This lab will have you build some arithmetic circuits, measure their performance using the ISE synthesis report, and build one of them using the components in your lab kit. The new features of the tools that will be introduced are:

- the ability to mix schematics and Verilog code in the same project by wrapping a symbol around some Verilog code and using that in a schematic
- getting information from the synthesis report on what the performance of the circuit would be if it were mapped onto the Xilinx part
- how to use wired components from your lab kit together with the XSA/XST board combination (also see the XSA/XST Pins document on the web site)
- how to use vector data types in your Verilog code, and how to use vectors in your schematics
- how to use some of the other I/O devices on your XSA/XST board combination

Overview

You will design three different four-bit adders using different adder architectures. You will simulate each of these to make sure that they function correctly. You will then synthesize each of them (without downloading them to the board) and look at the synthesis reports to see what the projected performance of the different implementations is. Finally, you will use one of the adders to implement a four-bit adder-subtractor on your XSA/XST board combination using red switches from your lab kit as input and the LEDs on the XSA board as output. For extra credit you can also augment this with a signed digit display using additional LEDs on the XST board.

Don’t panic! This lab handout is long, but it’s mostly because I’m trying to describe things very carefully. It’s not all that much work. That being said, it wouldn’t be good to leave it to the last minute either…

Part I: Adder Design

In Part I of the lab you will design three different four-bit adders. You will simulate the adders using the ISE simulator to verify their functionality. You will then synthesize each adder and look at the synthesis report to see what the projected performance is.
1. Design a one-bit full adder using any general logic gates from the Logic category in the symbols menu in ISE. Name your one-bit adder `fa_bit`. Simulate it to make sure that it functions correctly. Make a schematic symbol for this component.

2. Use your `fa_bit` to make a four-bit ripple-carry adder in a schematic named `add_four`. Your adder should have inputs `X[3:0]`, `Y[3:0]` and `Cin`, and outputs `Sum[3:0]`, `Cout`, and `OVF` for four-bit signed overflow. Use buses for the multiple-bit paths (details in the appendix of this lab handout). Make a symbol for this schematic. Here’s what mine looks like. Notice that the vectors have bus pins that expect the wire connecting to them to be a bus wire.

3. Simulate the `add_four` schematic to verify functionality. Make sure your testbench is self-checking! Also, if you’re testing exhaustively, don’t bother to print out all 512 input combinations (8 data inputs and Cin means 9 input bits)! Make sure that the testbench is self-checking and then let the testbench do the checking. All you really need to see (assuming that things are working) is a line that says which cell you’re checking, that you’re starting the check, and that the check is complete. If the check is complete without any error messages (and your testbench checks things correctly…) then you can assume your circuit is functioning correctly. Note that you can use arithmetic expressions in Verilog in your check statement (see Section 5.5.6 in your book). Remember that you may have to use concatenation to make the right assignment statement (as in Figure 5.33).

4. Synthesize your `add_four` schematic. To synthesize use the **Synthesize-XST** process in Webpack ISE. If you have no mistakes in your schematic you should get a green check mark in the **Process** tab.
5. Now expand the **Synthesize-XST** tab and click on the **View Synthesis Report** option.

This will open a new window with lots of information about how your add_four design was translated into the Xilinx FPGA. There’s a lot of information in this report. One thing you can see is how many basic logic elements (BELS) your project is using. My version used 21 basic gate equivalents (note that these are not mapped to LUTs yet because we haven’t executed the Implement Design phase yet).

```
Cell Usage :
  # BELS            : 21
  # AND2           : 12
  # OR3            : 4
  # XOR2           : 1
  # XOR3           : 4
  # IO Buffers      : 15
  # IBUF           : 9
  # OBUF           : 6
```

If you keep scrolling down you will see what the performance estimate is for this design if it were mapped to the FPGA. My design would take (estimated) 22.202ns for the worst-case path. The worst case path has 11 levels of logic and consists of the path from the *Cin* input to the *OVF* output. You can see how much delay is estimated for each gate and each piece of wire between the gates.
Record the number of gates and the worst-case delay for \textit{add\_four}.

If you’re curious you can also open views that show how the design was synthesized by opening the \textbf{View RTL Schematic} and \textbf{View Technology Schematic} options. Here’s what my four-bit ripple adder looked like:

6. Now repeat this experiment with a four-bit Carry Look Ahead Adder. Design a four-bit CLA adder in a schematic called \textit{cla4} again using Logic gates, and with the exact same inputs and outputs as your \textit{add\_four} design. Simulate \textit{cla4} to demonstrate functionality. You can cut and paste the “guts” of this testbench from
your previous testbench, but make sure to print something for this simulation that marks it as simulating the cla4 design. A simple $display() statement saying which cell you’re simulating will work. Synthesize and record the number of BELS and the worst-case delay for this adder (both the time and the source and destination of the worst-case delay path) from the synthesis report.

**NOTE** that you may have to make the schematic a larger sheet to fit all the components into that sheet. If you click on **Properties** of the schematic, you can change the size of the sheet.

**NOTE:** when you have multiple separate designs in the same project, you need to set one to be the Top Level Module. To make cla4 the top level module, in the **Synthesis/Implementation** view right-click the cla4.v file and select **Set as Top Module**. Now when you **Synthesize** it will be the cla4 schematic that is synthesized.

**NOTE:** When you have multiple separate designs and you’re only synthesizing one, it appears that ISE will try to look at all the designs in the project regardless of whether they are used in the schematic you are synthesizing at the moment. So, don’t have any half-finished modules in your project when you **Synthesize**.

**NOTE:** If you do have half-finished or other unrelated designs that are in your project and appear to be messing up the synthesis you can always Remove them. This will take them out of the list of active sources in the project, but will not delete them. You can add them back to the project later with **Project→Add Source**.

7. Now repeat this experiment a third time with a schematic named **xadd4**. This time use an add4 component from the Arithmetic category in the ISE symbol pane. These are built-in arithmetic primitives from Xilinx. You should again use the same inputs and outputs to the design, simulate the design (again, make sure that the simulation output prints something that identifies this as the xadd4 simulation), synthesize the design, and record the number of BELS and the worst-case delay.

8. What did you learn about the three different adder styles with respect to size and worst-case delay? Who “won” the speed competition? Why do you think the winner was fastest?

9. **Print and hand in the following:**
   a. Print four schematics (**fa_bit, add_four, cla4, and xadd4**). Remember that neatness and clarity are important in schematics.
   b. Print each testbench that you used to test each of the three four-bit adders (**add_four, cla4, and xadd4**)
   c. Print the results of your self-checking testbench from each of your three four-bit adders.
d. Print the output waveform from the cla4 simulation only. Zoom into a part of the waveform just to show how things are going. We don’t need to see the whole thing.

e. Report the number BELS, the worst-case path (source and destination), and the worst-case timing for each of the three four-bit adders. Include a sentence or two about why you think the results turned out the way they did.

Part II: Adder/Subtractor implementation using XSA/XST/kit-parts

In this part of the lab you’ll use your cla4 adder as the foundation for a four-bit adder-subtractor that you’ll map to the Xilinx part. The inputs will be switches that you wire on the prototyping area of your XST board. The output will be the 7-segment display on the attached XSA board.

1. Modify the BCD to 7-segment LED circuit that you designed for Lab2 so that it displays normal Arabic numbers instead of Klingon, and instead of displaying E for inputs in the range of 10-15 it displays the correct hex digit. That is, modify the Verilog code so that it’s a HEX to 7-segment display. The patterns on the LEDs should look like:

```
01234567
89abcdef
```

All 16 hex digits on the seven-segment display

You can do this by modifying the Verilog assign statements for each of the seven outputs \((a, b, c, d, e, f, g)\). You can also re-write this using vectors for the input \((b[3:0])\) for example) and the outputs \((\text{led}[6:0])\) for example). See section 5.5.3 for details about using vectors in Verilog. You can also use a Verilog case statement
to make things easier if you like. Look ahead to Section 6.6.3 for details. A code fragment from my Verilog code for this looks like this:

```verilog
module hex_v(b, led);
  input [3:0] b;
  output reg [6:0] led;
  always @(b)
    case (b)
      4'b0000 : led = 7'b1111110; // 0
      4'b0001 : led = 7'b0110000; // 1
      4'b0010 : led = 7'b1101101; // 2
      ...etc...
      default : led = 7'b0000000; // nothing
    endcase
  endmodule
```

2. When your hex to 7-segment Verilog code is working, create a schematic symbol for it using the Design Utilities. You can now use this module in a schematic just like it was made of gates. The symbol will show up in your Lab3 project symbols in the same way as the `fa_bit` symbol does.

3. Now use the `cla4` and your new hex to 7-segment display to make an adder-subtractor for four-bit signed numbers. The schematic should be named `add_sub4`. See section 5.3.3 for details. You can assume that there is no separate carry-in for this add/sub unit. You can use the `Cin` signal in your `cla4` to achieve the subtract function.

4. You should have the following inputs and outputs to your add/sub unit (named `add_sub4`):

   Inputs: `X[3:0], Y[3:0], control`
   Outputs: `LED[6:0], Cout, OVF`

   The `control` input should be 0 when you want to add and 1 when you want to subtract. For subtraction you should compute `X-Y`. The `OVF` signal should be high if there is an overflow from the signed operation. The `Cout` is the carry-out of the unsigned operation.

   The output should show up as a four-bit hex number on the 7-segment display. The `Cout` signal should be connected to the DP (decimal point) LED on the 7-segment display. The `OVF` signal should be connected to bar-graph LED segment 1 on the XST board (P68). See the XSA/XST pins handout for details on which pin assignments to use on the FPGA to make things come out on those LEDs. Use the pin assignment technique from Lab2 to assign signals to the correct pins.
Inputs for X and Y should come from a pair of red switch packs from your lab kit. You should wire those switches into the prototyping area on the XST board. Make sure that all outputs from the switches each have their own pullup resistors (remember from Lab1). Make sure that the white switches are all pointing to right for 0 and the left for 1. You can pick which pins you want to use on the FPGA for these inputs, but read the XSA/XST Pins handout carefully. You will wire your switch outputs to the prototyping header to make them connect to the FPGA.

**NOTE:** Don’t bother to put the switch components (from the 3700Lib) in the schematic. They won’t synthesize to the Xilinx part so to avoid confusion you can just leave them out of the schematic. You still need to wire them up though! If you really want to put them in the schematic, you’ll need yet another level of hierarchy in the schematic. Make your `add_sub4` schematic and make a symbol for it. Then make another schematic (named `top`, for example), and in that `top` schematic put the 8 switches (with pull-ups) and one instance of `add_sub4`. Then when you synthesize and implement, make sure that `add_sub4` is defined as the Top Module so that Xilinx doesn’t try to synthesize the red switches.

The input for the control signal should come from switch number 8 on the 8-way DIP switch (DIPSW8, pin P79) on the XST board. Again, consult the XSA/XST Pins handout for details.

5. When your schematic is complete you should have two 4-bit input buses for X and Y, and a single-bit for control as input. The rest of the circuit computes X+Y when control is 0 and X-Y when control is 1. The four-bit sum should go through your hex to 7-segment circuit to display on the 7-segment display as a hex digit. Cout and OVF should go to pins that are connected to the other LEDs as described earlier.

6. Simulate this circuit to make sure it functions correctly.

7. Make sure you have all the correct pin assignments in your UCF file (remember Lab2 for details about assigning pin locations). Then synthesize, implement, and generate programming file for this circuit.

8. Wire your switches for the X and Y inputs. Wire the switch outputs to the correct prototype header pins (that you chose as inputs to the FPGA for X and Y). Use XSLOAD to download your add/sub circuit (as a .bit file) to the Xilinx part and demonstrate that it functions. The TA will flip inputs and check that the correct sum shows up on the 7-segment display, and that Cout and OVF are showing up correctly.

**NOTE:** Make sure that you put your XSA board into the XST board correctly. The fastest way to destroy the XSA board is to put it in the XST headers backwards and then apply power! Poof. There are notes and arrows on the XST board that tell you how the XSA board should be oriented.
NOTE: Because of interactions between the various parts of these boards, it’s important that you put all the unused DIP switches in their OFF position before applying power, especially when you’re using both the XSA and XST boards together.

9. Print and hand in the following for Part II:
   a. Print the schematic for your add_sub4 circuit.
   b. Print your Verilog code for your hex to 7-segment circuit
   c. Print your testbench that you used for testing
   d. Print the result of your testing. Remember we don’t want pages and pages of correct circuit output printed out. Make your testbench self-checking.
Extra Credit (15 points):

This is totally optional and will add up to 15 points to your lab score (nominally 100 points). For extra credit take the four-bit sum from your add/sub circuit and design a new circuit to also show the Sum output as a signed number on the two 7-segment LEDs on the XST board. You should use XST-LED2 as the magnitude portion of the signed number, and segment G on XST-LED1 for a negative sign. So, if the Sum from the add-sub circuit is 1011 which shows up as B on the XSA display, for example, it would show up as –5 in the other two LEDs on the XST board. You will need to figure out how to design this circuit (I recommend doing this in Verilog!). The pin assignments for the two 7-segment displays are found in the XSA/XST Pin handout.

NOTE: Be very careful! If you do this you MUST also load the dwnldpa2.sva file to your XSA board BEFORE downloading your circuit configuration .bit file! You must also make sure that all the four DIP switches on the XSA board are in the OFF position. Read the XSA/XST Pins handout carefully before you attempt the extra credit part!

Turn in the Verilog code for your new display circuit, the modified schematic for add_sub4, and the simulation testbench and output. The TA will make note of the display during the demo.
Appendix: Using Buses in ISE Schematics.

Buses are a great way of representing groups of bits that should be treated as a unit. They are the equivalent of vector data in Verilog. The only real trick to using buses in Schematics is to name the wire with a bus name. This will then turn the wire into a thick bus wire. You can then connect individual wires to that bus using a “bus tap.” Here’s an example.

Start by putting down some components in a schematic. The following example uses four random components that I will connect to two input buses and one output bus. I’ve started by making wires hanging out in space that will become my buses.

![Diagram showing components and wires]

Now I can turn those wires into buses by giving them bus (vector) names. The names should have a variable name part and a bus description. Something like A[3:0] or mybus[3:0]. These names will expand into four separate names like A[3], A[2], A[1], A[0]. Note that even though I’m using square brackets because that is what Verilog uses, the schematic will use round parenthesis. It all works together without complaint though.

Select the Add Net Name widget (just to the right of the Add Wire widget). Now in the Options box enter the bus name.
Now when you select one of the wires in the schematic, it takes that name and turns into a thick wire. Here’s the example after naming all three buses.

Now you can make bus taps for each of the individual signals and attach those taps to the bus. Select the **Add Bus Tap** tool (looks like 📑). Now select the direction for the tap, and select one of the buses to get the **Selected Bus Name** filled in. Now as you click on inputs to the gates, a named bus wire will be made.

Note that each tap has name of the wire that is being tapped from that bus. You can also make an individual tap by typing the **Net Name** in directly. Here’s a finished example circuit. Note that I changed the direction of the tap for the output bus.
You can now simulate this circuit and call out each bus signal individually in the testbench using the \texttt{A[2]} or \texttt{mybus[1]} notation, or refer to a bus as a whole as in \texttt{A = 4'b0110}; or even talk about part of a bus with \texttt{A[2:1] = 2'b11};

Buses are the exact same thing as vector types in Verilog. So, if you define a vector in Verilog and ask for a schematic symbol, you’ll get a bus connection on the symbol. If you have buses in your schematic, they’ll act like vectors in the testbench.