STARS 2024

Lab Experiment 5: Introduction to SystemVerilog and the FPGA Development Environment

---

Objectives

- To gain an understanding of the various tools used to simulate Verilog, or synthesize, place-n-route, and flash it to an FPGA.
- To learn how to convert a given digital circuit design into Verilog using structural and dataflow syntax.
- To learn how to use the FPGA board simulator to test these designs with simulated push buttons and LEDs.
- To learn how to write a testbench that will programmatically change inputs and check the outputs to verify the functionality of a design.

Step 1: Implement and test various logic gates

From the lecture, we discussed how we can try out simple logic gates by creating signals as inputs and outputs, connecting them to the gate being tested, and observing the output as the inputs change. Let's quickly exercise our understanding of that syntax.

To be able to visualize this, we will use the FPGA board simulator to quickly test out some gates, and then try them on the physical FPGA connected to your lab machine. For reference, the simulator can be accessed at https://verilog.ecn.purdue.edu/.

An example: Recall the syntax for instantiating a gate in structural Verilog (the output is always the first port, followed by any number of inputs):

```
and (out, in1, in2);
```

As well as the syntax for creating one in dataflow Verilog:

```
assign out = in1 && in2;
```

To try this out on the FPGA simulator, we need to make use of the push buttons and LEDs so we can see the effect of the resulting circuit we implement in Verilog. The inputs available to you are the push buttons **(pb)**, and the outputs available are the two sets of red LEDs on either side of the board **(left/right)**, the center RGB LED **(red/green/blue)**, and the seven-segment displays. We suggest using the **red/green/blue** LEDs for testing single gates, and the individual bits of **left/right** for multiple-gate circuits.

In the case of this AND gate, we can replace the signals used above with push buttons 1 and 0, which are **pb[1]** and **pb[0]**, and the **right[0]** LED as the output. Writing the Verilog correctly and simulating it should turn on the rightmost red LED on the **right** bus of LEDs when you press both push buttons 1 and 0 at the same time.

Do this for each of the following gates, with the outputs connected to different LEDs as directed, but with the same pushbuttons.

- An AND gate connected to **right[0]**.
- An OR gate connected to **right[1]**.
- An NAND gate connected to **right[2]**.
- An NOR gate connected to **right[3]**.
- An XOR gate connected to **right[4]**.

Write code to achieve this in both **structural** and **dataflow** Verilog. Make sure to have one or the other commented out - you cannot assign two things to the same output!

Checkpoint L5C1: On the FPGA simulator, **demonstrate to your peer mentor/instructor** your understanding of the differences between the structural and dataflow Verilog, explain what output you expect from each gate for all 4 combinations of the two input push buttons, and demonstrate those outputs. **Make sure to document everything in your lab journal.**

Step 2: Set up your FPGA development environment

Now that you have **simulated** your Verilog on the FPGA dev board simulator, and you now know what to expect when you **flash** the same Verilog on your physical FPGA development board, let's set up our development environment on the lab machine to be able to use the physical FPGA board.

While logged in, open a terminal by pressing Ctrl-Alt-T on your keyboard, or press the Windows key on your keyboard and type 'terminal' and hit Enter to open it. A command-line interface will simplify setting up your tools, so we highly recommend learning to get used to the Linux-based terminal.

By default, your terminal should have opened at your home folder (where your Desktop and Downloads folder live). You can also find this folder if you press the Windows key and type 'files' to open the Files app. Type the following into the window that appears:

```
git clone https://github.com/STARS-Design-Track-2024/FPGA_Template
cd fpga-template
code .
```

This downloads a template project into a new folder "fpga-template" in which you can write Verilog, simulate it with a testbench, and/or configure your FPGA with the designs you write. You then "change directory" into the "fpga-template" folder, and start Visual Studio Code inside the folder (so that it opens the folder as a "workspace").

Visual Studio Code (or VScode) should appear as shown below (Figure 1). On the left sidebar, you should see a set of files included in the fpga-template folder. Their purposes are as follows:

- **top.sv**: The main file into which you will write the **Verilog design** you wish to implement. Since you will mostly be using this template to work with the physical FPGA, the default module included is the **top** module that presents push buttons (pb) and LEDs (left/right/ss7-ss0/red/green/blue). Don't worry about the "UART" ports - we won't use those.

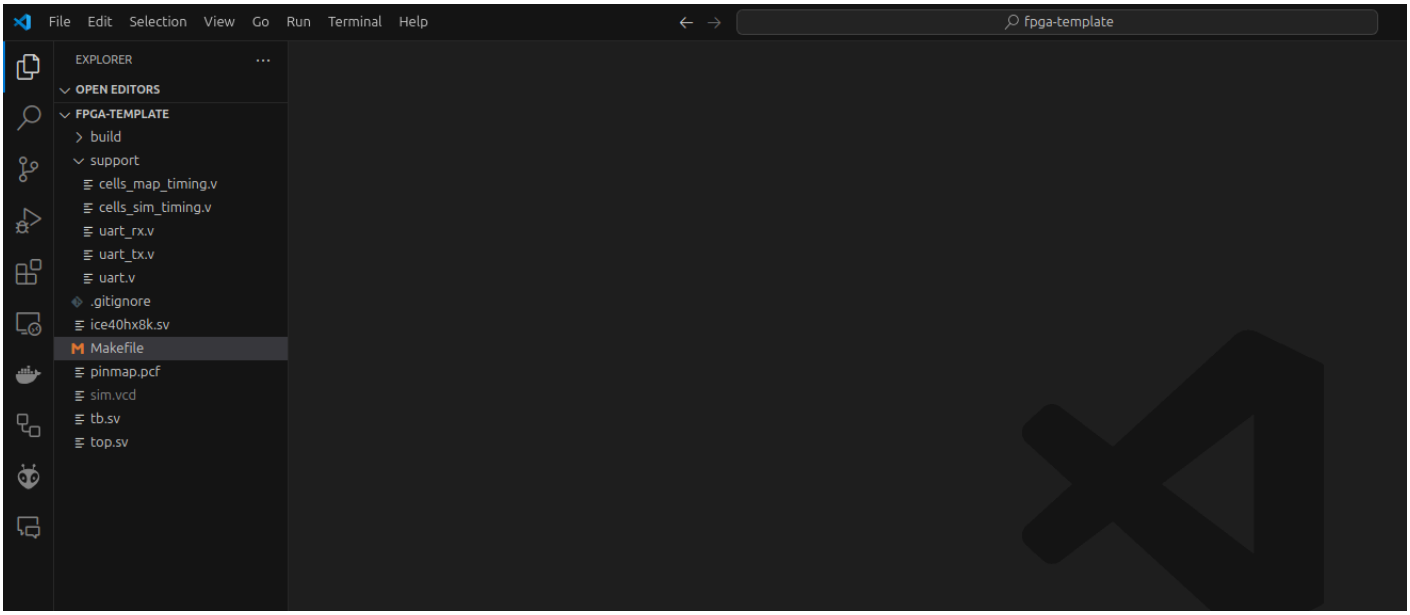


Figure 1: Visual Studio Code with opened folder in workspace

- **tb.sv** - If you wish to simulate your Verilog and see the outputs printed out before you try it out on the FPGA, you can write a **testbench** in this file and simulate it with the **top.sv** file. We'll dive into running testbenches in the next step.

- **Makefile** - we use the **Make build system** to automatically run a list of commands when we type something like "make some_command_here" into the terminal while cd'ed into **fpga-template**.
    - For example, to flash the FPGA with the Verilog you have written in **top.sv**, you would type **make cram** into a terminal. The "**cram**" target in the Makefile will perform the process laid out in Figure 2.
    - A quick glossary:
        - **Linting** is another word for checking for syntax errors.
        - A **netlist** contains the list of FPGA cells to be used to implement the circuit described in Verilog. It is produced as a JSON file by Yosys, a synthesis tool, from the Verilog you write.
        - **Place-n-route** is a process for allocating FPGA resources (muxes and flip-flops) to **place** the cells from the netlist file. It then generates data to **route** the cells to each other, and/or to input and output pins on the FPGA. We use NextPnR for this process, which generates an ASC file.

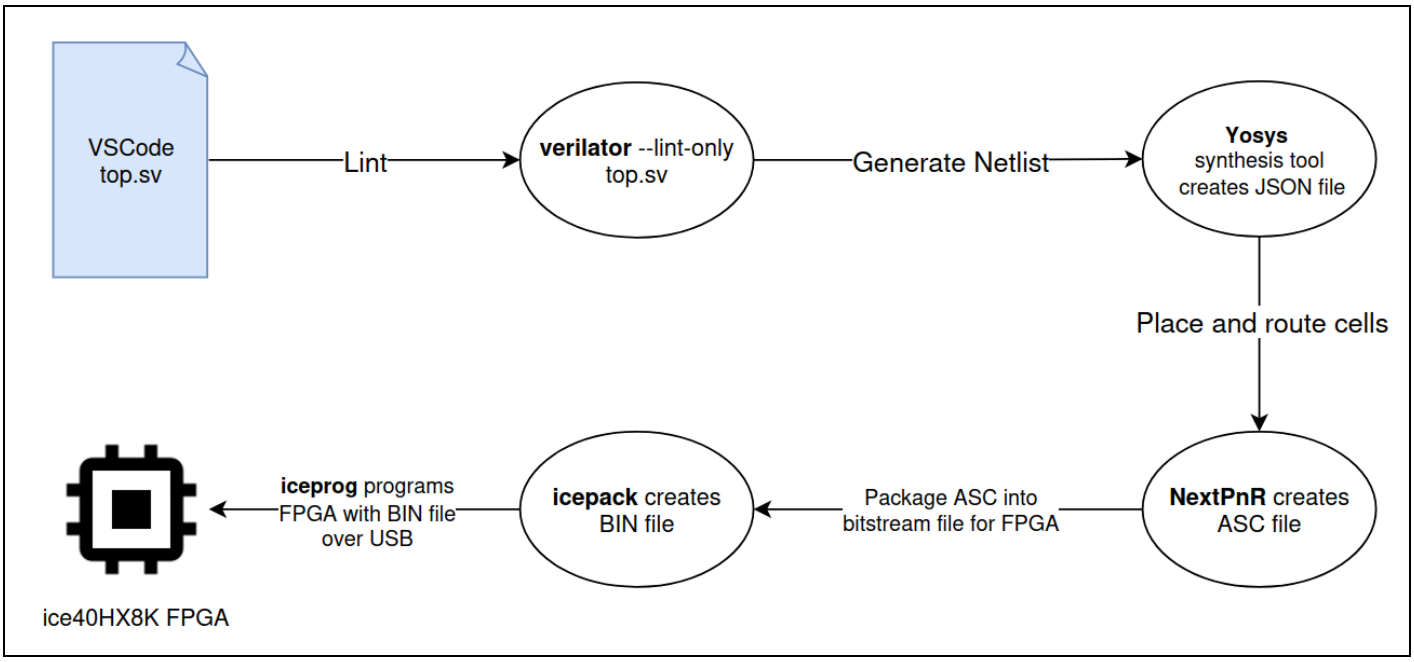


Figure 2: Transforming SystemVerilog code into FPGA bitstream file

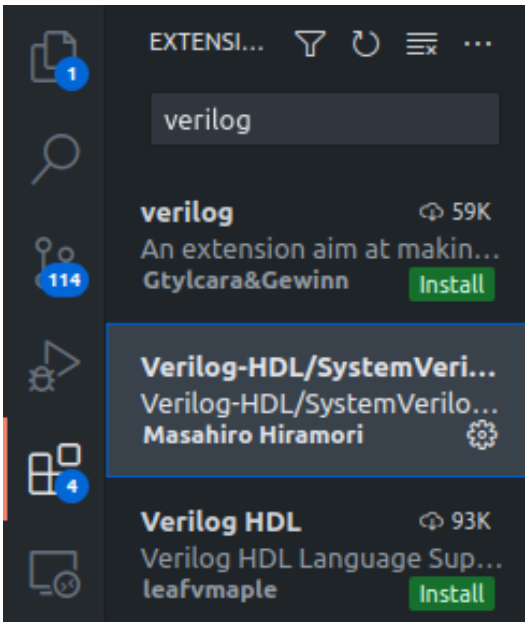
  - **The ASC** (Architecture-Specific Configuration) file is then transformed by **icepack** into a binary format (**BIN**) that can be sent to the FPGA in order to program it.
  - **iceprog** takes the resulting BIN file and flashes it into the FPGA's configuration RAM, or **cram**!

- **ice40hx8k.sv**: This is actually a top module for your own **top** module that you will write Verilog into, and better corresponds to the signals provided by the physical FPGA. It implements the **reset** signal, and divides down the FPGA's default 12 MHz clock into a new 100 Hz clock signal, **hz100**. You will use both of these in sequential Verilog later on.

- **pinmap.pcf:** This matches the inputs and outputs in our top-level Verilog module in the file above to the physical pins on the FPGA. This is a very important file because we need to ensure that our resulting circuit ends up using the correct pins for the push buttons and LEDs!

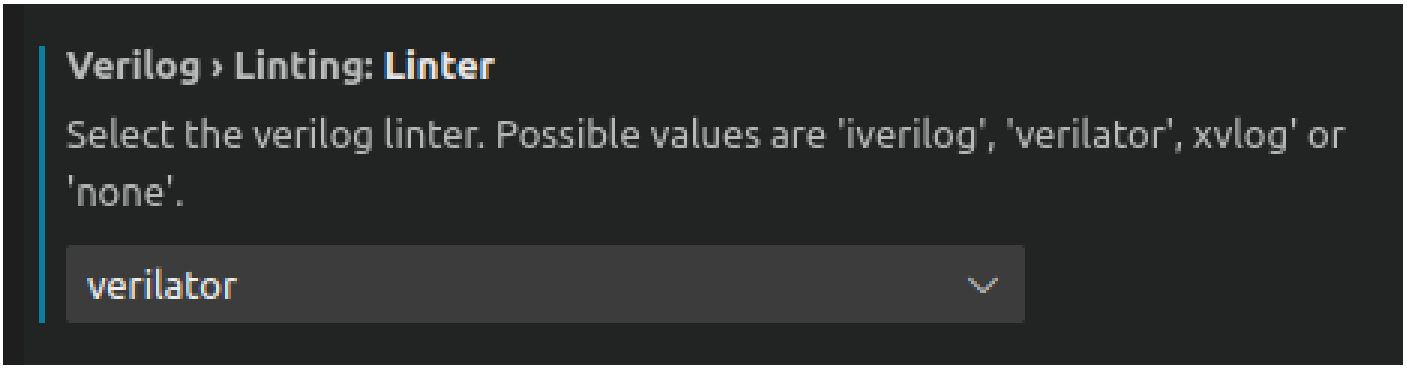
- **support**: Provides the cell libraries that you will use for simulating with testbenches, and necessary UART IP to provide a communication interface with our circuit if we need it.

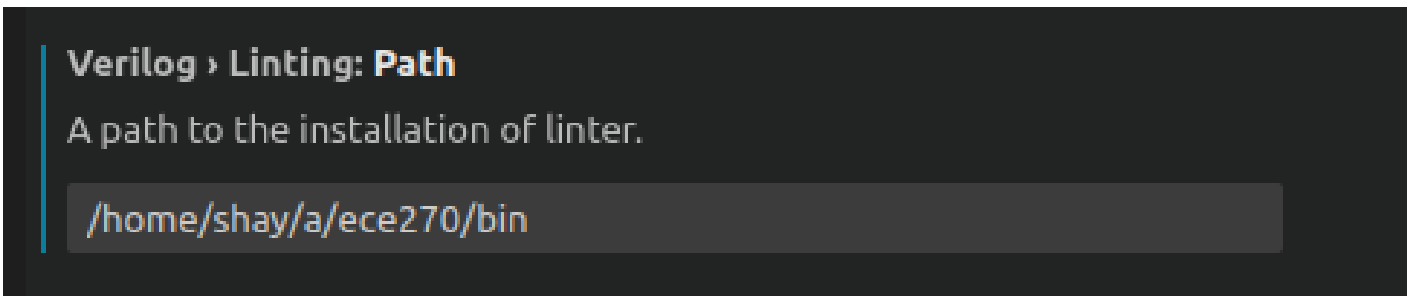
Before we start writing SystemVerilog, we will add and configure an extension to give us syntax highlighting and linting for our code. Click on the Extension button on the left sidebar of the program and search for Verilog (or press Ctrl-Shift-X), click the Verilog-HDL/SystemVerilog extension and install it. The extension should match the image on the right.



We'll configure the extension to use Verilator as a linter so that we don't have to keep typing "make cram" to do the linting for us - instead, the extension will report errors within the file editor itself. Right click on the extension and open up the "Extension Settings" menu.

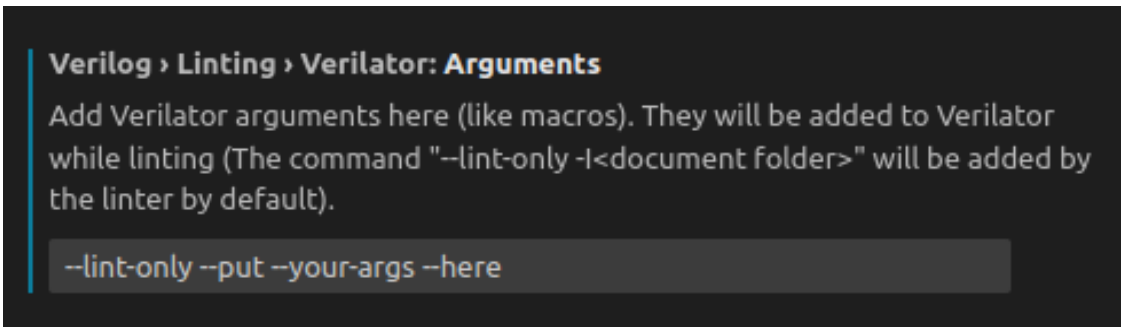
Scroll down to find the following settings and set them to the values as shown. This tells the extension that we wish to use **verilator** for linting, and we can find the **verilator** executable at the path **/home/shay/a/ece270/bin**.





Next, set the following Verilator arguments to the following under **Verilog > Formatting > Verilator: Arguments**. These flags are intended to **promote** some important warnings to errors, and to permit the use of timing delays in testbenches.

```
    --lint-only --timing -Werror-WIDTH -Werror-SELRANGE
-Werror-COMBDLY -Werror-LATCH -Werror-MULTIDRIVEN
```



Now that you have the linter and extension installed, your code should have syntax highlighting and warnings/errors (try typing in something incorrectly). If it doesn't seem to be working, ask for help.

Any errors in your code will be reported in the problems tab at the bottom of VSCode, every time you save the file (get used to pressing Ctrl-S!). An example of an error is shown below. You should pay careful attention to these errors, as they will tell you (with a bit of deciphering) what is causing your code to not function.

Checkpoint L5C2: **Demonstrate to your peer mentor/instructor** that you set up VScode with the extension and that when you open top.sv, the Verilog syntax is highlighted properly.

Copy in the code you wrote on the FPGA simulator into **top.sv**, and run **make cram** on the terminal. Show that it functions exactly the same way as the FPGA board simulator. **Make sure to document everything in your lab journal.**

Step 3: Implement and verify a 4-bit "ripple-carry adder"

An adder is a type of combinational circuit that takes three inputs, A and B, and a carry-in Cin and produces a sum S and a carry-out value Cout.

The "4" in 4-bit adder means that the size of the inputs A and B, and the output S, will be 4 bits in length, and the adder will actually be composed of four 1-bit full adders. We therefore need to create the full adders.

*The word "full" indicates the presence of a "carry-in" bit. Conversely a **half adder** will not have a carry-in bit.*

3.1: Create a truth table for a 1-bit full adder

To build up this circuit, we will first need to create a truth table for a **1-bit full adder** for all the possible inputs and corresponding outputs. Fill out the output values for **C** and **S** below.

A	B	Cin	Cout	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Do not forget that this is **binary addition**, not regular **decimal addition!** The 1s and 0s represent logic high and low respectively in a circuit.

3.2: Create equivalent expressions for Sum and Carry-Out

Now that you have a truth table for your adder, use it to form the necessary Boolean expressions for **Cout** and **S** based on **A**, **B** and **Cin**.

Then, convert these Boolean expressions to Verilog dataflow expressions. They should start as follows:

```
assign Cout = ...
assign S = ...
```

3.3: Create and test the 1-bit full adder module (fa)

Write out the Verilog to create a new module named **fa** with the following port information:

Port	Type	Length
A	input	1
B	input	1
Cin	input	1
S	output	1
Cout	output	1

In it, add the equivalent Verilog dataflow expressions you wrote earlier.

Add this module to your **top.sv** file. Do not replace the **top** module with it! We'll **instantiate** the full adder module in there later so we can use push buttons and LEDs to test the module as well.

3.4. Write a testbench to test your module

To give you an idea of how a good testbench should look so that you can write it yourself for the 4-bit adder when you create it, we will provide one for the 1-bit full adder you've written. Copy the testbench below, and replace the contents of the **tb.sv** file with it.

```
`timescale 1ms/10ps
module tb;

 logic A, B, Cin, S, Cout;
 fa fulladder (.A(A), .B(B), .Cin(Cin), .S(S), .Cout(Cout));

 initial begin
   // make sure to dump the signals so we can see them in the waveform
   $dumpfile("sim.vcd");
   $dumpvars(0, tb);

   // for loop to test all possible inputs
   for (integer i = 0; i <= 1; i++) begin
     for (integer j = 0; j <= 1; j++) begin
       for (integer k = 0; k <= 1; k++) begin
         // set our input signals
         A = i; B = j; Cin = k;
         #1;
         // display inputs and outputs
         $display("A=%b, B=%b, Cin=%b, Cout=%b, S=%b", A, B, Cin, Cout, S);
       end
     end
   end

   // finish the simulation
   #1 $finish;
 end

endmodule
```

3.5. Run the testbench

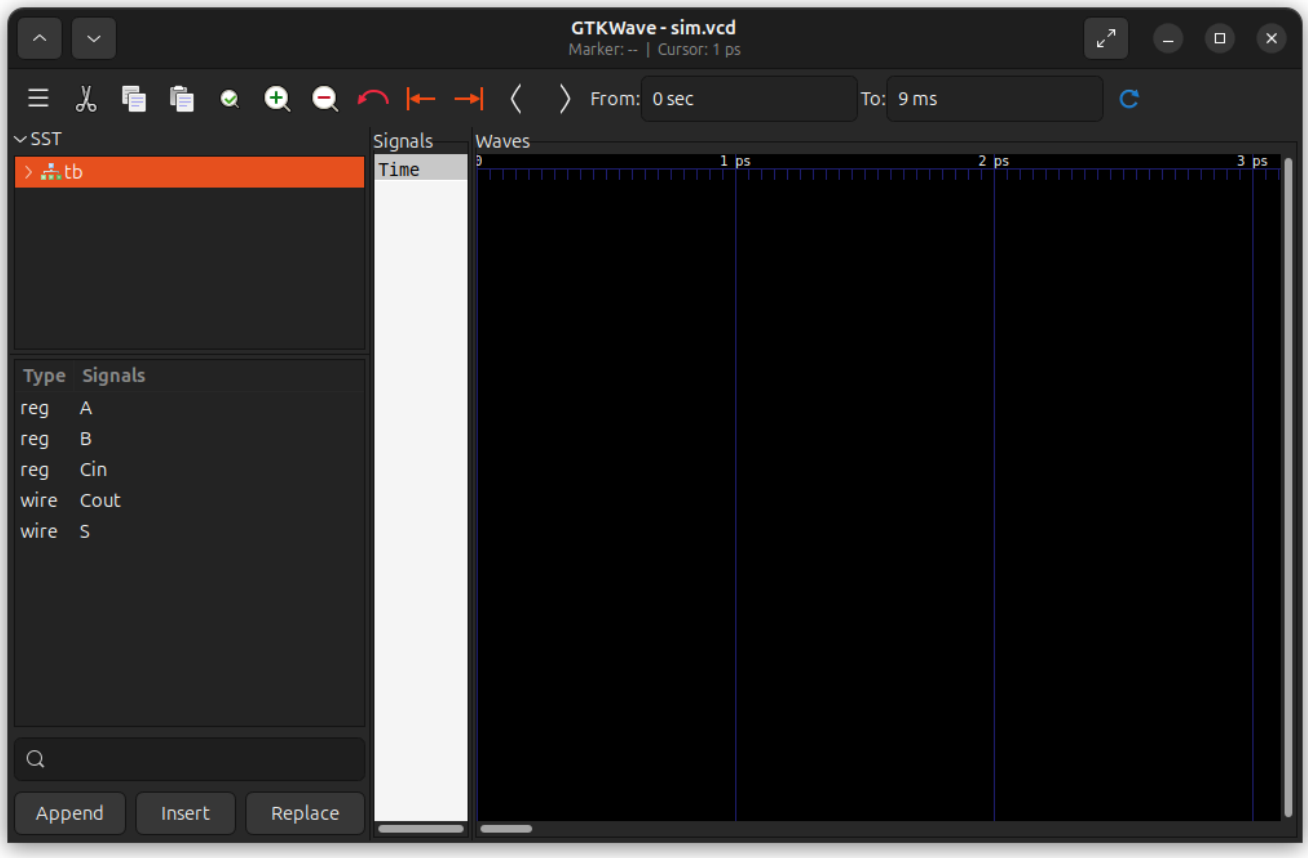
Ensure that you have the **fa** module in **top.sv**, and that the **tb.sv** file contains this testbench, and then in the terminal from where you opened VScode, run **make sim**.

The **sim** target will run half the process that we described back in Step 2 - it checks your Verilog for errors and synthesizes the Verilog to FPGA cells, but instead of placing and routing them, it writes the cells and connections between them back to Verilog, and simulates

the resulting Verilog with your testbench. (It is possible to simulate the Verilog directly, but simulating it at the cell level offers us more accuracy in circuit behavior). Notice that the **$display** calls we added will print out the input and output values.

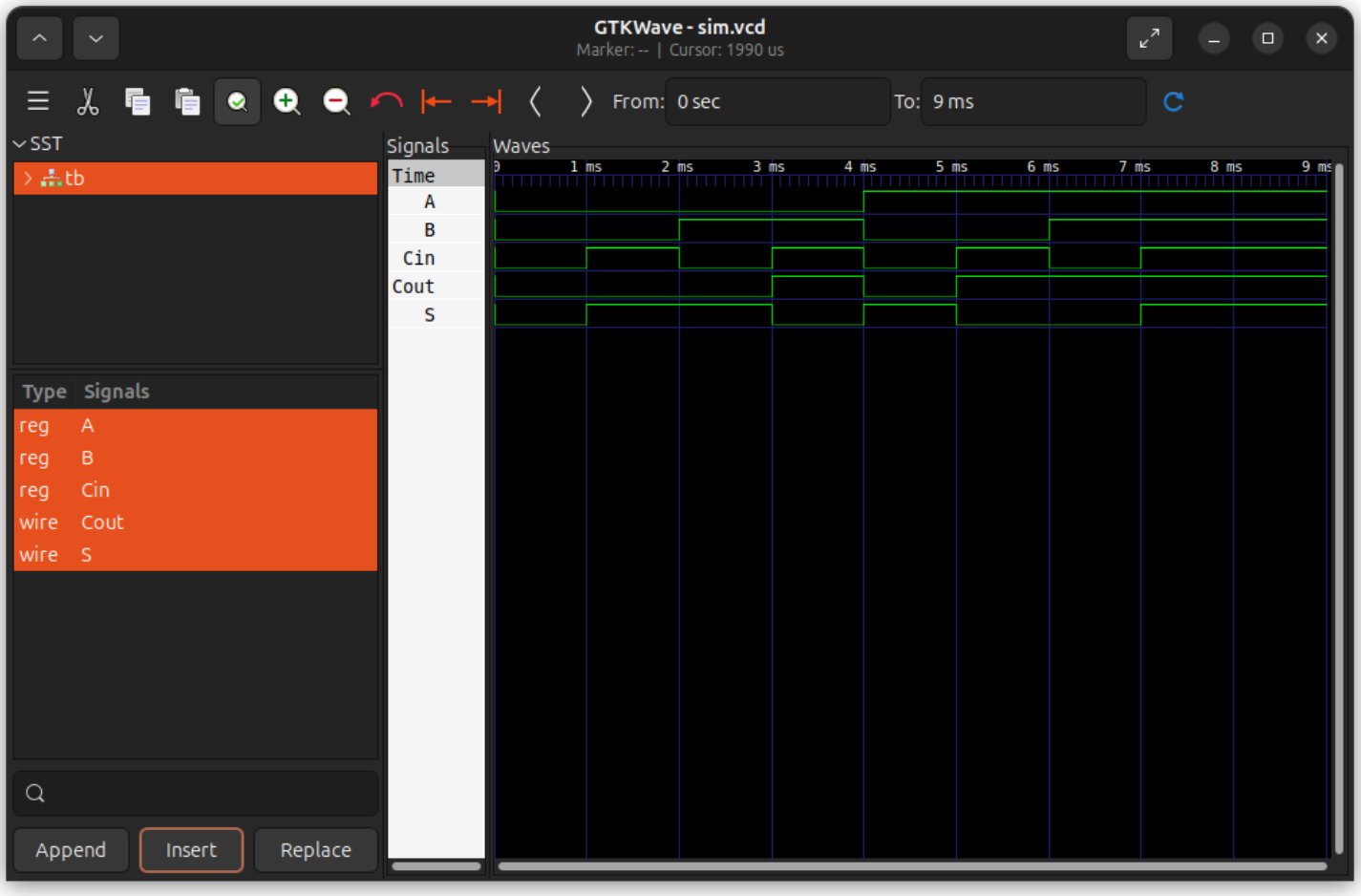
```
VCD info: dumpfile sim.vcd opened for output.
A=0, B=0, Cin=0, S=0, Cout=0
A=0, B=0, Cin=1, S=1, Cout=0
A=0, B=1, Cin=0, S=1, Cout=0
A=0, B=1, Cin=1, S=0, Cout=1
A=1, B=0, Cin=0, S=1, Cout=0
A=1, B=0, Cin=1, S=0, Cout=1
A=1, B=1, Cin=0, S=0, Cout=1
A=1, B=1, Cin=1, S=1, Cout=1
tb.sv:24: $finish called at 900000000 (10ps)
```

At the same time, you should see this program called GTKwave appear. GTKwave is an example of a program called a **waveform viewer**, and it allows you to visualize the execution of your testbench with waveforms, kind of like a "digital oscilloscope".



Highlight the **tb** module in the left sidebar as shown, and double-click each of the signals. Note that they are all the same signals defined in the testbench, and connected to the **fa** module instance.

After adding all the signals to the central Waves area, click the first magnifying glass (the fifth icon on the top toolbar) which will zoom out all the way so that the Waves area shows the results of the entire simulation, which should look like this (if your **fa** module is correct):



Note and answer the following:

- The inputs appear to change every millisecond. What statement does this? Why milliseconds?
- What are the values of the inputs for each “millisecond portion” of the waveform? Do the corresponding output values make sense for those outputs?

3.5. Try out your 1-bit adder on the physical FPGA

Having successfully simulated our adder with a testbench, let's try it on the physical FPGA. Instantiate the **fa** module in **top** under **top.sv**, and connect the ports as follows:

- A to push button 2 (**pb[2]**)
- B to push button 1 (**pb[1]**)
- Cin to push button 0 (**pb[0]**)
- Cout to the 2nd rightmost LED on **right** (**right[1]**)
- S to the rightmost LED on **right** (**right[0]**)

Run **make cram** to flash your adder to the FPGA. Try pushing the buttons and observing the outputs, and ensure that the correct LEDs turn on for each combination of the buttons 2, 1, and 0.

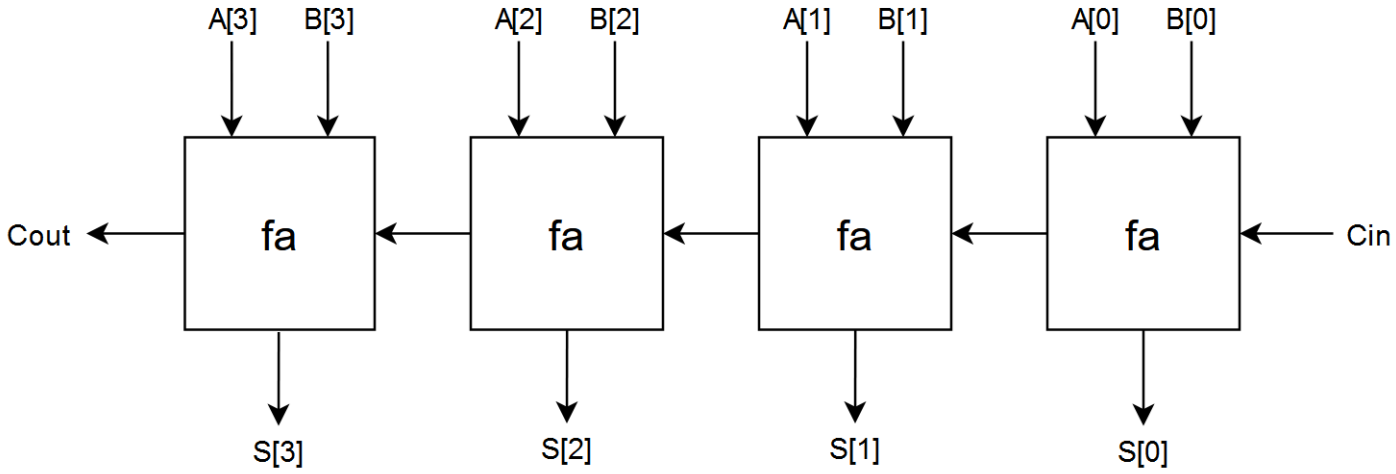
3.6. Implement and verify a 4-bit ripple carry adder

Now that you have a single 1-bit adder, create the 4-bit ripple carry adder simply by using 4 instances of the 1-bit adder. The 4-bit ripple carry adder should be named **fa4**, and have the following ports:

Port	Type	Length
A	input	4
B	input	4
Cin	input	1
S	output	4
Cout	output	1

Keep in mind, the ports are not the same! The length of A, B and S has changed so they are now 4-bit buses.

Create the module definition and port header. Inside the module, create 4 instances of the full adder you created earlier as shown in this block diagram.



The connections between the **fa** modules represent the connections between the carry-out of the right instance and the carry-in of the left instance.

Next, create a testbench for the **fa4** module. Use the **fa** module testbench we gave you earlier and extend it so that it tests all possible values of the 4-bit inputs.

Use **make sim** to run the testbench, and **make cram** to try it on the physical FPGA. If you're not in lab, you can use the Protobench tool (https://verilog.ecn.purdue.edu/protobench/) to run the testbench, or the FPGA board simulator (https://verilog.ecn.purdue.edu/).

Before testing the module, write down some example inputs and expected outputs so that you know what to look for from the testbench and/or FPGA board.

Checkpoint L5C3: Demonstrate your fa4 module and your testbench to your peer mentor/instructor. Show the set of inputs and outputs you wrote down, and show that your **fa4** module is producing the correct outputs for those inputs. Mention any difficulties you came across, and how you resolved them. **Make sure to document everything in your lab journal.**

A quick summary of methods and when to use which one:

It might feel like we just went over numerous ways to test the exact same Verilog, which can be confusing. Here are our suggestions on when you should use each method, assuming that you have just written out a Verilog module to try.

For simulating very small designs that can be tested with push buttons and LEDs anywhere: use the FPGA board simulator at https://verilog.ecn.purdue.edu/.

For prototyping short testbenches anywhere: use the testbench prototyping tool at https://verilog.ecn.purdue.edu/protobench/. Don't run very long testbenches or huge designs on the website - it needs to render the very long resulting waveforms in your browser, which is a slow process. Use the lab machine infrastructure for that.

If you're already in lab and want to run a testbench against your design: write your Verilog modules into top.sv under the **top** module, and run **make sim**.

If you're already in lab and want to physically try out your design: write your Verilog modules into top.sv beneath the **top** module, instantiate them in the **top** module so that you can use the physical push buttons and LEDs with your modules, and run **make cram**.