RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

ECE493 SPECIAL TOPICS

Hardware/Software Design of Embedded Systems Laboratory

Fall 2013

Last Updated:
October 5, 2013

Contents

1 Lab 1 - Introduction to FPGA's and VHDL 3
1.1 Introduction . . . 3
1.2 The DE2-115 FPGA Development Board . . . 3
1.3 Pre-lab . . . 4
1.4 Working with Quartus II . . . 4
1.4.1 Creating a New Project . . . 4
1.4.2 Creating an Empty File . . . 6
1.4.3 Writing VHDL Code . . . 7
1.4.4 Setting Pin Assignments . . . 7
1.4.5 Compiling Hardware . . . 8
1.4.6 Testing Hardware with Waveforms . . . 9
1.4.7 Uploading Hardware to Device . . . 9
1.5 Activities . . . 11
1.5.1 Implementing Logic . . . 11
1.5.2 7 Segment Display Decoder . . . 11
1.6 Lab Report . . . 12

2 Lab 2 - Latches, Flip-Flops, and Counters 13
2.1 Introduction . . . 13
2.2 Pre-lab . . . 13
2.3 Lab Activities . . . 13
2.3.1 Latches . . . 13
2.3.2 Flip-Flop . . . 14
2.3.3 Counters . . . 14
2.4 Lab Report . . . 14

3 Lab 3 - Complex Addition Systems 16
3.1 Introduction . . . 16
3.2 Pre-lab . . . 16
3.3 Lab Activities . . . 16
3.3.1 Half Adder . . . 16
3.3.2 Full Adder . . . 16
3.3.3 Full Adder Based ALU . . . 17
3.4 Lab Report . . . 18

4 Lab 4 - Finite-State Machines 19
4.1 Introduction . . . 19
4.2 Pre-lab . . . 19
4.3 Lab Activities . . . 19
4.3.1 FSM . . . 19
4.3.2 Vending Machine . . . 21
4.4 Lab Report . . . 22

5 Lab 5 - A Simple Processor 23
5.1 Introduction . . . 23
5.2 Pre-lab . . . 23
5.3 Lab Activities . . . 23
5.3.1 CPU Controller . . . 23
5.3.2 Design an ALU . . . 24

5.3.3 Initiate the FLASH Memory . . . 24
5.3.4 FLASH Memory Controller . . . 25
5.3.5 Program Counter . . . 26
5.3.6 Results Display . . . 26
5.4 Lab Report . . . 27

1 Lab 1 - Introduction to FPGA's and VHDL

1.1 Introduction

This lab will introduce you to the Altera DE2-115 FPGA Development Board. The DE2-115 contains all of the hardware necessary to prototype and create various hardware configurations on the Altera Cyclone IV FPGA chip that will be used throughout the course of this lab. By completing this lab, you will have an understanding of all the hardware contained on the FPGA development board, along with an understanding of how to connect peripherals to the development board. Lastly, this lab will go over the standard template for designing hardware in the VHDL programming language. All this will be accomplished by following the Quartus II introductory packet along with the following activities.

1.2 The DE2-115 FPGA Development Board

An overview of the DE2-115 FPGA Development Board can be found in Figure 1. Take note of the layout of the board. It is expected that by the end of this course you should be able to point out and explain the various clocks, configuration devices, and peripheral ports on the development board.

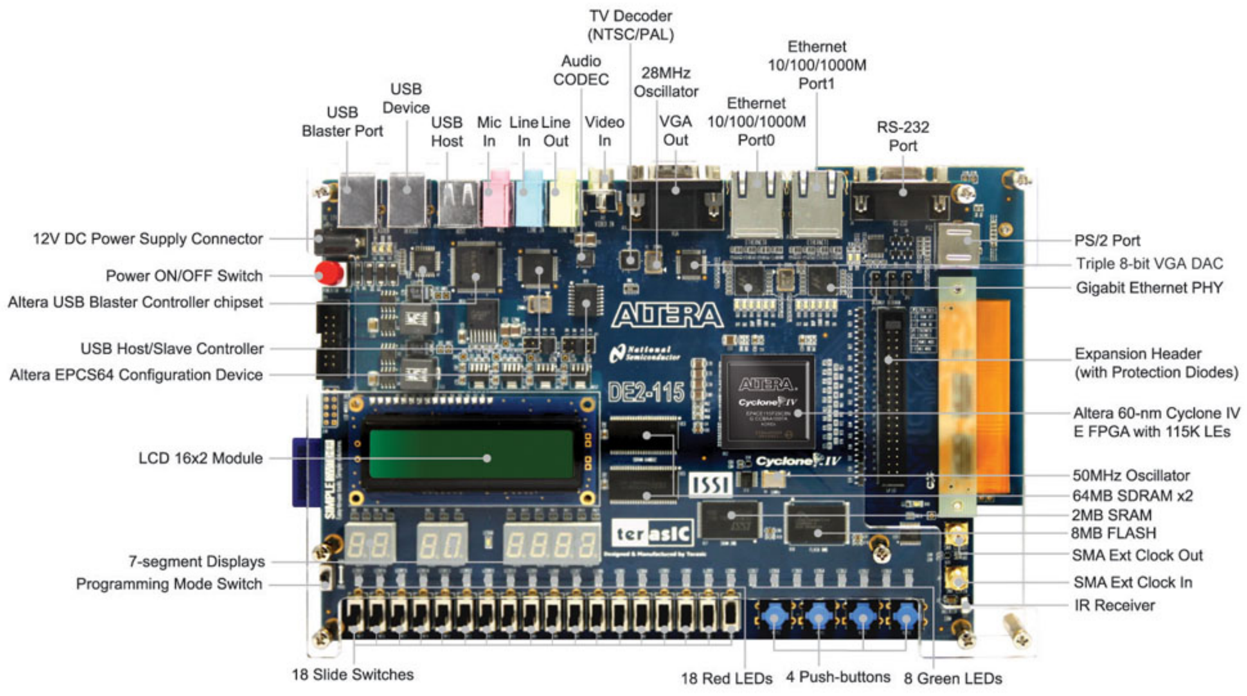


Figure 1: The DE2-115 FPGA Development Board

Figure 2 is an overview of how all the components of the board are connected to the FPGA device.

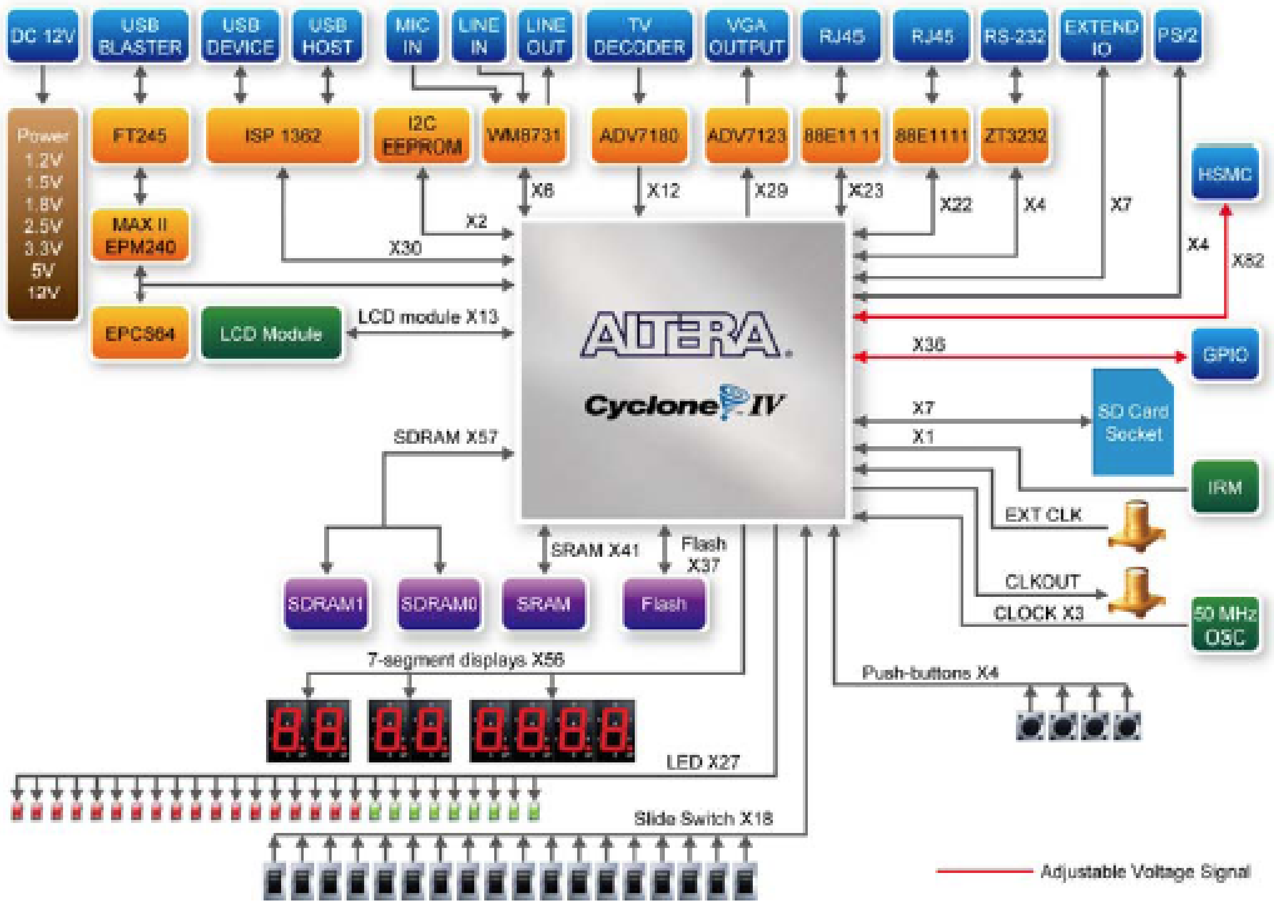


Figure 2: The DE2-115 Block Diagram

1.3 Pre-lab

Before attempting this lab, you should complete the following:

- Download and install the *Quartus II 13.0 Web Edition Software* located at:
  `http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html`
- Download and install the *Altera University Installer Software 13.0* located at:
  `http://www.altera.com/education/univ/software/upds/unv-upds.html`
- Download and read the following documents from the Sakai Resources pags; *DE2-115 User Manual, Quartus II Introduction.*

1.4 Working with Quartus II

1.4.1 Creating a New Project

1. In order to begin working on your first FPGA project, you must open the program Altera Quartus II. To begin a new project go to **File** → **New Project Wizard** as shown in Figure 3.

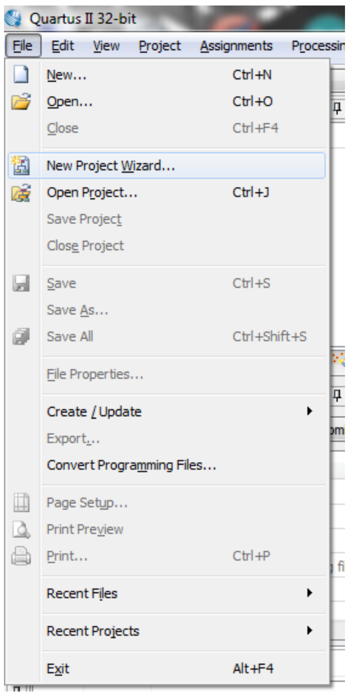
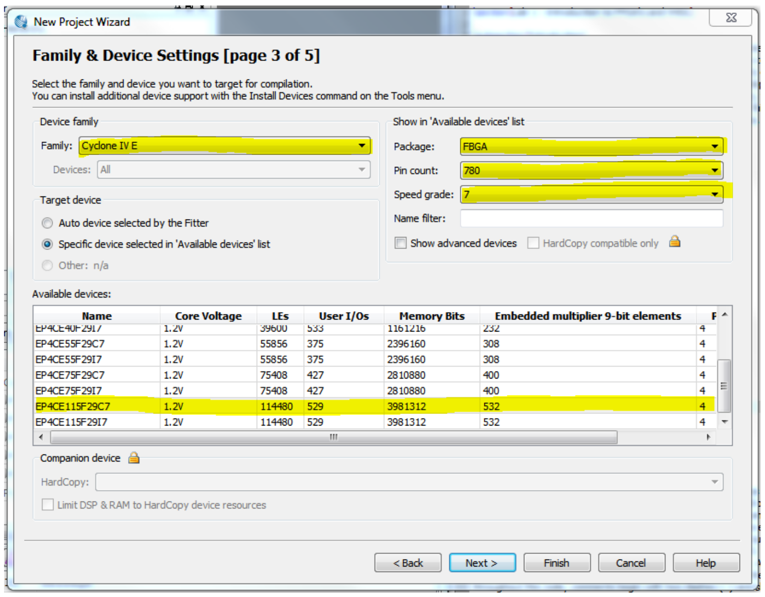


Figure 3: Create a new project menu

2. When the *New Project Wizard* window opens click **NEXT**.
3. Create a folder in your Z-drive called *FPGA Lab*.
4. Create a folder in FPGA Lab called *lab1*.
5. Set the working directory to *lab1*.
6. Name the project *lab1*.
7. Click **NEXT** to proceed.
8. Skip this step, click **NEXT** to proceed.
9. Under Device Family select *Cyclone IV E*, set the package to *FBGA*, pin count to *780*, and speed grade to *7*. In the Available devices list, look for and select *EP4CE115F29C7* as shown in Figure 4.



New Project Wizard

Family & Device Settings [page 3 of 5]

Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.

Device family
Family: Cyclone IV E
Devices: All

Show in 'Available devices' list
Package: FBGA
Pin count: 780
Speed grade: 7
Name filter:
Show advanced devices
HardCopy compatible only

Target device
Auto device selected by the Fitter
Specific device selected in 'Available devices' list
Other: n/a

Available devices:

Name	Core Voltage	LEs	User I/Os	Memory Bits	Embedded multiplier 9-bit elements	P
EP4CE40F29I7	1.2V	39600	533	1161216	232	4
EP4CE55F29C7	1.2V	55856	375	2396160	308	4
EP4CE55F29I7	1.2V	55856	375	2396160	308	4
EP4CE75F29C7	1.2V	75408	427	2810880	400	4
EP4CE75F29I7	1.2V	75408	427	2810880	400	4
EP4CE115F29C7	1.2V	114480	529	3981312	532	4
EP4CE115F29I7	1.2V	114480	529	3981312	532	4

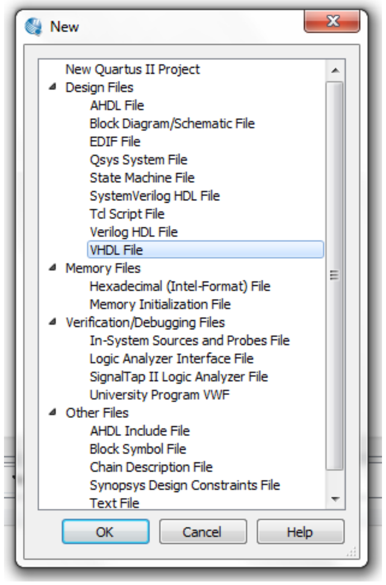
Companion device
HardCopy:
Limit DSP & RAM to HardCopy device resources

Figure 4: Device selection menu

10. Click **FINISH** to begin your new project.

1.4.2 Creating an Empty File

1. Go to **File** → **New** → **VHDL File** → **OK** as shown in Figure 5.



New

New Quartus II Project
Design Files
AHDL File
Block Diagram/Schematic File
EDIF File
Qsys System File
State Machine File
SystemVerilog HDL File
Tcl Script File
Verilog HDL File
VHDL File
Memory Files
Hexadecimal (Intel-Format) File
Memory Initialization File
Verification/Debugging Files
In-System Sources and Probes File
Logic Analyzer Interface File
SignalTap II Logic Analyzer File
University Program VWF
Other Files
AHDL Include File
Block Symbol File
Chain Description File
Synopsys Design Constraints File
Text File

Figure 5: Create a new file menu

2. Save this new file by going to **File** → **Save As** → ***lab1.vhd*** → **SAVE** as shown in Figure 6.

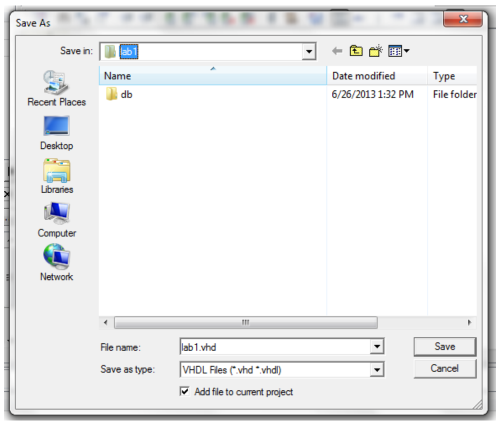


Figure 6: Save As dialog box

1.4.3 Writing VHDL Code

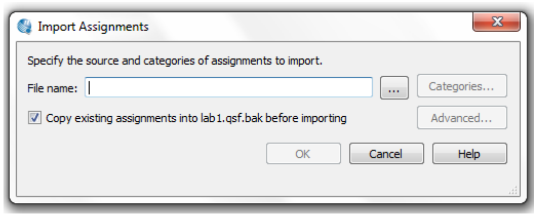
The following code block shows how to interact with the switches and LEDs on the DE2-115. Notice how the program begins with importing the ieee library which contains all of the basic logic primitives as established within the IEEE standard 1164. When working in industry it is common for large companies to create their own libraries as well. Every VHDL file should contain at least one entity (module) that is the same as the name of the file. An entity contains information about the structure of the module such as how many inputs/outputs (I/O) and what type of logic to expect on the I/O.Once the port structure for the entity is defined, its logical behavior is described within an architecture block. As can be seen, the code below is setting the red LEDs as defined in the array to the accompanying switches on the board. Take note on the use of comments throughout the code, comments begin with two dashes (–) and should always be used to describe what you are trying to accomplish, this way someone else who reads your code will understand it easily and your code will look more professional.

```vhdl
-- Import logic primitives
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Simple module that connects the SW switches to the LEDR lights
ENTITY lab1 IS
PORT ( SW: IN STD_LOGIC_VECTOR(17 DOWNTO 0); -- Initialize switches as an input
      LEDR: OUT STD_LOGIC_VECTOR(17 DOWNTO 0)); -- Initialize red LEDs as an output
END lab1;

-- Define characteristics of the entity lab1
ARCHITECTURE Behavior OF lab1 IS
BEGIN
      LEDR <= SW; -- Assign each switch to one red LED
END Behavior;
```

1.4.4 Setting Pin Assignments

The Cyclone IV chip on the DE2-115 contains 780 pins. Most of the pins have been predetermined through their routing on the PCB to directly control all of the interfaces on the board. As a result, Altera has provided a file that contains all of the pin assignments so that you can interact directly with the hardware on the board through your VHDL code. To set the pin assignments:

1. Go to **Assignments → Import Assignments → Select *DE2-115.qsf*** as shown in Figure 7. This file should be downloaded from Sakai under resources.
2. Then click **ADVANCED → check Global Assignments → Ok** as shown in Figure 8.

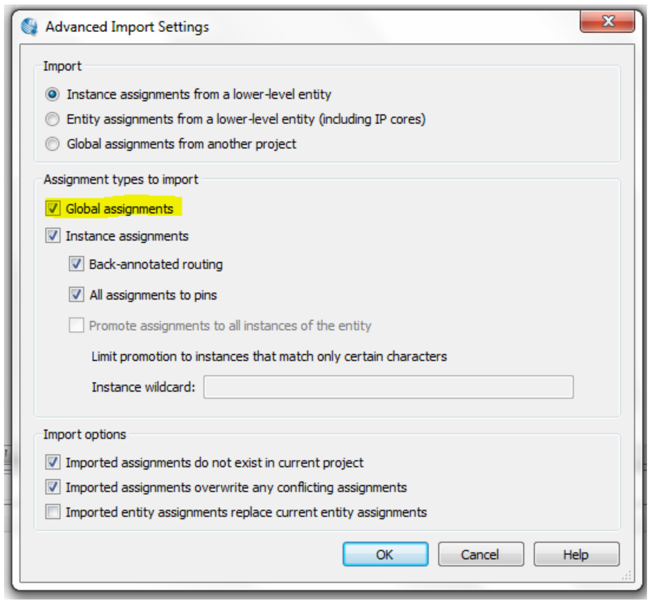

Import Assignments

Specify the source and categories of assignments to import.

File name:

[✓] Copy existing assignments into lab1.qsf.bak before importing

Figure 7: Import assignments dialog box


Advanced Import Settings

Import
- ● Instance assignments from a lower-level entity
- ○ Entity assignments from a lower-level entity (including IP cores)
- ○ Global assignments from another project

Assignment types to import
- [✓] Global assignments
- [✓] Instance assignments
  - [✓] Back-annotated routing
  - [✓] All assignments to pins
  - [ ] Promote assignments to all instances of the entity
  - Limit promotion to instances that match only certain characters
  - Instance wildcard:

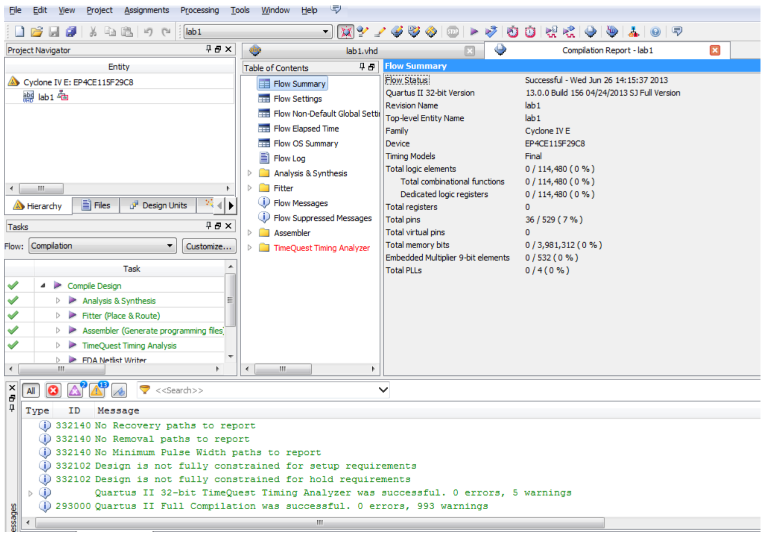
Import options
- [✓] Imported assignments do not exist in current project
- [✓] Imported assignments overwrite any conflicting assignments
- [ ] Imported entity assignments replace current entity assignments

Figure 8: Import assignments advanced options menu

1.4.5 Compiling Hardware

1. To begin compiling your hardware, go to **Processing → Start Compilation** or you may press ***Ctrl* + *L*** on your keyboard.

If the project compiles successfully, you may proceed to uploading the hardware. Otherwise if you have any errors you should debug your code. It is helpful to note that the first error should be solved first which will make it easier to solve the other errors. A successful compilation will look like Figure 9.



Flow Status	Successful - Wed Jun 26 14:15:37 2013
Quartus II 32-bit Version	13.0.0 Build 156 04/24/2013 SJ Full Version
Revision Name	lab1
Top-level Entity Name	lab1
Family	Cyclone IV E
Device	EP4CE115F29C8
Timing Models	Final
Total logic elements	0 / 114,480 (0 %)
Total combinational functions	0 / 114,480 (0 %)
Dedicated logic registers	0 / 114,480 (0 %)
Total registers	0
Total pins	36 / 529 (7 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

```
Type  ID     Message
      332140 No Recovery paths to report
      332140 No Removal paths to report
      332140 No Minimum Pulse Width paths to report
      332102 Design is not fully constrained for setup requirements
      332102 Design is not fully constrained for hold requirements
             Quartus II 32-bit TimeQuest Timing Analyzer was successful. 0 errors, 5 warnings
      293000 Quartus II Full Compilation was successful. 0 errors, 993 warnings
```

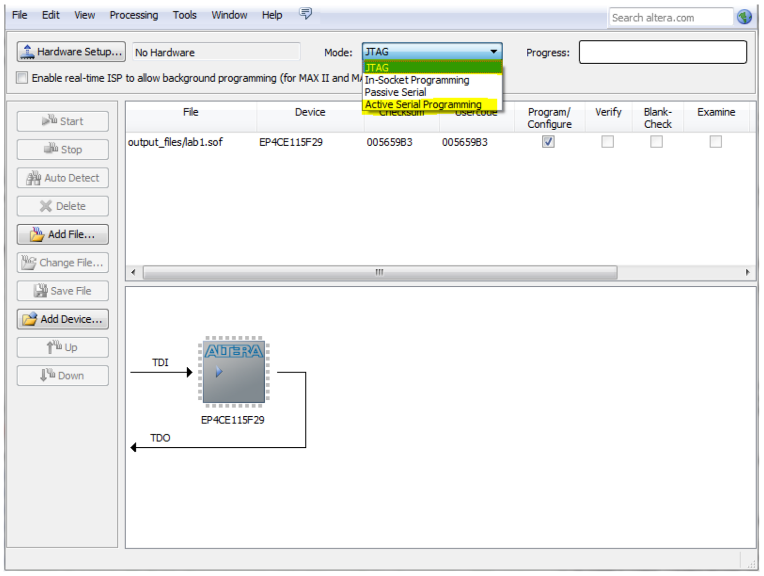
Figure 9: Successful completion of compilation

1.4.6 Testing Hardware with Waveforms

All hardware implementations should be tested for correctness before the VHDL code is uploaded to the FPGA device. To accomplish this, please follow the tutorial titled *Quartus_II_Simulation.pdf* listed under Sakai resources.

1.4.7 Uploading Hardware to Device

Once the hardware has compiled successfully, go to **Tools** → **Programmer** to open the hardware upload options. There are two typical modes for uploading hardware and it is important to understand when to use them. This can be seen in Figure 10.



Mode: JTAG

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine
output_files/lab1.sof	EP4CE115F29	005659B3	005659B3	✓			

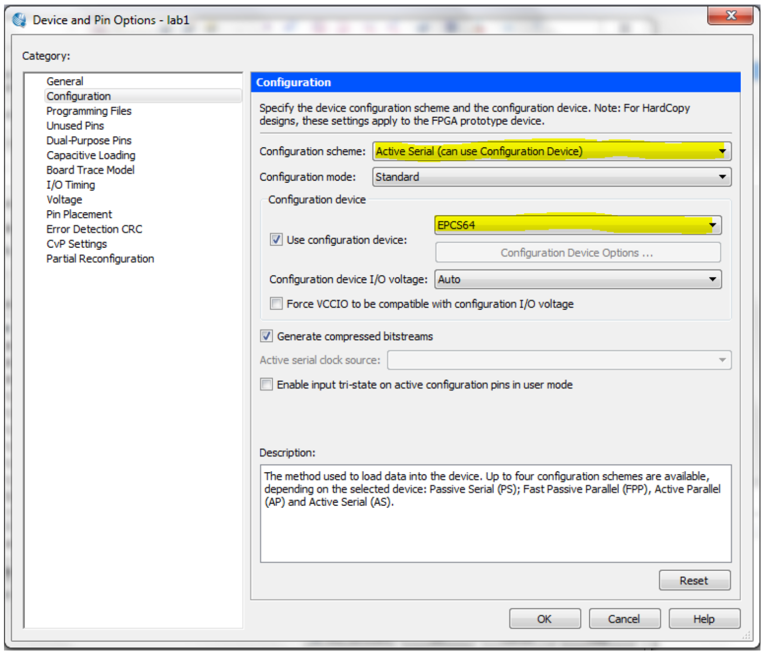
Figure 10: Modes for uploading hardware to the FPGA device

1. JTAG or Joint Test Action Group
    - This method loads the VHDL code directly to the FPGA chip.

- The FPGA is unable to save its current state so if the power is turned off the programmed hardware will disappear.
- To program the FPGA with this method all you need to do is connect the USB cable to the development board and ensure that under **Hardware Setup** that USB-Blaster is selected. Then you must go to **Add Files** and add your compiled *lab1.sof* file.
- Press **START** to upload your hardware, in a few moments you should see your development board behaving as instructed by your code.

2. Active Serial

- This method loads the hardware on to the on-board configuration device. What this means is that the hardware description is saved into memory and is loaded onto the FPGA chip whenever the board is powered on. This method is more desirable because it allows the FPGA to work without being connected to the computer and only to an external power source.
- Before beginning this method you should first check that under **Assignments** → **Device** → **Device and Pin Options** are configured in the same way as shown in Figure 11. If not, you must set the configuration scheme to *Active Serial* and also set the configuration device to *EPCS64*. If this was not set you must recompile your hardware.



Specify the device configuration scheme and the configuration device. Note: For HardCopy designs, these settings apply to the FPGA prototype device.

Configuration scheme: Active Serial (can use Configuration Device)

Configuration mode: Standard

Configuration device

[✓] Use configuration device: EPCS64

Configuration device I/O voltage: Auto

[ ] Force VCCIO to be compatible with configuration I/O voltage

[✓] Generate compressed bitstreams

Active serial clock source:

[ ] Enable input tri-state on active configuration pins in user mode

Description:

The method used to load data into the device. Up to four configuration schemes are available, depending on the selected device: Passive Serial (PS); Fast Passive Parallel (FPP), Active Parallel (AP) and Active Serial (AS).

Figure 11: Active Serial Configuration Settings

- Next once again ensure that the hardware is set to *USB-Blaster* and that the mode is set to *Active Serial*
- Click on **Add Files**, and select *lab1.pof.*
- Ensure that the development board is switched to *PROG.*
- Click **START** to begin programming. This method takes slightly longer.
- Switch the board back into the *RUN* position and verify that your logic is behaving properly.

1.5 Activities

1.5.1 Implementing Logic

Implement the hardware from the circuit in Figure 12. The inputs should come from SW(1) and SW(2) and the output should be shown on any of the available LEDs. Use the implemented circuit to test and create a truth table with your results and place it within a comment in the program file.

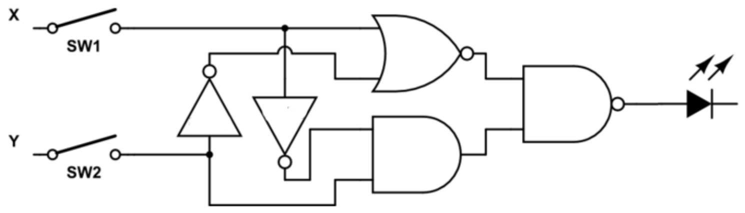
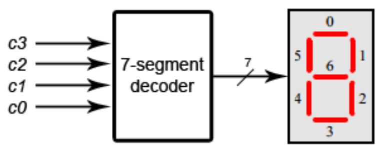


Figure 12: Circuit for activity 1

1.5.2 7 Segment Display Decoder

The 7-segment display is comprised of 7 LEDs that are arranged in such a way that allows for the creation of the numbers 0-9 and a select few characters with some clever use. Figure 13 shows the block diagram and output table. Your task is to create a 4 input, 7 output decoder that will display a number from 0-9 and the letters A-F. To accomplish this task, you should program the switches SW(3) - SW(0) to act as a 4-bit input to the decoder and output the result across all of the displays, HEX7 - HEX0.



c3c2c1c0	character	*c3c2c1c0*	character
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Figure 13: 7 segment display and decoder

Tips:

- You should create two entities, one labled part2 that contains the logic for the switch input and output to the displays and another labled bcd7seg that acts as a decoder for the display.
- The eight 7 segment displays can be accessed with the 7-bit signal vectors HEX7...HEX0. For example, to output to the first display (HEX0) you can either set each bit individually (HEX0(5) <= '0';) or set the whole vector with (HEX0 <= '00000000') which would display the number 8. Keep in mind that the LED segments use inverted logic.
- Figure 14 shows how to correctly display all the required characters.

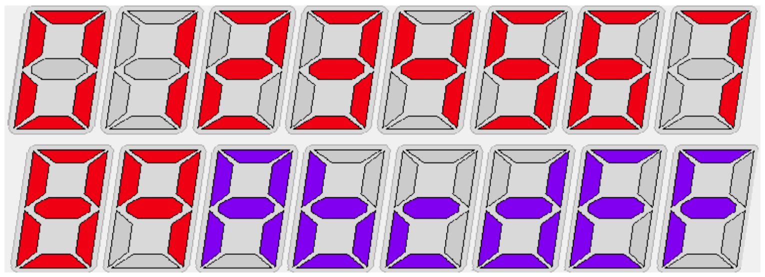


Figure 14: 7 segment displays showing all combinations for 0-9 and A-F

1.6 Lab Report

Your lab report should be upload to Sakai in a zip folder that includes; your commented VHDL code, your VHDL test bench, a pdf of your waveforms, and a text file answering any questions from the activities.

2 Lab 2 - Latches, Flip-Flops, and Counters

2.1 Introduction

Elementary latches and flip-flops have been used for years as a means to store temporary data either from the outputs of logic operations or by setting them to configure logic to behave in certain ways. This lab will go into the aspects of creating latches and flip-flops which will then be used to create a counter.

2.2 Pre-lab

Before you begin this lab you should complete the following and upload to Sakai:

- Write down the truth table for a D-latch, SR-latch and J-K flip-flop
- Design a block diagram for an 8-bit synchronous counter using J-K flip-flops

2.3 Lab Activities

2.3.1 Latches

The following VHDL code implements the logic for a D-latch based off of the schematic in Figure 15.

```vhdl
-- A gated D latch
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dlatch IS
     PORT (    Clk, D     : IN STD_LOGIC;
               Q, Qbar    : OUT     STD_LOGIC);
END dlatch;

ARCHITECTURE rtl OF dlatch IS

     SIGNAL D1, D2, Qa, Qb : STD_LOGIC; -- Intermediate signals
     ATTRIBUTE keep: boolean; -- For waveform results
     ATTRIBUTE keep of D1, D2, Qa, Qb : signal is true;

BEGIN

     D1 <= NOT (D AND CLK);
     D2 <= NOT (D1 AND CLK);
     Qa <= NOT (D1 AND Qb);
     Qb <= NOT (D2 AND Qa);

     Q <= Qa;
     Qbar <=Qb;

END rtl;
```

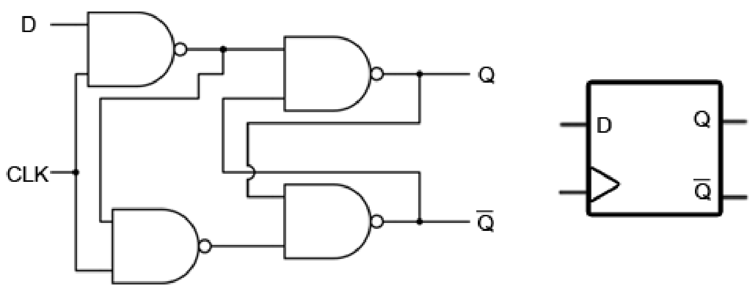


Figure 15: D-latch circuit and block diagram

Using this as a reference, design VHDL for an SR-latch with a clock input. Verify with waveforms that the circuit behaves the same as the truth table you created in the pre-lab.

2.3.2 Flip-Flop

Design VHDL code that implements the logic for a J-K flip-flop from Figure 16. Verify with waveforms that the circuit behaves the same as the truth table you created in the pre-lab

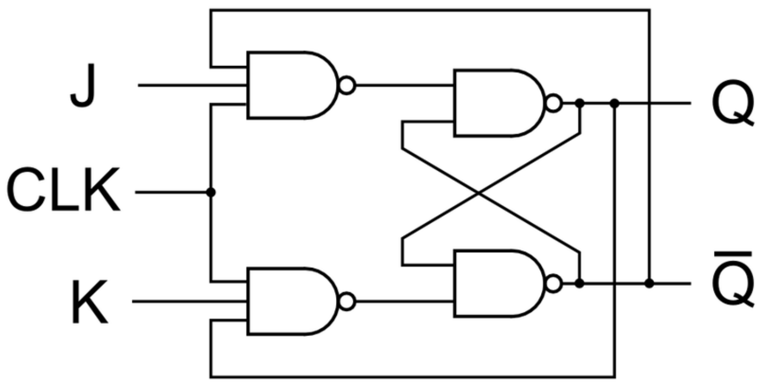


Figure 16: J-K fip-flop circuit

2.3.3 Counters

In the pre-lab, you created a block diagram for an 8-bit counter using J-K flip flops. Using the same VHDL code you created for implementing the J-K flip-flop, implement an 8-bit counter that increments when you press KEY0 on the DE2-115 development board. Link the binary output of the flip-flops to the red LEDs and then convert the binary value into hexadecimal to be shown on the 7-segment displays. *Hint: you should refer to your code for the 7-segment display driver designed in the previous lab.*

2.4 Lab Report

Your lab report submission should break down as follows:

- Extract from the fpga lab folder the VHDL file from each project, upload each of them to the Sakai Assignment page. Ex: part1.vhdl and part2.vhdl
  - Make sure that your code is well commented.

- Waveforms for the SR latch and JK flipflop (if JK doesn't work, try implementing it as a truth table with if statements)
- For Part 3 after compilation go to Tools > Netlist Viewers > RTL Viewer. Print a pdf of this page and discuss what you see.
- In a separate txt or pdf document, prepare a discussion on your compilation results. Be sure to include specifics about the amount of hardware used and where the numbers in the compilation results came from. Do the numbers make sense? Why? How do these results impact the use of FPGAs in industry? This should be no more than one page long, less is preferred.

3 Lab 3 - Complex Addition Systems

3.1 Introduction

From your pocket calculator to inside modern CPUs adders have long been used as more than just a simple way to sum numbers together. This lab will go into the logic structure of the adder as well as provide a method for converting the simple full adder in to an *arithmetic logic unit* (ALU) capable of handling 12 operations.

3.2 Pre-lab

Before coming to the lab, please complete the following and upload to Sakai:

- A truth table for a half adder and a full adder
- The logic equation for a 4-bit ripple carry adder

3.3 Lab Activities

3.3.1 Half Adder

Build the circuit in Figure 17, create waveforms to verify that the logic is correct.

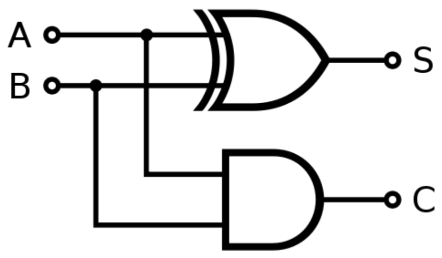


Figure 17: Circuit for a 1-bit half adder

3.3.2 Full Adder

The circuit in Figure 18 implements a full 1-bit adder. Implement this circuit in VHDL, create a waveform, and verify that the logic behaves as expected.

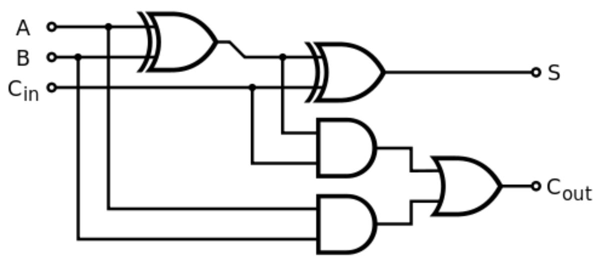


Figure 18: Circuit for a 1-bit full adder

Now that you have a working 1-bit full adder, implement a 4-bit ripple carry adder that sums the binary numbers "0110" and "0101." A block diagram for the 4-bit ripple carry adder is shown in Figure 19. Verify your results by creating a waveform and simulating the circuit.

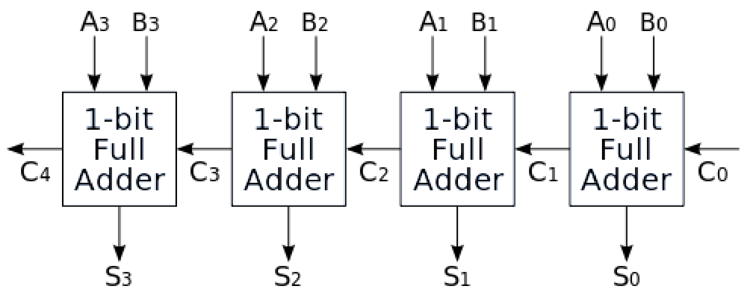


Figure 19: 4-bit ripple carry adder block diagram

3.3.3 Full Adder Based ALU

The block diagram in Figure 20 is an example of how the regular 1-bit full adder can be manipulated to implement additional functionality. For this activity, you must build VHDL code that implements a 4-bit complex adder ALU, the list of instructions can be found in Table 1.

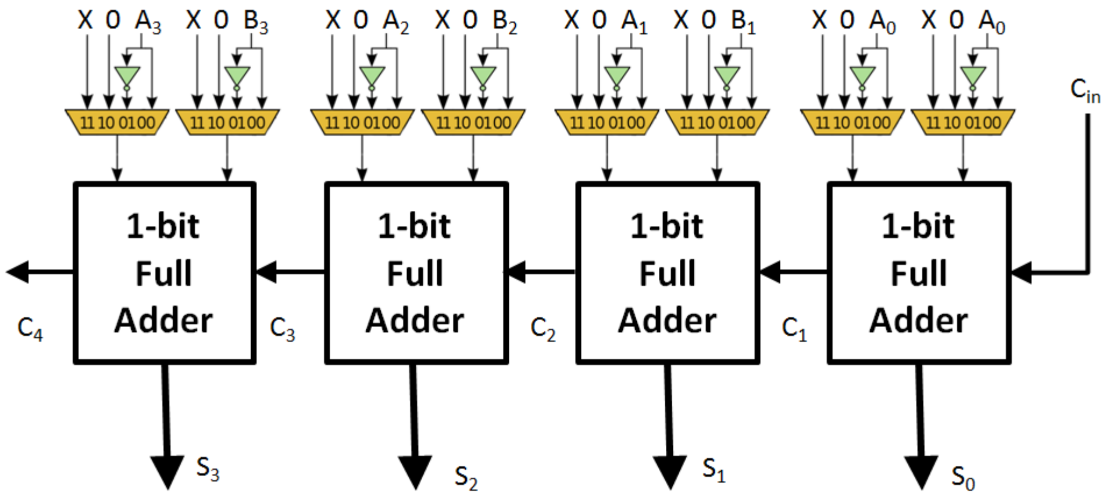


Figure 20: Block diagram for a 4-bit ripple carry adder ALU with 12 operations

After writing and testing your VHDL code, upload it on the DE2-115 FPGA Development board. Connect the inputs **A3-A0** to SW3 - SW0 and the inputs **B3-B0** to SW4(7)- SW(4). Connect the select lines for the **A** multiplexer to SW(8) and SW(9) while the select lines for **B** should connect to SW10 and SW11. Lastly, the carry in input should connect to SW12. Display inputs **A** and **B** in hexadecimal on HEX7 and HEX5, respectively and the output **S**, on HEX3. If the result over-flows display **C_4** on LEDG0. If the result is zero turn on LEDR1. Test and verify all twelve operations are correct and make a table that includes the values for **A**, **B**, **S**, **C_4**, and **Zero** for each operation.

A$_i$	**B**$_i$	**Carry In**	**Result**
Set to 0	Set to 0	0	0
Set to 0	Set to 0	1	1
A	Set to 0	0	A
Set to 0	B	0	B
A	Set to 0	1	A + 1
Set to 0	B	1	B + 1
A	B	0	A + B
Set to invert	B	1	B − A
Set to invert	Set to 0	0	$\overline{A}$
Set to invert	Set to 0	1	−A
Set to 0	Set to invert	0	$\overline{B}$
Set to 0	Set to invert	1	−B

Table 1: List of opperations for the adder based ALU

When you complete building the VHDL upload your code to the FPGA board. Test and verify all twelve operations are correct.

3.4 Lab Report

Your lab report submission should break down as follows:

- Extract from the fpga lab folder the VHDL file from each project, upload each of them to the Sakai Assignment page. Ex: part1.vhdl and part2.vhdl
  - Make sure that your code is well commented.
- Waveforms for the Half Adder and Full Adder.
- For Part 3 after compilation go to Tools > Netlist Viewers > RTL Viewer. Print a pdf of this page and discuss what you see.
- Your report should include a one page discussion on the number of pins used and your table from part 3.

4 Lab 4 - Finite-State Machines

4.1 Introduction

A *finite-state machine* (FSM) is a model for sequential logic that consists of a set of states (including an initial or reset state) a set of inputs/outputs and a transition function that shows the transition from one state to other states. The state the FSM is in at any given time is called the *current* state. The FSM can change from one state to another when initiated by a triggering event or condition (input); this is called a transition. This lab will examine a simple FSM to give the general idea of how they work, and build upon it to create a system modeled off of a real world example, a vending machine.

4.2 Pre-lab

Before coming to the lab, please complete the following activity and upload the results to Sakai.

- By examining the code in Listing 1 write out its state machine transition graph.

4.3 Lab Activities

4.3.1 FSM

Given the state machine transition graph in Figure 21, design VHDL code for the state transitions of this diagram. Note that in the state transition graph, we show the transition behavior as input/output where the input is denoted as D_1D_0 and the output is denoted as Z_1Z_0. For example 00/01 means D_1D_0 = 00 and causes Z_1Z_0 = 01. Assume that state A is the reset state for this machine.

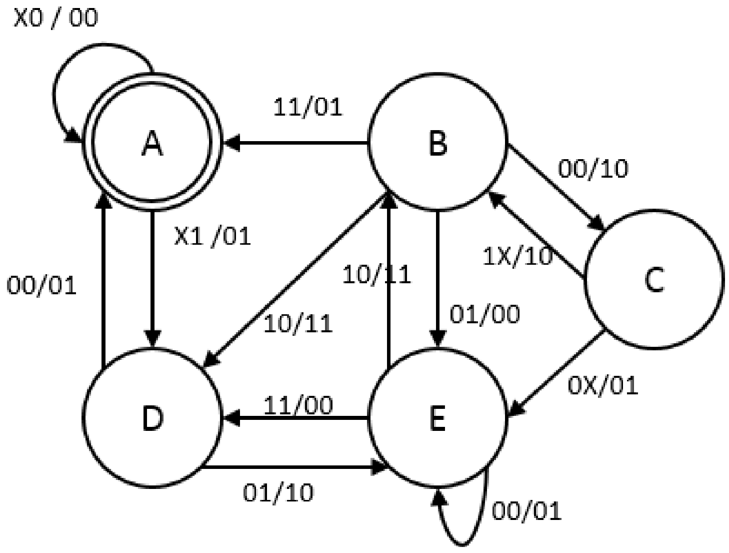


Figure 21: State machine transition graph

The code below shows a simple example of a FSM. You can build upon this code in completing this lab.

Listing 1: Sample code for a Finite State Machine

```
-- User-Encoded State Machine
library ieee;
use ieee.std_logic_1164.all;

```

```vhdl
entity state_machine is
    port(clk        : in std_logic;
         reset       : in std_logic;
         input       : in std_logic;
         output      : out std_logic);

end entity;

architecture rtl of state_machine is
    -- Build an enumerated type for the state machine
    type count_state is (A, B, C, D);

    -- Registers to hold the current state and the next state
    signal present_state, next_state      : count_state;

    -- Attribute to declare a specific encoding for the states
    attribute syn_encoding                        : string;
    attribute syn_encoding of count_state : type is "11 01 10 00";

begin
    -- Move to the next state
    process(clk, reset)
    begin
        if reset = '1' then
            present_state <= A;
        elsif (rising_edge(clk)) then
            present_state <= next_state;
        end if;
    end process;

    -- Determine what the next state will be, and set the output bits
    process (present_state, input)
    begin
        case present_state is
            when A =>
                if (input = '0') then
                    next_state <= B;
                    output <= '0';
                else
                    next_state <= D;
                    output <= '0';
                end if;
            when B =>
                if (input = '0') then
                    next_state <= C;
                    output <= '1';
                else
                    next_state <= A;
                    output <= '0';
                end if;
            when C =>
                if (input = '0') then
                    next_state <= D;
                    output <= '0';
                else
                    next_state <= B;
                    output <= '1';
                end if;
            when D =>
                if (input = '0') then
                    next_state <= A;
                    output <= '0';
                else
                    next_state <= C;
                    output <= '1';
                end if;
        end case;
    end process;

end rtl;
```

4.3.2 Vending Machine

Design a custom finite-state machine to control a vending machine to dispense products. The design has the following specifications:

1. The state machine should have three states:

   - *IDLE*: This is the default (reset) state for the machine. The machine should stay in this state until at least one product is selected. Being that this is the reset state, you should initialize the signals QUARTERS, COST, and DISPENSE_READY to zero. Make sure that LEDR0 to LEDR17 along with LEDG1 to LED8 are set to off. While in this state, you should display a dash across all HEX displays.
   - *PRODUCT_SELECT*: The state machine will move into this state if product(s) are selected through use of the switches (SW0 - SW15) on the FPGA board. Display the total cost in number of quarters needed of all products selected on HEX5 - HEX4 in hexadecimal. KEY3 and KEY2 should increment QUARTERS by a dollar and quarter respectively when pressed. Display the value of QUARTERS on HEX1 - HEX0. When the correct number of quarters have been inserted, the signal DISPENSE_READY should go HIGH (which should be shown on LEDG8) and the state should transition to DISPENSE. A list of available products and their cost can be found in Table 2.
   - *DISPENSE*: Once the proper amount of quarters have been deposited, the item should be dispensed from the machine. To show that the item(s) have been dispensed, turn on LEDR0-LEDR17 and set the next state to idle.

2. KEY0 will act as a CLOCK input and KEY1 will act as a coin return or rest. The states should transition on the rising edge. This means that you should select a product and then send a CLOCK pulse to calculate the cost of the item(s) selected. Then add quarters into the machine and send another CLOCK pulse when there is the right amount of quarters. If there is not enough quarters inserted, the state should not change. Once the item(s) are dispensed, send another CLOCK pulse to return to IDLE.

3. The current state; *IDLE*, *PRODUCT_SELECT*, and *DISPENSE* should be displayed by turning on LEDG0 - LEDG2 respectively.

4. Both signals QUARTERS and COST should be of the type INTEGER with a range of 0 to 40. You will need to include the package *ieee.numeric_std.all*.

Product	**Cost**	**Switch**
SODA_CAN[0:3]	$1.00	SW12 - SW15
CHIPS[0:3]	$0.75	SW8 - SW11
CHOCOLATE[0:3]	$0.50	SW4 - SW7
BUBBLE_GUM[0:3]	$0.25	SW0 - SW3

Table 2: List of vending machine items, cost, and switch correspondence

4.4 Lab Report

After completing the activities in this lab you should create a zip folder with the following and then submit it to Sakai:

- Commented VHDL code.
- Waveform for the FSM activity.
- Pictures of the results from the vending machine activity.
- A discussion on the results of compilation including longest path delay, the total number of logic elements used, and issues you encountered while performing the lab.

5 Lab 5 - A Simple Processor

5.1 Introduction

In this lab you will create a simple processor that can compute 8 operations. Figure 22 shows the components of a simple processor. The processor consists of an ALU, a FLASH memory, a control unit, a program counter (PC), and a display.

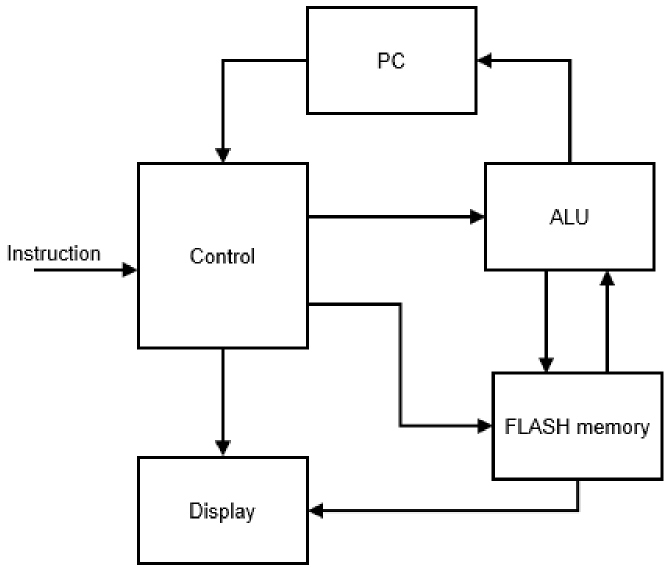


Figure 22: Block overview of processor

5.2 Pre-lab

Before coming to the lab, please complete the following and upload to Sakai:

- Read through the lab handout (specially Section 5.3.1) and construct a top level finite state machine transition graph for the processor. Note that the different states are already defined in Section 5.3.1.
- Read through the datasheet for working with the FLASH memory chip on the FPGA board. This can be found under Sakai Resources.

5.3 Lab Activities

5.3.1 CPU Controller

Instructions are fed to the CPU controller as an 18 bit instruction IR, the format of an instruction can be seen in Table 3.

OPCODE	Destination Address	Source Address 1	Source Address 2 / Shift Amount
3 bits	5 bits	5 bits	5 bits

Table 3: 18 bit instruction format

The Control module should take apart the instruction IR and break it down into signals to be used by the other modules. It is best to think of Control as a "Top" level state machine for the entire processor. The processor should have the following states *FETCH*, *DECODE*, *EXECUTE*, *MEMORY_WRITE*. Note that during each state, the display will shows the relative information to that state, given in Section 5.3.6.

The device should initiate in the *FETCH* state by sending an active low RESET signal in your test bench. If RESET is HIGH, take in the instruction IR and proceed to the *DECODE* state.

If the device is in the *DECODE* state, you should break down the instruction IR into the various signals required for the modules ALU and FLASH controller, you should also read in the data stored from Source Address 1 and Source Address 2 and store it into a register. Once this is completed, proceed to the *EXECUTE* state.

In the *EXECUTE* state, you should preform the ALU operation as determined from the processor instruction and store the result into a register. Once completed, Increment the Program Counter and move to the next state *MEMORY_WRITE*.

After the instruction is completed, take the result and write it into the Destination Address on the FLASH memory. Proceed back to *FETCH* and begin working on the next instruction.

Each state should transition on the positive edge of a clock pulse which will come from pressing KEY0.

5.3.2 Design an ALU

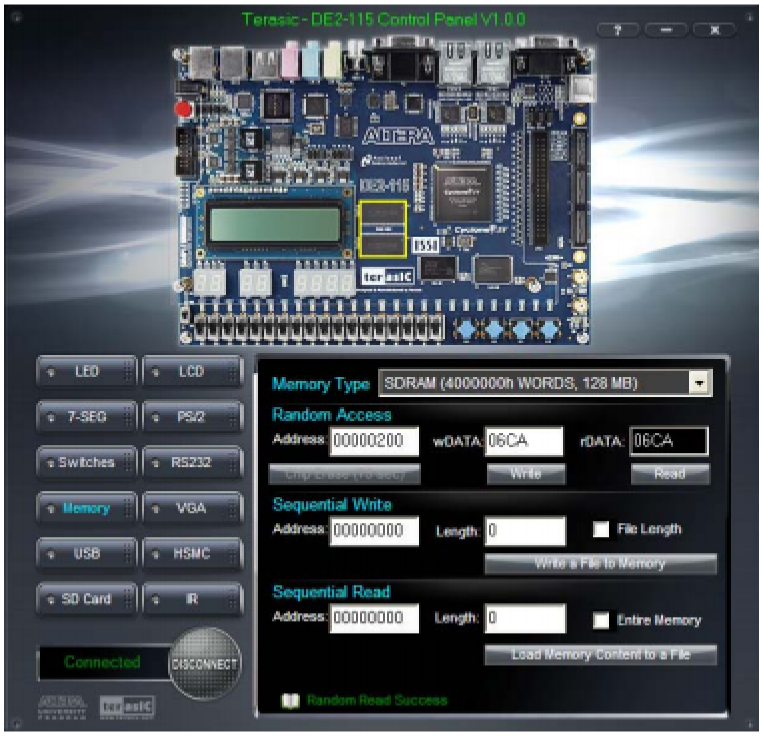
Using the operations listed in Table 4, create an ALU in VHDL that takes in two 8 bit values and returns the result to the processor. Required signals: **A**, **B**, **opcode**, and **ALU_out**. Note that the inputs **A** and **B** should be read from the FLASH memory controller , the **ALU_out** is a shift register in the ALU where the result of the operation will be stored prior to the *MEMORY_WRITE* state and the **opcode** comes from IR.

OPCODE	Instruction
000	AND
001	OR
010	NAND
011	NOR
100	XOR
101	ADD
110	SUB
111	Shift Right Logical

Table 4: List of ALU operations

5.3.3 Initiate the FLASH Memory

Using the DE2-115 Control Panel, initiate arbitrary values into the FLASH memory and keep a record of this data to compare your results. Figure 23 shows the option screen for loading values.



Memory Type: SDRAM (4000000h WORDS, 128 MB)

Random Access
Address: 00000200 wDATA: 06CA rDATA: 06CA

Sequential Write
Address: 00000000 Length: 0 ☐ File Length

Sequential Read
Address: 00000000 Length: 0 ☐ Entire Memory

Random Read Success

Figure 23: DE2-115 Control Panel - Accessing SDRAM/FLASH/EEPROM Menu

5.3.4 FLASH Memory Controller

The Altera DE2-115 FPGA Development Board contains an 8 MB FLASH memory chip. For this activity you will implement a FLASH memory controller that can read and write to the first 256 bits of the chip. A list of pins used for interacting with the chip can be found in Table 5. Notice that addresses are normally 23 bits long, however for this exercise you should set FL_ADDR[19] to FL_ADDR[22] to the source address and set the remaining bits to zero. You should pay careful attention to how data is read and written to the chip. The signal FL_CE_N should be kept HIGH to keep the chip enabled. If you want to read data from the chip, you must first tell the chip the address of where the data is stored and then set the signal FL_OE_N to HIGH, in order to send the data onto the data lines. Once the data is read, set the signal FL_OE_N back to LOW. If you want to write data to the chip, you must tell the chip the address location, and set the data on the data lines then set FL_WE_N to HIGH. When the operation is completed, make sure to set the signal FL_WE_N to LOW. NOTE: you should never set both FL_OE_N and FL_WE_N to HIGH at the same time as you may damage the device. FL_RY may be useful to determine if an operation has been completed.

Signal	Description
FL_ADDR[0] to FL_ADDR[22]	FLASH Address[0] to FLASH Address[22]
FL_DQ[0] to FL_DQ[7]	FLASH Data[0] to FLASH Data[7]
FL_CE_N	FLASH Chip Enable (set HIGH)
FL_OE_N	FLASH Output Enable
FL_RST_N	FLASH Reset (set LOW)
FL_RY	FLASH Ready/Busy output
FL_WE_N	FLASH Write Enable
FL_WP_N	Flash Write Protect (set LOW)

Table 5: Signal assignments for interfacing with the flash memory chip

5.3.5 Program Counter

Design a simple program counter that keeps track of the number of operations completed and increments by one at the end of MEMORY_WRITE. Store the result to a register to be shown through the display driver.

5.3.6 Results Display

As has been done in previous labs, create a display driver for showing information relative the the current state.

1. LEDG3 through LEDG0 should display the current state; FETCH, DECODE, EXECUTE, and MEMORY_WRITE respectively.
2. *FETCH*
   - Display a dash "-" across all displays.
3. *DECODE*
   - HEX7 and HEX6 should display the Source Address 1 from IR.
   - HEX5 and HEX4 should display the Source Address 2 from IR.
   - HEX3 should display the ALU operation from IR.
   - HEX0 should display the current Program Counter.
4. *EXECUTE*
   - HEX7 and HEX6 should display the value stored in Source Address 1.
   - HEX5 and HEX4 should display the value stored in Source Address 2.
   - HEX3 and HEX2 should display ALU_out the result of the operation.
   - HEX0 should display the current Program Counter.
5. *MEMORY_WRITE*
   - HEX7 and HEX6 should display the Destination Address.
   - HEX3 and HEX2 should display the value stored in the Destination Address.
   - HEX0 should display the incremented Program Counter.

5.4 Lab Report

After completing the activities in this lab you should create a zip folder with the following and then submit it to Sakai:

- Commented VHDL code.
- VHDL test bench with at least one instruction for all 8 operations.
- Photos of results from the test bench
- This processor is single-cycle which means it can only run one instruction at a time, how would you make the processor run in parallel? How many instructions can theoretically be worked on at the same time? Provide a diagram along with a written response.
- A discussion on the results of compilation including longest path delay, the total number of logic elements used, and issues you encountered while performing the lab.