EE25266 – ASIC/FPGA Chip Design

Mahdi Shabany
Electrical Engineering Department
Sharif University of Technology

Assignment #3

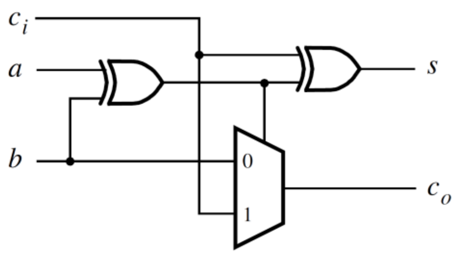
Registers, Adders and Counters, Finite State Machines and Pre-designed Cores

Introduction

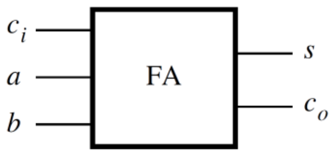
The purpose of this laboratory exercise is to investigate registers, adders, counters, Finite State Machines and Pre-designed Cores. We will begin with the design of a ripple-carry adder and then use it in conjunction with registers to perform simple arithmetic computations. We will also explore counters as the part of the exercise. Moreover, Finite State Machines (FSMs) and pre-designed cores will be described in this assignment. 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.
To simplify some of the steps a starter kit has been provided on the course website under Assignment #3. The starter kit is a ZIP archive containing a Quartus II projects for each part of the lab. Retrieve and unzip the archive into a working directory called *Assignment3*.

Part I:

Fig. 1a shows a circuit for a *full adder*, which has the inputs a, b, and c_i, and produces the outputs s and co. Parts b and c of the figure show a circuit symbol and truth table for the full adder, which produces the two-bit binary sum $c_o s = a + b + c_i$ (Note: in this expression, the + symbol means *addition*, as opposed to logical OR). Fig. 1d shows how four instances of this full adder module can be used to design a circuit that adds two four-bit numbers. This type of circuit is usually called a *ripple-carry* adder, because of the way that the carry signals are passed from one full adder to the next.



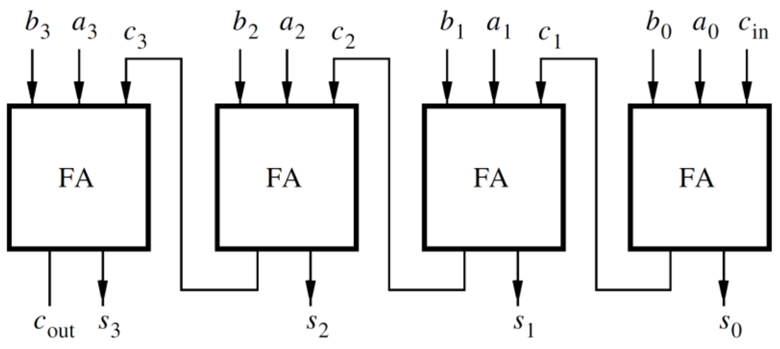
a) Full adder circuit



b) Full adder symbol

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c) Full adder truth table



d) Four-bit ripple-carry adder circuit

Fig. 1. A ripple-carry adder circuit.

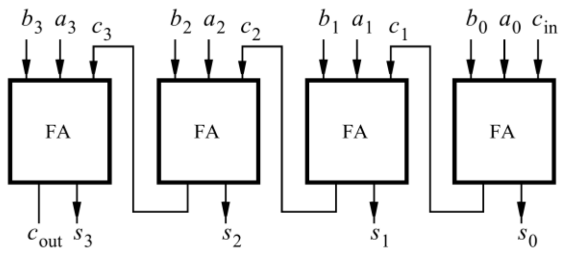
Write Verilog code that implements this circuit, as described below.

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. Write a Verilog module for the full adder subcircuit and write a top-level Verilog module that instantiates four instances of this full adder.
3. Use switches SW7−4 and SW3−0 to represent the inputs A (which is made up of $a_3a_2a_1a_0$) and B (which is made up of of $b_3b_2b_1b_0$) respectively. Use SW8 for the carry-in cin of the adder. Connect the SW switches to their corresponding red lights LEDR, and connect the outputs of the adder, cout and S (which comprises $s_3s_2s_1s_0$), to the green lights LEDG.
4. Simulate your circuit by trying different values for numbers A, B, and cin.
5. If you have not already done so, include the necessary pin assignments for the board, compile the circuit, and download it into the FPGA chip.

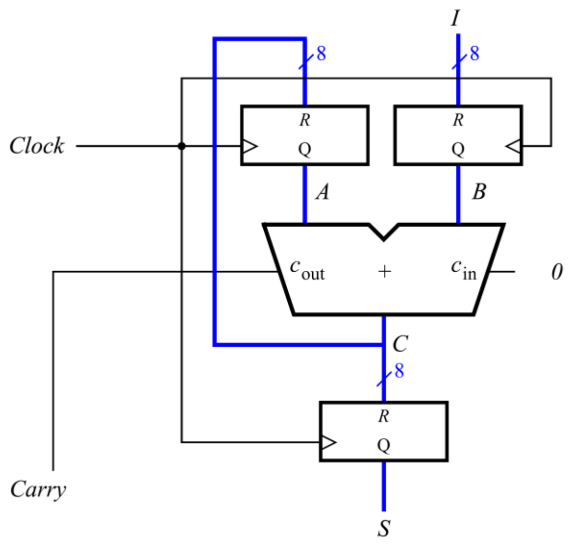
DE2: Download the circuit onto the board and test its functionality.

Part II :

Consider again the four-bit ripple-carry adder circuit from Part I; a diagram of this circuit is reproduced in Fig. 2a. You are to create an 8-bit version of the adder with additional registers, and include it in the circuit shown in Fig. 2b. Your circuit should be designed to support signed numbers in 2's-complement form. Note that the boxes labeled "R" in the figure are 8-bit registers - that is 8 D-type flip-flops with a common clock.



a) Four-bit ripple-carry adder circuit



b) Eight-bit registered adder circuit

Fig. 2. An 8-bit signed adder with registered inputs and outputs.

Perform the steps shown below:

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. Write Verilog code that describes the circuit in Figure 2b. Use the circuit structure in Figure 2a to describe your adder.

3. Include the required input and output ports in your project to implement the adder circuit on the board. Connect the input B to switches *SW*7−0. Use *KEY*0 as an active-low asynchronous reset input for the registers, and use *KEY*1 as a manual clock input. Display the sum outputs of the adder on the red

*LEDR*7−0 lights and display the carry output on the green *LEDG*8 light. The hexadecimal values of A and B should be shown on the displays *HEX7-6* and *HEX5-4*, and the hexadecimal value of S should appear on *HEX1-0*.

4. Compile your code and use timing simulation to verify the correct operation of the circuit.

5. Open the Quartus II Compilation Report and examine the results reported by the Timing Analyzer. What is the maximum operating frequency, *fmax*, of your circuit? What is the longest path in the circuit in terms of delay?

DE2: Download the circuit onto the board and test its functionality by using different values of B. Use some test cases where the circuit produces arithmetic overflow and ensure that you understand why your circuit produces incorrect results for these cases.

Part III:

Modify your circuit from Part II so that it can perform both addition and subtraction of eight-bit numbers. Use switch *SW*16 to specify whether addition or subtraction should be performed. Connect the other switches, lights, and displays as described for Part II.

1. Simulate your adder/subtractor circuit to show that it functions properly, and then download it onto the board and test it by using different switch settings.

2. Open the Quartus II Compilation Report and examine the results reported by the Timing Analyzer. What is the *fmax* of your circuit? What is the longest path in the circuit in terms of delay?

Part IV :

Implement a 3-digit BCD counter. Display the contents of the counter on the 7-segment displays *HEX2−0*. Derive a control signal, from the 50-MHz clock signal provided on the Altera board, to increment the contents of the counter at one-second intervals. Use the pushbutton switch *KEY*0 to reset the counter to 0. A diagram of the circuit is shown in Fig. 3.

IMPORTANT: In your design ensure that you do NOT *gate* the clock input. That is, the clock input to your circuit MUST be connected directly to the *clock* input of each flip-flop in your design. When a need arises to disable a flip-flop for a number of clock cycles, use an *enable* signal to keep the flip-flop’s stored value from changing.

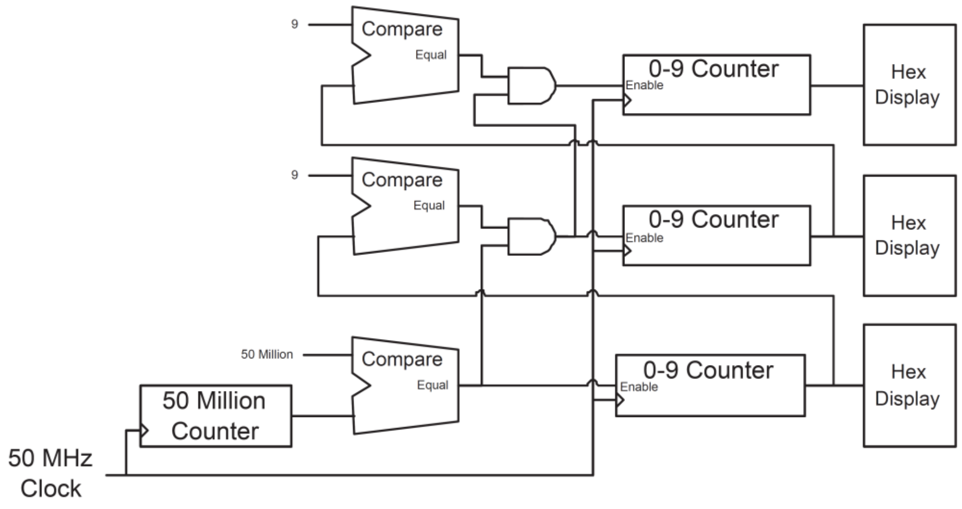


Fig. 3. Block Diagram of a 3-digit BCD Counter

Perform the following steps to complete this part:

1. The project for this part is provided in the starter kit. Open the project named *part4* in the *part4* subdirectory to begin your work.
2. Write a Verilog file that specifies the desired circuit.
3. Include the Verilog file in your project and compile the circuit.
4. Simulate the designed circuit to verify its functionality.
5. Assign the pins on the FPGA to connect to the 7-segment displays and the pushbutton switch.

DE2: Recompile the circuit and download it into the FPGA chip. Verify that your circuit works correctly by observing the display.

Part V:

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. In this case we want to recognize two sequences: 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. Fig. 4 illustrates the required relationship between w and z.

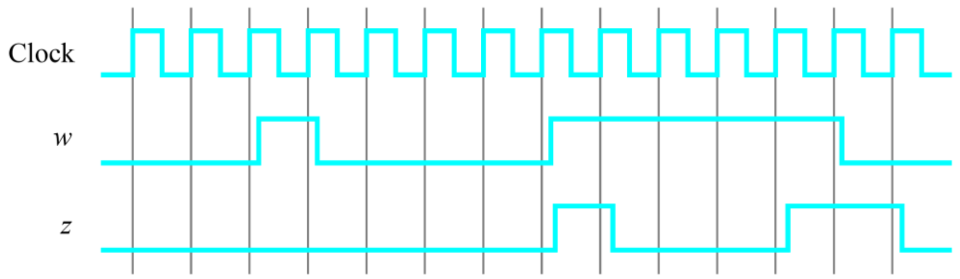


Fig. 4. Required timing for the output z.

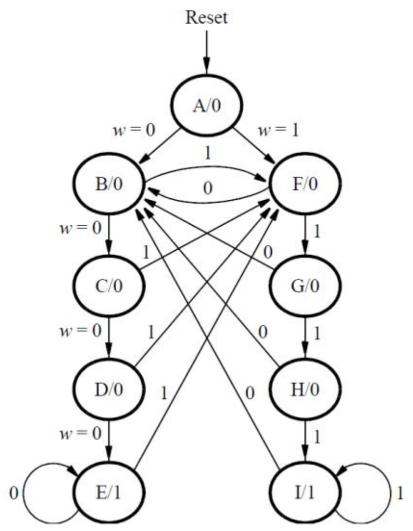


Fig. 5. A state diagram for the FSM.

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.

A state diagram for this FSM is shown in Fig. 5. 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 y8, . . . , y0 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.

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 part5 in the part5 subdirectory to begin your work.

2. Use the toggle switch SW0 on the Altera board as an active-low synchronous reset input for the FSM (causing the FSM to reset to state A, with only y0 = 1), use SW1 as the w input, and the pushbutton KEY0 as the clock input which is applied manually. Use the green light LEDG0 as the output z, and assign the state flip-flop outputs to the red lights LEDR8 to LEDR0.

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.

DE2: 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 LEDG0.

Part VI :

In this part you will implement a similar sequence recognizer as in Part V, 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 Fig. 5. In this version, you are to detect the same pattern as in part V. 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 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, y3, . . . , y0, 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).
A suggested skeleton of the Verilog code is given in Fig. 6. 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.

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.

```
module part6 ( . . . );
//. . . 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
```

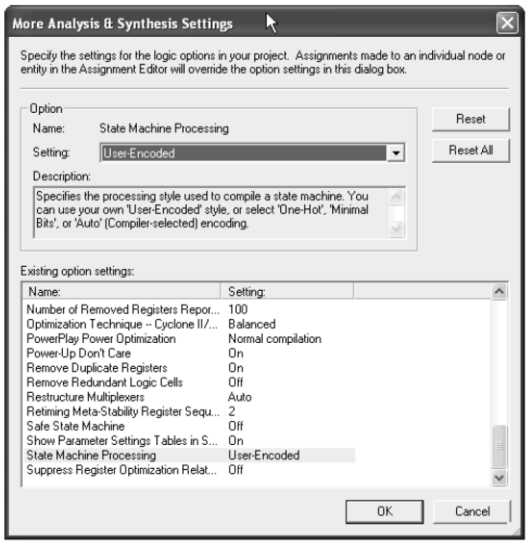
Fig. 6. Skeleton Verilog code for the FSM.

Implement your circuit as follows.

1. The project for this part is provided in the starter kit. Open the project named part6 in the part6 subdirectory to begin your work.

2. Include in the project your Verilog file that uses the style of code in Figure 6. Use the toggle switch SW0 on the Altera board as an active-low synchronous reset input for the FSM, use SW1 as the w input, and the pushbutton KEY0 as the clock input which is applied manually. Use the green light LEDG0 as the output z, and assign the state flip-flop outputs to the red lights LEDR3 to LEDR0. Connect the wait_done input signal to switch SW2. 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 Fig. 7, change the parameter State Machine Processing to the setting User-Encoded.



Specify the settings for the logic options in your project. Assignments made to an individual node or entity in the Assignment Editor will override the option settings in this dialog box.

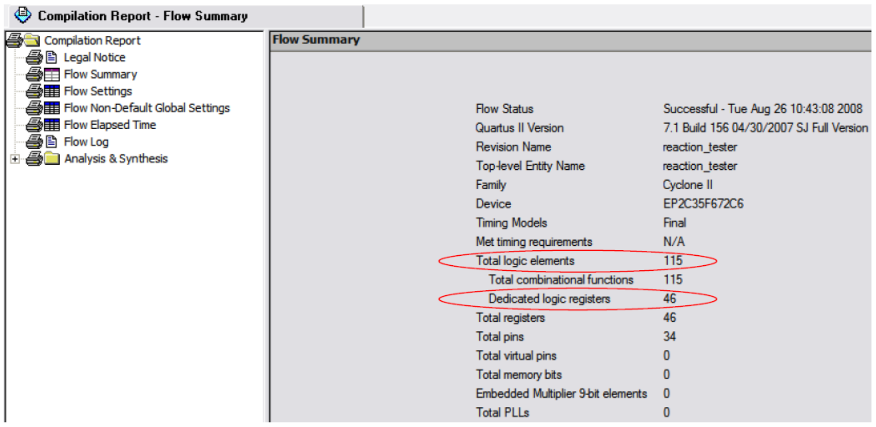
Option
Name: State Machine Processing
Setting: User-Encoded
Description:
Specifies the processing style used to compile a state machine. You can use your own 'User-Encoded' style, or select 'One-Hot', 'Minimal Bits', or 'Auto' (Compiler-selected) encoding.

Existing option settings:

Name:	Setting:
Number of Removed Registers Repor...	100
Optimization Technique -- Cyclone II/...	Balanced
PowerPlay Power Optimization	Normal compilation
Power-Up Don't Care	On
Remove Duplicate Registers	On
Remove Redundant Logic Cells	Off
Restructure Multiplexers	Auto
Retiming Meta-Stability Register Sequ...	2
Safe State Machine	Off
Show Parameter Settings Tables in S...	On
State Machine Processing	User-Encoded
Suppress Register Optimization Relat...	Off

Fig. 7. Specifying the state assignment method in Quartus II.

4. To examine the circuit produced by Quartus II open the RTL Viewer tool. The RTL viewer can be launched from the Quartus menu (Tools->Netlist viewers->RTL viewer). 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 Fig. 5 (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 Fig. 8 for a sample compilation report.



Flow Status	Successful - Tue Aug 26 10:43:08 2008
Quartus II Version	7.1 Build 156 04/30/2007 SJ Full Version
Revision Name	reaction_tester
Top-level Entity Name	reaction_tester
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	N/A
Total logic elements	115
Total combinational functions	115
Dedicated logic registers	46
Total registers	46
Total pins	34
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Fig. 8. Circuit size and number of registers in the compilation report.

5. Simulate the behavior of your circuit.

DE2: 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 LEDG0. Be sure to test for proper operation of the wait_done signal.

6. 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.

Part VII :

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 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.

Fig. 9 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 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 VIIc) the Handshaking Interface circuit indicated in Fig. 9 that connects the PS/2 Controller to the Synthesizer core, which is then connected to an amplifier (on the board) and then (by you) to an external speaker. The Synthesizer core is also provided to you as part of the lab startup kit.

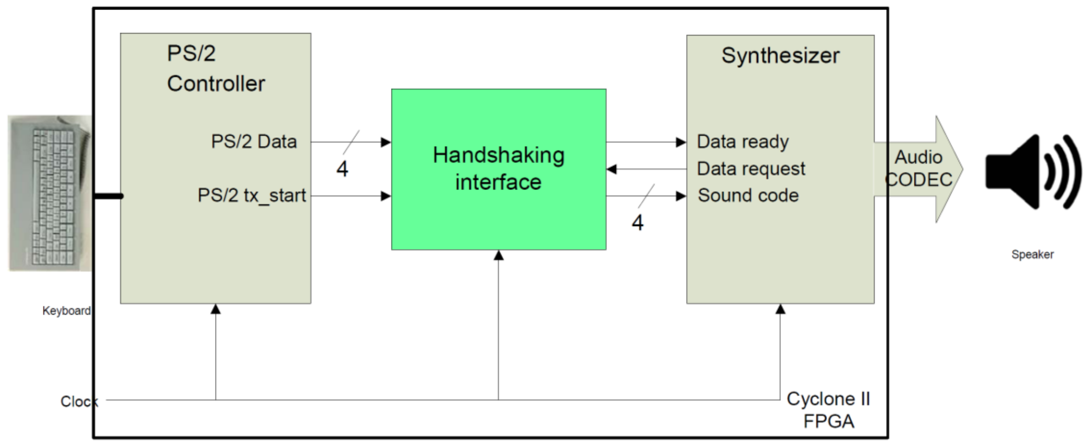


Fig. 9. High level diagram of the system.

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).

Part VII.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 Fig. 9 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, . . . , 0111. The teaching assistant can show you how to connect the computer keyboard into the PS/2 plug (purple plug) on the board.

In this part you will not yet design the full circuit shown in Fig. 9. 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 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 board. Go to the part3 folder in the starter kit and the sub-folder part3a. This folder contains a project that has the predesigned PS/2 Controller core and a skeleton Verilog file named part7a.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 Fig. 9. 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 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 board.

Part VII.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 Fig. 9, 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 Fig. 9. The Synthesizer will raise the data request line to a logic 1 (which you should connect to an LED on the 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 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 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 part7b. It contains a project that includes the

Synthesizer core. In the main code file part7b.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 Fig. 9 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 board to the speakers at your workstation. The TA can show you how to do this.

Part VII.c: Building the Interface between the Keyboard and the Synthesizer

Your final task is to create the whole circuit in Fig. 9 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 VIIa. 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 Fig. 10. 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.

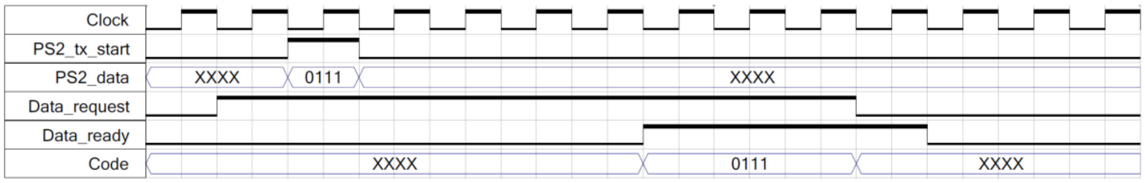


Fig. 10: Timing waveform for one cycle, sending data value 0111 from the PS/2 interface to your Synthesizer.

The top-level Veriog file is provided in the folder part7c of the starter kit. The file part7c.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 VIIa and b as appropriate. You should not need to make changes in the top-level file part7c.v.

DE2: Download the circuit into the FPGA chip. Test the functionality of your design by applying the inputs on the keyboard.