

Icarus Verilog

About

Icarus Verilog is a free Verilog simulation and synthesis tool. It compiles source code written in Verilog (IEEE-1364) into some target format. It also generates netlists for the synthesis part. It's by far the best free tool and many people work on that making it more and more complete day by day.

Well, because I use Ubuntu (and specifically 13.10, which doesn't mean anything for the whole guide), I'll show you everything with commands for that OS.

Installation

Open a terminal and type (or copy-paste) the below commands.

Firstly, add a new ppa:

```
$ sudo add-apt-repository ppa:team-electronics/ppa
```

Then, update the local repository cache:

```
$ sudo apt-get update
```

Finally, install (update) the Icarus Verilog package

```
$ sudo apt-get install iverilog
```

Usage

Let's say you have the source files `one.v` and `two.v` to compile. A quick way to compile them is just by typing

```
$ iverilog one.v two.v
```

In that way, `a.out` is created and that's our binary:

```
$ ls
```

```
$ a.out one.v two.v
```

Now we can execute `a.out` with `vvp` command:

```
$ vvp a.out
```

If you want to specify another name for the binary, you can use the `-o` option just like in `gcc`:

```
$ iverilog -o my_out.out one.v two.v
```

```
$ ls
```

```
$ one.v two.v my_out.v
```

Well, all these are ok, but if you want to write on big, or complex verilog code, you should also use during the compilation the `-Wall` option:

```
$ iverilog -Wall -o my_out.out one.v two.v
```

`-Wall` option works in the same logic as in gcc. It enables more warnings during the compilation, that can help you find that little thing that destroys the functionality of your hardware. And yes, warnings in our case can affect the functionality of our design more often than programming in C. So `-Wall` is a must. You should also consider on `-Winfloop`. You can imagine its usage.

Well, that's all from me about Icarus Verilog. If you want to learn more about it, you should visit it's official website at <http://iverilog.icarus.com/> and for further How Tos and support its official wiki at http://iverilog.wikia.com/wiki/Main_Page. I also recommend you to subscribe to the mailing list <https://lists.sourceforge.net/lists/listinfo/iverilog-devel>.

GTKWave

About

GTKWave is a fully featured GTK+ wave viewer for Unix, Win32, and Mac OSX which reads LXT, LXT2, VZT, FST, and GHW files as well as standard Verilog VCD/EVCD files and allows their viewing. Its official website is at <http://gtkwave.sourceforge.net/>.

GTKWave is the best free wave viewer and is the recommended viewer by Icarus Verilog developer.

Installation

Open a terminal and type (or copy-paste) the below commands.


Firstly, update the local repository cache:

```
$ sudo apt-get update
```

Finally, install (update) the GTKWave package

```
$ sudo apt-get install gtkwave
```

Usage

It's usage it's really easy. In order to open GTKWave you can either type `gtkwave` in terminal, or by clicking on its icon: 

As you can imagine, there's nothing you can do by simply opening it alone. In order to see some waveform, you must open through GTKWave some saved dumpfiles. In our case we will use `.vcd` files, which are produced for our Icarus Verilog tool.

Creation of .vcd files

In our verilog code in an `initial` block we should include these two lines:

```
module Test;
    ...
    initial begin
        $dumpfile("my_dumpfile.vcd");
        $dumpvars(0, my_module_name);
    end
    ...
endmodule
```

NOTE: In case you have defined arrays in our design. E.g.:

```
module Mem (bla, bla, bla, ...);  
    ...  
    reg [M:0] data [N:0];  
    ...  
endmodule
```

its important that you include also these lines in your `initial` block (for every array you defined), in order to be able to see the array signals in GTKWave:

```
module Test;  
    integer i;  
    ...  
    initial begin  
        $dumpfile("my_dumpfile.vcd");  
        $dumpvars(0, my_module_name);  
        for(i = 0; i < M; i = i + 1)  
            $dumpvars(1, full.path.to.array.data[i]);  
    end  
    ...  
endmodule
```

It's probable that the last action will result a Warning type like that below during the compilation. You don't need to worry:

```
VCD warning: array word cpu_tb.cpu0.cpu_dp.data_mem.data[39] will  
conflict with an escaped identifier.
```

Opening VCD files with GTKWave

You can open o VCD file either from terminal or by the GUI. In this guide I'll use as much as I can the terminal way. It's more catchy and for sure it'll save as useful time, especially when we test a hardware design. So here we go:

```
$ gtkwave my_dumpfile.vcd
```

As you can see, there aren't any waves in the Wave window. That's because the user has to choose which signals want to see. In order to choose the signal you want to see, you should go in the left window with the SST name. Here you can see your hardware hierarchy. By clicking the + of every instance, you can see the signals that are related with that instance. Then you can drag&drop them, or copy&paste them in the Signals window. Voilà! Here are the waveforms.

NOTE: When I close and re-open GTKWave with the same VCD file, the signals that I have chosen before are lost. Should I always spend all this time by only inserting signals to be shown?

The answer is “of course not!”. After you have decided on the signals you want to be shown you can click **File** → **Write Save File As** and save a file with a .gtkw ending. Let's say for this example config.gtkw. From now and then you can open it with your .vcd file and get instantly to the point, which is debugging ;) :

```
$ gtkwave my_dumpfile.vcd config.gtkw
```

A better view on things

Now you are able to see the values of the signals in a faster way. But how about the interpretation? The values of the signals are in hexadecimal format and all waves are colored green. Yes! You can change these properties.

You can change them by right-clicking on the signals and choosing **Data Format** or **Color Format**. You are free to experiment with these options. When you have the desired optical result you can save your configuration by going **File** → **Write Save File**.

All crystal clear until now...

Take GTKWave by your side

When you right-click on a signal and especially in the **Data Format** option you can see the choice **Translate Filter File**. We'll stick with that option for the rest of this guide. It's a very powerful option, but it goes by the most times as there's not there.

With that option you can translate the values of the signals in a more human way. You can do this by writing a .txt translation filter. It isn't as complex as it sounds. Let's see an example:

You write a MIPS processor in Verilog. You have a Register File with the register of this architecture. You don't want to see the names of them as hex numbers, even as decimal. So you can simply write a .txt file like this:

```
00    $0 - $zero
01    $1 - $at
02    $2 - $v0
...
```

The first column represents the hexadecimal values that a signal that represents registers can have. The second more complex column is what we want to see instead of the first column. And that's it!. Now we only have to go to the signal we want to change, right-click on it, choose **Data Format** → **Translate Filter File** → **Enable and Select**. Then you have to browse to the filter you wrote, choose it and hit OK. Voilà! (Make sure that the data format is in hex before you choose the filter). Now you can save again your configuration. You're ready to go!

In that way you can filter whatever you want. From registers' names and opcodes to full instructions! Full instructions???

Supposing that we talk about a MIPS implementation, we initialize the Instruction Memory array with the verilog command `$readmemb()` from a file. This file must be of such format:

```
@0      000000_01000_00100_01001_00000_100000 // add  $t1, $t0, $a0
@4      101011_01011_01001_0000000000000101 // sw   $t1, 5($t3)
@8      100011_01011_10010_0000000000000101 // lw   $s2, 5($t3)
@c      000000_01000_10011_01001_00000_101010 // slt  $t1, $t0, $s3
@10     000100_01001_10010_1111111111111011 // beq  $t1, $s2, -5
```

where the first column is the address in hex of the command (in MIPS every address takes 32 bits) and the second column is the binary representation of the MIPS assembly instruction. You can include some C++ style comments if you like to.

This special command is used like that in an `initial` block:

```
$readmemb("program.mbin", full.path.to.data);
```

Before we continue with my favorite part, let's make our life a lot easier.

Makefile: Our best friend

Let's create a simple make file with what we saw up to here. The scenario is that we have a design in Verilog and we want to run it and see the waveforms instantly, without touching the mouse at all:

```

CC = iverilog
FLAGS = -Wall -Winfloop

library_input: one.v two.v testbench.v
    $(CC) $(FLAGS) -o test one.v two.v testbench.v

vvp test

gtkwave my_dumpfile.vcd config.gtkw

```

The makefile should be in the same directory with the other .v, .vcd and .gtkw files in our example. You simply execute this kind of script by just typing make in the terminal.

```
$ make
```

Now, when you want to run your verilog code you can just type make and everything comes up for you.

No pain No gain

Now comes the last but the best part. How can you represent the instructions in GTKWave from the moment you have to change them from test to test? It's time consuming. It's also time consuming to convert an instruction to its binary representation in order to test the processor. There are some online converters, but they restrict you in the way that you cannot insert a whole program and get its binary. Furthermore, its boring to fill the file with addresses (@0, @4, ...). There are times you want to run a big program, and you spend the ¾ time in converting it in a format that \$readmemb() can read it.

For this reason there's no solution but to develop your own parser. You'll spend a few hours to finish it, but the results will save you a lot more hours in the testing of your code. For me I developed a MIPS assembly to MIPS binary parser. I named it MASMBIN and works like this: It takes as first argument the .masm assembly file (input), and as a second argument the .mbin binary file (output):

```
$ ./masmbin input.masm output.mbin
```

In this stage, it can take 2 options:

```
-c
```

Shows in .mbin file in comments the parsed instruction

```
-f filter.txt
```

Outputs a .txt filter file for GTKWave

So as you see, this parser does another job with the right option: It creates the filter that we wanted to represent the values of signals in GTKWave as higher level instructions.

Now we're going to put everything together and have a nice MIPS assembly simulator.

Let's start from the conventions you should take:

- In `$readmemb()`, the file must be the same (for our example, *program.mbin*)
- The filter for our instructions in GTKWave should have a stable name (for our example, *translate_instruction.txt*)

Ok, now let's write our final makefile:

```
CC = iverilog
FLAGS = -Wall -Winfloop

library_input: one.v two.v testbench.v
    $(CC) $(FLAGS) -o test one.v two.v testbench.v

    ./masmbin -c -f translate_instruction.txt program.masm
program.mbin
    vvp test
    gtkwave my_dumpfile.vcd config.gtkw
```

Now, everything is in place. You can edit the `program.masm` file as you want, save it, and execute `make`. Everything will come up automatically.

If you want to download the new version of MASMBIN you can do so at

https://dl.dropboxusercontent.com/u/63037297/MIPS_asm_to_bin_parser_v2.0.tar.gz.

For the previous version refer to the github repository:

<https://github.com/gon1332/MIPS-asm-to-bin-parser>