

Chapter 1

Solutions: The Digital Abstraction

1.1 (0.3,2,2), the worst case noise margin is larger

- 1.2**
1. 0.5V
 2. -0.5V
 3. 0.7V
 4. 0.4V

1.3 $V_N \leq 0.4V$

1.4 GNDA can be 0.2V lower than GNDB or or 0.3V higher

1.8

$$\begin{aligned} \max \left| \frac{dV_{out}}{dV_{in}} \right| &\geq \frac{V_{OH} - V_{OL}}{V_{IH} - V_{IL}} \\ \max \left| \frac{dV_{out}}{dV_{in}} \right| &\geq \frac{2.1 - 0.2}{1.7 - 0.7} = 1.9 \end{aligned}$$

1.9

000	68
001	70
011	72
010	74
110	76
111	78
101	80
100	82

1.10 We assume a truncation scheme, encoding $2\lfloor \frac{T}{2} \rfloor$. For binary weightings:

$$TempA[i] = \left\lfloor \frac{T - 68}{2^{i+1}} \right\rfloor \mod 2 \quad (1.1)$$

For thermometer encodings:

$$TempB[i] = \begin{cases} 0 : & T - 70 - 2i < 0 \\ 1 : & T - 70 - 2i \geq 0 \end{cases} \quad (1.2)$$

1.11 For both (a) and (b) we can use a 6-bit scheme where the top 2 bits represent the suit. The bottom 4 bits represent the rank in suit. In both representations, the lower four bits are used to do any comparison of rank. The upper two are used to do a comparison of suits.

1.15 The rules are to go west if the current address is greater than the destination and east if the current is less than the destination. If the two addresses are equal, the current processor is the destination.

- 1.16**
- | | | | |
|------|------|------|------|
| 0000 | 0001 | 0010 | 0011 |
| 0100 | 0101 | 0110 | 0111 |
| 1000 | 1001 | 1010 | 1011 |
| 1100 | 1101 | 1110 | 1111 |
- 1.
 2. Using the rules of the previous problem, the lower two bits determine east/west routing (east is higher) and the upper two bits represent north/south (south is higher).
 3. By splitting the north/south and east/west addresses, it becomes much simpler to determine direction. If nodes had been assigned meaningless IDs, a routing table (or more complex logic) is required to determine direction.

1.17 One example is:

0000	0
0001	1
0011	2
0111	3
0101	4
0100	5

Chapter 2

Solutions: The Practice of Digital System Design

- 2.1** Student solutions should include information about how the system connects to the TV (HDMI), resolution (1080p), the processor, DRAM, and video. The games could be burned into the system, on DVD, or downloadable (wired or wireless Internet?). The controllers could be wired or wireless, etc.
- 2.2** Version 2 of the console should probably be better than the first version, so upgrades to core components such as DRAM, graphics card, and processor are probably necessary. The controllers and video output can potentially remain the same. Another option is to take advantage of advances in fabrication technology to make a version of the console with the same performance. Presumably, this would make the device cheaper and draw less power.
- 2.4** In our example, we would buy the network interface and HDMI output since they are commodity parts that our team could only mess up. We would build the motherboard, for example, to tie together our components in a custom fashion.
- 2.9** To average the four inputs on every cycle we need 3 sets of 32 flip-flops (28.8kgrids) and 3 adders (90kgrids). We don't need any multipliers because we can shift the sum of the four numbers by 2 (in binary). The total is 118.8kgrids (or 488kgrids with a multiplier). To do the weighted average, we need 4 multipliers (1.2Mgrids) and storage for 128 bits of data (256 grids in ROM and 3072 in SRAM). The sum is approximately 1.3Mgrids regardless of the storage mechanism.
- 2.10** See answers below:
1. We need storage for 2.4×10^7 bits of SRAM. This is 5.76×10^8 grids, or 4.8mm^2 .

2. It would take $500^3 \times 5ns = 0.625s$ to complete the full operation.
3. The remaining area is $5.2mm^2$, enough area for 1890 functional units.
The matrix multiplication takes $330\mu s$.

2.17 In 2015 there would be over 15B transistors and in 2020, 115B.

Chapter 3

Solutions: Boolean Algebra

3.1 We prove absorption by enumerating all possible input combinations:

x	y	$x \wedge (x \vee y)$	x	$x \vee (x \wedge y)$	x
0	0	0	0	0	0
0	1	0	0	0	0
1	0	1	1	1	1
1	1	1	1	1	1

3.2

x	$x \wedge x$	$x \vee x$
0	0	0
1	1	1

3.6 Omitting several uninteresting cases, the truth table is:

w	x	y	z	$\overline{w \wedge x \wedge y \wedge z}$	$\overline{w \vee \bar{x} \vee \bar{y} \vee \bar{z}}$	$\overline{w \vee x \vee y \vee z}$	$\overline{w \wedge \bar{x} \wedge \bar{y} \wedge \bar{z}}$
0	0	0	0	1	1	1	1
0	0	0	1	1	1	0	0
...	1	1	0	0
1	1	1	0	1	1	0	0
1	1	1	1	0	0	0	0

3.9 We simplify this equation by using the commutative and associative properties followed by the combining property (twice):

$$\begin{aligned}
 & (x \wedge y \wedge z) \vee (\bar{x} \wedge y) \vee (x \wedge y \wedge \bar{z}) \\
 = & ((x \wedge y \wedge z) \vee (x \wedge y \wedge \bar{z})) \vee (\bar{x} \wedge y) \\
 = & (x \wedge y) \vee (\bar{x} \wedge y) \\
 = & y
 \end{aligned}$$

3.10

$$\begin{aligned}
& ((y \wedge \bar{z}) \vee (\bar{x} \wedge w)) \wedge ((x \wedge \bar{y}) \vee (z \wedge \bar{w})) \\
= & ((y \wedge \bar{z}) \wedge ((x \wedge \bar{y}) \vee (z \wedge \bar{w}))) \vee ((\bar{x} \wedge w) \wedge ((x \wedge \bar{y}) \vee (z \wedge \bar{w}))) && \text{Distributive} \\
= & ((y \wedge \bar{z} \wedge x \wedge \bar{y}) \vee (y \wedge \bar{z} \wedge z \wedge \bar{w})) \vee ((\bar{x} \wedge w \wedge x \wedge \bar{y}) \vee (\bar{x} \wedge w \wedge z \wedge \bar{w})) && \text{Distributive} \\
= & (0 \vee 0) \wedge (0 \vee 0) && \text{Complementation} \\
= & 0
\end{aligned}$$

3.13 The dual is found by simply replaced \vee with \wedge and \wedge with \vee .

$$\begin{aligned}
f(x, y) &= (x \wedge \bar{y}) \vee (\bar{x} \wedge y) \\
f^D(x, y) &= (x \vee \bar{y}) \wedge (\bar{x} \vee y)
\end{aligned}$$

This equation is true when x and y are the same, thus in normal form:

$$f^D(x, y) = (x \wedge y) \vee (\bar{x} \wedge \bar{y})$$

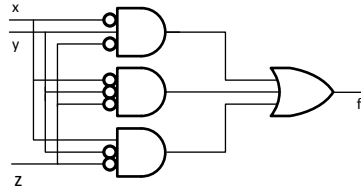
3.17 The normal form of $f(x, y, z) = x$ includes all minterms with $x = 1$:

$$f(x, y, z) = (x \wedge \bar{y} \wedge \bar{z}) \vee (x \wedge \bar{y} \wedge z) \vee (x \wedge y \wedge \bar{z}) \vee (x \wedge y \wedge z)$$

3.20 We first directly write the equation from the schematic and then simplify using DeMorgan's law:

$$\begin{aligned}
f(x, y, z) &= \overline{(\bar{x} \vee y) \wedge (x \vee \bar{y})} \\
&= (x \wedge \bar{y}) \vee (\bar{x} \wedge y)
\end{aligned}$$

3.23 We directly draw the schematic from the equations, using inversion bubbles instead of full inverters:



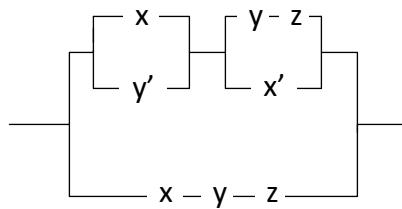
Chapter 4

Solutions: CMOS Logic Circuits

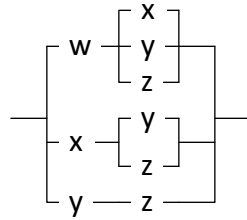
4.1 We enumerate all paths through the switch and write the logic in sum-of-products from:

$$f(e, d, c, b, a) = (a \wedge c \wedge e) \vee (a \wedge d) \vee (b \wedge c \wedge d) \vee (b \wedge e)$$

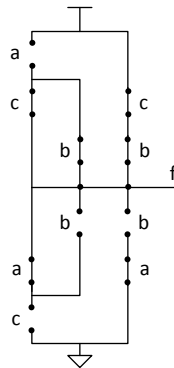
4.3 We find the solution by synthesizing the expression step-by-step, starting with $(y \wedge z) \vee \bar{x}$. The final solution is:



4.5 The solution is:



4.11 By substituting a closed switch for PFETs with a 0 input and NFETs with a 1 input (and open for PFET/1 and NFET/0), we can see that the circuit outputs a 1 for the given input:



4.15 We make our calculations using the following equations:

$$\begin{aligned}
 R_{s,N} &= \frac{L}{W} K_{RN} \\
 W_P &= \frac{1}{0.5} W_N K_p \\
 C_g &= WLK_c \\
 R_{s,N,130} &= 1k\Omega \\
 R_{s,N,28} &= 2.1k\Omega \\
 C_{g,N,130} &= 4fF \\
 C_{g,N,28} &= 0.56fF \\
 W_{N,130} &= 100L_{min} \\
 W_{N,28} &= 52L_{min} \\
 C_{g,P,130} &= 20fF \\
 C_{g,P,28} &= 1.5F
 \end{aligned}$$

4.17 1.

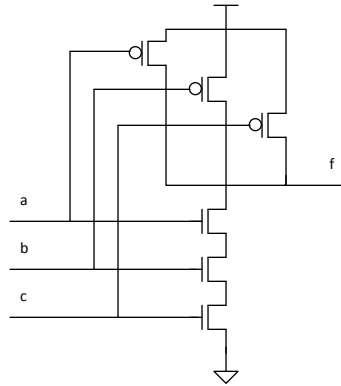
$$\begin{aligned}
 N &= \frac{1mm^2}{1 \times 10^6 L_{min}^2 nm^2} \frac{1 \times 10^{12} nm^2}{1mm^2} \\
 N_{130} &= 59 \\
 N_{28} &= 1276
 \end{aligned}$$

2.

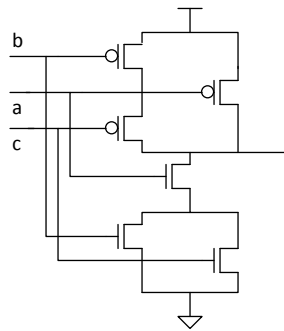
$$\begin{aligned}
 T &= 400(1 + K_P)\tau_n \\
 T_{130} &= 5.46ns \\
 T_{28} &= 1.10ns
 \end{aligned}$$

3.

$$\begin{aligned}
 \Theta &= \frac{N}{T} \\
 \Theta_{130} &= 10.9 \frac{Gops}{s} \\
 \Theta_{28} &= 1.16 \frac{Tops}{s}
 \end{aligned}$$



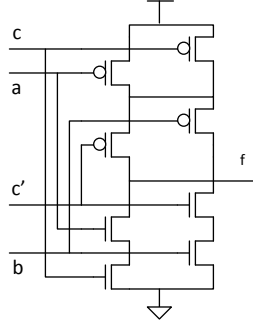
4.18



4.20

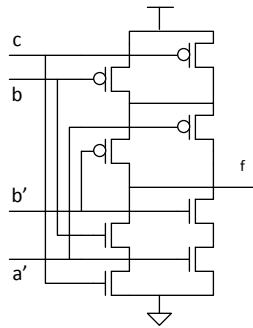
4.23 We write the equation, simplify it, and then draw the circuit:

$$\begin{aligned}
 f(c, b, a) &= \overline{(\bar{c} \wedge b \wedge \bar{a}) \vee (\bar{c} \wedge b \wedge a) \vee (c \wedge \bar{b} \wedge a) \vee (c \wedge b \wedge a)} \\
 &= \overline{(\bar{c} \wedge b) \vee (c \wedge a)}
 \end{aligned}$$



4.26 Because CMOS gates are inverting, we write the equation using the minterms that set $f = 0$ (000, 100, 110, 111). Next, we simplify and draw the gate:

$$\begin{aligned} f(c, b, a) &= \overline{(\bar{c} \wedge \bar{b} \wedge \bar{a}) \vee (c \wedge \bar{b} \wedge \bar{a}) \vee (c \wedge b \wedge \bar{a}) \vee (c \wedge b \wedge a)} \\ &= \overline{(\bar{b} \wedge \bar{a}) \vee (c \wedge b)} \end{aligned}$$



4.30 We can write the equation by analyzing the NFETs:

$$f(d, c, b, a) = \overline{d \vee c \vee (b \wedge a)}$$

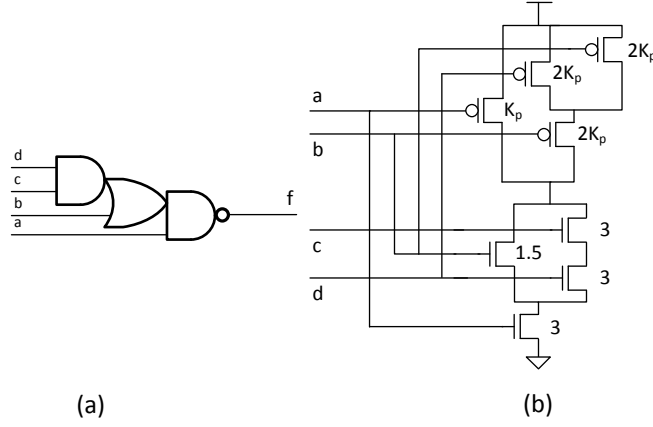
Chapter 5

Solutions: Delay and Power of CMOS Circuits

- 5.1** The pull-down (NFET) transistor has equal resistance to that of a minimum inverter, giving a maximum (and minimum) fall time of the product of fanout and inverter delay. That is $t_{fmax} = 4t_{inv}$. The rise time, however, has a resistance to one-third that of a minimum inverter. Thus, $t_{rmax} = \frac{4}{3}t_{inv}$.
- 5.4** In order to drive a load of $256C_{inv}$ using a chain of FO2 inverters, we need $\log_2(256) = 8$ inverters. The total delay is the product of number of stages (8) and the fanout of each stage (2), or $16t_{inv}$. The total energy of transitioning up then down again (or down then up) is found (starting with the output of stage 1) as:

$$\begin{aligned} E &= \sum_{i=0}^n C_{i+1} V^2 \\ &= E_{inv} \sum_{i=0}^n \frac{C_{i+1}}{C_{inv}} \\ &= (2 + 4 + 8 + 16 + 32 + 64 + 128 + 256) E_{inv} \\ &= 510 E_{inv} \end{aligned}$$

- 5.8** For part (a), we draw the schematic below as a AND-OR-ANDI gate. To size the transistor diagram, we want the maximum pull-up and pull-down resistance to be that of a minimum inverter. Recall that resistance is inversely proportional to width and we find the resistance of the pull-down path of c-d-a to be $\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$. Pull-down path of b-a is $\frac{2}{3} + \frac{1}{3}$. In this case having $W_a = 2$, $W_c = W_d = 4$, and $W_b = 2$ is also a valid answer. We apply a similar methodology to the pull-up network.



To calculate the logical effort of each input, we find the total input capacitance and divide by that of an inverter: $1 + K_p$:

$$\begin{aligned}
 LE_a &= \frac{3 + K_p}{1 + K_p} \\
 LE_b &= \frac{1.5 + 2K_p}{1 + K_p} \\
 LE_c &= \frac{3 + 2K_p}{1 + K_p} \\
 LE_d &= \frac{3 + 2K_p}{1 + K_p}
 \end{aligned}$$

5.11

Signal	Fanout	Logical Effort ($K_p = 1.3$)	Delay
i to $i + 1$	i to $i + 1$	$i + 1$	i to $i + 1$
b	2	1.87	3.74
cN	2	2.70	5.4
d	1	1	1
eN	5	1	5
TOTAL a to eN			$15.14t_{inv}$

5.14 To find the minimum delay, we first find the total effort. Next, we find

the stage effort to be the 4th root of the total effort:

$$\begin{aligned}
 TE &= \prod_i LE_i \times FO_i \\
 TE &= 1.87 * 2.7 * 1 * 1 * 20 \\
 TE &= 101 \\
 SE &= TE^{1/4} \\
 SE &= 3.17
 \end{aligned}$$

Using this stage effort ($FO \times LE$) we find the new optimal sizes and delay:

Signal	Fanout	Size of Driven Gate	Logical Effort ($K_p = 1.3$)	Delay
i to $i + 1$	$i + 1$	i to $i + 1$	$i + 1$	i to $i + 1$
b	1.70	3.19	1.87	3.17
cN	1.17	5.4	2.7	3.17
d	3.17	6.3	1	3.17
eN	3.17	20	1	3.17
TOTAL a to eN			$12.68t_{inv}$	

5.17 To solve this problem, we use the same methodology as in Example ??:

The size of our inverter is $W_N = 20 * 8L_{\min} = 160L_{\min}$.

$$\begin{aligned}
 R_w &= 10 \frac{\Omega}{\mu\text{m}} \times 500\mu\text{m} = 5000\Omega, \\
 C_w &= 0.18 \frac{fF}{\mu\text{m}} \times 500\mu\text{m} = 90\text{fF}, \\
 R_r &= \frac{K_{RN}}{W_N} = \frac{4.2 \times 10^4}{160} = 263\Omega, \\
 C_r &= W_N(1 + K_P)K_C = 160(1 + 1.3)2.8 \times 10^{-17} = 10.3\text{fF}.
 \end{aligned}$$

Using Equation (5.18) we compute the delay of each segment as:

$$\begin{aligned}
 D_l &= 0.4R_wC_w + R_rC_w + (R_r + R_w)C_r \\
 &= (0.4)(5000)(90) + (263)(90) + (5000 + 263)(10.3) \\
 &= 180000 + 23,670 + 54,209 = 258\text{ps}
 \end{aligned}$$

The delay of the entire wire, 20 segments, is $4D_l = (20)(258) = 5.16\text{ns}$.

- 5.20c** 1. As was shown in the text, the delay per millimeter of an optimally sized and spaced wire ($61\mu\text{m}$) is 228ps. 5mm is just 5 times that amount: 1.14ns. The energy to transmit one bit ($V_{dd} = 1$) is:

$$\begin{aligned} E &= \frac{1}{2}V_{dd}^2 * (C_{wire} + C_d) \\ E &= \frac{1}{2}V_{dd}^2 * (5000\mu\text{m} * 0.18 \frac{fF}{\mu\text{m}} + 82 * 108(1 + 1.3)2.8 \times 10^{-17}) \\ E &= 0.75pJ \end{aligned}$$

We found the total driver capacitance by multiplying the capacitance of each driver by the number of drivers.

2. By doubling the spacing between wires to $122\mu\text{m}$, we only need 41 segments. This reduces our driver energy by half and gives an energy per bit of 0.6pJ. The delay calculation is:

$$\begin{aligned} R_w &= 10 \frac{\Omega}{\mu\text{m}} \times 122\mu\text{m} = 1220\Omega, \\ C_w &= 0.18 \frac{fF}{\mu\text{m}} \times 122\mu\text{m} = 22fF, \\ R_r &= \frac{K_{RN}}{W_N} = \frac{4.2 \times 10^4}{108} = 389\Omega, \\ C_r &= W_N(1 + K_P)K_C = 108(1 + 1.3)2.8 \times 10^{-17} = 7.0fF \\ D_l &= 0.4R_wC_w + R_rC_w + (R_r + R_w)C_r \\ &= (0.4)(1220)(22) + (389)(22) + (1220 + 389)(7.0) \\ &= 180000 + 23,670 + 54,209 = 31ps \\ D &= 1.27ns \end{aligned}$$

- 5.22** We calculate energy in the same way we did in Exercise 5.4. Remember that the input capacitance is the product of the size and logical effort of a gate:

$$\begin{aligned} E_{101} &= \sum_{i=0}^n C_{i+1}V^2 \\ &= E_{inv} \sum_{i=0}^n \frac{C_{i+1}}{C_{inv}} \\ &= (1 + 2 * 1.87 + 4 * 2.7 + 4 + 20)E_{inv} \\ &= 39.5E_{inv} \end{aligned}$$

Chapter 6

Solutions: Combinational Logic Design

6.1 Circuits A, B, and D are all combinational. C is not combinational since the output of the middle block feeds the input of the leftmost block and the output of the leftmost block feeds the input of the middle block. No other diagram contains a cyclic composition of blocks.

6.2 (a)

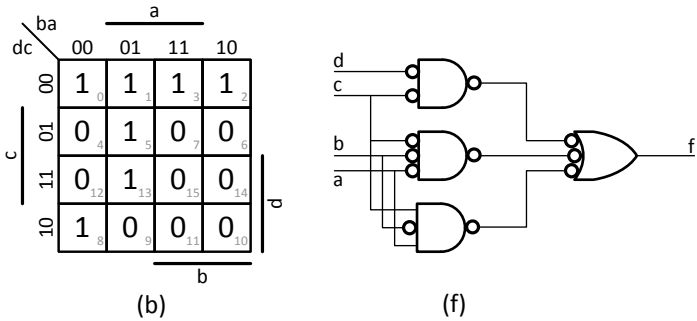
No.	in	out
0	0000	1
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	0
8	1000	1
9	1001	0
10	1010	0
11	1011	0
12	1100	0
13	1101	1
14	1110	0
15	1111	0

(b) See figure below, part (b)

(c)

Number of variables			
4	3	2	1
0000	000X	00XX	
0001	00X0		
0010	X000		
0011	00X1		
0101	0X01		
1000	001X		
1101	X101		

- The prime implicants of this function are 00XX, X000, 0X01, and X101.
- (d) The essential prime implicants of the function are 00XX (only cover of 2,3), X000 (only one to cover 8), and X101 (only one to cover 13).
- (e) The function is covered by the three essential prime implicants.
- (f) See figure below:



6.4 (a)

No.	in	out
0	0000	1
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	0
8	1000	1
9	1001	0
10	1010	X
11	1011	X
12	1100	X
13	1101	X
14	1110	X
15	1111	X

(b) See figure below, part (b)

Number of variables			
4	3	2	1
0000	000X	00XX	
0001	00X0	X0X0	
0010	X000	1XX0	
0011	00X1		
0101	0X01		
1000	001X		
	X010		
	X011		
	X101		
	10X0		
	1X00		

The prime implicants of this function are 00XX, X0X0, 1XX0, 0X01, X101, and X011.

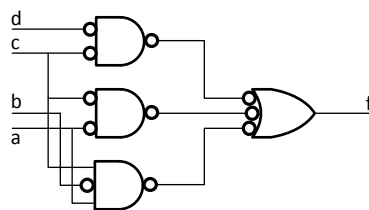
(c) The essential prime implicant of the function is 00XX (only cover of 3).

(d) The function is covered by the implicants 00XX, X0X0, and X101.

(e) See figure below:

		a			
		00	01	11	10
dc	ba				
	00	1 ₀	1 ₁	1 ₃	1 ₂
	01	0 ₄	1 ₅	0 ₇	0 ₆
	11	X ₁₂	X ₁₃	X ₁₅	X ₁₄
	10	1 ₈	0 ₉	X ₁₁	X ₁₀
		b			

(b)



(f)

6.9 To solve this problem, we list all implicants and minimize:

Number of variables				
5	4	3	2	1
00010	0001X	X0X11		
00011	00X11			
00101	0X011			
00111	X0011			
01011	001X1			
01101	0X101			
10001	X1101			
10011	100X1			
10111	10X11			
11101	1X111			
11111	111X1			

The essential prime implicants are: 0001X, 0X011, and 100X1. A possible cover of the function is: 0001x, 0x011, 100X1, X0X11, 0X101, and 111X1. One solutions in sum-of-products for is:

$$f(e, d, c, b, a) = (\bar{e} \wedge \bar{d} \wedge \bar{c} \wedge b) \vee (\bar{e} \wedge \bar{c} \wedge b \wedge a) \vee (e \wedge \bar{d} \wedge \bar{c} \wedge a) \vee (\bar{d} \wedge b \wedge a) \vee (\bar{e} \wedge c \wedge \bar{b} \wedge a) \vee (e \wedge d \wedge c \wedge a)$$

6.12 Using the same Karnaugh-map of Exercise 6.4, we can see the cover of maxterms (when the output is 0) is OR(0XX0), OR(X00X), and OR(X0X1). Our function is:

$$f(d, c, b, a) = (\bar{d} \vee \bar{a}) \wedge (\bar{c} \vee \bar{b}) \wedge (\bar{c} \vee a)$$

6.14 Segment 0 is lit for inputs 2, 3, 4, 5, 6, 8, 9, A, B, D, E, F. We show the Karnaugh-map below:

		a			
		00	01	11	10
dc	ba				
	00	0 ₀	0 ₁	1 ₃	1 ₂
	01	1 ₄	1 ₅	0 ₇	1 ₆
	11	0 ₁₂	1 ₁₃	1 ₁₅	1 ₁₄
	10	1 ₈	1 ₉	1 ₁₁	1 ₁₀

One possible cover is:

$$f(d, c, b, a) = (b \wedge \bar{a}) \vee (d \wedge \bar{c}) \vee (d \wedge a) \vee (\bar{d} \wedge c \wedge \bar{b}) \vee (\bar{d} \wedge \bar{c} \wedge b)$$

6.15 Segment 1 is lit for inputs 0, 4, 5, 6, 8, 9, A, B, C, E, F. We show the Karnaugh-map below:

		a			
		00	01	11	10
dc	ba				
	00	1 ₀	0 ₁	0 ₃	0 ₂
	01	1 ₄	1 ₅	0 ₇	1 ₆
	11	1 ₁₂	0 ₁₃	1 ₁₅	1 ₁₄
	10	1 ₈	1 ₉	1 ₁₁	1 ₁₀

One possible cover is:

$$f(d, c, b, a) = (\bar{b} \wedge \bar{a}) \vee (d \wedge \bar{c}) \vee (\bar{d} \wedge c \wedge \bar{b}) \vee (c \wedge b \wedge \bar{a}) \vee (d \vee c \vee b)$$

6.21 The Karnaugh-map is below:

		a			
		00	01	11	10
dc	ba				
	00	0 ₀	0 ₁	1 ₃	1 ₂
	01	1 ₄	1 ₅	0 ₇	1 ₆
	11	X ₁₂	X ₁₃	X ₁₅	X ₁₄
	10	1 ₈	1 ₉	X ₁₁	X ₁₀

One possible cover is:

$$f(d, c, b, a) = (b \wedge \bar{a}) \vee (d) \vee (c \wedge \bar{b}) \vee (\bar{d} \wedge \bar{c} \wedge b)$$

6.22 The Karnaugh-map is below:

		a			
		00	01	11	10
c	ba				
	dc				
	00	1 ₀	0 ₁	0 ₃	0 ₂
	01	1 ₄	1 ₅	0 ₇	1 ₆
	11	X ₁₂	X ₁₃	X ₁₅	X ₁₄
	10	1 ₈	1 ₉	X ₁₁	X ₁₀

One possible cover is:

$$f(d, c, b, a) = (\bar{b} \wedge \bar{a}) \vee (d) \vee (c \wedge \bar{b}) \vee (c \wedge \bar{a})$$

6.28 Using the same Karnaugh-map as Exercise 6.14, we can find a cover of maxterms as:

$$f(d, c, b, a) = (d \vee c \vee b) \wedge (d \vee \bar{c} \vee \bar{a}) \wedge (\bar{d} \vee \bar{c} \vee b \vee a)$$

6.35 Using the same Karnaugh-map as Exercise 6.21, we can find a cover of maxterms as:

$$f(d, c, b, a) = (d \vee c \vee b) \wedge (\bar{c} \wedge \bar{b} \wedge \bar{a})$$

6.42 Using the three Karnaugh-maps below (outputs 0, 1, 2 from L to R), we look for common terms in the coverage equations:

		a			
		00	01	11	10
c	ba				
	dc				
	00	0 ₀	0 ₁	1 ₃	1 ₂
	01	1 ₄	1 ₅	0 ₇	1 ₆
	11	X ₁₂	X ₁₃	X ₁₅	X ₁₄
	10	1 ₈	1 ₉	X ₁₁	X ₁₀

		a			
		00	01	11	10
c	ba				
	dc				
	00	1 ₀	0 ₁	0 ₃	0 ₂
	01	1 ₄	1 ₅	0 ₇	1 ₆
	11	X ₁₂	X ₁₃	X ₁₅	X ₁₄
	10	1 ₈	1 ₉	X ₁₁	X ₁₀

		a			
		00	01	11	10
c	ba				
	dc				
	00	1 ₀	0 ₁	0 ₃	1 ₂
	01	0 ₄	0 ₅	0 ₇	1 ₆
	11	X ₁₂	X ₁₃	X ₁₅	X ₁₄
	10	1 ₈	0 ₉	X ₁₁	X ₁₀

$$f(d, c, b, a)_0 = (b \wedge \bar{a}) \vee (d) \vee (c \wedge \bar{b}) \vee (\bar{d} \wedge \bar{c} \wedge b)$$

$$f(d, c, b, a)_1 = (\bar{b} \wedge \bar{a}) \vee (d) \vee (c \wedge \bar{b}) \vee (c \wedge \bar{a})$$

$$f(d, c, b, a)_2 = (\bar{c} \wedge \bar{a}) \vee (b \wedge \bar{a})$$

We can share the $b \wedge \bar{a}$ (XX10) term between outputs 0 and 2. We also share implicant $(c \wedge \bar{b})$ (X10X) between outputs 0 and 1.

- 6.44** The hazard occurs when $a = b = c = 1$ and then a toggles to 0. The output may go low for a period of time equal to that of the NOT-AND delay. The simplest fix for this problem is to remove the \bar{a} input from the AND gate. This simplifies the logic equation from $f = a \vee (c \wedge b \wedge \bar{a})$ to just $f = a \vee (c \wedge b)$.

Chapter 7

Solutions: VHDL Descriptions of Combinational Logic

7.1 *Fibonacci: case.*

```
library ieee;
use ieee.std_logic_1164.all;

entity fib_case is
  port( a: in std_logic_vector(3 downto 0);
        b: out std_logic );
end fib_case;

architecture impl of fib_case is
begin
  process(all) begin
    case a is
      when 4d"0" | 4d"1" | 4d"2" | 4d"3" | 4d"5" | 4d"8" | 4d"13" => b <= '1';
      when others => b <= '0';
    end case;
  end process;
end impl;
```

7.2 *Fibonacci: circuit: concurrent signal assignment.*

```
library ieee;
use ieee.std_logic_1164.all;

entity fib_assign is
  port( a: in std_logic_vector(3 downto 0);
        b: out std_logic );
```

```

end fib_assign;

architecture impl of fib_assign is
begin
    b <= (not a(3) and not a(2)) or
        (not a(2) and not a(1) and not a(0)) or
        (a(2) and not a(1) and a(0));
end impl;

```

7.3 Fibonacci: circuit: structural.

```

library ieee;
use ieee.std_logic_1164.all;

entity MY_AND2 is
    port( a, b : in std_logic;
          z: out std_logic );
end MY_AND2;

architecture impl of MY_AND2 is
begin
    z <= a and b;
end impl;

library ieee;
use ieee.std_logic_1164.all;

entity MY_AND3 is
    port( a, b, c : in std_logic;
          z: out std_logic );
end MY_AND3;

architecture impl of MY_AND3 is
begin
    z <= a AND b AND c;
end impl;

library ieee;
use ieee.std_logic_1164.all;

entity MY_OR3 is
    port( a, b, c: in std_logic;
          z: out std_logic );
end MY_OR3;

architecture impl of MY_OR3 is
begin
    z <= a or b or c;

```

```

end impl;

library ieee;
use ieee.std_logic_1164.all;

entity fib_struct is
  port( a : in std_logic_vector(3 downto 0);
        b : out std_logic );
end fib_struct;
architecture impl of fib_struct is
  component MY_AND2 is
    port( a, b : in std_logic;
          z: out std_logic );
  end component;
  component MY_AND3 is
    port( a, b, c : in std_logic;
          z: out std_logic );
  end component;
  component MY_OR3 is
    port( a, b, c : in std_logic;
          z: out std_logic );
  end component;
  signal t0, t1, t2 : std_logic;
begin
  andt0: MY_AND2 port map(not a(3), not a(2), t0);
  andt1: MY_AND3 port map(not a(2), not a(1), not a(0), t1);
  andt2: MY_AND3 port map(a(2), not a(1), a(0), t2);
  orb: MY_OR3 port map(t0, t1, t2, b);
end impl;

```

7.4 The testbench below requires the user to manually verify the output of dut0. It automatically checks that all 4 implementations provide the same answer.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity fib_tb is
end fib_tb;

architecture test of fib_tb is
  signal a: std_logic_vector(3 downto 0);
  signal o: std_logic_vector(2 downto 0);
  signal error: std_logic;
begin
  DUT0: entity work.fib_case(impl) port map (a, o(0));

```

```

DUT1: entity work.fib_assign(impl) port map(a, o(1));
DUT2: entity work.fib_struct(impl) port map(a, o(2));

process begin
  a <= 4x"0";
  error <= '0';
  for i in 0 to 15 loop
    wait for 10 ns;
    report to_string(to_integer(unsigned(a))) & " -> " & to_string(o(0));
    if o(0) /= o(1) then error <= '1'; end if;
    if o(0) /= o(2) then error <= '1'; end if;
    a <= a + 1;
  end loop;
  if error = '0' then report "Tests Passed";
  else report "Tests Failed"; end if;
  std.env.stop(0);
end process;
end test;

```

7.9 The following is the right approach as it allows us to simply encode the truth table.

```

library ieee;
use ieee.std_logic_1164.all;

entity dec_fib_case is
  port( a : in std_logic_vector(3 downto 0);
        b : out std_logic );
end dec_fib_case;

architecture impl of dec_fib_case is
begin
  process(all) begin
    case a is
      when 4d"0" | 4d"1" | 4d"2" | 4d"3" | 4d"5" | 4d"8" => b <= '1';
      when 4d"4" | 4d"6" | 4d"7" | 4d"9" => b <= '0';
      when others => b <= '-';
    end case;
  end process;
end impl;

```

7.14 *Bit reversal:*

```

library ieee;
use ieee.std_logic_1164.all;

entity bit_reverse is
  port( a : in std_logic_vector(4 downto 0);

```

```

        b : out std_logic_vector(4 downto 0) );
end bit_reverse;

```

```

architecture impl of bit_reverse is
begin
    b <= a(0) & a(1) & a(2) & a(3) & a(4);
end impl;

```

7.23 Find-first-one.

```

library ieee;
use ieee.std_logic_1164.all;

entity ffl is
    port( a: in std_logic_vector(15 downto 0);
          o: out std_logic_vector(3 downto 0);
          v: out std_logic );
end ffl;

architecture impl of ffl is
begin
    process(all) begin
        case? a is
            when 16b"1-----" => v <= '1'; o <= 4x"F";
            when 16b"01-----" => v <= '1'; o <= 4x"E";
            when 16b"001-----" => v <= '1'; o <= 4x"D";
            when 16b"0001-----" => v <= '1'; o <= 4x"C";
            when 16b"00001-----" => v <= '1'; o <= 4x"B";
            when 16b"000001-----" => v <= '1'; o <= 4x"A";
            when 16b"0000001-----" => v <= '1'; o <= 4x"9";
            when 16b"00000001-----" => v <= '1'; o <= 4x"8";
            when 16b"000000001-----" => v <= '1'; o <= 4x"7";
            when 16b"0000000001-----" => v <= '1'; o <= 4x"6";
            when 16b"00000000001-----" => v <= '1'; o <= 4x"5";
            when 16b"000000000001-----" => v <= '1'; o <= 4x"4";
            when 16b"0000000000001-----" => v <= '1'; o <= 4x"3";
            when 16b"00000000000001-----" => v <= '1'; o <= 4x"2";
            when 16b"000000000000001-----" => v <= '1'; o <= 4x"1";
            when 16b"0000000000000001" => v <= '1'; o <= 4x"0";
            when others => v <= '0'; o <= 4x"0";
        end case?;
    end process;
end impl;

```


Chapter 8

Solutions: Combinational Building Blocks

8.1 *Decoder.*

```
library ieee;
use ieee.std_logic_1164.all;

entity dec38a is
  port( a : in std_logic_vector(2 downto 0);
        b : out std_logic_vector(7 downto 0) );
end dec38a;

architecture impl of dec38a is
begin
  b(0) <= not a(0) and not a(1) and not a(2);
  b(1) <=  a(0) and not a(1) and not a(2);
  b(2) <= not a(0) and  a(1) and not a(2);
  b(3) <=  a(0) and  a(1) and not a(2);
  b(4) <= not a(0) and not a(1) and  a(2);
  b(5) <=  a(0) and not a(1) and  a(2);
  b(6) <= not a(0) and  a(1) and  a(2);
  b(7) <=  a(0) and  a(1) and  a(2);
end impl;
```

8.2 *Decoder logic.*

```
library ieee;
use ieee.std_logic_1164.all;
use work.sseg_constants.all; -- for SS_0, ... defined in ch07_sseg.vhd
use work.ch8.all; -- for Dec

entity ssdec is
  port( a : in std_logic_vector(3 downto 0);
```

```

        b : out std_logic_vector(6 downto 0) );
end ssdec;

architecture impl of ssdec is
    signal w : std_logic_vector(15 downto 0);
begin
    b <= (SS_0 and (6 downto 0 => w(0))) or
        (SS_1 and (6 downto 0 => w(1))) or
        (SS_2 and (6 downto 0 => w(2))) or
        (SS_3 and (6 downto 0 => w(3))) or
        (SS_4 and (6 downto 0 => w(4))) or
        (SS_5 and (6 downto 0 => w(5))) or
        (SS_6 and (6 downto 0 => w(6))) or
        (SS_7 and (6 downto 0 => w(6)));
    Decoder: Dec generic map(4,16) port map(a,w);
end impl;

```

8.4 Large decoder, I.

```

library ieee;
use ieee.std_logic_1164.all;
use work.ch8.all; -- for Dec

entity dec532 is
    port( a : in std_logic_vector(4 downto 0);
          b : out std_logic_vector(31 downto 0) );
end dec532;

architecture impl of dec532 is
    signal x: std_logic_vector(7 downto 0);
    signal y: std_logic_vector(3 downto 0);
begin
    b(7 downto 0)  <= x and (7 downto 0 => y(0));
    b(15 downto 8) <= x and (7 downto 0 => y(1));
    b(23 downto 16) <= x and (7 downto 0 => y(2));
    b(31 downto 24) <= x and (7 downto 0 => y(3));

    D0: Dec generic map(3,8) port map( a(2 downto 0), x);
    D1: Dec generic map(2,4) port map( a(4 downto 3), y);
end impl;

```

8.8 Multiplexer logic.

```

library ieee;
use ieee.std_logic_1164.all;
use work.ch8.all; -- for Muxb8

entity multFib is

```



```

    port( a : in std_logic_vector(3 downto 0);
          b : out std_logic );
end multFib;

architecture impl of multFib is
    signal na3, bv: std_logic_vector(0 downto 0);
begin
    na3(0) <= not a(3);
    MUX: Muxb8 generic map(1)
        port map("0","0","1","0",na3, na3, na3, "1", a(2 downto 0), bv);
    b <= bv(0);
end impl;

```

- 8.11 For this problem, we had to reverse the priority polarity from the example given in the chapter. To do so, we no compute C from the MSB to the LSB.

```

library ieee;
use ieee.std_logic_1164.all;
use work.ch8.all; -- for Enc83

entity progPriEnc83 is
    port( a, p: in std_logic_vector(7 downto 0);
          b: out std_logic_vector(2 downto 0) );
end progPriEnc83;

architecture impl of progPriEnc83 is
    signal g: std_logic_vector(7 downto 0);
    signal c: std_logic_vector(15 downto 0);
begin
    c <= (('0' & c(15 downto 1)) and ('0' & not a & not a(7 downto 1))) or
        (p & 8x"0");
    g <= a and (c(15 downto 8) or c(7 downto 0));
    ENC: Enc83 port map(g, b);
end impl;

```

- 8.14 *Comparator.*

```

library ieee;
use ieee.std_logic_1164.all;

entity MagCompML is
    generic( k : integer := 8 );
    port( a, b: in std_logic_vector(k-1 downto 0);
          gt: out std_logic );
end MagCompML;

architecture impl of MagCompML is

```

```

    signal eqi, gti: std_logic_vector(k-1 downto 0);
    signal eqa, gta: std_logic_vector(k downto 0);
begin
    eqi <= a nand b;
    gti <= a and not b;
    eqa <= '1' & (eqi(k-1 downto 0) and eqa(k downto 1));
    gta <= '0' & (gta(k downto 1) or (gti(k-1 downto 0) and eqa(k downto 1)));
    gt <= gta(0);
end impl;

```

8.17 Funnel-shifter.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity funnelShift is
    generic( i: integer := 16;
             j: integer := 8;
             l: integer := 3 );
    port( a: in std_logic_vector(i-1 downto 0);
          n: in std_logic_vector(l-1 downto 0);
          b: out std_logic_vector(j-1 downto 0) );
end funnelShift;

architecture impl of funnelShift is
    signal shift: integer;
    signal tmp: std_logic_vector(i-1 downto 0);
begin
    shift <= to_integer(unsigned(n));
    tmp <= std_logic_vector(unsigned(a) srl shift);
    b <= tmp(7 downto 0);
end impl;

```

8.19 Using building blocks, II. Below we use the magnitude comparator from exercise 8.14.

```

library ieee;
use ieee.std_logic_1164.all;
use work.ch8.all; -- for Mux3

entity findMin is
    generic( n: integer := 16 );
    port( a, b, c: in std_logic_vector(n-1 downto 0);
          z: out std_logic_vector(n-1 downto 0) );
end findMin;

architecture impl of findMin is

```

```

    signal agtb, agtc, bgtc: std_logic;
begin
    AB: entity work.MagCompML(impl) generic map(n) port map(a,b,agtb);
    AC: entity work.MagCompML(impl) generic map(n) port map(a,c,agtc);
    BC: entity work.MagCompML(impl) generic map(n) port map(b,c,bgtc);

    MOUT: Mux3 generic map(n)
        port map(a, b, c, (not agtb and not agtc) & (not bgtc and agtb) & (agtc and bgtc), z);
end impl;

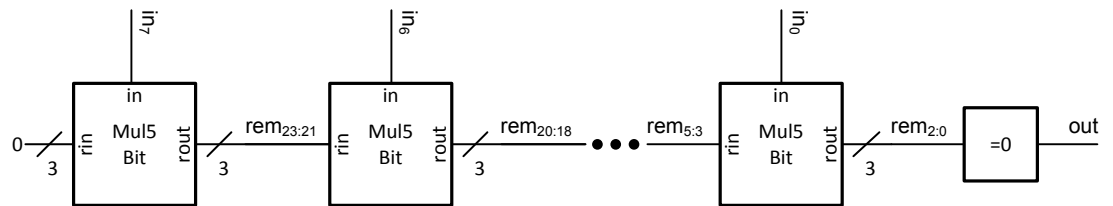
```

- 8.21** Our ROM would need to have 16 entries (one for each number). Addressed with the 4 input bits, each value in our table would be a single bit. We store a 1 in locations indexed by a prime number (2, 3, 5, 7, etc.) and a 0 everywhere else.

Chapter 9

Solutions: Combinational Examples

9.3 The way we build our multiple-of-5 circuit is analogous to the multiple of 3, as shown below:



We keep the intermediate remainder in a 3 bit format, in order to represent 0-4. We show the logic for computing next remainder in the next solution.

9.4 *Multiple-of-5 circuit, implementation.*

```
library ieee;
use ieee.std_logic_1164.all;

entity Multiple_of_5_bit is
  port( inp: in std_logic;
        rin: in std_logic_vector(2 downto 0);
        rout: out std_logic_vector(2 downto 0) );
end Multiple_of_5_bit;

architecture impl of Multiple_of_5_bit is
begin
  process(all) begin
```

```

        case rin & inp is
            when "0000" => rout <= 3d"0";
            when "0001" => rout <= 3d"1";
            when "0010" => rout <= 3d"2";
            when "0011" => rout <= 3d"3";
            when "0100" => rout <= 3d"4";
            when "0101" => rout <= 3d"0";
            when "0110" => rout <= 3d"1";
            when "0111" => rout <= 3d"2";
            when "1000" => rout <= 3d"3";
            when "1001" => rout <= 3d"4";
            when others => rout <= "---";
        end case;
    end process;
end impl;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all; -- for or_reduce

entity Multiple_of_5 is
    port( i: in std_logic_vector(7 downto 0);
          o: out std_logic );
end Multiple_of_5;

architecture impl of Multiple_of_5 is
    signal r: std_logic_vector(23 downto 0);
begin
    B7: entity work.Multiple_of_5_bit(impl) port map(i(7),"000",r(23 downto 21));
    B6: entity work.Multiple_of_5_bit(impl) port map(i(6),r(23 downto 21),r(20 downto 18));
    B5: entity work.Multiple_of_5_bit(impl) port map(i(5),r(20 downto 18),r(17 downto 15));
    B4: entity work.Multiple_of_5_bit(impl) port map(i(4),r(17 downto 15),r(14 downto 12));
    B3: entity work.Multiple_of_5_bit(impl) port map(i(3),r(14 downto 12),r(11 downto 9));
    B2: entity work.Multiple_of_5_bit(impl) port map(i(2),r(11 downto 9),r(8 downto 6));
    B1: entity work.Multiple_of_5_bit(impl) port map(i(1),r(8 downto 6),r(5 downto 3));
    B0: entity work.Multiple_of_5_bit(impl) port map(i(0),r(5 downto 3),r(2 downto 0));
    o <= not or_reduce(r(2 downto 0));
end impl;

-- pragma translate_off
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity testMul5 is
end testMul5;

```

```

architecture test of testMul5 is
    signal input: std_logic_vector(7 downto 0);
    signal error, output, mod5: std_logic;
begin
    DUT: entity work.Multiple_of_5(impl) port map(input,output);

    process begin
        input <= 8x"0";
        error <= '0';

        for i in 0 to 255 loop
            wait for 5 ns;
            if ((to_integer(unsigned(input)) mod 5) = 0) then
                mod5 <= '1';
            else mod5 <= '0'; end if;
            wait for 5 ns; -- allow update to mod5 to take effect before next line
            if output /= mod5 then
                report "ERROR " & to_string(input) & " -> " & to_string(output);
                error <= '1';
            end if;
            input <= input + 1;
        end loop;

        if error = '0' then report "MUL5 PASS";
        else report "MUL5 Errors found"; end if;
        std.env.stop(0);
    end process;
end test;
-- pragma translate_on

```

9.9 One correct solution is to implement this function using a case statement. For example: **when** MONDAY => tomorrow <= TUESDAY;.

9.10 One possible solution is to pass the year input to the DaysInMonth module. We can then modify the case statement:

```

case? year(1 downto 0) & month is
    when "--0100" => days <= 5d"30";
    when "--0110" => days <= 5d"30";
    when "--1001" => days <= 5d"30";
    when "--1011" => days <= 5d"30";
    when "000010" => days <= 5d"29"; -- year divisible by 4 => leap year
    when "--0010" => days <= 5d"28";
    when others => days <= 5d"31";
end case?;

```

9.12 There are two different options when designing this circuit. First, we could modify the comparator block to output a signal **gteq** for greater than or

equal. Leaving the rest of the logic as is, the arbiter would break ties to the higher input. A second option is to simply switch the **a** and **b** inputs to each comparator.

9.16 The simplest way to encode the lowest output is to place an inverter at the output of each magnitude comparator. That way, the comparator output is the \leq value. Ties are broken in favor of the higher number input.

9.18 A version of **OneInRow** and **OneInArray** are shown below. We do not use any sort of strategy in selecting the next position to play. We integrated the **OneInArray** function into the main module as the 2nd lowest priority. The **OneInArray** module itself is nearly identical to the **TwoInArray** module.

```

library ieee;
use ieee.std_logic_1164.all;

entity OneInRow is
  port( ain, bin: in std_logic_vector(2 downto 0));
    cout: out std_logic_vector(2 downto 0) );
end OneInRow;

architecture impl of OneInRow is
begin
  -- cout(0) = X__ or __X
  cout(0) <= (ain(2) and not (ain(1) or bin(1)) and not (ain(0) or bin(0))) or
    (not (ain(2) or bin(2)) and ain(1) and not (ain(0) or bin(0)));
  -- cout(1) = __X
  cout(1) <= not (ain(2) or bin(2)) and not (ain(1) or bin(1)) and ain(0);
  cout(2) <= '0';
end impl;

library ieee;
use ieee.std_logic_1164.all;

entity OneInArray is
  port( ain, bin: in std_logic_vector(8 downto 0));
    cout: out std_logic_vector(8 downto 0) );
end OneInArray;

architecture impl of OneInArray is
  signal rows, cols, cc: std_logic_vector(8 downto 0);
  signal ddiag, udiag: std_logic_vector(2 downto 0);
begin

  -- check each row
  TOPR: entity work.OneInRow(impl)
    port map(ain(2 downto 0),bin(2 downto 0),rows(2 downto 0));

```



```

MIDR: entity work.OneInRow(impl)
    port map(ain(5 downto 3),bin(5 downto 3),rows(5 downto 3));
BOTR: entity work.OneInRow(impl)
    port map(ain(8 downto 6),bin(8 downto 6),rows(8 downto 6));

-- check each column
LEFTC: entity work.OneInRow(impl)
    port map( ain(6) & ain(3) & ain(0),
              bin(6) & bin(3) & bin(0),
              cc(8 downto 6) );
(cols(6),cols(3),cols(0)) <= cc(8 downto 6);

MIDC: entity work.OneInRow(impl)
    port map( ain(7) & ain(4) & ain(1),
              bin(7) & bin(4) & bin(1),
              cc(5 downto 3) );
(cols(7),cols(4),cols(1)) <= cc(5 downto 3);

RIGHTC: entity work.OneInRow(impl)
    port map( ain(8) & ain(5) & ain(2),
              bin(8) & bin(5) & bin(2),
              cc(2 downto 0) );
(cols(8),cols(5),cols(2)) <= cc(2 downto 0);

-- check both diagonals
DNDIAGX: entity work.OneInRow(impl)
    port map(ain(8) & ain(4) & ain(0), bin(8) & bin(4) & bin(0), ddiag);
UPDIAGX: entity work.OneInRow(impl)
    port map(ain(6) & ain(4) & ain(2), bin(6) & bin(4) & bin(2), udiag);

-- OR together the outputs
cout <= rows or cols or
        (ddiag(2)&'0'&'0'&'0'&ddiag(1)&'0'&'0'&'0'&ddiag(0)) or
        ('0'&'0'&udiag(2)&'0'&udiag(1)&'0'&udiag(0)&'0'&'0');
end impl;

```

9.21 Tic-tac-toe, input check.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_unsigned.all;
use work.ch8.all;

entity tttLegal is
    port( a, b: in std_logic_vector(8 downto 0);
          legal: out std_logic );
end tttLegal;

```

```

architecture impl of tttLegal is
  signal noConflict: std_logic;
  signal nX, n0: std_logic_vector(3 downto 0);
  signal illegalCnt: std_logic_vector(1 downto 0);
begin
  noConflict <= not or_reduce(a and b);
  -- Count the number of ones
  nX <= ("000" & a(8)) + a(7) + a(6) + a(5) +
    a(4) + a(3) + a(2) + a(1) + a(0);
  n0 <= ("000" & b(8)) + b(7) + b(6) + b(5) +
    b(4) + b(3) + b(2) + b(1) + b(0);
  -- want nx + 1 >= n0 & n0 >= nx
  --      ~(nX + 1 < n0) & ~(nX > n0)
  M0: MagComp generic map(4) port map(n0, nX + 1,illegalCnt(0));
  M1: MagComp generic map(4) port map(nX, n0, illegalCnt(1));
  legal <= noConflict and not illegalCnt(1) and not illegalCnt(0);
end impl;

```

Chapter 10

Solutions: Arithmetic Circuits

10.1 $817_{10} = 1100110001_2 = 331_{16}$

10.2 $1492_{10} = 10111010100_2 = 5D4_{16}$

10.5 $001100110001_2 = 2^9 + 2^8 + 2^5 + 2^4 + 1 = 817_{10}$

10.9 $2C_{16} = 00101100_2 = 11 * 16^3 = 44_{10}$

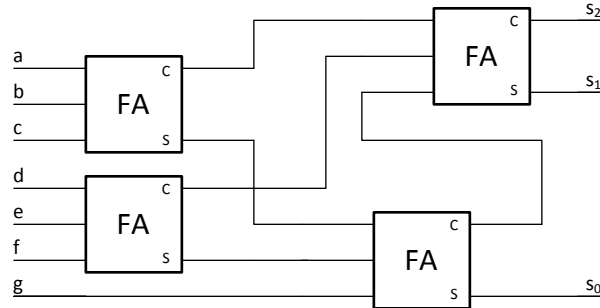
10.10 $BEEF_{16} = 1011111011101111_2 = 11 * 16^3 + 14 * 16^2 + 14 * 16 + 15 = 48879_{10}$

10.14
$$\begin{array}{r} 1010 \\ + 0111 \\ \hline 10001 \end{array}$$

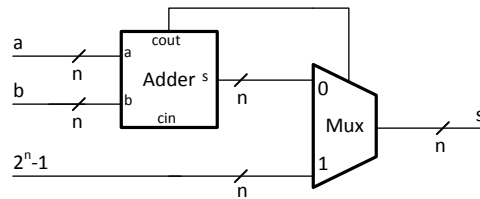
10.16 We can either perform this addition in hexadecimal, or convert to another base and then convert the resulting answer back to hexadecimal. We'll do both below (carries are shown in the top row):

$$\begin{array}{rcl} 1 & 0111 & 000 \\ 2A & 0010 & 1010_2 \\ +3C & 0011 & 1100_2 \\ \hline 66 & 0110 & 0110_2 \end{array}$$

10.18 To count the seven input bits, labeled $abcdefg$, we need four full adders. The first two have binary weight 0 and count the number of 1s in inputs abc and def . Next is a third FA with weight 0 that counts the two previous sums and the seventh input, g . Finally, the three carry-outs of the FAs are sent to a final adder with weight 2^1 . The schematic is shown below:



10.20 Since we are not concerned about subtraction (yet), an overflow condition is detected when any **cout** is produced by the adder. This output selects, using a multiplexer, between the computed sum and $2^n - 1$ as shown below:



10.25

Sign-magnitude	0 001 0001
1's complement	0001 0001
2's complement	0001 0001

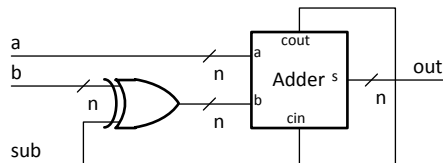
10.26

Sign-magnitude	1 001 0001
1's complement	1110 1110
2's complement	1110 1111

10.29 To do the subtraction, we first take the 2's complement of $-0110_2 = 1010_2$. Next, we add $0101_2 + 1010_2 = 1111_2 = -1_{10}$.

10.30 To do the subtraction, we first take the 2's complement of $-1110_2 = 0010_2$. Next, we add $0101_2 + 0010_2 = 0111_2 = 7_{10}$.

10.33 Our 1s complement adder is shown below. When subtracting, we simply negate (take the 1s complement) the b input. Unlike a 2s complement adder, however, we connect the carry-out of the adder back onto the carry-in. To see why, examine the number wheel of Figure 10.9(b). When we add two numbers, we count off positions going clockwise. When counting



10.40 The multiplication is shown below:

[illegible]

Diagram of a ripple-carry adder block. The block is labeled "Adder s". It has two inputs, "a" and "b", both labeled "n+2". It has two outputs, "cout" and "f", both labeled "n+3". The "cout" output is connected to the "cin" input of the next block. The "f" output is connected to the "f" input of the next block.

10.52 The division is shown below:

$$\begin{array}{r}
 001001 \\
 101 \overline{)101110} \\
 \underline{-101000} \\
 00110 \\
 \underline{-101} \\
 001
 \end{array}$$

10.54 We could do the division by converting to binary (or decimal) and then converting back to hex. Instead, we list the multiplication table for E to find the quotient. By looking in the table below, we find that $AE \div E = C$. The remainder is $AE - A8 = 6$.

×	E
0	0
1	E
2	1C
3	2A
4	38
5	46
6	54
7	62
8	70
9	7E
A	8C
B	9A
C	A8
D	B6
E	C4
F	D2

Chapter 11

Solutions: Fixed- and Floating-Point Numbers

11.1

$$1.0101_2 = 1 \times 1 + 1 \times 0.25 + 1 \times 0.0625 = 1.3125_{10}$$

11.2 The number represented is $-\frac{11}{16}$.

11.5 The first two bits are 01 to represent a positive one integer. We find the remaining digits as follows:

$$\begin{array}{r} 0.5999_{10} \quad 0.00000_2 \\ \underline{-0.5_{10}} \quad \underline{+0.10000_2} \\ 0.0999_{10} \quad 0.0010_2 \\ \underline{-0.0625_{10}} \quad \underline{+0.0001_2} \\ 0.0374_{10} \quad 0.10010_2 \\ \underline{-0.03125_{10}} \quad \underline{+0.00001_2} \\ 0.00615_{10} \quad 0.10011_2 \end{array}$$

Our fixed-point number is 01.10011_2 or 1.59375_{10} . The absolute error is 0.00615 and the relative error is 0.38%. We did not round our fixed-point value because $0.00615 < \frac{1}{64}$.

11.6 The first two bits are 00 to represent a positive number with no integer component. We find the remaining digits as follows:

$$\begin{array}{r} 0.3775_{10} \quad 0.00000_2 \\ \underline{-0.25_{10}} \quad \underline{+0.01_2} \\ 0.1275_{10} \quad 0.01000_2 \\ \underline{-0.125_{10}} \quad \underline{+0.001_2} \\ 0.0025_{10} \quad 0.01100_2 \end{array}$$

Our fixed-point number is 00.01100_2 or 0.375_{10} . The absolute error is 0.0025 and the relative error is 0.66%.

11.9 If we assume a basic rounding scheme, the precision of our system is half our resolution. Since the resolution is $\frac{1}{32}$, our precision and maximum absolute error is $\frac{1}{64}$. Any odd multiple of $\frac{1}{64}$ has maximal absolute error.

11.10 The maximal relative error occurs at $\frac{7}{64} = 0.109375$. To view this graphically, you can use the following WolframAlpha¹® command:

`Maximize[{Abs[Round[x 32]/32 - x]/x, 0.1 <= x <= 0.2}, {x}]`

11.11 Because we have both positive and negative numbers, we need a sign bit. We also need 4 integral bits to represent magnitudes from 0 to 10_{10} . The minimal accuracy is $\frac{1}{16} = 0.0625$ which gives a resolution of $\frac{1}{8}$. Thus, our format is s4.3.

11.13 The value is $\frac{15}{16} \times 2^{7-3} = 15$.

11.14 The value is $-1 \times \frac{4}{8} \times 2^{1-3} = -0.125$.

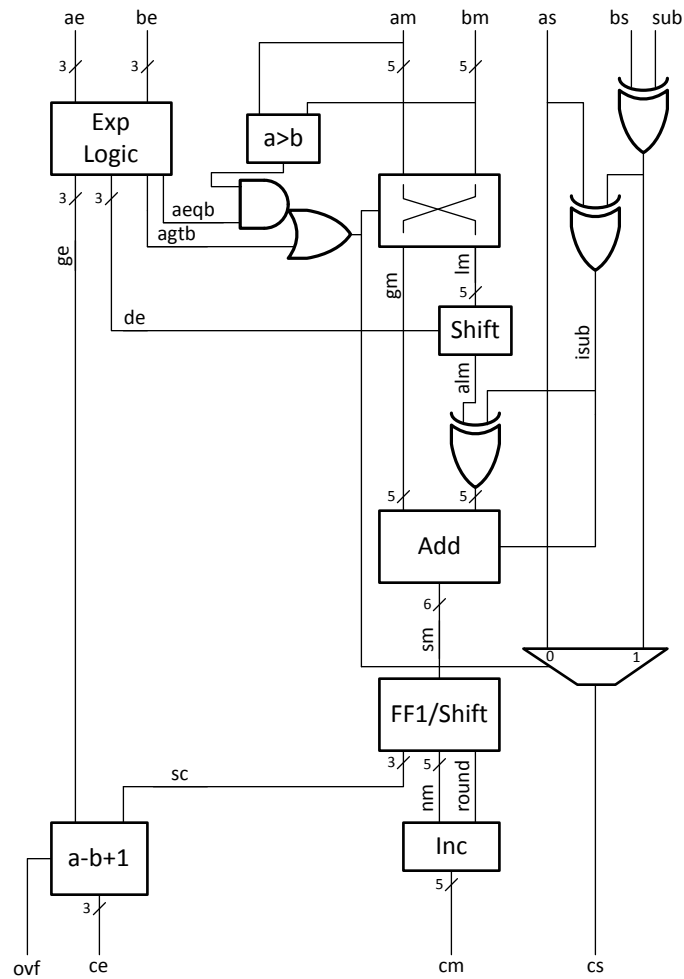
11.17 The sign bit is 1, since the number is negative. 23 in binary is 10011, which is rounded to $101 \times 2^2 = \frac{5_{10}}{8_{10}} \times 2^5$. The exponent is the sum of the bias and 5: 01101. The answer is 1101E01101. The absolute error is $24 - 23 = 1$ and the relative error is 4.3%.

11.18 The sign bit is 0, and $1000000_{10} = F4240_{16} \approx 100 \times 2^{18} = \frac{4_{10}}{8_{10}} \times 2^{21}$. Adding the bias gives a final answer 0100E11101. The number represented is $2^{20} = 1048576$, an error of 48 576, or 4.9%.

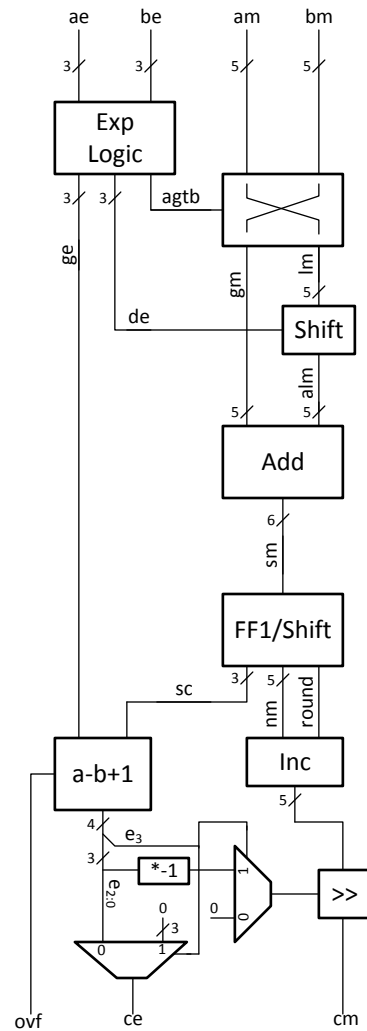
11.21 The mantissa of this representation needs to be 4 bits to bound the error to 10%. The smallest magnitude that must be represented accurately is $\frac{1}{32} = \frac{8}{16} \times 2^{-4}$. Our exponent bias is -4 and the maximum exponent is 4, 9 different values. The final representation is s4E4.

11.25 Our additions to the floating point adder presented are shown below. The initial design included a FF1 shift unit capable of shifting the LSB to MSB, so we did not modify it. We selectively invert the number with the smaller magnitude based on the XOR of both input signs and the subtract signal. The output sign, *cs* is that of the input with the highest magnitude (*bs* is inverted in subtraction). Note that the highest magnitude comparison must include the mantissa bits of the input to handle exponent ties.

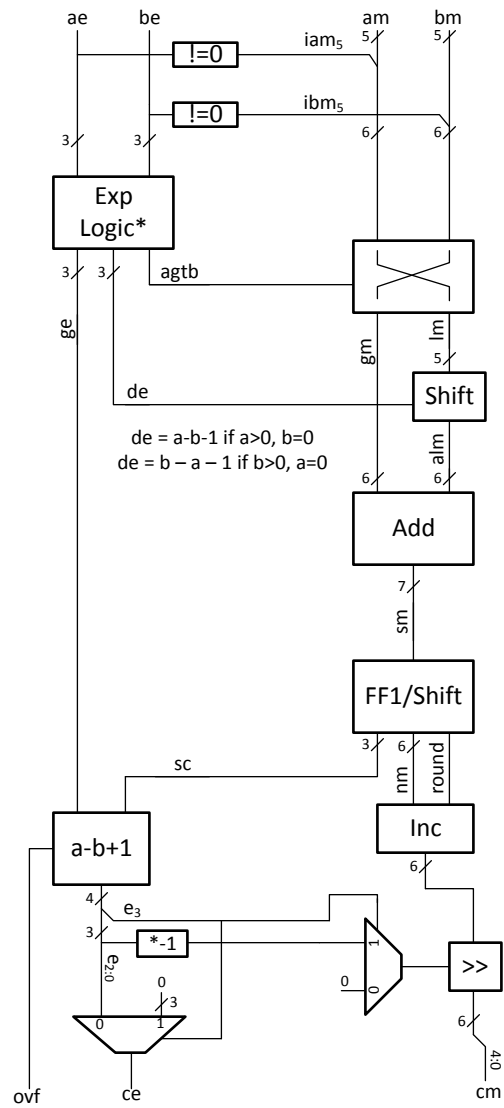
¹www.wolframalpha.com



11.29 Shown below, we handle gradual underflow by right shifting the mantissa if the newly computed exponent is less than 0. We also include a MUX to clamp the output exponent to 0.



11.32 Our adder first must include the implicit-1 if present. We must also modify the shift logic to account for the fact that exponents of 0 and 1 weight the mantissa equally. We then use the same logic as in the previous problem (omitting the mantissa MSB) for final exponent and mantissa calculation.



Chapter 12

Solutions: Fast Arithmetic Circuits

12.1 We use the propagate and generate equations shown in Equations 12.10-12.12 to formulate our comparator, setting the output to the generate signal of the final comparator. (The propagate signal of the final comparator signals equality.)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;

entity PG5 is
  port( pi, gi: in std_logic_vector(4 downto 0);
        po, go: out std_logic );
end PG5;

architecture impl of PG5 is
begin
  po <= and_reduce(pi);
  go <= gi(4) or (gi(3) and pi(4)) or (gi(2) and pi(4) and pi(3)) or
      (gi(1) and and_reduce(pi(4 downto 2))) or
      (gi(0) and and_reduce(pi(4 downto 1)));
end impl;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;

entity PG6 is
  port( pi, gi: in std_logic_vector(5 downto 0);
        po, go: out std_logic );
end PG6;
```

```

architecture impl of PG6 is
begin
    po <= and_reduce(pi);
    go <= gi(5) or (gi(4) and pi(5)) or (gi(3) and pi(5) and pi(4)) or
        (gi(2) and pi(5) and pi(4) and pi(3)) or (gi(1) and and_reduce(pi(5 downto 2))) or
        (gi(0) and and_reduce(pi(5 downto 1)));
end impl;

library ieee;
use ieee.std_logic_1164.all;

entity comp32 is
    port( a, b: in std_logic_vector(31 downto 0);
        agtb: out std_logic );
end comp32;

architecture impl of comp32 is
    signal g, p: std_logic_vector(31 downto 0);
    signal p6, g6: std_logic_vector(5 downto 0);
    signal p32: std_logic;
begin
    g <= a and not b;
    p <= not(a xor b);
    PG10: entity work.PG6(impl) port map(p(5 downto 0), g(5 downto 0),
        p6(0), g6(0));
    PG11: entity work.PG6(impl) port map(p(11 downto 6), g(11 downto 6),
        p6(1), g6(1));
    PG12: entity work.PG5(impl) port map(p(16 downto 12), g(16 downto 12), p6(2), g6(2));
    PG13: entity work.PG5(impl) port map(p(21 downto 17), g(21 downto 17), p6(3), g6(3));
    PG14: entity work.PG5(impl) port map(p(26 downto 22), g(26 downto 22), p6(4), g6(4));
    PG15: entity work.PG5(impl) port map(p(31 downto 27), g(31 downto 27), p6(5), g6(5));

    PG2: entity work.PG6(impl) port map(p6(5 downto 0), g6(5 downto 0), p32, agtb);
end impl;

```

12.3 The code is shown below and does not use PG modules. Instead, we use a look ahead tree to detect the presence of a 1 in any lower (higher priority) bits.

```

library ieee;
use ieee.std_logic_1164.all;

entity arb4 is
    port( bi: in std_logic;
        g: in std_logic_vector(2 downto 0);
        bo: out std_logic_vector(2 downto 0) );
end arb4;

```

```

architecture impl of arb4 is
begin
    bo(0) <= bi or g(0);
    bo(1) <= bi or g(0) or g(1);
    bo(2) <= bi or g(0) or g(1) or g(2);
end impl;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;

entity claArb is
    port( req: in std_logic_vector(31 downto 0);
          gnt: out std_logic_vector(31 downto 0) );
end claArb;

architecture impl of claArb is
    component arb4 is
        port( bi: in std_logic;
              g: in std_logic_vector(2 downto 0);
              bo: out std_logic_vector(2 downto 0) );
    end component;
    signal g4: std_logic_vector(7 downto 0);
    signal g8: std_logic_vector(1 downto 0);
    signal b: std_logic_vector(31 downto 0); -- 1 below this number?
    signal b_A20, b_A21, b_A10, b_A11, b_A12: std_logic_vector(2 downto 0);
    signal b_A13, b_A14, b_A15, b_A16, b_A17: std_logic_vector(2 downto 0);
begin
    g4(0) <= or_reduce(req(3 downto 0));
    g4(1) <= or_reduce(req(7 downto 4));
    g4(2) <= or_reduce(req(11 downto 8));
    g4(3) <= or_reduce(req(15 downto 11));
    g4(4) <= or_reduce(req(19 downto 16));
    g4(5) <= or_reduce(req(23 downto 20));
    g4(6) <= or_reduce(req(27 downto 24));
    g4(7) <= or_reduce(req(31 downto 28));

    g8(0) <= or_reduce(g4(3 downto 0)); -- bits 15:0
    g8(1) <= or_reduce(g4(7 downto 4)); -- bits 31:0

    b(0) <= '0';
    b(16) <= g8(0);
    A20: arb4 port map(b(0), g4(2 downto 0), b_A20);
    (b(12), b(8), b(4)) <= b_A20;

```

```

A21: arb4 port map(b(16), g4(6 downto 4), b_A21);
(b(28), b(24), b(20)) <= b_A21;

A10: arb4 port map(b(0), req(2 downto 0), b_A10);
(b(3),b(2),b(1)) <= b_A10;

A11: arb4 port map(b(4), req(6 downto 4), b_A11);
(b(7),b(6),b(5)) <= b_A11;

A12: arb4 port map(b(8), req(10 downto 8), b_A12);
(b(11),b(10),b(9)) <= b_A12;

A13: arb4 port map(b(12), req(14 downto 12), b_A13);
(b(15),b(14),b(13)) <= b_A13;

A14: arb4 port map(b(16), req(18 downto 16), b_A14);
(b(19),b(18),b(17)) <= b_A14;

A15: arb4 port map(b(20), req(22 downto 20), b_A15);
(b(23),b(22),b(21)) <= b_A15;

A16: arb4 port map(b(24), req(26 downto 24), b_A16);
(b(27),b(26),b(25)) <= b_A16;

A17: arb4 port map(b(28), req(30 downto 28), b_A17);
(b(31),b(30),b(29)) <= b_A17;

gnt <= not b and req;
end impl;

```

12.5 One (possibly the best) option for implementing this solution is to precompute the value $3a$ and modify the booth recoders to look at two bits at a time and select between $\{0, a, 2a, 3a\}$. We also need to remove all sign extensions internal to the multiplier. The other option is to append 0 to the MSB of both a and b , converting the $n \times m$ multiplication to $(n + 1) \times (m + 1)$.

12.6 The two tables are below. We can quickly check that the first table is correct because the sum of each column gives the correct weight from each bit position. The VHDL would closely follow, using a selection multiplexer, inverter, and precomputed $3 \times$ sum.

bit	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	b_{-1}
weight	-256	128	64	32	16	8	4	2	1	N/A
d_2	-256	128	64	64						
d_1				-32	16	8	8			
d_0							-4	2	1	1

b_{3i+2}	b_{2i+1}	b_{3i}	b_{3i-1}	d_i
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	2
0	1	0	0	2
0	1	0	1	3
0	1	1	0	3
0	1	1	1	4
1	0	0	0	-4
1	0	0	1	-3
1	0	1	0	-3
1	0	1	1	-2
1	1	0	0	-2
1	1	0	1	-1
1	1	1	0	-1
1	1	1	1	0

12.11 The tables are shown below:

lg weight	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
pps	8	9	7	8	6	7	5	6	4	5	3	4	2	3	1	2
stage 1	6	5	6	5	4	5	4	3	4	3	2	3	2	1		
stage 2	4	4	4	3	4	3	3	2	3	2	2	1				
stage 3	3	3	3	2	3	2	2	2	1							
stage 4	2	2	2	2	1											

lg weight	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
pps	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
stage 1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
stage 2	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
stage 3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
stage 4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

12.12 The first table below shows an updated version of Table 12.3, accounting for the new 7-input counter. The next two tables show the number of remaining terms to add at each bit-position. We were unable to save a stage of logic using this scheme, though this is not always the case.

in	out		
	i	$2i$	$3i$
1	1	0	0
2	1	1	0
3	1	1	0
4	2	1	0
5	2	2	0
6	2	2	0
7	1	1	1
8	2	1	1
9	2	2	1

lg weight	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
pps	8	9	7	8	6	7	5	6	4	5	3	4	2	3	1	2
stage 1	5	4	2	5	3	3	4	3	4	3	2	3	2	1		
stage 2	3	3	3	3	2	2	3	2	3	2	2	1				
stage 3	2	2	2	2	2	2	2	2	1							
stage 4																

lg weight	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
pps	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
stage 1	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
stage 2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4
stage 3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3
stage 4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

12.15 The tables are shown below. We must sign-extend the partial products.

in	out	
	i	$2i$
1	1	0
2	1	1
3	1	1
4	2	1
5	2	2
6	2	2
7	1	1
8	2	1
9	2	2
10	2	2
11	2	2
12	3	3

[illegible]

Chapter 13

Arithmetic Examples

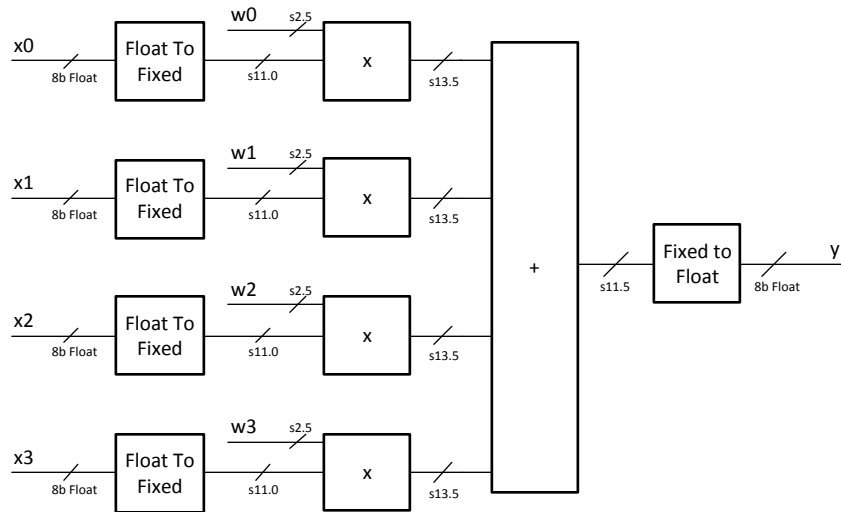
13.1 We can Booth-encode (Radix-4) and use the Wallace tree shown in the solution to Exercise 12.11. Slightly more interesting, however, is the negation on the input to the 2nd multiplier. To do this, we invert the input that is not the input to the Booth-recoders and add an additional carry in of 1 to the weight 0 term. This will not add any stages to the Wallace tree.

13.4 One of the simplest ways of fixing this bug is to add the following code to the VHDL:

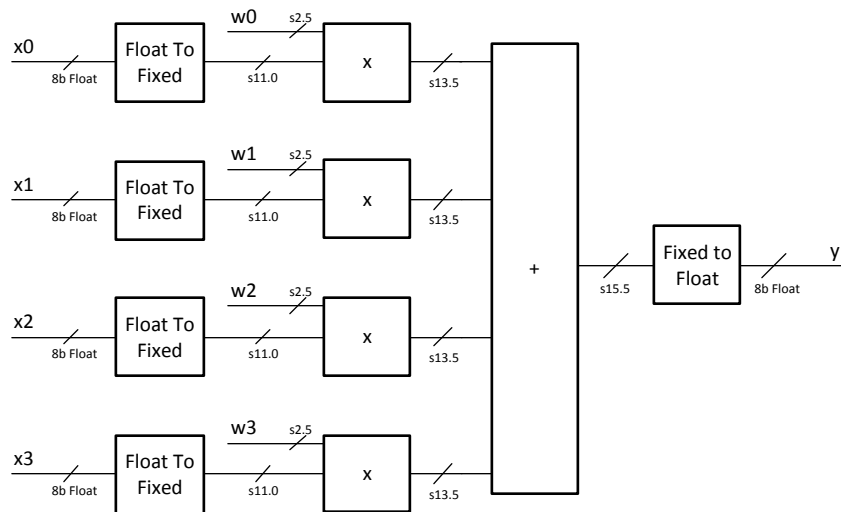
```
bugFix <= '1' when fixed = 12x"800" else '0';
float <= sign & 7x"7f" when (new_exp(3) or bugFix) else
    (sign & new_exp(2 downto 0) & mant(3 downto 0));
```

This will add minimum delay to the system, since the comparison is done in parallel to rest of the conversion process.

13.7 Shown below is our solution. We have increased the width of the multiplier output to accommodate for s2.5 weights. We did not widen the adder output since the sum of the weights had a magnitude less than one. This ensures that adder output can not have a magnitude greater than that of the greatest input.

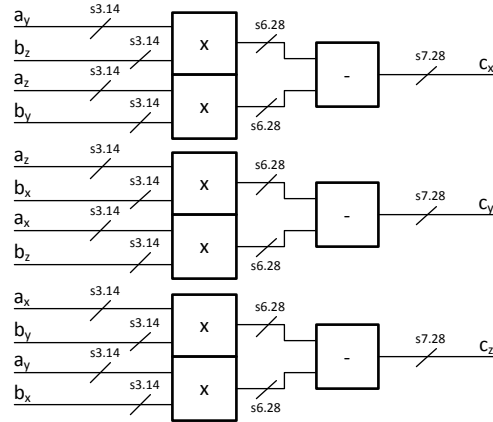


13.8 See the image below, which has a larger output of the adder. To detect overflow from the addition, we check that the upper 5 bits (sign, 4 MSB) are equivalent. If they are not, an overflow (relative to a s11.0 number) has occurred and the output is saturated.



13.11 Our cross product block diagram is shown below. We can factor the mul-

tiple and subtract module, and instantiate 3 copies of it. For extra speed, the full multiplies can be replaced with the ones used in Exercise 13.1.



13.12 The code is shown below. We find the values $x - 1$, $(x - 1)^2$, and $(x - 1)^3$ using VHDL's built-in multiplication. We then shift, sign extend, and manually line up the decimal points before the final add. We only include 20 bits in the final adder (instead of 27) because we discard the lower bits of $(x - 1)^3$ which cannot affect the round. The maximum error is 1.6% at $x = 2$.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity sqrtApprox is
  port (x: in std_logic_vector(8 downto 0);    -- 1.8
        y: out std_logic_vector(8 downto 0) ); -- 1.8
end sqrtApprox;

architecture impl of sqrtApprox is
  signal x_m1: signed(8 downto 0);
  signal x_m1_2: signed(17 downto 0);
  signal x_m1_3: signed(26 downto 0);
  signal y_t: signed(19 downto 0);
begin
  x_m1 <= signed(not x(8) & x(7 downto 0)); -- s.8 = x-1
  x_m1_2 <= x_m1 * x_m1; -- ss.16 = (x-1)*(x-1)
  x_m1_3 <= x_m1_2 * x_m1; -- sss.24 = (x-1)*(x-1)*(x-1)

```

```

y <= std_logic_vector(y_t(19 downto 11)) + std_logic(y_t(10)); -- round

y_t <= ( '1' & 19d"0" ) +
      ( x_m1(8) & x_m1 & 10d"0" ) -
      ( (2 downto 0 => x_m1_2(16)) & x_m1_2(16 downto 0) ) +
      ( (3 downto 0 => x_m1_3(24)) & x_m1_3(24 downto 9) );
end impl;

```

13.14 The simple converter, using VHDL addition and multiplication, is shown below.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bcd2bin is
  port(d: in std_logic_vector(15 downto 0);
        b: out std_logic_vector(13 downto 0) );
end bcd2bin;

architecture impl of bcd2bin is
  signal tmp: std_logic_vector(17 downto 0);
begin
  b <= tmp(13 downto 0);
  tmp <= d(15 downto 12) * 14d"1000" +
        d(11 downto 8) * 14d"100" +
        d(7 downto 4) * 14d"10" + d(3 downto 0);
end impl;

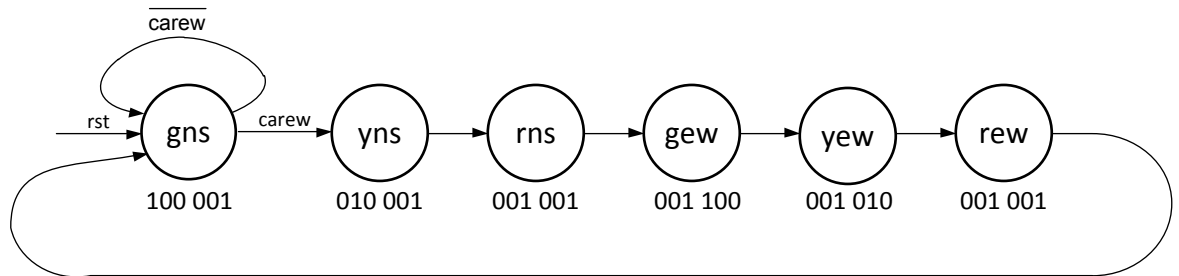
```


Chapter 14

Solutions: Sequential Logic

14.1 Leaving the input low for a minimum of 3 clock edges will put this FSM into state 00.

14.3 The new state diagram is shown below:



We have added new rns and rew states to set the lights to red in both directions. The state table is shown below:

state	next state		out
	carew=0	carew=1	
gns	gns	yns	100 001
yns	rns	rns	010 001
rns	gew	gew	001 001
gew	yew	yew	001 100
yew	rew	rew	001 010
rew	gns	gns	001 001

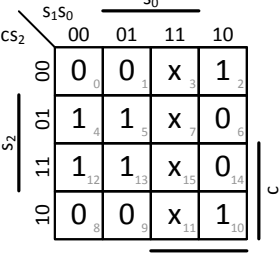
14.4 One possible binary state assignment is shown in the chart below.

state	encoding
gns	000
yns	001
rns	010
gew	100
yew	101
rew	110

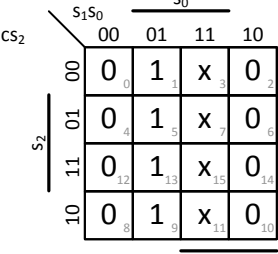
Next, we write the next state logic table:

state	carew	next state
000	0	000
000	1	001
001	0	010
001	1	010
010	0	100
010	1	100
100	0	101
100	1	101
101	0	110
101	1	110
110	0	000
110	1	000

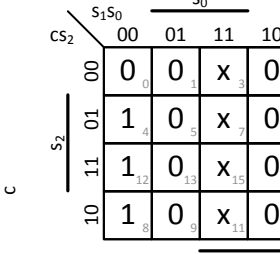
And find the next state logic:



$$ns2 = (s2 \wedge s1) \vee (s2 \wedge s1)$$



$$ns1 = s0$$



$$ns0 = (\bar{s1} \wedge \bar{s0} \wedge car) \vee (s2 \wedge s1 \wedge \bar{s0})$$

Finally, we compute the outputs:

$$\begin{aligned}
g_{\text{ns}} &= \overline{s_2} \wedge \overline{s_1} \wedge \overline{s_0} \\
y_{\text{ns}} &= \overline{s_2} \wedge s_0 \\
r_{\text{ns}} &= s_2 \vee s_1 \\
g_{\text{ew}} &= s_2 \wedge \overline{s_1} \wedge \overline{s_0} \\
y_{\text{ew}} &= s_2 \wedge s_0 \\
r_{\text{ew}} &= \overline{s_2} \vee s_1
\end{aligned}$$

14.5 The Verilog has very few changes compared to that supplied in the text:

```

library ieee;
use ieee.std_logic_1164.all;

package Traffic_Light_Codes is
  -- State assignment
  constant SWIDTH: integer := 3;
  subtype state_type is std_logic_vector(SWIDTH-1 downto 0);
  constant GNS: state_type := "000";
  constant YNS: state_type := "001";
  constant RNS: state_type := "010";
  constant GEW: state_type := "100";
  constant YEW: state_type := "101";
  constant REW: state_type := "110";
  -- define output codes
  subtype lights_type is std_logic_vector(5 downto 0);
  constant GNSL: lights_type := "100001";
  constant YNSL: lights_type := "010001";
  constant GEWL: lights_type := "001100";
  constant YEWL: lights_type := "001010";
  constant REDL: lights_type := "001001";
end package;

library ieee;
use ieee.std_logic_1164.all;
use work.Traffic_Light_Codes.all;
use work.ff.all;

entity Traffic_Light is
  port( clk, rst, carew: in std_logic;
        lights: out lights_type );
end Traffic_Light;

architecture impl of Traffic_Light is
  signal state, next_state, next1: state_type; -- current, next state, next state w/o reset

```

```

begin
  -- instantiate state register
  STATE_REG: vDFF generic map(SWIDTH) port map(clk, next_state, state) ;

  -- next state and output equations - this is combinational logic
  process (all) begin
    case state is
      when GNS =>
        if carew then next1 <= YNS;
        else next1 <= GNS; end if;
        lights <= GNSL;
      when YNS => next1 <= RNS; lights <= YNSL;
      when RNS => next1 <= GEW; lights <= REDL;
      when GEW => next1 <= YEW; lights <= GEWL;
      when YEW => next1 <= REW; lights <= YEWL;
      when REW => next1 <= GNS; lights <= REDL;
      when others =>
        next1 <= std_logic_vector'(SWIDTH-1 downto 0 => '-');
        lights <= "-----";
      end case;
    end process;
    -- add reset
    next_state <= GNS when rst else next1;
  end impl;

```

14.12 The new state diagram can be constructed by inserting a state between 3 and 4. This new state will be renamed 4, while 4 becomes 5, and 5 becomes 6. The new state transitions every cycle.

14.13 The state table is shown below. We omit a column indicating that the *rst* input cases a transition to state R.

state	next state		out
	a=0	a=1	
R	0	1	0
1	2	2	1
2	3	3	0
3	4	4	0
4	5	1	0
5	M	1	0
M	2	L	1
L	2	2	0

We also include the state table from the modified pulse filler of the previous exercise:

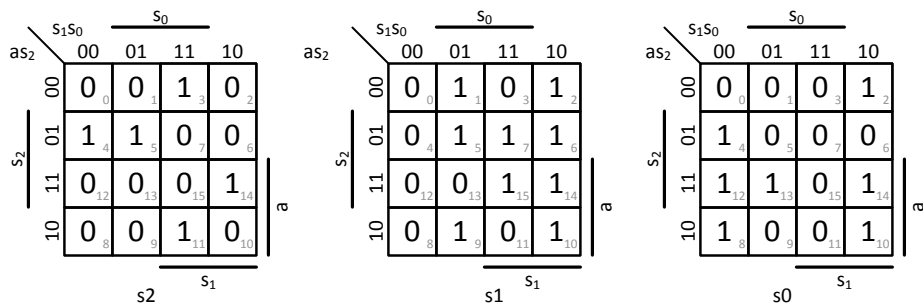
state	next state		out
	a=0	a=1	
R	0	1	0
1	2	2	1
2	3	3	0
3	4	4	0
4	5	5	0
5	6	1	0
6	M	1	0
M	2	L	1
L	2	2	0

14.14 We wrote a program to permute all possible state assignments and got the following:

state	encoding
R	000
1	001
2	011
3	111
4	101
5	100
M	110
L	010

There are only two non-single bit transitions.

14.15 The K-maps are shown below:



The state logic is:

$$\begin{aligned}
s_2 &= (\bar{a} \wedge s_2 \bar{s}_1) \vee (\bar{s}_2 \wedge s_1 \wedge s_0) \vee (a \wedge s_2 \wedge s_1 \wedge \bar{s}_0) \\
s_1 &= (s_2 \wedge s_1) \vee (s_1 \wedge \bar{s}_0) \vee (\bar{a} \wedge \bar{s}_1 \wedge s_0) \vee (\bar{s}_2 \wedge \bar{s}_1 \wedge s_0) \\
s_0 &= (a \wedge \bar{s}_0) \vee (s_2 \wedge \bar{s}_1 \wedge \bar{s}_0) \vee (a \wedge s_2 \wedge \bar{s}_1) \vee (\bar{s}_2 \wedge s_1 \wedge \bar{s}_0)
\end{aligned}$$

14.20 In our state table, shown below, we assume that only one (or zero) input goes high each cycle. If money is inserted while vending, we go to the appropriate state.

state	next state			{vend,change}
	n=0,d=0	n=1,d=0	n=0,d=1	
in00	in00	in05	in10	00
in05	in05	in10	in15	00
in10	in10	in15	in20	00
in15	in15	in20	in25	00
in20	in20	in25	in30	00
in25	in25	in30	in35	00
in30	in30	in35	in40	00
in35	in35	in40	in45	00
in40	in00	in05	in10	10
in45	in00	in05	in10	11

14.25 See the state table below. We use the state names up25k0 to up25k4 when counting the smooth signal. When we list up25kX, we are indicating any of the 5 states prefixed by “up25k.”

state	alt10k	alt25k	smooth	next state	{noelect, belt}
gnd	0	x	x	gnd	11
gnd	1	x	x	up10k	11
up10k	0	0	x	up10k	01
up10k	1	0	x	gnd	01
up10k	0	1	x	up25k	01
up25k0	0	0	1	up25k1	01
up25k1	0	0	1	up25k2	01
up25k2	0	0	1	up25k3	01
up25k3	0	0	1	up25k4	01
up25k4	0	0	1	up25kS	01
up25kS	0	0	1	up25kS	00
up25kX	0	1	x	up10k	
up25kX	0	0	0	up25k0	

Chapter 15

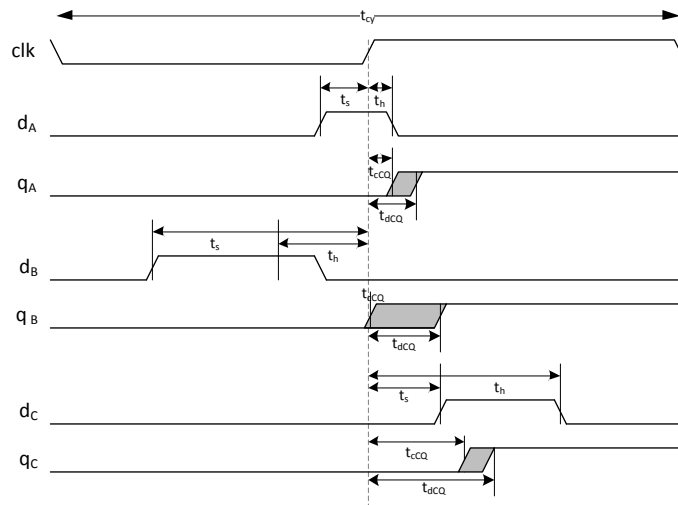
Solutions: Timing Constraints

- 15.1**
- $t_{cax} = t_{dax} = 20\text{ps}$
 - $t_{cbx} = 3030, t_{dbx} = 40\text{ps}$
 - $t_{ccx} = 3030, t_{dcx} = 40\text{ps}$
 - $t_{cdx} = 3030, t_{ddx} = 50\text{ps}$

15.2 $t_{cax} = 35\text{ps}, t_{dax} = 60\text{ps}$

15.3 $t_{cbx} = 35\text{ps}, t_{dbx} = 40\text{ps}$

15.7 The solution for the next 3 problems is shown below:



15.8 See above.

15.9 See above.

15.10 A 2GHz cycle time gives $t_{cy} = 500\text{ps}$. Using Equations 15.3 and 15.4 gives:

$$\begin{aligned}
 t_{cy} &\geq t_{dCQ} + t_{d\text{Max}} + t_s \\
 500 &\geq 20 + 400 + 20 \\
 500 &\geq 440 \\
 t_h &\leq t_{cCQ} + t_{c\text{Min}} \\
 10 &\leq 10 + 10 \\
 10 &\leq 20
 \end{aligned}$$

There are no errors.

15.11 Again using Equations 15.3 and 15.4:

$$\begin{aligned}
 t_{cy} &\geq t_{dCQ} + t_{d\text{Max}} + t_s \\
 500 &\geq 30 + 400 + 100 \\
 500 &\geq 530 \\
 t_h &\leq t_{cCQ} + t_{c\text{Min}} \\
 -20 &\leq 2 + 10 \\
 -20 &\leq 12
 \end{aligned}$$

There is a setup violation. We must increase the cycle time to 530ps for correct operation.

15.13 Using the setup and hold equations:

$$\begin{aligned}
 t_{cy} &\geq t_{dCQ} + t_{d\text{Max}} + t_s \\
 1000 &\geq 20 + t_{d\text{Max}} + 20 \\
 t_{d\text{Max}} &\leq 960\text{ps} \\
 t_h &\leq t_{cCQ} + t_{c\text{Min}} \\
 10 &\leq 10 + t_{c\text{Min}} \\
 t_{c\text{Min}} &\geq 0\text{ps}
 \end{aligned}$$

15.14 Using the setup and hold equations:

$$\begin{aligned}
 t_{cy} &\geq t_{dCQ} + t_{dMax} + t_s \\
 1000 &\geq 30 + t_{dMax} + 100 \\
 t_{dMax} &\leq 870\text{ps} \\
 t_h &\leq t_{cCQ} + t_{cMin} \\
 -20 &\leq 2 + t_{cMin} \\
 t_{cMin} &\geq -18\text{ps}
 \end{aligned}$$

15.16 We want the flip-flop to work even when there is no combinational logic, thus:

$$\begin{aligned}
 t_h &\leq t_{cCQ} + 0 \\
 t_h - t_{cCQ} &\leq 0
 \end{aligned}$$

15.18 To test if the violation is a setup-time problem, simply increase the cycle time. If running at a slower frequency fixes the problem (to a first order) it is a setup violation. Hold time violations cannot be detected by varying the clock. (Saying “if it is not setup then it is hold” is not a valid test strategy.) Because hold violations occur when the contamination delay of a circuit is too fast, any method of slowing logic such as increased temperature or lower voltage would make the circuit work. See Section 20.2.6 for more information on this type of characterization.

15.19 The new $t_s = t_h = 10\text{ps}$ since the outer clock effectively arrives 40ps earlier than the inner clock. There is a 40ps delay on the output, which gives $t_{dcq} = t_{ccq} = 120\text{ps}$.

15.22 We make the table below for all logic paths. In it, we list the time that a clock can come early to Y so a signal from X does not cause a setup violation. We also provide the how late a clock can come to Y so a signal from X does not a hold violation.

From	To	Early(ps)	Late(ps)
X	Y	1860	100
X	Z	1910	30
Y	X	-	-
Y	Z	1910	30
Z	X	1560	10
Z	Y	-	-

Note that in the table the clock to X from Z cannot actually be 1910ps early since that is equivalent to a 1860ps early clock to Z from X (a violation). We've updated the table below:

From	To	Early(ps)	Late(ps)
X	Y	1860	100
X	Z	10	30
Y	X	100	1860
Y	Z	1910	30
Z	X	30	10
Z	Y	30	1910

Chapter 16

Solutions: Data Path Sequential Logic

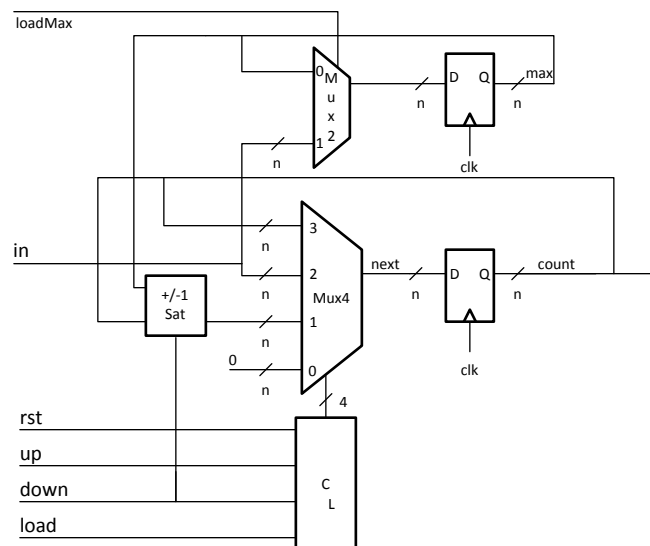
16.1 The sequence of states is shown in the table below. It counts through 8 different states only changing a single bit at a time.

State	Next
0000	0001
0001	0011
0011	0111
0111	1111
1111	1110
1110	1100
1100	1000
1000	0000

16.2 The sequence of 15 states (all except 0000) is shown in the table below. This counts through the 15 numbers in a pseudo-random order.

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

16.4 The solution is shown below. We have added a register for storing the maximum value, the logic to load it, and modified the counter. The new counter (not shown) will not increment when the *count* equals *max* and will not decrement when the *count* equals 0.



16.6 The Verilog is shown below:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```

use work.ff.all;
use work.ch8.all;

entity UDL_Count3Rd is
  generic( n: integer := 4 );
  port(clk, rst, up, down, load: in std_logic;
        input: in std_logic_vector(n-1 downto 0);
        rd: in std_logic_vector(1 downto 0);
        r0, r1, r2, r3: out std_logic_vector(n-1 downto 0) );
end UDL_Count3Rd;

architecture impl of UDL_Count3Rd is
  signal outpm1, nxt: std_logic_vector(n-1 downto 0);
  signal n0, n1, n2, n3, src: std_logic_vector(n-1 downto 0);
begin
  COUNT0: vDFF generic map(n) port map(clk, n0, r0);
  COUNT1: vDFF generic map(n) port map(clk, n1, r1);
  COUNT2: vDFF generic map(n) port map(clk, n2, r2);
  COUNT3: vDFF generic map(n) port map(clk, n3, r3);

  process(all) begin
    case rd is
      when "00" => (src, n3, n2, n1, n0) <= std_logic_vector'(r0 & r3 & r2 & r1 & nxt);
      when "01" => (src, n3, n2, n1, n0) <= std_logic_vector'(r1 & r3 & r2 & nxt & r0);
      when "10" => (src, n3, n2, n1, n0) <= std_logic_vector'(r2 & r3 & nxt & r1 & r0);
      when "11" => (src, n3, n2, n1, n0) <= std_logic_vector'(r3 & nxt & r2 & r1 & r0);
      when others => (src, n3, n2, n1, n0) <= std_logic_vector'(5*n-1 downto 0 => '-');
    end case;
  end process;

  outpm1 <= src + ((n-1 downto 1 => down) & '1');

  MUX: Mux4 generic map(n)
    port map(src, input, outpm1, (others => '0'),
      ((not rst and not up and not down and not load) &
      (not rst and load) &
      (not rst and (up or down)) &
      rst),
      nxt);
end impl;

```

16.7 The Verilog is shown below:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.ff.all;
use work.ch8.all;

```

```

entity UDL_Count3Rs is
  generic( n: integer := 4 );
  port( clk, rst, up, down, load: in std_logic;
        input: in std_logic_vector(n-1 downto 0);
        rd, rs: in std_logic_vector(1 downto 0);
        r0, r1, r2, r3: out std_logic_vector(n-1 downto 0) );
end UDL_Count3Rs;

architecture impl of UDL_Count3Rs is
  signal outpm1, nxt: std_logic_vector(n-1 downto 0);
  signal n0, n1, n2, n3, src: std_logic_vector(n-1 downto 0);
begin
  COUNT0: vDFF generic map(n) port map(clk, n0, r0);
  COUNT1: vDFF generic map(n) port map(clk, n1, r1);
  COUNT2: vDFF generic map(n) port map(clk, n2, r2);
  COUNT3: vDFF generic map(n) port map(clk, n3, r3);

  process(all) begin
    case rs is
      when "00" => src <= r0;
      when "01" => src <= r1;
      when "10" => src <= r2;
      when "11" => src <= r3;
      when others => src <= (others => '-');
    end case;
  end process;

  process(all) begin
    case rd is
      when "00" => (n3, n2, n1, n0) <= std_logic_vector'(r3 & r2 & r1 & nxt);
      when "01" => (n3, n2, n1, n0) <= std_logic_vector'(r3 & r2 & nxt & r0);
      when "10" => (n3, n2, n1, n0) <= std_logic_vector'(r3 & nxt & r1 & r0);
      when "11" => (n3, n2, n1, n0) <= std_logic_vector'(nxt & r2 & r1 & r0);
      when others => (n3, n2, n1, n0) <= std_logic_vector'(4*n-1 downto 0 => '-');
    end case;
  end process;

  outpm1 <= src + ((n-1 downto 1 => down) & '1');

  MUX: Mux4 generic map(n)
    port map(src, input, outpm1, (others => '0'),
      ((not rst and not up and not down and not load) &
      (not rst and load) &
      (not rst and (up or down)) &
      rst),

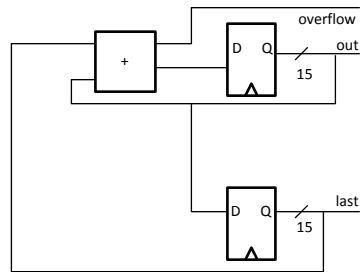
```

```

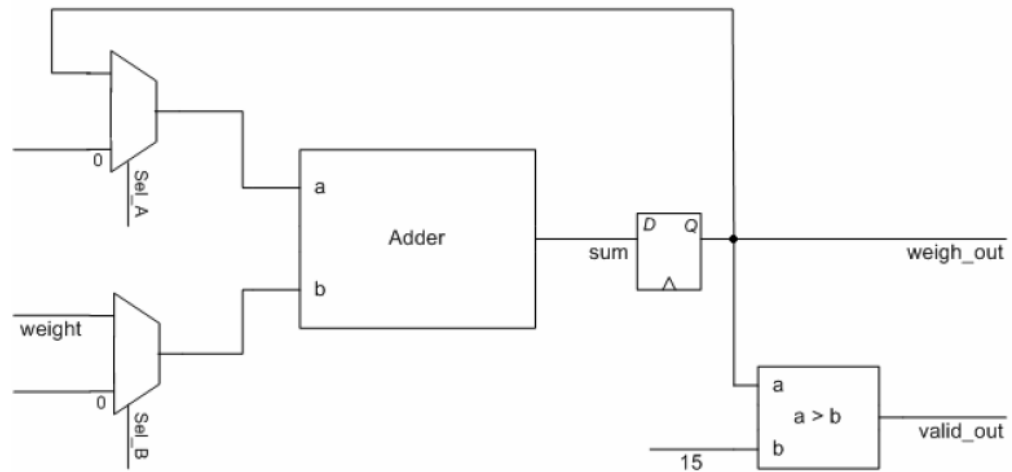
                                nxt);
end impl;

```

16.8 See below for a possible solution. We use two registers: one for the current number and one for the previous number. When we reset (not shown), we set the **out** register to 0 and the **last** register to 1. That gives the correct sequence of 0, 1, 1, 2, 3, ... on the output.



16.14 The datapath is shown below:



16.15 The two control signals are:

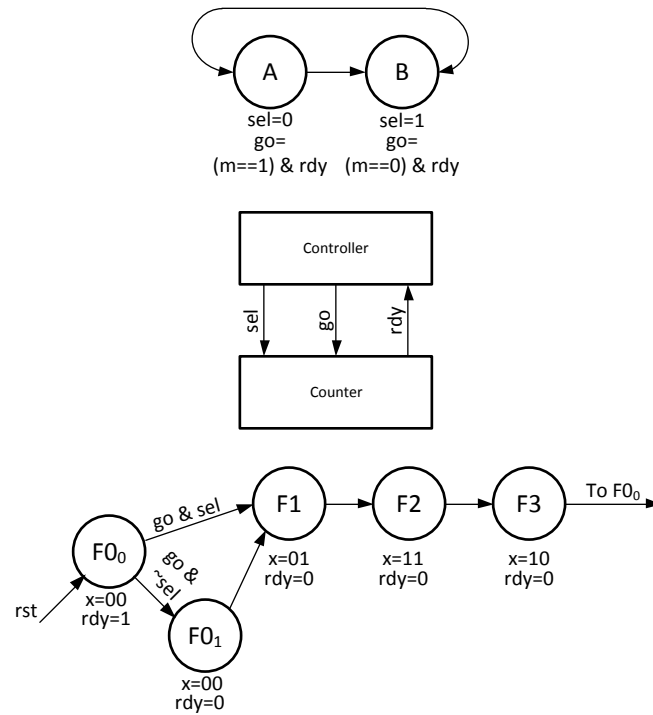
$$\text{Sel_A} = \text{rst} \vee \text{valid_out}$$

$$\text{Sel_B} = \text{rst} \vee \overline{\text{valid_out}}$$

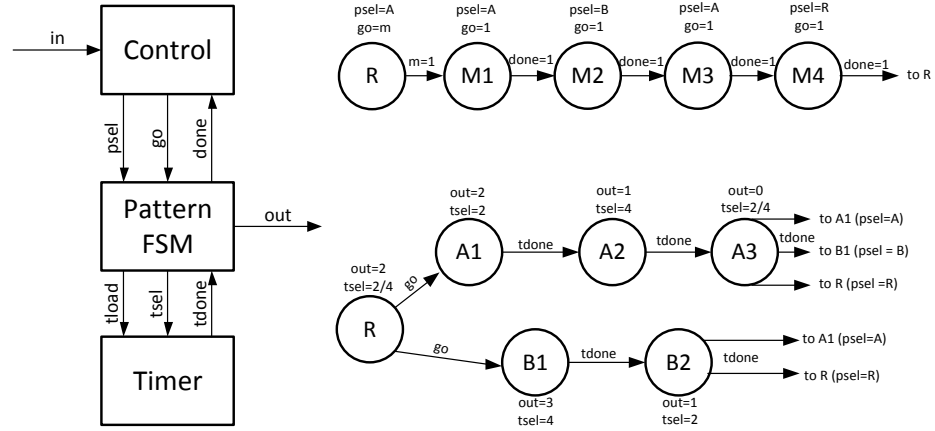
Chapter 17

Solutions: Factoring Finite State Machines

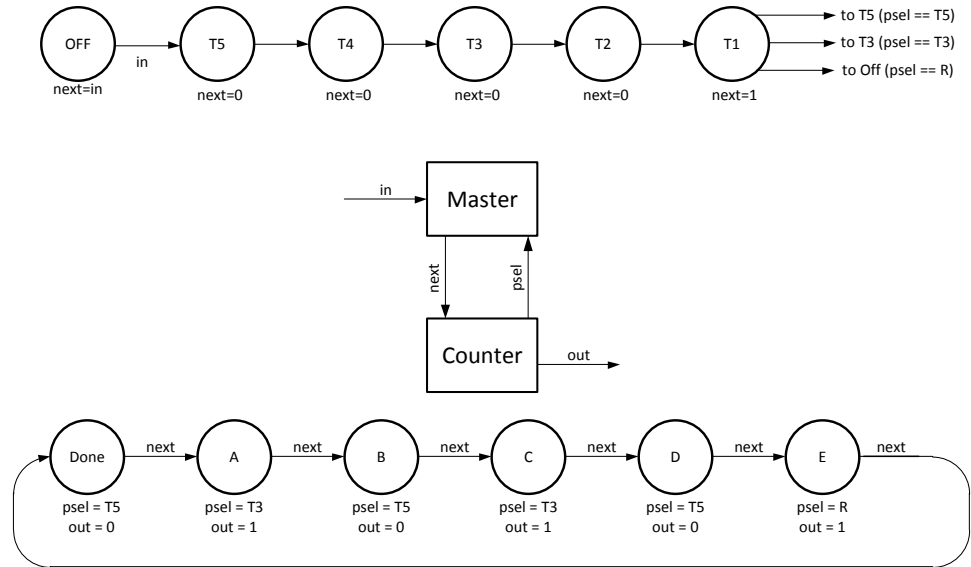
17.1 The diagram below shows our solution. We have factored the states B, C, D, F, G, H, and I into a separate counter state machine (bottom of the image). When the *go* signal is asserted, it walks through the Gray-code count (possibly through state FO_1). The controller toggles between states A and B and goes to the next state when the counter signals ready and the input is the correct value.



17.5 We have factored the FSM (see below) twice. First, we add a timer (either 2 or 4 cycles) that counts down the time spent at any one output state. Next, we factor the sequence of output states into two distinct patterns: 2-1-0, labeled A1-A3 and 3-1, labeled B1-B2. The master controller (top-right) selects the pattern and, when done, moves to the next state.



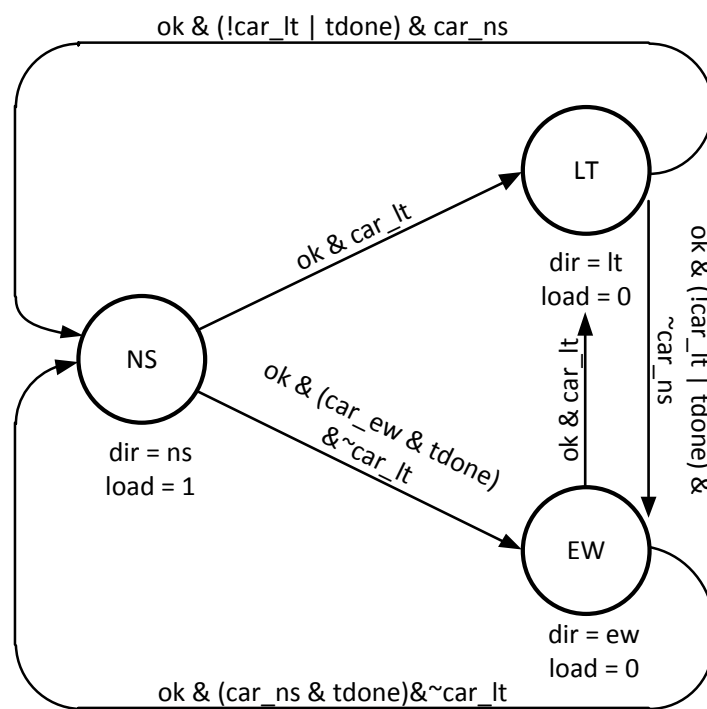
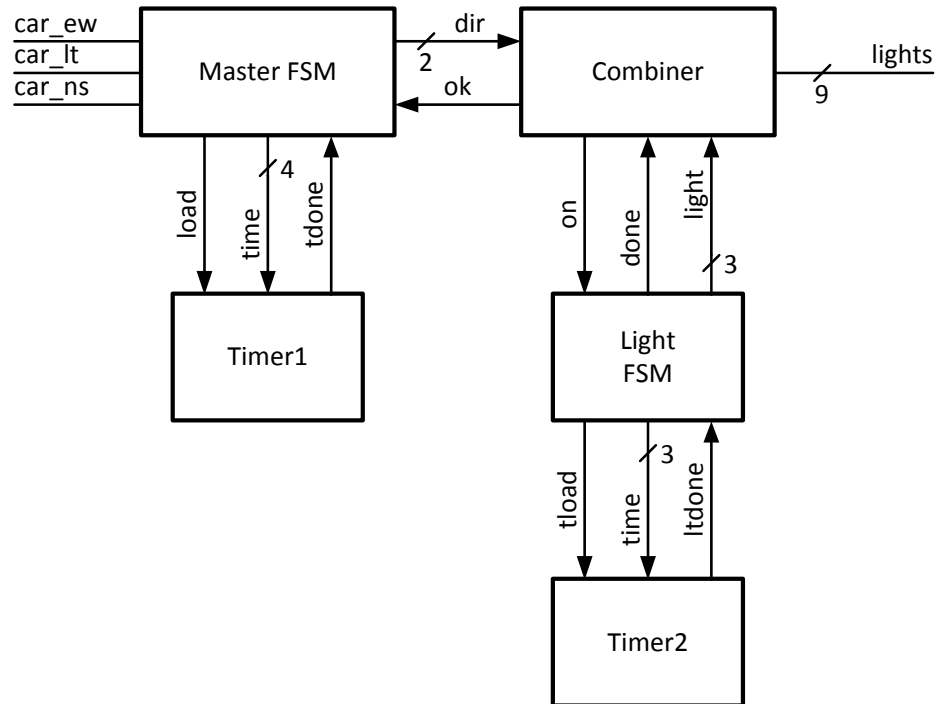
17.9 The inverted controller is shown below. We have an explicit timer at the top level that communications with the counter using *next* and *psel* signals. When the timer reaches 0, it asserts *next* and goes to the state indicated by *psel*. The output is driven directly by the counter.



17.14 To implement this functionality, we must modify both the master, timer, and the count. In the timer, the *tssel* signal must now be widened to 2 bits

to select between a count of 5, 15, or 4 cycles. The counter must output the LSB of the count itself (odd or even). Finally, the Master FSM must set *tssel* correctly to take into account the count.

17.17 The benefits of factoring is seen in this problem, as we only need to change 1 of the 5 state machines (shown below). We add the *car_ns* signal to the system and modify the controller to be fair (while still giving *car_lt* priority).



Chapter 18

Solutions: Microcode

18.1 The new microcode table is shown below. We have added the additional input signal, *car_ns*, and another address bit. We also modified the state transitions from GEW to go to YEW only when *car_ns* is asserted.

Address	State	<i>car_ew, car_ns</i>	Next State	Output	Data
00000	GNS (000)	00	GNS (000)	100001	000100001
00001	GNS (000)	01	GNS (000)	100001	000100001
00010	GNS (000)	10	YNS (001)	100001	001100001
00011	GNS (000)	11	YNS (001)	100001	001100001
00100	YNS (001)	00	GEW (010)	010001	010010001
00101	YNS (001)	01	GEW (010)	010001	010010001
00110	YNS (001)	10	GEW (010)	010001	010010001
00111	YNS (001)	11	GEW (010)	010001	010010001
01000	GEW (010)	00	GEW (010)	001100	010001100
01001	GEW (010)	01	YEW (011)	001100	011001100
01010	GEW (010)	10	GEW (010)	001100	010001100
01011	GEW (010)	11	YEW (011)	001100	011001100
01100	YEW (011)	00	GNS (000)	001010	000001010
01101	YEW (011)	01	GNS (000)	001010	000001010
01110	YEW (011)	10	GNS (000)	001010	000001010
01111	YEW (011)	11	GNS (000)	001010	000001010

18.3 We must change a total of 2 bits in our storage, see the new table below:

Address	State	<i>car_ew, car_ns</i>	Next State	Output	Data
00000	GNS (000)	00	GNS (000)	100001	000100001
00001	GNS (000)	01	GNS (000)	100001	000100001
00010	GNS (000)	10	YNS (001)	100001	001100001
00011	GNS (000)	11	GNS (001)	100001	000100001
00100	YNS (001)	00	GEW (010)	010001	010010001
00101	YNS (001)	01	GEW (010)	010001	010010001
00110	YNS (001)	10	GEW (010)	010001	010010001
00111	YNS (001)	11	GEW (010)	010001	010010001
01000	GEW (010)	00	GEW (010)	001100	010001100
01001	GEW (010)	01	YEW (011)	001100	011001100
01010	GEW (010)	10	GEW (010)	001100	010001100
01011	GEW (010)	11	GEW (010)	001100	010001100
01100	YEW (011)	00	GNS (000)	001010	000001010
01101	YEW (011)	01	GNS (000)	001010	000001010
01110	YEW (011)	10	GNS (000)	001010	000001010
01111	YEW (011)	11	GNS (000)	001010	000001010

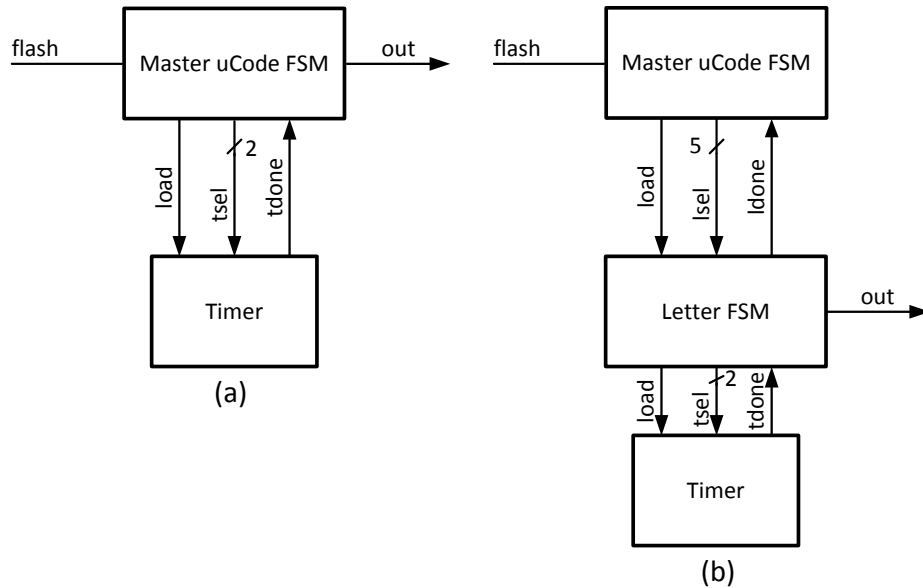
18.4 Below is our sequence, using the same inputs and outputs as described in the text. In the BASE state, we loop until an input is asserted. We then branch based on the input and perform one or more instructions, such as adding a value to the total or dispensing.

State	Br Inst	Target	SelVal	sub	selNext	Serv	Change	Next
BASE	inputs=0	BASE	0000	0	100	0	0	0
B2Q	quarter	Q1	0000	0	100	0	0	0
B2D	dime	D1	0000	0	100	0	0	0
B2N	nickel	N1	0000	0	100	0	0	0
B2Disp	dispense & enough	S1	0000	0	100	0	0	0
ToBase	Always	BASE	0000	0	100	0	0	0
Q1	Always	BASE	0001	0	010	0	0	1
D1	Always	BASE	0010	0	010	0	0	1
N1	Always	BASE	0100	0	010	0	0	1
S1	Never	X	1000	1	010	1	0	1
S1A	~done	S1A	0000	0	100	0	0	1
S2A	done	S2A	0000	0	100	0	0	1
S2B	zero	BASE	0000	0	100	0	0	0
C1	Never	X	0100	1	010	0	1	0
C1A	~done	C1A	0000	0	100	0	0	1
C2A	done	C2A	0000	0	100	0	0	1
C2B	zero	BASE	0000	0	100	0	0	1
C2C	Always	C1	0000	0	100	0	0	1

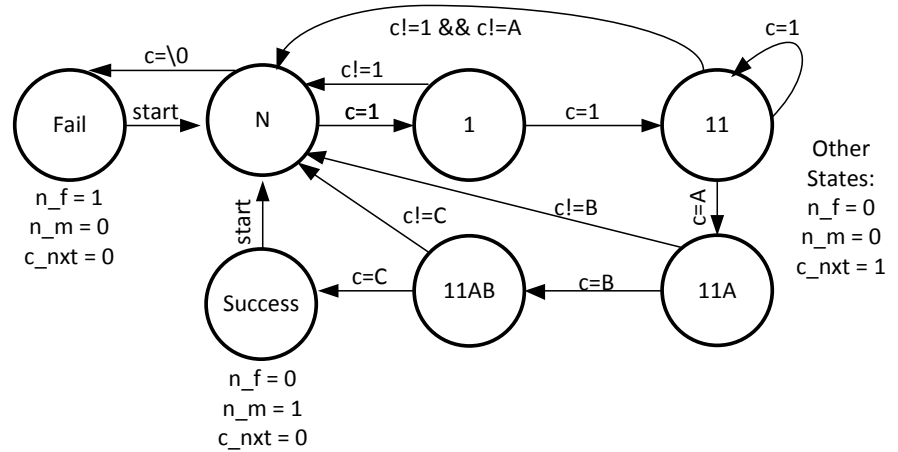
18.6 A portion of the micro-code table is shown below. It simply iterates through each state in the flash sequence if the input *flash* is high and reverts to state RST if it goes low.

State	<i>flash</i>	Next State	Output
RST	0	RST	0
RST	1	F1	0
X	0	RST	0
F1	1	F2	1
F2	1	F3	0

18.7 Two potential solutions are shown below. In (a), the microcode takes both *flash* and *tdone* as inputs and sets the output and loads the timer when appropriate. The code must both sequence each sub-component of a letter (dot, dash, space) and the letters themselves. Solution (b) factors out reusable letter FSM (potentially also microcoded) and the master FSM only needs to sequence the letters ‘SOS’.



18.11 The state diagram and simplified microcode are shown below. The microcode is fairly basic, except with respect to states S11p0 and S11p1. Here, we must check if the input character is either a ‘1’ or an ‘A’. We first check for an ‘A’ and if that is not a match, the FSM will not assert *c_nxt* and instead check against the value ‘1’.



State	Start	End	Match	Next	s_c	n_f	n_m	c_nxt
FAIL	0	X	X	FAIL	x	1	0	0
FAIL	1	X	X	N	x	0	0	1
X	0	1	X	FAIL	x	1	0	1
N	0	0	0	N	1	0	0	1
N	0	0	1	S1	1	0	0	1
S1	0	0	0	N	1	0	0	1
S1	0	0	1	S11p0	1	0	0	1
S11p0	0	0	0	S11p1	A	0	0	0
S11p0	0	0	1	S11A	A	0	0	1
S11p1	0	0	0	N	1	0	0	1
S11p1	0	0	1	S11p0	1	0	0	1
S11A	0	0	0	N	B	0	0	1
S11A	0	0	1	S11AB	B	0	0	1
S11AB	0	0	0	N	C	0	0	1
S11AC	0	0	1	SUCCESS	C	0	1	1
SUCCESS	0	X	X	SUCCESS	x	0	1	0
SUCCESS	1	X	X	N	x	0	0	1

18.17 The code used to write this program is shown below. It relies on a series of immediate loads, adds, and subtracts to compute each character. The code itself cannot be easily changed to spell out different strings. Another solution would be to write “HELLO WORLD” into RAM memory at initialization then load and output each character in turn. This would enable the code to be easily changed to spell out other phrases.

<pre> #48, 45, 4C, 4F, 20, #57, 4F, 52, 4C, 44 LDAI 0100 STA T2 LDAI 0010 SH T2 STA T1 #T1 = 0x20 LDA T2 #ACC = 0x04 SH T2 #ACC = 0x40 STA T2 #T2 = 0x40 LDAI 1000 OR T2 #ACC = 0x48 STA 01 #01 = 0x48 = H LDAI 0011 STA T0 #T0 = 3 LDA 01 SUB T0 STA 01 #01 = 0x45 = E LDAI 0111 ADD 01 STA 01 #01 = 0x4C = L STA 01 #01 = 0x4C = L LDAI 0011 #To next col </pre>	<pre> #From bottom of left ADD 01 STA 01 #01 = 0x4F = 0 STA T0 #T0 = 0x4F LDA T1 STA 01 #01 = 0x20 = ' ' LDAI 1000 ADD T0 #Acc = 57 STA 01 #01 = 0x57 LDA T0 STA 01 #01 = 0x4F LDAI 0011 ADD 01 STA 01 #01 = 0x52 LDAI 1110 STA T0 #T0 = 0xE LDA 01 SUB T0 #ACC = 0x44 STA T0 LDAI 1000 ADD T0 STA 01 LDA T0 STA 01 </pre>
--	--

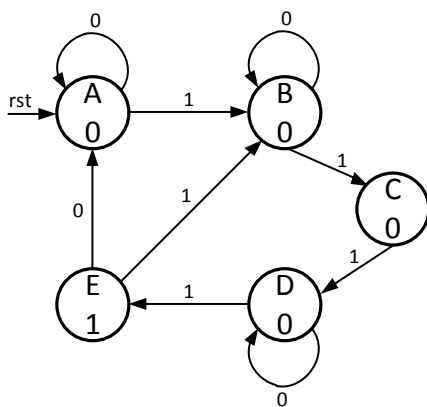
18.18 The output from our testbench is shown below.

PC: 0000, o1: 0000 i:01100100	# PC: 0017, o1: 004f i:01111010
# PC: 0001, o1: 0000 i:01111100	# PC: 0018, o1: 004f i:01011011
# PC: 0002, o1: 0000 i:01100010	# PC: 0019, o1: 004f i:01110011
# PC: 0003, o1: 0000 i:10111100	# PC: 001a, o1: 0020 i:01101000
# PC: 0004, o1: 0000 i:01111011	# PC: 001b, o1: 0020 i:10001010
# PC: 0005, o1: 0000 i:01011100	# PC: 001c, o1: 0020 i:01110011
# PC: 0006, o1: 0000 i:10111100	# PC: 001d, o1: 0057 i:01011010
# PC: 0007, o1: 0000 i:01111100	# PC: 001e, o1: 0057 i:01110011
# PC: 0008, o1: 0000 i:01101000	# PC: 001f, o1: 004f i:01100011
# PC: 0009, o1: 0000 i:11101100	# PC: 0020, o1: 004f i:10000011
# PC: 000a, o1: 0000 i:01110011	# PC: 0021, o1: 004f i:01110011
# PC: 000b, o1: 0048 i:01100011	# PC: 0022, o1: 0052 i:01101110
# PC: 000c, o1: 0048 i:01111010	# PC: 0023, o1: 0052 i:01111010
# PC: 000d, o1: 0048 i:01010011	# PC: 0024, o1: 0052 i:01010011
# PC: 000e, o1: 0048 i:10011010	# PC: 0025, o1: 0052 i:10011010
# PC: 000f, o1: 0048 i:01110011	# PC: 0026, o1: 0052 i:01111010
# PC: 0010, o1: 0045 i:01100111	# PC: 0027, o1: 0052 i:01101000
# PC: 0011, o1: 0045 i:10000011	# PC: 0028, o1: 0052 i:10001010
# PC: 0012, o1: 0045 i:01110011	# PC: 0029, o1: 0052 i:01110011
# PC: 0013, o1: 004c i:01110011	# PC: 002a, o1: 004c i:01011010
# PC: 0014, o1: 004c i:01100011	# PC: 002b, o1: 004c i:01110011
# PC: 0015, o1: 004c i:10000011	# PC: 002c, o1: 0044 i:xxxxxxxx
# PC: 0016, o1: 004c i:01110011	

Chapter 19

Solutions: Sequential Examples

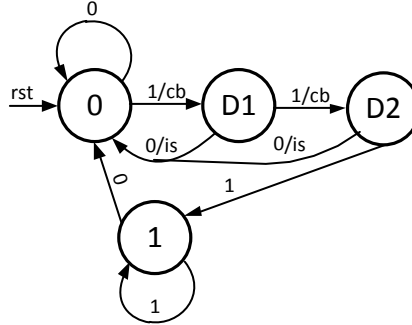
19.1 In the new state diagram, below, we have added another stage compared to that of the divide-by-3 counter.



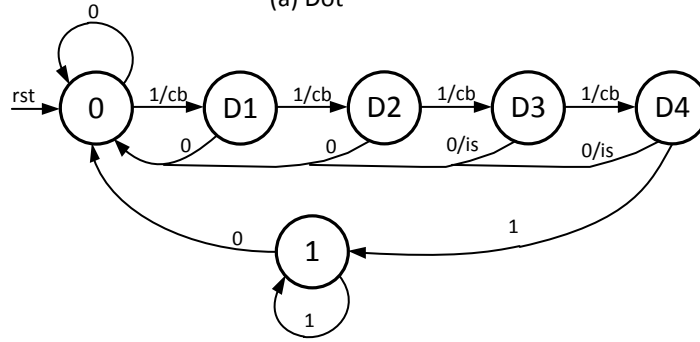
19.2 We can make a divide-by-9 counter by attaching the output of one divide-by-3 counter to the input of the next. With this structure, however, the divide-by-9 signal will be one cycle later than if we had built a single state machine.

19.4 The new dot (a) and dash (b) state machines are below. If there is a fourth consecutive 1 in the dash detector, we leave the *cb* signal asserted and move to state D4. There, if there is a 0 (or if there is a zero in D3), the FSM signals *is* and resets back to state 0. A fifth one goes into state

1 and does not assert *is*. The dot detector operates in a similar fashion.

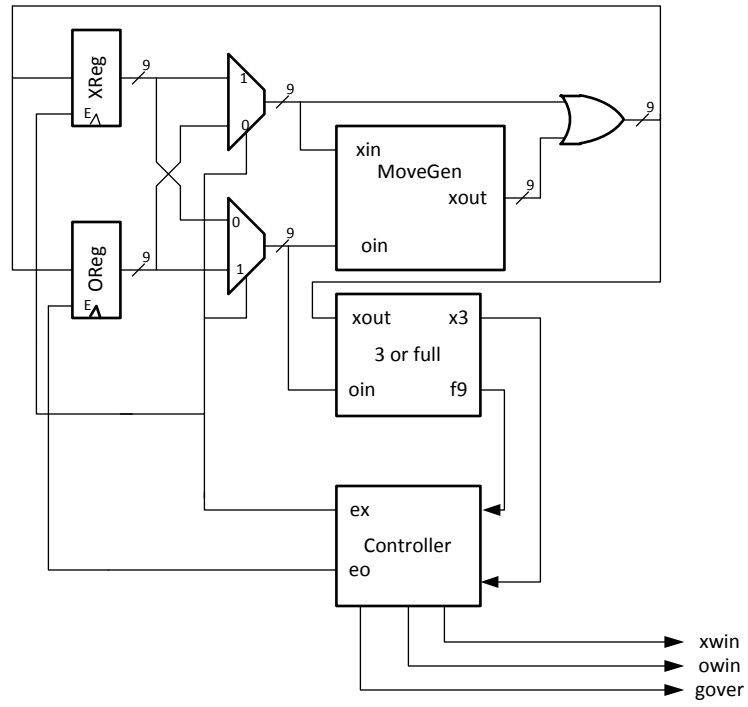


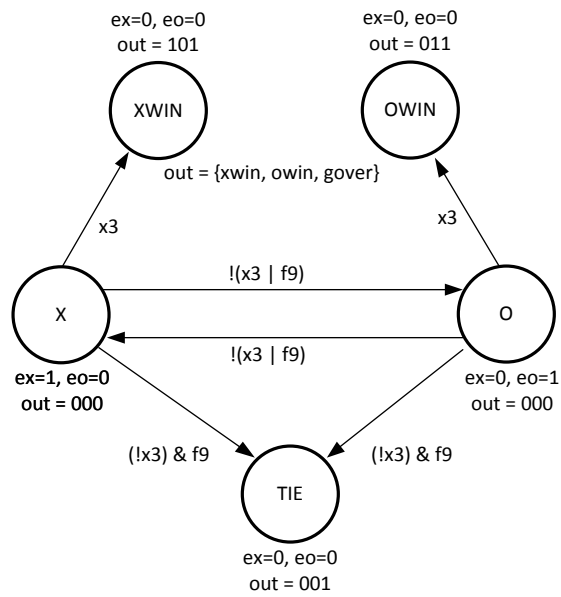
(a) Dot



(b) Dash

19.6 The top level diagram of the Tic-Tac-Toe machine is shown below. We have added a combinational module, 3-or-full, and sequential module, Controller. The 3-or-full asserts *x3* if the xout signal has 3 in a row. It sets *f9* high when no free squares remain. The controller state machine is shown in the second image (omitting resets of the game). It toggles between X playing and O playing until the board is full or the currently playing side wins.





19.12 The state table for the machine is below. We must also include a counter that increments when the next state is 10 and resets to zero when the state becomes 00.

State	<i>in</i>	Next State	<i>out</i>
00	0	00	0
00	1	01	0
01	0	11	0
01	1	00	0
11	0	00	0
11	1	10	0
10	0	11	1
10	1	00	1

Chapter 20

Solutions: Verification and Test

20.1 A sample listing of features is detailed in the table below. This list is not exhaustive, but provides a sampling of different features.

Designation	Name	Description
I	Input	Values are successfully input into the system
I.1	1-digit numbers	All single digit numbers can be input
I.2, I.3, I.4	Multi-digit numbers	2-, 3-, 4-digit numbers are input correctly
I.5+	5-digits or more	Inputs with more than 5 digits generate the expected (illegal) behavior.
I.d	Decimal	Input decimal numbers
I.lz	Leading 0	Inputting any number of leading 0s to a number gives expected behavior
I.neg	Negative numbers	User can input negative numbers
I.fun	Function	User can successfully indicate which of the 4 arithmetic operations can be performed.
I.op	2nd number	Inputting a number after the input of a function
I.Eq1	Equals, expected	The equals input works after entering a number, function, and another number
I.Eq2	Equals, unexpected	The equals input works after entering a number and a function (no second number) or just after a single number.
I.post	After calculation	Correctly handling input after the user inputs the equals sign
A	Arithmetic	Testing the math computations themselves.
A.[+ - × ÷]	Addition	The basic operations work without overflow
A.over	Overflowed math	The basic operations work with overflow
A.div0	Divide by 0	Correct error behavior when the user divides by 0
A	...	Other math operations
D	Display	Operations that numbers, decimal points, and negative signs display correctly

20.2 This feature list would be similar to that of Table 20.1. Users can also add functionality such as a stop watch (including lap times) or multiple time zones. The feature list should include all input buttons and their function.

20.5 Below are six possible test patterns to be applied to the adder:

Input 1 (hex)	Input 2 (hex)	Reason
0000 0000	0000 0000	Ensure the combinational unit has no x outputs or other glaring errors
AAAA AAAA	5555 5555	Check that all sum bits work correctly when there are no carries
3FFF FFFF	0000 0001	Check for correct carry propagation
3FFF FFFF	3FFF FFFF	Largest addition of 2 positive numbers without overflow
7FFF FFFF	0000 0001	Positive overflow condition
FFFF FFFF	0000 0001	Check that adding a negative and positive number does not give overflow

20.8 All four input combinations are needed to test that every one of the outputs are assigned correctly.

20.10 The faults and test vectors are shown below. We only need inputs $\{a, b, cin\} = \{001, 010, 011, 110\}$ to test for all possible faults.

Fault	$\{a, b, cin\}$	$cout_{good}$	$cout_{bad}$
$g'-0$	110	1	0
$g'-1$	001	0	1
$p'-0$	001	0	1
$p'-1$	011	1	0
$cin-0$	011	1	0
$cin-1$	010	0	1

20.11 We need to use vectors 001, 011, and 010 to test Q3.

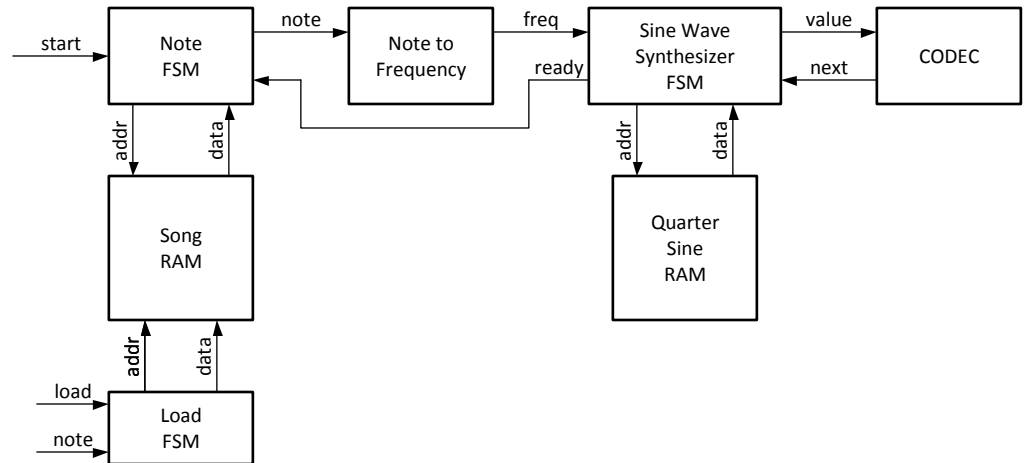
Fault	$\{a, b, cin\}$	s_{good}	s_{bad}
$p'-0$	001	1	0
$p'-1$	011	0	1
$cin-0$	011	0	1
$cin-1$	010	1	0

Chapter 21

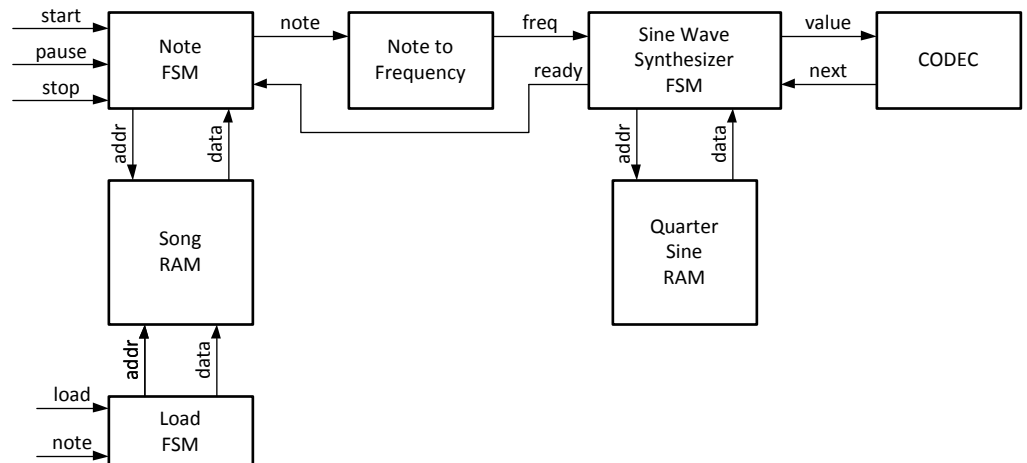
Solutions: System-Level Design

21.1 Topics mentioned in the text that are unspecified are the velocity of a serve, paddle size, etc. Other areas that need specification are the time-step, speed of the ball (grids per time-step), and the speed of the paddle. Moreover, the description should include the victory condition, what happens when the victory condition is obtained, and how to restart the game. One possible edge case is if the bottom of the paddle strikes the top of the ball. Users may also need to make sure that the ball does not have speed enough to “go through” the paddle and not detect the collision. Does the direction of the paddle during a collision or where on the paddle a collision occurs effect the direction of the ball?

21.5 The new block diagram is shown below, adding a new *Load FSM* module. When load is asserted and *mode* is idle, the load FSM will read each input node and save it into the RAM. The *load* signal is asserted during song playback. Playback cannot begin until after a song has been loaded and *load* has been deasserted.



21.6 The new block diagram is shown below. We have also updated the mode table to explain the functionality of the *pause* and *stop* signals.



Name	Description
idle	No music being played, goes to playback starting at note 0 on <i>start</i> and load on <i>load</i> .
playback	Generating audible output, goes to pause on <i>pause</i> and idle on <i>stop</i> .
pause	Not playing music, but will continue playing from current node on either <i>pause</i> or <i>start</i> . Will ignore <i>load</i> inputs.
loading	Loading a song, ignores all inputs that are not part of the load FSM.

Chapter 22

Solutions: Interface and System-Level Timing

- 22.1** Three further examples are: a timer for counting seconds, almost all visual displays (always valid to the observer), or signals that indicate a current long-term mode of device operation.
- 22.2** Periodic signals include a vending machine's coin inputs, the output of the arithmetic unit in dataflow FSMs, or messages sent across a bus (see Chapter 24).
- 22.4** The Verilog is shown below. We save the incoming whenever the register is not **full**, setting **full** to 0 when **count** reaches 4.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.ff.all;

entity rv2per is
  port(input: in std_logic_vector(7 downto 0);
        in_v: in std_logic;
        in_r: out std_logic;
        rst, clk: in std_logic;
        output: out std_logic_vector(7 downto 0) );
end rv2per;

architecture impl of rv2per is
  signal count: std_logic_vector(2 downto 0);
  signal nxt_count: std_logic_vector(2 downto 0);
  signal countIs5, full, saveData, nxt_full, nxt_full_r: std_logic;
  signal nxt_data: std_logic_vector(7 downto 0);
begin
  countIs5 <= '1' when (count = "100") else '0';
```

```

nxt_count <= "000" when (countIs5 or rst) else count + 1;

-- Save data if the data ff is empty, or at the end of the 5th
saveData <= countIs5 or not full;
nxt_data <= 8d"0" when rst else
    input when saveData else
    output;

process(all) begin
    case? std_logic_vector'(countIs5 & full & in_v) is
        when "000" => (nxt_full, in_r) <= std_logic_vector'("00");
        when "001" => (nxt_full, in_r) <= std_logic_vector'("11");
        when "01-" => (nxt_full, in_r) <= std_logic_vector'("10");
        when "1-0" => (nxt_full, in_r) <= std_logic_vector'("00");
        when "1-1" => (nxt_full, in_r) <= std_logic_vector'("11");
        when others => (nxt_full, in_r) <= std_logic_vector'("00");
    end case?;
end process;

nxt_full_r <= '0' when rst else nxt_full;

C: vDFF generic map(3) port map(clk, nxt_count, count);
D: vDFF generic map(8) port map(clk, nxt_data, output);
E: sDFF generic map(1) port map(clk, nxt_full_r, full);
end impl;

```

22.6 The double buffer design of Figure 23.11 meets the stated goals. We could have also designed a module where we wrote (and read) from the two flip-flops, alternating on every ready cycle.

22.8 One possible implementation of the serializer is show below:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.ff.all;

entity serializer is
    port(input: in std_logic_vector(63 downto 0);
         output: out std_logic_vector(7 downto 0);
         clk, sync: in std_logic );
end serializer;

architecture impl of serializer is
    signal count: std_logic_vector(2 downto 0);
    signal nxt_count: std_logic_vector(2 downto 0);
    signal countIs7, store: std_logic;
    signal data, nxt_data: std_logic_vector(63 downto 0);

```

```

begin
  -- Use to trigger the 1st valid 64 bits instead of a reset
  countIs7 <= '1' when (count = "111") else '0';
  store <= countIs7 or sync;
  nxt_count <= "000" when store else count + 1;

  nxt_data <= input when store else data;

  -- LSB first
  process(all) begin
    case (count) is
      when 3d"0" => output <= data(7 downto 0);
      when 3d"1" => output <= data(15 downto 8);
      when 3d"2" => output <= data(23 downto 16);
      when 3d"3" => output <= data(31 downto 24);
      when 3d"4" => output <= data(39 downto 32);
      when 3d"5" => output <= data(47 downto 40);
      when 3d"6" => output <= data(55 downto 48);
      when 3d"7" => output <= data(63 downto 56);
      when others => output <= (others => '-');
    end case;
  end process;

  C: vDFF generic map(3) port map(clk, nxt_count, count);
  D: vDFF generic map(64) port map(clk, nxt_data, data);
end impl;

```

22.14 An example timing table is below:

cycle	ball pos _x	ball pos _y	paddle _y	serve	score	points _x
i	5	50	3	0	0	x
i + 1	5	50	3	0	0	x
i + 20	4	50	4	0	0	x
i + 40	3	50	5	0	0	x
i + 60	2	50	6	0	0	x
i + 80	1	50	7	0	0	x
i + 100	0	50	8	0	1	x
i + 101	3	50	8	0	0	x+1
i + 102	3	50	8	0	0	x+1
...	3	50	8	0	0	x+1
i + j	3	50	8	1	0	x+1
i + j+1	4	50	8	0	0	x+1
i + j+31	5	50	8	0	0	x+1

22.15 The new timing table is below. We are able to start a new round of decryption 2 cycles (instead of 3) after Round 16 of a failed block.

Cycle	FK	NK	KGen	KSel	FB	NB	CT	SD	DES	Check
-1	1				1					
0			Key 0	Key 0			Block 0	1		
1			Key 1	Key 0					Round 1	
2				Key 0					Round 2	
...				Key 0					...	
15				Key 0		1			Round 15	
16				Key 0			Block 1	1	Round 16	
17		1	Key 1	Key 0	1			1	Round 1	Not PT
18			Key 2	Key 1			Block 0		Round 1	
19				Key 1					Round 2	
20				Key 1					Round 3	
...				Key 1					...	

Chapter 23

Solutions: Pipelines

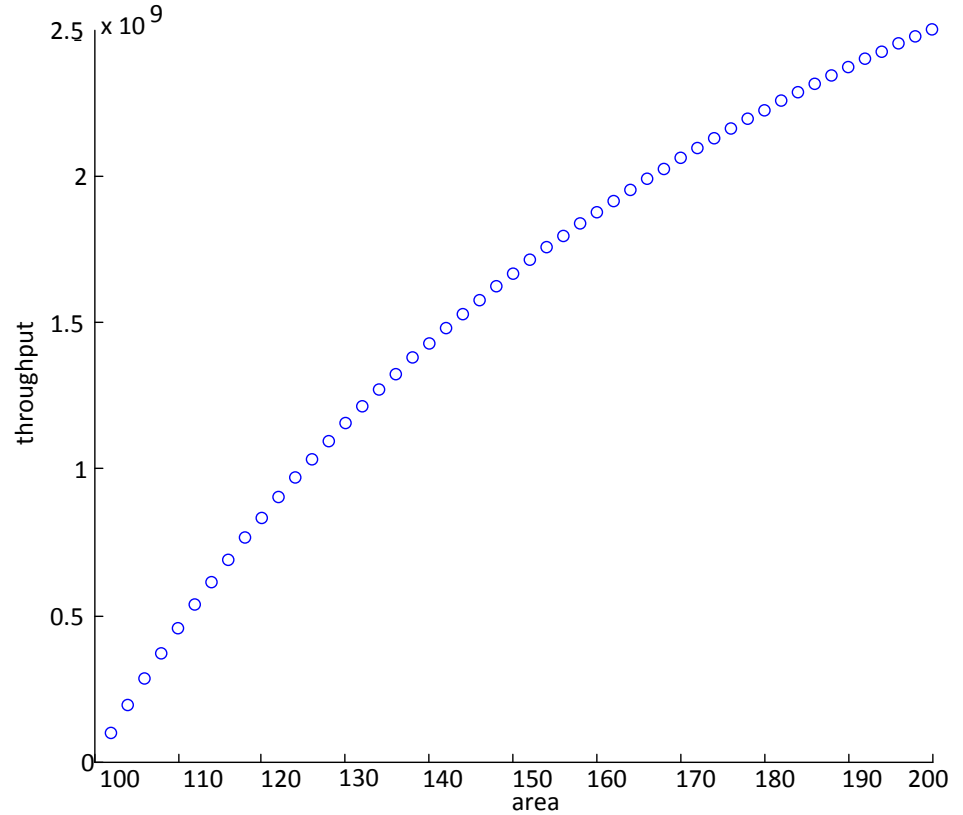
23.1 Using the equations from Section 23.1, the latency is 20.5ns and throughput is 48 780 000 s⁻¹.

23.2 Using the equations from Section 23.1, the latency is 20.5ns and throughput is 243 900 000 s⁻¹.

23.3 Using the equations from Section 23.1, the latency is 22.5ns and throughput is 222 222 222 s⁻¹.

23.4 Using the equations from Section 23.1, the latency is 22.5ns and throughput is 1 111 111 111 s⁻¹.

23.5 Our plot is shown below and shows that deep pipelining offers diminishing returns. *Errata: Ask the students to calculate the power (energy divided by clock time) for each pipeline depth.*



- 23.7**
1. It will take 200.5ns to complete the work (10 units of 20ns each, plus the final register delay)
 2. It will take 40.5ns to complete the work, since each of the 5 units will complete a task in 20ns.
 3. It will take 22.5ns to complete the first task (traversing the entire pipeline), and 4.5ns to complete each of the 9 subsequent tasks. This gives a total time of 63ns.
 4. The final answers are 20 000.5ns, 4 000.5ns, and 4 518ns. Note that the pipeline latency-penalty decreased from 50% to 13% as the batch size got larger.

23.14 The second stage is the bottleneck. The utilizations are 50%, 100%, 25%, and 33%.

23.15 To have no idle stages whatsoever, we need have a throughput of each stage equal to $200\,000\,000\text{ s}^{-1}$ we must use the following replication scheme:

6 copies of stage 1, 12 copies of stage 2, 3 copies of stage 3, and 4 copies of stage 4.

23.18 The expected number of total accesses in each cycle is 2. Thus, the expected utilizations of the stages are 100%, 66%, and 50% for $n=2,3,4$ (respectively). There are a handful of ways to compute the probabilities of conflicts, but one of which is to observe that there are 16 equally likely combinations of pipeline requests. Of these 16 (from 0000 to 1111): 1 has 0 requests, 4 have 1 request, 6 have 2 requests, 4 have 3 requests, and 1 has 4 requests. The over-subscription probabilities are $\frac{5}{16}$, $\frac{1}{16}$, and 0. This assumes a simplified model where subsequent requests are independent.

Chapter 24

Solutions: Interconnect

24.1 We can add full ready-valid flow control by adding three signals, as hinted in the problem statement. `ct_ready` is an input that indicates that the client is ready to receive data. `bt_ready` is an output to the bus that indicates that the bus interface is ready to receive data from the bus. `br_ready` is an input that indicates that the bus is ready to receive data. To send data three conditions need to be met: First, the sending client must have valid data (`cr_valid` asserted). Second, the receiving client must have space available (`ct_ready` asserted). Third, the arbiter must grant the sending client's request.

To enable back pressure from the receiving client to the sending client we assume a sending client with valid data first requests bus access by setting `arb_req` high. When the request is granted (`arb_grant` asserted), the sending client presents the receiving address and data to transmit to the bus on signals `br_addr` and `br_data` and sets `br_valid`. The bus interface `br_valid` signals are OR-ed together and fed back to the `bt_valid` to indicate the bus has valid address and data. When the address of the bus data is valid and equals the address of the receiving client the bus interface at the receiving client indicates to the client there is valid data by asserting `ct_valid`. The receiving client will only accept the valid data when it asserts `ct_ready`. To convey this ready status back to the sender, the receiving client sets `bt_ready` to `ct_ready` when the address of the bus data is valid and equals the address of the receiving client. The bus interface `bt_ready` signals are OR-ed together and fed back to all bus interfaces via `br_ready`. Finally, to complete the transfer, the `cr_ready` signal is asserted when the arbiter has granted the request and the `br_ready` indicates the receiving client has space.

The updated VHDL from 24.3 is below:

```
-- Combinational Bus Interface with full flow control
-- t (transmit) and r (receive) in signal names are from the
-- perspective of the bus
```

```

library ieee;
use ieee.std_logic_1164.all;

entity BusInt is
  generic( aw: integer := 2;  -- address width
           dw: integer := 4 ); -- data width
  port(
    -- bus rx - to the bus (client to bus interface)
    cr_valid: in std_logic;
    cr_ready: out std_logic;
    cr_addr: in std_logic_vector(aw-1 downto 0);
    cr_data: in std_logic_vector(dw-1 downto 0);

    -- bus tx - from the bus (bus interface to client)
    ct_valid: out std_logic;
    ct_data: out std_logic_vector(dw-1 downto 0);
    ct_ready: out std_logic;

    -- to the bus (bus interface to bus)
    br_addr: out std_logic_vector(aw-1 downto 0);
    br_data: out std_logic_vector(dw-1 downto 0);
    br_valid: out std_logic;
    br_ready: in std_logic;

    -- from the bus (bus to bus interface)
    bt_addr: in std_logic_vector(aw-1 downto 0);
    bt_data: in std_logic_vector(dw-1 downto 0);
    bt_valid: in std_logic;
    bt_ready: out std_logic;

    -- the arbiter
    arb_req: out std_logic;
    arb_grant: in std_logic;

    -- address of this interface
    my_addr: in std_logic_vector(aw-1 downto 0)
  );
end BusInt;

architecture impl of BusInt is
begin
  -- arbitration
  arb_req <= cr_valid;
  cr_ready <= arb_grant and br_ready;

  -- bus drive
  br_valid <= arb_grant;

```

```

br_addr <= cr_addr when arb_grant else (others => '0');
br_data <= cr_data when arb_grant else (others => '0');

-- bus receive
ct_valid <= '1' when (?? bt_valid) and (bt_addr = my_addr) else '0';
ct_data  <= bt_data;
bt_ready <= ct_ready when (?? bt_valid) and (bt_addr = my_addr) else '0';
end impl;

```

24.4 The VHDL for the bus interface below assumes the bus is ready to receive data from the sending client when **br_nready** (“bus receive NOT ready”) is ‘0’ and is not ready when **br_nready** is ‘1’. The signal **br_nready** is driven by OR-ing together the signals **bt_nready** driven by the individual client bus interfaces. The multibit signal **br_vector** is bit-wise OR-ed together and fed back to clients as **bt_vector**. The signal **bt_valid** is driven by OR-ing together the **br_valid** signals from individual bus interfaces.

The operation of a bus transfer is as follows: First, when the sending client asserts **cr_valid**, its bus interface asserts **arb_req** to request bus access. Second, after the arbiter grants this request by asserting **arb_grant**, the bus interface drives the multicast address **cr_vector** to **br_vector** along with the data to send. Note, the sending bus interface does not yet assert **br_valid**. The bus interfaces for clients that are not granted set **cr_vector** to all zeros.

Third, each of the receiving client interfaces reads the bit corresponding to its address from **bt_vector** which will only be ‘1’ if the sender has been granted bus access and wishes to send to this client. All receiving clients of the multicast that are not ready to receive new data will set **bt_nready** to ‘1’. Clients that have not been addressed by the multicast will drive ‘0’ to **bt_nready**. Thus, **br_nready** will be ‘1’ only if there is a receiving client that is not ready to accept data.

Fourth, once all receiving clients are ready **br_nready** will be ‘0’ which will cause the sending client bus interface to assert **cr_ready** and **br_valid** to indicate to the sending client and to the receiving client interfaces, respectively, that the data transfer can now proceed.

The updated VHDL from 24.3 is below. This builds on the VHDL from exercise 24.1.

```

-- Combinational Bus Interface with full flow control and multicast support
-- t (transmit) and r (receive) in signal names are from the
-- perspective of the bus
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity BusInt is
    generic( aw: integer := 2;    -- address width

```

```

        dw: integer := 4;    -- data width
        vw: integer := 2 ** aw ); -- vector width - corresponds to number
                                   -- of clients in address space, 2^aw

    port(
        -- bus rx - to the bus
        cr_valid: in std_logic;
        cr_ready: out std_logic;
        cr_data:  in std_logic_vector(dw-1 downto 0);
        cr_vector: in std_logic_vector(vw-1 downto 0);

        -- bus tx - from the bus
        ct_valid: out std_logic;
        ct_ready: in std_logic;
        ct_data:  out std_logic_vector(dw-1 downto 0);

        -- to the bus
        br_vector: out std_logic_vector(vw-1 downto 0);
        br_data:   out std_logic_vector(dw-1 downto 0);
        br_valid:  out std_logic;
        br_nready: in std_logic;

        -- from the bus
        bt_vector: in std_logic_vector(vw-1 downto 0);
        bt_data:   in std_logic_vector(dw-1 downto 0);
        bt_valid:  in std_logic;
        bt_nready: out std_logic;

        -- the arbiter
        arb_req:   out std_logic;
        arb_grant: in std_logic;

        -- address of this interface
        my_addr:   in std_logic_vector(aw-1 downto 0)
    );
end BusInt;

architecture impl of BusInt is
begin
    -- arbitration
    arb_req  <= cr_valid;
    cr_ready <= arb_grant and not br_nready;

    -- bus drive
    br_valid <= arb_grant and not br_nready;
    br_vector <= cr_vector when arb_grant else (others => '0');
    br_data  <= cr_data  when arb_grant else (others => '0');

```

```

-- bus receive
ct_valid <= bt_valid and bt_vector(to_integer(unsigned(my_addr)));
ct_data  <= bt_data;
bt_nready <= (not ct_ready) and bt_vector(to_integer(unsigned(my_addr)));
end impl;

```

24.4 To expand from a 2x2 crossbar to a 4x4 crossbar, we increase the size of the request/grant matrices. The VHDL below is an expansion of 24.5.

```

-- 4x4 Crossbar switch - full flow control
library ieee;
use ieee.std_logic_1164.all;

entity Xbar44 is
  generic( dw: integer := 4 ); -- data width
  port( -- client 0
    c0r_valid, c0t_ready: in std_logic;           -- r-v handshakes
    c0r_ready, c0t_valid: out std_logic;
    c0r_addr: in std_logic_vector(1 downto 0);    -- address
    c0r_data: in std_logic_vector(dw-1 downto 0); -- data
    c0t_data: out std_logic_vector(dw-1 downto 0);

    -- client 1
    c1r_valid, c1t_ready: in std_logic;
    c1r_ready, c1t_valid: out std_logic;
    c1r_addr: in std_logic_vector(1 downto 0);
    c1r_data: in std_logic_vector(dw-1 downto 0);
    c1t_data: out std_logic_vector(dw-1 downto 0);

    -- client 2
    c2r_valid, c2t_ready: in std_logic;
    c2r_ready, c2t_valid: out std_logic;
    c2r_addr: in std_logic_vector(1 downto 0);
    c2r_data: in std_logic_vector(dw-1 downto 0);
    c2t_data: out std_logic_vector(dw-1 downto 0);

    -- client 3
    c3r_valid, c3t_ready: in std_logic;
    c3r_ready, c3t_valid: out std_logic;
    c3r_addr: in std_logic_vector(1 downto 0);
    c3r_data: in std_logic_vector(dw-1 downto 0);
    c3t_data: out std_logic_vector(dw-1 downto 0) );
end Xbar44;

architecture impl of Xbar44 is
  signal req00, req01, req02, req03: std_logic;
  signal req10, req11, req12, req13: std_logic;
  signal req20, req21, req22, req23: std_logic;

```

```

signal req30, req31, req32, req33: std_logic;
signal grant00, grant01, grant02, grant03: std_logic;
signal grant10, grant11, grant12, grant13: std_logic;
signal grant20, grant21, grant22, grant23: std_logic;
signal grant30, grant31, grant32, grant33: std_logic;
begin

    -- request matrix
    req00 <= '1' when (c0r_addr = 2d"0") and (?? c0r_valid) else '0';
    req01 <= '1' when (c0r_addr = 2d"1") and (?? c0r_valid) else '0';
    req02 <= '1' when (c0r_addr = 2d"2") and (?? c0r_valid) else '0';
    req03 <= '1' when (c0r_addr = 2d"3") and (?? c0r_valid) else '0';
    req10 <= '1' when (c1r_addr = 2d"0") and (?? c1r_valid) else '0';
    req11 <= '1' when (c1r_addr = 2d"1") and (?? c1r_valid) else '0';
    req12 <= '1' when (c1r_addr = 2d"2") and (?? c1r_valid) else '0';
    req13 <= '1' when (c1r_addr = 2d"3") and (?? c1r_valid) else '0';
    req20 <= '1' when (c2r_addr = 2d"0") and (?? c2r_valid) else '0';
    req21 <= '1' when (c2r_addr = 2d"1") and (?? c2r_valid) else '0';
    req22 <= '1' when (c2r_addr = 2d"2") and (?? c2r_valid) else '0';
    req23 <= '1' when (c2r_addr = 2d"3") and (?? c2r_valid) else '0';
    req30 <= '1' when (c3r_addr = 2d"0") and (?? c3r_valid) else '0';
    req31 <= '1' when (c3r_addr = 2d"1") and (?? c3r_valid) else '0';
    req32 <= '1' when (c3r_addr = 2d"2") and (?? c3r_valid) else '0';
    req33 <= '1' when (c3r_addr = 2d"3") and (?? c3r_valid) else '0';

    -- arbitration 0 wins --> 3 loses
    grant00 <= req00;
    grant01 <= req01;
    grant02 <= req02;
    grant03 <= req03;

    grant10 <= req10 and not req00;
    grant11 <= req11 and not req01;
    grant12 <= req12 and not req02;
    grant13 <= req13 and not req03;

    grant20 <= req20 and not req10 and not req00;
    grant21 <= req21 and not req11 and not req01;
    grant22 <= req22 and not req12 and not req02;
    grant23 <= req23 and not req13 and not req03;

    grant30 <= req30 and not req20 and not req10 and not req00;
    grant31 <= req31 and not req21 and not req11 and not req01;
    grant32 <= req32 and not req22 and not req12 and not req02;
    grant33 <= req33 and not req23 and not req13 and not req03;

    -- connections

```

```

c0t_valid <= (grant00 and c0r_valid) or (grant10 and c1r_valid) or
              (grant20 and c2r_valid) or (grant30 and c3r_valid);
c0t_data  <= ((dw-1 downto 0 => grant00) and c0r_data) or
              ((dw-1 downto 0 => grant10) and c1r_data) or
              ((dw-1 downto 0 => grant20) and c2r_data) or
              ((dw-1 downto 0 => grant30) and c3r_data);

c1t_valid <= (grant01 and c0r_valid) or (grant11 and c1r_valid) or
              (grant21 and c2r_valid) or (grant31 and c3r_valid);
c1t_data  <= ((dw-1 downto 0 => grant01) and c0r_data) or
              ((dw-1 downto 0 => grant11) and c1r_data) or
              ((dw-1 downto 0 => grant21) and c2r_data) or
              ((dw-1 downto 0 => grant31) and c3r_data);

c2t_valid <= (grant02 and c0r_valid) or (grant12 and c1r_valid) or
              (grant22 and c2r_valid) or (grant32 and c3r_valid);
c2t_data  <= ((dw-1 downto 0 => grant02) and c0r_data) or
              ((dw-1 downto 0 => grant12) and c1r_data) or
              ((dw-1 downto 0 => grant22) and c2r_data) or
              ((dw-1 downto 0 => grant32) and c3r_data);

c3t_valid <= (grant03 and c0r_valid) or (grant13 and c1r_valid) or
              (grant23 and c2r_valid) or (grant33 and c3r_valid);
c3t_data  <= ((dw-1 downto 0 => grant03) and c0r_data) or
              ((dw-1 downto 0 => grant13) and c1r_data) or
              ((dw-1 downto 0 => grant23) and c2r_data) or
              ((dw-1 downto 0 => grant33) and c3r_data);

-- ready
c0r_ready <= (grant00 and c0t_ready) or (grant01 and c1t_ready) or
              (grant02 and c2t_ready) or (grant03 and c3t_ready);
c1r_ready <= (grant10 and c0t_ready) or (grant11 and c1t_ready) or
              (grant12 and c2t_ready) or (grant13 and c3t_ready);
c2r_ready <= (grant20 and c0t_ready) or (grant21 and c1t_ready) or
              (grant22 and c2t_ready) or (grant23 and c3t_ready);
c3r_ready <= (grant30 and c0t_ready) or (grant31 and c1t_ready) or
              (grant32 and c2t_ready) or (grant33 and c3t_ready);
end impl;

```

24.10 To build a buffered crossbar, we will expand upon the 2x2 crossbar in 24.5. We will start by creating a parametrized first-in, first-out (FIFO) buffer. This requires a random-access memory (RAM) and a counter.

```

-- A simple random access memory (RAM)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

```

```

entity RAM is
  generic( dw : integer := 4; -- data width
           aw : integer := 2; -- address width
           elements : integer := 2 ** aw ); -- number of elements in RAM
  port ( clk : in std_logic; -- clock input
        data_in: in std_logic_vector(dw-1 downto 0); -- data input
        data_out: out std_logic_vector(dw-1 downto 0);
        addr_in: in std_logic_vector(aw-1 downto 0); -- write address
        addr_out: in std_logic_vector(aw-1 downto 0); -- read address
        write_en: in std_logic ); -- write enable
end RAM;

architecture impl of RAM is
  -- memory array
  type mem_t is array(0 to elements-1) of std_logic_vector(dw-1 downto 0);
  signal memory: mem_t;
begin
  -- read/write memory on positive edge of clock
  process (clk) begin
    if rising_edge(clk) then
      data_out <= memory (to_integer(unsigned(addr_out))); -- get output value

      if (write_en) then
        memory (to_integer(unsigned(addr_in))) <= data_in; -- write to memory if enable high
      end if;
    end if;
  end process;
end impl;

-- A simple positive edge triggered D-flip flop
library ieee;
use ieee.std_logic_1164.all;

entity DFF is
  generic( width: integer := 1 );
  port( clk: in std_logic;
        d: in std_logic_vector(width-1 downto 0);
        q: out std_logic_vector(width-1 downto 0) );
end DFF;

architecture impl of DFF is
begin
  process (clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;

```



```

end impl;

-- A simple counter. Increments by 1 on every clock that enable is high.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
    generic( width: integer := 1 );
    port( clk: in std_logic; -- positive edge clocked
          rst: in std_logic; -- synchronous block reset
          count: out std_logic_vector(width-1 downto 0);
          enable: in std_logic );
end counter;

architecture impl of counter is
    signal next_count: std_logic_vector(width-1 downto 0);
begin
    -- increment by 1 if enabled - if reset is high, reset to 0
    next_count <= (others => '0') when rst else
        count + 1 when enable else
        count ;

    -- D-flip flop to store count value
    COUNT_REG: entity work.DFF(impl)
        generic map(width)
        port map (clk=>clk, d=>next_count, q=>count);
end impl;

-- A simple first-in, first-out (FIFO) buffer.
-- read data is always ready
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity FIFO is
    generic( dw: integer := 4; -- data width
            aw: integer := 2 ); -- address width for RAM in FIFO buffer
    port( clk: in std_logic; -- clock input
          rst: in std_logic; -- synchronous reset
          data_in: in std_logic_vector(dw-1 downto 0);
          data_out: out std_logic_vector(dw-1 downto 0);
          write_en: in std_logic; -- write enable
          read_en: in std_logic; -- read enable
          full: out std_logic;
          empty: out std_logic );
end FIFO;

```

```

architecture impl of FIFO is
  -- read and write pointers in buffer
  signal read_addr, write_addr: std_logic_vector(aw-1 downto 0);
begin
  -- read and write counters
  READ_COUNTER: entity work.counter generic map(aw)
    port map (clk=>clk, rst=>rst, count=>read_addr, enable=>read_en);
  WRITE_COUNTER: entity work.counter generic map(aw)
    port map (clk=>clk, rst=>rst, count=>write_addr, enable=>write_en);

  -- buffer memory
  MEMORY: entity work.RAM generic map(aw=>aw, dw=>dw)
    port map( clk=>clk, data_in=>data_in, data_out=>data_out,
      addr_in=>write_addr, addr_out=>read_addr,
      write_en=>write_en );

  -- buffer is empty when read and write pointers are equal
  empty <= '1' when write_addr = read_addr else '0';

  -- buffer is full when write pointer is 1 below read pointer
  full <= '1' when (write_addr - 1) = read_addr else '0';
end impl;

```

Now, we modify the previous 2x2 crossbar to add the FIFO buffers.

```

-- 2x2 Buffered crossbar switch - full flow control
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity BufXbar22 is
  generic( dw: integer := 4; -- data width
    aw: integer := 2 ); -- address width
  port( clk, rst: in std_logic; -- clk and rst to fifo buffers
    -- client 0
    c0r_valid, c0t_ready: in std_logic; -- r-v handshakes
    c0r_ready, c0t_valid: out std_logic;
    c0r_addr: in std_logic; -- address
    c0r_data: in std_logic_vector(dw-1 downto 0); -- data
    c0t_data: out std_logic_vector(dw-1 downto 0);

    -- client 1
    c1r_valid, c1t_ready: in std_logic; -- r-v handshakes
    c1r_ready, c1t_valid: out std_logic;
    c1r_addr: in std_logic; -- address
    c1r_data: in std_logic_vector(dw-1 downto 0); -- data
    c1t_data: out std_logic_vector(dw-1 downto 0) );

```

```
end BufXbar22;
```

```
architecture impl of BufXbar22 is
```

```
    -- buffer wires
```

```
    signal buf00_empty, buf01_empty, buf10_empty, buf11_empty: std_logic;
```

```
    signal buf00_full, buf01_full, buf10_full, buf11_full: std_logic;
```

```
    signal buf00_data, buf01_data, buf10_data, buf11_data: std_logic_vector(dw-1 downto 0);
```

```
    signal req00, req01, req10, req11: std_logic;
```

```
    signal grant00, grant01, grant10, grant11: std_logic;
```

```
begin
```

```
    -- request matrix
```

```
    req00 <= '1' when (c0r_addr = '0') and (?? c0r_valid) else '0';
```

```
    req01 <= '1' when (c0r_addr = '1') and (?? c0r_valid) else '0';
```

```
    req10 <= '1' when (c1r_addr = '0') and (?? c1r_valid) else '0';
```

```
    req11 <= '1' when (c1r_addr = '1') and (?? c1r_valid) else '0';
```

```
    -- arbitration 0 wins
```

```
    grant00 <= not buf00_empty;
```

```
    grant01 <= not buf01_empty;
```

```
    grant10 <= not buf10_empty and buf00_empty;
```

```
    grant11 <= not buf11_empty and buf01_empty;
```

```
    -- connections
```

```
    c0t_valid <= grant00 or grant10;
```

```
    c0t_data <= ((dw-1 downto 0 => grant00) and buf00_data) or
                ((dw-1 downto 0 => grant10) and buf10_data);
```

```
    c1t_valid <= grant01 or grant11;
```

```
    c1t_data <= ((dw-1 downto 0 => grant01) and buf01_data) or
                ((dw-1 downto 0 => grant11) and buf11_data);
```

```
    -- ready when all buffers for input are not full
```

```
    c0r_ready <= not buf00_full and not buf01_full;
```

```
    c1r_ready <= not buf10_full and not buf11_full;
```

```
    -- buffer instantiations
```

```
    BUF00: entity work.FIFO generic map( dw=>dw, aw=>aw)
```

```
        port map ( clk=>clk, rst=>rst,
```

```
                    data_out=>buf00_data, data_in=>c0r_data,
```

```
                    write_en=>req00, read_en=>grant00,
```

```
                    full=>buf00_full, empty=>buf00_empty);
```

```
    BUF01: entity work.FIFO generic map( dw=>dw, aw=>aw)
```

```
        port map ( clk=>clk, rst=>rst,
```

```
                    data_out=>buf01_data, data_in=>c0r_data,
```

```
                    write_en=>req01, read_en=>grant01,
```

```
                    full=>buf01_full, empty=>buf01_empty);
```

```
    BUF10: entity work.FIFO generic map( dw=>dw, aw=>aw)
```

```

    port map ( clk=>clk, rst=>rst,
               data_out=>buf10_data, data_in=>clr_data,
                                   write_en=>req10, read_en=>grant10,
                                   full=>buf10_full, empty=>buf10_empty);
BUF11: entity work.FIFO generic map( dw=>dw, aw=>aw)
    port map ( clk=>clk, rst=>rst,
               data_out=>buf11_data, data_in=>clr_data,
                                   write_en=>req11, read_en=>grant11,
                                   full=>buf11_full, empty=>buf11_empty);

end impl;

```

Chapter 25

Solutions: Memory Systems

25-1 1. 13-bits total are needed

$$\{\text{word}, \text{byte}\} = \{\text{address}[12:2], \text{address}[1:0]\}$$

2. The size of one word is $8 \times 2 = 16\text{B}$. We need a total of 10 bits to address each of the 1024 words.

$$\{\text{word}, \text{byte}\} = \{\text{address}[13:4], \text{address}[3:0]\}$$

3. 17 bits total are needed.

$$\{\text{word}, \text{bank}, \text{byte}\} = \{\text{address}[16:8], \text{address}[7:4], \text{address}[3:0]\}$$

4. Each word is $16 \times 8 = 128\text{B}$.

$$\{\text{word}, \text{bank}, \text{byte}\} = \{\text{address}[19:10], \text{address}[9:7], \text{address}[6:0]\}$$

25-3 1. See the timing table below. The total time is 130 cycles.

Command	Time
Activate R0	5
Read C1	5
Read C2	5
Read C3	5
Precharge R0	5
Act R1	5
Read C0	5
RAS: Wait to PC	2
Precharge R1	5
Act R2	5
Read C0	5
RAS: Wait to PC	2
Precharge R2	5
Act Ra	5
Read C3	5
RAS: Wait to PC	2
Precharge Ra	5
Act Rb	5
Read C3	5
RAS: Wait to PC	2
Precharge Rb	5
Act R0	5
Read C4	5
RAS: Wait to PC	2
Precharge R0	5
Act Rb	5
Read C1	5
Read C2	5
Precharge Rb	5
Total	130 cycles

2. See the timing table below with rearranged values. This solutions uses a greedy algorithm to group all accesses to one row together. The total time is 106 cycles.

Command	Time
Activate R0	5
Read C1	5
Read C2	5
Read C3	5
Read C4	5
Precharge R0	5
Act R1	5
Read C0	5
RAS: Wait to PC	2
Precharge R1	5
Act R2	5
Read C0	5
RAS: Wait to PC	2
Precharge R2	5
Act Ra	5
Read C3	5
RAS: Wait to PC	2
Precharge Ra	5
Act Rb	5
Read C3	5
Read C1	5
Read C2	5
Precharge Rb	5
Total	106 cycles

25.6 With a single word cache line, the cache hit rate will be 85% (the baseline). The sequence of addresses does not matter. With a line size of n , however, we can eliminate (with $P = 0.95$) the next $n-1$ misses. As a simplified example with 1000 cache accesses, we have a total of 150 cache misses. With a line size of 2, the number of misses is reduced by about 71 (92% hit rate). With line sizes of 4 and 8, the number is reduced by 107 (95.6%) and 125 (97.5%), respectively. The memory bandwidth required for *data* increasing with line size because unneeded words can potentially be fetched. The control bandwidth decreases, as there are less requests.

- 25-8**
1. We simply need two address such that $(A_1 \bmod n) = (A_2 \bmod n)$. Each access will conflict, evicting the previous address's line.
 2. A sequence of $n+1$ unique addresses will cause a miss on every access, assuming we evict the least recently used value.
 3. A sequence of $w+1$ addresses that map to the same set will never hit in the cache.

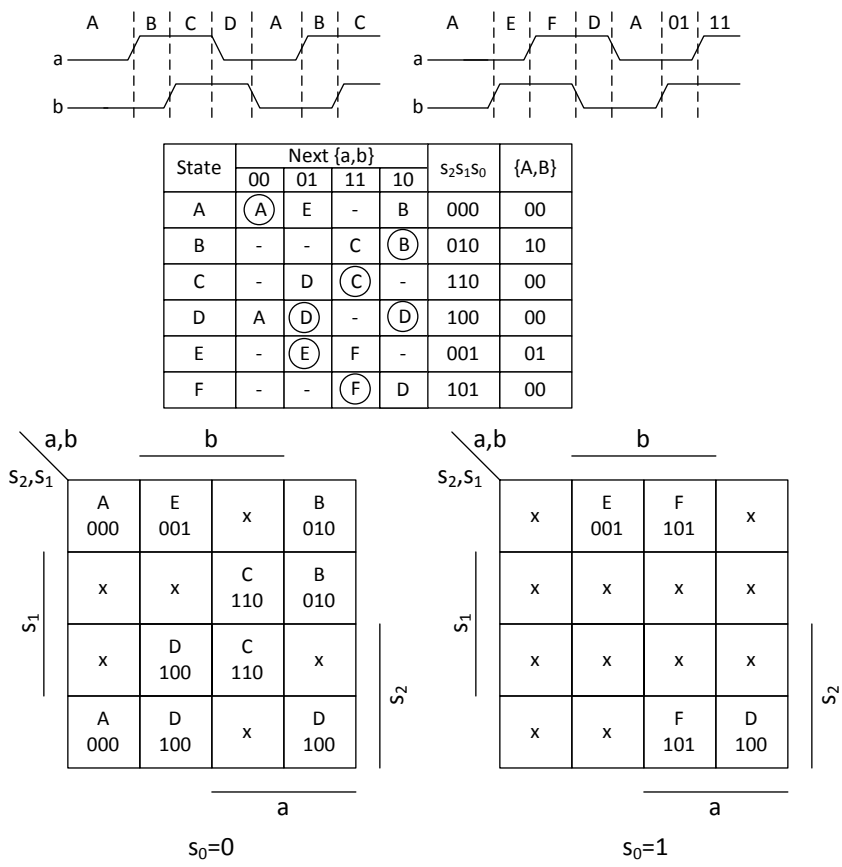
Chapter 26

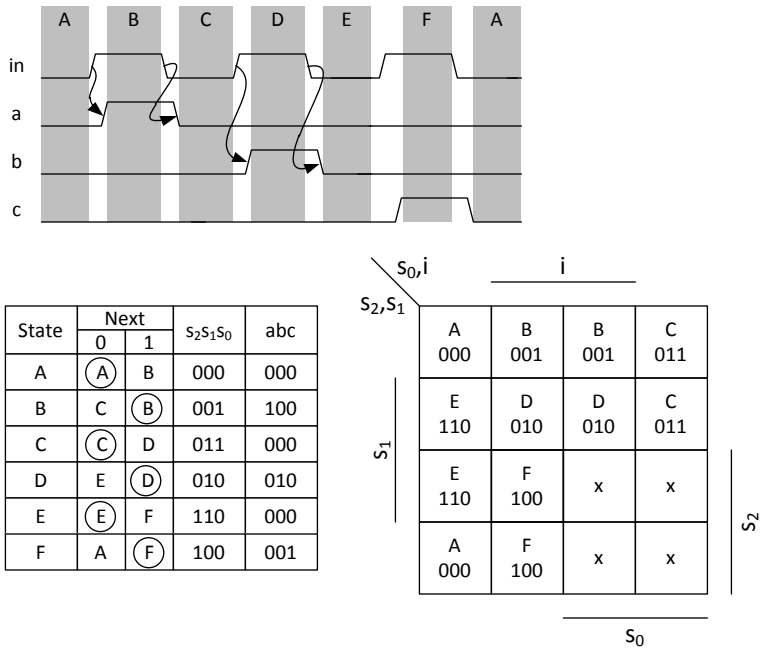
Solutions: Asynchronous Sequential Circuits

26.1 The flow table is shown below. The circuit sets itself into the 0 or 1 state when both of its inputs are 0 or 1, respectively. When the inputs are 01, the state toggles, and inputs 10 cause the state to hold.

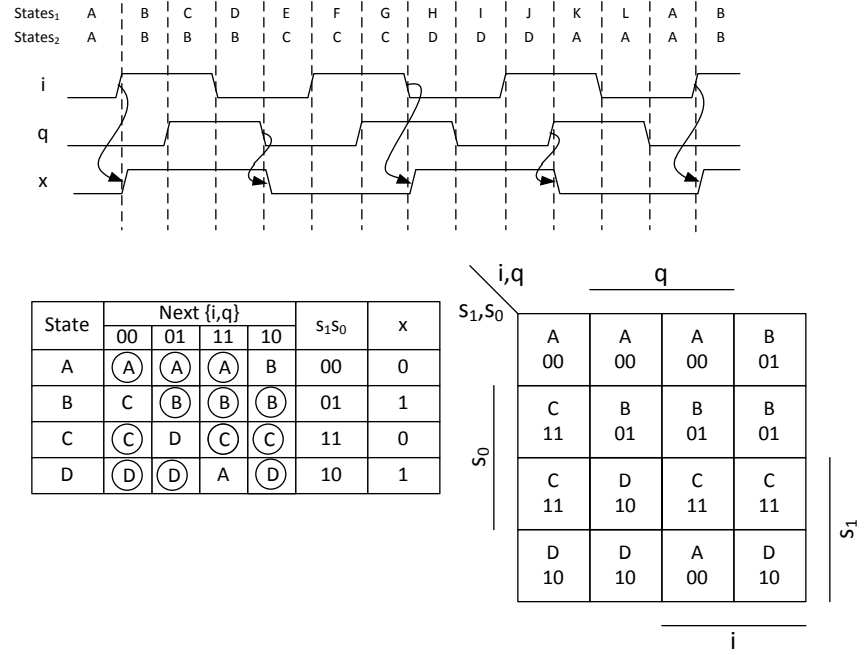
State	Next			
	00	01	11	10
0	0	1	1	0
1	0	0	1	1

26.2 The waveform, state transition table, and K-maps are shown below. We had to use 6 different states, and do not include transitions that are inconsistent with the problem description.

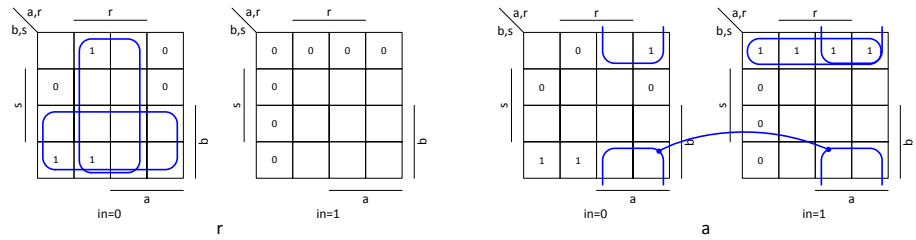




26.9 The solution is presented below. Taking the simple approach yields a total of 12 different states. However, if i is the first signal to rise, and i and q have the same frequency, only 4 states are necessary. We show the K-map for only the 4-state solution.



26.12 We allow the transition to B to go through 10000 or 10011 using the K-maps below. We changed the value for r at 10001 to be the desired final value of 0. The values at 10000 and 10000 must also be changed from x to the desired final state value so having a race results in the correct final state.



$$r = (b \wedge \overline{in}) \vee (r \wedge \overline{in})$$

$$a = (\overline{b} \wedge \overline{s} \wedge in) \vee (a \wedge \overline{s})$$

Chapter 27

Solutions: Flip-Flops

27.1 In the transition of D from 1 to 0, both inputs to U_4 must stabilize through the path of $U_1 + U_3 + U_5$ and separately through U_2 . The inputs to U_5 must be stable from a D transition from 0 to 1.

$$t_s = \max(t_1 + t_3 + t_5, t_2 + t_4)$$

27.2 The hold time in this situation is 0. Once the g input falls to 0, no change in d can change s' , r' or the outputs.

$$t_h = 0$$

27.3 The maximum delay comes from a transition from 1 to 0.

$$t_{dDQ} = \max(t_1 + t_3 + t_5, t_2) + t_4$$

27.4 The enable to q delay is given by:

$$t_{dGQ} = \max(t_3 + t_5, t_2) + t_4$$

27.5 As stated in the text, the setup time is that of the master.

$$t_s = \max(t_2 + t_4 + t_6, t_3 + t_5)$$

27.8 The setup time is simply $t_g + t_2$.

27.17 The state transitions are shown in the table below. The state is listed from the output of the top NAND gate to the bottom NAND gate. When the clock is off, the state toggles between 1110 and 0111 depending on d (via 0110 and 1111). When in one of the stable states (1110 or 0111) and the clock goes high, the system moves into either 1010 or 0101. This causes the output of the final RS latch to take on the last value of d . While the clock remains high, no change to the input is reflected in the output.

State	Next {d,c}				q
	00	01	11	10	
1110	1111	-	1010	1110	q
1111	0111	-	-	-	q
0111	0111	0101	-	0110	q
0110	-	-	-	1110	q
1010	-	1011	1010	1110	1
1011	1111	1011	1010	-	1
0101	0111	0101	0101	0111	0

Chapter 28

Solutions: Metastability and Synchronization Failure

28.1 The system will settle in $4.1\tau_s$.

$$t_s = -\tau_s \log(0.016) = 4.1\tau_s$$

28.3 The minimum required voltage difference is 0.91mV

$$\Delta V(0) = 1V \exp\left(-\frac{7\tau_s}{\tau_s}\right) = 0.91mV$$

28.6 There will be an error on about 40% of the asynchronous signal transitions.

$$P_E = \frac{t_s + t_h}{t_{cy}} = 0.4$$

28.9 The asynchronous signal must transition no more than 250 times a second.

$$f_E = \frac{f_E}{\frac{t_s + t_h}{t_{cy}}} = 250Hz$$

28.12 The answers are shown below. We can compute the ratio of errors in FF1 vs. FF2 below:

$$\begin{aligned} \frac{P_1}{P_2} &= \frac{(t_{s1} + t_{h1}) \exp\left(\frac{-t_w}{\tau_{s1}}\right)}{(t_{s2} + t_{h2}) \exp\left(\frac{-t_w}{\tau_{s2}}\right)} \\ \frac{P_1}{P_2} &= 0.2 \exp(5 * 10^{10} t_w) \end{aligned}$$

1. The first flip-flop is most desirable because it is least likely to enter an illegal state.
2. The second flip-flop will have an error about $2.4\times$ less often.

Chapter 29

Solutions: Synchronizer Design

29.2 See below:

$$\begin{aligned}P_{ES} &= \left(\frac{t_s + t_h}{t_{cy}} \right) \exp \left(\frac{-(5t_{cy} - t_s - t_{dCQ})}{\tau_s} \right) \\&= 5.8 \times 10^{-28} \\f_{ES} &= f_a P_{ES} \\&= 1.2 \times 10^{-21} \\MTBF &= 8.63 \times 10^{18} s\end{aligned}$$

29.3 See below, noting that 5 flip-flops gives a total wait time of 4 cycles:

$$\begin{aligned}P_{ES} &= \left(\frac{t_s + t_h}{t_{cy}} \right) \exp \left(\frac{-4(t_{cy} - t_s - t_{dCQ})}{\tau_s} \right) \\&= 3.0 \times 10^{-20} \\f_{ES} &= f_a P_{ES} \\&= 5.9 \times 10^{-12} \\MTBF &= 1.7 \times 10^{11} s\end{aligned}$$

29.5 The final answer is that we must only wait 2 clock cycles. First, we calculate the failure frequency:

$$\begin{aligned}MTBF &= (10^6)(30)(365)(24)(3600) = 9.5 \times 10^{14} s \\f_{ES} &= 1.1 \times 10^{-15} Hz\end{aligned}$$

Next, we can compute the probability of error and finally the error window:

$$\begin{aligned}
 P_{ES} &= \frac{f_{ES}}{f_a} = 3.2 \times 10^{-16} \\
 \exp\left(\frac{-t_w}{\tau_s}\right) &= P_{ES} \frac{t_{cy}}{t_s + t_h} = 4.5 \times 10^{-15} \\
 t_w &= -\tau_s \log(4.5 \times 10^{-15}) = 1.32ns \\
 N &\geq \frac{t_w + t_s + t_{dCQ}}{t_{cy}} \geq 1.4
 \end{aligned}$$

29.7 The time between bit transitions must be at least $t_{cy,o} + t_s + t_h$. This is to ensure that two transitions can enter into an illegal state in consecutive cycles.

29.9 In the simplest form of the problem requires placing synchronizers on the *increment* and *decrement* signals. This is a case in which the input logic does not require knowledge of the output signal (can increment past f_{16}). We may want to include logic that will only move the counter once for every positive edge of an input and not continuously.

29.11 The figure below shows the control data-path for indicating if a particular register is ready for new data (not present) or valid (present). We construct an FSM with 4 states, keeping 1 bit of state in each clock domain. In 2 of the states (00, 11) the register is considered empty, while it is considered full in the others. The state diagram is shown at the bottom of the figure. Note that only logic from one clock domain can only change the state bit in the same domain.

