Custom 4-bit Signed Fast Multiplier Report

Tony Han

4/23/2021

I. Introduction

In this project, we are intended to construct a 4-bit multiplier that takes 2 4-bit binary and returns an 8-bit product. A system controller is needed to control shifters, accumulators, adders, and counter to cooperate. All the modules will be implemented in VHDL and simulate on the Xilinx ISim toolkit.

II. Design

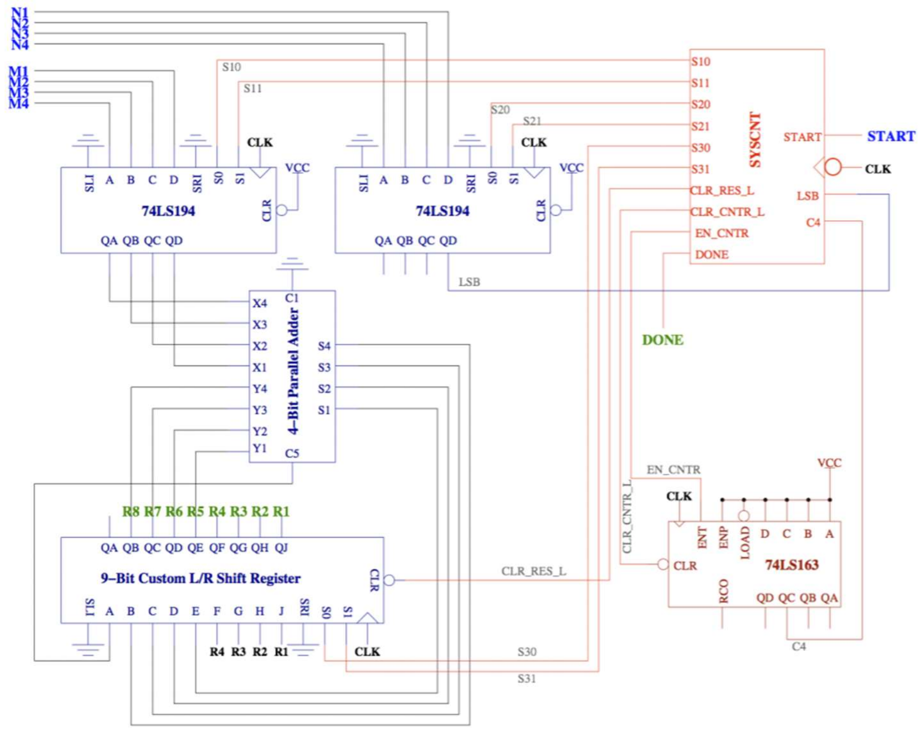


Figure 1. Multiplication Circuit

Since our task is to calculate the product of two 4-bit binary numbers, we need some components to store the data flow of two binaries. With the demand of dealing with shifting data flow, a 4-bit parallel load bidirectional shift register is a good choice. Also, we need a storage component to store the result, up to an 8-bit binary. Noted that the carry of the product also matters as we are shifting the resulting over time, we actually need a 9-bit parallel load bidirectional shift register. A 4-bit adder is also needed to calculate the internal addition for multiplier and higher 4-bit of the temporary result. So, a 4-bit parallel adder is needed. Since the system shifts specific times to get the actual result, we need a counter to store how many shifts the system performed. Finally, a system controller is needed to accept all external input such as START, CLK, and generate the DONE signal. More importantly, it is responsible to generate internal control signals for all the components. From the state diagram, we can see that there are 5 states in total. And all outputs are aligned with the state, which means the system controller can be implemented by a Moore machine. Finally, a top-level multiplier component is needed to instantiate all the previous components mentioned and map all the internal signals.

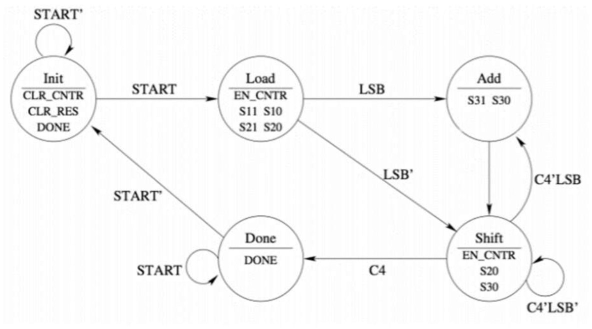


Figure 2. System Controller State Diagram

III. Result

System Controller

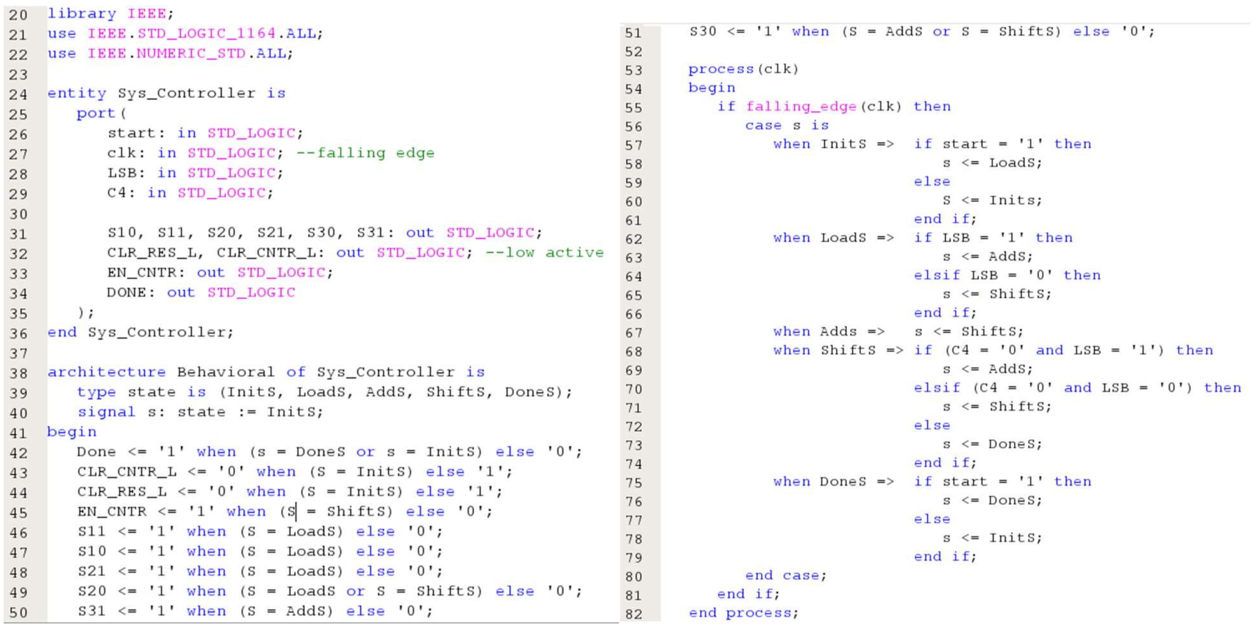
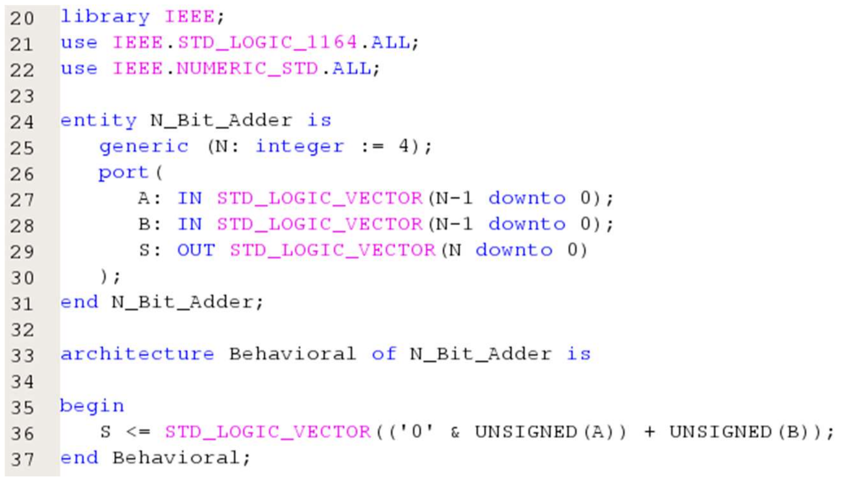

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Sys_Controller is
   port(
      start: in STD_LOGIC;
      clk: in STD_LOGIC; --falling edge
      LSB: in STD_LOGIC;
      C4: in STD_LOGIC;

      S10, S11, S20, S21, S30, S31: out STD_LOGIC;
      CLR_RES_L, CLR_CNTR_L: out STD_LOGIC; --low active
      EN_CNTR: out STD_LOGIC;
      DONE: out STD_LOGIC
   );
end Sys_Controller;

architecture Behavioral of Sys_Controller is
   type state is (InitS, LoadS, AddS, ShiftS, DoneS);
   signal s: state := InitS;
begin
   Done <= '1' when (s = DoneS or s = InitS) else '0';
   CLR_CNTR_L <= '0' when (S = InitS) else '1';
   CLR_RES_L <= '0' when (S = InitS) else '1';
   EN_CNTR <= '1' when (S = ShiftS) else '0';
   S11 <= '1' when (S = LoadS) else '0';
   S10 <= '1' when (S = LoadS) else '0';
   S21 <= '1' when (S = LoadS) else '0';
   S20 <= '1' when (S = LoadS or S = ShiftS) else '0';
   S31 <= '1' when (S = AddS) else '0';
   S30 <= '1' when (S = AddS or S = ShiftS) else '0';

   process(clk)
   begin
      if falling_edge(clk) then
         case s is
            when InitS =>  if start = '1' then
                              s <= LoadS;
                           else
                              S <= InitS;
                           end if;
            when LoadS =>  if LSB = '1' then
                              s <= AddS;
                           elsif LSB = '0' then
                              s <= ShiftS;
                           end if;
            when AddS =>   s <= ShiftS;
            when ShiftS => if (C4 = '0' and LSB = '1') then
                              s <= AddS;
                           elsif (C4 = '0' and LSB = '0') then
                              s <= ShiftS;
                           else
                              s <= DoneS;
                           end if;
            when DoneS =>  if start = '1' then
                              s <= DoneS;
                           else
                              s <= InitS;
                           end if;
         end case;
      end if;
   end process;
```

Figure 3. System Controller VHDL Implementation

Figure 3 shows the VHDL implementation of the system controller. It first generates outputs based on the current state. Then it updates the next state based on the input. Notice that the system controller is falling-edge triggered as opposed to all other components, which are raising-edge triggered. And this falling-edge triggered design is intentional to better sync the inputs and outputs from other components.

4-Bit Parallel Adder

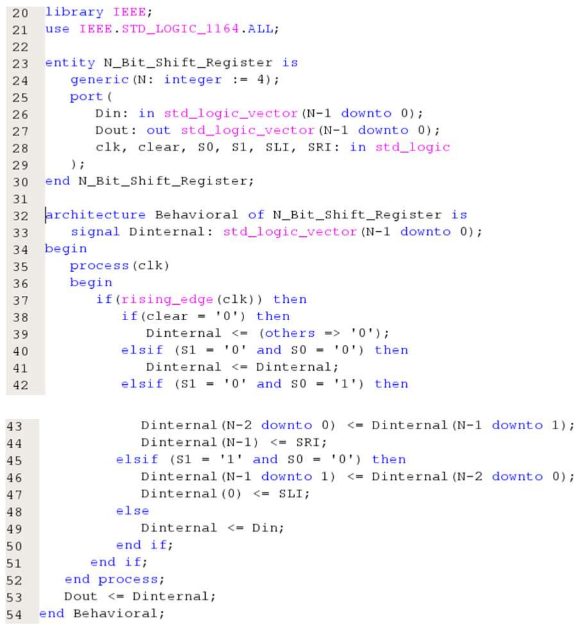


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity N_Bit_Adder is
   generic (N: integer := 4);
   port(
      A: IN STD_LOGIC_VECTOR(N-1 downto 0);
      B: IN STD_LOGIC_VECTOR(N-1 downto 0);
      S: OUT STD_LOGIC_VECTOR(N downto 0)
   );
end N_Bit_Adder;

architecture Behavioral of N_Bit_Adder is

begin
   S <= STD_LOGIC_VECTOR(('0' & UNSIGNED(A)) + UNSIGNED(B));
end Behavioral;
```

Figure 4. 4-bit adder VHDL implementation

As shown in Figure 4, the adder simply takes two 4-bit binary numbers and calculates their sum. Notice that for convenience, the carry is integrated into the sum result. As a result, the sum output is a 5-bit binary number. And the most significant bit is the carry.

N-bit Parallel Load Bidirectional Shift Register

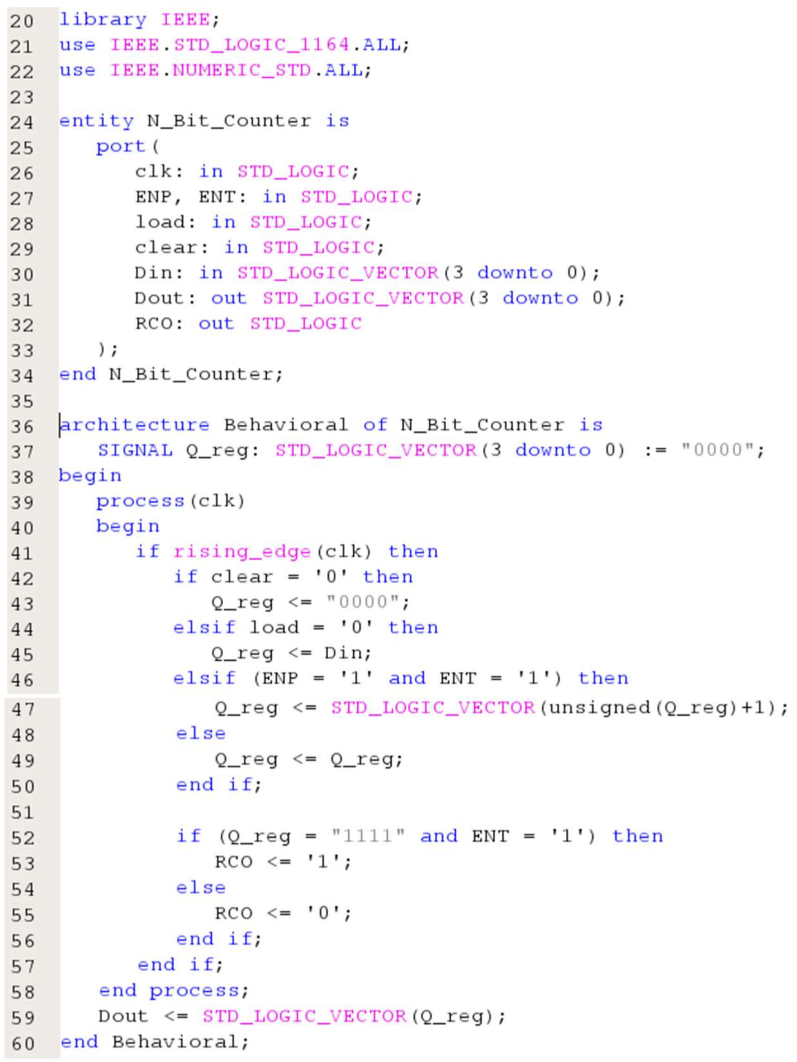


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity N_Bit_Shift_Register is
   generic(N: integer := 4);
   port(
      Din: in std_logic_vector(N-1 downto 0);
      Dout: out std_logic_vector(N-1 downto 0);
      clk, clear, S0, S1, SLI, SRI: in std_logic
   );
end N_Bit_Shift_Register;

architecture Behavioral of N_Bit_Shift_Register is
   signal Dinternal: std_logic_vector(N-1 downto 0);
begin
   process(clk)
   begin
      if(rising_edge(clk)) then
         if(clear = '0') then
            Dinternal <= (others => '0');
         elsif (S1 = '0' and S0 = '0') then
            Dinternal <= Dinternal;
         elsif (S1 = '0' and S0 = '1') then
            Dinternal(N-2 downto 0) <= Dinternal(N-1 downto 1);
            Dinternal(N-1) <= SRI;
         elsif (S1 = '1' and S0 = '0') then
            Dinternal(N-1 downto 1) <= Dinternal(N-2 downto 0);
            Dinternal(0) <= SLI;
         else
            Dinternal <= Din;
         end if;
      end if;
   end process;
   Dout <= Dinternal;
end Behavioral;
```

Figure 5. N-bit Parallel Load Bidirectional Shift Register VHDL implementation

As shown in Figure 5, a generic design is used to fit the demand of both 4-bit and 9-bit. Implementation is nothing different from a 74LS194 but takes n-bit input. Also, the control signals and logics are the same.

4-bit Parallel Load Counter

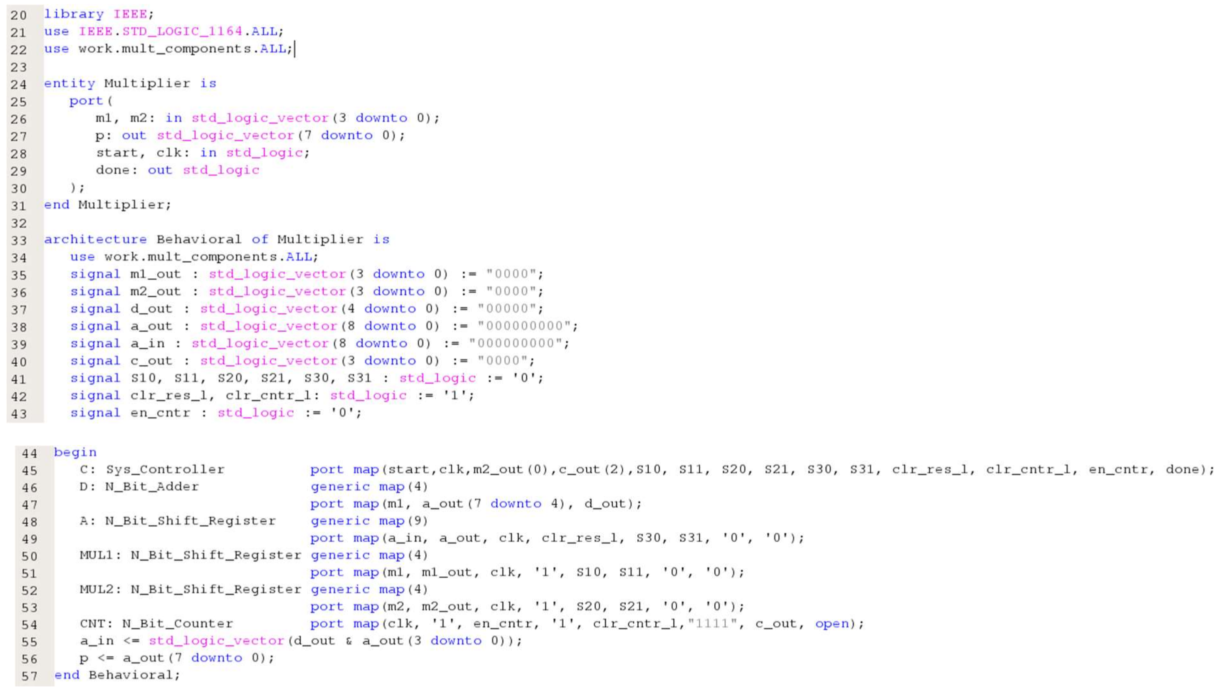


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity N_Bit_Counter is
   port(
      clk: in STD_LOGIC;
      ENP, ENT: in STD_LOGIC;
      load: in STD_LOGIC;
      clear: in STD_LOGIC;
      Din: in STD_LOGIC_VECTOR(3 downto 0);
      Dout: out STD_LOGIC_VECTOR(3 downto 0);
      RCO: out STD_LOGIC
   );
end N_Bit_Counter;

architecture Behavioral of N_Bit_Counter is
   SIGNAL Q_reg: STD_LOGIC_VECTOR(3 downto 0) := "0000";
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if clear = '0' then
            Q_reg <= "0000";
         elsif load = '0' then
            Q_reg <= Din;
         elsif (ENP = '1' and ENT = '1') then
            Q_reg <= STD_LOGIC_VECTOR(unsigned(Q_reg)+1);
         else
            Q_reg <= Q_reg;
         end if;

         if (Q_reg = "1111" and ENT = '1') then
            RCO <= '1';
         else
            RCO <= '0';
         end if;
      end if;
   end process;
   Dout <= STD_LOGIC_VECTOR(Q_reg);
end Behavioral;
```

Figure 6. 4-bit Parallel Load Counter VHDL implementation

As shown in Figure 6, a 4-bit adder is implemented. It is able to count from a pre-loaded value. But this function is never used in our multiplier design. Also, it has a ripple carry output to indicate an overflow happened. Again, this feature is not used in our multiplier design.

Top-level Multiplier



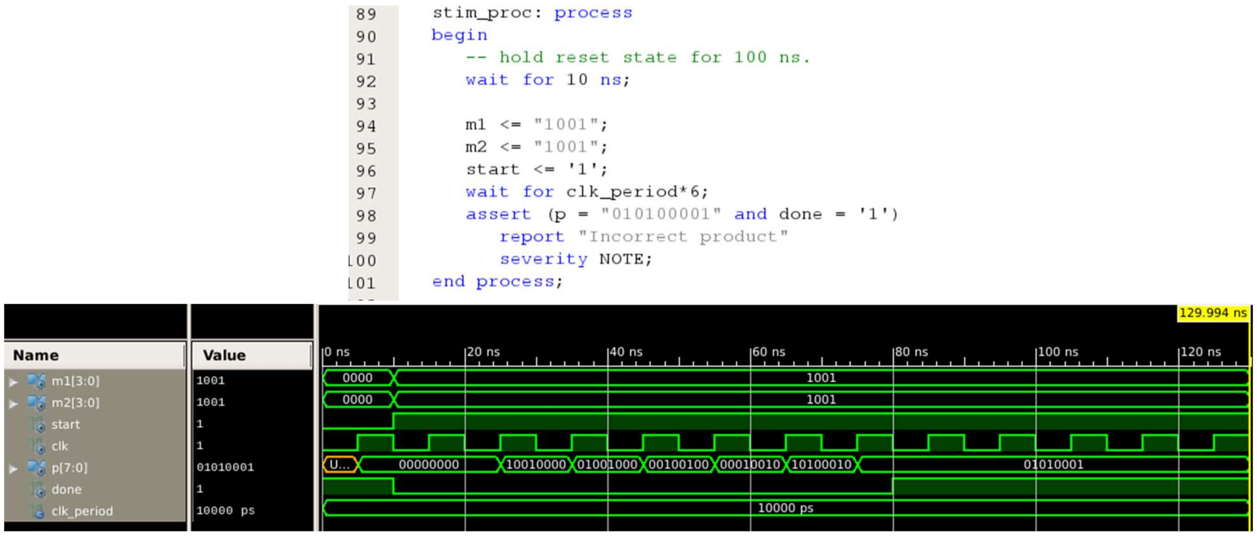
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.mult_components.ALL;

entity Multiplier is
   port(
      m1, m2: in std_logic_vector(3 downto 0);
      p: out std_logic_vector(7 downto 0);
      start, clk: in std_logic;
      done: out std_logic
   );
end Multiplier;

architecture Behavioral of Multiplier is
   use work.mult_components.ALL;
   signal m1_out : std_logic_vector(3 downto 0) := "0000";
   signal m2_out : std_logic_vector(3 downto 0) := "0000";
   signal d_out : std_logic_vector(4 downto 0) := "00000";
   signal a_out : std_logic_vector(8 downto 0) := "000000000";
   signal a_in : std_logic_vector(8 downto 0) := "000000000";
   signal c_out : std_logic_vector(3 downto 0) := "0000";
   signal S10, S11, S20, S21, S30, S31 : std_logic := '0';
   signal clr_res_l, clr_cntr_l: std_logic := '1';
   signal en_cntr : std_logic := '0';

begin
   C: Sys_Controller          port map(start,clk,m2_out(0),c_out(2),S10, S11, S20, S21, S30, S31, clr_res_l, clr_cntr_l, en_cntr, done);
   D: N_Bit_Adder             generic map(4)
                              port map(m1, a_out(7 downto 4), d_out);
   A: N_Bit_Shift_Register    generic map(9)
                              port map(a_in, a_out, clk, clr_res_l, S30, S31, '0', '0');
   MUL1: N_Bit_Shift_Register generic map(4)
                              port map(m1, m1_out, clk, '1', S10, S11, '0', '0');
   MUL2: N_Bit_Shift_Register generic map(4)
                              port map(m2, m2_out, clk, '1', S20, S21, '0', '0');
   CNT: N_Bit_Counter         port map(clk, '1', en_cntr, '1', clr_cntr_l,"1111", c_out, open);
   a_in <= std_logic_vector(d_out & a_out(3 downto 0));
   p <= a_out(7 downto 0);
end Behavioral;
```

Figure 7. Top-level Multiplier VHDL Implementation

Finally, a top-level multiplier is implemented to instantiate all previous components. Also, it declares all the internal and external signals needed. Besides, it maps all the internal and external signals to each component acting like wires in the circuit. The VHDL implementation is shown in Figure 7.

Test Bench

Test case 1

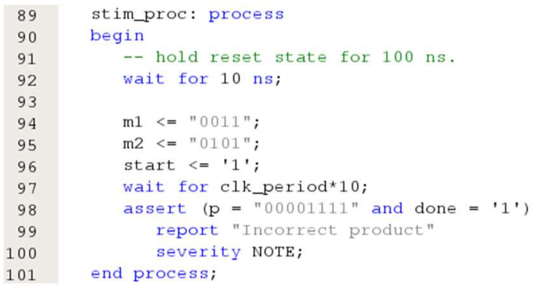


```vhdl
stim_proc: process
begin
   -- hold reset state for 100 ns.
   wait for 10 ns;

   m1 <= "1001";
   m2 <= "1001";
   start <= '1';
   wait for clk_period*6;
   assert (p = "010100001" and done = '1')
      report "Incorrect product"
      severity NOTE;
end process;
```

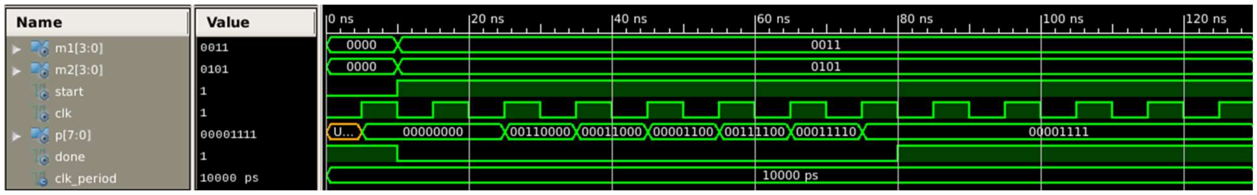
Figure 8. Test bench 1 with the simulation result

For simplicity, only the simulation process is included for VHDL code as shown in Figure 8. This test case calculates the product of 1001b*1001b. We can see that for a certain amount of clock cycles, then we get the result 0101001 with done set to high indicating the calculation is finished. The assertion showed that we are getting the correct result.

Test case 2



```vhdl
stim_proc: process
begin
   -- hold reset state for 100 ns.
   wait for 10 ns;

   m1 <= "0011";
   m2 <= "0101";
   start <= '1';
   wait for clk_period*10;
   assert (p = "00001111" and done = '1')
      report "Incorrect product"
      severity NOTE;
end process;
```



Name	Value
m1[3:0]	0011
m2[3:0]	0101
start	1
clk	1
p[7:0]	00001111
done	1
clk_period	10000 ps

Figure 9. Test bench 2 with the simulation result

Another test case is performed to test the correctness of the output. From Figure 8 we can see that after done is asserted, the product is 00001111, which passes the assertion and indicated that the result is correct.

IV. Comparison

One thing I noticed during the programming and testing is that the counter actually starts counting as from the load state. Since the output is updated as soon as the new state is switched, the result always shifts 1 bit less than the expected result, which makes the product always 1 time larger than the correct answer. I fixed this by stop triggering the counter in the load process and only count in the shifting process. Finally, the result is correct.

V. Conclusion

After certain testing, the result shown that this implementation of 4-bit multiplier is correct. And the component do have the ability to calculate any 4-bit binary multiplication.

Appendix

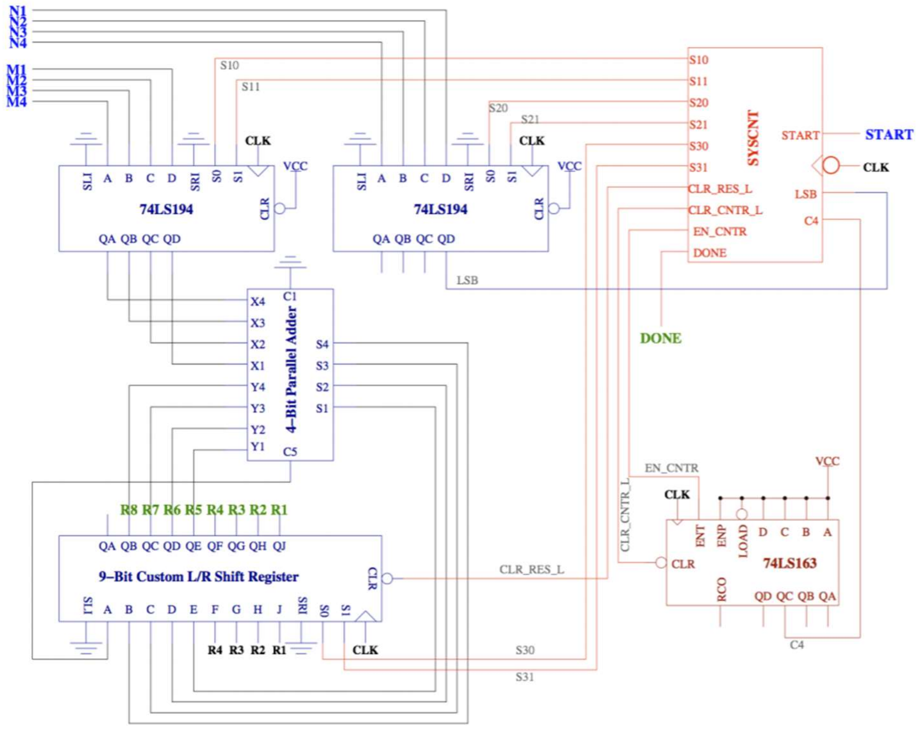


Figure 1. Multiplication Circuit

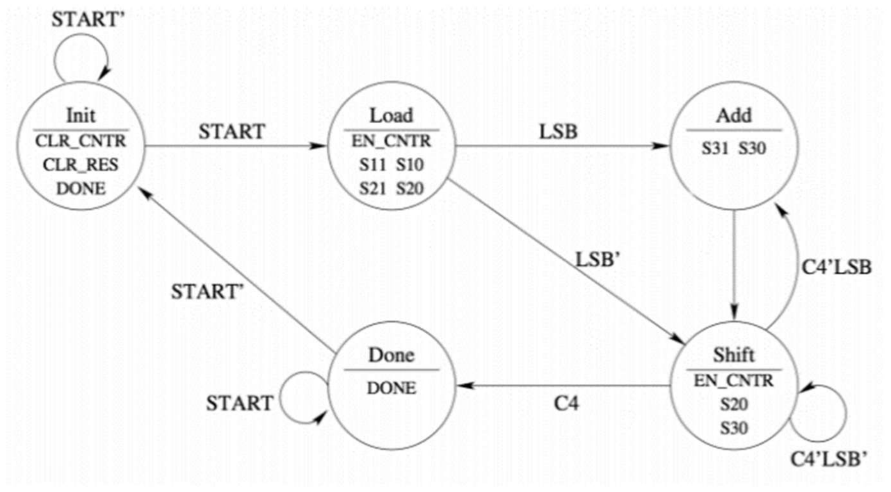
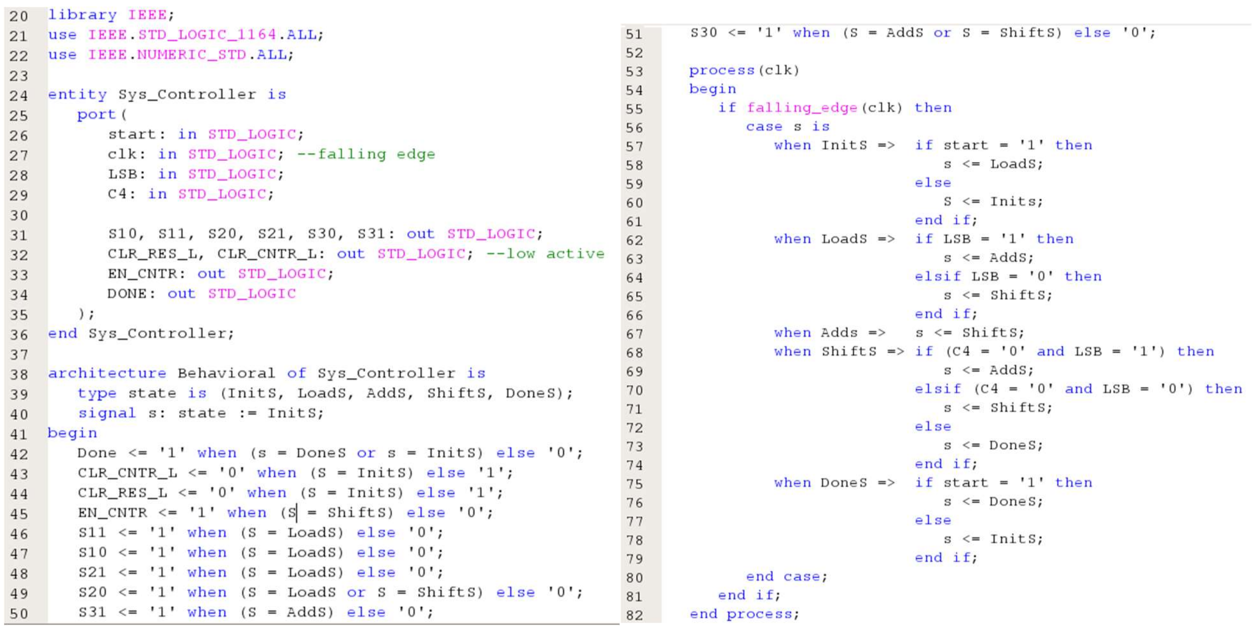
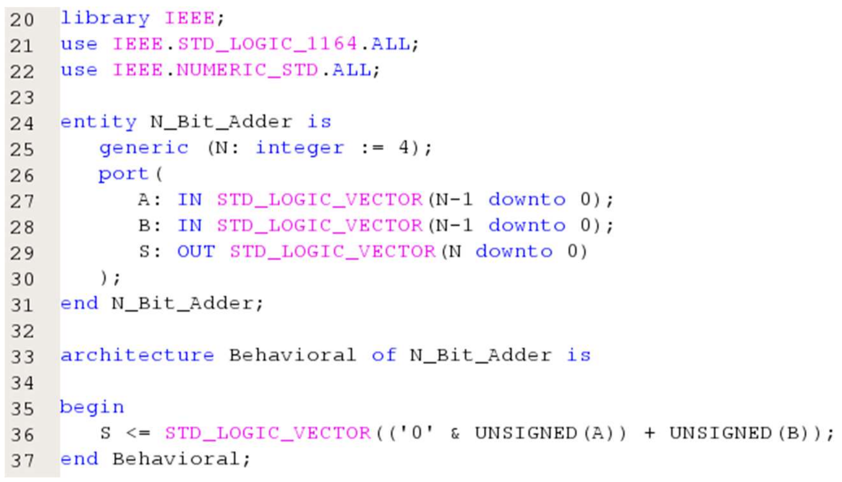


Figure 2. System Controller State Diagram



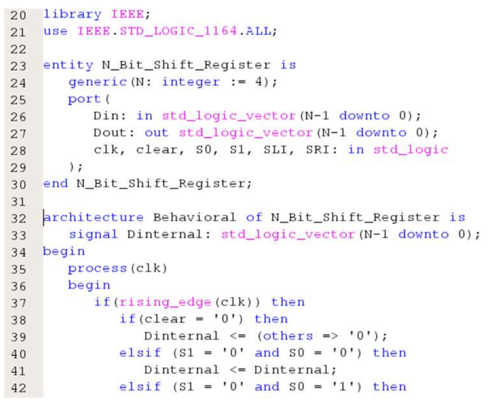
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Sys_Controller is
   port(
      start: in STD_LOGIC;
      clk: in STD_LOGIC; --falling edge
      LSB: in STD_LOGIC;
      C4: in STD_LOGIC;

      S10, S11, S20, S21, S30, S31: out STD_LOGIC;
      CLR_RES_L, CLR_CNTR_L: out STD_LOGIC; --low active
      EN_CNTR: out STD_LOGIC;
      DONE: out STD_LOGIC
   );
end Sys_Controller;

architecture Behavioral of Sys_Controller is
   type state is (InitS, LoadS, AddS, ShiftS, DoneS);
   signal s: state := InitS;
begin
   Done <= '1' when (s = DoneS or s = InitS) else '0';
   CLR_CNTR_L <= '0' when (S = InitS) else '1';
   CLR_RES_L <= '0' when (S = InitS) else '1';
   EN_CNTR <= '1' when (S = ShiftS) else '0';
   S11 <= '1' when (S = LoadS) else '0';
   S10 <= '1' when (S = LoadS) else '0';
   S21 <= '1' when (S = LoadS) else '0';
   S20 <= '1' when (S = LoadS or S = ShiftS) else '0';
   S31 <= '1' when (S = AddS) else '0';
   S30 <= '1' when (S = AddS or S = ShiftS) else '0';

   process(clk)
   begin
      if falling_edge(clk) then
         case s is
            when InitS =>  if start = '1' then
                              s <= LoadS;
                           else
                              S <= InitS;
                           end if;
            when LoadS =>  if LSB = '1' then
                              s <= AddS;
                           elsif LSB = '0' then
                              s <= ShiftS;
                           end if;
            when AddS =>   s <= ShiftS;
            when ShiftS => if (C4 = '0' and LSB = '1') then
                              s <= AddS;
                           elsif (C4 = '0' and LSB = '0') then
                              s <= ShiftS;
                           else
                              s <= DoneS;
                           end if;
            when DoneS =>  if start = '1' then
                              s <= DoneS;
                           else
                              s <= InitS;
                           end if;
         end case;
      end if;
   end process;
```

Figure 3. System Controller VHDL Implementation

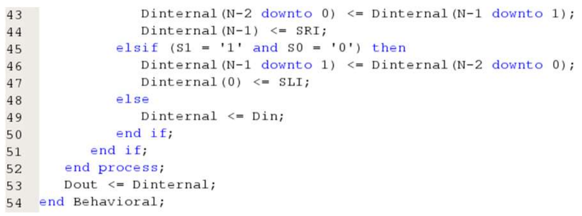


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity N_Bit_Adder is
   generic (N: integer := 4);
   port(
      A: IN STD_LOGIC_VECTOR(N-1 downto 0);
      B: IN STD_LOGIC_VECTOR(N-1 downto 0);
      S: OUT STD_LOGIC_VECTOR(N downto 0)
   );
end N_Bit_Adder;

architecture Behavioral of N_Bit_Adder is

begin
   S <= STD_LOGIC_VECTOR(('0' & UNSIGNED(A)) + UNSIGNED(B));
end Behavioral;
```

Figure 4. 4-bit adder VHDL implementation

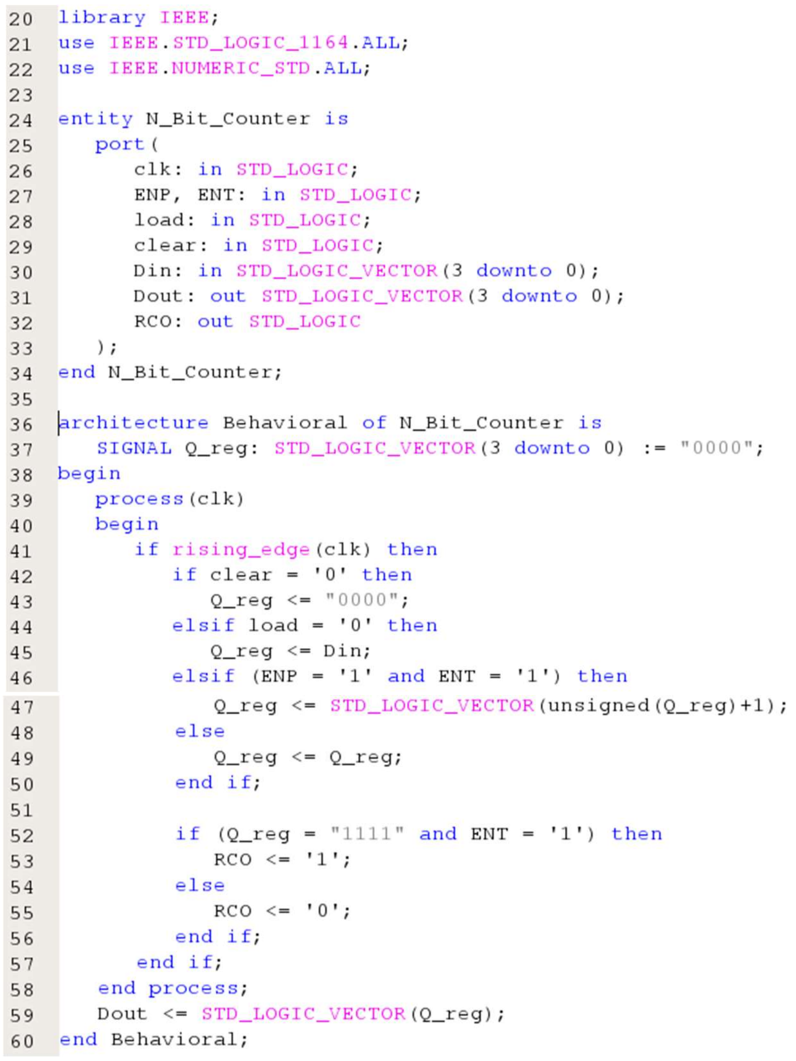


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity N_Bit_Shift_Register is
   generic(N: integer := 4);
   port(
      Din: in std_logic_vector(N-1 downto 0);
      Dout: out std_logic_vector(N-1 downto 0);
      clk, clear, S0, S1, SLI, SRI: in std_logic
   );
end N_Bit_Shift_Register;

architecture Behavioral of N_Bit_Shift_Register is
   signal Dinternal: std_logic_vector(N-1 downto 0);
begin
   process(clk)
   begin
      if(rising_edge(clk)) then
         if(clear = '0') then
            Dinternal <= (others => '0');
         elsif (S1 = '0' and S0 = '0') then
            Dinternal <= Dinternal;
         elsif (S1 = '0' and S0 = '1') then
```



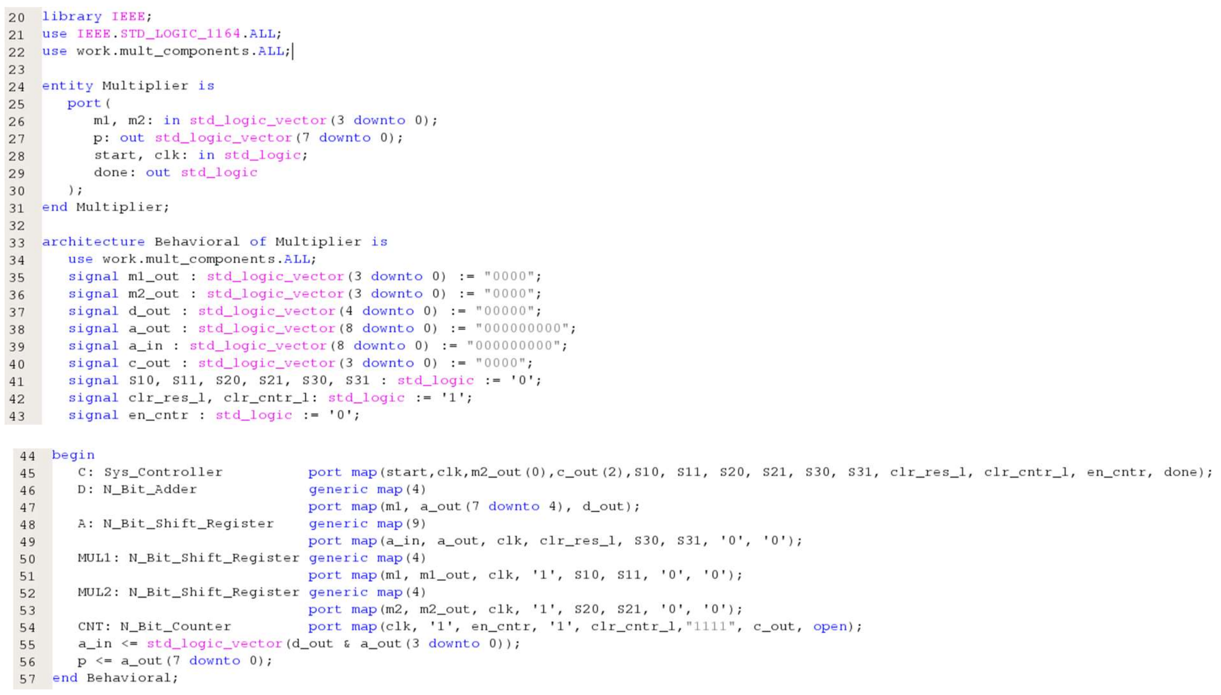
```vhdl
            Dinternal(N-2 downto 0) <= Dinternal(N-1 downto 1);
            Dinternal(N-1) <= SRI;
         elsif (S1 = '1' and S0 = '0') then
            Dinternal(N-1 downto 1) <= Dinternal(N-2 downto 0);
            Dinternal(0) <= SLI;
         else
            Dinternal <= Din;
         end if;
      end if;
   end process;
   Dout <= Dinternal;
end Behavioral;
```

Figure 5. N-bit Parallel Load Bidirectional Shift Register VHDL implementation



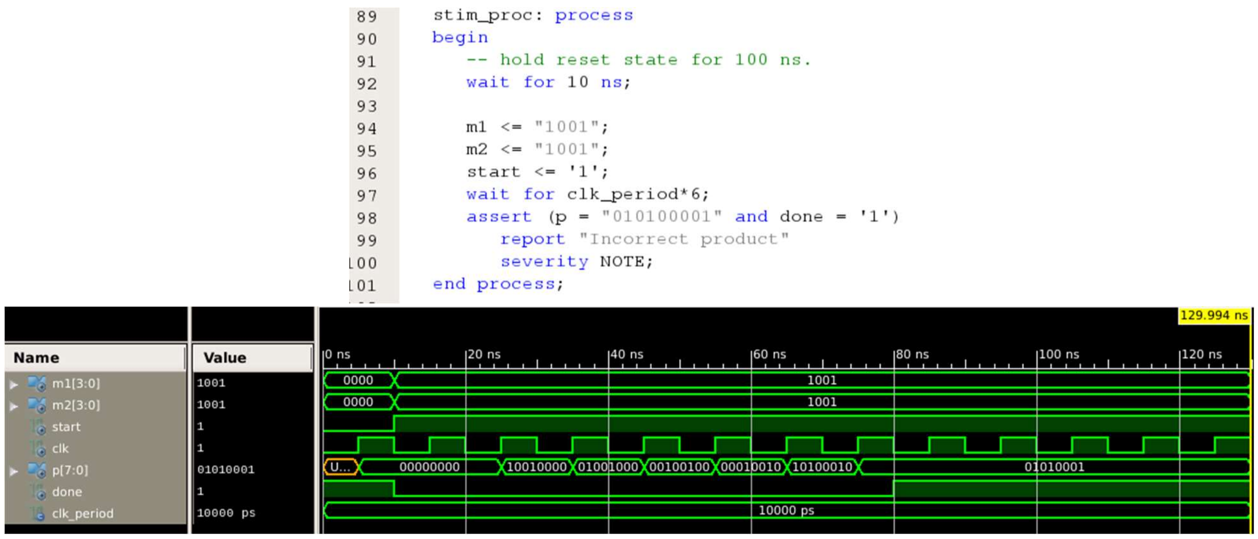
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity N_Bit_Counter is
   port(
      clk: in STD_LOGIC;
      ENP, ENT: in STD_LOGIC;
      load: in STD_LOGIC;
      clear: in STD_LOGIC;
      Din: in STD_LOGIC_VECTOR(3 downto 0);
      Dout: out STD_LOGIC_VECTOR(3 downto 0);
      RCO: out STD_LOGIC
   );
end N_Bit_Counter;

architecture Behavioral of N_Bit_Counter is
   SIGNAL Q_reg: STD_LOGIC_VECTOR(3 downto 0) := "0000";
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if clear = '0' then
            Q_reg <= "0000";
         elsif load = '0' then
            Q_reg <= Din;
         elsif (ENP = '1' and ENT = '1') then
            Q_reg <= STD_LOGIC_VECTOR(unsigned(Q_reg)+1);
         else
            Q_reg <= Q_reg;
         end if;

         if (Q_reg = "1111" and ENT = '1') then
            RCO <= '1';
         else
            RCO <= '0';
         end if;
      end if;
   end process;
   Dout <= STD_LOGIC_VECTOR(Q_reg);
end Behavioral;
```

Figure 6. 4-bit Parallel Load Counter VHDL implementation



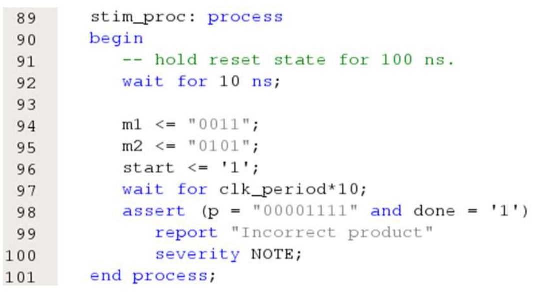
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.mult_components.ALL;

entity Multiplier is
   port(
      m1, m2: in std_logic_vector(3 downto 0);
      p: out std_logic_vector(7 downto 0);
      start, clk: in std_logic;
      done: out std_logic
   );
end Multiplier;

architecture Behavioral of Multiplier is
   use work.mult_components.ALL;
   signal m1_out : std_logic_vector(3 downto 0) := "0000";
   signal m2_out : std_logic_vector(3 downto 0) := "0000";
   signal d_out : std_logic_vector(4 downto 0) := "00000";
   signal a_out : std_logic_vector(8 downto 0) := "000000000";
   signal a_in : std_logic_vector(8 downto 0) := "000000000";
   signal c_out : std_logic_vector(3 downto 0) := "0000";
   signal S10, S11, S20, S21, S30, S31 : std_logic := '0';
   signal clr_res_l, clr_cntr_l: std_logic := '1';
   signal en_cntr : std_logic := '0';

begin
   C: Sys_Controller          port map(start,clk,m2_out(0),c_out(2),S10, S11, S20, S21, S30, S31, clr_res_l, clr_cntr_l, en_cntr, done);
   D: N_Bit_Adder             generic map(4)
                              port map(m1, a_out(7 downto 4), d_out);
   A: N_Bit_Shift_Register    generic map(9)
                              port map(a_in, a_out, clk, clr_res_l, S30, S31, '0', '0');
   MUL1: N_Bit_Shift_Register generic map(4)
                              port map(m1, m1_out, clk, '1', S10, S11, '0', '0');
   MUL2: N_Bit_Shift_Register generic map(4)
                              port map(m2, m2_out, clk, '1', S20, S21, '0', '0');
   CNT: N_Bit_Counter         port map(clk, '1', en_cntr, '1', clr_cntr_l,"1111", c_out, open);
   a_in <= std_logic_vector(d_out & a_out(3 downto 0));
   p <= a_out(7 downto 0);
end Behavioral;
```

Figure 7. Top-level Multiplier VHDL Implementation



```vhdl
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 10 ns;

      m1 <= "1001";
      m2 <= "1001";
      start <= '1';
      wait for clk_period*6;
      assert (p = "010100001" and done = '1')
         report "Incorrect product"
         severity NOTE;
   end process;
```

Figure 8. Test bench 1 with the simulation result



```
stim_proc: process
begin
   -- hold reset state for 100 ns.
   wait for 10 ns;

   m1 <= "0011";
   m2 <= "0101";
   start <= '1';
   wait for clk_period*10;
   assert (p = "00001111" and done = '1')
      report "Incorrect product"
      severity NOTE;
end process;
```

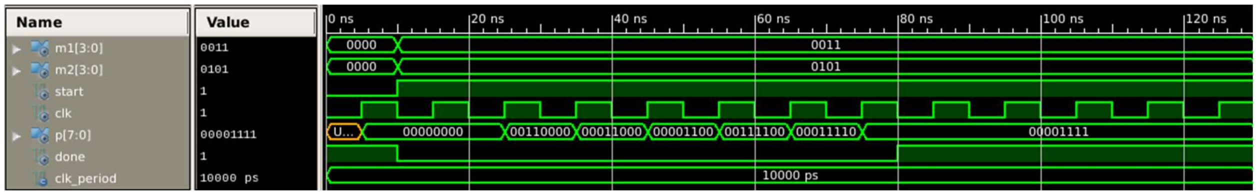


Figure 9. Test bench 2 with the simulation result