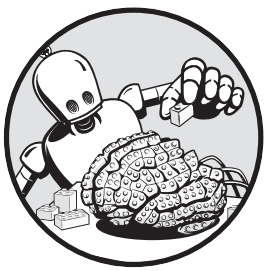
4

STORING STATE WITH THE FLIP-FLOP



Alongside the look-up table, the other main component in an FPGA is the *flip-flop*. Flip-flops give FPGAs the ability to remember, or store, state. In this chapter, we'll explore how flip-flops work and learn why they're important to the functioning of FPGAs.

Flip-flops make up for a shortcoming of look-up tables. LUTs generate output as soon as they're provided input. If all you had to work with was LUTs, your FPGA could perform all the Boolean algebra you might want, but your outputs would be determined solely based on the current inputs. The FPGA would know nothing about its past state. This would be very limiting. Implementing a counter would be impractical, since a counter requires knowledge of a previous value that can be incremented; so would storing the result of some math operation as a variable. Even something as critical as having a concept of time is impractical with just LUTs; you can only calculate values based on the now, not on anything in the past. The

flip-flop enables these interesting capabilities, which is why it's critical to the operation of an FPGA.

How a Flip-Flop Works

A flip-flop stores state in the form of a high or low voltage, corresponding to a binary 1 or 0 or a true/false value. It does this by periodically checking the value on its input, passing that value along to its output, and holding it there. Consider the basic diagram of a *D flip-flop* shown in Figure 4-1. D flip-flops are the most common type of flip-flop in FPGAs, and they're the focus of this chapter. (I'll drop the *D* in front of *flip-flop* going forward.)

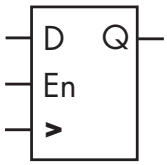


Figure 4-1: A diagram of a D flip-flop

Notice that the component has three inputs on the left and one output on the right. The top-left input, labeled *D*, is the *data input* to the flip-flop. It's where data, in the form of 1s or 0s, comes in. The bottom-left input, labeled with what looks like a greater-than (>) sign, is the *clock input*, which synchronizes the performance of the flip-flop. At regular intervals, the clock input triggers the flip-flop to take the value from the data input and pass it to the output (labeled *Q* in the diagram).

The middle-left input, labeled *En*, is the *clock enable*. As long as the clock enable is high, the clock input will continue to trigger the flip-flop to update its output. If the clock enable input goes low, however, the flip-flop will ignore its clock and data inputs, essentially freezing its current output value.

To better understand how a flip-flop operates, we need to look more closely at the signal coming in to the clock input.

FLIP-FLOP COMPONENT TERMINOLOGY

The component presented in Figure 4-1, a flip-flop with a clock enable pin, isn't always called a flip-flop. FPGA manufacturers such as AMD and Intel do in fact use that terminology in their reference information, but a more technically accurate name is a *clocked D latch*. It's not valuable getting into the details about why one name is better than another; instead, for the purposes of this book, we'll use the real-world terminology that the FPGA manufacturers use and refer to these components as flip-flops.

The Clock Signal

A *clock signal*, often just called a *clock*, is a digital signal that steadily alternates between high and low, as shown in Figure 4-2. This signal is usually provided via a dedicated electronic component external to the FPGA. A clock is key to how FPGAs operate: it triggers other components, such as flip-flops, to perform their tasks. If you think of an FPGA as a set of gears, the clock is like the big gear that turns all the other gears. If the main gear isn't spinning, the others won't spin either. You could also think of the clock as the heart of the system, since it keeps the beat for the entire FPGA. Every flip-flop in the FPGA will be updated on the pulse of the clock's heartbeat.

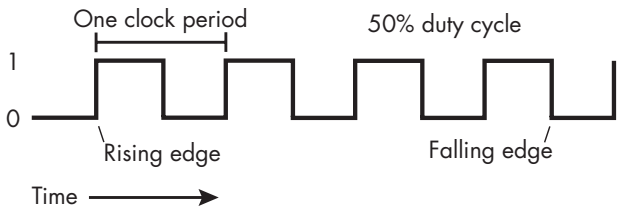


Figure 4-2: A clock signal

Notice the vertical lines in the clock signal diagram, where the signal jumps from low to high or high to low. These abrupt changes in the signal are called *edges*. When the clock goes from low to high, it's called a *rising edge*, and when it goes from high to low, it's called a *falling edge*. Flip-flops are conventionally triggered on each rising edge of the clock: whenever the clock signal changes from low to high, the flip-flop updates its output to match its data input.

NOTE *It's possible to trigger a flip-flop with the falling edges of a clock, but this is* much *less common than using the rising edge.*

Every clock has a *duty cycle*, the fraction of time that the signal is high. For example, a signal with a 25 percent duty cycle is high one-quarter of the time and low three-quarters of the time. Almost all clocks, including the one shown in Figure 4-2, have a 50 percent duty cycle: they're half-on, half-off.

A clock also has a *frequency*, which is the number of repetitions from low to high and back again (called a cycle) in a second. Frequency is measured in hertz (Hz), or cycles per second. You may be familiar with your computer's CPU frequency, which can be measured in gigahertz (GHz), where 1 GHz is 1 billion Hz. FPGAs don't often run quite that quickly. More commonly, FPGA clock signals run in the tens to hundreds of megahertz (MHz), where 1 MHz is 1 million Hz. As an example, the clock on the Go Board (discussed in Appendix A) runs at 25 MHz, or 25 million cycles per second.

Another way to describe a clock's speed is to refer to its *period*, the duration of a single clock cycle. You can calculate the period by finding

1 ÷ *frequency*. In the case of the Go Board, for instance, the clock period is 40 nanoseconds (ns).

A Flip-Flop in Action

A flip-flop operates on the transitions of its clock input. As mentioned previously, when a flip-flop sees a rising edge of the clock, it checks the state of the data input signal and replicates it at the output—assuming the clock enable pin is set to high. This process is called *registering*, as in, "the flip-flop *registers* the input data." Thanks to this terminology, a group of flip-flops is known as a *register*, and by extension, a single flip-flop can also be called a *one-bit register*. One flip-flop by itself is able to register a single bit of data.

To see how registering works in practice, we'll examine a few example inputs to a flip-flop and their corresponding outputs. First, consider Figure 4-3.

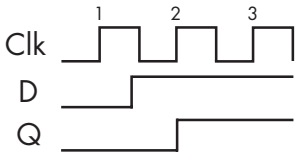


Figure 4-3: An example of flip-flop behavior

This figure shows three waveforms: the top one (Clk) represents an FPGA's clock signal, the middle one (D) is the data input of a flip-flop, and the bottom one (Q) is the flip-flop's output. Let's assume the clock enable is high, so the flip-flop is always enabled. We can see the waveforms across three cycles of the clock; the rising edge of each clock cycle is indicated with the numbers 1, 2, and 3. In between the first and second rising edges of the clock, the D input goes from low to high, but notice that the output doesn't immediately go high when the input does. Instead, it takes a bit of time for the flip-flop to register the change in the input. Specifically, it takes until the *next rising clock edge* for the flip-flop output to follow the input.

The flip-flop looks at the input data and makes the output match the input only at the rising edge of the clock, never between edges. In this case, at the rising edge of the second clock cycle, the output Q sees that D has gone from low to high. At this point, Q takes on the same value as D. On the third rising edge, Q again checks the value of D and registers it. Since D hasn't changed, Q stays high. Q also registered D at the rising edge of the first clock cycle, but since both D and Q were low at that point, Q didn't change.

Now consider Figure 4-4, which shows how a flip-flop responds to another example scenario.

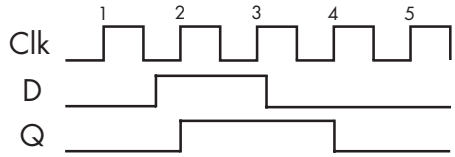


Figure 4-4: Another example of flip-flop behavior

Here we again see a flip-flop operating over several cycles of the clock. Again, let's assume the flip-flop is always enabled. Between the clock's first and second rising edges, input D goes from low to high. On the second rising edge, Q sees that D has gone high, so it toggles from low to high as well. On the third rising edge, Q sees D has stayed high, so it stays high, too. Between the third and fourth rising edges, D goes low, and the output similarly goes low on the fourth rising edge. On the last rising edge, D is still low, so Q stays low as well.

The previous examples have all assumed the clock enable input is high. Let's now show what happens when the flip-flop's clock enable isn't always high. Figure 4-5 shows the exact same Clk and D waveforms as Figure 4-4, but instead of the clock enable remaining high the whole time, it's only high at the third rising edge.

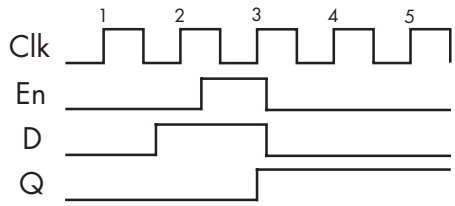


Figure 4-5: Flip-flop behavior with the clock enable signal

With the clock enable (En) now in play, a completely different output Q is generated. Q no longer "sees" that D has gone high on clock cycle two, since the clock enable is low at that point. Instead, Q only changes its output from low to high on clock cycle three, when the clock enable is high. On clock cycle four, D has gone low, but Q doesn't follow D. Instead, it stays high. This is because the clock enable has gone low at that point, locking the output in place. The flip-flop will no longer register any changes on D to Q.

These examples demonstrate flip-flop behavior, showing how a flip-flop's activity is coordinated by a clock. Additionally, we've seen how turning off the clock enable pin allows flip-flops to retain state, even when the input D is changing. This gives flip-flops the ability to store data for a long time.

A Chain of Flip-Flops

Flip-flops are commonly chained together, with the output from one flip-flop going directly into the data input of another flip-flop. For example, Figure 4-6 shows a chain of four flip-flops. For simplicity, let's assume these are always enabled.

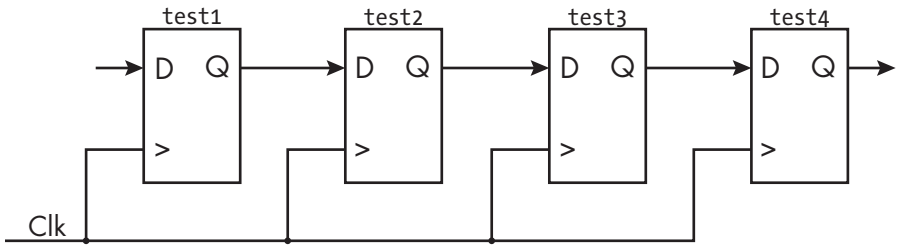


Figure 4-6: A chain of four flip-flops

The four flip-flops, labeled `test1` through `test4`, are chained such that the output of `test1` goes to the input of `test2`, the output of `test2` goes to the input of `test3`, and so on. All four flip-flops are driven by the same clock. The clock synchronizes their operation: with each rising edge of the clock, all four flip-flops will check the value on their input and register that value to their output.

Suppose the `test1` flip-flop registers a change at its input. Figure 4-7 illustrates how that change will propagate through the flip-flop chain, all the way to the output of `test4`.

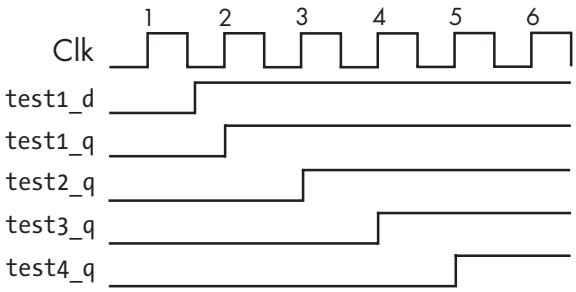


Figure 4-7: A change of input propagating through the flip-flop chain

The figure shows waveforms for the clock signal, the input and output of the `test1` flip-flop (`test1_d` and `test1_q`, respectively), and the output of each subsequent flip-flop. On the first clock cycle rising edge (labeled 1), `test1_d` is low, so `test1_q` stays low as well. It's not until the second rising clock edge that the first flip-flop "sees" that the input has changed to high and registers that to its output. The `test1` flip-flop's output is also the input to the `test2` flip-flop, but notice that the output of `test2` doesn't immediately change to high when the output of `test1` does. Instead, `test2_q` changes one clock cycle later, on the third rising clock edge. Then, on the fourth rising edge, we see `test3_q` go high, and finally on the fifth rising edge `test4_q` goes high and stays high.

By adding three flip-flops behind `test1`, we've delayed the output by three clock cycles as the signal propagates through the chain. Each flip-flop in the chain adds a single clock cycle of delay. This technique of delaying signals by adding a chain of flip-flops is a useful design practice when working with FPGAs. Among other things, designers may chain flip-flops to

create circuits that can delay or remember data for some amount of time, or to convert serial data to parallel data (or vice versa).

OTHER KINDS OF FLIP-FLOPS

In this chapter we're focusing on the D flip-flop. If you've taken a digital electronics course in college, there's a good chance your professor spent time talking about other types of flip-flops as well, including the *T flip-flop* and the *JK flip-flop*. In practice, however, you're unlikely to need to know anything about these other types of flip-flops to use an FPGA, as most FPGAs are made with D flip-flops. For this reason, I won't burden you with information about how the other kinds of flip-flops work, although it's important to acknowledge that they exist.

Project #3: Blinking an LED

Now that you know how flip-flops work, we'll make use of a couple of them in a project where the FPGA must remember information about its own state. Specifically, we're going to toggle the state of an LED each time a switch is released. If the LED was off before the switch is released, it should turn on, and if the LED was on, it should turn off.

This project uses two flip-flops. The first is for remembering the state of the LED: whether it's on or off. Without this memory, the FPGA would have no way of knowing whether to toggle the LED each time the switch is released; it won't know if the LED is on and needs to be turned off, or off and needs to be turned on.

The second flip-flop allows the FPGA to detect when the switch is released. Specifically, we're looking for the falling edge of the switch's electrical signal: its transition from high to low. A good way to look for a falling edge in an FPGA is to register the signal in question by passing it through a flip-flop. When the input value of the flip-flop (that is, the unregistered value) is equal to 0 but the previous output value (the registered value) is equal to 1, then we know that a falling edge has occurred. The falling edge of the switch is not to be confused with the rising edge of the clock; we're still using the rising edge of the clock to drive all of our flip-flops. Figure 4-8 shows the pattern to look for.

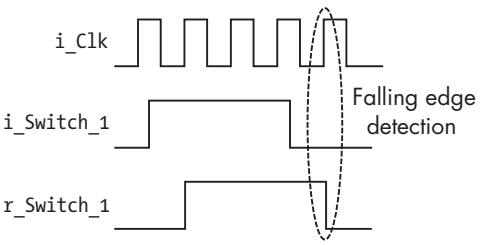


Figure 4-8: Falling edge detection using flip-flop

Here, `i_Clk` is the clock signal; `i_Switch_1` represents the electrical signal from the switch, which passes into a flip-flop; and `r_Switch_1` is the flip-flop's output. At the circled rising clock edge, we can see that `i_Switch_1` is low, but `r_Switch_1` is high. This pattern is how we can detect the falling edge of a signal. One thing to note is that while `r_Switch_1` does go low on the rising clock edge, when the logic evaluates the state of `r_Switch_1` at that same rising clock edge, it will still "see" that `r_Switch_1` is high. Only after some small delay will the output of `r_Switch_1` go low, following the state of `i_Switch_1`.

This project will also require some logic between the two flip-flops, which will be implemented in the form of a LUT. This will be your first glimpse of how flip-flops and LUTs work together in an FPGA to accomplish tasks. Figure 4-9 shows an overall block diagram for this project.

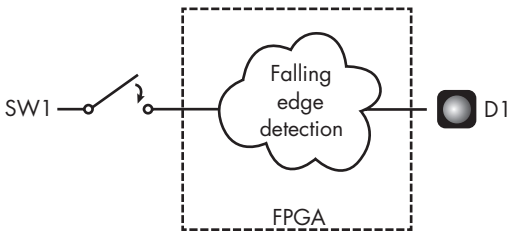


Figure 4-9: The Project #3 block diagram

The output of one of the switches on your development board (SW1) goes into the FPGA, where the falling edge detection logic is implemented. The output of this logic drives one of the board's LEDs (D1). Now we'll look at how to implement this design.

Writing the Code

We can write our LED-toggling code using Verilog or VHDL:

Verilog

```
module LED_Toggle_Project(
  input  i_Clk,
  input  i_Switch_1,
  output o_LED_1);

❶ reg r_LED_1    = 1'b0;
  reg r_Switch_1 = 1'b0;

❷ always @(posedge i_Clk)
  begin
  ❸ r_Switch_1 <= i_Switch_1;

  ❹ if (i_Switch_1 == 1'b0 && r_Switch_1 == 1'b1)
    begin
    ❺ r_LED_1 <= ~r_LED_1;
```

```
    end
  end

  assign o_LED_1 = r_LED_1;

endmodule
```

VHDL

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

entity LED_Toggle_Project is
  port (
    i_Clk      : in  std_logic;
    i_Switch_1 : in  std_logic;
    o_LED_1    : out std_logic
  );
end entity LED_Toggle_Project;

architecture RTL of LED_Toggle_Project is

❶ signal r_LED_1    : std_logic := '0';
  signal r_Switch_1 : std_logic := '0';

begin

❷ process (i_Clk) is
  begin
    if rising_edge(i_Clk) then
    ❸ r_Switch_1 <= i_Switch_1;
    ❹ if i_Switch_1 = '0' and r_Switch_1 = '1' then
      ❺ r_LED_1 <= not r_LED_1;
      end if;
    end if;
  end process;

  o_LED_1 <= r_LED_1;

end architecture RTL;
```

We begin by defining two inputs (the clock and the switch) and a single output (the LED). Then we create two signals ❶: `r_LED_1` and `r_Switch_1`. We do this using the `reg` keyword (short for *register*) in Verilog, or the `signal` keyword in VHDL. Ultimately these signals will be implemented as flip-flops, or registers, so we prefix their names with the letter `r`. It's good practice to label any signals that you know will become registers `r_`*`signal_name`*, as it helps keep your code organized and easy to search.

Next, we initiate what's known as an `always` block in Verilog or a `process` block in VHDL ❷. This type of code block is triggered by changes in one or more signals, as specified by the code block's *sensitivity list*, which is given in parentheses when the block is declared. In this case, the block is sensitive to the clock signal, `i_Clk`. Specifically, this block will be triggered any time the

clock changes from a 0 to a 1; that is, at each rising clock edge. Remember, when you use a clock to trigger logic within your FPGA, you'll almost always be using the clock's rising edges. In Verilog, we indicate this with the keyword `posedge` (short for *positive edge*, another term for *rising edge*) within the sensitivity list itself: `always @(posedge i_Clk)`. In VHDL, however, we only put the signal name in the sensitivity list, and specify to watch for rising edges two lines later, with `if rising_edge(i_Clk) then`.

Within the `always` or `process` block, we create the first flip-flop of this project by taking the input signal `i_Switch_1` and registering it into `r_Switch_1` ❸. This line of code will generate a flip-flop with `i_Switch_1` on the D input, `r_Switch_1` on the Q output, and `i_Clk` going into the clock input. The output of this flip-flop will generate a one-clock-cycle delay of any changes to the input. This effectively gives us access to the previous state of the switch, which we need to know in order to detect the falling edge of the switch's signal.

We next check to see if the switch has been released ❹. To do this, we compare the current state of the switch with its previous state, using the flip-flop we just created ❸. If the current state (`i_Switch_1`) is 0 *and* the previous state (`r_Switch_1`) is 1, then we've detected a falling edge, meaning the switch has been released. The *and* check will be accomplished with a LUT.

At this point, perhaps you've noticed something surprising. First we assigned `i_Switch_1` to `r_Switch_1` ❸, then we checked if `i_Switch_1` is 0 and `r_Switch_1` is 1 ❹. You might think that since we just assigned `i_Switch_1` to `r_Switch_1`, they'd always be equal, and the `if` statement would never be true. Right? Wrong! Assignments in an `always` or `process` block that use `<=` don't occur immediately. Instead, they take place on each rising edge of the clock and therefore *are all executed at the same time*. If at a rising clock edge `i_Switch_1` is 0 and `r_Switch_1` is 1, the `if` statement will evaluate as true, even as `r_Switch_1` is simultaneously switching from a 1 to a 0 to match `i_Switch_1`.

Now we're thinking in parallel instead of serially! We've generated assignments that occur all at once, instead of one at a time. This is completely different from traditional programming languages like C and Python, where assignments occur one after the other. To further drive this point home, you could move the assignment of `r_Switch_1` to the last line of the `always` or `process` block, and everything would still work the same. Formally, we call the `<=` assignment a *non-blocking assignment*, meaning it doesn't prevent ("block") other assignments from taking place at the same time. In Chapter 10, we'll revisit this concept and compare non-blocking assignments with blocking assignments.

Once we're inside the `if` statement, we toggle the state of the LED ❺. Doing so generates the second flip-flop used in this project. We take the current value of `r_LED_1`, invert it, and store the result back into the flip-flop. That might sound impossible, but it's perfectly valid. The output of the flip-flop will pass through a LUT, acting here as a NOT gate, and then be fed back into the flip-flop's input. This way, if the LED was on it'll turn off, and vice versa.

Adding Constraints

Once the code is ready, it's time to run the tools to build the FPGA image and program your board. First, since this project uses a clock, you need to

add a constraint telling the FPGA tool about the clock's period. The clock period tells the timing tool how much time is available to route wires between flip-flops. As clock speed increases, it gets harder for the FPGA to *meet timing*, or achieve all the desired tasks within each clock cycle. For slower clocks, with frequencies on the order of tens of megahertz, you shouldn't have any problems meeting timing. In general, it's only when you deal with clocks that are faster than 100 MHz that you may start to run into timing issues.

The clock period will vary from one development board to another, and can be found in your board's documentation. To tell Lattice iCEcube2 about the clock period, create a new text file with a *.sdc* file extension containing something like the following:

```
create_clock  -period 40.00 -name {i_Clk} [get_ports {i_Clk}]
```

This creates a clock with a 40 ns period (25 MHz frequency) and assigns that constraint to the signal called `i_Clk` in your design. This constraint will work for the Go Board, as an example, but if your board has a different clock period, replace `40.00` with the appropriate value.

Right-click **Constraint Files** under **Synthesis Tool** and select the *.sdc* file to add it to your project in iCEcube2. Remember from Chapter 2 that we previously had a single *.pcf* constraint file telling the tools which signals to map to which pins. Now we have an additional constraint file just for the clock. Both are critical for getting your FPGA to work correctly.

We also need to update the *.pcf* file to include the pin corresponding to the new clock signal. On the Go Board, for example, the clock is connected to pin 15 of the FPGA, so you would need to add the following pin constraint:

```
set_io i_Clk 15
```

Check the schematic for your development board to see which pin has the clock as an input, and replace the `15` as appropriate.

Building and Programming the FPGA

You're now ready to run the build. When you do this, the tools will generate some reports. The synthesis report should look something like this:

```
--snip--
Resource Usage Report for LED_Toggle_Project

Mapping to part: ice40hx1kvq100
Cell usage:
SB_DFF 2 uses
SB_LUT4 1 use

I/O ports: 3
I/O primitives: 3
```

```
  SB_GB_IO 1 use
  SB_IO 2 uses

  I/O Register bits: 0
❶ Register bits not including I/Os: 2 (0%)
  Total load per clock:
   ❷ LED_Toggle_Project|i_Clk: 1
  Mapping Summary:
❸ Total LUTs: 1 (0%)
```

This report tells us that we're using two register bits ❶, meaning our design includes two flip-flops. This is exactly what we expected. The report also shows that we're using one LUT ❸. This single LUT will be able to perform both the AND and NOT operations required in the code. Notice, too, that the tools identified the signal `i_Clk` as a clock ❷.

Now let's look at the place and route reports, which you can view in iCEcube2 by going to **P&R Flow ▸ Output Files ▸ Reports**. There are two reports here. The first is a pin report, which tells you which signals were mapped to which pins. You can use this to confirm that your signals were mapped correctly. The second is the timing report. It has a section labeled "Clock Frequency Summary" that should look something like this:

```
--snip--
 1::Clock Frequency Summary
============================================================
Number of clocks: 1
Clock: i_Clk | Frequency: 654.05 MHz | Target: 25.00 MHz |
--snip--
```

This section tells you if the constraint file was accepted correctly. Here we see that the tools have found our clock, `i_Clk`. The `Target` property indicates the tools have recognized a 25 MHz constraint placed on the clock (your number will vary, depending on your development board), while the `Frequency` property tells us the maximum frequency at which the FPGA could theoretically run our code successfully. In this case, we could run this FPGA at 654.05 MHz and it would still be guaranteed to work correctly. That's quite fast! As long as the `Frequency` property is higher than the `Target` property, you shouldn't have any issues running your code. A problem would show up here in the form of a *timing error*, which happens when the target clock speed is greater than the frequency that the tools can achieve. In Chapter 7, we'll take a deeper look at what causes timing errors and how to fix them.

Now that you've successfully built the FPGA design, you can program your board and test the project. Try pushing the switch several times. You should see the LED toggle on or off each time the switch is released. Congratulations, you've got your first flip-flop working!

However, you may notice something strange going on. The LED may not appear to change its state with each release. You might think that the FPGA isn't registering the releases of the switch, but in fact the LED is

toggling two or more times with each release, so quickly that your eyes don't see it. The cause is related to the physical workings of the switch itself. To solve this issue, the switch needs to be *debounced*. You'll learn what this means and how to do it in the next chapter.

Combinational Logic vs. Sequential Logic

There are two kinds of logic that can take place inside an FPGA: combinational logic and sequential logic. *Combinational logic* is logic for which the outputs are determined by the present inputs, with no memory of the previous state. This kind of logic is achieved with LUTs, which you'll recall generate their output based only on their current inputs. *Sequential logic*, on the other hand, is logic for which the outputs are determined both by present inputs and previous outputs. Sequential logic is achieved with flip-flops, since flip-flops don't immediately register changes on their inputs to their outputs, but rather wait until the rising edge of the clock to act on the new input data.

NOTE *You may also see combinational logic and sequential logic referred to as* combinatorial logic *and* synchronous logic*, respectively.*

It might not be obvious that a flip-flop's output depends on its previous output, so let's explore an example to make this more concrete. Suppose the flip-flop is enabled, its input is low, its clock is low, and the output is low. Then suddenly the input goes high, then back low again quickly. What will the output do? Nothing! It stays low, since there was no clock edge to trigger a change. Now, what happens if that same flip-flop has the same initial conditions, except the output is high? In this case, of course, the output will stay high. But if we only looked at the inputs (D, En, and Clk), *we would be unable to predict the output state*. You need to know what the output of the flip-flop was (its previous state) to determine the flip-flop's current state. That's why a flip-flop is sequential.

Knowing if your code is going to instantiate LUTs (combinational logic) or flip-flops (sequential logic) is critical to being a good FPGA designer, but sometimes it can be hard to tell the difference. In particular, an `always` block (in Verilog) or `process` block (in VHDL) can define a block of either combinational logic or sequential logic. We'll consider examples of each to see how they differ.

First, here's an example of a combinational implementation in Verilog and VHDL:

Verilog

```
always @ (input_1 or input_2)
  begin
    and_gate <= input_1 & input_2;
  end
```

VHDL

```
process (input_1, input_2)
begin
  and_gate <= input_1 and input_2;
end process;
```

Here we've created an `always` or `process` block with a sensitivity list (the signals in the parentheses) that includes two signals: `input_1` and `input_2`. The code block performs an AND operation on the two signals.

This block of Verilog or VHDL code will only generate LUTs; it won't generate any flip-flops. For our purposes, flip-flops require a clock input, and there is no clock. Since no flip-flops are generated, this is combinational logic.

Now consider a slight modification to the examples just shown:

Verilog

```
always @ (posedge i_Clk)
  begin
    and_gate <= input_1 & input_2;
  end
```

VHDL

```
process (i_Clk)
begin
  if rising_edge(i_Clk) then
    and_gate <= input_1 and input_2;
  end if;
end process;
```

This code looks very similar to the previous examples, except now the `always` or `process` block's sensitivity list has changed to be sensitive to the signal `i_Clk`. Since the block is sensitive to a clock, it's now considered sequential logic. This block will actually still require a LUT to perform the AND operation, but in addition to that the output will utilize a flip-flop, since the clock is gating the output from updating all the time.

While all the examples in this section are valid code, I'm going to make a suggestion, especially for FPGA beginners: when writing your code, only create *sequential* `always` blocks (in Verilog) or `process` blocks (in VHDL). The way to do this is to ensure that the block's sensitivity list only has a clock in it. (A clock and a reset is OK too, as we'll discuss later in the chapter.) Combinational `always` blocks and `process` blocks can get you into trouble: you can generate a latch by accident. We'll explore latches in the next section, but basically, they're bad. Additionally, I find code is more readable if you know that every time you come across an `always` block or `process` block, it will always be generating sequential logic.

As for combinational-only logic, write it outside of an `always` block or `process` block. In Verilog, the keyword `assign` is useful. In VHDL, you can simply use the `<=` assignment to create combinational logic.

The Dangers of Latches

A *latch* is a digital component that can store state without the use of a clock. In this way, latches perform a similar function as flip-flops (namely, storing state), but the method they use is different since there's no clock involved. Latches are dangerous and can be inadvertently generated when working with combinational code. In my career, I've never once generated a latch *on purpose*, only by accident. It's highly unlikely that you'd ever actually want to generate a latch either, so it's important to understand how to avoid them.

You always want your FPGA designs to be predictable. Latches are dangerous because they violate this principle. FPGA tools have a very difficult time understanding the timing relationship of a latch and how other components connected to it will perform. If you do manage to create a latch with your code, the FPGA tools will scream at you with warnings about the fact that you've done a horrible thing. Please don't ignore these warnings.

So how can this happen? A latch is created when you write a combinational `process` block or conditional assignment (in VHDL) or a combinational `always` block (in Verilog) with an *incomplete assignment*, meaning the output isn't assigned under all possible input conditions. This is bad and should be avoided. Table 4-1 shows an example of a truth table that would generate a latch.

Table 4-1: A Truth Table That Creates a Latch

Input A	Input B	Output Q
0	0	0
0	1	1
1	0	1
1	1	Undefined

This truth table has two inputs and one output. The output is 0 when both inputs are 0, and it's 1 when input A is 0 and input B is 1, or when input A is 1 and input B is 0. But what happens when both inputs are 1? We haven't explicitly stated what will occur. In this case, the FPGA tools assume that the output should retain its previous state, much like a flip-flop is capable of doing, but without the use of a clock. For example, if the output is 0 and both inputs go high, the output will stay 0. If the output is 1 and both inputs go high, the output will stay 1. This is the behavior that a latch creates: the ability to store state without a clock.

Let's take a look at how this truth table could be created in Verilog and VHDL. Don't write code like this!

Verilog

```
❶ always @ (i_A or i_B)
begin
  if (i_A == 1'b0 && i_B == 1'b0)
    o_Q <= 1'b0;
```

```
      else if (i_A == 1'b0 && i_B == 1'b1)
        o_Q <= 1'b1;
      else if (i_A == 1'b1 && i_B == 1'b0)
        o_Q <= 1'b1;
  ❷ // Missing one last ELSE statement!
end
```

VHDL

```
❶ process (i_A, i_B)
begin
  if i_A = '0' and i_B = '0' then
    o_Q <= '0';
  elsif i_A = '0' and i_B = '1' then
    o_Q <= '1';
  elsif i_A = '1' and i_B = '0' then
    o_Q <= '1';
❷ -- Missing one last ELSE statement!
  end if;
end process;
```

Here, our `always` or `process` block is combinational because there's no clock in the sensitivity list ❶ or the block itself, just two inputs, `i_A` and `i_B`. We mimic the incomplete truth table assignment of the output `o_Q` using conditional checks. Notice that we don't explicitly check the condition where `i_A` and `i_B` are both 1. Big mistake!

If you were to try to synthesize this faulty code, the FPGA tools would generate a latch and warn you about it in the synthesis report. The warning would look something like this:

```
@W: CL118 :"C:\Test.v":8:4:8:5|Latch generated from always block for signal
o_Q; possible missing assignment in an if or case statement.
```

The tools are pretty good. They tell you that there's a latch, they tell you which signal it is (`o_Q`), and they tell you why it might be occurring.

To avoid generating a latch, we could add an `else` statement ❷, which will cover all remaining possibilities. As long as the output is defined for all possible inputs, we'll be safe. An even better solution, however, would be not to use a combinational `always` or `process` block at all. I discourage the use of combinational `always` or `process` blocks precisely because it's easy to make this mistake of omitting an `else` statement. Instead, we can use a sequential `always` or `process` block. Here's what that looks like:

Verilog

```
❶ always @ (posedge i_Clk)
begin
  if (i_A == 1'b0 && i_B == 1'b0)
    o_Q <= 1'b0;
  else if (i_A == 1'b0 && i_B == 1'b1)
    o_Q <= 1'b1;
  else if (i_A == 1'b1 && i_B == 1'b0)
    o_Q <= 1'b1;
end
```

VHDL
```
❶ process (i_Clk)
begin
  if rising_edge(i_Clk) then
    if i_A = '0' and i_B = '0' then
      o_Q <= '0';
    elsif i_A = '0' and i_B = '1' then
      o_Q <= '1';
    elsif i_A = '1' and i_B = '0' then
      o_Q <= '1';
    end if;
  end if;
end process;
```

We now have a sequential `always` or `process` block, because we're using a clock in the sensitivity list ❶ and within the block itself. As a result, `o_Q` will create a flip-flop rather than a latch. Flip-flops don't have the same unpredictable timing issues that latches do. Remember that the flip-flop can utilize its `en` input to retain a value. The flip-flop's `en` input will be disabled when `i_A` and `i_B` are both high. This will retain the flip-flop's output with whatever state it had previously, performing the same behavior as the latch, but in a safe, predictable way.

One side effect of switching to a sequential `always` or `process` block is that it now takes a single clock cycle for the output to be updated. If it's critical that this logic be combinational—with the output updating as soon as one of the inputs changes, with no clock delay—then you need to ensure that the output is specified for all possible input conditions.

There's one other way to generate latches in VHDL. VHDL has the keyword `when`, which can be used in a conditional assignment. Verilog has no equivalent syntax, so this code snippet is for VHDL only:

```
o_Q <= '0' when (i_A = '0' and i_B = '0') else
       '1' when (i_A = '0' and i_B = '1') else
       '1' when (i_A = '1' and i_B = '0');
```

This code exists outside of a `process` block, and again we haven't explicitly stated what `o_Q` should be assigned to when `i_A` and `i_B` are both 1, so the FPGA tools will infer a latch here. The latch will enable the output to keep its previous state, but that's likely not what we intended. Instead, we should be specific with our code and ensure that we have an `else` condition that sets `o_Q` for all possible inputs.

Resetting a Flip-Flop

Flip-flops have an additional input that we haven't discussed yet, called *set/reset*, or often just *reset*. This pin resets the flip-flop back to an initial state, which could be 0 or 1. Resetting flip-flops is useful when the FPGA first powers up and initializes. For example, you might want to reset your flip-flops that control a state machine to the initial state (we'll discuss state machines in Chapter 8). You might also want to reset a counter to some

initial value, or reset a filter back to zero. Resetting flip-flops is one method to ensure your flip-flops are in a specific state prior to operation.

There are two types of resets: synchronous and asynchronous. *Synchronous resets* occur at the same time as the clock edge, whereas *asynchronous resets* can occur at any time. You might trigger an asynchronous reset with a button press external to the FPGA, for example, since the button press can come at any point in time. Let's look at how to code a reset, starting with a synchronous one:

Verilog

```
❶ always @ (posedge i_Clk)
begin
❷ if (i_Reset)
    o_Q <= 1'b1;
❸ else
--snip--
```

VHDL

```
❶ process (i_Clk)
begin
  if rising_edge(i_Clk) then
  ❷ if i_Reset = '1' then
      o_Q <= '1';
  ❸ else
--snip--
```

Here we have an `always` or `process` block with a normal sensitivity list; it's only sensitive to changes of the clock ❶. Inside the block, we first check the state of `i_Reset` ❷. If it's high, then we reset the signal `o_Q` to 1. This is our synchronous reset, since it's happening on the edge of the clock. If `i_Reset` is low, we proceed with the `else` branch of the block ❸, where we'd write whatever code we want to be executed under normal operating (non-reset) conditions.

Notice that in this example we're checking if the reset is high. Sometimes resets can be active low, however, which is usually indicated by `_L` or `_n` at the end of the signal name. If this were an active low reset, we would check for the signal being 0 rather than 1.

Now let's take a look at an asynchronous reset:

Verilog

```
❶ always @ (posedge i_Clk or i_Reset)
begin
❷ if (i_Reset)
    o_Q <= 1'b1;
❸ else
--snip--
```

VHDL

```
❶ process (i_Clk, i_Reset)
begin
❷ if (i_Reset = '1') then
    o_Q <= '1';
```

```
❸ elsif rising_edge(i_Clk) then
--snip--
```

Notice that we've added `i_Reset` into the `always` or `process` block's sensitivity list ❶. Now, rather than checking the clock state first, we check the reset state first ❷. If it's high, then we perform whatever reset conditions we want, in this case setting `o_Q` to 1. Otherwise, we proceed normally ❸.

The choice between synchronous and asynchronous resets should be documented in the user guide for your specific FPGA—some FPGAs are optimized to handle one or the other. Additionally, resets can create strange bugs if they're not treated properly. Therefore, I strongly recommend consulting the documentation to make sure you're resetting flip-flops correctly for your device.

Look-Up Tables and Flip-Flops on a Real FPGA

Now you understand that LUTs and flip-flops exist on FPGAs, but they may still seem a bit abstract. To get a more concrete picture, let's look at how LUTs and flip-flops are actually wired together in a real FPGA. The image in Figure 4-10 is taken from the datasheet for the Lattice iCE40 LP/HX family of FPGAs, the type of FPGA compatible with iCEcube2.

Datasheets are used throughout the electronics industry to explain the details of how a component works. Each FPGA will have at least a few unique datasheets with different pieces of information, and more complicated FPGAs can have dozens of them.

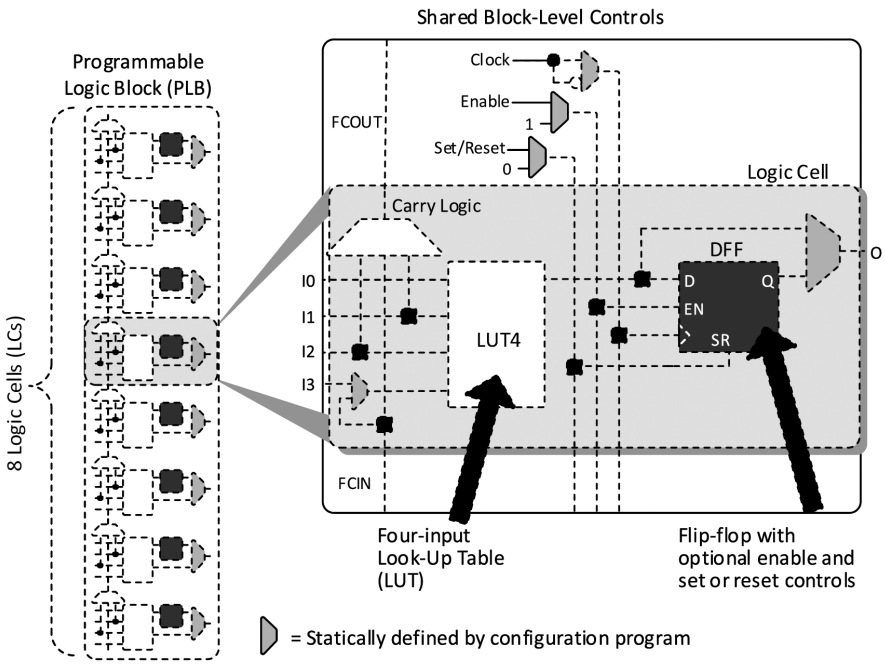


Figure 4-10: LUTs and flip-flops in a real FPGA

Every FPGA, whether from Lattice, AMD, Intel, or whoever else, will have an image very similar to Figure 4-10 in its specific family datasheet. This particular image shows the basic building block of Lattice iCE40 FPGAs, which Lattice calls the *Programmable Logic Block (PLB)*. Each FPGA company has its own unique name for these basic building blocks; for example, AMD calls them *Configurable Logic Blocks (CLBs)*, while Intel uses *Adaptive Logic Modules (ALMs)*. We'll look at the details of how the PLB from Lattice works as an example.

Looking at the left side of the image, we see there are eight logic cells in each PLB. The right side shows a zoomed-in version of a single logic cell. Inside it, notice that there's a rectangle labeled LUT4. This is a four-input look-up table! There's also a dark gray box labeled DFF. This is a D flip-flop! The LUT and the flip-flop truly are the two most critical components inside an FPGA.

This diagram is telling us that at the most fundamental level there's one LUT and one flip-flop inside each logic cell, and there are eight logic cells in a PLB. The PLB is copy-pasted hundreds or thousands of times inside the FPGA to provide enough LUTs and flip-flops to do all the required work.

On the left side of the DFF component (the flip-flop), notice the same three inputs we originally saw in Figure 4-1: data (D), clock enable (EN), and clock (>). The fourth input at the bottom of the component is the set/reset (SR) input we discussed in the previous section.

As you've seen, the clock enable input allows the flip-flop to keep its output state for multiple clock cycles. Without the En input, the output would just follow the input with one clock cycle of delay. Adding the En input lets the flip-flop store a state for a longer duration.

The last thing to notice in the diagram is the carry logic block, shown above and to the left of the LUT4. This block is mostly used to speed up arithmetic functions, such as addition, subtraction, and comparison.

While reviewing this diagram gave us an interesting look inside an FPGA and highlighted the central role of the LUT and the flip-flop, it isn't critical to memorize every detail of the PLB's architecture. You don't need to remember all the connections and how each is wired to its neighbor. In the real world, you write your Verilog or VHDL, and the FPGA tools take care of mapping that code onto the FPGA's resources. This is particularly useful if you want to switch from one type of FPGA to another (say, from a Lattice to an AMD). The beauty of Verilog and VHDL is that the code is generally portable; the same code works on different FPGAs, provided they have enough LUTs and flip-flops to do what you want.

Summary

In this chapter you learned about the flip-flop, which, along with the LUT, is one of the two most important components in an FPGA. You saw how flip-flops allow FPGAs to keep state, or remember past values, by only registering data from the input to the output on the positive edges of a clock signal. You learned how logic driven by flip-flops and clock signals is sequential, in contrast to the combinational logic of LUTs, and you got your first glimpse

of how flip-flops and LUTs work together through a project toggling an LED. You also learned how to avoid generating latches and how to reset a flip-flop to a default state.

In future chapters, as you build more complex blocks of code, you'll become more familiar with how flip-flops and LUTs interact and see how you can use just these two kinds of components to create large, sophisticated FPGA designs. You'll also see the role flip-flops play in keeping track of counters and state machines.