EECS150: Lab 2, Mapping Circuit Elements to FPGAs

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

1 Time Table

ASSIGNED	Thursday, January 28^{th}
DUE	Week 4: February $9^{th} - 11^{th}$, at beginning of your assigned lab section

2 Objectives

In this lab, you will be working with Verilog HDL (hardware description language) to architect a practical circuit. After designing your circuit, you will debug and verify it using a hardware-based test harness. Finally, after your circuit is working correctly, you will conduct a resource and timing analysis that will show you exactly how your design was actually implemented on the FPGA.

Through conducting these tests, you will gain experience working with non-trivial circuits on real hardware. Additionally, you will learn about design partitioning, the process where primitive gates and flip-flops in an HDL, like Verilog, are mapped down to primitive elements on the FPGA. Lastly, you will learn how to use hardware-based test harnesses, along with various tools, to help you debug your designs.

3 Addendum to the CAD Flow

In the FPGA Editor lab, you learned how a file[1] that contains a circuit description, mapped to a specific FPGA, can be modified and used to configure actual hardware. In this lab, we will be exploring an extension to this CAD flow that, in the end, generates the circuit description file. As in last lab, we will use the .ncd to generate a .bit file which we will then use to verify our design in hardware.

Figure 1 shows the additions to the tool flow that we will be using in this lab. The parts of the flow in the gray box were explored in the FPGA Editor lab. In this exercise, you will be introduced to **Design Entry** (Section 3.1), **Design Partitioning** (Section 3.2) and **Translate/Map/PAR** (Section 3.3). Collectively, Design Partitioning and PAR are known as **Partition-Place-and-Route** or **PPR**. It is in these steps that you will be performing a majority of your analysis.

3.1 Design Entry

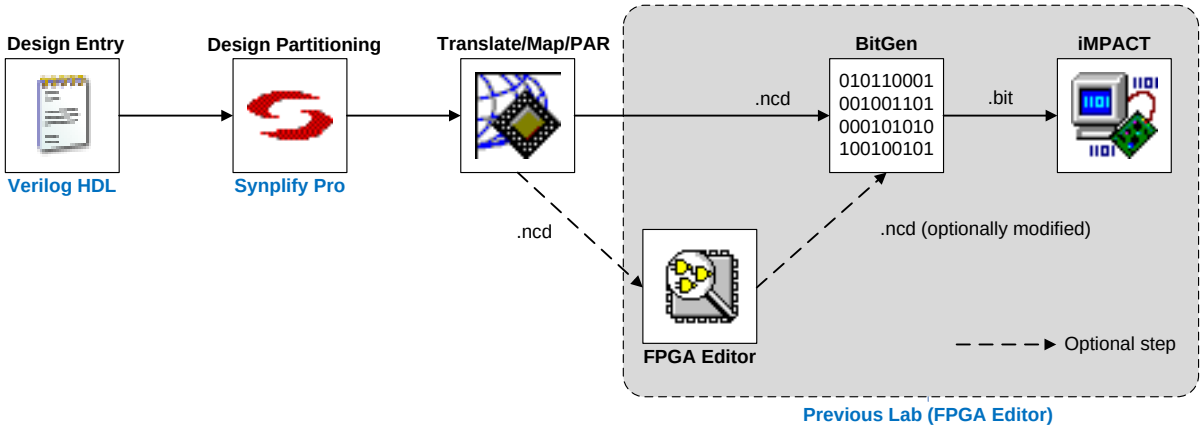
The first step in logic design is to conceptualize your design. Once you have a good idea about the function and structure of your circuit and maybe a few block diagram sketches, you can start the implementation process by specifying your circuit in a more formal manner.

In this class we will use a Hardware Description Language (HDL) called Verilog. HDLs have several advantages over other methods of circuit specification:

1. Ease of editing, since files can be written using any text editor

[1] Specifically, we are referring to the .ncd file that FPGA Editor works with.

Figure 1 Structural Verilog ⟶ .bit file tool flow.



2. Ease of management when dealing with large designs

3. The ability to use a high-level behavioral description of a circuit.

In this class we will default to using basic text editors to write Verilog. Fancier editors are available, and in fact are included with the CAD tools such as Xilinx ISE and ModelSim; however these tools are slow and will often hinder you.

In this lab, you will only be using a small subset of Verilog called Structural Verilog. Specifically, you will be designing down to primitive gates (such as **and** and **or**) and flip-flops.

3.2 Design Partitioning

Design partitioning is the process of transforming the primitive gates and flip-flops that you wrote in Verilog into LUTs and other primitive FPGA elements. For example, if you described a circuit composed of many gates, but ultimately of 6 inputs and 1 output, Design Partitioning will map your circuit down to a single 6LUT. Likewise, if you described a flip-flop, it will be mapped to a specific type of flip-flop which actually exists on the FPGA. The final product of the design partitioning phase is a netlist file, a text file that contains a list of all the instances of primitive components in the translated circuit and a description of how they are connected.

3.3 Translate, Map, Place and Route

From the netlist produced by the design partitioning tools, we must somehow create a file that can be directly translated into a bitstream to configure the FPGA. This is the job of the Translate, Map, and the Place and Route (PAR) tools.

3.3.1 Translate

Translate takes as input a netlist file from the design partitioning tools and outputs a Xilinx database file, which is the same thing as the netlist, reduced to logic elements expressed in terms that Xilinx-specific devices can understand.

3.3.2 Map

Map takes as input the database file which was output from Translate and 'maps' it to a specific Xilinx FPGA. This is necessary because different FPGAs have different architectures, resources, and components.

3.3.3 Placement

Placement takes as input the result of the "Map" step and determines exactly where, physically, on the FPGA each LUT, flip-flop, and logic gate should be placed. For example, a 6LUT implementing the function of a 6-input NAND gate in a netlist could be placed in any of the 69,120 6LUTs in a Xilinx Virtex5 xc5vlx110t FPGA chip. Clever choice of placement will make the subsequent routing easier and result a circuit with less delay.

3.3.4 Routing

Once the components are placed, the proper connections must be made. This step is called routing, because the tools must choose, for each signal, one of the millions of paths to get that signal from its source to its destination.

Because the number of possible paths for a given signal is very large, and there are many signals, this is typically the most time consuming part of implementing a design, aside from specification. Planning your design well and making it compact and efficient will significantly reduce how long this step takes. **Designing your circuit well can cut the time it takes to route from 30 min to 30 sec.**

3.3.5 PPR: Partitioning vs. Place and Route

Unlike design partitioning, which only needs to know a set of primitive components to express its result, placement and routing are dependent upon the specific size and structure of the target FPGA. Because of this the FPGA vendor, Xilinx in our case, usually provides the placement and routing programs, whereas a third party, Synplicity, can often provide more powerful and more general design partitioning tools.

3.4 After PAR: Design Tuning and Device Configuration

The output of the PAR tools is the .ncd file that contains the fully placed and routed design mapped down to specific LUTs and interconnects. At this point, you can run BitGen to produce a .bit file directly, or modify the design further with a program such as FPGA Editor. For explanations on either of these CAD flow steps, feel free to refer to the appropriate Sections in the FPGA Editor lab.

4 Lab Prerequisites

Before reading any datasheets or writing any Verilog, read the entirety of this section. It contains a detailed description of the Verilog constructs that you will be allowed to use as well as of the circuit you will be designing. Additionally, it provides a soft introduction to the Xilinx primitives that your design will need to instantiate. If you have any questions regarding the material in this section, be sure to ask a TA ahead of time.

4.1 Allowed Verilog Constructs

In this lab you will be using only **Structural Verilog**. Specifically, you will be allowed to:

1. Instantiate simple modules.
2. Use the **wire** construct.
3. Instantiate the primitive gates (using any number of inputs):
   (a) **and**
   (b) **or**
   (c) **not**
   (d) **xor**

Additionally, although they are not stricly Structural Verilog constructs, you will be using Verilog **generate** and **parameter** statements.

If any of the constructs mentioned above are unclear or you feel that you need to brush up on Verilog in general, please refer to your text or to the code examples provided in the lab (specifically ALU.v and Mux21.v). With regards to using any source to learn Verilog, remember that you are only allowed to instantiate primitive gates, modules and wires in this lab. **In general, the only allowed syntax for this lab can be found in one or both of ALU.v and Mux21.v.**

4.2 Using Xilinx FPGA Primitives

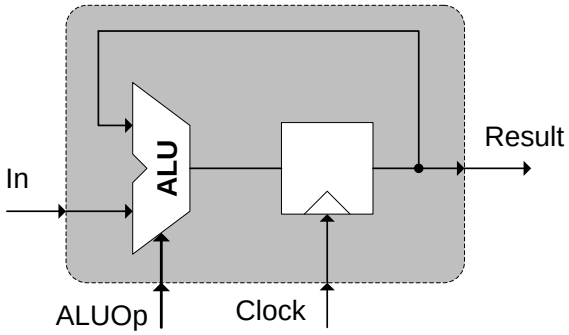
In addition to writing Structural Verilog, you will also instantiate **FPGA Primitives**. An FPGA primitive is either an actual piece of hardware on the FPGA or a wrapper that gets transformed into one. For example, the Virtex5 xc5vlx110t FPGA has flip-flops that can be set synchronously or asynchronously. There are different FPGA primitives, that users can instantiate in their Verilog, for both settings. Both primitives map to the same flip-flop, however. The difference is that depending on what type of primitive you choose, the actual element on the FPGA that it gets mapped to will be configured differently. This is ultimately a convenience, allowing users to select which elements in the FPGA they want to use (and how those elements are configured) during Design Entry.

In this lab, you will be using the Xilinx FDRSE primitive. Specifically, this is a D flip-flop with some additional properties of interest. As a later part of the PreLab (see Part 1b), you will read about the details of this primitive and how to use it in your designs.

4.3 Structural Accumulator

The main part of this lab will be spent building, testing, and analyzing the implementation of an N-bit Accumulator circuit. Regardless of its width, accumulators follow the basic structure shown in Figure 2.

Figure 2 The general structure of an accumulator

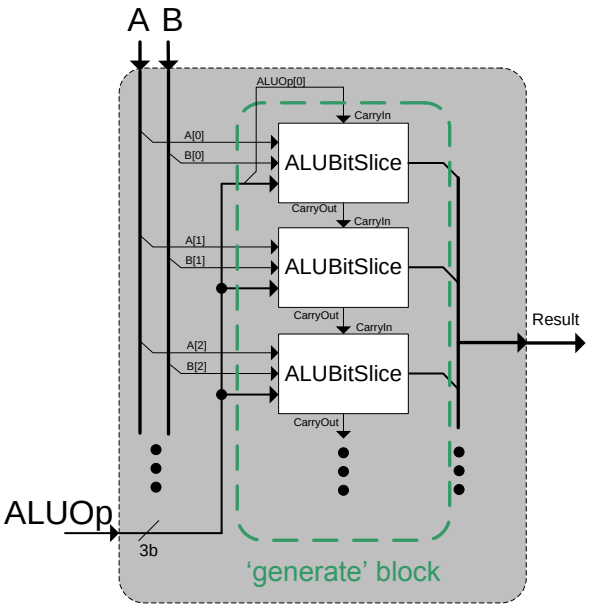


As shown in Figure 2, the Accumulator has a Clock, a standard data input In and an ALUOp[2]. In is a new value from outside the Accumulator which is to be processed by the ALU. ALUOp indicates the operation (**and**, **xor** or + for example) that the ALU will perform. The other input to the ALU is the output of a register which captures the result of the ALU from the cycle before. Thus, when the ALUOp is set to addition, the Accumulator 'accumulates,' hence the name.

We will be implementing our ALU using a bit-slice approach. A bit-sliced ALU is an N-bit ALU implemented as 1-bit ALUs. Bit-sliced ALUs are very efficiently mapped to FPGAs because each 1-bit ALU maps to a small number of FPGA resources. Additionally, by implementing your ALU as a chain of 1-bit ALUs, it will be easy to use Verilog **generate** statements to parameterize your ALU to N-bits (where you can choose the N). An example of using **generate** to build an N-bit ALU is shown pictorially in Figure 3.

[2]Accumulators do not typically have an ALUOp as they typically use a plain Adder circuit. Our Accumulator has an ALUOp because it uses a full-blown ALU.

Figure 3 Extending a bit-slice ALU out to N-bits using **generate**



As you will soon come to appreciate, the ease of changing the ALU's bit-width is extremely useful for performing experiments. Specifically, instead of recoding an ALU of a different width, you can just change a **parameter** that sets the width and the generate statement will take care of rebuilding the circuit automatically.

4.3.1 Coding the Accumulator

You will find a Verilog framework in the /Files directory of this lab. The framework only specifies the interfaces that you must abide by in order to make your Accumulator behave properly with our Tester module, which will be provided for you at the start of lab. **Feel free to modify any and all Verilog we give you. The only restriction on this policy is that you cannot modify the port specifications (interface) of ALU.v or Accumulator.v.**

As part of the PreLab (see Part 2a in Section 5), you will have to implement:

1. ALUBitSlice: A 1-bit ALU.
2. Accumulator: The Accumulator.

You will find the following Verilog written for you (feel free to modify it, but do not modify the port interface on the ALU):

1. Mux21: A 2:1 multiplexer.
2. ALU: The N-bit ALU.

The N-bit ALU will show you exactly how the ALUBitSlice modules are used (as it instantiates them) and how to use a Verilog **generate** statement to parameterize your circuits. You can extend this concept to the Accumulator, which requires the same support, but is not done for you. **Keep in mind, you must still incorperate the flip-flop FDRSE primitive element in your Accumulator. This has not been provided and you are expected to become aquanted with the Xilinx documentation (see PreLab Part 1b) on how to use and instantiate the FDRSE.**

Aside from its interface, which is specified by the framework because it is instantiated in ALU, ALUBitSlice's implementation is entirely up to you. Take note, however, that part of the ALUBitSlice's

interface is exactly what ALUOp encoding it should support (see Table 1). If your ALUBitSlice doesn't follow this ALUOp encoding, it will produce different results when run alongside the Tester.

Table 1 ALUOp operation encoding

Binary Code	Operation
000	Result = A + B
001	Result = A − B
010	Result = A & B (bit-wise **and**)
011	Result = A | B (bit-wise **or**)
100	Result = A ^ B (bit-wise **xor**)
101	Result = ~A (bit-wise **not**)
110	Result = A (passthrough)
111	unused

As you probably notice in Table 1, our ALU supports many more functions than a mere 2:1 mux can choose between. You will have to find a way to join 2:1 muxes together in order to create a mux large enough to support all of the operations shown in Table 1. Feel free to use Mux21 as a starting point. **It has been provided as an example of what Verilog constructs/syntax you are allowed to use in the rest of your design.**

5 PreLab

Please make sure to complete the prelab before you attend your lab section. This week's lab will be very long and frustrating if you do not do the prelab ahead of time.

1. Reading
   (a) Read Sections 3 and 4 and ask questions ahead of time if anything is unclear.
      - Pay particular attention to Section 4.3 as it will tell you what you have to implement for PreLab Part 2a specifically.
   (b) **Read the "FDRSE"** section in the Virtex-5 Libraries Guide for HDL Designs.
2. Verilog and Design
   (a) Write all the Verilog specified in Section 4.3 ahead of time.
      - Since Verilog is nothing more than a bunch of standard text in a file with a *.v extension, you can complete this part in your favorite text editor (we recommend **emacs** in Verilog mode or **Notepad++**).
      - Don't worry about debugging your Verilog. The first main part of the lab constitutes debugging your design.
3. Questions: Answer all questions on the Check-off sheet of this lab packet.

6 Lab Procedure

This section, and those beyond, assumes that you have coded the Structural Accumulator.

Before we analyze our circuit, we have to make sure it works correctly. To debug it properly, we will use several tools. First, we will run **Synplify Pro** (see Figure 1) to check our Verilog for syntax errors. Next, we will use the **Synplify RTL Schematic** to produce a gate and register level diagram of our circuit. This will allow us to inspect our design for obvious errors which are harder to see in Verilog code. Finally, we will actually test our circuit on actual hardware against a test harness that is also built in hardware.

6.1 Circuit Debugging

As our first step, we must resolve trivial typos and remaining bugs in our circuit. To accomplish this, we must first setup a **Xilinx ISE Project** to negotiate with the tools properly.

6.1.1 Project Setup

1. Move all of your Verilog files into C:\Users\cs150−xxx\Lab2Verilog[3].
2. Unzip and move all of **our** Verilog files from the /Framework directory of the lab to the same directory as was listed in the previous step.
3. **Double-Click** the **Xilinx ISE icon** on the desktop to start Xilinx ISE
4. Click **File** → **New Project** and type in a project name (i.e. **Lab2**)
   (a) Set the project location to **C:\Users\cs150-xx** (Xilinx ISE will create a subdirectory for the project).
   (b) The **Top-Level Module Type** should be set to **HDL**.
   (c) Click **Next**.
5. A new dialog will appear with configuration settings
   (a) Device Family: **Virtex5**
   (b) Device: **xc5vlx110t**
   (c) Package: **ff1136**
   (d) Speed Grade: **-1**
   (e) Synthesis Tool: **Synplify Pro**
   (f) Simulator: **ModelSimSE-Verilog**
   (g) Leave the "Enable Enhanced Design Summary," "Enable Message Filtering," and "Display Incremental Messages" checkboxes set to their defaults.
   (h) Click **Next**
6. Skip the **Add New Sources** dialog by clicking **Next**.
7. **Take a moment to review the project settings**. Then click **Finish**.
8. **Right-Click** in the **Sources In Project** box in the upper left corner of Xilinx ISE, and select **Add Source** (not Add Copy of Source).
   (a) Navigate to the **C:\Users\cs150-xxx\Lab2Verilog** folder and select **all of your Verilog files**, then click **Open**.
   (b) Use shift-click and control-click to select multiple files.
   (c) Click **OK**.
9. You should now have a Xilinx ISE project, which looks like Figure 4.

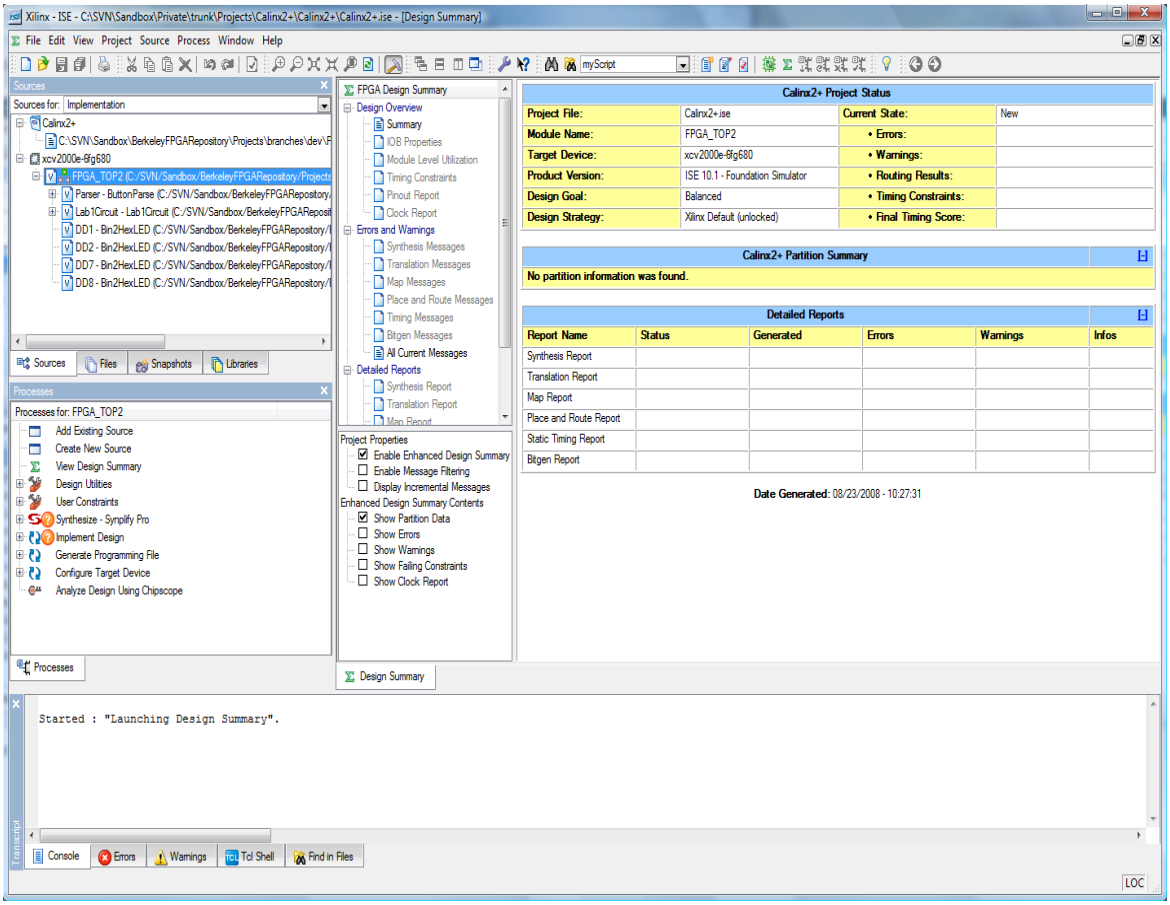
Xilinx ISE, as shown in Figure 4, will allow you to manage your files and invoke the various CAD tools from a central location.

Before continuing, take a moment to orient yourself with the Xilinx ISE IDE. In the upper left is the "**Sources**" box, where you can see all the modules that are part of your project, as well as which modules they depend on (or test) and which files that are in. In the middle left, you can see the "**Processes**" box, which will show all of the tools which can be applied to the currently selected source file.

You might have noticed that several Verilog files you had no part in modifying (the Verilog in the /Framework directory) have appeared in the **Sources** area. These include FPGA_TOP_ML505 and its children: several TestHarness modules. FPGA_TOP_ML505 is (by default) the **top level** Verilog file in

[3]The directory doesn't matter; however, we will use this one for the remainder of this tutorial.

Figure 4 A Complete Project



your design. It contains the assignments for various pins on the chip to wires that you can use in your design. Inside FPGA_TOP_ML505 you will find several TestHarness modules, and only inside them will you find your actual Accumulator or ALU. Working with the TestHarness modules will be the focus of Section 6.1.3.

Before you run any tests on hardware, however, you must reconcile any syntax errors in your Verilog handiwork.

6.1.2 Running Synplify: Syntax Checking and Schematic Generation

This step will acquaint you with **Synplify Pro**, including its syntax checker and ability to generate **schematics from your Verilog**. Beneath the covers, this step will be performing the **Design Partitioning** step (see Section 3.2) in the CAD flow. A fringe benefit of this tool is that it comes with several extremely useful utilities for analyzing and checking your design.

1. Select **Implementation** from the **Sources for:** pull down in the **Sources** box.
2. In **Xilinx ISE** select **FPGA_TOP_ML505** from the **Sources** box.
   - This will cause a long list of implementation steps to appear in the **Processes** box.
3. Constrain the clock frequency.
   (a) **Right-click Synthesize - Synplify Pro** and click **Properties**.
   (b) **Uncheck Auto Constrain** and set **Frequency** to 1.0.

(c) **Click Ok.**

4. **Double-click Synthesize - Synplify Pro.**
   - This will run the **Design Partitioning** tools on your design.
   - If there is an **X** or a **!** next to the **Synthesize - Synplify Pro** step, this means that there has been an error or warning.
   - **To see the errors and warnings from Synplify Pro, double-click the Synthesize - Synplify Pro → View Synthesis Report step.**

Once you find that you have errors in your **Synthesis Report** (you probably will), you must go through the Synthesis Report and fix them. This can be very daunting as the Synthesis Report is quite dense. Program 1 shows an example fragment from a Synthesis Report.

Program 1 Synthesis report example

```
@N: CG364 :"C:\Test.v":21:7:21:10|Synthesizing module Test
@W: CG133 :"C:\Test.v":26:15:26:19|No assignment to IfOut
@W: CL153 :"C:\Test.v":26:15:26:19|*Unassigned bits of IfOut
        have been referenced and are being tied to 0 - simulation
        mismatch possible
@W: CL159 :"C:\Test.v":24:14:24:18|Input Reset is unused
```

Consider the first line. @N denotes the entry type (there is one entry per line and they may be @W **warnings**, @E **errors** or @N **notes**). C:\Test.v denotes the path to the module that is responsible for throwing the entry. 21 (the left-most number) denotes the line number in the parent module. Lastly, the text at the far right offers a brief summary of the entry.

If running Synplify Pro fails, it is because you have @E or error entries in your design. **You must fix, rerun, fix, rerun, ... etc these errors until running Synplify Pro does not throw errors anymore.** Don't worry about @W or @N. You are guaranteed to get them, and they will mostly be benign in the case of this circuit.

After your design has passed the point of having no errors (as far as Synplify Pro is concerned), it is time to look at the schematic of your circuit for visual aided debugging.

1. To view a schematic of the circuit **double-click** on the **Synthesize - Synplify Pro → Launch Tools → View RTL Schematic** step.
   (a) This will launch **Synplify Pro** and automatically open the **RTL Schematic**.
   (b) Navigate through the schematic using the Synplify Navigation Bar (see Figure 5).
      - You can look inside modules using the "Push/Pop" Navigation Bar buttons.

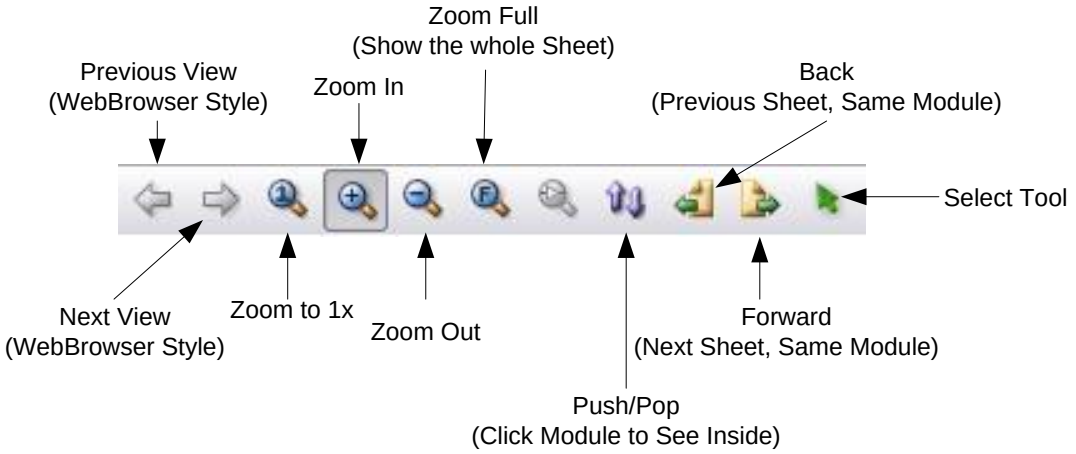
Does your circuit look correct? Remember that the first step in the design entry process is to come up with a design that can be scribbled on a piece of paper. The **RTL Schematic** should have replicated your vision to the gate and register. **Look carefully for wires that you think should be connected in your design but are disconnected or connected incorrectly.** This is often due to misspelling a wire in your circuit[4].

6.1.3 Hardware Verification

If your circuit looks correct, its time to test your circuit on actual hardware using our TestHarness modules. The TestHarness (or more accurately the Tester that is instantiated within the TestHarness, shown in Figure 6) works by comparing the output of your circuit with the output of a working circuit

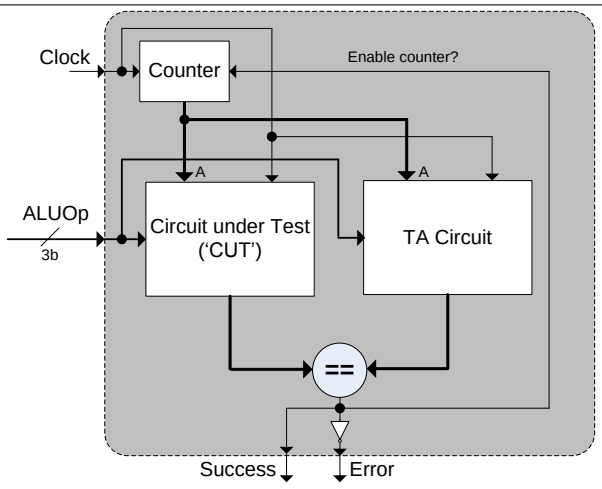
[4] **Synplify Pro will not normally throw an error if your design contains wires that are spelled incorrectly.** It is a part of the Verilog standard to initialize unknown wires as 1-bit wires. This will cause many a bug in your circuits throughout this semester. **Using the RTL Schematic to quickly pinpoint disconnected wires will save you hours over the course of the project.**

Figure 5 The Synplify RTL Navigation Toolbar



that does the same thing. Specifically, there are two TestHarness modules: one for the ALU and one for the Accumulator (this is purely for your convenience, as it is useful to be able to verify the ALU without the Accumulator). Both work in exactly the same way: your circuit, alonside ours as a "**black box**[5]," is fed a stream of inputs from a simple up-counter (a circuit that counts up by '1' every cycle). When the counter reaches its maximum value (its "saturation value") it stops. At this point, if your circuit passes all tests, a success LED turns on. If there was an error, an error LED turns on to indicate failure (and the counter will pause at that point).

Figure 6 The simplified Tester module



By default, the TestHarness circuits offer very little visibility into why your design fails if your design does indeed fail. In order to make the TestHarness modules more useful, you are allowed to modify them to fit your debugging needs. **Specifically, you may add any Structural Verilog to FPGA_TOP_ML505 or to the TestHarness modules.** For example, you can connect LEDs up to

[5]In order to make this assignment realistic we have given you an **EDIF black box** for our implementations of ALUBitSlice and AccumulatorBitSlice, namely **BehavioralALUBitSlice.edf** and **BehavioralAccumulatorBitSlice.edf**. These files can be **easily synthesized**, but are nearly **nearly impossible to read**.

various wires in each TestHarness for added visibility into why your design fails[6].

When you finish making your modifications, it is time to push your Verilog from **Synplify Pro** to the FPGA. In Xilinx ISE, this process is fairly straightforward, but is better broken up into two parts. First, in order to properly synthesize a black box, such as the ALUBehavioralBitSlice.edf file we have given you, you must take a few extra steps before running the tools that will push your design to hardware:

1. Make sure to add the following shell Verilog file(s) to your project:
   (a) ALUBehavioralBitSlice.v
   (b) AccumulatorBehavioralBitSlice.v
2. Set the Macro Search Path
   (a) Make sure **FPGA_TOP_ML505** is highlighted in the **Sources** box.
   (b) **Right-Click** on Implement Design in the **Processes** box.
   (c) Go to the Translate Properties tab.
   (d) Set the Macro Search Path to the **exact** directory where your copy(ies) of the .edf/.ngc files reside.
3. Your project should now be able to build with black box files and core files properly.

Second, and at last: you must invoke all of the tools on your design to bring your Verilog to life:

1. In **Xilinx ISE** make sure **FPGA_TOP_ML505** is still selected in the **Sources** box.
2. **Double-click Synthesize - Synplify Pro** and fix any errors that it reports back through the Synthesis Report.
3. Invoke the Xilinx **Place And Route** tools by **double-clicking** on the **Implement Design** step in the **Processes** box.
   - This will run three sub tools: Translate, Map and PAR.
   - Ignore any warnings from these steps only in this lab. They will often give warnings that can be safely ignored.
4. **Double-click Generate Programming File** (BitGen).
5. **Double-click Configure Target Device** (iMPACT[7]).

In the future, you don't have to explicitly click all of the above tabs in order to run the tools. If you click **Configure Target Device**, all of the tools that it depends on will run in automatically.

As you find bugs in your design and have to make changes, keep in mind the time it takes to generate a schematic versus the time it takes to verify on hardware. It is sometimes, for a design of this size, very easy to find a bug in an **RTL Schematic**. Additionally, you do not need to run all of the tools and then configure the board to see the schematic. Consider these development time tradeoffs when debugging your circuit: *you will save yourself a lot of time!*

6.2 Circuit Analysis

Now that your circuit is functioning the way it is supposed to, we are going to perform a timing and resource analysis on it This will allow us to see what is happening to our Verilog, insofar as it is being mapped to FPGA components, so that we can make more informed design decisions in the future. Proper use of the skills you learn in this exercise will carry you far in digital design, as space and time are major considerations for any design.

[6] For some code examples of how to attach LEDs and DIP switches to your design, refer to the commented sections in FPGA_TOP_ML505.

[7] If you have any problems running iMPACT, refer to the FPGA Editor lab for a refresher.

6.2.1 Resource Analysis

In the PreLab (see Question 1a on the check-off sheet), you were asked to find out how many resources a 1-bit ALU mapped to on the FPGA. We would like to extend this concept to our working N-bit Accumulator.

Specifically, we are interested how much of the following is taken up in a given N-width implementation of our Accumulator:

1. Occupied SLICEs (see check-off Question 2a).
2. SLICE LUTs (see check-off Question 2b).
3. SLICE Registers (used as flip-flops) (see check-off Question 2c).

Based on your thought process developed in the PreLab, you can probably derive a decent guess for these numbers right now. To verify your guess, we will use the **generate** block that we used to implement our Accumulator to change the width of the Accumulator without rewriting any Verilog.

1. In the **Sources** box, **right-click** on Accumulator which is nested underneath FPGA_TOP_ML505 and select **Set as Top Module**.
   - This will tell the tools to only PPR our Accumulator, as opposed to the Tester and any other baggage in FPGA_TOP_ML505.
   - We can now get an accurate resource estimate of *only* the resources taken up by the Accumulator.
2. In Accumulator.v, modify the value assigned to the **parameter** called Width and rerun the tools to find out how many resources your Accumulator takes up.

In order to test your resource consumption theory, you will tweak Width in Step 2, above, until you can come up with a generalized formula for determining resource consumption.

Of course, the last step of this story is how to actually use the tools to find resource consumption. This is very simple; specifically:

1. **Double-click Implement Design** step in the **Processes For Source** box.
2. After the **Map** process completes (i.e. has a green check next to its name), **double-click** on **View Design Summary** in the **Processes For Source** box.
3. Inspect the **Design Summary** for resource totals.

Before proceeding to Section 6.2.2, answer the check-off Questions 2a, 2b and 2c based on your observations and experiments.

6.2.2 Timing Analysis

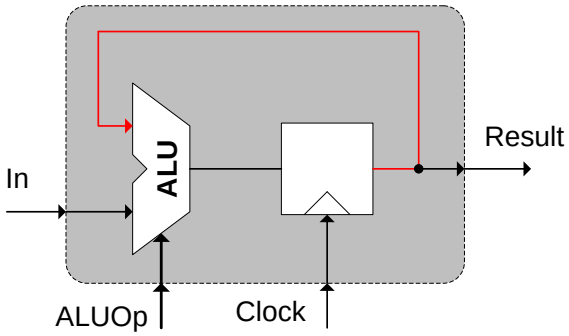
The second experiment that we would like to run analyzes the delay on the wire output from the FDRSE to the input of an ALU. This corresponds to the red path in Figure 7.

Of all of the paths in the Accumulator, this one (along with the delay through the ALU and the delay from the output of the ALU to the input of the FDRSE) is the most interesting because it is likely to be your critical path. We'll leave out the delay through the ALU and the delay from the output of the ALU to the input of the FDRSE for now, as the ALU is implemented in terms of LUTs (which we know the delay through) and connections from LUTs to flip-flops, both in the same slice, are statically routed. The only part of the path with varying delay is the path from the output of the FDRSE to the input of the ALU, which is exactly the path highlighted in Figure 7.

Before we proceed, we have to set FPGA_TOP_ML505 as our 'top' module again[8]. Remember, we only made Accumulator our top module in Section 6.2.1 because we wanted to see only its resources get

[8]For instructions on how to specify a 'top' module, refer to Step 1 in Section 6.2.1. The **Technology Schematic** sits right alonside the **RTL Schematic** in the Synplify tree.

Figure 7 The path in the Accumulator whose delay we would like to analyze



totalled by the tools. If we are going to perform any useful timing analysis, we need to see Accumulator in the context of a larger design. For the sake of this experiment, we'll use the Accumulator that was instantiated within the Tester module.

In order to compute delay, you first have to find the net that you are wanting to compute the delay for. This can be done in FPGA Editor, but is sometimes non-trivial because FPGA Editor doesn't present an abstract view of your circuit. As such, we will use the **Synplify Pro Technology Schematic**[9]. The **Technology Schematic** is a sister to the **RTL Schematic** that shows you a graphic view of your design after Partitioning: namely, at the level of LUTs and flip-flops. Nets (or wires) that you will find in the Technology Schematic will **have the same names** in FPGA Editor. Furthermore, SLICEs that drive a single net are typically named after that net. Thus, the **Technology Schematic** will allow you to easily find a net or SLICE of interest in FPGA Editor.

After you find the requested net (see Figure 7), open FPGA Editor using the PPR'ed .ncd file.

1. **Double-click Implement Design** step in the **Processes For Source** box.
2. After the **Place and Route** process completes, open FPGA Editor by **double-clicking View/Edit Routed Design (FPGA Editor)**, which can be found under **Implement Design** $\longrightarrow$ **Place and Route**.
3. Once in FPGA Editor, find the net that you located by name in the **Technology Schematic**.
4. How was this net routed in your design? In other words, how does it get from the output of the FDRSE to the input of the ALU? Answer this question on the check-off sheet (Question 3a).
5. **Left-click** on this net and click on the **delay** button at the far right of the screen on the **Button bar**.
   - The **Console Output** window will show you the delay from the net's **driver** to everywhere else in your design that the net is connected.
   - Connections will only tell you which SLICEs are connected to your net. As such, you will have to look at each SLICE (they will probably have useful names) to determine which one implements the 1-bit ALU and FDRSE that you are interested in.
6. Find the delay on the net shown in Figure 7 and mark this delay down on the check-off sheet for Question 3b.

Now, you built an N-bit accumulator. We have just seen the delay on a single 1-bit wire that feeds back from an FDRSE to the input of the ALU. What about the other wires? Will they share similar delay or differing amounts of delay? Based on your answer for Question 3a and from direct inspection, answer this question on the check-off sheet (Question 3b).

[9] To open the **Technology Schematic**, follow the instructions for opening the **RTL Schematic** (Step 1 in Section 6.1.2).

7 Lab 2 Checkoff

ASSIGNED	Thursday, January 28^{th}
DUE	Week 4: February $9^{th} - 11^{th}$, at beginning of your assigned lab section

Man Hours Spent	**Total Points**	**TA's Initial**	**Date**	**Time**
	/100		/ /	
Name	**SID**	**Section**		

1. PreLab ..... _____ (30%)

   (a) What FPGA resources (be precise) does a single ALUBitSlice instance map to on the Virtex5 xc5vlx110t FPGA?

   (b) What happens to the state of the FDRSE when its R and S inputs are both high?

   (c) In Question 1b, when do the values on the R and S lines actually matter? Any time? At the rising edge of the clock? Explain why, based on the FDRSE's description.

   (d) Imagine an Accumulator such as the one in Figure 2 without the flip-flop at the output. In other words, the output feeds **directly** into the second input of the ALU. Does this circuit make sense? Explain is behavior.

2. Resource Analysis ..... _____ (35%)

   (a) # of occupied SLICEs as a function of Width

   (b) # of SLICE LUTs as a function of Width

   (c) # of SLICE registers (used as flip-flops) as a function of Width

3. Timing Analysis ..... _____ (35%)

(a) How was the net from the output of the FDRSE to the input of the ALU routed?

(b) Delay (in ns) from the output of the FDRSE to the input of a LUT that implements an ALUBitSlice

---

(c) Is there significant difference in the delay between the different wires that make up the bus in an N-wide implementation? Why?

Rev.	Name	Date	Description
C	Brandon Myers	1/26/2010	Moved to Spring 2010.
B	John Wawrzynek & Chen Sun & Ilia Lebedev & Chris Fletcher	1/30/2009	Proofread and fixed various issues
A	Chris Fletcher & John Wawrzynek	1/22/2009	Wrote new Lab; Designed to replace old "Design with Verilog" and "Intro to CAD Tools" labs (some sections of pre-PAR tool flow and project setup taken from "Intro to CAD Tools" lab). Synplify Pro "Synthesis Report" (Section 6.1.2) material integrated from Chen Sun's Lab Lecture slides from Fall 2008.