CPE 166 Advance Logic Design

Harpreet Sidhu

Lab Section #3

Lab 3 Report

David Quintanilla

Table of Contents

Introduction

Lab 3 introduced a wide variety of topics including the introduction to Hamming Code, Linear Feedback Shift Registers and Algorithmic State Machines (ASM). In Part 1, I learned a basic understanding of Hamming Code. Consisting of bits and Parity bits, they key of Hamming code, is the use of extra parity bits to allow the correction of a single bit error. Furthermore learning that an even parity but is the bit added to the data block that ensures that the total number of buts in the message is even for error detection. Out of all the parts in Lab 3, Hamming Code was the one that stuck with me the most. Part 2 introduced the Linear Feedback Shift Register, which is a shift register whose input but is the output of a linear logic function of two or more of its previous states. The LFSR was something I was very familiar with so Part 2 was fairly simple to follow along.

Moving along to Part 3, Algorithmic State Machines (ASM) were introduced and were to be written in VHDL. ASM's consisted of a State box, Optional decision box and a Optional conditional output box. Part 3 was quite challenging for me due to my lack of familiarity with VHDL, however the concepts of ASM i understood after a couple days practice. ASM were also a key for Part 4. The final part, Part 4 asked to design a Stop watch following a set of guidelines giving in the procedures. Part 4 was by far the hardest part of the lab and required several attempts to finally have a complete running waveform.

Part 1 : Hamming Code Generator

Step 1 Parity Function

```
library ieee;
use ieee.std_logic_1164.all;
package MY_PACK is
function PARITY (D : std_logic_vector) -- declaration of function
return std_logic;
end MY_PACK;
package body MY_PACK is
function PARITY (D : std_logic_vector) -- implementation of function inside the
package body
return std_logic is
variable TMP : std_logic;
begin
 TMP := D(0);
 for J in 1 to D'high loop
 TMP := TMP xor D(J);
 end loop; -- works for any size of D
return TMP;
end PARITY; -- even parity
end MY_PACK;
```

Step 2 Even Parity Bit of 3-bit input Data (Verilog)

```
library ieee;
```

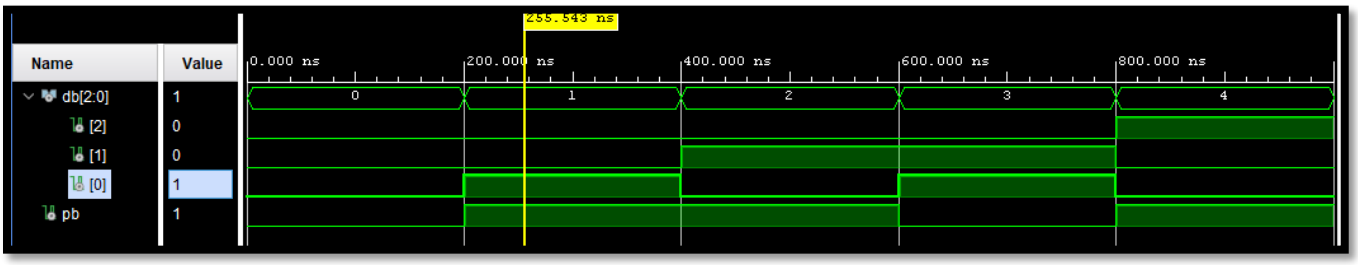
```
use ieee.std_logic_1164.all;
use work.MY_PACK.all;
entity PAR is
port( db: in std_logic_vector(2 downto 0);
 pb: out std_logic);
end PAR;

architecture ARCH of PAR is
   begin
      pb <= PARITY(db);
   end ARCH;
```

Step 2 Even Parity Bit of 3-bit input Data (Test Bench)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity PAR_tb is

end PAR_tb;

architecture Behavioral of PAR_tb is
   component PAR
      port( db: in std_logic_vector (2 downto 0) ;
           pb: out std_logic );
end component;
```

```
signal db: std_logic_vector (2 downto 0) ;
signal pb: std_logic;

begin

 uut: PAR port map(db => db, pb => pb);
 process begin
        db<="000";
        wait for 200 ns;
        db<="001";
        wait for 200 ns;
        db<="010";
        wait for 200 ns;
        db<="011";
        wait for 200 ns;
        db<="100";
        wait for 200 ns;
        db<="101";
        wait for 200 ns;
        db<="110";
        wait for 200 ns;
        db<="111";
        wait for 200 ns;
        wait;
     end process;
end Behavioral;
```

Wave Form (PAR)



Step 3 : (7, 4) Hamming Code Generator (Verilog)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.MY_PACK.all;

entity hamming is

port( d3: in std_logic;
   d5: in std_logic;
   d6: in std_logic;
   d7: in std_logic;
   dout: out std_logic_vector(7 downto 1));

end hamming;
```

```
architecture behavioral of hamming is
  begin

   dout <=  ((D7) & (D6) &(D5) & (PARITY(D7&D6&D5)) & (D3) &
(PARITY(D7&D6&D3)) & (PARITY(D7&D5&D3)));
end behavioral;
```

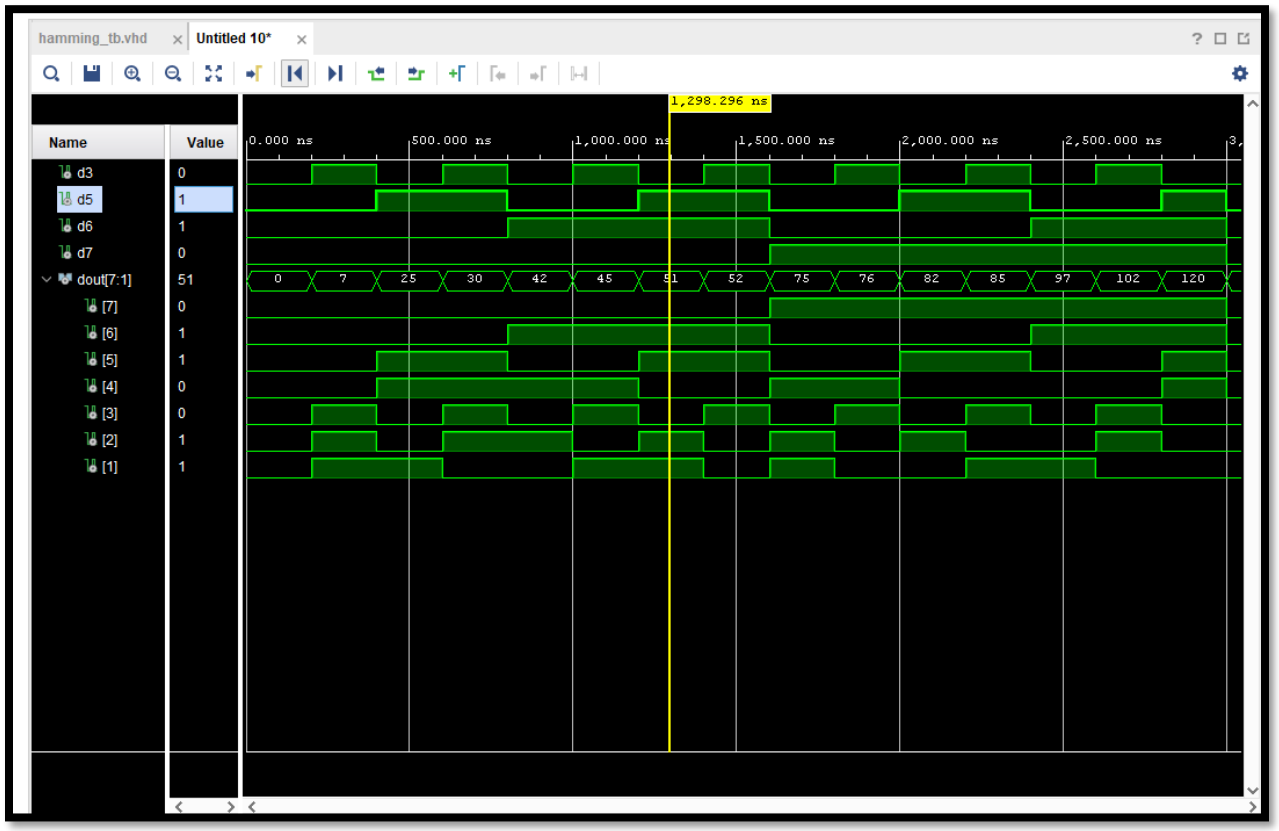
Test Bench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity hamming_tb is


end hamming_tb;



architecture Behavioral of hamming_tb is

component hamming
 port( d3: in std_logic;
   d5: in std_logic;
   d6: in std_logic;
   d7: in std_logic;
   dout: out std_logic_vector(7 downto 1));

end component;
```

```
signal d3:  std_logic;
signal d5:  std_logic;
signal d6:  std_logic;
signal d7:  std_logic;
signal dout:  std_logic_vector(7 downto 1);

begin

    uut: hamming port map( d3 => d3 , d5 => d5 , d6 => d6,  d7 => d7 , dout => dout);

    process begin

    d7<='0';d6<='0'; d5<='0'; d3<='0';
    wait for 200 ns;
    d7<='0';d6<='0'; d5<='0'; d3<='1';
     wait for 200 ns;
    d7<='0';d6<='0'; d5<='1'; d3<='0';
     wait for 200 ns;
    d7<='0';d6<='0'; d5<='1'; d3<='1';
     wait for 200 ns;
    d7<='0';d6<='1'; d5<='0'; d3<='0';
     wait for 200 ns;
    d7<='0';d6<='1'; d5<='0'; d3<='1';
     wait for 200 ns;
    d7<='0';d6<='1'; d5<='1'; d3<='0';
     wait for 200 ns;
    d7<='0';d6<='1'; d5<='1'; d3<='1';
     wait for 200 ns;
    d7<='1';d6<='0'; d5<='0'; d3<='0';
     wait for 200 ns;
```

```
    d7<='1';d6<='0'; d5<='0'; d3<='1';
     wait for 200 ns;
    d7<='1';d6<='0'; d5<='1'; d3<='0';
     wait for 200 ns;
    d7<='1';d6<='0'; d5<='1'; d3<='1';
     wait for 200 ns;
    d7<='1';d6<='1'; d5<='0'; d3<='0';
     wait for 200 ns;
    d7<='1';d6<='1'; d5<='0'; d3<='1';
     wait for 200 ns;
    d7<='1';d6<='1'; d5<='1'; d3<='0';
     wait for 200 ns;
    d7<='1';d6<='1'; d5<='1'; d3<='1';


    end process;


end Behavioral;
```

Wave Form


hamming_tb.vhd
Untitled 10*
1,298.296 ns
Name
Value
0.000 ns
500.000 ns
1,000.000 ns
1,500.000 ns
2,000.000 ns
2,500.000 ns
d3
0
d5
1
d6
1
d7
0
dout[7:1]
51
0
7
25
30
42
45
51
52
75
76
82
85
97
102
120
[7]
0
[6]
1
[5]
1
[4]
0
[3]
0
[2]
1
[1]
1

Part 2 Pseudorandom Number Generator

Step 1, 2, 3 LFSR Design (Verilog)

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity LSFR is
        Port ( reset, clk: in std_logic;
         Q: out std_logic_vector (4 downto 0) );
End LSFR;
Architecture beh of LSFR is
Signal m: std_logic_vector (4 downto 0);
Begin
         Process(reset, clk)
         Begin
         If ( reset = '1') then
         m <= ( 0=> '1', others =>'0'); --- value of “0001”
         elsif (rising_edge(clk) ) then
         m(4 downto 1) <= m(3 downto 0)
         m(0) <= m(2) xor m(3);
        end if;
 end process;
 Q <= m;
```

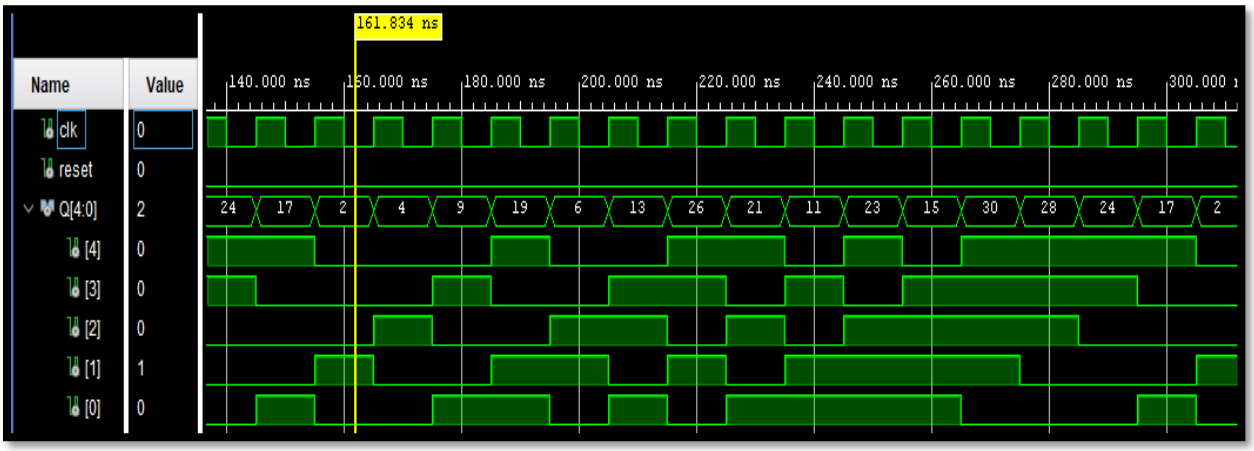
```
 end beh;
```

Step 1, 2, 3 LFSR Design (Test Bench)

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity LSFR_tb is
End LSFR_tb;
Architecture tb of LSFR_tb is
signal clk, reset: std_logic;
signal Q: std_logic_vector (4 downto 0);
component LSFR
 Port ( reset, clk: in std_logic;
 Q: out std_logic_vector (4 downto 0) );
End component;
Begin
        uut: LSFR port map(reset, clk, Q);
        Process Begin
                Clk <= '0';
                Wait for 5 ns;
                Clk <= '1';
                Wait for 5 ns;
 End process;
 Process Begin
        Reset <= '1';
        Wait for 2 ns;
        Reset <= '0';
        Wait for 200ns;
         Wait;
  End process;
```

End tb;

Wave Form



Part 3: Algorithmic State Machine (ASM) Charts

- Steps 1 & 2 are review of ASM charts and code that we learned in class

Step 3. Final Project Work

```
Library ieee;
Use ieee.std_logic_1164.all;

entity ASM is

 port(

   reset, clk, x:   in std_logic;
   y1: out std_logic;
   y0, z: out std_logic_vector (1 downto 0);
   ckcs, ckns: out std_logic_vector (1 downto 0)
   );

end ASM;

architecture Behavioral of ASM is

        constant S0: std_logic_vector(1 downto 0) := "00";
        constant S1: std_logic_vector(1 downto 0) := "01";
        constant S2: std_logic_vector(1 downto 0) := "10";
```

```
        constant S3: std_logic_vector(1 downto 0) := "11";

        signal cs, ns: std_logic_vector(1 downto 0);
begin

  ckns <= ns;
  ckcs <= cs;

  process(reset, clk) begin

    if(reset = '1') then
      cs <= S0;
    elsif(rising_edge(clk)) then
      cs <= ns;
    end if;
  end process;

  process(cs, x) begin
    case(cs) is
      when S0 =>
        if(x = '1') then
          ns <= S1;
        else
          ns <= S0;
        end if;

      when S1 =>
        if(x ='1') then
          ns <= S1;
        else
```

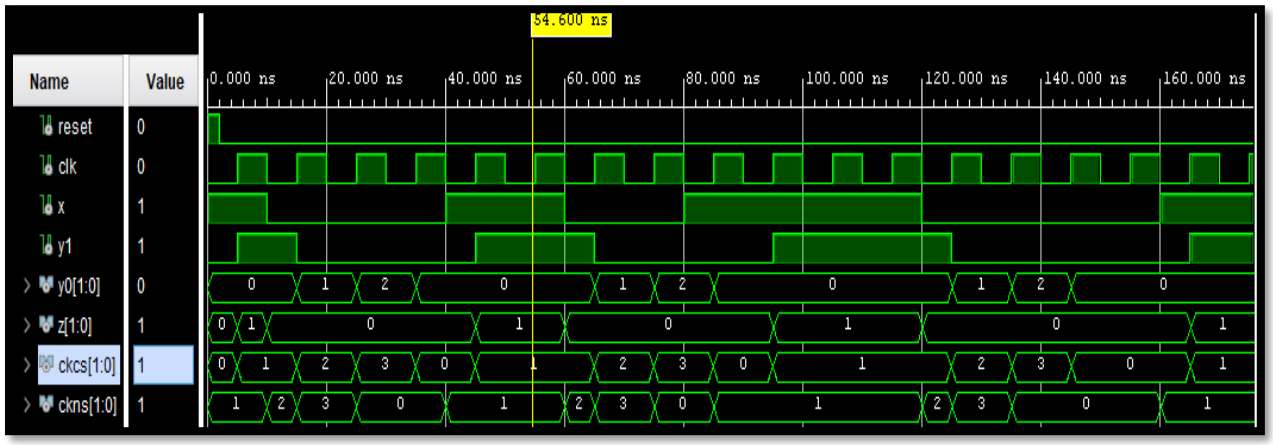
```
            ns <= S2;
          end if;

        when S2 =>
          if(x ='1') then
            ns <= S1;
          else
            ns <= S3;
          end if;

        when S3 =>
          ns <= S0;
        when others =>
          ns <= S0;
      end case;
end process;

   y1 <= '1' when (cs = S1) else '0';

   y0(0) <= '1' when (cs = S2) else '0';

   y0(1) <= '1' when (cs = S3) else '0';

   z(0) <= '1' when (cs = S1) and (x = '1') else '0';

   z(1) <= '1' when (cs = S2) and (x = '1') else '0';

end Behavioral;
```

Step 3. Final Project Work (Test Bench)

```
library IEEE;
use IEEE.std_logic_1164.ALL;


entity ASM_tb is

end ASM_tb;

architecture Behavioral of ASM_tb is
    component ASM
    port(
        reset, clk, x: in std_logic;
        y1: out std_logic;
        y0, z: out std_logic_vector (1 downto 0);
        ckcs, ckns: out std_logic_vector (1 downto 0)
    );
    end component;
    signal reset, clk, x, y1: std_logic;
    signal y0, z: std_logic_vector (1 downto 0);
    signal ckcs, ckns: std_logic_vector (1 downto 0);

begin

    uut: ASM port map(reset => reset, clk => clk, x => x, y1 => y1, y0 => y0, z => z, ckcs =>
ckcs, ckns => ckns);
```

```
    process begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns;
     end process;


      x <= '1', '0' after 10 ns, '1' after 40 ns, '0' after 60 ns, '1' after 80ns, '0' after 120ns,  '1' after
160 ns, '0' after 200 ns, '1' after 300 ns, '0' after 350 ns;


      process begin
        reset <= '1';
        wait for 2 ns;
        reset <= '0';
        wait for 200 ns;
        wait;
      end process;

end Behavioral;
```

Wave Form


54.600 ns
Name
Value
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 ns
reset
0
clk
0
x
1
y1
1
y0[1:0]
0
z[1:0]
1
ckcs[1:0]
1
ckns[1:0]
1

Part 4 : Stopwatch Design

Step 1: clkdiv block (Verilog)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clkdiv is

port(clkin: in std_logic;
    clkout: out std_logic := '0' );
end clkdiv;

architecture Behavioral of clkdiv is
signal cnt : integer range 0 to 10 := 0;

begin
   process(clkin) begin
      if(rising_edge(clkin)) then
         if(cnt = 10) then
            cnt <= 0;
            clkout <= '1';
         elsif (cnt < 3) then
           cnt <= cnt + 1;
           clkout <= '1';
         else
            cnt <= cnt + 1;
            clkout <= '0';
```

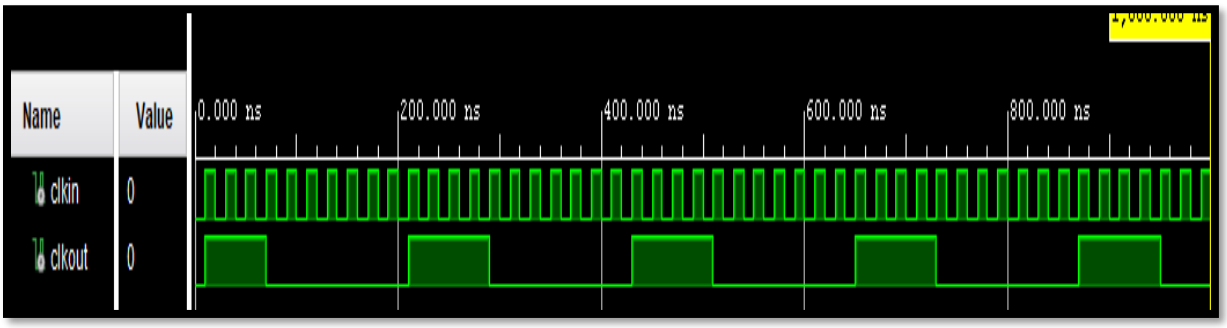
```
            end if;
        end if;
    end process;
end Behavioral;
```

Step 1 clkdiv (Test Bench)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clkdiv_tb is
--  Port ( );
end clkdiv_tb;

architecture Behavioral of clkdiv_tb is
    component clkdiv
        port ( clkin : in std_logic;
               clkout : out std_logic );
    end component;
    signal clkin, clkout : std_logic := '0';
begin
    DUT : clkdiv
    port map(clkin=>clkin,clkout=>clkout);

    process begin
        wait for 10 ns;
        clkin <= not clkin;
    end process;
```

```
end Behavioral;
```

WaveForm



Step 2 : FSM Block

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FSM is
    Port (clk, start, stop, reset: in std_logic;
          en: out std_logic);
end FSM;

architecture Behavioral of FSM is

    constant s0: std_logic_vector(1 downto 0) := "00";
    constant s1: std_logic_vector(1 downto 0) := "01";
```

```
signal cs, ns: std_logic_vector(1 downto 0);

begin
   process(reset, clk)
      begin
         if(reset = '1') then
          cs <= S0;

      elsif (rising_edge(clk)) then

         cs <= ns;

      end if;
   end process;

   process(cs, start, stop)
   begin
      case(cs) is

         when S0 => if (start='1') then

            ns <= S1;

            else

               ns <= S0;

            end if;
```

```
            when S1 => if (stop='1') then
               ns <= S0;
               else
               ns <= S1;
               end if;
            when others => ns <= S0;
         end case;
      end process;
      en <= '1' when (cs = S1 and Start = '1') else '0';
end Behavioral;
```

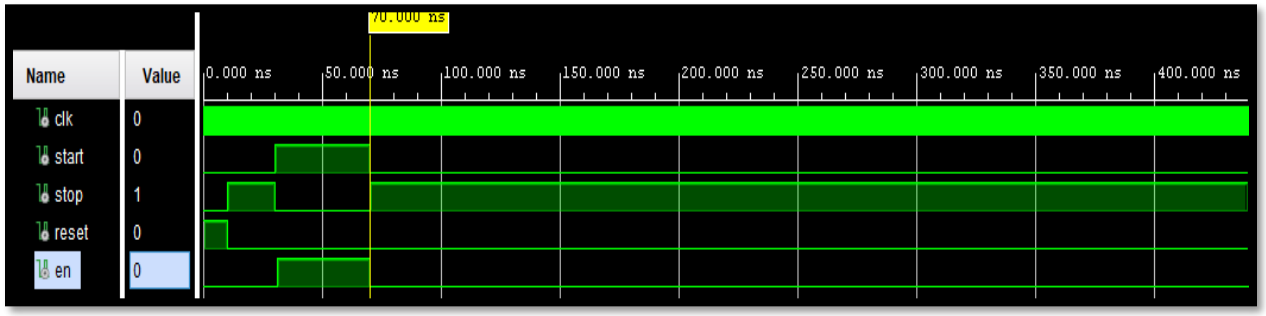
Step 2 FSM Block (Test Bench)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fsm_tb is
end fsm_tb;

architecture Behavioral of fsm_tb is
   component fsm
      port (clk, start, stop, reset: in std_logic;
            en: out std_logic);
   end component;
   signal clk, start, stop, reset, en: std_logic;
   begin
      DUT: fsm port map(clk, start, stop, reset, en);
```

```
process
begin

    clk <= '0';
    wait for 1 ns;
    clk <= '1';
    wait for 1 ns;

end process;

process
begin
    start <= '0';
    stop <= '0';
    reset <= '1';

    wait for 10 ns;
    reset <= '0';
    stop <= '1';

    wait for 20 ns;
    stop <= '0';
    start <= '1';

    wait for 40 ns;
    start <= '0';
    stop <= '1';
```

```
        wait for 100 ns;
        wait;
    end process;
end Behavioral;
```

Step 2 Wave Form



Step 3 : Watch (Verilog)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;


entity watch is
    Port (en, clk, reset: in std_logic;
          y0, y1, y2: out std_logic_vector(3 downto 0));
end watch;
```

```
architecture Behavioral of watch is

    signal y0_reg: std_logic_vector(3 downto 0);
    signal y1_reg: std_logic_vector(3 downto 0);
    signal y2_reg: std_logic_vector(3 downto 0);

    begin

        process(clk, reset)
        begin
            if(reset = '1') then

                y0_reg <= (others=>'0');
                y1_reg <= (others=>'0');
                y2_reg <= (others=>'0');

                elsif rising_edge(clk) then

                    if(y0_reg = 9) then

                        y0_reg <= (others=>'0');

                    elsif(en = '1') then

                        y0_reg <= y0_reg + 1;

                end if;

                if(y0_reg = 9) then
```

```
            if(y1_reg = 9) then
                y1_reg <= (others=>'0');
            else
                y1_reg <= y1_reg + 1;
            end if;
        end if;
        if(y0_reg = 9 and y1_reg = 9) then
            if(y2_reg = 9) then
                y2_reg <= (others=>'0');
            else
                y2_reg <= y2_reg + 1;
            end if;
        end if;
    end if;
end process;
```

```
    y0 <= y0_reg when (en = '1') or (reset = '1');
    y1 <= y1_reg when (en = '1') or (reset = '1');
    y2 <= y2_reg when (en = '1') or (reset = '1');

end Behavioral;
```

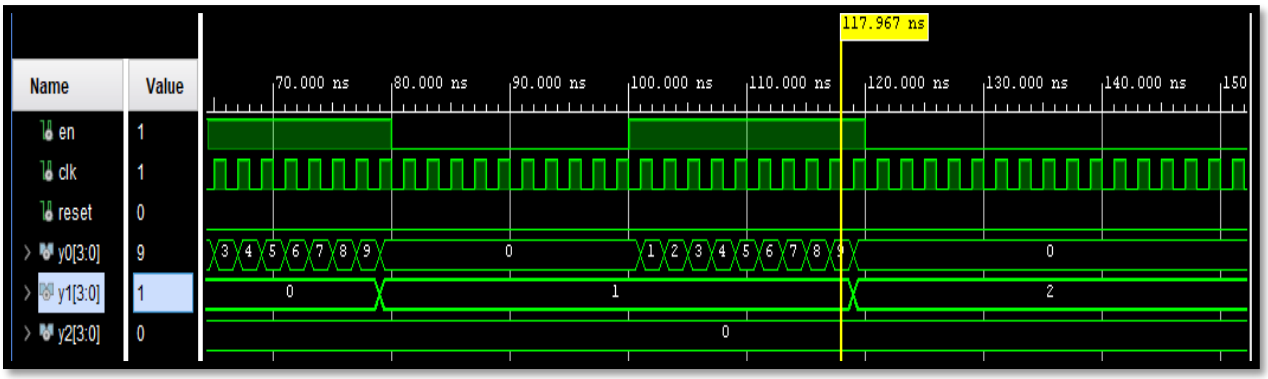
Step 3: Watch (Test Bench)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity watch_tb is
end watch_tb;

architecture Behavioral of watch_tb is
   component watch
      port (en, clk, reset: in std_logic;
           y0, y1, y2: out std_logic_vector(3 downto 0));
   end component;
   signal en, clk, reset: std_logic;
   signal y0, y1, y2: std_logic_vector(3 downto 0);
   begin

      DUT: watch port map(en, clk, reset, y0, y1, y2);

     process

      begin

         clk <= '0';
```

```
    wait for 1 ns;

    clk <= '1';

    wait for 1 ns;
end process;

process
begin

    en <= '0';

    reset <= '1';

    wait for 20 ns;

    en <= '1';

    wait for 20 ns;

    reset <= '1';

    wait for 20 ns;

    reset <= '0';

    wait for 20 ns;

    en <= '0';
```

```
        wait for 20 ns;

        en <= '1';

        wait for 20 ns;

        en <= '0';

        wait;

    end process;
end Behavioral;
```

Step 3 (Wave Form)



Step 4 Final Stopwatch (Verilog)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity stopwatch is
    Port (start, stop, clk, reset: in std_logic;
          y0, y1, y2: out std_logic_vector(3 downto 0));
end stopwatch;

architecture Behavioral of stopwatch is

    signal en, clk2: std_logic;

    component clkdiv

        port (clkin: in std_logic;

              clkout: out std_logic);

    end component;

    component fsm
        port (clk, start, stop, reset: in std_logic;
```

```
        en: out std_logic);

    end component;

    component watch

        port (en, clk, reset: in std_logic;
            y0, y1, y2: out std_logic_vector(3 downto 0));

    end component;

    begin
        g1: clkdiv port map(clkin => clk, clkout => clk2);
        g2: fsm port map(clk => clk2, start => start, stop => stop, reset => reset, en => en);
        g3: watch port map(en => en, clk => clk2, reset => reset, y0 => y0, y1 => y1, y2 => y2);

end Behavioral;
```

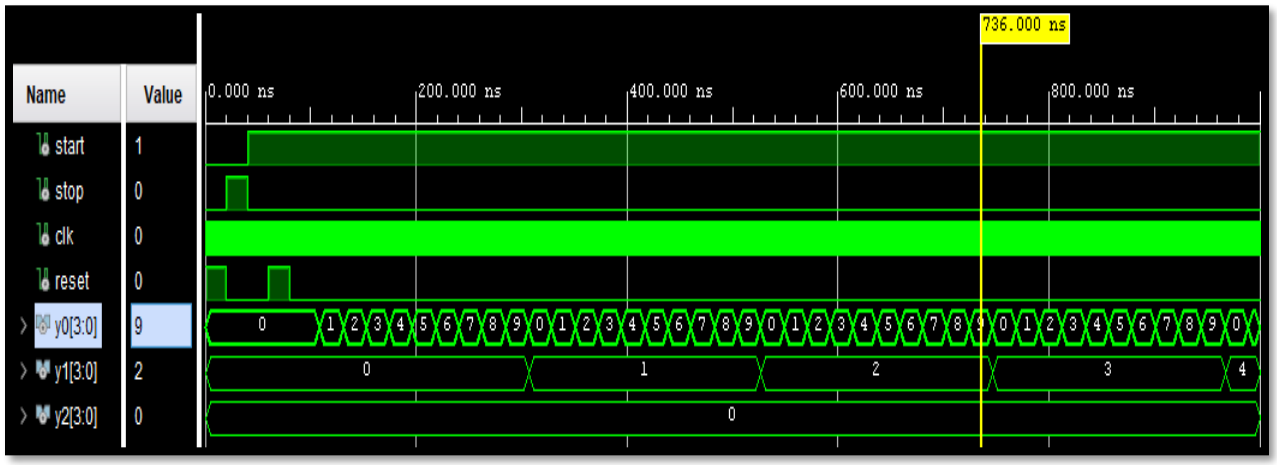
Step 4 Final Stopwatch (Test Bench)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity stopwatch_tb is
end stopwatch_tb;

architecture Behavioral of stopwatch_tb is
    component stopwatch
        port (start, stop, clk, reset: in std_logic;
            y0, y1, y2: out std_logic_vector(3 downto 0));
```

```
end component;

signal start, stop, clk, reset: std_logic;
signal y0, y1, y2: std_logic_vector(3 downto 0);
begin

   DUT: stopwatch port map(start, stop, clk, reset, y0, y1, y2);

   process

   begin
      clk <= '0';
      wait for 1 ns;
      clk <= '1';
      wait for 1 ns;

   end process;

   process
   begin

      start <= '0';
      stop <= '0';
      reset <= '1';

      wait for 20 ns;
      reset <= '0';
      stop <= '1';

      wait for 20 ns;
```

```
        stop <= '0';
        start <= '1';

        wait for 20 ns;

        reset <= '1';

        wait for 20 ns;
        reset <= '0';

        wait;
    end process;

end Behavioral
```

Step 4 Final Stop watch (Wave Form)



Conclusion

Lab 3 had its on set of challenges that were both familiar and new. One of the biggest lessons learned in this lab was understanding VHDL in a deeper context and the concept of Hamming Code. Some parts of this lab overlap with previous labs such as the clk divider and FSM. A key concept of lab 3 that I took away was “port mapping”. Although this is not a new concept to me, I really understood the concept of porting in lab 3.