Department of Electrical & Electronic Engineering, Imperial College London

EIE 2 Instruction Set Architecture & Compiler (IAC)

Lab 1 – Learning System Verilog with Verilator and Vbuddy

Peter Cheung, v1.1 - 19 Oct 2022

Objectives

By the end of this experiment, you should have able to:

- write a basic **System Verilog module** for a binary counter
- write a basic **testbench** in C++ to verify the counter is working
- make an executable model of the counter and testbench using **Verilator**
- understand how the counter hardware is **mapped to C++ model**
- use **GTKwave** to examine the waveforms
- use the **Vbuddy board** to display outputs from the simulation model
- design a **programmable up/down counter** with different incremental steps
- single step the counter using the rotary encoder switch on Vbuddy
- implement in System Verilog a circuit to translate a binary number to a BCD number (optional)

Pre-requisites

Before you start this Lab Session, you should have completed Lab 0 at home on your laptop. This means that you have the following packages installed and have verified that they are working properly:

- Microsoft VS Code
- VS Code extensions for System Verilog, C++, Verilog, Markdown
- Git and Github
- Verilator
- GTKwave
- GNU toolchain

You should create a github account. I will be putting on github all my lab instructions and other supporting materials. You are also expected to keep your work for this module on github. If you do not already know how to use git and github, and watch the youtube video introducing how to use git and github.

What is System Verilog?

System Verilog is a hardware description and verification language (HDL) mostly for designing digital hardware. It is based on Verilog, the original language introduced in 2002 with improvement that helps verification. System Verilog is a superset of Verilog that is particularly suitable for register-transfer level (RTL) design. It also includes object-oriented programming constructs that are dedicated to verification but

are not synthesizable. System Verilog is now the most commonly used language for specifying and designing digital integrated circuits, including microprocessors such as RISC-V.

What is Verilator?

Verilator is software package to simulate Verilog/System Verilog designs. It differs fundamentally from other commercial simulators such as Modelsim, Synopsys VCS, iCarus etc. in approach. Most other simulators use a method known as “event-driven” simulation. Every change in an input signal, an event is created in an event queue. Such event causes other signals to change states, and more events are generated. All events are propagated and evaluated through the circuit model until no events remains in the queue. The advantage of this approach is that we can model timing through each gate or flip-flop, and signal levels can take on various signal states (not only ‘0’ and ‘1’).

Verilator instead is a cycle-accurate simulator, which means that circuits states are only evaluated once in a clock cycle. It does not evaluate time within a clock cycle and is therefore only suitable for functional verification without timing information. For example, Verilator cannot evaluate glitches in a circuit. Furthermore, Verilator signals can only be true (1) or false (0). It cannot be unknown, high impedance, floating or in other signal states.

So why use Verilator? Most modern digital circuits are synchronous, i.e. driven by a clock signal controlling registers with combination circuits in between. If we care about the functionality of the circuit at register transfer level (RTL) only, Verilog is much faster than event-driven simulators. Verilator only accepts synthesizable Verilog/System Verilog code. Therefore, a Verilator verified design is almost guaranteed to be synthesizable by modern logic synthesis tools. Finally, Verilator is free and open-source – great for educational users.

Why is Verilator so fast?

The reason why Verilator is so fast is fundamentally because it translates Verilog/System Verilog into C++ (or System C) code. Then it uses standard, highly efficient C++ compiler to produce natively executable “model” or program of the design (known as Device-Under-Test or DUT). Simulation happens when you run (or execute) this program, just like any other C++ application that you compiled!

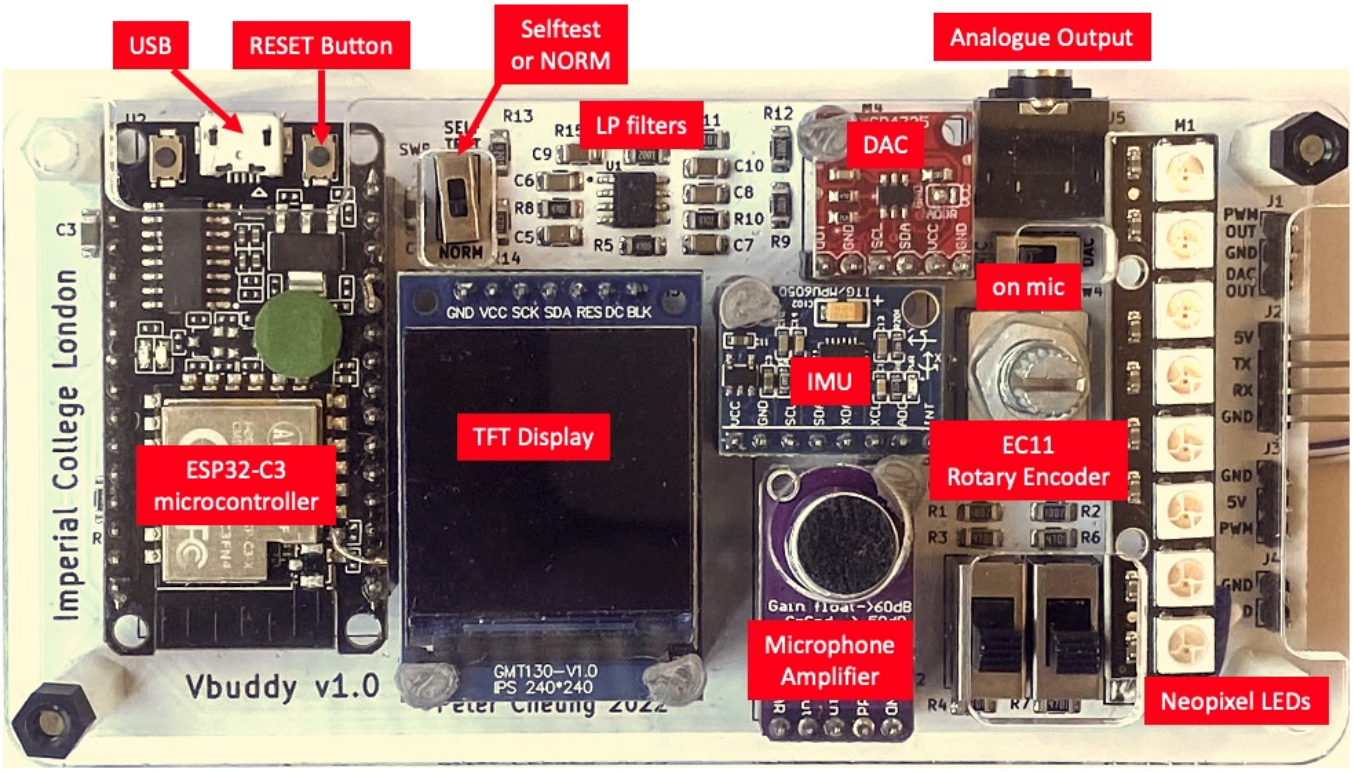
What is a Verilator testbench?

One disadvantage of translating a Verilog design into C++ executable is that you normally cannot interact with your simulator. Say, the executable program produced by Verilator is called “Vcounter”, then simulation is done running Vcounter itself. To know if the DUT is working, you have written a “wrapper” program that instantiate the DUT, provide input signals at the correct time, and display or compare the output signals. The wrapper program is known as a testbench for the DUT. When producing the executable model of the counter, it must include the DUT as well as the testbench code.

What is Vbuddy?

Normally using Verilator to simulate a digital circuit makes the entire process rather like software development – you compile a program, then run it! There is no easy way of interacting with the model of the DUT. For example, when simulating a counter, you cannot use the simulation model to drive a physical 7-

segment display. The lack of connection between software simulation and physical hardware makes the entire process of learning digital design “artificial”. For this reason, Vbuddy was created to provide a bridge between the Verilator simulator and actual physical electronics such as microphone signal and 7-segment displays.



Vbuddy is so named because it is a companion (or buddy) to Verilator, Verilog and RISC-V. It consists of an ESP-C3 RISC-V microcontroller, a 240x240 TFT colour display, a rotary encoder (EC11), a microphone with an autogain amplifier (MAX9814), an Inertia Measure Unit (MPU6050), a 12-bit DAC converter (MCP4725) and two channel lowpass filter. You will be using Vbuddy throughout the practical sessions of this module in the Autumn term.

Task1 - Simulating a basic 8-bit binary counter

Step 1: Create a folder tree on your local disk for this module.

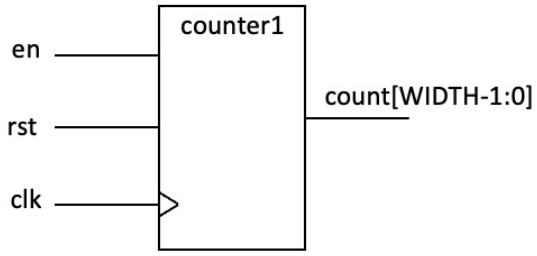
The easiest way to achieve this is to "fork" **this github repository** (repo) to your github account. If you do not already have one, you need to create one for this module. Now you need to FORK this repo to your github account, and then CLONE it onto your local disk. After successfuly clone the the Lab 1 repo to your disk, you should find a folder called Lab1-Counter. This is your personal folder for Lab 1. You can add files to it and push them back to your github repo under your github account.

Step 2: Run VS Code and open the Lab1-Counter folder with:

File -> Open Folder (then select the task1 folder)

Step 3: Create a new System Verilog file (counter.sv), and enter the code shown on the right. You should include all the comments for future reference. Colour highlight is automatic if you have installed the System Verilog package for VSC. The schematic representation of this basic counter is shown below. It counts on

the positive edge of clk if enable is '1'. It is synchronously reset to 0 if rst is asserted. Save this file as counter.sv.

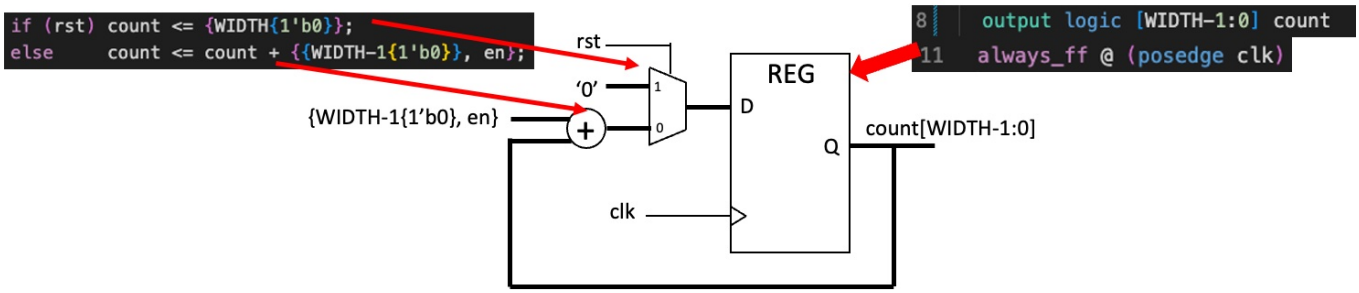



```
1  module counter #(
2    parameter WIDTH = 8
3  )(
4    // interface signals
5    input  logic             clk,      // clock
6    input  logic             rst,      // reset
7    input  logic             en,       // counter enable
8    output logic [WIDTH-1:0] count     // count output
9  );
10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

Note the following:

1. The file name and the module name must be the same.
2. The number of bits in the counter is specified with the parameter WIDTH. It is currently set to 8-bit.
3. The always_ff @ (posedge clk) is the way that one specifies a clocked circuit.
4. '<=' in line 12 and 13 are non-block assignments which should be used within an always_ff block.
5. {WIDTH{1'b0}} in line 12 uses the concatenation operator { } to form WIDTH bits of '0'. (Can you explain the construct in line 13?)

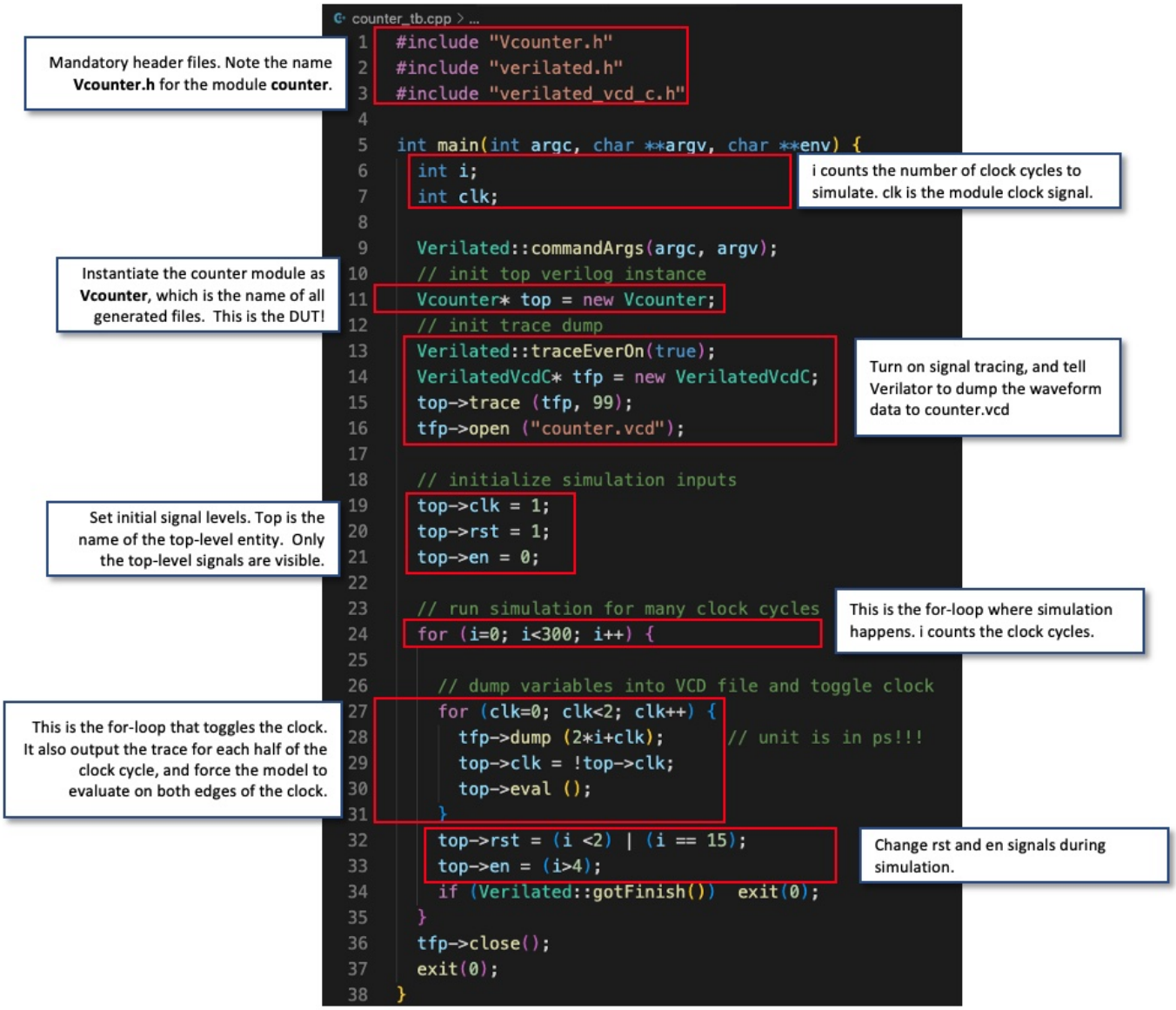
Here is the mapping between System Verilog and the counter circuit "synthesized" via Verilator:



Step 4: Create the testbench file ***counter_tb.cpp*** in C++ using VS Code.

We need to do this before we can combine everything to make the executable model. The listing for ***counter_tb.cpp*** is shown below. Again, you are required to type this in (instead of copying) to make sure that you think about each line of this program.

This testbench file is a template for all other testbench files. It consists of various sections, which are mandatory (except for the trace dump section if you don't want to see the waveforms). Make sure that you understand what each section is for!



Step 5: Compile the System Verilog model with the testbench

Open a terminal window and enter the following command:

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

This runs Verilator to translate counter.sv into C++ code, and merge with ***counter_tb.cpp*** to produces a number of files in a new folder: **obj_dir**. It also automatically generates a ***.mk*** file called ***Vcounter.mk***, which will produce the final simulation model ***Vcounter***.

Next enter:

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

This makes ***Vcounter***, which is the executable model of our counter!

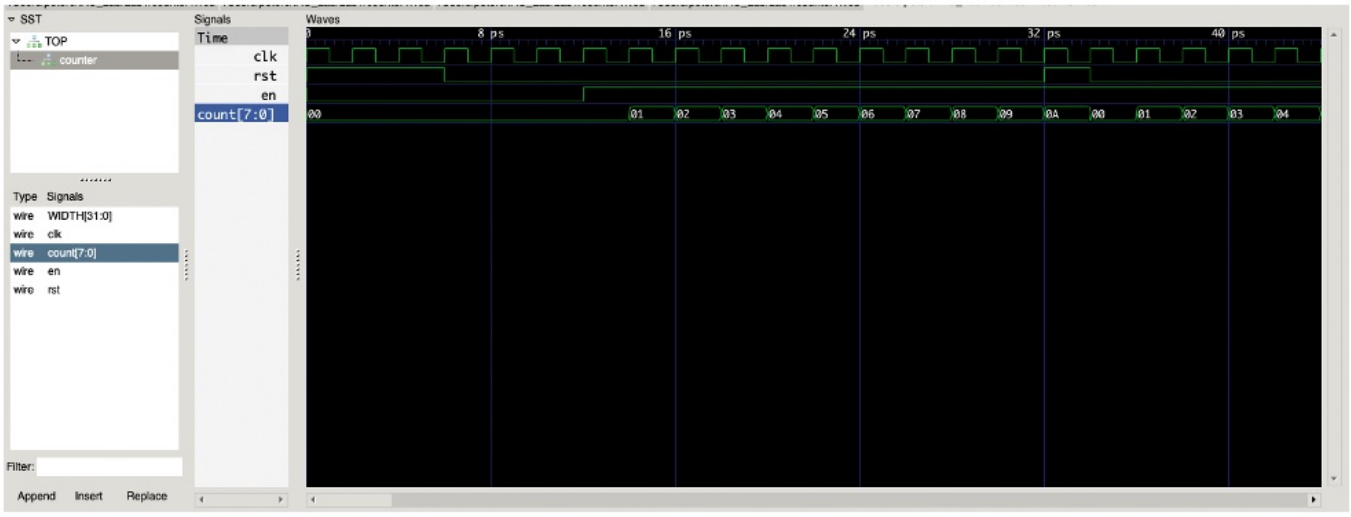
We are now ready to simulate by simply running ***Vcounter***, which is again in the ***obj_dir/.*** directory, by entering:

```
# run executable simulation file
obj_dir/Vcounter
```

This runs very fast, and you may think that nothing happened. However, if you examine the folder task1, you will see a file ***Vcounter.vcd*** has been generated. This is the trace waveform file and contains the simulation results.

Step 6: Plot the counter waveforms with GTKwave:

Start the GTKwave program on your laptop. Select *File -> Open New Tab* and select ***Vcounter.vcd*** file. A GTKwave window will appear. Click *Top -> counter*, followed by the signals: ***clk, rst, en*** and ***count[7:0]***. Use the + and - icons to adjust the zoom level. You should see the following.



Make sure you understand all waveform signals.

Why is the time axis in ps? Does it matter?

Congratulations! You have successfully simulated your first System Verilog digital hardware module.

Shortcut

To avoid retyping all the commands to make the executable model, create a shell script doit.sh that contains the following lines:

You can now run all the commands in one go by typing:

```
$ 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
```

You can now run all the commands in one go by typing:

```
source ./doit.sh
```

Alternatively, you can change the permission of ***doit.sh*** to executable with the ***chmod +x doit.sh*** command, and simply type

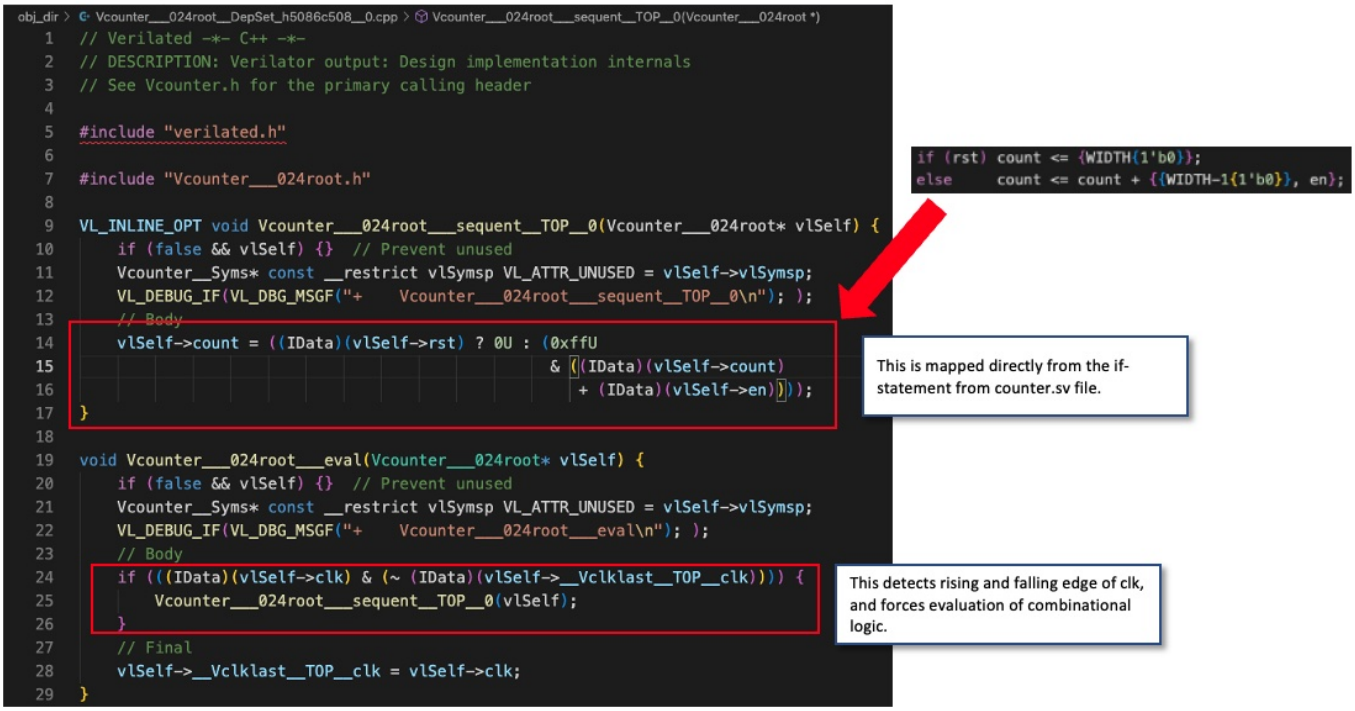
```
./doit.sh
```

to run the script

Deep Dive

Verilator produces the executable model ***obj_dir/Vcounter*** and the trace waveform file ***./counter.vcd***. In addition, there are many other files in the ***obj_dir*** directory. Details of what these files contains can be found on: https://verilator.org/guide/latest/files.html.

Explore what files are created by Verilator in **_obj_dir/ _** and now open in VS code the file: ***Vcounter__024root.....0.cpp***.



TEST YOURSELF CHALLENGES:

1. Modify the testbench so that you stop counting for 3 cycles once the counter reaches 0x9, and then resume counting. You may also need to change the stimulus for *rst*.
2. The current counter has a synchronous reset. To implement asynchronous reset, you can change line 11 of counter.sv to detect change in *rst* signal. (See notes.)

Make these modification, compile, and run. Examine the waveform with GTKwave and explain what you see.

Task 2: Linking Verilator simulation with Vbuddy

In this task, you will learn how to modify the testbench to interface with the Vbuddy board. Before you start, copy ***counter.sv*** and ***counter_tb.cpp*** to the task2 folder.

You also need to make a copy of the files ***Vbuddy.cpp*** and ***Vbuddy.cfg*** from your Lab 1 folder to here.

Step 1: Set up the Vbuddy interface

Connect Vbuddy to your computer's USB port using a USB cable provided. Find out the name of the USB device used. How? This depends on whether you are using a MacBook or a PC.

Macbook Users For Mac users, enter:

```
ls /dev/tty.u*
```

You should see device name as something similar this:

```
/dev/tty.usbserial-110
```

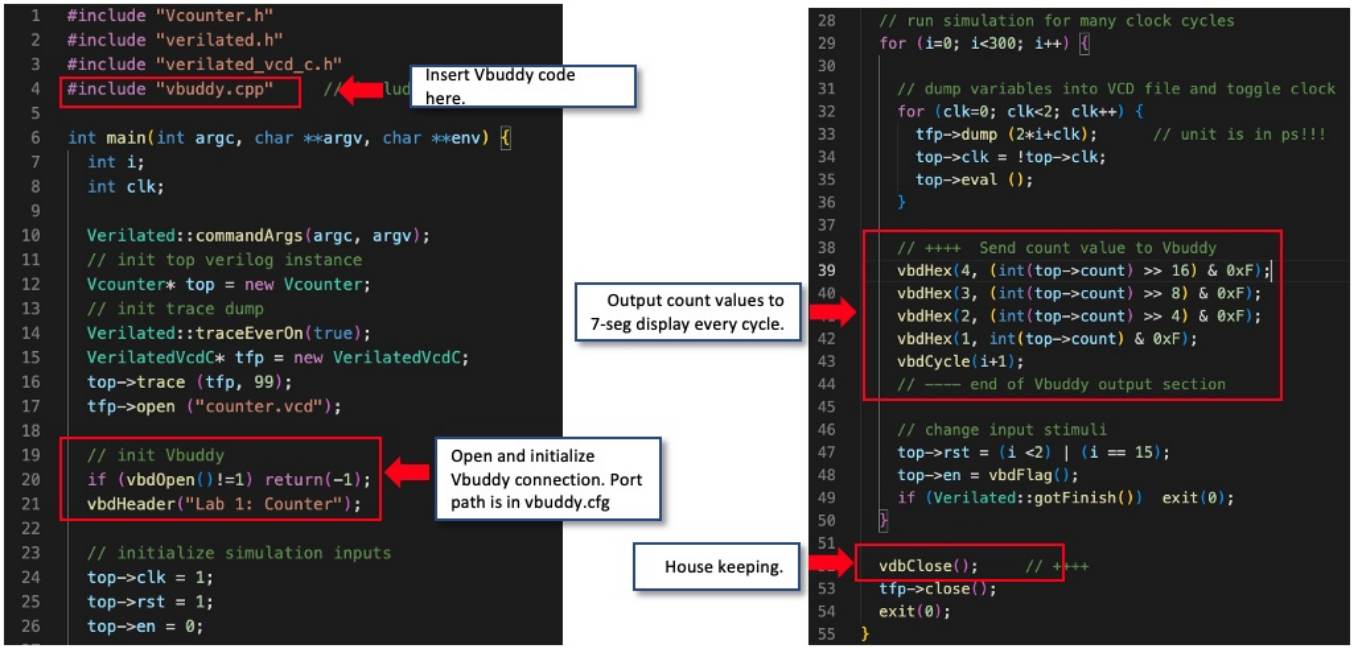
Window Users (This instruction for PC is to be updated.) For PC users, you need to find the COMS Port used to connect to Vbuddy. To do this, open the Device Manager, and you should see which COMS port (e.g. COM1 or COM6) is connect to Vbuddy via the USB cable.

Create a file: ***vbuddy.cfg***, which contains the device name as the only line (terminated with CR). You may use VS Code or the Linux nano editor to do this.

Step 2: Modify testbench for Vbuddy

Copy ***counter.sv*** and ***count_tb.cpp*** to your task2 folder.

Modify the testbench file ***count_tb.cpp*** to include **Vbuddy function** as shown in the diagram below. What they do are self-explanatory. You can find a list of the current Vbuddy functions which you can include in the testbench file here. These functions are of the form ***vbdXxxx***.



Recompile and test.

Step 3: Explore flag feature on Vbuddy

Vbuddy's rotary encode has a push-button switch. Vbuddy keeps an internal flag which, by default, will toggle between '0' and '1' each time the button is pressed. You can interrogate this toggle switch state with ***vbdFlag();***, which will return its current state and then toggle. A little postbox showing the flag state is drawn in the footer of the TFT display.

You can use this feature to enable and disable the counter by modifying line 48 of the testbench with:

```
    top->en = vbdFlag();
```

Re-compile and test.

Instead of showing count values on 7-segment displays, you may also plot this on the TFT by replacing the ***vdbHex()*** section with the command ***vbdPlot()***. You may want to increase the number of clock cycles to simulate because plotting a dot is much faster than outputting to the 7-segment display. You can start/stop the counter with the flag.

```
    vbdPlot(int(top->count), 0, 255);
```

TEST YOURSELF CHALLENGE:

Modify your counter and testbench files so that the ***en*** signal controls the direction of counting: '1' for up and '0' for down, via the ***vbdFlag()*** function.

WELL DONE! YOU MANAGE TO SPECIFY AND VERIFY YOUR DESIGN AND USE VBUDDY TO SEE THE RESULTS OF YOUR CIRCUIT. THE NEXT TWO TASK: TASK3 and TASK4 ARE OPTIONAL. YOU CAN SKIP THEM IF YOU RUN OUT OF TIME.

Task 3: Vbuddy parameter & flag in one-shot mode (OPTIONAL)

The rotary encodes (EC11) provides input from Vbuddy to the Verilator simulation model. Turning the encoder changes a stored parameter value on Vbuddy independently from the Verilator simulation. This parameter value can be read using the ***vbdValue( )*** function, and is displayed on the bottom left corner of the TFT screen in both decimal and hexadecimal format.

Step 1: Loadable counter

Copy across to the task3 foler the required files: ***counter.sv, counter_tb.cpp, vbuddy.cpp, doit.sh*** and ***vbuddy.cfg***.

Vbuddy's flag register has two modes of operation. The default mode is **TOGGLE**, which means that everything the rotary encoder switch is pressed, the flag will toggle as indicated at the bottom of the TFT screen.

However, using the ***vbdSetMode(1)*** function, you can set the mode to ONE-SHOT behaviour. Whenever the switch is pressed, the flag register is set to '1' as before – now the flag is **"ARMED"** ready to fire. However, when the flag register is read, it immediate resets to '0'.

Modify **counter.sv** so that pressing the switch on EC11 forces the counter to pre-set to Vbuddy's parameter value. (How?) Compile and test your design.

Step 2: Single stepping

Using the one-shot behaviour of the Vbuddy flag, it is possible to provide one clock pulse each time you press the rotary encoder switch. In other words, you can single step the counting action.

Modify **counter.sv** so that you only increment the counter each time you press the switch.

Task 4: Displaying count as Binary Coded Decimal (BCD) numbers (OPTIONAL)

So far, you have only simulated a single module. In this task, you will create a top-level module (**top.sv**), which has the counter module, and a second module that converts the 8-bit binary number into three BCD digits. You can find out how the binary to BCD conversion algorithm works from the course webpage.

Copy and paste the following code to **top.sv**:

```
module top #(
  parameter WIDTH = 8,
  parameter DIGITS = 3
)(
  // interface signals
  input  wire             clk,      // clock
  input  wire             rst,      // reset
  input  wire             en,       // enable
  input  wire [WIDTH-1:0] v,        // value to preload
  output wire [11:0]      bcd       // count output
);

  wire  [WIDTH-1:0]       count;    // interconnect wire

counter myCounter (
  .clk (clk),
  .rst (rst),
  .en (en),
  .count (count)
);

bin2bcd myDecoder (
  .x (count),
  .BCD (bcd)
);

endmodule
```

Modify the testbench file ***top_tb.cpp*** accordingly.
Modify the ***doit.sh*** file from task 3 to include all the modules (***top.sv, counter.sv, bin2bcd.sv*** and ***top_tb.sv***). Compile and run the Verilated model. Check that it works according to expectation.