Laboratory Exercise #6
Introduction to Logic Simulation and Verilog

ECEN 248: Introduction to Digital Design

Department of Electrical and Computer Engineering
Texas A&M University

1 Introduction

In the previous labs, we tested and debugged our digital designs by bread-boarding the actual circuits using ICs. For small designs, this method proved to be quite effective, and in the late 1960's and early 1970's, this was standard practice. However, as the gate counts of our circuits increased, this method became far less attractive. Thankfully, with advances in computer technology, there are alternatives to bread-boarding circuits. In this day and age, we can effectively simulate the operation of our digital circuits during the design process without being burdened with the tedium you all have experienced up to this point. In lab this week, we will demonstrate the concept of digital circuit simulation and, in doing so, introduce Verilog® HDL, an *Institute of Electrical and Electronics Engineers* (IEEE) standard Hardware Description Language (HDL).

2 Background

The following subsections will provide you with some of the background necessary to understand and appreciate the way modern digital design is carried out. For a more in-depth treatment, please consult the course text book.

2.1 Verilog HDL

Programming languages have proven to be effective for software development. Interestingly, a similar truth holds for hardware development. Just as programming languages boost developer productivity, Hardware Description Languages (HDLs) do so through various levels of abstraction, while providing a means to succinctly describe digital circuits. What differentiates a traditional programming language from a standard HDL is the manner in which operations described within the language are handled. In a tradition programming language such as C, operations happen sequentially, and conditional statements facilitate changes in program flow. In contrast, the operations described in an HDL happen concurrently, and signals (often called nets) allow concurrent operations to interact with one another.

In the 1980's, two competing HDLs were developed, namely VHDL and Verilog. Today, both languages co-exist, encouraging tool vendors to support VHDL and Verilog simultaneously within a single design. For our work in this lab, we will concentrate on Verilog; however, it is important to note that many of the HDL principals demonstrated here apply to VHDL as well. Verilog offers various levels of abstraction for describing digital circuits with structural at the bottom and behavioral at the top. Any given design could have a mixture of these abstraction levels, but in this lab assignment, we will start at the lowest level and move up. The example shown below uses structural Verilog and the gate-level primitives built into Verilog to describe the 1-bit wide, 2:1 multiplexer (MUX) you designed in the previous lab. The corresponding

gate-level schematic in Figure 1 labels the module ports, internal wires, and gate instance names according to the provided Verilog code.

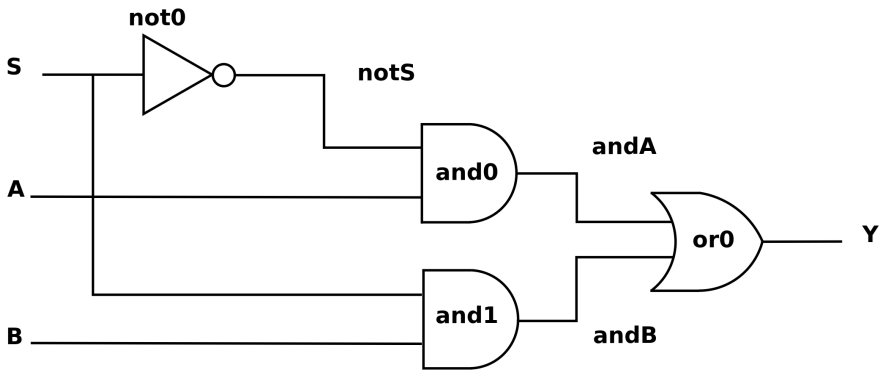


Figure 1: Gate-level Schematic of a 1-bit wide, 2:1 MUX

```
1 `timescale 1ns / 1ps
  `default_nettype none
3 /*This module describes a 1-bit wide multiplexer using structural constructs *
   *and gate-level primitives built into Verilog.                              */
5
  module two_one_mux(Y, A, B, S); //define the module name and its interface
7
      /*declare output and input ports*/
9     output wire Y; //declare output of type wire
      input wire A, B, S; //declare inputs of type wire
11
      /*declare internal nets*/
13    wire notS; //inverse of S
      wire andA; //output of and0 gate
15    wire andB; //output of and1 gate

17    /*instantiate gate-level modules*/
      not not0(notS, S);
19    and and0(andA, notS, A);
      and and1(andB, S, B);
21    or or0(Y, andA, andB);

23 endmodule //designate end of module
```

Syntactically, Verilog is very similar to the C programming language. Comments are designated in a similar fashion (i.e. * */ for comments which span multiple lines or \\ for single line comments), and semicolons end declarations, assignments, and statements. Verilog is case-sensitive and all keywords are lowercase. The keyword, **module**, marks the beginning of the Verilog module, while the keyword,

endmodule, marks the end. In the example above, the module name is *two_one_mux*, and its ports are Y, A, B, and S. The port order is not enforced by the language but follows a common convention in which the output ports are first, followed by the input ports. The keyword, **output**, is used to declare a net as an output port, while **input** declares a net as an input port. Nets can be of type **wire** or **reg**. For now, we will only be concerned with wires. Nets can take on one of four values, namely '1', '0', 'Z', and 'X' for high, low, floating, and undefined, respectively. Modules, whether built-in or user-defined, are instantiated as seen in lines 14 through 19 of the code above. The module name is followed by the instance name. The instance name must obviously be unique; however, multiple instances of the same module may exist. Notice the order of ports for the built-in primitives (i.e. **not**, **and**, **or**) follows the convention discussed above such that the output port is first, followed by the input ports. Since these operations are associative, the order of the inputs is not important.

2.2 Xilinx Vivado

Xilinx® Vivado Design Suite (Vivado™) version 15.2 is the hardware development tool that we will be using in this laboratory. Xilinx manufacturers a wide variety of reprogrammable logic devices (we will discuss these further in the next lab) and supplies Vivado as a means to develop for these devices. Consequently, Vivado is an excellent teaching tool for modern digital design. It supports digital circuit simulation, which is the focus of this week's lab assignment, in addition to all of the implementation processes necessary to load a design into an Xilinx part. It comes with an easy-to-use Graphical User Interface (GUI) and waveform viewer and supports both VHDL and Verilog. Xilinx Vivado is part of a larger class of computer-aided design (CAD) tools used to develop digital circuits. For additional information on Xilinx and Vivado 15.2, consult the manufacturer's website at: `www.xilinx.com`

2.3 Linux OS

Linux is an open-source Unix-like operating system (OS) commonly used in industry. The open-source license allows it to be used free of charge, which makes it a big win for the university setting. The Unix aspect allows Linux to be compatible with many of the CAD tools out on the market today. The Linux kernel along with the various support packages makeup the specific Linux distribution. The machines in the ECEN248 lab have a variant of Red Hat Linux called CentOS installed on them because many of the tool vendors will provide support for only Red Hat Linux.

The CentOS Linux installation has a window manager, which gives you the look and feel similar to that of MS Windows or MacOS. However, the Linux terminal is available for those acquainted with Unix. As a result, you will be provided with commands to execute in the terminal throughout the lab session. Please try to understand what these commands do so that you can execute them in future labs without instruction. The format of the commands provided is shown below. The command to be executed follows the > symbol.

```
>command argument0 argument1
```

Note: Please ensure that you have a Unix account with the ECE department before arriving to your lab session. It is imperative that you are able to log into the CentOS machines in ECEN 248 lab

3 Pre-lab

We will reuse the designs created in Lab 3 and Lab 4 for this week's lab so no pre-lab submission is required. However, you will be expected to have the appropriate material ready for lab so please read over the entire lab manual before coming to your lab session.

4 Lab Procedure

The lab experiments below will guide you through the development of a simple 4-bit ALU in Verilog using the Vivado Design suite. Each module that you create will be individually tested using the built-in digital circuit simulator, Vivado Simulator, with test bench files that we provide to you.

4.1 Experiment 1

The objective of this lab experiment is to familiarize you with the Vivado development environment. Please take note of what it is you are doing as you will be expected to perform similar exercises on your own in future experiments.

1. Launch Vivado and create a new design project.

   (a) Open a terminal window in the CentOS workstation by selecting **Applications** → **Accessories** → **Terminal** and type the following command and hit 'Enter' to create a work directory:

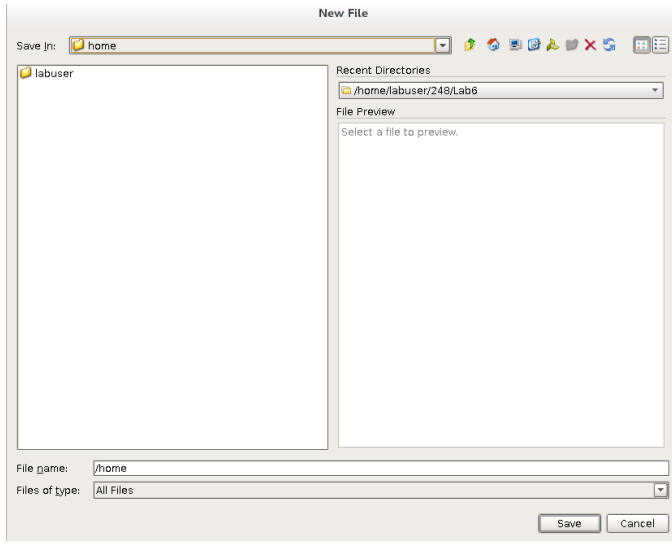
   ```
   >mkdir $HOME/ecen248
   ```

   (b) Now execute the following commands to run Vivado:

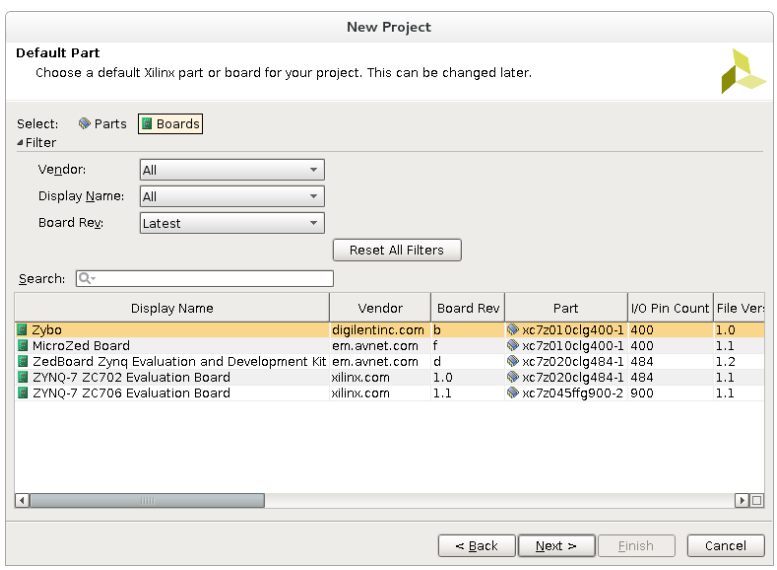
   ```
   >source /softwares/Linux/xilinx/Vivado/2015.2/settings64.sh
   >vivado
   ```

   The first sets up the environment in order to run Vivado and the second command starts the Vivado Suite

   (c) Once Vivado loads, select **File** → **New Project**
   The New Project Wizard will open. Click 'Next' and a window as illustrated below will open.

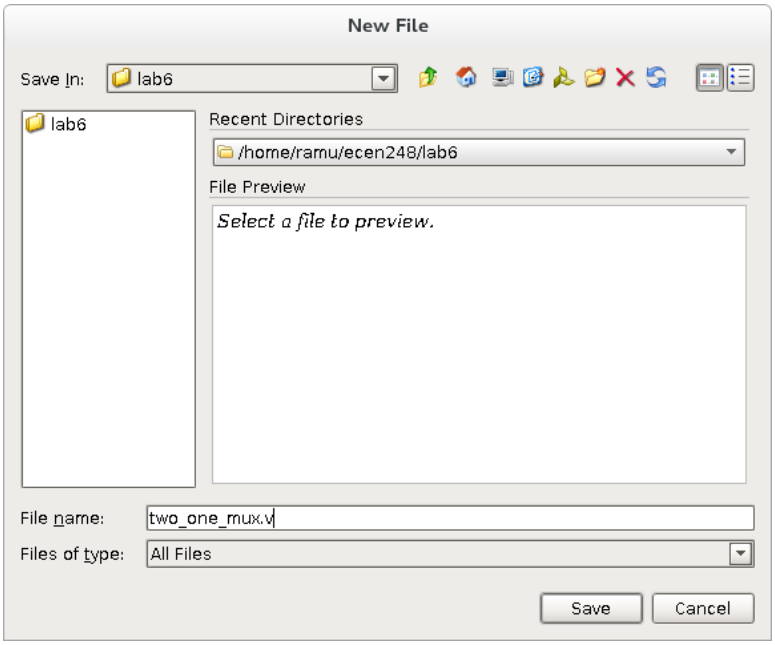


Type a Project Name (ex.lab6) and a Project Location (ex. /ecen248/lab6 in your home directory). Click 'Next' and select 'RTL Project'. Click 'Next' and the 'Add Sources' window will appear. Select Target language as 'Verilog' and simulator language as 'Verilog'. Click 'Next' until you reach the 'Default Part' window. Select 'Board' and select 'Zybo' as shown below. Click 'Next' and select 'Finish' to create new project.



2. Create a Verilog source file which describes the 2:1 MUX in Figure 1.

   (a) From within Vivado, select **File** → **New File** to create a new file.

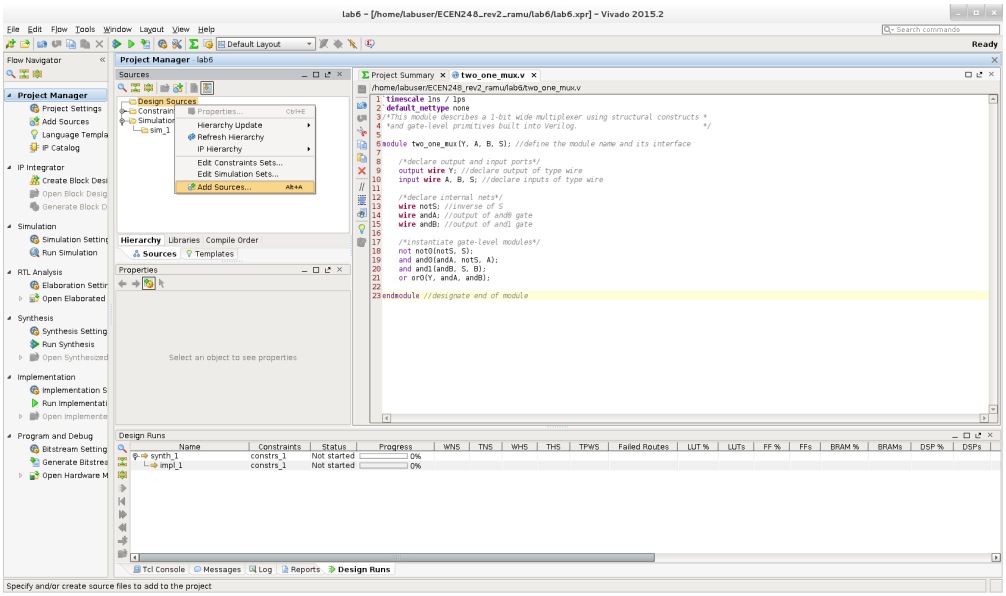


(b) A window will appear as seen above. Type the name as 'two_one_mux.v' and save it your lab 6 directory. You will see the text file opened in Vivado.

(c) Type the Verilog code example found the in the Background section of this manual into the Vivado text editor window.
Note: Do **NOT** copy and paste text from the PDF into the editor window. Symbols within the code do not always copy properly and will cause syntax errors when you attempt to build the project.

3. Add the 2:1 MUX source file and test bench to the Vivado project.

(a) Right-click on the Design Sources within the Hierarchy window under Sources and select 'Add Sources..' as shown in below.



(b) In the Add Sources window select 'Add or create design sources' and click 'Next'.

(c) Click on the green '+' button and select 'Add files'. Navigate to the source file you just created and hit 'OK.' Click 'Finish' to add the source file to the project.

(d) Ensure the source file you just added appears under the Design Sources in the Hierarchy window.

(e) Open a new terminal window and copy the appropriate test bench file from the ECEN248 directory into you lab6 directory with the following commands:

```
>cd $HOME/ecen248/lab6
>cp /homes/faculty/shared/ECEN248/two_one_mux_tb.v .
```

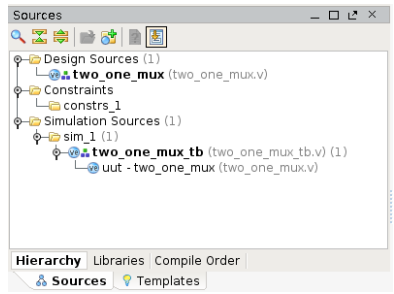
The first command changes the current directory to your lab6 directory, while the second command copies the 2:1 MUX test bench into the current directory. You could also perform this operation in a single command by replacing the '.' at the end of the copy command with the path to your lab6 directory. '.' represents current directory.

(f) Right-click on the Simulation Sources within the Hierarchy window under Sources and select 'Add Sources...' as shown earlier.

(g) In the Add Sources window select 'Add or create simulation sources' and click 'Next'. Add the test bench and click Finish.

(h) Add the 'two_one_mux_tb.v' file to your Vivado project, leaving the source file properties as default again.

(i) Ensure the hierarchy appears as seen in below.



(j) Open the 2:1 MUX test bench file (two_one_mux_tb.v) and take a moment to examine the contents. The test bench utilizes advanced concepts in Verilog that we have not discussed yet, but some things should look familiar. A test bench file is nothing more than a Verilog module which instantiates the Unit Under Test (UUT) and stimulates the inputs for testing. Notice that the test bench module does not have any ports of its own. The input and output ports of the UUT are the focus, and when we simulate the test bench file, we will examine those signals.

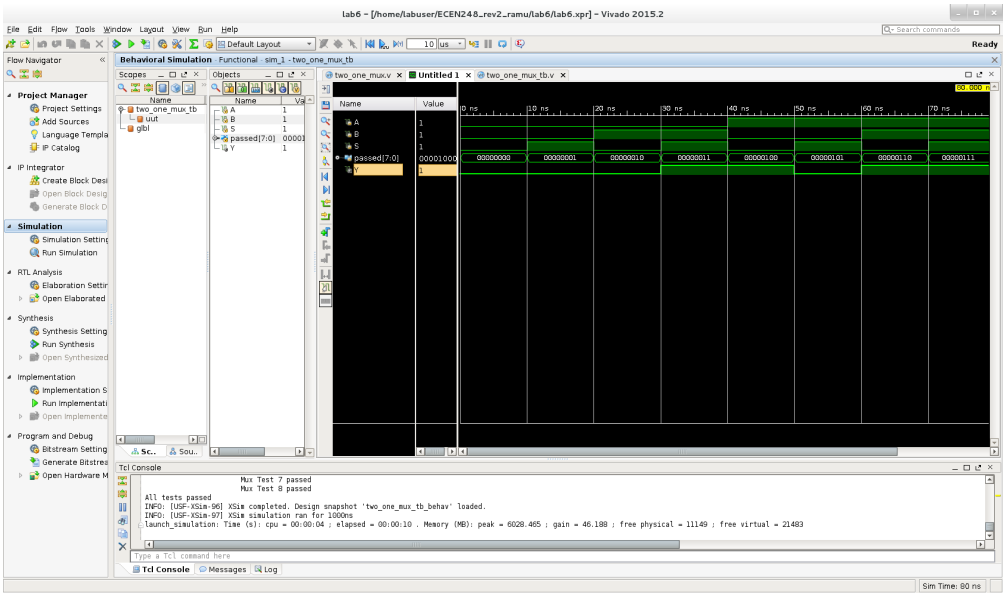
4. Simulate the 2:1 MUX test bench.

(a) To simulate a file using a test bench, right click on the test bench file and select 'Set as Top'. Here right click on 'two_one_mux_tb.v' and make it top.

(b) Select the 2:1 MUX test bench file in the hierarchy window and then click 'Run Simulation' from within the Flow Navigator panel. Select 'Run Behavioural Simulation'.

(c) If there are any errors or warnings, they will show up in the Console panel. Correct any errors and warnings at this time. You may re-run the simulation process once you have fixed your source code.

(d) The waveform panel shown below will open once your design successfully compiles. Take note of the waveform panel in the top-right corner and the console panel at the bottom. Notice that the test bench exercises the multiplexer through all of the possible input combinations. Ensure that your design passes all of these tests. Please include a screenshot of the waveform in you lab write-up along with the console output of the test bench.



5 Experiment 2

The purpose of this experiment is to design the modules necessary to build our simple 4-bit ALU, while introducing a level of abstraction available in Verilog. Each component will be tested using Vivado with the provided test benches. You may need to look back at Experiment 1 if you forgot how to perform a procedure listed below in Vivado.

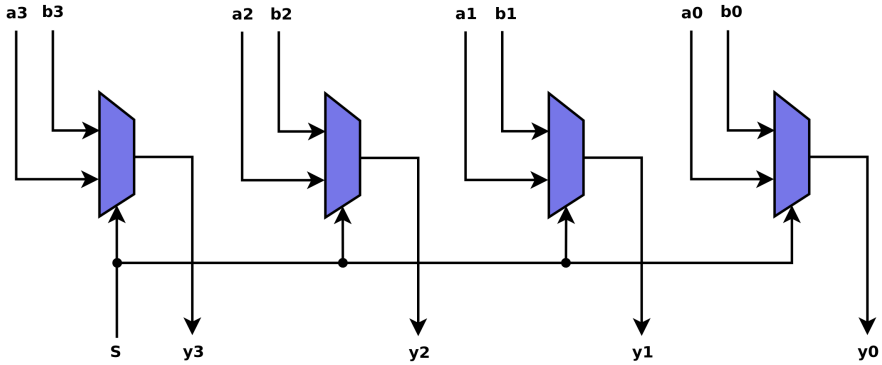


Figure 2: 4-bit wide, 2:1 MUX

1. Figure 2 illustrates how to connect four 1-bit, 2:1 multiplexers together to create a single 4-bit, 2:1

multiplexer. This diagram should reflect the bread board circuit from the previous lab.

(a) Using the aforementioned diagram and the code below as a starting point, describe a 4-bit, 2:1 multiplexer, which uses the module you created in Experiment 1.

```
`timescale 1ns / 1ps
`default_nettype none
/*This module connects four 1-bit, 2:1 MUXs together to    *
 *create a single 4-bit, 2:1 MUX                            */

module four_bit_mux(Y, A, B, S);

    /*declare output and input ports*/
    //output is a 4-bit wide wire
    input wire [3:0] A, B; //A and B are 4-bit wide wires
    input wire S;//select is still 1 bit wide

    /*instantiate user-defined modules*/
    two_one_mux MUX0(Y[0], A[0], B[0], S);
    two_one_mux MUX1(Y[1], A[1], B[1], S);
    //you need two more module instantiations here...

endmodule
```

(b) Add the file you just created to your current Vivado project.

(c) Copy the appropriate test bench file, 'four_bit_mux_tb.v', into your lab6 directory. As a refresher, the commands to do so may be found below.

```
>cd $HOME/ecen248/lab6
>cp /homes/faculty/shared/ECEN248/four_bit_mux_tb.v .
```

Note: If you use the same terminal that you used to do the last copy, you do not need to execute the first command!

(d) Add the test bench that you just copied over to your Vivado project and simulate the logic behavior as you did in the previous experiment. To simulate four_bit_mux_tb.v, right click on the file and select 'Set as Top'. Please include a screenshot of the waveform in you lab write-up along with the console output of the test bench.

2. In the previous steps, we used structural Verilog to describe a 1-bit, 2:1 MUX. Then we used structural Verilog again to create a 4-bit, 2:1 MUX from our 1-bit MUX module. For the addition/subtraction unit, we will start with a slightly higher level of abstraction available in Verilog, commonly referred to as dataflow. The following steps will guide you through the process.

(a) The **assign** statement in Verilog allows us to describe how data should flow from one wire to the next using arithmetic and logic operators available in most programming languages. The

operand and result wires can be an arbitrary width; however, the width of each wire should match the operation being performed. For example, if you are describing a bit-wise AND operation on two 4-bit wires (**assign** Result = A & B;), the result wire should be 4-bits in width as well. Using the gate-level schematic for a full-adder illustrated in Figure 3 and the code snippet below as a template, describe a full-adder in Verilog using dataflow abstraction.
Hint: The full-adder schematic provides suggested intermediate signal names.

```
`timescale 1ns / 1ps
`default_nettype none
/*This module describes the gate-level model of  *
 *a full-adder in Verilog                         */

module full_adder(S, Cout, A, B, Cin);

    /*declare output and input ports*/
    //1-bit wires
    input wire A, B, Cin;//1-bit wires

    /*delcare internal nets*/
    wire andBCin, andACin;//1-bit wires (missing something???)

    /*use dataflow to describe the gate-level activity*/
    assign S = A ^ B ^ Cin; //the hat (^) is for XOR
    assign andAB = A & B;//the ampersand (&) is for and
    //fill in code for andBC, andAC
    assign Cout = andAB | andBCin; //pipe (|) is for or
    //oh btw, the above line is missing something...

endmodule
```

(b) Copy over the full-adder test bench file, 'full_adder_tb.v', from the course directory and add it to your current Vivado project. Simulate the full-adder design and ensure it works properly. Include a screenshot of waveform in your lab write-up along with the console output from the test bench.

(c) Now that we understand the basics of structural and dataflow Verilog, we can use a mixture of the two abstraction levels to create the addition/subtraction unit found in Figure 4. As with the previous procedures, use the code below as a starting point.

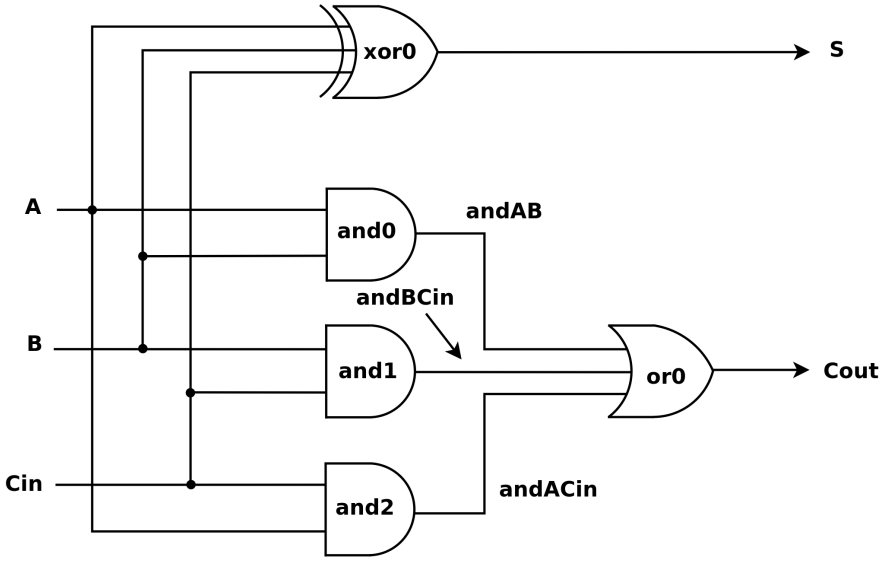


Figure 3: Full-Adder Gate-level Schematic

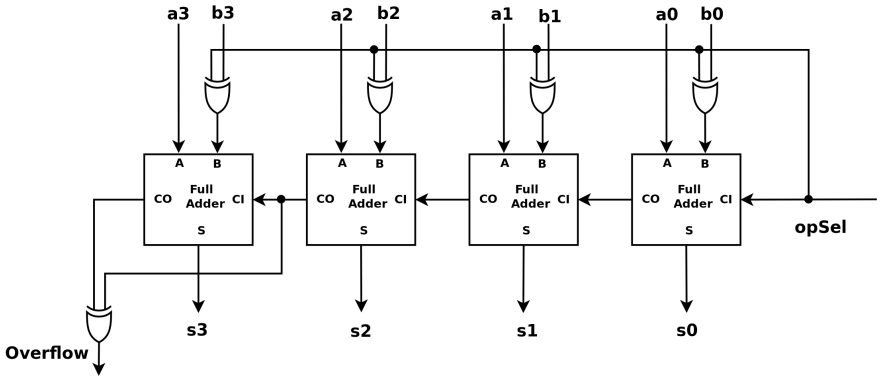


Figure 4: Addition/Subtraction Unit

```
1 ‘timescale 1ns / 1ps
  ‘default_nettype none
3 /*This Verilog module describes a 4-bit addition/subtraction *
   *unit using full-adder modules which have already been      *
5  *designed and tested.                                       */

7 module add_sub(
      /*declare output and input ports*/
9     output wire [3:0] Sum, //4-bit result
```

```
        output wire Overflow, //1-bit wire for overflow
        input wire [3:0] opA, opB, //4-bit operands
        input wire opSel//opSel = 1 for subtract
); //in Verilog, we can describe a module interface in this manner as well!

    /*declare internal nets*/
    wire [3:0] notB;
    wire c0, c1, c2, c3;

    /*create complement logic*/
    assign notB[0] = opB[0] ^ opSel;//if opSel == 1, complement
    //fill in the rest...

    /*wire up full adders to create a ripple carry adder*/
    full_adder adder0(Sum[0], c0, opA[0], notB[0], opSel);
    //something goes here...
    full_adder adder3(Sum[3], c3, opA[3], notB[3], c2);

    /*overflow detection logic*/
    assign Overflow = ;//finish this line

endmodule
```

(d) Copy 'add_sub_tb.v' from the course directory into your lab6 directory and simulate the test bench. As with the previous simulations, include the waveform and console output of the test bench simulation in your lab writeup.

6 Experiment 3

For this experiment, you will use your new found Verilog skills to create the simple 4-bit ALU described in the previous lab.

1. Using your corrected block diagram of the 4-bit ALU from the previous lab submission, create a top-level module with the following interface:

```
module four_bit_alu(
  output wire [3:0] Result,//4-bit output
  output wire Overflow, //1-bit signal for overflow
  input wire [3:0] opA, opB,//4-bit operands
  /*ctrl | operation*
   * 00  |    AND   *
   * 01  |    ADD   *
   * 10  |    AND   *
   * 11  |    SUB   */
  input wire [1:0] ctrl //2-bit operation select
);
```

Hint: Your code should instantiate the 4-bit, 2:1 MUX and the addition/subtraction unit which you have already designed.

2. Copy the 4-bit ALU test bench, 'four_bit_alu_tb.v', file from the course directory and add it to your current Vivado project.

3. Simulate the test bench and add the appropriate material to your lab submission. Demonstrate your progress to the TA once your circuit simulates properly without errors or warnings.

4. **(Honors)** In the last lab, we asked that you add a signal to your ALU which is asserted HIGH when performing a subtraction and $A \geq B$. Using the module interface below and a guide, modify your ALU to include this wire. Read 'four_bit_alu_tb.v' carefully and try to understand each line. Modify 'four_bit_alu_tb.v' to test the new added wire in your ALU and simulate your modified ALU with the modified testbench.

```
module four_bit_alu_honors(
  output wire [3:0] Result, //4-bit output
  output wire Gteq,
  output wire Overflow, //1-bit wire for overflow
  input wire [3:0] opA, opB, //4-bit operands
  /* ctrl | operation*
   * 00  |    AND    *
   * 01  |    ADD    *
   * 10  |    AND    *
   * 11  |    SUB    */
  input wire [1:0] ctrl //2-bit operation select
);
```

7 Post-lab Deliverables

Please include the following items in your post-lab write-up in addition to the deliverables mentioned in the *Policies and Procedures* document.

1. Include the source code with comments for **all** modules you simulated. You do **not** have to include test bench code. Code without comments will not be accepted!

2. Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.

3. Examine the 1-bit, 2:1 MUX test bench code. Attempt to understand what is going on in the code. The test bench is written using behavior Verilog, which will read much like a programming language. Explain briefly what it is the test bench is doing.

4. Examine the 4-bit, 2:1 MUX test bench code. Are all of the possible input cases being tested? Why or why not?

5. In this lab, we approached circuit design in a different way compared to previous labs. Compare and contrast bread-boarding techniques with circuit simulation. Discuss the advantages and disadvantages of both. Which do you prefer? Similarly, provide some insight as to why HDLs might be preferred over schematics for circuit representation. Are there any disadvantages to describing a circuit using an HDL compared to a schematic? Again, which would you prefer.

6. Two different levels of abstraction were introduced in this lab, namely structural and dataflow. Provide a comparison of these approaches. When might you use one over the other?

8 Important Student Feedback

The last part of the lab requests your feedback. We are continually trying to improve the laboratory exercises to enhance your learning experience, and we are unable to do so without your feedback. Please include the following post-lab deliverables in your lab write-up.

Note: If you have any other comments regarding the lab that you wish to bring to your instructor's attention, please feel free to include them as well.

1. What did you like most about the lab assignment and why? What did you like least about it and why?

2. Were there any sections of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?

3. What suggestions do you have to improve the overall lab assignment?