

Lab 1: FPGA Tools & Combinational Logic Design

Digital Design and Computer Architecture: RISC-V Edition (Harris & Harris, Elsevier © 2021)

Objective

The purpose of this lab is to learn to use Field Programmable Gate Array (FPGA) tools to simulate a SystemVerilog description of combinational logic, then synthesize it onto the FPGA and download it onto an FPGA board. The lab tutorial will walk you through a full adder and then you will design an instruction decoder circuit.

1. Tutorial: Altera FPGA Tools

All labs will be using the Intel® Quartus® Prime Lite Edition Design Software Version (Version 21 is the newest version, but you are encouraged to use **Version 20** for a better ModelSim experiment. Link: [Intel® Quartus® Prime Lite Edition Design Software Version 20.1.1 for Windows](#)) and the Altera DE0-CV (You also can use the DE10-Lite.) evaluation board with the Cyclone V 5CEBA4F23C7N¹ chip. You can download and install the software on your own Windows PC to do parts of the labs from home but will need to use a DE0-CV board (you can spend \$100 to get one for yourself).

In this tutorial, you will design a full adder, simulate it in ModelSim, and implement it on the DE0-CV board (optional). You will hook up three switches for input and two LEDs for output and check that the circuit behaves correctly. The instruction steps are as followed:

- Make sure the DE0-CV board is powered on with a wall adapter plugged into the DC 5V jack and the USB Blaster (J13) port is plugged into the computer you are using. Press the red button to turn on power and confirm that the POWER LED (D15) is glowing blue.
- Create a new folder in your temp storage directory for this tutorial, such as lab0_xx, where xx are your initials. Open Quartus Prime. It is found under the **Start** menu under **INTEL FPGA 18... Lite Edition → Quartus**. You will be greeted with a getting started window. Click on the **New Project Wizard**.
 - If the getting started screen is not present, you can reach the same wizard by selecting **File → New Project Wizard**.
 - If the Introduction screen appears, click the **Don't show me this introduction** again box and click on **Next**.

¹ 5C indicates the Cyclone V family of chips. The E indicates Enhanced logic/memory. The B indicates no hardware PCIe or memory controller. The A4 indicates the number of logic elements (49k, a medium-sized chip). F23 indicates that the chip is in a 484-pin ball grid array package. C indicates commercial temperature grade, and 7 is the medium speed grade for this chip. N indicates lead-free packaging (standard these days).

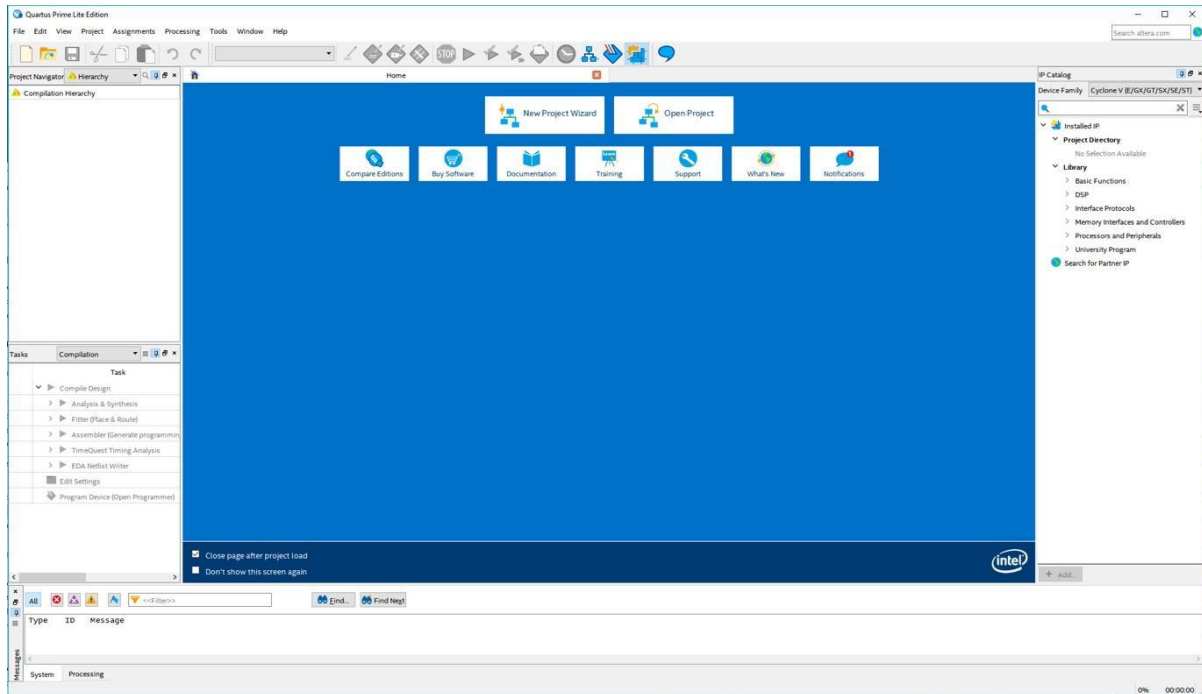


Figure 1. Open Quartus Prime Lite

- On the Directory, Name, Top-Level Entity screen, change the working location of the project to the folder you created, change the name of the project to something suitable such as lab1_XX. Set the top-level design entity to **fulladder**.

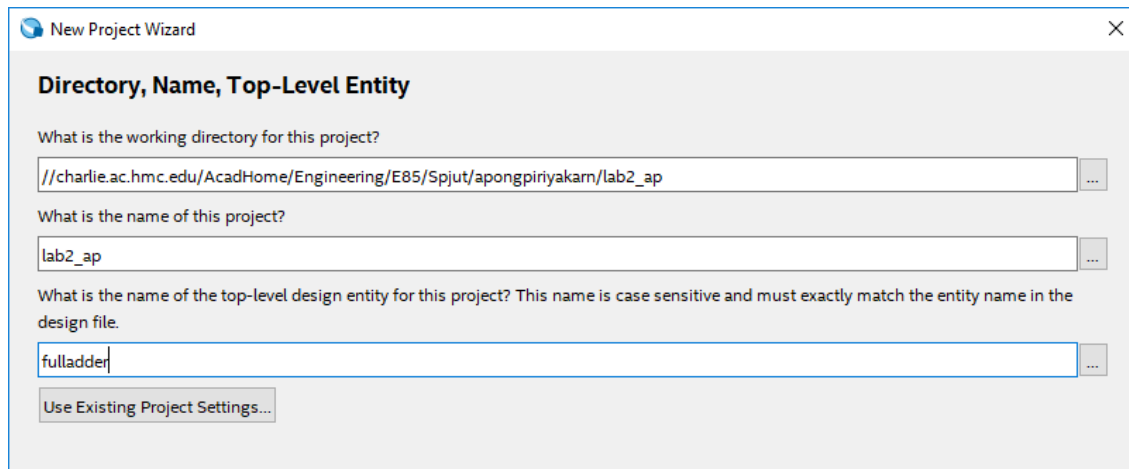
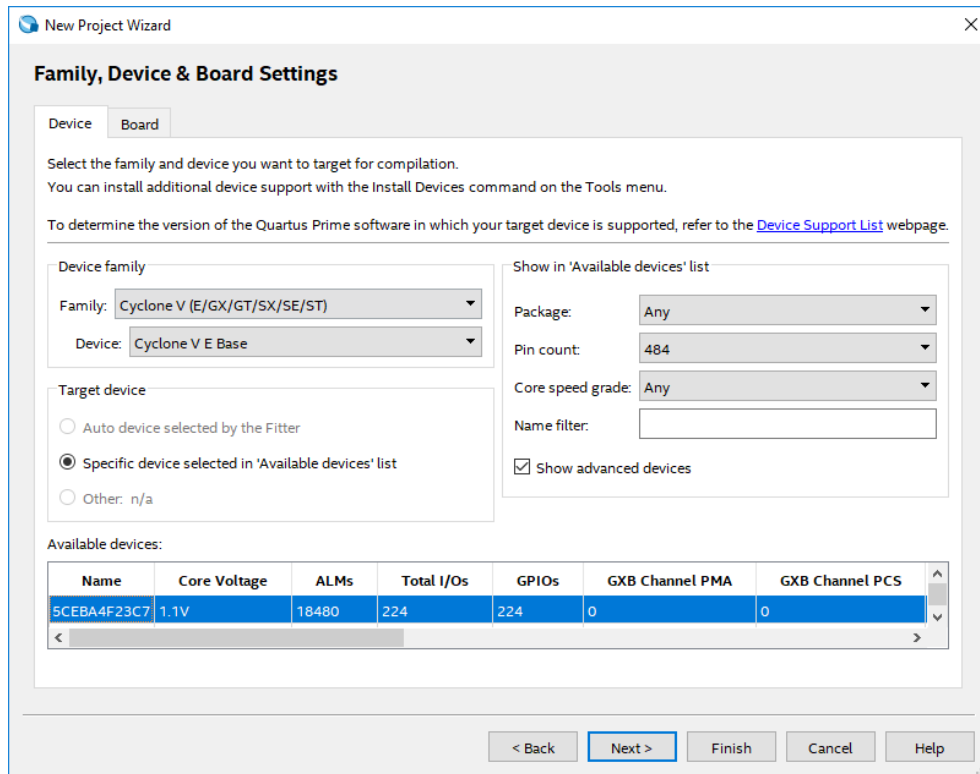


Figure 2. New Project Wizard

- Click **Next**. On the Project Type screen, select **Empty project**, and click **Next**.
- Click **Next** on the Add Files page as we have no files to add. The next page will set the specific FPGA we want the tool to target.
- Select **Pin Count** → **484**, then **Device** → **Cyclone V E Base**; this will greatly reduce the choices. Click **5CEBA4F23C7** on the available devices and click next.



Family, Device & Board Settings

Device | Board

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.

To determine the version of the Quartus Prime software in which your target device is supported, refer to the [Device Support List](#) webpage.

Device family

Family: Cyclone V (E/GX/GT/SX/SE/ST)

Device: Cyclone V E Base

Target device

☐ Auto device selected by the Fitter

☒ Specific device selected in 'Available devices' list

☐ Other: n/a

Show in 'Available devices' list

Package: Any

Pin count: 484

Core speed grade: Any

Name filter:

☒ Show advanced devices

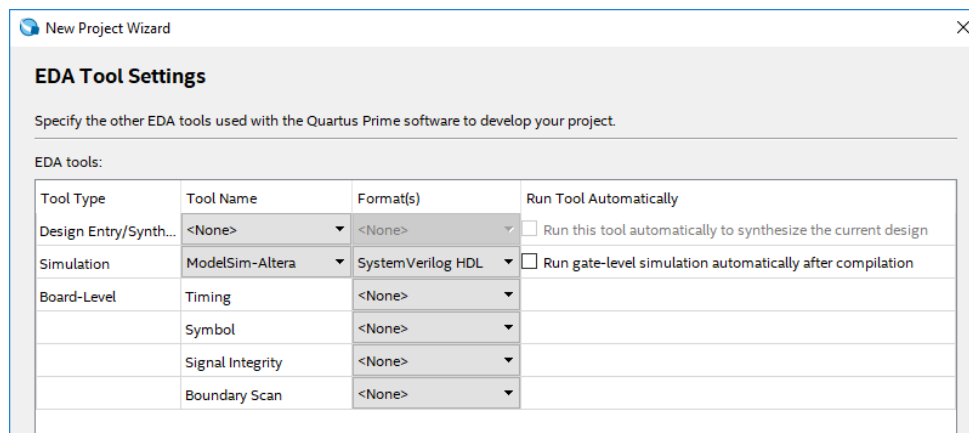
Available devices:

Name	Core Voltage	ALMs	Total I/Os	GPIOs	GXB Channel PMA	GXB Channel PCS
5CEBA4F23C7	1.1V	18480	224	224	0	0

< Back | Next > | Finish | Cancel | Help

Figure 3. Select target FPGA

- On the next page, change **Simulation** to **ModelSim-Altera** and the **Format** to **SystemVerilog HDL**, click next, then Finish.



EDA Tool Settings

Specify the other EDA tools used with the Quartus Prime software to develop your project.

EDA tools:

Tool Type	Tool Name	Format(s)	Run Tool Automatically
Design Entry/Synth...	<None>	<None>	<input type="checkbox"/> Run this tool automatically to synthesize the current design
Simulation	ModelSim-Altera	SystemVerilog HDL	<input type="checkbox"/> Run gate-level simulation automatically after compilation
Board-Level	Timing	<None>	
	Symbol	<None>	
	Signal Integrity	<None>	
	Boundary Scan	<None>	

Figure 4. Setting ModelSim and SystemVerilog HDL Settings

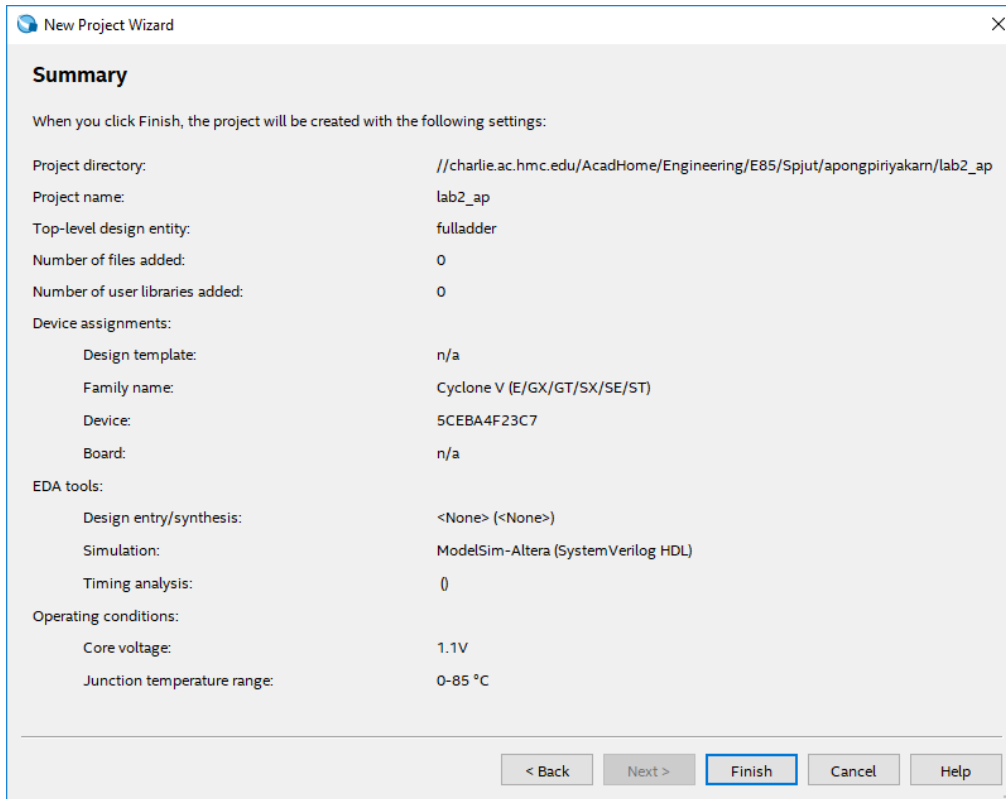


Figure 5. Project Summary

For this tutorial we will create a full adder.

- Choose **File** → **New** → **SystemVerilog HDL File**.

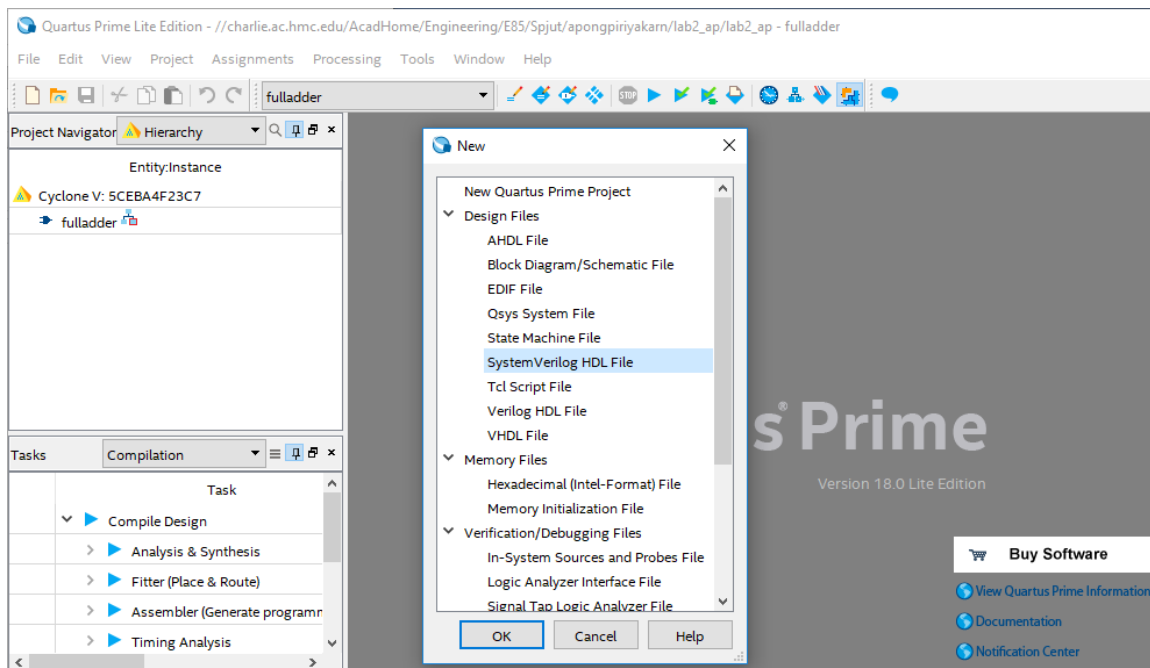


Figure 6. Create new SystemVerilog HDL File

- **Enter the HDL** for the full adder below into the file. Although it is not required, it might be more useful to type up the code, instead of copying and pasting it, to get comfortable with SystemVerilog now. Then, save your file as fulladder.sv in your lab1_xx directory.

```
// Behavioral Verilog explains relationships between inputs and outputs
// For example: assign y = a & b;
// Structural Verilog describes structures formed by simpler components
// For example: and g1(y, a, b);
// Sections 4.1-4.3 in the book (starting at p. 171) describes these
// differences in detail
```

```
// Is this module structural or behavioral?
```

```
module fulladder(input logic a, b, cin,
                 output logic sum, cout);
```

```
    // Declare 5 internal logic signals or local variables
    // which can only be used inside of this module
    logic ns, n1, n2, n3, n4;
```

```
    // The following logic gates are part of SystemVerilog Spec
    // (built-in primitives).
    // The first signal (e.g., ns) is the output. The rest(e.g., a, b) are
    // inputs.
```





```
    // sum logic
    xor x1(ns, a, b);           // ns = a XOR b
    xor x2(sum, ns, cin);       // sum = ns XOR cin
```

```
    // carry logic
    and a1(n1, a, b);           // n1 = a & b
    and a2(n2, a, cin);         // n2 = a & cin
    and a3(n3, b, cin);         // n3 = b & cin
    or o1(n4, n1, n2);          // n4 = n1 | n2
    or o2(cout, n3, n4);        // cout = n3 | n4
```

```
    // This example is Structural Verilog because the module is described
    // structurally using more fundamental building blocks
endmodule
```

1.1 Synthesis

Having completed the code, we can now synthesize it into hardware. Quartus Prime calls this process compilation.

- Choose **Processing** → **Start Compilation** (or click the Start Compilation arrow  next to the STOP icon). Watch for warnings , critical warnings , errors , and other notes in the bottom panel. It is a good habit to learn which warnings are normal and to track down the root cause of abnormal warnings that can signal something awry that would otherwise take you hours to debug.

In addition to other warnings, you should get a critical warning that the pins have not been assigned. Now you will need to assign the proper pins so that the signals in your design connect to the desired switches and LEDs on the board.

- Look at section 3.2, starting on Page 21, of the [DE0-CV User Manual](#) for the FPGA pin numbers for each function including pushbuttons, slide-switches, and LEDs on the board.

Now that synthesis has run, Quartus knows what signals are used by your top-level module, so you can assign them to pins. Let's assign inputs a, b, and c to SW0, SW1, and SW2, respectively. The manual shows that SW0 is PIN_U13 on your FPGA.

- Choose **Assignments** → **Pin Planner** and set **Location** for input a to PIN_U13. Likewise, set b to PIN_V13 and cin to PIN_T13. Hook sum to LEDR0 (PIN_AA2) and look up the pin assignment for LEDR1 for cout. Then close the Pin Planner and synthesize again. You should see two critical warnings that *Synopsys Design Constraints File* not found because you have specified no timing requirements for your circuit, but the critical warnings about pin assignments should go away. The rest of the lab should operate properly even with the set of warnings that are left. The file [QuartusFulladderCompOut.txt](#) on the website contains typical output at this stage. If you have *lots* of spare time, you can see how many of the remaining warnings you can make go away.

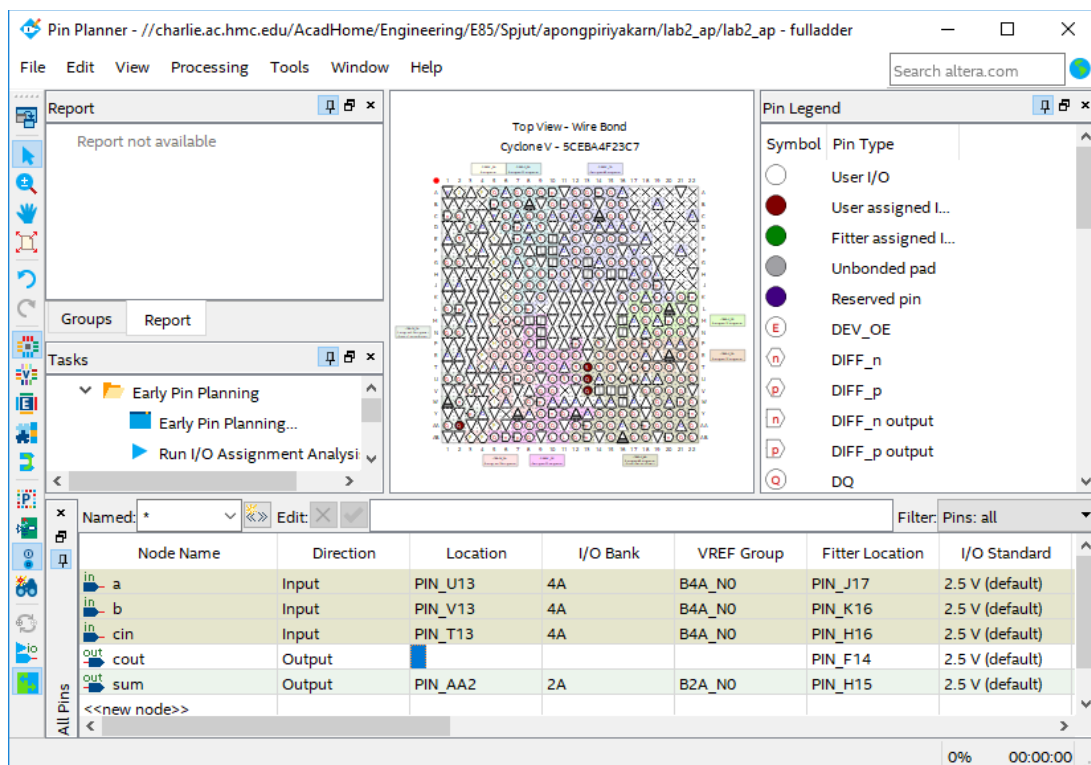


Figure 7. Pin Planner

1.2 RTL Viewer

Now we will look at what the synthesizer created using the register transfer level (RTL) viewer.

- Choose **Tools** → **Netlist viewer** → **RTL Viewer**. You should see the following circuit that matches your code.

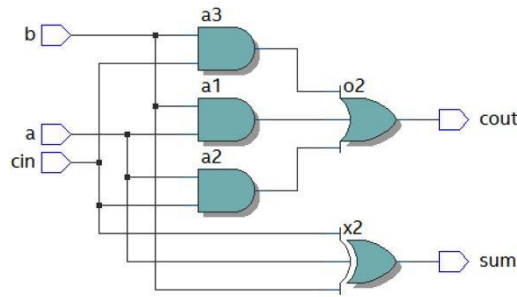


Figure 8. RTL Viewer

1.3 Simulation

Next, we will simulate our circuit to make sure it performs the intended function. The best way to do a simulation is with a self-checking testbench written in SystemVerilog. The testbench applies inputs and checks that the outputs match expectations. If you find a mistake, you can correct the design and rerun the simulation to confirm. This process reduces the tedium and risk of introducing errors when running simulations and checking the results manually.

SystemVerilog is powerful in that it supports both hardware modeling and testbenches, but you will have to be careful not to use the kinds of programming language constructs of a testbench when you intend to imply hardware.

- Create a new SystemVerilog file and enter the following code into it. Green lines are comments, so you do not need to copy them. Again, you have an option of copying/pasting the code or typing it up yourself. You'll have this option for the rest of sample codes provided in class as well. Save the code as testbench.sv in your lab2_xx directory. Observe that this code is a very different style of SystemVerilog than you have previously seen; instead of implying physical hardware, it reads inputs called test vectors from a file, applies them, and checks the result.
 - Note that if you copy and paste the code from this document, you might want to replace any smart quotes (‘ ’, “ ”) with standard quotes (' ') in Quartus

```

//// Testbench module tests another module called the device under test(DUT).
// It applies inputs to DUT and check if outputs are as expected.
// User provides patterns of inputs & desired outputs called testvectors.
module testbench();
    logic                clk, reset;
    // 'clk' & 'reset' are common names for the clock and the reset,
    // but they're not reserved.
    logic                a, b, cin, s, cout, sexpected, coutexpected;
    // These variables or signals represent 3 inputs, 2 outputs, 2 expected
    // outputs, respectively.
    logic [31:0] vectornum, errors;
    // '[31:0]' indicates that the following signals, vectornum and errors // in this case, are 32-bit long (start
    // from bit 0 to bit 31) in little // endian order (the least significant bit at the lowest address or
    // [msb:lsb]).
    // vectornum shows the number of test vectors that has been applied.
    // errors represents the number of errors found.

```

```

// The size of 'int' data type is 4 bytes, thus 32 bits.
logic [4:0] testvectors[10000:0];
// Above is a 5-bit binary array named testvectors with index 0 to 10000
// (testvectors[0],testvectors[1],testvectors[2],...,testvectors[10000]).
// In other words, testvectors contains 10001 elements, each of which is
// a 5-bit binary number. The number of bits represent the sum of the
// number of input and output bits (eg. three 1-bit inputs + two 1-bit
// outputs = one 5-bit testvector).
// In this tutorial, we will only
// use 8 test vectors (found in .tv file below), however it doesn't hurt
// to set up array to support more so we could easily add test vectors
// later.

//// Instantiate device under test (DUT).
// Inputs: a, b, cin. Outputs: s, cout.
fulladder dut(a, b, cin, s, cout);

//// Generate clock.
always
// 'always' statement causes the statements in the block to be
// continuously re-evaluated.
begin
    //// Create clock with period of 10 time units.
    // Set the clk signal HIGH(1) for 5 units, LOW(0) for 5 units
    clk=1; #5;
    clk=0; #5;
end

//// Start of test.
initial
// 'initial' is used only in testbench simulation.
begin
    //// Load vectors stored as 0s and 1s (binary) in .tv file.
    $readmemb("fulladder.tv", testvectors);
    // $readmemb reads binarys, $readmemh reads hexadecimal.

    // Initialize the number of vectors applied & the amount of
    // errors detected.
    vectornum=0;
    errors=0;
    // Both signals hold 0 at the beginning of the test.

    //// Pulse reset for 22 time units(2.2 cycles) so the reset
    // signal falls after a clk edge.
    reset=1; #22;
    reset=0;
    // The signal starts HIGH(1) for 22 time units then remains LOW(0)
    // for the rest of the test.
end

//// Apply test vectors on rising edge of clk.
always @(posedge clk)
// Notice that this 'always' has the sensitivity list that controls when all
// statements in the block will start to be evaluated. '@(posedge clk)' means
// at positive or rising edge of clock.
begin

```



```

        /// Apply testvectors 1 time unit after rising edge of clock to
        /// avoid data changes concurrently with the clock.
        #1;
        /// Break the current 5-bit test vector into 3 inputs and 2
        /// expected outputs.
        {a,b,cin, coutexpected,sexpected} = testvectors[vectornum];

    end

    /// Check results on falling edge of clk.
    always @(negedge clk)
    // This line of code lets the program execute the following indented
    // statements in the block at the negative edge of clock.
        /// Don't do anything during reset. Otherwise, check result.
        if (~reset) begin
            /// Detect error by checking if outputs from DUT match
            /// expectation.
            if (s !== sexpected || cout !== coutexpected) begin
                /// If error is detected, print all 3 inputs, 2 outputs,
                /// 2 expected outputs.
                $display("Error: inputs = %b", {a, b, cin});
                /// '$display' prints any statement inside the quotation to
                /// the simulator window.
                /// %b, %d, and %h indicate values in binary, decimal, and
                /// hexadecimal, respectively.
                /// {a, b, cin} create a vector containing three signals.
                $display(" outputs = %b %b (%b %b expected)", s, cout, sexpected, coutexpected);
                /// Increment the count of errors.
                errors = errors + 1;
            end
            /// In any event, increment the count of vectors.
            vectornum = vectornum + 1;
            /// When the test vector becomes all 'x', that means all the
            /// vectors that were initially loaded have been processed, thus
            /// the test is complete.
            if (testvectors[vectornum] === 5'bx) begin
                /// '===&'!==' can compare unknown & floating values (X&Z),unlike
                /// '==&'!==' which can only compare 0s and 1s.
                /// 5'bx is 5-bit binary of x's or xxxxx.
                /// If the current testvector is xxxxx, report the number of
                /// vectors applied & errors detected.
                $display("%d tests completed with %d errors", vectornum, errors);
                /// Then stop the simulation.
                $stop;
            end
        end

    end

    // In summary, new inputs are applied on the positive clock edge and the
    // outputs are checked against the expected outputs on the negative clock
    // edge. Errors are reported simultaneously. The process repeats until there
    // are no more valid test vectors in the testvectors arrays. At the end of the
    // simulation, the module prints the total number of test vectors applied and
    // the total number of errors detected.
endmodule

```

- Create another file called fulladder.tv and add the following lines (easiest in a different text editor, such as Notepad. Save with the “All Files” format). Each line has 5 bits corresponding to the three inputs and two expected outputs (essentially the truth table). Underscores in the test vector file are ignored, so the underscores are placed between the inputs and expected outputs to make them easier to read. The // line is a comment and is also ignored. For example, this test vector file indicates that a, b, and cin will all be read in as 0 and used in the simulation for the first test, and that s (sum) and cout are expected to both be 0 on this test. On the second test, cin becomes 1 and the expected s becomes 1 as well, but the other read-in and expected values are still 0. Since the logic is all combinational (no flip-flops or memory) and there are three 1-bit inputs, then there are $2^3 = 8$ possible inputs with their corresponding outputs.

```
// a b cin _ cout s
000_00
001_01
010_01
011_10
100_01
101_10
110_10
111_11
```

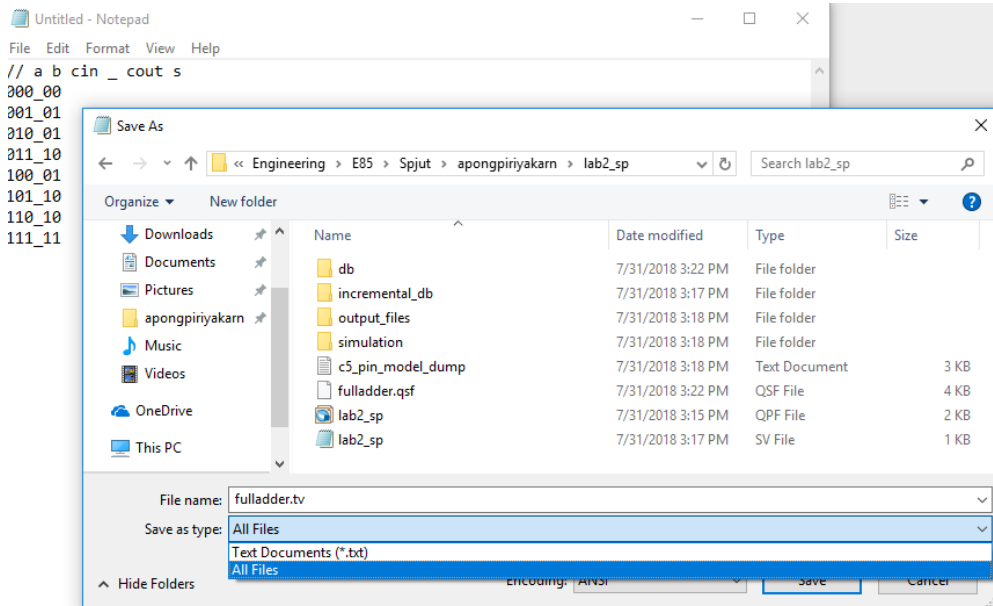


Figure 9. Saving fulladder.tv

We will use ModelSim, a commercial hardware description language (HDL) simulator made by Mentor Graphics. You can download and install ModelSim either as part of the Quartus Prime installation or directly from Mentor Graphics on your computer if you wish. On the lab computers it is found under the **Start** menu under **INTEL FPGA 18... Lite Edition** → **ModelSim**.

- Choose **File** → **New** → **Project...** and create a project named lab0_ms_xx in your Charlie directory. Click **OK**.

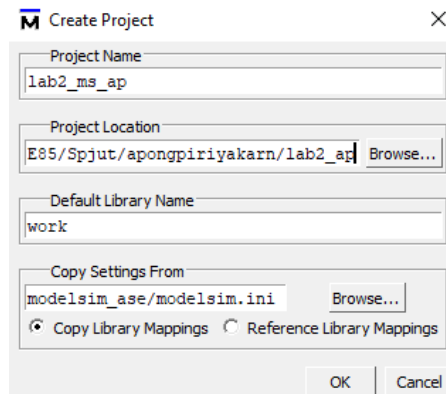


Figure 10. Create Project in ModelSim

- Click **Add Existing File** and browse to add your fulladder.sv and testbench.sv files. This step might take a few seconds. Choose **Compile** → **Compile All**. You should see a message “2 compiles, 0 failed with no errors.” If you do get errors, click on the red errors message to bring up the errors, and correct the bad file, then compile again.

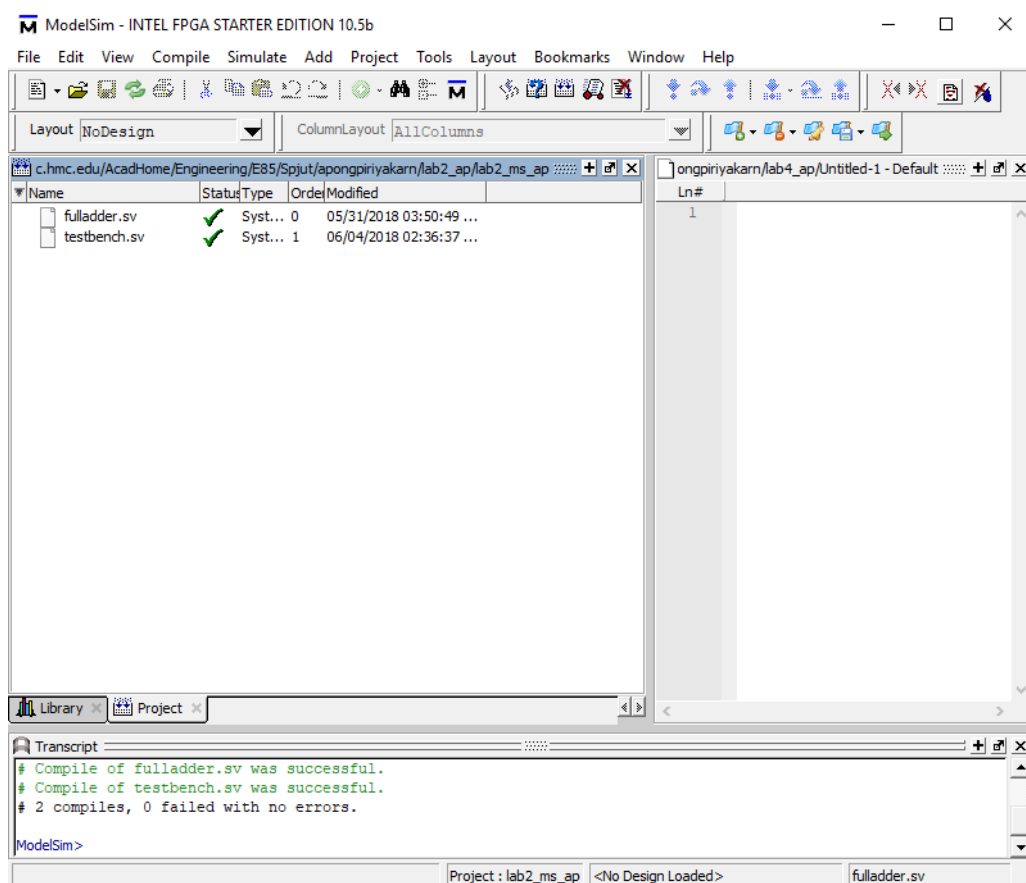


Figure 11. Compile files

- Choose **Simulate** → **Start Simulation...** Expand the + symbol next to the work library, then click on your testbench module. Choose **OK** to simulate it.

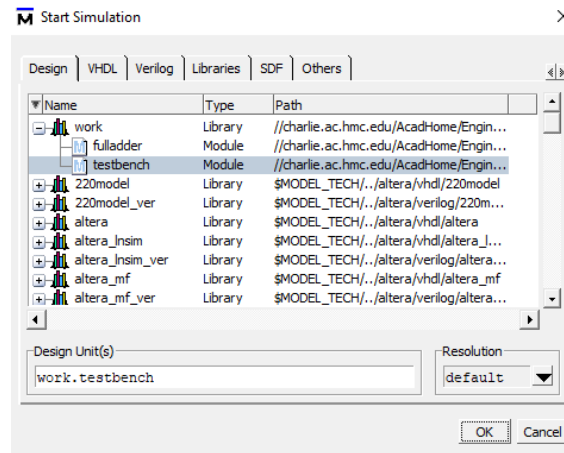


Figure 12. Simulate testbench

- In the Objects pane, select all the signals, then choose **Add** → **To Wave** → **Selected Signals** so that all your inputs and outputs show up in a waveform viewer.

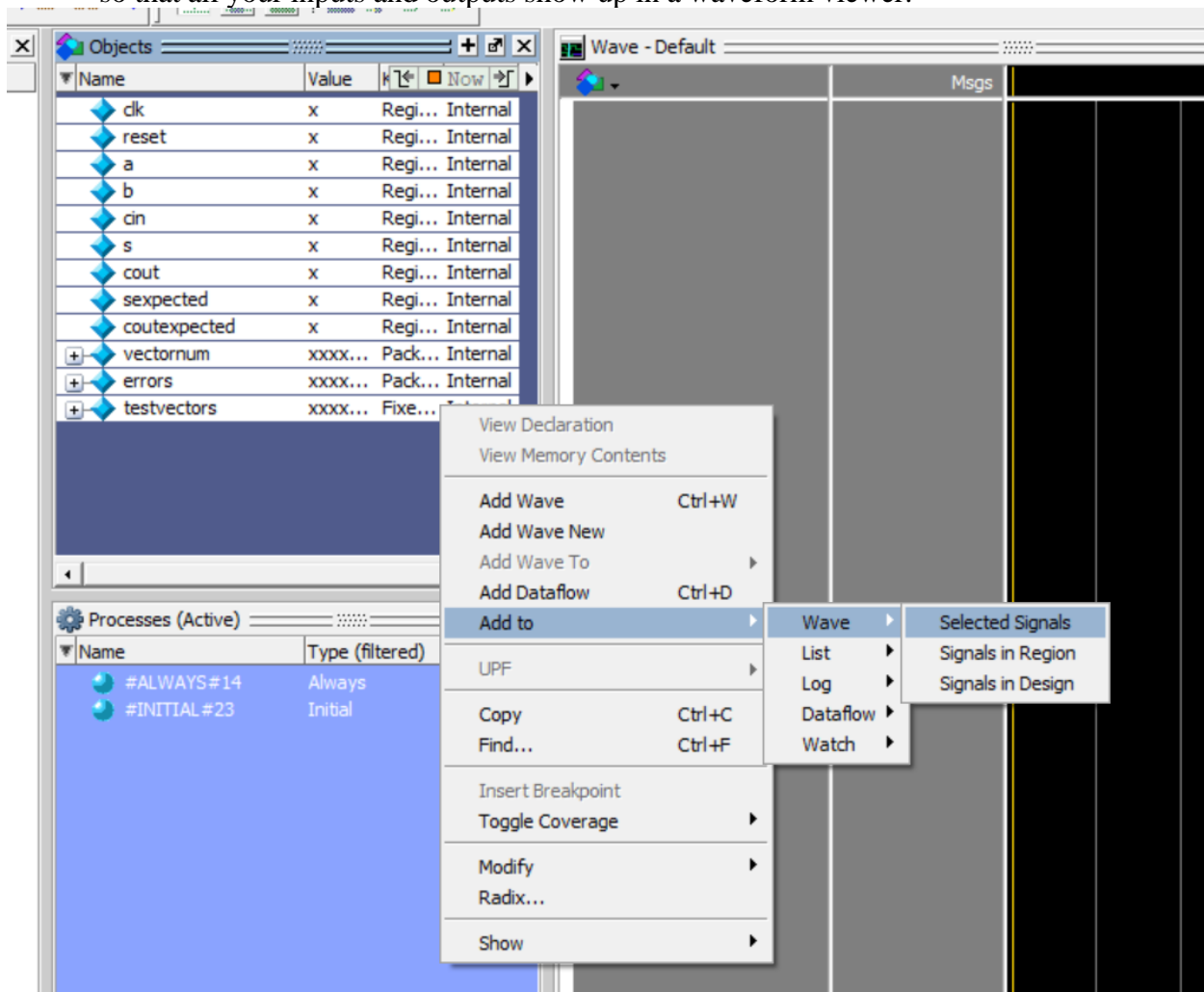


Figure 13. Add signals to wave

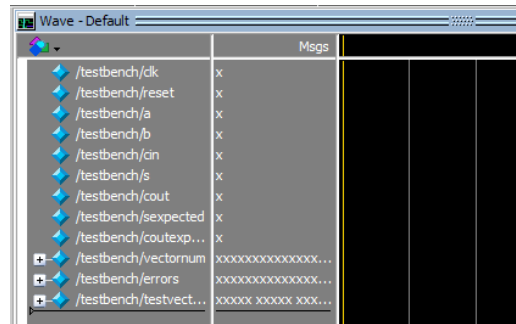


Figure 14. View Wave window

- Type **run 200** in the Transcript pane to run the simulation for 200 time-units. You should see a message “8 tests completed with 0 errors.” You can make the waveforms reappear by using the **Wave** tab at the bottom of the side window. Click on the **Zoom Full (F)** icon to see the full sweep.

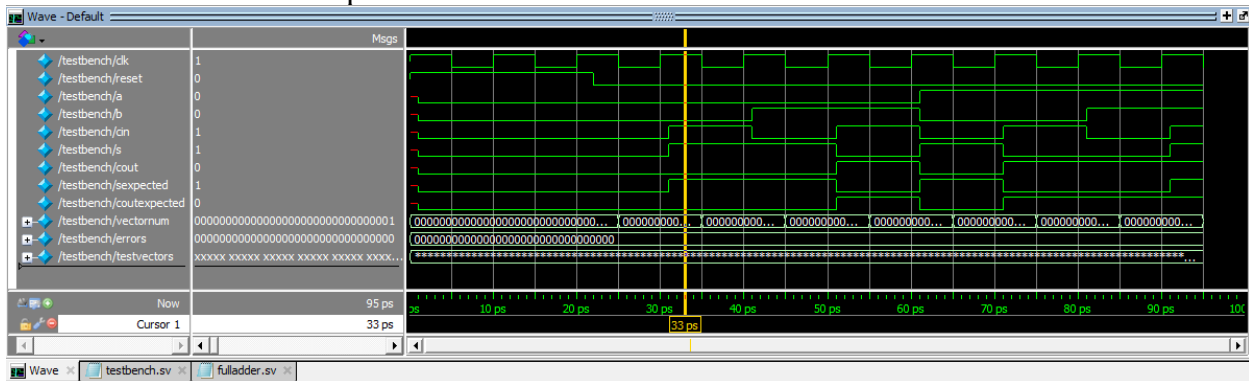


Figure 15. Simulation waveform

- If you see a warning that ModelSim can't find your fulladder.tv file, move it to the same directory that you chose for your ModelSim project (lab0_ms_xx). Then type **restart -f** in the Transcript pane to restart your simulation and **run 200** to rerun. **restart -f** forces a restart of the simulation without recompiling. It's useful if you want to change your test vectors without changing your simulation or testbench code. You DO lose your waveforms. Everything starts over from 0. It lets you save your LIST of displayed signals in the waveform window. If you change anything but the test vectors, you need to recompile and go through the other steps as well.
- If you ever need to stop a runaway simulation, you can use the **Simulate → Break** menu.
- If you make any changes to your code, be sure to choose Compile All again before rerunning, or you'll resimulate the old code.

- Choose **Tools** → **Programmer**.
 - If the programmer window does not say **USB-Blaster** next to **Hardware Setup**, then use the **Hardware Setup** button to set it to **USB-Blaster**.
 - If the **5CEBA4F23** icon in the above image isn't visible, click **Add File...** and browse to the "output_files" folder of your project. Select the .SOF file. It may be pre-selected.
- Click the **Start** button. It should program the FPGA and run to 100% successful.

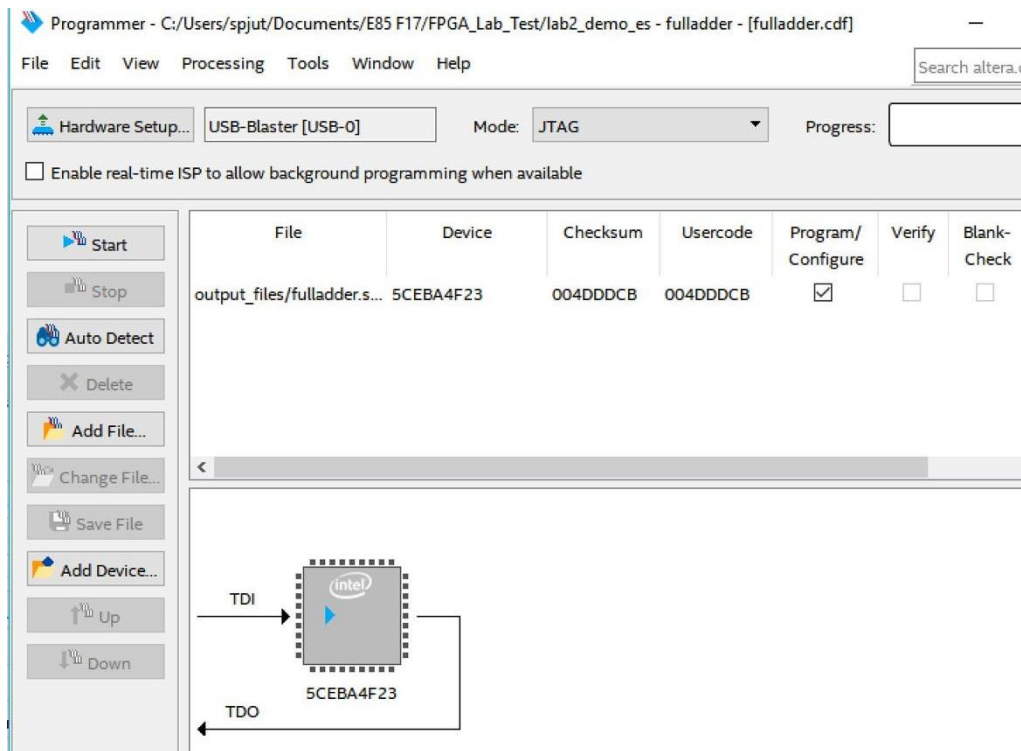


Figure 18. Program FPGA on DE0-CV board

Now you can move the toggle switches **SW[2:0]** on the DE0-CV board and look at the red LEDs just above the switches. Check that your adder adds properly.

2. DE0-CV Board

The Altera / Intel DE0-CV board contains an Altera Cyclone V 5CEBA4F23 FPGA, a power supply, a USB interface to download configuration from the host computer, and some LEDs, switches, and expansion pins.



Figure 19. DE0-CV Board

If you would like to know what is happening under the hood, skim through the [DE0-CV User Manual](#) on the class website. The **DE0CV_ControlPanel** on the lab computers will let you play with many of the features.

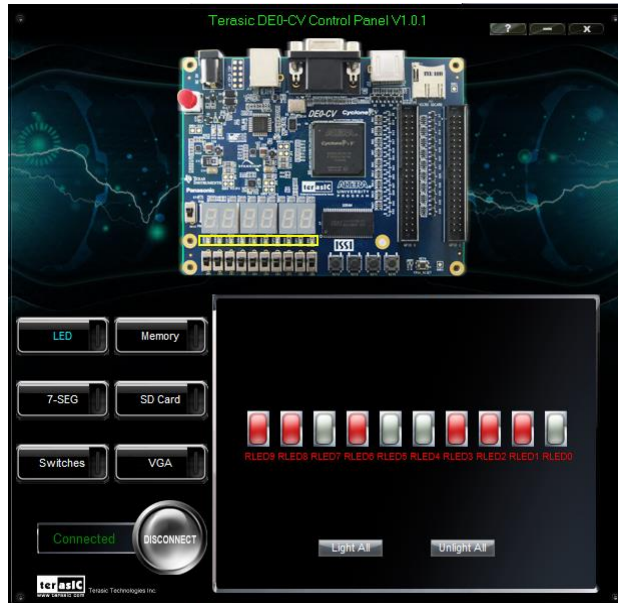


Figure 20.DE0-CV_ControlPanel

3. ALU Decoder

Now it is your turn to design a combinational logic circuit and build it on your FPGA board.

Table 1 describes an ALU Decoder, a circuit with seven input bits ($ALUOp_{1:0}$, $funct3_{2:0}$, op_5 , and $funct7_5$) and a 3-bit output ($ALUControl_{2:0}$). We will use this circuit in a later lab to control an arithmetic/logic unit (ALU) in a microprocessor.

Table 1. ALU Decoder Truth Table

$ALUOp_{1:0}$	$funct_{3:0}$	$\{op_5, funct_7\}$	$ALUControl_{2:0}$	Operation
00	x	x	000	Add
01	x	x	001	Subtract
10	000	00, 01, 10	000	Add
	000	11	001	Subtract
	010	x	101	SLT
	110	x	011	OR
	111	x	010	AND

For the purposes of this lab, you can assume that your circuit only must correctly handle the inputs in the table, and that the output for all other cases are don't cares. The following are the steps for building the ALU Decoder:

- Write Boolean equations for each of the three output bits ($ALUControl_{2:0}$) and sketch a schematic of a circuit that implements your equations.
- Write structural SystemVerilog code to implement your schematic. Name the module **aludec**. You may use the built-in structural gates, such as `and a1(y, a, b, c)` – which is a 3-input AND gate. For the built-in gates, the output is first (in this case, y) followed by the inputs (in this case, a, b, and c). The module interface should look like this:

```
module aludec(input logic   opb5,
              input logic [2:0] funct3,
              input logic   funct7b5,
              input logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

    // structural SystemVerilog body goes here

endmodule
```

- Build a self-checking testbench and testvector file that applies all the interesting inputs and checks the output. Name the testbench **testbench_aludec** and name the testvector file **aludec.tv**.
- Simulate your testbench and check that your aludec module performs the function you intended; debug any discrepancies. Display the signals in binary in this order (top to bottom): ALUOp, funct3, opb5, funct7b5, ALUControl, ALUControlExpected.
- Assign pins for your FPGA, using SW6 through SW0 to provide inputs and LEDR2 through LEDR0 to display the outputs.
- Synthesize your SystemVerilog code. Then examine it in the RTL Viewer and check that it matches your expectations.
- Download it onto the DE0-CV board and apply the inputs with the switches and check that the outputs match expectations.

Hints:

- Make sure there is a new line after the last vector. ModelSim couldn't read one person's vectors when they didn't have the new blank line.
- Name your testvector file: lab0.txt.
- If ModelSim can't find the path, you can try giving the full path, such as:
`$readmemb("C:/Users/harris/Documents/lab0/lab0_dh.tv");`

What to Turn In

1. Please indicate how many hours you spent in this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Boolean equations for your ALU Decoder
3. Gate-level schematic of your ALU Decoder
4. Structural SystemVerilog code for your ALU Decoder
5. RTL Viewer schematics of your synthesized ALU Decoder
6. Self-checking test bench for your ALU Decoder with a test vector file
7. Simulation waveforms showing the ALU Decoder simulation. Display the signals in binary in this order (top to bottom): ALUOp, funct3, opb5, funct7b5, ALUControl, ALUControlExpected. Did it work correctly?
8. Did the ALU Decoder function correctly on the DE0-CV Board? **(Optional)**

Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.