Chapter 4: Introduction to synchronous logic

Objectives of Chapter 4

- Learn how logic gates are connected to create the flip-flop
- Learn the different categories of flip-flops and their behavior
- Learn about the different circuit configurations of the D-type flip-flop, including frequency dividers, counter, shift registers and storage registers
- Learn how the D-type flip-flop is used to synchronize the transitions in state machines

Up to now we've limited our study of digital logic systems to ***asynchronous*** logic. Asynchronous means "not synchronized". In our system, this means that the change in the state of the output variable depends only on the state of the input variables and the combinatorial logic that links them. Change an input variable and the output variable would change to make the logic correct. There is no delay in process, other than the propagation delay through the combinatorial logic gates. However, within a computer, with millions of logic gates, we must be able to synchronize the change of logic state with some master signal (the clock) so that everything within the microprocessor can progress through a sequence of well-defined states. Therefore, let's now turn our attention to an analysis of synchronous logic.

Look at the circuit configurations in figure 4.1. Consider the upper circuit. Notice how the outputs of the gates are returned to the input of the opposite gate. We call this configuration ***feedback***. Feedback is the screeching you hear when you place a microphone too close to the loudspeaker the amplified sound is coming from. Let's analyze the circuits in figure 4.1. The circuit has two inputs, A and B and two outputs, Q and ~Q. As you'll soon see, these outputs are always the complement of each other, but for now, we'll just call them Q and ~Q.

According to figure 4.1, inputs A and B and output Q are all at logic level '1' and output ~Q is at logic '0'. This means that the upper NAND gate's two inputs are 1 and 0, the lower NAND gate's inputs are both 1. From the truth table for a NAND gate, we see that this circuit is stable because the inputs and outputs are all in their correct logic state.

Now let's perturb this stable system by applying a negative pulse to input A. Things will change very rapidly as a result of the pulse on input A. The output of the lower NAND gate goes to '1' because the inputs are now '1' and '0'. The '1' is now applied to the input of the upper NAND gate, so the output goes to '0', since both inputs are '1'. The output of the upper NAND gate is simultaneously applied to the input of the lower NAND gate, so both inputs are '0' and the output is '1'.

Finally, we remove the pulse that started it all. By removing the pulse, we are simply returning the signal at input A to its prior state. Notice that even though input A returns to logic ‘1’, the outputs remain in their new state because the positive feedback from the upper gate forces the circuit to remain in the state, Q = 0 and ~Q = 1.

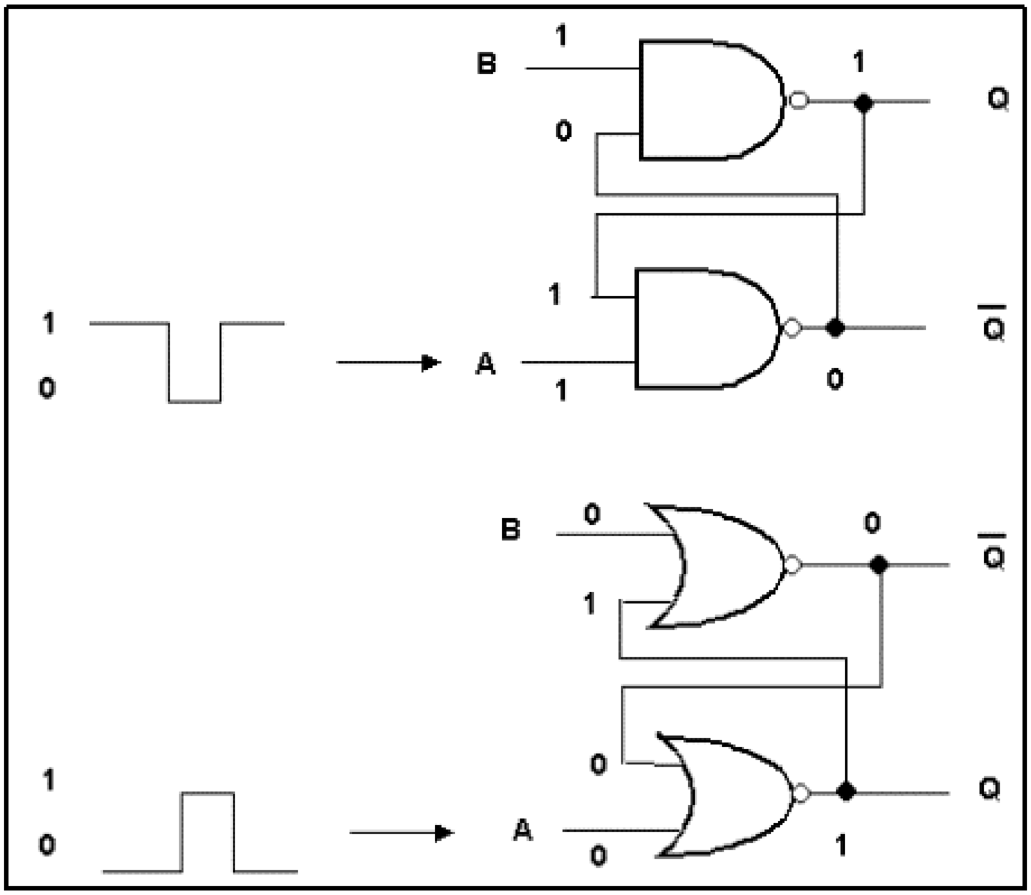


Figure 4.1: The Set/Reset (RS) Flip-Flop. The NAND gate design of the upper circuit is triggered by a negative going pulse. The NOR gate design of the lower circuit is triggered by a positive going pulse.

It should be apparent to you that we could repeat the process with a negative going pulse on input B and the outputs would flip back to their original state. You should be able to repeat the analysis of the NAND gate in figure 4.1 with the NOR gate. In this case, it is a positive going pulse that initiates the state transition, rather than a negative pulse.

Flip-flops

If we apply a second negative pulse to input A of the NAND gate of figure 4.1 the system remains unchanged because the output of the upper NAND gate is still 0. The only way to SET the circuit to the way it was is to provide a negative pulse at input B. Thus, we can see that one input is the SET input (setting Q = 1) and the other input is the RESET input (setting Q = 0). This type of circuit element is called a ***flip-flop***, because the two outputs flip and flop back and forth like a playground teeter-totter. This particular type of flip-flop is an ***RS flip-flop*** because the two inputs alternately reset (R) or set (S) the outputs. We also refer to this type of behavior as ***toggling*** between two states, much like the toggle switch on the wall that controls the lights in a room. In that case, the switch toggles the lights on or off. Figure 4.2 is a schematic representation of the RS flip-flop as a unique circuit element. Here we've taken the two NAND or NOR gates from figure 1 and drawn a different circuit symbol to represent them.

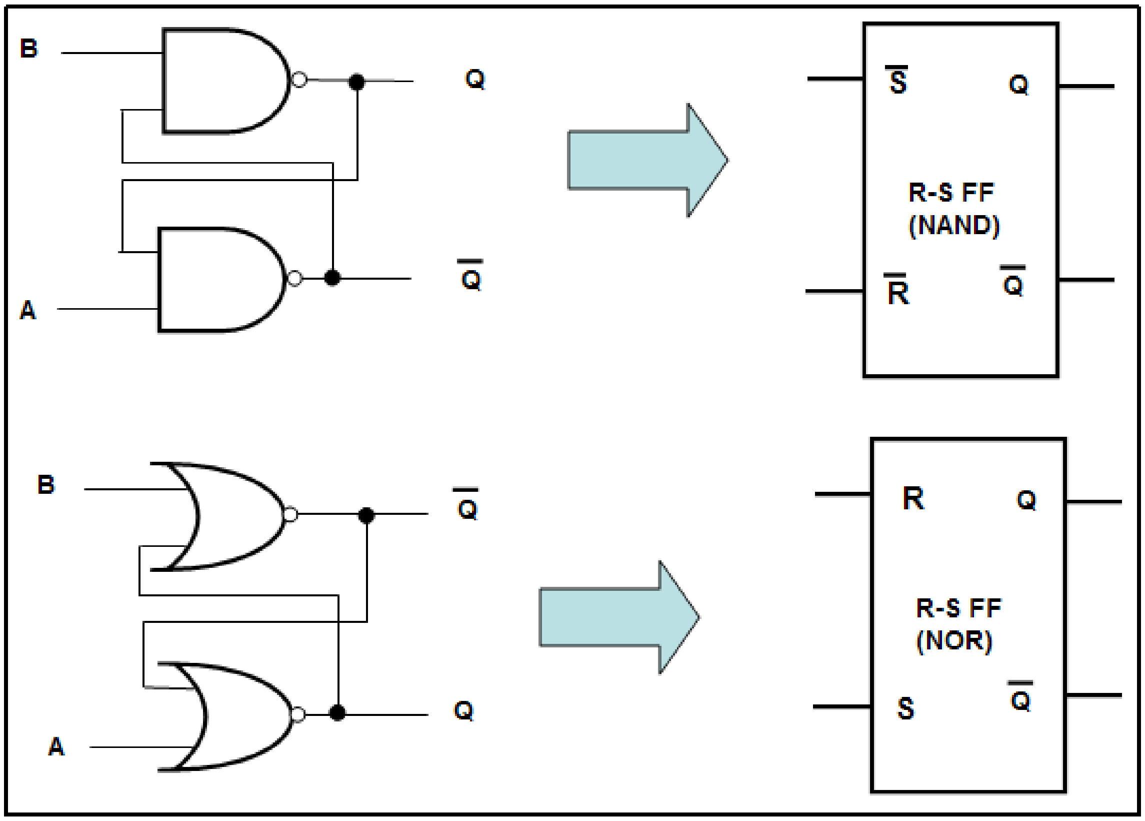


Figure 4.2: The Set/Reset (RS) Flip-Flop as a unique circuit symbol

Figure 4.2 is a trivial example, but it introduces an important concept. In chapters 1 through 3 we've been gradually increasing the complexity of the circuits being considered. For example, we looked at how an electronic switching element, a MOSFET transistor, could be configured as a simple inverter gate and how that basic configuration could be extended to more complex gates, such as NAND. We also saw how a compound gate, the XOR, is given a unique symbol in order to simplify the representation of circuits that contain it. Here, we are extending the concept as we begin to create new and more powerful circuit configurations out of the simpler building blocks. This is a process that we'll be continuing to use throughout this chapter and the rest of the book. Onward!

The RS flip-flop is important because it introduced the concept of ***state dependency***. The state of the RS flip-flop's two outputs is not only dependent upon the state of the two input variables, A and B, it is also dependent upon the previous state of the outputs. This is very different behavior from what we observed with asynchronous, combinatorial logic. Now, the state of the outputs will be a function of the state of the inputs plus the previous state of the outputs. Even though the RS flip-flop is important from a "new concept" point of view, it has limited usefulness in real life. So let's spend some time and see if we can see how this concept is applied in more practical circuit configurations.

The RS flip-flop is an asynchronous device. Pulsing either the S (SET) or R (RESET) inputs will drive the Q and ~Q outputs accordingly. When the inputs are active, the outputs respond. There is no clock signal involved. Thus, we have to ask the question, "Where's the synchronization in all of this?". Good question. It isn't there yet. Let's make use of the gating properties of our NAND and NOR gates to introduce the concept of a clock. Figure 4.3 shows just such a flip-flop design. We call this a ***clocked R-S flip-flop***. For simplicity, we'll only use the NAND-based designs, but you can easily convert it to a NOR design with the appropriate substitutions.

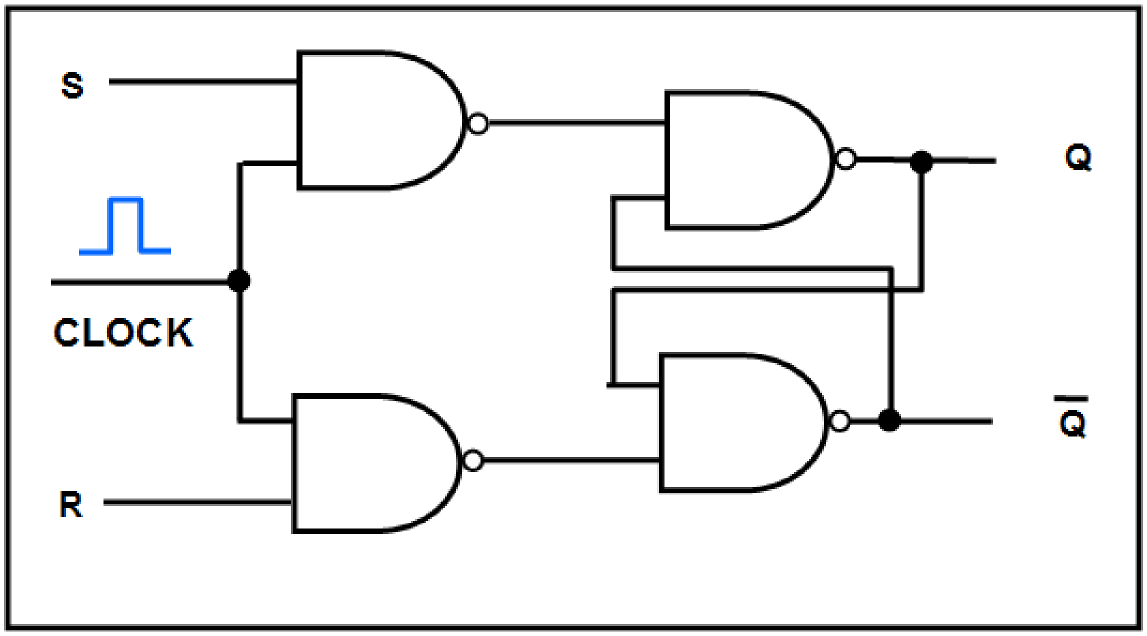


Figure 4.3: The clocked RS flip-flop

Introducing the additional two NAND gates now gives us a method of controlling the R and S inputs with a third input, the clock. Also, the R and S inputs are now active HIGH rather than active LOW with the RS flip-flop because of the inverting action of the two additional gates. Assume that Q = 0 and ~Q = 1. We wish to toggle the flip-flop to the opposite state. Assuming that the clock input is LOW, bringing the SET input HIGH will have no effect because the output of a NAND gate will go LOW only when both inputs are simultaneously HIGH. Thus, we can bring the SET input HIGH, but it isn't until the clock input also goes high will the other flip-flop pair toggle their states.

This is really getting close to what we want, but we still aren't there quite yet. We have introduced a clock synchronization mechanism, but it is a weak synchronization. The problem is that we are dependent upon the actual clock width for the degree of synchronization that we get. If the clock signal remains high for a length of time that is long compared to the time frames of the S and R inputs, then we've lost any synchronization that we hoped to gain. As long as the clock input is HIGH, we can continue to pulse the S and R inputs and cause the outputs to toggle continuously. Ideally, we would like to synchronize the flip-flop with either the rising or falling edge of the clock, rather than the level of the clock. Remember, the edge of a logic signal is the transition from HIGH to LOW or LOW to HIGH. For stability, we require that the transition occur very rapidly, independent of how long the signal remains in the HIGH or LOW state. Thus, even if we have a logic signal that changes state every 12 hours (the PM indicator on a digital clock) we still want that transition to occur in a few nanoseconds.

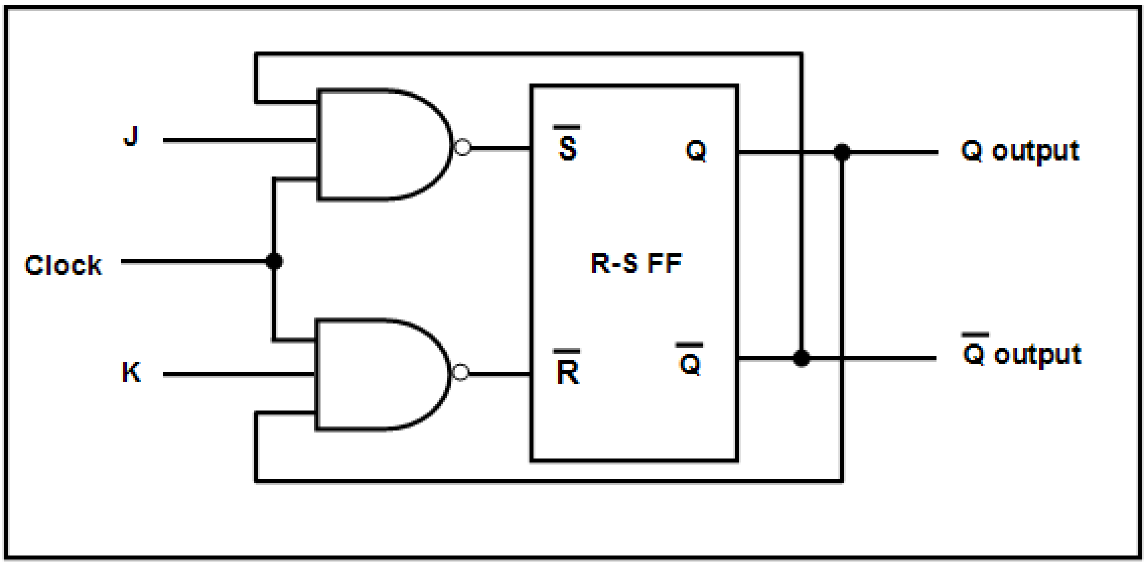


Figure 4.4: The JK flip-flop.

The circuit configuration of figure 4.4 is perhaps the most famous flip-flop of all. It is called the ***JK flip-flop (or "JK FF" for short)***. According to Null and Lobur[1] the JK flip-flop was given its name in honor of Jack Kilby, an engineer with Texas Instruments Corporation who was one of the first inventors of the integrated circuit. Here we've taken the basic clocked RS flip-flop and added two important new features:

1. A second set of feedback loops were added from the Q and ~Q outputs to the inputs of the NAND gates,
2. A second set of inputs, J and K, were added to the NAND gates.

Conceptually, all that we've done is to add the additional feedback path to the circuit by way of converting the gating NAND gates from two input gates to three input gates. However, in the process we've come very close to building a JK flip-flop that actually works the way we want it to. Unfortunately, it doesn't work quite right. The problem is traceable back to the original problem with our clocked RS flip-flop. We were synchronizing with the level of the clock and not the edge of the clock. Here's how we want the JK flip-flop to work. Table 4.1 is the truth table for a JK flip-flop. The last condition, J = 1 and K = 1 is supposed to enable the flip-flop to toggle back and forth each time the clock is pulsed. As you'll soon see this is a desirable circuit behavior for many reasons. However, because we are still at the mercy of the duration of the clock, and when all three inputs are HIGH, we create an unstable state for the flip-flop. To see this, return to figure 4.1 and analyze the circuit for the conditions when both inputs A and B go low at the same time. This type of behavior is also called a ***race condition*** and often leads to unstable and unpredictable circuit operation.

J	K	Q after clock
0	0	no change
1	0	Q = 1
0	1	Q = 0
1	1	outputs toggle

Table 4.1 Truth table for JK flip-flop

Let's try to analyze the behavior of the circuit and see what might be causing the problem. Assume that the flip-flop is in the RESET state (Q=0, ~Q =1). J and K are both HIGH and the clock is LOW. Since we have connected the ~Q output back to one of the three inputs of the upper NAND gate, when the clock goes HIGH all three inputs will be HIGH and its output will go LOW. The LOW output will cause the RS portion of the flip-flop to toggle. That's good, so far, because that's the desired behavior.

As soon as the Q and ~Q outputs toggle, the output of the upper NAND gate will start to go HIGH again. However, the clock signal is still HIGH, so the lower NAND gate now goes LOW and toggles it back the other way. The circuit now exhibits one of two possible behaviors, neither of which can be predicted. Either both the Q and ~Q outputs go high and stay HIGH as long as the clock is HIGH, or the outputs toggle rapidly due to the instability introduced by the race condition. Only careful circuit analysis of the propagation delays and switching conditions will determine exactly how it will behave. In any event, it isn't what we want.

What we are lacking is a mechanism to shift control to the clock edge. The solution is to create a circuit with two gated RS flip-flops in sequence. Figure 4.5 is just the circuit that we need. While this might look quite a bit more complicated, we've only made a few modifications to the basic circuit configurations of figures 4.3 and 4.4. Recall that the basic problem arose when we created the two feedback loops in figure 4.4 in an attempt to create a circuit that would toggle when J = K = 1 and the clock was pulsed high.

In figure 4.5 we've solved the race problem with the addition of the single NOT gate (gate #9) and by adding a second gated flip-flop. The new flip-flop design is called a ***master-slave*** JK flip-flop. Now, before you begin to get visions of chains and whips let me assure you that this is completely benign. The first grouping, the master, is used to bring in the data and stabilize it, and then the data is transferred to the slave, where it appears on the Q and ~Q outputs. The NOT gate and the slave portion of the circuit together create the circuit function that we've so far been lacking. When the clock signal goes HIGH on the master section of the flip-flop (gates 1 through 4) the output of the NOT gate goes low and locks the slave flip-flop, preventing any changes from occurring in its RS output section.

When the clock goes low again, the master flip-flop is disabled and its RS section cannot change state. However, the low going clock enables the slave flip-flop and it changes to agree with its gating inputs.

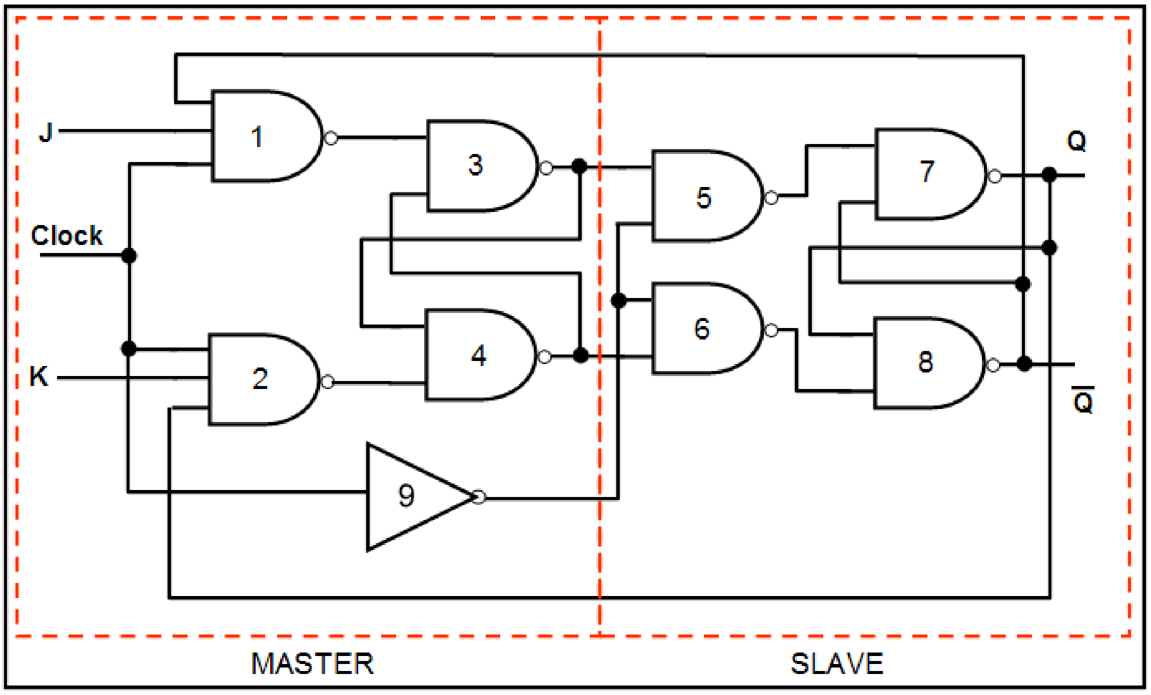
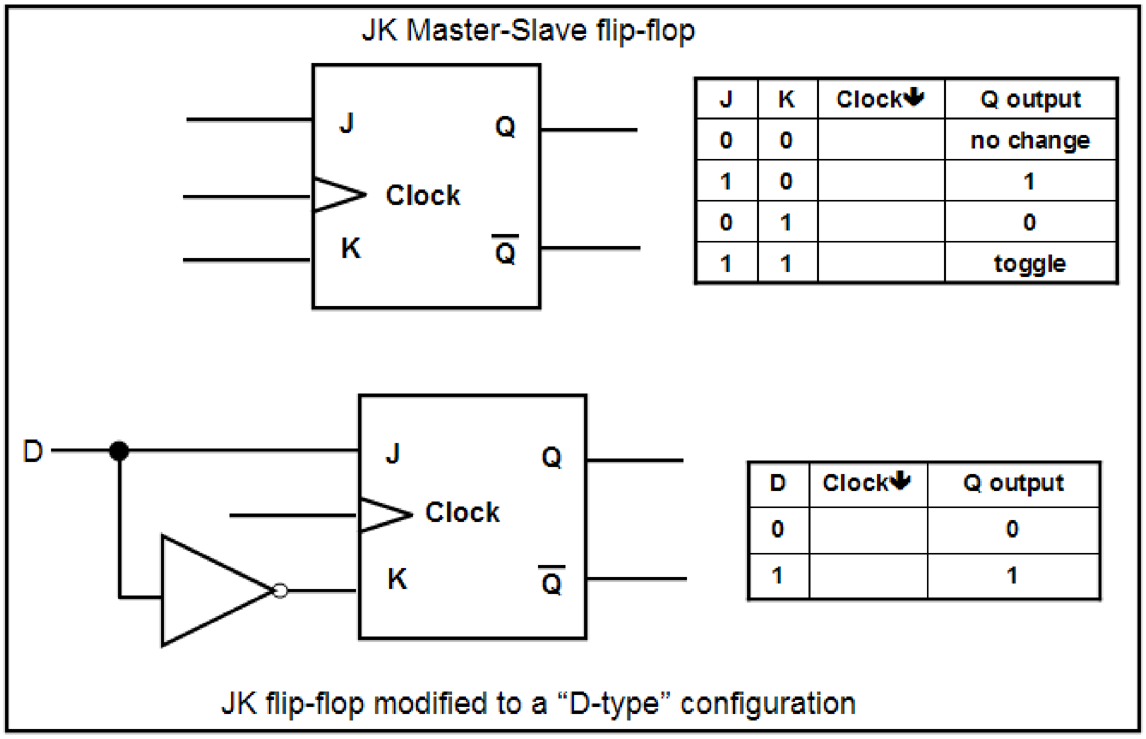


Figure 4.5: The Master-slave JK flip-flop.

The two feedback loops which created all the previous race condition problems now operate properly because we've stopped the race condition. Each section of the flip-flop now operates on either the HIGH or LOW level of the clock, or on ***alternate phases*** of the clock. In other words, the race condition came about because the feedback signal from output to input was free to "race" around the circuit as long as the clock was HIGH. Now we've changed the circuit so that the feedback loop is effectively blocked because only ½ of the circuit is active at a time. The master-slave circuit configuration finally gives us the behavior that we need. The circuit is only active on the transitions of the clock and not on the level of the clock.

Figure 4.6 shows the JK master-slave flip-flop as a unique circuit block, and also shows its truth table. Note that now we use a down arrow to indicate that the outputs change state on the negative transition (HIGH to LOW) of the clock, or when the master portion transfers information to the slave portion. Also shown in figure 4.6 is a slightly modified version of the circuit. By adding an inverter gate as shown, we limit the behavior of the JK flip-flop to only the conditions when J = 1 and K = 0 or vice versa. The inverter gate prevents the other two possible combinations from occurring, J = K = 0 and J = K = 1.

This new configuration, which we'll call the ***D flip-flop (D-ff, or D-flop, for short)***, is the most important version of the flip-flop families and we will focus our attention on this circuit almost exclusively through the rest of the book.


JK Master-Slave flip-flop
J
Q
Clock
K
$\overline{Q}$
D
J
Q
Clock
K
$\overline{Q}$
JK flip-flop modified to a “D-type” configuration

J	K	Clock↓	Q output
0	0		no change
1	0		1
0	1		0
1	1		toggle

D	Clock↓	Q output
0		0
1		1

Figure 4.6: The Master-slave JK flip-flop slightly modified to a ‘D’-type configuration

If you look at the truth table for the D flip-flop in the lower portion of figure 4.6 you might notice an interesting fact. When the clock goes low, Q output changes to agree with the D input. An alternative way of describing this is to say, “On the falling edge of the clock, the data value appearing on the D input is stored in the cell and appears on the Q output.” Now we know why it is called a “D” flip-flop. The ‘D’ means data. This is a memory cell.

Before turn our exclusive attention to the D-FF, we need to make a few introductory remarks. As previously stated, even though we are able to express more complex logical functions in terms of more primitive gates, it doesn’t necessarily follow that the actual design of the more complex function will be truly represented by the logical gate design. Always keep in mind that in most cases, there are circuit shortcuts that can be taken to reduce the overall complexity of the design. Therefore, we should always keep in mind that we are often looking at a logical, rather than actual, circuit implementation. Also, we may want to add additional functionality to a circuit block, or change slightly the behavior of the circuit we are dealing with.

If you examine the digital logic data book of any of the integrated circuit manufacturers, you’ll see perhaps three or four different variations of the standard circuits. For example, you may find a JK flip-flop which changes state on the rising edge of the clock rather than the falling edge. The manufacturers do this to broaden the appeal of the part to the circuit design community who uses them.

For example, if the only JK FF that your company offers is a negative edge-triggered FF and a large customer doesn’t want to incur the added expense of inserting an additional NOT gate into their circuit just so they can convert a positive going clock edge to a negative going clock edge, then you have an unhappy customer. The customer can beat on your Marketing Department and convince them that you need to add a positive edge triggered JK FF to your product offerings, or they can take their business elsewhere, to a company that does offer the parts that they want.

The D-FF is a good example of this. Almost all D flip-flops in use today are positive edge triggered devices. Data is transferred from the D input to the Q output on the rising edge of the clock. In addition, the original functionality of the RS inputs has been retained to give an asynchronous method of setting or resetting the device. We may summarize the behavior of the D-FF as follows:

The logic level present on the D input just prior to the rising edge of the clock is transferred to the Q output on the low-to-high transition (rising edge) of the clock. The Q output will continue to retain the data until the next rising edge on the clock input.

Figure 4.7 illustrates the behavior of the D flip-flop circuit.

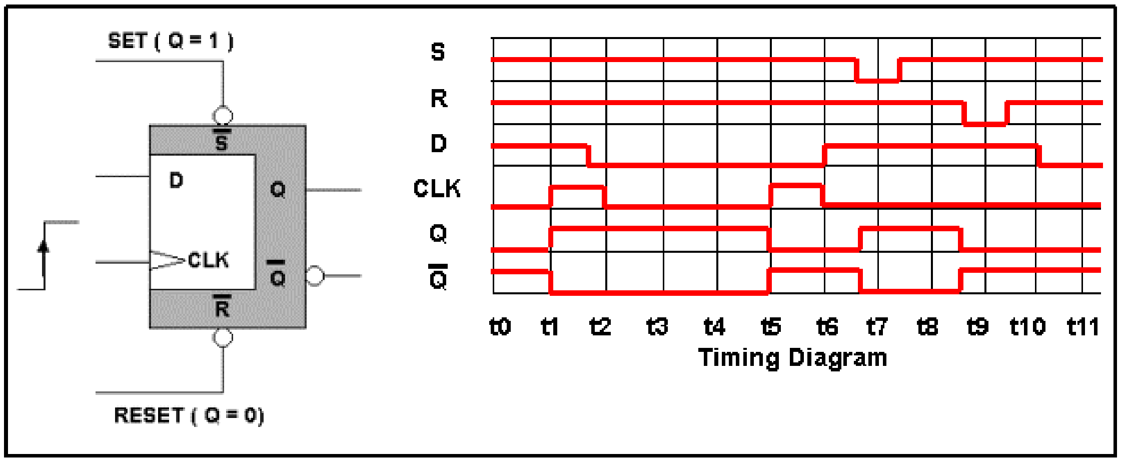


Figure 4.7: The 'D' type flip-flop. The shaded region represents the portion of the circuit that replicates the functionality of the RS flip-flop. The clear portion is the master-slave circuitry which adds the clock and 'D' (data) inputs. The timing diagram for the 'D' flip-flop is on the right.

The timing diagram is just a way of looking at the relationship of several signals with respect to time. If you've ever seen a polygraph (lie detector) test administered on TV or in the movies (or for real!) then you've seen that the strip chart recorder is simultaneously plotting several body conditions at the same time. In other words, our timing diagram is just a polygraph test of a D-FF.

Notice the up-arrow,🡅, on the clock (CLK) input. This symbol indicates that this input is ***edge-triggered.*** That is, it only becomes active when there is a ***rising edge*** on the input logic signal. Thus, only in the briefest instant of time during the transition of the clock input from a low state to a high state will the value on the D input be captured and transferred to the Q output. We indicate that a particular input is edge-triggered by placing the '▷' symbol by the input pin of the device. Inputs that have this type of behavior are only sensitive to the transition of the logic level (a high-to-low transition is also possible) and are not sensitive to the absolute value of the logic level.

The R and S inputs use bubble notation on the inputs to indicate that they are ***active, or asserted, LOW***. We can see this in the timing diagram in figure 4.7. Just after time t6, ~S goes low. After a bit of a propagation delay, Q goes HIGH and ~Q goes LOW. Notice that no CLK input was present. ~R and ~S are asynchronous inputs. They can override the action of the clock. Similarly, ~R is asserted after t8 and the outputs switch back.

The D flip-flop in figure 4.7 is comprised of two separate circuit functions. We know from the previous discussion that the device is actually a master-slave configuration. The ~R and ~S inputs, combined with the Q and ~Q outputs, form an RS flip-flop like the one we studied in figure 4.1. These inputs can override the D and CLK inputs of the "D-type" portion of the circuit so that the outputs may be forced to any state at anytime with the assertion of the appropriate low-going ~R or ~S signal.

Figure 4.8 shows the logical implementation of a D-type flip-flop. Although the master-slave relationship is clearly visible, the circuit implementation is not as easy to discern as the JK example that we looked at earlier.

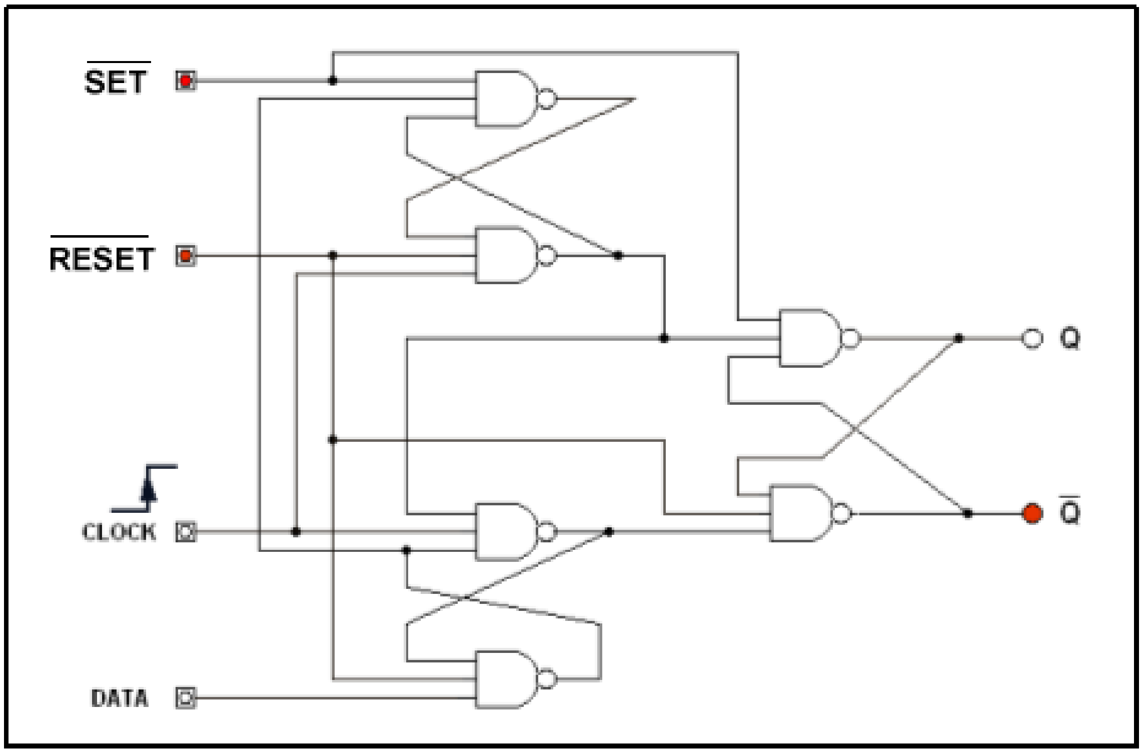


Figure 4.8: Gate level description of a D-type flip-flop.

The circuit in figure 4.8 uses 6 NAND gates to implement the design of the D flip-flop. We needed 8 gates, plus 2 inverter gates, to convert the JK flip-flop to a D flip-flop. Also, notice that there is no feedback from the outputs to the inputs to implement the toggle function. In the case of the D-FF it isn't one of its operational modes, so the circuit is not implemented in a way that adds the feedback.

The toggle function, that is having the outputs change state each time there is a pulse on the clock input, is a very useful feature to have in a digital system, so let's look at it in greater depth. Figure 4.9 shows a D-FF with the toggle function added back in. To add the toggle function, the ~Q output is returned to the D input. In the JK FF the toggle function was implemented by returning the Q output to the K input and the ~Q output to the J input, respectively.

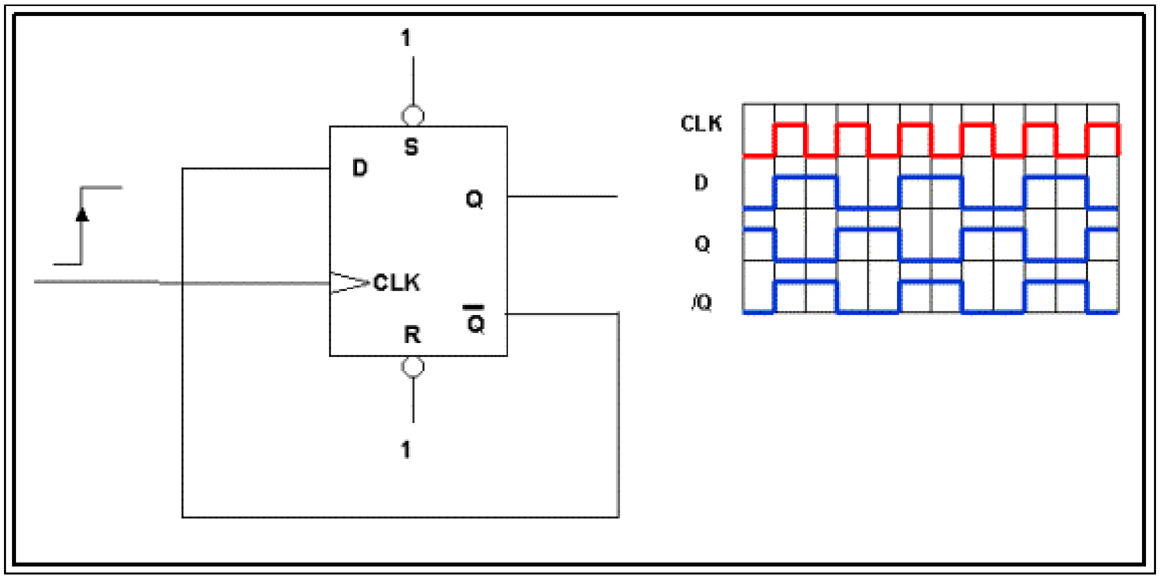


Figure 4.9: 'D' flip-flop connected in a divide-by-two configuration.

Referring to the timing diagram in figure 4.9, we see that when the flip-flop is configured in the toggle configuration, every two clock pulses results in one complete clock pulse on the Q or ~Q outputs. Each time we have a clock pulse it flips the outputs, so we need two clock pulses to flip the outputs back and forth.

Another way of saying this is that the period of the waveform at the Q and ~Q outputs have a period of twice that of the clock itself. Since the frequency is the reciprocal of the period, the frequency of the waveform at the Q output is ½ that of the frequency of the waveform at the clock input. Thus, if the clock input has a frequency of 1 MHz, the Q output has a frequency of 1/2 MHz, or 500 KHz.

If we placed another one of these circuits to the right of the one in figure 4.9 and feed the ~Q output into the clock input of the next one, the frequency would be halved again. We can see this in figure 4.10. Referring to figure 4.10, the waveforms for the input clock and the resulting changes in the Q1 and ~Q1 outputs are the red blue and green lines, respectively. This is identical to the circuit of figure 4.9. Now, if we connect ~Q1 so that it becomes the clock input for the second D-FF on the right, then the circuit behavior is exactly the same, except that the period of all the resulting waveforms have been doubled (frequencies have been halved).

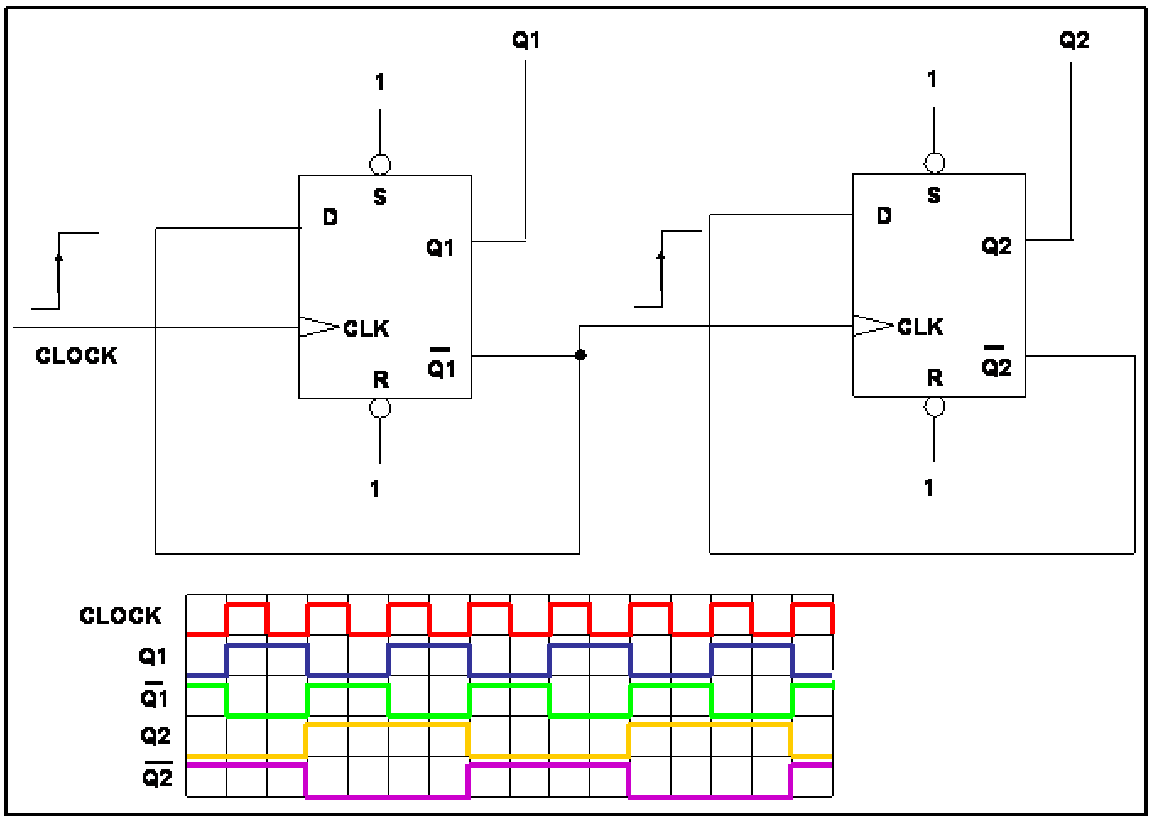


Figure 4.10: Two D-FF's connected in series. The ~Q output of one FF becomes the clock input for the next FF. The resulting transitions of the clock input and Q outputs are shown in the timing diagram.

Clearly we could continue to do this exercise with as many D-FF devices as we want to add. Each time we add a D-FF, the output period of that device is twice the period of the device that feeds it. In general, for a string of N D-FF's, the period of the N^{th} FF is simply 2^N times the period of the input clock to the first flip-flop in the chain. With enough flip-flops in a chain we could easily reduce an input waveform of several megahertz to a leisurely 1 clock cycle per second.

Now let's go a step further in order to better see another property of this chain of D-FF's. In figure 4.11 we see a chain of 4 D-FF's. The only other difference is that the ~R inputs are all tied together and presumably go to some location that can assert a RESET pulse at some time. Thus, whenever we apply a negative pulse to the four ~R inputs of this circuit, all of the Q outputs will be forced to go to 0, independent of the arrival of a clock signal. This is very convenient if we want to start the circuit running from a known state. In fact, that's exactly what the **RESET** button does on your PC.

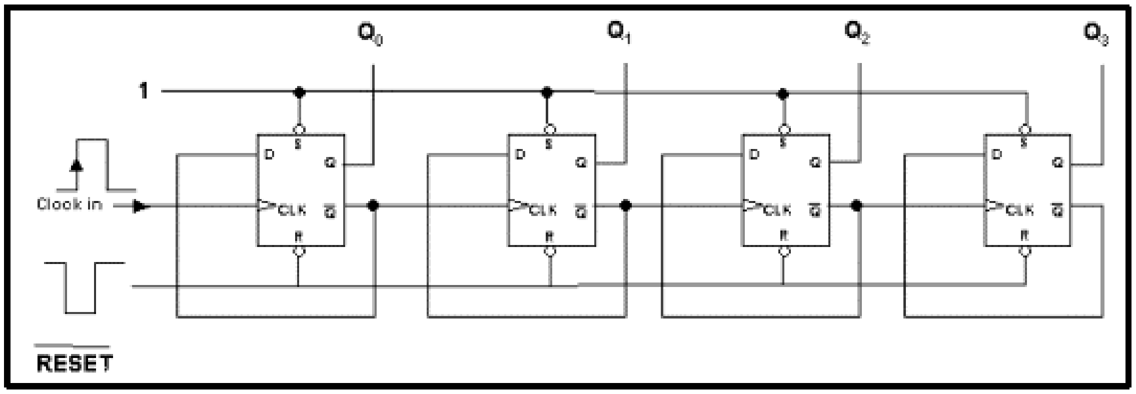


Figure 4.11: 'D' flip-flops configured as a 4-bit counter and divide-by-16 frequency divider.

In figure 4.11 we show four of the circuits from figure 4.9. Notice that we've tied all of the ~S inputs together and we're going to permanently hold them at a logic level of 1. Thus, we'll never have an occasion to "set" the Q outputs to 1. Tying an unused input to either logic level 1 or 0 is quite common in digital design and assures us that the circuit won't accidentally change state if that input were to see some noise. From our discussion of the RS flip-flop, we know that asserting the ~R input low will force the Q outputs to go to 0, thus giving us a known starting condition.

From our previous discussion, we can see that each successive D flip-flop, or stage, of the frequency divider divides the incoming clock by a factor of 2, so the output frequency, at the Q_3 output is one 16th of the frequency at the clock input, four stages to the left. Thus, a 16 MHz clock signal at the CLK input would be a 1 MHz clock signal at the Q_3 output. We can see the frequency division process in action by referring to the logic analyzer display of figure 4.12. Unlike the oscilloscope display that shows us a high-fidelity view of the waveform, the logic analyzer display can show multiple waveforms at one time, but loses some information in the process. Thus, the logic analyzer view is more like figure 3.15.

Let's look at figure 4.12 in some detail. The clock is the uppermost waveform and the Q output of each successive D flip-flop is represented by the next lower waveform. Let's compare the CLK waveform with the Q0 waveform directly below it. Figure 4.13 is a magnified portion of figure 4.12. The rising edges of the clock are labeled for ease of identification. Notice that on each rising edge of the CLK input, the Q0 output changes state. However, the D flip-flop requires a rising edge to cause the output to change state, so the net effect is that it requires 2 input rising edges to cause the Q0 output to go through one complete cycle itself.

There's one more interesting feature of the circuit configuration of figure 4.11 that we need to reiterate. We previously discussed that all of the ~R inputs are tied together so that asserting them will force the Q outputs to 0, independent of the state of the clock input. Thus, to start the system from a known state (all outputs = 0) we must assert the RESET input.

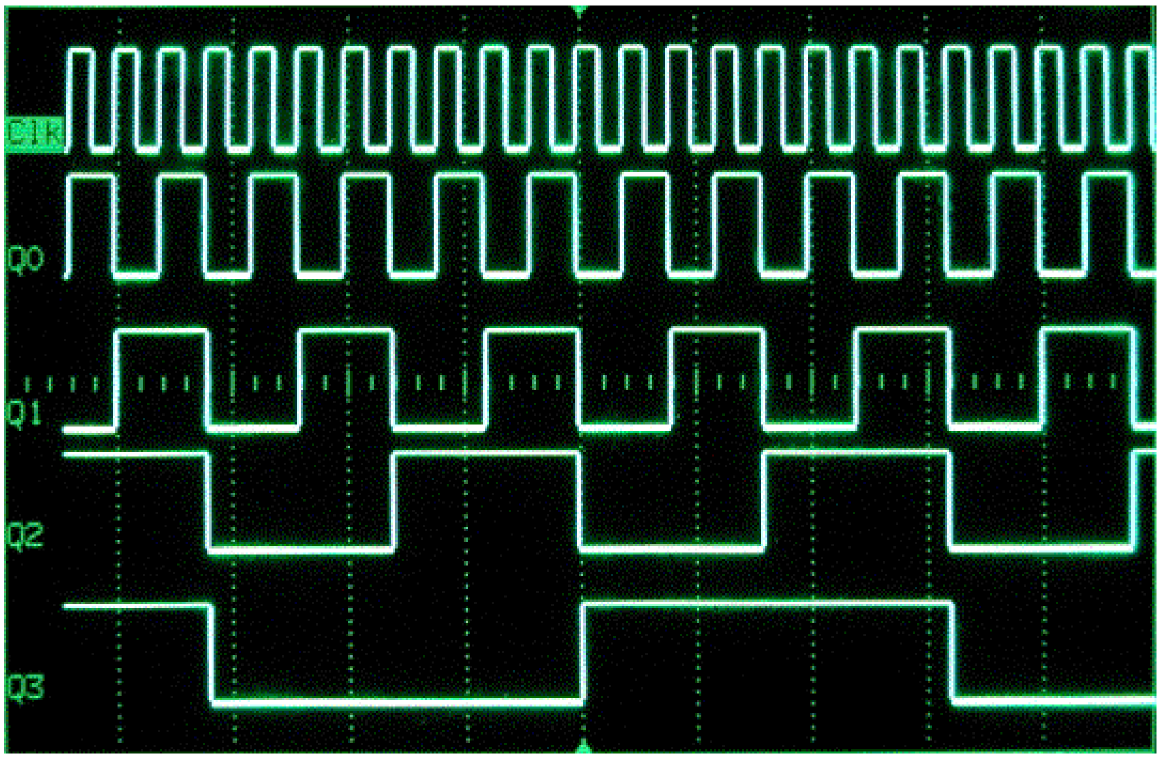


Figure 4.12: Logic analyzer display of a divide-by-16 circuit made from 4 ‘D’ type flip-flops.

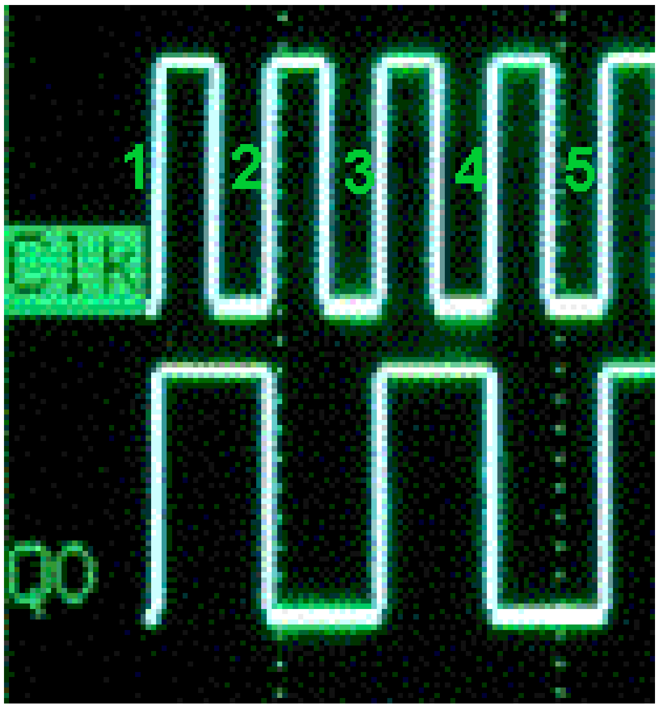


Figure 4.13: Magnified view of the clock input and Q0 output part of figure 4.12

Assume that we've just asserted RESET and outputs Q_0 through Q_3 are all 0. There are no clock pulses coming in at the CLK input on the left. The circuit is quiet and life is good. Suddenly, there's a knock at the door. Oops, wrong book. Suddenly, a single clock pulse is received at the CLK input. Since the ~Q output = 1, the D input is 1, so after the clock pulse, the Q output of the leftmost D-flop goes to 1. The ~Q output goes from 1 to 0. That's a falling edge, so there is no change at the second D-flop.

When the second clock pulse arrives at the CLK input, the first D-flop changes back and its output goes to 0. However, since its ~Q output now goes from 0 to 1, Q_1 now goes to 1. This is just what we would see on figure 4.12 if we looked down a column on the waveform display after each CLK input arrives. As an exercise, make a truth table listing each clock pulse as a numeric entry down the leftmost column and the values of Q0 through Q3 as the four output columns. Start from clock pulse #0, or the RESET condition.

If you complete the suggested exercise, you'll quickly see that the circuit is also a binary counter that counts from 0000 to 1111 and then rolls around to 0000 on the 16th clock pulse. Not surprising, but we could continue to add counting stages to form a counter of any arbitrary length. This particular counting configuration is called a ***ripple counter***, because the count ripples through the counter as each stage changes state. The fact that the count 'ripples' through the counter means that we could be receiving the next pulse before the count change from the previous pulse completely ripples through the counter. Although the counter will still maintain an accurate count, it does limit our ability to accurately read the current count in the counter.

Up to now we've been focusing on digital systems where each input or output variable occupies its own wire. If we want to transport 32 bits of data within a microprocessor, we need to create a bundle of 32 wires, or traces, in order to move the data around. This is called ***parallel*** data distribution because each individual data bit, or variable, travels in parallel with all the other data bits. However, there are some very good reasons to use ***serial*** data transmission methods. In serial data transmission, all of the data bits travel in sequence along a single wire, or at most, only a few wires. Some examples of parallel data transmission protocols that you might be familiar with are:

- Parallel port, LPT:
- PCI bus
- AGP bus
- IEEE 488 (HP-IB)

Examples of serial data transmission protocols are:

- RS232 (COM ports, COM1…COM4)
- Ethernet
- USB
- Firewire

We'll now look at a circuit, built from the D flip-flops, that allows us to go from serial data formats to parallel data formats. Obviously, this is quite important because we must have a way to manipulate the data within the computer once the serial data is received. The circuit structure that we use is called a ***shift register***. Consider figure 4.14.

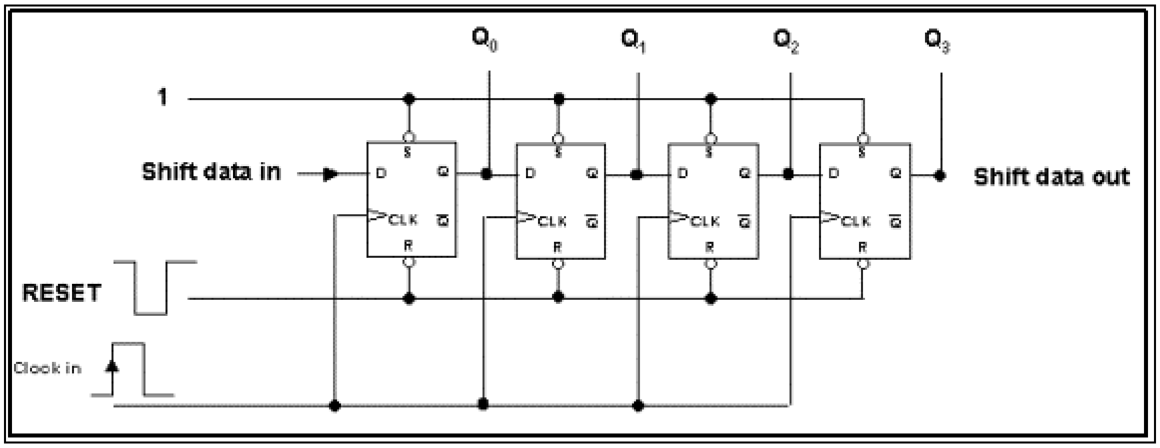


Figure 4.14: 4-bit shift register built from 'D' Type flip-flops.

The RESET inputs (R) are tied together, just like the ripple counter of figure 4.11. This allows us to simultaneously drive all of the outputs to 0. However, the big difference is that we have also tied the CLK (clock) inputs together and each Q output goes directly to the D input of the next stage. Thus, each D-flop will capture the value of the Q output to its left when the rising edge of the clock is applied to its CLK input, but because all of the CLK inputs are activated simultaneously, the data applied sequentially at the leftmost D-flop appears to move left-to-right through the shift register with each rising edge of the clock.

Let's see how this would look as a waveform. Suppose we want to send the number 6, or 0110 in binary, from one device to another along a serial cable. The method of translating the parallel data to serial is similar to the process of translating it back to parallel, but we won't consider that process here. We'll just assume that the data, 0110, now appears as the waveform traveling with a clock signal, as shown in figure 4.15.

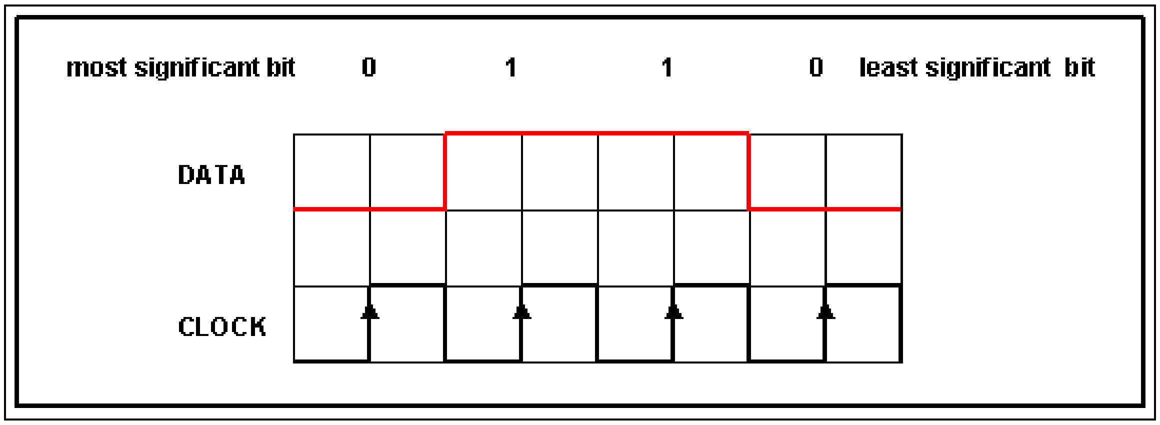


Figure 4.15: Serial data transmission of the binary number 6.

The data value is synchronized with the rising edge of the clock, which may, or may not, be transmitted along with the data. Sometimes, a clock signal will be generated locally by the receiving device, rather than transmitted with the data. This is the way the COM ports on your computer operate. Other techniques involve combining the clock with the data and transmitting them together along a single wire. Special circuitry is used to recover the clock signals from the data at the receiving end. However, in all cases, a clock signal is required to synchronize the capturing of the data at the D input of the flip-flop. Thus, along with each of the four clock pulses there corresponds the receipt of one of the four data bits.

Figure 4.16 shows the progression of the four data bits through the D flip-flops. Let's assume that in this example we are transmitting both the data and the clock along two separate wires, just as we've shown in figure 4.15. Remember that figure 4.15 is actually a graph in time, so that if we had a very fast pencil and paper, each time that a rising edge of the clock occurs, we would record the state of the data on the other wire, we would see ‘0’, ‘1’, ‘1’, ‘0’, respectively. Although it may appear a little confusing at first, we can analyze the behavior as follows. Just prior to the clock edge at t1, the D input to the leftmost D-flop is ‘0’ and all outputs have been RESET to ‘0’. At t1, the 0’s at the D inputs are transferred to the Q outputs, but since they were already ‘0’, nothing appears to happen.

Now just prior to t2, the value at the leftmost D input becomes ‘1’. Just after t2, Q0 goes to 1, to reflect the value at the D input. All other outputs are still 0 because we are still moving the first 0 through the chain. At t3, the third data bit, a ‘1’, is clocked into the leftmost D-flop, and everything else moves to the right. Thus, just after t3, the Q0 through Q3 outputs are 1100, respectively. Finally, just after t4, the data bits shift one more time and the data is now completely stored in the shift register as a parallel value.

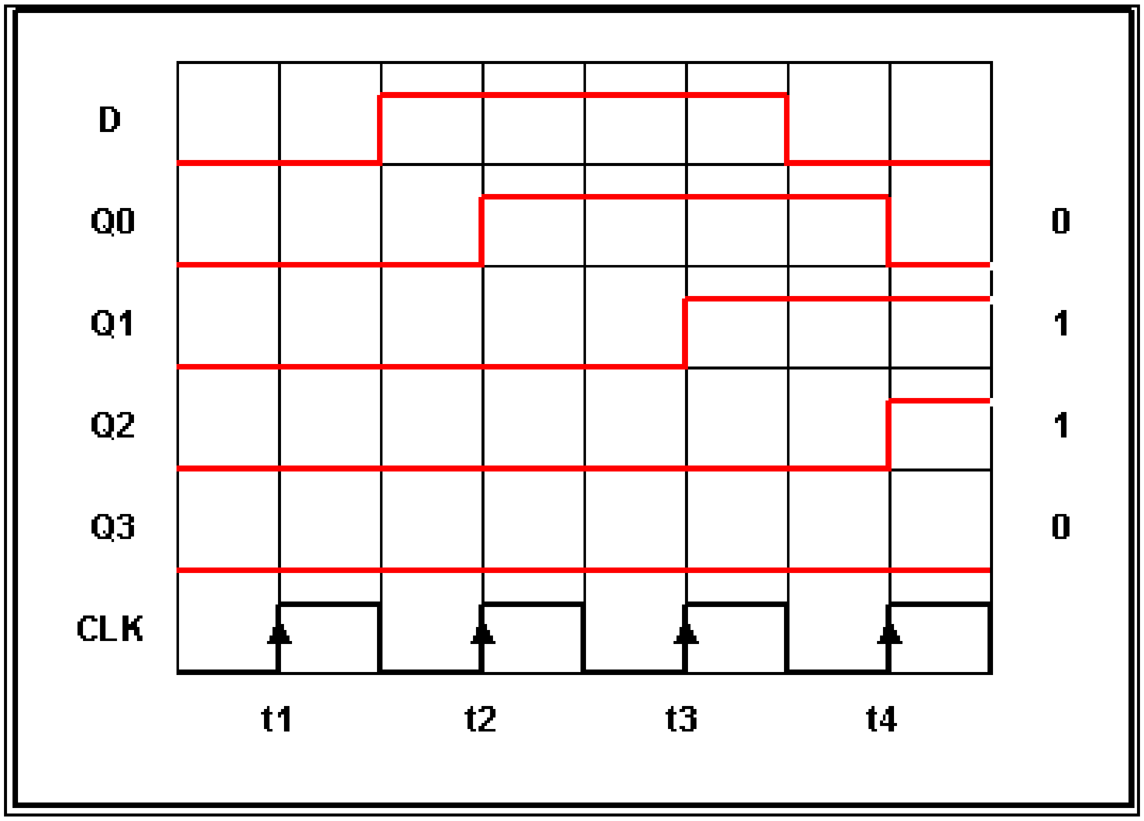


Figure 4.16: Propagation of the serial data through the 4-bit shift register

Storage Register

The final example of a D-flop circuit configuration that we're going to look at is perhaps the most important of all, the ***storage register***. The storage register is a grouping of D-flops designed to receive and hold a digital data value. The value can be a bit, nibble, byte, word, long word, double word, and so on. The key point is that a single clock signal forces all the D-flops to store the data at their respective D inputs at the same time. In this sense, it is a parallel in/parallel out device.

In contrast to the storage register, the shift register of figure 4.14 is a serial in/parallel out device. The reason that we need the storage register is that the data flowing through computers is extremely transitory, and we must find temporary storage locations where we can keep variables until we need them, or to store the results of instructions until we need to move them somewhere else.

Figure 4.17 shows eight D-flops in the circuit configuration of a storage register. We can see that any data present at the D inputs (the blue wires in the figure) will be stored by the flip-flops in the register after a rising clock edge. The data will then be available for the output data (red wires) to transmit. However, the key point is that the input data values (the signals on the blue wires) can now change, but the register is still holding the previous data stored in it.

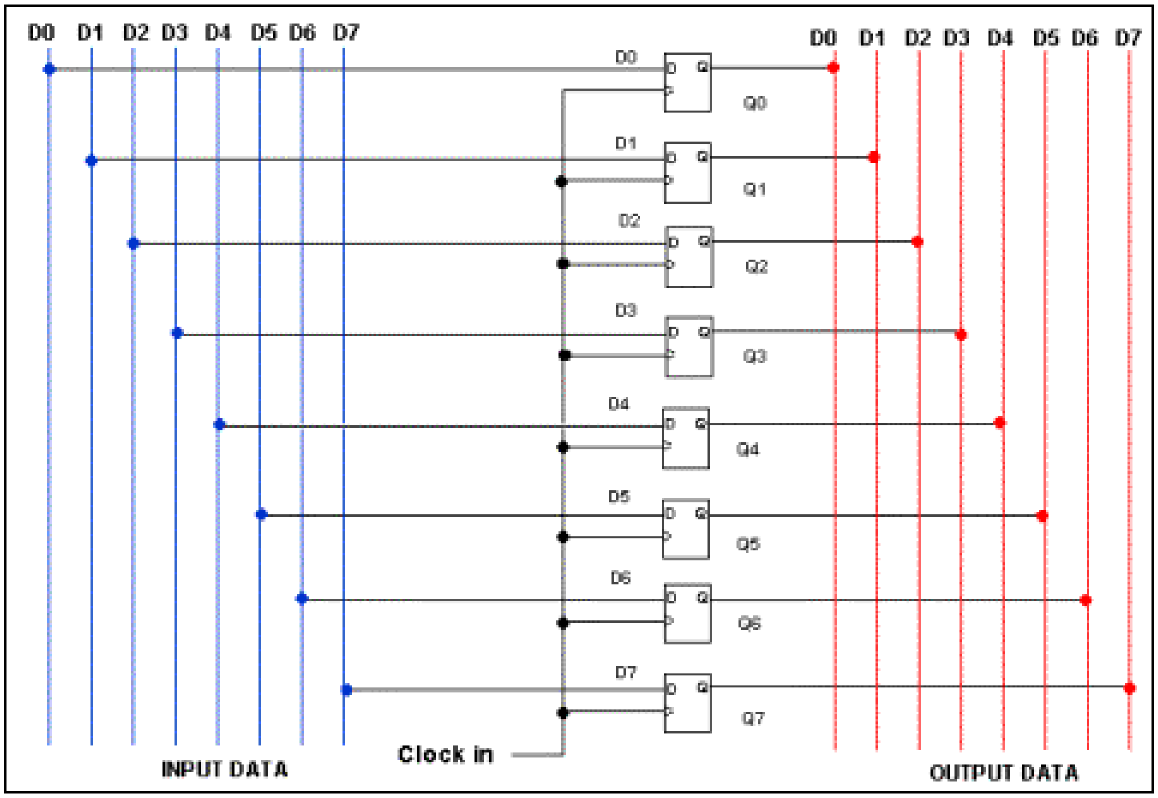


Figure 4.17: An 8-bit storage register. The input data (blue wires) is captured in the D-flops on the rising edge of the clock. The data then appears on the outputs (red wires). When D-flops are used in registers they typically do not have SET and RESET inputs, which is why those inputs are not shown in the figure.

In most computers, the number of D-flops that form the storage register matches the number of data bits in the computer's data path. Today, we use computers with data path widths from 4 bits wide to 128 bits wide. These registers are the key "data containers" in modern computers and microprocessors. You'll see in later chapters how, in many ways, the number and type of storage registers in a computer defines the architecture of that computer.

In order to reinforce the concept of the storage register as a unique circuit element instead of a collection of D-flops, we can redraw figure 4.17 as shown in figure 4.18.

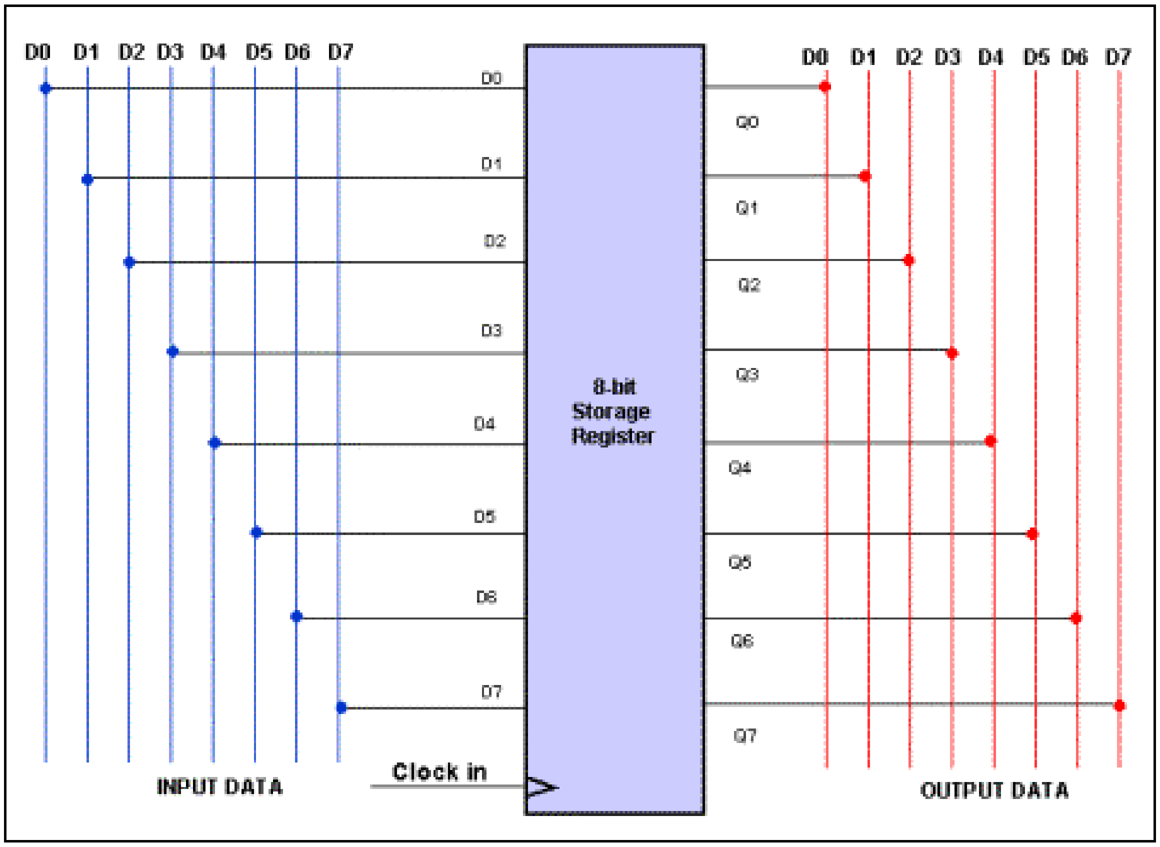
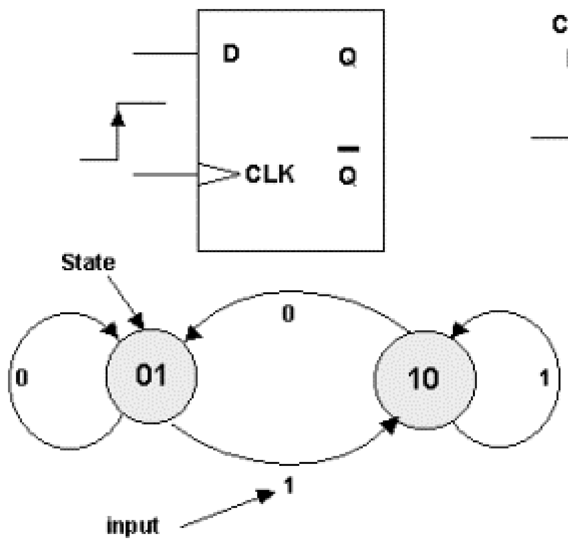


Figure 4.18: The circuit of figure 4.17 is redrawn to show the 8 individual D-flops aggregated into a single device. The storage register is an integral part of most computer systems so it should be considered apart from the D-flop.

Earlier in this chapter we've repeatedly referred to the "state" of a flip-flop. By that we were really saying that in order to know what the new values of the Q and ~Q outputs will be, we had to know what the current value (or state) of the inputs and outputs were just prior to the arrival of the appropriate clock edge. What we are leading up to with this simple example is the concept of a ***state machine***.

The states of a state machine represent all of the possible combinations that the outputs and inputs of a digital system can exist in; as well as the paths that the system may take as it transitions through the different states of the machine. Also, and perhaps most importantly, the state machine can change the path it takes because of changing conditions on its inputs.

Unlike the asynchronous combinatorial logic that we studied earlier, state machines are synchronous devices. This means that a state machine can only change state with the arrival of a clock edge. We can infer from this that somehow the D-flop and the state machine have some common lineage, and you're correct! We can best describe the behavior of a state machine with a table or a graphical picture (called a ***state transition diagram***). Consider figure 4.19.



Current Input D	Current State Q	Next State Q^n
0	0	0
1	1	1
1	0	1
0	1	0

Figure 4.19: State transition diagram and truth table for a 'D' type flip-flop. The asynchronous SET and RESET inputs are omitted for clarity.

The shaded circles in figure 4.19 are the states of the system. We have two states in the system, 01 and 10, corresponding to the conditions Q = 0, ~Q = 1 and Q = 1, ~Q = 0, respectively. The arrows represent the possible state transitions and the numbers on the arrows represent the input condition at the time of the transition. Thus, when the device is in the state 01, it will remain in that state as long as D = 0, but will transition to the 10 state if D = 1. Similarly, if it is in state 10, it will remain in that state if D = 1 when the clock edge arrives, but it will transition to the 01 state if D = 0.

The transition between states occurs on the rising edge of the clock, which is implied by the arrows. Even though the D-FF remains in a given state after the clock edge, we still use an arrow to show that it transitioned back to its current value.

The truth table view (the right side of figure 4.19) provides the same information, but is less intuitive than the state transition diagram. For simplicity, we've omitted the ~Q output of the D-FF in the truth table. The 4 rows of the table are the 4 possible state transitions that can occur. Note, however, that the output variable appears in two contexts, its value ***before and after*** the arrival of the rising edge of the clock pulse.

Figure 4.20 is the state transition diagram for the 4-bit counter example we considered in figure 4.11. Since there are no other synchronous inputs to the counter, other than the clock, the diagram is very straightforward. No matter what state the counter is in, the next state is always predetermined. If the RESET input could be made synchronous (and it is easy to do, but we won't bother) then every bubble would have an arrow back to the 0000 state. These RESET arrows would have the label RESET = 0 on them to indicate that a 0 on the RESET input would force the counter to return to 0000 on the next clock pulse. All of the other arrows that indicate a transition to the next counter state would have the label RESET = 1, to indicate a normal state transition.

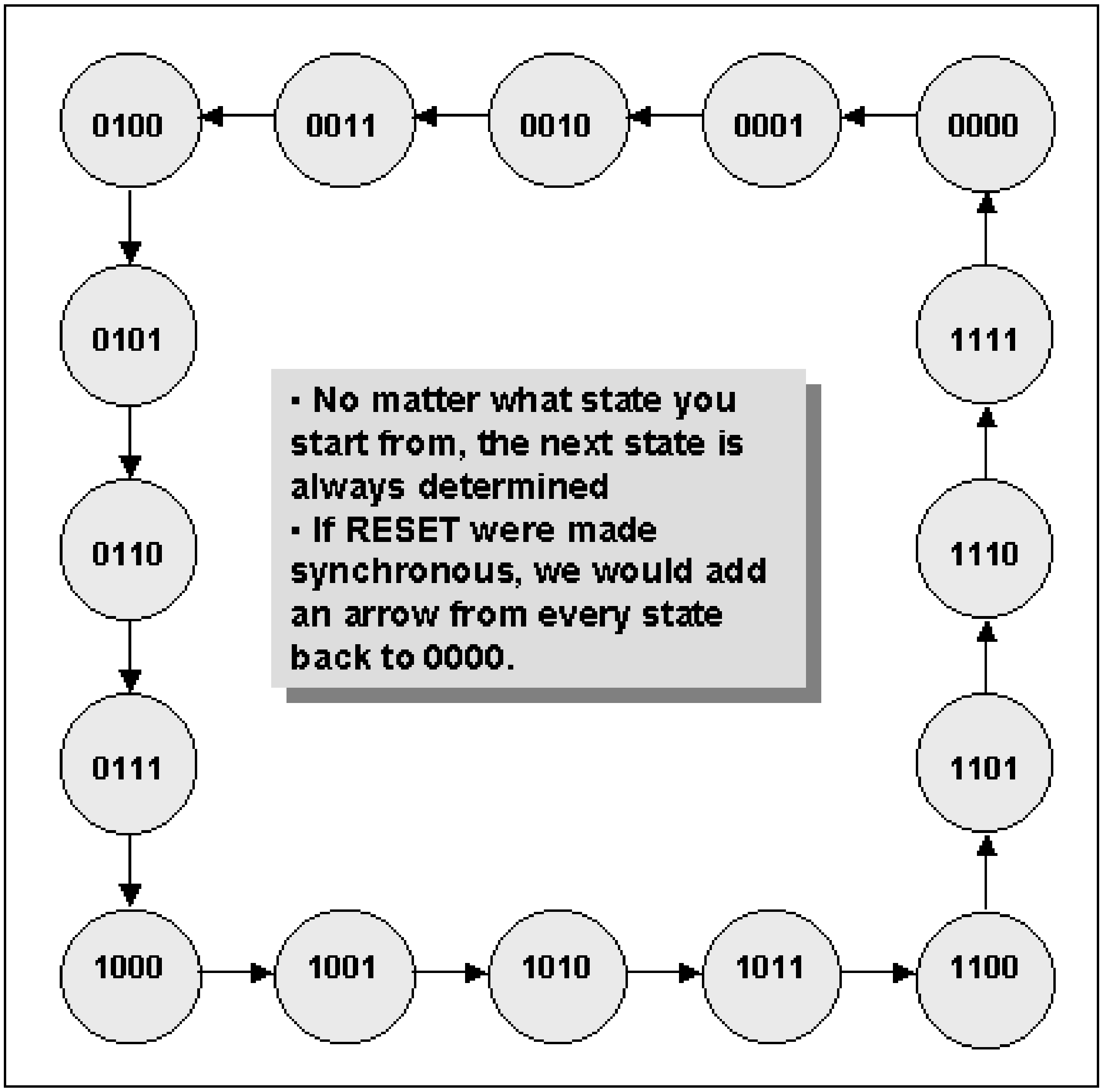

0100
0011
0010
0001
0000
0101
1111
0110
1110
0111
1101
1000
1001
1010
1011
1100
• No matter what state you start from, the next state is always determined
• If RESET were made synchronous, we would add an arrow from every state back to 0000.

Figure 4.20: State transition diagram for a 4-bit counter.

Earlier in chapter 3 we considered that one possible way to implement asynchronous logic was to directly implement the truth table in a memory device, such as figure 3.9. This would eliminate all of the gate design issues at once. We didn't discuss the reason for one system over the other, but we book-marked the idea for later elaboration. Let's elaborate! However, before we dive into the state machine it might be helpful to review the architecture of a computer memory. We'll be studying memories in much more detail in a later chapter, but for now, let's review what you already know about memories from your programming classes.

A memory device, such as a RAM or ROM memory chip contains a total number of ***memory cells***. Each cell is capable of storing a single 1 or 0 value. We can imagine that the cells may be organized into various combinations to form memories of different ***memory widths***. Once we know the total number of cells in the memory and the width of the memory we know how many ***memory addresses*** are required for this memory device. An example would probably help here. Suppose that we have a memory device that contains a total of 16,384 memory cells. Each memory cell stores a single **bit** of data. We want to design this memory device in such a way that each memory access will give us 8 bits of data. In other words, we want to store ***chars*** in this memory. Therefore, the number of unique addresses is 16,384 / 8 = 2,048. Thus, our memory device will contain:

- 16,384 total memory cells
- 8 lines for reading and writing data to the memory
- 2,048 uniquely addressable memory locations
- 11 address lines for supplying 2,048 unique address combinations.

You might be wondering where the 11 address lines came from. Consider that we need to be able to address 2,048 unique address locations. Thus, the first address location should be given address 0000 and the last address location is given the address 2,047. It might seem strange that all zeroes is a valid address, but Computer Scientists are a hardy breed and we can deal with it!

In binary, the 2,048 unique addresses are represented as going from 000 0000 0000 to 111 1111 1111. Notice that we need 11 binary digits to represent all the possible addresses from 0000 to 2,047. It is a general rule that since 0000 is a valid address that the highest address number (all binary digits equal 1) will be one less than the number of unique addresses.

So, you can think of this particular memory device as being organized as 2,048 by 8. We can change our specification and say that we want to store 32-bit wide integers in this memory. Since there are more memory cells per address location, there must be fewer addresses to deal with. Since the memory is 4 times wider then before, we'll have 4 times fewer addresses. Our new memory device would have:

- 16,384 total memory cells
- 32 lines for reading and writing data to the memory
- 512 unique memory addresses
- 9 lines for supplying 512 unique address combinations.

So what does this have to do with state machines? Recall that Figure 3.9 was a truth table that looked suspiciously like a memory. The truth table in Figure 3.9 had 4 independent input variables, A through D and two dependent output variables, E and F. From our previous discussion, this is a memory with 4 address lines and a 2-bit wide output. The memory contains 32 memory cells.

The above discussion is not meant to imply that a memory is the only method that we could use to store the state transition information for our state machine. Since the memory is really an exact mapping of the truth table we could use the methods that we learned earlier in this chapter. That is, build a Karnaugh Map for each output variable and develop the simplified minterm equations for each. Then, we could construct the gate representation of the truth table and have our state machine. How exactly you choose to implement the design is determined by many factors. We'll put our trust in the hardware designer to do the right thing.

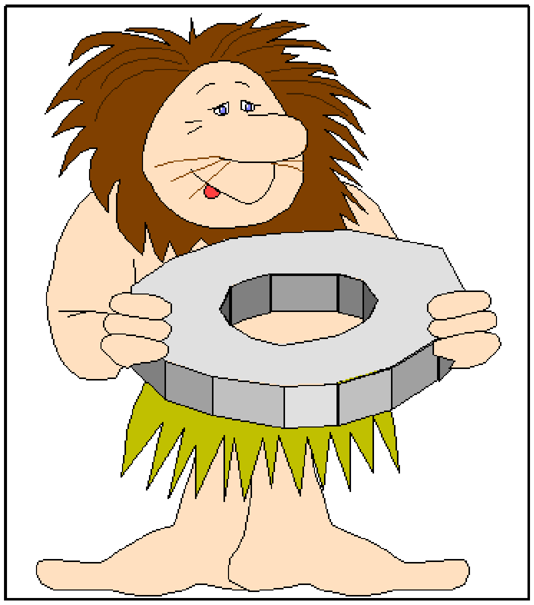


Figure 4.21: Generic Hardware Designer[2]

Now, suppose we want to build a typical state machine. A microprocessor is a good example of a state machine. How would we do it? Well first, we would probably need to take a few more courses in Computer Science and Electrical Engineering, but we won't let that stop us. We'll look at the big picture. If we really want to build a microprocessor, we would have to design a state machine for its brain. The state machine would have thousands of possible states, with hundreds of input and output signals (variables). Such a state machine would be difficult to design using logic gates, but would be easier to design using a memory to implement the logical truth table. Consider figure 4.22.

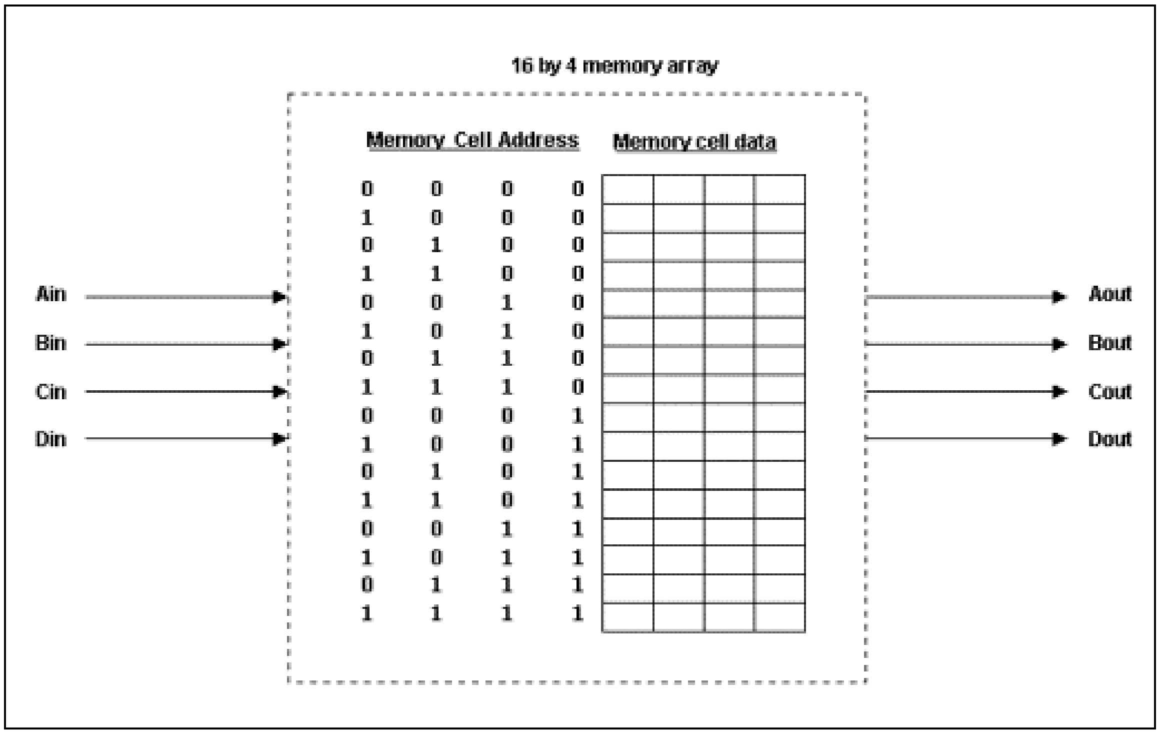


Figure 4.22: Organization of a 16 by 4 memory array as the truth table for a state machine

There are four input variables, A_{in} through D_{in}, and four output variables, A_{out} through D_{out}, respectively. The input variables form the address of a particular memory cell, 0000 through

1111 and output variables are just the data contained in the cell with the input address. This is shown in figure 4.23.

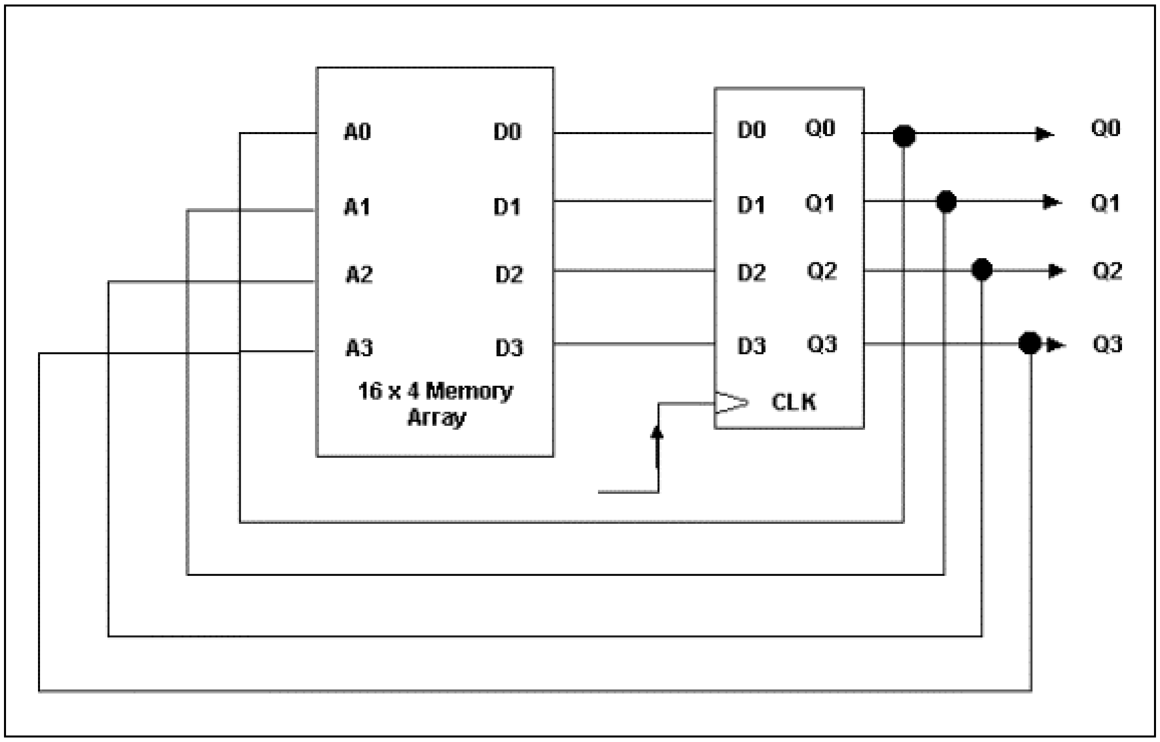


Figure 4.23: State machine implementation using a 16x4 memory array and a 4-bit storage register. The output variables are fed back to the input to provide the address value for the next state.

In figure 4.23, A0 through A3 are the addresses of the memory cells and D0 through D3 are the corresponding 4-bits of data stored in each cell. The outputs of the state machine, Q0 through Q3 drive out to the rest of the circuit, and are also routed back to provide the address for the next state. The 4-bit storage register provides the critical synchronization function. Since the data only transfers from the D inputs to the Q outputs on the positive clock edge, the register isolates the input and output transitions from each other and prevents the circuit from running in an uncontrollable fashion.

The 4-bit 'D' storage register provides exactly the same functionality in the state machine as the master-slave circuit did in the edge-triggered flip-flop. Without the D-FF to provide the appropriate synchronization our circuit would race uncontrollably. To see why this is so, consider the circuit shown in figure 4.24.

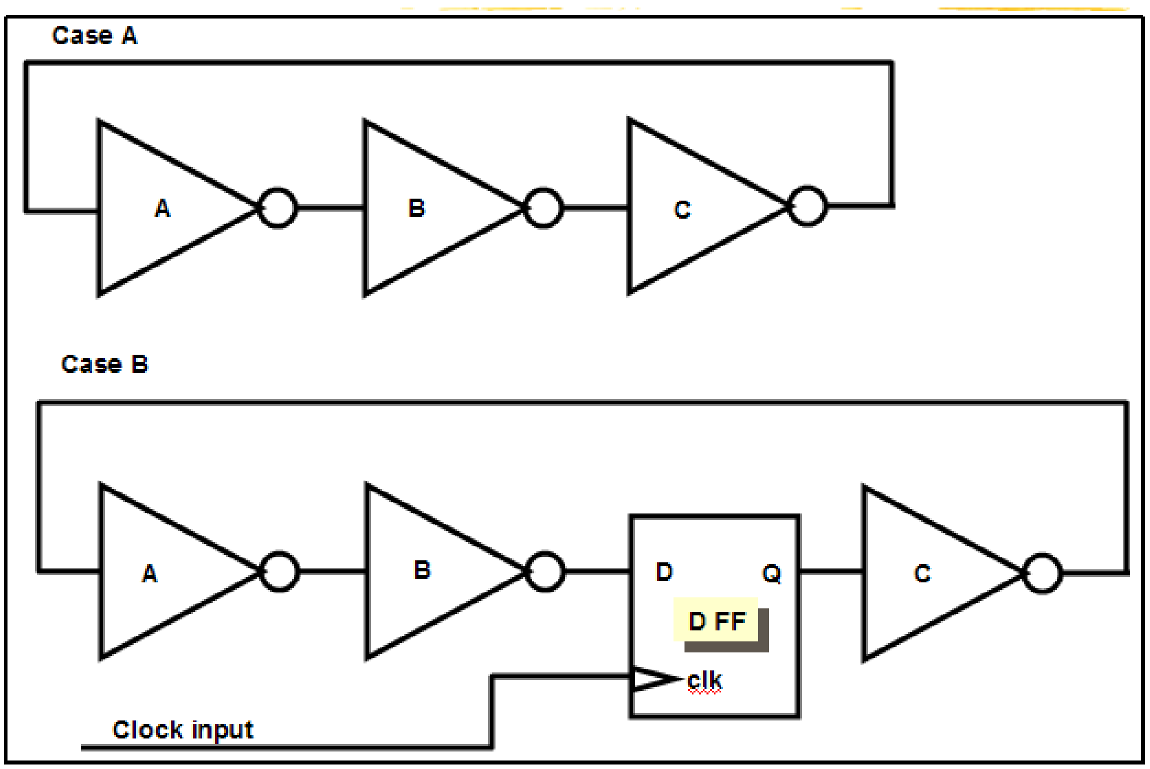


Figure 4.24: Case A: A circuit with a race condition. The signal that the output feeds back to the input is always the opposite polarity to the current input signal, the circuit can never establish a stable condition. Case B: The D-FF stabilizes the circuit and limits transitions to the rising edge of the clock.

In Case A, the circuit consists of three NOT gates (although any odd number of NOT gates greater than 1would also work) connected in a loop. We can analyze the circuit as follows: Suppose that at a particular instant of time the input to gate A goes high. After an appropriate time lag, determined by the propagation delay of gate A (let's assume that this is 10 ns), the output of gate A goes low. Since this is the input to gate B the process repeats itself from gate B to gate C and back to gate A. So, 30 ns after the input to gate A goes high, the input changes state to low, and the process repeats itself again. After 60 ns the input to gate 'A' goes high again and the cycle repeats itself. This is a classical ***race condition*** and generally it's a bad thing to have happen.

Actually, it's not that bad if we want it to work that way. This is one of the ways we can generate clock pulses. We call this circuit an ***oscillator***. With a few simple circuit manipulations we can add a crystal to the circuit and the frequency and periods of the oscillation become very precise. In the example circuit of figure 4.24 the frequency would be the reciprocal of 60 ns, or approximately 16.7 MHz.

The circuit of Case B clearly stops the oscillations. Even though the value at the D input toggles with every rising edge of the clock, the circuit is stable. There are no uncontrolled signals changing state and all transitions in the system are limited to when the rising edge of the clock updates the D-FF.

What does this pleasant little diversion have to do with state machines? Well, everything! If we didn't have the D flip-flops in the circuit, then feeding the outputs of the memory back to the inputs could create a race condition and the state machine would be useless to us. With the D flip-flop in the circuit, we maintain control over the moment in time when we allow the outputs to feed back to the inputs and change the address of the memory cell. This is the state transition that we desire.

The design of this flip-flop isolates the current state (outputs Q0 through Qn) from the next state. Only during the briefest of times when the clock edge rises can the outputs of the flip-flops change state to agree with the D inputs. At that point we have transitioned to the new state and the outputs of the current are fed back to the address inputs of the memory to provide the address for the next state. The data values stored in that address provide the values for the next state transition when the next rising edge of the clock occurs.

Thus, we can summarize this circuit architecture as follows:

- The memory cells hold all the possible states of the system just as the truth table contains all possible logical terms for the system.
- The output of the memory array (truth table) is the input to the D storage register and represents the next state of the state machine after the rising edge of the next clock pulse.
- The output of the D storage register is the current state of the system. The output is used to control external inputs and also provides the address to the memory array for the next state.
- The D storage register isolates the output of the state machine from the inputs to the state machine. State transitions can only occur on the rising edge of the clock signal.

Let’s summarize what we’ve learned in this chapter.

- The action of feedback creates a circuit element that has state dependency. That is, the state of the outputs depends not only on the state of the inputs, but on a clock pulse and the state of the outputs just prior to the arrival of the clock pulse. The simplest form of this circuit, an RS flip-flop, is of limited usefulness.
- Once we add the additional functionality of the master-slave flip-flop, then we can overcome many of the instabilities that are inherent in an RS design.
- The JK flip-flop is the most versatile form of the device.
- A derivative design, the D-type flip-flop has the added property of a data storage element.
- The D-FF can be configured as a counter, a shift-register and a storage register.
- The D-FF is the key element in the implementation of a state machine because it limits the possible transitions of the system to only the instant in time during the rising edge of the clock.

Bibliography and references

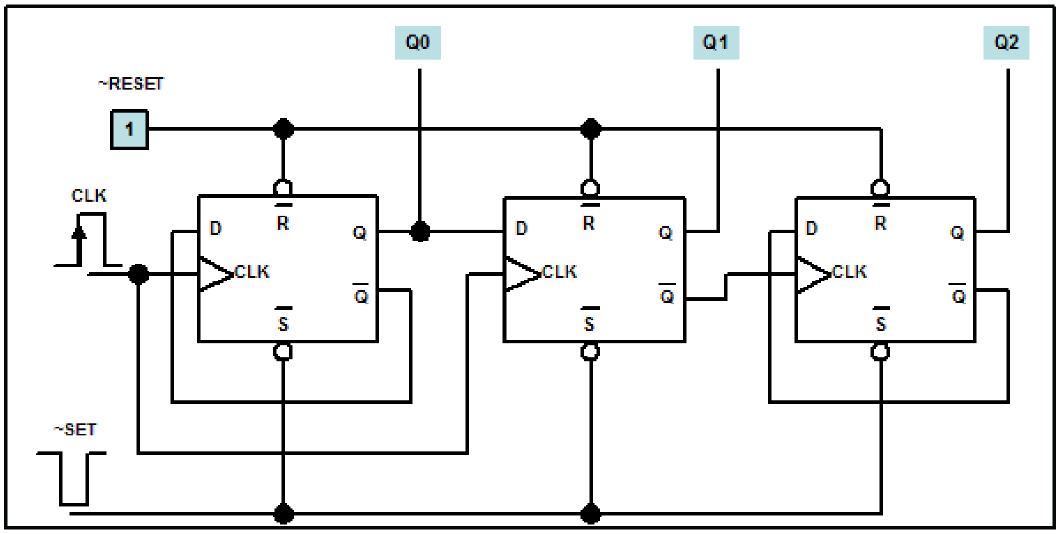
1- Linda Null and Julia Lobur, *the essentials of Computer Organization and Architecture*, ISBN 0-7637-0444-X, Jones and Bartlett Publishers, Sudbury, MA, 2003, Pg. 116

2- The particular division of Hewlett-Packard®, where I was formerly employed for many years, had code names for the hardware and software designers. Hardware Designers were called "Toads" and Software Designers were called "Turkeys". I'm not sure where the naming convention came from; they just were Turkeys and Toads. Engineers who could design hardware and software were sometimes called "Capons", but we won't go there.

Over the years I've made some additional observations about hardware and software types. Hardware Designers tend to wear Nike running shoes with untied laces and Software Designers tend to wear sandals or Birkenstocks. I use to wear running shoes but recently I noticed that I'm wearing my Birkenstocks more and more. Could teaching in a CS Department be remolding my essence? Only time will tell.

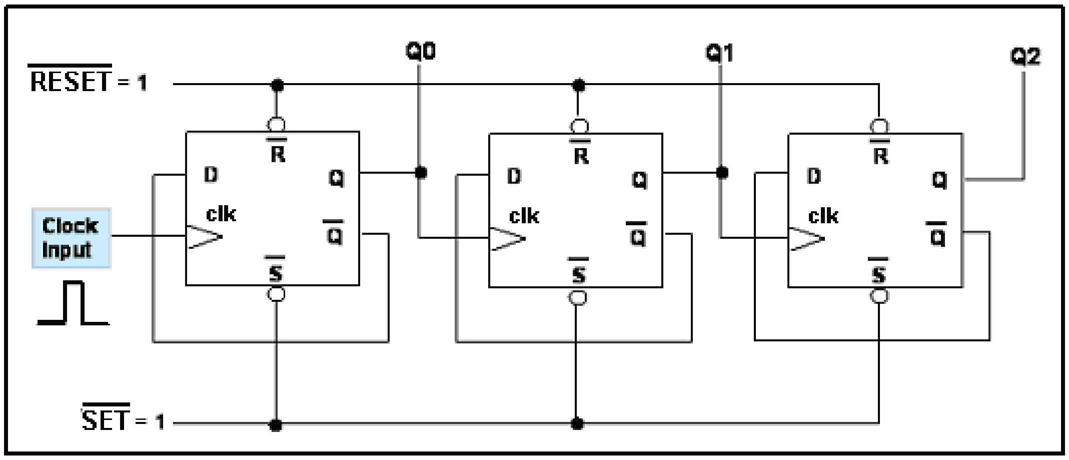
Exercises for chapter 4 (Note problems denoted by an asterisk are especially challenging)

1- The circuit shown below consists of 3 D-type flip flops. The black dots indicate those wires that are physically connected to each other. The ~RESET inputs are permanently tied to logic 1 and are never asserted. Before any clock pulses are received, all the ~SET inputs receive a negative pulse to establish the initial conditions for the circuit.



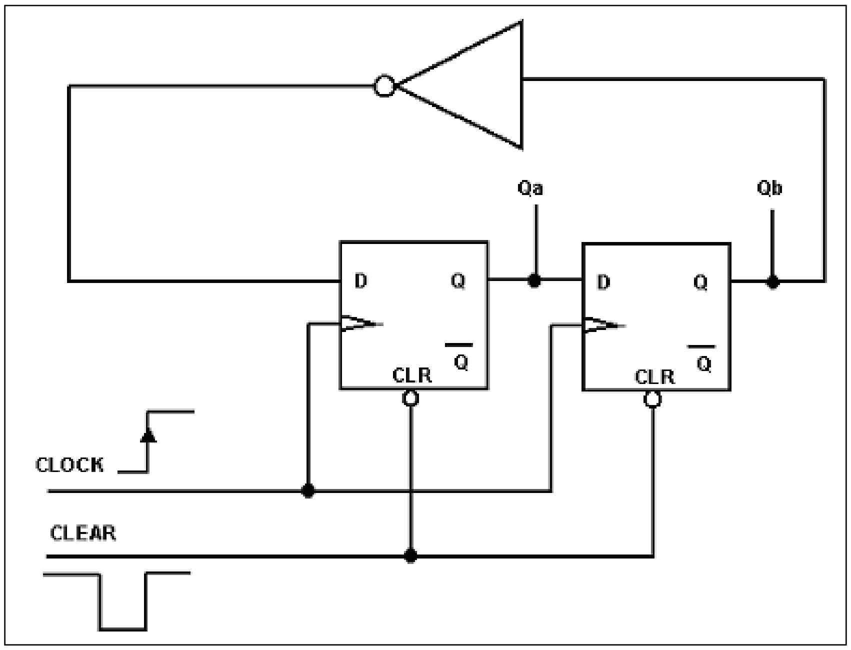
Draw the state transition diagram for this circuit. Please denote the states in the order Q0,Q1,Q2. Be sure to show the starting state, as indicated in the problem specification, and all subsequent states. Use arrows to indicate the transitions from one state to the next.

2- Shown below is a 3-bit counter made up of positive edge-triggered, D- type flip-flops:

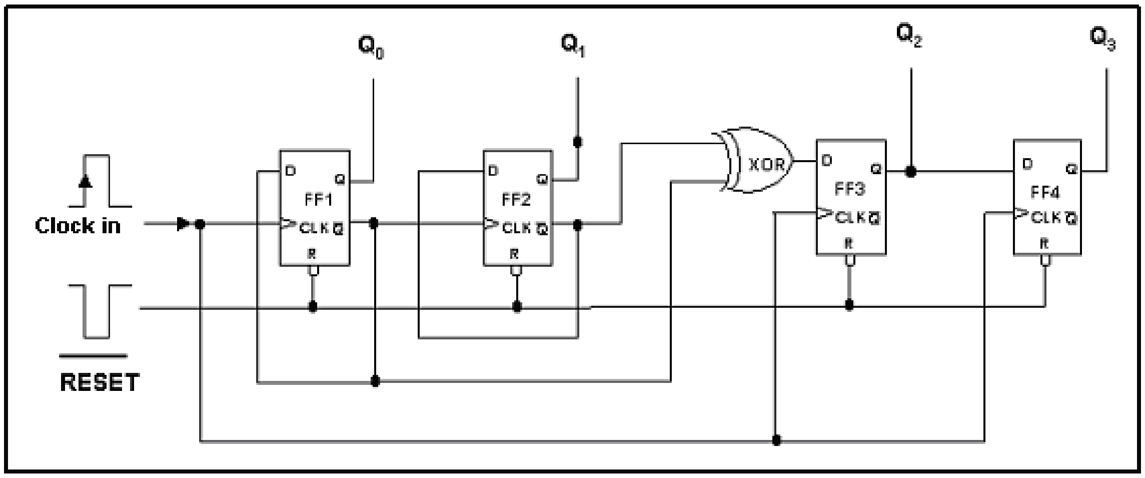


Assume a ~RESET pulse has just been asserted so that the outputs Q0,Q1 and Q2 are in the low state at time T0. Draw the Complete the timing diagram for outputs Q0, Q1 and Q2. For simplicity sake you may assume that there is no delay through a flip-flop, so that the outputs change state coincidentally with the rising edge of the input clock.

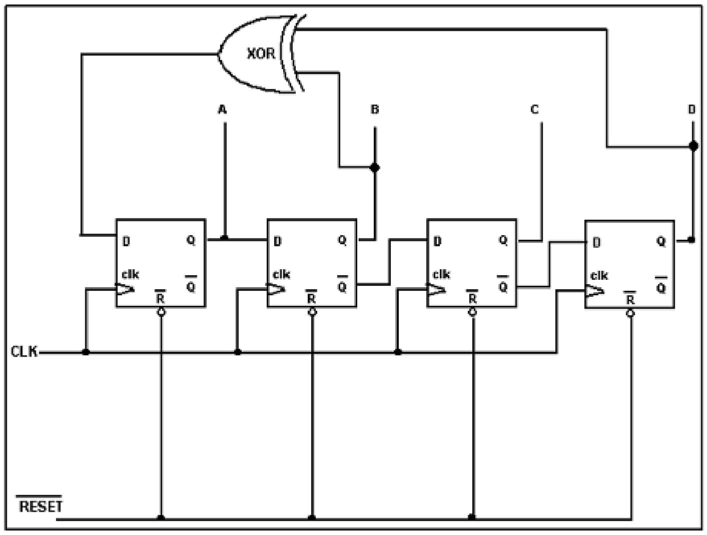
3- Consider the circuit shown below. Assume that a CLEAR pulse has been asserted. Create a truth table for this circuit, showing the state of the outputs after 4 clock pulses.



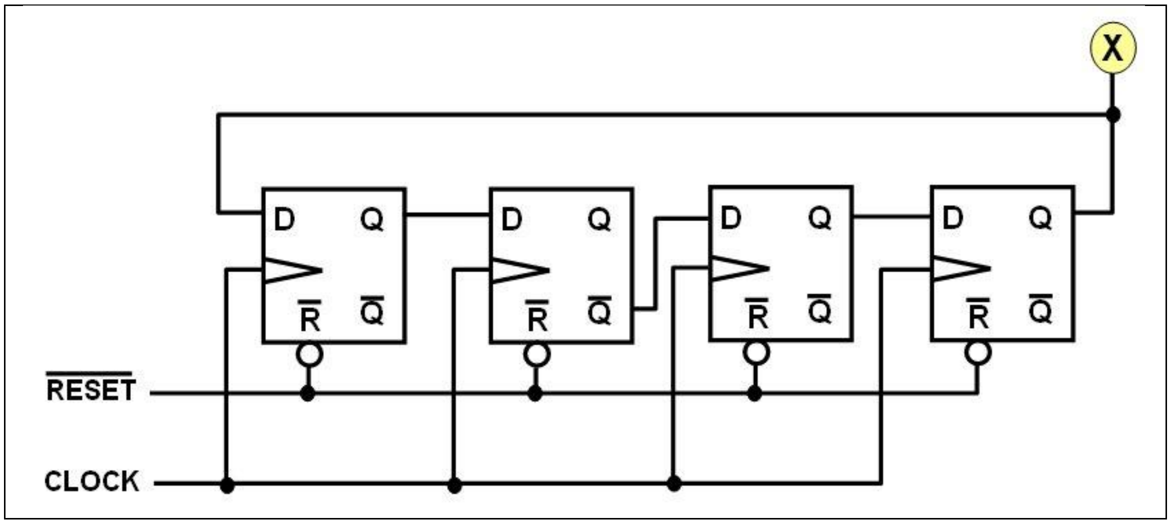
4- The figure below shows four D-type flip-flops and an exclusive OR (XOR) gate. A RESET pulse clears all of the Q outputs to zero. Wires that cross each other in the figure are only connected if there is a heavy black dot at the intersections. Create a truth table showing the state of outputs Q0 through Q3 after 4 clock pulses.



5- Consider the circuit shown below, comprised of 4, D-type flip flops and an exclusive or (XOR) gate. Assume that a RESET pulse is issued to the circuit and the outputs A,B,C,D all go to zero. Create a truth table showing the state of the outputs after 8 clock pulses. Does the pattern repeat itself? If so, when?

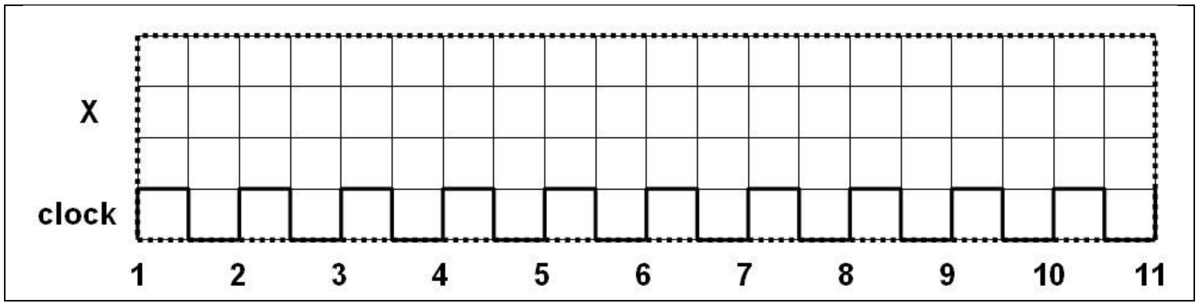


6- The circuit shown below consists of 4 “D” type flip-flops arranged as a type of shift register.



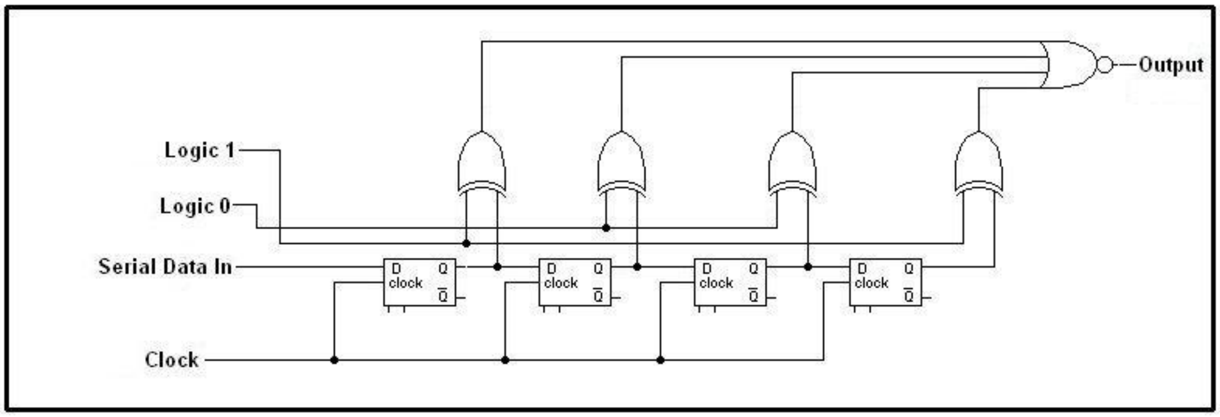
The graph shown below allows you to make a plot of the value of the circuit at point ‘X’ as a function of each clock pulse applied to the circuit. Assume that the ~RESET pulse has been applied to the circuit. Make a copy of the timing diagram shown below:

A-Draw the waveform that you would see at point X
B- Assume that along the horizontal axis each square of the graph represents 25 nanoseconds in time. What are the frequency and the period of the waveform from part A? Express your answer in the appropriate engineering units.



7- Consider a counter/frequency divider made up of 8 stages of "D" flip-flops connected in series. Assume that the propagation delay through an individual flip-flop is 25 ns. In this case, the propagation delay is the time from the rising edge of the clock to the time that the output change has stabilized. What is the maximum possible input clock frequency that we can have before the last stage of the counter is still changing state when the next clock signal arrives. You may assume a square wave signal for the clock input.

8- In a sentence or two, write down the purpose of the circuit shown below. Note that the signal lines labeled Logic 1 and Logic 0 may be considered to be constant values of 1, and 0, respectively. A stream of serial data comes into the circuit on the left side. The data is synchronous with a clock signal.

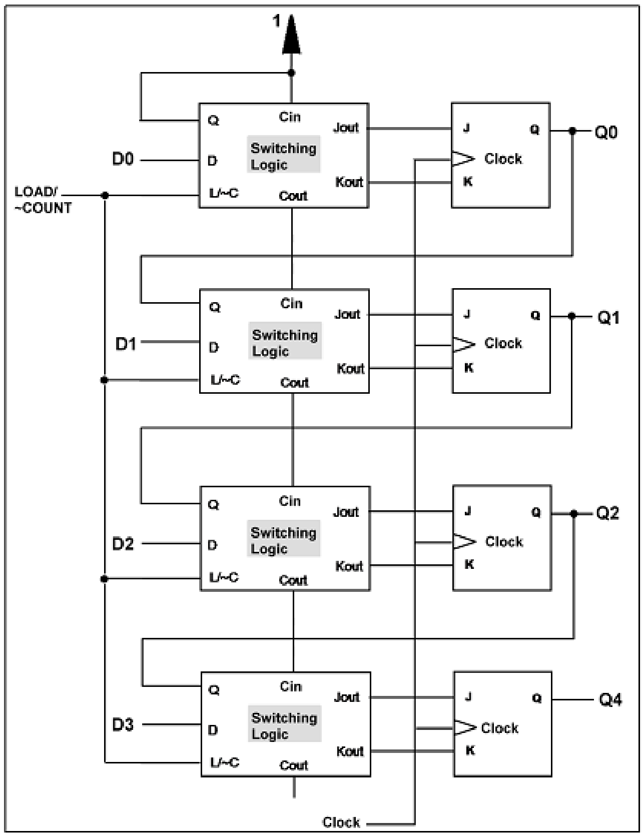


9- One of the most common uses of the JK-FF is to build synchronous counting circuits. A synchronous counter differs from a ripple counter in that the clock goes to all of the FF stages in parallel and each stage either toggles or remains in its current state depending upon the state of the J and K inputs. Thus, the counter is finished counting much quicker than a ripple counter. Design a 4-stage synchronous counter using JK flip-flops, such as the ones shown in figure 4.6. *Hint: The only external gating that you'll need are three, 2-input AND gates.*

10*- The circuit shown below is called a synchronous counter with parallel load. It is particularly useful in computer applications. The LOAD/~COUNT input determines if the circuit functions as a synchronous up-counter or as a parallel load register. For example, holding the LOAD/~COUNT input HIGH and then pulsing the clock input loads the values D0 through D3 into the JK flip-flops. If the LOAD/~COUNT input is then brought LOW, subsequent pulses on the Clock input causes the counter to synchronously count up, commencing from the previously loaded value. Thus, if the value (D0, D1, D2, D3) = (0,1,1,0) is loaded into the circuit with the LOAD/~COUNT input HIGH, then the next Clock pulse after LOAD/~COUNT is brought low will cause the Q0 to Q3 outputs to count up to (1, 1, 1, 0).

The box labeled "Switching Logic" contains the circuitry which implements the basic functionality of the synchronous counter with parallel load. Design the gate circuitry necessary to implement the switching logic for this device.

Hint: Solving the previous problem will help in understanding the requirements of this circuit design. Once that problem #7 is completed it should be relatively straight-forward to construct the truth table and K-Map for the switching logic.


1
D0
D1
D2
D3
LOAD/ ~COUNT
Q
Cin
Jout
Switching Logic
D
Kout
L/~C
Cout
J
Q
Clock
K
Q0
Q1
Q2
Q4
Clock