Project 7 Latches and Flip-Flops

The basic memory elements used in sequential circuits

Introduction

This project introduces the concept of electronic memory in digital circuits. Digital circuits need memory in order to create ordered sequences of events, like playing one note after another to reproduce a digitized song or to track and respond to sequences of events, like ensuring a particular sequence of button presses is used to open a digital combination lock.

Most digital circuits use electronic memory. Electronic memory circuits store the state of an input signal at some particular time, and their memorized outputs (or data) can be used at a later time. Digital circuits that use electronic **memory** are called **sequential** circuits because they can use memorized data to create or sense ordered sequences of events. In contrast, **combinational** circuits do not use memory, and so they can only change their outputs to reflect the current state of their **inputs**. Creating sequences is fundamental to all of computing, and many 100's of billions of computing circuits are in use today.

This project presents one of the most common memory devices used in digital circuit design - the **D-flip-flop**. This device **memorizes** a single **input** when triggered by a **timing signal**, and has a single **output** that always shows the state of the memorized signal. How D-flip-flop responds to the timing signal – the D-flip-flop only "samples" (or reads) **input at the rising edge of the timing signal**.

Before you begin, you should:

- Have a good amount of experience using Vivado and the Boolean Board;
- Be able to write and execute a Verilog testbench;
- Know how to model and simulate circuit delays.

After you're done, you should:

- Understand the operation of latches and flip-flops;
- Be able to describe memory circuits in behavioral Verilog.

Requirements

1. Parallel In Parallel Out (PIPO) Shift Register

Define an **8-bit PIPO register** in Verilog, with **8 slide switches** connected to the **inputs**. Connect **each of the 8 outputs to the "I1" input of a 2:1 mux,** and connect the **slide switches to the "I0" input of the mux**. Connect the **mux outputs** to **8 green LEDs**, connect **1 pushbutton to the PIPO register's clock input**, and connect a **second pushbutton to the mux select.** Verify that you can use the **first button to 'memorize' the switch positions**, and use the **second button to 'recall' them onto the LEDs.**

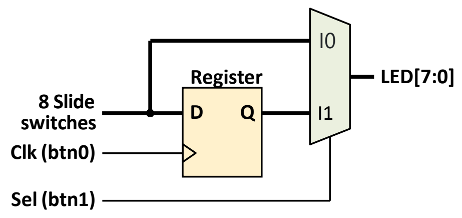


Figure 1. Using **a button as a clock**

Note: Clock signals and inputs are very **noise sensitive,** and the **clock networks inside the FPGA** are carefully **managed** by the design tools. Under normal conditions, only certain signal sources are allowed to drive clock inputs, including dedicated clock input pins, the specialized "clock management tile" inside the FPGA, and signals arising directly from other flip-flops. For this requirement, you are asked to use a **non-conforming input (from a pushbutton) as a clock signal to better illustrate basic operations**. **Vivado will only allow this if you include the line "set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets btn[0]];" in the .xdc file.**

Logic **signals** are often used to **control** when data is **loaded in a register**. Using a logic **signal as a clock** is generally considered poor design practice. A far better solution is to drive the **flip-flop's clock from a proper clock source**, and then use the **logic signal as a "clock enable" signal.** This protects the integrity of the clock network while preserving the same functionality.

Modify your **code** to connect the **FPGA's main clock** to the **clock input**, and use **btn[0] as an enable**. The FPGA's **main clock** signal is called **clk** in the **uploaded constraint file**, so you **must include clk as an input to your module**.

A complete constraint file (that includes clock) for the Booleanboard is uploaded in StudIP - you can use this file for all remaining projects in this course.

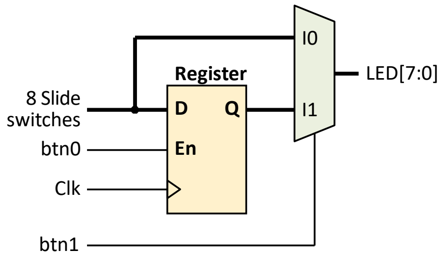


Figure 2. Using **a Clock Enable**

Counters, Clock Dividers

Measuring time and Creating Derivative Clocks

Introduction

This project continues exploring sequential circuits, building on the foundational concepts and devices that were introduced in a previous module. In this project, several sequential components like counters and registers are presented. These circuits can be used by themselves to solve certain problems, but they are more typically used as components in larger, more involved sequential designs.

Measuring and managing time accurately is one of the most fundamental tasks in a digital system. Time is measured by electronic circuits that are stabilized by the resonant oscillations of quartz crystals or silicon oscillators. These highly stable and repeatable mechanical vibrations are used to create a reliable “master clock” signal that can be routed to the various IC’s in a digital system. Clock management circuits inside a given IC use the input clock signal to drive counters that measure time intervals and/or produce derivative clock signals.

This project presents various counter circuits that can accurately measure time intervals, and clock divider circuits that can be used to produce new clock signals that are derivatives of the input “master clock” signal.

Before you begin, you should:

- Know Vivado, how to write testbenches, and how to use the simulator;
- Know how to describe memory devices in Verilog.

After you’re done, you should:

Know how to describe counters in Verilog;
Understand clock dividers and how to design them;
Understand procedural statements in Verilog.

Requirements

2. Design a clock divider based on an asynchronous counter

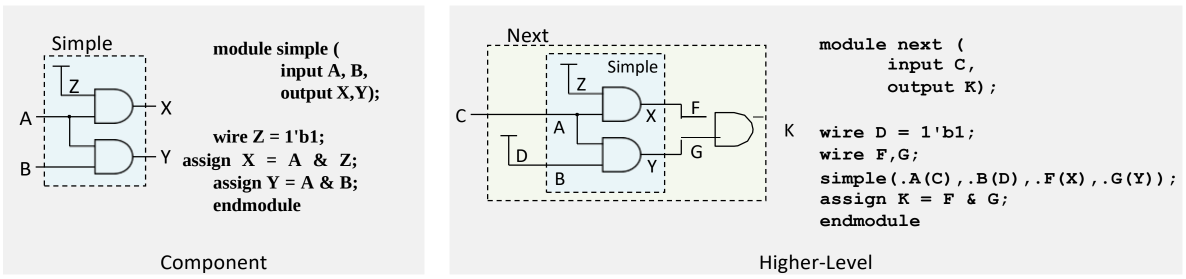
Create a **clock divider** that uses a structural **asynchronous counter built from Xilinx flip-flop primitives**. The **counter** uses the **main Boolean board 100Mhz clock as an input**, and it should **generate a clock signal below 1Hz to drive the LED.** The following tutorial may help you.

Structural Implementation of an Asynchronous Counter

Structural Verilog using Xilinx flip-flop primitives

Structural Verilog techniques can be useful when **preexisting circuit blocks can be reused in a new design**. Xilinx makes **many basic components** available in the Vivado tool, including D-FF's. Here, we present **an asynchronous counter built from Xilinx flip-flop primitives.**

Recall that structural Verilog builds a circuit by interconnecting preexisting circuit components. A component is instantiated in a Verilog file by including its name and then listing all the port connections. An example is presented below. The component called "simple" is instantiated and used in the higher-level design called "next". In the module next, the component simple is instantiated by including its name, followed by a port-connection list in parenthesis. In the port connection list, the component name is written first, immediately after a period, and then the higher-level port to which it connects is shown in parenthesis.



```
module simple (
        input A, B,
        output X,Y);

    wire Z = 1'b1;
assign  X  =  A  &  Z;
    assign Y = A & B;
    endmodule
```

```
module next (
      input C,
      output K);

wire D = 1'b1;
wire F,G;
simple(.A(C),.B(D),.F(X),.G(Y));
assign K = F & G;
endmodule
```

Figure 1. Structural Verilog Example

For this **asynchronous counter design, the Xilinx flip-flop primitive called "dff_inst0" can be used**. The Verilog code below shows how this component can be instantiated in a Verilog source file.

```
dff dff_inst0 (
    .clk(clk),
    .rst(rst),
    .D(din[0]),
    .Q(clkdiv[0])
);
```

The clock divider in this design receives a 100MHz clock, and it must produce a clock that toggles at less than 1Hz – this requires 27 flip-flops (since 100MHz / 2^27 is .745Hz). The figure below shows the circuit.

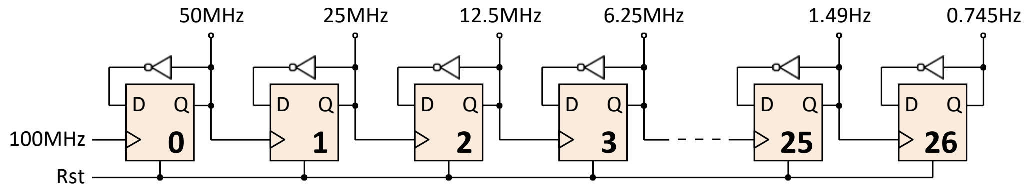


Figure 2. A clock divider based on an asynchronous counter

To create this circuit, you could instantiate 27 copies of the DFF Verilog code above, and then manually connect all the clock signals between neighboring flip-flops. Or, you can use a Verilog for loop to automatically "tile out" the flip-flops for you. The for loop can create copies of the flip-flop, and we can create buses to define the input and output signals between the flip-flops.

```
genvar i;
generate
for (i = 1; i < 27; i=i+1)
begin : dff_gen_label
    dff dff_inst (
        .clk(clkdiv[i-1]),
        .rst(rst),
        .D(din[i]),
        .Q(clkdiv[i])
    );
    end
endgenerate
```

In the code below, the flip-flop 0 is instantiated on its own, and then the for loop creates the remaining 26. Note the "assign din = ~clkdiv" statement towards the bottom of the code listing – that statement inverts every flip-flop's output before returning it to the input. The Verilog file for this design is below. Take the time to look through the code, and be sure you understand what's happening.

```
`timescale 1ns / 1ps
module clk_divider(
    input clk,
    input rst,
    output led
    );

wire [26:0] din;
wire [26:0] clkdiv;

dff dff_inst0 (
    .clk(clk),
    .rst(rst),
    .D(din[0]),
    .Q(clkdiv[0])
);

genvar i;
generate
for (i = 1; i < 27; i=i+1)
begin : dff_gen_label
    dff dff_inst (
        .clk(clkdiv[i-1]),
        .rst(rst),
        .D(din[i]),
        .Q(clkdiv[i])
    );
    end
endgenerate

assign din = ~clkdiv;

assign led = clkdiv[26];

endmodule
```

A Test Bench to Simulate the Clock Divider Circuit

As with the combinational circuit we have designed in previous projects, you can draft a test bench to test the circuit out before implementing it on-chip. However, in this test bench, we need to emulate the clock signal and the rst signal. The clock signal is actually a constantly oscillating signal. Using the Boolean Board as an example, the input clock frequency is 100 MHz, i.e., the period of the clock is 10 ns. For half of the period, the clock is high, and half of the period clock islow. In other words, every half of the period, 5 ns in this case, the clock will flip itself. To simulate the clock signal, instead of putting it in the initialized statement, we will use an always statement.

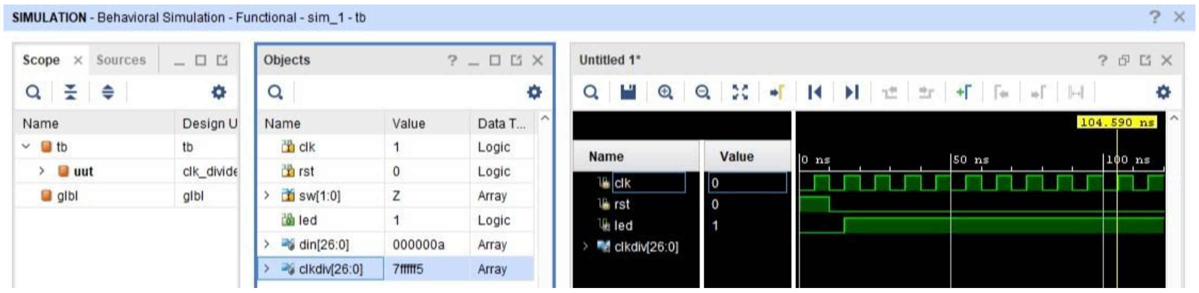
```
`timescale 1ns / 1ps

...

reg clk;

always
    #5 clk = ~clk;
```

In the initialize block, we will initialize `clk` signal to 0 and hold `rst` high for 10ns to reset the clock divider. So, the Verilog Test Bench will look like this:

```
`timescale 1ns / 1ps

module tb;

    // Inputs
    reg clk;
    reg rst;

    // Outputs
    wire led;

    // Instantiate the Unit Under Test (UUT)
    clk_divider uut (
        .clk(clk),
        .rst(rst),
        .led(led)
    );

    always
        #5 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 1;

        #10 rst = 0;

        // Wait 100 ns for global reset to finish
        #100;

    end

endmodule
```

Simulating the Clock Divider

When you first run behavioral simulation, the internal signals such as `clkdiv[26:0]` won't appear in the simulation window. You can access these signals by **opening windows Scope and Objects** on the left side of the simulation window and clicking on **uut under Scope window**. Now you can see `clkdiv[26:0]` under Objects window, go ahead and **click and drag this signal to the simulation window under the Name column as shown in Figure 3.** You can also drag `clkdiv[26:0]` signals into the simulation window one by one in any order you want.


SIMULATION - Behavioral Simulation - Functional - sim_1 - tb
Scope
Sources
Objects
Untitled 1*
104.590 ns
0 ns
50 ns
100 ns

Name	Design U
tb	tb
uut	clk_divide
glbl	glbl

Name	Value	Data T...
clk	1	Logic
rst	0	Logic
sw[1:0]	Z	Array
led	1	Logic
din[26:0]	000000a	Array
clkdiv[26:0]	7ffff5	Array

Name	Value
clk	0
rst	0
led	1
clkdiv[26:0]	

Figure 3. Accessing Internal Signals

Now you have `clkdiv[26:0]` in your simulation window, but it doesn't have any signals. In order to see those signals, you need to **relaunch** your simulation by clicking the Relaunch Simulation icon as shown in Figure 4. You can also achieve the same result by clicking on the Restart icon to restart the simulation and then clicking on the Run All icon to start the simulation again, both of which can be found on the same toolbar as the Relaunch Simulation icon.

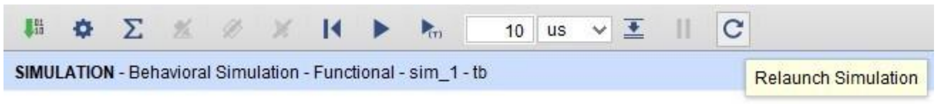
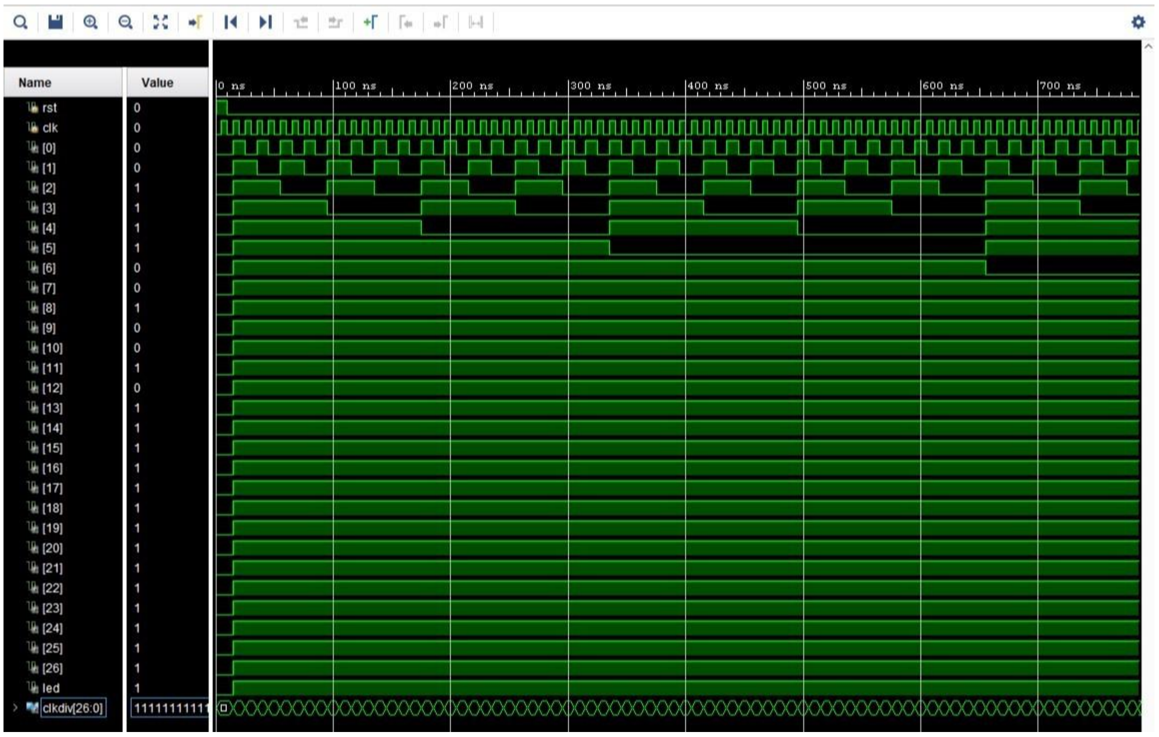

10
us
SIMULATION - Behavioral Simulation - Functional - sim_1 - tb
Relaunch Simulation

Figure 4. Relaunching Simulation

The simulated waveform is shown in Fig. 5 below. You can see in the waveform that clkdiv[0] is **half** of the frequency of clk, and that clkdiv[1] is half of the frequency of clkdiv[2].



Name	Value
rst	0
clk	0
[0]	0
[1]	0
[2]	1
[3]	1
[4]	1
[5]	1
[6]	0
[7]	0
[8]	1
[9]	0
[10]	0
[11]	1
[12]	0
[13]	1
[14]	1
[15]	1
[16]	1
[17]	1
[18]	1
[19]	1
[20]	1
[21]	1
[22]	1
[23]	1
[24]	1
[25]	1
[26]	1
led	1
clkdiv[26:0]	11111111111

Figure 5. Clock Divider Simulation Waveform

3. BCD Counter

Create a **4-bit decimal** (BCD) counter that continuously counts **0-9**. Drive the counter from the **1Hz clock**, connect its **outputs to one digit of the 7seg display**, and verify it counts through all digits at a rate of one digit per second. (**Note:** you need to place a seven-segment decoder between the BCD counter and the seven-segment display cathode signals).