ECE 451 LAB 6

Pyramid Counter

Ishani Gowaikar
CSU ID: 831702439

Objective:

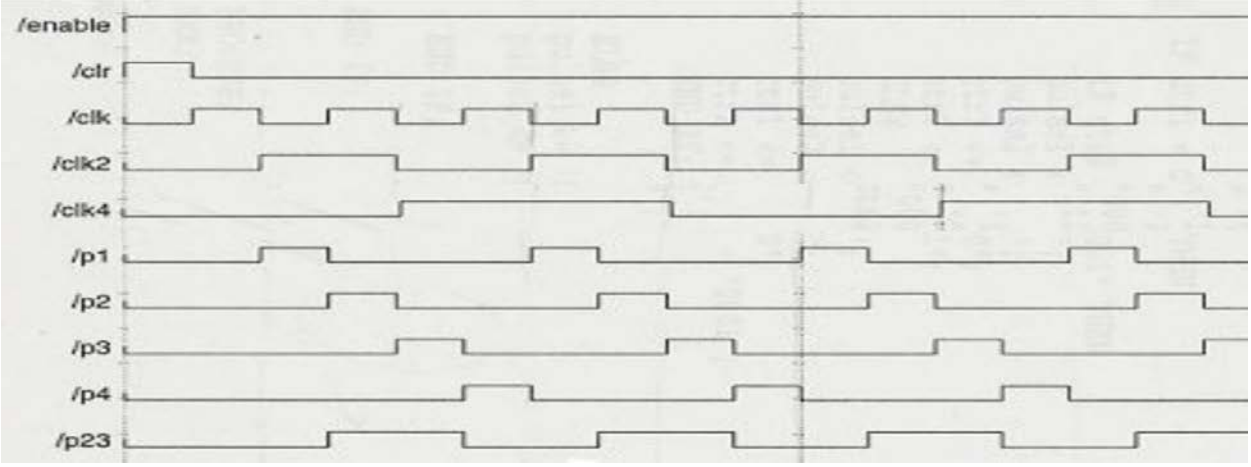
The objective of this lab is to apply sequential logic design techniques to design and build two different blocks, a clock generator and a pyramid counter.

Introduction:

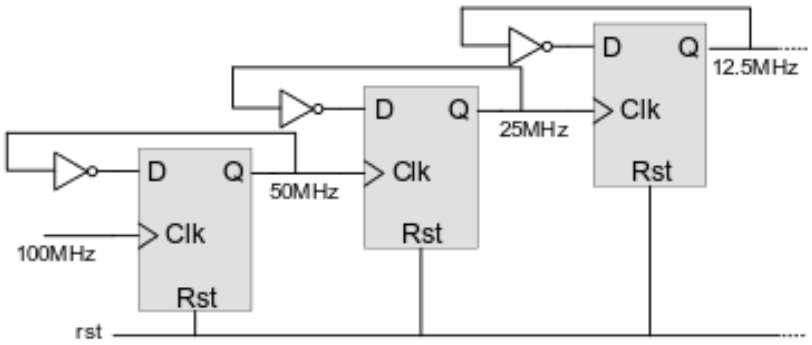
Pulse and Clock Generator:

The function of this circuit is to take an input clock signal and output the following signals:

- **Clk2**: A clock signal whose period is twice the input clock signal.
- **Clk4**: A clock signal whose period is four times the input clock signal.
- **P1, P2, P3 and P4**: Four different pulse signals whose pulse width is half the cycle period of the input clock signal and whose period is twice the input clock signal. In addition, only one of the four pulses must occupy the cycle period of these pulse signals are equally divided into four sub-phases (sub-periods) and each sub-phase.
- **P23**: A pulse signal whose period is the same as signals P1, P2, P3 and P4 but the width and the location of the pulse are equal to the sum of the second and third sub-phases.



The design used for generating the above clocks was the basic clock divider circuit



Pyramid Counter:

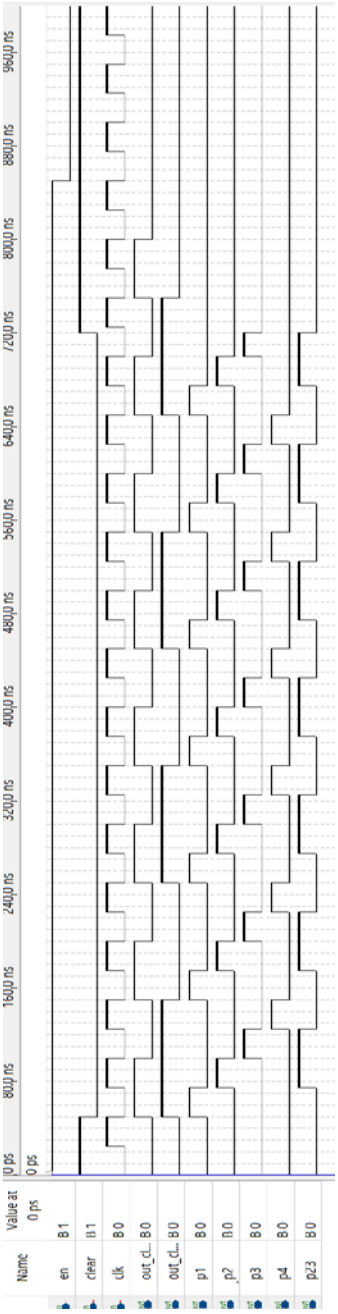
A counter that counts from 0 to 15, then pulses a control signal, Pulse1, which reconfigures the counter to count from 0 to 14 (or 1 to 15) and pulses the control signal again, then counts 0 to 13 (or 2 to 15) and pulses the control signal, etc. When the count goes from 0 to 1 (or 14 to 15) a second control signal, Pulse2, is sent.

Pulse Clock Code:

```
module pulse_clock (clk, out_clk1, out_clk2, out_clk3, out_clk4, en, clear, p1, p2, p3, p4, p23)

input clk;
input en;
input clear;

output reg out_clk1;
output reg out_clk2;
output reg out_clk3;
output reg out_clk4;


output reg p1, p2, p3, p4, p23;

wire [1:0] clock;

parameter limit1=1;

reg [25:0] count1 = 0;
reg [25:0] count2 = 0;



   always@(negedge clk)
   begin
      count1 = count1 + 1;
      if(count1 == limit1)
      begin
            count1 <= 0;
            out_clk1 <= ~out_clk1;
      end
   end

   always@(posedge out_clk1)
   begin
      count2 = count2 + 1;
      if(count2 == limit1) begin
            count2 <= 0    ;
            out_clk2 <= ~out_clk2;
      end
   end

   assign clock={out_clk1, clk};


   always@(clk)
      begin

      out_clk3 <= out_clk1;
      out_clk4 <= out_clk2;

      case (en)
```

```
53
54          1'b0:
55          begin
56             out_clk3 <= 1'b0;
57             out_clk4 <= 1'b0;
58
59          end
60
61                1'b1:
62
63                case (clear)
64                1'b1:
65                begin
66
67                   p1    <= 1'b0;
68                   p2    <= 1'b0;
69                   p3    <= 1'b0;
70                   p4    <= 1'b0;
71                   p23   <= 1'b0;
72                end
73
74
75                1'b0:
76
77                case(clock)
78                2'b00:
79                begin
80                   p1    <= 1'b0;
81                   p2    <= 1'b0;
82                   p3    <= 1'b1;
83                   p4    <= 1'b0;
84                   p23   <= 1'b1;
85                end
86
87                2'b01:
88                begin
89                   p1    <= 1'b0;
90                   p2    <= 1'b0;
91                   p3    <= 1'b0;
92                   p4    <= 1'b1;
93                   p23   <= 1'b0;
94                end
95
96                2'b10:
97                begin
98                   p1    <= 1'b1;
99                   p2    <= 1'b0;
100                  p3    <= 1'b0;
101                  p4    <= 1'b0;
102                  p23   <= 1'b0;
103               end
104
105             2'b11:
106             begin
107                p1    <= 1'b0;
108                p2    <= 1'b1;
109                p3    <= 1'b0;
110                p4    <= 1'b0;
111                p23   <= 1'b1;
112             end
113             endcase
114
115         endcase
116
117     endcase
118     end
119
120  endmodule
121
```

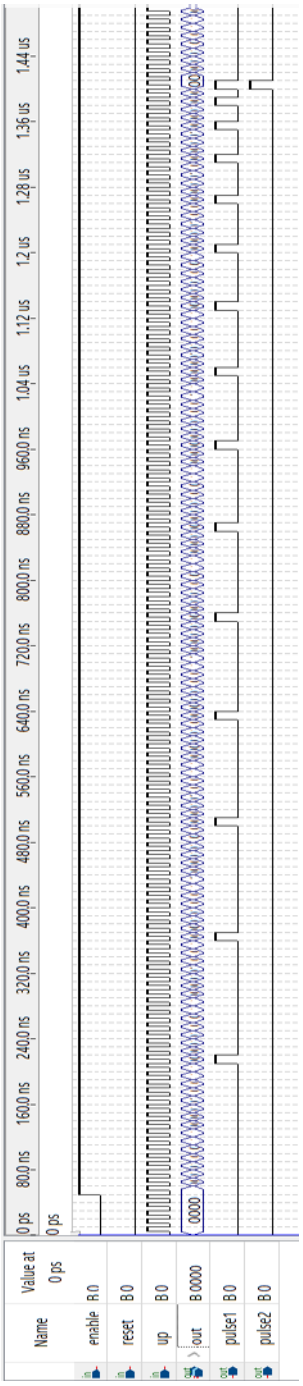
Pulse Clock Output:



Pyramid counter Code:

```
module pyramid_cntr (up, out, reset, pulse1, pulse2, enable);

input up, reset, enable;
output reg [3:0] out;
output reg pulse1, pulse2;

reg [3:0] temp = 4'b1111;
reg [3:0] flag = 4'b1111;

always @(negedge up)
   begin
      case(enable)
         1'b1:
         begin
            case(reset)
               1'b0:
               begin

                  out<= out + 1'b1;

                     case(out)
                        flag:
                        begin

                           pulse1 <= pulse1 + 1;
                           flag= flag-1;
                           out <= 4'b0000;

                              case(flag)
                                 4'b0000:
                                    pulse2 <= pulse2 + 1;

                                 default:
                                    begin
                                       pulse1 <= pulse1 + 1;
                                       pulse2 <= 0;
                                    end
                              endcase
                        end

               default:
               begin

                  pulse1 = 1'b0;
                  pulse2 = 1'b0;
               end
               endcase
               end
               1'b1:
                  out <= 4'b0000;
            endcase
               1'b0:
               begin
                  pulse1 = 1'b0;
                  pulse2 = 1'b0;
                  out <= 4'b0000;
               end
            endcase
         end
   endmodule


```

Pyramid Counter Output:



Conclusion:

The lab has been implemented in Verilog. The clock divider (pulse clock) module and the pyramid counter both modules have been designed using the specifications provided in the lab manual. Both modules have been designed using the 'case' statements instead of 'if-else' statements.

Questions:

- Describe how you would implement a hierarchical design in Verilog.

A Verilog module has a module name and an interface in the form of a port list. One module in Verilog can contain other modules, with module instantiation, which creates a module hierarchy, where the modules are connected with nets and the ports are attached to these nets by either position or name. Since, hierarchy is a programming construct and real hardware does not have the same boundaries, therefore, cross-module/hierarchical references (XMRs) are needed to make connections, which are difficult to describe using the port-binding instantiation method. Hierarchical references can be utilizes for providing parameter values for the modules instantiated deep inside, from the top-level module [2] .
For instance, from [1],

```
top_ver.v

module top_ver (q, p, r, out);

input     q, p, r;
output     out;
reg     out, intsig;

bottom1 u1(.a(q), .b(p), .c(intsig));
bottom2 u2(.l(intsig), .m(r), .n(out));

endmodule
```

bottom1.v

```
module bottom1(a, b, c);

input     a, b;
output     c;
reg      c;

always
begin
     c<=a & b; end endmodule  </pre>
```

bottom2.v

```
module bottom2(l, m, n);

input     l, m;
output    n;
reg       n;

always
begin
     n<=l | m; end endmodule  </pre>
```

Reference:

[1] https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/verilog/ver_hier.html

[2] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-884-complex-digital-systems-spring-2005/lecture-notes/l02_verilog.pdf