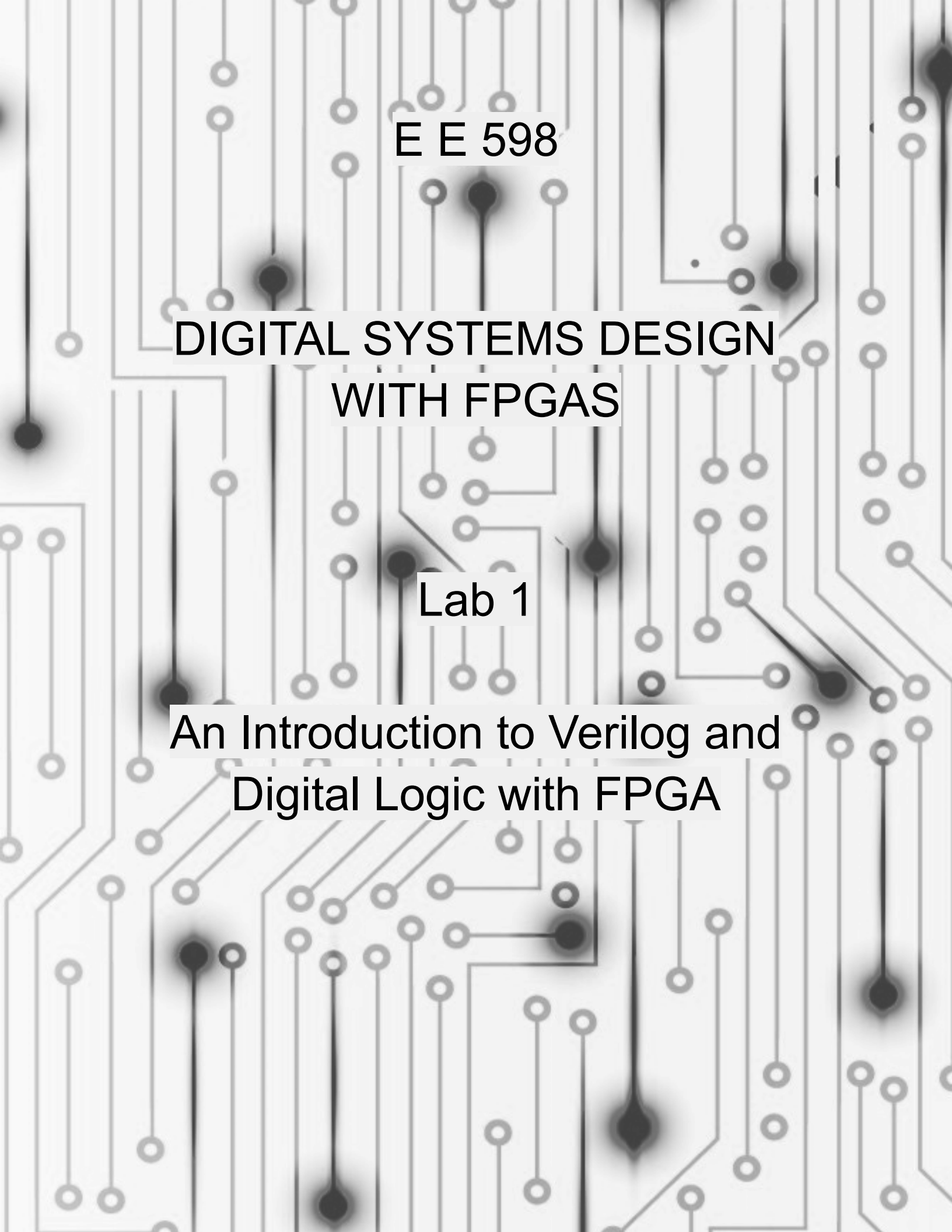


E E 598

DIGITAL SYSTEMS DESIGN WITH FPGAS

Lab 1

An Introduction to Verilog and Digital Logic with FPGA

General Lab Information

Overall Plans for the Labs

The main goal of the lab components for this class is to help you apply the Digital Systems theories and the FPGA concepts you learn in lectures to a physical device - DE1-SoC. You will be working on 4 lab assignments, the last of which will be your final project for this course.

Lab 1 is an introduction to the elements of software and hardware. Then you will practice Combinational Logic and High-Level (RTL). Lab 2 will focus on applying Sequential Logic in Finite State Machine (FSM) and implementing Linear Feedback Shift Register (LFSR). Lab 3 is targeted at memory access and Algorithmic State Machine with Datapath (ASMD). Lab 4 is a self-guided final project to show your comprehension of the material covered in this course, and your ability to apply this material.

Lab Material

Hardware:
The physical board that we will use is the Altera's Terasic **DE1-SoC** Development board. DE1-SoC is a System-on-Chip (SoC) FPGA board. It includes a Cyclone V version of the FPGA unit that gives enough computing power to design and practice the digital logic. It has a variety of GPIO ports, switches, LEDs, and buttons on the board. It also includes hardware such as an analog-to-digital converter (ADC), G-sensor (accelerometer), video and audio capabilities, and an Ethernet connection, which will offer different possibilities for labs and the final project.

Check and make sure you have all the listed items in your kits:

- White Box X 1
- DE1-SoC Development board X 1
- Power adapter X 1
- USB-A data cable X 1
- Custom Breadboard PCB X 1 (We will not use)
- Jumper Cables X 1 pack (We will not use)
- Digital Components X 1 set (We will not use)
- LED matrix board X 1 (We will not use)

You are responsible for your lab kit and we expect that you will return the kit in good working order with all pieces intact.

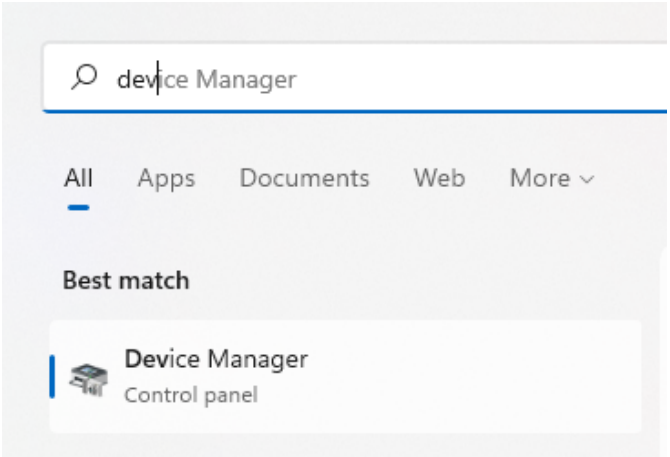
Software:
Download the files into a folder: Link

- The software that we will use to program code, map pins, and upload code is **Quartus Prime**.
- The software that we will use to simulate the functionality of Verilog design is **ModelSim**. It is a powerful tool to verify your work before uploading codes to the physical board.

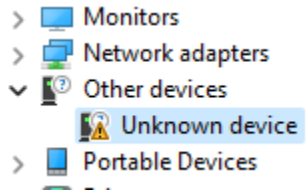
Click **QuartusLiteSetup-20.1.1.720-windows** to install into your PC (you should use windows machine, we do not support mac systems at this point)

NOTE:
If your PC can't recognize DE1, following the steps below:

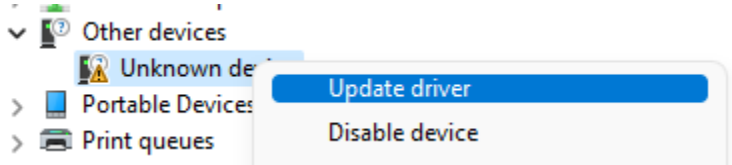
- Run **Device Manager**



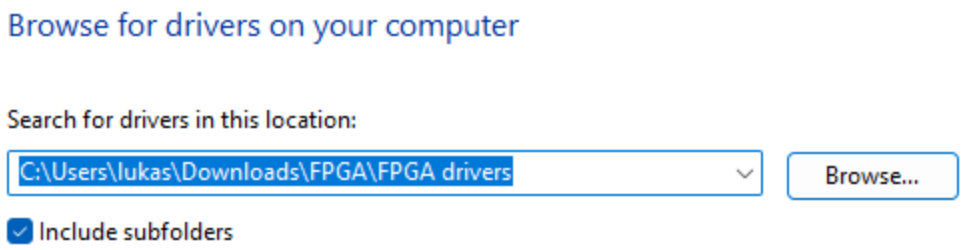
- Located **Unknown device**



- Right click **Update Drivers**



- Choose **Browse my computer for drivers** and set direction to the unzip location of the **FPGA drivers.**



- Then follow the instructions to finish

Demo and Submission

Submission requirements are listed in each lab. Also, as a general rule, you should know that you will need to demo your work with the video (insert video link to the report), you will also need to answer in the report. You will need to fully understand how the lab was completed. Submission should include your full report and your programs, (Due dates on Canvas.)
A report template has been provided and can be found HERE as well.

Before going through the lab, you should review the Verilog tutorial

Lab 1 - Introduction to Digital Logic

Lab 1 - Learning Objectives

This lab will be separated into three parts.

- Part 1:
  - Familiarity with the DE1-SoC board
  - Familiarity with the Quartus and ModelSim.

- Part 2:
  - Familiarity with Combinational Logic Design

- Part 3:
  - Understanding an alternative way to optimize using high-level (RTL) Verilog, where you tell the CAD tools the output to look like, and it automatically does the Boolean Algebra for you (sometimes it can be a real pain to optimize all the way down at gate level.)

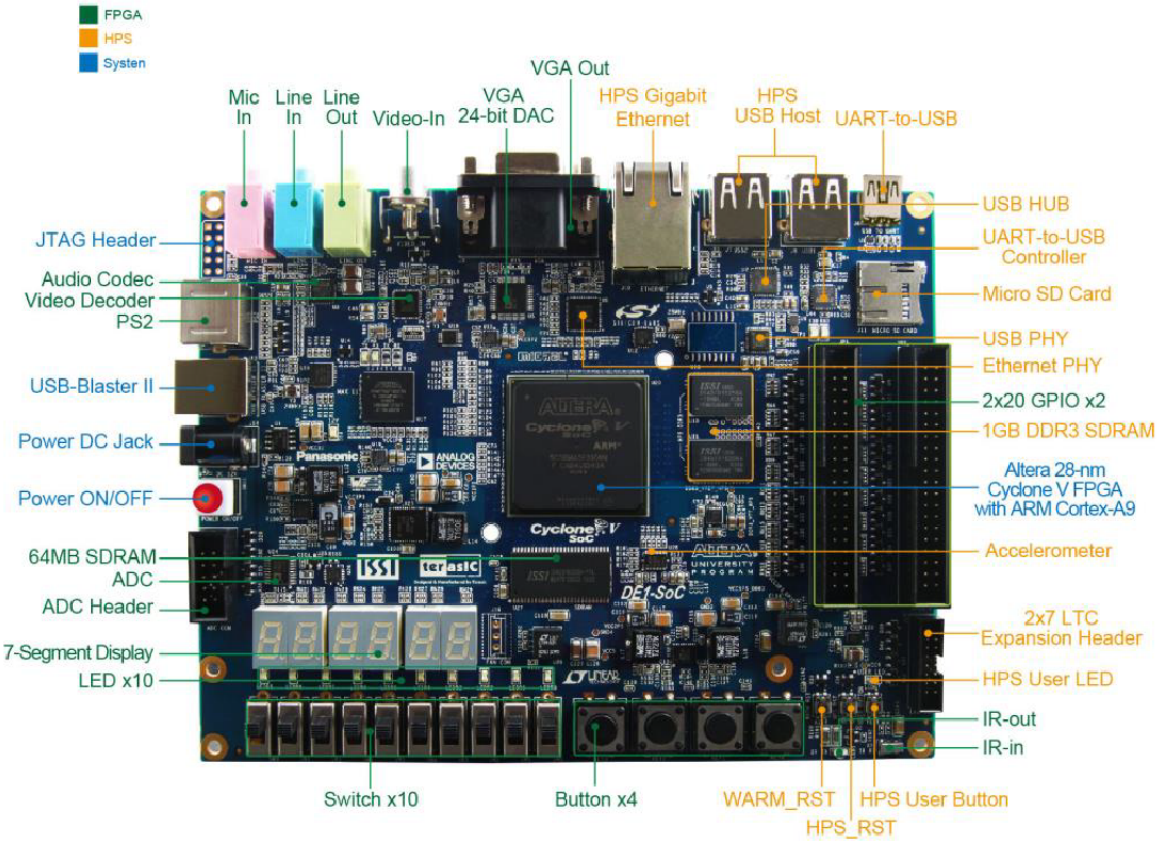
If you have any questions at any point in the lab, make sure you have read the entire lab thoroughly. If you still cannot find the answer you should ask your TA. And as always, there are no dumb questions, please ask them; you don't want to damage any of the expensive hardware that is provided for you.

Note: With all the labs, keep your files when you are done with the lab – they will often get reused in subsequent labs!

Lab 1 Instructions

1. An Introduction to Verilog

1.1 Altera/Terasic DE1-SoC Development Board



The picture above is a diagram of the DE1 with most of the major components highlighted. For the first few labs you will be using the various input/output devices located directly on the board such as the switches and LEDs. Further details will be provided in each lab. Take note of the large FPGA (Field Programmable Gate Array) that is in the middle of the board. Later on in the quarter, you will be programming this and directly interfacing with devices on the board. Think of it like a universal logic unit that all the devices can talk to. For now there is a program loaded into the FPGA to allow you to use the input/output connectors on the board to make the earlier labs easier.

- **The FPGA**
  "FPGA" is an acronym for Field Programmable Gate Array. Essentially it is a large array of logical elements that have been connected together. However, in an FPGA the connections between these logical elements can be programmed and reprogrammed,

which means the FPGA can be used to build many different kinds of hardware all on the same chip. If you look at the lights when you turn on the board, you will see that it has a premade display running. This is a program that we have written and programmed into your DE1's non-volatile memory. By programming into the non-volatile memory, each time the board is turned on this startup program will be loaded, regardless of what you did previously.

- More information for the DE1-SoC in Appendix A

1.2 Quartus and ModelSim Tutorial

- Download the project folder HERE.
- Practice with sections 1-6 of the tutorials to be familiar with software and complete the following assignment.
  (Both of these components are very important for all future labs so please pay attention and do not rush through them.)

1.3 Assigned Task – Verilog Design and Simulation

- Copy the previous project folder and rename it lab1-1.3.
- You will perform the steps in the previous tutorial all the way through section 7. This will have you simulate two designs, mux2_1, and mux4_1. Be sure to do all the steps, and ask the TAs in the lab for help if you get stuck.

- Write a simple explanation of what the mux4_1 actually does (Note: we do NOT want a written description of the circuit gates, we want a description of what it actually does – if you're not sure, see how we described how the mux2_1 works).
- Save a screenshot of the mux4_1 simulation.

2. Combinational Logic Design

2.1 Assigned Task – Multi-Digit Recognizer on the DE1 FPGA

- Copy the previous project folder and rename it lab1-2.1.
- Develop a Verilog program that will check the bottom two digits of your friend's student ID number with yours. If both digits are the same it will light up a LED. SW[7:0] will be the input method and LEDR[0] will be the indicator LED.

SW3	SW2	SW1	SW0	Meaning
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

- First, complete the Quartus tutorial you started in part 1. This will involve loading the mux2_1 design into the DE1 FPGA and testing it with the switches on the board. This will show you the complete flow for creating FPGA-based designs.

- Next, if you followed the tutorial, you should have a SystemVerilog file called DE1_SoC.sv with a module named DE1_SoC within it. Edit the DE1_SoC module to implement the logic for a circuit that will recognize the bottom two digits of your student ID number. SW3-SW0 will be the bottom digit. SW7-SW4 will be the next digit up, using the similar code (i.e. when SW7==1 and SW6, SW5, and SW4 are each 0, the digit encoded is 8). To make things easier I have provided a structure for the file on the next page that will help you get started, and hook things up to the proper inputs and outputs. Note that we do NOT care about the number of gates in the SystemVerilog version – we'll start worrying about efficiency in later labs.

```
// Top-level module that defines the I/Os for the DE1 SoC 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;
   input  logic  [3:0]    KEY;
   input  logic  [9:0]      SW;

   // Default values, turns off the HEX displays
   assign HEX0 = 7'b1111111;
   assign HEX1 = 7'b1111111;
   assign HEX2 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   assign HEX4 = 7'b1111111;
   assign HEX5 = 7'b1111111;

   // Logic to check if SW[3]..SW[0] match your bottom digit,
   // and SW[7]..SW[4] match the next.
   // Result should drive LEDR[0].

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);

   // Try all combinations of inputs.
   integer i;
   initial begin
      SW[9] = 1'b0;
      SW[8] = 1'b0;
      for(i = 0; i <256; i++) begin
         SW[7:0] = i; #10;
      end
   end
endmodule
```

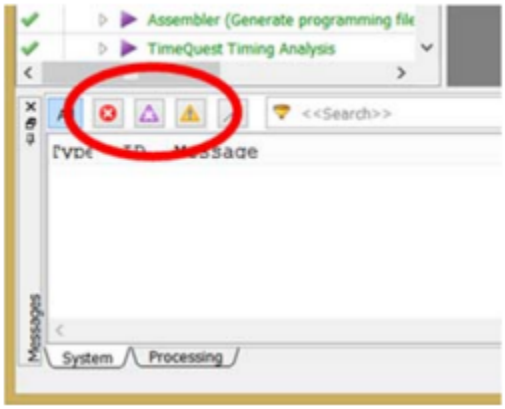
Note: That the testbench uses an “initial” statement to generate the input patterns, with #10 delays, for loops, etc. These are great for quickly writing testbenches, but should not appear in the code that actually gets put into the FPGA. **When you program the FPGA board, the programmer looks for the ‘DE1_SoC’ module and that is the only module that gets loaded onto the FPGA.** The testbench is essentially invisible to the board - although it lives in the same .sv file as the DE1_SoC module, it is not instantiated (referenced) by the DE1_SoC module - thus, none of its code gets loaded to the board. Also note that this means you also do not need to comment out your testbench before loading to the board.

Make sure to test your design in simulation BEFORE mapping it to the FPGA. This will speed up your development time for this lab, and will be critical as your designs get larger.

- **Quartus Tip**: Errors will block compilation of your design but warnings will not. However, some warnings will cause bugs in your program - many of the warning messages that Quartus prints are actually early warnings of errors in your design. Generally, you should try to fix your code to remove all warnings before simulating your design. However, do be aware that Quartus will give warnings we don't actually care about. This is on a case-by-case basis, but here are some general guidelines:

    - You should try to fix: warnings that mention a specific line number or an actual signal you are trying to use (i.e. SW[0], LEDR[5])
    - You probably don't need to worry about: warnings that mention timing constraints, settings files, or anything not referencing your written code
    - Feel free to ask if you are unsure!

  To quickly find errors or warnings, the upper-right corner of the message window has filtering icons that will just show errors, critical warnings, or warnings (left to right) in the message window:



  Note: However the Quartus II does have some useless warning messages. We have set up the tool to filter out many of them, but if you find a warning message that you don't think anyone should see, please show your TA. We will augment the system to ignore these, or explain why that actually is a real problem with your code.

- During simulation you'll likely notice two new kinds of waveforms – red and blue. Red (or the value X) represents an unknown value – since our testbench doesn't specify the value of the KEY inputs, their value is unknown. This will commonly occur if, for example, you declare SW[9:0] in your testbench but only end up actually using, say, SW[0]. SW[0] will have a green waveform but SW[9] - SW[1] will all be red since they are declared but not used. It's recommended to set inputs you are not using to 0 just so that all declared signals are defined. If you see Blue lines (or the value Z) representing a

signal that is disconnected – since our circuit doesn't make any connection to the LEDRs other than LEDR[0], the others are disconnected. Red and Blue lines are a way for the simulator to get your attention and have you think about whether those values are what you actually want.

- Remember to save a screenshot of the simulation of this project.

2.2 Assigned Task – Multi-level logic on the DE1 FPGA

Copy the previous project folder and rename it lab1-2.2.

- In order to speed up processing of returns at Nordstrom, the customer service department wants an electronic detector device. Its goal is to both determine those items that have been discounted, so that the proper rebate can be calculated, as well as help find shoplifters returning their ill-gotten gains.

- There are six products being sold. The UPC code for each is shown, as well as whether it was ever on sale (i.e. sold at a discounted price), and whether it is expensive, and thus is marked when sold.

Item Name	UPC Code	Discounted?	Expensive?
Shoes	0 0 0	No	Yes
Costume Jewelry	0 0 1	No	No
Christmas Ornament	0 1 1	Yes	No
Business Suit	1 0 0	No	Yes
Winter Coat	1 0 1	Yes	Yes
Socks	1 1 0	Yes	No

Note that since there are only 6 items, not all UPC codes are used. The behavior of your circuit for these situations is unimportant (i.e. Don't Care).

- You will be given the UPC code of the item under test (signals "U", "P", and "C" for simplicity), and a detector will check for a secret mark ("M"). Your circuit should have one "discounted" light that lights up whenever a discounted item's UPC code is applied, as well as a "stolen" light that lights up whenever a theft is detected.

- Nordstrom has a special method for finding shoplifters. Whenever they sell an expensive item, they put a secret mark onto it. Thus, expensive items that are purchased are specially marked, while stolen expensive items will not be so marked (inexpensive items are never marked, since it is too expensive to mark everything sold). When there is a return, we want to make sure someone didn't steal the item, then return the stolen item for cash.

- With the rules given, there are four cases for the stolen light logic to consider:
    - An expensive item with the mark is not stolen.
    - An expensive item without the mark is stolen.
    - A non-expensive item without the mark is not stolen.
    - A non-expensive item with the mark will never occur, so is a Don't Care.

- For this circuit, create a design by hand for each of the outputs. You will need to use **K-Maps** or **Boolean Algebra** to optimize the design (K-Maps will likely be the best way). Then, write the corresponding code in Verilog in Quartus II and simulate it. Finally, download the design to the FPGA, and use the switches and lights on the board to connect to the circuit.

Once you have the design working on the FPGA, draw the schematic of the design. This will be used for evaluating the quality of your design.

Note that for all design problems, you will be graded 100 points on correctness, style, testing, etc. For this lab, the bonus goal is to use as FEW logic gates as possible. Each gate (an individual Inverter, an individual AND, and an individual OR) appearing in your hand-drawn circuit diagram costs the same, and gates (other than inverters) can have as many inputs as you want. Unlike lab #2, inverters on the inputs DO count towards your total; each and every gate on your schematic will count as 1 gate. Note that you can only use standard gates (AND, OR, NAND, NOR, NOT, XOR, XNOR) – no AND gates with one input inverted, etc.

- Please switch Sw9, Sw8, Sw7 for U, P, C, respectively, and Sw0 for the secret Mark.

3. High-Level (RTL) Verilog

3.1 Assigned Task – Seven-segment display

- Copy previous project folder and rename it lab1-3.1.
- RTL code for the seven segment display is given below. Enter the code below into a new System Verilog file. Then, create a new module that instantiates the seg7 code twice – one taking an input from SW3 – SW0 and displaying on Hex0 of the board, another taking input from SW7 – SW4 and displaying on Hex1 of the board.

```
module seg7 (bcd, leds);
  input  logic  [3:0] bcd;
  output logic  [6:0] leds;

  always_comb begin
      case (bcd)
        //          Light: 6543210
        4'b0000: leds = 7'b0111111; // 0
        4'b0001: leds = 7'b0000110; // 1
        4'b0010: leds = 7'b1011011; // 2
        4'b0011: leds = 7'b1001111; // 3
        4'b0100: leds = 7'b1100110; // 4
        4'b0101: leds = 7'b1101101; // 5
        4'b0110: leds = 7'b1111101; // 6
        4'b0111: leds = 7'b0000111; // 7
        4'b1000: leds = 7'b1111111; // 8
        4'b1001: leds = 7'b1101111; // 9
        default: leds = 7'bX;
      endcase
  end
endmodule
```

3.2 Assigned Task – UPC code to display

Copy previous project folder and rename it lab1-3.2.

- In section 2 we built a system that took in a UPC code and output whether the item was on sale and whether it was stolen. A neighboring store, Fred's House of Useful Stuff, wants a similar system, but has found that people are changing the UPC sticker on their stuff. To combat that, they'd like you to add a display on Hex5 – Hex0 that describes each product when it is entered – if the description is different from the product actually entered, they know someone is trying to cheat! Note that, since Fred pays his employees embarrassingly little, they tend to be not the brightest folks, so the Hex display should be as simple as possible.

- Since Fred doesn't want to pay for a redesign of the circuit, we will use the same UPC codes, On Sale values, and Expensive markings as before BUT, Fred is willing to sell whatever products you will like him to (Hint: pick products that make good Hex displays!), though the expensive ones must use UPC codes designated "expensive", and the inexpensive ones must use UPC codes designated "inexpensive". BE CREATIVE! And you MUST sell 6 items.

- Determine your items and Hex displays. You can use any or all of the lights on the hex display you choose, upper and lower case, pictograms, whatever. So, for example you could put in "Dress Shoes" as an item, and have the hex display show "ShOE". Note that Fred was beaten with a pair of loafers as a kid and so is unwilling to sell shoes (i.e. no copying from me! And all your items must be DIFFERENT than Lab 2.2's).

- Once you have figured out what you'll display, create a high-level design for the circuit. It will be similar to the seg7 module, with 3 inputs (U, P, and C), but up to 6 7-bit outputs (for each of the 7-seg displays). It will be written in RTL. Simulate it in ModelSim, then hook it to the switches and lights of your board to make sure it works. Finally, create a new module that instantiates one copy of your new display code, and one copy of your lab #3. It should hook them both together, so the system simultaneously computes the HEX display, and the Sale and Stolen lights. Test and debug with ModelSim, then load onto your board.

  ModelSim Tip: When you put your entire design together and simulate in ModelSim, you'll probably need some help organizing all your signals. A couple tips:

  1.) ModelSim can have signals from multiple modules displayed at the same time. Simply select in the left pane the module whose signals you want to display, then drag the signals you want from the middle pane to the waveform window.
  2.) To order the signals, you can click and drag names in the waveform window.
  3.) To organize signals for each module, highlight all of the signal names related to one module in the waveform window, right-click on one of the signal names, and select "Group". Give it a memorable name, then hit okay. You can now move the signals as a group, and hide/expose them easily.
  4.) Once you have organized signals the way you like, remember to save the formatting into the <modulename>_wave.do file.

- You will be graded 100 points on correctness, style, testing, test coverage, etc. Your bonus is the quality of results. In this lab "quality of results" means how nice/creative your display is. Results that are particularly well done, pretty, amusing, or otherwise makes your TA laugh or applaud will get more points.

  NOTE: Your design has outputs for only 6 of the 8 possible UPC codes. For the other two cases, a line such as "default: LEDs = 7'bx;" tells Quartus II that it can treat these cases as a Don't Care condition (if you didn't do this, go back and correct it to do so). Test your design on the circuit board, and record the pattern it shows for these Don't Care conditions.

Deliverables

What's in the report:

Assigned Task in section 1.3:

- A simple, correct explanation of what the mux4_1 circuit does

- The mux4_1 simulation result screenshot in ModelSim

Assigned Task in section 2.1:

- The Multi-Digit Recognizer simulation result screenshot in ModelSim
- A video link to show your program work correctly
    - Demonstrate that the solution will accept your two digits, but no other combinations (all 256 combinations do not have to be demonstrated, just a reasonable amount)

Assigned Task in section 2.2:

- The Multi-level logic simulation result screenshot in ModelSim
- UPC Schematic
- A video link to show your program work correctly
    - Demonstrate all 16 combinations with the 4 switches

Assigned Task in section 3.1:

- The “2 7-seg display” simulation result screenshot in ModelSim
- A video link to show your program work correctly
    - Demonstrate all 16 combinations for each digit (32 total combinations)

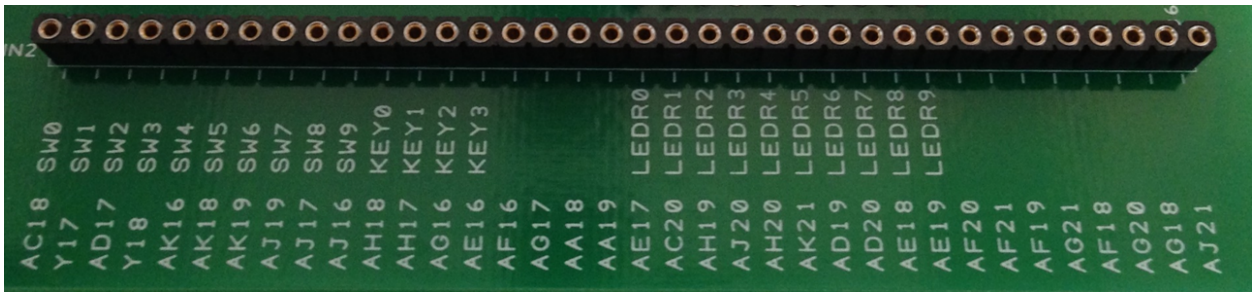
Assigned Task in section 3.2:

- The “UPC code to display” simulation result screenshot in ModelSim
- A video link to show your program work correctly
    - Demonstrate all 16 combinations with the 4 switches

Compress the report and all the project folders into a zip file. Then Submit it to the canvas. If the video is too large, you are allowed to use a direct google drive link and add it as a comment.

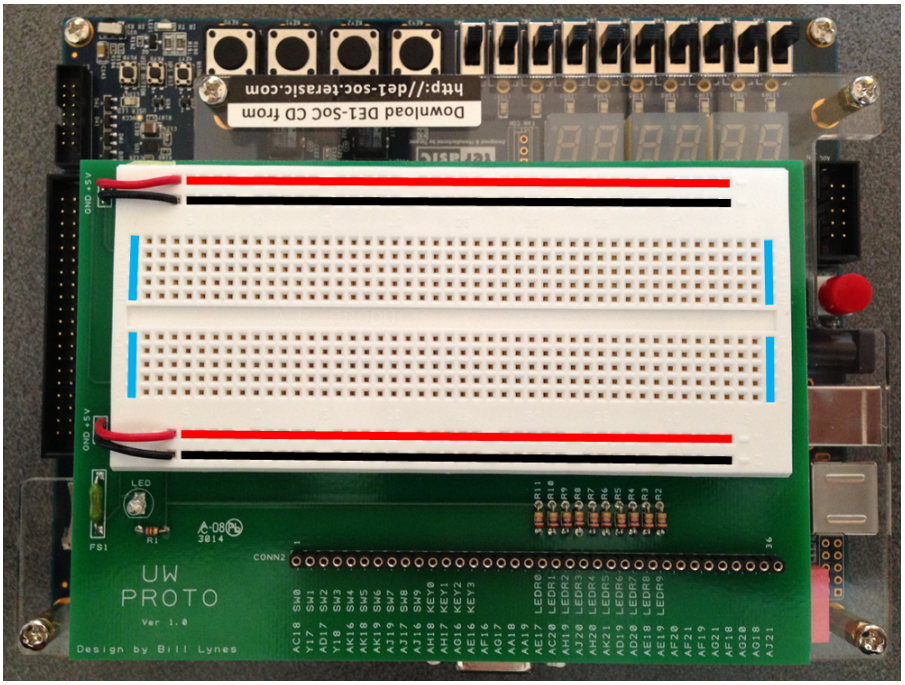
Appendix A

- **Switches, Keys, and LEDs**



The image is a zoomed in picture of the prototyping board. As you can see, each little hole has a corresponding pin mapping (more on this in later labs) and a pre-programmed functionality. Remember the program we put in the non-volatile memory? Well in addition to having the visual display, the program also maps those holes to the indicated switches, keys, or LEDs. So for example, if I hooked up the hole labeled "AJ16 SW9" to the hole labeled "AE19 LEDR9" with a wire, I would be controlling the red LED labeled LEDR9 on the DE1 board with the switch labeled SW9.

- **The Solderless Breadboard**



The white solderless breadboard attached to your DE1 is where you will be building your first few circuits. Notice the red and black wires going to each of the rows at the top and bottom of the board. The red wire denotes VDD (+5 Volts) and the black wire denotes Ground (0 Volts) Never directly connect these two rows together in any way. You should have both a VDD and Ground at the top of your board and a set at the bottom. Even though there are spaces between the rows all of the holes are directly connected to each other. This means you have rails of VDD and Ground both at the top and bottom of your board.

In addition to the rails of VDD and Ground provided for you on the board there are two 48-column grids of holes separated by an indented divider. All 5 holes in one column are connected to each other; however the column on the top of the divider is not connected to the column below the divider. All of these connections are underneath the board so you cannot see them. So remember, if you connect VDD into one of the five holes in a column, all five holes now have VDD running across them.
The red lines mark where VDD runs across the board, and the black lines mark where GND runs across the board. The holes on the board are interconnected as the blue line shows, up-and-down for each 5 hole segment.

- **Wiring your Solderless Breadboard**
  Wires should be inserted as perpendicular as possible to the breadboard and should slide in and out with just a little force. If you find that there is a lot of resistance in either direction, first review the wiring guidelines below, and if that doesn't work, then please talk to the TAs.
  Do not try to force anything larger than stripped wires into the holes, because this could damage the breadboard (at great cost).
  Before doing any work on the breadboard such as wiring and inserting/removing chips, be sure the power is OFF. That is, unplug the power connector while you are constructing the circuit. After you have finished wiring up your design and before you turn on the power, double check the power and ground connections.

- **Wiring Guidelines**
  Wiring your circuit together can often feel tedious, especially in the beginning. However, if you are patient and wire your circuit nicely, you will find that you will spend a lot less time tracking down wiring errors. To aid you in this, here are a few tips to consider while wiring up your circuit. If anything is unclear, ask your TA for an example.
  Make sure your wires are stripped carefully. This means that when you put the wire in your bread board, there shouldn't be any un-insulated wire visible and the wire shouldn't crunch against the bottom of the board.
  Your wires should always be nice and straight. There should be no twists or kinks in them, as they can cause your board to short out when you insert them in your bread board.

Arrange the chips on the breadboard so that only short wire connections are needed. Put tightly connected chips closer together. Chips will only fit with one set of pins on one side of the center divider and another set of pins on the other side of the divider.
Do not make a jungle of wires. Long looping wires that go way into the air are easy to pull out (a hard bug to find later when the circuit doesn't work as intended).
Try to maintain a low wiring profile so that you can reach the pins of the chips and so the chips can be replaced if necessary. The best connections are those that lie flat on the board. Try to avoid wiring over any chips so that chips can be removed easily and replaced.