

TUTORIAL ON USING XILINX ISE DESIGN SUITE 14.6: Modeling State Machines for Spartan-6 (NEXYS 3 Board)

Shawki Areibi

January 11, 2016

1 Introduction

The examples that have been discussed so far were combinational and sequential circuits in isolation. Processes that model combinational circuits are sensitive to the inputs, being activated whenever an event occurs on an input signal. In contrast, sequential circuits retain information stored in internal devices such as flip flops and latches. In this tutorial we will illustrate how finite state machines can be modeled using the process construct.

2 A General Finite State Machine

Figure 1 shows a general model of a finite state machine. The circuit consists of a combinational component and a sequential component. The sequential component consists of memory elements such as edge-triggered

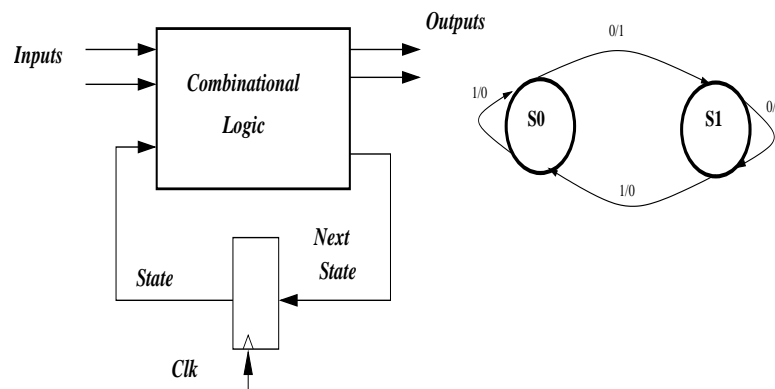


Figure 1: A Behavioral Model of a Finite State Machine.

flip-flops, that record the state and are updated synchronously on the rising edge of the clock signal. The combinational component is made up of logic gates that compute two Boolean functions. The *output function* computes the values of the output signals. the *next-state* function computes the new values of the memory elements (i.e. the value of the next state).

From the state diagram in Figure 1 we know that if the machine is in state S_0 and the input signal has value of '0', then the machine transitions to state S_1 while setting the value of the output signal to '1'. The behavior of the state machine can be described in a similar manner.

3 Implementations: Mealy versus Moore FSMs

Finite State Machines can be classified as either **Mealy** or **Moore**. These two kinds of FSMs differ in how their outputs are computed. Figure 2 shows the two state machines.

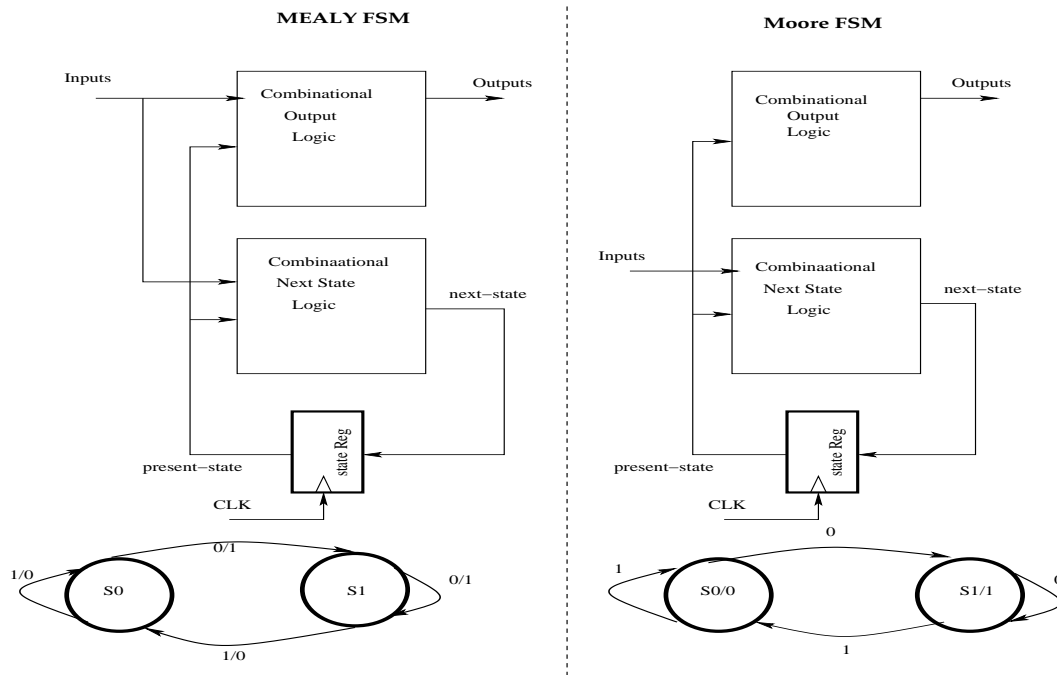


Figure 2: Mealy FSM versus Moore FSM

3.1 Mealy FSMs

A Mealy FSM's outputs are a function of both inputs and its present state. Because of this, **the outputs of a Mealy FSM can change as soon as any of its inputs change**. However, its state still cannot change until a triggering clock edge. Implementations #1, #2 in this tutorial demonstrate how we can realize a Mealy FSM.

3.2 Moore FSMs

A Moore FSM's output are a function of only its present state. The block diagram of the Mealy FSM in Figure 2 (on the left hand side) is modified to represent a Moore FSM (right hand side of the figure) by removing all external inputs to the output logic. It is clear from this diagram that since a Moore FSM's outputs are a function of only its present state, they can **change only at a triggering clock edge**. Implementation #3 in this tutorial demonstrates how we can realize a Moore FSM.

3.3 Mealy and Moore Functional Equivalence

In general, any sequential function can be implemented by either a Mealy FSM or a Moore FSM. However, while a Moore FSM is often conceptually simpler, it usually requires more states than a Mealy FSM to implement the same function. The example in this tutorial we will require the same number of states for both Machines. Because a Mealy FSM's outputs are a function of its inputs and state, an output can have different values in a single state.

3.4 Implementation #1: ‘Mealy FSM’ using Two Processes

Figure 1 suggests a VHDL implementation using communicating concurrent processes. The combinational component can be implemented within one process. This process is sensitive to events on the input signals and changes in the `present_state`. Thus, if any of the input signals or the `present_state` variables change value, the process is executed to compute new values for the output signals and the new state variables.

The sequential component can be implemented within a second process. This process is sensitive to the rising edge of the clock signal. When it is executed, the state variables are updated to reflect the value of the next state computed by the combinational component. The VHDL description of such a state machine is shown in Implementation #1. The model is structured as *two* communicating processes, with the signals **`present_state`** and **`next_state`** used to communicate values between them.

It is important to notice the following from the model described in Implementation #1:

1. The process can be naturally extended to state machines composed of a larger number of states. The behavior of each additional state can be described as an additional branch of the case statement in the process `comb_process`.
2. The signal **`next_state`** is changing with the input signal, whereas the signal **`present_state`** is not. This is because **`present_state`** is updated only on the rising clock edge, while **`next_state`** changes when ever the input signal `x` changes.

```
-- Implementation #1 (Two processes to implement the Mealy FSM)
library IEEE;
use IEEE.std_logic_1164.all;

entity state_machine is
port (reset, clk, x: in std_logic;
      z: out std_logic);
end entity state_machine;

architecture behavioral of state_machine is
    type statetype is (state0, state1);
    signal present_state, next_state: statetype:= state0;
begin
    comb_process: process(present_state,x) is
    begin
        case present_state is           -- depending upon the current state
            when state0 =>              -- set output signals and next state
                if x = '0' then
                    next_state <= state1;
                    z <= '1';
                else
                    next_state <= state0;
                    z <= '0';
                end if;
            when state1 =>
                if x = '1' then
                    next_state <= state0;
                    z <= '0';
                else
                    next_state <= state1;
```

```

        z <= '1';
    end if;
end case;
end process comb_process;

clk_process: process is
begin
    wait until (rising_egdge(clk));    -- wait until the rising edge
    if reset = '1' then                -- check for reset and initialize state
        present_state <= statetype'left;
    else
        present_state <= next_state;
    end if;
end process clk_process;
end architecture behavioral;

```

3.5 Implementation #2: ‘Mealy FSM’ using Three Processes

We could just easily have written the preceding example as a single process whose execution is initiated by the clock edge. In that case, the computation of the next state, the output signals, and the state transition are all synchronized by the clock.

Alternatively, we could construct a model in which outputs are computed asynchronously with the computation of the next state. Such a model is shown in Implementation #2. This model is constructed with three process: one each for the output function, **next_state** function, and the **present_state** transition.

It is important to note that the **output_process** has the **present_state** and inputs in the sensitivity list of the process. Note how the model is constructed from the structure of the hardware: Concurrency in the circuit naturally appears as multiple concurrent processes in the VHDL model. State machines that compute the output signal values only as a function of the current state are referred to as Moore machines. State machines that compute the output values as a function of both the current state and the input values are Mealy machines.

```
-- Implementation #2
-- Three processes to implement state machine
library IEEE;
use IEEE.std_logic_1164.all;

entity state_machine is
port (reset, clk, x: in std_logic;
      z: out std_logic);
end entity state_machine;

architecture behavioral of state_machine is
    type statetype is (state0, state1);
    signal present_state, next_state: statetype:= state0;

begin

output_process: process(present_state,x) is
begin
    case present_state is          -- depending upon the current state and input
        when state0 =>            -- set output signals
            if x = '1' then
                z <= '0';
            else
                z <= '1';
            end if;
        when state1 =>
            if x = '1' then
                z <= '0';
            else
                z <= '1';
            end if;
        end case;
    end process output_process;

next_state_process: process(present_state,x) is
begin
```

```

case present_state is          -- depending upon the current state and input
  when state0 =>               -- set next state
    if x = '1' then
      next_state <= state0;
    else
      next_state <= state1;
    end if;
  when state1 =>
    if x = '1' then
      next_state <= state0;
    else
      next_state <= state1;
    end if;
end case;
end process next_state_process;

clk_process: process is
begin
  wait until (rising_egdge(clk)); -- wait until the rising edge
  if reset = '1' then             -- check for reset and initialize state
    present_state <= statetype'left;
  else
    present_state <= next_state;
  end if;
end process clk_process;
end architecture behavioral;

```

3.6 Implementation #3: ‘Moore FSM’ using Three Processes

Comparing implementation #2 (for the Mealy Machine) and implementation #3 below (for the Moore Machine) we notice that they are similar. **However**, it is important to note that the **output_process** of the Moore FSM has only the present_state in the sensitivity list of the process.

```
-- Implementation #3
-- Three processes to implement Moore FSM
library IEEE;
use IEEE.std_logic_1164.all;

entity state_machine is
port (reset, clk, x: in std_logic;
      z: out std_logic);
end entity state_machine;

architecture behavioral of state_machine is
    type statetype is (state0, state1);
    signal present_state, next_state: statetype:= state0;
begin

    output_process: process(present_state) is
    begin
        case present_state is           -- depending upon the current state only
            when state0 =>              -- set output signals
                z <= '1';
            when state1 =>
                z <= '0';
        end case;
    end process output_process;

    next_state_process: process(present_state,x) is
    begin
        case present_state is          -- depending upon the current state and input
            when state0 =>              -- set next state
                if x = '1' then
                    next_state <= state0;
                else
                    next_state <= state1;
                end if;
            when state1 =>
                if x = '1' then
                    next_state <= state0;
                else
                    next_state <= state1;
                end if;
        end case;
    end process next_state_process;

    clk_process: process is
```

```

begin
    wait until (rising_edge(clk));    -- wait until the rising edge
    if reset = '1' then               -- check for reset and initialize state
        present_state <= statetype'left;
    else
        present_state <= next_state;
    end if;
end process clk_process;
end architecture behavioral;

```


4 Implementation

The following is a simple User File Constraint (UFC) that connects any of the three implementations to the I/O (LEDs, Switches, Push Buttons) of the NEXYS 3 Board.




```
NET reset LOC = T5;    # Reset is Switch 7
NET clk LOC = D9;      # Clk is Push Button BTNR
NET x LOC = T10;       # Input x is Switch 0
NET z LOC = T11;       # Output z is LED 7
NET s0 LOC = U16;      # State 0 indicator is LED 0
NET s1 LOC = V16;      # State 1 indicator is LED 1
```

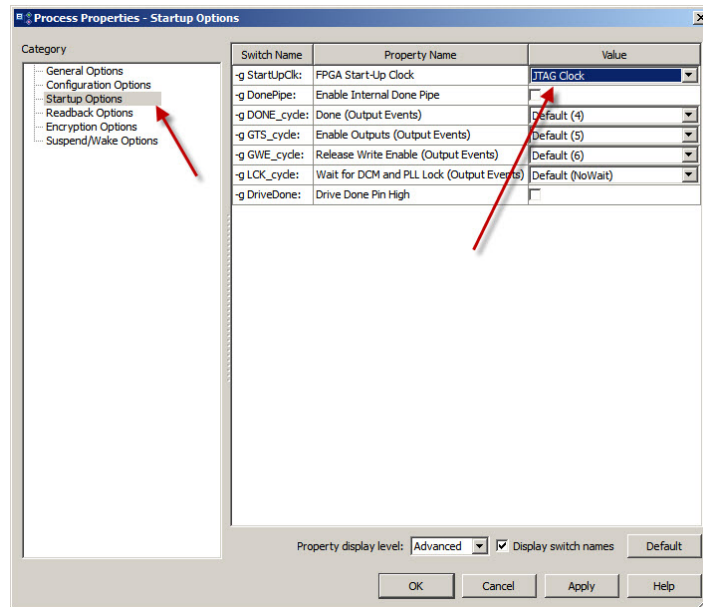
1. Create a new project. Make sure that the top level design is VHDL.
2. Enter the VHDL code for each implementation of the state machine. You may copy the file associated with the tutorial.
3. Simulate the design.
4. Synthesize, Implement the design.
5. Generate a bit-stream and map onto the FPGA.

Make sure you understand the different implementations and how they differ in terms of their output as the clock is applied to the Finite State Machine (FSM).


5 Generating a Bit Stream “Compiling the Design”

Hitherto, we have examined how to design a digital circuit using the Xilinx ISE Design Suite 14.6 Software. We will now look at how to compile (generate the bit-stream) and download the design to the Digilent NEXYS 3 board.

1. Now go back to the Project Navigator window. Highlight your main design **VHDL or Schematic** in the **Hierarchy** pane. Left double click  **Synthesize - XST** in the **Processes** pane to synthesize the design.
2. When the synthesis stage has completed, you will see the following message on the console panel (process “synthesis – xst” completed successfully).
3. Left double click  **Implement Design** to implement the design.
4. When the implement stage has completed successfully you will see the following message on the console panel (process “Generate Post-Place & Route static Timing” completed successfully).
5. Right click on  **Generate Programming File** and choose properties. A window will pop as seen below:



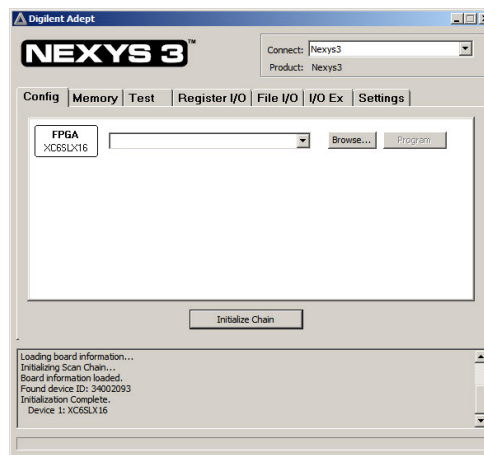
Highlight startup options and change the switch name “-g StartUpClk” value to JTAG clock. Press **Apply** then **OK**.

6. Finally, left double click  **Generate Programming File** to generate the programming file that will be downloaded to the FPGA. It is possible to go straight to generating the programming file, and the Project Navigator will determine which of the previous step need to be run to produce an up to date programming file.
7. When the generation of the bit-stream stage has completed successfully you will see the following message on the console panel (process “Generate Programming File” completed successfully).

6 Downloading the design to the FPGA

Downloading a bit-stream to the FPGA is accomplished via either the **iMPACT** tool within the ISE Project Navigator or the Digilent **Adept** tool for NEXYS 3 board.


1. Steps to download the design using the **iMPACT** tool within the ISE Project Navigator:
 - (a) Select “Configure Target Device”. **Double click** and a new window will appear stating that “No iMPACT project file exists”. You can ignore the message and **press OK**.
 - (b) A new window will then appear (ISE iMPACT).
 - (c) **Double click** on the Boundary Scan icon  to start the ISE iMPACT tool.
 - (d) Click on the boundary scan box in the top menu “Initialize Chain”. Make sure that your FPGA is powered on and connected to the host. A small window will appear with the message “Do you want to continue and assign the configuration files(s)?”. **Press yes**.
 - (e) A new window will appear (Assign New Configuration File). Here you will zoom to the directory where your *.bit file exists.
 - (f) **Double click** on the file name.
 - (g) A Message window titled **Attach SPI or BPI PROM** will appear. **Press Yes**.
 - (h) A new window titled “Add Prom File” will appear. Just simply **close** the window.
 - (i) A message box will appear (Device Programming Properties). **Click OK**.
 - (j) Move the cursor over the device that appears in the **Boundary-Scan** tab and right click the mouse button. A menu will appear. **Press** the left mouse button and select **Program**.
 - (k) If the programming succeeds you will see the following message “Program Succeeded”.
2. Steps to download the design using the Digilent **Adept** tool:
 - (a) Load the Digilent Adept from the  → **All Programs** → **Digilent** → **Adept** → .
 - (b) The Digilent Adept window will appear as seen in the Figure below.
 - (c) Click the **Browse** icon. A new window will appear to choose your bit file.
 - (d) Zoom onto the directory where your bit file resides and double click it.
 - (e) Click the **Program** button.
 - (f) The system will start to program the device and at the bottom of the Adept Tool you will see some messages indicating that it has successfully programmed the device.

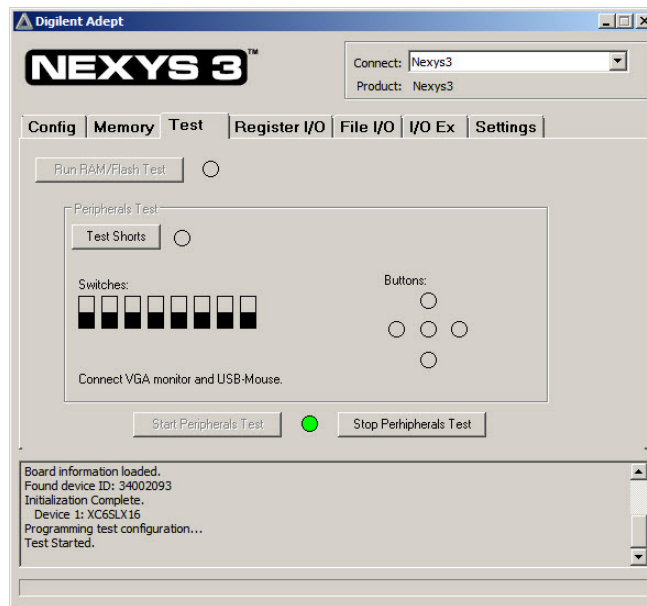


7 Appendix A - Setting-up and Testing the NEXYS3 board

This is intended to allow the student to quickly set up the NEXYS 3 board for this tutorial. It does not attempt to explain the configuration and is in no way a substitute for the documentation provided with the board. It will allow you to use the slide switches as input and the LEDs as outputs.

1. Connect the USB cable to the NEXYS 3 board.
2. Connect the host computer to your USB cable.
3. When the power switch to the board is on a small yellow LED labeled *Done* should glow.

You can test if the Digilent NEXYS3 Board is operational by using the Digilent Adept Tool. Double click on the  icon and you will see the Digilent Adept GUI on your screen. Press the **Test** icon. A new menu will appear. Press the “Start Peripherals Test”.



The test will display different values on the 7-segment display. You can also test the switches and light emitting diodes by sliding the switches to the on-off position. Once a switch is turned on the corresponding LED will glow. You will also notice that the switches on the Digilent Adept tool will change value. You can also test the push buttons by pressing on them. You will see the color of the corresponding button on the Adept tool change from transparent to black. Once you are satisfied that the FPGA board is operational you can press the “Stop Peripherals Test”. By pressing the “Reset Button” on the FPGA you will reset the board to the factory setting where it tests all other modules on the PCB board. Power off the board using the slide switch found at the top left part of the board.

8 Appendix B - LEDs, 7-Segments and Switches

The following sections explain the connection and location of the DIP switches and LEDs of the Digilent NEXYS 3 Board.

8.1 LEDs

The Digilent NEXYS 3 Board provides a series of eight LEDs (LD0–LD7) for use. All of these LEDs are **Logic Active High** meaning that an LED segment will glow when a logic-high is applied to it. The following table show the connection from the NEXYS 3 Board to LEDs expressed as UCF constraints.

—Description	—Location
NET LD0	LOC=U16
NET LD1	LOC=V16
NET LD2	LOC=U15
NET LD3	LOC=V15
NET LD4	LOC=M11
NET LD5	LOC=N11
NET LD6	LOC=R11
NET LD7	LOC=T11

Table 1: NEXYS 3 (Light Emitting Diodes) LEDs

8.2 Seven Segment Displays

The Digilent NEXYS 3 Board provides four multiplexed 7-segment displays for use. The following tables show the connection from the NEXYS 3 Board to the 7-segment displays expressed as UCF constraints.

—Description	—Location
NET CA	LOC=T17;
NET CB	LOC=T18;
NET CC	LOC=U17;
NET CD	LOC=U18;
NET CE	LOC=M14;
NET CF	LOC=N14;
NET CG	LOC=L14;
NET DP	LOC=M13;
NET AN0	LOC=N16;
NET AN1	LOC=N15;
NET AN2	LOC=P19;
NET AN3	LOC=P17

Table 2: NEXYS 3 (7-Segment display)

8.3 Slide Switches

The Digilent NEXYS 3 board has a bank of eight slide switches which are accessible by the user.

When closed or ON, each DIP switch pulls the connected pin of the NEXYS 3 Board to ground. When the DIP switch is open or OFF, the pin is pulled high through a $10K\Omega$ resistor.

The table below shows the connections from the Digilent NEXYS 3 Board to the switches expressed as UCF constraints.

—Description	—Location
NET SW0	LOC=T10
NET SW1	LOC=T9
NET SW2	LOC=V9
NET SW3	LOC=M8
NET SW4	LOC=N8
NET SW5	LOC=U8
NET SW6	LOC=V8
NET SW7	LOC=T5

Table 3: NEXYS 3 (Slide Switches)

8.4 Push Buttons

The Digilent NEXYS 3 board has five pushbuttons (labeled BTNS through BTNR) which are accessible by the user.

When pressed, each pushbutton pulls the connected pin of the NEXYS 3 Board to ground. Otherwise, the pin is pulled high through a $10K\Omega$ resistor. The table below shows the connections from the the Digilent NEXYS 3 Board to the push buttons expressed as UCF constraints.

—Description	—Location
NET BTNS	LOC=B8
NET BTNU	LOC=A8
NET BTNL	LOC=C4
NET BTND	LOC=C9
NET BTNR	LOC=D9

Table 4: NEXYS 3 (Pushbuttons)