

Verilog Workshop

In this workshop, we will focus on sequential circuits over combinational circuits.

Counter

We will start by writing code for the counter and display this as a seven segment display.

Decoder

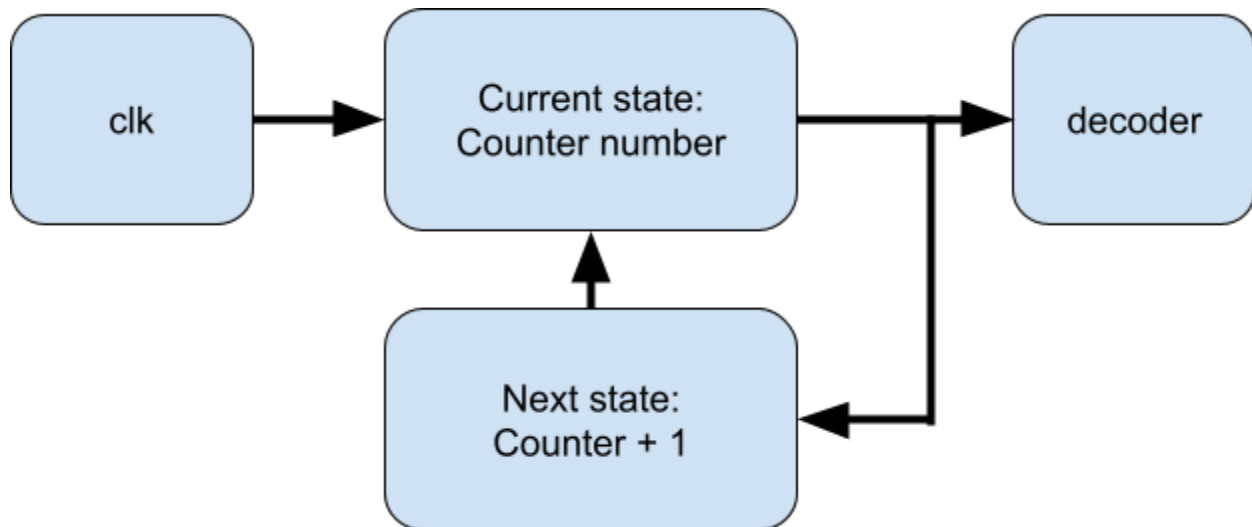
We first quickly construct a 4 bit to seven segment display decoder. This is a combinational circuit.

```
module binary2hex(  
    input [3:0] b,  
    output reg [7:0] seg  
);  
  
    always@(b) begin  
        case (b)  
            4'b0000: seg = 8'b00000011;  
            4'b0001: seg = 8'b10011111;  
            4'b0010: seg = 8'b00100101;  
            4'b0011: seg = 8'b01100001;  
            4'b0100: seg = 8'b10011001;  
            4'b0101: seg = 8'b01001001;  
            4'b0110: seg = 8'b01000001;  
            4'b0111: seg = 8'b00011111;  
            4'b1000: seg = 8'b00000001;  
            4'b1001: seg = 8'b00001001;  
            4'b1010: seg = 8'b00010001;  
            4'b1011: seg = 8'b11000001;  
            4'b1100: seg = 8'b01100011;  
            4'b1101: seg = 8'b10000101;  
            4'b1110: seg = 8'b01100001;  
            4'b1111: seg = 8'b01110001;  
        endcase  
    end  
endmodule
```

This is not the main focus of the workshop, so it's okay to just copy and paste this into a Verilog Module. The idea is basically every time any input changes, the section updates and the 7 bit segment updates. Depending on what the 4 bit is, the corresponding 7 segment arrangement lights up.

Counter

A counter is a sequential circuit, it needs to hold the value and respond accordingly. For instance, if a counter is currently at 6, it needs to hold the value 6 to know the next time someone presses the button, it turns 7.



For any module, we start with declaring inputs and outputs.

```
module counter(  
    input clk, reset,  
    output [7:0] C, // goes to the 7 segment cathode  
    output [3:0] A // goes to the 7 segment anode  
);
```

For any state machine, it is good practice to start with this format:

```
module counter(  
    input clk, reset,  
    output [7:0] C, // goes to the 7 segment cathode  
    output [3:0] A // goes to the 7 segment anode  
);  
  
// Signal declaration
```

```

reg [3:0] state, next_state;

// state register
always@(posedge clk, posedge reset) begin
    // put stuff here
end

// Next State Logic
always@* begin
    // put stuff here
end

// Output

endmodule

```

Fill in the required logic to construct a functional counter. If unsure, please reach out for help. This will be guided in the workshop.

UCF file

When this is completed, we will need the output to go into a spot on the FPGA development board. We need to write a UCF file. Syntax is as shown below.

```

NET "C[0]" LOC = "?";
NET "C[1]" LOC = "?";
NET "C[2]" LOC = "?";
NET "C[3]" LOC = "?";
NET "C[4]" LOC = "?";
NET "C[5]" LOC = "?";
NET "C[6]" LOC = "?";
NET "C[7]" LOC = "?";

NET "clk" LOC = "?";
NET "reset" LOC = "?";

NET "A[3]" LOC = "?";
NET "A[2]" LOC = "?";
NET "A[1]" LOC = "?";
NET "A[0]" LOC = "?";

```

The “?” are left empty deliberately as you need to learn to read the manual, this

will be guided in the workshop.

Then upload this on the board and see what happens

The counter doesn't really count properly, can we explain why?

Extension:

Write a module that can eliminate the problem we encounter. (Won't be covered in the workshop).

Musical Tone

We then proceed to use the switches to make the buzzer produce our desired sound.

Timer

A timer is needed to provide a correct digital frequency. We need a module such that the output oscillates depending on the input signal given.

This input signal needs to be a meaningful value that allows the system to understand the frequency it oscillates at.

We start by declaring input signal.

```
module timer(  
    input [31:0] period,  
    input clk,  
    input reset,  
    output reg sound  
);
```

Period is 32 bits because I said so (ensure it is enough to reach audible range).

We also need a constantly oscillating clock, which will come from the board itself. According to the manual, the board has an oscillating clock at 10MHz.

Note the output is a register, because the output itself is a state machine and it holds a value and it's not purely dependent on the input. This will be explained in the workshop.

Once again, we use this template to write a sequential circuit. Our timer clearly holds a value and hence is a sequential circuit.

```

module timer(
    input [31:0] period,
    input clk,
    input reset,
    output reg sound
);

// Signal declaration
reg [31:0] state, next_state;

// State register
always@(posedge clk, posedge reset) begin
    // put stuff here
end

// Next-state Logic
always@* begin
    // put stuff here
end

// Output Logic
always@* begin
    // put stuff here
end

endmodule

```

Once again, please fill in the blanks. This will be guided in the workshop.

Switches

We then need to configure switches that select a value to send into the timer. This is essentially a combination of a priority encoder and a decoder.

The correct tone parameters are defined here, locate this in the beginning of the code.

```

localparam C4 = 32'd191130;
localparam D4 = 32'd170262;
localparam E4 = 32'd151699;
localparam F4 = 32'd143184;
localparam G4 = 32'd127551;

```

```
localparam A4 = 32'd113636;  
localparam B4 = 32'd101235;  
localparam C5 = 32'd95547;
```

Declare the input and output of the module as usual.

Now to program a priority encoder, we can conduct it in this matter:

```
reg [31:0] period;  
// a priority encoder  
always@* begin  
    if (SW >= 8'b10000000)  
        period <= C5;  
    else if (SW >= 8'b01000000)  
        period <= B4;  
    else if (SW >= 8'b00100000)  
        period <= A4;  
    else if (SW >= 8'b00010000)  
        period <= G4;  
    else if (SW >= 8'b00001000)  
        period <= F4;  
    else if (SW >= 8'b00000100)  
        period <= E4;  
    else if (SW >= 8'b00000010)  
        period <= D4;  
    else if (SW >= 8'b00000001)  
        period <= C4;  
    else  
        period <= 0;  
end
```

Where period is a register that holds a value such that we can pass it into the timer.

Follow the workshop to finish off this section.

UCF file

Write a UCF file to configure this, please read the manual or follow the workshop. Upon completion, run this in the development board and see the working sound being made.