Project 3: Go Board Project – Using Registers and Clocks

Video: https://youtu.be/_7K-ty3Mffg?t=4
Link: https://nandland.com/project-3-the-flip-flop-aka-register/

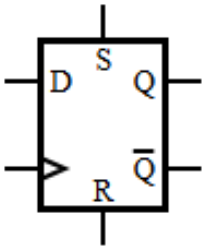
Introducing the D Flip-Flop

The previous project introduced **Look-Up Tables (LUTs).** This project will introduce a fundamental FPGA component: **The Flip-Flop**. Flip-Flops are a critical component to making FPGAs work. A Flip-Flop can also be called a Register, the two are used interchangeably.
Flip-Flops allow the FPGA to have knowledge of **state**. By state I mean that the FPGA is able to know what happened previously, and use that information to dictate what happens next.
For example, if you want to look for a sequence of button pushes in order to illuminate an LED, you need to have information about state. Let's say you want to illuminate LED 1 if the user first presses and releases Switch 1, then presses and releases Switch 2. *This would not be possible using LUTs alone*.
Or what about a situation where you want to increment a counter every time a button is pressed? The only way to increment a counter is to have knowledge of its previous value, and then add 1 to the previous value to create the new value. Again, this is not possible without Flip-Flops.

I also created a YouTube video for this project, should you prefer to follow along with that.

So What is A Flip-Flop?

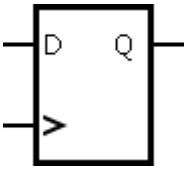
If you've taken a logic class before, you've probably been taught about the different types of Flip-Flops: D Flip-Flop, JK Flip-Flop, T Flip-Flop. I don't really understand why professors like to talk about all the different types of Flip-Flops. In the real world, 99.9999% of Flip-Flops are the D Flip-Flop. I've been designing FPGAs for over 10 years and I've never once seen or used a different type of Flip-Flop other than the D Flip-Flop (DFF).



The D Flip Flop! (Source: Wikipedia)

Let's look at a picture of a D Flip-Flop from Wikipedia. The image above shows every single pin on a normal Flip-Flop. I'll go through them quickly now, then I'll tell you to ignore most of them and only focus on three.
The top of the device contains the pin **S**, this is the **Set input**.
On the bottom, there's the pin labelled **R**, this is the **Reset input**.
On the right side is an output pin labelled **Qbar**,
this contains the opposite value of the **output Q**.

Now that I told you what those pins are for, let's completely forget about them! Let's instead look at the three most important pins on a Flip-Flop and focus on those.



D Data Input to Flip-Flop
Q Data Output of Flip-Flop (Registered)
> Clock Input to Flip-Flop

The first question you might be asking yourself is, *what is a clock?*

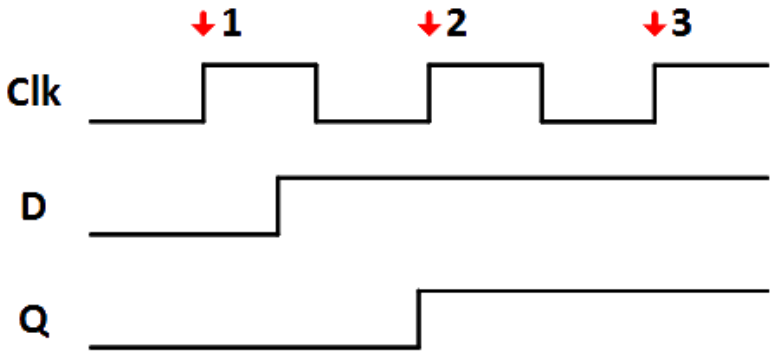
No, not this type of clock

This type of clock!

Digital clocks are what makes almost all digital circuits operate. One way to think about the clock in a system is to think of set of gears.
In order for any gear in the system to turn, there must be some master gear that drives all the others.
That is effectively the purpose of the clock in digital logic. It provides a steady stream of **low-to-high-to-low again** transitions of a voltage that lets your FPGA chug along.
Plus, the analogy is fun because the gears look a bit like the square wave of a digital clock.

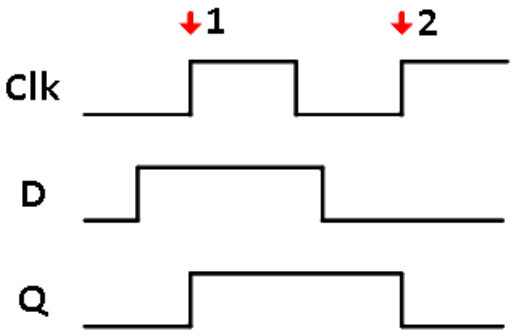
The clock on the Go Board oscillates at **25 MHz**, which means 25 million cycles per second.
A single clock cycle is from one rising edge to the next rising edge.

The clock is what allows a Flip-Flop to be used as a data storage element.
Any data storage elements are known as **sequential logic** or **registered logic**.
Sequential logic operates on the *transitions* of a clock. 99.9% of the time this will be the rising edge (when the clock goes from 0 to 1).
When a Flip-Flop sees a rising edge of the clock, it *registers* the data from the Input D to the Output Q.
Flip-Flops are what make complex FPGAs possible!
Let's look at a waveform of a few events.



D Flip-Flop Input to Output

The above waveform shows **three clock cycle events**, represented by the red arrow on the rising edge of the clock.
In between the first and second rising edge of the clock, the D input goes from low to high. The output Q sees that D has gone from low to high at the rising edge of the second clock cycle.
The rising edge is when the Flip Flop looks at the input data. At this point, Q becomes the same value as input D.
On the third rising edge, Q again checks the value of D and **registers** it (this is why flip-flops are often referred to as registers). Since it has not changed, Q stays high. Let's look at another waveform.



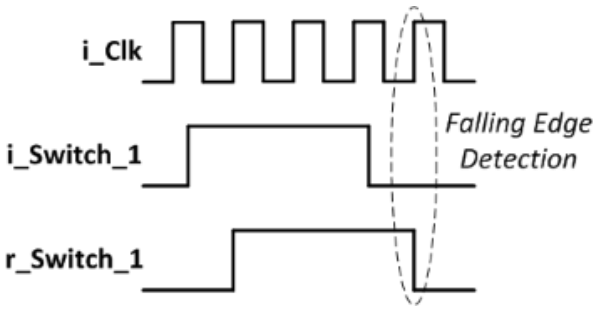
D Flip-Flop Sequence of Events

The above image shows a waveform of two inputs and one output for a D Flip-Flop.
The D Flip-Flop is sensitive to the rising edge of the clock, so when the rising edge comes along, the input D is passed along to the output Q. *This only occurs on the edges*.
On the first clock cycle, Q sees that D has become 1, so it toggles from 0 to 1.
On the second clock edge, Q again checks the value of D and sees that it is low again, so it becomes low.
If you want more examples showing Flip-Flop waveforms, check out the YouTube video I made.
Now that you know about clocks and Flip-Flops, let's talk about our next project.

Project Description

This project should toggle the state of LED 1, only when Switch 1 is released.

By toggle, I mean that if the LED is off, it should turn on. If the LED is on, it should turn off.
It might not be obvious yet, but this project requires Flip-Flops to complete. Without Flip-Flops, when the switch is released, how should it continue to stay on?
There needs to be some storage element that "remembers" that the LED is ON.
That indicates that we will need at least one Flip-Flop.



Additionally, the project description states that the LED should toggle ***only when Switch 1 is released***.
In order to know when it is released, that requires an additional Flip-Flop.
This Flip-Flop will be used to detect a transition on Switch 1. In this case, we are looking for the falling-edge of Switch 1.
A good way to look for a falling edge in an FPGA is to register the signal that you want to look at.
When the current value (unregistered) is equal to a zero,
but the previous value (registered) is equal to a one,
then we know that a falling edge has occurred.

This project will also require some logic between the two Flip-Flops. This will be implemented in the form of a LUT.
This project starts to show how Flip-Flops and LUTs work together to accomplish your goals.
You're welcome to try this on your own, should you need some hints, you can refer to the VHDL and Verilog below.

VHDL Code – Clocked_Logic_Intro.vhd:

```
 1library ieee;
 2use ieee.std_logic_1164.all;
 3
 4entity Clocked_Logic_Intro is
 5  port (
 6    i_Clk       : in  std_logic;
 7    i_Switch_1  : in  std_logic;
 8    o_LED_1     : out std_logic
 9    );
10end entity Clocked_Logic_Intro;
11
12architecture RTL of Clocked_Logic_Intro is
13
14  signal r_LED_1    : std_logic := '0';
15  signal r_Switch_1 : std_logic := '0';
16
17begin
18
19  -- Purpose: Toggle LED output when i_Switch_1 is released.
20  p_Register : process (i_Clk) is
21  begin
22    if rising_edge(i_Clk) then
23      r_Switch_1 <= i_Switch_1;         -- Creates a Register
```

```
24
25    -- This conditional expression looks for a falling edge on i_Switch_1.
26    -- Here, the current value (i_Switch_1) is low, but the previous value
27    -- (r_Switch_1) is high.  This means that we found a falling edge.
28    if i_Switch_1 = '0' and r_Switch_1 = '1' then
29        r_LED_1 <= not r_LED_1;         -- Toggle LED output
30    end if;
31    end if;
32  end process p_Register;
33
34  o_LED_1 <= r_LED_1;
35
36end architecture RTL;
```

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity Clocked_Logic_Intro is
5   port (
6     i_Clk       : in  std_logic;
7     i_Switch_1  : in  std_logic;
8     o_LED_1     : out std_logic
9     );
10end entity Clocked_Logic_Intro;
11
12architecture RTL of Clocked_Logic_Intro is
13
14  signal r_LED_1    : std_logic := '0';
15  signal r_Switch_1 : std_logic := '0';
16
17begin
18
19  -- Purpose: Toggle LED output when i_Switch_1 is released.
20  p_Register : process (i_Clk) is
21  begin
```

```
22    if rising_edge(i_Clk) then
23      r_Switch_1 <= i_Switch_1;         -- Creates a Register
24
25    -- This conditional expression looks for a falling edge on i_Switch_1.
26    -- Here, the current value (i_Switch_1) is low, but the previous value
27    -- (r_Switch_1) is high.  This means that we found a falling edge.
28    if i_Switch_1 = '0' and r_Switch_1 = '1' then
29        r_LED_1 <= not r_LED_1;         -- Toggle LED output
30    end if;
31    end if;
32  end process p_Register;
33
34  o_LED_1 <= r_LED_1;
35
36end architecture RTL;
```

Notice in the code above we have created a few new signals between the ***architecture declaration*** and the ***begin*** statement. These signals I named **r_LED_1** and **r_Switch_1**, because I know that they will be turned into Flip-Flops (Registers).
All signals that I know will become registers I call **r_.** It helps to keep the code organized and it helps when searching for signal names in a large file.
A signal **r_Data** is much easier to find than a signal called data.

The next fundamental part of the code introduces a new VHDL keyword: **process**.
A process is a block of code that is triggered by signals in the **sensitivity list**.
The sensitivity list is the list of signals in the parenthesis. In this case, this process will be triggered whenever the signal **i_Clk** changes. **i_Clk** will change when it goes from 0 to 1 or from 1 to 0 for example.
Processes are very complicated and require their own tutorial.
Luckily I already wrote one. I definitely recommend reading that article to get familiar with processes, they're going to become very important.

The statement **if rising_edge(i_Clk) then** is specifically looking for ***rising* edges** of i_Clk to perform all of the statements within the process.
Like I said previously, **a rising edge** of a clock will be used for 99.99% of the logic within your FPGA.
The rest of the code is used to create the logic we require to detect a falling edge on i_Switch_1, and then toggle the state of the LED.

Verilog Code – Clocked_Logic_Intro.v:

```
1module Clocked_Logic_Intro
2  (input  i_Clk,
3   input  i_Switch_1,
4   output o_LED_1);
5
6  reg r_LED_1    = 1'b0;
7  reg r_Switch_1 = 1'b0;
8
9  // Purpose: Toggle LED output when i_Switch_1 is released.
10 always @(posedge i_Clk)
11 begin
12   r_Switch_1 <= i_Switch_1;         // Creates a Register
13
14 // This conditional expression looks for a falling edge on i_Switch_1.
15 // Here, the current value (i_Switch_1) is low, but the previous value
16 // (r_Switch_1) is high.  This means that we found a falling edge.
17   if (i_Switch_1 == 1'b0 && r_Switch_1 == 1'b1)
18   begin
19     r_LED_1 <= ~r_LED_1;         // Toggle LED output
20   end
21 end
22
23 assign o_LED_1 = r_LED_1;
24
25endmodule
```


```
1 module Clocked_Logic_Intro
2   (input  i_Clk,
3    input  i_Switch_1,
4    output o_LED_1);
5
6   reg r_LED_1    = 1'b0;
7   reg r_Switch_1 = 1'b0;
8
9   // Purpose: Toggle LED output when i_Switch_1 is released.
10  always @(posedge i_Clk)
11  begin
12    r_Switch_1 <= i_Switch_1;         // Creates a Register
```

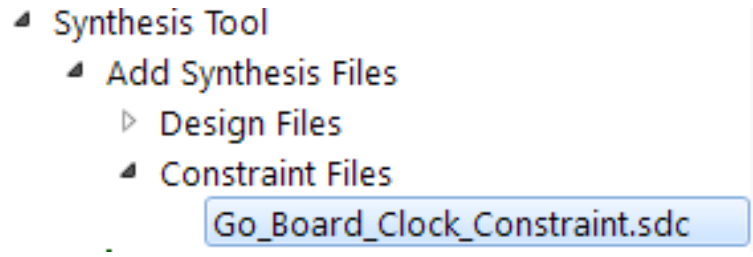
```
13
14  // This conditional expression looks for a falling edge on i_Switch_1.
15  // Here, the current value (i_Switch_1) is low, but the previous value
16  // (r_Switch_1) is high.  This means that we found a falling edge.
17    if (i_Switch_1 == 1'b0 && r_Switch_1 == 1'b1)
18    begin
19      r_LED_1 <= ~r_LED_1;         // Toggle LED output
20    end
21  end
22
23  assign o_LED_1 = r_LED_1;
24
25endmodule
```

Notice in the code above we have created a few new signals defined as **reg**.
These signals I named **r_LED_1** and **r_Switch_1**, because I know that they will be turned into Flip-Flops (Registers).
All signals that I know will become registers I call **r_**. It helps to keep the code organized and it helps when searching for signal names in a large file.
A signal **r_Data** is much easier to find than a signal called data.

The next fundamental part of the code introduces a new VHDL keyword: **always**.
An always block is a block of code that is triggered by signals in the **sensitivity list**.
The sensitivity list is the list of signals in the parenthesis after the **@ sign**.
In this case, this **always block** will be triggered whenever the signal **i_Clk** changes from a 0 to a 1, indicated by the Verilog keyword **posedge**.
Like I said previously, a rising edge of a clock will be used for 99.99% of the logic within your FPGA. Always blocks are very complicated and require their own tutorial.
Luckily I already wrote one. I definitely recommend reading that article to get familiar with always blocks, they're going to become very important.

The rest of the code is used to create the logic we require to detect a falling edge on i_Switch_1, and then toggle the state of the LED.

Building the Project



Since this project is the first one that we've done with a clock, we need to tell the FPGA tool about it.
Specifically, the tool needs to know if you have a 10 MHz clock, or a 50 MHz clock, or whatever.
In the case of the Go Board, the on-board clock is a **25 MHz clock**. A 25 MHz clock has a period of 40 nanoseconds (ns). The clock period tells the timing tools how much time it has to route wires between Flip-Flops.
As clock speed increases, it gets harder to "meet timing" during Place & Route.
Since the Go Board clock is relatively low, you shouldn't have any problems meeting timing.
In general, it's only when you have to deal with clocks that are faster than 100 MHz that you start to run into timing issues.
To learn more about this, you can read about Propagation Delay.
So now let's tell the tools about the clock period, by right-clicking on Constraint Files under Synthesis Tool, then add this Constraint File.
Note that these constraints get carried automatically into Place and Route.

Now we have two constraint files.
One is telling the tools which signals to map to which pins.
The other is telling the tools the frequency of our clocked logic on the Go Board.
Both are critical for getting your FPGA to work correctly.

Now let's run the FPGA build and look at the Synthesis and Place and Route report files.
In the synthesis report, we see that we are using 1 LUT and 2 DFF (D Flip-Flops). This is exactly what we expected!
Notice too that the tools identified the signal i_Clk as a clock.

Synthesis Report

```
Resource Usage Report for Clocked_Logic_Intro

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
Register bits not including I/Os:   2 (0%)
Total load per clock:
   i_Clk: 1

Mapping Summary:
Total  LUTs: 1 (0%)
```

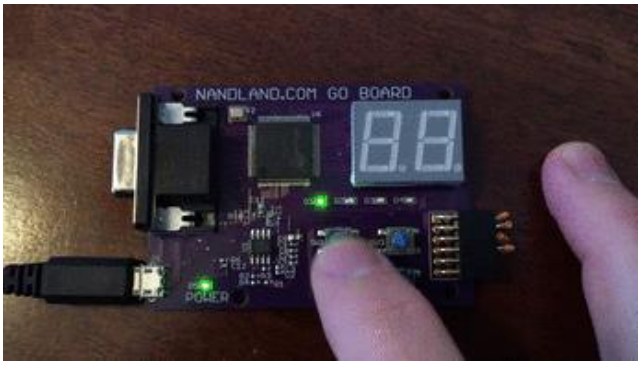
Place and Route Report

Now let's look at the **Place and Route report**. This can be viewed in iCEcube2 by clicking on **Reports** under **Output Files**, which is under P&R Flow.
There are two reports here. The first is a pin report, which tells you which signals were mapped to which pins. You can confirm here that your signals were mapped to the correct pins.
The second is the timing report. Let's look at that one.

In the timing report we see a section labeled **"Clock Frequency Summary"** which shows us whether we got our constraint file to be accepted correctly.
Here we see that the tools have found our clock i_Clk and it sees that we placed a 25 MHz constraint on it.
In my case, I was able to get the code to route at an actual frequency of 172.41 MHz. This means that I could run this FPGA at 172.41 MHz and it would still be guaranteed to work correctly. As long as the actual frequency is higher than the requested frequency, you shouldn't have any timing errors.

There's a lot of information in the Place and Route timing report, showing detailed timing information. Take a look at it.
In general, as long as you don't have timing errors, you shouldn't need to look at this file in detail.
Timing errors are a *pain* to deal with, but the Go Board was specifically designed to help you avoid them. Dealing with timing errors is a subject for an advanced FPGA project, which is out of the scope of the Go Board projects.

```
 ###################################################################
                    1::Clock Frequency Summary
 ===================================================================
Number of clocks: 1
Clock: i_Clk                          | Frequency: 172.41 MHz  | Target: 25.00 MHz   |
```



Pressing Switch 1 Toggles LED… Sometimes…

Conclusion

Congratulations! You've completed another project, the most complicated one yet!
Things have been a bit slow going, but now that you've learned the basics, the pace will pick-up quickly. Did you notice that the LED on this project only toggles sometimes?
Approximately half the time you pushed the button the LED toggles.
That seems strange, since you would think that a single button press would coincide with a single LED toggle. However, what is happening is that the switch is subject to ***mechanical bouncing***.
If you want to fix this bouncing problem, you need to create a **debouncer** in your FPGA logic, which is exactly what we will do in the next project.
This debounce project also introduces how to count up in an FPGA design.

Before we get to that, let's just take a step back and discussed what we've seen.
The way FPGAs work is fundamentally different than the way a traditional computer works.
A traditional computer has one central processor and all instructions go through that.
A microcontroller might have some built-in peripherals like SPI or I2C, but they need to be available to you on the hardware, it's usually not possible to create them if they're not already there.
An FPGA on the other hand has no central processor. With just the two elements that we've introduced so far, the LUT and the Flip-Flop, we can perform a huge amount of functionality.
An FPGA is just a grid of Flip-Flops and LUTs ready for whatever you want to throw at them.

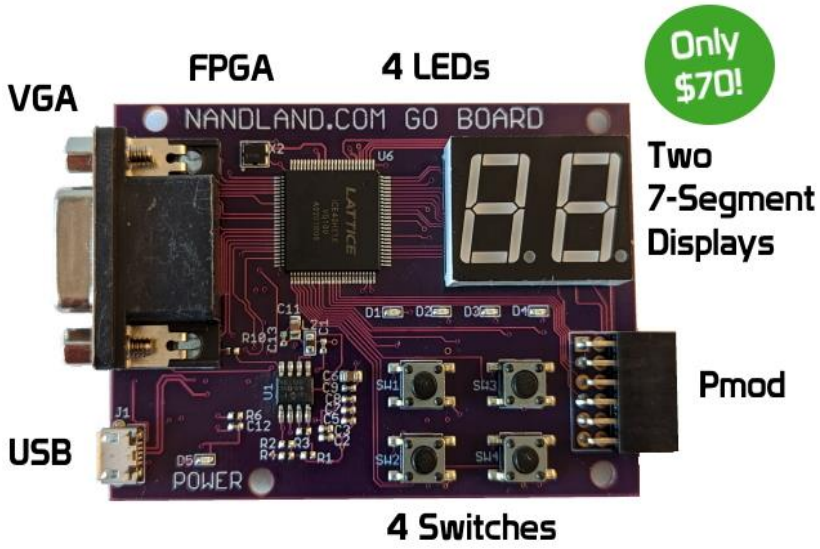
By giving the designer total control, FPGAs have a huge amount of versatility.
You're able to do things like interface with I2C, SPI, PCI, ADCs, DACs, find Bitcoins, design Pong, create a UART, talk to external memory, and much more, all in a single device!

The remaining projects will show you what other peripherals are available on the Go Board, all while reinforcing what you've already been exposed to.
Don't worry if you're still unclear about the details of FPGAs, as you do more of these projects the concepts will become more clear.
So let's learn about debouncing switches, which will show how counters work in an FPGA.

Your Next Go Board Project: Debouncing a Switch

Go Board Tutorials

- Download and Install the FPGA Tools
- Project 1 – Your First Go Board Project
- Project 2 – The Look-Up Table (LUT)
- Project 3 – The Flip-Flop (AKA Register)
- Project 4 – Debounce A Switch
- Project 5 – Seven Segment Display
- Project 6 – How To Simulate Your FPGA Designs
- Project 7 – UART Part 1: Receive Data From Computer
- Project 8 – UART Part 2: Transmit Data To Computer
- Project 9 – VGA Introduction (Driving Test Patterns to VGA Monitor)
- Project 10 – PONG
- Go Board Support Stuff



BUY NOW

One Comment

1. 

Hans Offenfrish June 14, 2024 at 10:55 pm - Reply

I noticed that if I power off my computer and then turn it on a day or more later, the LEDs that I had programmed to light up prior to powering down light up again. Which component on the Go Board is responsible for remembering the binary code even when the Go Board is powered off? Is this ability to remember your code a characteristic of all FPGAs?