Imperial College London

Lecture 3

Verilator, Testbench and Vbuddy

Prof Peter YK Cheung
Imperial College London

URL: www.ee.ic.ac.uk/pcheung/teaching/EIE2-IAC/
E-mail: p.cheung@imperial.ac.uk

This lecture is mostly about HOW to verify a SystemVerilog design.

Learning outcomes

- Different types of **simulators**
- Verify a SystemVerilog (SV) module with **Verilator**
- Template for a **Verilator testbench**
- Using a **shell script** as shortcut
- Verify a SV module using **gtkWave** waveform viewer
- Verify a SV module using **Vbuddy**
- What is in **Lab 1**?

Slides in this lecture are partly derived and modified from:

"*Verilator: Fast, Free, But for me?",* a talk by Wilson Snyder (created of Verilator) – http://www.veripool.org/papers

Here are a list of learning outcome for this lecture. It is also tightly coupled with Lab 1, which will take you through the steps in verifying a SystemVerilog module is working properly using a number of tools.

Verilator History

- Verilator was born in 1994
  - Verilog was the new Synthesis Language
  - C++ was the Test-bench Language
  - Paul Wasson synthesized Verilog into C++
- Wilson Snyder created Verilator since 2001
  - Open-source and free
  - Strong community with many contributors
  - Works on all platforms (PC, Linux, MacOS)
  - Fast, particularly with multithreading

Some history about Verilator.

It started life as a synthesis program that takes Verilog description and turn it into C++ code that one can compile and run just like any other C++ programme.

This initial work was then further developed by Wilson Snyder since 2001, and now grow into an open-source community with many contributors.

I have chosen this in preference to what was used previously for teaching this module because:

1. It is free;
2. It works on all platforms (even with Apple silicon);
3. It is fast, particularly for processor simulation.

Verilator User Base

All trademarks registered by respective owners.
Users based on correspondence; there is no official way to determine "users" since there's no license!

Verilator is now adopted by many companies and universities. This is their list of users back in 2010. Imperial College was already a user!

Now the list is actually too large to include in a slide like this. Basically almost all of electronic chip design places and many many universities have adopted Verilator either for research, teaching, or both!

Three types of simulator

1. **Instruction level simulator**
   - Processor instruction-level function
   - No hardware representation, no concept of clock
   - Written in high level language such as C or C++
   - GNU RISC-V toolchain includes such a simulator
2. **Cycle accurate simulator**
   - Simulate HDL specification of hardware, e.g. SystemVerilog
   - All signal values correct cycle-by-cycle
   - No time delay information
   - Verilator is an example – only two values: ”0” or “1”
3. **Event-driven simulator**
   - Change signal produces event in an event queue
   - Simulate delay using timing model
   - Capture glitches
   - Slower and costly. Example: ModelSim

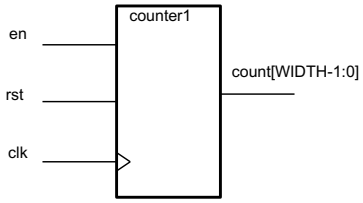
GNU toolchain for RISC-V provides an instruction level simulator. This is fast, but is devoid of any reference or specification of hardware. It is NOT useful to produce a final processor, but can be used to generate expected results. It is very fast, but lacks details.

Verilator is the 2nd type, which is cycle-accurate. Each clock cycle marks a time step, and between each cycle, all logic specifications are evaluated. Verilator synthesis compiler is excellent in checking the synthesizability of the SV specification. If Verilator can produce a working design, this design is likely to be synthesizable on FPGA or other commercial tools to produce an actual physical chip design. What is lacking is timing information. The chip will work, but not necessary at a frequency you want or need. Furthermore, Verilator only understands two-level logic: ’0’ or ‘1’. Therefore tri-state busses can be a problem to simulate.

Finally the most comprehensive type of simulator for digital is ”event-driven” simulator. This type of simulators generates an event when an input signal changes state, which is put onto the event queue. The impact of this event is evaluated, and produces other events internal to the circuit. These are also put onto the event queue. A simulation step is completed when ALL events in the event queue are evaluated for the current time. Event-driven simulator can handle time delay and is therefore able to evaluate propagation delay, race conditions and glitches.

Example: Simple Counter

```
1  module counter #(
2    parameter WIDTH = 8
3  )(
4    // interface signals
5    input  logic             clk,      // clock
6    input  logic             rst,      // reset
7    input  logic             en,       // counter enable
8    output logic [WIDTH-1:0] count     // count output
9  );
10
11 always_ff @ (posedge clk)
12   if (rst) count <= {WIDTH{1'b0}};
13   else     count <= count + {{WIDTH-1{1'b0}}, en};
14
15 endmodule
```



In Lab 1, you are to design an 8-bit counter as shown here. The counter has three inputs:

1. clk – the clock signal (positive edge triggered)
2. rst – the reset signal (high reset), synchronous to clk
3. en – the enable signal, i.e. counting only if en = '1'

The counter has output count[7:0].

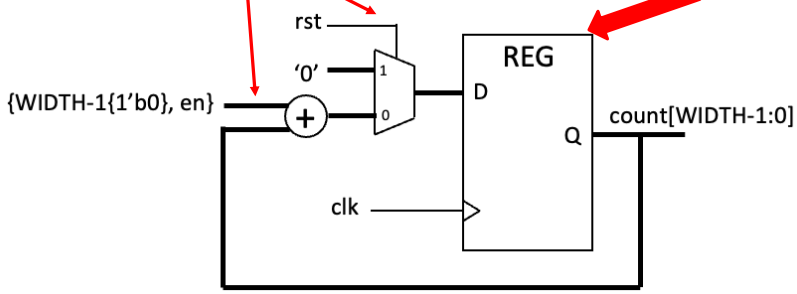
Note the following:

1. We use parameter to define the width of the counter to be 8-bit. The use of parameter allows the same module to be used with different counter width (covered in next lecture).
2. The use of concatenation {.} to create 8-bit value with the LSB = en signal.

Mapping from SV to hardware

```
if (rst) count <= {WIDTH{1'b0}};
else     count <= count + {{WIDTH-1{1'b0}}, en};
```

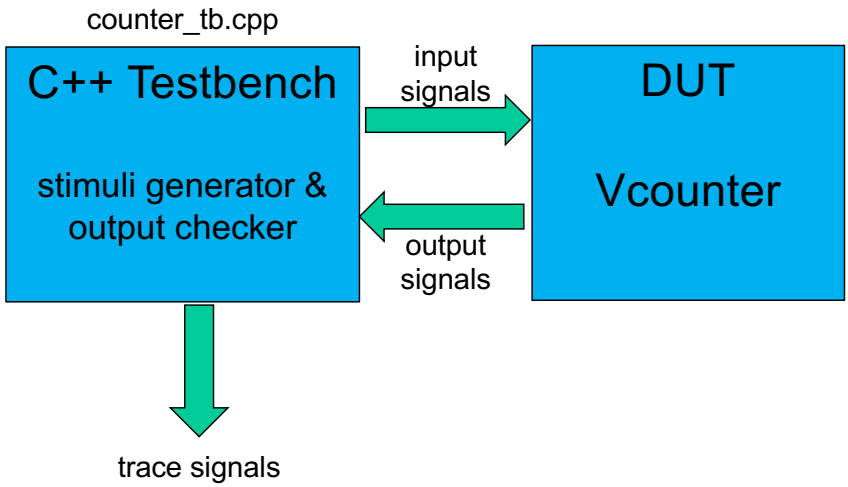
```
8    output logic [WIDTH-1:0] count
11   always_ff @ (posedge clk)
```



This is how the SV code is mapped to the actual hardware synthesized by Verilator.

The if-else statement is mapped to the MUX. The couting action is achieved via the adder on the feedback path of the register.

Testbench

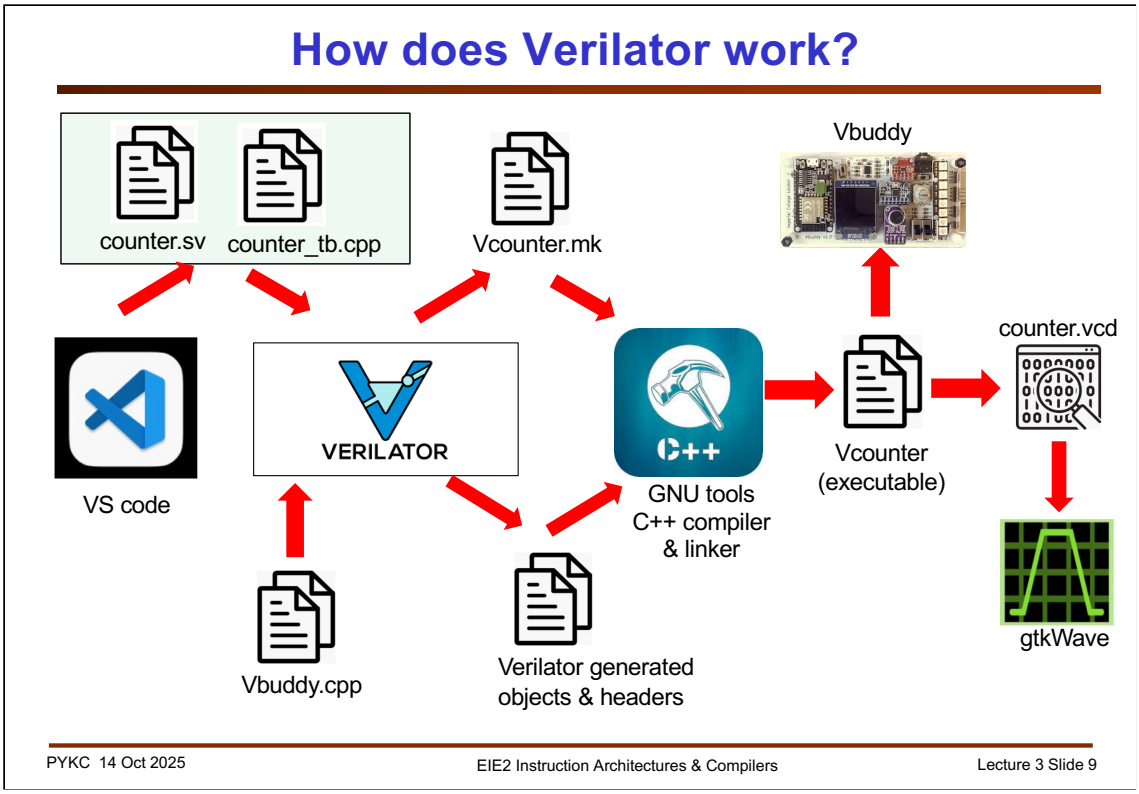


Before this module can be simulated and verified, we need to create a testbench program.

SystemVerilog was designed also with verification in mind. However, Verilator is designed to synthesize hardware, and does not cope with those parts of SV that are not synthesizable. Therefore, to test our counter, we need to write a separate testbench file in C++.

This testbench file is essentially a “wrapper” for the module counter, which is the Device Under Test (or DUT). It produces the input stimuli to the DUT, receive the results (which is the signal count[7:0]) and help the user to determine whether the DUT is doing what it is suppose to.

One way to verify the hardware functionality is to produce a trace file which contain a record of all the signals (at the top-level) over time. This can then be interrogated and displayed using another program, such as gtkWave, which is an open-source wave viewing program.

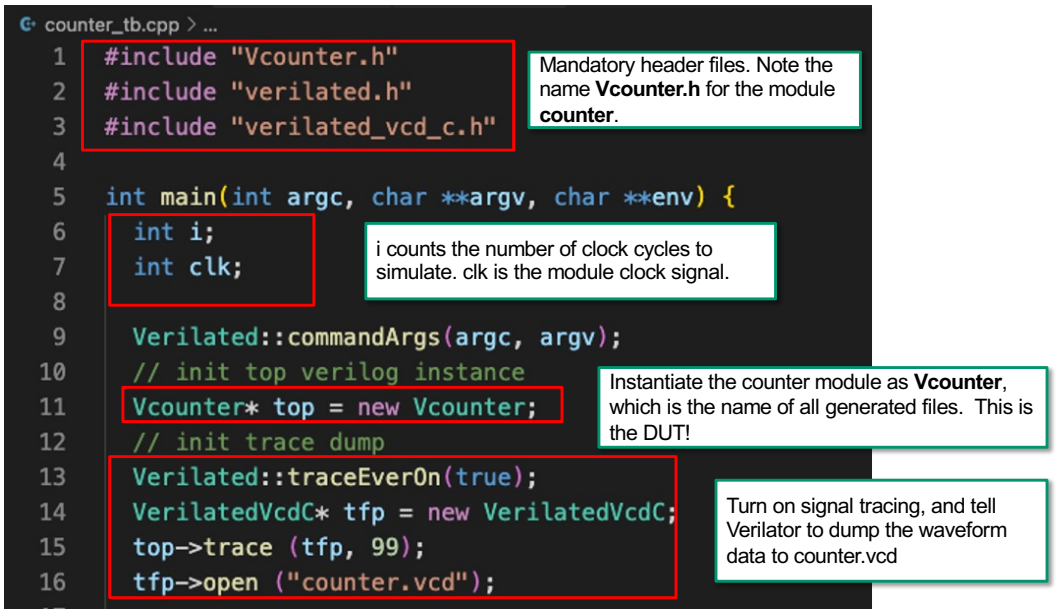


This slide summaries the various programs required to take a SystemVerilog source code, and turn it into an executeable program.

Consider our “counter” module. Here are the steps:

1. Use VS Code to create and edit counter.sv.
2. Use VS code to create and edit the testbench file counter_tb.cpp.
3. Use Verilator to compile the HDL source code (.sv) with the testbench (.cpp) and the Vbuddy API (Vbuddy.cpp) to produce the C++ program which contains the synthesized hardware, the testbench procedures etc., and a bunch of header files.
4. Verilator also produces a make file (Vcount.mk) which tells the C++ how to compile and link everything together to produce the final executeable model of the counter with built-in testbench (the Vcounter binary file).
5. This executeable binary of the counter is a native ce program which can be executed by the computer to produce the output signals, which can be displayed as waveform or even drive an external unit such as the Vbuddy.

Format of the Testbench (1)



The testbench file has a standard structure with different sections. They are highlighted here in separate RED boxes with explanation.

1. If the module is called counter, a header file "Vcounter.h" is produced, which defines the interface signals for this module. Verilated.h must always be included. If you want Verilator to produce the trace file for laer inspection, Verilated_vcd_c.h must also be included.
2. The next section declares internal variables (they are not signals) for the testbench. In this case, i counts the number of clock cycles, and clk is another internal variable to count the phase of the clock (high, low).
3. `Vcounter* top = new Vcounter;` is the statement that instantiate the module (or DUT). If the module is called "encoder", then this line will be:

   `Vencoder* top = new Vencoder;`
4. This segment is the same for all testbenches. It tells Verilator to produce the trace file "counter.vcd" for you to inspect later.

Format of the Testbench (2)

```
18   // initialize simulation inputs
19   top->clk = 1;
20   top->rst = 1;
21   top->en = 0;
22
23   // run simulation for many clock cycles
24   for (i=0; i<300; i++) {
25
26     // dump variables into VCD file and toggle clock
27     for (clk=0; clk<2; clk++) {
28       tfp->dump (2*i+clk);       // ur
29       top->clk = !top->clk;
30       top->eval ();
31     }
32     top->rst = (i <2) | (i == 15);
33     top->en = (i>4);
34     if (Verilated::gotFinish())  exit(0);
35   }
36   tfp->close();
37   exit(0);
38 }
```

Set initial signal levels. Top is the name of the top-level entity. Only the top-level signals are visible.

This is the for-loop where simulation happens. i counts the clock cycles.

This is the for-loop that toggles the clock. It also output the trace for each half of the clock cycle, and force the model to evaluate on both edges of the clock.

Change rst and en signals during simulation.

5. This segment initialize the state of the input signals before clocking the circuit. “top” is the name given to the DUT in 3) above. In C++ terms, this is a pointer pointing to a structure that has all the input and output signals.

   Therefore top->rst = 1 initializes the rst signal of counter to a ‘1’.
6. The next segment is the first for-loop which cycle through the clock cycles i. this is th
7. The inner for-loop is the part that does the business. It dumps the signal values to the trace file specified in 4), toggles the clk signal top->clk, and then perform a simulation step with top->eval() function. It does it only twice because each clock cycle has two phases: negative and positive edges.
8. The final section changes the stimuli signals within the simulation loop.

These 8 segments of the testbench require modifications depending on the DUT, the required signal stimuli and how you might want to observe the output signals (such as using Vbuddy).

Making the final simulation model

$ doit.sh

```
#!/bin/sh

# cleanup
rm -rf obj_dir
rm -f counter.vcd

# run Verilator to translate Verilog into C++, including C++ testbench
verilator -Wall --cc --trace counter.sv --exe counter_tb.cpp

# build C++ project via make automatically generated by Verilator
make -j -C obj_dir/ -f Vcounter.mk Vcounter

# run executable simulation file
obj_dir/Vcounter
```


Once the SV source file counter.sv and the testbench file counter_tb.cpp have been created, the following steps need to be taken:

1. Run Verilator with this command in the terminal window:

```
# run Verilator to translate Verilog into C++, including C++ testbench
verilator -Wall --cc --trace counter.sv --exe counter_tb.cpp
```

   -Wall : report all warnings

   --cc : generate C++ instead of SystemC codes

   --trace : produce trace for all signals

   --exe : produce an executeable model Vcount using counter_tb.cpp as testbench.

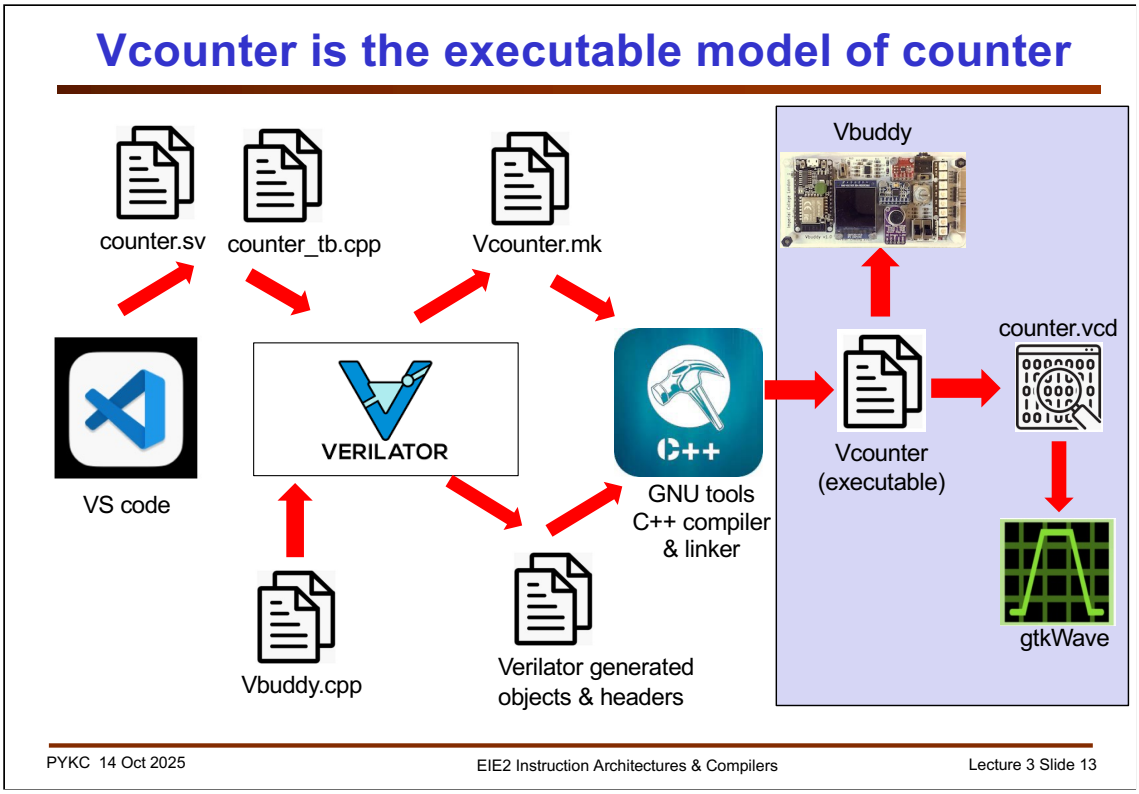
2. Verilator produces a make file (Vcounter.mk), which uses the make utility to build the final simulation model. The command is:

```
# build C++ project via make automatically generated by Verilator
make -j -C obj_dir/ -f Vcounter.mk Vcounter
```

   Run this make command telling it to that object and header files are found in the obj_dir/ folder, using the make rules found in Vcounter.mk.

   The result of this make is to produce an executeable model of the counter in the obj_dir/ folder called Vcounter.

3. The final step is simply to run the binary file in obj_dir/Vcounter.

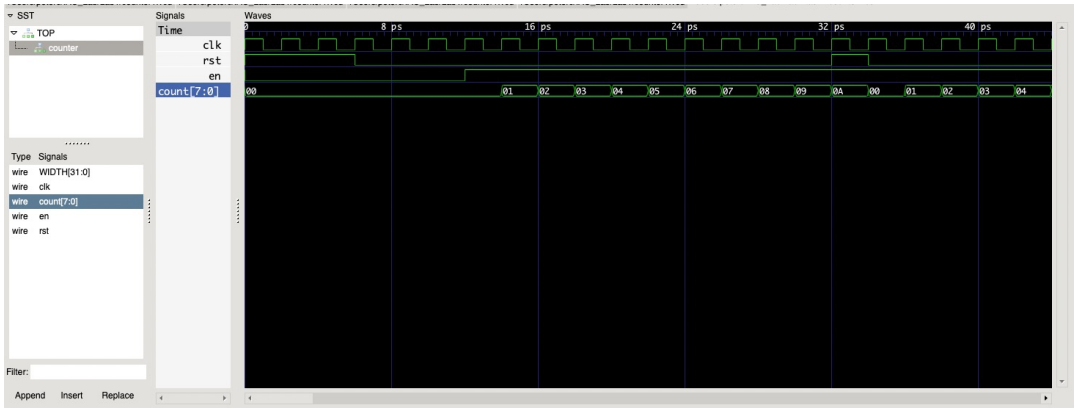


Once the executeable model of the counter, Vcounter, is produce, we can run it.

This has two consequences. The testbench program will send input signals to the counter and a set of waveform traces will be produced in the file counter.vcd.

Also, you can use the Vbuddy.cpp API calls to send messages to the Vbuddy board, which will display the results of the DUT for you to inspect. You will find how this works during Lab 1.

Checking the simulation results

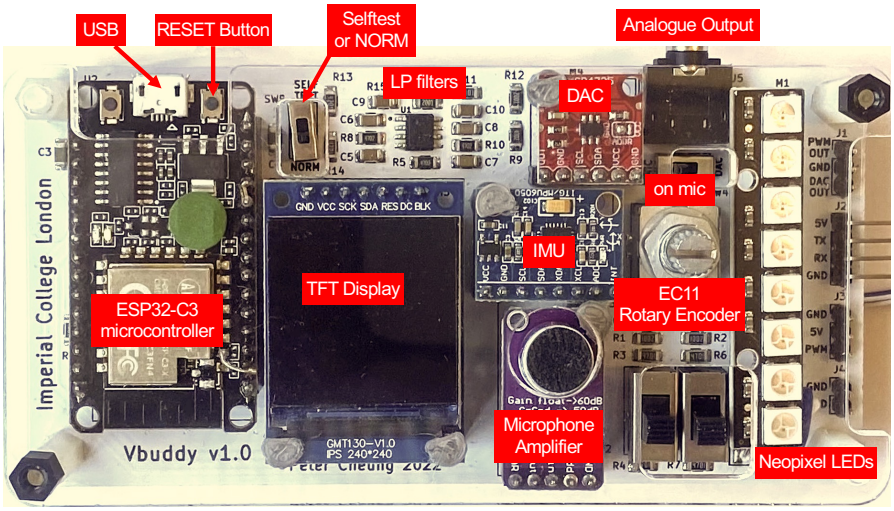


Once the executeable model terminates, the trace file counter.vcd is produced. You can open and load this trace file using the waveform viewer app gtkWave. Here is a sample of the waveforms being displayed by gtkWave.

Since Verilator is cycle-by-cycle simulator, the time axis has no significant. Each clock tick (i.e. half a clock cycle) is display as 1ps, which is of course nonsense.

Nevertheless, the states that the circuit goes through are accurate. The counter output count[7:0] shows values that is incrementing one every clock cycle. The effect of rst and en signals can also be observed.

Vbuddy



Inspecting waveforms to verify a piece of digital hardware working is really boring. Therefore, I have designed a board called Vbuddy to spice things up. I called it Vbuddy because it is a good companion to V, which stands for RISC-V, Verilog or Verilator – take your pick!

Vbuddy receives messages over the USB cable from the computer running the Verilated model of your hardware. The microcontroller board is itself a RISC-V processor, the ESP32-C3 made by Espressif. The board is from AI-Thinker. It also consists of various components as shown above.

A set of API calls (written in C++) are provided for use with the testbench to do various things such as displaying data as a waveform on the TFT display while simulation is happening, or show the counter output value on 7-segment displays (simulated) or capture realtime audio signal with the microphone, amplifier and A-to-D converter.