EECS 151/251A ASIC Lab 2: Simulation

Written by Nathan Narevsky (2014, 2017) and Brian Zimmer (2014)
Modified by John Wright (2015, 2016) and Arya Reais-Parsi (2019)

Overview

In lecture, you have learned how to use Verilog to describe hardware at the register-transfer-level (RTL). In this lab, you will first learn how to simulate the hardware that you have *described* in Verilog with a CAD tool called Synopsys VCS. The goal of this lab is to introduce the Verilog simulation environment, as well as demonstrate how useful and critical simulation will be as we move forward with the necessary CAD tools for VLSI design. While going through the lab, keep in mind that understanding how the simulation environment works will lead to confidence in designs after running through the CAD tools, and significantly less hassle in debugging problems when they occur.

This lab uses tools that may not be installed on all servers, so we recommend that you login to one of the class servers which are physically located in Cory 125, which are named `c125m-1.eecs.berkeley.edu` through `c125m-16.eecs.berkeley.edu`. You can access them remotely through SSH (see the last section of the Lab 1 handout). You may also try the `hpse-10.eecs.berkeley.edu` through `hpse-15.eecs.berkeley.edu` if you are having trouble with the `c125m` machines.

Take this opportunity to download the VCS user guide from the `eecs151` class-account home directory: `/home/ff/eecs151/labs/manuals/vcsmx_ug.pdf`.

To begin this lab, get the project files by typing the following command

```
git clone /home/ff/eecs151/labs/lab2
cd lab2
```

RTL-level simulation: FIR filter

For this lab, we will be using Verilog code that implements a very simple FIR (Finite Impulse Response) filter. A schematic of the filter is shown below.

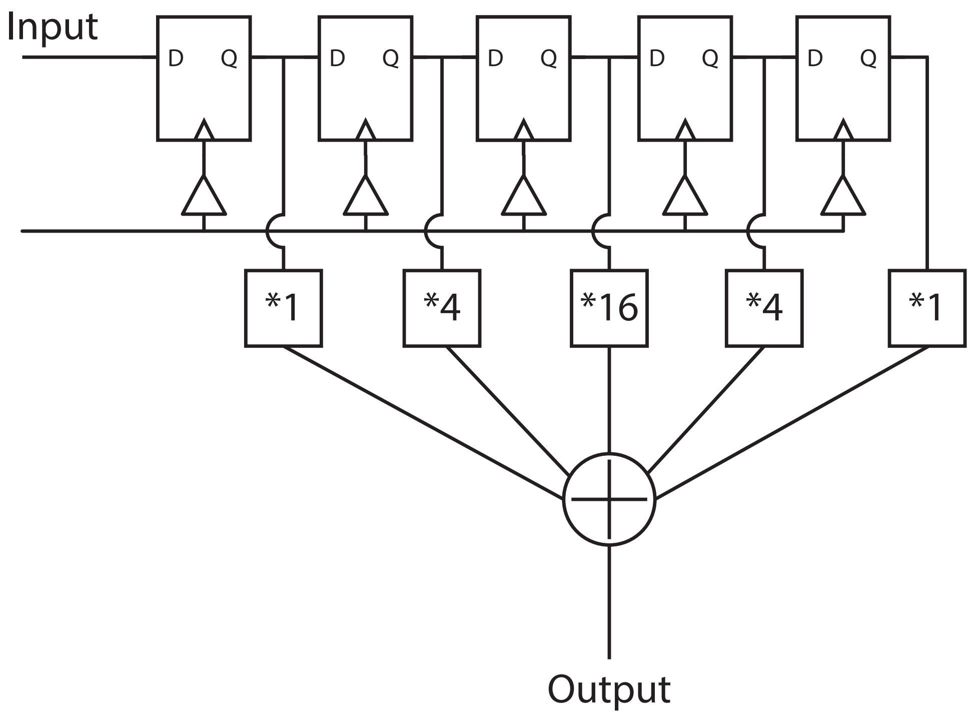


Figure 1: FIR Filter

There is an input signal and a clock input, and 5 delayed versions on the input are kept, multiplied by different coefficients and then summed together. The math expression for this particular filter is:

$$y[n] = 1 * x[n] + 4 * x[n-1] + 16 * x[n-2] + 4 * x[n-3] + 1 * x[n-4]$$

The input in our example is a 4 bit signed 2's compliment number, and the output is a larger bitwidth signed number to ensure that there is no overflow. The purpose of this class and this lab in particular is not to focus on filter design, but on implementation of digital circuits from Verilog code. As such, Verilog code for this FIR filter is provided in the src folder.

For simulating Verilog, we will be using a tool from Synopsys called VCS. VCS works by compiling Verilog modules into a binary file, and then executes that binary file to produce the desired outputs. This simulation framework is very fast and can be easily scripted, as you will see throughout this lab. VCS is a command line based tool, so for your convenience we have provided a Makefile that will run VCS with different options, which will be explained throughout the lab.

To get started let us look through a part of the Makefile included in the project files folder. Below are some selective lines copied and pasted to highlight the basic functionality of this Makefile.

```
include ./Makefrag

default : all

basedir  = ./
```

```
# Verilog sources

srcdir = $(basedir)/src
vsrcs = \
    $(srcdir)/fir.v \
    $(srcdir)/addertree.v \
    $(srcdir)/fir_tb.v \

#--------------------------------------------------------------------
# Build rules
#--------------------------------------------------------------------

VCS      = vcs -full64
VCS_OPTS = -notice -PP -line +lint=all,noVCDE +v2k -timescale=1ns/10ps -debug


#--------------------------------------------------------------------
# Build the simulator
#--------------------------------------------------------------------

vcs_sim = simv
$(vcs_sim) : Makefile $(vsrcs)
    $(VCS) $(VCS_OPTS) +incdir+$(srcdir) -o $(vcs_sim) \
           +define+CLOCK_PERIOD=$(vcs_clock_period) \
           -sverilog $(vsrcs)


#--------------------------------------------------------------------
# Run
#--------------------------------------------------------------------

vpd = vcdplus.vpd
$(vpd): $(vcs_sim)
    ./simv +verbose=1
    date > timestamp

run: $(vpd)

#--------------------------------------------------------------------
# Default make target
#--------------------------------------------------------------------

.PHONY: run

all : $(vcs_sim)
```

There is a lot of code here, but let us walk through the different pieces to show what each of them does. The first few lines setup the files that will be necessary for running the tool, and make sure that the libraries are defined properly. The Makefrag file that gets included sets up a few of the variables that get used in the makefile, such as the `$(vcs_clock_period)` variable. Under the comment that says "Build rules" two things are defined, `VCS` and `VCS_OPTS`. `VCS` is the command to call the vcs simulator, and `VCS_OPTS` defines some of the options thare passed to the simulator. One important setting here is the `-timescale` option, which sets the timescale for the Verilog testbench. We will discuss what this actually means later on in the lab. Under the comment that says "Run" the makefile sets up the process of running the executable output of VCS to generate a ".vpd" file. This file contains simulation history data, and is generated in the Verilog testbench with the `$vcdpluson` and `$vcdplusoff` system tasks. A system task in Verilog is a function that is created by the compiler to make things easier for you, and usually begins with a `$` character. We will not dig deeper into system tasks in this lab, but they are very powerful tools that are worth investing the time in learning.

Below the "Default make target" comment, the final lines set up run as a phony make target, which means that it is simply a shortcut for running a variable target, and will not have it's own file that it creates. The "all" target was setup as the default target towards the beginning of the file, and here it points to the `$(vcs_sim)` target.

To actually run the makefile, we first need to setup the path for the tools. To do this run the following command:

```
source /home/ff/eecs151/tutorials/eecs151.bashrc
```

This sets up the correct environment variables to be able to run VCS and the other CAD tools that we will use in later labs. Run the Makefile to create an output as well as the .vpd waveform file with the following command:

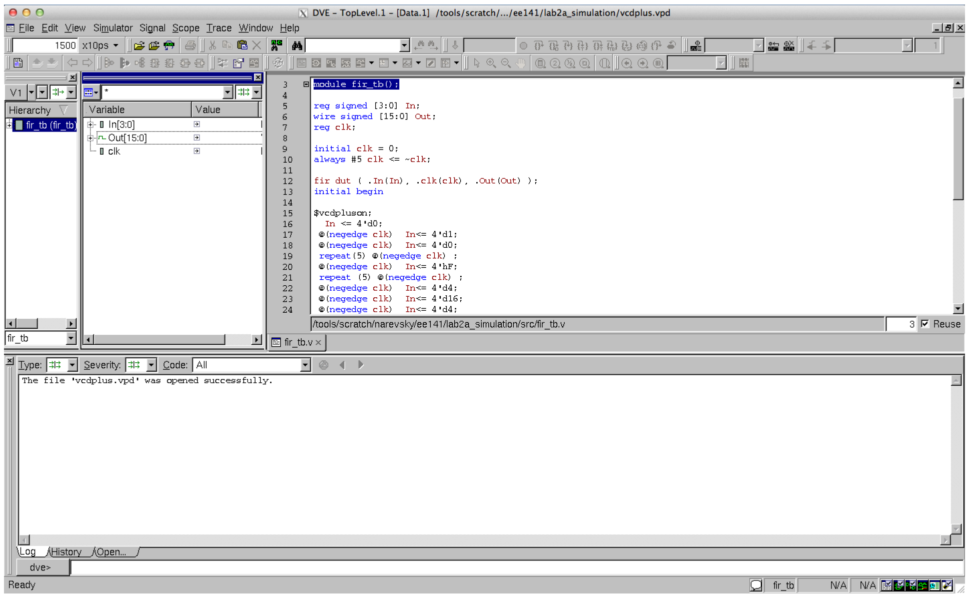
```
make run
```

The first line you should see is the command that the Makefile created for you. Next, VCS compiles your Verilog code. If your code has errors, they will be shown during this step. There are also "Lint" errors, that attempt to catch Verilog mistakes that aren't actual syntax errors, but cause common bugs. We intentionally left code in the design to trigger these messages. In this case, assigning a number like "1" instead of `16'd1` gives Verilog less information about how you want the bits interpreted, which could cause a design bug that is hard to track down. At some point, you will see `./simv` which means that compilation has finished, and the simulator has started.

At the end of the output text, the testbench should report the observed values with the expected values. You can even automate the checking of these outputs in either the Verilog testbench itself or through parsing the output text directly, but that will be explained in more detail later. Let us load up the waveforms in a graphical viewer DVE (Discovery Visualization Environment). (Note: this will require X11 forwarding, so if you ssh from a terminal be sure to use either the -X or -Y flag. If you are using Putty, make sure to enable X11 forwarding and have the proper X server software installed. An example of an X server is a program called Xming, although there are a ton of other options). You can also use X2Go, NoMachine, or VNC.

```
dve -vpd vcdplus.vpd &
```

When DVE pops up, you should see the window below:


DVE - TopLevel.1 - [Data.1] /tools/scratch/.../ee141/lab2a_simulation/vcdplus.vpd
File Edit View Simulator Signal Scope Trace Window Help
1500 x10ps
Hierarchy
fir_tb (fir_tb)
Variable
Value
In[3:0]
Out[15:0]
clk
fir_tb
/tools/scratch/narevsky/ee141/lab2a_simulation/src/fir_tb.v
Reuse
fir_tb.v
Type: Severity: Code: All
Log History Open...
dve>
Ready
fir_tb N/A N/A

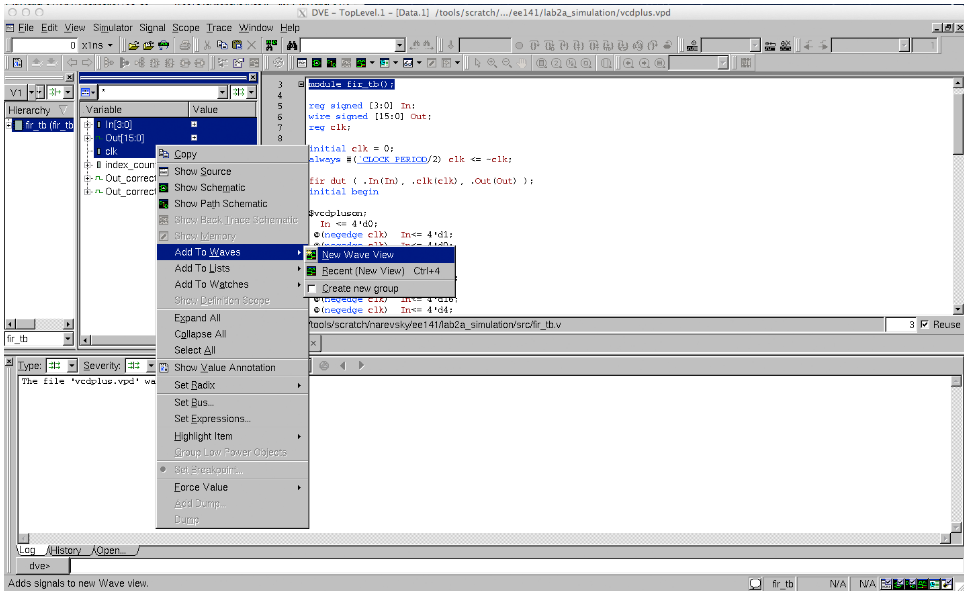
```
module fir_tb();

reg signed [3:0] In;
wire signed [15:0] Out;
reg clk;

initial clk = 0;
always #5 clk <= ~clk;

fir dut ( .In(In), .clk(clk), .Out(Out) );
initial begin

$vcdpluson;
  In <= 4'd0;
 @(negedge clk)   In<= 4'd1;
 @(negedge clk)   In<= 4'd0;
 repeat(5) @(negedge clk) ;
 @(negedge clk)   In<= 4'hF;
 repeat (5) @(negedge clk) ;
 @(negedge clk)   In<= 4'd4;
 @(negedge clk)   In<= 4'd16;
 @(negedge clk)   In<= 4'd4;
```

```
The file 'vcdplus.vpd' was opened successfully.
```

Figure 2: DVE Window

This window will contain the design hierarchy, the signals in that level, as well as a lot of other buttons and options. One important feature is towards the top left of the window where it has a textfield and next to that says "x10ps". This represents the units of time that the signals are plotted on. Change this to be in units of nanoseconds.

You can add waveforms to a new view by selecting a few of them (In, Out and clk) and the right clicking and going to "Add To Waves" and then clicking either "New Wave View" or "Recent". Since there are no other wave views open these both accomplish the same thing. If you already had a wave view open, then you can selectively add signals to one wave view or another. A screenshot of adding these waveforms is shown below.



```
module fir_tb();

reg signed [3:0] In;
wire signed [15:0] Out;
reg clk;

initial clk = 0;
always #(`CLOCK_PERIOD/2) clk <= ~clk;

fir dut ( .In(In), .clk(clk), .Out(Out) );
initial begin

$vcdpluson;
  In <= 4'd0;
 @(negedge clk)   In<= 4'd1;
 @(negedge clk)   In<= 4'd16;
 @(negedge clk)   In<= 4'd4;
```

/tools/scratch/narevsky/ee141/lab2a_simulation/src/fir_tb.v

The file 'vcdplus.vpd' wa

Figure 3: Adding waveforms

The picture below displays the output of the FIR filter as a step waveform. This can be achieved by setting the radix of the signal to be 2's compliment (right-click — Set Radix — Twos compliment) and then changing the Set Draw Style Scheme to be Vector: Analog. You then need to change the properties of the signal since the beginning of the waveform does not utilize the full scale. If you right click and go to properties, you can set the Analog Waveform properties to be User, and set the Min to be -32 and the max to be 32. Then you should see something similar to the waveform below.

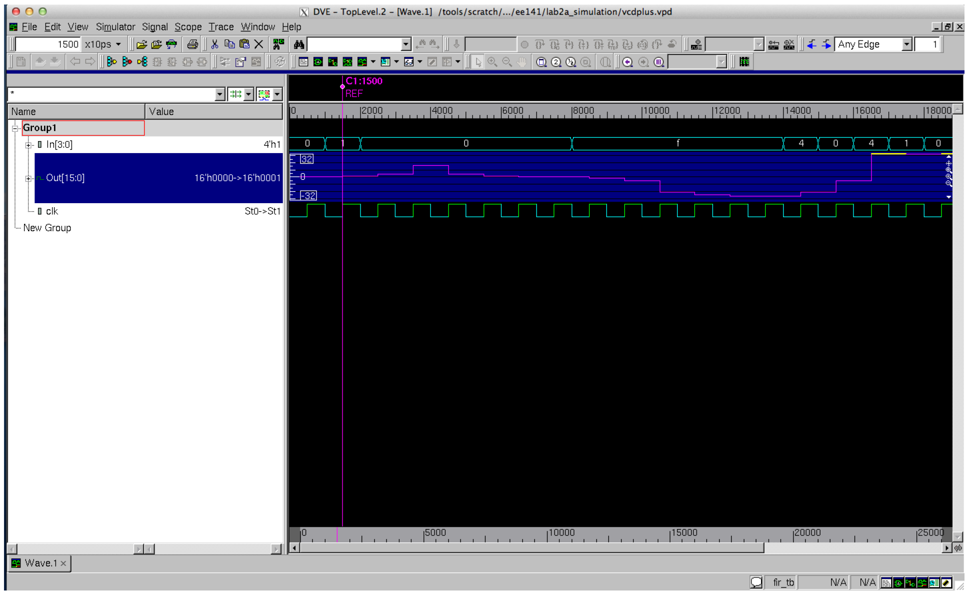


Figure 4: Displaying Waveforms

For those of you who know a bit about FIR filters, this beginning section of the waveform is the filter's impulse response.

Testbenches

Testbenches are how you simulate a design. Specifically they setup the inputs and check the outputs of the submodule that you are trying to test. If you look at the `fir_tb.v` file in the `src/` folder, there are a few important parts that you will need to understand in order to write your own testbench. The first important piece is generating the clock waveform. This is accomplished by the following lines of Verilog code:

```
reg clk;

initial clk = 0;
always #(`CLOCK_PERIOD/2) clk <= ~clk;
```

This creates a register named clk, which is initially 0 and toggles every CLOCK_PERIOD divided by 2 (in order to generate a rising edge every CLOCK_PERIOD). This clock period is defined through the Makefile process that we will discuss in more detail in the next section in the lab. After we have a clock waveform, the next step is to setup the inputs. There is an initial begin block that is the body of the testbench that does this work. Let us look at a piece of it:

```
initial begin
$vcdpluson;
  In <= 4'd0;
 @(negedge clk)  In<= 4'd1;
 @(negedge clk)  In<= 4'd0;
 .
 .
 .
 @(negedge clk)  In<= 4'd13;
 @(negedge clk)  In<= 4'd14;
 @(negedge clk)  In<= 4'd15;
$vcdplusoff;
$finish;
end
```

The `$vcdpluson` and `$vcdplusoff` are system tasks that setup the vpd file generation that we used to look at the waveforms in the previous section. The other lines setup the register In to take on different values after the negative edge of the clock. In this block, the lines are executed after each other, so the next `@(negedge clk)` call waits until the next negative edge of the clock before executing the code that follows it. This allows us to set up a series of values for the inputs sequentially in time. The reason that we are operating on the negative edge of the clock is that the registers that are sampling the inputs operate on the positive edge, so we want to make sure that the correct value is sampled when that edge occurs. If we were to change the inputs on the same edge, we could cause a hold time violation later when delay gets annotated.

The above code sets up the values sequentially, but you can do this in a much cleaner way by reading values from a file. Take a look at the file `src/fir_tb_file.v`, parts of which are copied below:

```
initial begin
$vcdpluson;
 repeat (26) @(negedge clk);
$vcdplusoff;
$finish;

end

initial begin
    $readmemb("data_b.txt", Out_correct_array);
    $readmemb("input.txt", input_array);
end
assign Out_correct = Out_correct_array[index_counter];
assign In = input_array[index_counter];
always @(negedge clk) begin
    $display($time, ": Out should be %d, got %d", Out_correct, Out);
    index_counter <= index_counter + 1;
end
```

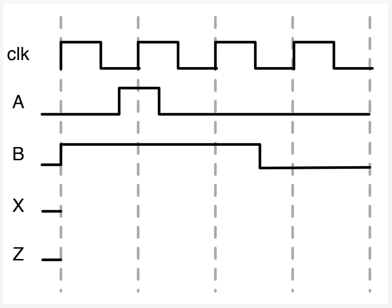
This testbench file uses text files to pull both the inputs and expected outputs, as well as displays what the output is and what it thinks the correct output should be. This is done using the `$readmemb` command, which reads a file into a verilog memory that you instantiate in the testbench. By then using a counter to loop through both the input and the desired output, the testbench simply needs to run for the right number of cycles, which is shown using the `repeat` syntax inside the first initial begin block. Finally, this testbench uses `$display`, which is verilog's print statement to print out information about the waveforms to the console so that you do not need to look at the waveforms. More sophisticated testbenches can be created this way so that you do not have to look at the waveforms to gather the necessary information, allowing you to automate your testing procedures. While this is a small enough design that you could in theory debug using only the waveforms, the project later in this course will be much more complicated so learning how to build automated testbenches will be very important.

Question 1: Conceptually translating between waveforms, Verilog, and schematics

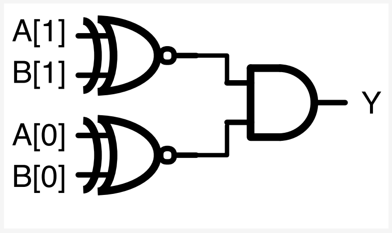
When asked to write Verilog, include the module definition. There are multiple correct solutions, we will accept any solution that works.

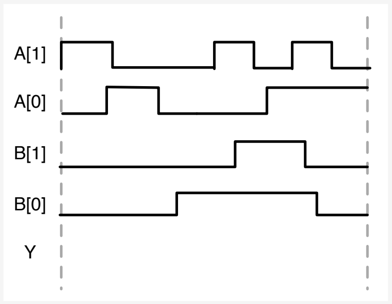
a) Using the provided Verilog code and input waveforms, submit the equivalent schematic and draw/ sketch the output waveforms. Note that the initial conditions of X and Z are given.

```
module dut (
  input A, B, clk,
  output reg X, Z
);
always @(posedge clk) begin
  X <= B;
  Z <= (Z & X) | A;
end
endmodule
```

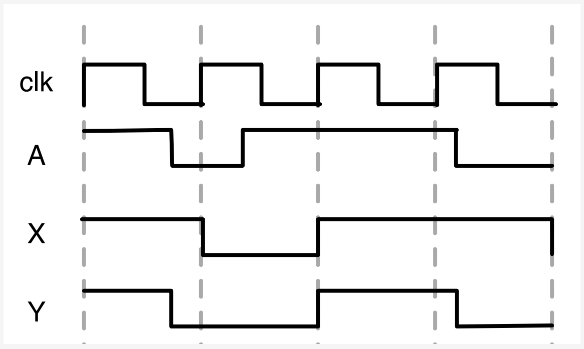


b) Using the schematic below and input waveforms, submit the equivalent Verilog code and a sketch of output waveforms





c) Using the input and output waveforms provided below, submit the equivalent schematic and Verilog code. Hint: Use 1 flip-flop and 1 logic gate only. A is an input, X and Y are outputs.



Question 2: Writing a testbench

a) Using the Verilog code in Question 1a, generate a testbench that emulates the input waveforms, then simulate the block using VCS and compare to your previous answer. Submit your Verilog testbench and a screenshot of the simulation waveforms showing all of the input and output pins.

Gate-level simulation: Incorporating delay estimates into the FIR filter

The RTL design of the FIR filter, `fir.v`, conceptually describes hardware, but cannot be implemented. During synthesis, a CAD tool translates RTL into logic gates from a particular technology library. In Lab 3a, you will learn how to create this file yourself, but for now we have provided the output result as `src/fir.mapped.v`.

To simulate using the gate level netlist, you simply need to make a few changes to the VCS setup. If you look at the Makefile, these changes have been done for you already, but it is important to understand these differences. The first difference in the Makefile is the difference between `vsrcs` and `vsrcs_gates`. These variables setup which Verilog files to include for the simulation, and for `vsrcs_gates` we also include a `cells.v` file which contains Verilog models of the standard cells. You will learn more about these standard cells in the next lab, but just know that you need to include that file to run the simulation otherwise the simulator will error since it cannot find the definitions for those modules. The extra options in the new VCS section of the Makefile are simply to deal with these standard cell models. One important new setup is in the vpd generation, where it says `-ucli -do run.tcl`. A UCLI (Unified Command-Line Interface) file allows a designer to issue commands directly to the simulator for various purposes. What we are using these files for in this lab is to set initial conditions for the registers in the design, since although we do not know their initial state, verilog simulators do not properly simulate with unknown or 'x' valued inputs. To truly test whether a real design will work or not you should be able to start with any set of initial conditions and the design will still work, which we will not do in this lab due to time constraints. This previous command also calls a run.tcl script, which actually sources the UCLI file, and runs the simulation. Note: this can only be done with the `-P access.tab` flag setup in the vcs section.

Go ahead and run the gate level simulation by entering the following command:

```
make run-gates
```

Notice that the waveforms look exactly the same as the results from simulating the RTL version of the design. By default, the logic gates behave ideally. Depending the operating conditions of the chip—voltage, process variation, temperature—the delay through a gate will be known. CAD tools do this calculation for you, and annotate the delay onto the gates using an SDF file.

Open `src/fir.mapped.sdf`, and go to line 1535.

```
(CELL
  (CELLTYPE "INVX1_RVT")
  (INSTANCE add0/U47)
  (DELAY
    (ABSOLUTE
    (IOPATH A Y (0.013:0.013:0.013) (0.011:0.011:0.011))
    )
  )
)
```

The above text describes the delay for a cell of type `INVX1_RVT` for the instance `add0/U47`. The format of the delay is minimum:typical:maximum, which refer to different operating regions that will be discussed in more detail in future labs. We will be simulating only using the typical numbers (which are in the middle) for this lab. For this specific instance that means that there will be a delay of either 0.013ns or 0.011ns, depending on whether the data is transitioning from low to high or from high to low.

Remember that previously we had talked about the `VCS_OPTS`, and how there was a `-timescale` flag. This was specified to have the value `1ns/10ps`, which means that a delay of 1 would correspond to an actual delay of 1ns, with a resolution of 10ps.

To tell the simulator about these delays, navigate to the `$(vcs_sim_gates)` target in the Makefile and add the following options to the `$(VCS)` command:

```
+sdfverbose -sdf typ:fir:src/fir.mapped.sdf
```

In doing so be sure to remove the following flags as well:

```
+notimingcheck +delay_mode_zero
```

Simulate the design again, and open the resulting waveform. Zooming in on the same point in time that we looked at earlier (12.5ns in), from the DVE window figure out the delay of the first flip-flop in the chain (delay_chain0) is relative to its own clock edge (the clk-to-Q) as well as the `clk` input pin. Try out some of the other options in the wave viewer to try and figure out what is going on. If you get stuck on anything that you are trying to do, you can look up the Synopsys DVE User Guide, which has a significant amount of information about DVE, in the `eecs151` home directory: `/home/ff/eecs151/labs/manuals/dve_ug.pdf`.

Question 3: Calculating delays from the DVE window

Calculate the delay of the first flip-flop in the chain (delay_chain0) relative to the input pin `clk` at 12.5ns. What is the delay, and how were you able to get it from the viewer?

Now that we are simulating with delay, we can run the circuit too quickly. When this happens, there is a 'setup' violation and the design will not function correctly

Question 4: Creating and fixing setup times

Edit the Makefrag file to make the clock period 0.5ns and simulate again. Does it still work? Why or why not? Run it again at a larger clock period (but less than the initial 5.0ns) and report both the clock period you used and whether or not it successfully meets timing.

Re-run the design with the clock period set to 5.0ns again. Saved the vpd file as a different name, such as "vcdplus_gates.vpd" After you have renamed the previous vpd file, run:

```
make run-gates-hold
```

This will run the simulation with a different delay file, which intentionally has an error in it. You should now have two different vpd files, and you can load them in the DVE waveform viewer to see the differences. Add the delay_chain signals as well as the clk0-clk5 signals for both vpd files. Zoom into the clock edges near 7.53ns into the simulation. There should be a significant difference between the two vpd files, and one of them will have a signal that is incorrectly getting captured on the wrong cycle. This is an exaggerated case of a hold time violation, which occurs when a specific delay path is too small relative to another.

Question 5: Fixing hold times

Setup times can be fixed by increasing the clock period, but hold times must be fixed at design time. Later, you will learn how the CAD tools do this for you, but in this problem you will manually fix the design.

a) Explain the differences between the waveforms in the two vpd files. What signal(s) are different and why?

b) Modify the `src/fir.mapped.hold.sdf` file to fix the hold time. You are only allowed to increase the delays in one cell. Submit the line you changed (line number and new text), as well as simulation waveforms showing correct output and the text printout of the simulation showing that the results are correct.