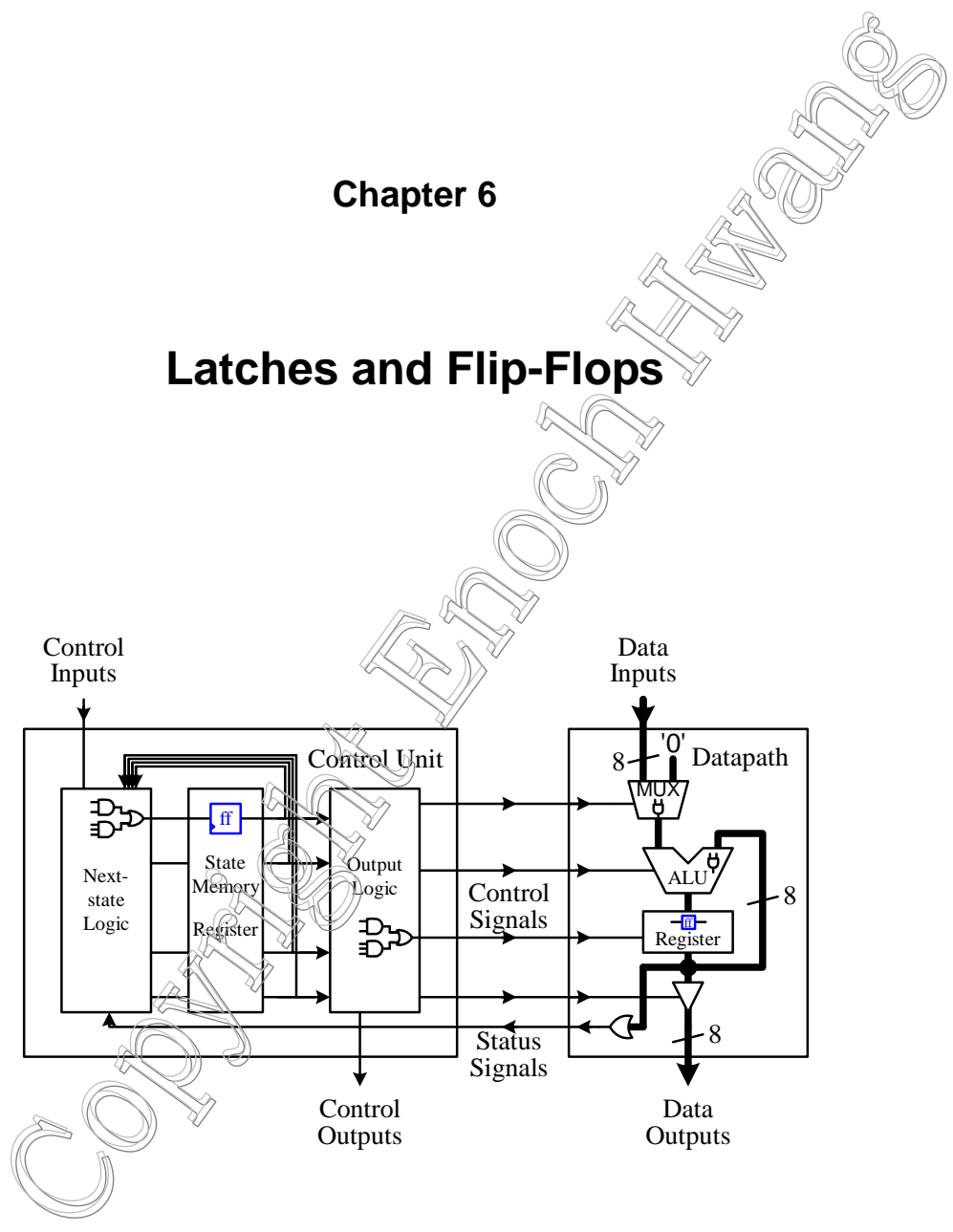


Contents

Latches and Flip-Flops	2
6.1 Bistable Element	3
6.2 SR Latch	4
6.3 SR Latch with Enable	6
6.4 D Latch	7
6.5 D Latch with Enable	8
6.6 Clock	9
6.7 D Flip-Flop	10
6.7.1 * Alternative Smaller Circuit	11
6.8 D Flip-Flop with Enable	12
6.9 Asynchronous Inputs	13
6.10 Description of a Flip-Flop	14
6.10.1 Characteristic Table	14
6.10.2 Characteristic Equation	14
6.10.3 State Diagram	15
6.10.4 Excitation Table	15
6.11 * Timing Issues	15
6.12 Designing a Car Security System—Version 2	16
6.13 VHDL for Latches and Flip-Flops	17
6.13.1 Implied Memory Element	17
6.13.2 VHDL Code for a D Latch with Enable	18
6.13.3 VHDL Code for a D Flip-Flop	18
6.13.4 VHDL Code for a D Flip-Flop with Enable and Asynchronous Set and Clear	21
6.14 * Other Flip-Flop Types	22
6.14.1 SR Flip-Flop	22
6.14.2 JK Flip-Flop	23
6.14.3 T Flip-Flop	24
6.15 Summary Checklist	25
6.16 Problems	25
Index	27

Chapter 6

Latches and Flip-Flops



So far, we have been looking at the design of combinational circuits. We will now turn our attention to the design of **sequential circuits**. Recall that the outputs of sequential circuits are dependent on not only their current inputs (as in combinational circuits), but also on all their past inputs. Because of this necessity to remember the history of inputs, sequential circuits must contain memory elements.

The car security system from Section 2.9 is an example of a combinational circuit. In that example, the siren is turned on when the master switch is on and someone opens the door. If you close the door afterwards, then the siren will turn off immediately. For a more realistic car security system, we would like the siren to remain on even if you close the door after it was first triggered. In order for this modified system to work correctly, the siren must be dependent on not only the master switch and the door switch but also on whether the siren is currently on or off. In other words, this modified system is a sequential circuit that is dependent on both the current and on the past inputs to the system.

In order to remember this history of inputs, sequential circuits must have memory elements. Memory elements, however, are just like combinational circuits in the sense that they are made up of the same basic logic gates. What makes them different is in the way these logic gates are connected together. In order for a circuit to “remember” its current value, we have to connect the output of a logic gate directly or indirectly back to the input of that same gate. We call this a **feedback loop** circuit, and it forms the basis for all memory elements. Combinational circuits do not have any feedback loops.

Latches and **flip-flops** are the basic memory elements for storing information. Hence, they are the fundamental building blocks for all sequential circuits. A single latch or flip-flop can store only one bit of information. This bit of information that is stored in a latch or flip-flop is referred to as the **state** of the latch or flip-flop. Hence, a single latch or flip-flop can be in either one of two states: 0 or 1. We say that a latch or a flip-flop changes state when its content changes from a 0 to a 1 or vice versa. This state value is always available at the output. Consequently, the content of a latch or a flip-flop is the state value, and is always equal to its output value.

The main difference between a latch and a flip-flop is that for a latch, its state or output is constantly affected by its input as long as its enable signal is asserted. In other words, when a latch is enabled, its state changes immediately when its input changes. When a latch is disabled, its state remains constant, thereby, remembering its previous value. On the other hand, a flip-flop changes state only at the active edge of its enable signal, i.e., at precisely the moment when either its enable signal rises from a 0 to a 1 (referred to as the rising edge of the signal), or from a 1 to a 0 (the falling edge). However, after the rising or falling edge of the enable signal, and during the time when the enable signal is at a constant 1 or 0, the flip-flop’s state remains constant even if the input changes.

In a microprocessor system, we usually want changes to occur at precisely the same moment. Hence, flip-flops are used more often than latches, since they can all be synchronized to change only at the active edge of the enable signal. This enable signal for the flip-flops is usually the global controlling clock signal.

Historically, there are basically four main types of flip-flops: SR, D, JK, and T. The major differences between them are the number of inputs they have and how their contents change. Any given sequential circuit can be built using any of these types of flip-flops (or combinations of them). However, selecting one type of flip-flop over another type to use in a particular sequential circuit can affect the overall size of the circuit. Today, sequential circuits are designed mainly with D flip-flops because of their ease of use. This is simply a tradeoff issue between ease of circuit design versus circuit size. Thus, we will focus mainly on the D flip-flop. Discussions about the other types of flip-flops can be found in Section 6.14.

In this chapter, we will look at how latches and flip-flops are designed and how they work. Since flip-flops are at the heart of all sequential circuits, a good understanding of their design and operation is very important in the design of microprocessors.

6.1 Bistable Element

Let us look at the inverter. If you provide the inverter input with a 1, the inverter will output a 0. If you do not provide the inverter with an input (that is neither a 0 nor a 1), the inverter will not have a value to output. If you want to construct a memory circuit using the inverter, you would want the inverter to continue to output the 0 even after you remove the 1 input. In order for the inverter to continue to output a 0, you need the inverter to self-provide its own input. In other words, you want the output to feed back the 0 to the input. However, you cannot connect the output of the inverter directly to its input, because you will have a 0 connected to a 1 and so creating a short circuit.

The solution is to connect two inverters in series, as shown in Figure 6.1. This circuit is called a **bistable element**, and it is the simplest memory circuit. The bistable element has two symmetrical nodes labeled Q and Q' , both of which can be viewed as either an input or an output signal. Since Q and Q' are symmetrical, we can arbitrarily use Q as the state variable, so that the state of the circuit is the value at Q . Let us assume that Q originally has the value 0 when power is first applied to the circuit. Since Q is the input to the bottom inverter, therefore Q' is a 1. A 1 going to the input of the top inverter will produce a 0 at the output Q , which is what we started off with. Hence, the value at Q will remain at a 0 indefinitely. Similarly, if we power-up the circuit with $Q = 1$, we will get $Q' = 0$, and again, we get a stable situation with Q remaining at a 1 indefinitely. Thus, the circuit has two stable states: $Q = 0$ and $Q = 1$; hence, the name “bistable.”

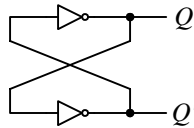


Figure 6.1 Bistable element circuit.

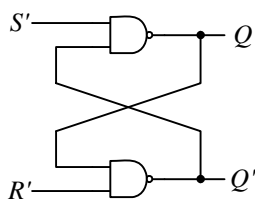
We say that the bistable element has memory because it can remember its state (i.e., keep the value at Q constant) indefinitely. Unfortunately, we cannot change its state (i.e., cannot change the value at Q). We cannot just input a different value to Q , because it will create a short circuit by connecting a 0 to a 1. For example, let us assume that Q is currently 0. If we want to change the state, we need to set Q to a 1, but in so doing we will be connecting a 1 to a 0, thus creating a short. Another way of looking at this problem is that we can think of both Q and Q' as being the primary outputs, which means that the circuit does not have any external inputs. Therefore, there is no way for us to input a different value.

6.2 SR Latch

In order to change the state for the bistable element, we need to add external inputs to the circuit. The simplest way to add extra inputs is to replace the two inverters with two NAND gates, as shown in Figure 6.2(a). This circuit is called an **SR latch**. In addition to the two outputs Q and Q' , there are two inputs S' and R' for *set* and *reset*, respectively. Just like the bistable element, the SR latch can be in one of two states: a set state when $Q = 1$, or a reset state when $Q = 0$. Following the convention, the primes in S and R denote that these inputs are active-low (i.e., a 0 asserts them, and a 1 de-asserts them).

To make the SR latch go to the set state, we simply assert the S' input by setting it to 0 (and de-asserting R'). It doesn't matter what the other NAND gate input is, because 0 NAND anything gives a 1, hence $Q = 1$, and the latch is set. If S' remains at 0 so that Q (which is connected to one input of the bottom NAND gate) remains at 1, and if we now de-assert R' (i.e., set R' to a 1) then the output of the bottom NAND gate will be 0, and so, $Q' = 0$. This situation is shown in Figure 6.2(d) at time t_0 . From this current situation, if we now de-assert S' so that $S' = R' = 1$, the latch will remain in the set state because Q' (the second input to the top NAND gate) is 0, which will keep $Q = 1$, as shown at time t_1 . At time t_2 , we reset the latch by making $R' = 0$ (and $S' = 1$). With R' being a 0, Q' will go to a 1. At the top NAND gate, 1 NAND 1 is 0, thus forcing Q to go to 0. If we de-assert R' next so that, again, we have $S' = R' = 1$, this time the latch will remain in the reset state, as shown at time t_3 .

Notice the two times (at t_1 and t_3) when both S' and R' are de-asserted (i.e., $S' = R' = 1$). At t_1 , Q is at a 1; whereas, at t_3 , Q is at a 0. Why is this so? What is different between these two times? The difference is in the value of Q immediately before those times. The value of Q right before t_1 is 1; whereas, the value of Q right before t_3 is 0. When both inputs are de-asserted, the SR latch remembers its previous state. Previous to t_1 , Q has the value 1, so at t_1 , Q remains at a 1. Similarly, previous to t_3 , Q has the value 0, so at t_3 , Q remains at a 0.



S'	R'	Q	Q_{next}	Q'_{next}
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

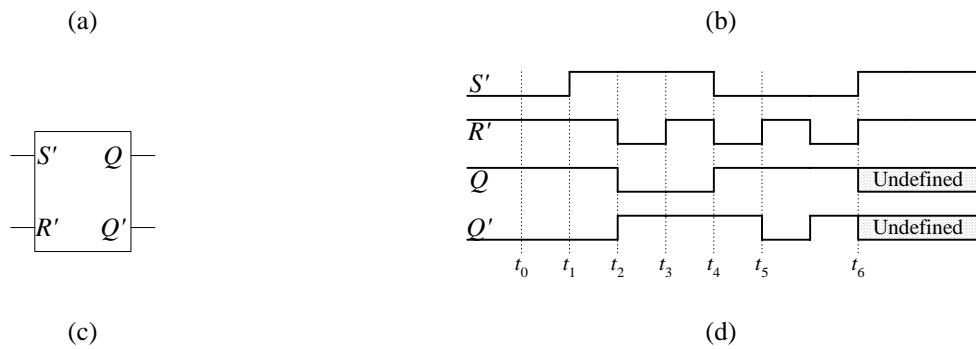


Figure 6.2 SR latch: (a) circuit using NAND gates; (b) truth table; (c) logic symbol; (d) sample trace.

If both S' and R' are asserted (i.e., $S' = R' = 0$), then both Q and Q' are equal to a 1, as shown at time t_4 , since 0 NAND anything gives a 1. Note that there is nothing wrong with having Q equal to Q' . It is just because we named these two points Q and Q' that we don't like them to be equal. However, we could have used another name say, P instead of Q' .

If one of the input signals is de-asserted earlier than the other, the latch will end up in the state forced by the signal that is de-asserted later, as shown at time t_5 . At t_5 , R' is de-asserted first, so the latch goes into the set state with $Q = 1$, and $Q' = 0$.

A problem exists if both S' and R' are de-asserted at *exactly* the same time, as shown at time t_6 . Let us assume for a moment that both gates have exactly the same delay and that the two wire connections between the output of one gate to the input of the other gate also have exactly the same delay. Currently, both Q and Q' are at a 1. If we set S' and R' to a 1 at exactly the same time, then both NAND gates will perform a 1 NAND 1 and will both output a 0 at exactly the same time. The two 0's will be fed back to the two gate inputs at exactly the same time, because the two wire connections have the same delay. This time around, the two NAND gates will perform a 1 NAND 0 and will both produce a 1 again at exactly the same time. This time, two 1's will be fed back to the inputs, which again will produce a 0 at the outputs, and so on and on. This oscillating behavior, called the **critical race**, will continue indefinitely until one outpaces the other. If the two gates do not have exactly the same delay, then the situation is similar to de-asserting one input before the other, and so, the latch will go into one state or the other. However, since we do not know which is the faster gate, we do not know which state the latch will end up in. Thus, the latch's next state is undefined.

Of course, in practice, it is next to impossible to manufacture two gates and make the two connections with precisely the same delay. In addition, both S' and R' need to be de-asserted at exactly the same time. Nevertheless, if this circuit is used in controlling some mission-critical device, we don't want even this slim chance to happen.

In order to avoid this non-deterministic behavior, we must make sure that the two inputs are never de-asserted at the same time. Note that we do want the situation when both of them are de-asserted, as in times t_1 and t_3 , so that the circuit can remember its current content. We want to de-assert one input after de-asserting the other, but just not de-asserting both of them at *exactly* the same time. In practice, it is very difficult to guarantee that these two signals are never de-asserted at the same time, so we relax the condition slightly by not having both of them asserted together. In other words, if one is asserted, then the other one cannot be asserted. Therefore, if both of them are never asserted simultaneously, then they cannot be de-asserted at the same time. A minor side benefit for not having both of them asserted together is that Q and Q' are never equal to each other. Recall that, from the names that we have given these two nodes, we do want them to be inverses of each other.

From the above analysis, we obtain the truth table in Figure 6.2(b) for the NAND implementation of the SR latch. In the truth table, Q and Q_{next} actually represent the same point in the circuit. The difference is that Q is the current value at that point, while Q_{next} is the new value to be updated in the next time period. Another way of looking at it is that Q is the input to a gate, and Q_{next} is the output from a gate. In other words, the signal Q goes into a gate, propagates through the two gates, and arrives back at Q as the new signal Q_{next} . Figure 6.2(c) shows the logic symbol for the SR latch.

The SR latch can also be implemented using NOR gates, as shown in Figure 6.3(a). The truth table for this implementation is shown in Figure 6.3(b). From the truth table, we see that the main difference between this implementation and the NAND implementation is that, for the NOR implementation, the S and R inputs are active-high, so that setting S to 1 will set the latch, and setting R to 1 will reset the latch. However, just like the NAND implementation, the latch is set when $Q = 1$ and reset when $Q = 0$. The latch remembers its previous state when $S = R = 0$. When $S = R = 1$, both Q and Q' are 0. The logic symbol for the SR latch using NOR implementation is shown in Figure 6.3(c).

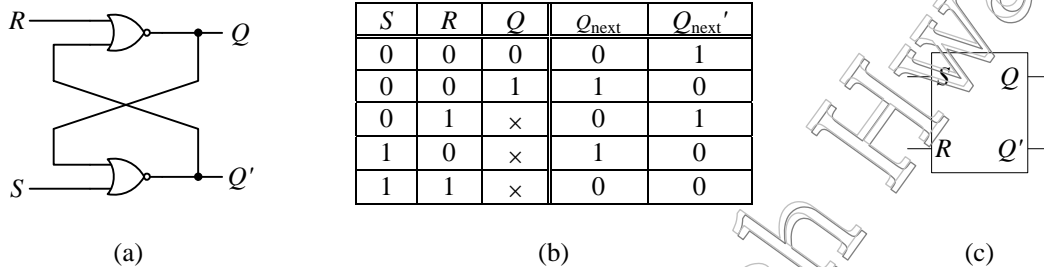


Figure 6.3 SR latch: (a) circuit using NOR gates; (b) truth table; (c) logic symbol.

6.3 SR Latch with Enable

The SR latch is sensitive to its inputs all the time. In other words, Q will always change when either S or R is asserted. It is sometimes useful to be able to disable the inputs so that asserting them will not cause the latch to change state but to keep its current state. Of course, this is achieved by de-asserting both S and R . Hence, what we want is just one enable signal that will de-assert both S and R . The **SR latch with enable** (also known as a **gated SR latch**) shown in Figure 6.4(a) accomplishes this by adding two extra NAND gates to the original NAND-gate implementation of the latch. These two new NAND gates are controlled by the enable input, E , which determines whether the latch is enabled or disabled. When $E = 1$, the circuit behaves like the normal NAND implementation of the SR latch, except that the new S and R inputs are active-high rather than active-low. When $E = 0$, then $S' = R' = 1$, and the latch will remain in its previous state, regardless of the S and R inputs. The truth table and the logic symbol for the SR latch with enable is shown in Figure 6.4(b) and (c), respectively.

A typical operation of the latch is shown in the sample trace in Figure 6.4(d). Between t_0 and t_1 , $E = 0$, so changing the S and R inputs do not affect the output. Between t_1 and t_2 , $E = 1$, and the trace is similar to the trace of Figure 6.2(d) except that the input signals are inverted.

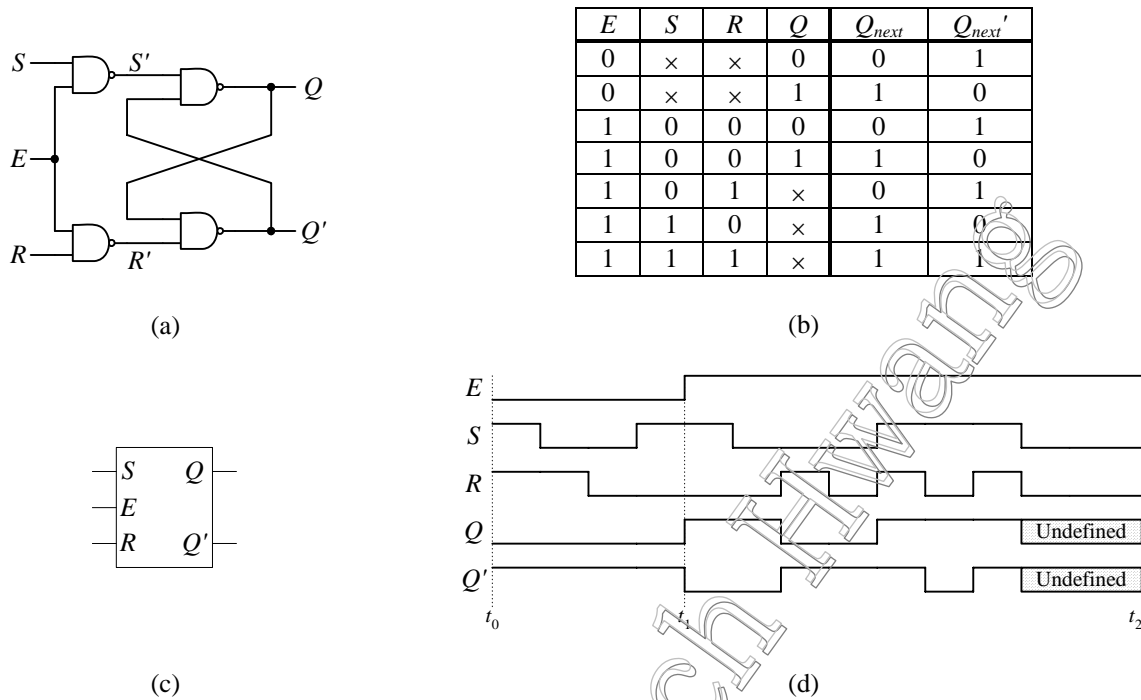


Figure 6.4 SR latch with enable: (a) circuit using NAND gates; (b) truth table; (c) logic symbol; (d) sample trace.

6.4 D Latch

Recall from Section 6.2 that the disadvantage with the SR latch is that we need to ensure that the two inputs, S and R , are never de-asserted at exactly the same time, and we said that we can guarantee this by not having both of them asserted. This situation is prevented in the **D latch** by adding an inverter between the original S' and R' inputs. This way, S' and R' will always be inverses of each other, and so, they will never be asserted together. The circuit using NAND gates and the inverter is shown in Figure 6.5(a). There is now only one input D (for data). When $D = 0$, then $S' = 1$ and $R' = 0$, so this is similar to resetting the SR latch by making $Q = 0$. Similarly, when $D = 1$, then $S' = 0$ and $R' = 1$, and Q will be set to 1. From this observation, we see that Q_{next} always gets the same value as the input D and is independent of the current value of Q . Hence, we obtain the truth table for the D latch, as shown in Figure 6.5(b).

Comparing the truth table for the D latch shown in Figure 6.5(b) with the truth table for the SR latch shown in Figure 6.2(b), it is obvious that we have eliminated not just one, but three rows, where $S' = R'$. The reason for adding the inverter to the SR-latch circuit was to eliminate the row where $S' = R' = 0$. However, we still need to have the other two rows where $S' = R' = 1$ in order for the circuit to remember its current value. By not being able to set both S' and R' to 1, this D-latch circuit has now lost its ability to remember. Q_{next} cannot remember the current value of Q , instead, it will always follow D . The end result is like having a piece of wire where the output is the same as the input!

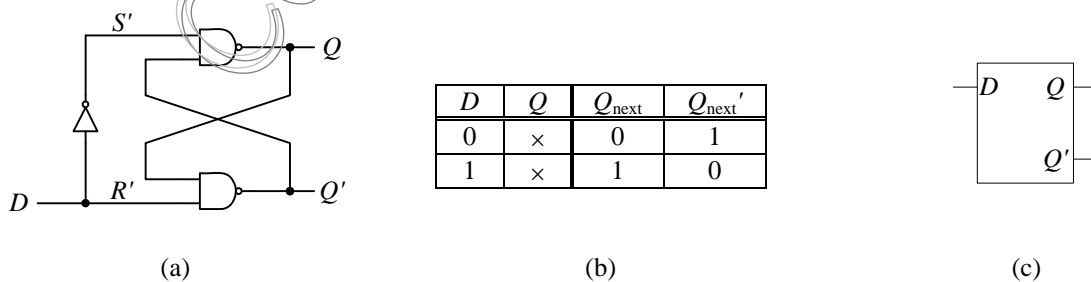


Figure 6.5 D latch: (a) circuit using NAND gates; (b) truth table; (c) logic symbol.

6.5 D Latch with Enable

In order to make the D latch remember the current value, we need to connect Q (the current state value) back to the input D , thus creating another feedback loop. Furthermore, we need to be able to select whether to loop Q back to D or input a new value for D . Otherwise, like the bistable element, we will not be able to change the state of the circuit. One way to achieve this is to use a 2-input multiplexer to select whether to feedback the current value of Q or pass an external input back to D . The circuit for the **D latch with enable** (also known as a **gated D latch**) is shown in Figure 6.6(a). The external input becomes the new D input, the output of the multiplexer is connected to the original D input, and the select line of the multiplexer is the enable signal E .

When the enable signal E is asserted ($E = 1$), the external D input passes through the multiplexer, and so Q_{next} (i.e., the output Q) follows the D input. On the other hand, when E is de-asserted ($E = 0$), the current value of Q loops back as the input to the circuit, and so Q_{next} retains its last value independent of the D input.

When the latch is enabled, the latch is said to be open, and the path from the input D to the output Q is transparent. In other words, Q follows D . Because of this characteristic, the D latch with enable circuit is often referred to as a **transparent latch**. When the latch is disabled, it is closed, and the latch remembers its current state. The truth table and the logic symbol for the D latch with enable are shown in Figure 6.6(b) and (c). A sample trace for the operation of the D latch with enable is shown in Figure 6.6(d). Between t_0 and t_1 , the latch is enabled with $E = 1$, so the output Q follows the input D . Between t_1 and t_2 , the latch is disabled, so Q remains stable even when D changes.

An alternative way to construct the D latch with enable circuit is shown in Figure 6.7. Instead of using the 2-input multiplexer, as shown in Figure 6.6(a), we start with the SR latch with enable circuit of Figure 6.4(a), and connect the S and R inputs together with an inverter. The functional operations of these two circuits are identical.

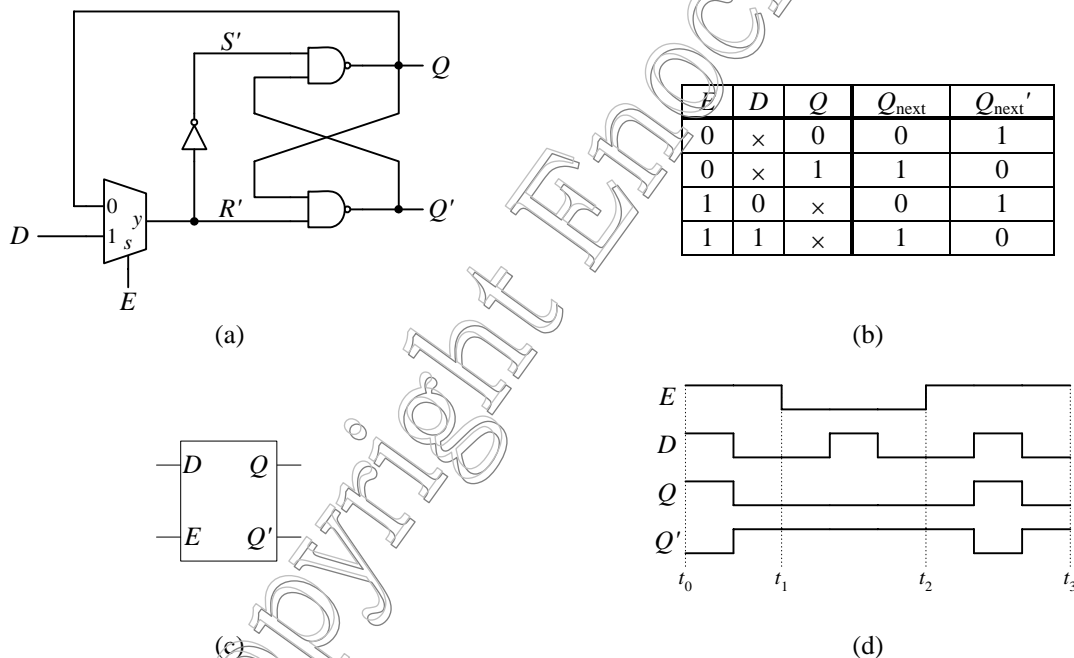


Figure 6.6 D latch with enable: (a) circuit; (b) truth table; (c) logic symbol; (d) sample trace.

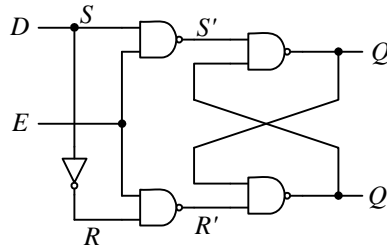


Figure 6.7 D latch with enable circuit using four NAND gates.

6.6 Clock

Latches are known as **level-sensitive** because their outputs are affected by their inputs as long as they are enabled. Their memory state can change during this entire time when the enable signal is asserted. In a computer circuit, however, we do not want the memory state to change at various times when the enable signal is asserted. Instead, we like to synchronize all of the state changes to happen at precisely the same moment and at regular intervals. In order to achieve this, two things are needed: (1) a synchronizing signal, and (2) a memory circuit that is not level-sensitive. The synchronizing signal, of course, is the **clock**, and the non-level-sensitive memory circuit is the flip-flop.

The clock is simply a very regular square wave signal, as shown in Figure 6.8. We call the edge of the clock signal when it changes from 0 to 1 the **rising edge**. Conversely, the **falling edge** of the clock is the edge when the signal changes from 1 to 0. We will use the symbol \uparrow to denote the rising edge and \downarrow for the falling edge. In a computer circuit, either the rising edge or the falling edge of the clock can be used as the synchronizing signal for writing data into a memory element. This edge signal is referred to as the **active edge** of the clock. In all of our examples, we will use the rising clock edge as the active edge. Therefore, at every rising edge, data will be clocked or stored into the memory element.

A **clock cycle** is the time from one rising edge to the next rising edge or from one falling edge to the next falling edge. The speed of the clock, measured in hertz (Hz), is the number of cycles per second. Typically, the clock speed for a microprocessor in an embedded system runs around 20 MHz, while the microprocessor in a personal computer runs upwards of 2 GHz and higher. A clock **period** is the time for one clock cycle (seconds per cycle), so it is just the inverse of the clock speed.

The speed of the clock is determined by how fast a circuit can produce valid results. For example, a two-level combinational circuit will have valid results at its output much sooner than, say, an ALU can. Of course, we want the clock speed to be as fast as possible, but it can only be as fast as the slowest circuit in the entire system. We want the clock period to be the time it takes for the slowest circuit to get its input from a memory element, operate on the data, and then write the data back into a memory element. More will be said on this in later sections.

Figure 6.9 shows a VHDL description of a clock-divider circuit that roughly cuts a 25 MHz clock down to 1 Hz.

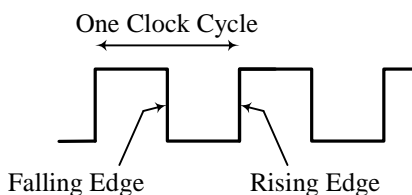


Figure 6.8 Clock signal.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Clockdiv IS PORT (
  Clk25Mhz: IN STD_LOGIC;

```

```

    Clk: OUT STD_LOGIC);
END Clockdiv;

ARCHITECTURE Behavior OF Clockdiv IS
    CONSTANT max: INTEGER := 25000000;
    CONSTANT half: INTEGER := max/2;
    SIGNAL count: INTEGER RANGE 0 TO max;
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clk25Mhz'EVENT and Clk25Mhz = '1';
        IF count < max THEN
            count <= count + 1;
        ELSE
            count <= 0;
        END IF;
        IF count < half THEN
            Clk <= '0';
        ELSE
            Clk <= '1';
        END IF;
    END PROCESS;
END Behavior;

```

Figure 6.9 VHDL behavioral description of a clock-divider circuit.

6.7 D Flip-Flop

Unlike the latch, a flip-flop is not level-sensitive, but rather **edge-triggered**. In other words, data gets stored into a flip-flop only at the active edge of the clock. An **edge-triggered D flip-flop** achieves this by combining in series a pair of D latches. Figure 6.10(a) shows a **positive edge-triggered D flip-flop**, where two D latches are connected in series. A clock signal *Clk* is connected to the *E* input of the two latches: one directly, and one through an inverter.

The first latch is called the *master* latch. The master latch is enabled when *Clk* = 0 because of the inverter, and so *QM* follows the primary input *D*. However, the signal at *QM* cannot pass over to the primary output *Q*, because the second latch (called the *slave* latch) is disabled when *Clk* = 0. When *Clk* = 1, the master latch is disabled, but the slave latch is enabled so that the output from the master latch, *QM*, is transferred to the primary output *Q*. The slave latch is enabled all the while that *Clk* = 1, but its content changes only at the rising edge of the clock, because once *Clk* is 1, the master latch is disabled, and the input to the slave latch, *QM*, will be constant. Therefore, when *Clk* = 1 and the slave latch is enabled, the primary output *Q* will not change because the input *QM* is not changing.

The circuit shown in Figure 6.10(a) is called a positive edge-triggered D flip-flop because the primary output *Q* on the slave latch changes only at the rising edge of the clock. If the slave latch is enabled when the clock is low (i.e., with the inverter output connected to the *E* of the slave latch), then it is referred to as a **negative edge-triggered** flip-flop. The circuit is also referred to as a **master-slave** D flip-flop because of the two D latches used in the circuit.

Figure 6.10(b) shows the operation table for the D flip-flop. The \uparrow symbol signifies the rising edge of the clock. When *Clk* is either at 0 or 1, the flip-flop retains its current value (i.e., $Q_{\text{next}} = Q$). Q_{next} changes and follows the primary input *D* only at the rising edge of the clock. The logic symbol for the positive edge-triggered D flip-flop is shown in Figure 6.10(c). The small triangle at the clock input indicates that the circuit is triggered by the edge of the signal, and so it is a flip-flop. Without the small triangle, the symbol would be that for a latch. If there is a circle in front of the clock line, then the flip-flop is triggered by the falling edge of the clock, making it a negative edge-triggered flip-flop. Figure 6.10(d) shows a sample trace for the D flip-flop. Notice that when *Clk* = 0, *QM* follows *D*, and the output of the slave latch, *Q*, remains constant. On the other hand, when *Clk* = 1, *Q* follows *QM*, and the output of the master latch, *QM*, remains constant.

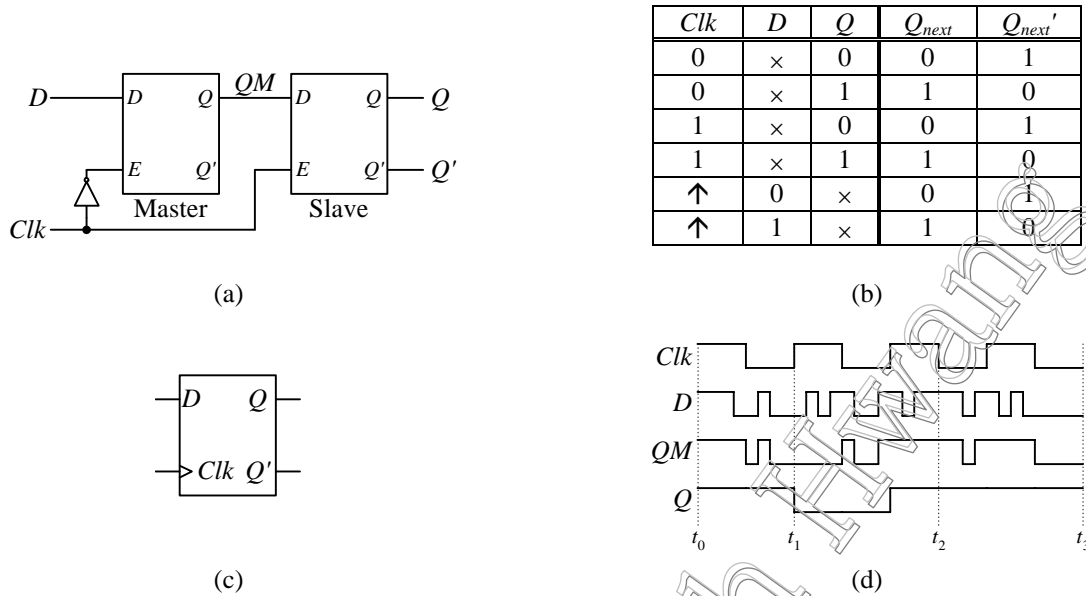


Figure 6.10 Master-slave positive edge-triggered D flip-flop: (a) circuit using D latches; (b) operation table; (c) logic symbol; (d) sample trace.

Figure 6.11 compares the different operations between a latch and a flip-flop. In Figure 6.11(a), we have a D latch with enable, a positive edge-triggered D flip-flop, and a negative edge-triggered D flip-flop, all having the same *D* input and controlled by the same clock signal. Figure 6.11(b) shows a sample trace of the circuit's operations. Notice that the gated D latch, *Q_a*, follows the *D* input as long as the clock is high (between times *t₀* and *t₁* and times *t₂* and *t₃*). The positive edge-triggered flip-flop, *Q_b*, follows the *D* input only at the rising edge of the clock at time *t₂*, while the negative edge-triggered flip-flop, *Q_c*, follows the *D* input only at the falling edge of the clock at times *t₁* and *t₃*.

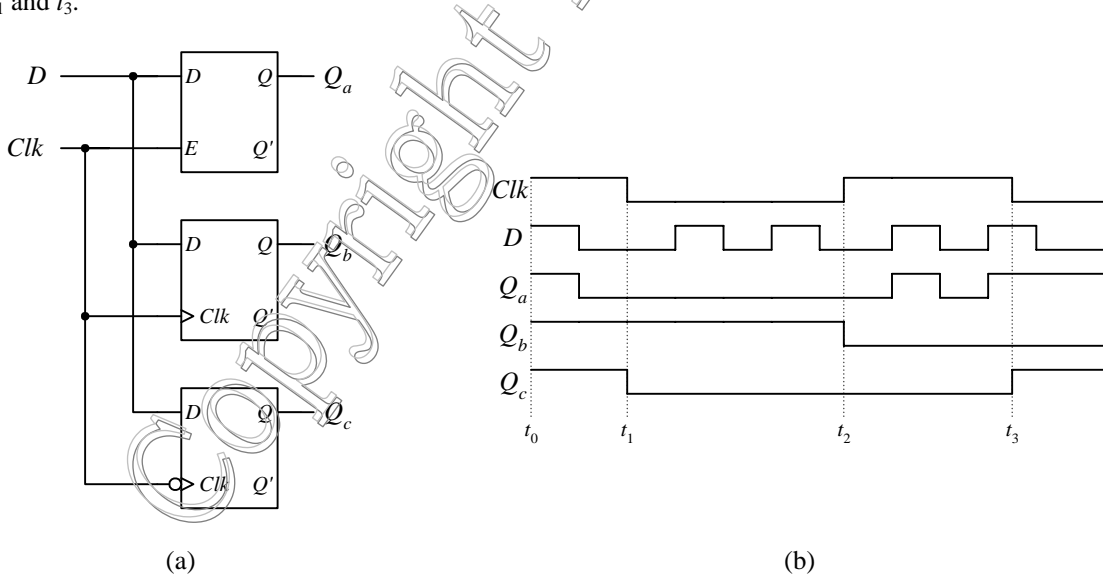


Figure 6.11 Comparison of a gated latch, a positive edge-triggered flip-flop, and a negative edge-triggered flip-flop: (a) circuit; (b) sample trace.

6.7.1 * Alternative Smaller Circuit

Not all master-slave flip-flops are edge-triggered. For instance, using two SR latches to construct a master-slave

flip-flop results in a flip-flop that is level-sensitive. Conversely, an edged-triggered D flip-flop can be constructed using SR latches instead of the master-slave D latches.

The circuit shown in Figure 6.12 shows how a positive edge-triggered D flip-flop can be constructed using three interconnected SR latches. The advantage of this circuit is that it uses only 6 NAND gates (26 transistors) as opposed to 11 gates (38 transistors) for the master-slave D flip-flop shown in Figure 6.10(a). The operation of the circuit is as follows. When $Clk = 0$, the outputs of gates 2 and 3 will be 1 (since $0 \text{ NAND } x = 1$). With $n_2 = n_3 = 1$, this will keep the output latch (comprising of gates 5 and 6) in its current state. At the same time, $n_4 = D'$ since one input to gate 4 is n_3 , which is a 1 ($1 \text{ NAND } x = x'$). Similarly, $n_1 = D$ since $n_2 = 1$, and the other input to gate 1 is n_4 , which is D' (again $1 \text{ NAND } x = x'$).

When Clk changes to 1, n_2 will be equal to D' because $1 \text{ NAND } n_1 = n_1'$, and $n_1 = D$. Similarly, n_3 will be equal to D when Clk changes to 1 because the other two inputs to gate 3 are both D' . Therefore, if $Clk = 1$ and $D = 0$, then n_2 (which is equal to D') will be 1 and n_3 (which is equal to D) will be 0. With $n_2 = 1$ and $n_3 = 0$, this will de-assert S' and assert R' , thus resetting the output latch Q to 0. On the other hand, if $Clk = 1$ and $D = 1$, then n_2 (which is equal to D') will be 0 and n_3 (which is equal to D) will be 1. This will assert S' and de-assert R' ; thus setting the output latch Q to 1. So at the rising edge of the Clk signal, Q will follow D .

The setting and resetting of the output latch occurs only at the rising edge of the Clk signal, because once Clk is at a 1 and remains at a 1, changing D will not change n_2 or n_3 . The reason, as noted in the previous paragraph, is that n_2 and n_3 are always inverses of each other. Furthermore, the following argument shows that both n_2 and n_3 will remain constant even if D changes. Let us first assume that n_2 is a 0. If $n_2 = 0$, then n_3 (the output of gate 3) will always be a 1 (since $0 \text{ NAND } x = 1$), regardless of what n_4 (the third input to gate 3) may be. Hence, if n_4 (the output of gate 4) cannot affect n_3 , then D (the input to gate 4) also cannot affect either n_2 or n_3 . On the other hand, if $n_2 = 1$, then $n_3 = 0$ ($n_3 = n_2'$). With a 0 from n_3 going to the input of gate 4, the output of gate 4 at n_4 will always be a 1 ($0 \text{ NAND } x = 1$), regardless of what D is. With the three inputs to gate 3 being all 1's, n_3 will continue to be 0. Therefore, as long as $Clk = 1$, changing D will not change n_2 or n_3 . And if n_2 and n_3 remain stable, then Q will also remain stable for the entire time that Clk is 1.

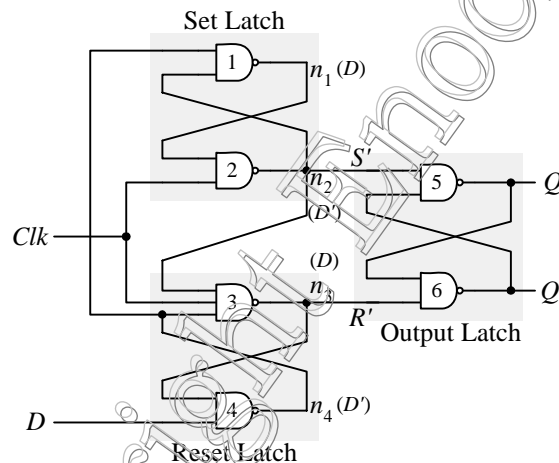


Figure 6.12. Positive edge-triggered D flip-flop.

6.8 D Flip-Flop with Enable

So far, with the construction of the different memory elements, it seems like every time we add a new feature we have also lost a feature that we need. The careful reader will have noticed that, in building the D flip-flop, we have again lost the most important property of a memory element—it can no longer remember its current content! At every active edge of the clock, the D flip-flop will load in a new value. So how do we get it to remember its current value and not load in a new value?

The answer, of course, is exactly the same as what we did with the D latch, and that is by adding an enable input, E , through a 2-input multiplexer, as shown in Figure 6.13(a). When $E = 1$, the primary input D signal will pass to the D input of the flip-flop, thus updating the content of the flip-flop at the active edge. When $E = 0$, the

current content of the flip-flop at Q is passed back to the D input of the flip-flop, thus keeping its current value. Notice that changes to the flip-flop value occur only at the active edge of the clock. Here, we are using the rising edge as the active edge. The operation table and the logic symbol for the D flip-flop with enable is shown in Figure 6.13(b) and (c) respectively.

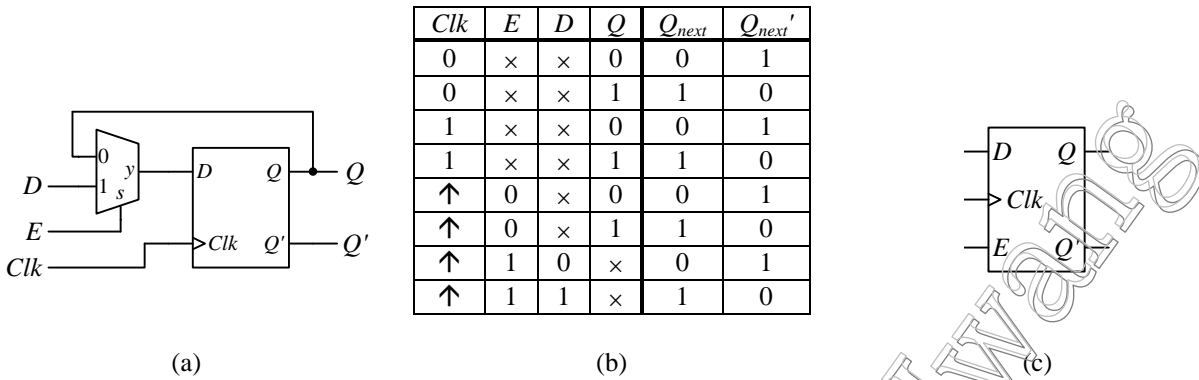
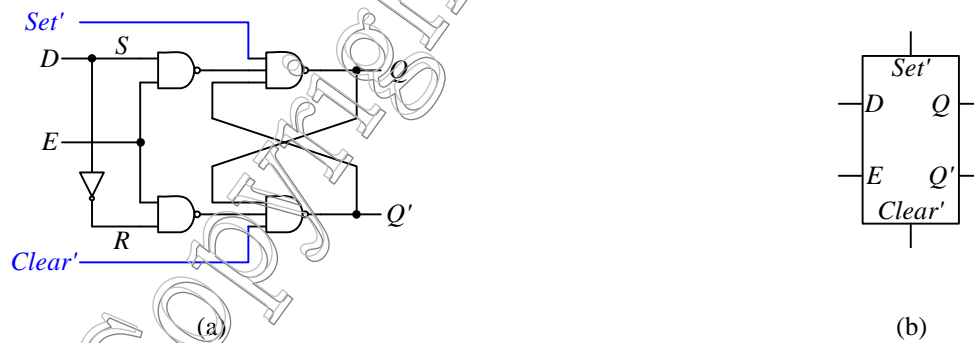


Figure 6.13 D flip-flop with enable: (a) circuit; (b) operation table; (c) logic symbol.

6.9 Asynchronous Inputs

Flip-flops (as we have seen so far) change states only at the rising or falling edge of a synchronizing clock signal. Many circuits require the initialization of flip-flops to a known state that is independent of the clock signal. Sequential circuits that change states whenever a change in input values occurs that is independent of the clock are referred to as **asynchronous** sequential circuits. **Synchronous** sequential circuits, on the other hand, change states only at the active edge of the clock signal. Asynchronous inputs usually are available for both flip-flops and latches, and they are used to either set or clear the storage element's content that is independent of the clock.

Figure 6.14(a) shows a gated D latch with asynchronous active-low Set' and $Clear'$ inputs, and (b) is the logic symbol for it. Figure 6.14(c) is the circuit for the D edge-triggered flip-flop with asynchronous Set' and $Clear'$ inputs, and (d) is the logic symbol for it. When Set' is asserted (set to 0) the content of the storage element is set to 1 immediately (i.e., without having to wait for the next rising clock edge), and when $Clear'$ is asserted (set to 0) the content of the storage element is set to 0 immediately.



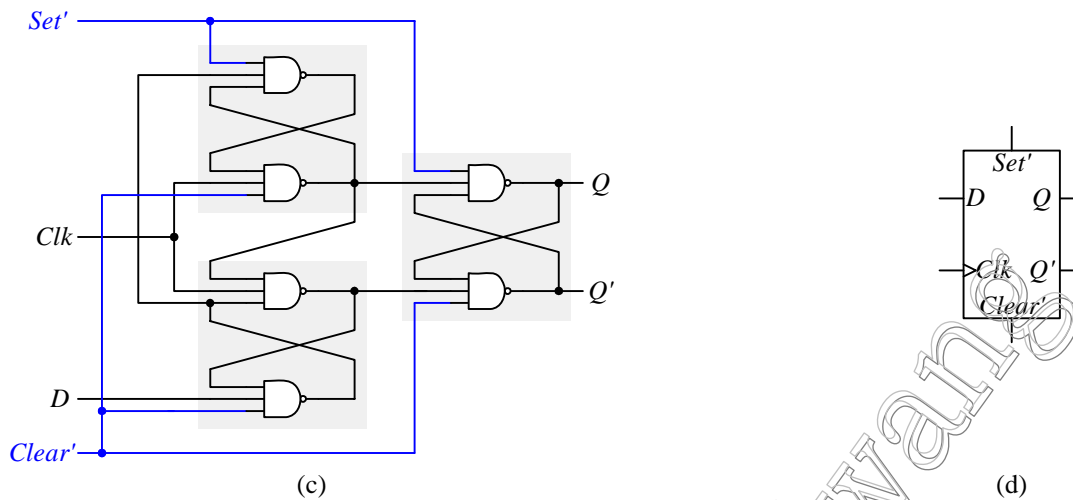


Figure 6.14 Storage elements with asynchronous inputs: (a) D latch with active-low set and clear; (b) logic symbol for (a); (c) D edge-triggered flip-flop with active-low set and clear; (d) logic symbol for (c).

6.10 Description of a Flip-Flop

Combinational circuits can be described with either a truth table or a Boolean equation. For describing the operation of a flip-flop or any sequential circuit in general, we use a characteristic table, a characteristic equation, a state diagram, or an excitation table, as discussed in the following subsections.

6.10.1 Characteristic Table

The **characteristic table** specifies the functional behavior of the flip-flop. It is a simplified version of the flip-flop's operational table by only listing how the state changes at the active clock edge. The table has the flip-flop's input signal(s) and current state (Q) listed in the input columns, and the next state (Q_{next}) listed in the output column. Q_{next}' is always assumed to be the inverse of Q_{next} , so it is not necessary to include this output column. The clock signal is also not included in the table, because it is a signal that we do not want to modify. Nevertheless, the clock signal is always assumed to exist. Furthermore, since all state changes for a flip-flop (i.e., changes to Q_{next}) occur at the active edge of the clock, therefore it is not necessary to list the situations from the operation table for when the clock is at a constant value.

The characteristic table for the D flip-flop is shown in Figure 6.15(a). It has two input columns (the input signal D , and the current state Q) and one output column for Q_{next} . From the operation table for the D flip-flop shown in Figure 6.10(b), we see that there are only two rows where Q_{next} is affected during the rising clock edge. Hence, these are the only two rows inserted into the characteristic table.

The characteristic table is used in the analysis of sequential circuits to answer the question of what is the next state, Q_{next} , when given the current state, Q , and input signals (D in the case of the D flip-flop).

6.10.2 Characteristic Equation

The **characteristic equation** is simply the Boolean equation that is derived directly from the characteristic table. Like the characteristic table, the characteristic equation specifies the flip-flop's next state, Q_{next} , as a function of its current state, Q , and input signals. The D flip-flop characteristic table has only one 1-minterm, which results in the simple characteristic equation for the D flip-flop shown in Figure 6.15(b).

6.10.3 State Diagram

A **state diagram** is a graph with nodes and directed edges connecting the nodes, as shown in Figure 6.15(c). The state diagram graphically portrays the operation of the flip-flop. The nodes are labeled with the states of the flip-flop, and the directed edges are labeled with the input signals that cause the transition to go from one state of the flip-flop to the next. Figure 6.15(c) shows the state diagram for the D flip-flop. It has two states, $Q = 0$ and $Q = 1$, which correspond to the two values that the flip-flop can contain. The operation of the D flip-flop is such that when it is in state 0, it will change to state 1 if the input D is a 1; otherwise, if the input D is a 0, then it will remain in state 0. Hence, there is an edge labeled $D = 1$ that goes from state $Q = 0$ to $Q = 1$, and a second edge labeled $D = 0$ that goes from state $Q = 0$ back to itself. Similarly, when the flip-flop is in state 1, it will change to state 0 if the input D is a 0; otherwise, it will remain in state 1. These two conditions correspond to the remaining two edges that go out from state $Q = 1$ in the state diagram.

6.10.4 Excitation Table

The **excitation table** is like the mirror image of the characteristic table by exchanging the input signal column(s) with the output (Q_{next}) column. The excitation table shows what the flip-flop's inputs should be in order to change from the flip-flop's current state to the next state desired. In other words, the excitation table answers the question of what the flip-flop's inputs should be when given the current state that the flip-flop is in and the next state that we want the flip-flop to go to. This table is used in the synthesis of sequential circuits.

Figure 6.15(d) shows the excitation table for the D flip-flop. As can be seen, this table can be obtained directly from the state diagram. For example, using the state diagram of the D flip-flop from Figure 6.15(c), if the current state is $Q = 0$ and we want the next state to be $Q_{next} = 0$, then the D input must be a 0 as shown by the label on the edge that goes from state 0 back to itself. On the other hand, if the current state is $Q = 0$ and we want the next state to be $Q_{next} = 1$, then the D input must be a 1.

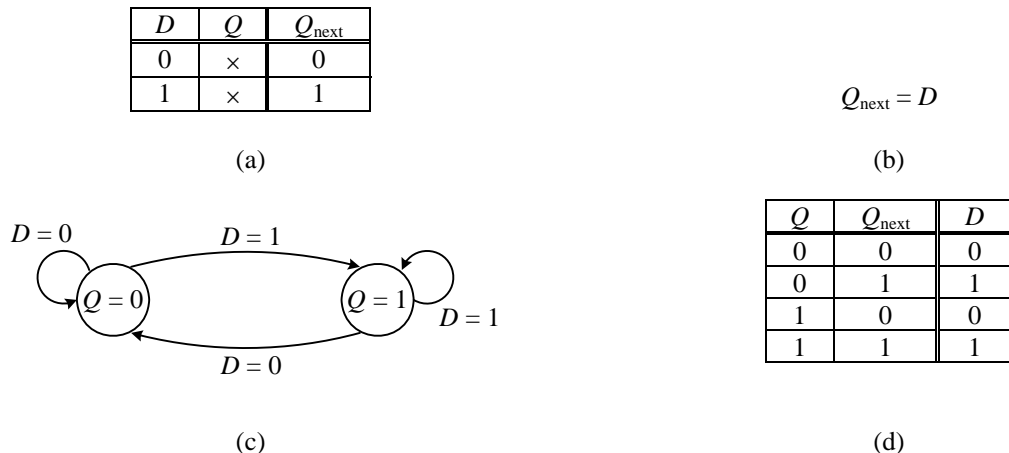


Figure 6.15 Description of a D flip-flop: (a) characteristic table; (b) characteristic equation; (c) state diagram; (d) excitation table.

6.11 * Timing Issues

So far in our discussion of latches and flip-flops, we have ignored timing issues and the effects of propagation delays. In practice, timing issues are very important in the correct design of sequential circuits. Consider again the D latch with enable circuit from Section 6.5 and redrawn in Figure 6.16(a). Signals from the inputs require some delay to propagate through the gates and finally to reach the outputs.

Assuming that the propagation delay for the inverter is 1 nanosecond (ns) and 2 ns for the NAND gates, the timing trace diagram would look like Figure 6.16(b) with the signal delays taken into consideration. The arrows denote which signal edge causes another signal edge. The number next to an arrow denotes the number of nanoseconds in delay for the resulting signal to change.

At time t_1 , signal D drops to 0. This causes R to rise to 1 after a 1 ns delay through the inverter. The D edge also causes S' to rise to 1, but after a delay of 2 ns through the NAND gate. After that, R' drops to 0 at 2 ns after R rises to 1. This in turn causes Q' to rise to 1 after 2 ns, followed by Q dropping to 0.

At time t_2 , signal E drops to 0, disabling the circuit. As a result, when D rises to 1 at time t_3 , both Q and Q' are not affected.

At time t_4 , signal E rises to 1 and re-enables the circuit. This causes S' to drop to 0 after 2 ns. R' remains unchanged at 1 since the two inputs to the NAND gate, E and R , are 1 and 0, respectively. With S' asserted and R' de-asserted, the latch is set with Q rising to 1 at 2 ns after S' drops to a 0. This is followed by Q' dropping to 0 after another 2 ns.

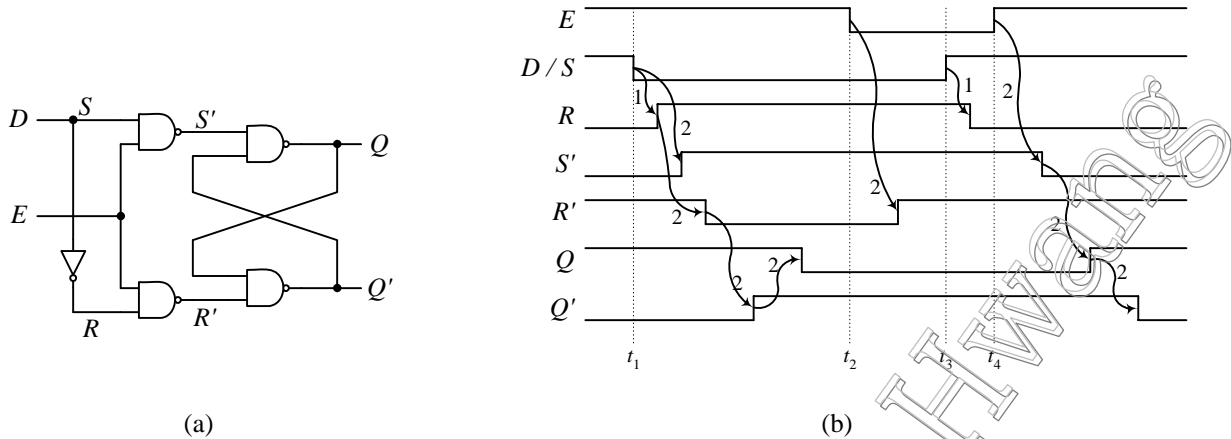


Figure 6.16 D latch with enable: (a) circuit; (b) timing diagram with delays.

Furthermore, for the D-latch circuit to latch in the data from input D correctly, there is a critical window of time right before and right after the falling edge of the enable signal, E , that must be observed. Within this time frame, the input signal, D , must not change. As shown in Figure 6.17, the time before the falling edge of E is referred to as the **setup time**, t_{setup} , and the time after the falling edge of E is referred to as the **hold time**, t_{hold} . The length of these two times is dependent on the implementation and manufacturing process and can be obtained from the component data sheet.

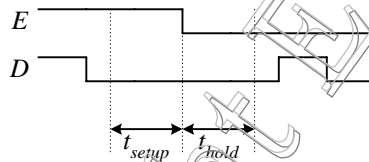


Figure 6.17 Setup and hold times for the gated D latch.

6.12 Car Security System—Version 2

In Section 2.9, we designed a combinational circuit for a car security system where the siren will come on when the master switch is on and either the door switch or the vibration switch is also on. However, as soon as both the door switch and the vibration switch are off, the siren will turn off immediately, even though the master switch is still on. In reality, what we really want is to have the siren remain on, even after both the door and vibration switches are off. In order to do so, we need to remember the state of the siren. In other words, for the siren to remain on, it should be dependent not only on whether the door or the vibration switch is on, but also on the fact that the siren is currently on.

We can use the state of a **SR latch** to remember the state of the siren (i.e., the output of the latch will drive the siren). The state of the latch is driven by the conditions of the input switches. The modified circuit, as shown in Figure 6.18, has an SR latch (in addition to its original combinational circuit) for remembering the current state of

the siren. The latch is set from the output of the combinational circuit. The latch's reset is connected to the master switch so that the siren can be turned off immediately.

A sample timing trace of the operation of this circuit is shown in Figure 6.19. At time 0, the siren is off, even though the door switch is on, because the master switch is off. At time 300 ns, the siren is turned on by the door switch since the master switch is also on. At time 500 ns, both the door and the vibration switches are off, but the siren is still on because it was turned on previously. The siren is turned off by the master switch at time 600 ns.

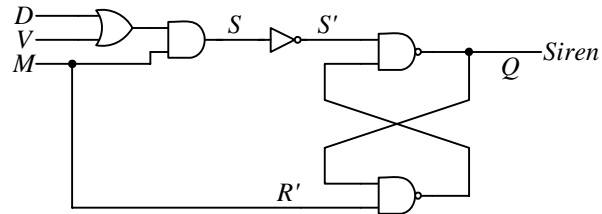


Figure 6.18 Modified car security system circuit with memory.

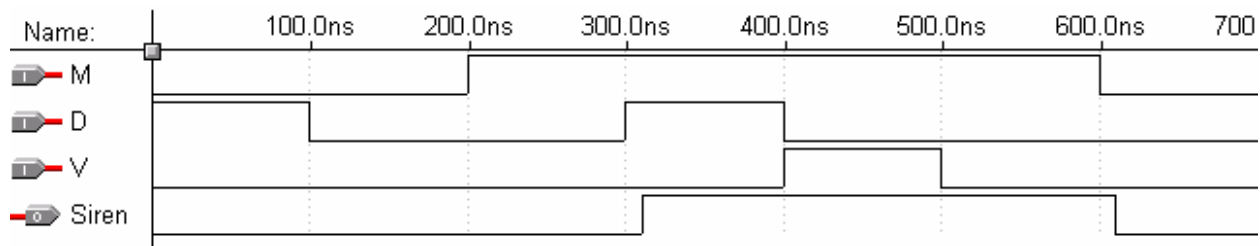


Figure 6.19 Sample timing trace of the modified car security system circuit with memory.

6.13 VHDL for Latches and Flip-Flops

6.13.1 Implied Memory Element

VHDL does not have any explicit object for defining a memory element. Instead, the semantics of the language provide for signals to be interpreted as a memory element. In other words, the memory element is declared depending on how these signals are assigned.

Consider the VHDL code in Figure 6.20. If *Enable* is 1, then *Q* gets the value of *D*; otherwise, *Q* gets a 0. In this code, *Q* is assigned a value for all possible outcomes of the test in the IF statement. With this construct, a combinational circuit is produced.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY no_memory_element IS PORT (
    D, Enable: IN STD_LOGIC;
    Q: OUT STD_LOGIC);
END no_memory_element;

ARCHITECTURE Behavior OF no_memory_element IS
BEGIN
    PROCESS(D, Enable)
    BEGIN
        IF (Enable = '1') THEN
            Q <= D;
        ELSE

```

```

    Q <= '0';
  END IF;
END PROCESS;
END Behavior;

```

Figure 6.20 Sample VHDL description of a combinational circuit.

If we remove the ELSE and the statement in the ELSE part, as shown in Figure 6.21, then we have a situation where no value is assigned to Q if $Enable$ is not 1. The key point here is that the VHDL semantics stipulate that, in cases where the code does not specify a value of a signal, the signal should retain its current value. In other words, the signal must remember its current value, and in order to do so, a memory element is implied.

6.13.2 VHDL Code for a D Latch with Enable

Figure 6.21 shows the VHDL code for a D latch with enable. If $Enable$ is 1, then Q gets the value of D . However, if $Enable$ is not 1, the code does not specify what Q should be; therefore, Q retains its current value by using a memory element. This code produces a latch and not a flip-flop, because Q follows D as long as $Enable$ is 1 and not only at the active edge of the $Enable$ signal. The process sensitivity list includes both D and $Enable$, because either one of these signals can cause a change in the value of the Q output.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY D_latch_with_enable IS PORT (
  D, Enable: IN STD_LOGIC;
  Q: OUT STD_LOGIC);
END D_latch_with_enable;

ARCHITECTURE Behavior OF D_latch_with_enable IS
BEGIN
  PROCESS(D, Enable)
  BEGIN
    IF (Enable = '1') THEN
      Q <= D;
    END IF;
  END PROCESS;
END Behavior;

```

Figure 6.21 VHDL code for a D latch with enable.

6.13.3 VHDL Code for a D Flip-Flop

Figure 6.22 shows the behavioral VHDL code for a positive edge-triggered D flip-flop. The only difference here is that Q follows D only at the rising edge of the clock, and it is specified here by the condition “ $Clock$ ’EVENT AND $Clock = '1'$.” The 'EVENT attribute refers to any changes in the qualifying $Clock$ signal. Therefore, when this happens and the resulting $Clock$ value is a 1, we have, in effect, a condition for a positive or rising clock edge. Again, the code does not specify what is assigned to Q when the condition in the IF statement is false, so it implies the use of a memory element. Note also that the process sensitivity list contains only the clock signal, because it is the only signal that can cause a change in the Q output.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY D_flipflop IS PORT (
  D, Clock: IN STD_LOGIC;

```

```

    Q: OUT STD_LOGIC);
END D_flipflop;

ARCHITECTURE Behavior OF D_flipflop IS
BEGIN
    PROCESS(Clock)                -- sensitivity list is used
    BEGIN
        IF (Clock'EVENT AND Clock = '1') THEN
            Q <= D;
        END IF;
    END PROCESS;
END Behavior;

```

Figure 6.22 Behavioral VHDL code for a positive edge-triggered D flip-flop using an IF statement.

Another way to describe a flip-flop is to use the WAIT statement instead of the IF statement, as shown in Figure 6.23. When execution reaches the WAIT statement, it stops until the condition in the statement is true before proceeding. The WAIT statement, when used in a process block for synthesis, must be the first statement in the process. Note also that the process sensitivity list is omitted, because the WAIT statement implies that the sensitivity list contains only the clock signal.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY D_flipflop IS PORT (
    D, Clock: IN STD_LOGIC;
    Q: OUT STD_LOGIC);
END D_flipflop;

ARCHITECTURE Behavioral OF D_flipflop IS
BEGIN
    PROCESS                -- sensitivity list is not used if WAIT is used
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '0';    -- negative edge triggered
        Q <= D;
    END PROCESS;
END Behavioral;

```

Figure 6.23 Behavioral VHDL code for a negative edge-triggered D flip-flop using a WAIT statement.

Alternatively, we can write a structural VHDL description for the positive edge-triggered D flip-flop, as shown in Figure 6.24. This VHDL code is based on the circuit for a positive edge-triggered D flip-flop, as given in Figure 6.12.

The simulation trace for the positive edge-triggered D flip-flop is shown in Figure 6.25. In the trace, before the first rising edge of the clock at time 100 ns, both Q and Q' (QN) are undefined because nothing has been stored in the flip-flop yet. Immediately after this rising clock edge at 100 ns, Q gets the value of D , and QN gets the inverse. At 200 ns, D changes to 1, but Q does not follow D immediately but is delayed until the next rising clock edge at 300 ns. At the same time, QN drops to 0. At 400 ns, when D drops to 0, Q again follows it at the next rising clock edge at 500 ns.

```

-- define the operation of the 2-input NAND gate
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY NAND_2 IS PORT (
    I0, I1: IN STD_LOGIC;

```

```

    O: OUT STD_LOGIC);
END NAND_2;

ARCHITECTURE Dataflow_NAND2 OF NAND_2 IS
BEGIN
    O <= I0 NAND I1;
END Dataflow_NAND2;

-- define the structural operation of the SR latch
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY SRLatch IS PORT (
    SN, RN: IN STD_LOGIC;
    Q, QN: BUFFER STD_LOGIC);
END SRLatch;

ARCHITECTURE Structural_SRLatch OF SRLatch IS
    COMPONENT NAND_2 PORT (
        I0, I1 : IN STD_LOGIC;
        O : OUT STD_LOGIC);
    END COMPONENT;
BEGIN
    U1: NAND_2 PORT MAP (SN, QN, Q);
    U2: NAND_2 PORT MAP (Q, RN, QN);
END Structural_SRLatch;

-- define the operation of the 3-input NAND gate
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY NAND_3 IS PORT (
    I0, I1, I2: IN STD_LOGIC;
    O: OUT STD_LOGIC);
END NAND_3;

ARCHITECTURE Dataflow_NAND3 OF NAND_3 IS
BEGIN
    O <= NOT (I0 AND I1 AND I2);
END Dataflow_NAND3;

-- define the structural operation of the D flip-flop
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY positive_edge_triggered_D_flipflop IS PORT (
    D, Clock: IN STD_LOGIC;
    Q, QN: BUFFER STD_LOGIC);
END positive_edge_triggered_D_flipflop;

ARCHITECTURE StructuralDFF OF positive_edge_triggered_D_flipflop IS
    SIGNAL N1, N2, N3, N4: STD_LOGIC;

    COMPONENT SRLatch PORT (
        SN, RN: IN STD_LOGIC;
        Q, QN: BUFFER STD_LOGIC);

```

```

END COMPONENT;
COMPONENT NAND_2 PORT (
  I0, I1: IN STD_LOGIC;
  O: OUT STD_LOGIC);
END COMPONENT;
COMPONENT NAND_3 PORT (
  I0, I1, I2: IN STD_LOGIC;
  O: OUT STD_LOGIC);
END COMPONENT;

BEGIN
  U1: Srlatch PORT MAP (N4, Clock, N1, N2);    -- set latch
  U2: Srlatch PORT MAP (N2, N3, Q, QN);      -- output latch
  U3: NAND_3 PORT MAP (N2, Clock, N4, N3);   -- reset latch
  U4: NAND_2 PORT MAP (N3, D, N4);         -- reset latch
END StructuralDFF;

```

Figure 6.24 Structural VHDL code for a positive edge-triggered D flip-flop.

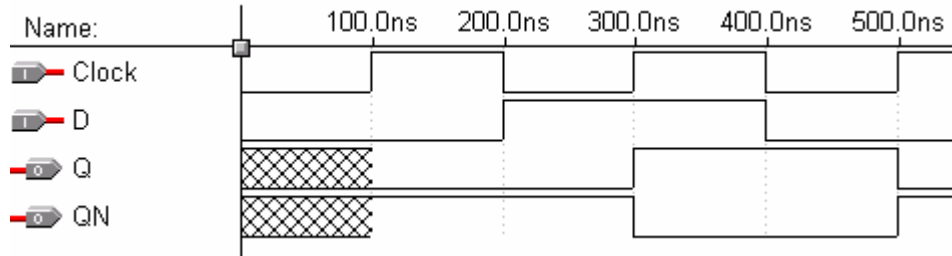


Figure 6.25 Simulation trace for the positive edge-triggered D flip-flop.

6.13.4 VHDL Code for a D Flip-Flop with Enable and Asynchronous Set and Clear

Figure 6.26 shows the VHDL code for a positive edge-triggered D flip-flop with enable and asynchronous active-high set and clear inputs. The two asynchronous inputs are checked independently of the clock event. When either the *Set* or the *Clear* input is asserted with a 1 (active-high), *Q* is immediately set to 1 or 0, respectively, independent of the clock. If *Enable* is asserted with a 1, then *Q* follows *D* at the rising edge of the clock; otherwise, *Q* keeps its previous content. Figure 6.27 shows the simulation trace for this flip-flop. Notice in the trace that when either *Set* or *Clear* is asserted (at 100 ns and 200 ns, respectively) *Q* changes immediately. However, when *Enable* is asserted at 400 ns, *Q* doesn't follow *D* until the next rising clock edge at 500 ns. Similarly, when *D* drops to 0 at 600 ns, *Q* doesn't change immediately but drops at the next rising edge at 700 ns. At 800 ns, when *D* changes to a 1, *Q* does not follow the change at the next rising edge at 900 ns, because *Enable* is now de-asserted.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY d_ff IS PORT (
  Clock: IN STD_LOGIC;
  Enable: IN STD_LOGIC;
  Set: IN STD_LOGIC;
  Clear: IN STD_LOGIC;
  D: IN STD_LOGIC;
  Q: OUT STD_LOGIC);
END d_ff;

ARCHITECTURE Behavioral OF d_ff IS

```

```

BEGIN
  PROCESS(Clock, Set, Clear)
  BEGIN
    IF (Set = '1') THEN
      Q <= '1';
    ELSIF (Clear = '1') THEN
      Q <= '0';
    ELSIF (Clock'EVENT AND Clock = '1') THEN
      IF Enable = '1' THEN
        Q <= D;
      END IF;
    END IF;
  END PROCESS;
END Behavioral;

```

Figure 6.26 Behavioral VHDL code for a positive edge-triggered D flip-flop with active-high enable and asynchronous set and clear inputs.

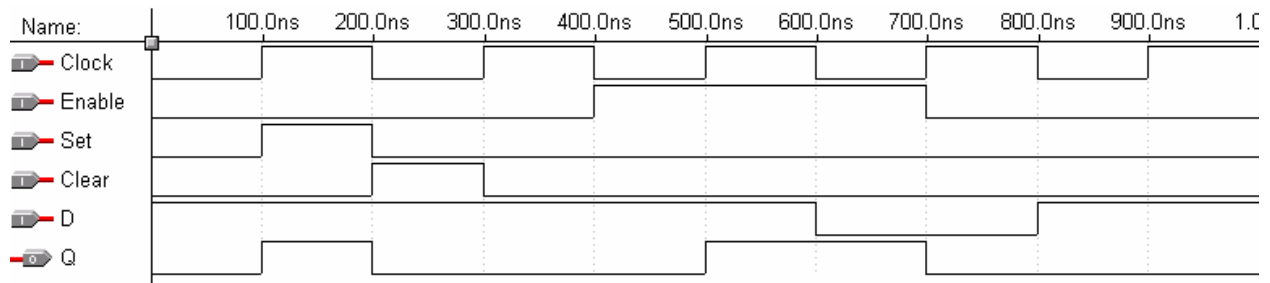


Figure 6.27 Simulation trace for the positive edge-triggered D flip-flop with active-high enable and asynchronous set and clear inputs.

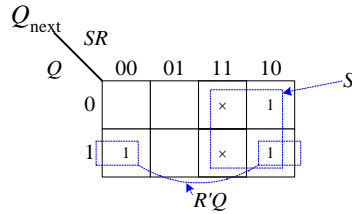
6.14 * Other Flip-Flop Types

There are basically four main types of flip-flops: D, SR, JK, and T. The major differences in these flip-flop types are in the number of inputs they have and how they change states. Like the D flip-flop, each type can also have different variations, such as active-high or -low inputs, whether they change state at the rising or falling edge of the clock signal, and whether they have any asynchronous inputs. Any given sequential circuit can be built using any of these types of flip-flops or combinations of them. However, selecting one type of flip-flop over another type to use in a particular circuit can affect the overall size of the circuit. Today, sequential circuits are designed primarily with D flip-flops only because of their simple operation. Of the four flip-flop's characteristic equations, the characteristic equation for the D flip-flop is the simplest.

6.14.1 SR Flip-Flop

Like SR latches, SR flip-flops are useful in control applications where we want to be able to set or reset the data bit. However, unlike SR latches, SR flip-flops change their content only at the active edge of the clock signal. Similar to SR latches, SR flip-flops can enter an undefined state when both inputs are asserted simultaneously. When the two inputs are de-asserted, then the next state is the same as the current state. The characteristic table, characteristic equation, state diagram, circuit, logic symbol, and excitation table for the SR flip-flop are shown in Figure 6.28.

The SR flip-flop truth table shown in Figure 6.28(a) is for active-high set and reset signals. Hence, the flip-flop state, Q_{next} , is set to 1 when S is asserted with a 1, and Q_{next} is reset to 0 when R is asserted with a 1. When both S and R are de-asserted with a 0, the flip-flop remembers its current state. From the truth table, we get the following K-map for Q_{next} , which results in the characteristic equation shown in Figure 6.28(b).



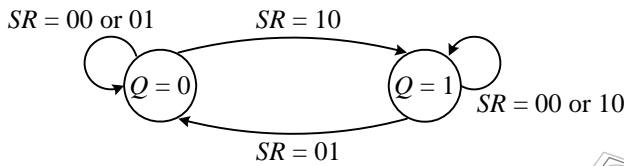
Notice that the SR flip-flop circuit shown in Figure 6.28(d) uses the D flip-flop. The signal for asserting the D input of the flip-flop is generated by the combinational circuit that is derived from the characteristic equation of the SR flip-flop, namely $D = Q_{next} = S + R'Q$.

S	R	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	x	x
1	1	1	x	x

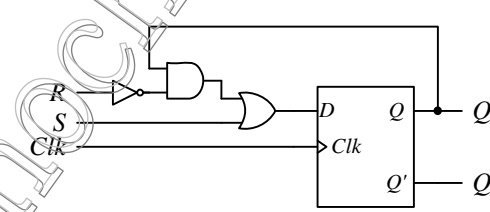
(a)

$$Q_{next} = S + R'Q$$

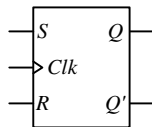
(b)



(c)



(d)



(e)

Q	Q_{next}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

(f)

Figure 6.28 SR flip-flop: (a) characteristic table; (b) characteristic equation; (c) state diagram; (d) circuit; (e) logic symbol; (f) excitation table.

6.14.2 JK Flip-Flop

The operation of the JK flip-flop is very similar to the SR flip-flop. The J input is just like the S input in the SR flip-flop in that, when asserted, it sets the flip-flop. Similarly, the K input is like the R input where it resets the flip-flop when asserted. The only difference is when both inputs, J and K , are asserted. For the SR flip-flop, the next state is undefined; whereas, for the JK flip-flop, the next state is the inverse of the current state. In other words, the JK flip-flop toggles its state when both inputs are asserted. The characteristic table, characteristic equation, state diagram, circuit, logic symbol, and excitation table for the JK flip-flop are shown in Figure 6.29.

6.14.3 T Flip-Flop

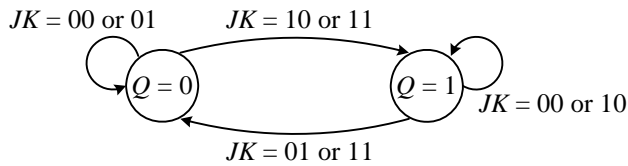
The T flip-flop has one input, T (which stands for toggle), in addition to the clock. When T is asserted ($T = 1$), the flip-flop state toggles back and forth at each active edge of the clock, and when T is de-asserted, the flip-flop keeps its current state. The characteristic table, characteristic equation, state diagram, circuit, logic symbol, and excitation table for the T flip-flop are shown in Figure 6.30.

J	K	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

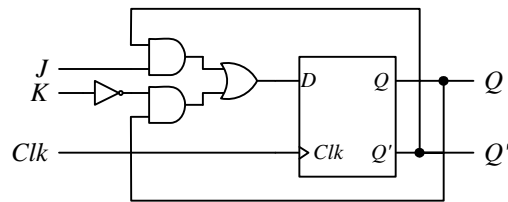
(a)

$$Q_{next} = K'Q + JQ'$$

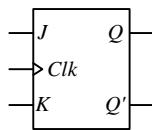
(b)



(c)



(d)



(e)

Q	Q_{next}	J	K
0	0	0	×
0	1	1	×
1	0	×	1
1	1	×	0

(f)

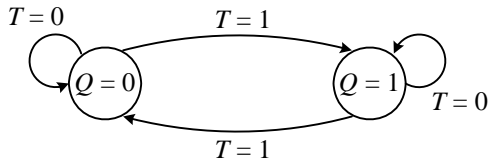
Figure 6.29 JK flip-flop: (a) characteristic table; (b) characteristic equation; (c) state diagram; (d) circuit; (e) logic symbol; (f) excitation table.

T	Q	Q_{next}	Q_{next}'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

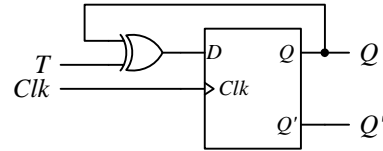
(a)

$$Q_{next} = TQ' + T'Q = T \oplus Q$$

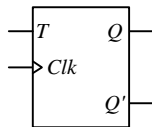
(b)



(c)



(d)



(e)

Q	Q_{next}	T
0	0	0
0	1	1
1	0	1
1	1	0

(f)

Figure 6.30 T flip-flop: (a) characteristic table; (b) characteristic equation; (c) state diagram; (d) circuit; (e) logic symbol; (f) excitation table.

6.15 Summary Checklist

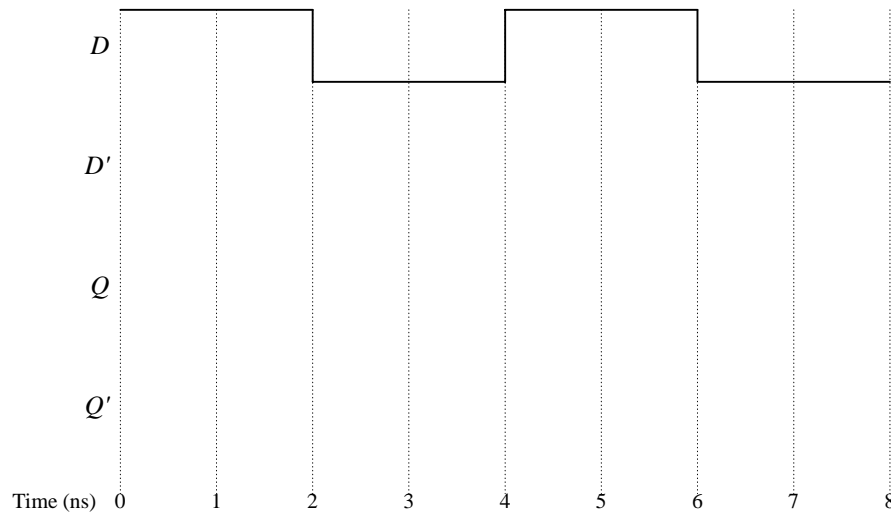
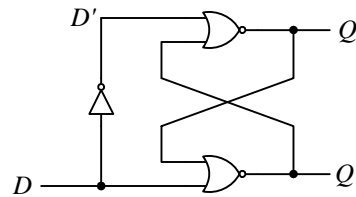
- Feedback loop
- Bistable element
- Latch
- Flip-flop
- Clock
 - Level-sensitive, active edge, rising/falling edge, clock cycle
- SR latch
- SR latch with enable
- D latch
- D latch with enable
- D flip-flop
 - Characteristic table, characteristic equation, state diagram circuit, excitation table
- Asynchronous inputs
- VHDL implied memory element
- SR flip-flop
 - Characteristic table, characteristic equation, state diagram circuit, excitation table
- JK flip-flop
 - Characteristic table, characteristic equation, state diagram circuit, excitation table
- T flip-flop
 - Characteristic table, characteristic equation, state diagram circuit, excitation table

6.16 Problems

1. Draw an SR latch with enable similar to that shown in Figure 6.4, but using NOR gates to implement the SR

latch. Derive the truth table for this circuit.

2. Draw the D latch using NOR gates
3. Draw the D latch with enable similar to the circuit in Figure 6.6(a), but use NAND gates instead of the multiplexer.
4. Draw the master-slave negative edge-triggered D flip-flop circuit.
5. Derive the truth table for a negative edge-triggered D flip-flop.
6. Draw the circuit for an SR flip-flop using SR latches.
7. 6.7 Draw the circuit for the JK flip-flop using SR latches.
8. 6.8 Draw the circuit for the T flip-flop using a JK flip-flop.
9. 6.9 Complete the following timing diagram for the following circuit. Assume that the signal delay through the NOR gates is 3 nanoseconds, and the delay through the NOT gate is 1 nanosecond.



Index

- A**
 Active edge, 3
 Asynchronous inputs, 13
- B**
 Bistable element, 3
- C**
 Characteristic equation, 14
 Characteristic table, 14
- D**
 D flip-flop, **10**
 with enable, 12
 D latch, **7**
 with enable, 8
- E**
 Edge-triggered flip flop, 10
 negative, 10
 positive, 10
 Excitation table, 15
- F**
 Falling edge, 3
 Feedback loop, 3
 Flip-flop, 3
 D, 10
 JK, 23
 SR, 22
 T, 24
- G**
 Gated D latch, 8
 Gated SR latch, 6
- H**
 Hold time, 16
- I**
 Implied memory element, 17
- J**
 JK flip-flop, 22, **23**
- L**
 Latch, 3
 Level sensitive, 9
- M**
 Master-slave D flip-flop, 10
- N**
 Negative edge-triggered flip flop, 10
- P**
 Positive edge-triggered flip flop, 10, 12
 Propagation delay, 15
- R**
 Rising edge, 3
- S**
 Setup time, 16
 SR flip-flop, 22
 SR latch, **4**
 with enable, 6
 State, 3
 State diagram, 14
- T**
 T flip-flop, 22, **24**
 Timing issues, 15
 Transparent latch, 8
- V**
 VHDL
 Clock' EVENT, 18
 flip-flops, 17
 Implied memory element, 17
 latches, 17
 WAIT, 19
 VHDL code
 D flip-flop (behavioral), 18
 D flip-flop (structural), 21
 D flip-flop with asynchronous inputs, 22
 D latch, 18