ECE241 - Digital Systems

University of Toronto

Lab #6 - Fall 2008

Finite State Machines and Pre-designed Cores

1 Introduction

Finite State Machines (FSMs) are digital circuits that are use to control *what happens* in a digital circuit (typically by controlling enable and reset signals on registers) and *when* it happens (during which clock period). The purpose of this lab is to gain experience working with finite state machines. You will begin with FSMs that represent sequence recognizers, similar to the ones discussed in class, and then show how finite state machines can be used as part of a communication mechanism between two circuits.

2 Preparation

Your preparation must contain the source code and the simulation results for circuits described in Part I through Part III.

Your preparation, to be shown to your TA at the beginning of the lab, must consist of the following:

1. The Verilog source code used to implement each of the circuits described in Parts I through III. The code should be **PRINTED** and placed in your lab book **before** you arrive in the lab.
2. The circuits of Part I, II and IIIc should be simulated. The simulation must clearly indicate the operation of each circuit. The simulation must be **PRINTED** and **COMMENTED** before the lab begins. The comments must explain how the simulation results show correctness of the circuit.
3. In Part I, view the circuit you have created in the RTL viewer. The RTL viewer can be launched from the Quartus menu (Tools->Netlist viewers->RTL viewer). Count the number of flip-flops using the RTL-viewer (you may want to make a copy of the RTL Viewer image to paste into your notebook). A sample RTL viewer window is shown in Figure 1, with a group of flip-flops highlighted. Note that flip-flops will often be referred to as *registers* in Quartus II. Also, the RTL-viewer may "bundle" flip-flops together. For example, in Figure 1, four flip-flops have been bundled together ([3...0]). Additionally, you should observe the number of flip-flops using the compilation report. An example compilation report is shown in Figure 2 where Quartus II refers to flip-flops as registers.

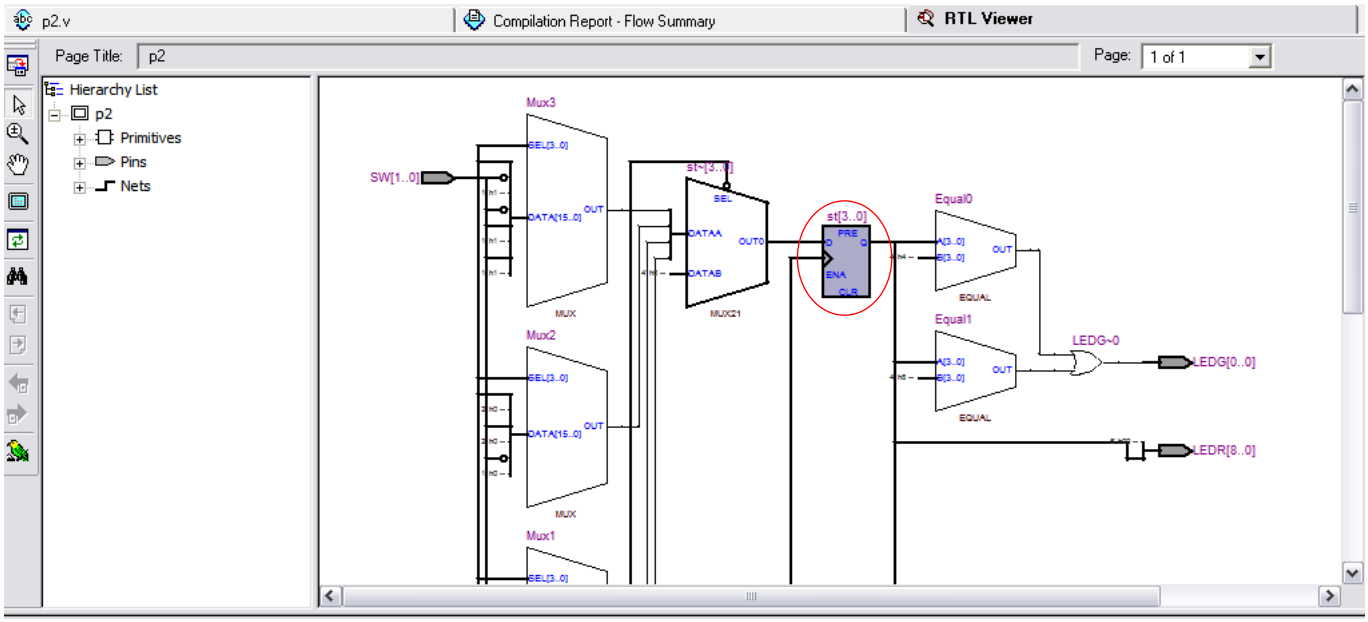

Figure 1: Sample RTL viewer window.

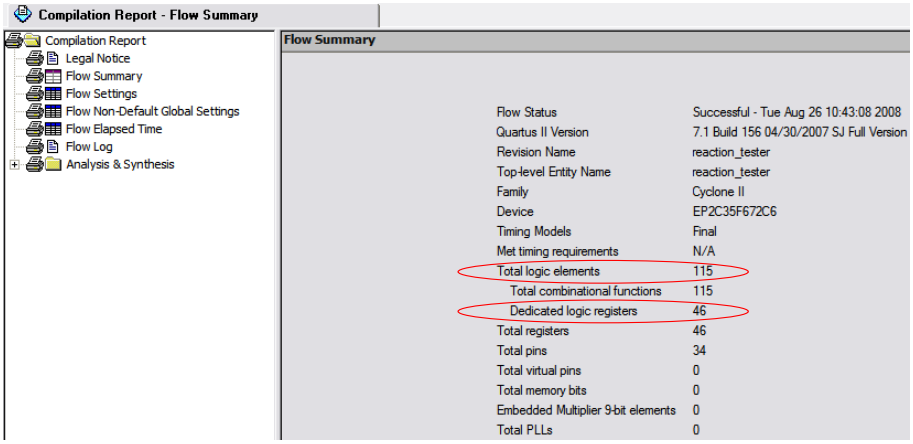

Figure 2: Circuit size and number of registers in the compilation report.

3 In the Lab

In the lab you will have to implement and test circuits described in the sections below. To simplify some of the steps a starter kit has been provided on the course website, located at:

`http://www.eecg.utoronto.ca/~jayar/ece241_08F/Lab6_starterkit.zip`

The starter kit is a ZIP archive containing a Quartus II projects for each part of the lab. Unzip the archive into a working directory called *lab6*.

3.1 Part I

The goal of this section is to implement a finite state machine that takes as input a serial input - a sequence of ones and zeroes on one input, a different one in every clock cycle, like the example described in class. In this case we want to recognize *two* sequences (rather than one in the example in class): either four consecutive 1s or four consecutive 0s. To describe it another way, assume that the input data is called w and the output is z. Whenever $w = 1$ or $w = 0$ for four consecutive clock periods the value of z should be set to 1; otherwise, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output z will be equal to 1 after the fourth and fifth clock periods. Figure 3 illustrates the required relationship between w and z.

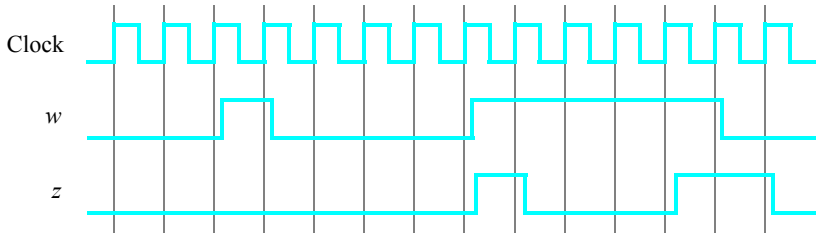

Figure 3: Required timing for the output z.

A state diagram for this FSM is shown in Figure 4. For this part you must to *manually* design an FSM circuit (i.e. **do not** use Verilog case statements, but create the logic and flip-flops directly) that implements this state diagram, including the logic expressions that feed each of the state flip-flops. To implement the FSM use nine state flip-flops called $y_8, \ldots, y_0$ and the one-hot state assignment given in Table 1. Note that the one-hot state codes enable you to derive these expressions by inspection, as described in class.

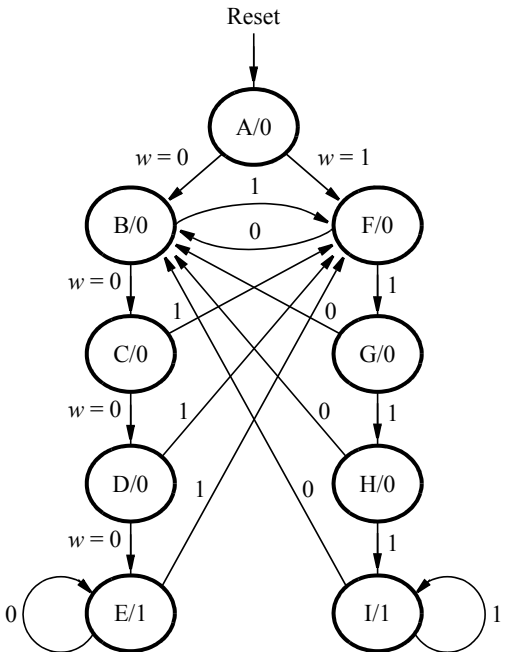

Figure 4: A state diagram for the FSM.

Name	State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$
A	000000001
B	000000010
C	000000100
D	000001000
E	000010000
F	000100000
G	001000000
H	010000000
I	100000000

Table 1: One-hot codes for the FSM.

Design and implement your circuit using Verilog. Write a Verilog file that instantiates the nine flip-flops in the circuit and which specifies the logic expressions that drive the flip-flop input ports. Use only simple **assign** statements in your Verilog code to specify the next-state logic feeding the flip-flops. You are not allowed to use behavioral types of Verilog statements, such as case statements, for this part of the lab exercise; specify the next-state logic using just simple **assign** statements.

Complete this part of the lab as follows:

1. The project for this part is provided in the starter kit. Open the project named *part1* in the *part1* subdirectory to begin your work.
2. Use the toggle switch SW_0 on the Altera DE2 board as an active-low synchronous reset input for the FSM (causing the FSM to reset to state A, with only $y_0 = 1$), use SW_1 as the w input, and the pushbutton KEY_0 as the clock input which is applied manually. Use the green light $LEDG_0$ as the output z, and assign the state flip-flop outputs to the red lights $LEDR_8$ to $LEDR_0$.
3. Include the Verilog file in your project, and assign the pins on the FPGA to connect to the switches and the LEDs. Compile the circuit.
4. Download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDG_0$.

3.2 Part II

In this part you will implement a similar sequence recognizer as in Part I, but you will use a different style of Verilog code, that makes it easier to create larger FSMs. You will build an extended version of the FSM in Figure 4. In this version, you are to detect the same pattern as in part I. However, you are to add one extra state called *Wait*. The FSM should go to the *Wait* state once one of the patterns is detected and stay there until an external signal, call *wait_done* is asserted. The external signal will be controlled by you using one of the switches on the DE2 board.

In this version of the code you should not manually derive the logic expressions needed for each state flip-flop. Instead, describe the state table for the FSM by using a Verilog **case** statement in an **always** block, and use another **always** block to instantiate the state flip-flops. You can use a third **always** block or simple assignment statements to specify the output z. Another difference from part I is that you should use four state flip-flops, $y_3, \ldots, y_0$, and encode the states using the binary codes as shown in Table 2. (The purpose in using this different method here

is to illustrate the difference between one-hot encoding, and a coding method that uses a minimum number of flip-flops).

Name	State Code $y_3y_2y_1y_0$
A	0000
B	0001
C	0010
D	0011
E	0100
F	0101
G	0110
H	0111
I	1000
Wait	1001

Table 2: Binary codes for the FSM.

A suggested skeleton of the Verilog code is given in Figure 5. While you do not have to follow the exact same structure, **you must use separate always blocks for the next state logic and the state flip-flops**. Doing so ensures that your design is well structured and easy to understand.

Implement your circuit as follows.

1. The project for this part is provided in the starter kit. Open the project named *part2* in the *part2* subdirectory to begin your work.
2. Include in the project your Verilog file that uses the style of code in Figure 5. Use the toggle switch SW_0 on the Altera DE2 board as an active-low synchronous reset input for the FSM, use SW_1 as the *w* input, and the pushbutton KEY_0 as the clock input which is applied manually. Use the green light $LEDG_0$ as the output *z*, and assign the state flip-flop outputs to the red lights $LEDR_3$ to $LEDR_0$. Connect the *wait_done* input signal to switch SW_2. Assign the pins on the FPGA to connect to the switches and the LEDs.
3. Before compiling your code it is necessary to explicitly tell the Synthesis tool in Quartus II that you wish to have the finite state machine implemented using the state assignment specified in your Verilog code. If you do not explicitly give this setting to Quartus II, the Synthesis tool will automatically use a state assignment of its own choosing, and it will ignore the state codes specified in your Verilog code. To make this setting, choose Assignments > Settings in Quartus II, click on the Analysis and Synthesis item on the left side of the window, and then click on More Settings. As indicated in Figure 6, change the parameter State Machine Processing to the setting User-Encoded.

```
module  part2 ( . . . );
    . . . define input and output ports

    . . . define signals
    reg [3:0] y_Q, Y_D;      // y_Q represents current state, Y_D represents next state
    parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
        F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000, Wait = 4'b1001;

    always @(w, y_Q)
    begin: state_table
        case (y_Q)
            A: if (!w) Y_D = B;
                else Y_D = F;
            . . . remainder of state table
            default: Y_D = 4'bxxxx;
        endcase
    end // state_table

    always @(posedge Clock)
    begin: state_FFs
        . . .
    end // state_FFS

    . . . assignments for output z and the LEDs
endmodule
```

Figure 5: Skeleton Verilog code for the FSM.

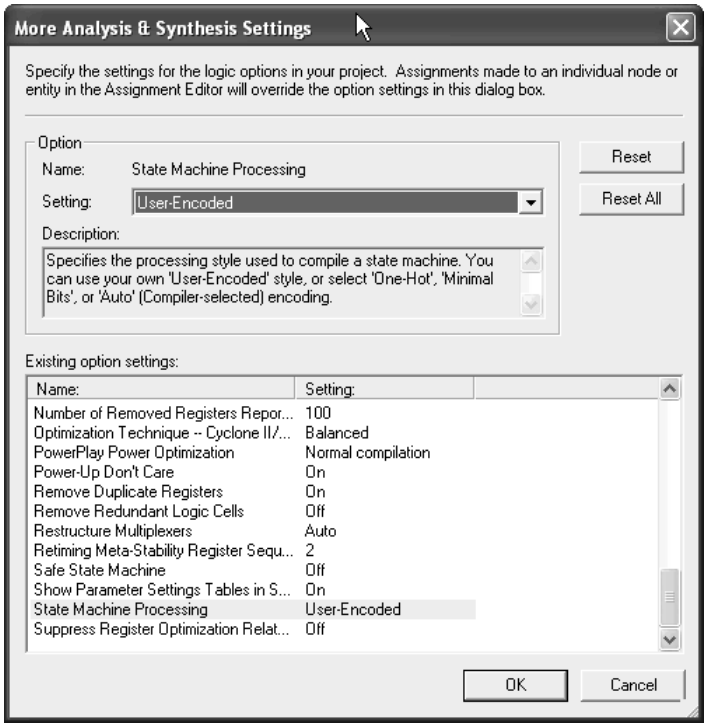

Figure 6: Specifying the state assignment method in Quartus II.

4. To examine the circuit produced by Quartus II open the RTL Viewer tool as described above. Double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one in Figure 4 (plus your extra Wait state). To see the state codes used for your FSM, open the Compilation Report, select the Analysis and Synthesis section of the report, and click on State Machines. You must also record the size of your circuit and the number of flip-flops (also called registers) used. The size is given as the number of Logic Elements (LEs) in the Compilation Report, and number of flip-flops is given as the total number of dedicated registers. You can refer back to Figure 2 for a sample compilation report.

5. Simulate the behavior of your circuit.

6. Once you are confident that the circuit works properly as a result of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDG_0$. Be sure to test for proper operation of the *wait_done* signal.

7. Now change the state encoding to back to one-hot. To do this, you just need to change the constants associated with the state names, and the size of the state register (i.e. encode state A as 000000001, state B as 000000010, and so on). Compile the circuit again and record the circuit size and the number of flip-flops used (remember, Quartus II will report flip-flops as registers in the compilation report). Show the new circuit to the TA and explain what you observe.

3.3 Part III

In this part you will work with two pre-designed circuits, or "cores" as they are called. The first core connects the FPGA to a computer keyboard; the second core takes a binary number as input and turns it into a musical tone, which you'll be able to hear by connecting a speaker to the DE2 board.

You will design a finite state machine to coordinate data transmission between two circuits so that keys from the keyboard can be used to play tones on the speaker. In working with these circuits you'll learn how to build systems in a modular fashion, by interconnecting prebuilt subcircuits.

Figure 7 illustrates a high-level view of the final circuit you will build. It includes a keyboard which is connected to the PS/2 data port (the purple plug) on the DE2 board. The wires from the PS/2 port are connected to the Cyclone II FPGA, and it communicates with the keyboard using a core called the *PS/2 Controller*; this core is given to you as part of the lab startup kit. You will ultimately build (in Part III*c*) the *Handshaking Interface* circuit indicated in Figure 7 that connects the PS/2 Controller to the *Synthesizer* core, which is then connected to an amplifier (on the DE2 board) and then (by you) to an external speaker. The Synthesizer core is also provided to you as part of the lab startup kit.

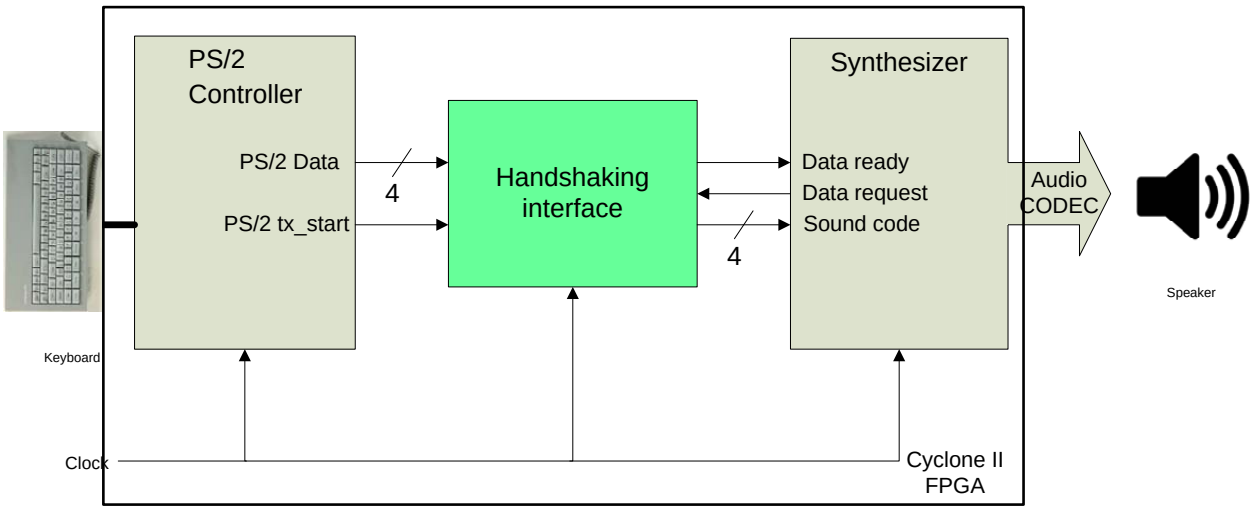

Figure 7: High level diagram of the system

You will design this circuit in three steps, by making the parts of the system work separately, and then together. (This, by the way, is the kind of discipline you'll need to learn to build successful projects, including the one in this course).

3.3.1 Part III*a*: Receiving Key Codes from the Keyboard and the Keyboard Interface

The keyboard uses a standard protocol known as PS/2. This protocol is fairly complex, so to prevent the need for you to understand its details, we have provided a greatly simplified interface in the core shown in Figure 7 called the PS/2 Controller. The PS/2 Controller has two outputs, $Data$ and tx_start. Whenever a valid key on the keyboard is pressed, a corresponding 4-bit key code will be provided on the $Data$ wires; this data is *valid* (i.e. ready to be taken) on a positive clock edge when the *tx_start* signal is high. You can only use the keyboard keys '1', '2', '3', '4', '5', '6', and '7' with this simplified PS/2 Controller, and they provide the four-bit codes $0000, \ldots, 0111$. The teaching assistant can show you how to connect the computer keyboard into the PS/2 plug (purple plug) on the DE2 board.

In this part you will not yet design the full circuit shown in Figure 7. Instead, you will design a simple circuit that can receive a single four-bit data value from the PS/2 Controller core and display this data on lights on the DE2 board. Each time you press a key on the keyboard you should see the corresponding data on the lights that you've used on the DE2 board. You are also required to observe the data transfer from the keyboard using the logic analyzer you first used in Lab 4.

Go to the part3 folder in the starter kit and the sub-folder part3a. This folder contains a project that has the pre-designed PS/2 Controller core and a skeleton Verilog file named part3a.v. In this code you will see a Verilog module that instantiates the PS/2 Controller in this line:

```
PS2_Controller PS2(.clk(CLOCK_50), .reset(~KEY[0]), .PS2_CLK(PS2_CLK),
.PS2_DAT(PS2_DAT), .ps2data(ps2data), .tx_start(ts));
```

The wires `ps2data` and `ts` are the output signals from the PS/2 Controller in Figure 7. Do not change the other signals connected to this core; you will not need to know what they do except that the main clock used is the 50 MHz signal `CLOCK_50`.

You should modify this circuit to

1. Capture the data from the controller into a 4-bit register. The output of the register should be displayed on four LED lights on the DE2 board.
2. Connect the ps2data and tx_start lines to the GPIO connector on the DE2 board so that you can look at these using the the logic analyzer. Refer to:

   `http://www.eecg.utoronto.ca/~jayar/ece241_08F/Logic_Scope_Tutorial.pdf`

   for the logic scope documentation and to Figure 8 for the expected waveform. The waveform shows a single key code transmission, including the PS/2 data bus and the expected key code value. Verify that the value that the logic analyzers sees is the same as the value you have on the the lights connected to your register.

Demonstrate the functioning circuit (including the logic scope readout) to the TA.

Figure 8: Timing waveform for one cycle, with a data value 0111.

3.3.2 Part III*b*:Using the Synthesizer Module to Make a Sound

In this part you will connect a few switches to the Synthesizer core and have it generate tones manually. You will manually perform the handshaking protocol that the Synthesizer core requires.

As indicated in Figure 7, the Synthesizer core provides a *data request* signal, and accepts a *data ready* signal and a 4-bit sound code. The Synthesizer will play a different note depending on the sound code provided. The sound codes accepted are 0000 to 0111.

You must communicate with the Synthesizer using the following *handshaking* protocol: there exist two synchronization signals, called *data request* and *data ready*, as illustrated in Figure 7. The Synthesizer will raise the *data request* line to a logic 1 (which you should connect to an LED on the DE2 board) once it is ready for a key code to "play". Once the *data request* line rises, you can set the data to be sent to the Synthesizer on the 4-bit sound code lines (which should be connected to switches on the DE2 board). To signal the Synthesizer that this data is ready, you must raise the *data ready* signal (which you should also connect to a switch) to a logic 1. *You must ensure that the data (the sound code) remains constant as long as the data request line is high*. Once the Synthesizer has taken the sound code data, it will lower the *data request* line to logic 0 and *wait until the data ready line is lowered*. Following this, the cycle repeats where the *data request* line will be raised once new data can be accepted.

To build a circuit that can implement this using switches, go to the folder in the starter kit named part3b. It contains a project that includes the Synthesizer core. In the main code file part3b.v you will see an instantiation of the Synthesizer that looks like this:

```
synthesizer s(CLOCK_50, !KEY[0], AUD_BCLK, AUD_DACLRCK,
AUD_DACDAT, AUD_XCK, I2C_SDAT, I2C_SCLK,
sound_code, data_rq, data_rd);
```

The wires *sound_code*, *data_rq*, *data_rd* are the signals given in Figure 7 as *sound code*, *data request* and *data ready*, respectively. Modify the code to connect these to the appropriate switches, and then compile and test the Synthesizer.

To test the Synthesizer, you must connect the audio output jack of the DE2 board to the speakers at your workstation. A teaching assistant can show you how to do this.

3.3.3 Part III*c*: Building the Interface between the Keyboard and the Synthesizer

Your final task is to create the whole circuit in Figure 7 by designing a finite state machine that connects between the PS/2 Controller and the Synthesizer. The handshaking is necessary, by the way, because the two modules operate at different and unknown frequencies where the sound takes multiple seconds to play, but the user might press keys either slower or faster than this. You are to design the interface that will handle this situation properly. The sound should be played based on the first key code entered after the previous sound finished playing. For example, if you press a new key while a note is playing, that key should be ignored.

The inputs to your state machine should be the synchronizing signals coming from the two cores (from the PS/2 Controller, the *tx_start* signal, and from the Synthesizer the *data request* signal). The outputs should be the *data ready* signal for the Synthesizer and the control of the register you needed in Part III*a*. The various data lines will have to be connected to that register in the appropriate way.

For your reference, we show one cycle of communication in Figure 9. When the handshaking interface FSM sees the *data request* signal asserted to 1 by the Synthesizer, it is ready to capture data from the keyboard. In the clock period when *tx_start* becomes 1, this data is captured. Here, we assume that the key 7 has been pressed, providing a data value of 0111. Some number of cycles later (depending on your design), the FSM asserts *data ready* to 1 and waits for the Synthesizer to deassert *data request* to complete the handshaking protocol.

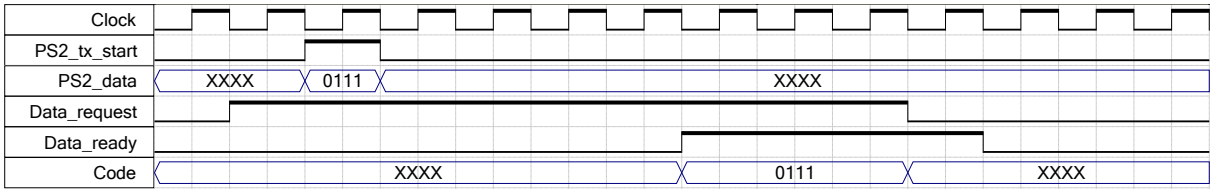

Figure 9: Timing waveform for one cycle, sending data value 0111 from the PS/2 interface to your Synthesizer.

The top-level Veriog file is provided in the folder part3/part3c of the starter kit. The file part3c.v is the top-level module that instantiates the keyboard PS/2 Controller, the Synthesizer, and the module you'll need to design called handshake_p3. You are to write your FSM code in the file called *handshake_p3.v*. The module interface is already in this file. You should reuse parts of your design from Parts III*a* and b as appropriate. You should not need to make changes in the top-level file *part3c.v*.