 0;
    #20 Start = 1;   // Cycle from S_idle to S_1
    #80 Start = 0;
    #70 Zero = 0;    // S_idle to S_1 to S_idle
```

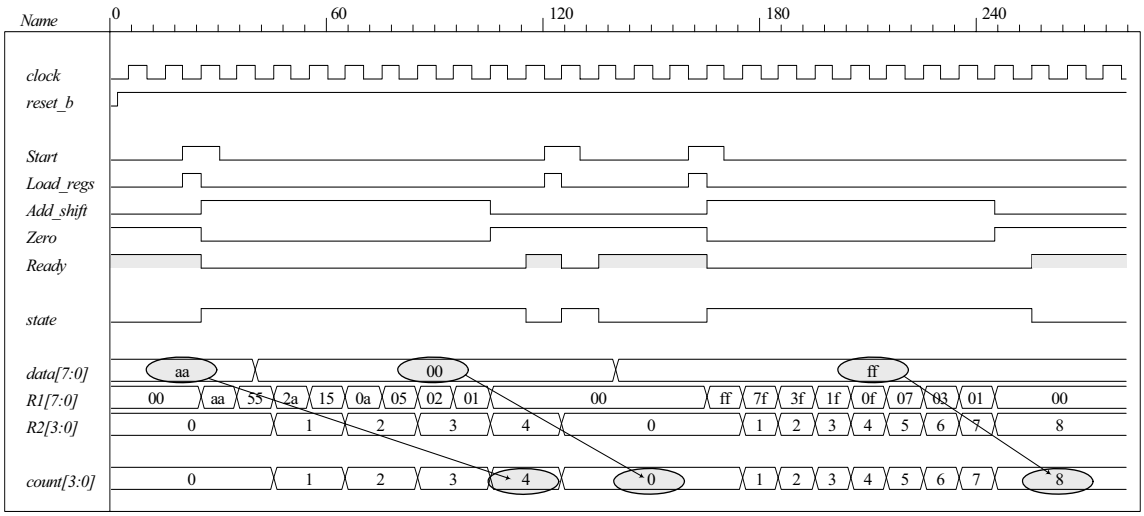
```
   #90 Zero = 1;   // Return to S_idle
  join
endmodule
```

Note: The state transitions and outputs of the controller match the ASMD chart.




```
module Count_of_Ones_2_Beh #(parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0] count,
  output Ready,
  input [dp_width -1: 0] data,
  input Start, clock, reset_b
);
  wire Load_regs, Add_shift, Zero;

  Controller_2_Beh M0 (Ready, Load_regs, Add_shift, Start, Zero, clock, reset_b);
  Datapath_Unit_2_Beh M1 (count, Zero, data, Load_regs, Add_shift, clock, reset_b);
endmodule

// Test plan for integrated system
// Test for data values of 8'haa, 8'h00, 8'hff.

// Test bench for integrated system

module t_Count_Ones_2_Beh ();
  parameter dp_width = 8, R2_width = 4;
  wire [R2_width -1: 0] count;
  reg [dp_width -1: 0] data;
  reg Start, clock, reset_b;

  Count_of_Ones_2_Beh M0 (count, Ready, data, Start, clock, reset_b);

  initial #700 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    data = 8'haa;          // Expect count = 4
    Start = 0;
    #20 Start = 1;
    #30 Start = 0;
    #40 data = 8'b00;   // Expect count = 0
    #120 Start = 1;
    #130 Start = 0;
```

```
  #140 data = 8'hff;
  #160 Start = 1;
  #170 Start = 0;

 join
endmodule
```



(c) T_0, T_1 are to be asserted when the state is in *S_idle*, *S_running*, respectively. Let $D0$, $D1$ denote the inputs to the one-hot flip-flops.

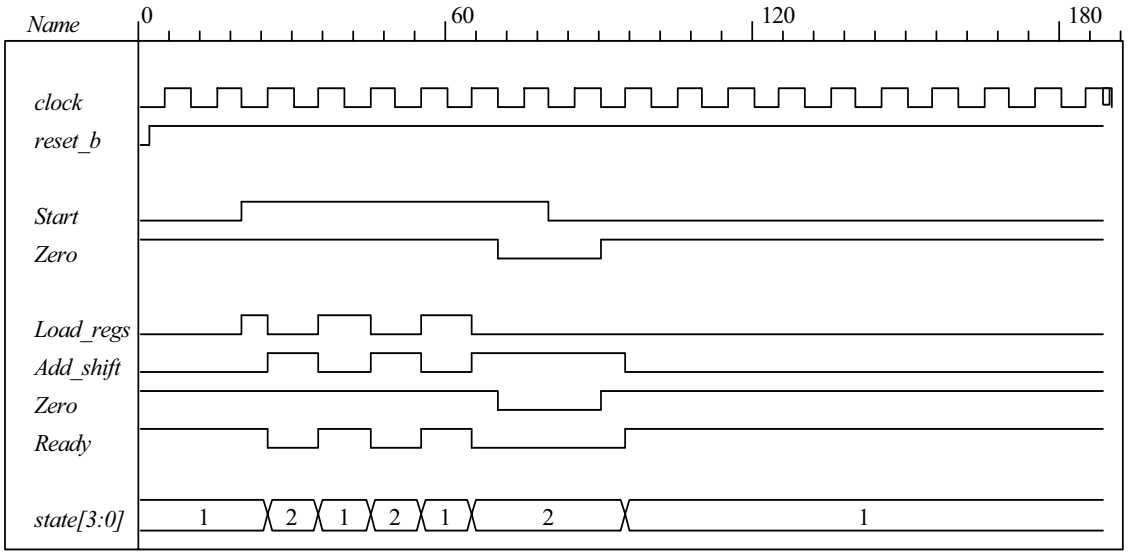
$D_0 = T_0\, Start' + T_1\, Zero$
$D_1 = T_0\, Start + T_1\, E'$

(d) Gate-level one-hot controller

```
module Controller_2_Gates_1Hot
(
  output      Ready, Load_regs, Add_shift,
  input       Start, Zero, clock, reset_b
);
  wire w1, w2, w3, w4;
  wire T0, T1;
  wire set;
  assign Ready = T0;
  assign Add_shift = T1;
  and (Load_regs, T0, Start);
  not (set, reset_b);
  DFF_S M0 (T0, D0, clock, set);     // Note: reset action must initialize S_idle = 2'b01
  DFF M1 (T1, D1, clock, reset_b);

  not (Start_b, Start);
  not (Zero_b, Zero);
  and (w1, T0, Start_b);
  and (w2, T1, Zero);
  or (D0, w1, w2);

  and (w3, T0, Start);
  and (w4, T1, Zero_b);
  or (D1, w3, w4);
endmodule
```

```
module DFF (output reg Q, input D, clock, reset_b);
  always @ (posedge clock, negedge reset_b)
    if (reset_b == 0) Q <= 0;
    else Q <= D;
endmodule
module DFF_S (output reg Q, input D, clock, set);
  always @ (posedge clock, posedge set)
    if (set == 1) Q <= 1;
    else Q <= D;
endmodule

// Test plan for Control Unit
// Verify that state enters S_idle with reset_b asserted.
// With reset_b de-asserted, verify that state enters S_running and asserts Load_Regs when
// Start is asserted.
// Verify that state returns to S_idle from S_running if Zero is asserted.
// Verify that state goes to S_running if Zero is not asserted.

// Test bench for One-Hot Control Unit

module t_Control_Unit ();
  wire          Ready, Load_regs, Add_shift;
  reg           Start, Zero, clock, reset_b;
  wire [3: 0]   state = {M0.T1, M0.T0};       // Observe one-hot state bits

  Controller_2_Gates_1Hot M0 (Ready, Load_regs, Add_shift, Start, Zero, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    Zero = 1;
    Start = 0;
    #20 Start = 1;   // Cycle from S_idle to S_1
    #80 Start = 0;
    #70 Zero = 0;   // S_idle to S_1 to S_idle
    #90 Zero = 1;   // Return to S_idle
  join
endmodule
```

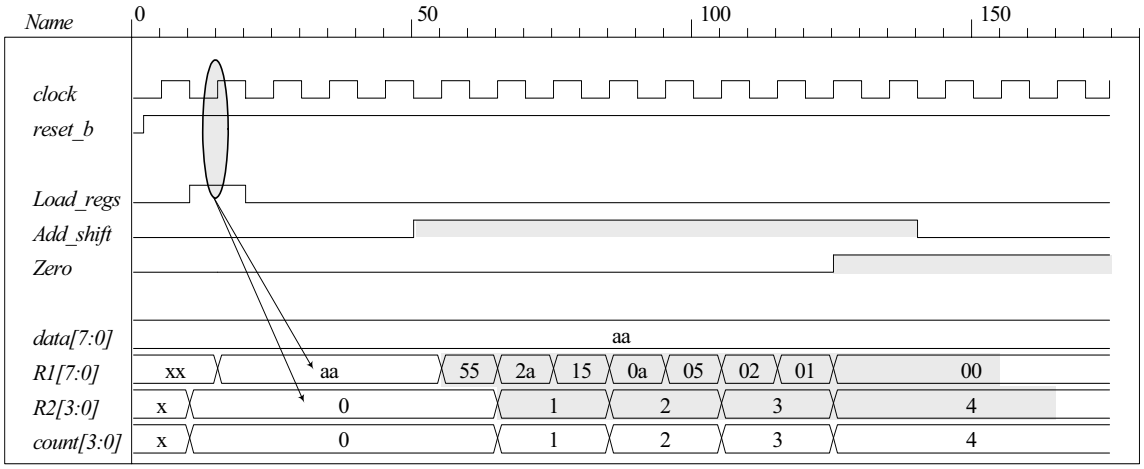
Simulation results show that the controller matches the ASMD chart.



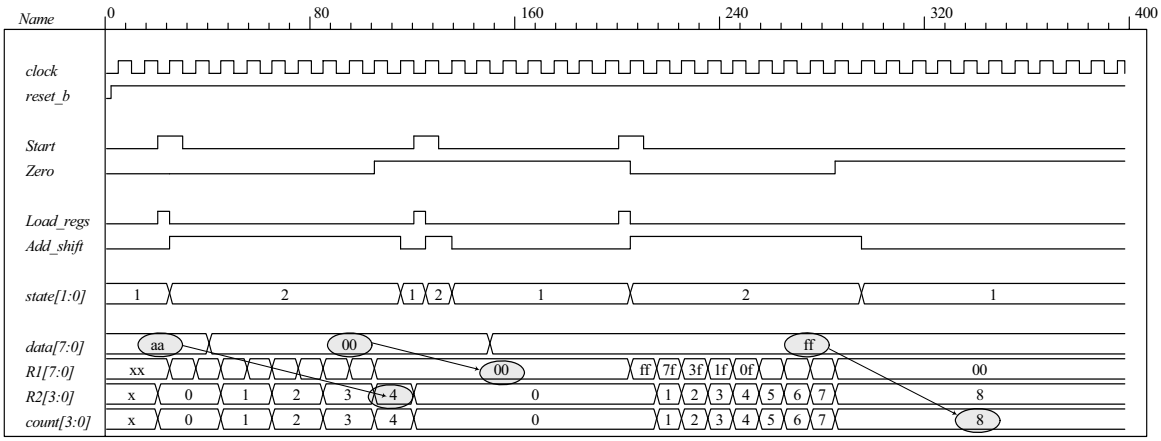

```
//  Datapath unit – structural model

module Datapath_2_STR
#(parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0]      count,
  output                       Zero,
  input [dp_width -1: 0]       data,
  input                        Load_regs, Add_shift, clock, reset_b);
  supply1                      pwr;
  supply0                      gnd;
  wire [dp_width -1: 0] R1_Dbus, R1;
  wire [R2_width -1: 0] R2_Dbus;
  wire DR1_0, DR1_1, DR1_2, DR1_3, DR1_4, DR1_5, DR1_6, DR1_7;
  wire R1_0, R1_1, R1_2, R1_3, R1_4, R1_5, R1_6, R1_7;
  wire R2_0, R2_1, R2_2, R2_3;
  wire [R2_width -1: 0] R2 = {R2_3, R2_2, R2_1, R2_0};
  assign count = {R2_3, R2_2, R2_1, R2_0};
  assign R1 = { R1_7, R1_6, R1_5, R1_4, R1_3, R1_2, R1_1, R1_0};
  assign DR1_0 = R1_Dbus[0];
  assign DR1_1 = R1_Dbus[1];
  assign DR1_2 = R1_Dbus[2];
  assign DR1_3 = R1_Dbus[3];
  assign DR1_4 = R1_Dbus[4];
  assign DR1_5 = R1_Dbus[5];
  assign DR1_6 = R1_Dbus[6];
  assign DR1_7 = R1_Dbus[7];

  nor (Zero, R1_0, R1_1, R1_2, R1_3, R1_4, R1_5, R1_6, R1_7);
  not (Load_regs_b, Load_regs);

  DFF DF_0 (R1_0, DR1_0, clock, pwr);      // Disable reset
  DFF DF_1 (R1_1, DR1_1, clock, pwr);
  DFF DF_2 (R1_2, DR1_2, clock, pwr);
  DFF DF_3 (R1_3, DR1_3, clock, pwr);
  DFF DF_4 (R1_4, DR1_4, clock, pwr);
  DFF DF_5 (R1_5, DR1_5, clock, pwr);
  DFF DF_6 (R1_6, DR1_6, clock, pwr);
  DFF DF_7 (R1_7, DR1_7, clock, pwr);
```

```
  DFF DR_0 (R2_0, DR2_0, clock, Load_regs_b);  // Load_regs (set) drives R2 to all ones
  DFF DR_1 (R2_1, DR2_1, clock, Load_regs_b);
  DFF DR_2 (R2_2, DR2_2, clock, Load_regs_b);
  DFF DR_3 (R2_3, DR2_3, clock, Load_regs_b);

  assign DR2_0 = R2_Dbus[0];
  assign DR2_1 = R2_Dbus[1];
  assign DR2_2 = R2_Dbus[2];
  assign DR2_3 = R2_Dbus[3];

  wire [1: 0]                sel = {Add_shift, Load_regs};
  wire [dp_width -1: 0]      R1_shifted = {1'b0, R1_7, R1_6, R1_5, R1_4, R1_3, R1_2, R1_1};
  wire [R2_width -1: 0]      sum = R2 + {3'b000, R1[0]};

  Mux8_4_x_1 M0 (R1_Dbus, R1, data, R1_shifted, R1, sel);
  Mux4_2_x_1 M1 (R2_Dbus, R2, sum, Add_shift);
endmodule
```

```
  module Mux8_4_x_1 #(parameter dp_width = 8) (output reg [dp_width -1: 0] mux_out,
    input [dp_width -1: 0] in0, in1, in2, in3, input [1: 0] sel);
    always @ (in0, in1, in2, in3, sel)
      case (sel)
        2'b00:  mux_out = in0;
        2'b01:  mux_out = in1;
        2'b10:  mux_out = in2;
        2'b11:  mux_out = in3;
      endcase
  endmodule

  module Mux4_2_x_1 #(parameter dp_width = 4) (output [dp_width -1: 0] mux_out,
    input [dp_width -1: 0] in0, in1, input  sel);
    assign mux_out = sel ? in1: in0;
endmodule

// Test Plan for Datapath Unit:
// Demonstrate action of Load_regs
// R1 gets data, R2 gets all ones
// Demonstrate action of Incr_R2
// Demonstrate action of Add_shift and detect Zero

// Test bench for datapath

module t_Datapath_Unit
#(parameter dp_width = 8, R2_width = 4)
( );
  wire [R2_width -1: 0]  count;
  wire                   Zero;
  reg [dp_width -1: 0]   data;
    reg                  Load_regs, Add_shift, clock, reset_b;

  Datapath_2_STR M0 (count,  Zero, data, Load_regs, Add_shift, clock, reset_b);

  initial #250 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    data = 8'haa;
    Load_regs = 0;
    Add_shift = 0;
    #10 Load_regs = 1;
    #20 Load_regs = 0;
    #50 Add_shift = 1;
    #140 Add_shift = 0;
  join
endmodule
```



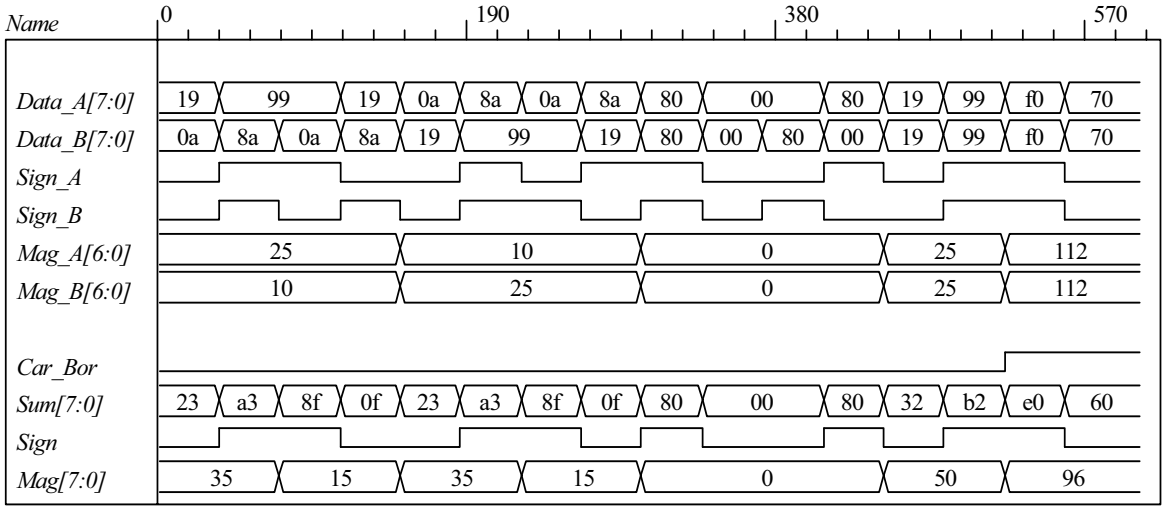

```
// Integrated system

module Count_Ones_2_Gates_1Hot_STR
# (parameter dp_width = 8, R2_width = 4)
(
  output [R2_width -1: 0]    count,
  input [dp_width -1: 0] data,
  input              Start, clock, reset_b
);
  wire     Load_regs, Add_shift, Zero;
  Controller_2_Gates_1Hot M0 (Ready, Load_regs, Add_shift, Start, Zero, clock, reset_b);
  Datapath_2_STR M1 (count,  Zero, data, Load_regs, Add_shift, clock, reset_b);
endmodule

// Test plan for integrated system
// Test for data values of 8'haa, 8'h00, 8'hff.

// Test bench for integrated system

module t_Count_Ones_2_Gates_1Hot_STR ();
  parameter              dp_width = 8, R2_width = 4;
  wire [R2_width -1: 0]  count;
  reg [dp_width -1: 0]   data;
  reg                    Start, clock, reset_b;
  wire [1: 0]            state = {M0.M0.T1, M0.M0.T0};

 Count_Ones_2_Gates_1Hot_STR  M0 (count, data, Start, clock, reset_b);

  initial #700 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin reset_b = 0; #2 reset_b = 1; end
  initial fork
    data = 8'haa;       // Expect count = 4
    Start = 0;
    #20 Start = 1;
    #30 Start = 0;
    #40 data = 8'b00;     // Expect count = 0
    #120 Start = 1;
    #130 Start = 0;
    #150 data = 8'hff;     // Expect count = 8
    #200 Start = 1;
    #210 Start = 0;
  join
endmodule
```



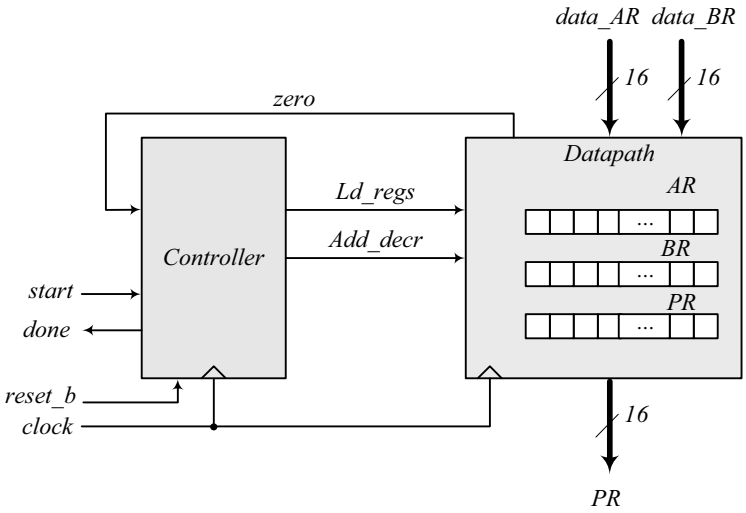
8.38

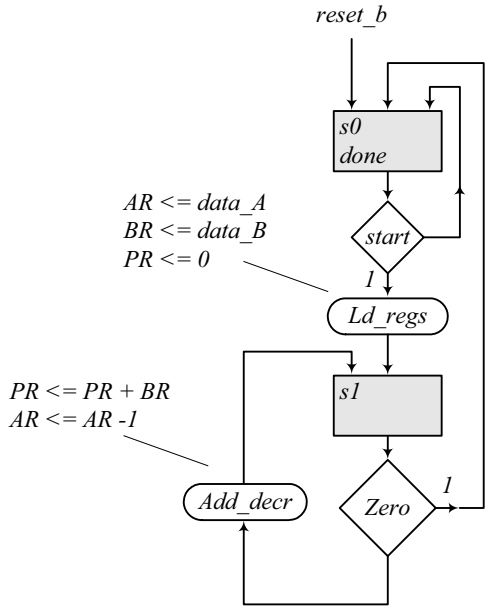
```
module Prob_8_38 (
 output reg [7: 0]    Sum,
 output reg           Car_Bor,
 input [7: 0]         Data_A, Data_B);
 reg [7: 0]           Reg_A, Reg_B;

 always @ (Data_A, Data_B)
    case ({Data_A[7], Data_B[7]})
      2'b00, 2'b11:        begin                              // ++, --
                             {Car_Bor, Sum[6: 0]} = Data_A[6: 0] + Data_B[6: 0];
                            Sum[7] = Data_A[7];
                           end

      default:             if (Data_A[6: 0] >= Data_B[6: 0]) begin          // +-, -+
                             {Car_Bor, Sum[6: 0]} = Data_A[6: 0] - Data_B[6: 0];
                             Sum[7] = Data_A[7];
                           end
                           else begin
                             {Car_Bor, Sum[6: 0]} = Data_B[6: 0] - Data_A[6: 0];
                             Sum[7] = Data_B[7];
                           end
    endcase
endmodule

module t_Prob_8_38 ();
  wire [7: 0]    Sum;
  wire           Car_Bor;
  reg [7: 0]     Data_A, Data_B;
  wire [6: 0]    Mag_A, Mag_B;
  assign         Mag_A = M0.Data_A[6: 0];          // Hierarchical dereferencing
  assign         Mag_B = M0.Data_B[6: 0];
  wire           Sign_A = M0.Data_A[7];
  wire           Sign_B = M0.Data_B[7];
  wire           Sign = Sum[7];
  wire [7: 0]    Mag = Sum[6: 0];

  Prob_8_38 M0 (Sum, Car_Bor,  Data_A, Data_B);

  initial #650 $finish;
```

```
initial fork

  // Addition                                  //  A        B
   #0 begin Data_A = {1'b0, 7'd25}; Data_B = {1'b0, 7'd10}; end          //+25, +10
  #40 begin Data_A = {1'b1, 7'd25}; Data_B = {1'b1, 7'd10}; end          // -25, -10
  #80 begin Data_A = {1'b1, 7'd25}; Data_B = {1'b0, 7'd10}; end          // -25, +10
 #120 begin Data_A = {1'b0, 7'd25}; Data_B = {1'b1, 7'd10}; end          // 25, -10
                                    //  B        A
 #160 begin Data_B = {1'b0, 7'd25}; Data_A = {1'b0, 7'd10}; end          //+25, +10
 #200 begin Data_B = {1'b1, 7'd25}; Data_A = {1'b1, 7'd10}; end          // -25, -10

 #240 begin Data_B = {1'b1, 7'd25}; Data_A = {1'b0, 7'd10}; end          // -25, +10
 #280 begin Data_B = {1'b0, 7'd25}; Data_A = {1'b1, 7'd10}; end          // +25, -10
  // Addition of matching numbers

 #320 begin Data_A = {1'b1,7'd0}; Data_B = {1'b1,7'd0}; end              // -0, -0
 #360 begin Data_A = {1'b0,7'd0}; Data_B = {1'b0,7'd0}; end              // +0, +0
 #400 begin Data_A = {1'b0,7'd0}; Data_B = {1'b1,7'd0}; end              // +0, -0
 #440 begin Data_A = {1'b1,7'd0}; Data_B = {1'b0,7'd0}; end              // -0, +0

 #480 begin Data_B = {1'b0, 7'd25}; Data_A = {1'b0, 7'd25}; end          // matching +
 #520 begin Data_B = {1'b1, 7'd25}; Data_A = {1'b1, 7'd25}; end          // matching –

  // Test of carry (negative numbers)
  #560 begin Data_A = 8'hf0; Data_B = 8'hf0; end                         // carry - -
  // Test of carry (positive numbers)
  #600 begin Data_A = 8'h70; Data_B = 8'h70; end                         // carry ++
 join
endmodule
```



8.39 Block diagram and ASMD chart:

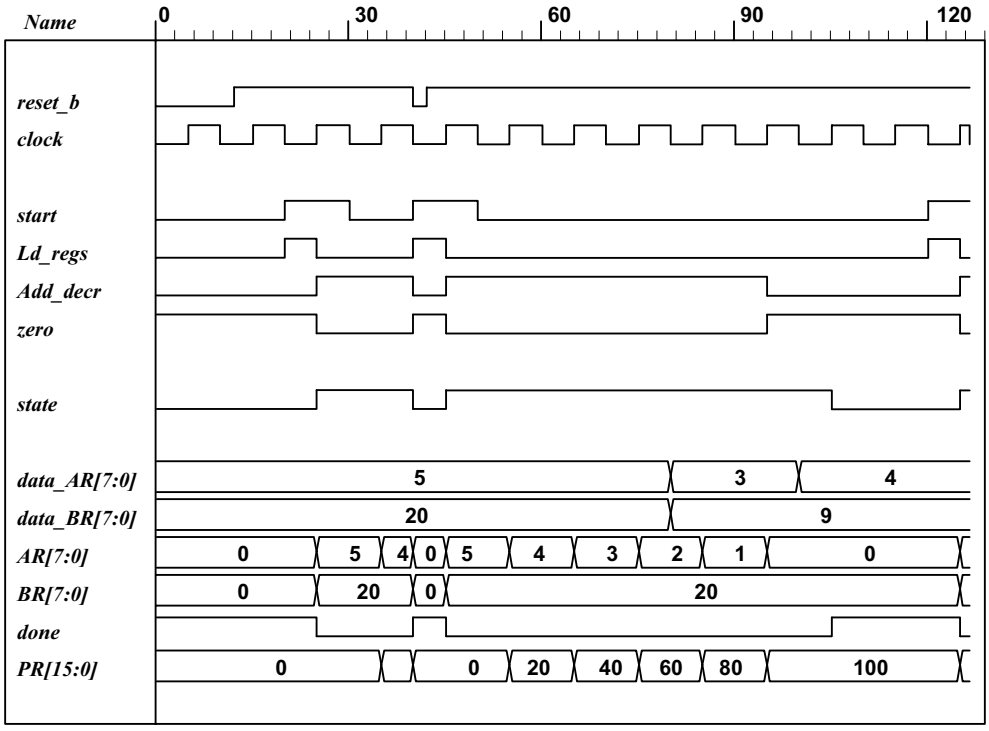




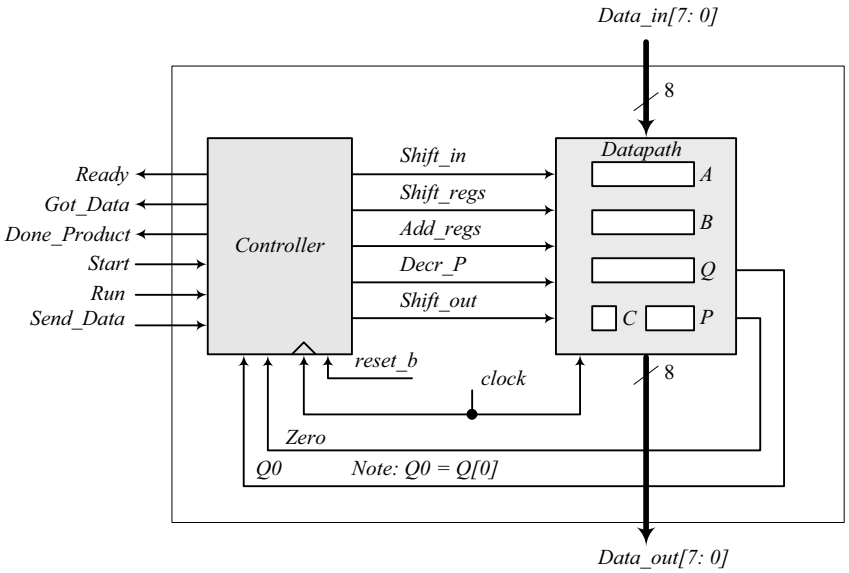

```
module Prob_8_39 (
  output [15: 0] PR, output done,
  input        [7: 0]   data_AR, data_BR, input  start, clock, reset_b
);

  Controller_P8_39 M0 (done, Ld_regs, Add_decr, start, zero, clock, reset_b);

 Datapath_P8_39 M1 (PR, zero, data_AR, data_BR, Ld_regs, Add_decr, clock, reset_b);
endmodule

module Controller_P8_16 (output done, output reg Ld_regs, Add_decr, input start, zero, clock, reset_b);
  parameter      s0 = 1'b0, s1 = 1'b1;
  reg          state, next_state;
  assign done = (state == s0);
```

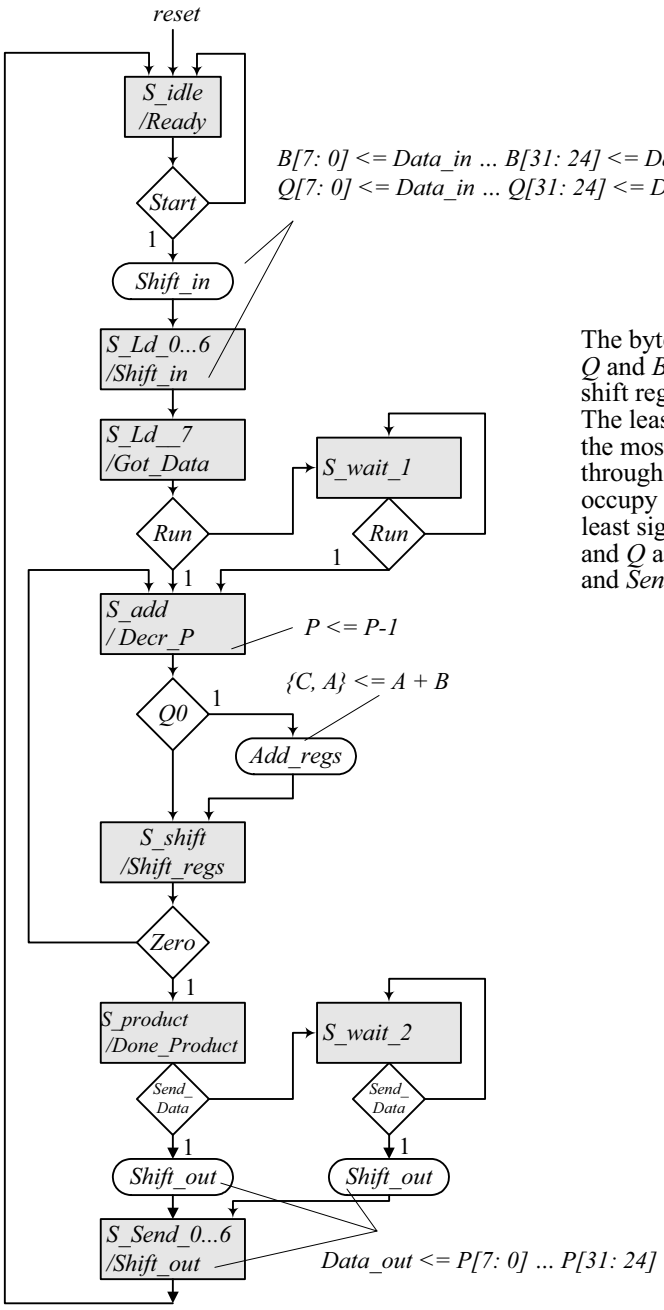
```
  always @ (posedge clock, negedge reset_b)
    if (!reset_b) state <= s0; else state <= next_state;

  always @ (state, start, zero) begin
    Ld_regs = 0;
    Add_decr = 0;
    case (state)
      s0:       if (start) begin Ld_regs = 1; next_state = s1; end
      s1:       if (zero) next_state = s0; else  begin next_state = s1; Add_decr = 1; end
      default: next_state = s0;
    endcase
  end
endmodule

module  Datapath_P8_16 (
  output reg     [15: 0] PR, output zero,
  input       [7: 0]   data_AR, data_BR, input Ld_regs, Add_decr, clock, reset_b
);

  reg          [7: 0]   AR, BR;
  assign       zero = ~( | AR);

  always @ (posedge clock, negedge reset_b)
    if (!reset_b) begin AR <= 8'b0; BR <= 8'b0; PR <= 16'b0; end
    else begin
      if (Ld_regs) begin AR <= data_AR; BR <= data_BR; PR <= 0; end
      else if (Add_decr) begin PR <= PR + BR; AR <= AR -1; end
    end
 endmodule

// Test plan – Verify;
// Power-up reset
// Data is loaded correctly
// Control signals assert correctly
// Status signals assert correctly
// start is ignored while multiplying
// Multiplication is correct
// Recovery from reset on-the-fly

module t_Prob_P8_16;
  wire          done;
  wire     [15: 0] PR;
  reg       [7: 0]   data_AR, data_BR;
  reg           start, clock, reset_b;

  Prob_8_16 M0 (PR, done, data_AR, data_BR, start, clock, reset_b);

  initial #500 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
     reset_b = 0;
      #12 reset_b = 1;
    #40 reset_b = 0;
    #42 reset_b = 1;
    #90 reset_b = 1;
    #92 reset_b = 1;
  join
```

```
 initial fork
   #20 start = 1;
   #30 start = 0;
   #40 start = 1;
   #50 start = 0;
   #120 start = 1;
   #120 start = 0;
 join

 initial fork
   data_AR = 8'd5;       // AR > 0
   data_BR = 8'd20;

   #80 data_AR = 8'd3;
   #80 data_BR = 8'd9;

   #100 data_AR = 8'd4;
   #100 data_BR = 8'd9;
 join
endmodule
```



8.40





The bytes of data will be read sequentially. Registers *Q* and *B* are organized to act as byte-wide parallel shift registers, taking 8 clock cycles to fill the pipe. The least significant byte of the multiplicand enters the most significant byte of *Q* and then moves through the bytes of Q to enter *B*, then proceed to occupy successive bytes of *B* until it occupies the least significant byte of *B*, and so forth until both *B* and *Q* are filled. Wait states are used to wait for *Run* and *Send_Data*.

```
module Prob_8_40 (
  output   [7: 0]      Data_out,
  output               Ready, Got_Data, Done_Product,
  input    [7: 0]      Data_in,
  input                Start, Run, Send_Data, clock, reset_b
);

Controller M0 (
   Ready, Shift_in, Got_Data, Done_Product, Decr_P, Add_regs, Shift_regs, Shift_out,
   Start, Run, Send_Data, Zero, Q0, clock, reset_b
);
Datapath M1(Data_out, Q0, Zero, Data_in,
  Start, Shift_in, Decr_P, Add_regs, Shift_regs, Shift_out, clock
);
endmodule

module Controller (
   output reg Ready, Shift_in, Got_Data, Done_Product, Decr_P, Add_regs,
     Shift_regs, Shift_out,
   input Start, Run, Send_Data, Zero, Q0, clock, reset_b
);

  parameter   S_idle =    5'd20,
              S_Ld_0 =  5'd0,
              S_Ld_1 =  5'd1,
              S_Ld_2 =  5'd2,
              S_Ld_3 =  5'd3,
              S_Ld_4 =  5'd4,
              S_Ld_5 =  5'd5,
              S_Ld_6 =  5'd6,
              S_Ld_7 =  5'd7,
              S_wait_1 = 5'd8,  // Wait state
              S_add =    5'd9,
              S_Shift =  5'd10,
              S_product =    5'd11,
              S_wait_2 =     5'd12,  // Wait state
              S_Send_0 =   5'd13,
              S_Send_1 =   5'd14,
              S_Send_2 =   5'd15,
              S_Send_3 =   5'd16,
              S_Send_4 =   5'd17,
              S_Send_5 =   5'd18,
              S_Send_6 =   5'd19;

  reg    [4: 0]          state, next_state;

  always @ (posedge clock, negedge reset_b)
    if (~reset_b) state <= S_idle; else state <= next_state;

  always @ (state, Start, Run, Q0, Zero, Send_Data) begin
    next_state = S_idle;                  // Prevent accidental synthesis of latches
    Ready = 0;
    Shift_in = 0;
    Shift_regs = 0;
    Add_regs = 0;
    Decr_P = 0;
    Shift_out = 0;
    Got_Data = 0;
    Done_Product = 0;
```

```
    case (state)     // Assign by exception to default values
     S_idle:         begin
                       Ready = 1;
                       if (Start) begin next_state = S_Ld_0; Shift_in = 1; end
                     end
     S_Ld_0:         begin next_state = S_Ld_1; Shift_in = 1; end
     S_Ld_1:         begin next_state = S_Ld_2; Shift_in = 1; end
     S_Ld_2:         begin next_state = S_Ld_3; Shift_in = 1; end
     S_Ld_3:         begin next_state = S_Ld_4; Shift_in = 1; end
     S_Ld_4:         begin next_state = S_Ld_5; Shift_in = 1; end
     S_Ld_5:         begin next_state = S_Ld_6; Shift_in = 1; end
     S_Ld_6:         begin next_state = S_Ld_7; Shift_in = 1; end
     S_Ld_7:         begin Got_Data = 1;
                       if (Run) next_state = S_add;
                       else next_state = S_wait_1;
                     end
     S_wait_1:       if (Run) next_state = S_add; else next_state = S_wait_1;
     S_add:          begin next_state = S_Shift; Decr_P = 1; if (Q0) Add_regs = 1; end
     S_Shift:        begin Shift_regs = 1; if (Zero) next_state = S_product;
                     else next_state = S_add; end
     S_product:      begin
                       Done_Product = 1;
                       if (Send_Data) begin  next_state = S_Send_0; Shift_out = 1; end
                       else next_state = S_wait_2; end
     S_wait_2:       if (Send_Data) begin next_state =S_Send_0; Shift_out = 1; end
                     else next_state = S_wait_2;
     S_Send_0:       begin next_state = S_Send_1; Shift_out = 1; end
     S_Send_1:       begin next_state = S_Send_2; Shift_out = 1; end
     S_Send_2:       begin next_state = S_Send_3; Shift_out = 1; end
     S_Send_3:       begin next_state = S_Send_4; Shift_out = 1; end
     S_Send_4:       begin next_state = S_Send_5; Shift_out = 1; end
     S_Send_5:       begin next_state = S_Send_6; Shift_out = 1; end
     S_Send_6:       begin next_state = S_idle; Shift_out = 1; end
     default:        next_state = S_idle;
    endcase
  end
endmodule

module Datapath #(parameter dp_width = 32, P_width = 6) (
  output   [7: 0]       Data_out,
  output                Q0, Zero,
  input    [7: 0]       Data_in,
  input                 Start, Shift_in, Decr_P, Add_regs, Shift_regs, Shift_out, clock
);
  reg   [dp_width - 1: 0]       A, B, Q;                // Sized for datapath
  reg                           C;
  reg   [P_width - 1: 0]        P;
  assign                        Q0 = Q[0];
  assign                        Zero = (P == 0);            // counter is zero
  assign                        Data_out = {C, A, Q};

  always @ (posedge clock) begin
    if (Shift_in) begin
      P <= dp_width;
      A <= 0;
      C <= 0;
      B[7: 0]     <= B[15: 8];        // Treat B and Q registers as a pipeline to load data bytes
      B[15: 8]    <= B[ 23: 16];
      B[23: 16]   <= B[31: 24];
      B[31: 24]   <= Q[7: 0];
      Q[7: 0]     <= Q[15: 8];
```

```
      Q[15: 8]    <= Q[ 23: 16];
      Q[23: 16]  <= Q[31: 24];
      Q[31: 24]  <= Data_in;
    end
    if (Add_regs) {C, A} <= A + B;
    if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
    if (Decr_P) P <= P -1;
    if (Shift_out) begin {C, A, Q} <= {C, A, Q} >> 8; end
  end
endmodule

module t_Prob_8_40;
  parameter          dp_width = 32;          // Width of datapath
  wire  [7: 0]       Data_out;
  wire               Ready, Got_Data, Done_Product;
  reg                Start, Run, Send_Data, clock, reset_b;

  integer            Exp_Value;
  reg                Error;
  wire  [7: 0]       Data_in;
  reg [dp_width -1: 0]      Multiplicand, Multiplier;
  reg [2*dp_width -1: 0]    Data_register;      // For test patterns
  assign                    Data_in = Data_register [7:0];
  wire [2*dp_width -1: 0]   product;
  assign                    product = {M0.M1.C, M0.M1.A, M0.M1.Q};
  Prob_8_40  M0 (
    Data_out,  Ready, Got_Data, Done_Product, Data_in, Start, Run, Send_Data, clock, reset_b
);

  initial #2000 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial fork
    reset_b = 1;
    #2 reset_b = 0;
    #3 reset_b = 1;
  join
  initial fork
    Start =0;
    Run = 0;
    Send_Data = 0;
    #10 Start = 1;
    #20 Start = 0;


    #50 Run= 1;          // Ignored by controller
    #60 Run = 0;
    #120 Run = 1;
    #130 Run = 0;

    #830 Send_Data = 1;
    #840 Send_Data = 0;
  join
// Test patterns for multiplication

  initial begin
    Multiplicand = 32'h0f_00_00_aa;
    Multiplier = 32'h0a_00_00_ff;
    Data_register = {Multiplier, Multiplicand};
  end

 initial begin           // Synchronize input data bytes
```

```
  @ (posedge Start)
     repeat (15) begin
       @ (negedge clock)
          Data_register <= Data_register >> 8;
     end
  end
endmodule
```

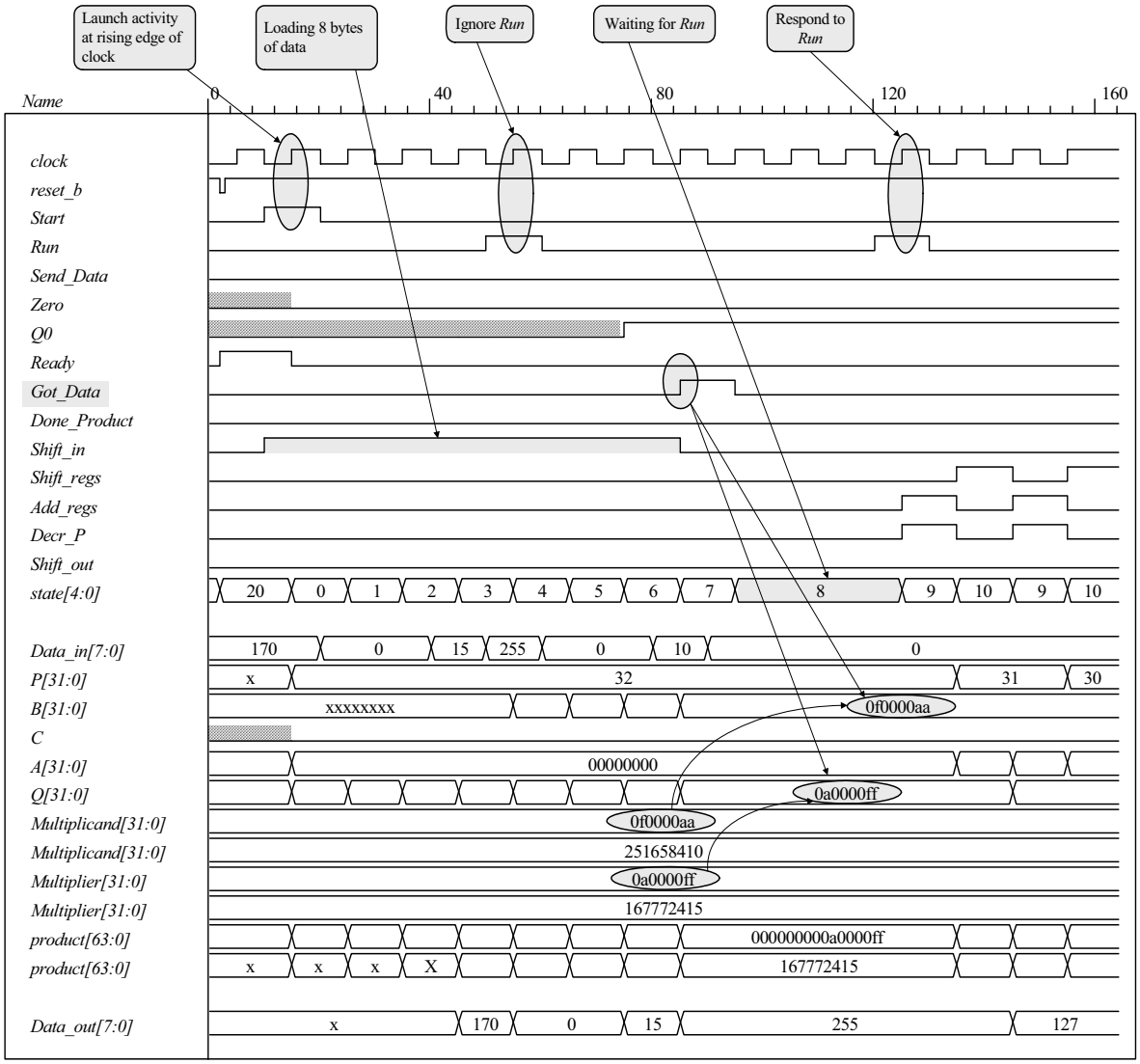
Simulation results: Loading multiplicand ($0f0000aa_H$) and multiplier ($0a0000ff_H$), 4 bytes each, in sequence, beginning with the least significant byte of the multiplicand.

Note: *Product* is not valid until *Done_Product* asserts. The value of *Product* shown here (255_{10}) reflects the contents of *{C, A, Q}* after the multiplier has been loaded, prior to multiplication.

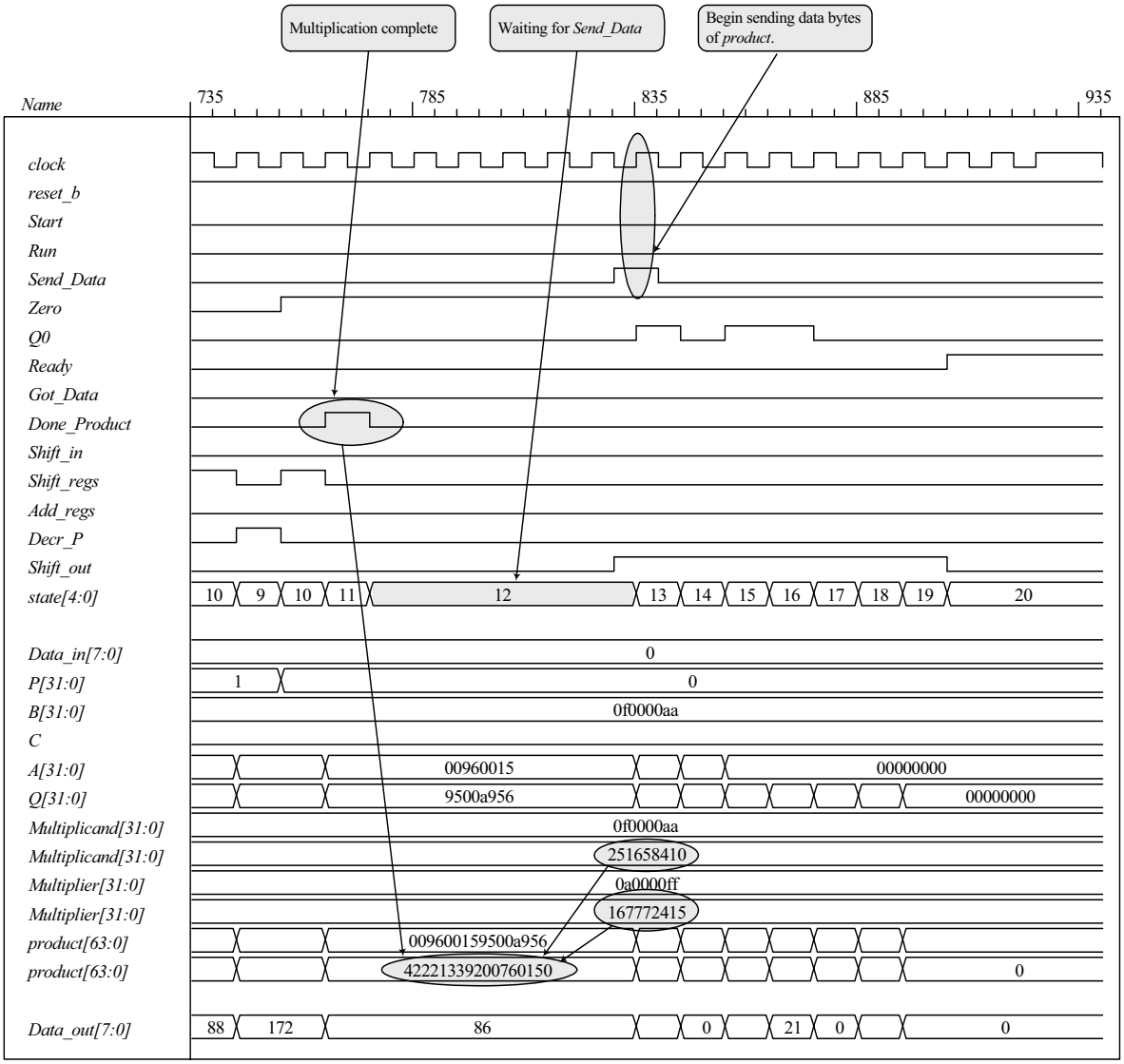
Note: The machine ignores a premature assertion of *Run*.

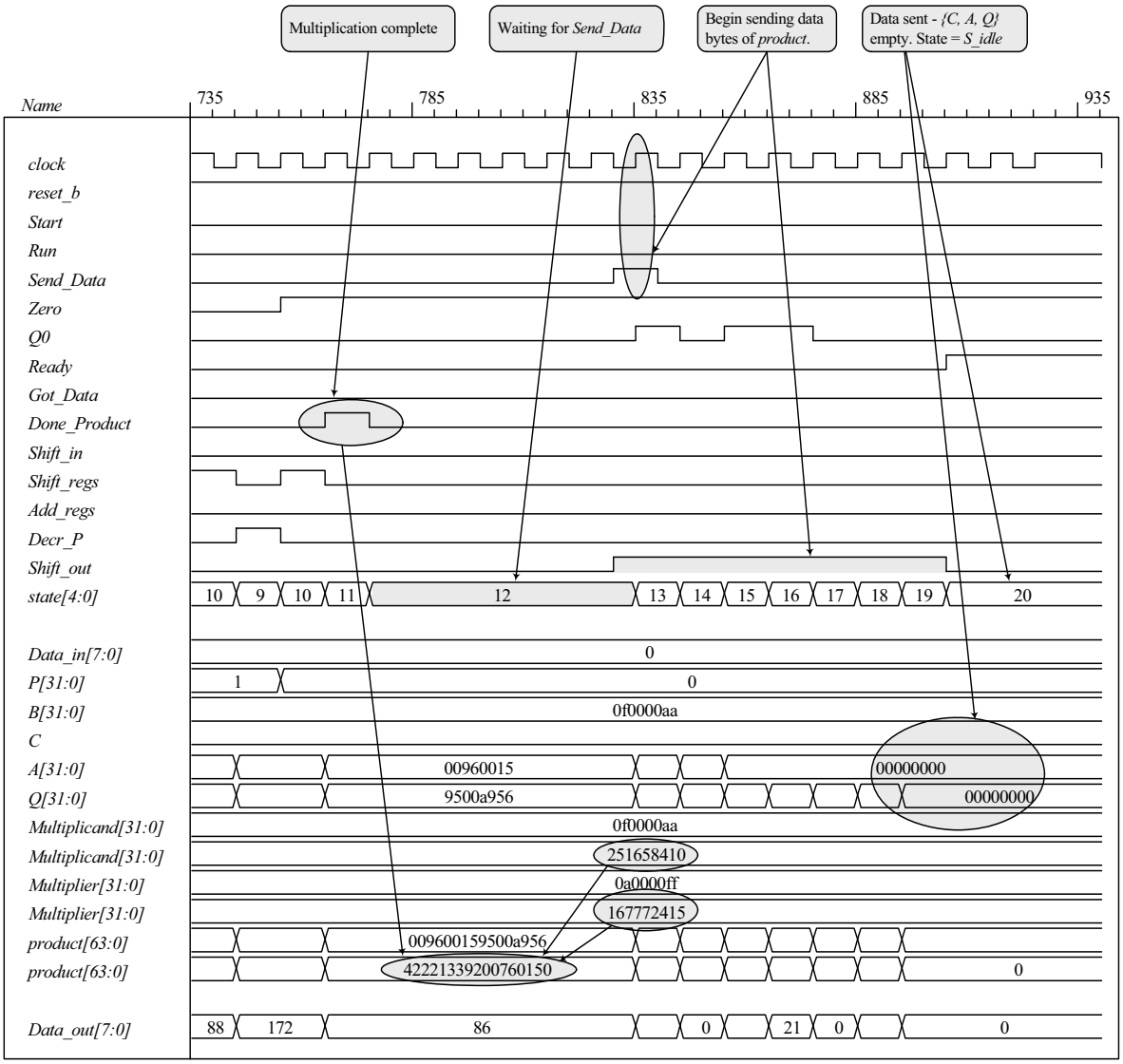
Note: *Got_Data* asserts at the 8^{th} clock after Start asserts, i.e., 8 clocks to load the data.

Note: *Product*, *Multiplier*, and *Multiplicand* are formed in the test bench.

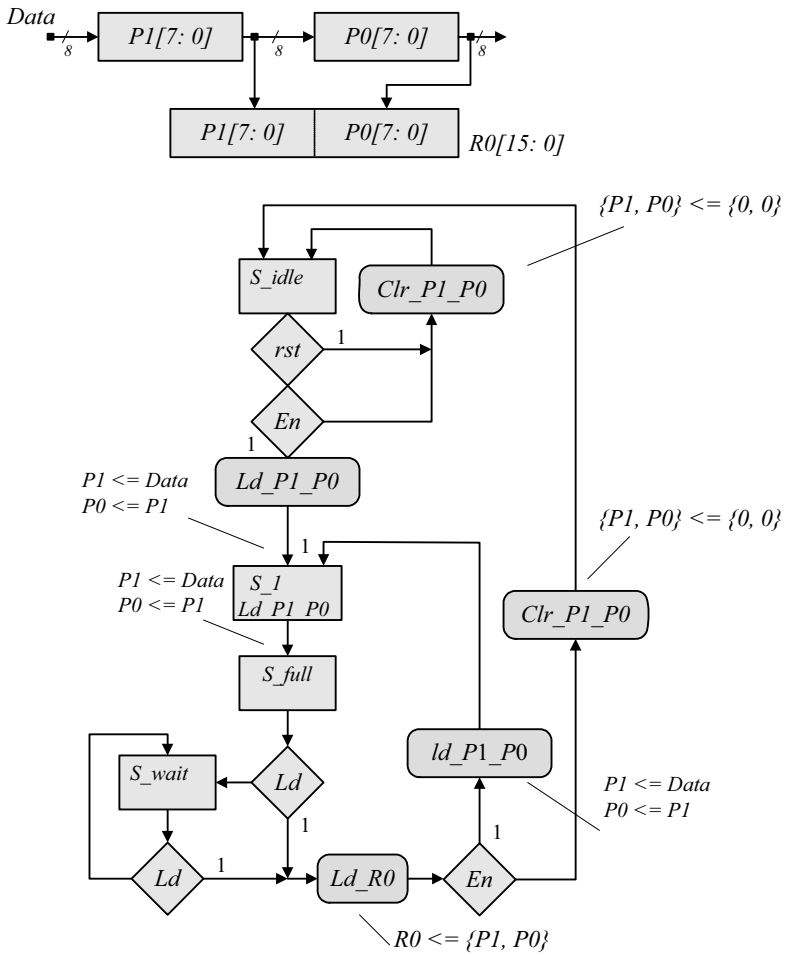


Note: Product (64 bits) is formed correctly





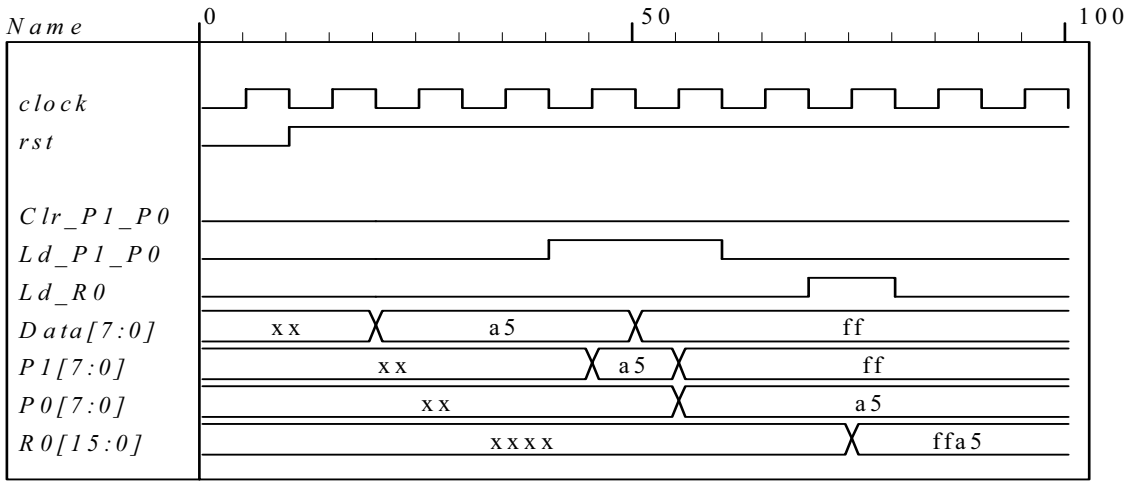
8.41 (a)



(b) HDL model, test bench and simulation results for datapath unit.

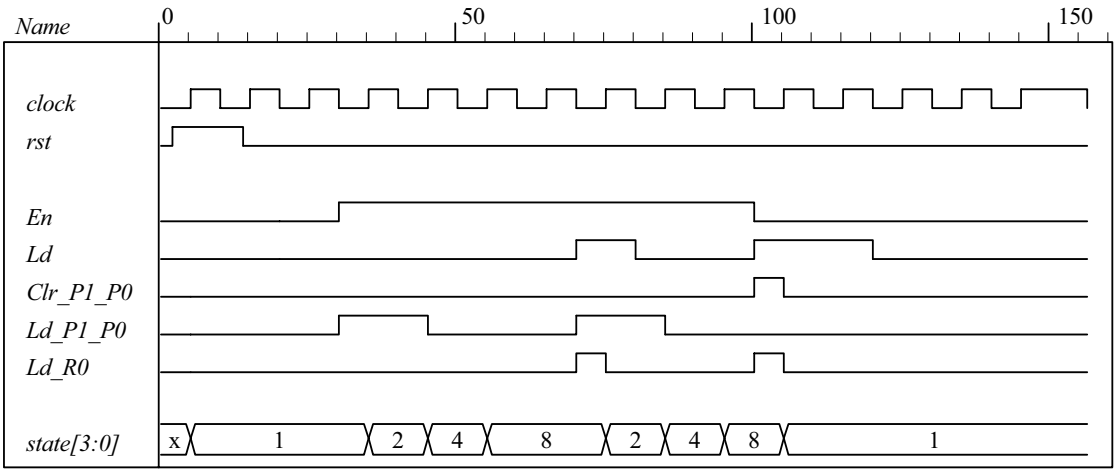
```
module Datapath_unit
(
  output reg [15: 0]    R0, input [7: 0] Data, input Clr_P1_P0, Ld_P1_P0, Ld_R0, clock, rst);
   reg [7: 0]      P1, P0;

  always @ (posedge clock) begin
    if (Clr_P1_P0) begin P1 <= 0; P0 <= 0; end
    if (Ld_P1_P0) begin P1 <= Data; P0 <= P1; end
    if (Ld_R0) R0 <= {P1, P0};
  end
endmodule
```

```
// Test bench for datapath
module t_Datapath_unit ();
  wire [15: 0]  R0;
  reg [7: 0]    Data;
  reg           Clr_P1_P0,  Ld_P1_P0,  Ld_R0,  clock,  rst;

   Datapath_unit M0 (R0, Data, Clr_P1_P0, Ld_P1_P0, Ld_R0, clock, rst);

  initial #100 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin rst = 0; #2 rst = 1; end
  initial fork
    #20 Clr_P1_P0 = 0;
    #20 Ld_P1_P0 = 0;
    #20 Ld_R0 = 0;
    #20 Data = 8'ha5;
    #40 Ld_P1_P0 = 1;
    #50 Data = 8'hff;
    #60 Ld_P1_P0 = 0;
    #70 Ld_R0 = 1;
    #80 Ld_R0 = 0;
  join
endmodule
```



(c) HDL model, test bench, and simulation results for the control unit.

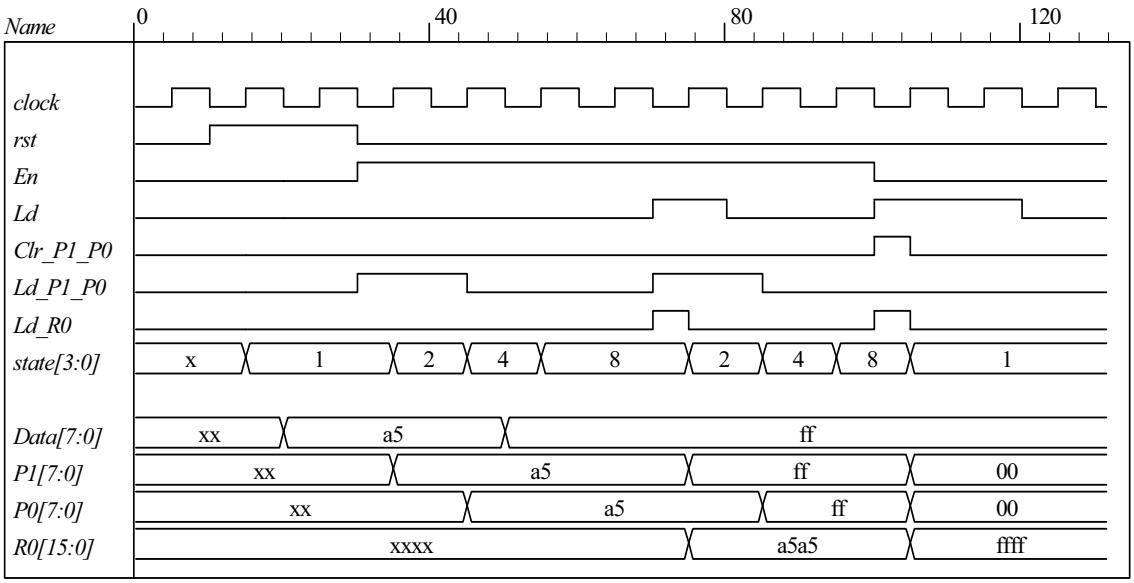
```
module Control_unit (output reg Clr_P1_P0, Ld_P1_P0, Ld_R0, input En, Ld, clock, rst);
  parameter S_idle = 4'b0001, S_1 = 4'b0010, S_full = 4'b0100, S_wait = 4'b1000;
  reg [3: 0] state, next_state;

  always @ (posedge clock)
    if (rst) state <= S_idle;
    else state <= next_state;

  always @ (state, Ld, En) begin
    Clr_P1_P0 = 0;              // Assign by exception
    Ld_P1_P0 = 0;
    Ld_R0 = 0;
    next_state = S_idle;
    case (state)
      S_idle:    if (En) begin Ld_P1_P0 = 1; next_state = S_1; end
                 else next_state = S_idle;

      S_1:       begin Ld_P1_P0 = 1; next_state = S_full; end

      S_full:    if (!Ld) next_state = S_wait;
                 else begin
                   Ld_R0 = 1;
                   if (En) begin Ld_P1_P0 = 1; next_state = S_1;  end
                   else begin Clr_P1_P0 = 1; next_state = S_idle; end
                 end

      S_wait:    if (!Ld) next_state = S_wait;
                 else begin
                   Ld_R0 = 1;
                   if (En) begin Ld_P1_P0 = 1; next_state = S_1; end
                   else begin Clr_P1_P0 = 1; next_state = S_idle;  end
                 end
      default:   next_state = S_idle;
    endcase
  end
endmodule

// Test bench for control unit
module t_Control_unit ();
  wire  Clr_P1_P0, Ld_P1_P0, Ld_R0;
  reg    En, Ld, clock, rst;

  Control_unit M0 (Clr_P1_P0, Ld_P1_P0, Ld_R0, En, Ld, clock, rst);

  initial #200 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin rst = 0; #2 rst = 1; #12 rst = 0; end
  initial fork
    #20 Ld = 0;
    #20 En = 0;
    #30 En = 1; // Drive to S_wait
    #70 Ld = 1; // Return to S_1 to S_full tp S_wait
    #80 Ld = 0;
     #100 Ld = 1;    // Drive to S_idle
    #100 En = 0;
    #110 En = 0;
    #120 Ld = 0;
  join
endmodule
```



(c) Integrated system Note that the test bench for the integrated system uses the input stimuli from the test bench for the control unit and displays the waveforms produced by the test bench for the datapath unit.:

```
module Prob_8_41 (output [15: 0] R0, input [7: 0] Data, input En, Ld, clock, rst);
  wire Clr_P1_P0, Ld_P1_P0, Ld_R0;

  Control_unit M0 (Clr_P1_P0, Ld_P1_P0, Ld_R0, En, Ld, clock, rst);
  Datapath_unit M1 (R0, Data, Clr_P1_P0, Ld_P1_P0, Ld_R0, clock);

endmodule

module Control_unit (output reg Clr_P1_P0, Ld_P1_P0, Ld_R0, input En, Ld, clock, rst);
  parameter S_idle = 4'b0001, S_1 = 4'b0010, S_full = 4'b0100, S_wait = 4'b1000;
  reg [3: 0] state, next_state;

  always @ (posedge clock)
    if (rst) state <= S_idle;
    else state <= next_state;

  always @ (state, Ld, En) begin
    Clr_P1_P0 = 0;              // Assign by exception
    Ld_P1_P0 = 0;
    Ld_R0 = 0;
    next_state = S_idle;
    case (state)
      S_idle:    if (En) begin Ld_P1_P0 = 1; next_state = S_1; end
                 else next_state = S_idle;

      S_1:       begin Ld_P1_P0 = 1; next_state = S_full; end

      S_full:    if (!Ld) next_state = S_wait;
                 else begin
                   Ld_R0 = 1;
                   if (En) begin Ld_P1_P0 = 1; next_state = S_1; end
                   else begin Clr_P1_P0 = 1; next_state = S_idle; end
                 end

      S_wait:    if (!Ld) next_state = S_wait;
                 else begin
```

```
                Ld_R0 = 1;
                if (En) begin Ld_P1_P0 = 1; next_state = S_1; end
                else begin Clr_P1_P0 = 1; next_state = S_idle; end
              end
    default:  next_state = S_idle;
   endcase
  end
endmodule

module Datapath_unit
(
  output reg [15: 0]  R0,
  input [7: 0]        Data,
  input               Clr_P1_P0,
                      Ld_P1_P0,
                      Ld_R0,
                      clock);
   reg [7: 0]    P1, P0;

  always @ (posedge clock) begin
    if (Clr_P1_P0) begin P1 <= 0; P0 <= 0; end
    if (Ld_P1_P0) begin P1 <= Data; P0 <= P1; end
    if (Ld_R0) R0 <= {P1, P0};
  end
endmodule


// Test bench for integrated system
module t_Prob_8_41 ();
  wire [15: 0]  R0;
  reg [7: 0]    Data;
  reg           En, Ld, clock, rst;

  Prob_8_41 M0 (R0, Data, En, Ld, clock, rst);

  initial #200 $finish;
  initial begin clock = 0; forever #5 clock = ~clock; end
  initial begin rst = 0; #10 rst = 1; #20 rst = 0; end
  initial fork

    #20 Data = 8'ha5;
    #50 Data = 8'hff;

    #20 Ld = 0;
    #20 En = 0;
    #30 En = 1; // Drive to S_wait
    #70 Ld = 1; // Return to S_1 to S_full tp S_wait
    #80 Ld = 0;
     #100 Ld = 1;    // Drive to S_idle
    #100 En = 0;
    #110 En = 0;
    #120 Ld = 0;
  join
endmodule
```



CHAPTER 9

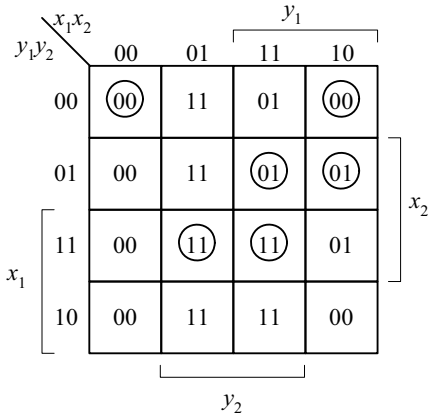
9.1 **(a)** Asynchronous circutis do not use clock pulses and change state in response to input changes. Synchronous circuits use clock pulses and a change of state occurs in reponse to the clock transition.

(b) The input signals change one at a time when the circuit is stable.

(c) The circuit is in a stable state when the excitation variables (Y) are equal to the secondary variables (y) (see F. 9.1). Unstable otherwise.

(d) The total state is the combination of binary values of the internal state and the inputs.

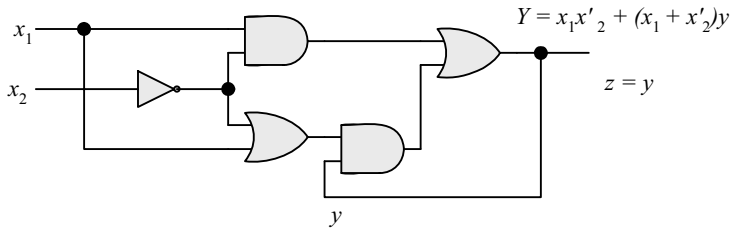
9.2 $Y_1 = x_1'x_2 + y_1x_2$ $\quad Y_2 = x_1y_2 + x_2$



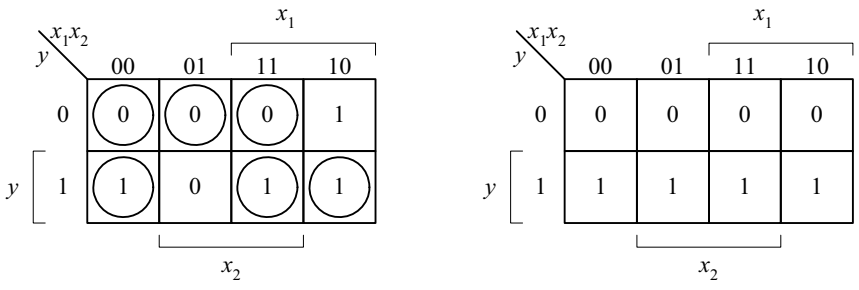
x_1x_2: 00, 10, 11, 01, 11, 10, 00

y_1y_2: 00, 00, 01, 11, 11, 01, 00

9.3 **(a)**



(b)



(c)