Getting Started With GHDL and Simulation

Yehowshua Immanuel and Saloni Oswal

January 17, 2020

Intro

A Hardware Description Language(HDL) is purely that. It merely describes hardware. There are a couple types of HDLs, digital and mixed(digital+analog). This tutorial deals with digital HDLs.

HDLs aren't very useful by themselves. Typically, they might be passed to a synthesizer which reduces the HDL into a gate list that describes all the connections between various gates. A gate list is more commonly known as a netlist.

There are different kinds of synthesizers. Some synthesizer frameworks target FPGAs while others target physical fabrication or VLSI. In VLSI, HDL is just the first step in a long laborious sequence of tasks that usually results in a finished physical chip.

After writing some HDL, you may wish to know whether or not it does what you want. This can be accomplished using an HDL simulator. Currently, the only Free and Open Source(FOSS) VHDL simulator being maintained is GHDL.

Installing GHDL

Ubuntu

```
$sudo apt update
$sudo apt install build-essential gnat git llvm clang zlib1g-dev llvm-dev
$mkdir -p ~/src; cd ~/src
$git clone https://github.com/ghdl/ghdl
$cd ghdl
$mkdir build; cd build
$../configure --with-llvm-config --prefix=/usr/local
$make -j8
$sudo make install
```

MacOS

Installing HomeBrew package manager

first you should check if you have brew so in the terminal type `$brew`

If that command returns,

`-bash: brew: command not found`

you don't have brew. So do the following:

```
$/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com
$/Homebrew/install/master/install)"
```

Installing GHDL on Mac

```
$brew install wget
$mkdir -p ~/src; cd ~/src
$wget https://github.com/ghdl/ghdl/releases/download/v0.36/ghdl-0.36-macosx-llvm.tgz
$mkdir -p ghdl
$tar -C ghdl -xf ghdl-0.36-macosx-llvm.tgz
$rm ghdl-0.36-macosx-llvm.tgz
$cd ghdl/bin
$echo "export PATH=\"`pwd`:\$PATH\"" >> ~/.bash_profile
```

Windows

Install bash for windows on windows and follow the instructions above for Ubuntu.

Verifying Installation

Open up a new terminal and do `ghdl --help`. You should see a list of some ghdl commands and their explanations.

Simulation

Now that we have installed GHDL, lets simulate the hardware described in `MIPS-SingleCycle.vhdl`. Simulating the hardware often involves a couple things:

- pulling the reset signal high initially and then setting it low for the remainder of the simulation(if you are simulating synchronous hardware)
- constantly toggling the clock(or clocks for multiple clock domain hardware) for the duration of the simulation
- if you are simulating asynchronous hardware, you usually just set signals of interest directly without toggling a clock.

There are a couple of common testbench approaches:

- write a testbench directly in the HDL language(in this case VHDL) and simulate the testbench in a simulator for that HDL(in this case GHDL)
- modern hardware is extremely complex and basic HDLs(such as VHDL and Verilog) provide poor abstraction. It is becoming increasingly popular to use a high level language such as Python and C++ to drive simulations.

In this section, I describe simulating with the first approach listed above. In the document, `Speedy_Debugging_With_Cocotb.pdf`, you can learn more about how to write testbenches in Python that can automatically perform checks on your hardware.

Writing the Testbench

You don't need to write the testbench as it has already been provided, but you should read the following to learn about how it works.

You'll notice two files, namely:

- MIPS-SingleCycle.vhdl
- MIPS-Assignment-1tb.vhdl

The first file contains the implementation of the MIPS computer, and the second file is the testbench. Inside the second file, we instantiate a top entity `MIPS_tb` which is connected to and drives the `MIPS` entity, which is implemented in the first file.

The important lines in the testbench are:

```
-- reset the logic
logic_reset :process
    begin
    creset <= '1';
    wait for 120 ns;
    creset <= '0';
    wait;
    end
process;
```

and

```
-- Clock process definitions
clock_process :process
    begin
    clock <= '0';
    wait for clock_period/2;
    clock <= '1';
    wait for clock_period/2;
    end
process;
```

Both code snippet stanzas above are VHDL processes. These processes, unlike the processes found in the `MIPS` implementation, cannot be synthesized into physical hardware because they contain special keywords such as `wait`. These are called un-synthesize-able processes and are used to simulate hardware. In VHDL, all processes are executed simultaneously.

The first code-snippet stanza above starts the simulation by driving the `creset` signal high, and then waiting for 120ns before driving it low. The signal remains low for the remainder of the simulation because the of the indefinite `wait` command that comes after `creset <= '0';`.

The second code-snippet stanza above starts the simulation by driving the `clock` signal low, and waiting for `clock_period/2 = 50` before driving it high. This toggling action continues because unlike the first process stanza, there is no indefinite wait.

Running the Simulation

To run the simulation, do `$make sim` in your terminal. You should see the following output:

```
mkdir -p sim_dir
ghdl -i --workdir=sim_dir MIPS-SingleCycle.vhdl MIPS-Assignment-1tb.vhdl
ghdl -m --ieee=synopsys -fexplicit -o sim_dir/MIPS_tb --workdir=sim_dir/ MIPS_tb
```

```
analyze MIPS-SingleCycle.vhdl
analyze MIPS-Assignment-1tb.vhdl
elaborate mips_tb
cd sim_dir;ghdl -r MIPS_tb --ieee-asserts=disable --stop-time=500ns
--wave=MIPS_tb.ghw --vcd=MIPS_tb.vcd
./mips_tb:info: simulation stopped by --stop-time @500ns
```

To learn more about how the simulator and how makefiles(in general) work, go ahead and read `A_Deeper_Look_At_GHDL.pdf`

Changing the Simulation Duration

To change the amount of time the simulation runs for: you can do `$TIME=900ns make sim` If you don't specify a simulation duration before running `$make sim`, the simulation will only run for `500ns` by default.

Cleaning up Your Simulation Environment

Sometimes, you've done something really bad and perhaps broken a part of your simulation environment and for some reason, your VHDL won't compile and simulate. A common rule of thumb in software development is to cleanup and delete all the compiled files using the command `$make clean`.

Updating Your Simulation

In general, you do not need to run `$make clean` very often. In the process of developing your VHDL code, you will need to update your simulation. To do this, simply run `$make sim` again. It is typically only necessary to run `$make clean` if something is wrong with your simulation and you believe you may have corrupted an output file. If you do however run `$make clean` before, running `$make sim`, be aware that it will take longer to rebuild the simulation than simply running `$make sim` on its own.

Viewing the Waveform

- GTKWave is a popular FOSS waveform viewer available for Linux, MacOS, and Windows. We will be using GTKWave in class.
- Scansion is a free waveform viewer available only for Macs. It offers nice integration with Mac touchpad gestures, however, it has cannot view the content of VHDL memories, so GTKWave if the preferred waveform viewer for this class.

Installing GTKWave on Ubuntu or Bash for Windows

```
apt install gtkwave
```

By default, you cannot view the display in bash for windows. To remedy this, do the following in bash for windows :

```
export DISPLAY=localhost:0.0
```

You'll also need to download, install, and open Xming. Once you open Xming, it should start a process in the backgroung.

After you have already run `make sim`, you can now run `gtkwave sim_dir/MIPS_tb.ghw`.

MacOS

Download the GTKWave app for MacOS: Once the app has downloaded, drag the app from your `Downloads` folder to your `Aplications` folder.

After you have already run `make sim`, you can now run `open sim_dir/MIPS_tb.ghw`.

Adding Signals to the Waveform

After you open up the waveform with GTKWave, you will see a window like the following with 4 major panes.

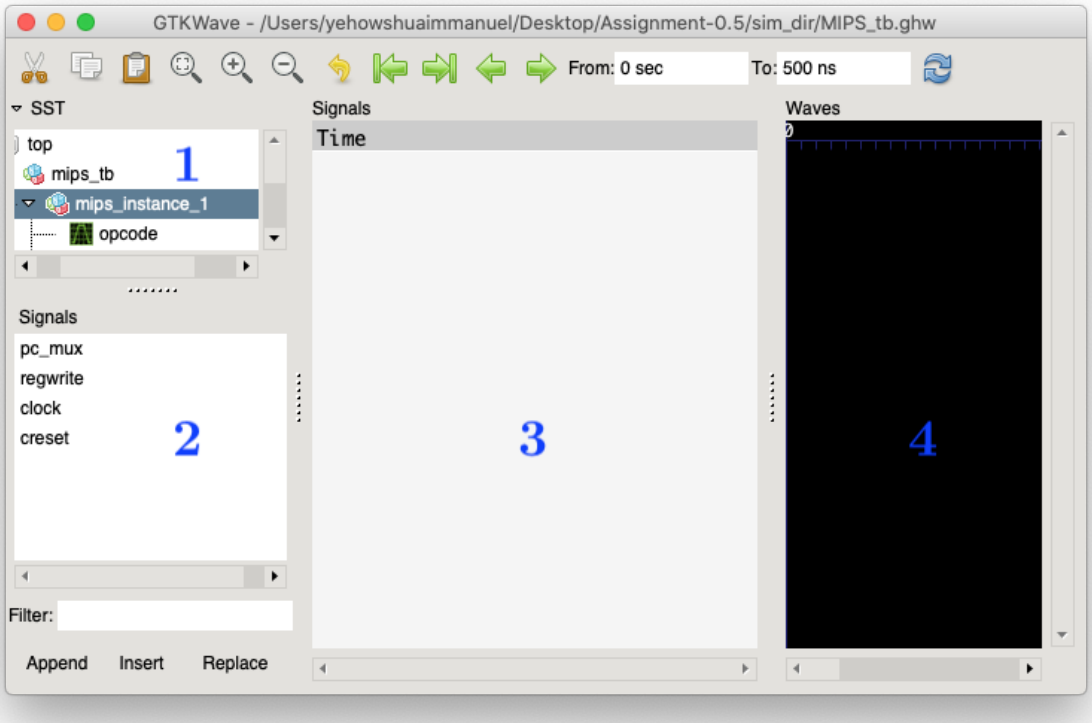

Figure 1: Default Simulation Window

`mips_instance-1` is currently selected in the figure above. Selecting this entity shows the signal names for the entities ports as well as some other signals GTKWave think might be important in pane 2.

Add `pc_mux`, `regwrite`, and `creset` to pane three by clicking on `pc_mux` and then shift+clicking on `creset`. Then click on the `append` button at the bottom of the window. You signals should now be inserted into pane 3 and their respective waveforms displayed in pane 4. You may need to zoom out on the waveform. The default timescale is `fs` but the testbench simulate inns“.

If you want to search for a signal, just type your term into the Filter panel. You can also import all the signals in a module at once into GTKWave. Just select your entity in pane 1, and **right click → recurse import → append**.

Refreshing/Updating the Waveform from Simulation

In the process of developing your VHDL code, you will need to re-simulation. After you rerun your simulation with `$make sim`, you do not need to re-open the waveform from the terminal, simply switch to the already open GTKWave window and click **File** → **Reload Waveform**.

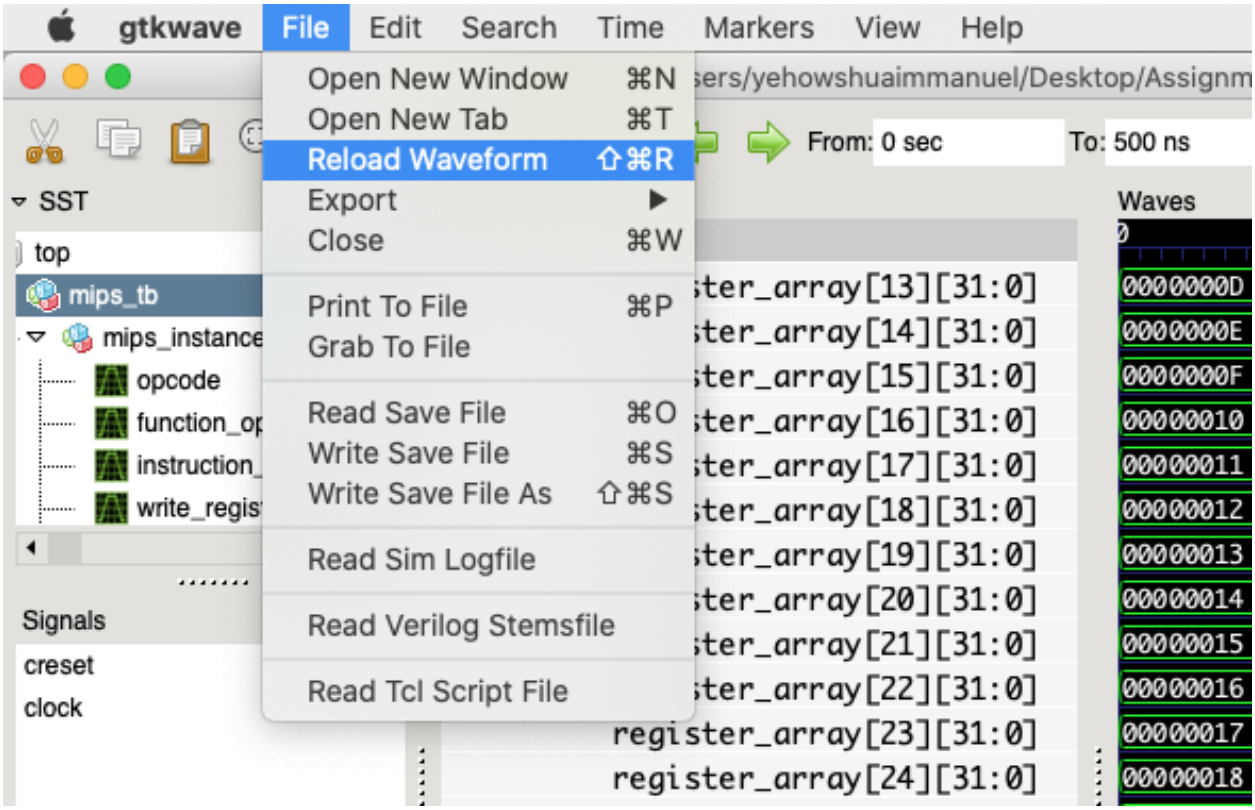

Figure 2: Updating the Waveform

1. You can launch gtkwave and open the file test_f.ghw
2. Or to launch gtkwave, type the following command. *gtkwave test_f.ghw*