IAC Lab 1

Task 1

Started off by creating a System Verilog file, `counter.sv`, which performs the functions of a counter module. Within the System Verilog file, we specify the interface signals -

- `clk` for the clock input signal
- `rst` for the reset input signal
- `en` for the counter enable input signal
- `count[7:0]` for the 8-bit counter output signal

```
module counter #(
  parameter WIDTH = 8
)(
  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // counter enable
  output logic [WIDTH-1:0] count     // count output
);

always_ff @ (posedge clk)
  if (rst) count <= {WIDTH{1'b0}};
  else     count <= count + {{WIDTH-1{1'b0}}, en};

endmodule
```

We also specify the logic which occurs on the positive, or rising edge of the clock signal -

- If the reset signal is activated, the count signal outputs 0. This is represented by the statement below, which sets a fixed number of bits equal to the width of the counter output signal (in this case 8 bits) to the binary value 0.

  `if (rst) count <= {WIDTH{1'b0}}`

- Else, the count signal is incremented by the below expression -

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

  We concatenate the enable signal `en`, which we know to be `1`, and `{{WIDTH-1{1'b0}}` which represents an 8-bit array of zeros as covered earlier. This gives us 1, hence the count signal increases by 1.

With the System Verilog file for our component in place, we develop a C++ testbench file, which will specify the inputs, simulate the component given the inputs, and dump the results into a `.vcd` file which we can easily visualise later on.

- We start by declaring the mandatory header files `Vcounter.h`, `verilated.h` and `verilated_vcd_c.h`
- In the main function, we initialise our counter module `Vcounter` as the variable `top`.
- We then dump the waveform data generated into a `counter.vcd` file, which would be used for viewing on GTKWave.
- We set the initial clock signal, `clk` to be 1, enable signal, `en` to be 0, and reset signal `rst` to be 1.
- We create a for-loop which runs the simulation for 300 clock cycles, evaluating and dumping the input and output signal on each clock edge.

```
// Mandatory header files
#include "Vcounter.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

int main(int argc, char **argv, char **env){

    Verilated::commandArgs(argc, argv);

    // Initialise counter module as Vcounter
    Vcounter* top = new Vcounter;

    // Dump waveform data to counter.vcd
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;
    top->trace(tfp, 99);
    tfp->open("counter.vcd");

    // Set initial signal levels
    top->clk = 1;
    top->rst = 1;
    top->en = 0;

    // Run simulation for 300 clock cycles
    for (int i=0; i<300; i++){

        // Toggle the clock twice, evaluating on both edges
        for (int clk=0; clk<2; clk++){
            tfp->dump(2*i+clk);
            top->clk = !top->clk;
            top->eval();
        }

        // Change rst and en signals during simulation
        top->rst = (i<2) | (i==5);
        top->en = (i>4);
        if (Verilated::gotFinish()) exit(0);
    }
    tfp->close();
    exit(0);
}
```

We change the input signals during simulation -

- The reset signal, `rst`, is high before the 2nd clock cycle (excluding) and on the 5th clock cycle.

  `top->rst = (i<2) | (i==5)`
- The enable signal, `en`, is high after the 4th clock cycle (excluding)

  `top->en = (i>4)`

We proceed to translate the `counter.sv` Verilog file into C++ and integrate it with the testbench code, then build and run the project.

```
# Run Verilator to translate Verilog into C++, including C++ testbench
```

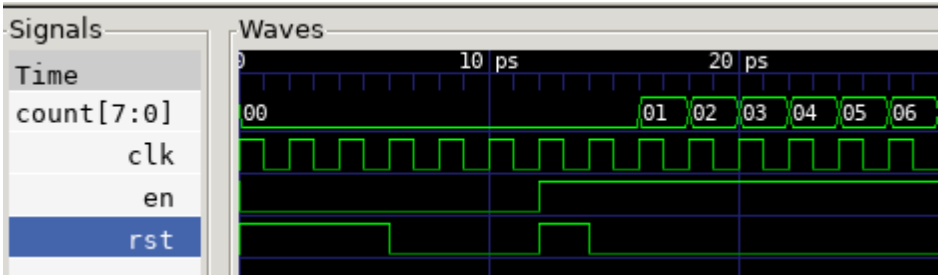
```
verilator -Wall --cc --trace counter.sv --exe counter_tb.cp
p

# Build C++ project via make automatically generated by Ver
ilator
make -j -C obj_dir/ -f Vcounter.mk Vcounter

# Run executable simulation file
obj_dir/Vcounter
```

Viewing `counter.vcd` on GTKWave, we observe the following -

- The input signals follow the logic conditions specified in counter_tb.cpp. The enable signal only activates on the rising edge of the 5th clock cycle (note that clock cycles start from 0 and the clock signal is initially high to begin with)
- However, as the reset signal is also high on the 5th cycle, the counter only starts incrementing on the 7th clock cycle.



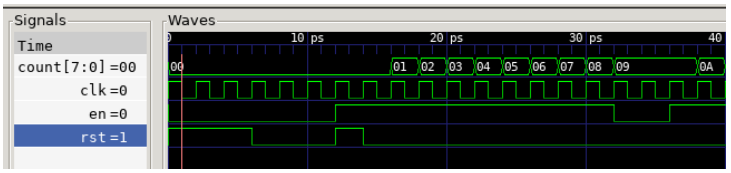
The time axis is in picoseconds as specified by the `$timescale` variable in `counter.vcd` as generated by Verilated.

```
$version Generated by VerilatedVcd $end
$date Thu Oct 10 12:04:56 2024 $end
$timescale 1ps $end
```

As a challenge, we modify the testbench to stop counting for 3 cycles once the counter reaches 0x9, and then resume counting. We do this by implementing further logic for the enable signal, deactivating it between the 15th and 16th clock cycle inclusive.

We start from the 15th cycle as the counter idles for 6 cycles, and takes 9 cycles to increment to 0x9. As the counter is synchronous, changes will not take place till the next clock cycle, hence we only deactivate the enable signal for 2 cycles.

```
// Change rst and en signals during simulation
top->rst = (i<2) | (i==5);
top->en = (i>4) and (i<15 or i>16);
if (Verilated::gotFinish()) exit(0);
```



Another challenge is to modify the current counter to support asynchronous reset, meaning that the counter resets on the same clock cycle as the reset signal when the reset signal is asserted.

We do this by specifying the flip-flop to update on both the rising edge of the clock and the reset signal.

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

During the rising edge of either signal, if the reset signal `RST` is asserted, the count is set to an array of zeroes, with the length of the `WIDTH` parameter (8).

Else, the count is set to increment by the concatenation of the array of zeroes and the enable signal. If the enable signal `EN` is asserted, the expression `{{WIDTH-1{1'b0}}, en}` resolves to 1, and the counter increases by 1.

Task 2

I then proceeded to make a copy of `Vbuddy.cpp` and `Vbuddy.cfg` from the GitHub repository on my local folder. I connected the VBuddy to my computer using a USB cable and ran the following script and listed the devices which are currently connected to the computer -

```
~/Documents/iac/lab0-devtools/tools/attach_usb.sh
ls /dev/ttyU*
```

I was able to observe a device `/dev/ttyUSB0` and inserted that line into the `vbuddy.cfg` file.

I then modified the testbench file `count_tb.cpp` to include functions to integrate it with the VBuddy.

- We insert `vbuddy.cpp` as a dependency at the start

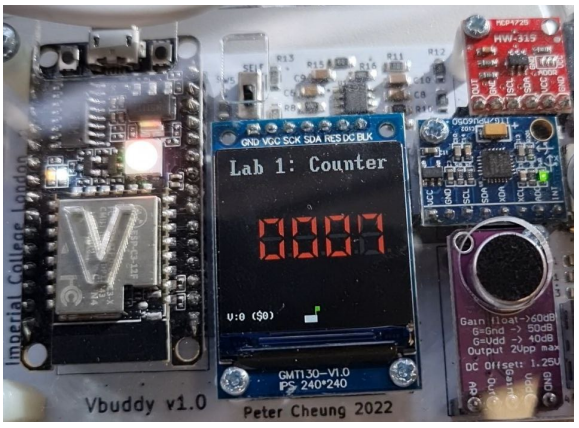
```
// Mandatory header files
#include "Vcounter.h"
#include "verilated.h"
#include "verilated_vcd_c.h"
#include "vbuddy.cpp"
```

- We insert code which opens and initializes the VBuddy connection. If VBuddy is not open the code returns. The header on the display of VBuddy is also specified.
- We obtain the count value of the counter at each clock cycle and output the value to the VBuddy display, which mimics a 7-segment display. For VBuddy display, which is able to display up to a 16-bit integer, we will right-shift 4, 8 and 12 bits and apply a bitmask which captures only the last 4 significant bits (`0xF = 0b0000 0000 0000 1111`) to get each 4 bit segment, which will be converted to a hexadecimal digit to be displayed.
- We then close the VBuddy connection with `vbdClose()`.

```
// Initialise VBuddy
if (vbdOpen() != 1) return (-1);
vbdHeader("Lab 1: Counter");


// Send count value to VBuddy
vbdHex(4, (int(top->bcd) >> 12) & 0xF);
vbdHex(3, (int(top->bcd) >> 8) & 0xF);
vbdHex(2, (int(top->bcd) >> 4) & 0xF);
vbdHex(1, int(top->bcd) & 0xF);
vbdCycle(i+1);
```

Compiling and loading the code onto VBuddy, we observe the display incrementing every clock cycle.



I then proceeded to explore the flag feature of VBuddy. VBuddy has a rotary encoder which also has a push-button switch.

The function `vbdFlag()` will return its current state of the push-button and we assign it to the enable input of the counter.

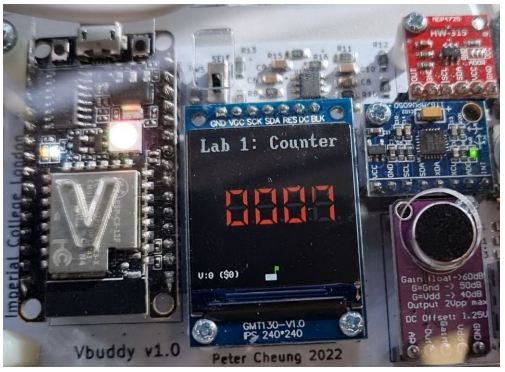
```
top->en = vbdFlag();
```

A little postbox showing the flag state is drawn in the footer of the TFT display shown on the right.

On recompilation, the counter initially starts incrementing. However, when we push down the button stops and the flag on the bottom of the display turns from green to red.

Instead of showing count values on 7-segment displays, I proceeded to plot this on the TFT by commenting out the `vdbHex()` section with the command `vbdPlot()`. This allows us to obtain a line display (which comprises of individual dots).

```
/* Send count value to VBuddy
vbdHex(4, (int(top->bcd) >> 12) & 0xF);
vbdHex(3, (int(top->bcd) >> 8) & 0xF);
vbdHex(2, (int(top->bcd) >> 4) & 0xF);
vbdHex(1, int(top->bcd) & 0xF);
*/
vbdPlot(int(top->count), 0, 255);
vbdCycle(i+1);
```

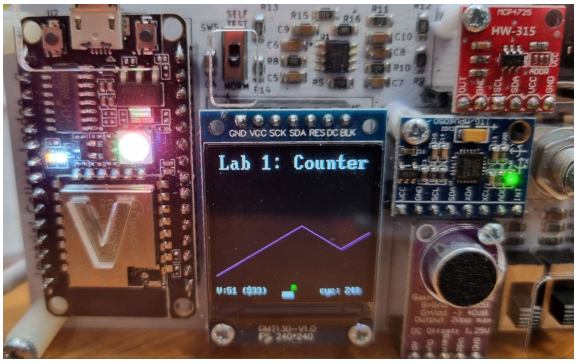


I moved onto the test challenge, which was to modify the counter and testbench files so that the enable signal `(EN)` controls the direction of counting - `1` for up and `0` for down, using `vbdFlag()` function.

As the `vbdFlag()` function is linked to the EN signal, we instead change the `counter.sv` file to reflect the following -

```
always_ff @ (posedge clk)
    if (rst) count <= {WIDTH{1'b0}};
    else if (en) count <= count + {{WIDTH-1{1'b0}},1'b1};
    else count <= count - {{WIDTH-1{1'b0}},1'b1};
```

- If the reset signal RST is asserted, count resets to 0
- If the enable signal EN is asserted, count increments by 1
- Else if the enable signal EN is de-asserted, the count decrements by 1



Loading the code into VBuddy, we observe that the line slopes up initially, then down when the button is pressed.

Task 3

The rotary encoder changes a stored parameter value on VBuddy, read using `vbdValue()` and displayed on the bottom left corner of the TFT screen in both decimal and hexadecimal format.

We seek to implement a loading function in the counter, such that when the rotary encoder is pressed, the stored parameter value is loaded into the counter. We start off with an edited version of counter.sv with a few changes -

```
module counter #(
  parameter WIDTH = 8
)(
  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             ld,       // load counter from data
  input  logic [WIDTH-1:0] v,        // value to preload
  output logic [WIDTH-1:0] count     // count output
);

always_ff @ (posedge clk)
  if (rst) count <= {WIDTH{1'b0}};
  else     count <= ld ? v : count + {{WIDTH-1{1'b0}},1'b1}
endmodule
```

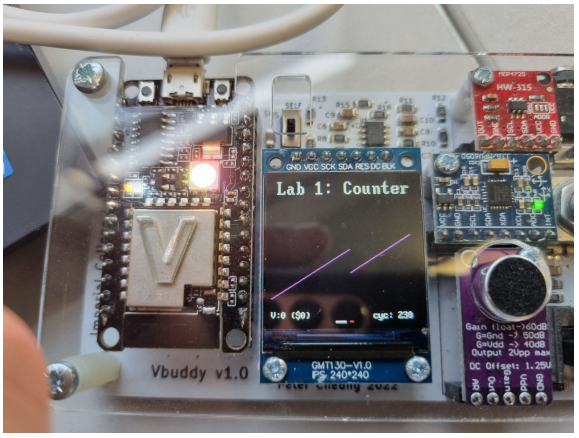
- The enable signal `EN` is replaced with a load signal `LD` . When asserted, the load signal will preload the value of the encoder.
- At each rising edge of the clock, if the load signal is asserted the count equals the preloaded value v, else the count increments by 1. This is nicely summed up in the ternary statement -

```
count <= ld ? v : count + {{WIDTH-1{1'b0}}, 1'b1}
```

We also specify `vbdSetMode(1)` to set the rotary encoder to one-shot behaviour. When the switch is pressed, the flag register is set to `1`. However, by the next clock cycle, the flag register is read and resets to `0`.

After recompiling and loading the program, we are able to preload the value of the counter to the parameter value of the rotary encoder when we press down on the encoder. This allows us to obtain the waveform as shown where we depress the encoder at parameter value of about `0x40`, or `(64)`

```
// Initialise VBuddy
if (vbdOpen() != 1) return (-1);
vbdHeader("Lab 1: Counter");
vbdSetMode(1);
```



Using the one-shot behaviour of the VBuddy flag, we can also provide one clock pulse each time you press the rotary encoder switch, thereby single stepping the counting action.

To do so, we modify `counter.sv` by adding this line, where we look out for the positive edge of the load signal `LD` instead of the clock signal `CLK`.

```
always_ff @ (posedge ld)
  if (rst) count <= {WIDTH{1'b0}};
  else     count <= count + {{WIDTH-1{1'b0}},1'b1}
```

Task 4

Lastly, I downloaded from the GitHub repository a top-level module `top.sv`, containing the counter module, and a second module that converts the 8-bit binary number into three BCD digits, `bin2bcd.sv`.

- The top-level module specifies the overall input signals (`CLK`, `RST`, `EN`) and output signals (`BCD`). To avoid unused signals we can delete `V`, the value to preload as there are no signals to load now.
- A logic connector, `count`, is specified to link the output of the counter to that of the BCD decoder.

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

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

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

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

endmodule
```

We also initialise the counter as `Vtop` instead of `Vcounter` as the name of the class now changes, and use the corresponding input and output signals to interface with it as specified by `top.sv`.

In the C++ testbench file, we also change the mandatory file from `Vcounter.h` to `Vtop.h`. This is because as the top-level System Verilog file is now `top.sv`, `Vtop.h` will be generated as the C++ translation.

```
// Mandatory header files
#include "Vtop.h"
#include "verilated.h"
#include "verilated_vcd_c.h"
#include "vbuddy.cpp"

int main(int argc, char **argv, char **env){

    Verilated::commandArgs(argc, argv);

    // Initialise counter module as Vcounter
    Vtop* top = new Vtop;
```

As the counter increments up till 256 (8 bits, so 2^8), we will need 12 bits of BCD (binary-coded decimal) to represent 3 groupings of 4 bits, each grouping for the hundreds, tens and ones bit. We can obtain them individually using the bitwise-shift and bitmask method as explained earlier.

```
// Send count value to VBuddy
vbdHex(3, (int(top->bcd) >> 8) & 0xF);
vbdHex(2, (int(top->bcd) >> 4) & 0xF);
vbdHex(1, int(top->bcd) & 0xF);
vbdCycle(i+1);
```

Lastly, we include all System Verilog files into the `generate.sh` script and specify `top_tb.cpp` instead of `counter_tb.cpp` as the testbench C++ file. We then specify the make directory and make file as `Vtop` and `Vtop.mk` respectively.

```
#!/bin/sh

# Cleanup old directories
rm -rf obj_dir
rm -f counter.vcd

# Run Verilator to translate Verilog into C++, including C++ testbench
verilator -Wall --cc --trace top.sv bin2bcd.sv counter.sv --exe top_tb.cpp

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

# Run executable simulation file
obj_dir/Vtop
```

Compiling and running it, we see that our counter increments in base 10 instead of base 16.