Australian National University

Lab Four

Finite State Machines

Overview

Finite state machines (FSMs) are used to control the sequential execution of functions based on a system's *state* and *inputs*. In this lab you will learn to implement FSMs on your FPGA boards to control the flow of your application.

In this lab you will

- Design a FSM to mimic the operation of a Microwave oven's control system.
- Design your own state machine
- Implement the FSMs in Verilog code and test the working using inputs and outputs available on your FPGA board

Best of luck and Let's have some fun!

Australian National University

Table of Contents

Australian National University

Lab Overview

Learning Outcomes

By completing this lab, you will learn to:

1. Instantiate modules in hierarchical designs
2. Use a combination of resets, enables and finite state machines to control complex sequential logic processes
3. Design a finite state machine for a given set of inputs and outputs

What's needed?

To complete this lab, you'll require access to:

1. Vivado Design Suite running on Windows 10/11 or a supported Linux operating system
2. Basys3 Artix-7 FPGA Development Board

Mark Allocation

1. Pre-Lab Task [1 Bonus Mark]
2. Activity 1 [10 Marks]
3. Activity 2 [10 Marks]

In total: 20 Marks + 1 Bonus Mark

Recommended Time Allocation

1. 1 hr. for Activity 1
2. 2 hr. for Activity 2

Australian National University

Pre-Lab Task:

Please refer to the lab document (below) to complete the following pre-lab tasks.

Activity 1: Microwave Control System

Task 1:

What is the difference between Moore and Mealy finite state machines? Also, briefly describe their advantages and disadvantages. **[0.25 Marks]**

Task 2:

Based on Fig.2 (System Diagram), complete the following state transition table. Write down the next state for each scenario at each entry. If the input is undefined, leave the entry empty. **[0.25 Marks]**

	Enter = 0	**Enter =1**	**Counter overflow =1**	**Counter overflow = 0**	**Reset**
IDLE	IDLE	SET_TIMER	X	X	IDLE
SET_TIMER					
COOK					
ALARM					

Activity 2: Help the blind man find his way to the Vaccine.

Task 3:

Draw state transition diagram for Activity 2. Please refer to Fig.2 as an example. **[0.25 Marks]**

Task 4:

Design and complete your state transition table for Activity 2. You can refer to Task 2 as an example. **[0.25 Marks]**

Activity 1: Microwave Control System
(Approx. 1hr) (10 Marks)

A Finite State Machine is a tool used to control the flow and execution sequence of your system. A Finite State Machine divides the control sequence of an application into discrete, interconnected states. These states determine which stage of operation we are in, based on which the same inputs may have different effects/outputs. A Finite State Machine is described using a state diagram and a state transition table.

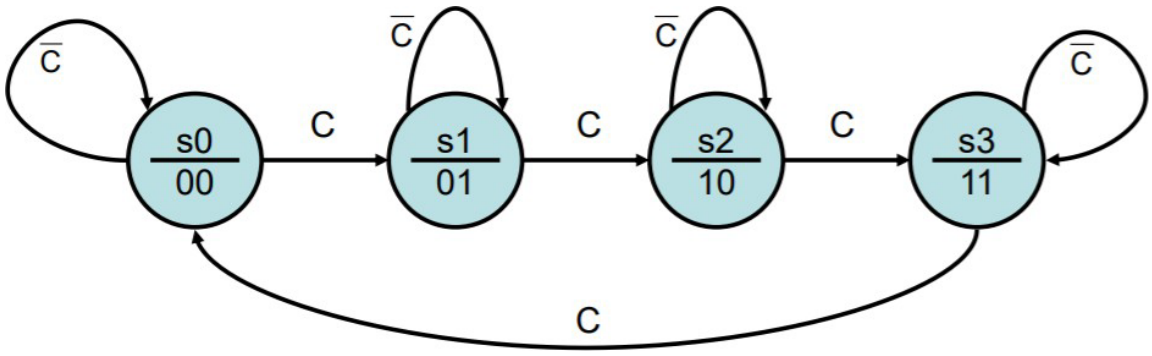

Figure 1: Example of a State Diagram for a Moore Machine (from the lecture slides)

FSMs can be of two main types, Moore and Mealy Machines. Both types have their own pros and cons. An example of a Moore machine state diagram is shown in Figure 1. Moreover, the same system may be described as both types. Which type we choose depends on the advantages and disadvantages of the decision.

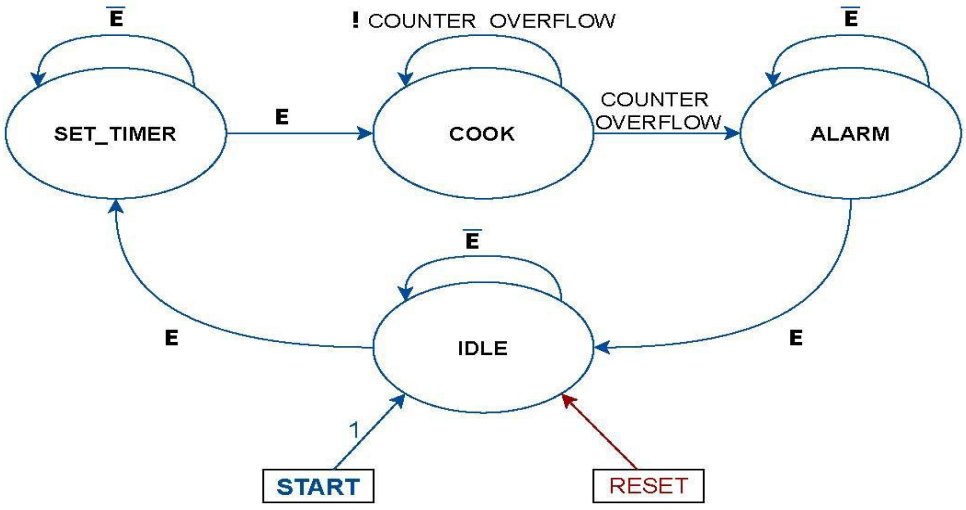

Figure 2: System Diagram

In the following Activity, you will design a State Machine to operate as the control system of a microwave oven. Figure 2 shows the system control sequence overview.

We shall use the LEDs as the outputs and the switches and push buttons as inputs. The system tries to mimic the working of a Microwave oven using the inputs and outputs available on our FPGA board. Thus, the system may be summed up as follows:

- **IDLE**: The system waits for user input. Press enter to start.
- **SET_TIMER**: Use the switches to set the timer by turning on the switch number corresponding to the number of seconds. E.g., Set *SW[7]* to wait 7 seconds. Only the MSB of *SW* is used, all the rest are ignored. Press enter to start cook.
- **COOK**: Wait until timer has elapsed
- **ALARM**: Blink LEDs to indicate cook complete. Press enter to restart.

Procedure

Here is how we will proceed.

Step 1: Create a Vivado project for this Activity

1. Create a new Project in Vivado. Call it '*MicrowaveFSM*'. We shall be using this project for the first activity.
2. Create a top-level module called '*MicrowaveFSM_top.v*'. This module will be used to connect all the different parts of the project. This module is important as it lets us pre-process input signals (Debounce/SPOT etc.) before passing them to the FSM. The project uses the usual inputs like `clk`, which accepts a 100 MHz external clock and `reset` connected to a push button. I'm sure you are familiar with their operation by now. This activity has 2 inputs, the Switch array and an Enter Key (called **E** in the state diagram) which is connected to a push-button.

   The output of this circuit drives the set of LEDs on the FPGA board.

```
module MicrowaveFSM_top (
      input wire clk,
      input wire reset,
      input wire enter,
      input wire [15:0] SW,
      output wire [15:0] LED
);
endmodule
```

3. We will use this module in the later stages but first let's start work on our Finite State Machine.

Step 2: Create a FSM module inside your project

1. Create another source file called '*FSM.v*'. This module will have the same ports as our top module.

2. In the FSM module, declare the output as a `reg` entity so we may drive this output inside a procedural block.
3. Before you start writing the code for the FSM, look at the state diagram shown in Figure 2 and assign state names (IDLE, COOK, etc.) to the state encoding values using *parameters*. This makes it easier to refer to the states in the Verilog code.

Step 3: Declare state variables and assign values to each state

1. Depending on the number of states calculate the number of bits required to represent all states. In our case, we have 4 states. Therefore, we shall use 2 bits to represent each state.
2. Assign bit values to each state and declare them as parameters inside your FSM module.

```
parameter IDLE=2'd0,COOK=2'd1,SET_TIMER=2'd2,ALARM=2'd3;
```

Step 4: Describe the Current State Register/ Transition block

1. A Finite State Machine consists of 3 main parts. The Nextstate Logic, the Output Logic and the Current State Register/Transition block.
2. The Current State Register/Transition block is the simplest of the three. It simply stores the current state of the system and loads the appropriate next state at every clock cycle. As you would have noticed, we declared two registers of data-width large enough to store our current and next state.

```
reg [1:0] state, nextstate;
```

3. Describe a procedural block that updates at every clock cycle and loads the contents of `nextstate` in `state`. Make sure that the `reset` signal is able to return the system to a known state/safe state.

Step 5: Next state Logic

1. The next state logic is crucial in controlling the sequence of events and states to match the required output logic.
2. Next state logic depends on the inputs of the system and the current state.
3. Describe the next state logic as a different procedural block. **Remember:** The next state logic *need not* to be synchronized with `clk`.

4. The next state logic is best described using combinatorial logic through a case statement.

```
always@(*)begin
      case(state)
          IDLE: begin
             if(enter) nextstate= SET_TIMER;
             else nextstate=IDLE;
           end
          SET_TIMER: begin
              ...
              ...
     endcase
```

5. The next state logic depends on the inputs and current state. Refer to the State diagram for more information. If you get stuck on the COOK state, refer to Step 6.

Step 6: Cook State logic

As you will have noticed, the system waits during the cook state until a timer runs out before proceeding to the next state, regardless of input. In this case we shall use a heartbeat generator to count the number of seconds.

1. Add a source for the *heartbeat* generator. You may use your *hearbeat* module from previous labs.
2. Instantiate this module to provide a beat every second by setting the right *THRESHOLD* value.
3. Now we need a counter to count down the number of seconds. A very easy and sleek way of doing this is to reuse the `LED` register to store the timer value set during SET_TIMER.
4. Instead of counting down the seconds using another module, we can simply right shift the `LED` register by 1 at every beat signal.

```
if(beat) LED <= LED >> 1;
```

5. Once the MSB is removed the correct time will have elapsed and we can move to the next state. Make sure you understand how we check if the `LED` register has all zeros. If not, contact your tutor.

```
if(&(~LED)) nextstate=ALARM;
```

Step 7: Output Logic

1. The output logic is the most important block in any FSM. This block dictates the behaviour of the system that is controlled by the FSM.
2. Based on your knowledge of how the system works or should work, describe another procedural block containing a case statement to drive your outputs.

```
always@(*)begin //should it be posedge clk?? read below
      case(state)
          IDLE: begin
             //Outputs for State IDLE
          end
          SET_TIMER: begin
              ...
     endcase
```

3. In our system, we can set all LEDs high to indicate READY/IDLE state. In state SET_TIMER we can connect `LED` to `SW` input directly to show the time SET.
4. In state COOK we shift the `LED` right at every beat signal. Since the `LED`s are also our output, this serves to indicate the amount of time left. In other cases, e.g., using the seven segment displays, we may need additional statements in our output logic for this state.
5. In the last state, ALARM, we flash LEDs to indicate that the COOK is completed. Here, we may either reuse the beat signal to toggle `LED`, or we could use a different clock divider and set `LED` high for half the time and low the other half. You can control the frequency of flashing by changing the THRESHOLD or TOPCOUNT of your clock divider/heartbeat module.

Important thinking point: In class you have learned that a FSM's output logic should be coded as combinational. However, in this design `LED` (the output) behaves as a shift register (or, if you like, a *one-hot down-counter with* `enable` *and* `reset`) and therefore must be a sequential component. How can this be?

The answer is that, strictly speaking, the `LED` shift register is not part of the microwave controller FSM as described in the state diagram of Figure 2 (the state diagram does not describe the counting steps!). It is actually a separate module of its own, which the main FSM controls.

To help you understand, compare the two snippets of code below, the first explicitly identifies the external counter logic (a `reg` called `ledcount` is created for the purpose),

```
always@(*)begin // FSM combinational output logic
      case(state) begin
            ...
            SET_TIME: LED = SW;
            COOK: LED = ledcount;
            ...
      endcase
end

always@(posedge clk)begin //counter logic - sequential
      if(state==SET_TIME)  ledcount <= SW; //reset
      else if(beat) ledcount <= ledcount >> 1; //o-h downcounter
end
```

the second bundles the counter in with the output logic

```
always@(posedge clk)begin //sequential "bundled" output logic
     case(state) begin
          ...
         SET_TIME: LED <= SW
         COOK: if(beat) LED <= LED >> 1;
          ...
    endcase
```

Now, since the only purpose of this FSM is to run the LEDs, the first version of the code – though formally "neater" – may be seen as unnecessarily laboured for this simple design. The two code options are largely equivalent and the bundled option is acceptable in this case where there is a very low risk of confusion in interpreting the structure of the system.

You can use either style for your coding today. As an exercise, try to draw the schematic for your Verilog in each case, and you might notice that the first one has a much more intuitive translation. That's why it is recommended that you keep to the template for more complex designs!

Step 8: Module interconnect in TOP module

The top is module is very useful in assembling all the different blocks/components of the system and connecting them appropriately. Here, we create instances of various modules that we have described and join them together using wires.

We also use the top module to pre-process input signals to

- Reduce noise
- Sample inputs at the right sampling frequency
- Buffer values for bulk processing
- Sync various asynchronous inputs

In this case we shall use the top module to

1. Debounce the enter button
2. Use Single Pulse on Transition (SPOT) on the debounced signal

After this, we can pass the pre-processed enter signal and other raw inputs to the FSM. Similarly, we can link the outputs from the FSM to the outputs of the TOP module.

Step 9: Constraints file for your TOP module

Write constraints for your system based on the ports in your TOP module.

You may copy these from your previous labs and change the port names to match based on your inputs and outputs. Alternatively, you may find the master XDC file for your board here:

Basys-3: https://github.com/Digilent/digilent-xdc/blob/master/Basys-3-Master.xdc

Step 10: Synthesize – Implement – Generate Bitstream

1. Now that all the pieces of your project are in place, synthesize your project. Make sure you check the messages window for any Errors or Critical Warnings. Debug your code.
2. Run Implementation and Generate Bitstream.
3. Upload your code to the board using Hardware Manager
4. Test to make sure it works correctly

Congratulations! You now know how to Code and Implement a Finite State Machine. Based on your knowledge of the theory of Moore and Mealy machines, can you identify if this is a Moore machine or a Mealy machine?

Please show your work to the tutor.

Australian National University

Activity 2: Design your own State Machine
(Approx. 1-2 hrs - 10 Marks)

This activity is aimed at helping you get comfortable designing and implementing your own State Machine. This skill will be especially handy in the next lab and the assignment when we expect you to write your own code from scratch. Since this is the first time, you'll be writing your own code, we thought we'd make this activity a little fun.

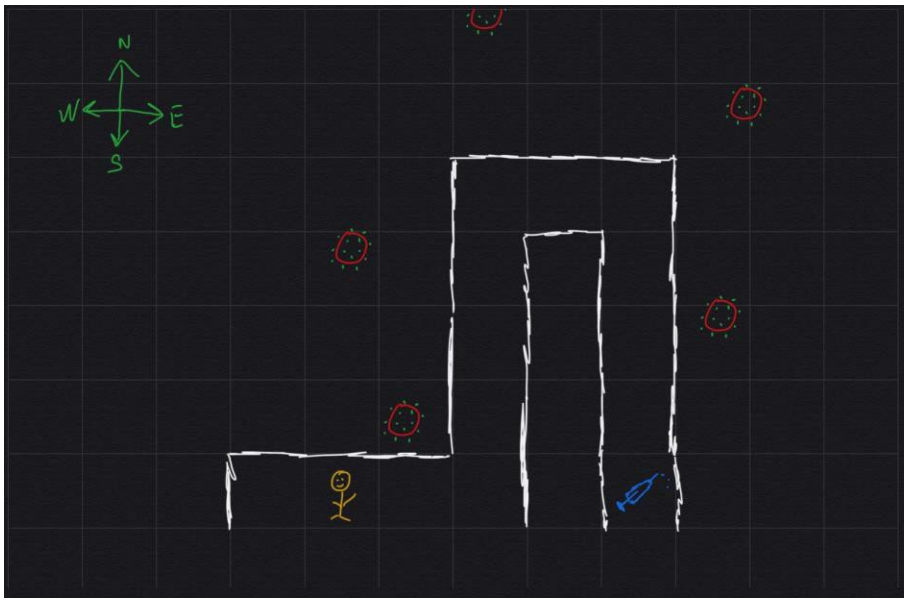

Figure 3: Map for Activity 2. Help the Blind man get to the Coronavirus Vaccine!

Your job is to help the blind man find his way to the Coronavirus vaccine. However, there are so many ways and viable options to get there. The path is also full of viruses and other dangers that can ruin our chances of a cure. Since the path is dark and full of terrors, he needs a device that can help him test his steps and plan his path to the vaccine. For your convenience, the correct path is highlighted with white borders. The device will accept a sequence of intended steps and, at the end, will deliver a response as to whether the entered path was correct by comparing it to a known correct sequence.

The circuit will have the usual inputs, `clk` and `reset`, along with the 4 pushbuttons for the 4 directions. `Btn_N`, `btn_S`, `btn_E` and `btn_W`.

Note: button west is unused so you may ignore it if you wish. One click of the button corresponds to 2 moves on the map. Eg. `Btn_E` takes you two boxes to the right. `Btn_N` takes you two boxes up as indicated in the top left corner.

This activity requires you to design and implement a state machine that can do the following:

- Receive a sequence of 6 pushbutton inputs. For every received input, a LED will be lit on the board to confirm the current count of entries.
- Once 6 button presses are received, compare the sequence with a known correct sequence (in our case, the correct combination is {E,N,N,E,S,S})

- Notify the user as to whether the sequence was correct or incorrect by either lighting up all LEDs (correct) or every second LED (incorrect).
- Be ready to receive a new sequence after a reset (centre button) event

There are a few possible different ways to implement this design. The tips that follow guide you through one possible design.

1. Draw a state diagram that has an IDLE state, 6 states to accept button inputs and a READY state to display the result.
2. Label all state transitions. What are the trigger signals for your system?
3. What output corresponds to each state?
4. A possible hardware implementation of this design involves the use of a SIPO shift register to cumulatively store the sequence as it is entered one button press at a time, and a comparator to compare (==) the *entered_steps* with the *correct_steps.* How do these relate to the state machine (i.e., what signals do they exchange, and what is their role in relation to inputs / transition triggers / outputs for the state machine)?
5. Do you need to condition your inputs (debouncer, spot, etc.) in any way for things to work properly?

Once you have an idea of what your system should look like, draw a block diagram sketch to consolidate your concept and get coding!

Some useful code snippets are below:

Parameter assignments make for more intuitive coding of the correct sequence:

```
parameter N=2'b10,S=2'b01,W=2'b00,E=2'b11;
wire [11:0] correct_steps;
assign correct_steps= {E,N,N,E,S,S};
```

Also, here are the port declarations for the fsm module:

```
module fsm(
    input clk,
    input reset,
    input btn_N,
    input btn_S,
    input btn_E,
    input btn_W,
    output reg [15:0] LED);
```

A good design is flexible. Once you have completed your design, as a test of flexibility of your code, ask yourself: could you change your system to work fine with a different correct sequence just by changing the one line `assign correct_steps= {E,N,N,E,S,S};`?

Or would you need to edit your code more substantially?

You are not required to change your code if you have completed a functioning implementation, however, I invite you to reflect on how different approaches to coding the FSM may lead to varying degrees of maintainability of the design.

Code, implement and run your design.

Please show your work to the tutor.

Congratulations! This task was not easy as you had to prepare all code on your own. If you have successfully completed it without too much handholding by the tutors, you are starting to demonstrate independence in the conception of digital designs, which is a key ability for the course and one that will come in very handy for both Lab 5 and the assignment.