LAB 2 – Mapping Your Circuit to FPGA

Goals

- Design a 4-bit adder using structural modeling.

- Learn how to interface to the components on an FPGA Board.

- Implement your design on the Basys 3 FPGA board and see your circuit running.

To Do

- The first step is designing a simple "1-Bit Adder" circuit.

- In the next step, you will instantiate 4 copies of the 1-bit adder to implement a "4-bit Adder".

- Then, you will learn how to use Xilinx Vivado for writing Verilog and how to connect to the Basys 3 board.

- Finally, you will program the FPGA and get the circuit running on the FPGA board.

- Follow the instructions. Paragraphs that have a gray background like the current paragraph denote descriptions that require you to do something.

- To complete the lab you have to show your work to an assistant before the deadline, there is nothing to hand in. The required tasks are clearly marked with gray background throughout this document.

Introduction

In the previous lab we drew some small circuits. We did not, however, see what our circuit would do. In this exercise, we will see how the drawing can be implemented on the FPGA board and see our circuit operating.

In the last week, our designs were relatively small. This week, we will design a slightly larger circuit using structural modeling, which we have already learned in class. First, we will design a small circuit that is able to add two 1-bit numbers. Then, we will combine multiple instances of the 1-bit adder circuit to make a larger 4-bit adder. Such a hierarchical design technique is very important in allowing us to build large and complex circuits.

Binary Addition

In one of the later lab exercises (i.e., Lab 5) we will work on the design of arithmetic operations circuits. Today, we will start simple by designing an adder circuit.

We give a small summary here about a simple method to add two binary numbers.

Essentially, the binary addition is the same as the decimal addition that you are used to. The only difference is that a digit in a binary number can only be 0 or 1, instead of the numbers from 0 to 9.

Consider the simple example given below:

```
    1  1
       8  7  3
 +     3  6  2
 ───────────────
    1  2  3  5
```

We start from the right and first add $3 + 2$, which results in 5. Since this number is less than 10, we can directly write it as it is. The next digit is $7 + 6$, which results in 13. Now we cannot directly write the result since it exceeds the largest possible number we can use in a digit. We write 3 to the result, and have a 'carry' that we move over to the next digit. In the third digit, we now have $1 + 8 + 3 = 12$ since we need to also add the carry from the right side. This result also produces a carry, which also gives the final result.

Now let us look at the same addition when expressed as binary numbers.

```
        1      1  1      1
    0  1  1  0  1  1  0  1  0  0  1
 +  0  0  1  0  1  1  0  1  0  1  0
 ─────────────────────────────────
    1  0  0  1  1  0  1  0  0  1  1
```

The principle is the same. We start from the right. Add each digit. $1 + 0$ is 1, $0 + 1$ is 1, $0 + 0$ is 0 and so on. Once we try to add $1 + 1$ we realize that the result $(2)_{10}$ can not be expressed in one binary digit $(10)_2$. So we have one 'carry' bit that is moved to the left.

Part 1 - One bit addition

Looking at the example, we see that to add two one-bit numbers A and B we would need two outputs. One 'Sum' output that is the result of the addition, and an additional 'CarryOut' output that indicates that the addition overflowed and produced a carry bit to the right. Such a circuit is called a half-adder.

The problem with the half adder is that it produces a 'CarryOut', but cannot accept a corresponding 'CarryIn' as an input. A circuit that can add three inputs A, B, CarryIn and has two outputs Sum, CarryOut is called a full-adder.

By sequentially connecting the CarryOut signal of one stage to the CarryIn stage of the following digit we can build much larger adders easily.

Fill out the truth table for a full adder circuit given below. We use the inputs A, B, CI and the outputs are called S and CO. Derive the Boolean equations for both outputs. Apply logic minimization techniques to come up with a simplified full adder circuit.

Part 1a

Complete the following truth table for the full-adder circuit.

CI	B	A	CO	S
0	0	0		
0	0	1		
0	1	0		
0	1	1	1	0
1	0	0	0	1
1	0	1		
1	1	0		
1	1	1		

Part 1b

Derive Boolean equations for CO and S (using whatever method you like).

CO   =

S   =

Part 1c

Draw the schematic of the full adder circuit according to the equations you have derived.

Now we have everything in place to implement a 1-bit full adder in Vivado.

Part 2 - Starting Vivado and Creating a Project

Please follow the instructions below.

1. **Start Vivado:** Click on the Start menu and go to

Programs → Xilinx Design Tools → Vivado 2016.4.

2. **Create a 'New Project':** Inside Vivado, choose File → New Project. This will bring up a new project wizard.

- Specify the "Project name". You can simply call it Lab2. You can also specify where the project files will be stored using the "Project location." Click next.

- Select "RTL Project" as the project type and click next.

- In the "Add Source" dialog, we do not have source files yet so just click next.

- In the "Add Existing IP" dialog, click next.

- In the "Add Constraints" dialog, click next.

- In "Default Part", we need to select the FPGA board that we are using. Type "xc7a35tcpg236-1" in the Search field and you should be left with only one option. Select and click next.

- In "New Project Summary", you can see the project configuration. Make sure that the default part and product family information reads:

Default Part: xc7a35tcpg236-1
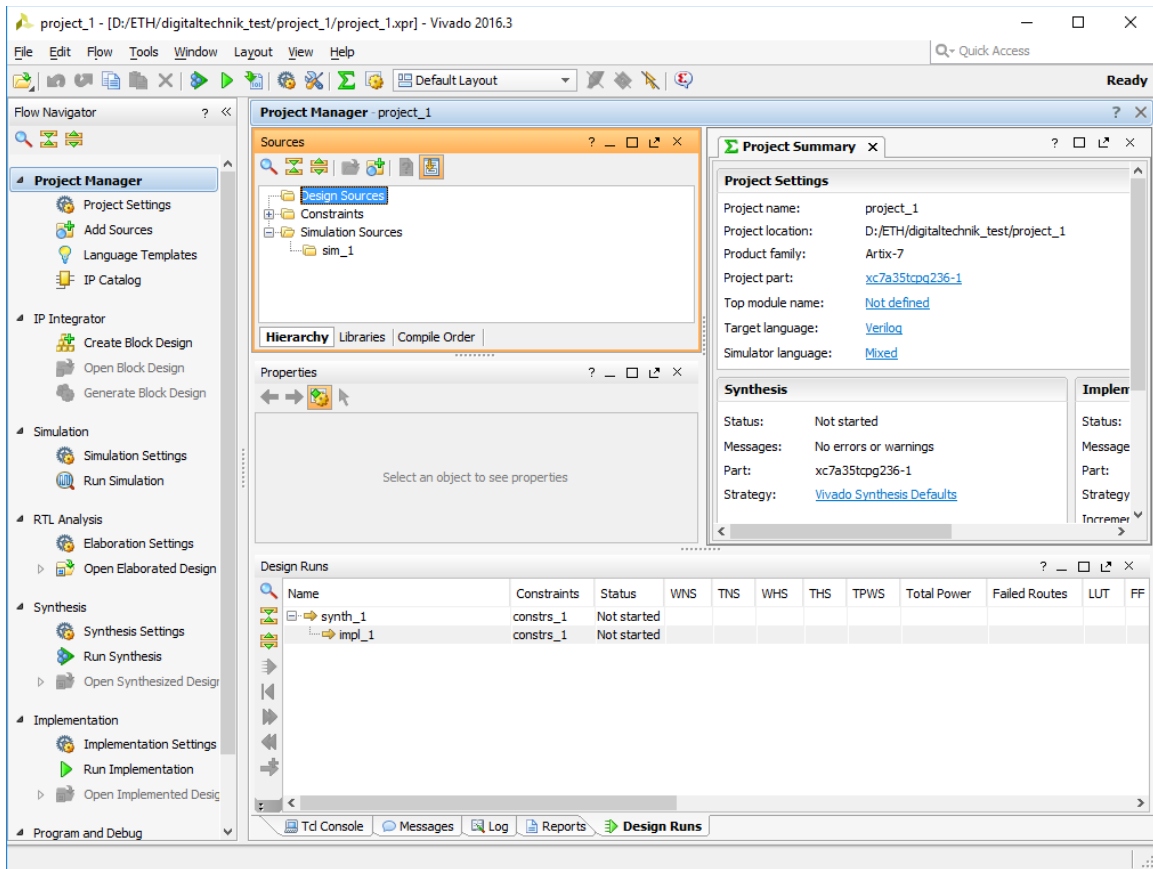
Product: Artix-7

Family: Artix-7

Package: cpg236

Speed Grade: -1

Once everything checks out, click finish to create the project!

After creating the project, you should see the following view:

On the left side of Vivado, you will find the "Flow Navigator" which you will extensively use to develop your project and finally program it to the FPGA board.

Let's start a new Verilog code to implement the 1-bit full adder.

Inside Flow Navigator, under "Project Manager", click on "Add Sources". Select "Add or create design sources", and click next. Click "Create File". A new dialog box as follows will pop up.
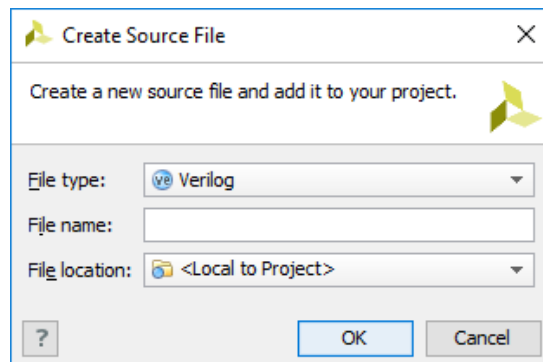
Keep the type as Verilog and choose a file name for your 1-bit FA. E.g., "FullAdder.v". Then click OK. The dialog will disappear and you will see your design in the list. Click Finish. A new dialog will popup asking you to define the module. Simply skip this process and press OK and ignore the warning message.

In the Project Manager box, you should see your newly created file. Double click on it to edit it.

Now, you are all set to start implementing the Full Adder in Verilog.

Part 3 – Implementing a Full Adder in Verilog

First, we will need to specify the input and output of the full adder.

Complete the input/output list of the FullAdder module. Your code should look something like:

```
module FullAdder(input a, input b, input ci, output s, output co);

        <your logic here>

endmodule
```

Implement the full adder logic inside the FullAdder module.

Part 4 – Implementing a 4-Bit Adder in Verilog

Now that we have designed a full adder, we can use it to design a 4-bit adder.

We have two 4-bit numbers (A and B) in our adder. The output is the sum (S) of these two 4-bit numbers and is 5 bits wide, 4 Sum bits of the individual full adders plus the Carry Out bit of the final (left most, most significant) full adder. All but the first (right most, least significant) full adders have their Carry In signals connected to the Carry Out of the full adder to their left. The first (right most, least significant) full adder does not have anything to its right, so there is no Carry In entering this full adder. We can connect the Carry In input to logic-0 (or ground, the symbol for this being called 'gnd'). You can see an example in the following figure.
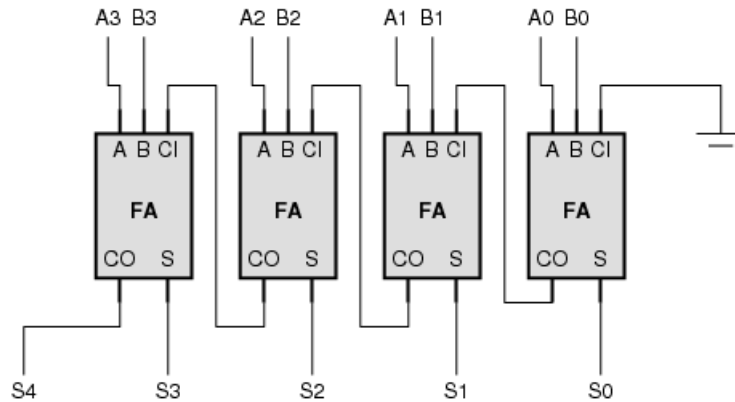
In the description (and the figure) the organization is so that the least significant bit is on the right and the most significant bit is on the left. This is partly for didactic purposes; the organization mimics the way we write numbers on paper.

Given the description, we can now start a Verilog code that contains four instances of the full adder (FA).

Similar as before, create another source file. You can call it FourBitAdder.v. Again, you should first complete the input/output list so that your code looks like:

module FourBitAdder(**input** [3:0] a, **input** [3:0] b, **output** [4:0] s);

    <your logic here>

endmodule

Afterwards, implement the combinational logic of the 4-bit adder by creating 4 instances of the full adder.

Part 5 – Running Your 4-Bit Adder on the Basys 3 FPGA Board

After designing the 4-bit adder, we can now program it to the FPGA board. An FPGA is a reconfigurable chip that allows you to map a given schematic (essentially a netlist of interconnected logic gates) in itself. Depending on the type an FPGA can map anywhere from a few hundred to more than a million gates.

In this course, we will be using the Basys 3 boards from Digilent, as shown below.

Power switch (Remember to turn it on!)

Artix®-7 FPGA

16 light emitting diodes (LEDs)
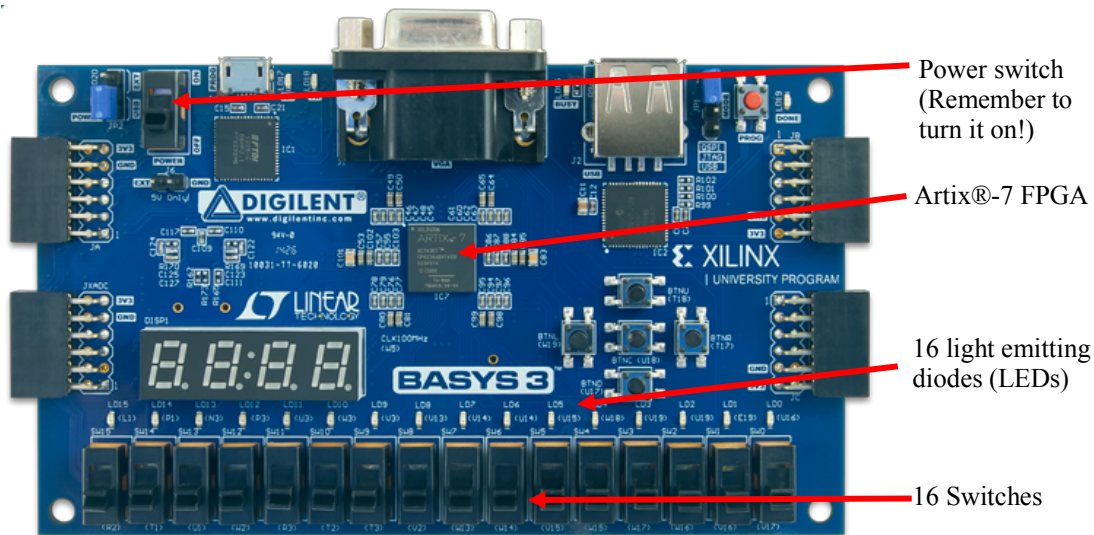
16 Switches

This board includes, among other devices, a Artix®-7 FPGA chip which we can program, some switches, some light-emitting diodes (LEDs), and some other cool features. You can learn more about the Basys 3 Starter Board from:

http://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/

In this lab we have two four-bit inputs that we can map to eight switches (e.g., labeled SW7 to SW0), and a 5-bit output which we can map to the LEDs (e.g., LD4 to LD0). From the board's reference manual,

https://reference.digilentinc.com/_media/basys3:basys3_rm.pdf

we can find out to which FPGA pins the switches and the LEDs are connected (page 15). If you look on the board you see that the pin numbers are also written next to the name of the LEDs and Switches. For example, the right most switch (SW0) is connected to pin V17.

What we now need to do is to tell Vivado which of our inputs and outputs should be connected to which pin exactly. This is done using a Xilinx Design Constraint File (XDC). Typically, a XDC contains also timing related constraints, telling Vivado how fast the circuit should be, etc., but at this moment we do not need these.

You can generate a XDC file the same way you generate a new source file by selecting *Add Sources* and then *Add or Create Constraints* under *Project Manager*.

Once you have generated the new file, you can select this file in the Project Manager, under Constraints.

If you double click on this entry you should start a text editor. Using the text editor enter the following constraints:

```
set_property PACKAGE_PIN V17 [get_ports {a[0]}]
set_property PACKAGE_PIN V16 [get_ports {a[1]}]
set_property PACKAGE_PIN W16 [get_ports {a[2]}]
set_property PACKAGE_PIN W17 [get_ports {a[3]}]
set_property PACKAGE_PIN W15 [get_ports {b[0]}]
set_property PACKAGE_PIN V15 [get_ports {b[1]}]
set_property PACKAGE_PIN W14 [get_ports {b[2]}]
set_property PACKAGE_PIN W13 [get_ports {b[3]}]
set_property PACKAGE_PIN U16 [get_ports {s[0]}]
set_property PACKAGE_PIN E19 [get_ports {s[1]}]
set_property PACKAGE_PIN U19 [get_ports {s[2]}]
set_property PACKAGE_PIN V19 [get_ports {s[3]}]
set_property PACKAGE_PIN W18 [get_ports {s[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {a b s}]
```

Each line of code essentially connects one of your inputs or outputs to a corresponding pin, which in turn connects your circuit to the peripherals on the board once it is downloaded to the FPGA. The last line sets the I/O standard of the ports. We do not get into the details here so simply include all the variable names in the last line.

Programming the FPGA. Now we are almost at the end, what we have to do is to compile (map) our design on to the FPGA. This process consists of several independent steps (we do not discuss them in detail here), but you can use a shortcut:

Select your circuit (FourBitAdder) in *Project Manager*. From the *Flow Navigator*, select

> *Program and Debug → Generate Bitstream*.

This will generate a binary file, which is the FPGA programming file that you will later download to the FPGA. To generate the programming file, all other steps (i.e., Synthesis and Implementation) have to be completed.

You can track the process on the top right corner, where it will say "Running write_bitstream" with a progress bar. If there are errors or problems try debugging your code or contact an assistant.

Now we have the program file we need to download this information to the FPGA. Make sure the FPGA board is powered on and that it is connected to the computer through the USB cable.

From the *Flow Navigator*, select

> *Program and Debug → Open Hardware Manager → Open Target → Auto Connect*

This will open a Hardware Manager.

Vivado should have automatically detected the FPGA board. You are now ready to download the code to the board.

From the *Flow Navigator*, select

Program and Debug → Hardware Manager → Program Device → xc7a35t_0

A dialog window will popup with two fields: Bitstream file and Debug probes file. The Bitstream file field should automatically contain the path of the previously generated bitstream file. Leave the Debug probes file empty. Simply press Program.

Now you are ready to test your circuit on the board. Use the switches and verify that your circuit functions correctly. Show the working circuit to an assistant. If there are errors, you should find the problem and correct it.

Last Words

We have (hopefully) managed to program the FPGA and have translated our design idea from paper, to Verilog code, and then to a physical working circuit.

It is quite possible that you have encountered errors during this exercise. The entire process consists of many steps, and at each step there are many parts where you could have made an error. From the lowest level the errors could be:

- You have made a mistake in the truth table

- Equations were not derived correctly

- Mistake in your Verilog implementation

- The constraint file could have errors

- Problems with the programming cable (USB port not working)

- The configuration switches on the board were not correct

- Board was not powered

There are so many things that could go wrong; it is almost normal that some of you experience problems during the exercise. You should not feel frustrated. It is a rather complex process involving many steps, and it requires some experience to get it right.

When compared to the capabilities of the FPGA, the circuit we have programmed is very small (less than 1% of its resources would be used). For such a small circuit some of the steps of the program may seem a little bit complex. However, the environment is designed for much larger circuits, and some of its 'quirks' start to make sense once we start making larger circuits.