ECE 152A LABORATORY 2

Objectives:

1. Understand the trade-off between time- and space-efficiency in the design of adders. **In this lab, adders operate on unsigned numbers.**
2. Learn how to write Verilog code for different adder implementations. Apply the principles of hierarchical design. Verify each design by simulation.
3. On an FPGA, implement a 4-bit adder that is made up of two 2-bit adders. The adder operates on unsigned numbers and should be able to detect overflow.

Deliverables and Due Dates:

Lab # 2 Problem Set is due at the beginning of your lab section when Lab # 2 starts. This comprises completing Part A below. (See the "Design Steps in Detail".)

In addition, you have to come to the lab having read the rest of this lab (i.e. Part B). If you come to the lab having written the Verilog code for the three different implementations, you will get a head start on this lab. **(Please note that we take the Honor Code very seriously: Copying Verilog code from someone outside your lab team will get you an F in this course and referral to the Committee on Student Misconduct. You must write the Verilog code with only your own team.)**

Lab # 2 check-out occurs in the following week. This comprises completing all of the remaining steps in this lab and demonstrating the correct functionality of your simulations and the final implementation in FPGA.

Grading will be done as follows for steps completed on time:

Part A: (due the beginning of the lab session when Lab # 2 starts.)

Step A.1:	15%
Step A.2:	15%
Step A.3:	15%
Step A.4:	5%

Part B: (due the lab session in the following week.)

Step B.1-B.2 for A.1:	10%
Step B.1-B.2 for A.3:	15%
Step B.3-B.4 for A.3:	25%

The grades for Part B are based on demonstration of functionality for each step. For each step in Part B, partial credit will usually NOT be given.

Design Strategy (Outline):

Part A. We will compute the area and delay of three different implementations of the 4-bit adder by using the simple gate delay model.

Step A.1. Analysis of the ripple carry adder implementation
Step A.2. Analysis of the two-level AND-OR implementation
Step A.3. Analysis of two 2-bit adders in cascade
Step A.4. Comparison

Part B. We will write the Verilog code for the implementations A.1 and A.3.
For implementations A.1 and A.3, do the following steps:

Step B.1: Write the design in Verilog.
Step B.2: Simulate the design using ModelSim and verify operation.

For implementation A.3, do the following steps in addition:

Step B.3: Download to the FPGA.
Step B.4: Test the FPGA implementation and verify operation.

Design Steps in Detail:

Part A: Worst-case delay and area cost analysis of different implementations

In this step, we will use analytical techniques and the simple gate delay model to compare three different implementations of the 4-bit adder with respect to their worst-case delay and area performance. (We will see more sophisticated adders such as the carry look-ahead adder later in the course. But the design of those adders use the same ideas of time- and space-efficiency.)

In all of the steps below,
(1) Your adder design must be able to detect overflow,
(2) You should have a 1-bit carry-in to the 4-bit adder. (This way, you can cascade such 4-bit adders, if you want to build larger adders.)

Step A.1. In this step, we implement the 4-bit adder as four full-adders in cascade. This is called a "ripple carry adder" because the carry bit has to ripple from one element to the next. We expect that the worst-case delay of the ripple carry adder will be dominated by the carry chain. In this exercise, we want to quantify such intuitions.

(a) Using the simple gate delay model (which associates a delay with each gate), find an expression for the maximum (i.e. worst-case) delay of the 4-bit ripple carry adder. In addition, find a "critical input transition" that results in this maximum delay.

(b) What is the number of 2-input gates that this implementation uses?

(c) Your textbook uses an "area cost function" defined as the sum of the number of inputs that are input to all the gates in the circuit plus the number of gates in the circuit. This cost function is reasonable for CMOS VLSI implementation of adders because the wires on a CMOS chip add significantly to the area of the circuit. In fact, roughly 80% of the area of a VLSI chip today is consumed by wires and only 20% by transistors.

Using this cost function, compute the area cost of this 4-bit adder implementation.

Step A.2. Now, we want to estimate the delay and the space-complexity of the 4-bit adder when it is designed as a combinational logic block from the truth table directly and then minimized using Karnaugh maps to arrive at a minimum sum-of-products expression for each output.

The problem is that the truth table for this 4-bit adder is large. (How many rows does it have?) So, in this section, instead of finding the explicit expressions, we will try to estimate the delay and space-complexity.

Without writing down the truth table, write the Boolean equation for each sum bit in terms of the internal carry bits. Now, we would have to substitute for these carry bits successively in terms of the inputs. You should do this substitution for sum[0] and

sum[1] and see what results in terms of AND, OR, NOT gates. By doing this for sum[0] and sum[1], you should be able to see how fast these terms multiply. Now, try to visualize what the resulting circuit will look like, without finding the expressions for sum[2] and sum[3]. Sketch the schematic. Then, answer the following questions:

(a) How many logic levels will there be in the final circuit?

(b) What will be the dependency between the variables? For example, sum[0] depends on which input variables? And sum[1] depends on which input variables? Similarly, what input variables does each of sum[2] and sum[3] depend on?

(c) In the final circuit, where do you expect the gates with large fan-in to be located? Where do you expect small fan-in?

(d) Along which path do you expect the worst-case (maximum) delay to lie? Is this a result of the number of logic levels that the input signal has to travel through?

(e) Estimate the number of gates and the area cost of this implementation.

[You should be aware that there are tools that can automate 2-level logic minimization. The tool "espresso" available for free from Berkeley at http://www-cad.eecs.berkeley.edu/Software/software.html is such a tool. If you would like to experiment with it. Download espresso and perform logic minimization for this example. Then, you can get a clear quantitative answer rather than just an estimate.]

Step A.3. Now, we implement the 4-bit adder by using two 2-bit adders. Each of the 2-bit adders is designed from a truth table and using logic minimization.

Write down the truth table for a 2-bit adder. Then, find minimum sum-of-products expressions for the outputs. Draw a block diagram schematic to show how to exactly cascade two 2-bit adders to get a 4-bit adder.

(a) Compute the worst-case delay and find a critical transition.

(b) Compute the number of gates and the area cost of this implementation.

Step A.4. Now, compare quantitatively the time- and space-efficiencies of the implementations in the previous steps. (For A.2, we have only estimates from our analysis.) What is your conclusion? When should we prefer which implementation?

Part B: Functional Simulation, FPGA download and testing for functionality

In this part of the lab, we will first write two of the implementations of the previous part in Verilog. Second, we will simulate each of these using ModelSim and verify their operation. (In the ModelSim simulation, we will not simulate the gate delays. We leave the timing simulation of circuits to later labs.)

Then, for simplicity, we will download to the FPGA only one of these 4-bit adders: the one that is comprised of two 2-bit adders in cascade. Then, we will test the correctness of this implementation in FPGA directly.

<u>Step B.1</u>:
In this step, we write the Verilog code for a particular implementation of the 4-bit adder.

Step B.1 for A.1:
In order to facilitate the design process, we use "hierarchical design". For example, in order to implement the 4-bit ripple carry adder, you will implement a full-adder module in Verilog and use these full-adder modules to implement a 4-bit ripple-carry adder. To do this, first, you should write down the module declarations for the full-adder and the ripple-carry adder. (The comments that begin with "Students:" is a comment for you, not a comment that should appear in your final program.)

```
//Students: Always start your module declarations with a description of what the
//module does:
//A.1: The module "fourBitAdder_FourByOne implements a 4-bit adder using
//four 1-bit full adders in cascade. The phrase "FourByOne" refers to the fact that
//we have four 1-bit adders in cascade.

//Students: write down the inputs and outputs, do not forget overflow detection.
module fourBitAdder_FourByOne (....);
        input ...;
        output ...;
        //Students: implementation goes in here.
endmodule

//Students: write down the inputs and outputs
module fullAdder (.....);
        input ...;
        output ...;
        //Students: implementation goes in here.

endmodule
```

Next, fill in the code for the fourBitAdder_FourByOne in Verilog assuming that you have the fullAdder module. Note that you will instantiate the full adder module inside the fourBitAdder_FourByOne module.

Then, fill in the Verilog code for the fullAdder assuming that you have gate-level primitives (e.g. and, or, not gates).

Run the Verilog compiler and check that it compiles.

Step B.1 for A.3:
Write the module for the A.3 implementation, just as we did for the A.1 implementation.

You HAVE to use the following name for this module:
fourBitAdder_TwoByTwo

(This refers to the fact that we have two 2-bit adders in cascade.)

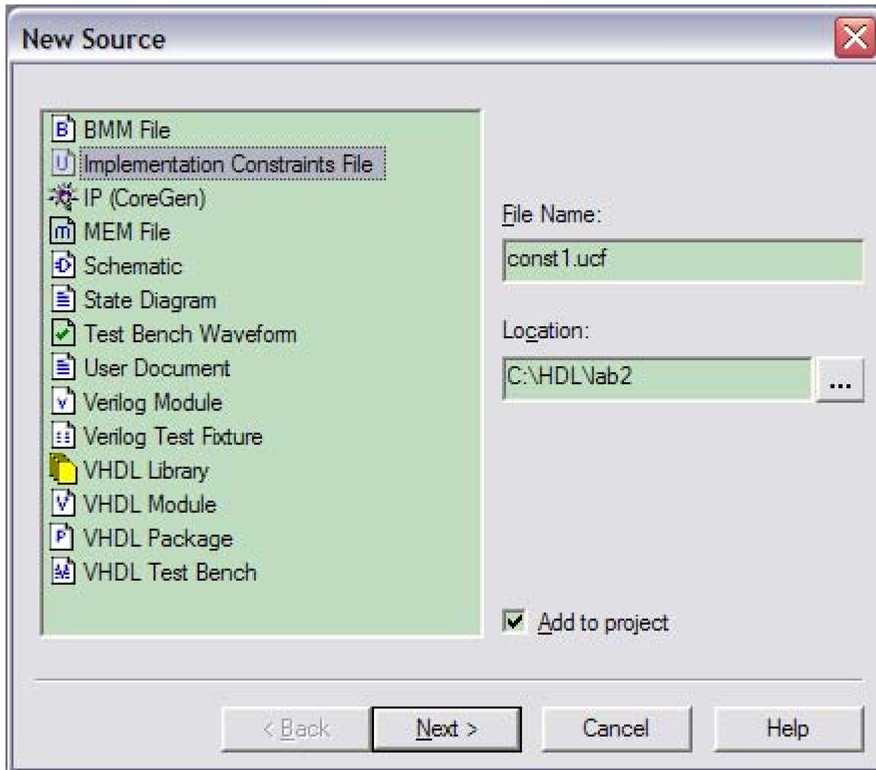
Step B.2:

In this step, simulate the design using ModelSim. To do so, you first have to create a test bench for the design in Xilinx Program Manager by creating a new source. Then, verify Modelsim waveforms that each of your adder implementations is operating correctly.

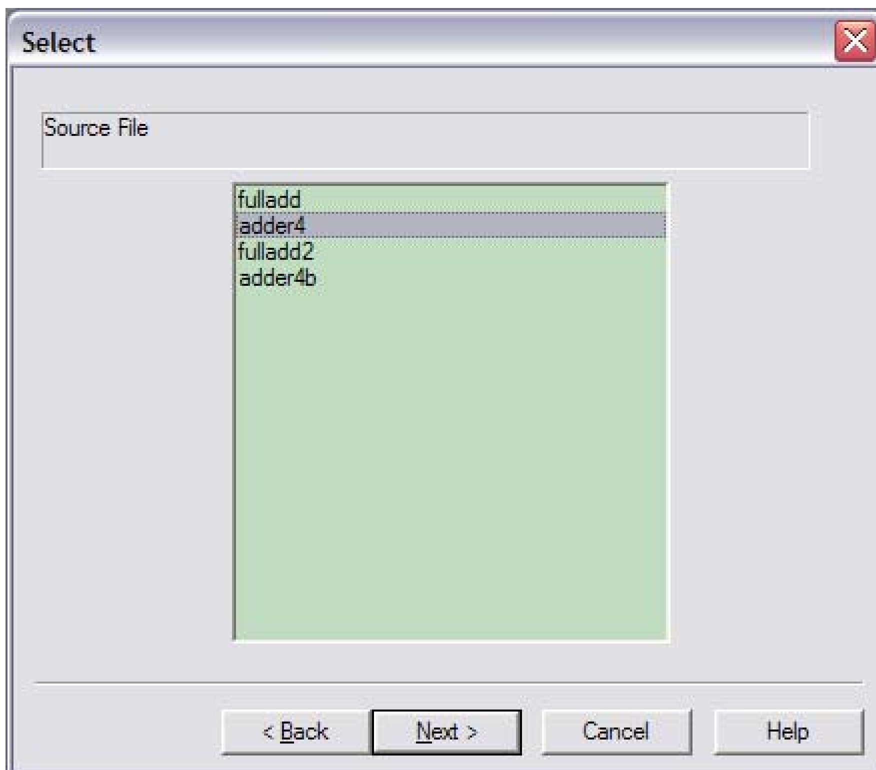
Step B.3:

In this step, we download the Verilog design to the FPGA (using Xilinx Program Manager). The steps start on the next page.

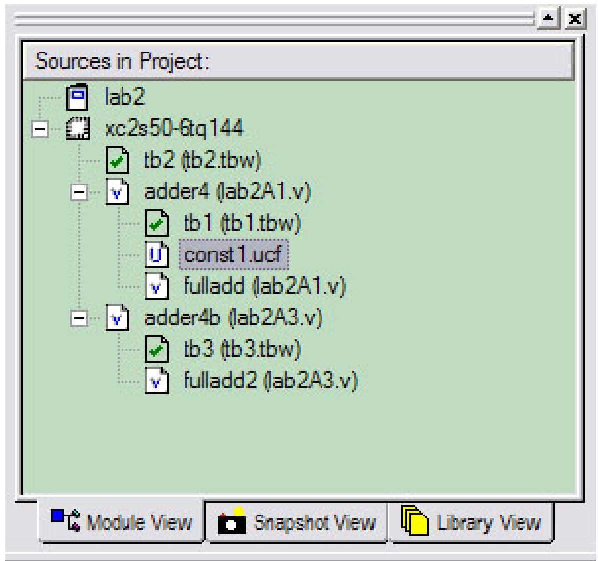
Creating the constraints file before FPGA implementation



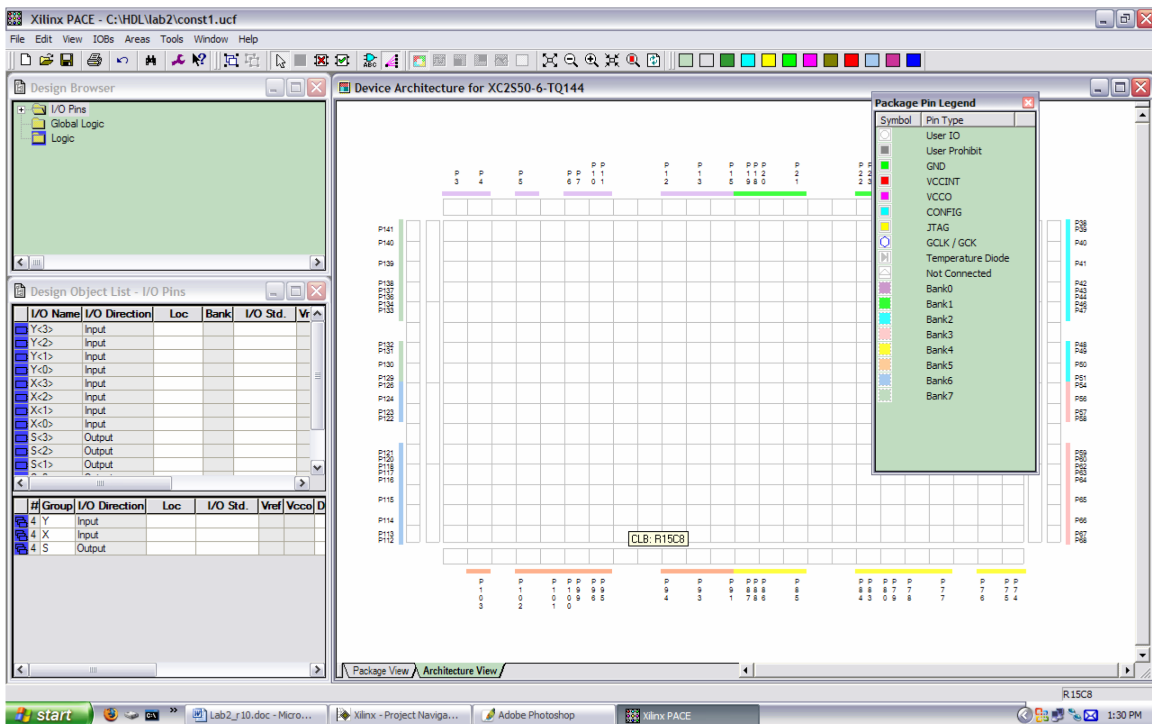
Select the Verilog file you want to associate with the constraints file.



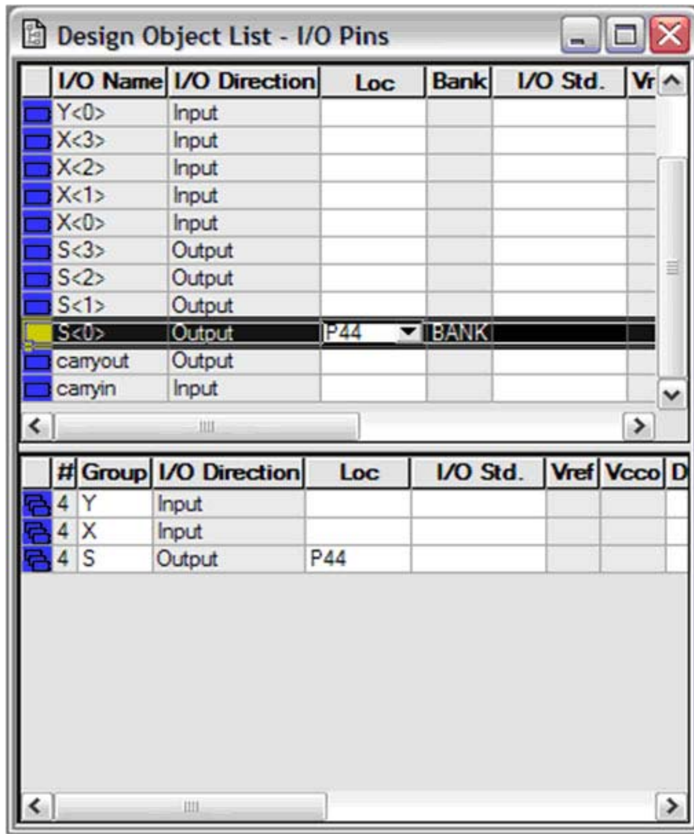
The following is the hierarchy view that should show up in your project list after creating the .ucf file.



Double click on the .ucf file you created. This new window should show up.



On the left side of the window, you can manually enter the pin number to implement at a certain FPGA pin.

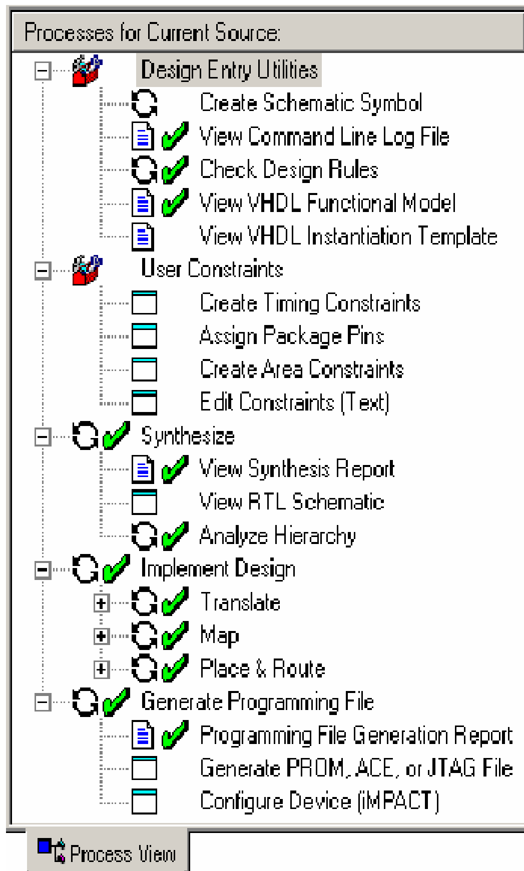


To find out which pins of the board you can use for external logic refer to the XSA50 manual. As you can see below, some of these pins can be used as general-purpose I/O with the bitstream.

> **Configuration Pins** (30*, 31*, 37, 38*, 39*, 44*, 46*, 49*, 57*, 60*, 62*, 67*, 68*, 69, 72, 106, 109, 111): These pins are used to load the SpartanII FPGA with a configuration bitstream. Some of these pins are dedicated to the configuration process and cannot be used as general-purpose I/O (37, 69, 72, 106, 109, 111). The rest can be used as general-purpose I/O after the FPGA is configured. If external logic is connected to these pins, you may have to disable it during the configuration process. The DONE pin (72) can be used for this purpose since it goes to a logic high only after the configuration process is completed.
>
> **Free Pins** (77*, 78*, 79*, 80*, 83*, 84*, 85*, 86*, 87*): These pins are not connected to any other devices on the XSA Board so they can be used without restrictions as general-purpose I/O through the prototyping header.

When entering a pin number always insert a "P" before the number: For example, pin location 54 should be entered as p54 or P54.

After entering all the pin locations, save and exit. Keep in mind that you need 9 input pins and 5 output pins.



2. Now, you will synthesize and implement the design. Perform all the steps that are check-marked in the figure.

3. To download the design to the FPGA, we need to generate a programming file. To do so, double-click on "generate programming file". They should be check-marked as shown in the figure. This creates a .bit file.

Now you are ready to configure your board with the ".bit" file you generated.
But before you do so, remember you still have to create the external logic part for your inputs and outputs.

External Logic:
In this step, you have to use 5 LEDs as your outputs: You need 4 LEDs for "sum0", "sum1", "sum2", "sum3", and one LED for "overflow". You will use 9 dip switches as your inputs: You need four of them for A0,A1,A2,A3; four of them for B0,B1,B2,B3; and one of them for Cin. You need to connect them through the general purpose I/O pins that we discussed above.

By carefully selecting the output pins, it's possible to utilize the 7-segment display and 4-bit dip-switch on the XSA board.

Step B.4:
In this step, we test the design on FPGA for all valuations (i.e. input combinations). To download the .bit file you will be using XSTOOLS. For further details, please refer to the XSA50 manual xsa-manual-v1_2.pdf , pages 13-15. Now you are ready to test varies combinations of inputs and observe the output. Verify that the outputs (LEDs) are functioning correctly.