Modeling Registers and Counters

Introduction

When several flip-flops are grouped together, with a common clock, to hold related information the resulting circuit is called a register. Just like flip-flops, registers may also have other control signals. You will understand the behavior of a register with additional control signals. Counters are widely used sequential circuits. In this lab you will model several ways of modeling registers and counters. *Please refer to the Vivado tutorial on how to use the Vivado tool for creating projects and verifying digital circuits.*

Objectives

After completing this lab, you will be able to:
- Model various types of registers
- Model various types of counters
- ***You will be asked to demo one of the different projects in this assignment. Make sure you have full understanding of each part.***

Registers   Part 1

In a computer system, related information is often stored at the same time. A **register** stores bits of information in such a way that systems can write to or read out all the bits simultaneously. Examples of registers include data, address, control, and status. Simple registers will have separate data input and output pins but clocked with the same clock source. A simple register model is shown below.

```
module Register
(
 input [3:0] D, input Clk,
 output reg [3:0] Q
 );
always @(posedge Clk)
          Q <= D;
endmodule
```

Notice that this is similar to a simple D Flip-flop with multiple data port.

The simple register will work where the information needs to be registered every clock cycle. However, there are situations where the register content should be updated only when certain condition occurs. For example, a status register in a computer system gets updated only when certain instructions are executed. In such case, a register clocking should be controlled using a control signal. Such registers will have a clock enable pin. A model of such register is given below.

```
module Register_with_synch_load_behavior
(
 input [3:0] D, input Clk, input load,
 output reg [3:0] Q
);
   always @(posedge Clk) if
      (load)
         Q <= D;
endmodule
```

Another desired behavior of registers is to reset the content when certain condition occurs. A simple model of a register with synchronous reset and load (reset has a higher priority over load) is shown below

```
module Register_with_synch_reset_load_behavior
(
 input [3:0] D, input Clk, input reset, input load,
 output reg [3:0] Q
 );
   always @(posedge Clk)
       if (reset)
       begin
          Q <= 4'b0;
       end
       else if (load)
       begin
           Q <= D;
       end
endmodule
```

1-1. Model a 4-bit register with synchronous reset and load using the model provided above. Develop a testbench and simulate the design. Assign Clk, D input, reset, load, and output Q. Verify the design in hardware.
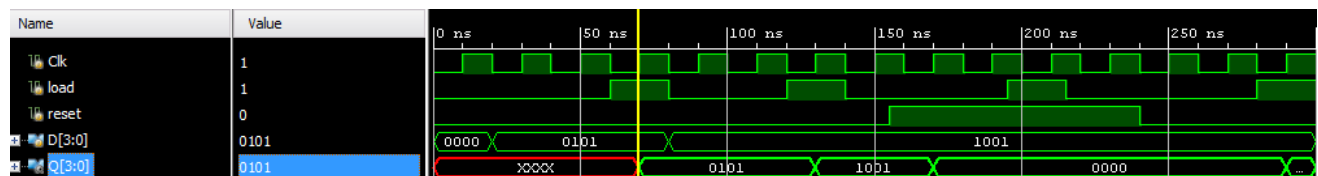
1-1-1. Create a blank project called 4-bit register.

1-1-2. Create and add the Verilog module that will model the 4-bit register with synchronous reset and load. Use the code provided in the above example.

1-1-3. Develop a testbench and simulate the design. Analyze the output.

1-1-4. Add the appropriate board related master XDC file to the project and edit it to include the related pins, assigning *Clk* to **SW15**, *D* input to **SW3-SW0**, *reset* to **SW4**, *load* to **SW5**, and *Q* to **LED3-LED0**.

1-1-5. Add the following line of code in the XDC file to allow SW15 be used as a clock

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets { clk }];
```

1-1-6. Synthesize the design.

1-1-7. Implement the design.

1-1-8. Generate the bitstream, download it into the Basys3 board, and verify the functionality.

In some situations, it is necessary to set the register to a pre-defined value. For such a case, another control signal, called set, is used. Typically, in such registers, reset will have a higher priority over set, and set will have a higher priority over load control signal.

1-2.   Model a 4-bit register with synchronous reset, <u>set</u>, and load signals. Assign Clk, D input, reset, set, load, and output Q. Verify the design in hardware.

```
 5
 6 module Register_with_synch_set_reset_load_behavior(
 7     input Clk,
 8     input [3:0] D,
 9     input set,
10     input reset,
11     input load,
12     output reg [3:0] Q
13     );
14
15     always @(posedge Clk)
16     if (reset)
17        Q <= 4'b0;
18     else if (set)
19        Q <= 4'b1111;
20     else if (load)
21        Q <= D;
22 endmodule
23
```

The above registers are categorized as parallel registers. ***There are another kind of registers called shift registers. A shift register is a register in which binary data can be stored and then shifted left or right when the control signal is asserted. Shift registers can further be sub-categorized into parallel load serial out, serial load parallel out, or serial load serial out shift registers.*** They may or may not have reset signals.

In Xilinx FPGA, LUT can be used as a serial shift register with one bit input and one bit output using one LUT as SRL32 providing efficient design (instead of cascading up to 32 flip-flops) provided the code is written properly. It may or may not have enable signal. When the enable signal is asserted, the internal content gets shifted by one bit position and a new bit value is shifted in. Here is a model for a simple one- bit serial shift in and shift out register without enable signal. It is modeled to shift for 32 clock cycles before the shifted bit brought out. **This model can be used to implement a delay line**.

```
module simple_one_bit_serial_shift_register_behavior
(input Clk, input ShiftIn,
 output ShiftOut
 );
    reg [31:0] shift_reg;

    always @(posedge Clk)
          shift_reg <= {shift_reg[30:0], ShiftIn};
    assign ShiftOut = shift_reg[31];

endmodule
```

The above model can be modified if we want to implement a delay line less than 32 clocks. Here is the model for the delay line of 3 clocks.

```
module delay_line3_behavior
(input Clk, input ShiftIn,
 output ShiftOut);

 reg [2:0] shift_reg;

    always @(posedge Clk)

          shift_reg <= {shift_reg[1:0], ShiftIn}; assign
```
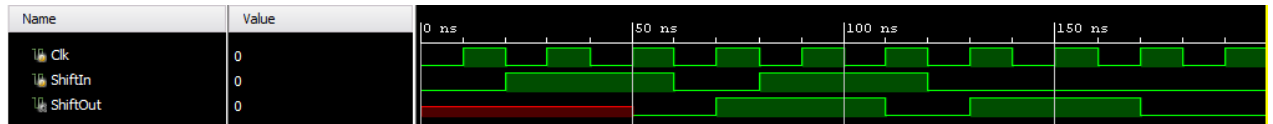
```
        ShiftOut = shift_reg[2];

endmodule
```

1-3.   Model a 1-bit delay line shift register using the above code. Develop a testbench and simulate the design using the stimuli provided below. Assign Clk, ShiftIn, and output ShiftOut.

The following code models a four-bit parallel in shift left register with load and shift enable signal..

```
module Parallel_in_serial_out_load_enable_behavior
(input Clk, input ShiftIn, input [3:0] ParallelIn, input load, input ShiftEn,
output ShiftOut, output [3:0] RegContent
 );
    reg [3:0] shift_reg;

always @(posedge Clk) if(load)
        shift_reg <= ParallelIn;
      else if (ShiftEn)
        shift_reg <= {shift_reg[2:0], ShiftIn};

    assign ShiftOut = shift_reg[3];
    assign RegContent = shift_reg;

endmodule
```

1-4.   Model a 4-bit parallel in left shift register using the above code. Develop a testbench and simulate the design using the stimuli provided below. Assign Clk, ParallelIn, load, ShiftEn, ShiftIn, RegContent, and ShiftOut. Verify the design.
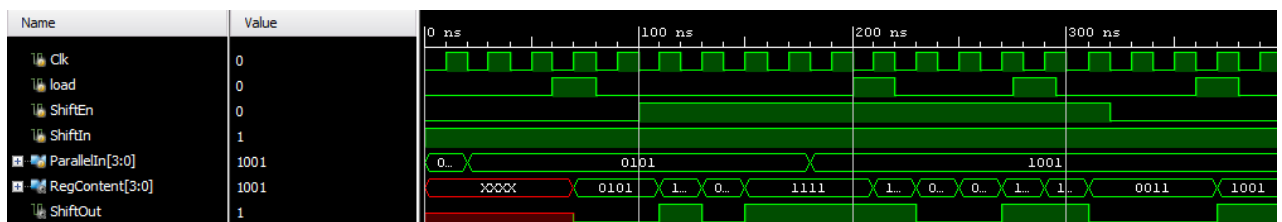
1-5.   Write a model for a 4-bit serial in parallel out shift register. Develop a testbench and simulate the design. Assign Clk, ShiftEn, ShiftIn, ParallelOut, and ShiftOut. Verify the design.
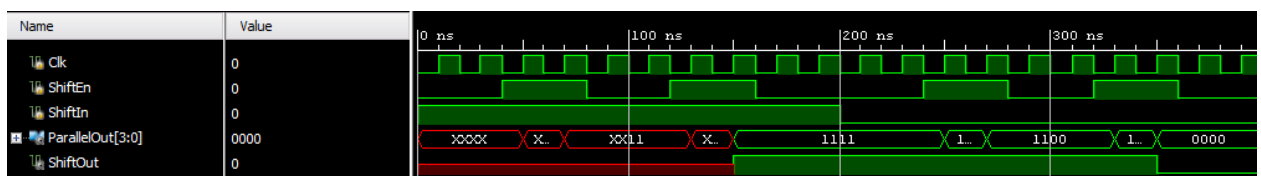
# Counters   Part 2

Counters can be asynchronous or synchronous. Asynchronous counters count the number of events solely using an event signal. Synchronous counters, on the other hand, use a common clock signal so that when several flip-flops must change state, the state changes occur simultaneously.

A binary counter is a simple counter which counts values up when an enable signal is asserted and will reset when the reset control signal is asserted. Of course, a clear signal will have a higher priority over the enable signal. The following diagram shows such a counter. Note that clear is an asynchronous negative logic signal whereas Enable is synchronous positive logic signal.
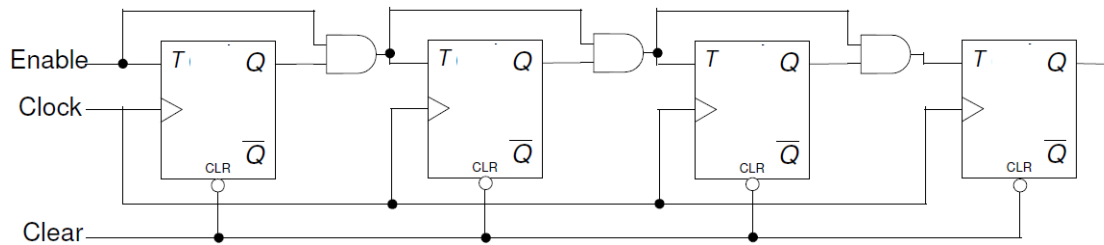
2-1.   Design a 8-bit counter using T flip-flops, extending the above structure to 8-bits. Your design needs to be hierarchical, using a T flip-flop in behavioral modeling, and rest either in dataflow or gate-level modeling. Develop a testbench and validate the design. Assign Clock input, Clear_n, Enable, and Q. Implement the design and verify the functionality.
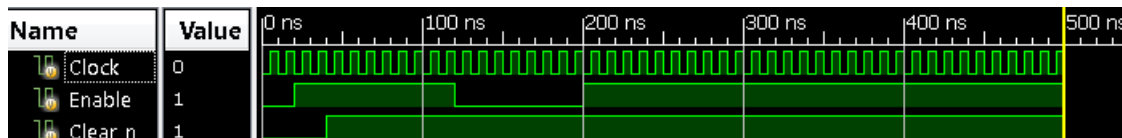
2-1-8.   Counters can also be implemented using D flip-flops since a T flip-flop can be constructed using a D flip-flop as shown below.
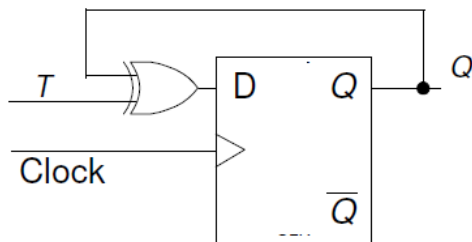
2-2.   Modify the 8-bit counter using D flip-flops. The design should be hierarchical, defining D flip-flop in behavioral modeling, creating T flip-flop from the D flip-flop, implementing additional functionality using dataflow modeling. Assign Clock input, Clear_n, Enable, and Q. Implement the design and verify the functionality.

Other types of binary counters include (i) up, (ii) down, and (iii) up-down. Each one of them may have count enable and reset as control signals. There may be situation where you may want to start the count up/down from some non-zero value and stop when some other desired value is reached. Here is an example of a 4-bit counter which starts with a value 10 and counts down to 0. When the count value 0 is reached, it will re-initialize the count to 10. At any time, if the enable signal is negated, the counter pauses counting until the signal is asserted back. It assumes that load signal is asserted to load the pre- defined value before counting has begun.

```
reg [3:0] count; wire
cnt_done;

assign cnt_done = ~| count;
assign Q = count;

always @(posedge Clock) if
  (Clear)
     count <= 0;
      else if (Enable)
     if (Load | cnt_done)
        count <= 4'b1010;   // decimal 10
     else
        count <= count - 1;
```

2-3.   Model a 4-bit down-counter with synchronous load, enable, and clear as given in the code above. Develop a testbench (similar to the waveform shown below) and verify the design works. Assign Clock input, Clear, Enable, Load, and Q. Implement the design and verify the functionality.
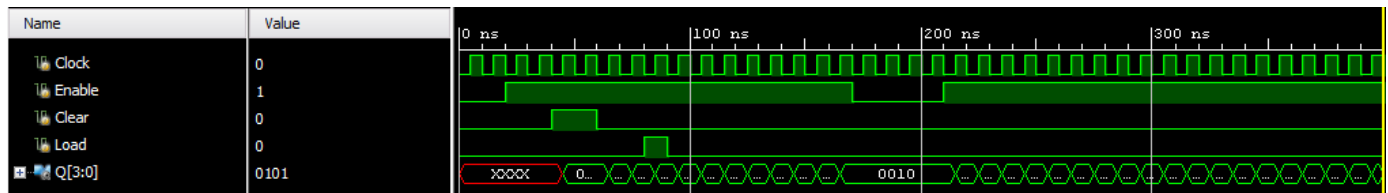
Conclusion

In this lab, you learned how various kinds of registers and counters work. You modeled and verified the functionality of these components. These components are widely used in a processor system design.

What to Submit?
Submit 100 to 150 words describing what you learned in this assignment and what you can improve on next time.