Lab 5

Intro to Verilog

Updated: February 11, 2020
Version: SystemVerilog 2017

5.1 Introduction

Previously, you built digital circuits using hardware devices. This becomes cumbersome very quickly. It would be possible to draw circuit schematics in software then "program" them onto an device. This is easier/faster than physically wiring by hand and allows for much larger, more complicated designs, but still requires a significant amount of click-and-place. The next step is to *describe* digital circuits with code, or a hardware description language (HDL). You should keep in mind that we are still generating a hardware implementation of the logic circuit described by the code, and as such, there are certain characteristics of a HDL that are different than standard programming languages (like C++, Java, Matlab, Python, etc.).

There are many HDLs, most of which are special purpose ones for particular part families. The two most common languages are Verilog, which has 2/3 of the business market and is similar to C, and VHDL, which has 2/3 of the academic market and is similar to Ada. Since Verilog is easier to learn and more industry oriented, we will use it.

5.2 Objectives

After completing this lab, you should be able to:

- Organize files and adhere to a given folder structure
- Know basic Verilog syntax and use it to create simple components/systems
- Explain the difference between gate-level, functional/behavioral, and structural coding styles
- Create a full adder using both gate-level and functional/behavioral styles
- Combine full adder components to create a 2-bit and 4-bit adder/subtractor (structural)
- Write test benches to verify your designs

5.3 Background

See the Introduction of the "SystemVerilog Tutorial" reference document.

Transistor Count of CPUs

To provide some historical context, Table 5.1 shows the steady increase in the complexity of devices over the last several decades.

Source: https://en.wikipedia.org/wiki/Transistor_count

5.4 Pre-Lab

1. Read Part 1 of the SystemVerilog Tutorial
2. Watch Vivado Simulation Tutorial and answer the following questions in preparation for the quiz (you don't need to submit your answers):
   (a) When do you specify the hardware (FPGA/board) that you want to use for a project?
   (b) Which of the following is NOT an option for how a file is used?
   (c) What is the purpose of a test bench?
   (d) How are simulation results viewed in Vivado?
3. Complete the pre-lab quiz on Canvas

Processor	Year	Transistor count	Notes
Intel 4004	1971	2,250	4-bit, first commercial microprocessor
Intel 8080	1974	6,000	8-bit, popular in early computing systems, calculators, cash registers, and robots
MOS Technology 6502	1975	4,528	8-bit, cheaper than Intel, variants were used in Apple II, Atari and original Nintendo (NES) gaming consoles
Intel 8086	1978	29,000	16-bit, first in a long line using the "x86 architecture"
Intel Pentium	1993	3,100,000	32-bit, popular processors for personal PCs in the 1990s
Intel Core i7 Skylake K	2015	1,750,000,000	64-bit, quad-core + GPU GT2
Xilinx Virtex-Ultrascale VU440	2015	20,000,000,000	FPGA
NVIDIA GV100 Volta	2017	21,100,000,000	Graphics Processing Unit (GPU)
Intel Stratix 10 GX 10M	2019	43,300,000,000	FPGA

Table 5.1: Transistor Count of CPUs (https://en.wikipedia.org/wiki/Transistor_count)

5.5 Procedure

5.5.1 Vivado Project

Create a new project in Vivado with

- Project Type = RTL Project
- Default Part = Board: Basys3

5.5.2 Half Adder

Gate-Level Circuit

Create a new *design* source with:

- File type = SystemVerilog
- File name = `halfadder`
- File location = <Local to Project> (default)

Vivado will create this file for you, add it to the project, and include a default block of code. Replace the large comment block with a short one and add the module name next to the `endmodule` statement. It should look like Listing 5.1. **All files you create should have these same basic elements.**

Add inputs `a` and `b` and outputs `c` and `s` and use gate primitives to define the behavior of this module.

Draw a diagram of this module as a solid rectangle with the input and output ports around the outside and the gates and wire connections inside, as shown in Figure 5.1. (Yours needs to be large enough to contain the half adder circuit inside it.)

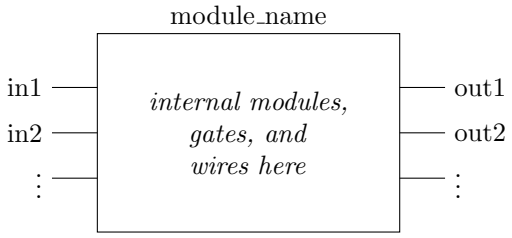

Figure 5.1: Module diagram example

Expected Results

Before you run a simulation, you should always created an expected results table (ERT). That is, what you *expect* your design to do. If you don't know what to expect, how can you know if the result is correct?!

For this lab, you already have ERTs in the form of truth tables from the previous two labs. We just need to put them in a form that is easy to compare with simulations. Table 5.2 provides an example for the half adder. Also see the LaTeX Tutorial for how to include your ERTs and simulation waveforms in your report.

Simulation

Create a new *simulation* source with

- Simulation set = sim_1

Table 5.2: Example expected results table (ERT)

Time (ns):	0	10	20	30
a	0	1	0	1
b	0	0	1	1
c	0	0	0	1
s	0	1	1	0

- File type = SystemVerilog
- File name = `halfadder_test`
- File location = <Local to Project> (default)

Like the design source, it should contain some initial code. Modify it to look like the code given in Listing 5.2 and add the remaining test cases for the half adder. Run a simulation and compare with the ERT. Do they match? If not, fix the issue or ask your instructor for help. Save a screen capture for your report.

5.5.3 Full Adder

Create a new design source named `fulladder`. Using Listing 5.3 as a starting point, define two instances of your `halfadder` and an OR gate to create a full adder.

Draw a diagram like the one in Figure 5.2 using the following conventions:

- Represent the half adders as blocks with the module name (`halfadder`) above the block and the instance name/identifier (`ha0`) in the middle of the block.
- Label their inputs and outputs *inside* the block.
- Label *internal* signals of the outer module above or below those lines (as done with `c1` and `s1`).
- Use the standard gate symbol for the OR gate. (i.e. It is okay to mix blocks and gates.)

Simulation

Convert your full adder truth table into an ERT.

Create a new simulation source with

- Simulation set = sim_2 ***new**
- File type = SystemVerilog

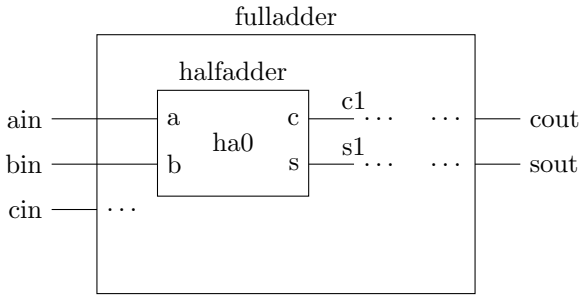

Figure 5.2: Module within a module diagram example

- File name = `fulladder_test`
- File location = <Local to Project> (default)

Use Listing 5.4 as a starting point. Run the simulation and confirm the results.

Note: You may need to right-click on `fulladder_test` under the `sim_2` folder and choose "Set as Top" to make it the primary file in that simulation set.

5.6 Adder/Subtractor

Repeat the same steps for a two-bit adder/subtractor. Name it `addsub`. Use 2-bit vectors for your inputs `a` and `b` and output `sum`, as done in Listing 5.5. Use two instances of your `fulladder` module and additional assignment statements for the XOR gates.

Draw a diagram. Note vector (multi-bit) signals using a slash with the number of bits above it, as shown in Figure 5.3.

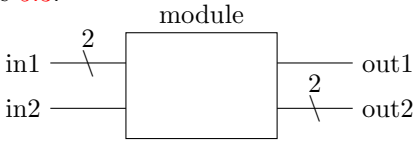

Figure 5.3: Vector (multi-bit) signal example

Simulation

Create an ERT, write a test bench, and run the simulation. Use the same test cases as in the previous lab, but do them for both addition and subtraction (so 14 cases total). Confirm that your simulation results match your theoretical results.

5.7 Deliverables

Submit a report containing the following:

1. All of your Verilog code (your half adder, full adder, and 2-bit adder/subtractor, and their test benches, 6 files total)
2. Block diagrams for each module (scan your hand-drawn diagrams and include as an image; make sure they are easy to read)
3. ERTs and screenshots of each of the three behavioral simulations (use the format in the LaTeX Tutorial)
4. Comment on whether the simulations match the expected output values
5. What is one thing that you still don't understand about Verilog?

5.8 Code Appendix

Listing 5.1: Half adder initial

```
`timescale 1ns / 1ps
// Your Name, ELC 2137, year-mo-dy

module halfadder(

   );

endmodule // halfadder
```

Listing 5.2: Half adder test bench initial

```
`timescale 1ns / 1ps
// Your Name, ELC 2137, year-mo-dy

module halfadder_test();

   // Define signals that will connect
   // to your module to be tested
   reg a, b;
   wire c, s;

   // Instantiate (create an instance
   // of) your design and connect
   // signals to it
   halfadder ha0(
      .a(a), .b(b),
      .c(c), .s(s)
   );

   // Define inputs/test cases
   initial begin
      // Test case #1 (wait 10ns)
      a=0; b=0; #10;

      // TO DO: add more test cases here

      $finish;
   end

endmodule // halfadder_test
```

Listing 5.3: Full adder initial

```
`timescale 1ns / 1ps
// Your Name, ELC 2137, year-mo-dy

module fulladder(
    input ain, bin, cin,
    output cout, sout
    );

    // Internal signals
    wire c1, c2, s1;

    // One instance of halfadder
    halfadder ha0(
      .a(ain), .b(bin),
      .c(c1), .s(s1)
      );

    // TODO: add another halfadder

    // Define carry output
    assign cout = c1 | c2;

endmodule // fulladder
```

Listing 5.4: Full adder test bench initial

```
`timescale 1ns / 1ps
// Your Name, ELC 2137, year-mo-dy

module fulladder_test();

    reg cin_t, a_t, b_t;
    wire cout_t, s_t;

    // TODO: instantiate your fulladder
    // module here

    initial begin
        cin_t = 0; a_t=0; b_t=0; #10;
        // TODO: add more test cases
        $finish;
    end
```

```
endmodule // fulladder_test
```

Listing 5.5: Adder/subtractor initial

```
`timescale 1ns / 1ps
// Your Name, ELC 2137, year-mo-dy

module addsub(
    input [1:0] a, b,
    input mode,
    output [1:0] sum,
    output cbout
    );

    // Internal signals
    wire c1, c2;
    wire [1:0] b_n;

    // Invert b input for subtraction
    assign b_n = ... // TODO

    fulladder fa0(
        .ain(a[0]), .bin(b_n[0]), .cin(
            mode),
        .cout(c1), .sout(sum[0])
        );

    // TODO: add second full adder

    // Convert carry to borrow when
    // subtracting
    assign cbout = ... // TODO

endmodule // addsub
```