CHAPTER 3

SEQUENTIAL LOGIC DESIGN -- CONTROLLERS

3.1 EXERCISES

Any problem noted with an asterisk (*) represents an especially challenging problem.

Section 3.2: Storing One Bit—Flip-Flops

3.1. Compute the clock period for the following clock frequencies.

a. 50 kHz (early computers)
b. 300 MHz (Sony Playstation 2 processor)
c. 3.4 GHz (Intel Pentium 4 processor)
d. 10 GHz (PCs of the early 2010s)
e. 1 THz (1 terahertz) (PCs of the future?)

a) 1/50,000 = 0.00002 s = 20 us
b) 1/300,000,000 = 3.33 ns
c) 1/3,400,000,000 = 294 ps = 0.294 ns
d) 1/10,000,000,000 = 100 ps = 0.1 ns
e) 1/1,000,000,000,000 = 1 ps

3.2 Compute the clock period for the following clock frequencies.

a. 32.768 kHz
b. 100 MHz
c. 1.5 GHz
d. 2.4 GHz

a) 1/32768 = 30.5 us
b) 1/100,000,000 = 10 ns
c) 1/1,500,000,000 = 0.66 ns = 667 ps
d) 1/ 2,400,000,000 = 0.416 ns = 416 ps

3.3 Compute the clock frequency for the following clock periods.

a. 1 s
b. 1 ms
c. 20 ns
d. 1 ns
e. 1.5 ps

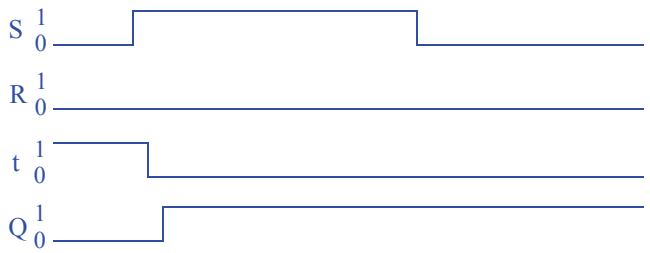
a) 1/1s = 1 Hz
b) 1/.001 = 1000 Hz = 1 kHz
c) 1/20ns = 50,000,000 Hz = 50 MHz
d) 1 /1ns = 1,000,000,000 = 1 GHz
e) 1/1.5ps = 666 GHz

3.4 Compute the clock frequency for the following clock periods.

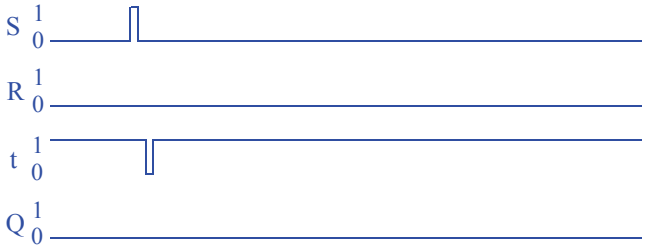
a. 500 ms
b. 400 ns
c. 4 ns
d. 20 ps

a) 1/500ms = 2 Hz
b) 1/400 ns = 2,500,000 Hz = 2.5 MHz
c) 1/4ns = 250,000,000 Hz = 250 MHz
d) 1/20ps = 50,000,000,000 Hz = 50 GHz

3.5 Trace the behavior of an SR latch for the following situation: Q, S, and R have been 0 for a long time, then S changes to 1 and stays 1 for a long time, then S changes back to 0. Using a timing diagram, show the values that appear on wires S, R, t, and Q. Assume logic gates have a tiny nonzero delay..



3.6 Repeat Exercise 3.5, but assume that S was changed to 1 just long enough for the signal to propagate through one logic gate, after which S was changed back to 0 -- in other words, S did not satisfy the hold time of the latch.



3.7 Trace the behavior of a level-sensitive SR latch (see Figure 3.16) for the input pattern in Figure 3.92. Assume S1, R1, and Q are initially 0. Complete the timing diagram, assuming logic gates have a tiny but non-zero delay.

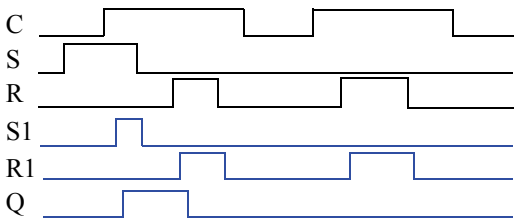


Figure 3.92

3.8 Trace the behavior of a level-sensitive SR latch (see Figure 3.16) for the input pattern in Figure 3.93. Assume S1, R1, and Q are initially 0. Complete the timing diagram, assuming logic gates have a tiny but non-zero delay.

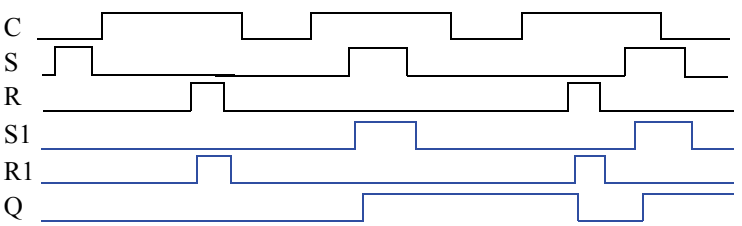


Figure 3.93

3.9 Trace the behavior of a level-sensitive SR latch (see Figure 3.16) for the input pattern in Figure 3.94. Assume S1, R1, and Q are initially 0. Complete the timing diagram, assuming logic gates have a tiny but non-zero delay..

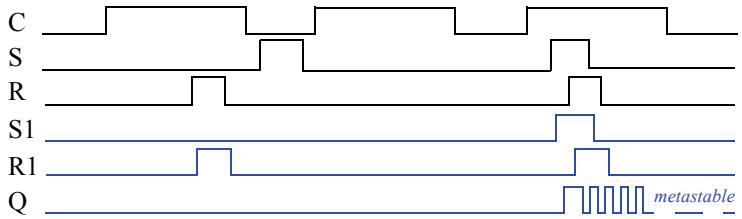


Figure 3.94

3.10 Trace the behavior of a D latch (see Figure 3.19) for the input pattern in Figure 3.95. Assume Q is initially 0. Complete the timing diagram, assuming logic gates have a tiny but non-zero delay.

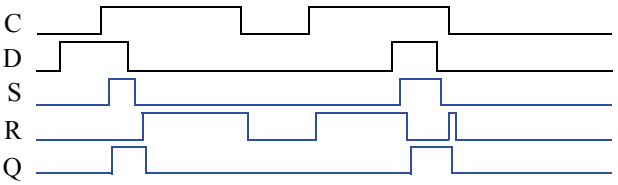


Figure 3.95

3.11 Trace the behavior of a D latch (see Figure 3.19) for the input pattern in Figure 3.96. Assume Q is initially 0. Complete the timing diagram, assuming logic gates have a tiny but non-zero delay.

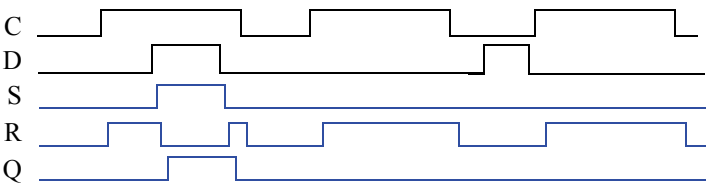


Figure 3.96

3.12 Trace the behavior of an edge-triggered D flip-flop using a master-servant design (see Figure 3.25) for the input pattern in Figure 3.97. Assume each internal latch initially stores a 0. Complete the timing diagram, assuming logic gates have a tiny but non-zero delay.

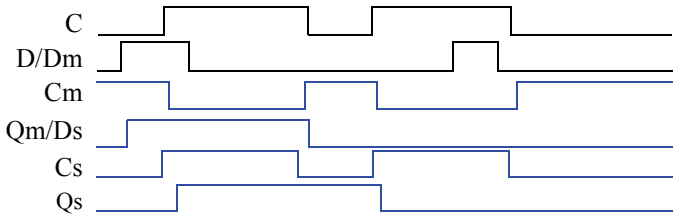


Figure 3.97

3.13 Trace the behavior of an edge-triggered D flip-flop using the master-servant design (see Figure 3.25) for the input pattern in Figure 3.98. Assume each internal latch initially stores a 0. Complete the timing diagram, assuming logic gates have a tiny but non-zero delay.

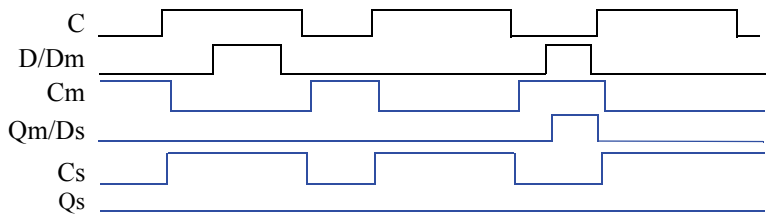


Figure 3.98

3.14 Compare the behavior of D latch and D flip-flop devices by completing the timing diagram in Figure 3.99. Provide a brief explanation of the behavior of each device. Assume each device initially stores a 0.

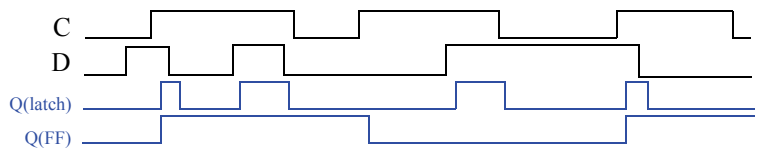


Figure 3.99

As long as the C (clock) input is 1, the D latch will store the value of D (after a short gate delay). The D flip-flop will only store the value of D on the rising edge of C (after a short gate delay).

3.15 Compare the behavior of D latch and D flip-flop devices by completing the timing diagram in Figure 3.100. Assume each device initially stores a 0. Provide a brief explanation of the behavior of each device.

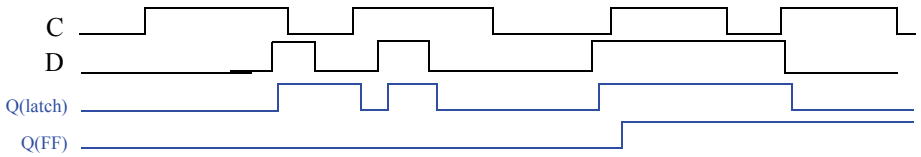
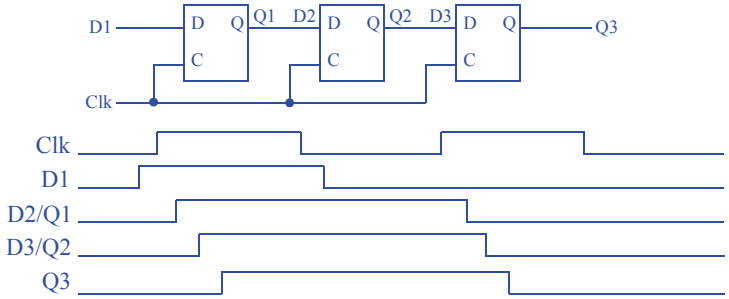


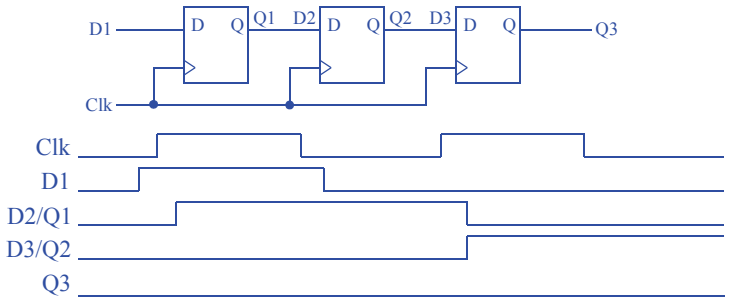
Figure 3.100

As long as the C (clock) input is 1, the D latch will store the value of D (after a short gate delay). The D flip-flop will only store the value of D on the rising edge of C (after a short gate delay).

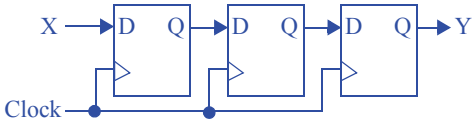
3.16 Create a circuit of three level-sensitive D latches connected in series (the output of one is connected to the input of the next). Use a timing diagram to show how a clock with a long high-time can cause the value at the input of the first D latch to trickle through more than one latch during the same clock cycle.



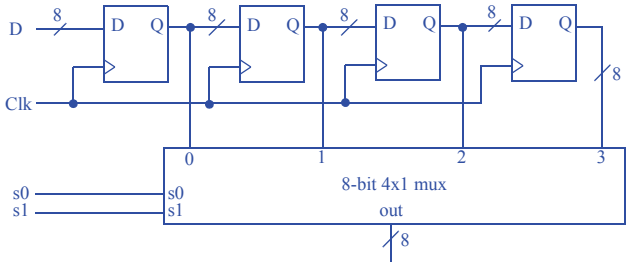
3.17 Repeat Exercise 3.16 using edge-triggered D flip-flops, and use a timing diagram to show how the input of the first D flip-flop does not trickle through to the next flip-flop no matter how long the clock signal is high.



3.18 A circuit has an input X that is connected to the input of a D flip-flop. Using additional D flip-flops, complete the circuit so that an output Y equals the output of X's flip-flop but delayed by two clock cycles.



3.19 Using four registers, design a circuit that stores the four values present at an 8-bit input D during the previous four clock cycles. The circuit should have a single 8-bit output that can be configured using two inputs s1 and s0 to output any one of the four registers. (Hint: use an 8-bit 4x1 mux.)



3.20 Consider three 4-bit registers connected as in Figure 3.101. Assume the initial values in the registers are unknown. Trace the behavior of the registers by completing the timing diagram of Figure 3.102.

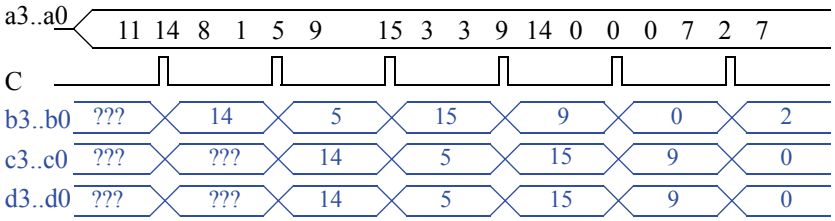


Figure 3.102

3.21 Consider three 4-bit registers connected as in Figure 3.103. Assume the initial values in the registers are unknown. Trace the behavior of the registers by completing the timing diagram of Figure 3.104.

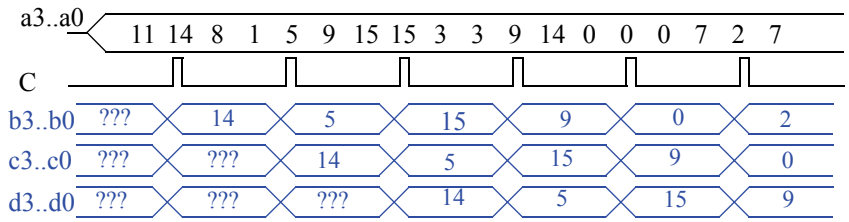
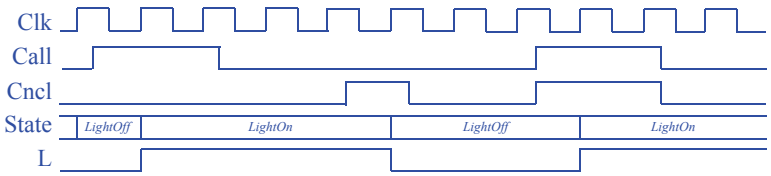


Figure 3.104

Section 3.3: Finite-State Machines (FSMs)

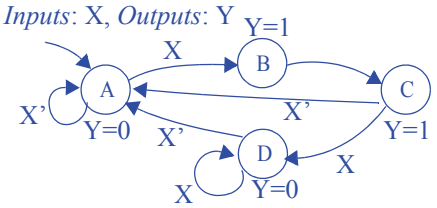
3.22 Draw a timing diagram (showing inputs, state, and outputs) for the flight-attendant call-button FSM of Figure 3.53 for the following scenario. Both inputs Call and Cncl are initially 0. Call becomes 1 for 2 cycles. Both inputs are 0 for 2 more cycles, then Cncl becomes 1 for 1 cycle. Both inputs are 0 for 2 more cycles, then both inputs Call and Cncl become 1 for 2 cycles. Both inputs become 0 for 1 last cycle. Assume any input changes occur halfway between two clock edges.



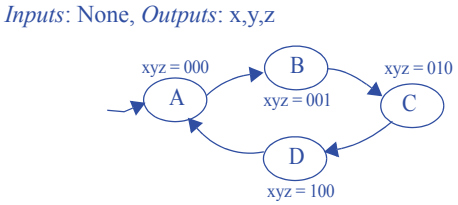
3.23 Draw a timing diagram (showing inputs, state, and outputs) for the code-detector FSM of Figure 3.58 for the following scenario. (Recall that when a button (or buttons) is pressed, a becomes 1 for exactly 1 clock cycle, no matter how long the button (or buttons) is pressed). Initially no button is pressed. The user then presses buttons in the following order: red, green, blue, red. Noticing the final state of the system, can you suggest an improvement to the system to better handle such incorrect code sequences?

Do not assign this exercise. The exercise refers to an earlier version of the figure, which was changed when creating the second edition, and thus the exercise description is not consistent with the figure.

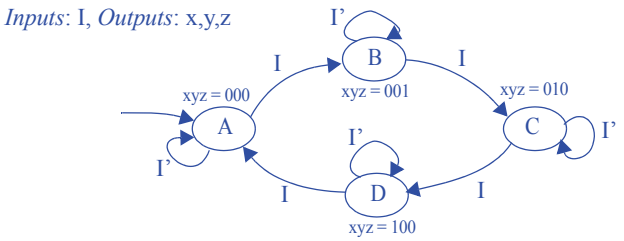
3.24 Draw a state diagram for an FSM that has an input X and an output Y. Whenever X changes from 0 to 1, Y should become 1 for two clock cycles and then return to 0 -- even if X is still 1. (Assume for this problem and all other FSM problems that an implicit rising clock is ANDed with every FSM transition condition.)



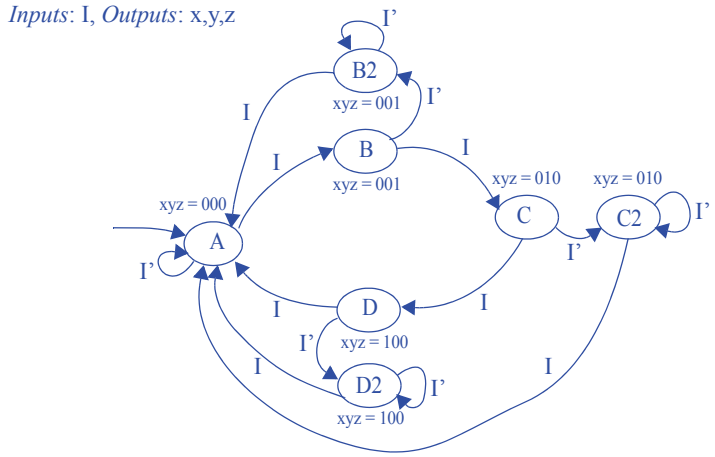
3.25 Draw a state diagram for an FSM with no inputs and three outputs x, y, and z. xyz should always exhibit the following sequence: 000, 001, 010, 100, repeat. The output should change only on a rising clock edge. Make 000 the initial state.



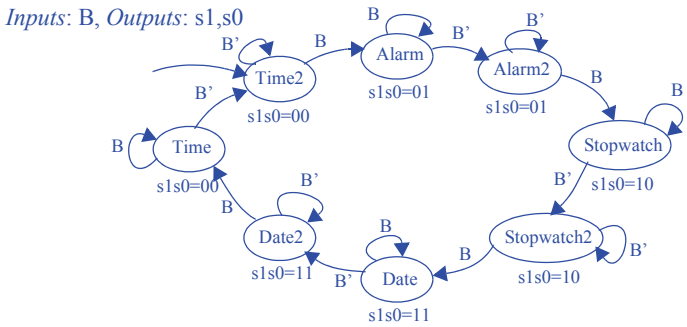
3.26 Do Exercise 3.25, but add an input I that can stop the sequence when set to 0. When input I returns to 1, the sequence resumes from where it left off.



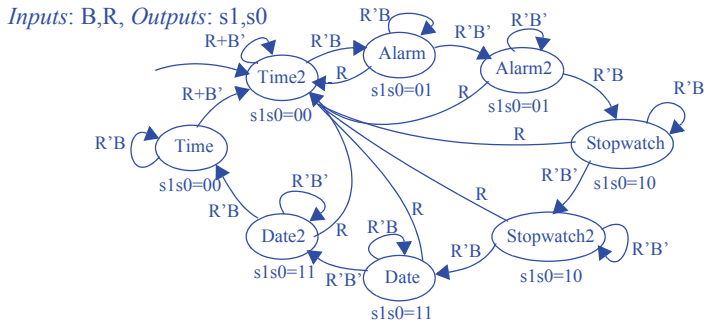
3.27 Do Exercise 3.25, but add an input I that can stop the sequence when set to 0. When I returns to 1, the sequence starts from 000 again..



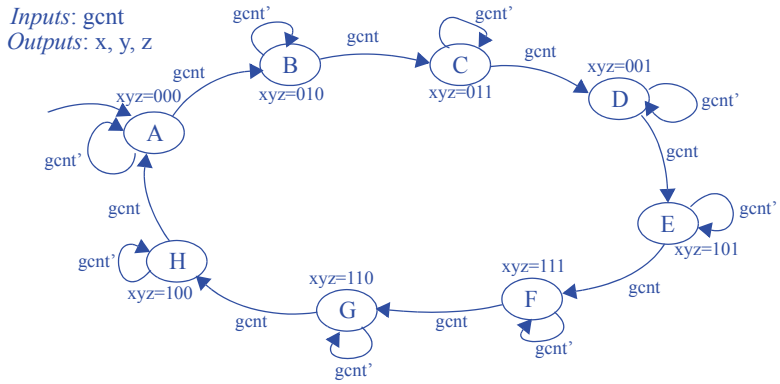
3.28 A wristwatch display can show one of four items: the time, the alarm, the stopwatch, or the date, controlled by two signals s1 and s0 (00 displays the time, 01 the alarm, 10 the stopwatch, and 11 the date—assume s1s0 control an N-bit mux that passes through the appropriate register). Pressing a button B (which sets B = 1) sequences the display to the next item. For example, if the presently displayed item is the date, the next item is the current time. Create a state diagram for an FSM describing this sequencing behavior, having an input bit B, and two output bits s1 and s0. Be sure to only sequence forward by one item each time the button is pressed, regardless of how long the button is pressed—in other words, be sure to wait for the button to be released after sequencing forward one item. Use short but descriptive names for each state. Make displaying the time be the initial state.



3.29 Extend the state diagram created in Exercise 3.28 by adding an input R. R=1 forces the FSM to return to the state that displays the time.



3.30 Draw a state diagram for an FSM with an input *gcnt* and three outputs, *x*, *y* and *z*. The *xyz* outputs generate a sequence called a Gray code in which exactly one of the three outputs changes from 0 to 1 or from 1 to 0. The Gray code sequence that the FSM should output is 000, 010, 011, 001, 101, 111, 110, 100, repeat. The output should change only on a rising clock edge when the input `gcnt` = 1. Make the initial state 000.



3.31 Trace through the execution of the FSM created in Exercise 3.30 by completing the timing diagram in Figure 3.107, where C is the clock input. Assume the initial state is the state that sets xyz to 000.

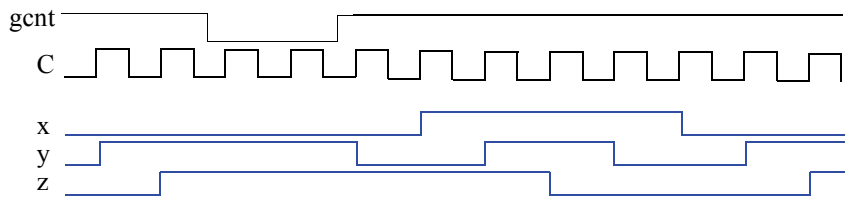
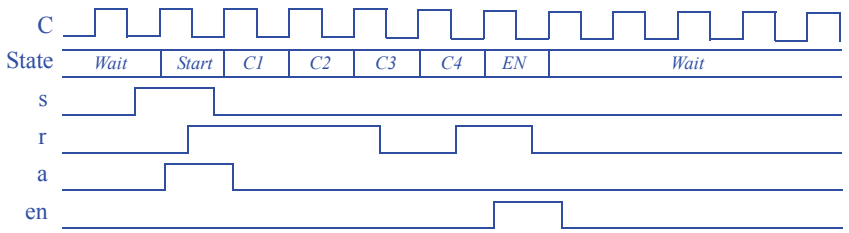


Figure 3.105

3.32 Draw a timing diagram for the FSM in Figure 3.108with the FSM starting in state *Wait*. Choose input values such that the FSM reaches state *EN,* and returns to *Wait*.



3.33 For FSMs with the following numbers of states, indicate the smallest possible number of bits for a state register representing those states:

a. 4
b. 8
c. 9
d. 23
e. 900

a) 2 bits
b) 3 bits
c) 4 bits
d) 5 bits
e) 10 bits

3.34 How many possible states can be represented by a 16-bit register?

$2^{16} = 65{,}536$ possible states

3.35 If an FSM has N states, what is the maximum number of possible transitions that could exist in the FSM? Assume that no pair of states has more than one transition in the same direction, and that no state has a transition point back to itself. Assuming there are a large number of inputs, meaning the number of transitions is not limited by the number of inputs? Hint: try for small N, and then generalize.

For two states A and B, there are only 2 possible transitions: A->B and B->A. For three states A, B, and C, possible transitions are A->B, A->C, B->A, B->C, C->A, and C->B, for 6 possible transitions. For each of N states, there can be up to N-1 transitions pointing to other states. Thus, the maximum possible is N*(N-1).

3.36 *Assuming one input and one output, how many possible four-state FSMs exist?

The complete solution to this challenge problem is not provided.The solution involves determining a way to enumerate all possible transitions from each state, and all possible actions in a state.

3.37 *Suppose you are given two FSMs that execute concurrently. Describe an approach for merging those two FSMs into a single FSM with identical functionality as the two separate FSMs, and provide an example. If the first FSM has N states and the second has M states, how many states will the merged FSM have?

The complete solution to this challenge problem is not provided. The solution involves creating the "cross product" of the two FSMs. If the first FSM has states n0 and n1, and the second has states m0, m1, and m2, then the cross product is an FSM having 2*3=6 states, which we might call n0m0, n0m1, n0m2, n1m0, n1m1, and n1m2. In each state, the actions of the two states from which that state is composed must all be included. Transitions must be combined also so that the transitions of the original FSMs are obeyed in the new FSM.

3.38 *Sometimes dividing a large FSM into two smaller FSMs results in simpler circuitry. Divide the FSM shown in Figure 3.111 into two FSMs, one containing G0-G3, the other containing G4-G7. You may add additional states, transitions, and inputs or outputs between the two FSMs, as required. Hint: you will need to introduce signals between the FSMs for one FSM to tell the other FSM to go to some state.

The solution idea involves the first FSM going to some new "idle" state rather than going to G4. Upon going to that idle state, the first FSM should tell the second FSM to go to G4. Meanwhile, the second FSM should be waiting in some new state until instructed to go to G4. Likewise, the second FSM should tell the first FSM when to go from its idle state to G0.

Section 3.4: Controller Design

3.39 Using the process for designing a controller, convert the FSM of Figure 3.109 to a controller, implementing the controller using a state register and logic gates.

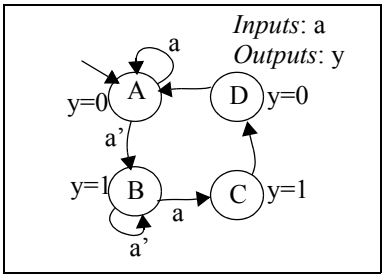
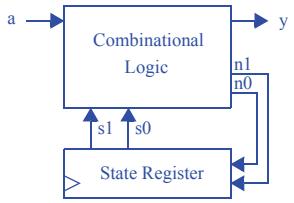


Figure 3.107

Step 1 - Capture the FSM
The appropriate FSM is given above.

Step 2A - Set up the architecture



Step 2B - Encode the states

A straightforward encoding is A=00, B=01, C=10, D=11.

Step 2C - Fill in the truth table

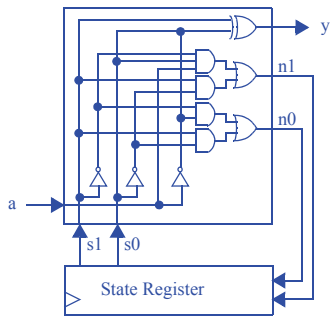
Inputs			Outputs		
s1	s0	a	n1	n0	y
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	0	0	0

Step 2D - Implement the combinational logic

n1 = s1's0a + s1s0'a' + s1s0'a = s1's0a + s1s0'

n0 = s1's0'a' + s1's0a' + s1s0'a' + s1s0'a = s1'a' + s1s0'

y = s1's0a' + s1's0a + s1s0'a' + s1s0'a = s1's0 + s1s0' = s1 xor s0



3.40 Using the process for designing a controller, convert the FSM of Figure 3.110 to a controller, implementing the controller using a state register and logic gates.

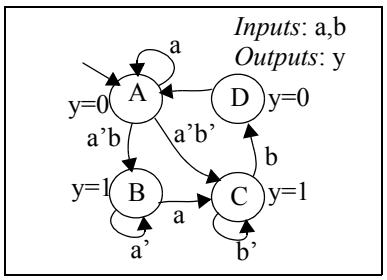
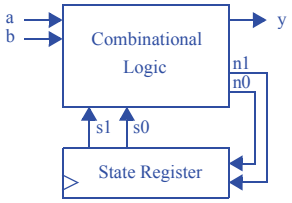


Figure 3.108

Step 1 - Capture the FSM
The appropriate FSM is given above.

Step 2A - Set up the architecture



Step 2B - Encode the states
A straightforward encoding is A=00, B=01, C=10, D=11.

Step 2C - Fill in the truth table

Inputs				Outputs		
s1	s0	a	b	n1	n0	y
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	0	1
1	0	0	1	1	1	1
1	0	1	0	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

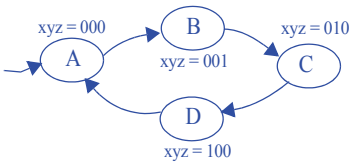
Step 2D - Implement the combinational logic

n1 = s1's0'a'b' + s1's0a + s1s0'
n0 = s1's0'a'b + s1's0a' + s1s0'b
y = s1's0 + s1s0'
Note: The above equations can be minimized further.

3.41 Using the process for designing a controller, convert the FSM you created for Exercise 3.24 to a controller, implementing the controller using a state register and logic gates.

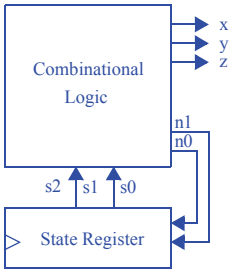
Step 1 - Capture the FSM

Inputs: None, *Outputs*: x,y,z



The FSM was created during Exercise 3.25.

Step 2A - Set up the architecture



Step 2B - Encode the states

A straightforward encoding is A=00, B=01, C=10, D=11.

Step 2C - Fill in the truth table

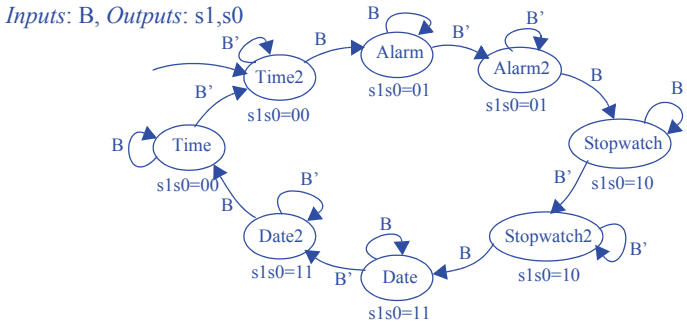
Inputs		Outputs				
s1	s0	n1	n0	x	y	z
0	0	0	1	0	0	0
0	1	1	0	0	0	1
1	0	1	1	0	1	0
1	1	0	0	1	0	0

Step 2D - Implement the combinational logic

n1 = s1's0 + s1s0' = s1 XOR s0
n0 = s1's0' + s1s0' = s0'
x = s1s0
y = s1s0'
z = s1's0

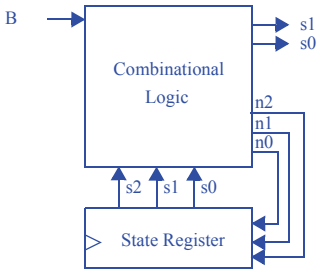
3.42 Using the process for designing a controller, convert the FSM you created for Exercise 3.28 to a controller, implementing the controller using a state register and logic gates.

Step 1 - Capture the FSM



The FSM was created during Exercise 3.28.

Step 2A - Set up the architecture



Step 2B - Encode the states

A straightforward encoding is Time2=000, Alarm=001, Alarm2=010, Stopwatch=011, Stopwatch2=100, Date=101, Date2=110, Time=111.

Step 2C - Fill in the truth table

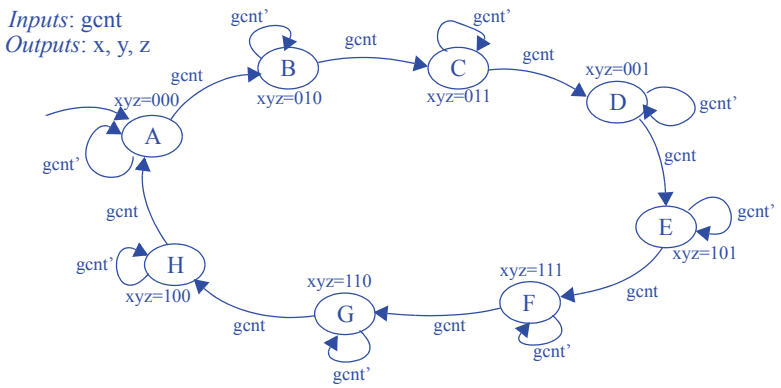
Inputs				Outputs				
s2	s1	s0	B	n2	n1	n0	s1	s0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0	1
0	0	1	1	0	0	1	0	1
0	1	0	0	0	1	0	0	1
0	1	0	1	0	1	1	0	1
0	1	1	0	1	0	0	1	0
0	1	1	1	0	1	1	1	0
1	0	0	0	1	0	0	1	0
1	0	0	1	1	0	1	1	0
1	0	1	0	1	0	1	1	1
1	0	1	1	1	1	0	1	1
1	1	0	0	1	1	0	1	1
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	1	0	0

Step 2D - Implement the combinational logic

n2 = s2's1s0B' + s2s1' + s2s0' + s2B
n1 = s1s0' + s1B + s2s0B + s2's1's0B'
n0 = s0'B + s2'B + s1B + s2s1's0B'
s1 = s2s0' + s2s1' + s2's1s0
s0 = s1 XOR s0

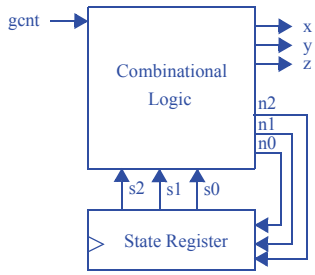
3.43 Using the process for designing a controller, convert the FSM you created for Exercise 3.30 to a controller, implementing the controller using a state register and logic gates.

Step 1 - Capture the FSM



The FSM was created during Exercise 3.30.

Step 2A - Set up the architecture



Step 2B - Encode the states

A straightforward encoding is A=000, B=001, C=010, D=011, E=100, F=101, G=110, H=111.

Step 2C - Fill in the truth table

	Inputs				Outputs					
	s2	s1	s0	gcnt	n2	n1	n0	x	y	z
A	0	0	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	1	0	0	0
B	0	0	1	0	0	0	1	0	1	0
	0	0	1	1	0	1	0	0	1	0
C	0	1	0	0	0	1	0	0	1	1
	0	1	0	1	0	1	1	0	1	1
D	0	1	1	0	0	1	1	0	0	1
	0	1	1	1	1	0	0	0	0	1
E	1	0	0	0	1	0	0	1	0	1
	1	0	0	1	1	0	1	1	0	1
F	1	0	1	0	1	0	1	1	1	1
	1	0	1	1	1	1	0	1	1	1
G	1	1	0	0	1	1	0	1	1	0
	1	1	0	1	1	1	1	1	1	0
H	1	1	1	0	1	1	1	1	0	0
	1	1	1	1	0	0	0	1	0	0

Step 2D - Implement the combinational logic

n2 = s2's1s0gcnt + s2s1' + s2s1s0' + s2s1s0gcnt'
n1 = s2's1's0gcnt + s2's1s0' + s2's1s0gcnt' + s2s1's0gcnt + s2s1s0' + s2s1s0gcnt'
n0 = s2's1's0'gcnt + s2's1's0gcnt' + s2's1s0'gcnt + s2's1s0gcnt' + s2s1's0'gcnt + s2s1's0gcnt' + s2s1s0'gcnt + s2s1s0gcnt'
x = s2
y = s2's1's0 + s2's1s0' + s2s1's0 + s2s1s0'
z = s2's1 + s2s1'
Note: The above equations can be minimized further.

3.44 Using the process for designing a controller, convert the FSM in Figure 3.111 to a controller, stopping once you have created the truth table. Note: your truth table will be quite large, having 32 rows -- you might therefore want to use a computer tool, like a word processor or spreadsheet, to draw the table.

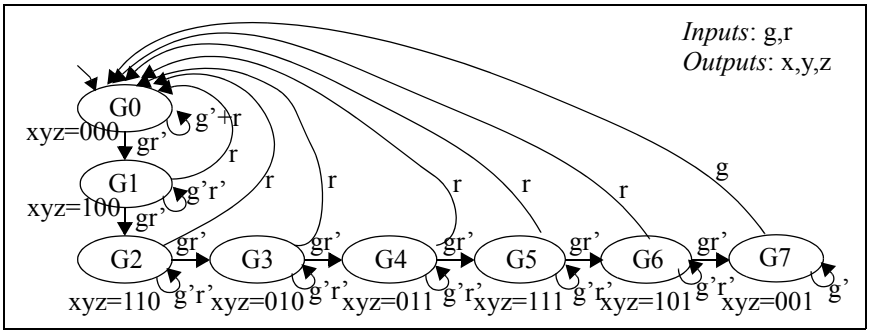
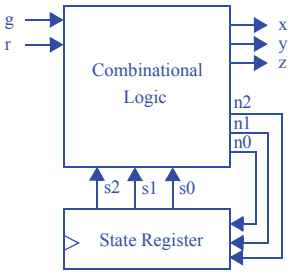


Figure 3.111

Step 1 - Capture the FSM

The FSM is given in Figure 3.111.

Step 2A - Set up the architecture



Step 2B - Encode the states

A straightforward encoding is G0=000, G1=001, G2=010, G3=011, G4=100, G5=101, G6=110, G7=111.

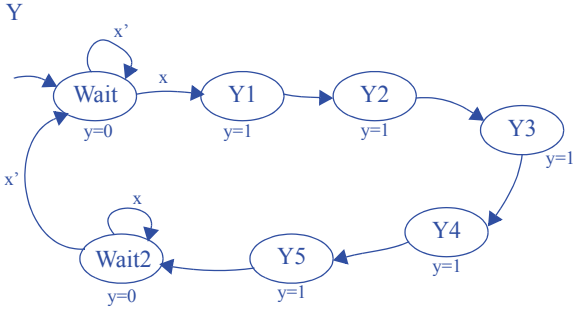
Step 2C - Fill in the truth table

	Inputs					Outputs					
	s3	s2	s1	g	r	n2	n1	n0	x	y	z
G0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	0	0	0
	0	0	0	1	0	0	0	1	0	0	0
	0	0	0	1	1	0	0	0	0	0	0
G1	0	0	1	0	0	0	0	1	1	0	0
	0	0	1	0	1	0	0	0	1	0	0
	0	0	1	1	0	0	1	0	1	0	0
	0	0	1	1	1	0	0	0	1	0	0
G2	0	1	0	0	0	0	1	0	1	1	0
	0	1	0	0	1	0	0	0	1	1	0
	0	1	0	1	0	0	1	1	1	1	0
	0	1	0	1	1	0	0	0	1	1	0
G3	0	1	1	0	0	0	1	1	0	1	0
	0	1	1	0	1	0	0	0	0	1	0
	0	1	1	1	0	1	0	0	0	1	0
	0	1	1	1	1	0	0	0	0	1	0
G4	1	0	0	0	0	1	0	0	0	1	1
	1	0	0	0	1	0	0	0	0	1	1
	1	0	0	1	0	1	0	1	0	1	1
	1	0	0	1	1	0	0	0	0	1	1
G5	1	0	1	0	0	1	0	1	1	1	1
	1	0	1	0	1	0	0	0	1	1	1
	1	0	1	1	0	1	1	0	1	1	1
	1	0	1	1	1	0	0	0	1	1	1
G6	1	1	0	0	0	1	1	0	1	0	1
	1	1	0	0	1	0	0	0	1	0	1
	1	1	0	1	0	1	1	1	1	0	1
	1	1	0	1	1	0	0	0	1	0	1
G7	1	1	1	0	0	1	1	1	0	0	1
	1	1	1	0	1	1	1	1	0	0	1
	1	1	1	1	0	0	0	0	0	0	1
	1	1	1	1	1	0	0	0	0	0	1

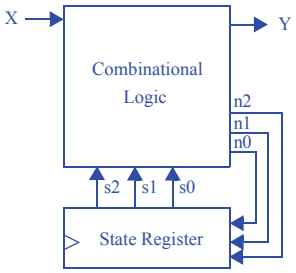
3.45 Create an FSM that has an input *X* and an output *Y*. Whenever *X* changes from 0 to 1, *Y* should become 1 for five clock cycles and then return to 0 -- even if *X* is still 1. Using the process for designing a controller, convert the FSM to a controller, stopping once you have created the truth table.

Step 1 - Capture the FSM

Inputs: X
Outputs: Y



Step 2A - Set up the architecture



Step 2B - Encode the states

A straightforward encoding is Wait=000, Y1=001, Y2=010, Y3=011, Y4=100, Y5=101, Wait2=110.

Step 2C - Create the state table

	Inputs				Outputs			
	s2	s1	s0	X	n2	n1	n0	Y
Wait	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	1	0
Y1	0	0	1	0	0	1	0	1
	0	0	1	1	0	1	0	1
Y2	0	1	0	0	0	1	1	1
	0	1	0	1	0	1	1	1
Y3	0	1	1	0	1	0	0	1
	0	1	1	1	1	0	0	1
Y4	1	0	0	0	1	0	1	1
	1	0	0	1	1	0	1	1
Y5	1	0	1	0	1	1	0	1
	1	0	1	1	1	1	0	1
Wait2	1	1	0	0	1	1	0	0
	1	1	0	1	0	0	0	0
	1	1	1	0	0	0	0	0
	1	1	1	1	0	0	0	0

Step 2D - Implement the combinational logic

n2 = s2s1’ + s2’s1s0 + s2s0’X’
n1 = s1’s0 + s2’s1s0’ + s1s0’X’
n0 = s2s1’s0’ + s2’s1s0’ + s2’s0’X
Y = (s2 xor s1) + s2’s0

3.46 The FSM in Figure 3.112 has two problems: one state has non-exclusive transitions, and another state has incomplete transitions. By ORing and ANDing the conditions for each state's transitions, prove that these problems exist. Then, fix these problems by refining the FSM, taking your best guess as to what was the FSM creator's intent.

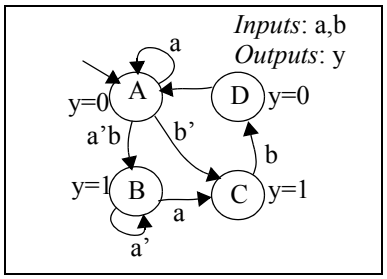
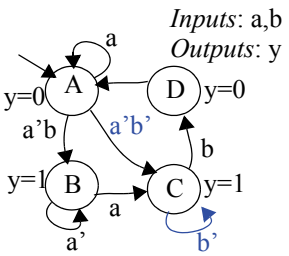


Figure 3.112

If we AND each pair of transitions with each other in state *A*, we get:
a * a'b = 0*b = 0
a'b * b' = a'*0 = 0
a*b' = ab', which is not equal to 0.
State A's transitions are thus not exclusive, i.e., both a and b' could be simultaneously true.

ORing state B's transitions yields:
a+a' = 1
ORing state C's transitions yields:
b
Clearly, state C's transitions are not completely specified, because their ORing doesn't result in 1. If b is 0, the FSM doesn't indicate what to do from state C.

We can address both of these problems with the following changes. The designer likely wanted to stay in state A when a is true, and go to B on a'b and go to C on a'b'. The designer likely wanted to stay in state C when b is 0.



3.47 Reverse engineer the poorly-designed three-cycles high circuit in Figure 3.41 to an FSM. Explain why the behavior of the circuit, as described by the FSM, is undesirable.

Step 2D was already completed, so we'll begin with Step 2C:

Step 2C - Fill in the truth table

Note that this circuit does not have the standard structure of a controller. However, we could say that the three flip-flops represent a 3-bit state register (so the leftmost flip-flop's value is the s2 signal, the middle flip-flop's value is the s1 signal, and the rightmost flip-flop's value is the s0 signal. Similarly, the input to the leftmost flip-flop, b, is n2, the signal from the output of the leftmost flip-flop to the input of the middle flip-flop is n1, and the signal from the output of the middle flip-flop is n0).

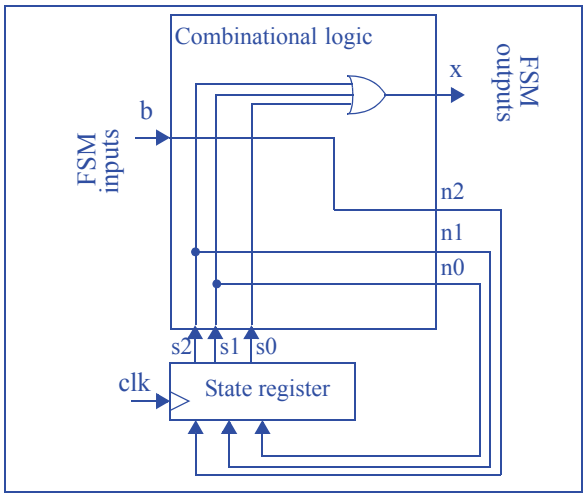
Inputs				Outputs			
s2	s1	s0	b	n2	n1	n0	x
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	0	0	1
0	0	1	1	1	0	0	1
0	1	0	0	0	0	1	1
0	1	0	1	1	0	1	1
0	1	1	0	0	0	1	1
0	1	1	1	1	0	1	1
1	0	0	0	0	1	0	1
1	0	0	1	1	1	0	1
1	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	1	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1

n2 = b; n1 = s2; n0 = s1; x = s2 + s1 + s0

Step 2B - Encode the states

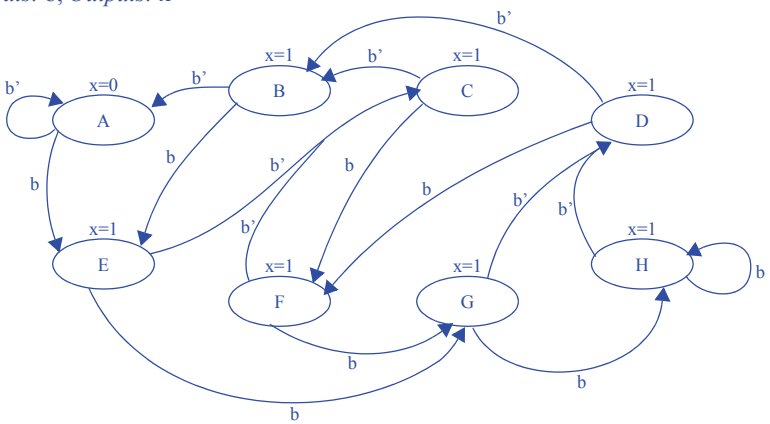
A straightforward encoding is A=000, B=001, C=010, D=011, E=100, F=101, G=110, H=111

Step 2A - Set up the architecture



Step 1: Capture the FSM

Inputs: b, *Outputs:* x



The behavior of this circuit is undesirable because if, after transitioning from A and before transitioning back to A, the user presses the button again, the output will stay on for more than three cycles.

3.48 Reverse engineer the behavior of the sequential circuit shown in Figure 3.113.

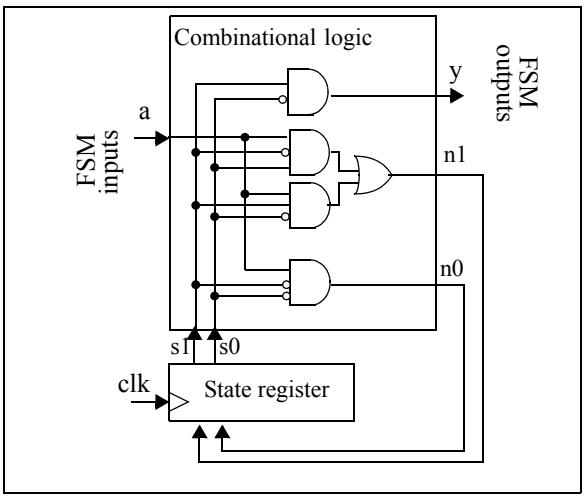


Figure 3.113

For this problem, we carry out the controller design process in reverse. We already have step 2D completed above, so we will begin with step 2C.

Step 2C - Fill in the truth table

Inputs			Outputs		
s1	s0	a	n1	n0	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	1	0	0	0

Step 2B - Encode the states

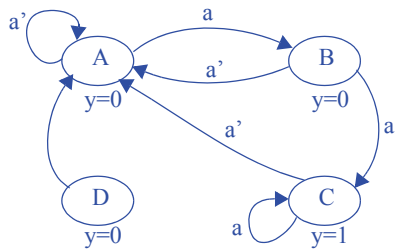
We will name the encodings as states as follows: 00=A, 01=B, 10=C, and 11=D.

Step 2A- Set up the architecture

The architecture has already been defined

Step 1 - Capture the FSM

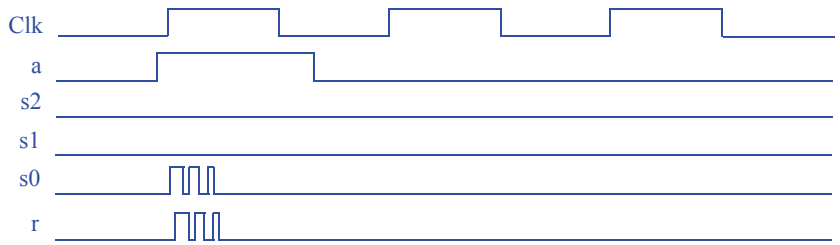
Inputs: a
Outputs: y



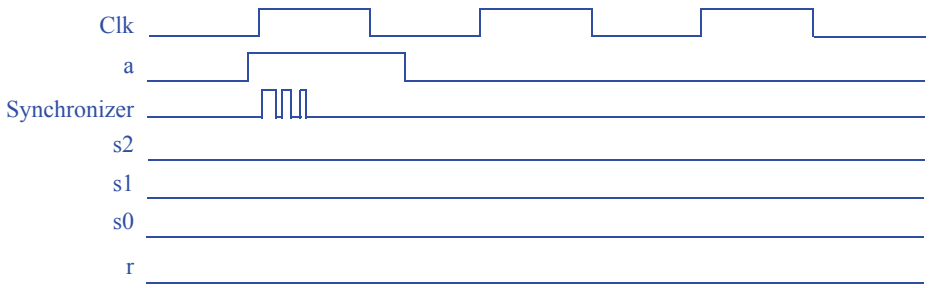
Section 3.5: More on Flip-Flops and Controllers

3.49 Use a timing diagram to illustrate how metastability can yield incorrect output for the secure car key controller of Figure 3.69. Use a second timing diagram to show how the synchronizer flip-flop introduced in Figure 3.84 may reduce the likelihood of such incorrect output.

Without Synchronizer:

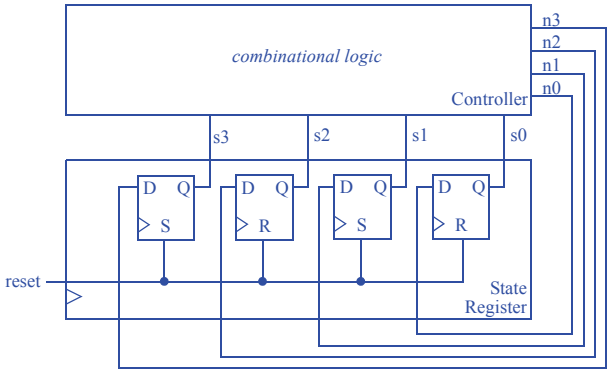


With Synchronizer:

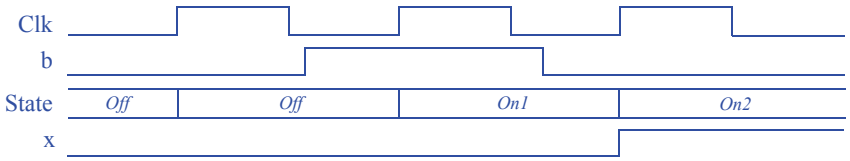


Note that in this case, even though metastability caused the Synchronizer flip-flop to end in zero (which caused us to miss the pulse on "a"), at least our state register did not go metastable, and as a result we did not experience incorrect output.

3.50 Design a controller with a 4-bit state register that gets synchronously initialized to state `1010` when an input *reset* is set to `1`.



3.51 Redraw the laser-timer controller timing diagram of Figure 3.63 for the case of the output being registered as in Figure 3.88.

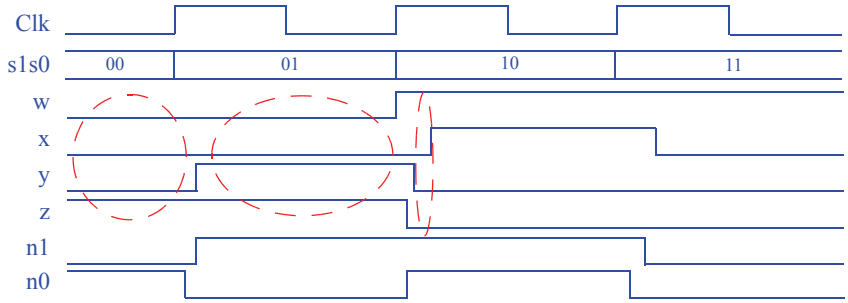


One more clock pulse has been added to show that the change of x is delayed by 1 pulse.

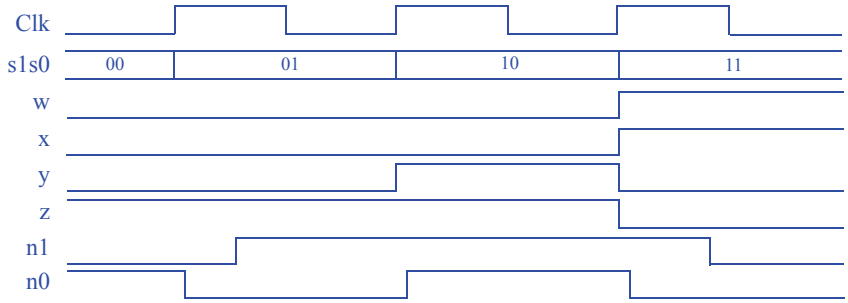
3.52 Draw a timing diagram for three clock cycles of the sequence generator controller of Figure 3.68 assuming that AND gates have a delay of 2 ns and inverters (including inversion bubbles) have a delay of 1 ns. The timing diagram should show the incorrect outputs that appear temporarily due to glitching. Then, introduce registered outputs to the controller using flip-flops at the outputs, and show a new timing diagram, which should no longer have glitches (but the output may be shifted in time).

Let's assume the delay of an XOR gate is the same as for an AND gate.

Unregistered Output:



Registered Output:



Note that we do not register the n1 or n0 outputs -- they are inputs to the state register.

Also note that the glitch here is not a temporary spurious ouput value on one control line, but a temporary spurious value on (wxyz) due to the varying delays for each of w, x, y, and z.