Verilator Introduction

Fadi Alzammar

May 15, 2023

What is Verilator?

Verilator is a free, open-source tool for converting Verilog and SystemVerilog code to a cycle-accurate C++[1] model. This model can then be run to simulate the behavior of our circuit. Simulating this way, as opposed to the traditional timestep-based approach directly from HDL in tools such as Xilinx Vivado, has several advantages:

- The simulation only needs to update each signal when there is a change to it, rather than every timestep. This means that simulations run *much* faster.
- We have the full power of C++ available to us. Though waveforms are definitely useful, we can also port our data into any program we can make to analyze the data, including even a VGA simulator!
    - Similarly, programming our testbenches in C++ allows us to easily complex scenarios that would be a huge pain to do in (System)Verilog.
- It is easy to analyze code coverage, showing you what parts of your HDL are most commonly used (if at all!). This is a great process for determining where you should place your focus when trying to improve the performance of a circuit.

Unfortunately, this does not come without downsides. The two main ones are:

- Only synthesize Verilog can be simulated, or in other words, Verilog that can actually be turned into hardware. This means that delay statements like `#10` and similar software-only constructs will not work.
- Verilator is a two-state simulator. Many traditional simulators, such as the one in Xilinx Vivado, are 4-state. Both, of course, support 0 and 1 states for each bit, but only the latter supports unknown (X) and high-impedance (Z) states.

Installation

Operating system preparation

Depending on your operating system, there may be some additional steps required to run Verilator.

Linux

If you are already on (a reasonably popular distribution of) Linux, great! There's nothing to do for this step.

MacOS

MacOS is almost as easy to get started with. Just install the Homebrew package manager by running the following in Terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

---

[1]Verilator can also convert to SystemC, but that will be outside the scope of these guides.

Windows

If you are on Windows, there is some more work. The easiest way for us to get started is with Windows Subsystem for Linux (WSL), which is essentially a Hyper-V virtual machine with strong integration to Windows. As long as you are on Windows 10 version 2004 or greater, or Windows 11, the installation is as easy as opening a PowerShell or Command Prompt as administrator and running:

```
wsl --install
```

By default, this will install Ubuntu. If you would like to install a different distribution, you can instead run:

```
wsl --install -d <Distribution Name>
```

You'll then be prompted to restart your PC, after which you can access the VM by searching for "Ubuntu" (or whichever distribution you chose), or by opening it as a new tab in the wonderful, modern "Terminal" application, where it should automatically be added as a dropdown option.

Verilator

Verilator can easily be installed from most package managers. For Debain/Ubuntu-based operating systems, this can be done with:

```
sudo apt install verilator
```

On MacOS, you will instead use Homebrew:

```
brew install verilator
```

If you have specific requirements beyond the scope of these guides, or you are not on a distribution which Verilator is packaged for, you may need to follow instructions on Verilator's installation page to build from source.l

GTKWave

In addition to Verilator, we will be using GTKWave, an open-source waveform viewer, after running our simulations. Similarly, it can be installed with:

```
sudo apt install gtkwave
```

Visual Studio Code

I will also be including instructions to integrate Verilator into Visual Studio Code, a popular code editor. I'd recommend installing this directly from their website for more frequent updates than a package manager.

If you are using WSL, you can install it on Windows and then add the WSL extension. From there, you can run `code <file.txt>` in WSL on any file you wish to edit to launch a connection from the . The first time you do this, it will automatically install the required package in your WSL distribution.

Configuration

For an optimal coding experience, install the C/C++ extension. Then, so that we can see Verilator constructs and functions in IntelliSense, add the Verilator header to the extension's include path:
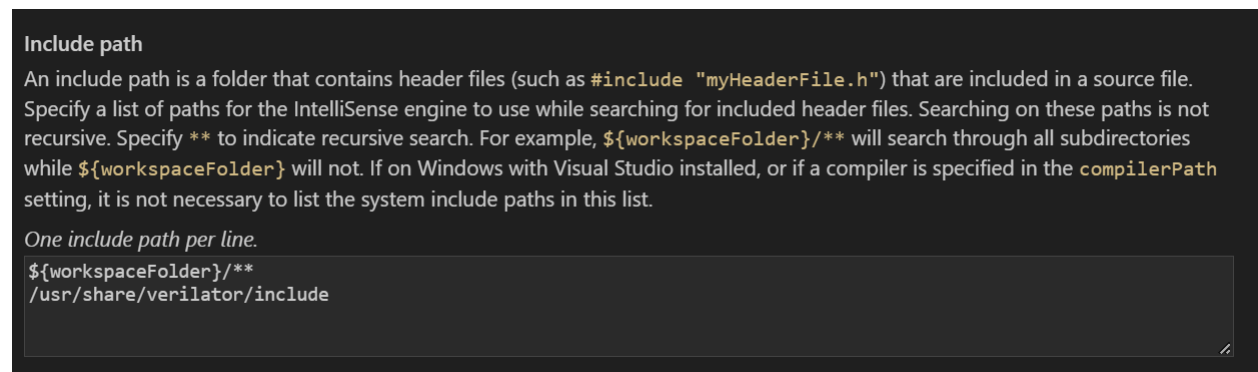
1. Open the Command Pallete by pressing Ctrl+Shift+P

2. Begin searching for "Edit Configurations" until you can click the following option:
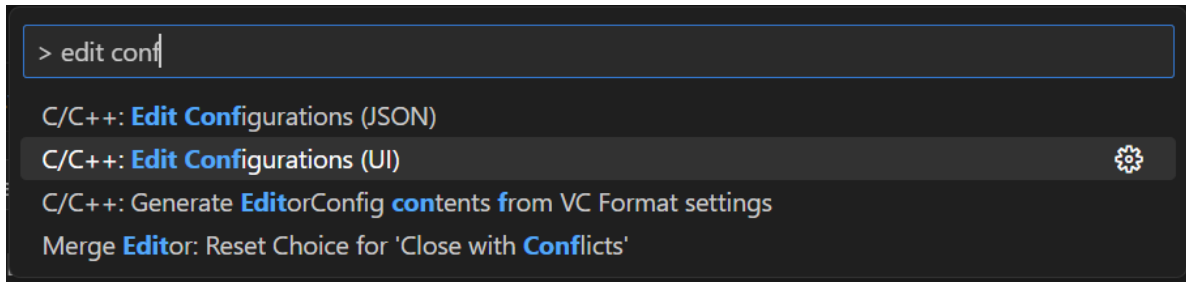
Add Verilator to the "Include path". If you installed Verilator with `apt`, you will want to add the following directory:

`/usr/share/verilator/include`

Your setting should now look something like this:

Our First Verilated Module

The HDL

As a simple example, let's begin with a simple counter. Below is the SystemVerilog code for the module.

```systemverilog
module counter #(
    parameter BITS = 8) (
    input clk,
    input reset,
    output logic [BITS-1:0] count
    );

    always_ff @ (posedge clk) begin
        if (reset)
            count <= 0;
        else
            count <= count + 1;
    end
endmodule
```

C++ Testbench

Now for the actual Verilator part. I will include my full testbench below, then deep-dive into what each part is doing.

```cpp
#include <stdlib.h>
#include <iostream>
#include <verilated.h>
#include <verilated_vcd_c.h>
#include "Vcounter.h"

int main(int argc, char** argv, char** env){
    // Initialize verilated module
    Vcounter *dut = new Vcounter;

    // Trace setup
    Verilated::traceEverOn(true);
    VerilatedVcdC *vcd = new VerilatedVcdC;
    dut->trace(vcd,5);
    vcd->open("waveform.vcd");
    int i = 0;

    // Initialize inputs
    dut->clk = 0;
    dut->reset = 1;

    // Reset
    dut->eval();
    vcd->dump(i);

    dut->reset = 0;
    dut->clk = 1;
    i++;
    dut->eval();
    vcd->dump(i);

    // Run simulation
    while(dut->count <= 0xF) {
        printf("Count: %i\n", dut->count);

        dut->clk = 0;
        i++;
        dut->eval();
        vcd->dump(i);


        dut->clk = 1;
        i++;
        dut->eval();
        vcd->dump(i);
    }

    // Cleanup
    vcd->close();
    delete dut;
    exit(EXIT_SUCCESS);
}
```

Headers

```
#include <stdlib.h>
#include <iostream>
#include <verilated.h>
#include <verilated_vcd_c.h>
#include "Vcounter.h"
```

- `stdlib.h` and `iostream` are standard C++ libraries for general functions and input-output operations
- `verilated.h` is the standard Verilator header
- `verilated_vcd_c.h` is an additional Verilator header required to output VCD trace files
- `Vcounter.h` is the header for our C++ model which Verilator will create from our Verilog module
  - `counter` should be replaced with whatever the name of your module is

Verilator Setup

```
// Initialize verilated module
Vcounter *dut = new Vcounter;
```

First, we create an object that will act as our DUT (device under test), or in other words, the module we are simulating. The `Vcounter.h` defines a `Vcounter` type that has all the inputs and outputs of our hardware module. `dut` is a pointer to an object of this type.

```
// Trace setup
Verilated::traceEverOn(true);
VerilatedVcdC *vcd = new VerilatedVcdC;
dut->trace(vcd,5);
vcd->open("waveform.vcd");
int i = 0;
```

`traceEverOn` is used to enable VCD trace writing, and `vcd` points to an object which stores the trace information. The function `trace(vcd,5)` associates our trace data to the module, with `5` being the maximum signal depth that will be recorded. We open `waveform.vcd`, the actual file that the trace data will be written to, and finally, `i` is the current timestep we are on in the simulation. Every time we advance in time, `i` must be incremented appropriately.

Initialization

```
// Initialize inputs
dut->clk = 0;
dut->reset = 1;
```

We want to set the values our inputs start at, so that we can ensure the same simulation behavior every time. More on this will come later.

Reset

Similarly, we want to start with a known state for all our internal signals by resetting our module.

```
// Reset
dut->eval();
vcd->dump(i);

dut->reset = 0;
dut->clk = 1;
i++;
```

```
dut->eval();
vcd->dump(i);
```

Because we initialized `reset` high, our module enters reset immediately. The state of our module, `dut` is evaluated with `eval()`, and we dump this information into `vcd`, recording it at timestep `i` (which is currently 0), with `dump(i)`.

At the next clock edge, we set `reset` low and `clk` high, `i` is then incremented, so that the next `dump()` will save to the next timestep.

Simulation

```cpp
// Run simulation
while(dut->count <= 0xF) {
    printf("Count: %i\n", dut->count);

    dut->clk = 0;
    dut->eval();
    vcd->dump(i);
    i++;

    dut->clk = 1;
    dut->eval();
    vcd->dump(i);
    i++;
}
```

The fun part! This simulation is running until our `count` output hits `0xF`, or 15. Each clock cycle, the current cycle is printed to the console.

Cleanup

```cpp
// Cleanup
vcd->close();
delete dut;
exit(EXIT_SUCCESS);
```

Finally, we just need to close the VCD file we opened, clear the memory we allocated for `dut`, and exit our program with a successful error code!

Simulating

To "verilate" our model, run:

```
verilator -Wall --trace --exe --build -cc tb.cpp counter.sv
```

Breaking it down:

- `-Wall` enables all warnings
- `--trace` enables Verilator to dump our waveform data to a VCD
- `--exe` links our C++ testbench to the Verilog code
- `--build` automatically builds the new C++ model that Verilator generates
- `--cc` specifies that we are using C++

After those flags, you just need the name of your C++ testbench and the top-level Verilog module. Successfully running this command should result in an **obj_dir** directory being created, which contains the C++ model

Verilator generated from our Verilog code and an executable of our simulation.

Finally, to run the simulation, we just need to run the aforementioned executable:
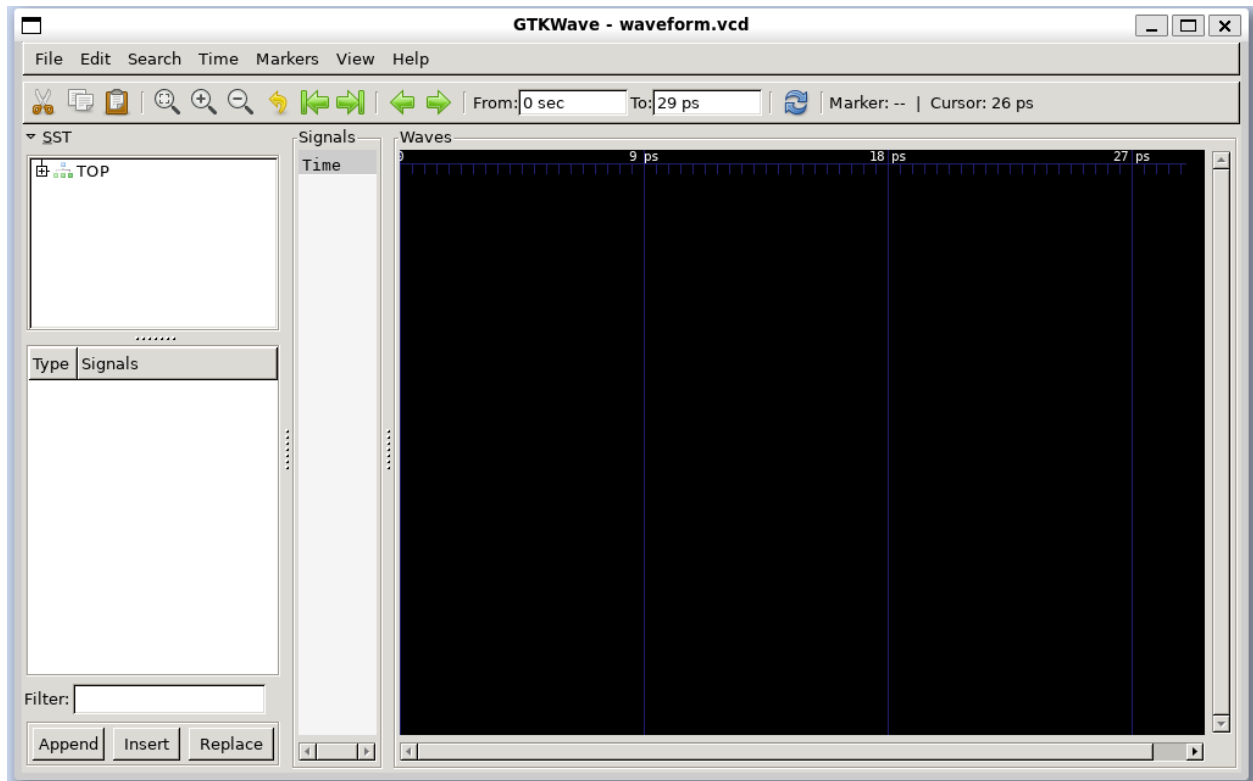
```
./obj_dir/Vcounter
```

If all went well, you should see counts 1-15 printed to the console, and you should now have a file `waveform.vcd` in the folder you ran this.
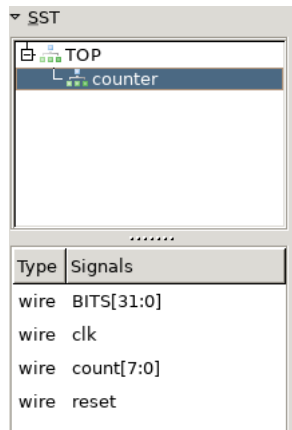
Viewing the Waveform

To open our newly created trace file in GTKWave, run:

```
gtkwave waveform.vcd
```

More than likely, it will look something like this:

Don't worry, this is normal! Expand the line labelled "TOP" to see our module, and click on each line to see what signals are visible at each level.

To view a signal, just drag it to the black area on the right. You can select multiple signals by holding Ctrl while you click on them, or alternatively click on one signal, hold Shift, and click another signal to select all signals between them. If you are doing this, make sure to continue holding Ctrl or Shift while dragging the signals.
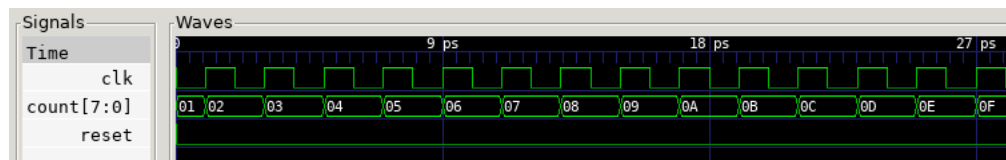
From here, you can use the buttons at the top to zoom in/out or scroll between edges.

Congrats, you've successfully Verilated a module, simulated it, and viewed its signals as waveform! Next, we will look at Verilating the OTTER.