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. In this lab, you will understand the behavior of a register with additional control signals. In addition to registers, counters are widely used sequential circuits. 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

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.

```
architecture Register_behavioral of Register is begin

signal D : std_logic_vector(3 downto 0);
signal Q : std_logic_vector(3 downto 0);

begin

process (clk) begin
      if rising_edge(clk) then
            Q <= D;
      end if;
end process;
end Register_behavioral;
```

Notice that this is essentially a simple D Flip-flop with multiple data ports.

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.

```
architecture Register_with_sync_load_behavior of Register_with_sync_load is

signal D : std_logic_vector(3 downto 0);
signal Q : std_logic_vector(3 downto 0);

begin
process (clk) begin
      if rising_edge(clk) then
            if (load = '1') then
                  Q <= D;
            end if;
      end if;
end process;
end Register_with_sync_load_behavior;
```

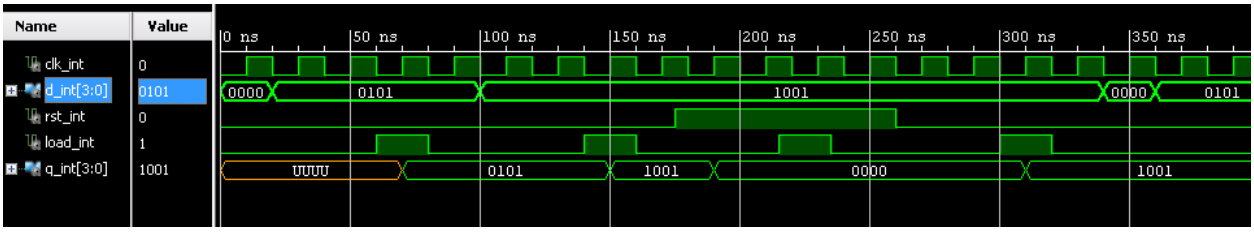
Another desired behavior of registers is to reset the content when certain conditions occur. A simple model of a register with synchronous reset and load (reset has a higher priority over load) is shown below

```
architecture Register_with_sync_reset_load_behavior of
Register_with_sync_reset_load is begin

signal D : std_logic_vector(3 downto 0);
signal Q : std_logic_vector(3 downto 0);

begin
process (clk)
      if rising_edge(clk) then
            if (reset = '1') then
                  Q <= "0000";
            elsif (load = '1') then
                  Q <= D;
            end if;
end process;
end Register_with_sync_reset_load_behavior;
```

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 to SW15, D input to SW3-SW0, reset to SW4, load to SW5, and output Q to LED3-LED0. Verify the design in hardware.



1-1-1. Open Vivado and create a blank project called lab6_1_1**.**

1-1-2. Create and add the VHDL 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. Synthesize the design.

1-1-5. Create and add the XDC file, assigning *Clk* to ***SW15***, *D* input to ***SW3-SW0***, *reset* to ***SW4***, *load* to ***SW5***, and *Q* to ***LED3-LED0***.

1-1-6. Implement the design.

Look at the Project Summary and note that 1 BUFG and 11 IOs are used.

If you open Utilization Report file, you will notice that there are 4 IOB registers are used.

1-1-7. Generate the bitstream, download it into the Nexys4 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, set, and load signals. Assign Clk to SW15, D input to SW3-SW0, reset to SW4, set to SW5, load to SW6, and output Q to LED3-LED0. Verify the design in hardware.

1-2-1. Open Vivado and create a blank project called lab6_1_2**.**

1-2-2. Create and add the VHDL module that will model the 4-bit register with synchronous reset, set, and load control signals.

1-2-3. Synthesize the design.

1-2-4. Create and add XDC file, assigning *Clk* to ***SW15***, *D* input to ***SW3-SW0***, *reset* to ***SW4***, *set* to ***SW5***, *load* to ***SW6***, and *Q* to ***LED3-LED0***.

1-2-5. Implement the design.

Look at the Project Summary and note the resources used. Understand the result.

1-2-6. Generate the bitstream, download it into the Nexys4 board, and verify the functionality.

The above registers are categorized as parallel registers. There are other kinds 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 FPGAs, a 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.

```
architecture simple_one_bit_serial_shift_register_behavior of
simple_one_bit_serial_shift_register is begin

signal shift_reg : std_logic_vector(31 downto 0);
signal shift_in : std_logic;
signal shift_out : std_logic;

begin
process (clk) begin
      if rising_edge(clk) then
            shift_reg <= shift_reg(30 downto 0) & shiftin;
      end if;
      shiftout <= shift_reg(31);
end process;
end simple_one_bit_serial_shift_register_behavior;
```

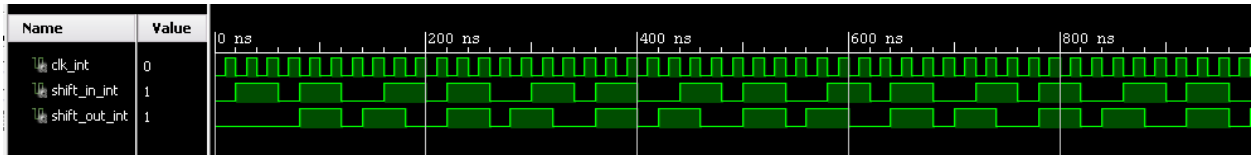
The above model can be modified if we want to implement a delay line less than 32 clocks. Here is the model for the delay line of 3 clocks.

```
architecture delay_line3_behavior of delay_line3 is begin

signal shift_reg : std_logic_vector(2 downto 0);
signal shift_in : std_logic;
signal shift_out : std_logic;

begin
      process (clk) begin
            if rising_edge(clk) then
                  shift_reg <= shift_reg(1 downto 0) & shiftin;
            end if;
            shiftout <= shift_reg(2);
      end process;
end delay_line3_behavior;
```

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 to SW15, ShiftIn to SW0, and output ShiftOut to LED0. Verify the design in hardware.



1-3-1. Open Vivado and create a blank project called lab6_1_3**.**

1-3-2. Create and add the VHDL module that will model the 1-bit delay line shift register using the provided code.

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

1-3-4. Synthesize the design.

1-3-5. Create and add the XDC file, assigning *Clk* to ***SW15***, *ShiftIn* to ***SW0***, and *ShiftOut* to ***LED0***.

1-3-6. Implement the design.

Look at the Project Summary and note the resources used. Understand the result.

1-3-7. Generate the bitstream, download it into the Nexys4 board, and verify the functionality.

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

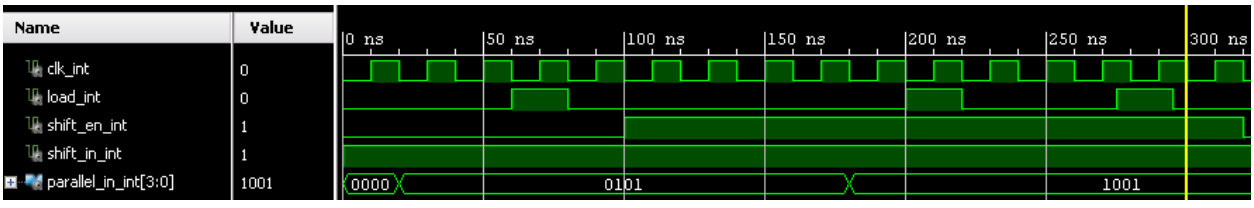
```
architecture Parallel_in_serial_out_load_enable_behavior of
Parallel_in_serial_out_load_enable is begin

signal shift_reg : std_logic_vector(3 downto 0);
signal parallelin : std_logic_vector(3 downto 0);

signal load : std_logic;
signal shift_en : std_logic;
signal shift_in : std_logic;
signal shift_out : std_logic;

begin
      process (clk) begin
            if rising_edge(clk) then
                  if (load = '1') then
                        shift_reg <= parallelin;
                  elsif (shiften = '1') then
                        shift_reg <= shift_reg(3 downto 0) & shiftin;
                  end if;
            shiftout <= shift_reg(3);
      end process;
end Parallel_in_serial_out_load_enable_behavior;
```

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 to SW15, ParallelIn to SW3-SW0, load to SW4, ShiftEn to SW5, ShiftIn to SW6, RegContent to LED3-LED0, and ShiftOut to LED7. Verify the design in hardware.



1-4-1. Open Vivado and create a blank project called lab6_1_4**.**

1-4-2. Create and add the VHDL module that will model the 4-bit parallel in left shift register using the provided code.

1-4-3. Develop a testbench and simulate the design.

1-4-4. Synthesize the design.

1-4-5. Create and add the XDC file, assigning *Clk* to ***SW15***, *ParallelIn* to ***SW3-SW0***, *load* to ***SW4***, *ShiftEn* to ***SW5***, *ShiftIn* to ***SW6***, *RegContent* to ***LED3-LED0***, and *ShiftOut* to ***LED7***.

1-4-6. Implement the design.

Look at the Project Summary and note the resources used. Understand the result.

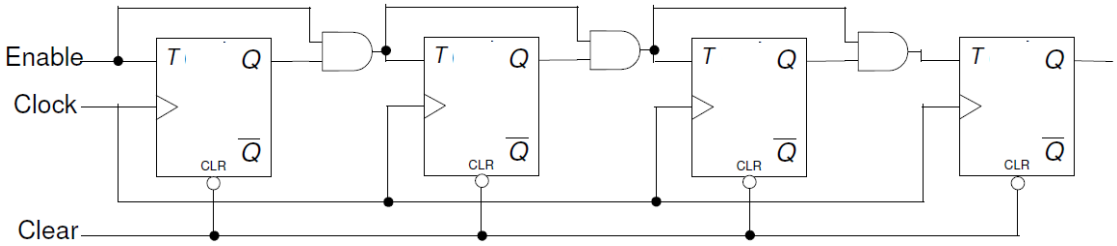
1-4-7. Generate the bitstream, download it into the Nexys4 board, and verify the functionality.

1-5. Write a model for a 4-bit serial in parallel out shift register. Develop a testbench and simulate the design. Assign Clk to SW15, ShiftEn to SW0, ShiftIn to SW1, ParallelOut to LED3-LED0, and ShiftOut to LED7. Verify the design in hardware.

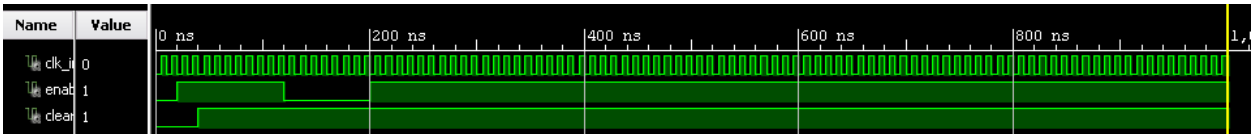
Counters Part 2

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 to SW15, Clear_n to SW0, Enable to SW1, and Q to LED7-LED0. Implement the design and verify the functionality in hardware.



2-1-1. Open Vivado and create a blank project called lab6_2_1.

2-1-2. Create and add the VHDL module to provide the desired functionality.

2-1-3. Synthesize the design and view the schematic under the Synthesized Design process group. Indicate what kind and how many resources are used..

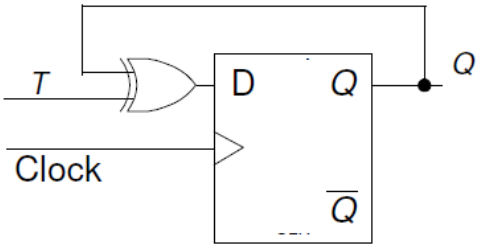
2-1-4. Develop a testbench and validate the design.

2-1-5. Create and add the XDC file, assigning *Clock* input to ***SW15***, *Clear_n* to ***SW0***, *Enable* to ***SW1***, and *Q* to ***LED7-LED0***.

2-1-6. Implement the design.

2-1-7. Generate the bitstream, download it into the Nexys4 board, and verify the functionality.

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 to SW15, Clear_n to SW0, Enable to SW1, and Q to LED7-LED0. Implement the design and verify the functionality in hardware.

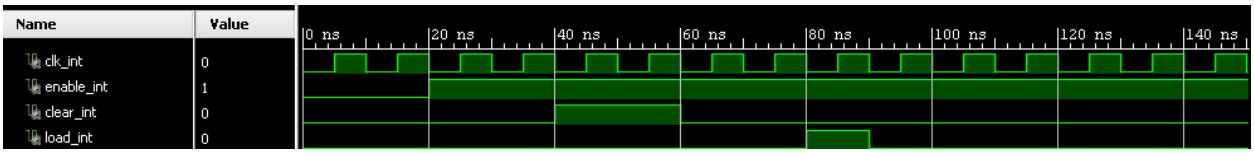
Other types of binary counters include (i) up, (ii) down, and (iii) up-down. Each one of them may have count enable and reset as control signals. There may be situation where you may want to start the count up/down from some non-zero value and stop when some other desired value is reached. Here is an 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.

```
…
signal count : std_logic_vector (3 downto 0);
signal cnt_done : std_logic;

…
cnt_done <= not(count(3) or count(2) or count(1) or count(0));
Q <= count;
if rising_edge(clk) then
      if (clear = '1') then
            count <= "0000";
      elsif (enable = '1') then
            if ((load or cnt_done) = '1') then
                  count <= "1010";
            else
                  count <= count - '1';  -- or you can use "0001"
            end if;
      end if;
end if;

…
```

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 to SW15, Clear to SW0, Enable to SW1, Load to SW2, and Q to LED3-LED0. Implement the design and verify the functionality in hardware.



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.