CPE166-04: Advanced Logic Design

Lab Number: 3

Brandon Sayareh

Table of Contents

Introduction

This is a four-part laboratory. The first part is focused on building a (7, 4) Hamming Code Generator that took 4 different parts and used two test benches. Next was a Pseudorandom Number Generator which was just one module and one test bench. In part 3 an Algorithmic State Machine (ASM) Chart is designed which uses new parts that focus a lot on the clock with the reuse of one multiplexer. The last design is the stopwatch in VHDL by using the hierarchical design approach shown in the figure with the engineering and design purpose in part 4. All code was run in the Vivado IDE.

Laboratory Report

Part 1: (7, 4) Hamming Code Generator

Design purpose

The key of the Hamming Code is the use of extra parity bits to allow the correction of a single bit error. Hamming (7,4) is a linear error-correcting code that encodes four bits of data into seven bits by adding three even parity bits shown in figure 1. This project is to design (7, 4) Hamming Code generator in VHDL.

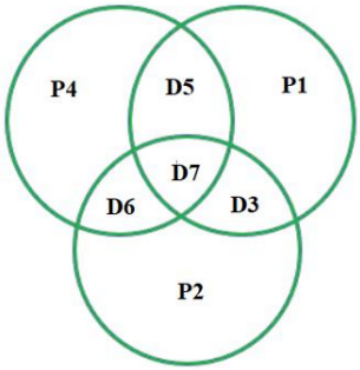


Figure 1. (7, 4) hamming code diagrams

The interface of this design is shown in Table 1.

Table 1. (7, 4) hamming code interface signals

Port Names	Port Direction	Port Size
D7	Input	1 bit
D6	Input	1 bit
D5	Input	1 bit
D3	Input	1 bit
DOUT	Output	7 bits (7 downto 1)

In the final design, D7, D6, D5, D3 are input data, DOUT is the final 7-bit hamming code.

VHDL code for MY_PACK

```
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;
```

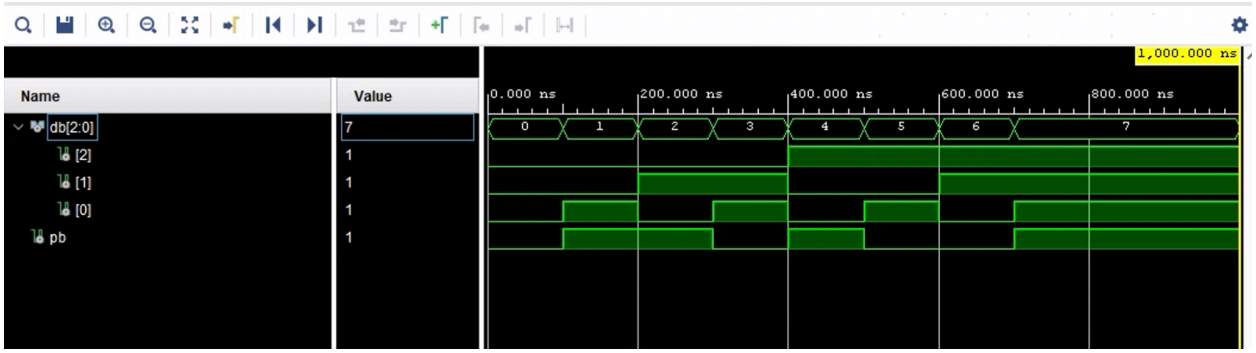
VHDL code PAR

```
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;
```

VHDL Test Bench for PAR

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

entity PAR_tb is
--  Port ( );
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 100 ns;
 db <= "001";
 wait for 100 ns;
 db <= "010";
 wait for 100 ns;
 db <= "011";
 wait for 100 ns;
 db <= "100";
 wait for 100 ns;
 db <= "101";
 wait for 100 ns;
 db <= "110";
 wait for 100 ns;
 db <= "111";
 wait for 100 ns;
 wait;
 end process;
end Behavioral;
```

Results of the Simulation



We have data inputs from 0-7. When we have an odd value we need a pb or parity bit to make it an even number of data.

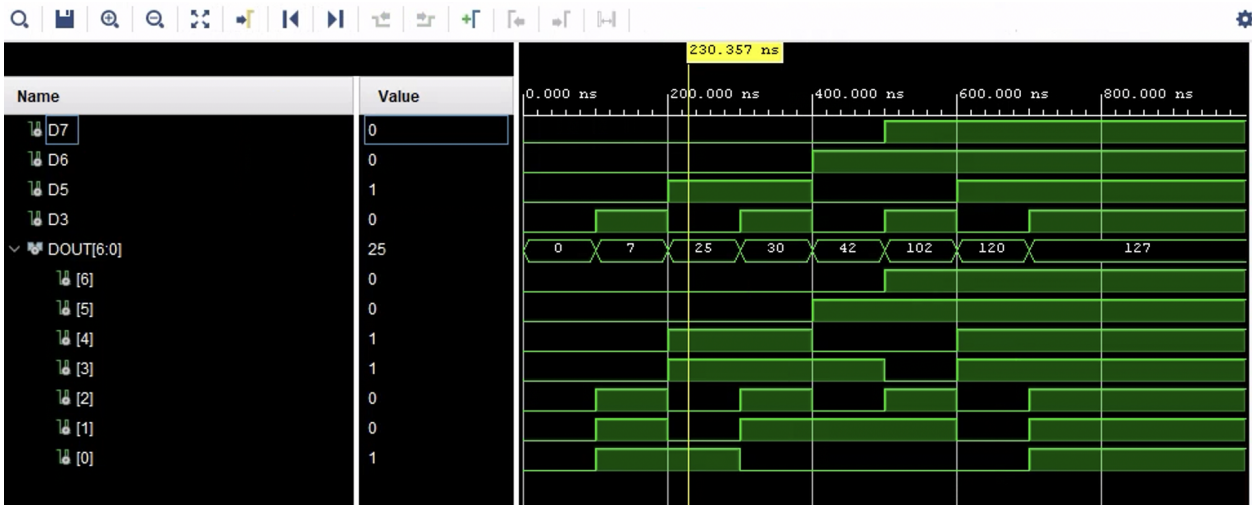
VHDL code for hamming

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

entity hamming is
    --  Port ( );
    -- inputs and outputs
    Port(D7, D6,D5,D3: in std_logic;
        DOUT: out std_logic_vector(6 downto 0)
    );
end hamming;

architecture Behavioral of hamming is
    --making the PAR commponent
    --changing from entity to component. in notes 10.example 11
    component PAR
    port(db: in std_logic_vector(2 downto 0);
    pb: out std_logic);
    end component;
begin
    DOUT(6)<=D7; DOUT(5)<=D6; DOUT(4)<=D5; DOUT(2)<=D3;
    PAR4: PAR port map(db(2)=>D7,db(1)=>D6,db(0)=>D5,pb=>DOUT(3));
    PAR2: PAR port map(db(2)=>D7,db(1)=>D6,db(0)=>D3,pb=>DOUT(1));
    PAR1: PAR port map(db(2)=>D7,db(1)=>D5,db(0)=>D3,pb=>DOUT(0));

end Behavioral;
```

VHDL testbench for hamming

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


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

architecture Behavioral of hamming_tb is
 component hamming
 Port (D7, D6, D5, D3: in std_logic;
 DOUT: out std_logic_vector(6 downto 0));
 end component;
 signal D7, D6, D5, D3 : std_logic;
 signal DOUT : std_logic_vector(6 downto 0);
begin
 uut: hamming port map(D7=>D7,D6=>D6,D5=>D5,D3=>D3,DOUT=>DOUT);
 process begin
 D7<='0'; D6<='0'; D5<='0'; D3<='0'; -- 0000
 wait for 100 ns;
 D7<='0'; D6<='0'; D5<='0'; D3<='1'; -- 0001
 wait for 100 ns;
 D7<='0'; D6<='0'; D5<='1'; D3<='0'; -- 0010
 wait for 100 ns;
 D7<='0'; D6<='0'; D5<='1'; D3<='1'; -- 0011
 wait for 100 ns;
 D7<='0'; D6<='1'; D5<='0'; D3<='0'; -- 0100
 wait for 100 ns;
 D7<='1'; D6<='1'; D5<='0'; D3<='1'; -- 1101
 wait for 100 ns;
 D7<='1'; D6<='1'; D5<='1'; D3<='0'; -- 1110
 wait for 100 ns;
 D7<='1'; D6<='1'; D5<='1'; D3<='1'; -- 1111
 wait for 100 ns;
 wait;
 end process;
end Behavioral;
```

Results of the simulation



This simulation shows that we need a parity bit for odd douts.

Part 2: Pseudorandom Number Generator

Design purpose

A linear feedback shift register (LFSR) is a shift register whose input bit is the output of a linear logic function of two or more of its previous states. The LFSR circuit can be used as a pseudorandom number generator. This project is a design of a 5-stage LFSR circuit.

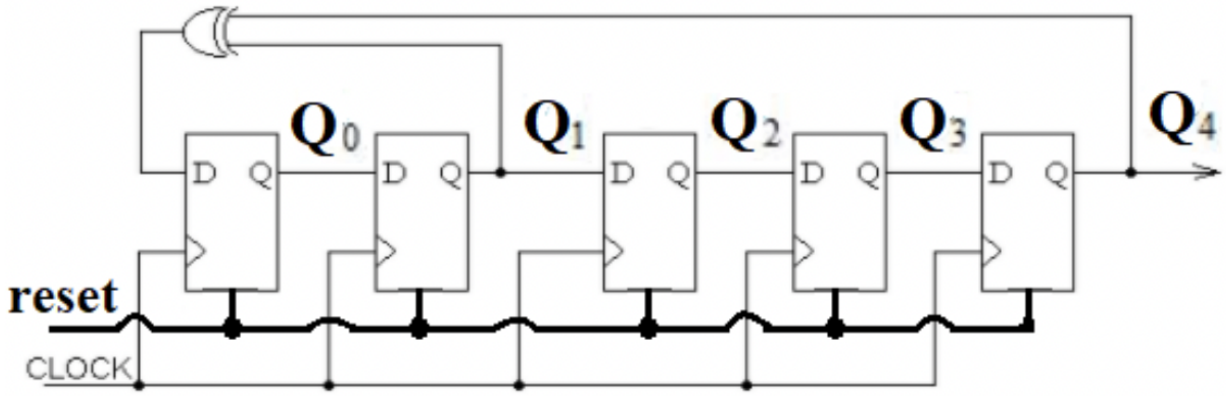


Figure 1. 5-stage LFSR diagram

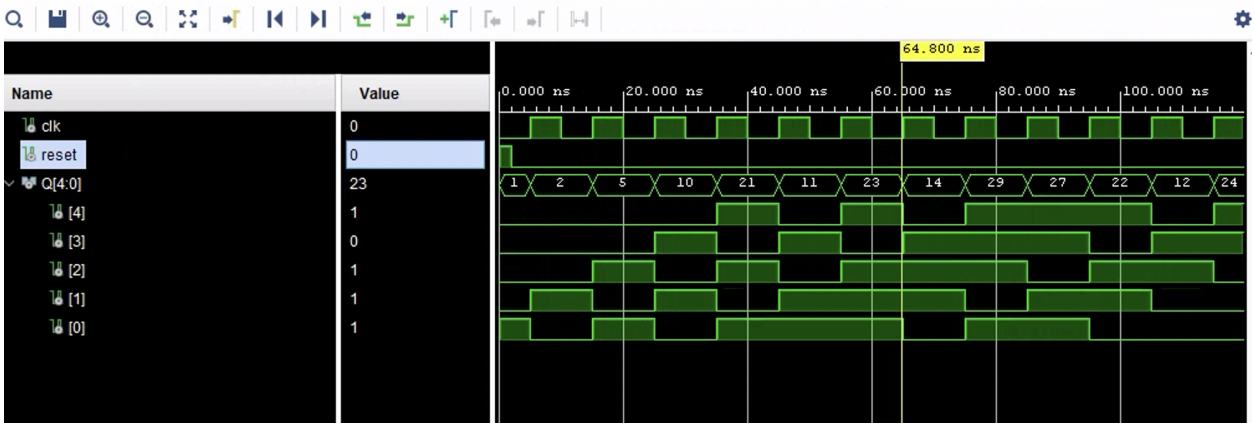
VHDL Code for pseudo_num_gen

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity pseudo_num_gen is
--  Port ( );
Port (reset, clk: in std_logic;
 Q: out std_logic_vector(4 downto 0));
end pseudo_num_gen;

architecture Behavioral of pseudo_num_gen 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(1) xor m(4);
 end if;
 end process;
 Q <= m;
end Behavioral;
```

VHDL Test Bench for pseudo_num_gen

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

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

architecture Behavioral of pseudo_num_gen_tb is
 signal clk, reset: std_logic;
 signal Q: std_logic_vector (4 downto 0);
 component pseudo_num_gen
 Port ( reset, clk: in std_logic;
 Q: out std_logic_vector (4 downto 0) );
 end component;
 begin
 uut: pseudo_num_gen 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 200 ns;
 wait;
 end process;
end Behavioral;
```

Results of the simulation

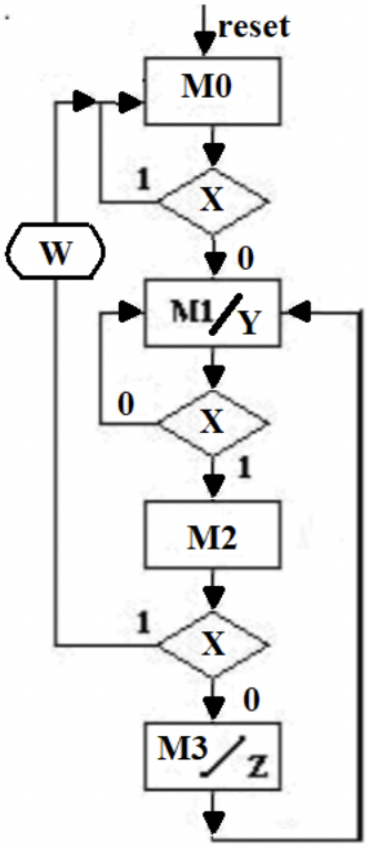


The value of Q (4 downto 0) repeats every 31 clock cycles.

Part 3: Algorithmic State Machine (ASM) Charts

Design Purpose

Implement the ASM charts and run simulations using VHDL. The ASM chart below has two moore outputs Y and Z. Y is asserted in the M1 state and Z is asserted in the M3 state. It has a mealy output, which is asserted when X is logic high in the M2 state.



Verilog Code for ASM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity ASM is
--  Port ( );
port (reset, clk, x: in std_logic;
        y, 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
   ckcs<=cs;
   ckns<=ns;

   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;
```

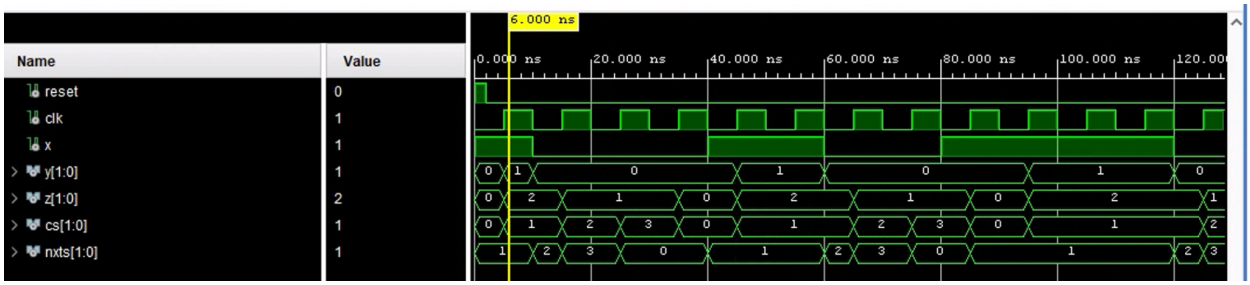
```
        when S2 => if(x='1') then
            ns<=S1;
        else
            ns<=S3;
        end if;
        when S3 => ns<=S0;
        when others => ns<=S0;
        end case;
    end process;

    y(0) <= '1' when ((cs=S2) or (cs=S3)) else '0';
    y(1) <= '1' when (cs=S1) 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;
```

Verilog Test Bench for ASM

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

entity ASM_tb is
end ASM_tb;

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

    signal reset, clk, x: std_logic;
    signal y, z: std_logic_vector (1 downto 0);
    signal cs, nxts: std_logic_vector (1 downto 0);

begin
    DUT: ASM port map(reset, clk, x, z(1 downto 0), y(1 downto 0), cs, nxts);

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

```
end process;
 process
    begin
      x <= '1';
      wait for 10 ns;
      x <= '0';
      wait for 30 ns;
      x <= '1';
      wait for 20 ns;
      x <= '0';
      wait for 20 ns;
      x <= '1';
      wait for 40 ns;
      x <= '0';
      wait for 40 ns;
      x <= '1';
      wait for 40 ns;
      x <= '0';
      wait for 10 ns;
      wait;
 end process;

process
 begin
    Reset <= '1';
     Wait for 2 ns;
    Reset <= '0';
    Wait for 400 ns;
    Wait;
 End process;
end tb;
```

Results of the simulation



The reset state is only high at start. After the clock cycle runs and the output pattern repeats with x, y, z, cs, and nxts.

Part 4 - Stopwatch Design

Design Purpose

The frequency of the input "clk" signal is 4 Hz. At any time, if the "reset" input is logic high, the output of the stopwatch will be zero. When the "start" input is logic high, the stopwatch will start counting. When the "stop" input is logic high, the stopwatch will stop, but the stopwatch output will maintain its value. After that, when the "start" input is logic high again, the stopwatch will resume counting from its old value.

The clkdiv block gets the 4 Hz input clkin signal and generates the 1 Hz output clkout signal.

For the fsm block (figure 2), when the reset signal is logic high, the state machine enters the first idle state. Idle state means that the current watch is not counting, and the displayed output value is 0. Output "en" as 0 to disable counting.

The watch block has three inputs: reset, en and 1 Hz clk. It has three 4-bit outputs: y2, y1, and y0. When "reset" is logic high, the watch will output 0. Only when "en" is logic high, the watch will increase its value every 1 second. Each of the y2, y1 and y0 signals ranges from $(0000)_2$ to $(1001)_2$.

The final stopwatch design is shown in Figure 1, with four 1-bit inputs, including reset, clk, start and stop, and three 4-bit outputs, including y2, y1, and y0. In the architecture part of the design, two internal signals must be declared, including "en" and "clk2", and three components must be declared, including clkdiv, fsm and watch. In addition, the main design part requires a clkdiv instance, an fsm instance and a watch instance. The two internal signals "en" and "clk2" are used for wiring of different modules.

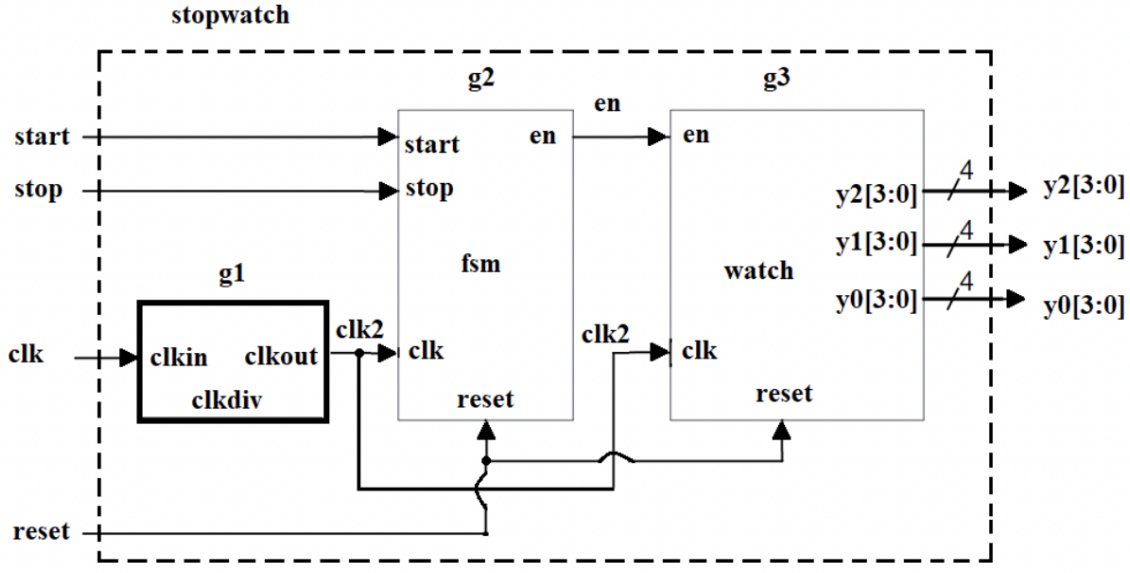


Figure 1. Stopwatch block diagram

Table 1. I/O ports for the stopwatch design

Port Names	Port Direction	Port Size
start	Input	1
stop	Input	1
clk	Input	1
reset	Input	1
y3	Output	4
y2	Output	4
y1	Output	4
y0	Output	4

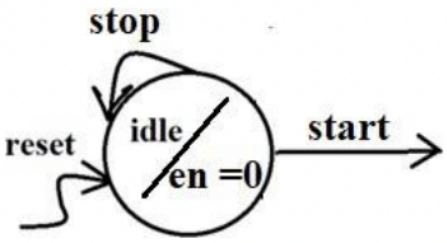


Figure 2. IDLE state of the fsm block

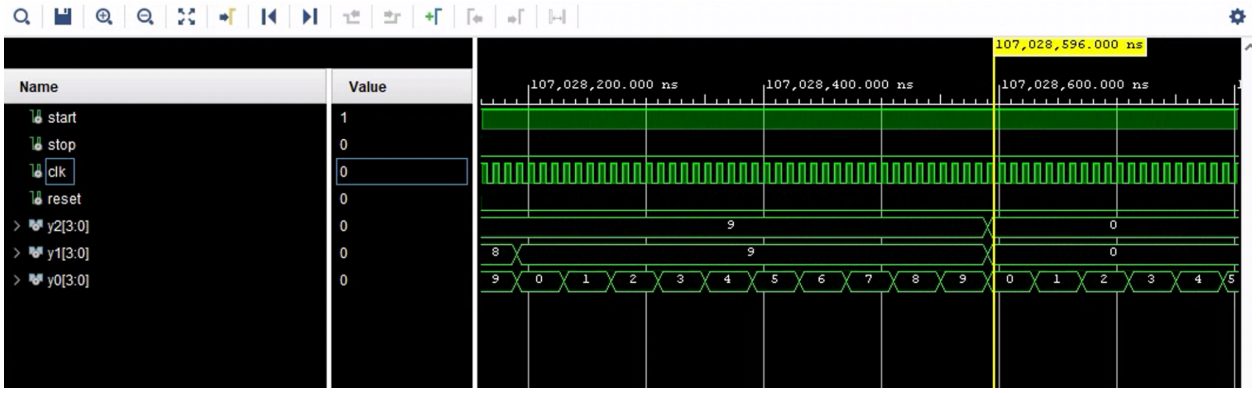
VHDL Code for the stopwatch

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

entity stopwatch is
--  Port ( );
   Port ( start : in STD_LOGIC;
   stop : in STD_LOGIC;
   clk : in STD_LOGIC;
   reset : in STD_LOGIC;
   y3,y2,y1,y0 : out STD_LOGIC_VECTOR (3 downto 0));
end stopwatch;

architecture Behavioral of stopwatch is
Component clkdiv is
   Port ( clkin : in STD_LOGIC;
   clkout : out STD_LOGIC);
end Component;

Component fsm is
Port ( start,stop,clk,reset : in STD_LOGIC;
en : out STD_LOGIC);
end Component;

Component watch is
   Port ( clk,en,reset : in STD_LOGIC;
   y2,y1,y0 : out STD_LOGIC_VECTOR (3 downto 0));
end Component;

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

end Behavioral;
```

VHDL Test bench Code for stopwatch

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity tb_stopwatch is
end tb_stopwatch;

architecture Behavioral of tb_stopwatch is
Component stopwatch is
Port ( start : in STD_LOGIC;
stop : in STD_LOGIC;
clk : in STD_LOGIC;
reset : in STD_LOGIC;
y2,y1,y0 : out STD_LOGIC_VECTOR (3 downto 0));
end Component;
signal start,stop,clk,reset:STD_LOGIC;
signal y2,y1,y0 :STD_LOGIC_VECTOR (3 downto 0);

begin
UUT:stopwatch port map
(start=>start,stop=>stop,clk=>clk,reset=>reset,y2=>y2,y1=>y1,y0=>y0);
process
begin
clk<='1';
wait for 5ns;
clk<='0';
wait for 5ns;
end process;

process
begin
reset<='1';
wait for 10 ns;
reset<='0';
start<='1';stop<='0';
wait for 500 ns;
reset<='0';
start<='0';stop<='1';
wait for 100 ns;
reset<='0';
start<='1';stop<='0';
wait for 100000 ns;
end process;

end Behavioral;
```

Results of the simulation



The output y2 y1 y0 is a 3-digit binary number, and each digit ranges from 0 to 9. Assuming y2 = (0110) 2 = 6, y1 = (0101) 2 = 5, y0 = (0111) 2 = 7, this means 657 seconds. If you press the "stop" button, the stopwatch will stop at 657 seconds and keep the value unchanged. After pressing the "start" button, the stopwatch will continue counting every second until: y2 = (1001) 2 = 9, y1 = (1001) 2 = 9, y0 = (1001) 2 = 9, which means 999 seconds. After 999 seconds, the stopwatch will return to 0 and then increase its value every second.

VHDL code for clkdiv

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity clkdiv is
--  Port ( );
    Port ( clkin : in STD_LOGIC;
    clkout : out STD_LOGIC);
end clkdiv;
architecture Behavioral of clkdiv is
    signal count:integer:=1;
begin

process(clkin)
    begin
    if (clkin'event and clkin='1')then
        count<=count+1;
        case count is
        when 1=> clkout<='1';
        when 2=> clkout<='0';
        when 4=> clkout<='1';count<=1;
        when others=>null;
        end case;
    end if;
end process;
end Behavioral;
```

VHDL code for fsm

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity fsm is
--  Port ( );
   Port ( start,stop,clk,reset : in STD_LOGIC;
   en : out STD_LOGIC);
end fsm;

architecture Behavioral of fsm is

begin
   process(clk,reset)
begin
   if(reset='1')then
      en<='0';
   elsif(clk'event and clk='1')then
      if(start='1' and stop='0')then
         en<='1';
      elsif(start='1' and stop='1')then
         en<='0';
   else
      en<='0';
   end if;
   end if;
end process;

end Behavioral;
```

VHDL code for watch

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.all;
use IEEE.std_logic_unsigned.all;

entity watch is
--  Port ( );
   Port ( clk,en,reset : in STD_LOGIC;
   y2,y1,y0 : out STD_LOGIC_VECTOR (3 downto 0));
end watch;

architecture Behavioral of watch is
```

```
   signal count_ssd,count_ssd1,count_ssd2,count_ssd3:STD_LOGIC_VECTOR (3
downto 0);
begin
process(clk,reset)
  begin
  if(reset='1')then
      count_ssd3<="0000";
      count_ssd2<="0000";
      count_ssd1<="0000";
   elsif(clk'event and clk='1')then
      if (en='1') then
        if count_ssd1="1001" then
        count_ssd2<=count_ssd2+1;
        count_ssd1<="0000";
        if count_ssd2="1001" then
            count_ssd3<=count_ssd3 +1;
            count_ssd2<="0000";
         if count_ssd3="1001" then
            count_ssd3<="0000";
         end if;
         end if;
               else
               count_ssd1<=count_ssd1+1;
               end if;
        else
           count_ssd3<=count_ssd3;
           count_ssd2<=count_ssd2;
           count_ssd1<=count_ssd1;
        end if;
      end if;
end process;
y2<=count_ssd3;
y1<=count_ssd2;
y0<=count_ssd1;

end Behavioral;
```

Conclusion

From this project, I refreshed on the little Verilog I knew and was able to practice the material taught in the lecture. Coding for some of the modules was easy while others were much harder. Overall code for the modules did not give me too much trouble for parts 1, 2, and 3. I was able to reference the notes and the logic made sense to me. I did have trouble with writing some of the test benches but I believe this was due to unfamiliarity. Part four with the state machine gave me trouble but it was because I was not coding for other states and only looking at the main instructions. Once clear on what to do I was able to code this and the testbench without a problem.

Demoing my code showed me a lot of useful information and tips about the IDE Vivado and Verilog itself. Some useful commands helped me code certain things faster like duplicating lines. I was also shown how to see the current state and next state displayed on the waveform for the state machine. The information given to me was by my professor about how I could code more efficiently, presenting code or waveforms in the field, and helping with my part 4 code. I will be re-coding the modules in this lab in order to solidify the knowledge I have gained. I feel comfortable with the modules, material covered and am pleased with what I have learned.