EECS 470 Lab 1 Assignment

Note:

- The lab should be completed individually
- The lab check-off is due by **Friday, January 19th**

Your assignment for this lab will be to access the CAEN Linux environment and go through the Verdi tutorial for the two bit branch predictor module. You will then implement your own finite state machine for an `arbiter` module.

This document begins by copying the early labs/projects readme and continues with a tutorial of the Verdi debugger, a debugging example, and ends with the `arbiter` module assignment.

You will submit the lab by having a lab instructor check-off your work during lab or office hours, by placing yourself on the help queue when ready. Submit a photo of their signature (or a screenshot of a unique code if virtual) to Gradescope.

1 EECS 470 Early Labs/Projects Readme

This readme will be included in the first few labs/projects, and is a beginner's guide and quick reference for getting started in EECS 470.

It starts with a guide to accessing the CAEN Linux environment, gives a refresher on using the terminal and shell, introduces the EECS-470 GitHub organization with an SSH key guide and refresher, introduces our general build tools and our Makefile workflow, and ends by mentioning the autograder submission script.

1.1 Accessing the CAEN Linux Environment

CAEN's Linux environment is our workspace for verilog programming in 470. It gives us access to the Synopsys build tools and the Verdi debugger and is the only location where we can use these tools. Luckily, there are many options for accessing the environment:

1.1.1 Lab computers!

Lab computers are the simplest way to access CAEN Linux - and they don't require two-factor authentication (2FA).

Go to any CAEN lab (try searching at this link by checking "CAEN Workstations") and open a computer:

- If your login screen is for Linux, you're done!

  Use your uniqname and password to log in and access the Linux desktop, then open a terminal with Ctrl+T or by right-clicking the desktop and selecting "Open Terminal"

- Windows computers will require a few more steps:
  - Most lab computers are dual boot, so you can often reboot straight into the Linux desktop:

    Press Ctrl+Alt+Del at the Windows login screen and select restart, then use the arrow keys to choose Linux when the option to choose an OS comes up

  - Or you can connect to a remote Linux desktop inside Windows by using CAEN VNC:

    Log in to Windows, then search for "CAEN VNC" in the "appsanywhere" page that automatically opens. Click the green launch button, then the second big green launch button in the cloudpaging player window. It will open a small window asking for your log in. Enter it, then log in again to access the Linux desktop!

1.1.2 Remote login!

You can still log into the CAEN Linux environment if you're not on campus, however you will need to use two-factor authentication with Duo every single time you access, which gets annoying. These tutorials will show you how to access either the desktop through VNC or the terminal through SSH.

- Log in to a desktop via VNC
- Log in to a terminal via SSH
  - And don't forget to set up an ssh config file (see below)

1.1.3 Visual Studio Code!

VS Code is a common editor and offers a great way to access projects in 470. You can use the Remote-SSH extension to access CAEN over SSH from a VS Code instance running on your local machine. If you follow the linked tutorial above, CAEN is already ready for ssh with the hostname `login-course.engin.umich.edu` and your username will be your uniqname, so connect to the remote: `YOUR_UNIQNAME@login-course.engin.umich.edu`. Unfortunately this too requires 2FA on every login, but you can set up ControlPersist in your ssh config (see below) to speed up reconnecting.

1.1.4 Configuring SSH

Most students will eventually want to access CAEN remotely, and will probably use ssh to do so. Setting up an ssh config file can save you a lot of hassle over the semester. Below is a good default ssh config for CAEN, it lets you type `ssh caen` instead of `ssh YOUR_UNIQNAME@login-course.engin.umich.edu` and also keeps your ssh connection alive so you don't have to reconnect and redo 2FA if you connect multiple times in a row. You can use it by copying it to the `~/.ssh/config` file, or if you're using VS Code, search for "remote-ssh configuration" and add it there.

```
Host caen login-course.engin.umich.edu
    HostName login-course.engin.umich.edu
    User YOUR_UNIQNAME
    ControlMaster auto
    ControlPath ~/.ssh/_%r@%h:%p
    ControlPersist yes
    ForwardX11 yes
```

Figure 1: A default SSH config for CAEN

1.2 Using the Terminal and Shell

Once you can access the CAEN Linux environment, you can open a terminal and run some shell commands. Open the terminal by pressing Ctrl+T or right-clicking the desktop and selecting "Open Terminal"

We assume you have basic terminal and shell knowledge already, but very quickly, here's what to remember:

- Each line is a command with space separated arguments, use quotes for arguments that contain spaces: `grep EECS 470 Makefile` vs `grep "EECS 470" Makefile`
- Press the up arrow to reuse previous commands
- Press tab to auto-complete typed filenames (`grep 470 Makef<TAB>`)
- Use `pwd` to print your working folder/directory, `cd new_dir` to change your directory, `mkdir new_dir` to make new directories, and `touch new_file` to make new files
- Output files raw with `cat`, view them with `less`:
  `cat README.md` vs `less README.md`

- Redirect command output to files with >:
  `echo EECS 470 rules! > file.txt; cat file.txt`
- Pipe the output of one command as the input to another with |:
  `echo "EECS 370 was a lot of work" | sed -e s/3/4/ -e s/wa/i/`
- Press Ctrl+C to Cancel a running command
  Press Ctrl+\ to force Quit a running command if Ctrl+C doesn't work
- Finally, read manuals on any command with `man command`, and get quick help with `command --help`

The specific build tools used in 470 (Makefile usage, running testbenches) are gone over in detail below.

1.3 GitHub Organization Guide

Lab and project sources are managed by the EECS-470 GitHub organization.

We use the organization to distribute source code, manage autograder submissions for projects, manage group teams for the final project, and as a central location for instructors to view and manage your source files.

All students should have received an email invite to the organization. This will prompt you to create a University of Michigan GitHub account if you aren't using one already (your account name doesn't have to match your uniqname, which is the source of much complication in automatic course tools).

Once you've accepted the invite (and if you're reading this, you've probably already accepted it), you should be able to see repositories for the labs and projects that have been released so far.

You will download and manage these repositories in the CAEN Linux Environment (see the tutorial above). Once you have terminal access and have accepted the Organization invite, you can generate an SSH key for your UofM GitHub account.

1.3.1 Generating SSH keys for GitHub

If you've already used SSH keys from CAEN linux before, feel free to reuse those. This section assumes you haven't.

This section is modelled on the tutorial: Connecting to GitHub with SSH

If you run into issues, you can ask an instructor, but they're generally going to refer back to that link. Generating SSH keys can be annoying, but hopefully this guide is easy-enough to follow.

Start in the terminal in CAEN Linux. Run the following commands:

```
ssh_key_file=~/.ssh/eecs470_github
# Your caen username is your uniqname
ssh-keygen -t ed25519 -C "$USER@umich.edu" -f $ssh_key_file
# Start the SSH agent (this might be redundant)
eval "$(ssh-agent -s)"
# Add the key
ssh-add $ssh_key_file
# Output the public key for copying:
cat $ssh_key_file.pub
```

Figure 2: Commands to generate an SSH key

The first command will prompt you to add an optional password.

The last command will output the public key to the terminal for copying.

You should then copy the key and upload it to your github account. Open github.com, log in, then navigate to Settings, SSH and GPG keys. (or run this command to open firefox at the website)

```
firefox https://github.com/settings/keys
```

Click 'New SSH key' and add your copied public key and a title (i.e. "CAEN 470 key").

On the key, click the "Configure SSO" dropdown and authorize the EECS-470 organization.

Finally, update your SSH configuration to use the new key. Add the following lines to the file `~/.ssh/config` (create it if it doesn't exist):

```
Host github.com
    ForwardAgent yes
    IdentityFile ~/.ssh/eecs470_github
```

Figure 3: A default SSH config for GitHub

Test that the key is added with: `ssh -T git@github.com`.

Test that the key is authorized and working by downloading a git repo.

1.3.2 Download git repos

With a newly added SSH key, you're ready to download your repositories!

On a repository in GitHub you can click the green "`<> Code`" dropdown to view the SSH link for cloning a repository. You can then use the `git clone <SSH_link> <directory>` command to clone the repo.

Example for a lab1 repo:

```
# Optionally make a folder for organizing repos first
mkdir eecs470; cd eecs470
# Clone your lab1 repository from Fall 2023
git clone git@github.com:EECS-470/lab1-f23.$USER.git lab1
```

Figure 4: An example for cloning the lab 1 repository

1.3.3 Basic git usage

We assume you've used git somewhat in the past, but here is an opinionated set of commands to remember:

- Clone a repo to a directory:
  `git clone <SSH_link> <directory>`
- View file edits:
  `git diff`
- Add edited files to your workspace:
  `git add <file1> <file2>`
- Commit added files:
  `git commit -m "<message>"`
- View a log of commits:
  `git log --oneline --graph --branches --max-count=10`

- Create and checkout a new branch:
  `git checkout -b <branch_name>`
- Upload a new branch to GitHub:
  `git push --set-upstream origin <branch_name>`
- Push commits to GitHub:
  `git push`
- Merge branches: Create a pull request on GitHub. Yes I'm serious.
- Get the most recent updates from GitHub:
  `git pull --ff-only`

1.4 Build Tools in EECS 470

In EECS 470 we're generally executing one of three types of commands in CAEN:

1. Compiling verilog testbenches with simulated and synthesized modules
2. Running the compiled executable
3. Debugging in Verdi, our verilog debugging environment

Here's our general workflow:

We start with a verilog testbench and module(s) it tests.

We compile these under simulation to an exectuable file that runs the testbench and prints whether the module passes or fails. If it fails, we use Verdi or display statements to inspect the output and iterate on either the testbench or the module until we're satisfied.

Once the testbench and module work in simulation, we change our compilation step. We synthesize the verilog modules against actual device cells representing structural hardware connections. We only synthesize the module, not the testbench, as the testbench may include features like *printing*, or *reading files* that don't exactly translate to raw silicon.

In synthesized modules, we generally focus on timing rather than area, and use the **slack** metric: the clock period minus the longest signal path from one register to another. If slack is negative, a signal might not reach its next register by the clock edge, leaving incorrect values floating and causing potential *undefined behavior* (scary!).

So when we synthesize modules, we check the slack of the compilation, and either increase the clock period or change the design if we find violations. When we're satisfied with the synthesis, we can compile the synthesized module with the original testbench to produce a new executable that we can debug again until it passes.

1.5 Using Makefiles in EECS 470

To manage this workflow, we use *GNU Make* as a build automation tool. We specify targets with dependencies and set variables in the special file "`Makefile`". At the shell, we run `make my_target` to compile or run a target and all of its dependencies. Make also only rebuilds targets or dependencies if their source files are newer than the previous build.

From the workflow above, we name our normal simulation executable `simv`, and our synthesis executable `syn_simv`. To build these, we need to specify our source and output files in these Make variables:

- `TESTBENCH` – Our verilog testbench for a given module(s)
- `SOURCES` – The source files for the module(s) used by the testbench
- `SYNTH_FILES` – The file(s) we'll use when compiling for synthesis

For simulation we compile `simv` directly from `SOURCES` and `TESTBENCH`. For synthesis we first synthesize `SOURCES` to `SYNTH_FILES` and then compile `syn_simv` from `SYNTH_FILES` and `TESTBENCH`.

We won't go over the specific build commands in detail, but our verilog compiler is `vcs`, which we run with many many arguments, and our synthesizer is `dc_shell`, which we run with a synthesis script: `470synth.tcl`. The script reads environment variables for source files and other configuration and defines the constraints that set up our timing requirements.

To use the Makefile, run `make my_target` in the shell when in a directory with a "`Makefile`" file. Here is the reference table of all standard targets in EECS 470 makefiles:

```
make sim       <- execute the simulation testbench (simv)
make simv      <- compiles simv from the testbench and SOURCES

make syn       <- execute the synthesized module testbench (syn_simv)
make syn_simv  <- compiles syn_simv from the testbench and *.vg SYNTH_FILES
make *.vg      <- synthesize the top level module in SOURCES for use in syn_simv
make slack     <- a phony command to print the slack of any synthesized modules

make verdi     <- runs the Verdi GUI debugger for simulation
make syn_verdi <- runs the Verdi GUI debugger for synthesis

make clean     <- remove files from testbench runs and compilations (not from synthesis)
make nuke      <- remove all files created by the makefile
make clean_run_files <- remove per-run output files
make clean_exe       <- remove compiled executable files
make clean_synth     <- remove generated synthesis files
```

Figure 5: Table of EECS 470 standard Makefile targets

1.6 Submitting Projects to the Autograder

Projects in EECS 470 will be submitted to the autograder with the public autograder submission script in CAEN. Run the script from CAEN:

```
/afs/umich.edu/class/eecs470/Public/470submit project_number
```

This will have the autograder download your graded files from the 'main' branch of your GitHub repository and run the autograder.

You will then receive an email with a summary of the public output of the autograder. This will only contain short descriptions of any errors, and is not representative of your final grade.

You should email the instructors to debug any issues with the autograder.

2 Verdi Debugger Tutorial and Synopsys Tools Intro

Now we will start the tutorial of the Verdi debugger and an intro to our two main Synopsys tools. This section will ask you to edit files and run commands, but can be rather long and detailed. So follow along, but feel free to skip ahead and reference previous sections later if it gets difficult to follow.

First, a quick overview of the two main Synopsys tools.

2.1 The Simulator: `simv`

Verilog, as a hardware description language, cannot be "compiled" in the same way that C or Haskell might be compiled. The way we test or run a design once it has been implemented in such a hardware description language is to simulate it. The easiest way to do this simulation is to build a software simulator automatically, which is what the Synopsys VCS tool does. It takes in a Verilog design description you've created and builds a simulator in C++ that changes values the way you've described and prints appropriately. This is generally used as a first pass to make sure that the design works the way that is intended.

It's time for you to try this out. In the directory you've extracted all the provided files, run the command: `make sim`.

This will have resulted in an error message. What went wrong? It turns out that we have a syntax error on a line of `two_bit_pred-a.sv`: `loggic` should be `logic`. Fix the error (open the file in your choice of editor, see above), save the file and then run `make sim` again. This time you should see the output of the `$monitor(...)` call in `two_bit_pred_test.sv`. The Make rule to simulate also redirects the output into a file called `program.out` (this is explained in more detail in Lab 5). This is particularly useful when you have a lot of output.

2.2 The Synthesis Tool: `dc_shell`

In this class we require that your hardware designs really represent hardware and we judge your final project on the overall speed of your design. To accomplish this you will need to synthesize your design. The synthesis tool attempts to create an actual circuit level implementation of your Verilog design description. This circuit level design is actually a Verilog file itself, but it is structural Verilog and uses a library of standard cells that another tool can lay out on a real chip. One benefit of the output being a Verilog file is that we can simulate the circuit level design in the same way we simulate your behavioral design.

Thus we can test your synthesized design and if it does not behave properly your design is not considered to be synthesizable and is therefore incorrect (though we do give partial credit).The clock speed of the circuit that it generates will be the clock speed we use for your design. We won't go into great detail about synthesis in this tutorial but we will show you the basic commands. We provide greater depth in Lab 2.

We interact with the synthesis tool, Synopsys Design Compiler, through a script of options and commands written in the Tool Command Language, which is abbreviated Tcl (`*.tcl`) and pronounced either T-C-L or like the word "tickle." We have provided you with a synthesis script called `470synth.tcl`.

The clock period is set in the Makefile and passed to the tcl script, Ctrl+F for `CLOCK_PERIOD` to change it. A higher clock period will yield faster synthesis times, while a lower one will take longer because it requires more effort on the part of the synthesis tool. A clock period set too small may not even be possible. We will explore this design constraint in Project 2.

Run `make syn` to synthesize and simulate the synthesized design. This will produce four files. Please open them and read them.

1. `two_bit_pred.chk` – This file contains errors if something went wrong, otherwise it will be empty (or occasionally contains a single number; you can ignore that).
2. `two_bit_pred.ddc` – This file contains the proprietary Synopsys representation of the synthesized design. It can be included like a Verilog design file in synthesis, either to be optimized again or as a black box.

3. `two_bit_pred.rep` – This file contains the timing report for your design. We will cover it in greater detail in Lab 2.
4. `two_bit_pred.vg` – This file contains the structural Verilog that the synthesis tool generated for your design and design constraints.

2.3 The Debugger: `verdi`

This simple error was easy to find and fix, but many (most) errors are not. For the more complicated errors, Synopsys provides a debugging tool, called Verdi, for hardware description designs. This debugger looks much more akin to the debugger you used in 270 (or your equivalent undergraduate digital design course) than the ones you would have used in 280/281 (or your equivalent undergraduate programming courses).

Note:

- An alternative option to Verdi is the GTK Waveform debugger. Verdi is a more powerful tool, but can be more slow when used remotely. You can reference this GTKwave Waveform Viewer tutorial if you would like to use it, although not all instructors are experienced with the tool.

The command to run the debugger is in another Make rule, and you can run it with the command: `make verdi`. Run that command now. This should open a window, which we will now refer to as the Verdi window. We will now walk through the default panes of the Verdi window, but you can add more panes yourself through the Window » Window Manager... menu.

2.3.1 Module Instance Pane

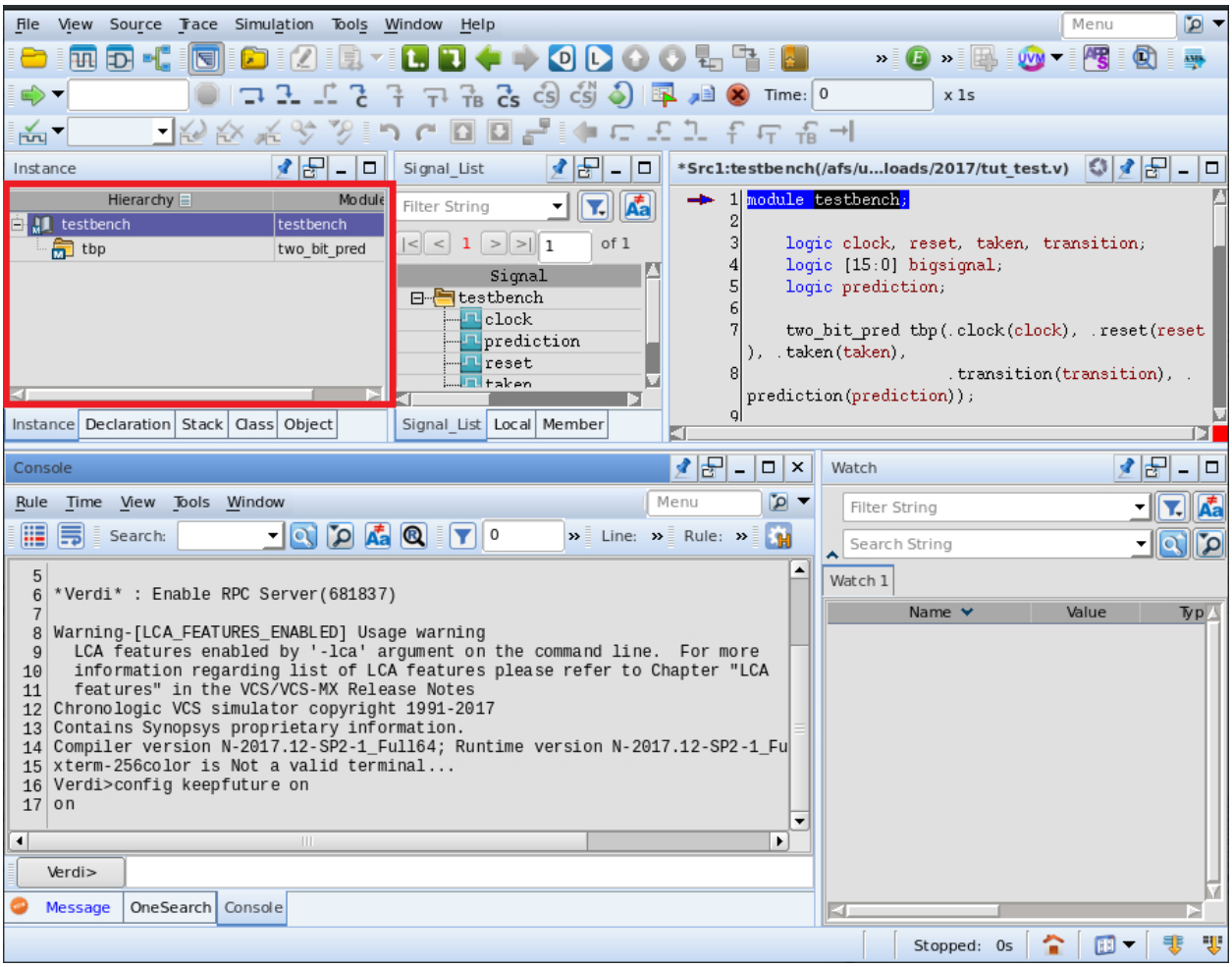

Figure 6: The Verdi Window with the Module Instance Pane highlighted

In fig. 6, the pane on the far left of the Verdi window is the Module Instance Pane. It contains the names of the modules you might be interested in, grouped hierarchically. You can use this list to select signals from a particular module you want to look at in other views and panes.

In our tutorial, we have a testbench which instantiates another module. Expand the testbench (click the plus icon to the left) in the pane to see the submodule. You'll notice that the submodule is called `tbp`, which is not the name of the `two_bit_pred` module, but rather the name of the instantiation in the testbench. Make sure to use meaningful names for instantiations when you implement your own designs.

2.3.2 Signal List Pane

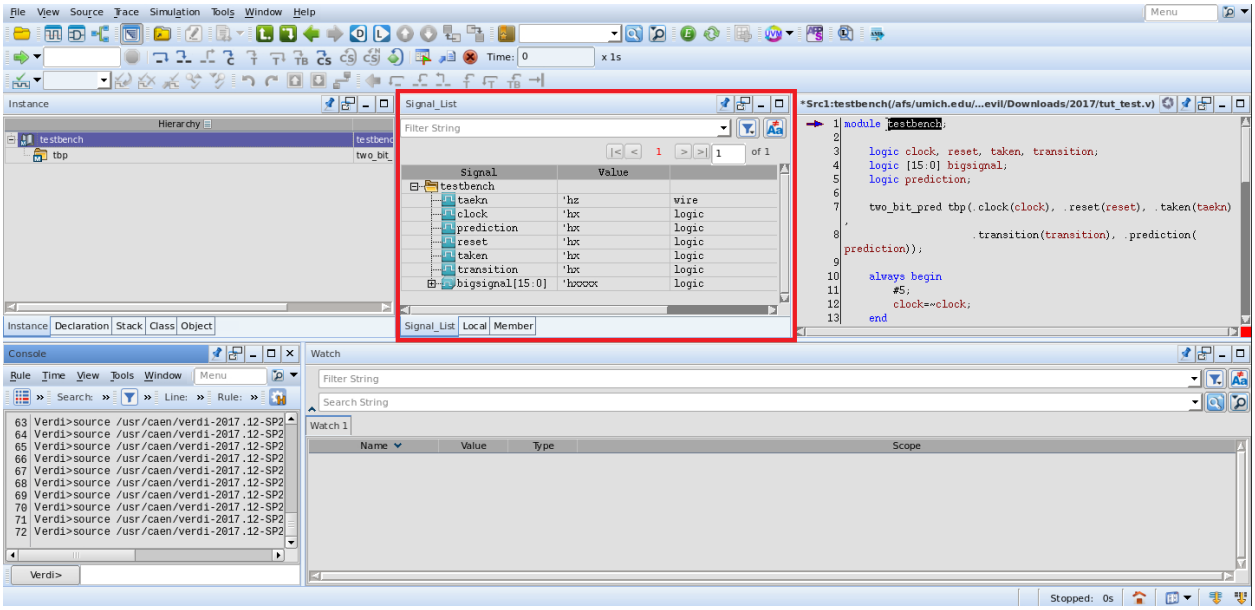

Figure 7: The Verdi Window with the Signal List highlighted

In fig. 7, the signal list pane is highlighted. You can add this pan through the Window » Window Manager... » Signal_List » Activate » Close menu. Double click on a module in the module instance pane to lists the signals in that module. From the Signal List pane, select the signals whose waveforms you would like to view. You can select multiple signals by holding down ⇧ Shift and clicking to select a group or holding down Ctrl and clicking to select multiple individual signals. Select all the signals available in the testbench and right click on the highlighted group, then go to Add to Waveform » New Waveform to open the waveform viewer. If, at some point later on, you need to add more signals to the waveform viewer, you can do so by selecting them here and selecting Add to Waveform » Add to [wave name].

2.3.3 Waveform Viewer

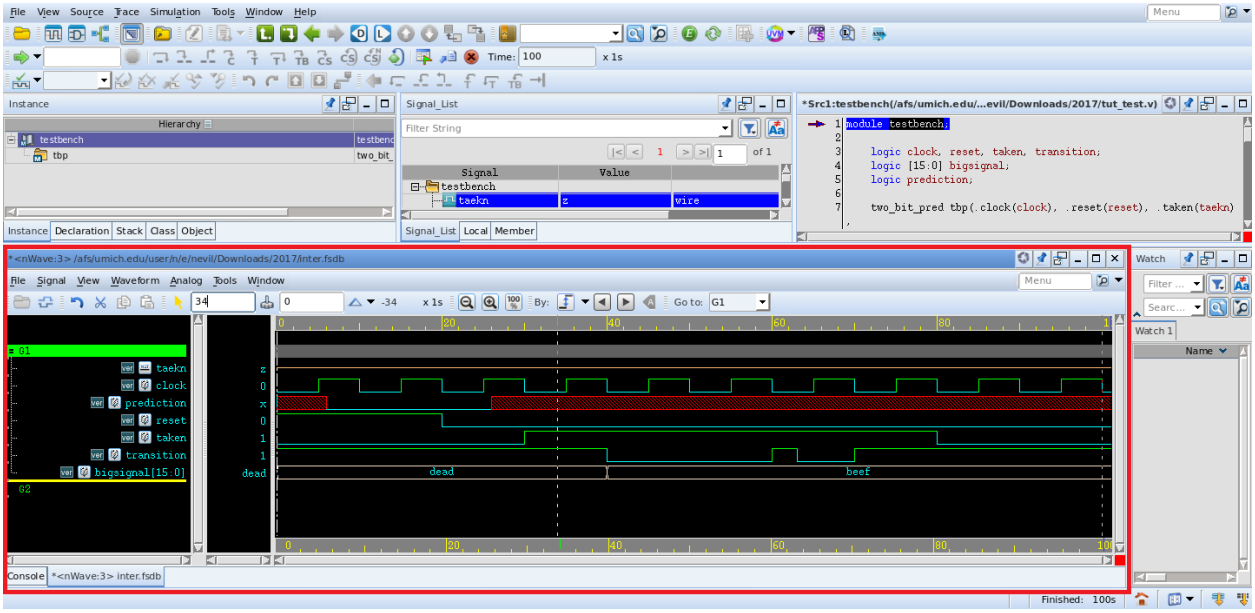

Figure 8: The Waveform Viewer Window

You will see a new window that shows waveforms, as in fig. 8. You need to go to Simulation » Run/continue or press F5 to start the simulation and populate the waveform viewer. At this point, if you added additional signals to this view, they would have no waveform, until you reran the simulator. This can be done by Simulation » Restart then Simulation » Run/continue or by pressing F5 twice.

The waveform viewer is intended to facilitate debugging by showing you how several signals change in relation to one another over time. The pane on the left hand side of this window lists the names of the signals currently being displayed. The right hand pane shows the signals themselves. At the bottom of this window there is a scroll bar which moves through time, when zoomed in to see specific signal transitions. Move to the right to see how the signals in the module change. Now, try zooming in and out, as well as getting all of time on the screen at once by going to View » Zoom or clicking one of the magnifying glass buttons at the top.

Signals in the waveform viewer can be displayed several different ways, largely related to the 4 state logic used in Verilog. The "good" signals `0` and `1` are shown as green signal lines with transitions between the two values as appropriate. The "bad" signals `X` and `Z` show up as a red block where the signal should be and a yellow line half way between `0` and `1`, respectively. Both of these can be seen in the provided module. Try to find them now. The signal `prediction` is an example of the unknown value, `X`, after time 26, and the `taekn` signal is an example of the unconnected value, `Z`. These all apply to 1-bit signals. When we have a larger bus, values are displayed with their hexadecimal value with marks where the value transitions. If any part of a bus is `X`, it will show the whole thing as `X`, and similarly for `Z`, it will show the hexadecimal value with the unconnected portions marked with a `z`. If you would rather have buses display their value in some other base, you can right click on the signal name in the left hand pane and choose the Radix option. You can also expand the bus into all its individual signals by double clicking on the signal name.

You can click on the waveform itself to drop a marker, which helps line up values across many signals at the same time. At the top of the waveform pane, once you've placed a marker, there is a number corresponding to the time you've set the marker at. You can use the options available to advance forward by an amount or to the next clock tick.

Signals can be dragged around using the middle button of your mouse. You can also remove signals by highlighting them and then pressing the delete key. You can also add custom signals that are a logical

operation of other signals by going to Signal » Logical Operation....

2.3.4 Source Pane

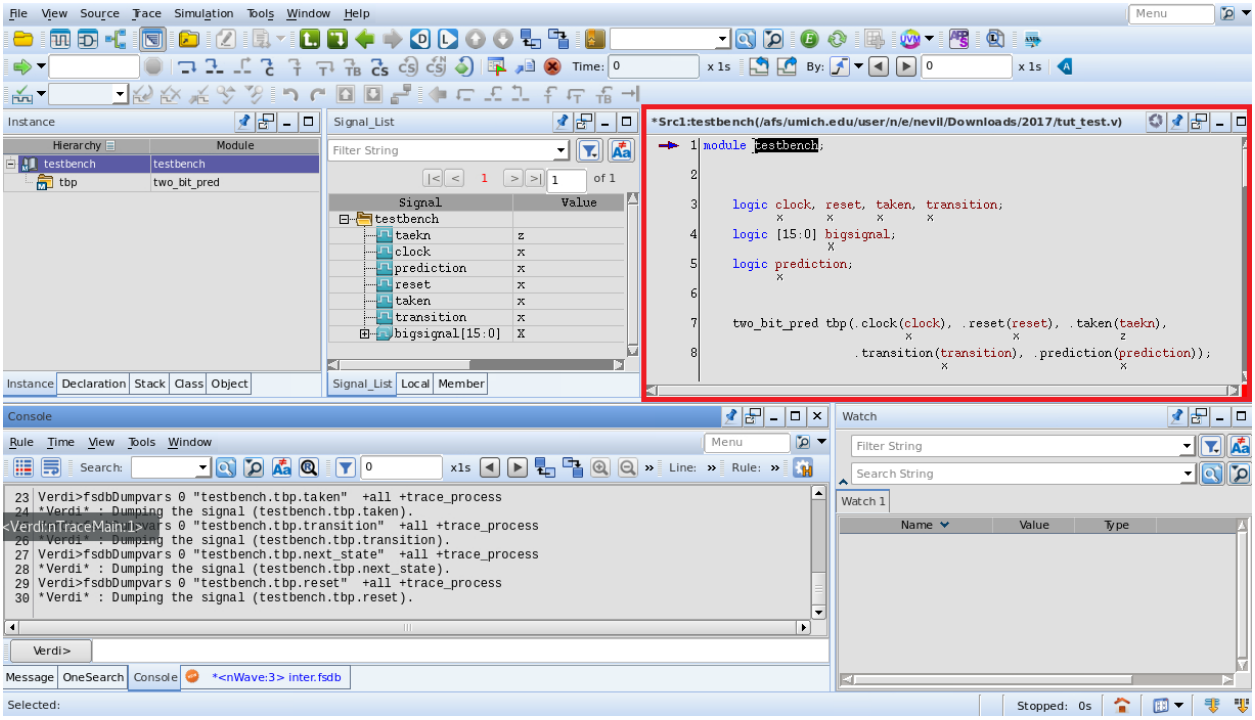

Figure 9: The Verdi Window with the Source Pane highlighted

The source pane is by default on the right hand side of the Verdi window, and is highlighted in fig. 9. It shows the Verilog source for the module you're currently examining. This allows you to examine what exactly is generating a signal. It can also let you double check that the file you've just edited to fix a bug is the one you're now compiling/simulating. To change the file being displayed, drag a module (using middle button of your mouse) from the instance pane over to see it. Similarly, dragging a signal name over to the source pane to highlight its definition. To view the current values of signals from the simulator in the source pane, right click on the source pane and select Active Annotation. You should now see values next to each signal.

2.3.5 Schematic Pane

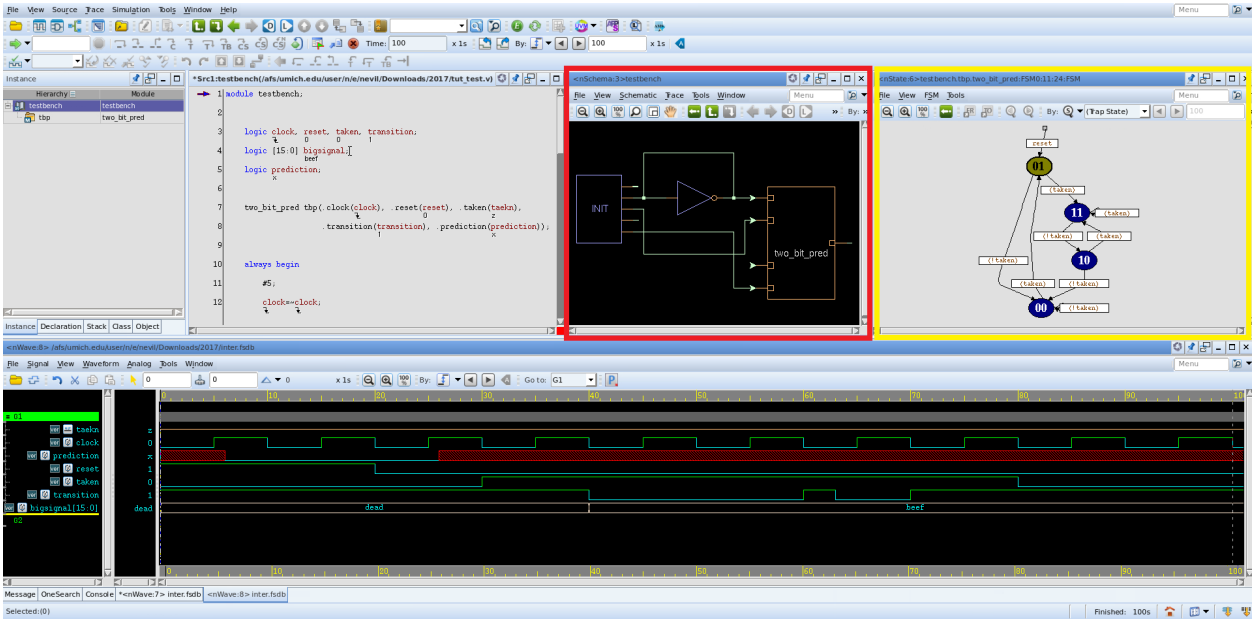

Figure 10: The Schematic Viewer

Figure 10 shows the schematic viewer. To open schematic go to Tools » New Schematic from Source » New Schematic. This can be useful in tracking where signals are coming from after synthesis. In the schematic pane, double click on `two_bit_pred` and then double click on `state`. You will see a FSM built as highlighted in yellow in Figure 10. Turn on View » Transition Condition to see the logic that causes state transitions. Similarly, you can turn on State Action and Transition Action. In this lab assignment, you will be asked to build a Moore state machine. The FSM state diagram generated through Verdi schematic window should looks similar to the state machine in the assignment.

3 Debugging Example

Now that you've learned about many of the features of Verdi, it's time to use it to debug the module you've been given.

The output of the `two_bit_pred` module is the `prediction` signal. This signal goes to `X` at time 26 and stays there for the rest of the simulation. Why would that happen? Let's figure it out.

For a signal to be unknown, one of its drivers must also be unknown, which if we follow this logic down all the way means that one of the inputs somewhere is unknown. So, we need to start by looking at how `prediction` gets set. The source for `tbp` should already be open, so let's use that to trace back to where the problem is. Let's also add the `tbp` module signals to the waveform viewer so we can see the signals in that module. Looking at the source, we can see that `prediction` is simply set equal to a bit of signal `state`. Looking at the waveform viewer, we see that, as expected, `state` is unknown at the same times. Now we want to check the signals that define `state`. We can see that `state` either takes on a known value of `01`, or it takes on the value of `next_state`. So again, now we need to trace back to which values set `next_state`. Looking at this, we can see that `next_state` is determined by `state` and `taken`. We already know state is broken, so we need to look at `taken`. On the waveform viewer, we see that `taken` always has a value of `Z`, unconnected. Because `taken` is an input into this module, we know the problem is in how the testbench is connecting to this module. Now let's drag the `testbench` module into the source viewer so we can look for the problem. Looking at where `tbp` is instantiated, we can see that there was a typo. `.taken(taekn)` should have been `.taken(taken)`. The compiler assumed `taekn` was just a wire not being driven and although it

gave a warning, it did not give a compiler error on this. These little bugs can often be very annoying to deal with.

Now that we've found the bug, we want to save the waveform viewer configuration so we can just easily reload them when we run the waveform viewer again. It may not seem to be a big deal right now, but eventually when you're working with hundreds of signals, only a few of which are related to a problem you're trying to debug, it can be very annoying to try to find and place them on the waveform viewer every time. To do this, on the waveform viewer go to File » Save Signal... . Type in a filename, like `signal.rc` or, in the future, any name you want. It will be useful to keep the names descriptive as there may be different sets of signals to look at for different parts of a module you're testing. Now exit Verdi.

Use your favorite text editor to fix that typo in `two_bit_pred_test.sv`. When you're finished, type make clean and then `make verdi` to recompile and reload the waveforms so we can double check that it's working properly now. Now on the Verdi window, go to Tools » New Waveform... » File » Restore Signal. Select your `signal.rc` file and press OK. The waveform viewer, complete with the original signals, should appear. Run the simulation again. Here we can see, that everything is working properly now.

Now, let's find another bug. Open up the provided `Makefile` and change the value of the `SOURCES` variable to `two_bit_pred-b.sv` This module has an infinite loop, which requires one more option in Verdi. First, run `make clean` then `make sim`. When you've noticed that it hangs without printing anything, kill the job with Ctrl+\ . Now, run `make clean` to remove the files the last command created and then run `make verdi` to start debugging the infinite loop.

The Verdi window should appear. Now let's find that infinite loop. Add the signals to the waveform viewer. To find out where it is, run the simulation. The waveform viewer should hang and a red dot in the toolbox above should get enabled. Now click on Simulation » Stop. You can now see that the simulation was hung at time 25. The problem must be around here somewhere. We are now at time 30. Now let's find out where we're looping. Click on Simulation » Step/Next » Next a few times and look at the tbp source code. We seem to be going between `two_bit_pred-b.sv:17` and `two_bit_pred-b.sv:18`. That means that lines 17 and 18 of `two_bit_pred-b.sv` are causing the problem. We now know that the problem is in our module (not our testbench). In the tbp source, right click and select Active Annotation to monitor the values of each wire. We want to see how we got to this state. If we look at the source window, we can see that lines 17 and 18 are assignments to `loop1` and `loop2`. If we click next a few times and watch the values, we can see that both `loop1` and `loop2` keep changing even though no time is actually passing. This is a sure sign of circular logic. Since `loop1` is dependent on `loop2`, and `loop2` is dependent on `loop1`, the circular path is fairly obvious in this case. We now have enough information to fix the bug, but in this example, there's nothing to really fix, since the `loop1` and `loop2` variables are not used for anything, so we can just leave the design alone and finish up.

Similar to debugging forward simulation, Verdi also supports debugging execution in reverse. This allows you to go back in time without setting checkpoints and avoids overstepping during debugging. To enable, go to Tools » Preferences... » Interactive Debug » Reverse Debug, enable Reverse Debug and click OK. To use reverse debugging, go to Simulation » Reverse Debug.

This concludes our introduction to Verdi, but there are a number of other useful features such as tracing signal values and searching across the design. Feel free to try and explore other things.

4 Assignment

For the lab assignment, you will write, test, and debug a simple design from scratch. We will be showing you how to do each of these steps in greater detail in future labs. For now, we will be largely copying from `two_bit_pred-a` and giving you specific steps to follow.

4.1 Design

The design we will be implementing is a state machine for an "`arbiter`" module which decides for two requests, A and B, which to grant control. We'll use our testbench to simulate these requesters, and will test the grant outputs of the module.

"`arbiter`" has the following module definition and state machine:

```
module arbiter (
    input clock, reset,
    input req_a, req_b,  // request lines
    output gnt_a, gnt_b  // grant lines
);
```

Figure 11: Definition for the `arbiter` module

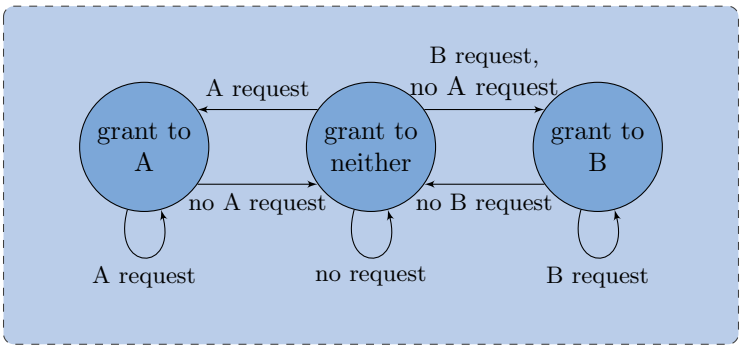

Figure 12: The Moore-type state machine for our arbiter

This module receives two requests, and has three possible outputs: grant to A, grant to B, and grant to neither. It will grant to A if both A and B request, and will continue granting to it's current grantee until they stop requesting, where it will have one cycle of granting to neither before granting again.

We can model this in verilog with two bits for the state, and we can test the output grant lines by checking multiple scenarios, described below.

4.2 Implementation

Go through the following steps to implement `arbiter` from a copy of the `two_bit_pred` files.

Copy the following files:

- Copy "`two_bit_pred-a.sv`" to "`arbiter.sv`"
- Copy "`two_bit_pred_test.sv`" to "`arbiter_test.sv`"

Make the following changes in `arbiter.sv`:

1. Change the module definition to match the definition in fig. 11.

2. Add assign statements that assign to both `gnt_a` and `gnt_b` based on bits of `state`.
3. Update the first `always_comb` block to use only three states, and set the `next_state` based on our input requests and the current `state` with reference to the state machine diagram. (also add a "default" at the end of the case statement which does nothing, to avoid creating a latch)
4. Update the final `always_ff` block to remove the condition on `transition` and to set the reset state to "grant to neither".

Make the following changes in `arbiter_test.sv`:

1. Update the testbench variables to contain:
   `logic clock, reset, req_a, req_b, gnt_a, gnt_b;`
2. Update the module instantiation to instantiate the arbiter.
3. Update the monitor statement to print out the above variables.
4. Update the initial variables to set `reset` to 1 and set all of `clock`, `req_a`, and `req_b` to 0.
5. Set reset to 0 after two negedges.
6. Add your own tests by updating the request variables. We will test by simply viewing the grant output from the monitor statement or by opening Verdi to view the waveform.
7. Add at least the following test cases:
   - Granting none, only A is requested
   - Granting A, only B is requested
   - Granting none, both A and B are requested
   - Granting none, only B is requested
   - Granting B, only A is requested

In the Makefile, edit the `TESTBENCH`, `SOURCES`, and `SYNTH_FILES` variables to match the arbiter files. Run the testbench in Verdi ("`make verdi`") and view the output of the two grant lines to verify your module.

You may also test the module under synthesis, although this isn't required.

4.3 Submission procedure

In EECS 470, labs are submitted by in-person check-offs during lab or office hours. You will place yourself on the office hours help queue, and an instructor will come by to do the check-off. They will talk to you about the design and have you show them certain files and explain some of the concepts you learned.

The lab instructor will give you their signature with the date, or if virtual, a per-student per-lab unique check-off code. You will turn in a photo of the signature or a screenshot of the code to Gradescope as your submission.

4.4 Submission

For lab 1, be ready to show that you've downloaded the git repo successfully, can run the `two_bit_pred` testbench with "`make sim`", and can open Verdi to show the output of your arbiter module on its testbench.

Place yourself on the help queue during lab or office hours once you're confident you've completed the lab satisfactorily. Turn in your check-off to Gradescope by the end of the day of next week's lab.