IAC Lab 2

Task 1

Started off the lab by specifying a 256×8 bit ROM (Read-Only Memory) component in System Verilog. The component contains an -

- 8-bit address input, `ADDR`
- 8-bit output, `DOUT`
- and a clock input, `CLK`

Initially, the ROM would load `sinerom.mem`, a file containing the individual bits of a cosine wave, scaled between 0 and 256.

The individual bits are generated by `sinegen.py`, a Python file, which iterates through a single cosine wave, splits it up into 256 intervals and at each interval, saves the value of the cosine wave (between 0 and 1) and scales it 256 times.

```
module rom #(
    parameter ADDRESS_WIDTH = 8,
              DATA_WIDTH = 8
)(
    input logic clk,
    input logic [ADDRESS_WIDTH-1:0] addr,
    output logic [DATA_WIDTH-1:0] dout
);

logic [DATA_WIDTH-1:0] rom_array [2**ADDRESS_WIDTH-1:0];

initial begin
        $display("Loading ROM");
        $readmemh("sinerom.mem", rom_array);
end;

always_ff @(posedge clk)
    dout <= rom_array[addr];

endmodule
```

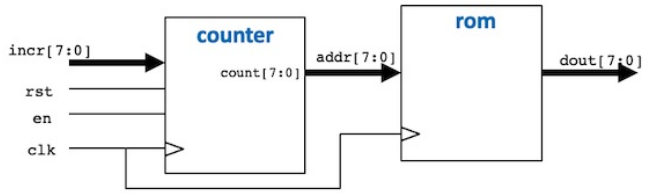
```
import math
import string
f = open("sinerom.mem","w")
for i in range(256):
    v = int(math.cos(2*3.1416*i/256)*127+127)
    if (i+1)%16 == 0:
        s = "{hex:2X}\n"
    else:
        s = "{hex:2X} "
    f.write(s.format(hex=v))

f.close()
```

```
FE FD FD FD FD FD FC FC FB FA FA F9 F8 F7 F6 F5
F4 F3 F1 F0 EF ED EB EA E8 E6 E5 E3 E1 DF DD DA
D8 D6 D4 D1 CF CD CA C8 C5 C2 C0 BD BA B8 B5 B2
AF AC A9 A6 A3 A0 9D 9A 97 94 91 8E 8B 88 85 82
7E 7B 78 75 72 6F 6C 69 66 63 60 5D 5A 57 54 51
4E 4B 48 45 43 40 3D 3B 38 35 33 30 2E 2C 29 27
25 23 20 1E 1C 1A 18 17 15 13 12 10  E  D  C  A
 9  8  7  6  5  4  3  3  2  1  1  0  0  0  0  0
 0  0  0  0  0  0  1  1  2  3  3  4  5  6  7  8
 9  A  C  D  E 10 12 13 15 17 18 1A 1C 1E 20 23
25 27 29 2C 2E 30 33 35 38 3B 3D 40 43 45 48 4B
4E 51 54 57 5A 5D 60 63 66 69 6C 6F 72 75 78 7B
7F 82 85 88 8B 8E 91 94 97 9A 9D A0 A3 A6 A9 AC
AF B2 B5 B8 BA BD C0 C2 C5 C8 CA CD CF D1 D4 D6
D8 DA DD DF E1 E3 E5 E6 E8 EA EB ED EF F0 F1 F3
F4 F5 F6 F7 F8 F9 FA FA FB FC FC FD FD FD FD FD
```

I then proceeded on to create a top-level System Verilog module, `sinegen.sv`, which contains both `counter.sv` which we had previous designed in Lab 1 and the above System Verilog file, `rom.sv`.

An 8-bit input which specifies the increment to the counter, `INCR`, is provided. For each clock cycle, the counter will increment by the value of `INCR` and the output of the counter will be fed to the ROM, `sineROM`, via an interconnecting wire `address`. The value of `address` will then specify the 8-bit value of the cosine wave to be output from memory.

```
module sinegen #(
    parameter ADDRESS_WIDTH = 8,
              DATA_WIDTH = 8
)(
  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // enable
  input  logic [ADDRESS_WIDTH-1:0] incr,
  output logic [DATA_WIDTH-1:0] dout
);

logic [ADDRESS_WIDTH-1:0] address;    // interconnect wire

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

rom sineROM (
  .clk (clk),
  .addr (address),
  .dout (dout)
);

endmodule
```



To adapt the new top-level module to our use case, we have to increase the number of cycles of the simulation to an arbitrarily large value, in this case 1000000 as specified by the line `int SIM_CYCLES = 1000000;`

We also plot the value of the output `dout` as a point with the line - `vbdPlot(int(sinegen->dout), 0, 255);`

Lastly, as the number of cycles is large and the program is unlikely to finish execution in a reasonable amount of time, we insert a `vbdGetKey()` function which will detect if the `q` key is pressed on the

```
int SIM_CYCLES = 1000000;
// Run simulation for 300 clock cycles
for (int i=0; i<SIM_CYCLES; i++){

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

    // Send count value to VBuddy
    vbdPlot(int(sinegen->dout), 0, 255);
    vbdCycle(i+1);
```

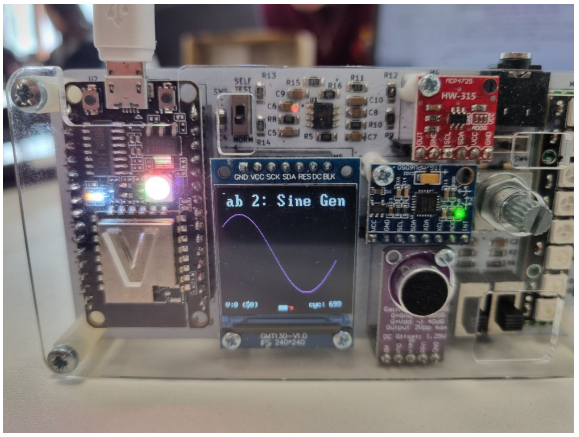
```
    // Change rst and en signals during simulation
    if (Verilated::gotFinish() || (vbdGetkey()=='q')) exit(0);
```

console, and if so, terminate and exit the program.

We then edit the shell script `generate.sh` to run the corresponding System Verilog files, with the top-level file `sinegen.sv` coming first, then the individual component files `rom.sv` and `counter.sv`, and the new testbench file `sinegen_tb.cpp`.

```
#!/bin/sh

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

# Run Verilator to translate Verilog into C++, including C++ testbench
# Top-level SV file should come first in cmd-line args
verilator -Wall --cc --trace sinegen.sv rom.sv counter.sv --exe sinegen_tb.cpp

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

# Run executable simulation file
obj_dir/Vsinegen
```

We then build the C++ project generated by Verilator, in this instance now called `Vsinegen`. Loading the code into the VBuddy connected to the computer via USB, as observed, we obtain a sine wave displayed on the VBuddy which varies with time.



As a challenge, we modify the design such that the `vbdValue()` function, which returns the value of the rotary encoder on VBuddy, is used as an input to change the frequency of the sinewave generated.

We define the range of possible values of the increment, from 1 to 10. The minimum increment `min_incr = 1` while the maximum possible increment `max_incr = 10`. We also define the range of possible values returned by `vbdValue()`, between 1 and 100. We set the initial increment value `INCR` to 1.

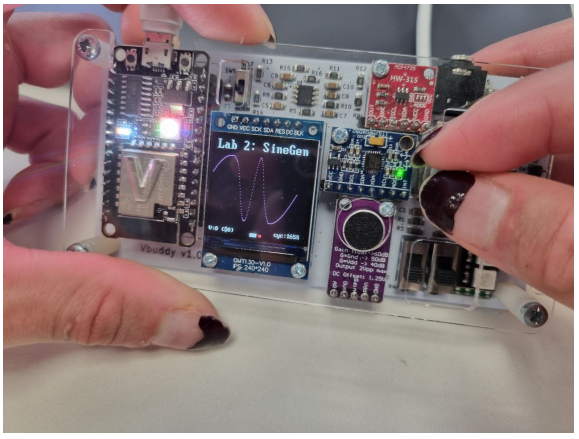
```
const int min_incr = 1;
const int max_incr = 10;
const int min_vbd_val = 1;
const int max_vbd_val = 100;

// Set initial signal levels
sinegen->clk = 1;
sinegen->rst = 0;
sinegen->en = 1;
sinegen->incr = min_incr;
```

At each clock cycle, we obtain the current `vbdValue()`, scale it to a decimal between 0 and 1, then multiply it by 100 to obtain a number between 0 and 100.

As a sanity check, we then ensure that it is within the maximum and minimum bounds of the possible increment value (if not, we set it to the maximum and minimum value respectively), then send it to the `sinegen` component.

This allows us to vary the frequency of the sine wave on the VBuddy as shown on the right.

```
int vbd_val = vbdValue();

double r = (double) (vbd_val - min_vbd_val) / (max_vbd_val - min_vbd_val);
int new_inc = r * (max_vbd_val - min_vbd_val) + min_vbd_val;
new_inc = new_inc < min_incr? min_incr : new_inc;
new_inc = new_inc > max_incr? max_incr: new_inc;
sinegen -> incr = new_inc;
```



Task 2

For this task, we seek to modify the design in Task 1 to generate 2 simultaneous sinusoidal signals which have different phases, where the phase offset between the two sinusoids is determined by the rotary encoder on VBuddy. We start off by copying the code from Task 1 to Task 2, then edited the ROM to be dual-port (accepts 2 addresses and outputs 2 values).

We simply duplicate the input parameter to include 2 8-bit inputs `addr1` and `addr2`, and output parameter to include 2 8-bit outputs `dout1` and `dout2`. Similar to `dout1`, `dout2` is assigned to output the bits at address `addr2` of the ROM `rom_array`.

```
module rom #(
    parameter ADDRESS_WIDTH = 8,
              DATA_WIDTH = 8
)(
    input logic clk,
    input logic [ADDRESS_WIDTH-1:0] addr1,
    input logic [ADDRESS_WIDTH-1:0] addr2,
    output logic [DATA_WIDTH-1:0] dout1,
    output logic [DATA_WIDTH-1:0] dout2
);

logic [DATA_WIDTH-1:0] rom_array [2**ADDRESS_WIDTH-1:0];

initial begin
        $display("Loading ROM");
        $readmemh("sinerom.mem", rom_array);
end;

always_ff @(posedge clk) begin
    dout1 <= rom_array[addr1];
    dout2 <= rom_array[addr2];
    end

endmodule
```

Lastly, we change the ROM to accept the corresponding interconnecting wires address1 and address2.

We change the corresponding top-level System Verilog design sheet `sinegen.sv` to include 2 outputs of 8 bits `dout1` and `dout2`. We also specify an additional 8-bit output offset which will receive values from the rotary encoder via `vbdValue()` and assign the value of `address2` to the sum of `address1` and the offset signal.

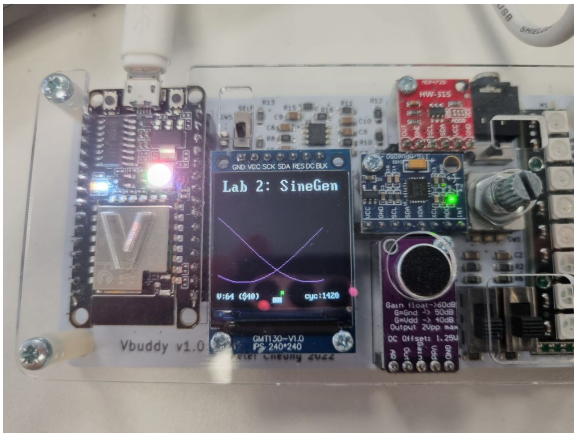
```
)(
  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // enable
  input  logic [ADDRESS_WIDTH-1:0] incr,
  input  logic [ADDRESS_WIDTH-1:0] offset,
  output logic [DATA_WIDTH-1:0] dout1,
  output logic [DATA_WIDTH-1:0] dout2
);

logic [ADDRESS_WIDTH-1:0] address1;    // interconnect wire
logic [ADDRESS_WIDTH-1:0] address2;
assign address2 = address1 + offset;
```

Lastly, we specify 2 plots for VBuddy by calling the function `vbdPlot()` twice, on the first output `dout1` and on the second output `dout2`.

```
// Send count value to VBuddy
vbdPlot(int(sinegen->dout1), 0, 255);
vbdPlot(int(sinegen->dout2), 0, 255);
vbdCycle(i+1);
```

We then compile and test the code on VBuddy and observed that the phase of the 2 signals can indeed be controlled by the rotary encoder.

Setting the offset value to 64, the 2 waveforms become 90 degrees apart in their phases, making one a sinewave and the other a cosine wave at the same frequency, as shown in the picture.



Task 3

For the last task, we seek to capture real audio signal using the microphone and amplifier module on VBuddy. The samples are then written to a dual-port RAM. Simultaneously, we read back from the RAM the store signal at a different address offset from the write address, such that the retrieved signal is a delayed version of the original signal.

We start off by implementing a RAM (random-access memory) component in System Verilog. Compared to ROM, there are some changes to the input and output signals of the RAM component.

```
module ram #(
    parameter ADDRESS_WIDTH = 9,
              DATA_WIDTH = 8
)(
    input logic clk,
    input logic readEn,
    input logic writeEn,
    input logic [ADDRESS_WIDTH-1:0] writeAddr,
    input logic [ADDRESS_WIDTH-1:0] readAddr,
    input logic [DATA_WIDTH-1:0] din,
    output logic [DATA_WIDTH-1:0] dout
);

logic [DATA_WIDTH-1:0] ram_array [2**ADDRESS_WIDTH-1:0];

always_ff @(posedge clk) begin
    if (writeEn == 1'b1)
        ram_array[writeAddr] <= din;
    if (readEn == 1'b1)
        dout <= ram_array[readAddr];
    end

endmodule
```

We change the inputs and outputs as such -

- `readEn` enables reading of RAM when asserted
- `writeEn` enables writing of RAM when asserted
- `writeAddr` specifies an 8-bit location to write data to, when `writeEn` is asserted
- `readAddr` specifies an 8-bit location to read data from, when `readEn` is asserted
- `din` specifies the 8-bit data to be written to `writeAddr`, when `writeEn` is asserted
- `dout` specifies the 8-bit data to be output from `readAddr`, when `readEn` is asserted

Observing the logic that occurs at the rising edge of the flip-flop, we see that the write operation takes precedence from the read operation.

The difference between the read and write addresses is the offset. We use the `vbdValue()` function to vary this offset using the rotary encoder.

On the top-level System Verilog schematic, sigdelay.sv, we specify the 8-bit input signal from the microphone as `mic_signal` and the delayed signal (delayed by the offset) as `delayed_signal`. Similar to the previous lab, we add the offset signal to the address we seek to read from.

However, now we specify the address returned by the counter as the read address `readAddr` and the address incremented by the offset as `writeAddr`.

This is so that data written to the RAM will always be written at a higher address than the data read from the RAM. After a fixed number of cycles (the offset) where no data is read (as no data has been written yet), the data read from the RAM will be prior data written to the RAM, hence achieving a delay.

```
)(
  // interface signals
  input  logic              clk,      // clock
  input  logic              rst,      // reset
  input  logic              wr,       // write
  input  logic              rd,       // read
  input  logic [ADDRESS_WIDTH-1:0] offset,
  input  logic [DATA_WIDTH-1:0] mic_signal,
  output logic [DATA_WIDTH-1:0] delayed_signal
);

logic [ADDRESS_WIDTH-1:0] address1;    // interconnect wire
logic [ADDRESS_WIDTH-1:0] address2;
assign address2 = address1 + offset;
```

```
ram sineRAM (
  .clk (clk),
  .readEn (rd),
  .writeEn (wr),
  .writeAddr (address2),
  .readAddr (address1),
  .din (mic_signal),
  .dout (delayed_signal)
);
```

> ⚠ The inverse can also work, where we assign data written to the RAM at a lower address to data read. However this means that the counter must read till the end of the RAM (512 bits), then overflow and return to zero, before data which was previously written can be read.
>
> This would mean a delay of `(512 - offset)` cycles.

We use ***sigdelay_tb.cpp,*** the provided C++ testbench file. We initialise VBuddy to accept microphone input, with buffer of 512 bits.

```
// intialize variables for analogue output
vbdInitMicIn(RAM_SZ);
```

Each clock cycle, we obtain the microphone sound value using `vbdMicValue()` and the rotary encoder value `vbdValue()`. We then plot both the microphone input signal `mic_signal` and delayed signal from RAM `delayed_signal` .

```
top->mic_signal = vbdMicValue();
top->offset = abs(vbdValue());     // adjust delay by changing incr

// plot RAM input/output, send sample to DAC buffer, and print cycle count
vbdPlot(int (top->mic_signal), 0, 255);
vbdPlot(int (top->delayed_signal), 0, 255);
vbdCycle(simcyc);
```

Playing a tone at 400Hz from an online tone generator website (Tone Generator), we obtain the following waveform. (For simplicity's sake we remove the microphone input waveform).