Tools

The purpose of this section is to get some experience of different tools used when developing, testing, and debugging digital designs.

1. Compile and install the Icarus Verilog simulator
2. Compile and run a Verilog simulation
3. View Verilog test-bench output as waveforms
4. Add assertions to a test-bench
5. Add print statements to a test-bench

Note that you are completely free to use ModelSim or other tools during your development (possibly you used it in 1st year), but in the assessment environment for the CPU deliverable we will use Icarus Verilog.

Installing a Verilog simulator

The simulator we will use is called Icarus Verilog. This is a free and open-source simulator, available under the LGPL license. It is purely a command-line simulator, and can be compiled and used under multiple operating systems including Linux, OSX, and Windows.

While Icarus can be installed via `apt get Verilog` under Ubuntu, this will not get the latest version, and you may encounter unsupported features or bugs. In particular, you may find that the Ubuntu versions don't support `always_comb` and `always_ff`. It is recommended that you compile and install it from scratch. This both ensures that you have the latest version, and also shows you the standard process for compiling and installing many software packages.

The main installation steps for Ubuntu (e.g. 20.04) are:

1. Clone the Icarus Verilog git repository and enter the directory:

   ```
   git clone https://github.com/steveicarus/iverilog.git
   cd iverilog
   ```

2. Install pre-requisite libraries and packages:

   ```
   apt get gperf autoconf bison gcc make
   ```

   This should be enough for Ubuntu 20.04, and probably other recent Ubuntus and other distros., though there may be some missing libraries that get reported in the next steps.

3. Use GNU autotools to generate a configuration script:

   ```
   sh autoconf.sh
   ```

   The output will be a script called `configure`, which is used to adapt the compilation process according to the current operating-system and

development tools. This step is part of what allows this software (and much other open-source software) to compile on a very wide range of unix-like platforms.

4. Run the configuration script:

   `./configure`

   This script will perform a number of checks on your environment, such as determining what kind of operating system it is, whether it is 64-bit or 32-bit, what libraries are installed, where header files are located, and many other things. The output of this will be a number of files describing exactly how the software should be built (compiled and linked) for your specific system.

5. Build the software using the tool `make`:

   `make`

   Makefiles are a common way of managing the compilation of software, particularly when multiple source files are being compiled, and when other tools need to be run as part of the build process. Describing the build steps as a makefile provides a number of advantages, such as minimal rebuilds that only recompile files that have changed, and parallel compilation across all your CPU cores. The ability to use other tools as part of the build process is also important for complex software, as it may be necessary to generate source files or analyse input files. During all the compilation messages that flash pass you might notice the build process call the `bison` parser generator, which is a tool that you will most likely want to use next term when writing your compiler.

   Once this process has completed, you'll see large numbers of `.o` files which have been compiled, and also a number of binaries have been built, such as `driver/iverilog`. However, the `iverilog` binary is not yet installed and in the search path, so if you try to run `iverilog` from any other directories it won‘t work.

6. Install the compiler:

   `sudo make install`

   This will install the binaries, support files, and other parts of the packages into standard system directories. For example, it will install the `iverilog` binary into `/usr/local/bin/`, and install the man pages (i.e. documentation) into `/usr/local/man`. At this point you should be able to run the program using the command `iverilog`, and get documentation on it using `man iverilog`.

This process involves a number of steps, and some stages such as compilation take a bit of time. If instead you were to perform `apt get Verilog` it would download pre-compiled binaries and data files, then install them directly. The

pre-compiled binary packages are much more convenient to use, particularly because distribution maintainers like Ubuntu take care to make sure that all their binary packages work together. However, you will often find the need to compile packages from source, for a number of reasons:

1. The packaged binary may be too old, and lack recent bug-fixes or features.
2. The packaged binary may be too *new*, and no longer support data files you already have or the features you need to use.
3. You may need to specify specific configuration flags, in order to disable or enable specific features at compile-time.
4. Often there is just no binary package available for a piece of software.

Compile and run a Verilog simulation

Just like software needs to be compiled before it can be executed, hardware simulations also need to be translated into a low-level form before they can be executed. The exact form of the simulation "binary" depends very much on the simulation tool, with each simulator having its own compiled representation. However, there are a number of common steps when simulating Verilog, VHDL, or any other hardware description language.

The files `and_gate.v`, `not_gate.v`, and `and_not_testbench.v` each describe Verilog modules, and we wish to simulate them as a single system.

-> Open the files in a text editor and look at the modules.

You should be able to see that one module *instantiates* the other two modules, i.e. it creates and contains instances of the modules. This defines the hierarchy of the system, which is a tree of module instantiations with parent-child relationships. The outer-most module at the root of the tree is not instantiated by any other module, and is called the *top-level module* (sometimes called "root module"). During simulation there must be a single top-level module representing the outer-most level of the design.

The top-level module is conceptually similar to the idea of `main` as the outer-most function in the tree of software calls in C++. But the analogy is not perfect:

- The module hierarchy is static, and cannot be changed at run-time. Once the module is synthesised to hardware, the hierarchy represents the fixed set of hardware resources dedicated to the design.
- The tree of function-calls made from `main` in software is dynamic, and may change every time the program runs according to the input data.

In some cases the simulation tool will infer the top-level module, but sometimes it may be necessary to explicitly identify the top-level module using simulator specific flags.

The main steps needed in an RTL simulation flow are:

1. **Compilation** : the input Verilog source files are passed, and the basic syntax is checked. In some tools multiple Verilog files are compiled independently, just like you might compile multiple C++ files into object files. In other tools compilation is combined with the elaboration stage.
2. **Elaboration**: the complete module hierarchy is built, with Verilog modules instantiated, and the links between them resolved. This is a bit like linked multiple object files into a binary, but there are a larger class of errors that can occur at this stage in hardware design.
3. **Simulation**: the compiled design is simulated over some requested time-period, modelling the changes in internal logic levels using events. During simulation the simulated design may interact with the external environment, for example printing messages to the console, or reading from files. It is also common to dump traces for certain signals, so that they can be viewed as wave-forms.

Icarus Verilog does not support separate compilation so there are two main phases:

1. **Compilation + elaboration**:

   The following is all one command, with the slash (\) used to indicate command continuation:

   ```
   $ iverilog -Wall -g 2012 -s and_not_testbench -o and_not_testbench \
       and_gate.v or_gate.v
   ```

   This command compiles the Verilog modules into a simulation binary called `and_not_testbench`. The command options are:

   - `-Wall` : Enable all warnings during compilation.
   - `-g 2012` : Use the 2012 revision of the Verilog language. This allows for additional language constructs compared with "classic" Verilog.
   - `-s and_not_testbench` : Specify that `and_not_testbench` should be the top-level module. In this case it could be inferred, but it does not hurt to specify explicitly.

2. Execution:

   ```
   $ ./and_not_testbench
   ```

   This causes the simulation to run. However, very little will actually appear to happen, as the simulation will complete without printing anything. We will need to either view waveforms, add assertions, or print output in order to see anything.

   The compiled file looks like a binary, but if you run:

   ```
   $ file and_not_testbench
   ```

   you will see that it is not. If you view `and_not_testbench` in a text editor, you can see the internal code for the compiled simulation. The shebang

at the start of the file serves the same purpose as the shebang in a shell script, and specifies the general simulation program for any compiled Icarus Verilog program.

View Verilog test-bench output as waveforms

One way of understanding what is happening in a simulation is to view how the values of signals change over time. This can help you understand the relationships between when different signals change. A useful tool for viewing signals is GTKWave, which is widely used in open-source EDA tool-chains for viewing signals. GTKWave is available under Linux, OSX, and Windows, and can be compiled from source (using a similar method to Icarus Verilog). In this case the pre-packaged binaries should be fine.

By default the simulation does not generate waveforms, so the test-bench needs to be modified to produce them.

1. Install GTKWave:

   ```
   sudo apt install gtkwave
   ```

2. Modify the test-bench to enable dumping of signals. At the point marked `/* INSERT WAVEFORM COMMANDS */` in `and_not_testbench.v` insert the following:

   ```
   $dumpfile("and_not_testbench_waves.vcd");
   $dumpvars(0,and_not_testbench);
   ```

   The first command specifies the file-name of the waveform to be written, while the second specifies which module to dump the signals from and the number of levels in the hierarchy to include.

3. Compile and run the simulation again, to generate the waveform file `and_not_testbench_waves.vcd`.

4. Open the waveform file `and_not_testbench_waves.vcd` in `gtkwave`:

   ```
   gtkwave and_not_testbench_waves.vcd
   ```

   At this point you should see a window pop-up which shows you a number of panels, but no waveforms yet. You need to specify which signals to actually display, as often there will be thousands of signals from which you need to pick a small sub-set.

   Note: [WSL on Windows only] if you are using WSL on Windows, be aware that (as of Aug 2021) there is no default support for GUI windows from within WSL (unless you are on a windows insiders preview). You have two main choices:

   - Install an X server so that WSL can display windows. You will probably learn quite a lot doing this, but don't do it under time pressure.

- Install GTWave in Windows, and use file-sharing between the WSL guest and windows host. For example, if you have a file `~/wibble.vcd` in WSL, you can open it as `wsl://home/USER/wibble.vcd` in a Windows programme.

5. In the top-left panel of GTKWave (marked SST), right-click on `and_not_testbench`, and select `Recurse Import -> Append`.

   You should now be able to see the individual waveforms for each signal in the heirarchy.

You may wish to play around with the interface, as there are many features making it easier to work with, such as:

- "Toggle Trace Heir" : Used to prefix the signals with their position in the hierarchy.
- Left-clicking in the "Waves" pane shows a vertical line, allowing you to compare the values of many signals at the same time instant.
- Right clicking on a signal in the "Signals" pane allows you to change the data format for that signal. For example, you could choose to have signals shown as decimal, and/or interpreted as twos-complement.
- Selecting a signal in the "Signals" pane and then pressing the "left" or "right" key allows you to move between transitions. For the `clk` signal this allows you to visit each cycle. For other signals it allows you to jump to key points when they change value.
- Zoom the time-scale, either using `ctrl`+mouse-wheel, or using the `+` and `-` buttons. If you open a simulation and the waveforms either look flat, or just are a solid block of transitions, you can zoom in or out till the signal transitions are clear.

Add assertions to a test-bench

Staring at waveforms can be useful while debugging a circuit, but is impractical when there are thousands of cycles of transitions to track. A human is not going to be able to reliably calculate the expected output signals and compare them, and even if they could it would take days to check one waveform. We need an automated way of indicating what the expected behaviour is, and then be informed if the behaviour is ever violated. In Verilog this is performed using `assert`, which has a similar intended purpose and behaviour to `assert` in C++.

The `and_not_testbench` currently contains no assertions, so it has no idea whether the `and` and `not` gate are correct:

1. Replace the comment `/* INSERT assert HERE */` with the statement

   ```
   assert(d==1);
   ```

2. Recompile and simulate the test-bench. It should still simulate as before.

3. Deliberately break the `not` gate by replacing `assign r = ~a;` with

   ```
   assign r = a;
   ```

4. Recompile and simulate the test-bench. It should now report an assertion error, and identify the line where the error occurred.

5. Restore the `not` gate functionality, and check the test-bench passes again.

6. Insert appropriate `assert` statements after the remaining three delay statements, and check that the test-bench passes.

It might seem odd to deliberately break the circuit, but it is often worth temporarily violating assertions in order to check that they are active and have some power.

Add print statements to a test-bench

Another debugging technique is to use `$display`, which is similar to `printf` in C, or `cout` in C++. This allows you to print out messages during simulation, including the values of pre-determined signals.

1. Replace the comment `/* INSERT display HERE */` with:

   ```
   $display("a=%d, b=%d, d=%d", a, b, d);
   ```

2. Compile and simulate the test-bench, and you should see that the values of the signals `a`, `b`, and `d` are printed out.

3. Repeat the process (i.e. insert `$display`) just after the other three delays.

4. Compile and simulate the test-bench, and check that all the values are printed out.

5. We can place `$display` inside conditional blocks to control how often they print. Replace all four existing `$display` statements with:

   ```
   if ( d != ~(a&b) ) begin
       $display("Error : a=%d, b=%d, d=%d,  expected=%d", a, b, dk, ~(a&b));
   end
   ```

   This should ensure that the printing only happens when `d` has the wrong value.

6. Compile and simulate the test-bench. Given the `and` and `nno` gates are correct, no output should be printed.

7. Break the `and` gate again, and then re-simulate. You should find that the incorrect values get printed out just before the assertions fail.

An advantage of `$display` over looking at waveforms is that you can print out variables under very specific conditions. So if you have thousands of signals over thousands of cycles, it can be used to extract the exact situation where a

problem happened. Waveforms are often more useful when you don't know why things failed, and you are looking for clues.