VHDL Process and FSM Tutorial

Purpose

The goal of this tutorial is to demonstrate how to design sequential circuits and finite state machines (FSMs) in VHDL through the use of a process statement. This tutorial accompanies Lab 6: Finite State Machines and VGA Controller.

Background Information

The most common way of implementing a clocked (sequential) circuit in VHDL is through the use of a VHDL process.

Unlike structural descriptions (i.e. using logical keywords like and, or, etc.), the process statement can be thought of as a purely behavioral description of a module’s functionality. When writing a process, you are no longer writing HDL that describes how a module is implemented using gates: you are describing how a module should behave, and you are leaving the conversion of this behavioral description into registers and gates up to the synthesis tool. There are pros and cons to this method of design. A behavioral description allows you to work at a higher level of abstraction, which can help to simplify larger, more complicated designs. At the same time, when you would like to implement a specific piece of hardware (for example, a single 65,536-by-two-byte block RAM), it can be difficult to get the synthesis tool to produce the exact results you would like. In cases like this, you usually need to adhere to template HDL snippets provided by the manufacturer that are guaranteed to synthesize in a specified manner.

Processes in VHDL are declared in the following manner:

```
process_name : process (sensitivity list)
begin
-- Process statements go here
end process;
```

There are three key components to this declaration. The first is the process name (process_name above), which is optional but is usually recommended so as to give some description of what the process does. The next is the sensitivity list, which is a comma-separated list of signals to which the process is sensitive: this means that whenever any one of these signals changes, the process is re-evaluated. You will usually see clock and reset signals in the sensitivity list for a process. The last part is the body of the process, which is the area between the begin and end process statements. This area will contain the behavioral description of the process.

For a simple example, consider the following process that computes the two’s complement of an eight-bit logic vector X, and assigns the result to Y.

```
negate : process (X)
begin
  Y <= not X + "00000001";
end process;
```
Because X is in the sensitivity list for this process, whenever the value of X changes, the process is re-evaluated and the value of Y is updated accordingly.

One of the main behavioral constructs that a process allows us to use is the if statement. This statement functions in exactly the same manner as most programming languages, and the syntax is provided below.

```vhdl
if (condition1) then
  -- Statements
elsif (condition2) then
  -- More statements
else
  -- Even more statements
end if;
```

If `condition1` evaluates to true, the first block of statements will be executed. If `condition2` evaluates to true, then the second block will be executed, and if neither is true, the last block will be executed. One thing to be careful of is that the standard “else if” keyword is combined to a single keyword `elsif` in VHDL (using `else if` instead will create a new if statement block inside the `else` portion of the first, which is functionally identical but requires an additional `end if` to compile).

The conditions for if statements in VHDL are of the “boolean” type. The result of equality (=) and inequality (<, <=, >, >=) operators, as well as logical (and, or, not, etc.) operators evaluate to a boolean type in this context, and can therefore be used as conditions.

In order to use a process to design sequential circuits, VHDL provides a signal attribute called an event. The event attribute evaluates to true whenever the signal is changing. One can use the event attribute with a clock signal, for example, to design processes that perform a specified action on the edge of a clock. See the following process for an example, which implements a positive edge-triggered D flip-flop with an asynchronous reset. Note that Q and D are declared as `STD_LOGIC` signals. The signal Q is the value stored in the flip-flop, while D is the input.

```vhdl
dff : process (clk, reset)
begin
  if (reset = '1') then
    -- Reset Q to 0 (asynchronous)
    Q <= '0';
  elsif (clk'event and clk = '1') then
    -- Update the value of Q on a rising edge
    Q <= D;
  end if;
end process;
```

This process has the `clk` (clock) and `reset` signals in the sensitivity list, and will therefore be re-evaluated whenever either of these signals changes value. If the `reset` signal is high, the value stored in the latch will be set to '0'. Observe that the condition in this if statement does not mention the clock, so this reset will happen regardless of the state of the clock. Therefore, the reset input here is asynchronous.
The condition for the `elsif` case is the most interesting. As discussed above, the value of `clk'event` evaluates to true when `clk` is on an edge. Combining this with the `clk = '1'` statement, the condition as a whole (`clk'event and clk = '1'`) will evaluate to true only on a rising edge. Thus, the statement inside this condition (that updates the value of Q to match the value of D) will happen only on a rising edge, and this design becomes edge-triggered.

**Tutorial Procedure**

The best way to learn to write your own FSMs in VHDL is to see an example. For the purposes of this tutorial, we will implement the color changing FSM used in Lab 6. The state transition diagram is shown in Figure 1.

![Figure 1: State Transition Diagram for Color Generation FSM](image)

**For the impatient, actions that you need to perform have key words in bold.**

1. With your Lab 6 project open in the Xilinx ISE, right click under the sources window and select **New Source**...

2. From the list of options given, select ”VHDL Module”. Name the new module appropriately, and select **Next**. See the image below for an example.
3. This module will need input connections to a clock signal (clk), a reset signal (reset), and a single one-bit input (X), and should have a two-bit output S. **Configure the Port Wizard** as seen in the following image.
4. **Continue** through the New Source Wizard and **create** the new ColorFSM module.

5. First, we’re going to **create a signal** to hold the current state of our FSM. For the purposes of this tutorial, let’s call this signal “state”, and **declare** it as a two-bit standard logic vector. See the following image for an example.

   ```vhdl
arbitrary architecture Behavioral of ColorFSM is

   signal state : STD_LOGIC_VECTOR(1 downto 0) := "00";

begin

Note that the value after the := assigns a default or starting value to this signal. Here we are assigning a starting value of zero.

6. Next, let’s **create a process** to implement our finite state machine. This process should be re-evaluated whenever the clock (clk) or reset signals change value, and therefore, clk and reset should be **placed in the sensitivity list**. See the following image for an example.

   ```vhdl
   Color_FSM : process (clk, reset)
   begin

   end process;

   Note that the signal X (the input to our FSM) is not being placed in the sensitivity list. This is because, in general, an FSM does not change state exactly when an input changes, it will wait for a clock edge to respond.

7. Next, let’s **add the reset condition** to our process. Recall from the discussion above that this is usually accomplished using an if statement. In the condition of that if statement, we will check to see if reset is ‘1’. If reset is ‘1’, we should set the state signal to the value for the reset state. Recall from the state diagram in Figure 1 that the reset state is S0. Assuming the default counting-order state assignment, this is state “00”. Therefore, state is assigned the value of “00”. See the following image for an example.

   ```vhdl
   Color_FSM : process (clk, reset)
   begin

   if (reset = '1') then
     state <= "00";
   end if;

   end process;

8. Next, let’s **add a clock edge condition** to our process. Since we want this FSM to change state on a rising clock edge, we can use the same condition as discussed above. See the following image for an example.
if (reset = '1') then  
   -- Reset state  
   state <= "00";
elsif (clk'event and clk = '1') then  
   -- Transitions will go inside here  
end if;

9. Next, it’s time to add the state transitions! The easiest way to implement this is as a nested if statement inside the clock edge condition we added in the previous step. To start simple, we will add the transition from S0 to S1 when X = ‘1’. To implement this, we will add an if statement inside the clock edge condition that checks two things: first, is the current state equal to S0 (“00”), and second, is the input X equal to the value for this transition (‘1’)? This can be implemented as follows:

   if (reset = '1') then  
      -- Reset state  
      state <= "00";
   elsif (clk'event and clk = '1') then  
      -- Transitions go inside here  
      if (state = "00" and X = '1') then  
         -- Transition to S1 from S0 when X is 1  
         state <= "01";
      end if;
   end if;

10. Finally, let’s add the rest of the state transitions in the same manner as the first one. For every transition edge in Figure 1, add a transition condition, inside which the state signal is updated to its new value. Once you’re done, the finished process should look like the following image:
Color_FSM : process (clk, reset)
begin
   if (reset = '1') then  
      -- Reset state  
      state <= "00";
   elsif (clk'event and clk = '1') then  
      -- Transitions go inside here  
      if (state = "00" and X = '1') then  
         state <= "01";
      elsif (state = "01" and X = '1') then  
         state <= "10";
      elsif (state = "10" and X = '1') then  
         state <= "11";
      elsif (state = "11" and X = '1') then  
         state <= "00";
      else  
         -- Do nothing, hold state  
      end if;
   end if;
end process;
11. To finish our module, we need to assign the value of the FSM output. For this FSM, the output is simply equal to the state variable, so all we need to do is assign state to the output S. This can be done outside of the Color_FSM process as shown in the image below.

    end process;

    S <= state;

    end Behavioral;

12. To verify that this module works as intended, you should conduct a test bench waveform simulation. Note that because we are now simulating a module with a clock, the “Initial Timing and Clock Wizard” will automatically detect that the input clk is our clock signal. Instead of choosing “Combinatorial” under “Clock Information”, we can leave this option set to “Single Clock”. Under the “Clock Timing Information” pane, set both the clock’s high and low time to 20 nanoseconds, and set the initial setup time and output valid delay to 10 nanoseconds each. Additionally, uncheck “GSR (FPGA)”. These changes are summarized in the following screenshot.
13. You should see that in the Test Bench Waveform that comes up, the clock signal is already configured for you. All you have to do is configure the X and reset inputs. You should set up an input pattern like the one shown in the following screenshot. Notice that the system is reset before any other inputs become 1.

14. **Simulate** the Test Bench Waveform, and **verify** that your FSM transitions from state to state as intended. Create a schematic symbol and take a screenshot of the simulation (and of the code) for your report. fc