Lab 1: Introduction to Digital Design using SystemVerilog and FPGA

The University of Washington | The Remote Hub Lab | Last Revised: September 2022

Summary

This lab introduces the Hardware Description Language (HDL) SystemVerilog. In this lab, you will learn about 1) Quartus for SystemVerilog synthesis, 2) ModelSim for testing and simulation, and 3) Writing code to Field Programmable Gate Arrays (FPGAs).

Pre-Lab

- Lab 0 to create an account on the remote lab platform.

Table of Contents

Task 1: Creating a schematic diagram and simulating a full adder in Quartus

The objective of this task is to see how circuits are constructed using logic gates if you were to build a circuit on a breadboard. Digital design, however, involves many components and therefore implementing digital circuits are done using Field Programmable Gate Arrays (FPGAs) not by breadboards.

To demonstrate the concept of schematic diagrams, diagrams that would be used to build small circuits on a breadboard, this task will be done in Quartus. In this task you will follow two video tutorials on creating a schematic diagram for a full adder circuit and simulate it with timing diagrams. Please note that these tutorials were created using an older version of Quartus and so to adjust to version 17.0, you will need to follow the document called **"schematic-tutorial"** on canvas which includes the links to the video tutorials. Note that this will be the only time we will use these tutorials and they are meant to demonstrate the concept of schematic diagrams. In this course we will be writing Verilog code to implement designs and will use ModelSim for simulation.

What is a Field Programmable Gate Array (FPGA)?

FPGAs are hardware chips with programmable logic cells -- electrical components which a designer can use to customize and configure circuits for specific purposes. Digital circuits typically consist of a large amount of logic components which necessitates a way to reconfigure circuits on scale, and this is why FPGAs are used.

To program an FPGA, we need to use a Hardware Description Language (HDL). This lab series uses an HDL called SystemVerilog, but note that there are other HDLs such as VHDL. After writing and synthesizing the code, it is good practice to simulate the output. This allows designers to check if the code results in what they expect before sending it to the FPGA. Sending a design to an FPGA means converting the HDL code into bitstreams (0s and 1s) that are understandable by the machine. This workflow is demonstrated in Figure 1.

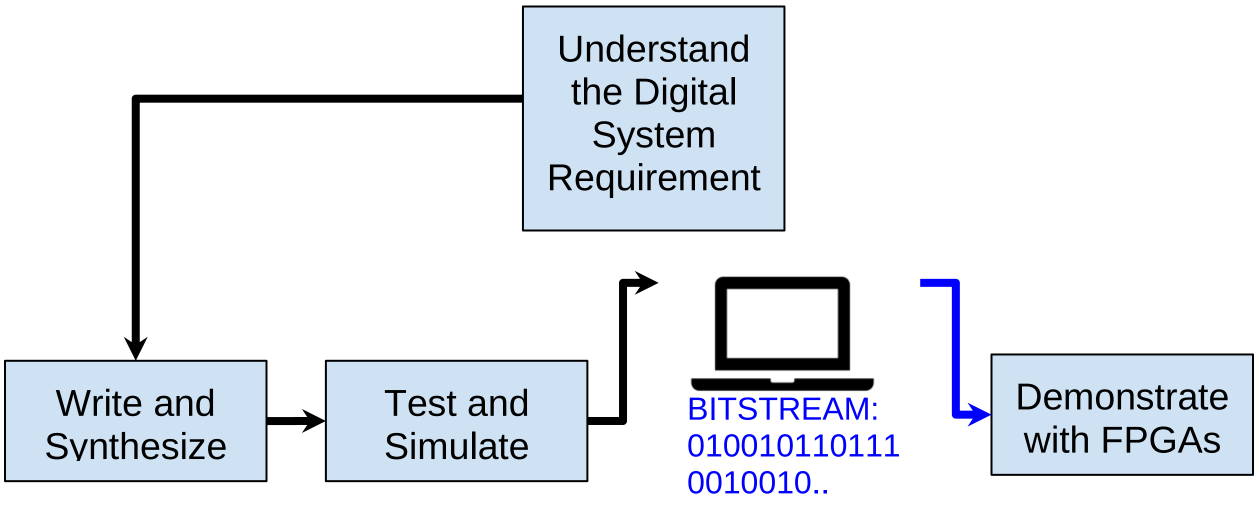

Understand the Digital System Requirement
Write and Synthesize
Test and Simulate
BITSTREAM: 010010110111 0010010..
Demonstrate with FPGAs

Figure 1: Understanding → SystemVerilog → Synthesis → Simulation→ FPGA Workflow

In the next task, we'll walk through each step of the workflow as depicted in Figure 1. We will use Quartus and ModelSim to write, verify and synthesize code before executing the code on the remote lab.

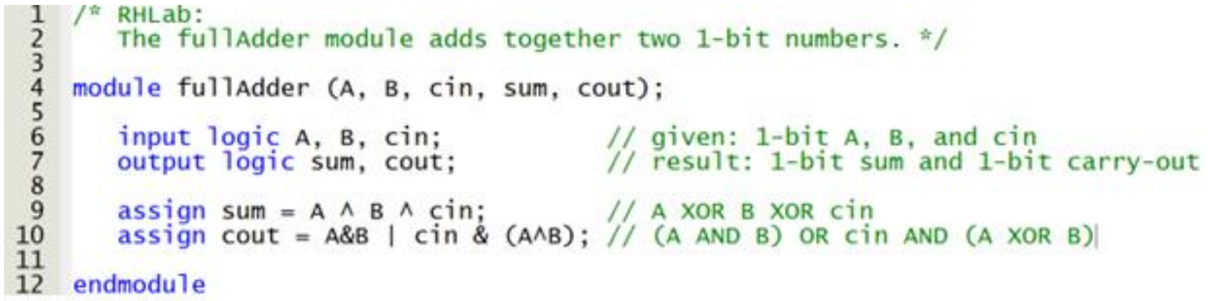
Task 2: Implementing Digital Logic Using a Hardware Description Language

Description

You should come to realize from task 1 that it is a tedious task to build circuits physically on breadboards especially as designs become more complicated. In this task we will do the same task we did in task1 but using SystemVerilog and ModelSim.

Creating a digital design involves three steps: Describing the design using HDL with a software like Quartus, verifying the design with a tool like ModelSim, and then implementing the verified design on an FPGA. In this section, we will introduce the major SystemVerilog Syntax through a full adder example which will suffice for the needs of this lab.

A full adder performs bitwise addition using two bits (A, B) and a carry-in value (cin), resulting in sum and carry-out (cout) outputs. Figure 2 shows an example of a SystemVerilog program which creates a full adder unit.



```
 1  /* RHLab:
 2     The fullAdder module adds together two 1-bit numbers. */
 3
 4  module fullAdder (A, B, cin, sum, cout);
 5
 6     input logic A, B, cin;          // given: 1-bit A, B, and cin
 7     output logic sum, cout;         // result: 1-bit sum and 1-bit carry-out
 8
 9     assign sum = A ^ B ^ cin;       // A XOR B XOR cin
10     assign cout = A&B | cin & (A^B); // (A AND B) OR cin AND (A XOR B)
11
12  endmodule
```

Figure 2: Full Adder Module

SystemVerilog Syntax and Main Components

general notes

- Notice how most lines in SystemVerilog except for “endmodule” end with a semicolon (**;**).
- Notice the indents and white space left between lines for readability.

modules

- Think of this like the “container” for your code. A module begins with the starting line syntax “module [name()]”. A module ends with the line “endmodule.”
- See lines 4 and 12 in Figure 2.

input and output logic

- These lines are akin to declaring your variables. You list your input signals after the syntax “input logic” , and you list your output signals after the syntax “output logic” respectively.
- See lines 6 and 7 in Figure 2.

assign statements

- assign statements define the variables’ meaning with Boolean expressions.
- See Figure 2’s lines 9-10 which use assign statements as supposed to lines 6-7’s variable declarations.

comments

- Leave comments in your code for readability. Use one double forward slash at the beginning of in-line comments (//) and a single forward slash and star at the beginning and end of multi-line comments (/*...*/).
- In Figure 2, see lines 6-10 for in-line comments and lines 1-2 for multi-line comments.

Defining Logic functions with Bitwise operators

- Bitwise operators are used to define logic functions such as the sum and cout equations in lines 9 and 10 of figure 2. Table 1 summarizes the bitwise operators used in SystemVerilog

***Table 1**: Bitwise Operators*

Operator	Syntax Symbol	Boolean Expression → SystemVerilog Example
AND	&	$A\ AND\ B \to A\ \&\ B$
OR	|	$A\ OR\ B \to A \mid B$
NOT	~	$\underline{A} \to \sim A$
NOR	(combination)	$A\ NOR\ B \to \sim(A \mid B)$
NAND	(combination)	$A\ NAND\ B \to \sim(A\ \&\ B)$
XOR	^	$A\ XOR\ B \to A\ \hat{}\ B$

Note: Do not use ‘+’ for the OR operator, and do not use ‘x’ for the AND operator in SystemVerilog code. They mean different things in SystemVerilog.

Quartus, ModelSim, and LabsLand Activity

The following steps will provide a walkthrough of the Understanding → SystemVerilog →

Synthesis → Simulation → FPGA workflow using the full adder example.

You will follow along a series of video tutorials that will walk you through the steps of creating your first design using SystemVerilog and simulating the design in ModelSim. For your reference, the source code used in the tutorials is in the appendix of this document.

1. **Understand the system requirements.**
   a) Before coding, we need to understand the system we will be implementing.
   b) One way to accomplish this first step is through truth tables and schematic diagrams. You created the schematic in task 1, however, you will not need to create a schematic diagram for every design before you implement it in SystemVerilog. A high level block diagram that puts the system components together and how they relate to each other should be enough.

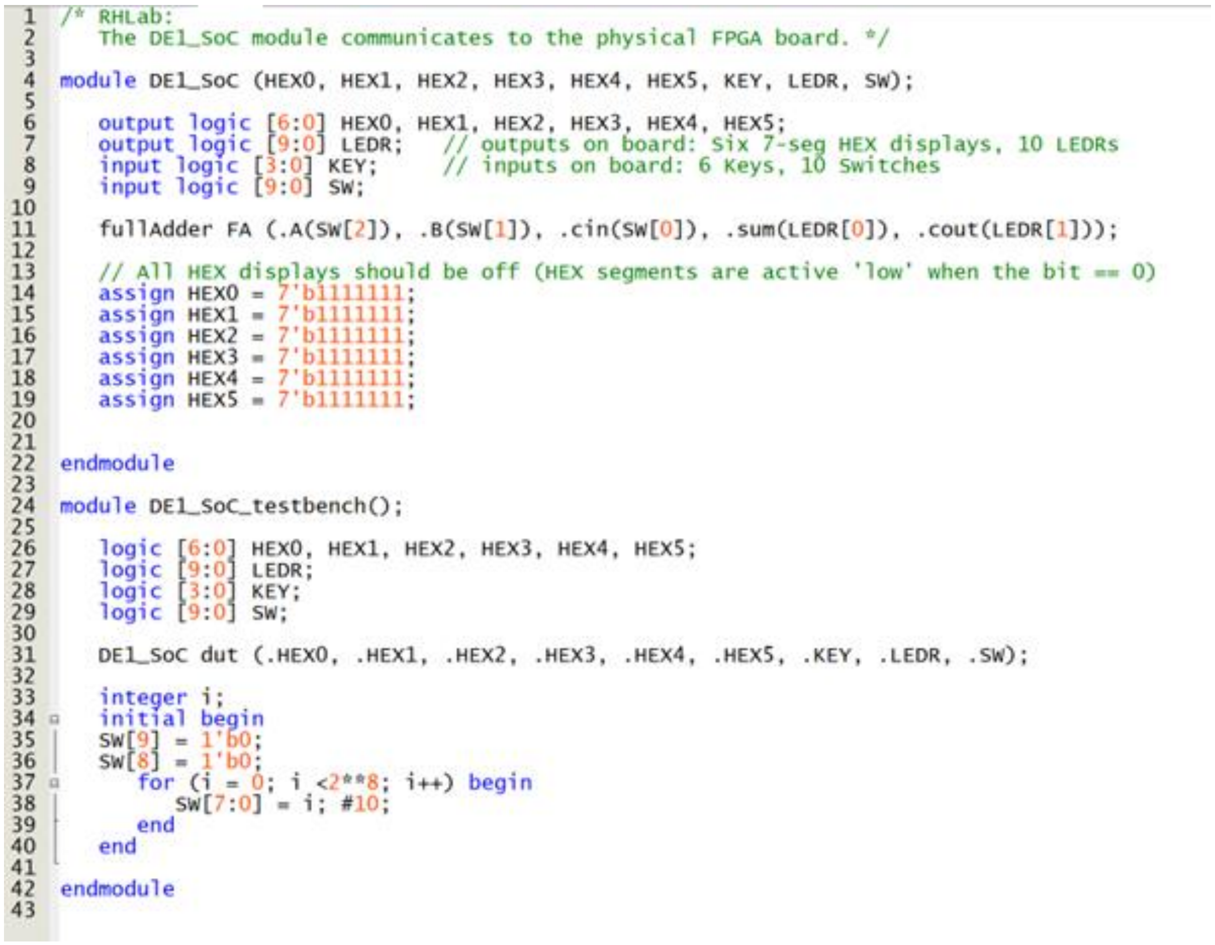
2. **Practice writing SystemVerilog in Quartus, synthesizing your system in Quartus, and simulating your system in ModelSim.**
   This tutorial will help you implement the SystemVerilog code in Figure 2 in addition to writing a verification code to simulate the design in ModelSim. Note that it is intentional that the code in figure 2 is not "copy-and-paste-able" to Quartus; you must type out the syntax of the code to gain familiarity with the (HDL).
   Please follow the tutorials in the following order:
   1. Launch the Quartus Prime software.
   2. Create a project from scratch. Please follow the steps in the following video and use the same project name as in the video https://youtu.be/iLbmSTG7bpA
   3. Implement the full adder using SystemVerilog and simulate it using ModelSim. Please follow the following video: https://youtu.be/BcvclrqZ2fc

   Please note that in the video when it refers to compiling the project for the first time, it may give you a compilation error. If you run the video for few more seconds it 'll tell you about setting the top level modules so that the program compiles.

   · After you successfully simulate the full adder, save a screenshot of the simulation. You can do that by going to File->Export->image and then save the image. You will need to include this image in your lab report as a proof that you went over the tutorials.

3. **Prepare your code for the LabsLand FPGA.**
   a) Follow the instructions in this video tutorial to create and simulate a SystemVerilog module called "DE1_SoC" for your FPGA: https://youtu.be/mnZt2iNNfp4.
   b) Refer to Figure 3 to view the DE1_SoC code for both the design module and testbench.
   c) Note that this tutorial helps you make another module which instantiates the full adder system so we can interact with FPGA inputs and outputs. This hierarchical structure is a common feature of HDL. Because we are making another module, we must synthesize and simulate at this higher level as well.

After you are done with these steps on quartus, close the current project before moving to the next task. You may quit Quartus at this point.



```
/* RHLab:
   The DE1_SoC module communicates to the physical FPGA board. */

module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;    // outputs on board: Six 7-seg HEX displays, 10 LEDRs
   input logic [3:0] KEY;      // inputs on board: 6 Keys, 10 Switches
   input logic [9:0] SW;

   fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

   // All HEX displays should be off (HEX segments are active 'low' when the bit == 0)
   assign HEX0 = 7'b1111111;
   assign HEX1 = 7'b1111111;
   assign HEX2 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   assign HEX4 = 7'b1111111;
   assign HEX5 = 7'b1111111;


endmodule

module DE1_SoC_testbench();

   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0] LEDR;
   logic [3:0] KEY;
   logic [9:0] SW;

   DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW);

   integer i;
   initial begin
   SW[9] = 1'b0;
   SW[8] = 1'b0;
      for (i = 0; i <2**8; i++) begin
         SW[7:0] = i; #10;
      end
   end

endmodule
```

Figure 3: DE1_SoC Module and Testbench

4. **Demonstrate the digital system on an FPGA.**
   a) Login to LabsLand, navigate to your main dashboard with the RHLab activities, and choose “Intel DE1-SoC”. Click the “Access this Lab” button.

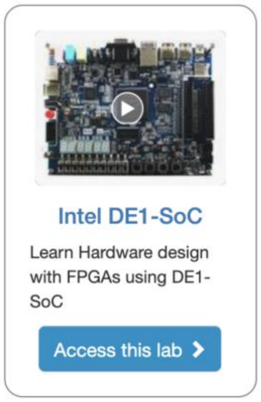
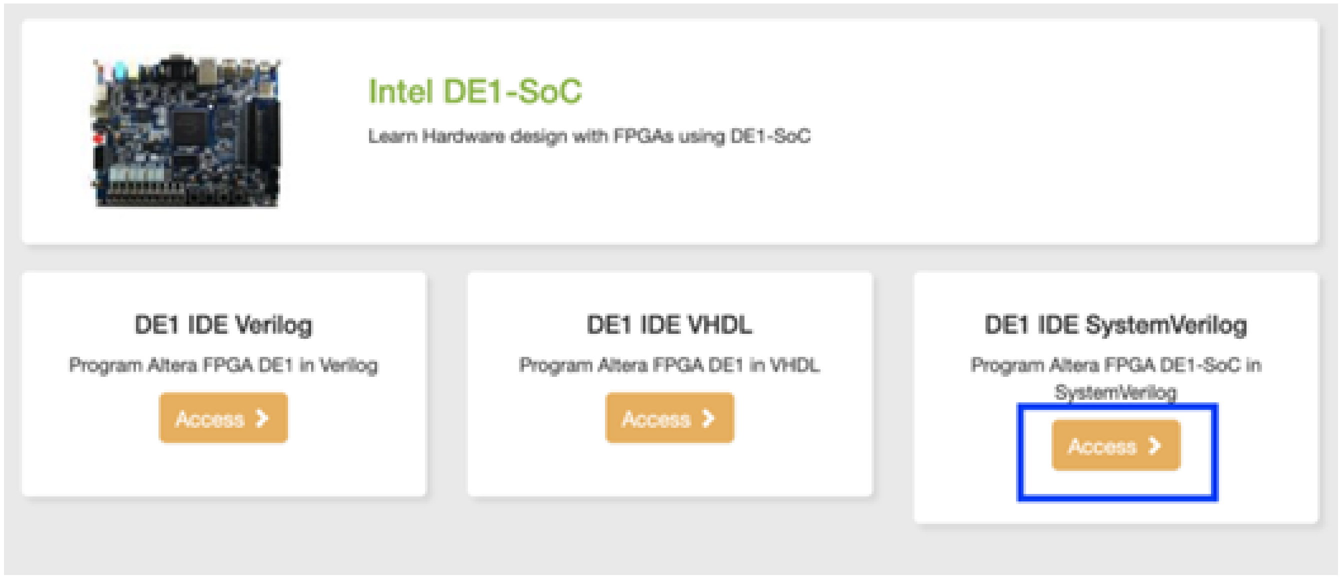


Figure 4: Intel DE1-SoC Activity

b) Locate “DE1 IDE SystemVerilog” and click the “Access” button below it. You will be directed to a new page called “SystemVerilog IDE for DE1-Soc”.



Intel DE1-SoC

Learn Hardware design with FPGAs using DE1-SoC

DE1 IDE Verilog

Program Altera FPGA DE1 in Verilog

Access

DE1 IDE VHDL

Program Altera FPGA DE1 in VHDL

Access

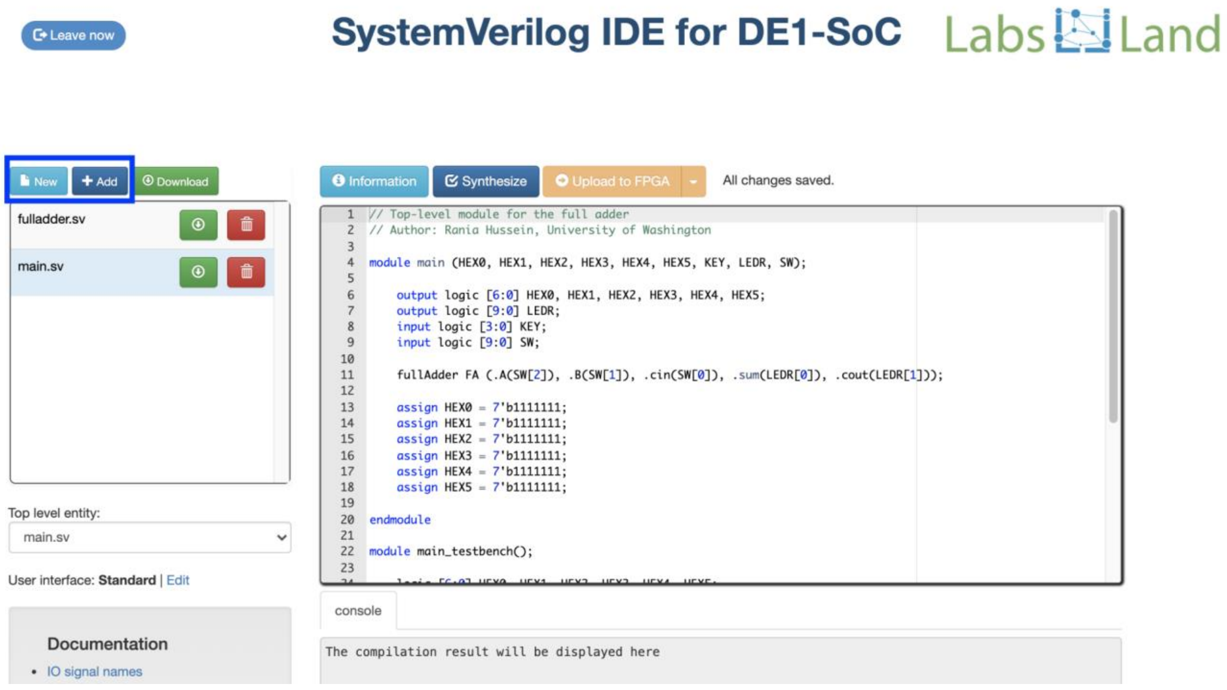
DE1 IDE SystemVerilog

Program Altera FPGA DE1-SoC in SystemVerilog

Access

Figure 5: DE1 SystemVerilog IDE

c) In the following page, select the “Add” button to import the top-module “DE1-SoC.sv” and file “full_adder.sv” that you created earlier into LabsLand.



fulladder.sv

main.sv

Top level entity: main.sv

User interface: **Standard** | Edit

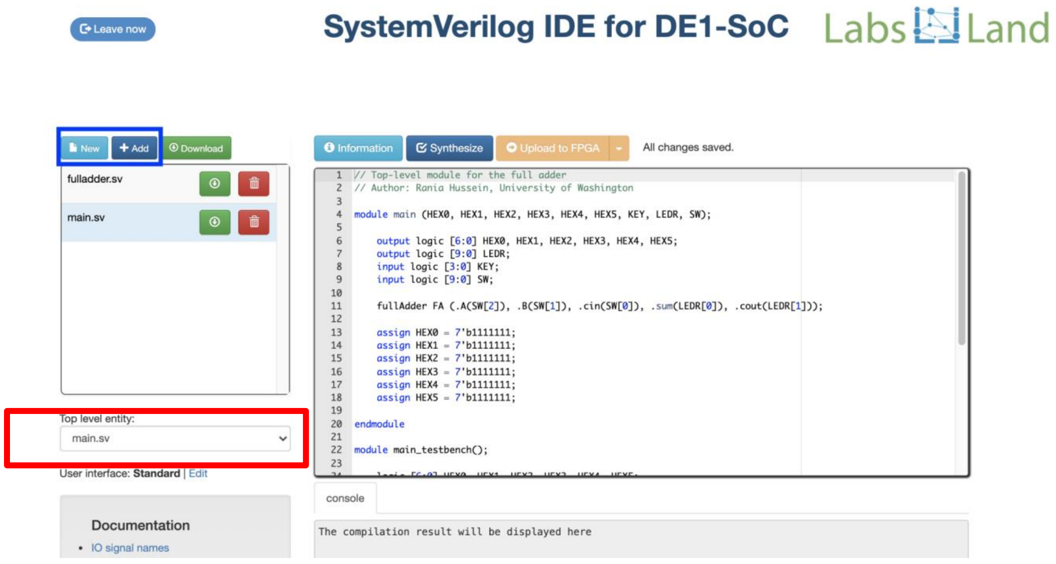
All changes saved.

```
// Top-level module for the full adder
// Author: Rania Hussein, University of Washington

module main (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

    assign HEX0 = 7'b1111111;
    assign HEX1 = 7'b1111111;
    assign HEX2 = 7'b1111111;
    assign HEX3 = 7'b1111111;
    assign HEX4 = 7'b1111111;
    assign HEX5 = 7'b1111111;

endmodule

module main_testbench();

```

```
The compilation result will be displayed here
```

Figure 6: Add Modules into LabsLand

d) Choose the top-module using the dropdown menu under “Top level entity” (boxed in red in Figure 7). Make sure you select “DE1-SoC.sv” as the top-module.



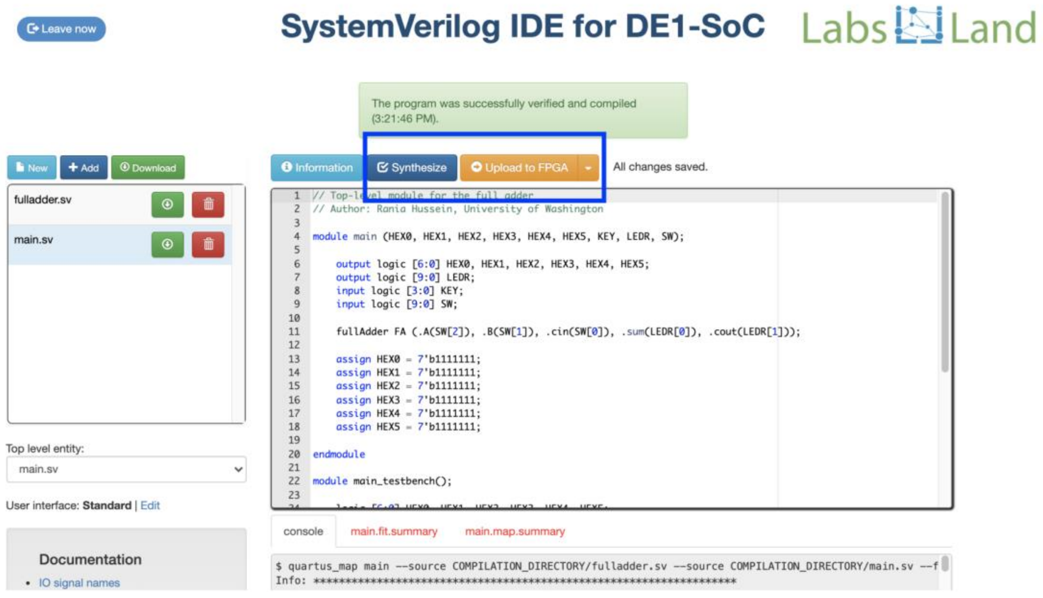
```
// Top-level module for the full adder
// Author: Rania Hussein, University of Washington

module main (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

    assign HEX0 = 7'b1111111;
    assign HEX1 = 7'b1111111;
    assign HEX2 = 7'b1111111;
    assign HEX3 = 7'b1111111;
    assign HEX4 = 7'b1111111;
    assign HEX5 = 7'b1111111;

endmodule

module main_testbench();
```

```
The compilation result will be displayed here
```

Figure 7: Setting the Top Level Module

(Alternatively, you may also create new files by clicking “New” and copy the provided full adder example under “Examples” found in the bottom left corner of the interface into the corresponding new files. The top-module is named “main.sv” in this case, so make sure to adjust the settings accordingly.)

e) You will then be able to synthesize the code using the button “Synthesize”. Once the synthesis is complete and succeeds without errors, you can click on “Upload to FPGA” to load your design onto an FPGA.



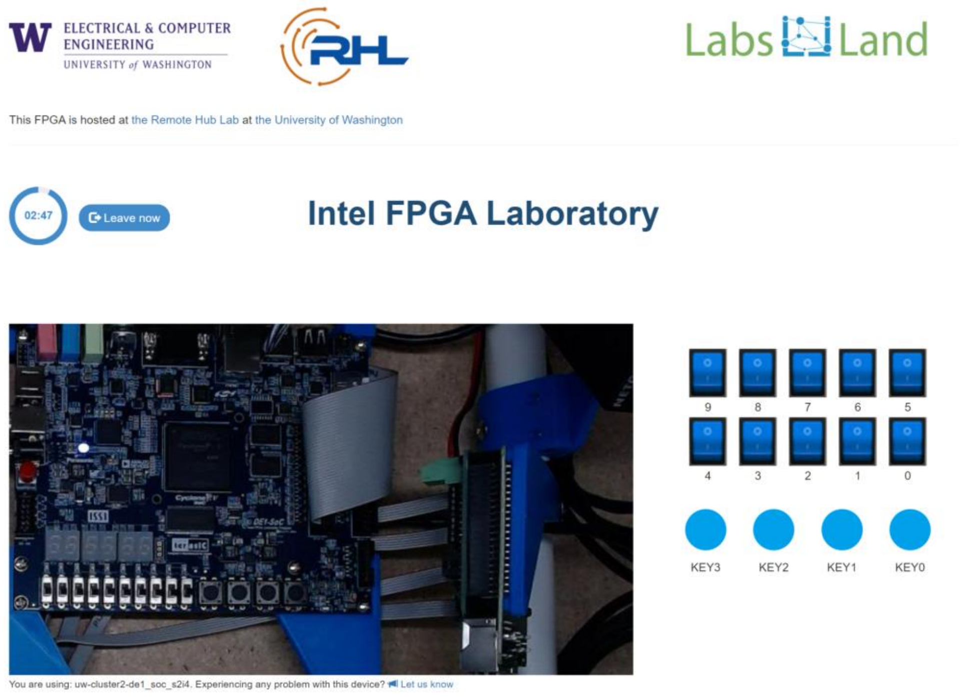
The program was successfully verified and compiled (3:21:46 PM).

```
// Top-level module for the full adder
// Author: Rania Hussein, University of Washington

module main (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    fullAdder FA (.A(SW[2]), .B(SW[1]), .cin(SW[0]), .sum(LEDR[0]), .cout(LEDR[1]));

    assign HEX0 = 7'b1111111;
    assign HEX1 = 7'b1111111;
    assign HEX2 = 7'b1111111;
    assign HEX3 = 7'b1111111;
    assign HEX4 = 7'b1111111;
    assign HEX5 = 7'b1111111;

endmodule

module main_testbench();
```

```
$ quartus_map main --source COMPILATION_DIRECTORY/fulladder.sv --source COMPILATION_DIRECTORY/main.sv --f
Info: ************************************************************
```

Figure 8: Synthesizing on LabsLand

f) After connecting to the remote FPGA, you will see a webpage like Figure 9's.



This FPGA is hosted at the Remote Hub Lab at the University of Washington

Intel FPGA Laboratory

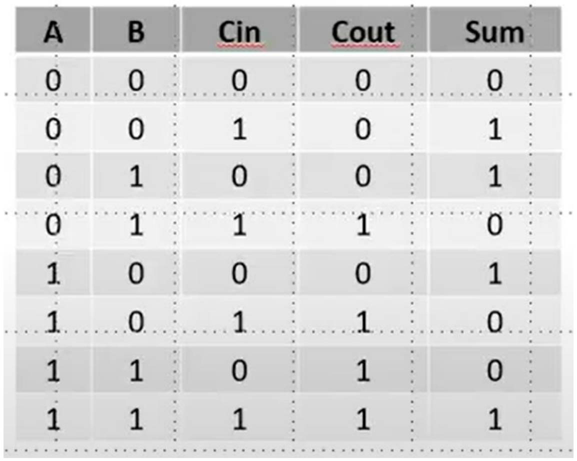
You are using: uw-cluster2-de1_soc_s2i4. Experiencing any problem with this device? Let us know

Figure 9: An FPGA on LabsLand

The right part of the page shows the buttons and keys of the FPGA. You can click on the buttons and keys accordingly as inputs. It is important to note that 'KEYS' need to be held down, as they do not function like switches.

g) Toggle Switch2 (A), Switch1 (B), and Switch0 (Cin) to test different input combinations. Check that your system in LabsLand outputs signals using LEDR1 (Cout) and LEDR0 (Sum) which are consistent with the Truth Table in Table 2.

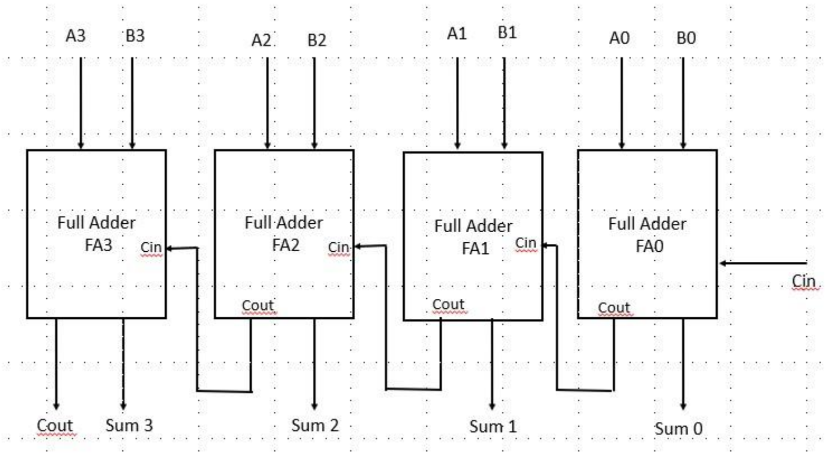
Table 2: Full Adder Truth Table



A	B	Cin	Cout	Sum
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

Task 3: Design a 4-bits adder

In the previous tasks, we created an adder that can add three 1-bit. However, in reality, the numbers consist of multiple bits and therefore we want to extend our design accordingly. Your task is to use the "fullAdder"
SystemVerilog module to extend the design to 4 bits numbers as shown in the following figure



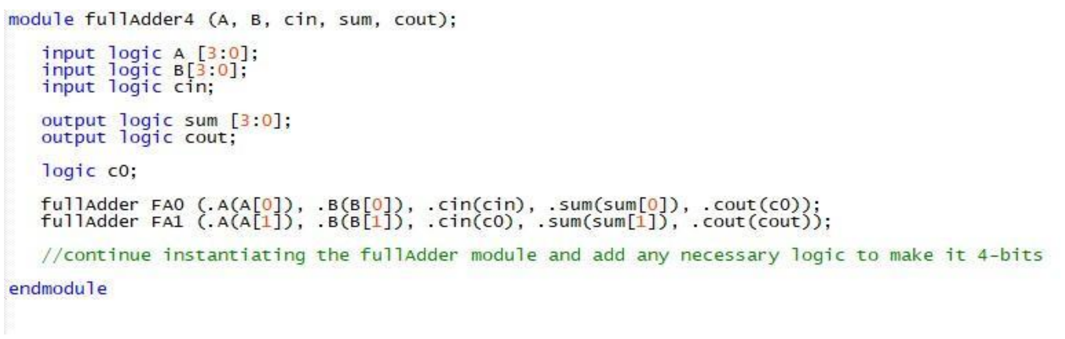
4-bits Adder

To get started, you need to do the following:

1. Instead of creating a project from scratch, make a copy of the lab1 folder that was created in task 2 and call it "lab1a". Launch quartus by double clicking on DE1_Soc.qpf file under the lab1a folder. If you followed all the steps of task 2, you should have two .sv files: DE1_SoC.sv and fullAdder.sv

2. Create a new SystemVerilog file and call it fullAdder4. This module will instantiate the fullAdder module created in task 2. To help you get started, here's a skeleton

of fullAdder4 please modify it accordingly. Make the fullAdder4 module your top level entity and simulate it in ModelSim and save a picture of the simulation for your lab report.

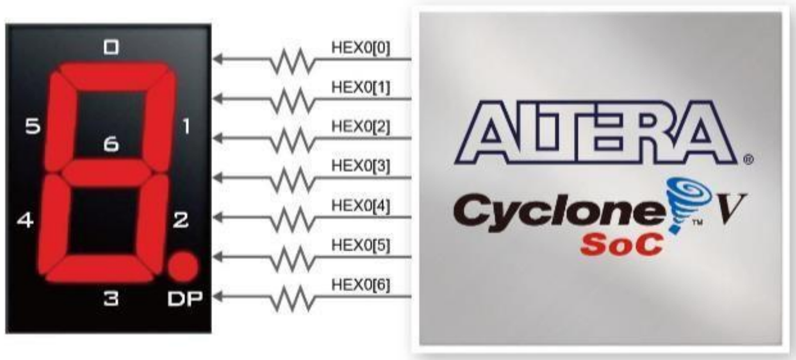


```
module fullAdder4 (A, B, cin, sum, cout);

   input logic A [3:0];
   input logic B[3:0];
   input logic cin;

   output logic sum [3:0];
   output logic cout;

   logic c0;

   fullAdder FA0 (.A(A[0]), .B(B[0]), .cin(cin), .sum(sum[0]), .cout(c0));
   fullAdder FA1 (.A(A[1]), .B(B[1]), .cin(c0), .sum(sum[1]), .cout(cout));

   //continue instantiating the fullAdder module and add any necessary logic to make it 4-bits

endmodule
```

3. Modify the DE1_SoC.sv such that you use 9 switches and 5 LEDs. SW0 is for Cin, SW4-SW1 for the 4-bits A and SW8-SW5 for B. The 5 LEDs should show Sum3-Sum0 and Cout. For better readability, show the output in the same order shown in Figure 1 from left to right (i.e Cout should be the left most LED and Sum0 is the right most LED).

4. Additionally, modify the assign statements for the HEX displays in the DE1_SoC.sv such that it displays the word "Adding". You will need to use 6 HEX displays to display the word "Adding". As an example, to make HEX5 shows the letter 'A', all the segments need to be turned on except segment 3 (HEX5[3]). The following figure is from the DE1_SoC datasheet and shows the 7-segments display connections. The segments are active low which means that a value of 0 is needed to turn on any segment.

HEX0 connections on the DE1_SoC board



Accordingly, to display the letter 'A' on HEX5, the following statement can be used

assign HEX5 = 7'b0001000; //displays 'A' where all segments except 3 are turned on.

5. Make DE1_SoC.sv your top level entity and simulate it in ModelSim and save a picture of the simulation for your lab report.

6. Similar to the steps you followed in task 2 to upload the program to Labsland, load the 4-bits program to the FPGA and test it. For demonstration, test the adder by adding 2 random 4 bits numbers For example you can try adding 1001 with 0111, and 0101 with 1111, or other random numbers of your choice.

Lab Demonstration and Submission Requirements

- Submit a video demonstrating the 4-bits adder. Your video is expected to be around 2 minutes or less. In the video demonstrate the functionality on the board using the switches and LEDs and the 7-segment display.
- Submit a short lab report (about 3 pages and it's ok if you go above 3 pages) that should include the following sections, detailed below.

  Results
  - Include a screenshot of the waveforms you created for task1
  - Include screenshots of ModelSim results for tasks 2 and 3
  - Describe what you tested in the simulation, and what the results in the screenshot show

  Reflection
  - Write down your initial thoughts about the Understanding → SystemVerilog → Synthesis → Simulation → FPGA workflow. What did you observe throughout the process?
  - What did you notice about the SystemVerilog code that you'd like to learn more about?
  - What questions do you have, if any?

- Submit the SystemVerilog files (files with extension .sv) of task 3. Make sure to follow the commenting guide provided. **A significant amount of grade will depend on the commenting style.**
- Submit your report, video and programs to Canvas.