Chapter 7: Sequential Logic Design

In this chapter, we begin looking at sequential logic design. Sequential logic design differs from combinational logic design in that the outputs of the circuit depend not only on the current values of the inputs but also on the *past* values of the inputs. This is different from the combinational logic design, where the output of the circuitry depends only on the current values of the inputs. The ability of a sequential logic circuit to base its outputs on both current and past inputs allows more sophisticated and intelligent systems to be created. We begin by looking at sequential logic storage devices, which are used to hold the past values of a system. This is followed by an investigation of the timing considerations of sequential logic circuits. We then look at some useful circuits that can be created using only sequential logic storage devices. Finally, we look at one of the most important logic circuits in digital systems, the finite-state machine (FSM). The goal of this chapter is to provide an understanding of the basic operation of sequential logic circuits.

Learning Outcomes—After completing this chapter, you will be able to:

7.1 Describe the operation of a sequential logic storage device.
7.2 Describe sequential logic timing considerations.
7.3 Design a variety of common circuits based on sequential storage devices (toggle flops, ripple counters, switch debouncers, and shift registers).
7.4 Design a FSM using the classical digital design approach.
7.5 Design a counter using the classical digital design approach and an HDL-based structural approach.
7.6 Describe the FSM reset condition.
7.7 Analyze a FSM to determine its functional operation and maximum clock frequency.

7.1 Sequential Logic Storage Devices

7.1.1 The Cross-Coupled Inverter Pair

The first thing that is needed in sequential logic is a storage device. The fundamental storage device in sequential logic is based on a positive feedback configuration. Consider the circuit in Fig. 7.1. This circuit configuration is called the *cross-coupled inverter pair*. In this circuit, if the input of U1 starts with a value of 1, it will produce an output of Q = 0. This output is fed back to the input of U2, thus producing an output of Qn = 1. Qn is fed back to the original input of U1, thus reinforcing the initial condition. This circuit will *hold*, or *store,* a logic 0 without being driven by any other inputs. This circuit operates in a complementary manner when the initial value of U1 is 0. With this input condition, the circuit will store a logic 1 without being driven by any other inputs.

B. J. LaMeres, *Introduction to Logic Circuits & Logic Design with Verilog*,
https://doi.org/10.1007/978-3-031-43946-9_7

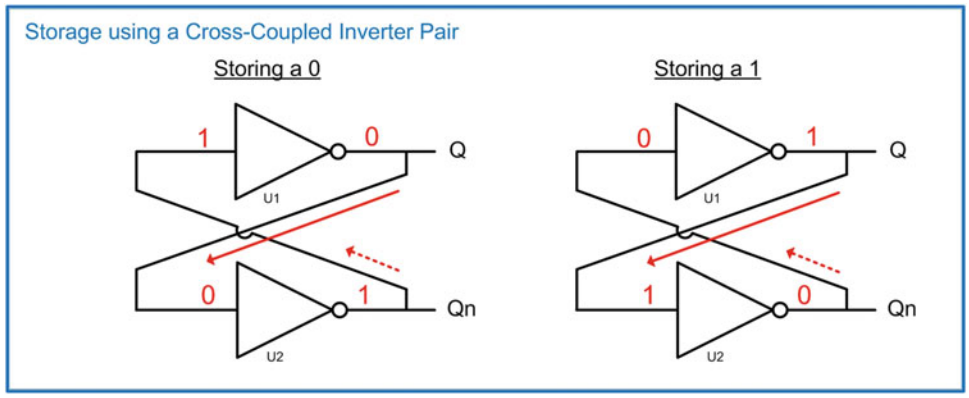


Fig. 7.1
Storage using a cross-coupled inverter pair

7.1.2 Metastability

The cross-coupled inverter pair in Fig. 7.1 exhibits what is called *metastable* behavior due to its positive feedback configuration. Metastability refers to when a system can exist in a state of equilibrium when undisturbed but can be moved to a different, more stable state of equilibrium when sufficiently disturbed. Systems that exhibit *high levels* of metastability have an equilibrium state that is highly unstable, meaning that if disturbed even slightly, the system will move rapidly to a more stable point of equilibrium. The cross-coupled inverter pair is a highly metastable system. This system actually contains three equilibrium states. The first is when the input of U1 is exactly between logic 0 and logic 1 (i.e., $V_{CC}/2$). In this state, the output of U1 is also exactly $V_{CC}/2$. This voltage is fed back to the input of U2, thus producing an output of exactly $V_{CC}/2$ on U2. This in turn is fed back to the original input on U1, reinforcing the initial state. Despite this system being at equilibrium in this condition, this state is highly unstable. With minimal disturbance to any of the nodes within the system, it will rapidly move to one of two more stable states. The two stable states for this system are when Q = 0 or when Q = 1 (see Fig. 7.1). Once the transition begins between the unstable equilibrium state toward one of the two more stable states, the positive feedback in the system continually reinforces the transition until the system reaches its final state. In electrical systems, this initial disturbance is caused by the presence of *noise* or unwanted voltage in the system. Noise can come from many sources, including the random thermal motion of charge carriers in semiconductor materials, electromagnetic energy, or naturally occurring ionizing particles. Noise is present in every electrical system, so the cross-coupled inverter pair will never be able to stay in the unstable equilibrium state where all nodes are at $V_{CC}/2$.

The cross-coupled inverter pair has two stable states; thus, it is called a *bistable* element. In order to understand the bistable behavior of this circuit, let's look at its behavior when the initial input value on U1 is set directly between logic 0 and logic 1 (i.e., $V_{CC}/2$) and how a small amount of noise will cause the system to move toward a stable state. Recall that an inverter is designed to have an output that quickly transitions between a logic LOW and HIGH in order to minimize the time spent in the uncertainty region. This is accomplished by designing the inverter to have what is called *gain*. Gain can be thought of as a multiplying factor that is applied to the input of the circuit when producing the output (i.e., $V_{out} = \text{gain} \cdot V_{in}$). The gain for an inverter will be negative since the output moves in the opposite direction of the input. The inverter is designed to have a very high gain, such that even the smallest change on the input when in the transition region will result in a large change on the output. Consider the behavior of this circuit shown in Fig. 7.2. In this example, let's represent the gain of the inverter as $-g$ and see how the system responds when a small positive voltage noise (V_n) is added to the $V_{CC}/2$ input on U1.

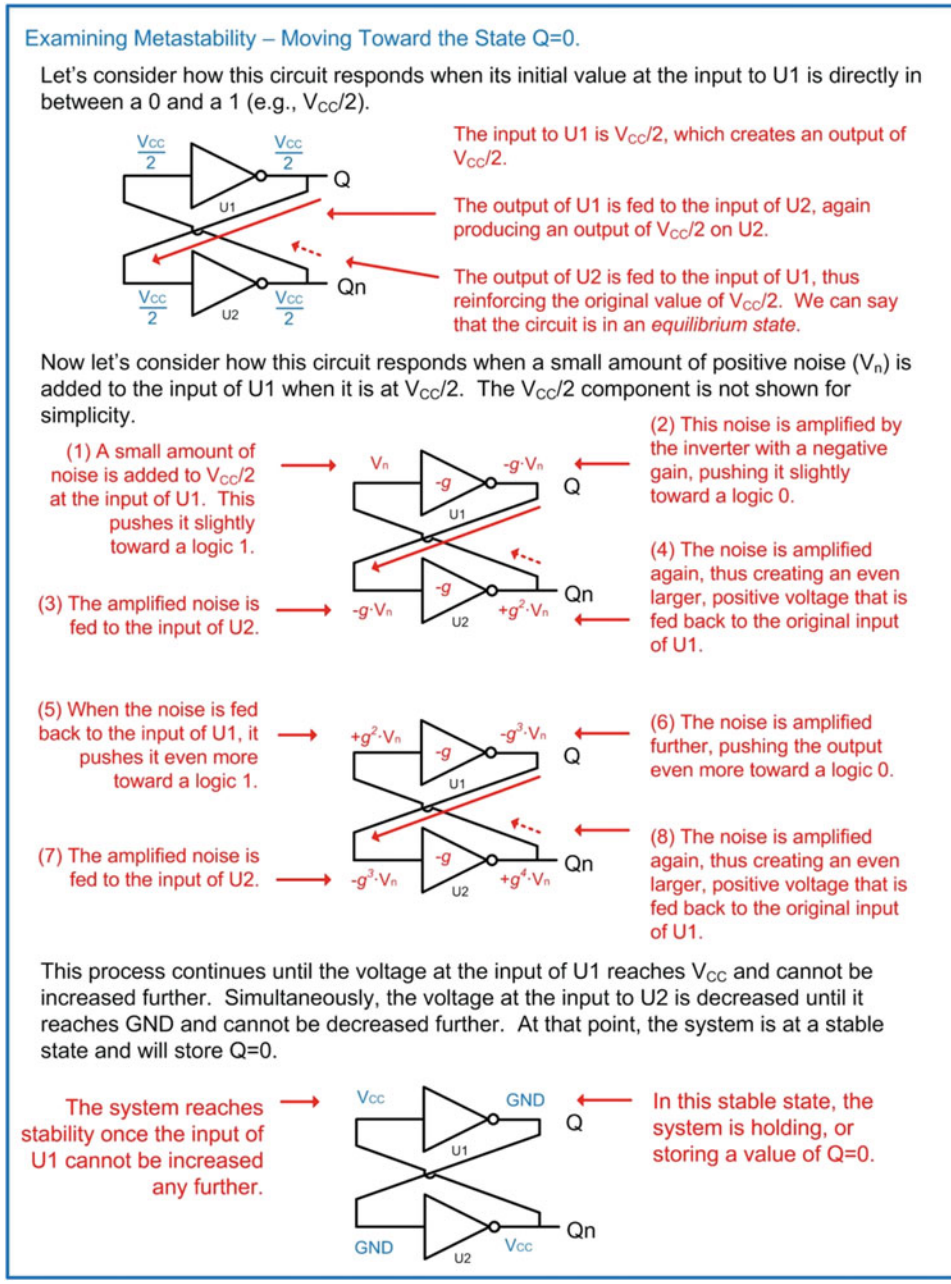


Fig. 7.2
Examining metastability moving toward the state Q = 0

Figure 7.3 shows how the system responds when a small negative voltage noise ($-V_n$) is added to the $V_{CC}/2$ input on U1.

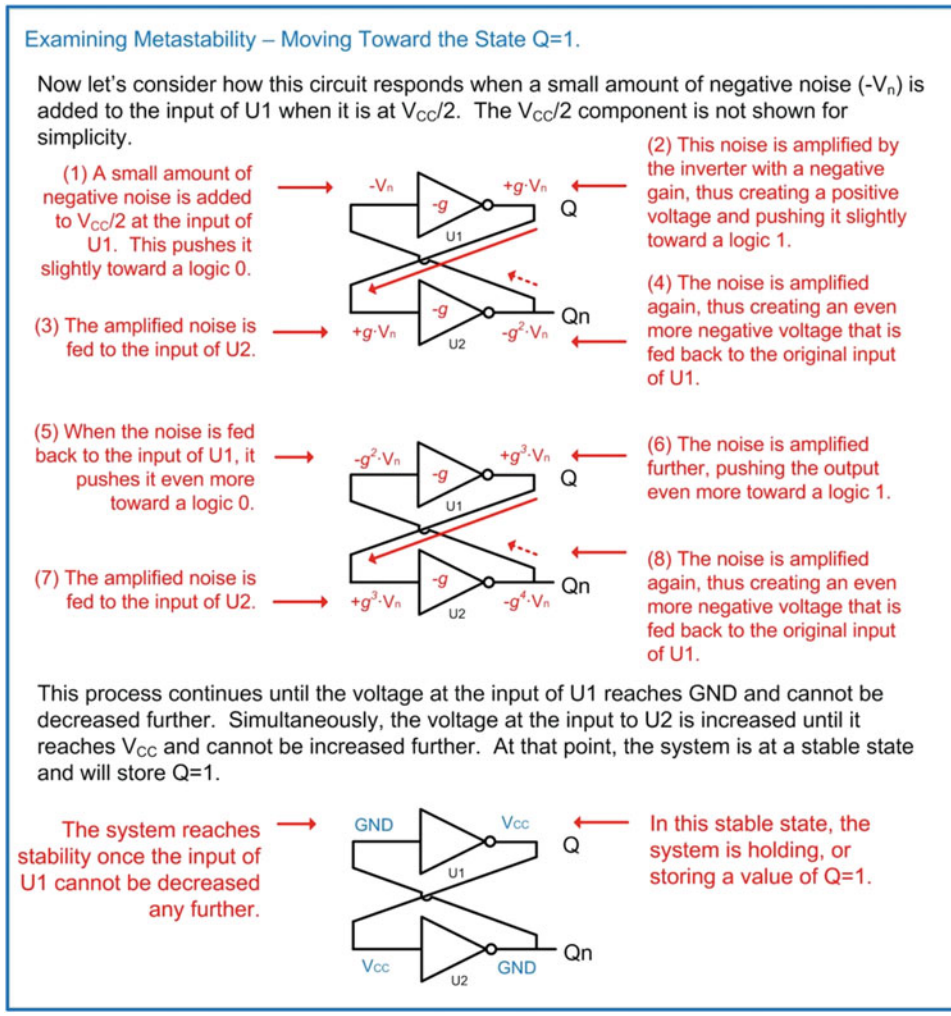


Fig. 7.3
Examining metastability moving toward the state Q = 1

7.1.3 The SR Latch

While the cross-coupled inverter pair is the fundamental storage concept for sequential logic, there is no mechanism to set the initial value of Q. All that is guaranteed is that the circuit will store a value in one of two stable states (Q = 0 or Q = 1). The *SR Latch* provides a means to control the initial values in this positive feedback configuration by replacing the inverters with NOR gates. In this circuit, S stands for *set* and indicates when the output is forced to a logic 1 (Q = 1), and R stands for *reset* and indicates when the output is forced to a logic 0 (Q = 0). When both S = 0 and R = 0, the SR *Latch* is put into *store* mode, and it will hold the last value of Q. In all of these input conditions, Qn is the complement of Q. Consider the behavior of the SR *Latch* during its store state, as shown in Fig. 7.4.

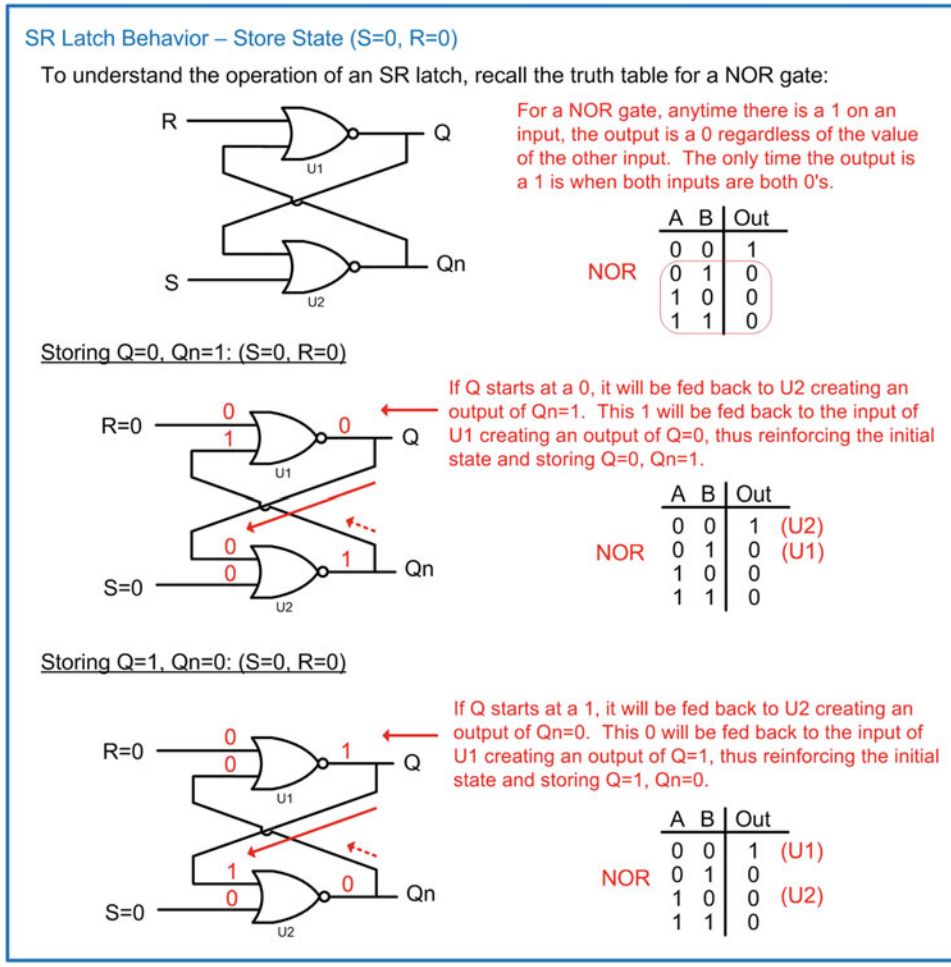


Fig. 7.4
SR latch behavior – Store state (S = 0, R = 0)

The SR *Latch* has two input conditions that will force the outputs to known values. The first condition is called the *set* state. In this state, the inputs are configured as S = 1 and R = 0. This input condition will force the outputs to Q = 1 (e.g., setting Q) and Qn = 0. The second input condition is called the *reset* state. In this state, the inputs are configured as S = 0 and R = 1. This input condition will force the outputs to Q = 0 (i.e., resetting Q) and Qn = 1. Consider the behavior of the SR *Latch* during its set and reset states, as shown in Fig. 7.5.

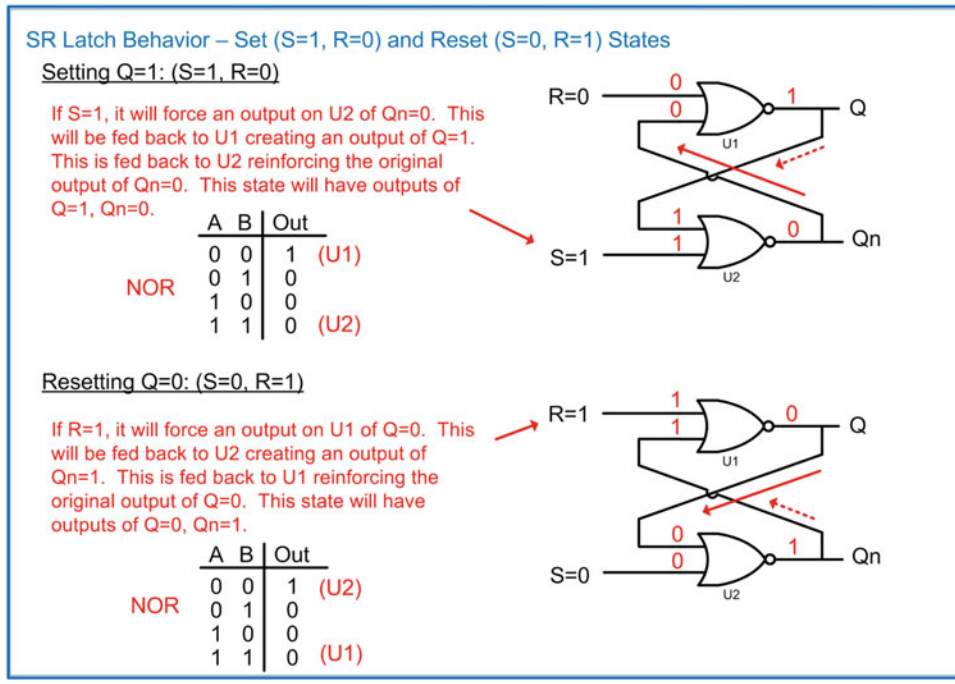


Fig. 7.5
SR latch behavior – Set (S = 1, R = 0) and reset (S = 0, R = 1) states

The final input condition for the SR *Latch* leads to potential metastability and should be avoided. When S = 1 and R = 1, the outputs of the SR *Latch* will both go to logic 0's. The problem with this state is that if the inputs subsequently change to the store state (S = 0, R = 0), the outputs will go metastable and then settle in one of the two stable states (Q = 0 or Q = 1). The reason this state is avoided is because the final resting state of the SR *Latch* is random and unknown. Consider this operation shown in Fig. 7.6.

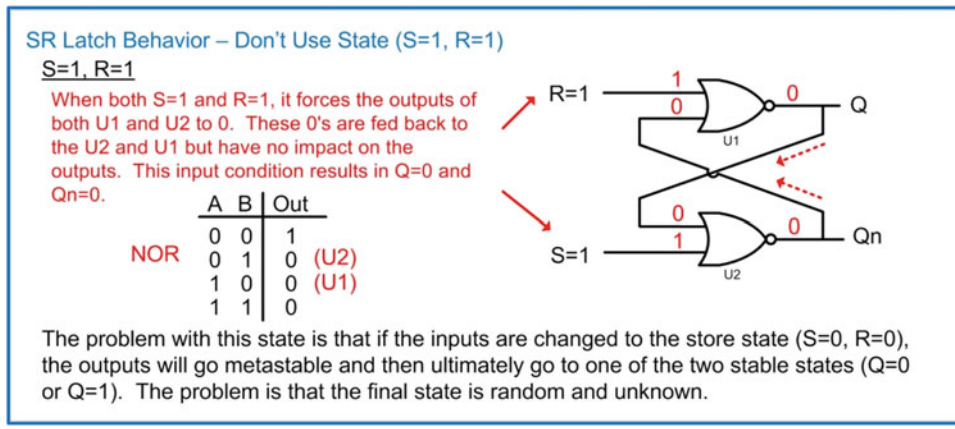


Fig. 7.6
SR latch behavior – Don't use state (S = 1 and R = 1)

Figure 7.7 shows the final truth table for the SR *Latch*.

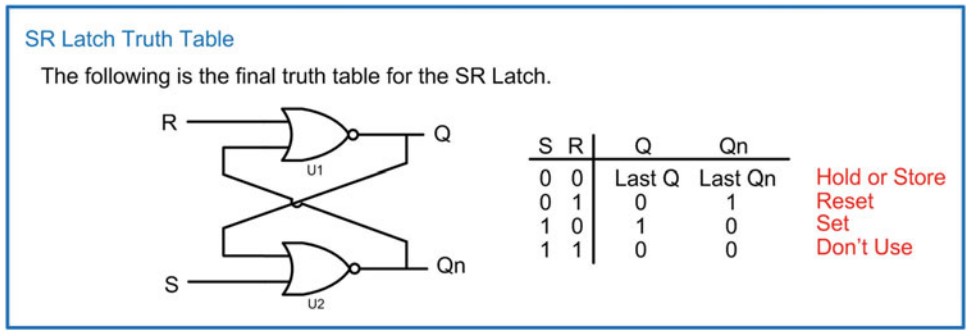


Fig. 7.7
SR latch truth table

The SR *Latch* has some drawbacks when it comes to implementation with real circuitry. First, it takes two independent inputs to control the outputs. Second, the state where S = 1 and R = 1 causes problems when real propagation delays are considered through the gates. Since it is impossible to match the delays exactly between U1 and U2, the SR *Latch* may occasionally enter this state and experience momentary metastable behavior. In order to address these issues, a number of improvements can be made to this circuit to create two of the most commonly used storage devices in sequential logic, the *D-Latch* and the *D-Flip-Flop*. In order to understand the operation of these storage devices, two incremental modifications are made to the SR Latch. The first is called the *S'R' Latch,* and the second is the *SR Latch with enable*. These two circuits are rarely implemented and are only explained to understand how the SR Latch is modified to create a D-Latch and ultimately a D-Flip-Flop.

7.1.4 The S'R' Latch

The S'R' Latch operates in a similar manner as the SR Latch with the exception that the input codes corresponding to the store, set, and reset states are complemented. To accomplish this complementary behavior, the S'R' Latch is implemented with NAND gates configured in a positive feedback configuration. In this configuration, the S'R' Latch will store the last output when S' = 1, R' = 1. It will set the output (Q = 1) when S' = 0 and R' = 1. Finally, it will reset the output (Q = 0) when S' = 1, R' = 0. Consider the behavior of the S'R' *Latch* during its store state, as shown in Fig. 7.8.

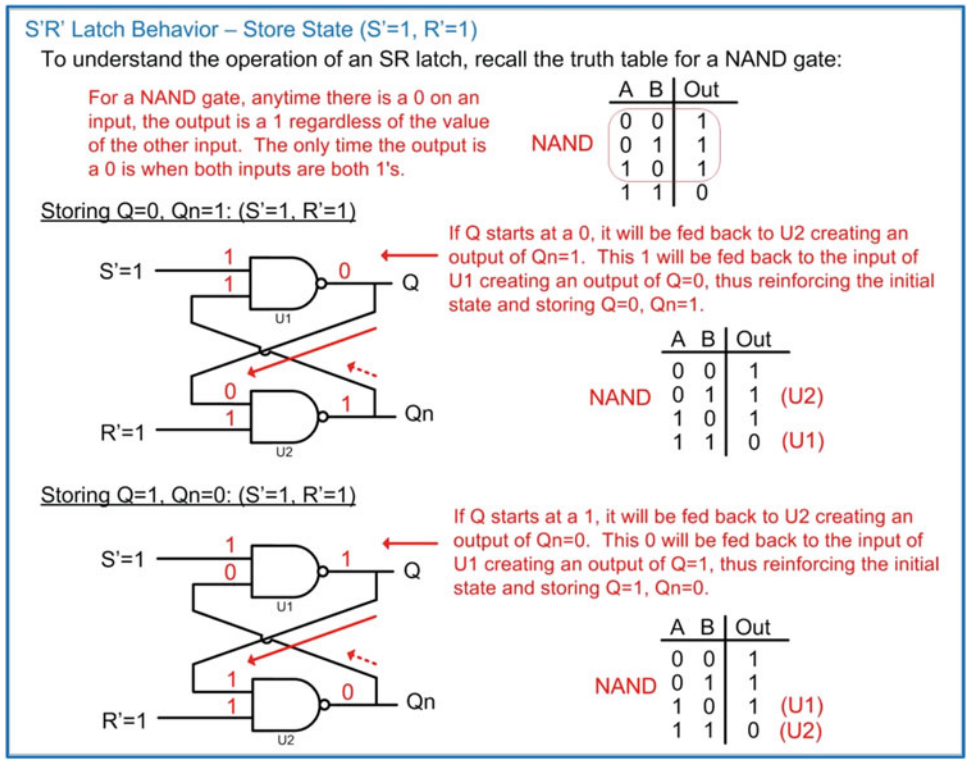


Fig. 7.8
S'R' latch behavior – Store state (S' = 1, R' = 1)

Just as with the SR *Latch*, the S'R' *Latch* has two input configurations to control the values of the outputs. Consider the behavior of the S'R' *Latch* during its set and reset states, as shown in Fig. 7.9.

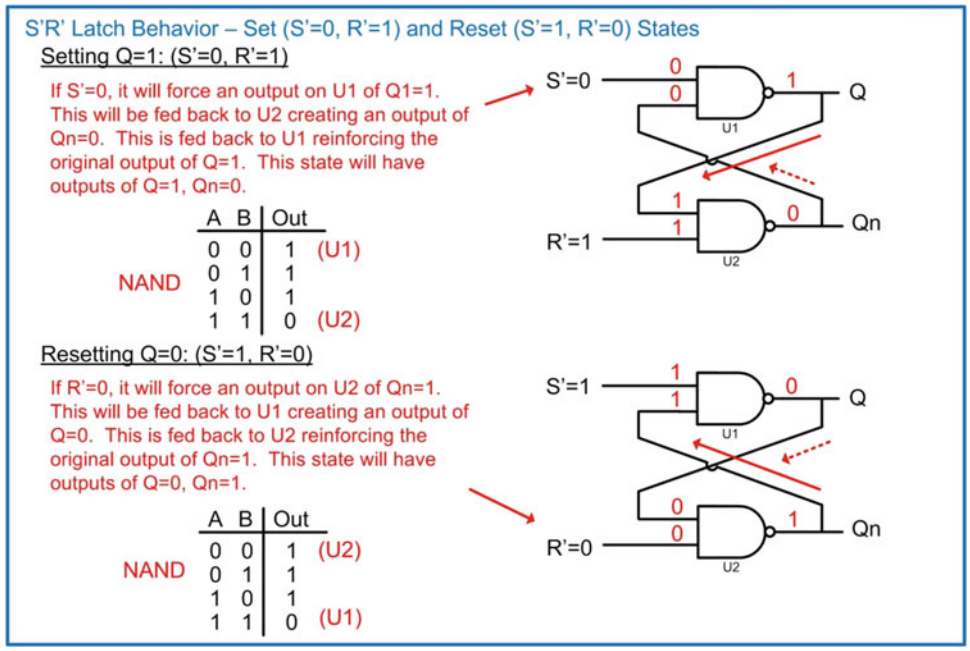


Fig. 7.9
S'R' latch behavior – Set (S' = 0, R' = 1) and reset (S' = 1, R' = 0) states

And finally, just as with the SR *Latch*, the S'R' *Latch* has a state that leads to potential metastability and should be avoided. Consider the operation of the S'R' *Latch* when the inputs are configured as S' = 0 and R' = 0, as shown in Fig. 7.10.

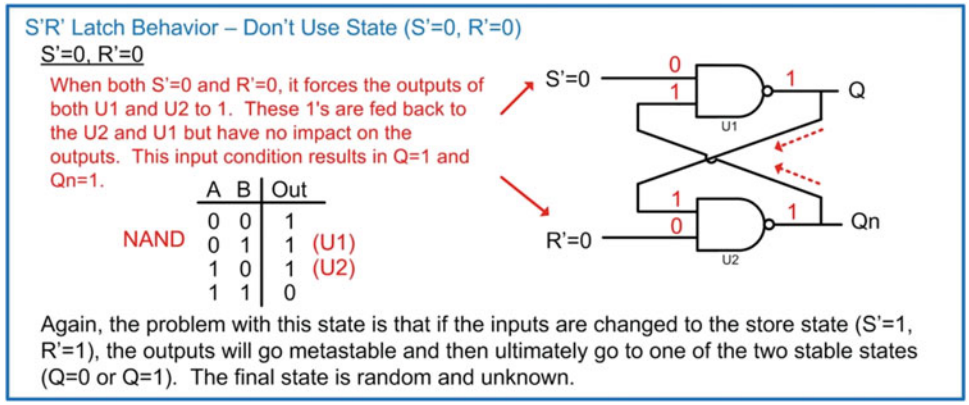


Fig. 7.10
S'R' latch behavior – Don't use state (S' = 0 and R' = 0)

The final truth table for the S'R' *Latch* is given in Fig. 7.11.

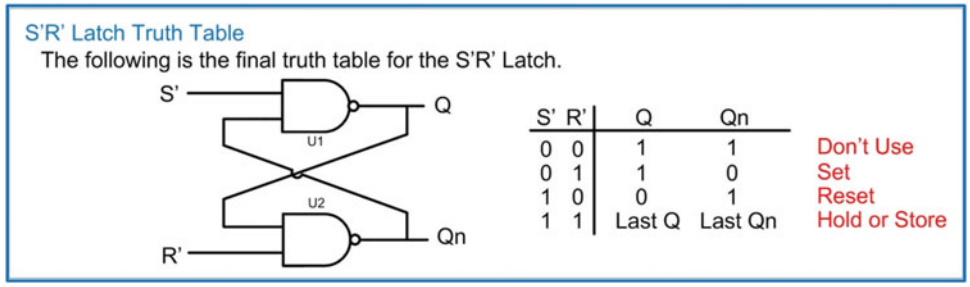


Fig. 7.11
S'R' latch truth table

7.1.5 SR Latch with Enable

The next modification that is made in order to move toward a D-Latch and ultimately a D-Flip-Flop is to add an *enable* line to the S'R' *Latch*. The enable is implemented by adding two NAND gates on the input stage of the S'R' *Latch*. The SR *Latch* with enable is shown in Fig. 7.12. In this topology, the use of NAND gates changes the polarity of the inputs, so this circuit once again has a set state where S = 1, R = 0, and a reset state of S = 0 and R = 1. The enable line is labeled *C*, which stands for *clock*. The rationale for this will be demonstrated upon moving through the explanation of the D-Latch.

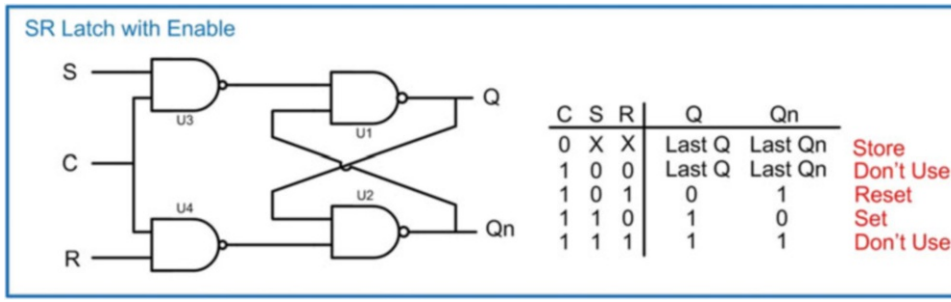


Fig. 7.12
SR latch with enable schematic

Recall that any time a 0 is present on one of the inputs to a NAND gate, the output will always be a 1, regardless of the value of the other inputs. In the SR *Latch* with enable configuration, any time C = 0, the outputs of U3 and U4 will be 1's and will be fed into the inputs of the cross-coupled NAND gate configuration (U1 and U2). Recall that the cross-coupled configuration of U1 and U2 is an S'R' *Latch* and will be put into a store state when S' = 1 and R' = 1. This is the *store state* (C = 0). When C = 1, it has the effect of inverting the values of the S and R inputs before they reach U1 and U2. This condition allows the *set state* to be entered when S = 1, R = 0, and C = 1, and the *reset state* to be entered when S = 0, R = 1, and C = 1. Consider this operation in Fig. 7.13.

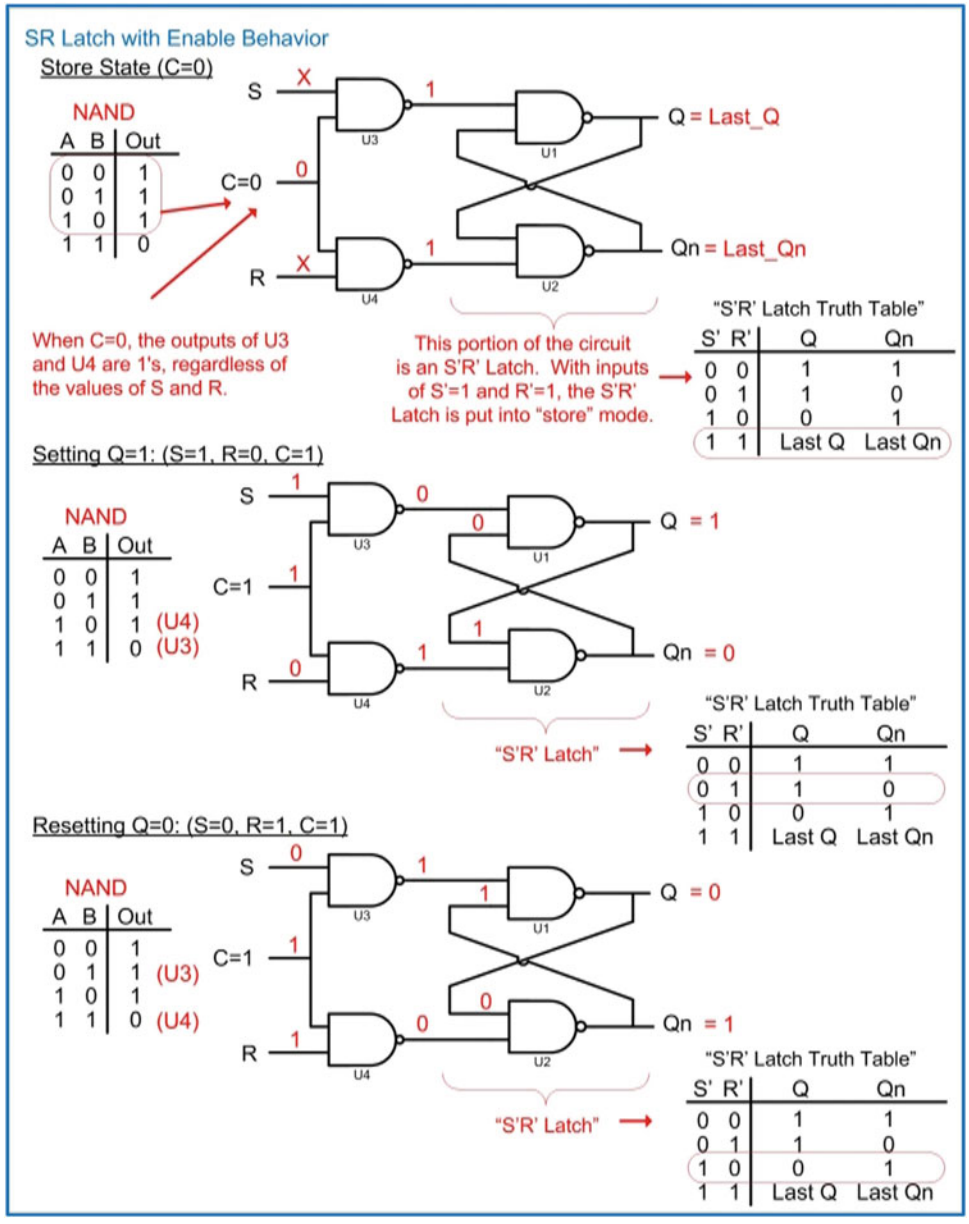


Fig. 7.13
SR latch with enable behavior – Store, set, and reset

Again, there is a potential metastable state when S = 1, R = 1, and C = 1 that should be avoided. There is also a second store state when S = 0, R = 0, and C = 1 that is not used because storage is to be dictated by the C input.

7.1.6 The D-Latch

The SR *Latch* with enable can be modified to create a new storage device called a D-Latch. Instead of having two separate input lines to control the outputs of the latch, the R input of the latch is instead driven by an inverted version of the S input. This prevents the S and R inputs from ever being the same value and removes the two "Don't Use" states in the truth table shown in Fig. 7.12. The new single input is renamed D to stand for *data*. This new circuit still has the behavior of storing the last values of Q and Qn when C = 0. When C = 1, the output will be Q = 1, when D = 1 and will be Q = 0 when D = 0. The behavior of the output when C = 1 is called *tracking* the input. The D-Latch schematic, symbol, and truth table are given in Fig. 7.14.

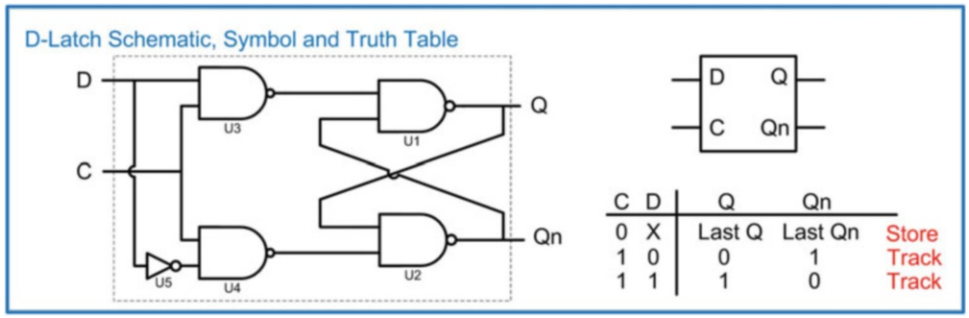


Fig. 7.14
D-latch schematic, symbol, and truth table

The timing diagram for the D-Latch is shown in Fig. 7.15.

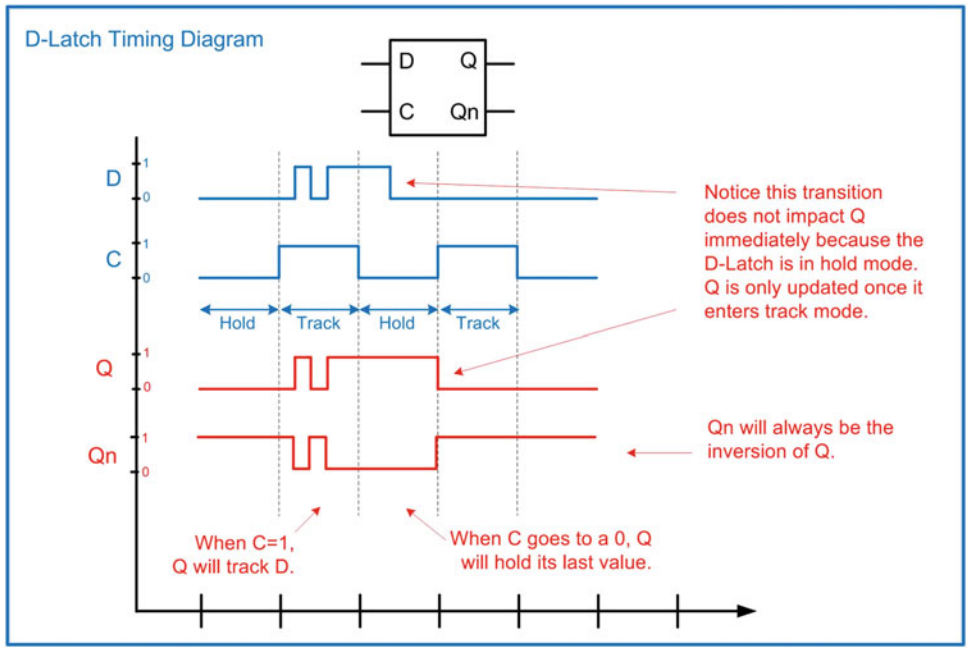


Fig. 7.15
D-latch timing diagram

7.1.7 The D-Flip-Flop

The final and most widely used storage device in sequential logic is the *D-Flip-Flop*. The D-Flip-Flop is similar in behavior to the D-Latch with the exception that the store mode is triggered by a transition, or edge, on the clock signal instead of a level. This allows the D-Flip-Flop to implement higher-frequency systems since the outputs are updated in a shorter amount of time. The schematic, symbol, and truth table are given in Fig. 7.16 for a rising edge-triggered D-Flip-Flop. To indicate that the device is edge-sensitive, the input for the clock is designated with a ">." The U3 inverter in this schematic creates the rising edge behavior. If U3 is omitted, this circuit would be a negative edge-triggered D-Flip-Flop.

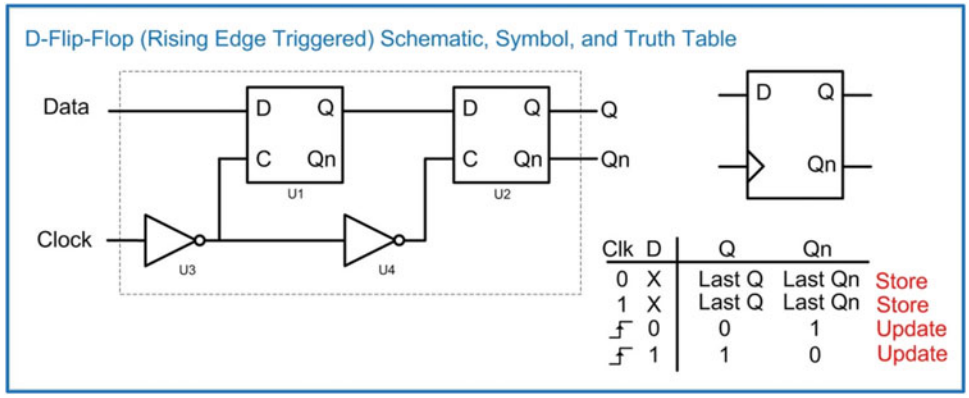


Fig. 7.16
D-Flip-Flop (rising edge triggered) schematic, symbol, and truth table

The D-Flip-Flop schematic shown above is called a *master/slave* configuration because of how the data is passed through the two D-Latches (U1 and U2). Due to the U4 inverter, the two D-Latches will always be in complementary modes. When U1 is in hold mode, U2 will be in track mode, and vice versa. When the clock signal transitions HIGH, U1 will store the last value of data. During the time when the clock is HIGH, U2 will enter track mode and pass this value to Q. In this way, the data are latched into the storage device on the rising edge of the clock and is present on Q. This is the *master* operation of the device because U1, or the first D-Latch, is holding the value, and the second D-Latch (the *slave*) is simply passing this value to the output Q. When the clock transitions LOW, U2 will store the output of U1. Since there is a finite delay through U1, the U2 D-Latch is able to store the value before U1 fully enters track mode. U2 will drive Q for the duration of the time that the clock is LOW. This is the *slave* operation of the device because U2, or the second D-Latch, is holding the value. During the time the clock is LOW, U1 is in track mode, which passes the input data to the middle of the D-Flip-Flop, preparing for the next rising edge of the clock. The master/slave configuration creates a behavior where the Q output of the D-Flip-Flop is only updated with the value of D on a rising edge of the clock. At all other times, Q holds the last value of D. An example timing diagram for the operation of a rising edge D-Flip-Flop is given in Fig. 7.17.

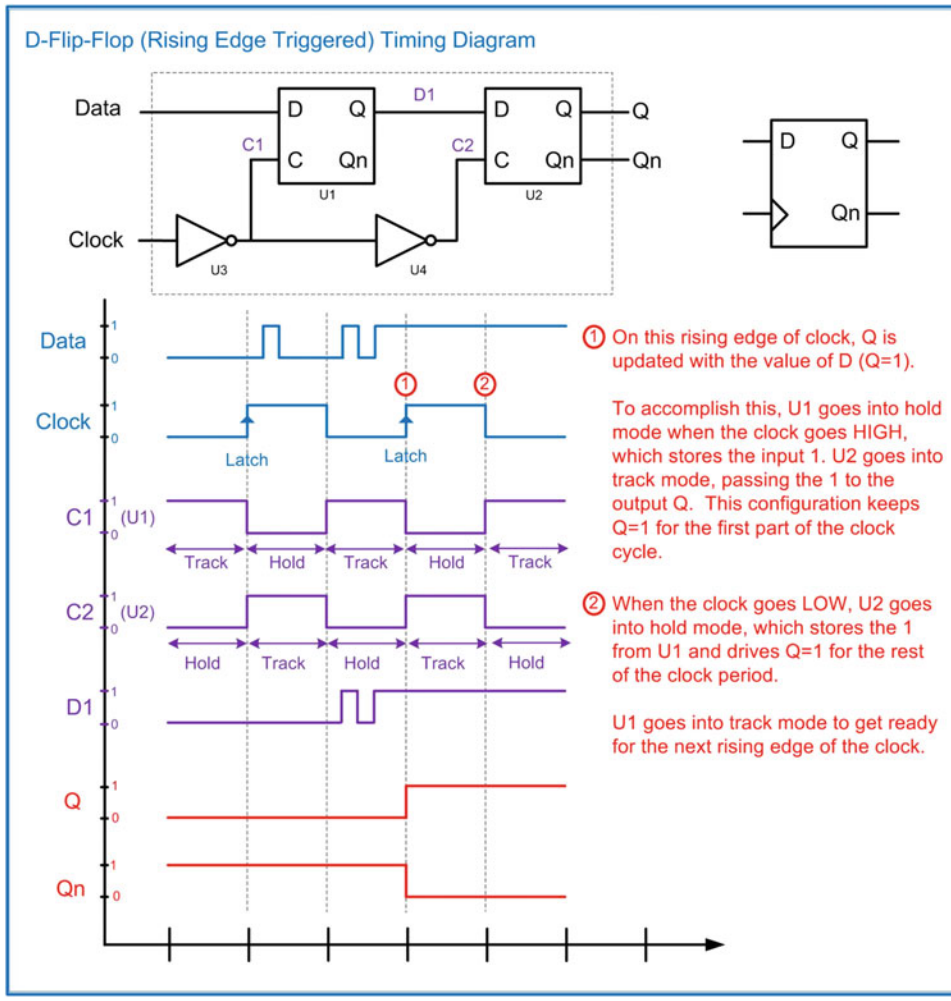
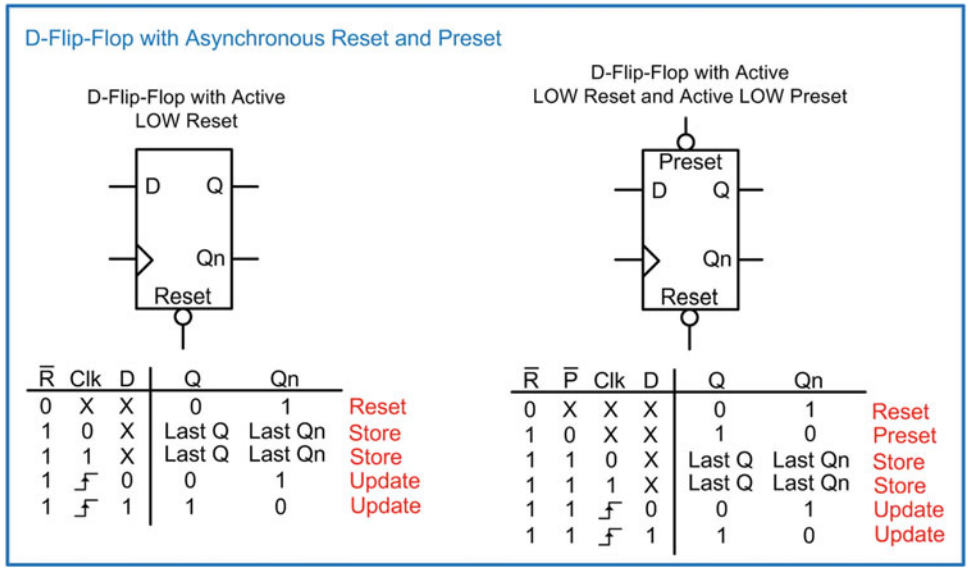


Fig. 7.17
D-Flip-Flop (rising edge triggered) timing diagram

D-Flip-Flops often have additional signals that will set the initial conditions of the outputs that are separate from the clock. A *reset* input is used to force the outputs to Q = 0, Qn = 1. A *preset* input is used to force the outputs to Q = 1, Qn = 0. In most modern D-Flip-Flops, these inputs are active LOW, meaning that the line is asserted when the input is a 0. Active LOW inputs are indicated by placing an inversion bubble on the input pin of the symbol. These lines are typically *asynchronous*, meaning that when they are asserted, action is immediately taken to alter the outputs. This is different from a *synchronous* input, in which action is only taken on the edge of the clock. Figure 7.18 shows the symbols and truth tables for two D-Flip-Flop variants, one with an active LOW reset and another with both an active LOW reset and an active LOW preset.

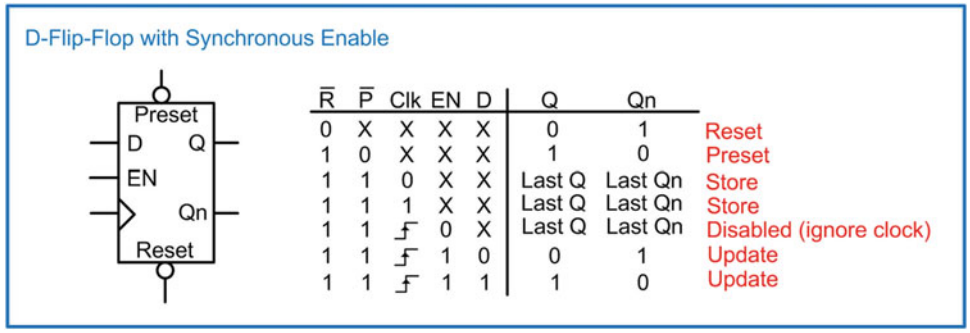


$\overline{R}$	Clk	D	Q	Qn	
0	X	X	0	1	Reset
1	0	X	Last Q	Last Qn	Store
1	1	X	Last Q	Last Qn	Store
1	↓	0	0	1	Update
1	↓	1	1	0	Update

$\overline{R}$	$\overline{P}$	Clk	D	Q	Qn	
0	X	X	X	0	1	Reset
1	0	X	X	1	0	Preset
1	1	0	X	Last Q	Last Qn	Store
1	1	1	X	Last Q	Last Qn	Store
1	1	↓	0	0	1	Update
1	1	↓	1	1	0	Update

Fig. 7.18
D-Flip-Flop with asynchronous reset and preset

D-Flip-Flops can also be created with an *enable* line. An enable line controls whether or not the output is updated. Enable lines are synchronous, meaning that when they are asserted, the outputs will be updated on the rising edge of the clock. When de-asserted, the outputs are not updated. This behavior, in effect, ignores the clock input when de-asserted. Figure 7.19 shows the symbol and truth table for a D-Flip-Flop with a synchronous enable.



$\overline{R}$	$\overline{P}$	Clk	EN	D	Q	Qn	
0	X	X	X	X	0	1	Reset
1	0	X	X	X	1	0	Preset
1	1	0	X	X	Last Q	Last Qn	Store
1	1	1	X	X	Last Q	Last Qn	Store
1	1	↓	0	X	Last Q	Last Qn	Disabled (ignore clock)
1	1	↓	1	0	0	1	Update
1	1	↓	1	1	1	0	Update

Fig. 7.19
D-Flip-Flop with synchronous enable

The behavior of the D-Flip-Flop allows us to design systems that are *synchronous* to a clock signal. A clock signal is a periodic square wave that dictates when events occur in a digital system. A synchronous system based on D-Flip-Flops will allow the outputs of its storage devices to be updated upon a rising edge of the clock. This is advantageous because when the Q outputs are storing values, they can be used as inputs for combinational logic circuits. Since combinational logic circuits contain a certain amount of propagation delay before the final output is calculated, the D-Flip-Flop can hold the inputs at a steady value while the output is generated. Since the input on a D-Flip-Flop is ignored at all

other times, the output of a combinational logic circuit can be fed back as an input to a D-Flip-Flop. This gives a system the ability to generate outputs based on the current values of inputs in addition to the past values of the inputs that are being held on the outputs of D-Flip-Flops. This is the definition of sequential logic. An example of a synchronous, sequential system is shown in Fig. 7.20.

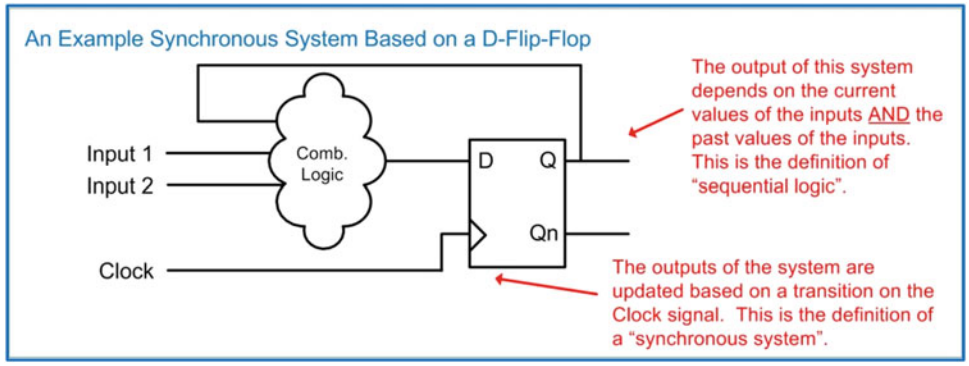


Fig. 7.20
An example synchronous system based on a D-Flip-Flop

Concept Check

CC7.1(a) What will always cause a digital storage device to come out of metastability and settle in one of its two stable states? Why?

A) The power supply: The power supply provides the necessary current for the device to overcome metastability.

B) Electrical noise: Noise will always push the storage device toward one state or another. Once the storage device starts moving toward one of its stable states, the positive feedback of the storage device will reinforce the transition until the output eventually comes to rest in a stable state.

C) A reset: A reset will put the device into a known stable state.

D) A rising edge of the clock: The clock also puts the device into a known stable state.

CC7.1(b) What was the purpose of replacing the inverters in the cross-coupled inverter pair with NOR gates to form the SR *Latch*?

A) NOR gates are easier to implement in CMOS.

B) To provide the additional output Qn.

C) To provide more drive strength for storage.

D) To provide inputs to explicitly set the value being stored.

7.2 Sequential Logic Timing Considerations

There are a variety of timing specifications that need to be met in order to successfully design circuits using sequential storage devices. The first specification is called the *setup time* (t_{setup} or t_s). The setup time specifies how long the data input needs to be in a steady state *before* the clock event. The second specification is called the *hold time* (t_{hold} or t_h). The hold time specifies how long the data input needs to be in a steady state *after* the clock event. If these specifications are violated (i.e., the input transitions too close to the clock transition), the storage device will not be able to determine whether the input was a 1 or 0 and will go metastable. The time a storage device will remain metastable is a deterministic value and is specified by the part manufacturer (t_{meta}). In general, metastability should be avoided; however, knowing the maximum duration of metastability for a storage device allows us to design circuits to overcome potential metastable conditions. During the time the device is metastable, the output will have random behavior. It may go to a steady state 1, a steady state 0, or toggle between 0 and 1 uncontrollably. Once the device comes out of metastability, it will come to rest in one of its two stable states (Q = 0 or Q = 1). The final resting state is random and unknown. Another specification for sequential storage devices is the delay from the time a clock transition occurs to the point that the data is present on the Q output. This specification is called the *Clock-to-Q* delay and is given the notation t_{CQ}. These specifications are shown in Fig. 7.21.

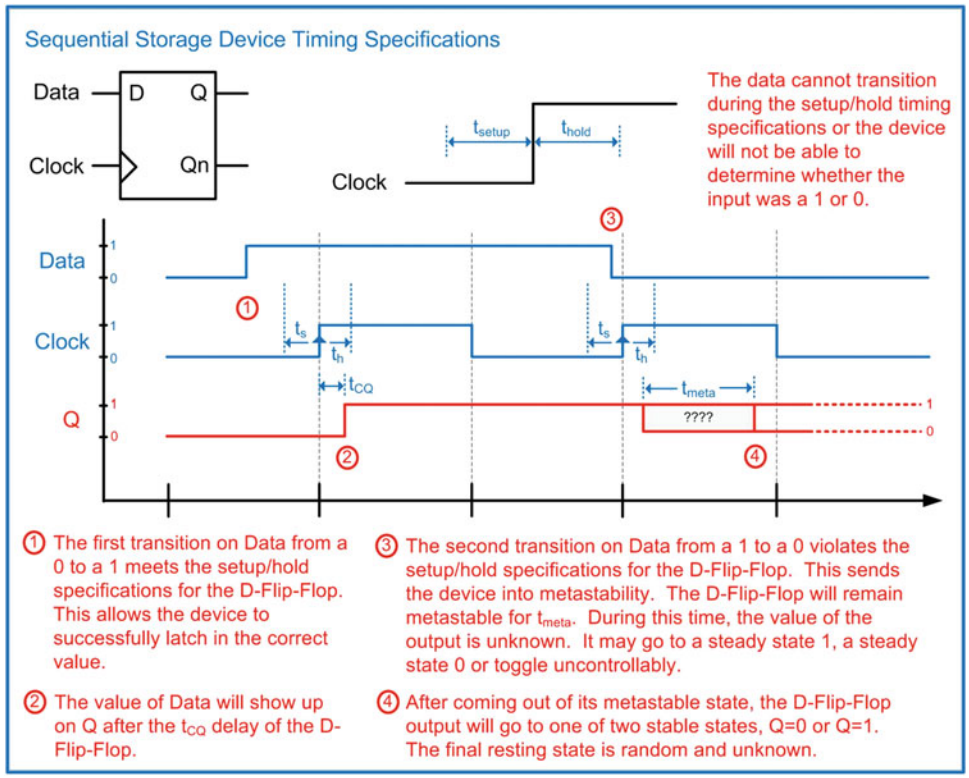


Fig. 7.21
Sequential storage device timing specifications

Concept Check

CC7.2 Which D-flop-flop timing specification requires all combinational logic circuits in the system to settle on their final output before a triggering clock edge can occur?

A) t_{setup} B) t_{hold} C) t_{CQ} D) t_{meta}

7.3 Common Circuits Based on Sequential Storage Devices

Sequential logic storage devices give us the ability to create sophisticated circuits that can make decisions based on the current and past values of the inputs; however, there are a variety of simple yet useful circuits that can be created with only these storage devices. This section will introduce a few of these circuits.

7.3.1 Toggle Flop Clock Divider

A *Toggle Flop* is a circuit that contains a D-Flip-Flop configured with its Qn output wired back to its D input. This configuration is also commonly referred to as a *T-Flip-Flop* or *T-Flop*. In this circuit, the only input is the clock signal. Let's examine the behavior of this circuit when its outputs are initialized to Q = 0, Qn = 1. Since Qn is wired to the D input, a logic 1 is present on the input before the first clock edge. Upon the rising edge of the clock, Q is updated with the value of D. This puts the outputs at Q = 1, Qn = 0. With these outputs, now a logic 0 is present on the input before the next clock edge. Upon the next rising edge of the clock, Q is updated with the value of D. This time the outputs go to Q = 0, Qn = 1. This behavior continues indefinitely. The circuit is called a toggle flop because the outputs simply toggle between 0 and 1 every time there is a rising edge of the clock. This configuration produces outputs that are square waves with exactly half the frequency of the incoming clock. As a result, this circuit is also called a *clock divider*. This circuit can be given its own symbol with a label of "T," indicating it is a toggle flop. The configuration of a Toggle Flop (T-Flop) and timing diagram are shown in Fig. 7.22.

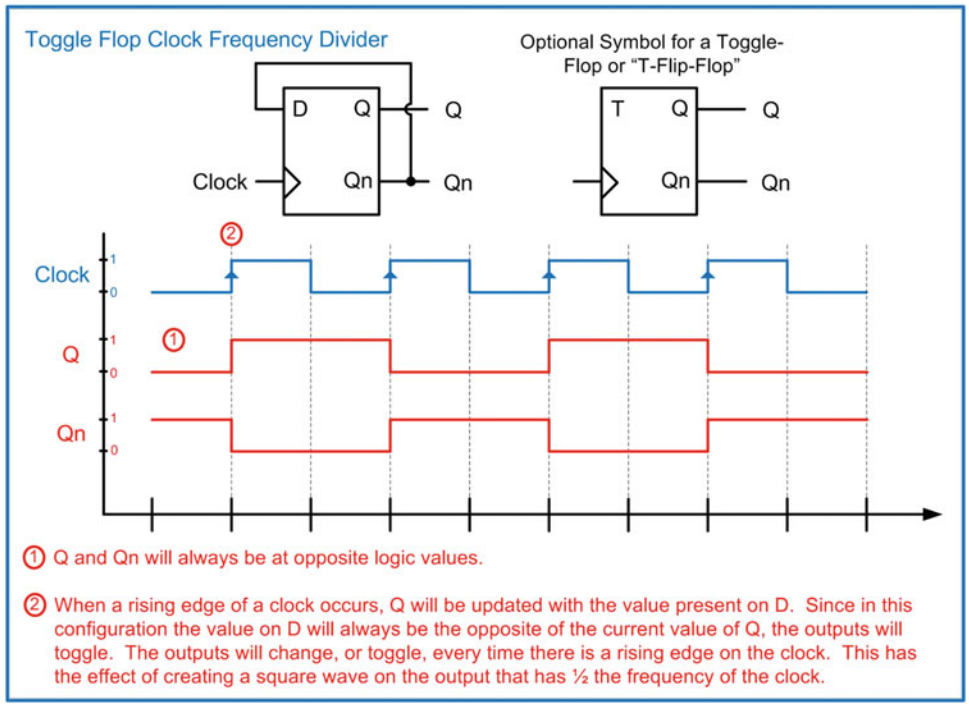


Fig. 7.22
Toggle flop clock frequency divider

7.3.2 Ripple Counter

The toggle flop configuration can be used to create a simple binary counter called a *ripple counter*. In this configuration, the Qn output of a toggle flop is used as the clock for a subsequent toggle flop. Since the output of the first toggle flop is a square wave that is ½ the frequency of the incoming clock, this configuration will produce an output on the second toggle flop that is ¼ the frequency of the incoming clock. This is, by nature, the behavior of a binary counter. The output of this counter is present on the Q pins of each toggle flop. Toggle flops are added until the desired width of the counter is achieved, with each toggle flop representing 1-bit of the counter. Since each toggle flop produces the clock for the subsequent latch, the clock is said to *ripple* through the circuit, hence the name ripple counter. A 3-bit ripple counter is shown in Fig. 7.23.

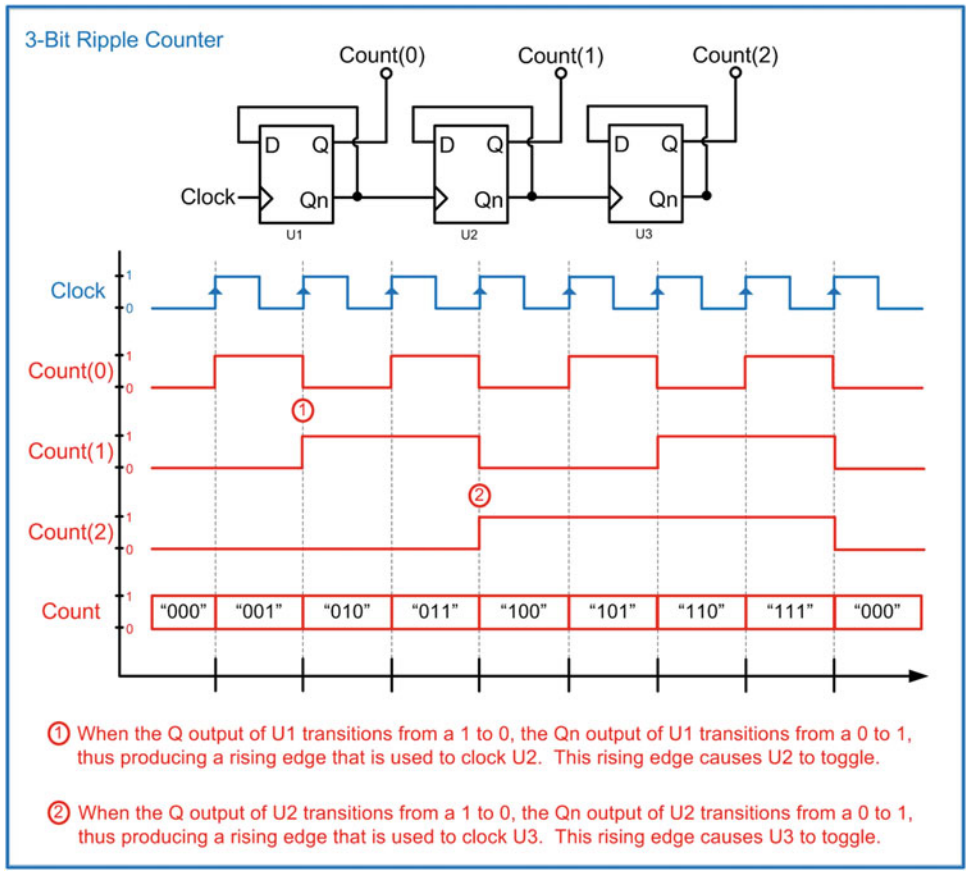


Fig. 7.23
3-Bit ripple counter

7.3.3 Switch Debouncing

Another useful circuit based on sequential storage devices is a switch debouncer. Mechanical switches have a well-known issue of not producing clean logic transitions on their outputs when pressed. This becomes problematic when using a switch to create an input for a digital device because it will cause unwanted logic level transitions on the output of the gate. In the case of a clock input, this unwanted transition can cause a storage device to unintentionally latch incorrect data.

The primary cause of these unclean logic transitions is due to the physical vibrations of the metal contacts when they collide with each other during a button press or switch actuation. Within a mechanical switch, there is typically one contact that is fixed and another that is designed to move when the button is pressed. The contact that is designed to move can be thought of as a beam that is fixed on one side and free on the other. As the free side of the beam moves toward the fixed contact in order to close the circuit, it will collide and then vibrate just as a tuning fork does when struck. The vibration will eventually diminish, and the contact will come to rest, thus making a clean electrical connection; however, during the vibration period, the moving contact will *bounce* up and down on the destination contact. This bouncing causes the switch to open and close multiple times before coming to rest in the closed position. This phenomenon is accurately referred to as *switch bounce*. Switch bounce is present in all mechanical switches and gets progressively worse as the switches are used more and more.

Figure 7.24 shows some of the common types of switches found in digital systems. The term *pole* is used to describe the number of separate circuits controlled by the switch. The term *throw* is used to describe the number of separate closed positions the switch can be in.

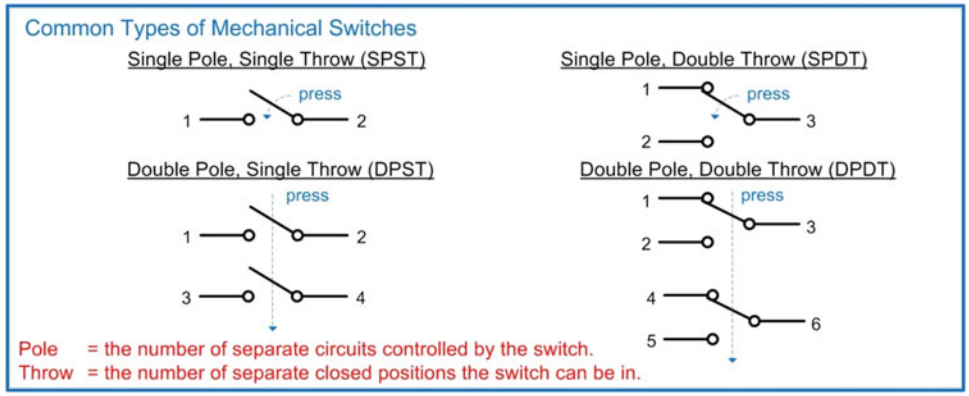


Fig. 7.24
Common types of mechanical switches

Let's look at switch bounce when using a single pole, single throw (SPST) switch to provide an input to a logic gate. An SPST requires a resistor and can be configured to provide either a logic HIGH or LOW when in the open position and the opposite logic level when in the closed position. The example configuration in Fig. 7.25 provides a logic LOW when in the open position and a logic HIGH when in the closed position. In the open position, the input to the gate (SW) is pulled to GND to create a logic LOW. In the closed position, the input to the gate is pulled to V_{CC} to create a logic HIGH. A resistor is necessary to prevent a short circuit between V_{CC} and GND when the switch is closed. Since the input current specification for a logic gate is very small, the voltage developed across the resistor due to the gate input current is negligible. This means that the resistor can be inserted into the pull-down network without developing a noticeable voltage. When the switch closes, the free-moving contact will bounce off of the destination contact numerous times before settling in the closed position. During the time the switch is bouncing, it will repeatedly toggle between the open (HIGH) and closed (LOW) positions.

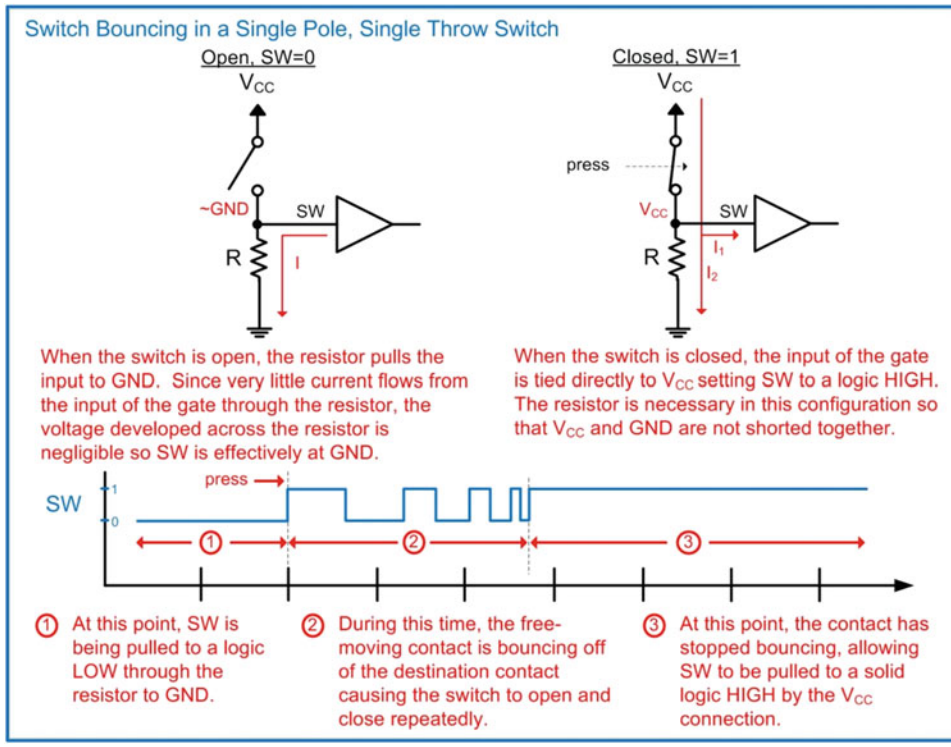


Fig. 7.25
Switch bouncing in a single pole, single throw switch

A possible solution to eliminate this switch bounce is to instead use a single pole, double throw (SPDT) switch in conjunction with a sequential storage device. Before looking at this solution, we need to examine an additional condition introduced by the SPDT switch. The SPDT switch has what is known as *break-before-make* behavior. The term *break* is used to describe when a switch is open, while the term *make* is used to describe when the switch is closed. When an SPDT switch is pressed, the input will be floating during the time when the free-moving contact is transitioning toward the destination contact. During this time, the output of the switch is unknown and can cause unwanted logic transitions if it is being used to drive the input of a logic gate.

Let's look at switch bounce when using an SPDT switch without additional circuitry to handle bouncing. An SPDT has two positions that the free-moving contact can make a connection to (i.e., double throw). When using this switch to drive a logic level into a gate, one position is configured as a logic HIGH and the other as a logic LOW. Consider the SPDT switch configuration in Fig. 7.26. Position 1 of the SPDT switch is connected to GND, while position 2 is connected to V_{CC}. When unpressed, the switch is in position 1. When pressed, the free-moving contact will transition from position 1 to 2. During the transition, the free-moving contact is floating. This creates a condition where the input to the gate (SW) is unknown. This floating input will cause unpredictable behavior on the output of the gate. Upon reaching position 2, the free-moving contact will bounce off of the destination contact. This will cause the input of the logic gate to toggle between a logic HIGH and floating repeatedly until the free-moving contact comes to rest in position 2.

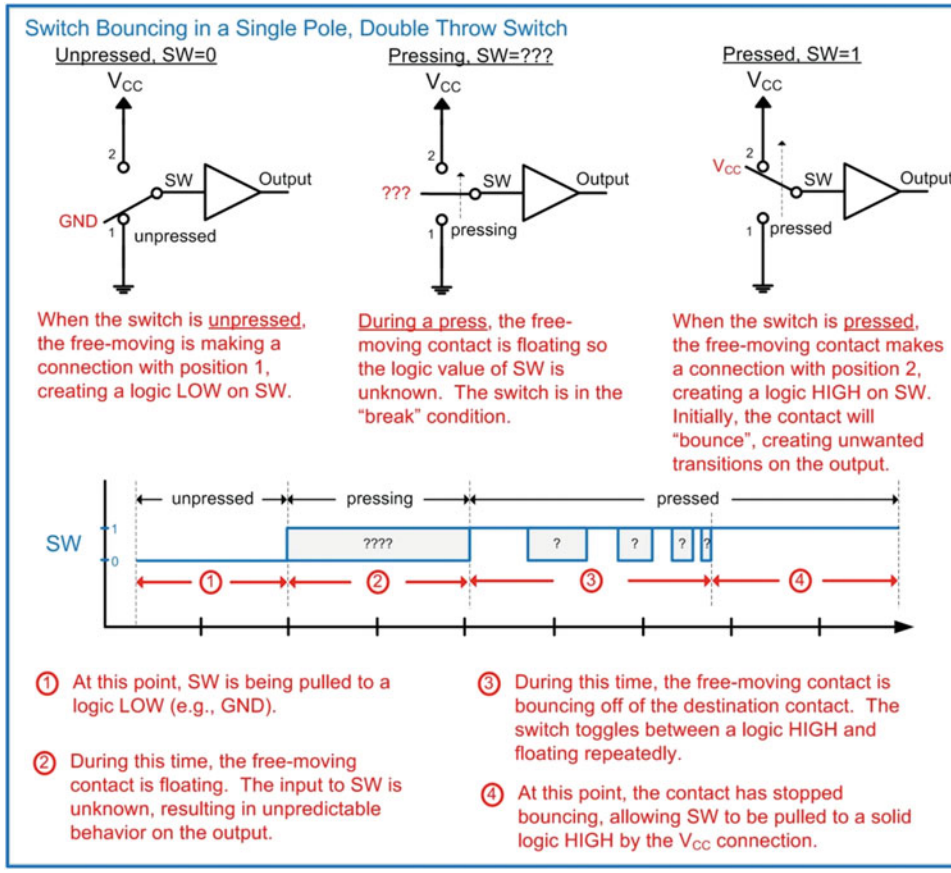


Fig. 7.26
Switch bouncing in a single pole, double throw switch

The SPDT switch is ideal for use with an S'R' *Latch* in order to produce a clean logic transition. This is because during the *break* portion of the transition, an S'R' *Latch* can be used to hold the last value of the switch. This is unique to the SPDT configuration. The SPST switch, in comparison, does not have the *break* characteristic; rather, it always drives a logic level in both of its possible positions. Consider the debounce circuit for an SPDT switch in Fig. 7.27. This circuit is based on an S'R' *Latch* with two pull-up resistors. Since the S'R' *Latch* is created using NAND gates, this circuit is commonly called a *NAND-Debounce* circuit. In the unpressed configuration, the switch drives S' = 0 and the R2 pull-up resistor drives R' = 1. This creates a logic 0 on the output of the circuit (Qn = 0). During a switch press, the free-moving contact is floating, thus it is not driving in a logic level into the S'R' Latch. Instead, both pull-up resistors pull S' and R' to 1's. This puts the latch into its hold mode, and the output will remain at a logic 0 (Qn = 0). Once the free-moving contact reaches the destination contact, the switch will drive R' = 0. Since at this point the R1 pull-up is driving S' = 1, the latch outputs a logic 1 (Qn = 1). When the free-moving contact bounces off of the destination contact, it will put the latch back into the hold mode; however, this time the last value that will be held is Qn = 1. As the switch continues to bounce, the latch will move between the Qn = 1 and Qn = "Last Qn" states, both of which produce an output of 1. In this way, the SPDT switch in conjunction with the S'R' *Latch* produces a clean 0 to 1 logic transition despite the break-before-make behavior of the switch and the contact bounce.

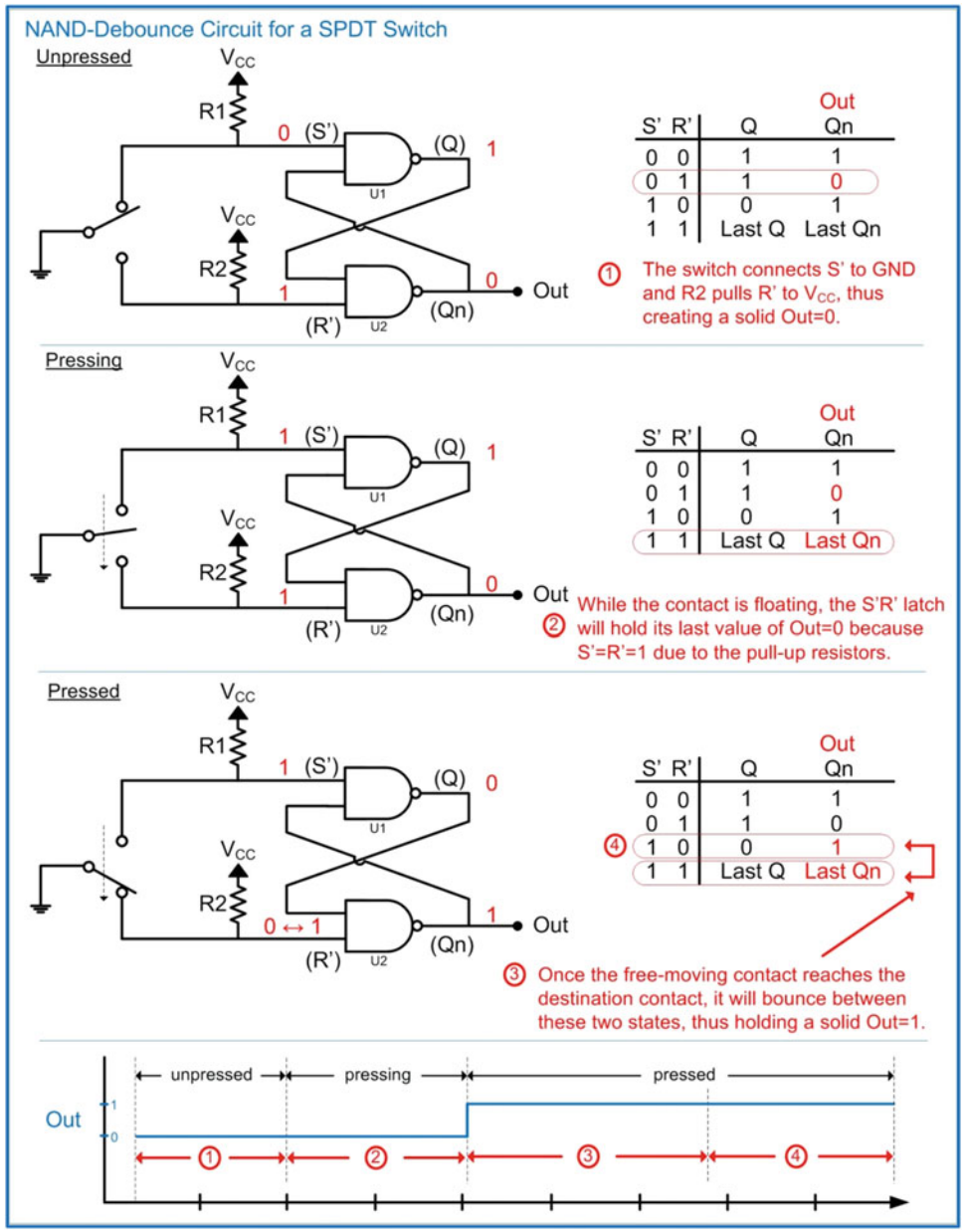


Fig. 7.27
NAND debounce circuit for an SPDT switch

7.3.4 Shift Registers

A *shift register* is a chain of D-Flip-Flops that are each connected to a common clock. The output of the first D-Flip-Flop is connected to the input of the second D-Flip-Flop. The output of the second D-Flip-Flop is connected to the input of the third D-Flip-Flop, and so on. When data is present on the input to the first D-Flip-Flop, it will be latched upon the first rising edge of the clock. On the second rising edge of the clock, the same data will be latched into the second D-Flip-Flop. This continues on each rising edge of the clock until the data has been *shifted* entirely through the chain of D-Flip-Flops. Shift registers are commonly used to convert a serial string of data into a parallel format. If an *n*-bit serial sequence of information is clocked into the shift register, after *n* clocks, the data will be held on each of the D-Flip-Flop

outputs. At this moment, the *n*-bits can be read as a parallel value. Consider the shift register configuration shown in Fig. 7.28.

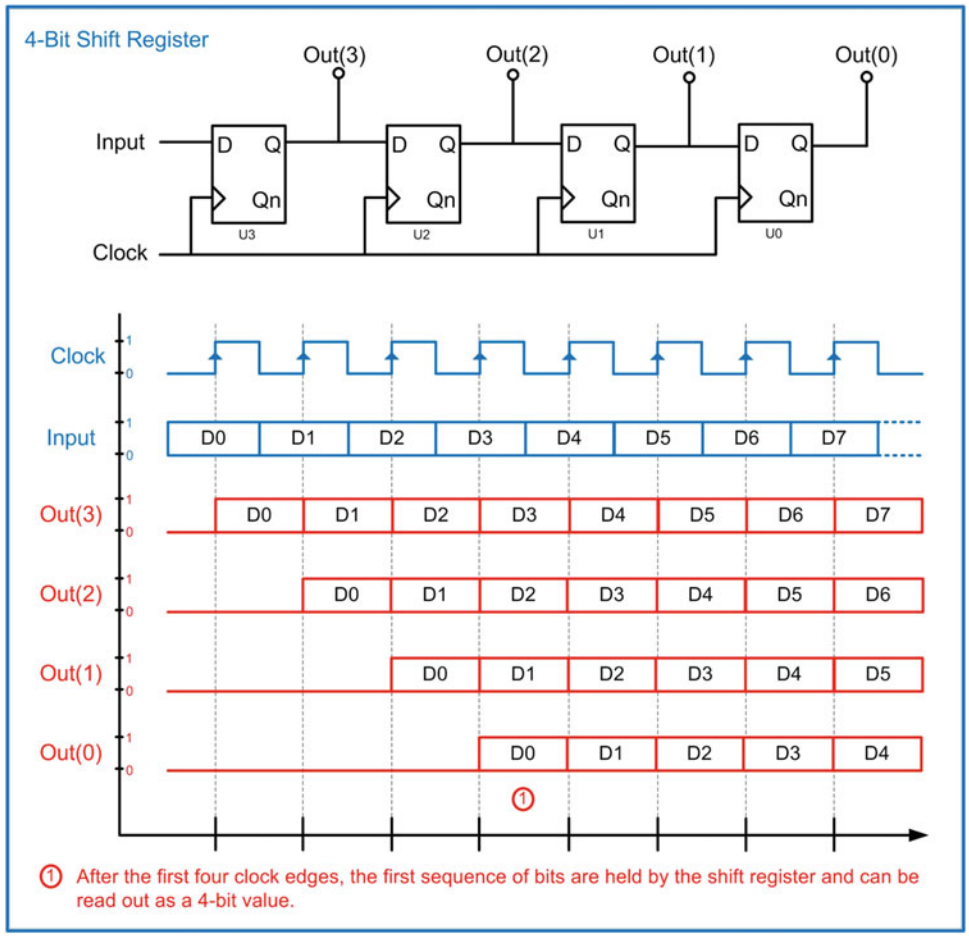


Fig. 7.28
4-Bit shift register

Concept Check

CC7.3 Which D-Flip-Flop timing specification is most responsible for the ripple delay in a ripple counter?

A) t_{setup} B) t_{hold} C) t_{CQ} D) t_{meta}

7.4 Finite-State Machines

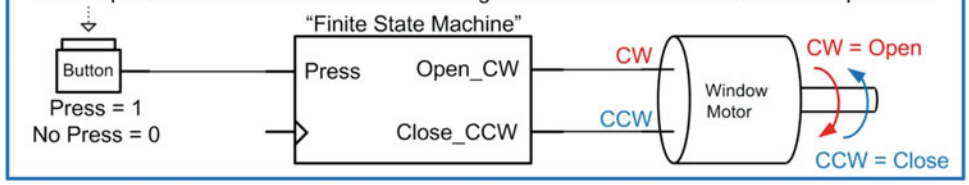
Now we turn our attention to one of the most powerful sequential logic circuits, the finite state machine (FSM). An FSM, or *state machine*, is a circuit that contains a pre-defined number of states (i.e., a finite number of states). The machine can exist in only one state at a time. The circuit *transitions* between states based on a triggering event, most commonly the edge of a clock, in addition to the values of any inputs of the machine. The number of states and all possible transitions are pre-defined. Through the use of states and a pre-defined sequence of transitions, the circuit is able to make decisions on the next state to transition to based on a history of past states. This allows the circuit to create outputs that are more intelligent compared to a simple combinational logic circuit that has outputs based only on the current values of the inputs.

7.4.1 Describing the Functionality of an FSM

The design of a state machine begins with an abstract word description of the desired circuit behavior. We will use a design example of a push-button motor controller to describe all of the steps involved in creating a FSM. Example 7.1 starts the FSM design process by stating the system's word description.

Example: Push-Button Window Controller - Word Description

Design a system that will allow a user to open and close a window with the push of a button. The window is connected to a motor that has two inputs. The first input to the motor is asserted when the motor needs to spin in a clockwise (CW) direction to open the window, while the second input is asserted when the motor needs to spin in a counterclockwise (CCW) direction to close the window. The signals to the motor do not need to be held for the duration of the window opening/closing. Once the motor observes an assertion on one of its inputs, it will spin until the window is in the correct position and then stop. The inputs are not allowed to be asserted at the same time. The user will press a single button to either open or close the window so the system must keep track of whether the window is in the open or closed position in order to send the correct signals to the motor when the button is pressed.



Example 7.1
Push-button window controller – Word description

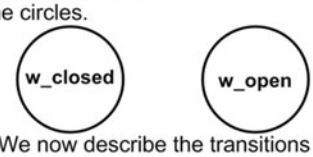
7.4.1.1 State Diagrams

A state diagram is a graphical way to describe the functionality of a FSM. A state diagram is a form of directed graph in which each state (or vertex) within the system is denoted as a circle and given a descriptive name. The names are written inside the circles. The transitions between states are denoted using arrows, with the input conditions causing the transitions written next to them. Transitions (or edges) can move to different states upon particular input conditions or remain in the same state. For a state machine implemented using sequential logic storage, an evaluation of when to transition states is triggered every time the storage devices update their outputs. For example, if the system was implemented using rising edge-triggered D-flip-Flops, then an evaluation would occur on every rising edge of the clock.

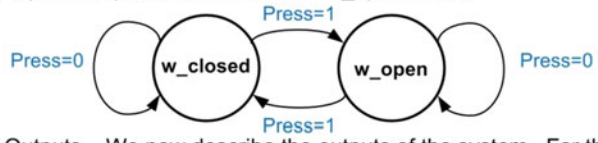
There are two different types of output conditions for a state machine. The first is when the output only depends on the current state of the machine. This type of system is called a *Moore Machine*. In this case, the outputs of the system are written inside the state circles. This indicates the output value that will be generated for each specific state. The second output condition is when the outputs depend on both the current state and the system inputs. This type of system is called a *Mealy Machine*. In this case, the outputs of the system are written next to the state transitions corresponding to the appropriate input values. Outputs in a state diagram are typically written inside parentheses. Example 7.2 shows the construction of the state diagram for our push-button window controller design.

Example: Push-Button Window Controller - State Diagram

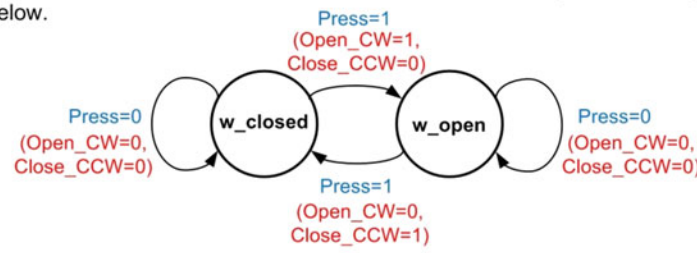
1) Defining the States - For this design, we will define two finite states. The first state is when the window is in the closed position. Let's call this state "w_closed". The second state is when the window is in the open position. Let's call this state "w_open". Each of these two states will be represented in the state diagram as circles. The names of the states are written inside of the circles.



2) Defining the Transitions - We now describe the transitions between states using arrows and labeling the arrows with the input conditions that trigger each transition. For this design, when the machine is in the "w_closed" state, a button press (Press=1) will cause a transition to the "w_open" state. When the button is not pressed, the machine will remain in the "w_closed" state (Press=0). When the machine is in the "w_open" state, a button press (Press=1) will cause a transition to the "w_closed" state, while the button not being pressed (Press=0) will keep the machine in the "w_open" state.



3) Defining the Outputs – We now describe the outputs of the system. For this design, the system will output the appropriate motor control signals upon a button press. This means that the outputs depend on both the current state and the current inputs. This is by definition a *Mealy Machine*. As such, the outputs are listed next to the state transitions. By listing the outputs in this location, both the current state and the input values producing the outputs are indicated. When this machine is in either the w_closed or w_open states and the button is NOT pressed, the outputs Open_CW and Close_CCW are both 0's. When the machine is in w_closed state and the button is pressed, the Open_CW output is asserted to rotate the motor clockwise and open the window. When the machine is in w_open state and the button is pressed, the Close_CCW output is asserted to rotate the motor counterclockwise and close the window. The final state diagram for this system is shown below.



Example 7.2
Push-button window controller – State diagram

7.4.1.2 State Transition Tables

The state diagram can now be described in a table format that is similar to a truth table. This puts the state machine's behavior in a form that makes logic synthesis straightforward. The table contains the same information as in the state diagram. The state that the machine exists in is called the *current state*. For each current state that the machine can reside in, every possible input condition is listed along with the destination state of each transition. The destination state for a transition is called the *next state*. Also listed in the table are the outputs corresponding to each current state and, in the case of a Mealy Machine, the output corresponding to each input condition. Example 7.3 shows the construction of the state transition table for the push-button window controller design. This information is identical to the state diagram given in Example 7.2.

Example: Push-Button Window Controller - State Transition Table

A state transition table contains the same information as the state diagram but in a tabular format. This format is similar to a truth table and makes logic synthesis straight forward. Each state and input condition is listed in the table along with the corresponding next state and outputs.

	(Input)		(Outputs)	
Current State	Press	Next State	Open_CW	Close_CCW
w_closed	0	w_closed	0	0
w_closed	1	w_open	1	0
w_open	0	w_open	0	0
w_open	1	w_closed	0	1

Example 7.3
Push-button window controller – State transition table

7.4.2 Logic Synthesis for an FSM

Once the behavior of the state machine has been described, it can be directly synthesized. There are three main components of a state machine: the state memory, the next-state logic, and the output logic. Figure 7.29 shows a block diagram of a state machine, highlighting these three components. The *next-state logic* block is a group of combinational logic that produces the next-state signals based on the current state and any system inputs. The *state memory* holds the current state of the system. The current state is updated with the next state on every rising edge of the clock, which is indicated with the ">" symbol within the block. This behavior is created using D-Flip-Flops, where the current state is held on the Q outputs of the D-Flip-Flops while the next state is present on the D inputs of the D-Flip-Flops. In this way, every rising edge of the clock will trigger an evaluation of which state to move to next. This decision is based on the current state and the current inputs. The *output logic* block is a group of combinational logic that creates the outputs of the system. This block always uses the current state as an input and, depending on the type of machine (Mealy vs. Moore), uses the system inputs. It is useful to keep this block diagram in mind when synthesizing FSMs, as it will aid in keeping the individual design steps separate and clear.

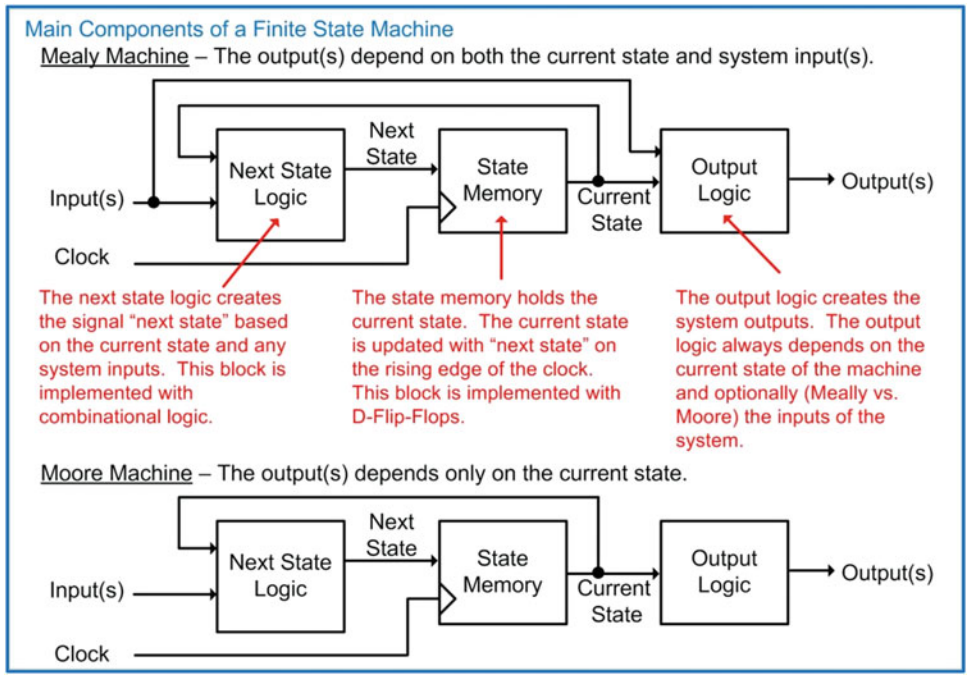


Fig. 7.29
Main components of a finite-state machine

7.4.2.1 State Memory

The state memory is the circuitry that will hold the current state of the machine. Upon the rising edge of a clock, it will update the current state with the next state. At all other times, the next-state input is ignored. This gives time for the next-state logic circuitry to compute the results for the next state. This behavior is identical to that of a D-Flip-Flop; thus, the state memory is simply one or more D-Flip-Flops. The number of D-Flip-Flops required depends on how the states are *encoded*. *State encoding* is the process of assigning a binary value to the descriptive names of the states from the state diagram and state transition tables. Once the descriptive names have been converted into representative codes using 1's and 0's, the states can be implemented in real circuitry. The assignment of codes is arbitrary and can be selected in order to minimize the circuitry needed in the machine.

There are three main styles of state encoding. The first is straight ***binary encoding***. In this approach, the state codes are simply a set of binary counts (i.e., 00, 01, 10, 11...). The binary counts are assigned starting at the beginning of the state diagram and incrementally toward the end. This type of encoding has the advantage that it is very efficient in minimizing the number of D-Flip-Flops needed for the state memory. With n D-Flip-Flops, 2^n states can be encoded. When a large number of states is required, the number of D-Flip-Flops can be calculated using the rules of logarithmic math. Example 7.4 shows how to solve for the number of bits needed in the binary state code based on the number of states in the machine.

Solving For the Number of Bits Needed for Binary State Encoding

Problem: You are designing a state machine that has 41 unique states and are going to encode the states in binary. How many D-Flip-Flops do you need?

Solution: Each D-Flip-Flops will hold one bit of the state code. If the state memory has n-bits, it can encode 2^n states using binary encoding. We can use logarithms in order to solve for the n in the exponent.

$$2^n = (\#\ \text{of states})$$

$$\log(2^n) = \log(\#\ \text{of states})$$

$$n \cdot \log(2) = \log(\#\ \text{of states})$$

$$n = \frac{\log(\#\ \text{of states})}{\log(2)}$$

$$n = \frac{\log(41)}{\log(2)}$$

$$n = 5.36$$

Rounding up to the next whole number means that we need 6 bits, or 6-D-Flip-Flops to encode 41 states in binary.

Check: To check this, let's plug 6 back into the original expression. If we have 6 bits, we can encode 2^6 states, or 64 states. This is enough to encode our 41 states. If we had 1 less bit (e.g., 5), we could only encode up to 2^5=32 states, so we require 6 bits for this state encoding. Note that not all of the possible binary values are used as state codes.

Example 7.4
Solving for the number of bits needed for binary state encoding

The second type of state encoding is called ***gray code encoding***. A gray code is one in which the value of a code differs by only 1-bit from any of its neighbors (i.e., 00, 01, 11, 10. . .). A gray code is useful for reducing the number of bit transitions on the state codes when the machine has a linear transition sequence. Reducing the number of bit transitions can reduce the amount of power consumption and noise generated by the circuit. When the state transitions of a machine are highly non-linear, a gray code encoding approach does not provide any benefit. Gray code is also an efficient coding approach. With *n* D-Flip-Flops, 2^n states can be encoded just as in binary encoding. Figure 7.30 shows the process of creating *n*-bit, gray code patterns.

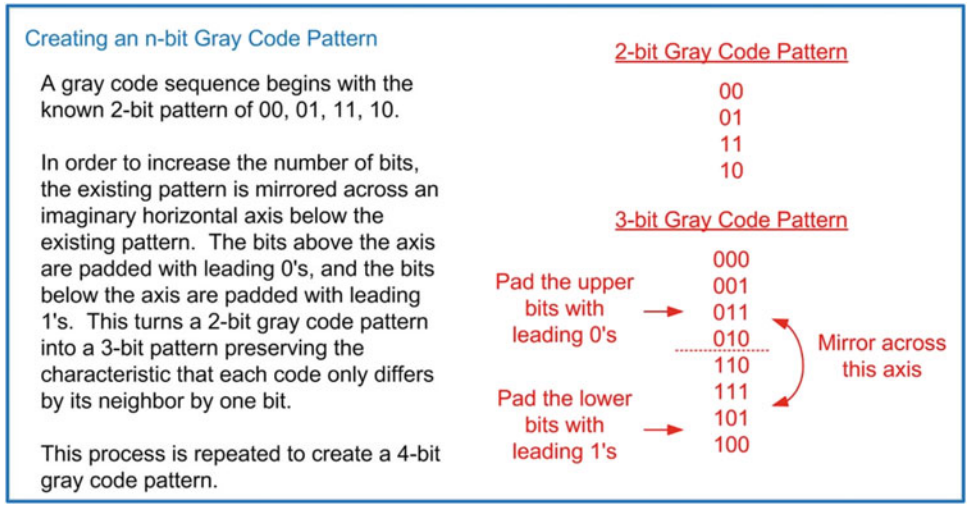


Fig. 7.30
Creating an *n*-bit gray code pattern

The third common technique to encode states is using ***one-hot encoding***. In this approach, a separate D-Flip-Flop is asserted for each state in the machine. For an *n-state* machine, this encoding approach requires *n* D-Flip-Flops. For example, if a machine had 3 states, the one-hot state codes would be "001," "010" and "100," This approach has the advantage that the next-state logic circuitry is very simple; further, there is less chance that the different propagation delays through the next-state logic will cause an inadvertent state to be entered. This approach is not as efficient as binary and gray code in terms of minimizing the number of D-Flip-Flops because it requires one D-Flip-Flop for each state; however, in modern digital integrated circuits that have abundant D-Flip-Flops, one-hot encoding is commonly used.

Comparison of Different State Encoding Approaches

A state machine has eight unique states named S0, S1, … S7. The following is an example of how these states can be encoded using binary, gray code and one-hot.

State Name	Binary	Gray Code	One-Hot
S0	000	000	00000001
S1	001	001	00000010
S2	010	011	00000100
S3	011	010	00001000
S4	100	110	00010000
S5	101	111	00100000
S6	110	101	01000000
S7	111	100	10000000

Fig. 7.31
Comparison of different state-encoding approaches

Figure 7.31 shows the differences between these three state-encoding approaches.

Once the codes have been assigned to the state names, each of the bits within the code must be given a unique signal name. The signal names are necessary because the individual bits within the state code are going to be implemented with real circuitry, so each signal name will correspond to an actual

node in the logic diagram. These individual signal names are called ***state variables***. Unique variable names are needed for both the current state and next-state signals. The current state variables are driven by the Q outputs of the D-Flip-Flops holding the state codes. The next-state variables are driven by the next-state logic circuitry and are connected to the D inputs of the D-Flip-Flops. State variable names are commonly chosen that are descriptive both in terms of their purpose and connection location. For example, current state variables are often given the names Q, Q_cur, or Q_current to indicate that they come from the Q outputs of the D-Flip-Flops. Next-state variables are given names such as Q*, Q_nxt, or Q_next to indicate that they are the *next* value of Q and are connected to the D input of the D-Flip-Flops. Once state codes and state variable names are assigned, the state transition table is updated with the detailed information.

Returning to our push-button window controller example, let's encode our states in straight binary and use the state variable names Q_cur and Q_nxt. Example 7.5 shows the process of state encoding and the new state transition table.

Example: Push-Button Window Controller - State Encoding

This state machine contains two states, w_closed and w_open. The following are the three possible ways these states could be encoded.

State Name	Binary	Gray Code	One-Hot
w_closed	0	0	01
w_open	1	1	10

Since this machine is so small, there is no difference between the binary and gray code approaches. Both of these techniques will require one D-Flip-Flop to hold the state code. The one-hot approach will require two D-Flip-Flops. Let's choose binary state encoding for this example. Let's use the state variable names Q_cur and Q_nxt.

Once the state codes and state variables are chosen, the state transition table is updated with the new detailed information about the design.

Current State		Input	Next State		Outputs	
	Q_cur	Press		Q_nxt	Open_CW	Close_CCW
w_closed	0	0	w_closed	0	0	0
w_closed	0	1	w_open	1	1	0
w_open	1	0	w_open	1	0	0
w_open	1	1	w_closed	0	0	1

Example 7.5
Push-button window controller – State encoding

7.4.2.2 Next-State Logic

The next step in the state machine design is to synthesize the next-state logic. The next-state logic will compute the values of the next-state variables based on the current state and the system inputs. Recall that a combinational logic function drives only one output bit. This means that every bit within the next-state code needs to have a dedicated combinational logic circuit. The state transition table contains all of the necessary information to synthesize the next-state logic, including the exact output values of each next-state variable for each and every input combination of state code and system input(s).

In our push-button window controller example, we only need to create one combinational logic circuit because there is only one next-state variable (Q_nxt). The inputs to the combinational logic circuit are Q_cur and Press. Notice that the state transition table was created such that the order of the input values is listed in a binary count, just as in a formal truth table formation. This makes synthesizing the combinational logic circuit straightforward. Example 7.6 shows the steps to synthesize the next-state logic for this push-button window controller.

Example: Push-Button Window Controller - Next State Logic

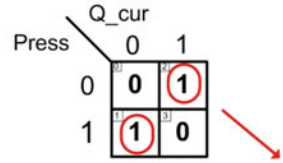
We need to synthesize the combinational logic circuit that will create the next state logic for Q_nxt. The behavior of this combinational logic circuit is described in the state transition table. In order to visualize where this information is within the table, let's pull it out and put it into a traditional truth table format.

Current State	Input	Next State	Outputs
	Q_cur	Press		Q_nxt	Open_CW	Close_CCW
w_closed	0	0	w_closed	0	0	0
w_closed	0	1	w_open	1	1	0
w_open	1	0	w_open	1	0	0
w_open	1	1	w_closed	0	0	1

These columns are the inputs to the next state logic.

This column is the desired output for the next state logic variable Q_nxt.

Q_cur	Press	Q_nxt
0	0	0
0	1	1
1	0	1
1	1	0



$Q\_nxt = (Q\_cur' \cdot Press) + (Q\_cur \cdot Press')$

or

$Q\_nxt = Q\_cur \oplus Press$

Example 7.6
Push-button window controller – Next-state logic

7.4.2.3 Output Logic

The next step in the state machine design is to synthesize the output logic. The output logic will compute the values of the system outputs based on the current state and, in the case of a Mealy machine, the system inputs. Each of the output signals will require a dedicated combinational logic circuit. Again, the state transition table contains all of the necessary information to synthesize the output logic.

In our push-button window controller example, we need to create one circuit to compute the output "Open_CW" and one circuit to compute the output "Close_CCW." In this example, the inputs to these circuits are the current state (Q_cur) and the system input (Press). Example 7.7 shows the steps to synthesize the output logic for the push-button window controller.

Example: Push-Button Window Controller - Output Logic

We need to synthesize the combinational logic circuits that will create the output logic for the signals "Open_CW" and "Close_CCW". The behavior of this combinational logic circuit is described in the state transition table. Again, in order to visualize where this information is within the table, let's pull it out and put it into traditional truth table formats.

Current State		Input	Next State		Outputs	
	Q_cur	Press		Q_nxt	Open_CW	Close_CCW
w_closed	0	0	w_closed	0	0	0
w_closed	0	1	w_open	1	1	0
w_open	1	0	w_open	1	0	0
w_open	1	1	w_closed	0	0	1

These columns are the inputs to the output logic.

These columns are the desired behavior of the outputs.

Q_cur	Press	Open_CW
0	0	0
0	1	1
1	0	0
1	1	0

Press \ Q_cur	0	1
0	0	0
1	1	0

Open_CW = Q_cur' · Press

Q_cur	Press	Close_CCW
0	0	0
0	1	0
1	0	0
1	1	1

Press \ Q_cur	0	1
0	0	0
1	0	1

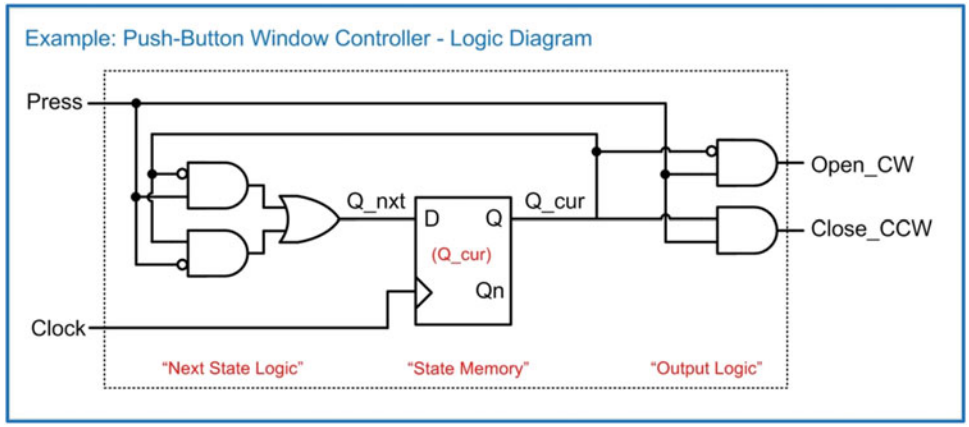
Close_CCW = Q_cur · Press

Example 7.7
Push-button window controller – Output logic

7.4.2.4 The Final Logic Diagram

The final step in the design of the state machine is to create the logic diagram. It is useful to recall the block diagram for a state machine from Fig. 7.29. A logic diagram begins by entering state memory. Recall that the state memory consists of D-Flip-Flops that hold the current state code. One D-Flip-Flop is needed for every current state variable. When entering the D-Flip-Flops, it is useful to label them with the current state variable they will be holding. The next part of the logic diagram is the next-state logic. Each of the combinational logic circuits that compute the next-state variables should be drawn to the left of D-Flip-Flop, holding the corresponding current state variable. The output of each next-state logic circuit is connected to the D input of the corresponding D-Flip-Flop. Finally, the output logic is entered, with the inputs to the logic coming from the current state and potentially from the system inputs.

Example 7.8 shows the process for creating the final logic diagram for our push-button window controller. Notice that the state memory is implemented with one D-Flip-Flop since there is only 1-bit in the current state code (Q_cur). The next-state logic is a combinational logic circuit that computes Q_nxt based on the values of Q_cur and Press. Finally, the output logic consists of two separate combinational logic circuits to compute the system outputs Open_CW and Close_CCW based on Q_cur and Press. In this diagram, the Qn output of the D-Flip-Flop could have been used for the inverted versions of Q_cur; however, inversion bubbles were used instead in order to make the diagram more readable.



Example 7.8
Push-button window controller – Logic diagram

7.4.3 FSM Design Process Overview

The entire FSM design process is given in Fig. 7.32.

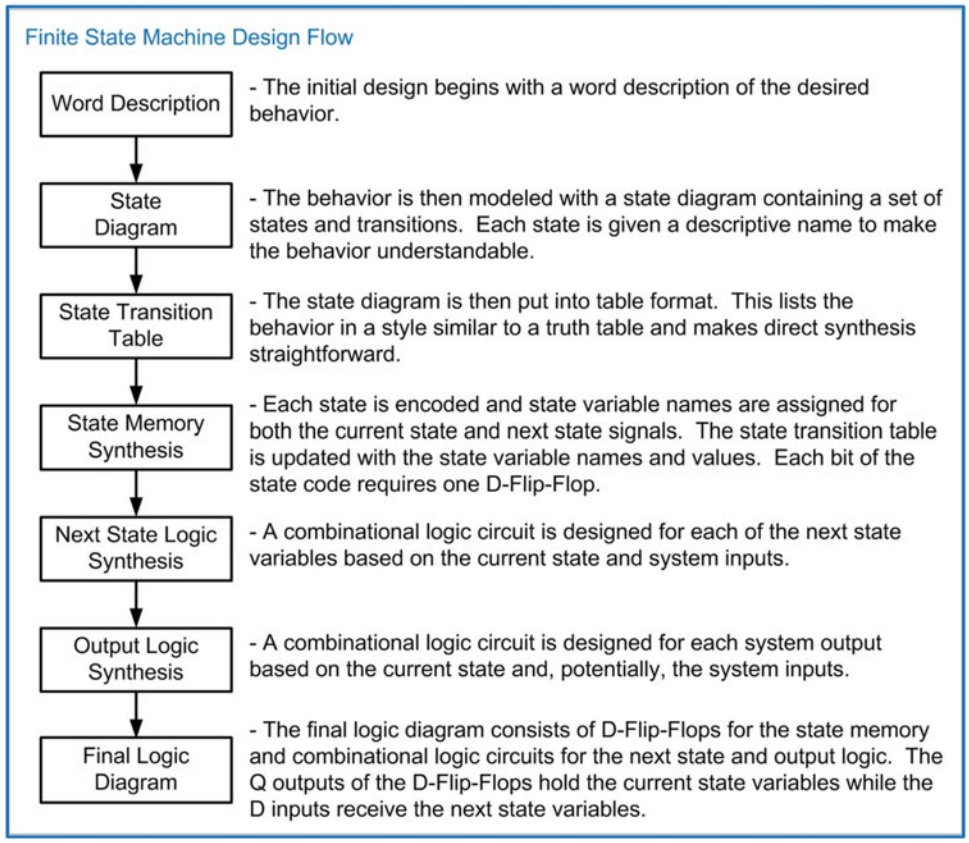


Fig. 7.32
Finite-state machine design flow

7.4.4 FSM Design Examples

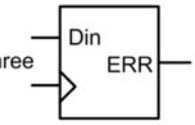
7.4.4.1 Serial Bit Sequence Detector

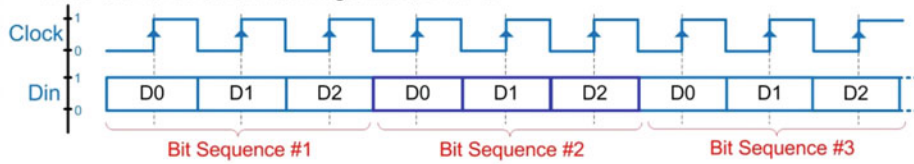
Let's consider the design of a 3-bit serial sequence detector. Example 7.9 provides the word description, state diagram, and state transition table for this FSM.

Example: Serial Bit Sequence Detector (Part 1)

Word Description

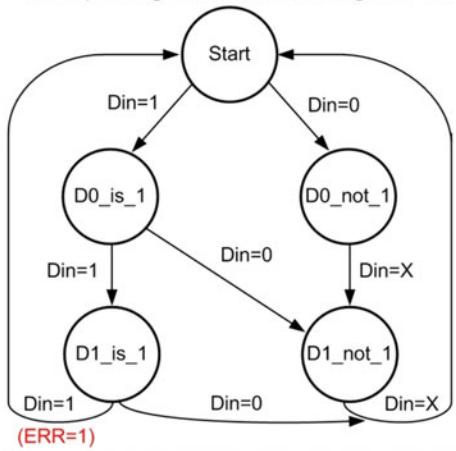
We are going to design a circuit that will monitor an incoming serial bit stream. The information in the bit stream represents data in groups of three bits. The code "111" represents that an error has occurred in the transmitter. Our system needs to monitor the incoming bit stream and assert a signal called "ERR" if the sequence "111" is detected. At all other times and for all other incoming codes, ERR=0.





State Diagram & State Transition Table

To implement this design, we need a machine that can keep track of the number of incoming bits. In this way, the machine will know once the three bits within a sequence have been received. The machine must also track if the sequence of incoming bits are 1's. In order to do this, let's create a sequence of states that will be traversed when Din=1. We also need a parallel sequence of states that will be traversed if an incoming bit is ever a 0. Each of these parallel paths must contain enough states to track that three bits of the sequence have been received before starting over and monitoring the next incoming sequence. The only time the output ERR will be asserted is when three 1's are received within one three bit data sequence. To simplify the state diagram, the output of ERR=1 is only listed once next to the corresponding transition in the diagram. It is assumed that at all other times, ERR=0.



	(Input)		(Output)
Current State	Din	Next State	ERR
Start	0	D0_not_1	0
Start	1	D0_is_1	0
D0_is_1	0	D1_not_1	0
D0_is_1	1	D1_is_1	0
D1_is_1	0	Start	0
D1_is_1	1	Start	1
D0_not_1	0	D1_not_1	0
D0_not_1	1	D1_not_1	0
D1_not_1	0	Start	0
D1_not_1	1	Start	0

Example 7.9
Serial bit sequence detector (Part 1)

Example 7.10 provides the state encoding and next-state logic synthesis for the 3-bit serial bit sequence detector.

Example: Serial Bit Sequence Detector (Part 2)

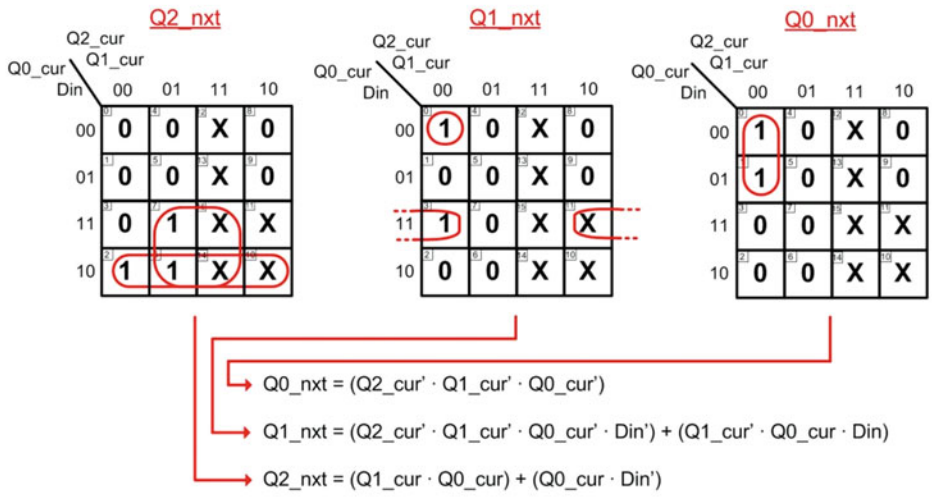
State Encoding

Let's encode the states in binary in order to minimize the number of D-Flip-Flops. Encoding in Gray Code will not benefit this design since the state transitions are not linear. Since there are 5 unique states, we'll need 3 bits to encode all of the states. At this point, we also need to assign the state variable names. Let's call the three variables for the current state Q2_cur, Q1_cur, and Q0_cur. Let's call the three variables for the next state Q2_nxt, Q1_nxt, and Q0_nxt. After the state codes are assigned, we can update the state transition table.

State	Code
Start	= "000"
D0_is_1	= "001"
D1_is_1	= "010"
D0_not_1	= "011"
D1_not_1	= "100"

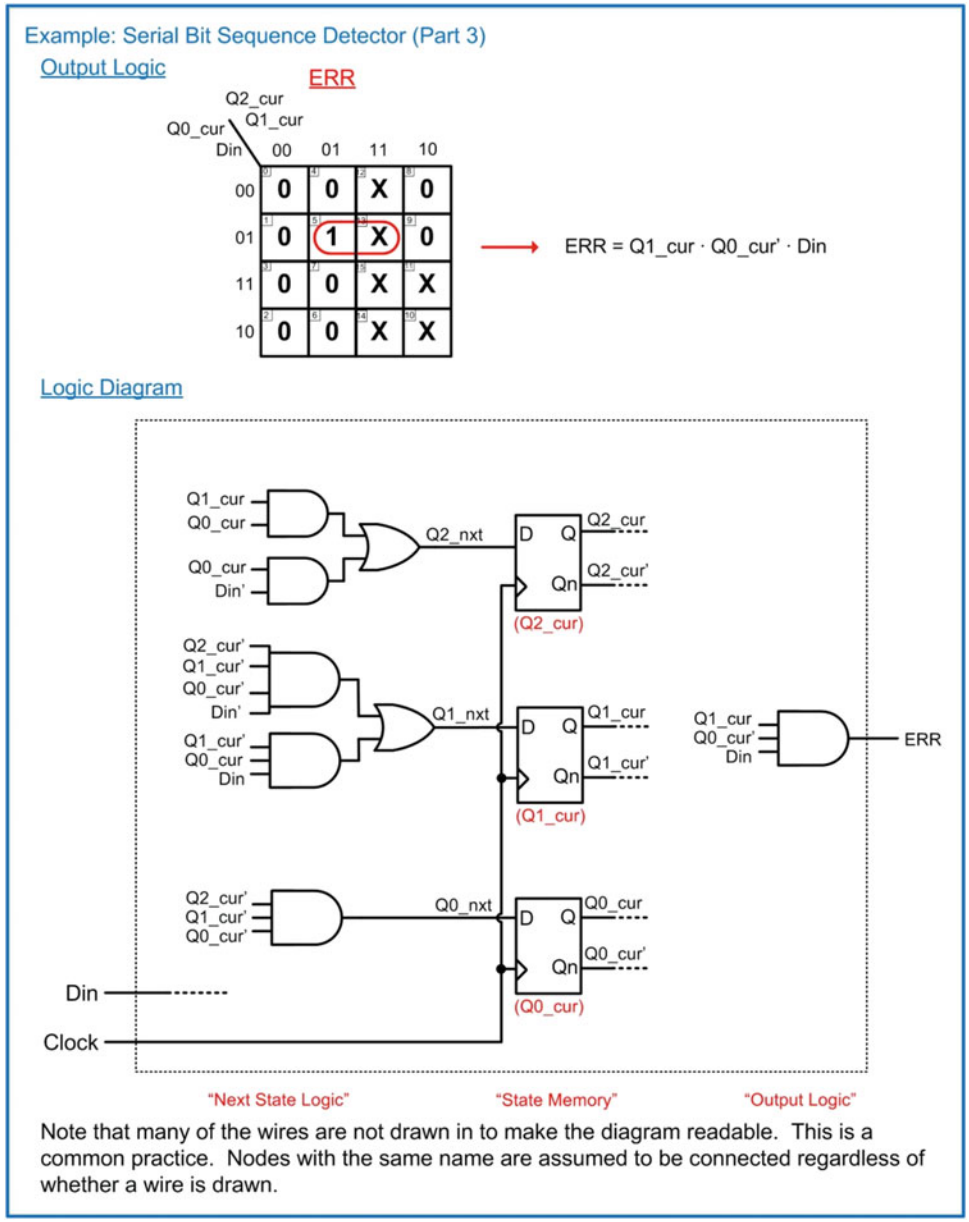
Current State				Input	Next State				Output
	Q2_cur	Q1_cur	Q0_cur	Din		Q2_nxt	Q1_nxt	Q0_nxt	ERR
Start	0	0	0	0	D0_not_1	0	1	1	0
Start	0	0	0	1	D0_is_1	0	0	1	0
D0_is_1	0	0	1	0	D1_not_1	1	0	0	0
D0_is_1	0	0	1	1	D1_is_1	0	1	0	0
D1_is_1	0	1	0	0	Start	0	0	0	0
D1_is_1	0	1	0	1	Start	0	0	0	1
D0_not_1	0	1	1	0	D1_not_1	1	0	0	0
D0_not_1	0	1	1	1	D1_not_1	1	0	0	0
D1_not_1	1	0	0	0	Start	0	0	0	0
D1_not_1	1	0	0	1	Start	0	0	0	0

Next State Logic



Example 7.10
Serial bit sequence detector (Part 2)

Example 7.11 shows the output logic synthesis and final logic diagram for the 3-bit serial bit sequence detector.



Note that many of the wires are not drawn in to make the diagram readable. This is a common practice. Nodes with the same name are assumed to be connected regardless of whether a wire is drawn.

Example 7.11
Serial bit sequence detector (Part 3)

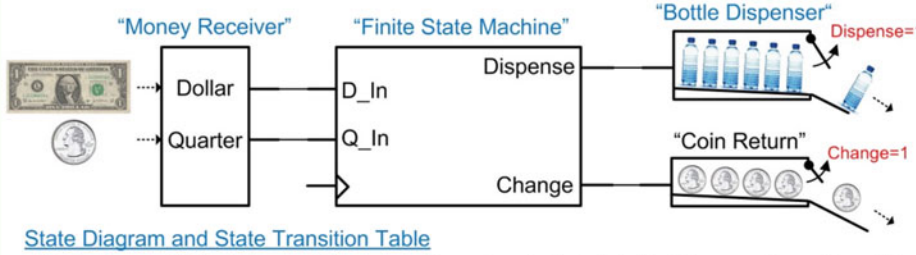
7.4.4.2 Vending Machine Controller

Let's now look at the design of a simple vending machine controller. Example 7.12 provides the word description, state diagram, and state transition table for this FSM.

Example: Vending Machine Controller (Part 1)

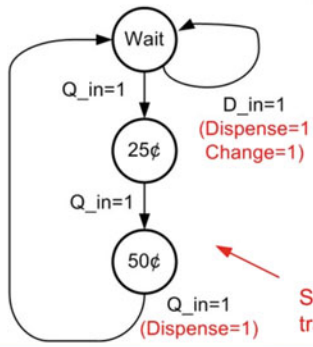
Word Description

We are going to design a simple vending machine controller. The vending machine will sell bottles of water for 75¢. Customers can enter either a dollar bill or quarters. Once a sufficient amount of money is entered, the vending machine will dispense a bottle of water. If the user entered a dollar it will return one quarter in change. A "Money Receiver" detects when money has been entered. The receiver sends two logic signals to our circuit indicating whether a dollar bill or quarter was received. A "Bottle Dispenser" system holds the water bottles and will release one bottle when its input signal is asserted. A "Coin Return" system holds quarters for change and will release one quarter when its input signal is asserted. The money receiver will reject money if a dollar and quarter are entered simultaneously or if a dollar is entered once the user has started entering quarters.



State Diagram and State Transition Table

To implement this state machine, we will need an initial state that the machine will wait in until a customer enters money (Wait). If a dollar is entered, the machine will assert the "Dispense" signal to release a bottle of water and assert the "Change" signal to give one quarter in change. We do not need an additional state for the condition of when a dollar is entered because the machine will simply assert the output signals and return to the Wait state. When the customer pays with quarters, our machine needs to keep track of how many quarters have been received. We'll need two interim states that keep track of how many quarters have been entered (25¢ and 50¢). Once the third quarter has been entered, our machine will assert the "Dispense" signal and return to the Wait state.



	(Inputs)			(Outputs)	
Current State	Q_in	D_in	Next State	Dispense	Change
Wait	0	0	Wait	0	0
Wait	0	1	Wait	1	1
Wait	1	0	25¢	0	0
25¢	0	X	25¢	0	0
25¢	1	X	50¢	0	0
50¢	0	X	50¢	0	0
50¢	1	X	Wait	1	0

State diagrams can be simplified by only drawing transitions when a signal is asserted.

Example 7.12
Vending machine controller (Part 1)

Example 7.13 provides the state encoding and next-state logic synthesis for the simple vending machine controller.

Example: Vending Machine Controller (Part 2)

State Encoding

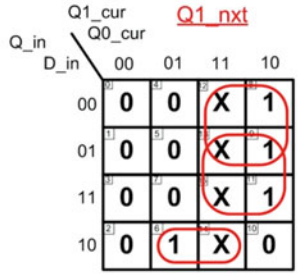
Let's encode the states in binary and name the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. In this table we list out all possible values the current state and the inputs to make the table more complete.

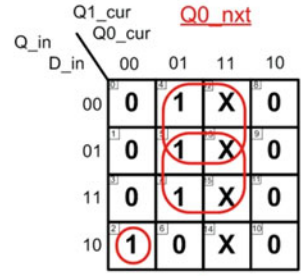
State	Code
Wait	= "00"
25¢	= "01"
50¢	= "10"

Current State			Input		Next State			Outputs	
	Q1_cur	Q0_cur	Q_in	D_in		Q1_nxt	Q0_nxt	Dispense	Change
Wait	0	0	0	0	Wait	0	0	0	0
Wait	0	0	0	1	Wait	0	0	1	1
Wait	0	0	1	0	25¢	0	1	0	0
Wait	0	0	1	1	Wait	0	0	0	0
25¢	0	1	0	0	25¢	0	1	0	0
25¢	0	1	0	1	25¢	0	1	0	0
25¢	0	1	1	0	50¢	1	0	0	0
25¢	0	1	1	1	25¢	0	1	0	0
50¢	1	0	0	0	50¢	1	0	0	0
50¢	1	0	0	1	50¢	1	0	0	0
50¢	1	0	1	0	Wait	0	0	1	0
50¢	1	0	1	1	50¢	1	0	0	0

Next State Logic

The next state logic for this counter depends on both the current state variables and the system input Up. We can again take advantage of don't cares for the unused state code to minimize the logic.



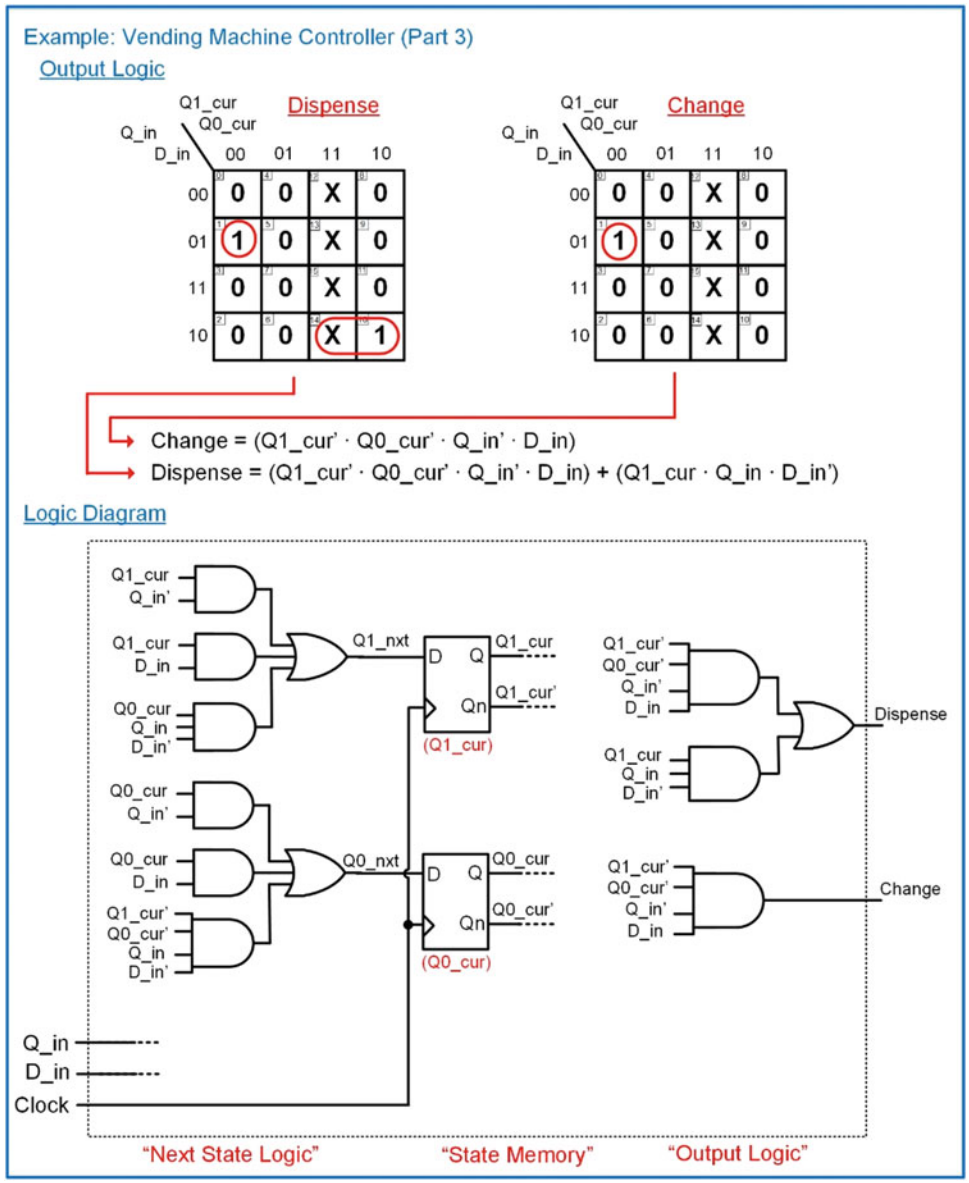


Q0_nxt = (Q0_cur · Q_in') + (Q0_cur · D_in) + (Q1_cur' · Q0_cur' · Q_in · D_in')

Q1_nxt = (Q1_cur · Q_in') + (Q1_cur · D_in) + (Q0_cur · Q_in · D_in')

Example 7.13
Vending machine controller (Part 2)

Example 7.14 shows the output logic synthesis and final logic diagram for the vending machine controller.



Example 7.14
Vending machine controller (Part 3)

Concept Check

CC7.4(a) What allows an FSM to make more intelligent decisions about the system outputs compared to combinational logic alone?

A) An FSM has knowledge about past inputs.

B) The D-flip-flops allow the outputs to be generated more rapidly.

C) The next-state and output logic allow the FSM to be more complex and implement larger truth tables.

D) A synchronous system is always more intelligent.

CC7.4(b) When designing an FSM, many of the details of the implementation can be abstracted. At what design step do the details of the implementation start being considered?

A) The state diagram step.

B) The state transition table step.

C) The state memory synthesis step.

D) The word description.

CC7.4(c) What impact does adding an additional state have on the implementation of the state memory logic in an FSM?

A) It adds an additional D-Flip-Flop.

B) It adds a new state code that must be supported.

C) It adds more combinational logic to the logic diagram.

D) It reduces the speed that the machine can run at.

CC7.4(d) Which of the following statements about the next-state logic is FALSE?

A) It is always combinational logic.

B) It always uses the current state as one of its inputs.

C) Its outputs are connected to the D inputs of the D-flip-flops in the state memory.

D) It uses the results of the output logic as part of its inputs.

CC7.4(e) Why does the output logic stage of an FSM always use the current state as one of its inputs?

A) If it didn't, it would simply be a separate combinational logic circuit and not be part of the FSM.

B) To make better decisions about what the system's outputs should be.

C) Because the next-state logic is located too far away.

D) Because the current state is produced on every triggering clock edge.

CC7.4(f) What impact does asserting a reset have on an FSM?

A) It will cause the output logic to produce all zeros.

B) It will cause the next-state logic to produce all zeros.

C) It will set the current state code to all zeros.

D) It will start the system clock.

7.5 Counters

A *counter* is a special type of FSM. A counter will traverse the states within a state diagram in a linear fashion, continually circling around all states. This behavior allows a special type of output topology called *state-encoded outputs*. Since each state in the counter represents a unique counter output, the states can be encoded with the associated counter output value. In this way, the current state code of the machine can be used as the output of the entire system.

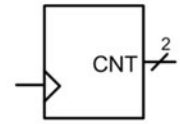
7.5.1 2-Bit Binary Up Counter

Let's consider the design of a 2-bit binary up counter. Example 7.15 provides the word description, state diagram, state transition table, and state encoding for this counter.

Example: 2-Bit Binary Up Counter (Part 1)

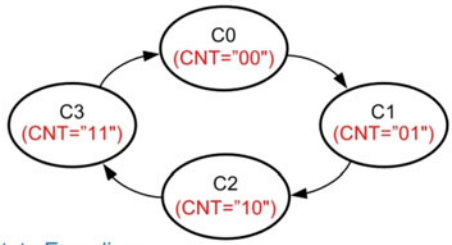
Word Description

We are going to design a 2-bit binary up counter. The counter will increment by 1 on every rising edge of the clock ("00", "01", "10", "11"). When the counter reaches "11", it will start over counting at "00". The output of the counter is called CNT.



State Diagram & State Transition Table

The state diagram for this counter is below. Notice that there are no inputs to the state machine. Also notice that the machine transitions in a linear pattern through the states and continually repeats the sequence of states. The outputs of this machine depend only on the current state so they are written inside of the state circles. This is a Moore machine.



Current State	Next State	(Output) CNT
C0	C1	"00"
C1	C2	"01"
C2	C3	"10"
C3	C0	"11"

State Encoding

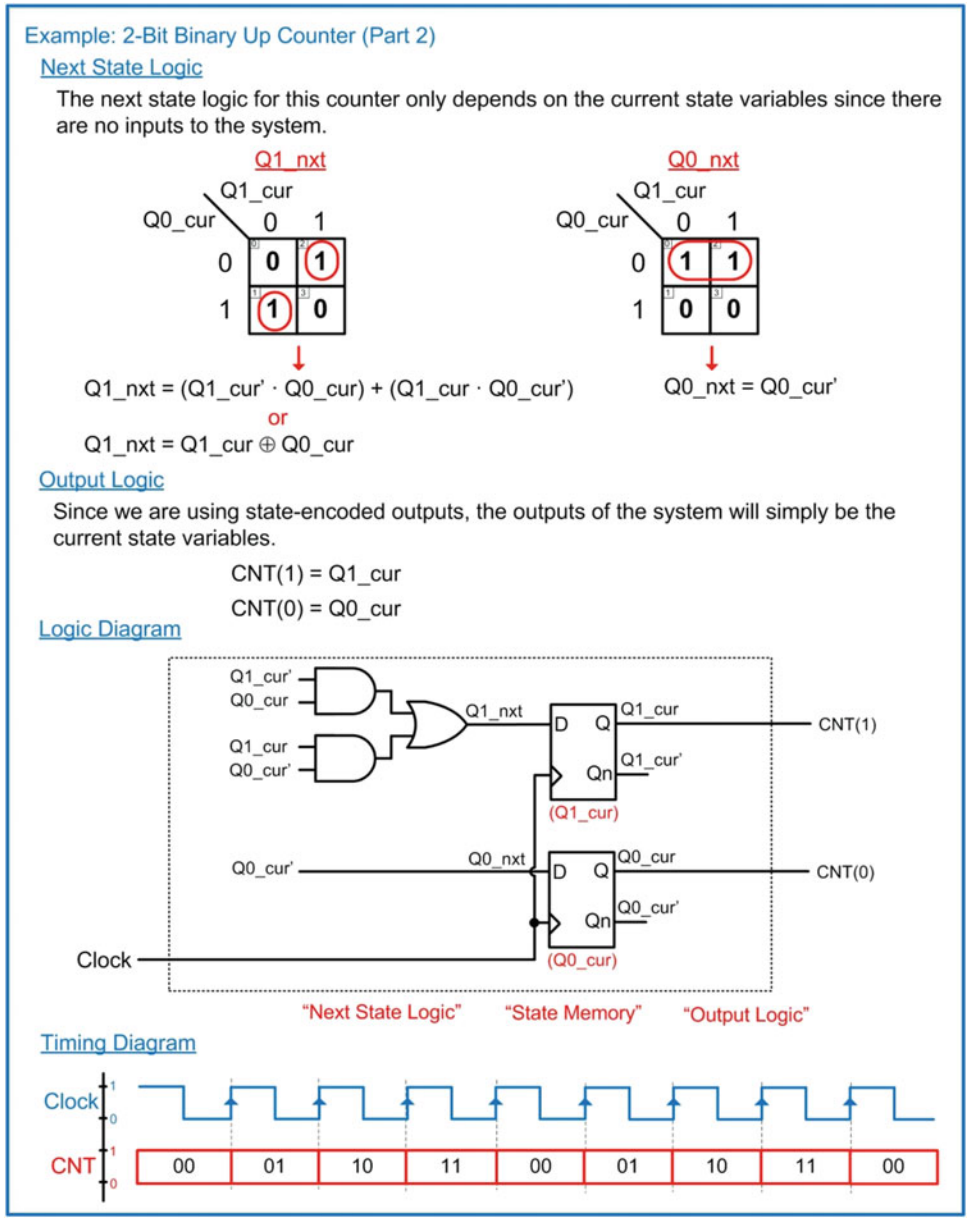
When implementing this counter, we can use "state-encoded outputs". This means that we choose the state codes so that they match the desired output at each state. This allows the machine to simply use the current state variables for the system outputs. Let's name the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State	Code
C0	= "00"
C1	= "01"
C2	= "10"
C3	= "11"

Current State			Next State			Outputs
	Q1_cur	Q0_cur		Q1_nxt	Q0_nxt	CNT
C0	0	0	C1	0	1	"00"
C1	0	1	C2	1	0	"01"
C2	1	0	C3	1	1	"10"
C3	1	1	C0	0	0	"11"

Example 7.15
2-Bit binary up counter (Part 1)

Example 7.16 shows the next-state and output logic synthesis, the final logic diagram, and the resultant representative timing diagram for the 2-bit binary up counter.



Example 7.16
2-Bit binary up counter (Part 2)

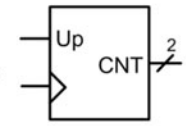
7.5.2 2-Bit Binary Up/Down Counter

Let's now consider a 2-bit binary up/down counter. In this type of counter, there is an input that dictates whether the counter increments or decrements. This counter can still be implemented as a Moore machine and use state-encoded outputs. Example 7.17 provides the word description, state diagram, state transition table, and state encoding for this counter.

Example: 2-Bit Binary Up/Down Counter (Part 1)

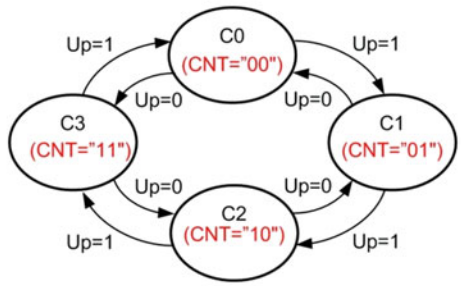
Word Description

We are going to design a 2-bit binary up/down counter. When the system input "Up" is asserted, the counter will increment by 1 on every rising edge of the clock. When Up=0, the counter will decrement by 1 on every rising edge of the clock. The output of the counter is called CNT.



State Diagram & State Transition Table

The state diagram for this counter is below. In this diagram, if the input Up=1, the machine will traverse the states in order to create an incrementing count. If the input Up=0, the machine will traverse the states in the opposite order. The outputs of this machine again only depend on the current state so they are written inside of the state circles. This is a Moore machine.



	(Input)		(Output)
Current State	Up	Next State	CNT
C0	0	C3	"00"
	1	C1	
C1	0	C0	"01"
	1	C2	
C2	0	C1	"10"
	1	C3	
C3	0	C2	"11"
	1	C0	

State Encoding

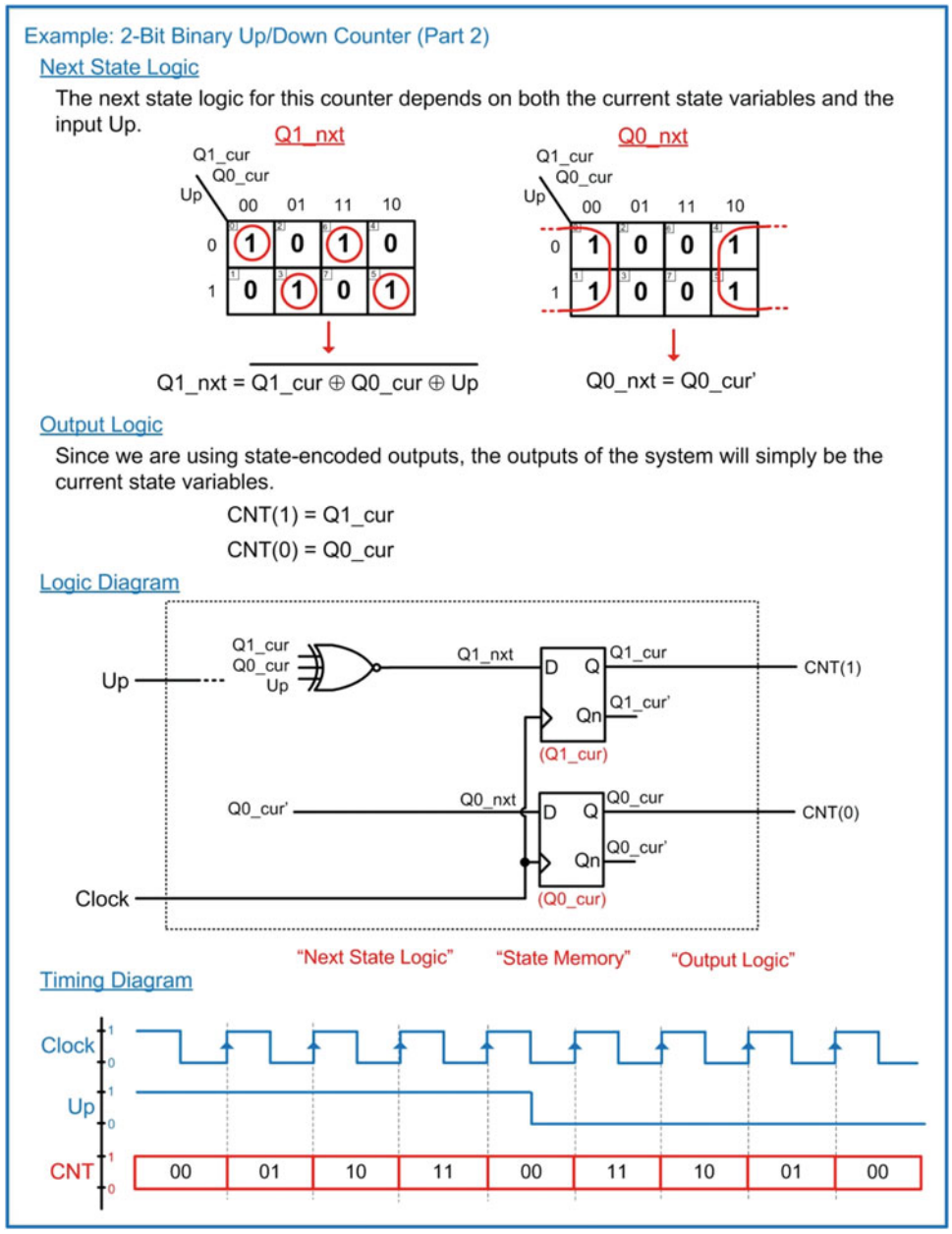
Again, this counter will use "state-encoded outputs". Let's name the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State	Code
C0	= "00"
C1	= "01"
C2	= "10"
C3	= "11"

Current State			Input	Next State			Outputs
	Q1_cur	Q0_cur	Up		Q1_nxt	Q0_nxt	CNT
C0	0	0	0	C3	1	1	"00"
C0	0	0	1	C1	0	1	"00"
C1	0	1	0	C0	0	0	"01"
C1	0	1	1	C2	1	0	"01"
C2	1	0	0	C1	0	1	"10"
C2	1	0	1	C3	1	1	"10"
C3	1	1	0	C2	1	0	"11"
C3	1	1	1	C0	0	0	"11"

Example 7.17
2-Bit binary up/down counter (Part 1)

Example 7.18 shows the next-state and output logic synthesis, the final logic diagram, and resultant representative timing diagram for the 2-bit binary up/down counter.



Example 7.18
2-Bit binary up/down counter (Part 2)

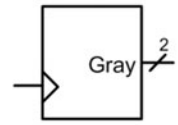
7.5.3 2-Bit Gray Code Up Counter

A gray code counter is one in which the output only differs by 1-bit from its prior value. This type of counter can be implemented using state-encoded outputs by simply encoding the states in gray code. Let's consider the design of a 2-bit gray code up counter. Example 7.19 provides the word description, state diagram, state transition table, and state encoding for this counter.

Example: 2-Bit Gray Code Up Counter (Part 1)

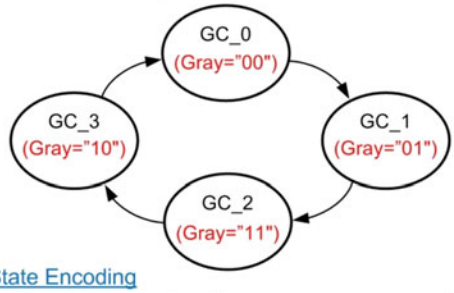
Word Description

We are going to design a 2-bit gray code up counter. The counter will output an incrementing gray code pattern on every rising edge of the clock ("00", "01", "11", "10). When the counter reaches "11", it will start over counting at "00". The output of the counter is called Gray.



State Diagram & State Transition Table

The state diagram for this counter is below. Notice that there are no inputs to the state machine. Also notice that the machine transitions in a linear pattern through the states and continually repeats the sequence of states. The outputs of this machine depend only on the current state, so they are written inside of the state circles. This is a Moore machine.



(Output)

Current State	Next State	Gray
GC_0	GC_1	"00"
GC_1	GC_2	"01"
GC_2	GC_3	"11"
GC_3	GC_0	"10"

State Encoding

When implementing this counter, we can use "state-encoded outputs". This means that we choose the state codes so that they match the desired output at each state. This allows the machine to simply use the current state variables for the system outputs. Let's name the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State	Code
GC_0	= "00"
GC_1	= "01"
GC_2	= "11"
GC_3	= "10"

Current State			Next State			Outputs
	Q1_cur	Q0_cur		Q1_nxt	Q0_nxt	Gray
GC_0	0	0	GC_1	0	1	"00"
GC_1	0	1	GC_2	1	1	"01"
GC_2	1	1	GC_3	1	0	"11"
GC_3	1	0	GC_0	0	0	"10"

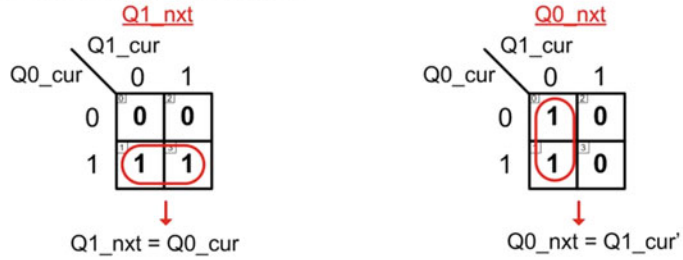
Example 7.19
2-Bit gray code up counter (Part 1)

Example 7.20 shows the next-state and output logic synthesis, the final logic diagram, and the resultant representative timing diagram for the 2-bit gray code up counter.

Example: 2-Bit Gray Code Up Counter (Part 2)

Next State Logic

The next state logic for this counter only depends on the current state variables since there are no inputs to the system. Care must be taken when synthesizing the next state logic because the order of the current state variable values in the state transition table is not in a binary count order as in prior examples.



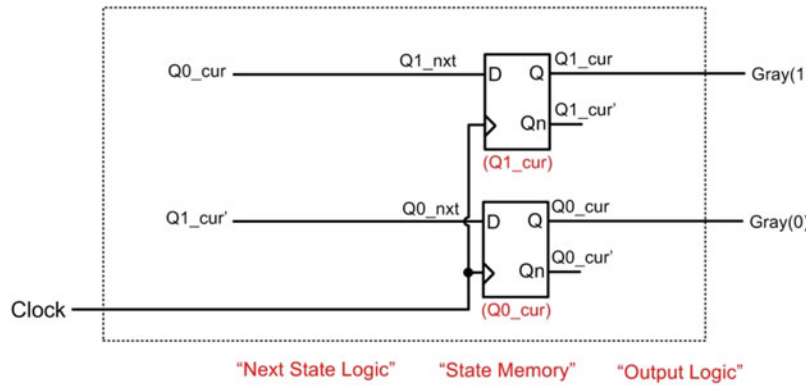
Output Logic

Since we are using state-encoded outputs, the outputs of the system will simply be the current state variables.

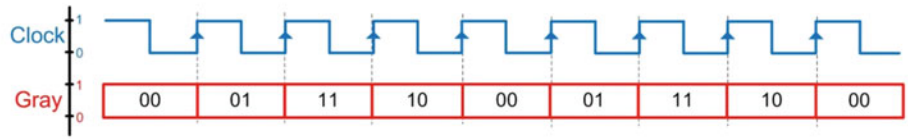
Gray(1) = Q1_cur

Gray(0) = Q0_cur

Logic Diagram



Timing Diagram



Example 7.20
2-Bit gray code up counter (Part 2)

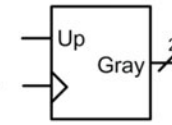
7.5.4 2-Bit Gray Code Up/Down Counter

Let's now consider a 2-bit gray code up/down counter. In this type of counter, there is an input that dictates whether the counter increments or decrements. This counter can still be implemented as a Moore machine and use state-encoded outputs. Example 7.21 provides the word description, state diagram, state transition table, and state encoding for this counter.

Example: 2-Bit Gray Code Up/Down Counter (Part 1)

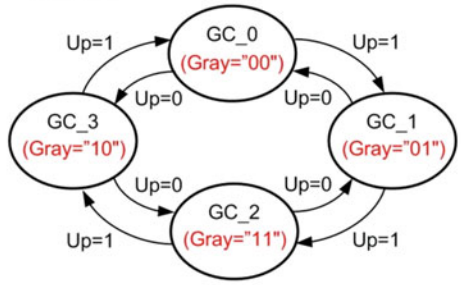
Word Description

We are going to design a 2-bit gray code up/down counter. When the system input "Up" is asserted, the counter will output an incrementing gray code pattern on every rising edge of the clock ("00", "01", "11", "10"). When the input Up=0, the counter will output a decrementing gray code pattern. The output of the counter is called Gray.



State Diagram & State Transition Table

The state diagram for this counter is below. The outputs of this machine again only depend on the current state, so they are written inside of the state circles. This is a Moore machine.



	(Input)		(Output)
Current State	Up	Next State	Gray
GC_0	0	GC_3	"00"
	1	GC_1	
GC_1	0	GC_0	"01"
	1	GC_2	
GC_2	0	GC_1	"11"
	1	GC_3	
GC_3	0	GC_2	"10"
	1	GC_0	

State Encoding

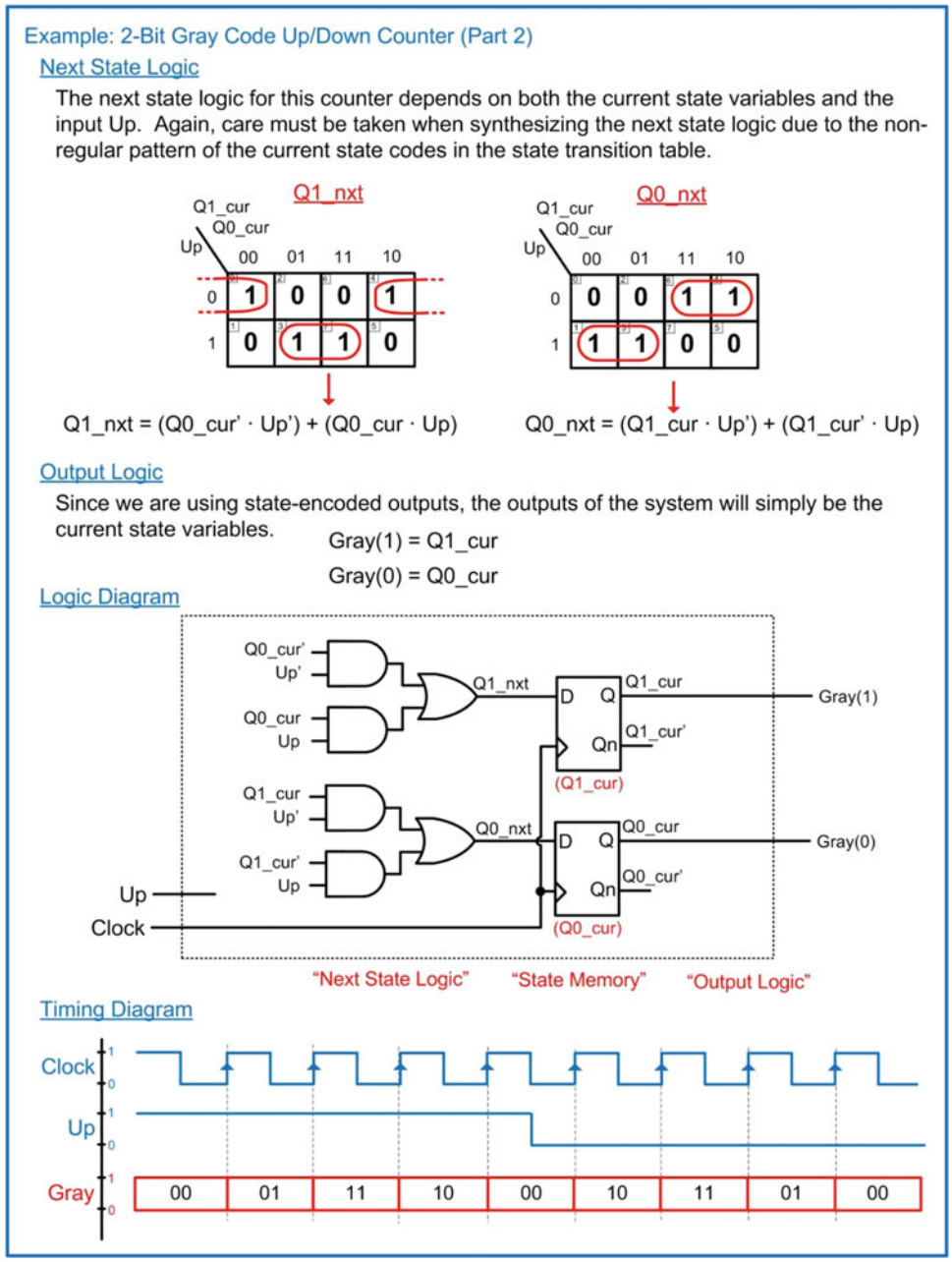
Again, this counter will use "state-encoded outputs". Let's name the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State	Code
GC_0	= "00"
GC_1	= "01"
GC_2	= "11"
GC_3	= "10"

Current State			Input	Next State			Outputs
	Q1_cur	Q0_cur	Up		Q1_nxt	Q0_nxt	Gray
GC_0	0	0	0	GC_3	1	0	"00"
GC_0	0	0	1	GC_1	0	1	"00"
GC_1	0	1	0	GC_0	0	0	"01"
GC_1	0	1	1	GC_2	1	1	"01"
GC_2	1	1	0	GC_1	0	1	"11"
GC_2	1	1	1	GC_3	1	0	"11"
GC_3	1	0	0	GC_2	1	1	"10"
GC_3	1	0	1	GC_0	0	0	"10"

Example 7.21
2-Bit gray code up/down counter (Part 1)

Example 7.22 shows the next-state and output logic synthesis, the final logic diagram, and resultant representative timing diagram for the 2-bit gray code up/down counter.



Example 7.22
2-Bit gray code up/down counter (Part 2)

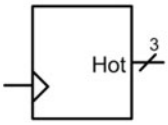
7.5.5 3-Bit One-Hot Up Counter

A one-hot counter creates an output in which one and only one bit is asserted at a time. In an *up counter* configuration, the assertion is made on the least significant bit first, followed by the next higher significant bit, and so on (i.e., 001, 010, 100, 001…). A one-hot counter can be created using state-encoded outputs. For an *n*-bit counter, the machine will require *n* D-Flip-Flops. Let's consider a 3-bit one-hot up counter. Example 7.23 provides the word description, state diagram, state transition table, and state encoding for this counter.

Example: 3-Bit One-Hot Up Counter (Part 1)

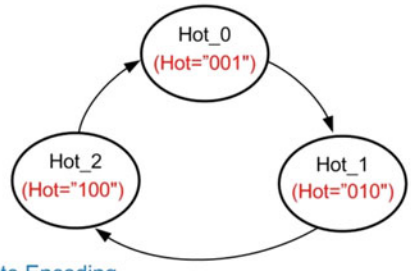
Word Description

We are going to design a 3-bit one-hot up counter. The counter will output an incrementing one-hot pattern on every rising edge of the clock ("001", "010", "100"). When the counter reaches "100", it will start over counting at "001". The output of the counter is called Hot.



State Diagram & State Transition Table

The state diagram for this counter is below. Notice that there are no inputs to the state machine. The outputs of this machine depend only on the current state so they are written inside of the state circles. This is a Moore machine.



Current State	Next State	(Output) Hot
Hot_0	Hot_1	"001"
Hot_1	Hot_2	"010"
Hot_2	Hot_0	"100"

State Encoding

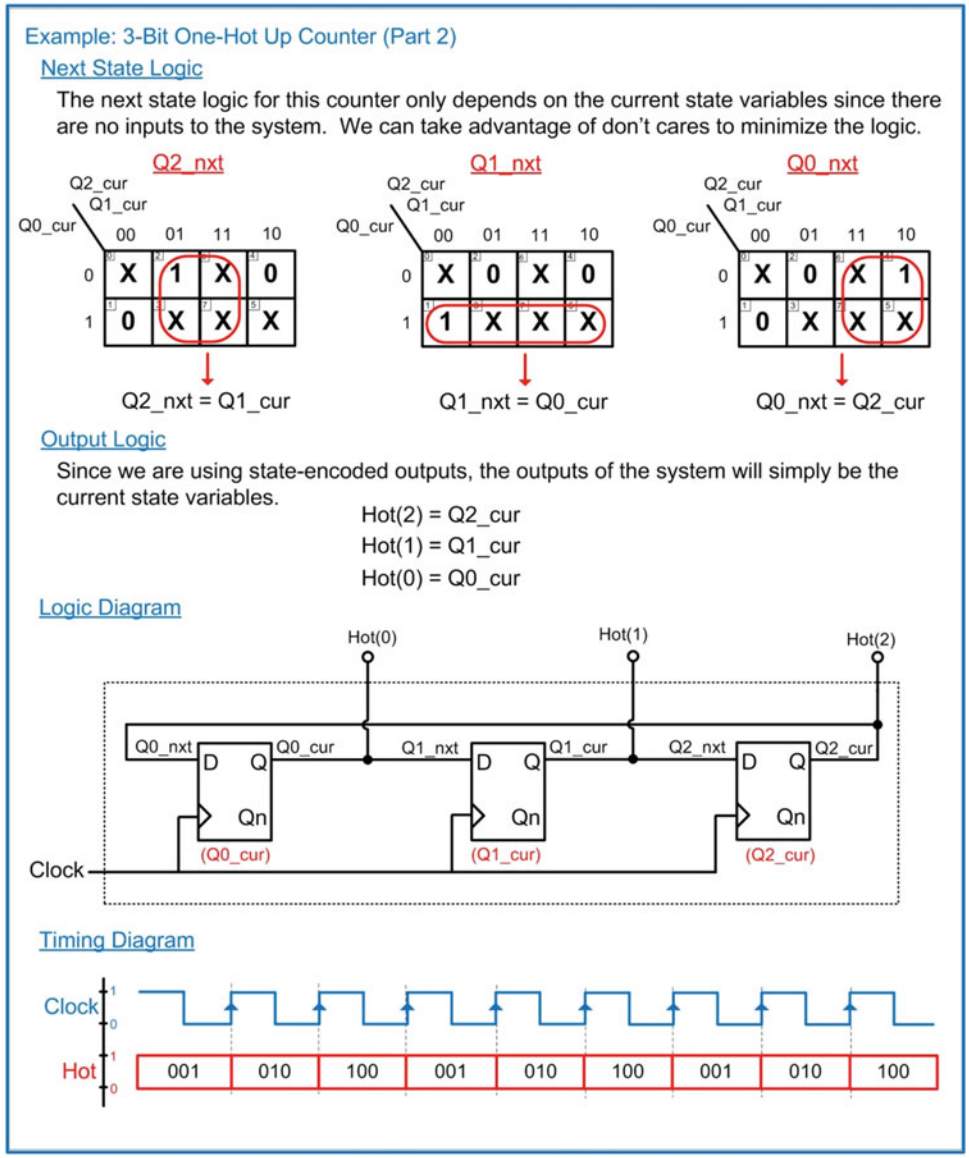
When implementing this counter, we can use "state-encoded outputs". Using one-hot state encoding requires three bits to encode the states. This means we'll need three variables for both the current state and next state. Let's name the current state variables Q2_cur, Q1_cur and Q0_cur and the next state variables Q2_nxt, Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State	Code
Hot_0	= "001"
Hot_1	= "010"
Hot_2	= "100"

Current State				Next State				Outputs
	Q2_cur	Q1_cur	Q0_cur		Q2_nxt	Q1_nxt	Q0_nxt	Hot
Hot_0	0	0	1	Hot_1	0	1	0	"001"
Hot_1	0	1	0	Hot_2	1	0	0	"010"
Hot_2	1	0	0	Hot_0	0	0	1	"100"

Example 7.23
3-Bit one-hot up counter (Part 1)

Example 7.24 shows the next-state and output logic synthesis, the final logic diagram, and the resultant representative timing diagram for the 3-bit one-hot up counter.



Example 7.24
3-Bit one-hot up counter (Part 2)

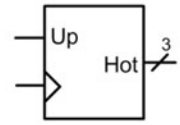
7.5.6 3-Bit One-Hot Up/Down Counter

Let's now consider a 3-bit one-hot up/down counter. In this type of counter, there is an input that dictates whether the counter increments or decrements. This counter can still be implemented as a Moore machine and use state-encoded outputs. Example 7.25 provides the word description, state diagram, state transition table, and state encoding for this counter.

Example: 3-Bit One-Hot Up/Down Counter (Part 1)

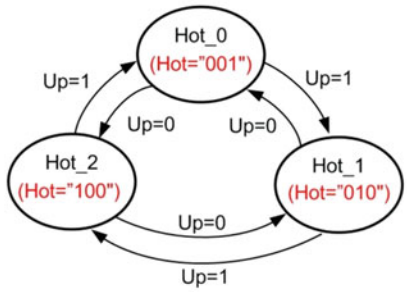
Word Description

We are going to design a 3-bit one-hot up/down counter. When the system input "Up" is asserted, the counter will output an incrementing one-hot pattern on every rising edge of the clock ("001", "010", "100"). When the input Up=0, the counter will output a decrementing one-hot pattern ("100", "010", "001"). The output of the counter is called Hot.



State Diagram & State Transition Table

The state diagram and state transition table for this counter are below.



	(Input)		(Output)
Current State	Up	Next State	Hot
Hot_0	0	Hot_2	"001"
Hot_0	1	Hot_1	"001"
Hot_1	0	Hot_0	"010"
Hot_1	1	Hot_2	"010"
Hot_2	0	Hot_1	"100"
Hot_2	1	Hot_0	"100"

State Encoding

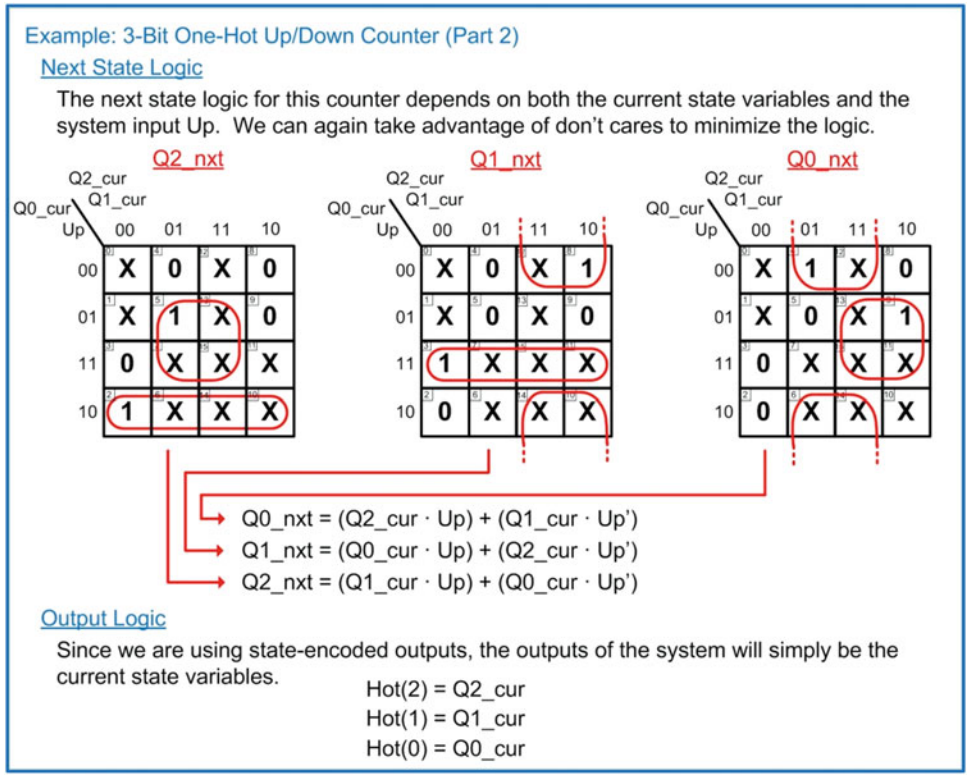
Let's use "state-encoded outputs" and name the current state variables Q2_cur, Q1_cur and Q0_cur and the next state variables Q2_nxt, Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State Code
Hot_0 = "001"
Hot_1 = "010"
Hot_2 = "100"

	Current State			Input	Next State			Outputs
	Q2_cur	Q1_cur	Q0_cur	Up		Q2_nxt	Q1_nxt	Q0_nxt	Hot
Hot_0	0	0	1	0	Hot_2	1	0	0	"001"
Hot_0	0	0	1	1	Hot_1	0	1	0	"001"
Hot_1	0	1	0	0	Hot_0	0	0	1	"010"
Hot_1	0	1	0	1	Hot_2	1	0	0	"010"
Hot_2	1	0	0	0	Hot_1	0	1	0	"100"
Hot_2	1	0	0	1	Hot_0	0	0	1	"100"

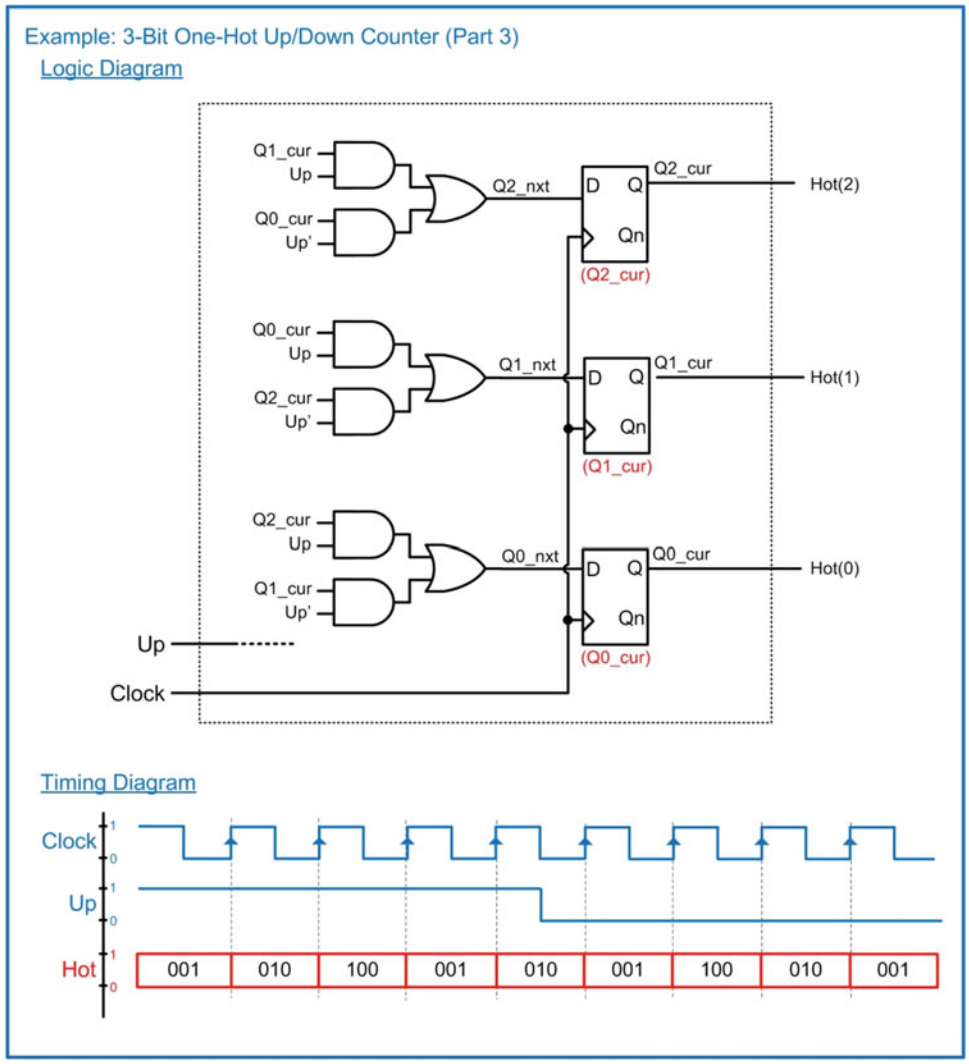
Example 7.25
3-Bit one-hot up/down counter (Part 1)

Example 7.26 shows the next-state and output logic synthesis for the 3-bit one-hot up/down counter.



Example 7.26
3-Bit one-hot up/down counter (Part 2)

Finally, Example 7.27 shows the logic diagram and resultant representative timing diagram for the counter.



Example 7.27
3-Bit one-hot up/down counter (Part 3)

Concept Check

CC7.5 What characteristic of a counter makes it a special case of a FSM?

A) The state transitions are mostly linear, which reduces the implementation complexity.

B) The outputs are always gray codes.

C) The next-state logic circuitry is typically just sum terms.

D) There is never a situation where a counter could be a Mealy machine.

7.6 Finite-State Machine's Reset Condition

The one-hot counter designs in Examples 7.23 and 7.25 were the first FSM examples that had an initial state that was not encoded with all 0's. Notice that all of the other FSM examples had initial states with state codes comprised of all 0's (e.g., w_closed = 0, S0 = "00," C0 = "00," GC_0 = "00, etc.). When the initial state is encoded with all 0's, the FSM can be put into this state by asserting the reset line of all of the D-Flip-Flops in the state memory. By asserting the reset line, the Q outputs of all of the D-Flip-Flips are forced to 0's. This sets the initial current state value to whatever state is encoded with all 0's. The initial state of a machine is often referred to as the *reset state*. The circuitry to initialize state machines is often omitted from the logic diagram, as it is assumed that the necessary circuitry will exist in order to put the state machine into the reset state. If the reset state is encoded with all 0's, then the reset line can be used alone; however, if the reset state code contains 1's, then both the reset *and* preset lines must be used to put the machine into the reset state upon start-up. Let's look at the behavior of the one-hot up counter again. Figure 7.33 shows how using the reset lines of the D-Flip-Flops alone will cause the circuit to operate incorrectly. Instead, a combination of the reset and preset lines must be used to get the one-hot counter into its initial state of Hot_0 = "001."

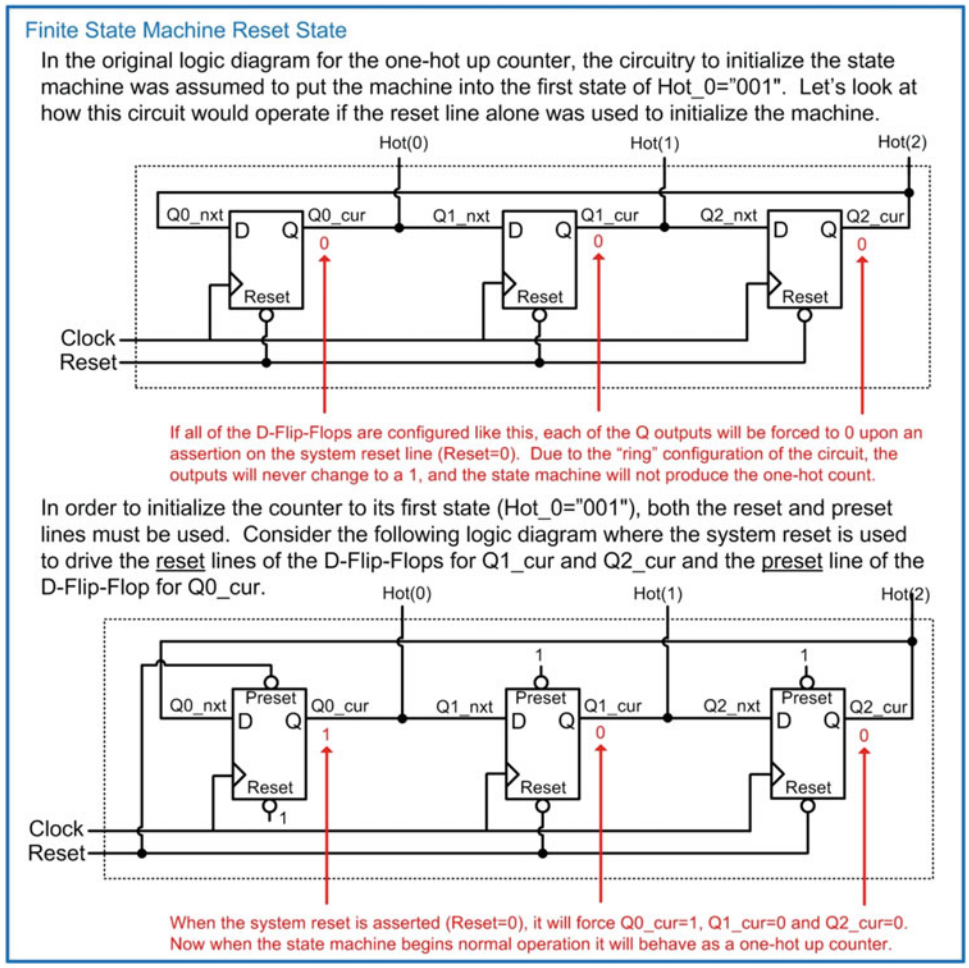


Fig. 7.33
Finite-state machine reset state

Resets are most often asynchronous so that they can immediately alter the state of the FSM. If a reset was implemented in a synchronous manner and there was a clock failure, the system could not be reset since there would be no more subsequent clock edges that would recognize that the reset line was asserted. An asynchronous reset allows the system to be fully restarted even in the event of a clock failure.

CONCEPT CHECK

CC7.6 What is the downside of using D-flip-flops that do not have preset capability in a FSM?

A) The FSM will run slower.

B) The next-state logic will be more complex.

C) The output logic will not be able to support both Mealy and Moore-type machine architectures.

D) The start-up state can never have a 1 in its state code.

7.7 Sequential Logic Analysis

Sequential logic analysis refers to the act of deciphering the operation of a circuit from its final logic diagram. This is similar to combinational logic analysis, with the exception that the storage capability of the D-flip-flops must be considered. This analysis is also used to understand the timing of a sequential logic circuit and can be used to predict the maximum clock rate that can be used.

7.7.1 Finding the State Equations and Output Logic Expressions of an FSM

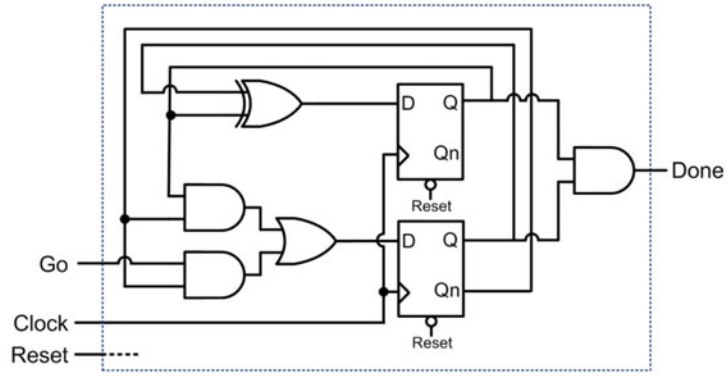
When given the logic diagram for a FSM and it is desired to reverse-engineer its behavior, the first step is to determine the next-state logic and output logic expressions. This can be accomplished by first labeling the current and next-state variables on the inputs and outputs of the D-Flip-Flops that are implementing the state memory of the FSM. The outputs of the D-Flip-Flops are labeled with arbitrary current state variable names (e.g., Q1_cur, Q0_cur, etc.), and the inputs are labeled with arbitrary next-state variable names (e.g., Q1_nxt, Q0_nxt, etc.). The numbering of the state variables can be assigned to the D-Flip-Flops arbitrarily as long as the current and next-state bit numberings match. For example, if a D-Flip-Flop is labeled to hold bit 0 of the state code, its output should be labeled Q**0**_cur and its input should be labeled Q**0**_nxt.

Once the current state variable nets are labeled in the logic diagram, the expressions for the next-state logic can be found by analyzing the combinational logic circuit, driving the next-state variables (e.g., Q1_nxt and Q0_nxt). The next-state logic expressions will be in terms of the current state variables (e.g., Q1_cur and Q0_cur) and any inputs to the FSM.

The output logic expressions can also be found by analyzing the combinational logic driving the outputs of the FSM. Again, these will be in terms of the current state variables and potentially the inputs to the FSM. When analyzing the output logic, the type of machine can be determined. If the output logic only depends on combinational logic that is driven by the current state variables, the FSM is a Moore machine. If the output logic depends on both the current state variables and the FSM inputs, the FSM is a Mealy machine. An example of this analysis approach is given in Example 7.28.

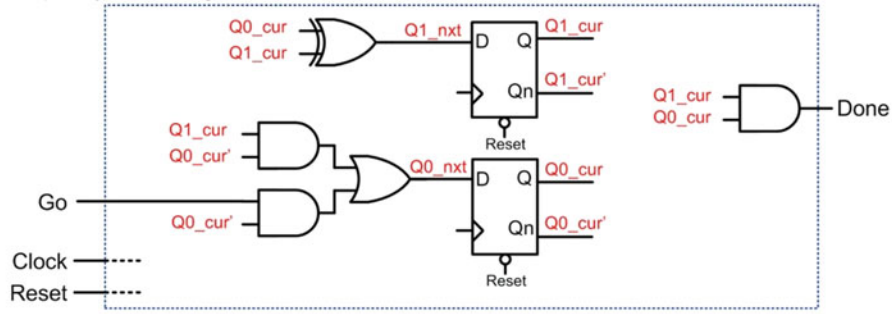
Example: Determining the Next State Logic and Output Logic Expressions of a FSM

Given: The following finite state machine logic diagram.



Find: The logic expressions for the next state and output logic.

Solution: First, we need to label the inputs and outputs of the D-Flip-Flops. Let's call the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. We can assign these node names to whichever D-flip-flop we wish as long as we match the next state and current state variable numbers (i.e., Q1_nxt with Q1_cur and Q0_nxt and Q0_cur). We can also redraw the diagram without all of the connecting nets to reduce the complexity of the diagram.



From this drawing the next state logic and output logic expressions can be found directly.

Q1_nxt = Q0_cur ⊕ Q1_cur
Q0_nxt = (Q1_cur · Q0_cur') + (Go · Q0_cur')

Done = Q1_cur · Q0_cur

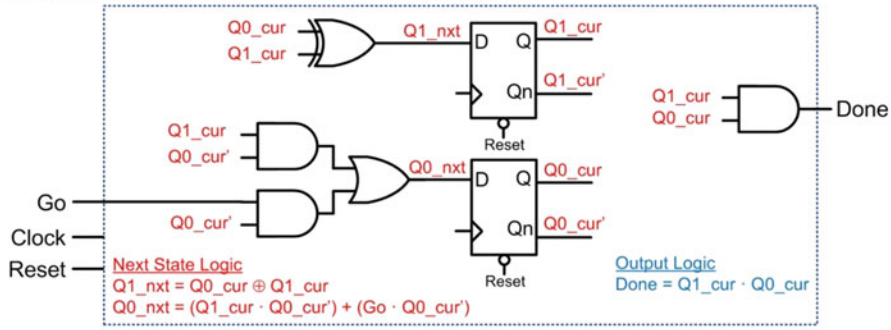
Example 7.28
Determining the next-state logic and output logic expression of an FSM

7.7.2 Finding the State Transition Table of an FSM

Once the next-state logic and output logic expressions are known, the state transition table can be created. It is useful to assign more descriptive names to all possible state codes in the FSM. The number of unique states possible depends on how many D-Flip-Flops are used in the state memory of the FSM. For example, if the FSM uses two D-flip-flops there are four unique state codes (i.e., 00, 01, 10, and 11). We can assign descriptive names such as S0 = 00, S1 = 01, S2 = 10, and S3 = 11. When first creating the transition table, we assign labels and list each possible state code. If a particular code is not used, it can be removed from the transition table at the end of the analysis. The state code that the machine will start in can be found by analyzing its reset and preset connections. This code is typically listed first in the table. The transition table is then populated with all possible combinations of current states and inputs. The next-state codes and output logic values can then be populated by evaluating the next-state logic and output logic expressions found earlier. An example of this analysis is shown in Example 7.29.

Example: Determining the State Transition Table of a FSM

Given: The following finite state machine logic diagram.



Find: The state transition table.

Solution: Since there are two D-Flip-Flops in this circuit there can be four unique state codes (00, 01, 10, and 11). We notice that the reset condition for this FSM will initialize this machine to state 00. We will insert this state code in the table as the first state. Also, we can assign four arbitrary state names to these codes. Let's use S0=00, S1=01, S2=10, and S3=11. We can list these state names and current state codes in the table along with every possible value of the input. The last step is to simply evaluate the logic expressions for the next state variables and the output to complete the table.

Current State			Input	Next State			Outputs
	Q1_cur	Q0_cur	Go		Q1_nxt	Q0_nxt	Done
S0	0	0	0	S0	0	0	0
S0	0	0	1	S1	0	1	0
S1	0	1	0	S2	1	0	0
S1	0	1	1	S2	1	0	0
S2	1	0	0	S3	1	1	0
S2	1	0	1	S3	1	1	0
S3	1	1	0	S0	0	0	1
S3	1	1	1	S0	0	0	1

The current state codes and Go are used as the inputs into the next state logic and output logic expressions.

These values are calculated using the next state logic and output logic expressions. The state names for the next states are added last.

Example 7.29
Determining the state transition table of an FSM

7.7.3 Finding the State Diagram of an FSM

Once the state transition table is found, creating the state diagram becomes possible. We start the diagram with the state corresponding to the reset state. We then draw how the FSM transitions between each of its possible states based on the inputs to the machine and list the corresponding outputs. An example of this analysis is shown in Example 7.30.

Example: Determining the State Diagram of a FSM

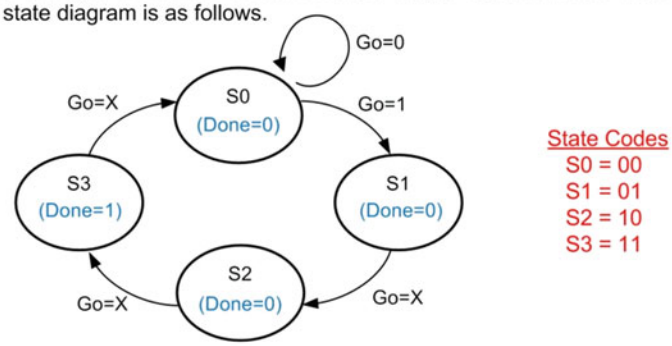
Given: The following state transition table that has been created from a FSM logic diagram.

Current State			Input	Next State			Outputs
	Q1_cur	Q0_cur	Go		Q1_nxt	Q0_nxt	Done
S0	0	0	0	S0	0	0	0
S0	0	0	1	S1	0	1	0
S1	0	1	0	S2	1	0	0
S1	0	1	1	S2	1	0	0
S2	1	0	0	S3	1	1	0
S2	1	0	1	S3	1	1	0
S3	1	1	0	S0	0	0	1
S3	1	1	1	S0	0	0	1

Find: The state diagram.

Solution: The reset condition for this FSM is S0=00 based on the way that the resets of the D-Flip-Flops were connected in the prior logic diagram. This allows us to begin drawing the state diagram starting in S0. From this state we simply list the next state based on the input Go. We notice that the machine will stay in S0 when Go=0 and will transition to S1 when Go=1. We then notice that the machine transitions from S1-to-S2, from S2-to-S3, and from S3-to-S0 regardless of the input value. We can draw these transitions with the input condition Go=X.

For the output Done, we notice that it only depends on the current state, thus this is a Moore machine. For this type of machine we can write the output value within the state bubbles. The final state diagram is as follows.



Example 7.30
Determining the state diagram of an FSM

7.7.4 Determining the Maximum Clock Frequency of an FSM

The maximum clock frequency is often one of the banner specifications for a digital system. The clock frequency of an FSM depends on a variety of timing specifications within the sequential circuit, including the setup and hold time of the D-Flip-Flop, the clock-to-Q delay of the D-Flip-Flop, the combinational logic delay driving the input of the D-Flip-Flop, the delay of the interconnect that wires the circuit together, and the desired margin for the circuit. The basic concept of analyzing the timing of FSM is to determine how long we must wait after a rising (assuming a rising edge-triggered D-Flip-Flop) clock edge occurs until the subsequent rising clock edge can occur. The amount of time that must be allowed between rising clock edges depends on how much delay exists in the system. A sufficient

amount of time must exist between clock edges to allow the logic computations to settle so that on the next clock edge, the D-Flip-Flops can latch in a new value on their inputs.

Let's examine all of the sources of delay in an FSM. Let's begin by assuming that all logic values are stable and we experience a rising clock edge. The value present on the D input of the D-flip-flop is latched into the storage device and will appear on the Q output after one clock-to-Q delay of the device (t_{CQ}). Once the new value is produced on the output of the D-flip-flop, it is then used by a variety of combinational logic circuits to produce the next-state codes and the outputs of the FSM. The next-state code computation is typically longer than the output computation, so let's examine that path. The new value on Q propagates through the combinational logic circuitry and produces the next-state code at the D input of the D-Flip-Flop. The delay to produce this next-state code includes wiring delays in addition to gate delays. When analyzing the delay of the combinational logic circuitry (t_{cmb}) and the delay of the interconnect (t_{int}), the worst-case path is always considered. Once the new logic value is produced by the next-state logic circuitry, it must remain stable for a certain amount of time in order to meet the D-flip-flop's setup specification (t_{setup}). Once this specification is met, the D-flip-flop *could* be clocked with the next clock edge; however, this represents a scenario without any *margin* in the timing. This means that if anything in the system caused the delay to increase even slightly, the D-flip-flop could go metastable. To avoid this situation, margin is included in the delay (t_{margin}). This provides some padding so that the system can reliably operate. A margin of 10% is typical in digital systems. The time that must exist between rising clock edges is then simply the sum of all of these sources of delay ($t_{CQ} + t_{cmb} + t_{int} + t_{setup} + t_{margin}$). Since the time between rising clock edges is defined as the period of the signal (T), this value is also the definition of the period of the fastest clock. Since the frequency of a signal is simply $f = 1/T$, the maximum clock frequency for the FSM is the reciprocal of the sum of the delays.

One specification that is not discussed in the above description is the hold time of the D-Flip-Flop (t_{hold}). The hold specification is the amount of time that the input to the D-Flip-Flop must remain constant after the clock edge. In modern storage devices, this time is typically very small and considerably less than the t_{CQ} specification. If the hold specification is less than $t_{CQ,}$ it can be ignored because the output of the D-Flip-Flop will not change until after one t_{CQ} anyway. This means that the hold requirements are inherently met. This is the situation with the majority of modern D-Flip-Flops. In the rare case that the hold time is greater than t_{CQ}, then it is used in place of t_{CQ} in the summation of delays. Figure 7.34 gives the summary of the maximum clock frequency calculation when analyzing an FSM.

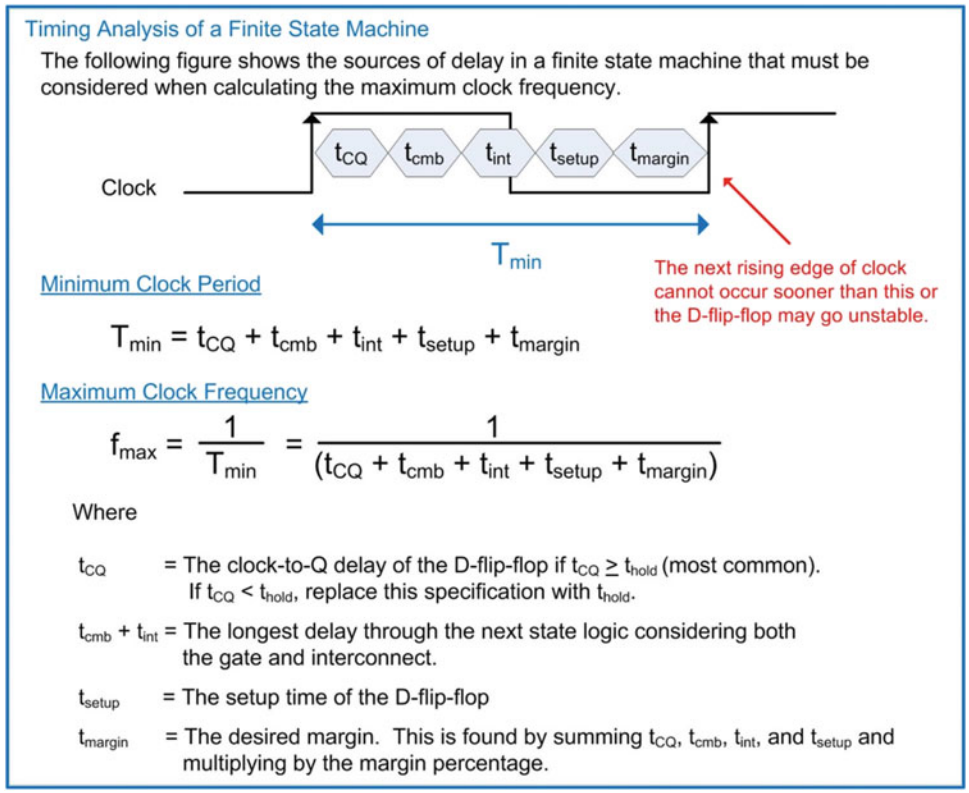
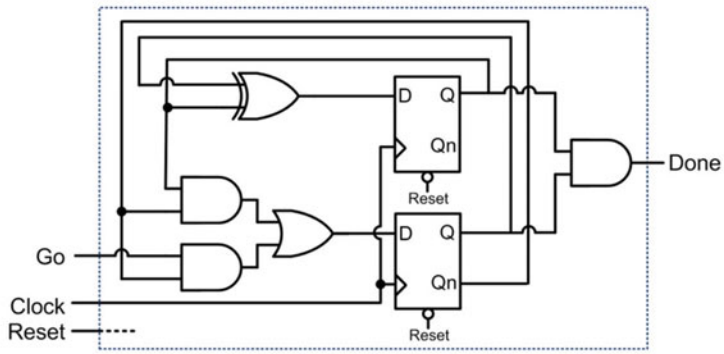


Fig. 7.34
Timing analysis of a finite-state machine

Let's take a look at an example of how to use this analysis. Example 7.31 shows this analysis for the FSM analyzed in prior sections, but this time considering the delay specifications of each device.

Example: Determining the Maximum Clock Frequency of a FSM

Given: The following finite state machine logic diagram with the associated delays.



t_{setup} = 0.5 ns
t_{hold} = 0.5 ns
t_{CQ} = 1 ns

t_{XOR} = 4 ns
t_{AND} = 1 ns
t_{OR} = 2 ns

t_{int} = 0.5 ns (for all paths)

Find: The maximum clock frequency this FSM can operate at with a timing margin of 10%.

Solution: First, we need to decide whether to use t_{CQ} or t_{hold} in our delay calculation. In this example, $t_{CQ} > t_{hold}$ so we will use t_{CQ}. When $t_{CQ} \geq t_{hold}$ the hold specification of the D-flip-flop is inherently met.

Next, we need to find the longest combinational logic and interconnect path. Since it is given that all interconnect paths are identical at 0.5ns, we simply need to find the longest gate delay path. There are three paths in this FSM. The first is the next state logic circuit using the XOR gate with 4ns of delay (t_{XOR}). The second is the next state logic expression using the SOP form with a delay of 3ns ($t_{AND} + t_{OR}$ =1ns + 2ns). The third path is through the output logic circuit with a delay of 1ns (t_{AND}). The longest combinational logic path is through the XOR gate so in our calculation we will use t_{cmb}=4ns.

Next, we need to calculate the exact value of the 10% margin required. The margin is found by summing all other real delays in the signal path and multiplying by the margin percentage. For this example:

$$t_{margin} = (t_{CQ} + t_{cmb} + t_{int} + t_{setup})\cdot(0.1)$$
$$t_{margin} = (1ns + 4ns + 0.5ns + 0.5ns)\cdot(0.1)$$
$$t_{margin} = 0.6ns$$

Now we can plug all of our delays directly into the equation for the maximum clock frequency:

$$f_{max} = \frac{1}{(t_{CQ} + t_{cmb} + t_{int} + t_{setup} + t_{margin})} = \frac{1}{(1ns + 4ns + 0.5ns + 0.5ns + 0.6ns)}$$

$$f_{max} = 151\ MHz$$

Example 7.31
Determining the maximum clock frequency of an FSM

Concept Check

CC7.7 What is the risk of running the clock above its maximum allowable frequency in a FSM?

A) The power consumption may drop below the recommended level.

B) The setup and hold specifications of the D-flip-flops may be violated, which may put the machine into an unwanted state.

C) The states may transition too quickly to be usable.

D) The crystal generating the clock may become unstable.

Summary

- Sequential logic refers to a circuit that bases its outputs on both the present and past values of the inputs. Past values are held in a sequential logic storage device.
- All sequential logic storage devices are based on a cross-coupled feedback loop. The positive feedback loop formed in this configuration will hold either a 1 or a 0. This is known as a bistable device.
- If the inputs of the feedback loop in a sequential logic storage device are driven to exactly between a 1 and a 0 (i.e., $V_{cc}/2$) and then released, the device will go *metastable*. Metastability refers to the behavior where the device will ultimately be pushed toward one of the two stable states in the system, typically by electrical noise. Once the device begins moving toward one of the stable states, the positive feedback will reinforce the transition until it reaches the stable state. The stable state that the device will move toward is random and unknown.
- Cross-coupled inverters are the most basic form of the positive feedback loop configuration. To give the storage device the ability to drive the outputs to known values, the inverters are replaced with NOR gates to form the SR *Latch*. A variety of other modifications can be made to the loop configuration to ultimately produce a D-latch and a D-Flip-Flop.
- A D-Flip-Flop will update its Q output with the value on its D input on every triggering edge of a clock. The amount of time that it takes for the Q output to update after a triggering clock edge is called the "t-clock-to-Q" (t_{CQ}) specification.
- The setup and hold times of a D-flip-flop describe how long before (t_{setup}) and after (t_{hold}) the triggering clock edge the data on the D input of the device must be stable. If the D input transitions too close to the triggering clock edge (i.e., violating a setup or hold specification), then the device will go metastable, and the ultimate value on Q is unknown.
- A synchronous system is one in which all logic transitions occur based on a single timing event. The timing event is typically the triggering edge of a clock.
- There are a variety of common circuits that can be accomplished using just sequential storage devices. Examples of these circuits include switch debouncing, toggle-flops, ripple counters, and shift registers.
- A finite state machine (FSM) is a system that produces outputs based on the current value of the inputs and a history of past inputs. The history of inputs is recorded as *states* that the machine has been in. As the machine responds to new inputs, it transitions between states. This allows a FSM to make more sophisticated decisions about what outputs to produce by knowing its history.
- A state diagram is a graphical way to describe the behavior of an FSM. States are represented using circles, and transitions are represented using arrows. Outputs are listed either inside the state circle or next to the transition arrow.
- A state transition table contains the same information as a state diagram but in tabular format. This allows the system to be more easily synthesized because the information is in a form similar to a truth table.
- The first step in FSM synthesis is creating the *state memory*. The state memory consists of a set of D-Flip-Flops that hold the current state of the FSM. Each state in the FSM must be assigned a binary code. The type of encoding is arbitrary; however, there are certain encoding types that are commonly used, such as binary, gray code, and one-hot. Once the codes are assigned, state variables need to be defined for each bit position for both the current state and the next-state codes. The state variables for the current state represent the Q outputs of the D-flip-flops, which hold the current state code. The state variables for the next-state code represent the D inputs of the D-Flip-Flops. A D-Flip-Flop is needed for each bit in the state code. On the triggering edge of a clock, the current state will be updated with the next-state code.
- The second step in FSM synthesis is creating the *next-state logic*. The next-state logic is combinational logic circuitry that produces the next-state codes based on the current state variables and any system inputs. The next-state logic drives the D inputs of the D-Flip-Flops in the state memory.
- The third step in FSM synthesis is creating the *output logic*. The output logic is combinational logic circuitry that produces the system outputs based on the current state and potentially the system inputs.
- The output logic always depends on the current state of an FSM. If the output logic also depends on the system inputs, the machine

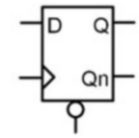
is a *Mealy* machine. If the output logic does not depend on the system inputs, the machine is a *Moore* machine.

- A counter is a special type of FSM in which the states are traversed linearly. The linear progression of states allows the next-state logic to be simplified. The complexity of the output logic in a counter can also be reduced by encoding the states with the desired counter output for that state. This technique, known as *state-encoded outputs,* allows the system outputs to simply be the current state of the FSM.
- The *reset state* of an FSM is the state that the machine will go to when it begins operation. The state code for the reset state must be configured using the reset and/or preset lines of the D-Flip-Flops. If only reset lines are used on the D-Flip-Flops, the reset state must be encoded using only zeros.
- Given the logic diagram for a state machine, the logic expression for the next-state memory and the output logic can be determined by analyzing the combinational logic driving the D inputs of the state memory (i.e., the next-state logic) and the combinational logic driving the system outputs (i.e., the output logic).
- Given the logic diagram for a state diagram, the state diagram can be determined by first finding the logic expressions for the next-state and output logic. The number of D-Flip-Flops in the logic diagram can then be used to calculate the possible number of state codes that the machine has. The state codes are then used to calculate the next-state logic and output values. From this information, a state transition table, and, in turn, the state diagram can be created.
- The maximum frequency of an FSM is found by summing all sources of time delay that must be accounted for before the next triggering edge of the clock can occur. These sources include t_{CQ}, the worst-case combinational logic path, the worst-case interconnect delay path, the setup/hold times of the D-Flip-Flops, and any margin that is to be included. The sum of these timing delays represents the smallest period (T) that the clock can have. This is then converted to frequency.
- If the t_{CQ} time is greater than or equal to the hold time, the hold time can be ignored in the maximum frequency calculation. This is because the outputs of the D-flip-flops are inherently *held* while the D-flip-flops are producing the next output value. The time it takes to change the outputs after a triggering clock edge is defined as t_{CQ}. This means that as long as $t_{CQ} \geq t_{hold}$, the hold time specification is inherently met since the logic driving the next-state codes uses the Q outputs of the D-Flip-Flops.

Exercise Problems

For some of the following exercise problems, you will be asked to design a Verilog model and perform a functional simulation. You will be provided with a test bench for each of these problems. The details of how to create your own Verilog test bench are provided later in Chap. 8. For some of the following exercise problems, you will be asked to use D-Flip-Flops as part of a Verilog design. You will be provided with the model of the D-Flip-Flop and can declare it as a sub-system in your design. The Verilog module definition for a D-Flip-Flop is given in Fig. 7.35. Keep in mind that this D-Flip-Flop has an active LOW reset. This means that when the reset line is pulled to 0, the outputs will go to Q = 0 and Qn = 1. When the reset line is LOW, the incoming clock is ignored. Once the reset line goes HIGH, the D-Flip-Flop resumes normal behavior. The details of how to create your own model of a D-Flip-Flop are provided later in Chap. 8.

Rising Edge Triggered D-Flip-Flop
with Active LOW Reset



dflipflop.v

```
module dflipflop
    (output reg  Q, Qn,
     input  wire Clock, Reset, D);

                    :

endmodule
```

Fig. 7.35
D-Flip-Flop module definition

Section 7.1: Sequential Logic Storage Devices

7.1.1 What does the term *metastability* refer to in a sequential storage device?

7.1.2 What does the term *bistable* refer to in a sequential storage device?

7.1.3 You are given a cross-coupled inverter pair in which all nodes are set to $V_{cc}/2$. Why will this configuration always move to a more stable state?

7.1.4 An SR *Latch* essentially implements the same cross-coupled feedback loop to store information as in a cross-coupled inverter pair. What is the purpose of using NOR gates instead of inverters in the SR *Latch* configuration?

7.1.5 Why isn't the input condition S = R = 1 used in an SR *Latch*?

7.1.6 How will the output Q behave in an SR *Latch* if the inputs continuously switch between S = 0, R = 1 and S = 1, R = 1 every 10 ns?

7.1.7 How do D-Flip-Flops enable synchronous systems?

7.1.8 What signal in the D-flip-flop in Fig. 7.35 has the highest priority?

7.1.9 For the timing diagram shown in Fig. 7.36, draw the outputs Q and Qn for a rising edge-triggered D-Flip-Flop with active LOW reset.

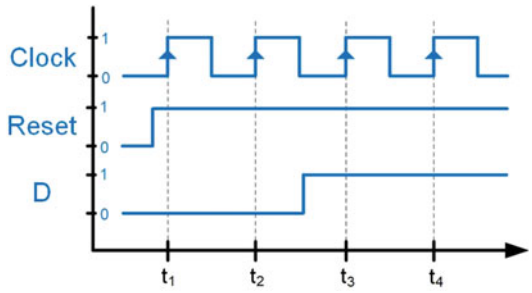


Fig. 7.36
D-Flip-Flop timing exercise 1

7.1.10 For the timing diagram shown in Fig. 7.37, draw the outputs Q and Qn for a rising edge-triggered D-flip-flop with active LOW reset.

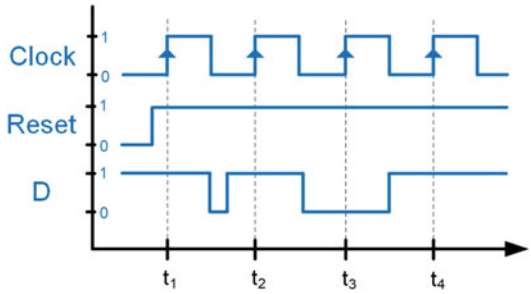


Fig. 7.37
D-Flip-Flop timing exercise 2

7.1.11 For the timing diagram shown in Fig. 7.38, draw the outputs Q and Qn for a rising edge triggered D-flip-flop with active LOW reset.

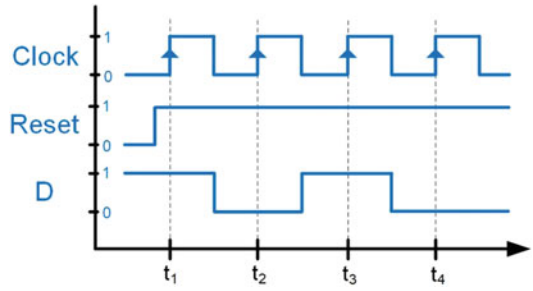


Fig. 7.38
D-Flip-Flop timing exercise 3

Section 7.2: Sequential Logic Timing Considerations

7.2.1 What timing specification is violated in a D-Flip-Flop when the data is not held long enough *before* the triggering clock edge occurs?

7.2.2 What timing specification is violated in a D-Flip-Flop when the data is not held long enough *after* the triggering clock edge occurs?

7.2.3 What is the timing specification for a D-Flip-Flop that describes how long after the triggering clock edge occurs that the new data will be present on the Q output?

7.2.4 What is the timing specification for a D-Flip-Flop that describes how long after the device goes metastable that the outputs will settle to known states?

7.2.5 If the Q output of a D-flip-flop is driving the D input of another D-flip-flop from the same logic family, can the hold time be ignored if it is less than the clock-to-Q delay? Provide an explanation as to why or why not.

Section 7.3: Common Circuits Based on Sequential Storage Devices

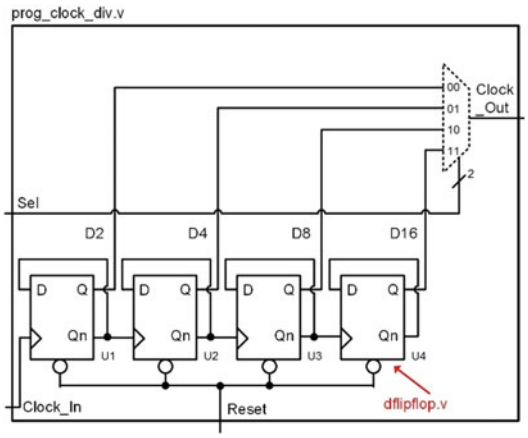
7.3.1 In a Toggle Flop (T-flop) configuration, the Qn output of the D-Flip-Flop is routed back to the D input. This can lead to a hold time violation if the output arrives at the input too quickly. Under what condition(s) is a hold time violation not an issue?

7.3.2 In a Toggle Flop (T-flop) configuration, what timing specifications dictate how quickly the next edge of the incoming clock can occur?

7.3.3 One drawback of a ripple counter is that the delay through the cascade of D-flip-flops can become considerable for large counters. At what point does the delay of a ripple counter prevent it from being useful?

7.3.4 A common use of a ripple counter is in the creation of a 2^n *programmable clock divider*. In a ripple counter, bit(0) has a frequency that is exactly $^1/_2$ of the incoming clock, bit(1) has a frequency that is exactly $^1/_4$ of the incoming

clock, bit(2) has a frequency that is exactly $^1/_8$ of the incoming clock, etc. This behavior can be exploited to create a divided-down output clock that is divided by multiples of 2^n by selecting a particular bit of the counter. The typical configuration of this programmable clock divider is to route each bit of the counter to an input of a multiplexer. The select lines going to the multiplexer choose which bit of the counter is used as the divided-down clock output. This architecture is shown in Fig. 7.39. Design a Verilog model to implement the programmable clock divider shown in this figure. Use the module port definition provided in this figure for your design. Use a 4-bit ripple counter to produce four divided versions of the clock (1/2, 1/4, 1/8, and 1/16). Your system will take in two select lines that will choose which version of the clock is to be routed to the output. Instantiate the D-flip-flop model provided to implement the ripple counter. Implement the 4-to-1 multiplexer using continuous assignment. The multiplexer does not need to be its own subsystem.




```
module prog_clock_div
    (output wire Clock_Out,
     input  wire Clock_In, Reset,
     input  wire Sel[1:0]);
                :
endmodule
```

Fig. 7.39
Programmable clock module description

7.3.5 What phenomenon causes switch bounce in an SPST switch?

7.3.6 What two phenomena cause switch bounce in an SPDT switch?

Section 7.4: Finite-State Machines

7.4.1 For the state diagram in Fig. 7.40, how many D-Flip-Flops will this machine take if the states are encoded in *binary*?

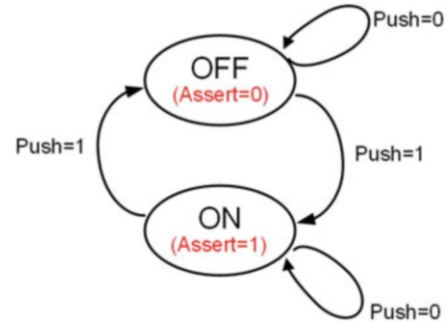


Fig. 7.40
Finite state machine 0 state diagram

7.4.2 For the state diagram in Fig. 7.40, how many D-Flip-Flops will this machine take if the states are encoded in *gray code*?

7.4.3 For the state diagram in Fig. 7.40, how many D-Flip-Flops will this machine take if the states are encoded in *one-hot*?

7.4.4 For the state diagram in Fig. 7.40, is this a Mealy or Moore machine?

The next set of questions is about the design of an FSM by hand to implement the behavior described by the state diagram in Fig. 7.40. For this design, you will name the current state variable Q0_cur and the next-state variable Q0_nxt. You will also use the following state codes:

OFF = '0'
ON = '1'

7.4.5 For the state diagram in Fig. 7.40, what is the next-state logic expression for Q0_nxt?

7.4.6 For the state diagram in Fig. 7.40, what is the output logic expression for assert?

7.4.7 For the state diagram in Fig. 7.40, provide the final logic diagram for this machine.

7.4.8 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.40. Use the module port definition provided in Fig. 7.41 for your design. Name the current state variables Q1_cur and Q0_cur, and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with logical operators* for the implementation of your next-state and output logic.

fsm0.v

```
module fsm0
    (output wire Assert,
     input  wire Clock, Reset,
     input  wire Push);
                  :
endmodule
```

Fig. 7.41
FSM 0 module definition

7.4.9 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.40. Use the module port definition provided in Fig. 7.41 for your design. Name the current state variables Q1_cur and Q0_cur, and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with conditional operators* for the implementation of your next-state and output logic.

7.4.10 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.40. Use the module port definition provided in Fig. 7.41 for your design. Name the current state variables Q1_cur and Q0_cur and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *User-Defined Primitives* for the implementation of your next-state and output logic.

7.4.11 For the state diagram in Fig. 7.42, how many D-Flip-Flops will this machine take if the states are encoded in *binary*?

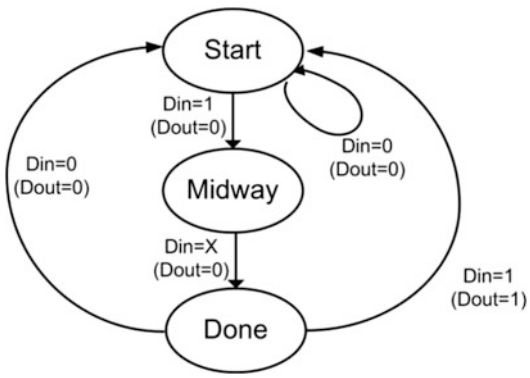


Fig. 7.42
Finite state machine 1 state diagram

7.4.12 For the state diagram in Fig. 7.42, how many D-Flip-Flops will this machine take if the states are encoded in *gray code*?

7.4.13 For the state diagram in Fig. 7.42, how many D-Flip-Flops will this machine take if the states are encoded in *one-hot*?

7.4.14 For the state diagram in Fig. 7.42, is this a Mealy or Moore machine?

The next set of questions is about the design of an FSM by hand to implement the behavior described by the state diagram in Fig. 7.42. For this design, you will name the current state variables Q1_cur and Q0_cur and the next-state variables Q1_nxt and Q0_nxt. You will also use the following state codes:

Start = "00"
Midway = "01"
Done = "10"

7.4.15 For the state diagram in Fig. 7.42, what is the next-state logic expression for Q1_nxt?

7.4.16 For the state diagram in Fig. 7.42, what is the next-state logic expression for Q0_nxt?

7.4.17 For the state diagram in Fig. 7.42, what is the output logic expression for Dout?

7.4.18 For the state diagram in Fig. 7.42, provide the final logic diagram for this machine.

7.4.19 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.42. Use the module port definition provided in Fig. 7.43 for your design. Name the current state variables Q1_cur and Q0_cur and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with logical operators* for the implementation of your next-state and output logic.

fsm1.v

```
module fsm1
    (output wire Dout,
     input  wire Clock, Reset,
     input  wire Din);
                  :
endmodule
```

Fig. 7.43
Finite state machine 1 module description

7.4.20 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.42. Use the module port definition provided in Fig. 7.43 for your design. Name the current state variables Q1_cur and Q0_cur and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with conditional operators* for the implementation of your next-state and output logic.

7.4.21 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.42. Use the module port definition provided in Fig. 7.43 for your design. Name the current state variables Q1_cur and

Q0_cur and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *User-Defined Primitives* for the implementation of your next-state and output logic.

7.4.22 For the state diagram in Fig. 7.44, how many D-Flip-Flops will this machine take if the states are encoded in *binary*?

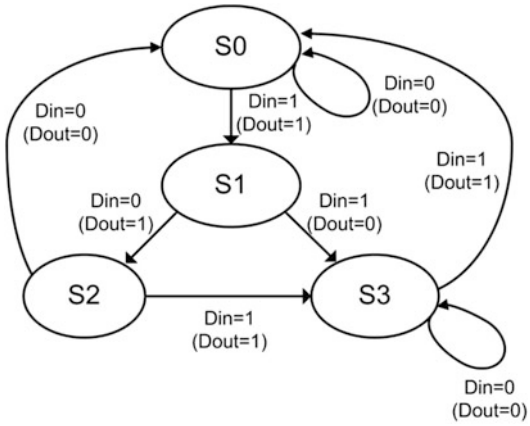


Fig. 7.44
Finite state machine 2 state diagram

7.4.23 For the state diagram in Fig. 7.44, how many D-Flip-Flops will this machine take if the states are encoded in *gray code*?

7.4.24 For the state diagram in Fig. 7.44, how many D-Flip-Flops will this machine take if the states are encoded in *one-hot*?

7.4.25 For the state diagram in Fig. 7.44, is this a Mealy or Moore machine?

The next set of questions are about the design of an FSM by hand to implement the behavior described by the state diagram in Fig. 7.44. For this design, you will name the current state variables Q1_cur and Q0_cur and name the next-state variables Q1_nxt and Q0_nxt. You will also use the following state codes:

S0 = "00"
S1 = "01"
S2 = "10"
S3 = "11"

7.4.26 For the state diagram in Fig. 7.44, what is the next-state logic expression for Q1_nxt?

7.4.27 For the state diagram in Fig. 7.44, what is the next-state logic expression for Q0_nxt?

7.4.28 For the state diagram in Fig. 7.44, what is the output logic expression for Dout?

7.4.29 For the state diagram in Fig. 7.44, provide the final logic diagram for this machine.

7.4.30 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.44. Use the module port definition provided in Fig. 7.45 for your design. Name the current state variables Q1_cur and Q0_cur, and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with logical operators* for the implementation of your next-state and output logic.

fsm2.v

```
module fsm2
    (output wire Dout,
     input  wire Clock, Reset,
     input  wire Din);
                  :
endmodule
```

Fig. 7.45
Finite state machine 2 module description

7.4.31 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.44. Use the module port definition provided in Fig. 7.45 for your design. Name the current state variables Q1_cur and Q0_cur, and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with conditional operators* for the implementation of your next-state and output logic.

7.4.32 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.44. Use the module port definition provided in Fig. 7.45 for your design. Name the current state variables Q1_cur and Q0_cur, and name the next-state variables Q1_nxt and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *User-Defined Primitives* for the implementation of your next-state and output logic.

7.4.33 For the state diagram in Fig. 7.46, how many D-Flip-Flops will this machine take if the states are encoded in *binary*?

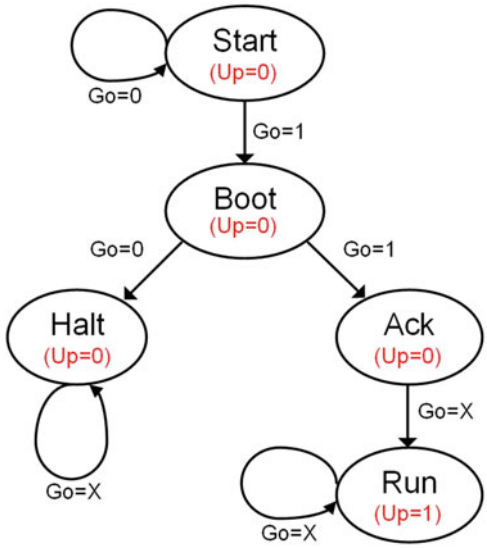


Fig. 7.46
Finite state machine 3 state diagram

7.4.34 For the state diagram in Fig. 7.46, how many D-Flip-Flops will this machine take if the states are encoded in *gray code*?

7.4.35 For the state diagram in Fig. 7.46, how many D-Flip-Flops will this machine take if the states are encoded in *one-hot*?

7.4.36 For the state diagram in Fig. 7.46, is this a Mealy or Moore machine?

The next set of questions is about the design of an FSM by hand to implement the behavior described by the state diagram in Fig. 7.46. For this design, you will name the current state variables Q2_cur, Q1_cur, and Q0_cur and the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. You will also use the following state codes:

Start = "000"
Boot = "001"
Halt = "010"
Ack = "011"
Run = "100"

7.4.37 For the state diagram in Fig. 7.46, what is the next-state logic expression for Q2_nxt?

7.4.38 For the state diagram in Fig. 7.46, what is the next-state logic expression for Q1_nxt?

7.4.39 For the state diagram in Fig. 7.46, what is the next-state logic expression for Q0_nxt?

7.4.40 For the state diagram in Fig. 7.46, what is the output logic expression for Up?

7.4.41 For the state diagram in Fig. 7.46, provide the final logic diagram for this machine.

7.4.42 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.46. Use the module port definition provided in Fig. 7.47 for your design. Name the current state variables Q2_cur, Q1_cur, and Q0_cur, and name the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with logical operators* for the implementation of your next-state and output logic.

fsm3.v

```
module fsm3
    (output wire Up,
     input  wire Clock, Reset,
     input  wire Go);
                  :
endmodule
```

Fig. 7.47
Finite-state machine 3 module description

7.4.43 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.46. Use the module port definition provided in Fig. 7.47 for your design. Name the current state variables Q2_cur, Q1_cur, and Q0_cur, and name the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *continuous assignment with conditional operators* for the implementation of your next-state and output logic.

7.4.44 Design a Verilog model to implement the behavior described by the state diagram in Fig. 7.46. Use the module port definition provided in Fig. 7.47 for your design. Name the current state variables Q2_cur, Q1_cur, and Q0_cur, and name the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. Instantiate the D-Flip-Flop model provided to implement your state memory. Use *User-Defined Primitives* for the implementation of your next-state and output logic.

The next set of questions is about the design of a 4-bit serial bit sequence detector by hand similar to the one described in Example 7.9. The input to your state detector is called *DIN* and the output is called *FOUND*. Your detector will assert FOUND anytime there is a 4-bit sequence of "0101." For all other input sequences, the output is not asserted.

7.4.45 For your 4-bit serial bit sequence detector, provide the *state diagram* for this FSM.

7.4.46 For your 4-bit serial bit sequence detector, how many D-Flip-Flops does it take to implement the state memory for this FSM if you encode your states in *binary*?

7.4.47 For your 4-bit serial bit sequence detector, provide the *state transition table* for this FSM.

7.4.48 For your 4-bit serial bit sequence detector, synthesize and provide the combinational logic expressions for the *next-state logic*.

7.4.49 For your 4-bit serial bit sequence detector, synthesize and provide the combinational logic expression for the *output logic* for FOUND.

7.4.50 For your 4-bit serial bit sequence detector, is this FSM a Mealy or Moore machine?

7.4.51 For your 4-bit serial bit sequence detector, provide the *logic diagram* for this FSM.

The next set of questions is about the design of a 20-cent vending machine controller by hand similar to the one described in Example 7.12. Your controller will take in nickels and dimes and dispense a product anytime the customer has entered 20 cents. Your FSM has two inputs, *Nin* and *Din*. Nin is asserted whenever the customer enters a nickel while Din is asserted anytime the customer enters a dime. Your FSM has two outputs, *Dispense* and *Change*. Dispense is asserted anytime the customer has entered at least 20 cents and Change is asserted anytime the customer has entered more than 20 cents and needs a nickel in change.

counter_3bit_binary_up.v

```
module counter_3bit_binary_up
    (output wire Count[2:0],
     input  wire Clock, Reset);
                :
endmodule
```

Fig. 7.48
3-Bit binary up counter module definition

7.4.52 For your 20-cent vending machine controller, provide the state diagram for this FSM.

7.4.53 For your 20-cent vending machine controller, how many D-Flip-Flops does it take to implement the state memory for this FSM if you encode your states in *binary*?

7.4.54 For your 20-cent vending machine controller, provide the *state transition table* for this FSM.

7.4.55 For your 20-cent vending machine controller, synthesize and provide the combinational logic expressions for the *next-state logic*.

7.4.56 For your 20-cent vending machine controller, synthesize and provide the combinational logic expression for the *output logic* for Dispense and Change.

7.4.57 For your 20-cent vending machine controller, is this FSM a Mealy or Moore machine?

7.4.58 For your 20-cent vending machine controller, provide the *logic diagram* for this FSM.

The next set of questions is about the design of an FSM by hand that controls a traffic light at the intersection of a busy highway and a seldom used side road. You will be designing the control signals for just the red, yellow, and green lights facing the highway. Under normal conditions, the highway has a green light. The side road has a car detector that indicates when a car pulls up by asserting a signal called *CAR*. When CAR is asserted, you will change the highway traffic light from green to yellow. Once yellow, you will always go to red. Once in the red position, a built-in timer will begin a countdown and provide your controller a signal called *TIMEOUT* when 15 seconds has passed. Once TIMEOUT is asserted, you will change the highway traffic light back to green. Your system will have three outputs *GRN*, *YLW*, and *RED* that control when the highway facing traffic lights are on (1 = ON, 0 = OFF).

7.4.59 For your traffic light controller, provide the state diagram for this FSM.

7.4.60 For your traffic light controller, how many D-Flip-Flops does it take to implement the state memory for this FSM if you encode your states in *binary*?

7.4.61 For your traffic light controller, provide the *state transition table* for this FSM.

7.4.62 For your traffic light controller, synthesize and provide the combinational logic expressions for the *next-state logic*.

7.4.63 For your traffic light controller, synthesize and provide the combinational logic expression for the *output logic* for GRN, YLW, and RED.

7.4.64 For your traffic light controller, is this FSM a Mealy or Moore machine?

7.4.65 For your traffic light controller, provide the *logic diagram* for this FSM.

Section 7.5: Counters

The next set of questions is about the design a *3-bit binary up counter* by hand. This state machine will need eight states and require 3-bits for the state variable codes. Name the current state variables Q2_cur, Q1_cur, and Q0_cur and the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. The output of your counter will be a 3-bit vector called *Count*.

7.5.1 For your 3-bit binary up counter, what is the next-state logic expression for Q2_nxt?

7.5.2 For your 3-bit binary up counter, what is the next-state logic expression for Q1_nxt?

7.5.3 For your 3-bit binary up counter, what is the next-state logic expression for Q0_nxt?

7.5.4 For your 3-bit binary up counter, what is the output logic expression for Count(2)?

7.5.5 For your 3-bit binary up counter, what is the output logic expression for Count(1)?

7.5.6 For your 3-bit binary up counter, what is the output logic expression for Count(0)?

7.5.7 For your 3-bit binary up counter, provide the logic diagram.

7.5.8 Design a Verilog model for a *3-bit binary up counter*. Instantiate the D-Flip-Flop model provided to implement your state memory.

Use whatever concurrent modeling approach you wish to model the next-state and output logic. Use the module port definition provided in Fig. 7.48 for your design.

The next set of questions is about the design a *3-bit binary up/down counter* by hand. The counter will have an input called *Up* that will dictate the direction of the counter. When Up = 1, the counter should increment and when Up = 0 it should decrement. This state machine will need eight states and require three bits for the state variable codes. Name the current state variables Q2_cur, Q1_cur, and Q0_cur and the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. The output of your counter will be a 3-bit vector called *Count*.

7.5.9 For your 3-bit binary up/down counter, what is the next-state logic expression for Q2_nxt?

7.5.10 For your 3-bit binary up/down counter, what is the next-state logic expression for Q1_nxt?

7.5.11 For your 3-bit binary up/down counter, what is the next-state logic expression for Q0_nxt?

7.5.12 For your 3-bit binary up/down counter, what is the output logic expression for Count(2)?

7.5.13 For your 3-bit binary up/down counter, what is the output logic expression for Count(1)?

7.5.14 For your 3-bit binary up/down counter, what is the output logic expression for Count(0)?

7.5.15 For your 3-bit binary up/down counter, provide the logic diagram.

7.5.16 Design a Verilog model for a *3-bit binary up/down counter*. Instantiate the D-Flip-Flop model provided to implement your state memory. Use whatever concurrent modeling approach you wish to model the next-state and output logic. Use the module port definition provided in Fig. 7.49 for your design.

The next set of questions is about the design a *3-bit gray code up counter* by hand. This counter will create a pattern of 000→001→011→010→110→111→101→100 and then repeat. This state machine will need eight states and require 3-bits for the state variable codes. Name the current state variables Q2_cur, Q1_cur, and Q0_cur and the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. The output of your counter will be a 3-bit vector called *Count*.

counter_3bit_binary_up_down.v

```
module counter_3bit_binary_up_down
    (output wire Count[2:0],
     input  wire Clock, Reset,
     input  wire Up);
                :
endmodule
```

Fig. 7.49
3-Bit binary up/down counter module definition

7.5.17 For your 3-bit gray code up counter, what is the next-state logic expression for Q2_nxt?

7.5.18 For your 3-bit gray code up counter, what is the next-state logic expression for Q1_nxt?

7.5.19 For your 3-bit gray code up counter, what is the next-state logic expression for Q0_nxt?

7.5.20 For your 3-bit gray code up counter, what is the output logic expression for Count(2)?

7.5.21 For your 3-bit gray code up counter, what is the output logic expression for Count(1)?

7.5.22 For your 3-bit gray code up counter, what is the output logic expression for Count(0)?

7.5.23 For your 3-bit gray code up counter, provide the logic diagram.

7.5.24 Design a Verilog model for a *3-bit gray code up counter*. Instantiate the D-Flip-Flop model provided to implement your state memory. Use whatever concurrent modeling approach you wish to model the next-state and output logic. Use the module port definition provided in Fig. 7.50 for your design.

The next set of questions is about the design a *3-bit gray code up/down counter* by hand. The counter will have an input called *Up* that will dictate the direction of the counter. When Up = 1, the counter should increment and when Up = 0 it should decrement. When incrementing up, the counter will create the pattern 000→001→011→010→110→111→101→100 and then repeat. This state machine will need eight states and require 3-bits for the state variable codes. Name the current state variables Q2_cur, Q1_cur, and Q0_cur and the next-state variables Q2_nxt, Q1_nxt, and Q0_nxt. The output of your counter will be a 3-bit vector called *Count*.

counter_3bit_graycode_up.v

```
module counter_3bit_graycode_up
    (output wire Count[2:0],
     input  wire Clock, Reset);
                :
endmodule
```

Fig. 7.50
3-Bit gray code up counter module definition

7.5.25 For your 3-bit gray code up/down counter, what is the next-state logic expression for Q2_nxt?

7.5.26 For your 3-bit gray code up/down counter, what is the next-state logic expression for Q1_nxt?

7.5.27 For your 3-bit gray code up/down counter, what is the next-state logic expression for Q0_nxt?

7.5.28 For your 3-bit gray code up/down counter, what is the output logic expression for Count(2)?

7.5.29 For your 3-bit gray code up/down counter, what is the output logic expression for Count(1)?

7.5.30 For your 3-bit gray code up/down counter, what is the output logic expression for Count(0)?

7.5.31 For your 3-bit gray code up/down counter, provide the logic diagram.

7.5.32 Design a Verilog model for a *3-bit gray code up/down counter*. Instantiate the D-Flip-Flop model provided to implement your state memory. Use whatever concurrent modeling approach you wish to model the next-state and output logic. Use the module port definition provided in Fig. 7.51 for your design.

The next set of questions is about the design a *4-bit one hot up counter* by hand. This counter will create a pattern of 0001 → 0010 → 0100 → 1000 and then repeat. This FSM should use state-encoded outputs to simplify the output logic. This state machine will need four states and require four bits for the state variable codes. Name the current state variables Q3_cur, Q2_cur, Q1_cur, and Q0_cur and the next-state variables Q3_nxt, Q2_nxt, Q1_nxt, and Q0_nxt. The output of your counter will be a 4-bit vector called *Count*.

counter_3bit_graycode_up_down.v

```
module counter_3bit_graycode_up_down
    (output wire Count[2:0],
     input  wire Clock, Reset,
     input  wire Up);
                  :
endmodule
```

Fig. 7.51
3-Bit gray code up/down counter module definition

7.5.33 For your 4-bit one-hot up counter, what is the next-state logic expression for Q3_nxt?

7.5.34 For your 4-bit one-hot up counter, what is the next-state logic expression for Q2_nxt?

7.5.35 For your 4-bit one-hot up counter, what is the next-state logic expression for Q1_nxt?

7.5.36 For your 4-bit one-hot up counter, what is the next-state logic expression for Q0_nxt?

7.5.37 For your 4-bit one-hot up counter, what is the output logic expression for Count(3)?

7.5.38 For your 4-bit one-hot up counter, what is the output logic expression for Count(2)?

7.5.39 For your 4-bit one-hot up counter, what is the output logic expression for Count(1)?

7.5.40 For your 4-bit one-hot up counter, what is the output logic expression for Count(0)?

7.5.41 For your 4-bit one-hot up counter, provide the logic diagram.

7.5.42 Design a Verilog model for a *4-bit one-hot up counter*. Instantiate the D-Flip-Flop model provided to implement your state memory. Use whatever concurrent signal assignment modeling approach you wish to model the next-state and output logic. Use the Verilog module definition provided in Fig. 7.52 for your design.

counter_4bit_onehot_up.v

```
module counter_4bit_onehot_up
    (output wire Count[3:0],
     input  wire Clock, Reset);
                  :
endmodule
```

Fig. 7.52
4-Bit one-hot up counter module description

The next set of questions is about the design a *binary counter that goes from 0000_2 (0_{10}) to 1001_2 (9_{10})* by hand. This counter will create a pattern of 0000→0001→0010→0011→0100→0101→0110→0111→1000→1001 and then repeat. This type of counter is known as a *binary coded decimal* (BCD) counter. This FSM should use state-encoded outputs to simplify the output logic. This state machine will need 10 states and require four bits for the state variable codes. Name the current state variables Q3_cur, Q2_cur, Q1_cur, and Q0_cur and the next-state variables Q3_nxt, Q2_nxt, Q1_nxt, and Q0_nxt. The output of your counter will be a 4-bit vector called *Count*.

7.5.43 For your BCD counter, what is the next-state logic expression for Q3_nxt?

7.5.44 For your BCD counter, what is the next-state logic expression for Q2_nxt?

7.5.45 For your BCD counter, what is the next-state logic expression for Q1_nxt?

7.5.46 For your BCD counter, what is the next-state logic expression for Q0_nxt?

7.5.47 For your BCD counter, what is the output logic expression for Count(3)?

7.5.48 For your BCD counter, what is the output logic expression for Count(2)?

7.5.49 For your BCD counter, what is the output logic expression for Count(1)?

7.5.50 For your BCD counter, what is the output logic expression for Count(0)?

7.5.51 For your BCD counter, provide the logic diagram.

Section 7.6: Finite-State Machine's Reset Condition

7.6.1 Are resets typically synchronous or asynchronous?

7.6.2 Why is it necessary to have a reset/preset condition in an FSM?

7.6.3 How does the reset/preset condition correspond to the behavior described in the state diagram?

7.6.4 When is it necessary to also use the preset line(s) of a D-Flip-Flop instead of just the reset line(s) when implementing the state memory of an FSM?

7.6.5 If an FSM has eight unique states that are encoded in binary and all D-Flip-Flops used for the state memory use their reset lines, what is the state code that the machine will go to upon reset?

Section 7.7: Sequential Logic Analysis

7.7.1 For the FSM logic diagram in Fig. 7.53, give the next-state logic expression for *Q_nxt*.

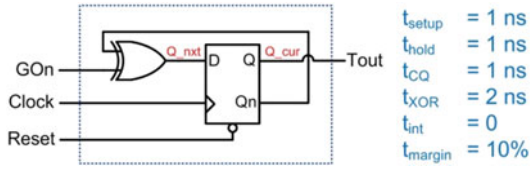


Fig. 7.53
Sequential logic analysis 1

7.7.2 For the FSM logic diagram in Fig. 7.53, give the output logic expression for *Tout*.

7.7.3 For the FSM logic diagram in Fig. 7.53, give the state transition table.

7.7.4 For the FSM logic diagram in Fig. 7.53, give the state diagram.

7.7.5 For the FSM logic diagram in Fig. 7.53, give the maximum clock frequency.

7.7.6 For the FSM logic diagram in Fig. 7.54, give the next-state logic expression for *Q_nxt*.

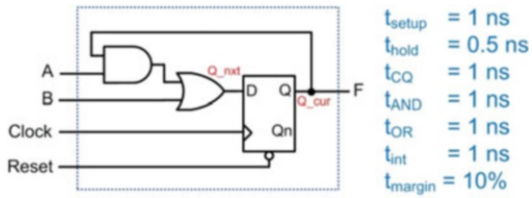


Fig. 7.54
Sequential logic analysis 2

7.7.7 For the FSM logic diagram in Fig. 7.54, give the output logic expression for *F*.

7.7.8 For the FSM logic diagram in Fig. 7.54, give the state transition table.

7.7.9 For the FSM logic diagram in Fig. 7.54, give the state diagram.

7.7.10 For the FSM logic diagram in Fig. 7.54, give the maximum clock frequency.

7.7.11 For the FSM logic diagram in Fig. 7.55, give the next-state logic expressions for *Q1_nxt* and *Q0_nxt*.

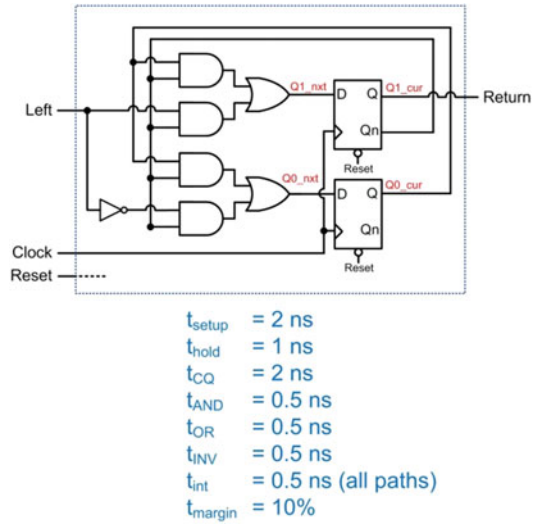


Fig. 7.55
Sequential logic analysis 3

7.7.12 For the FSM logic diagram in Fig. 7.55, give the output logic expression for *Return*.

7.7.13 For the FSM logic diagram in Fig. 7.55, give the state transition table.

7.7.14 For the FSM logic diagram in Fig. 7.55, give the state diagram.

7.7.15 For the FSM logic diagram in Fig. 7.55, give the maximum clock frequency.