Lab 3: Multiplier with Control Unit Design Using VHDL

ENGG3050: Reconfigurable Computing Systems

Instructor:
Dr. Shawki Areibi

Group 42: Tuesday-3:30 Section

Bilal Ayyache: 0988616
Anthony Granic: 0994824

Lab Start Date: October 3, 2019
Lab End Date: October 8, 2019

Contents

List of Figures

List of Tables

1 Project Implementation

1.1 Problem Statement

The main objective of lab 3 is to design and implement a multiplier that can multiply two 4-bit unsigned binary numbers using Structural VHDL.

1.2 Assumptions and Constraints

Constraints includes using an FPGA board, programming in VHDL, and testing project using Vivado as a design tool. Design must be mapped successfully on the FPGA board with no errors. VHDL code must be structural and not behavioral (for most parts). It was also assumed that the FPGA board was fully functional.

1.3 System Overview Justification Of Design

The system made incorporates a decoder based control unit to decide which stage of standard serial multiplication the system is at. After 32 iterations the program stops and displays the output. This method of multiplication incorporates multiple shift registers and an ALU to add the shifted values of the numbers together.

1.3.1 Designed System Overview

The final design incorporates 3 shift registers, an ALU (only adding is used), and a control unit. The final design has the desired output of the product of two unsigned 4-bit numbers displayed in a hexadecimal format.

A (binary)	B (binary)	Product (hex)
0100	0011	0C
1010	0100	28
0010	1111	20

Table 1: Example of multiplier behaviour

1.3.2 System Functionality and Reasons Behind Design

The system operates based on the decisions of the control unit. The control unit decides when the system should add, load, shift, or wait based on the current state of the least significant bit of the product or the iteration counter. The registers receive these signals and shift or output based on the signal received. Once the iteration counter reaches 32 the program waits for the next time it needs to multiply the operands.

This system has two 4-bit inputs that represent unsigned numbers. This was done using 8 switches. These inputs and the 8-bit output were mapped to a seven segment display to be represented as hexadecimal values. A button on the board initiates the multiplication of the currently inputted values. An LED is used to indicate the completion of multiplication.

This design was an efficient way to implement a serial multiplier and allowed complete control of when the operands were to be multiplied. This reduced error in the display of the hexadecimal values on the seven segment display as intermediate values were not displayed or multiplied.

1.4 System design diagrams

The design of this system was based off of these diagrams referenced from chapter 3.5 of Computer Organization and Design 5th edition by David A. Patterson and John L. Hennessy.

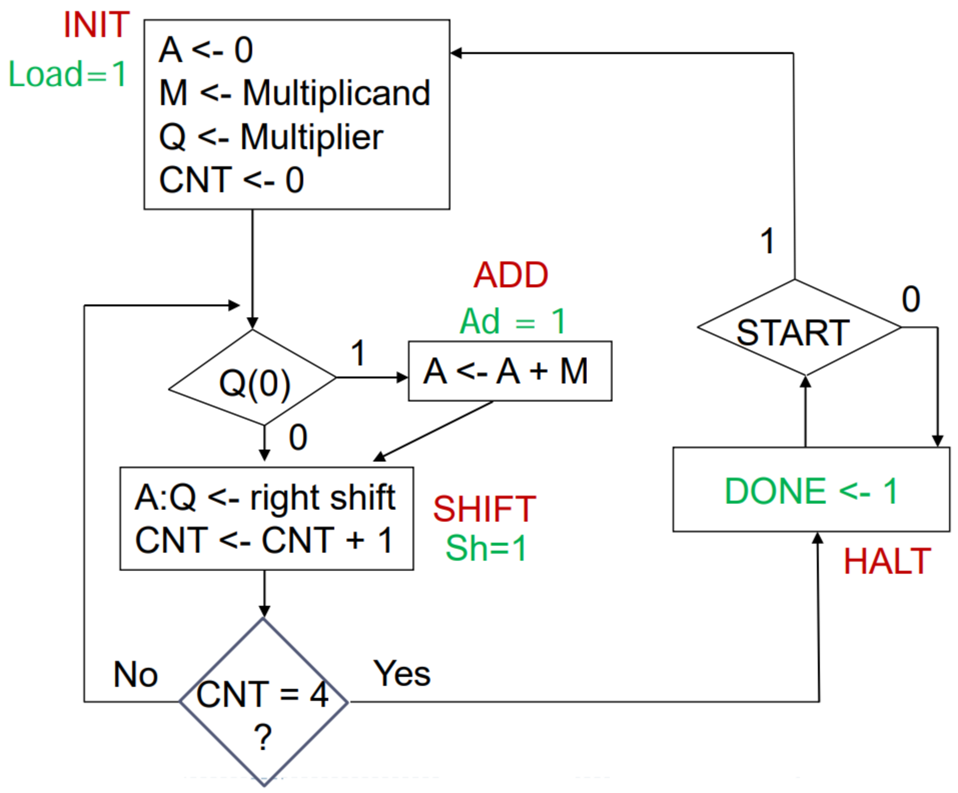
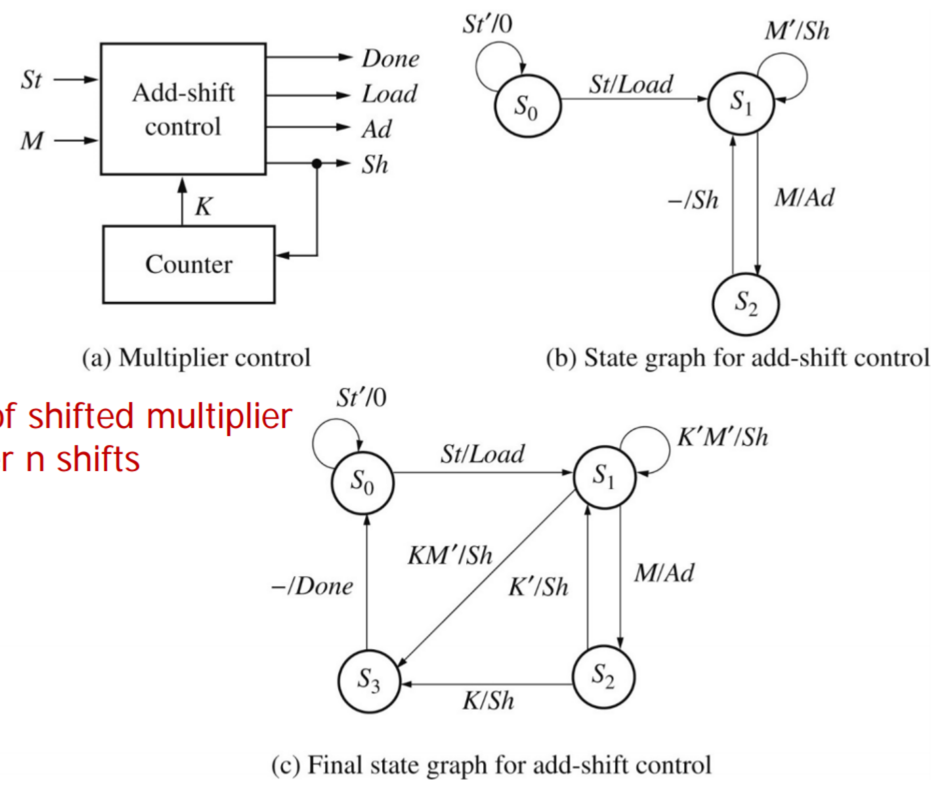


Figure 1: State Diagram



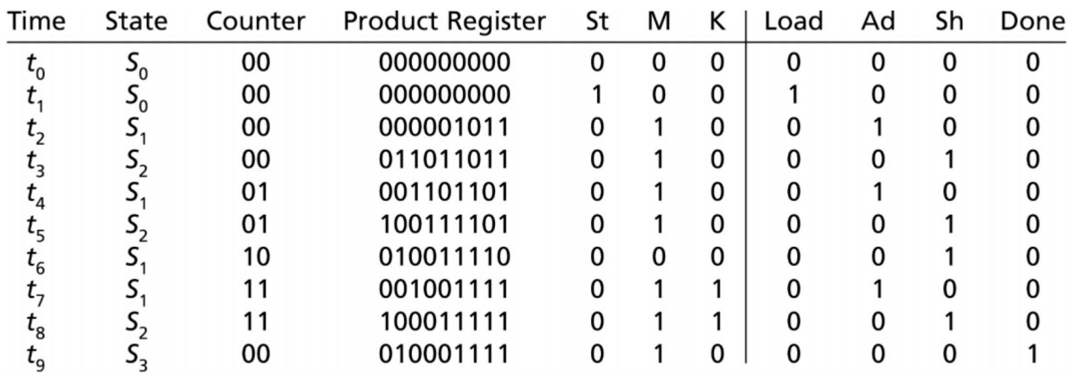
(a) Multiplier control

(b) State graph for add-shift control

M = LSB of shifted multiplier
K = 1 after n shifts

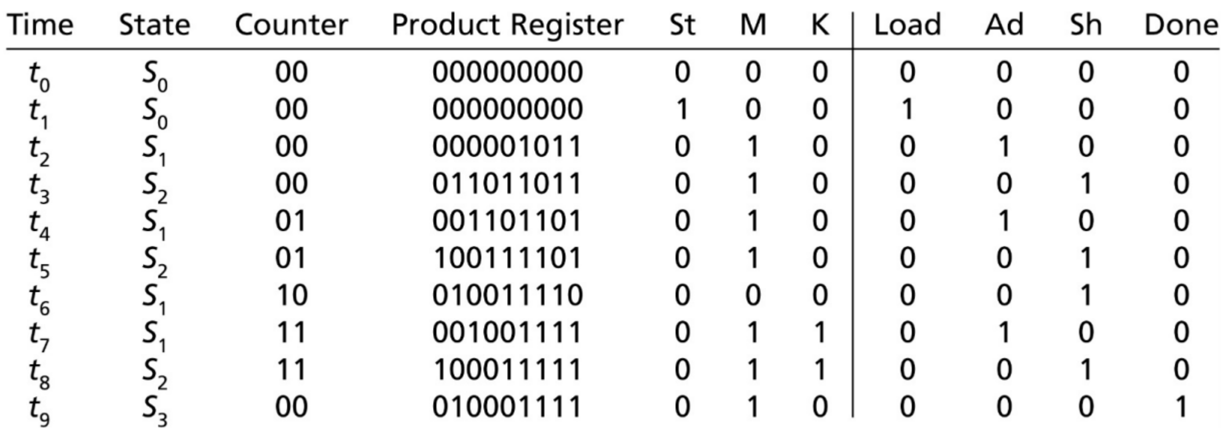
(c) Final state graph for add-shift control

Figure 2: Meele Machine



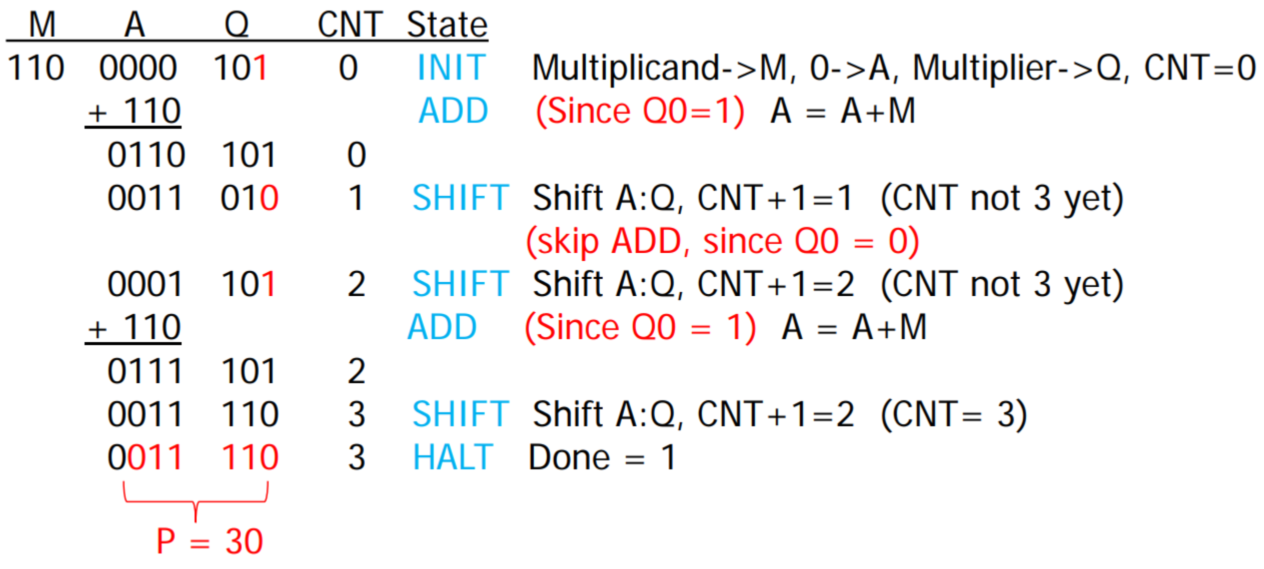
Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t_0	S_0	00	000000000	0	0	0	0	0	0	0
t_1	S_0	00	000000000	1	0	0	1	0	0	0
t_2	S_1	00	000001011	0	1	0	0	1	0	0
t_3	S_2	00	011011011	0	1	0	0	0	1	0
t_4	S_1	01	001101101	0	1	0	0	1	0	0
t_5	S_2	01	100111101	0	1	0	0	0	1	0
t_6	S_1	10	010011110	0	0	0	0	0	1	0
t_7	S_1	11	001001111	0	1	1	0	1	0	0
t_8	S_2	11	100011111	0	1	1	0	0	1	0
t_9	S_3	00	010001111	0	1	0	0	0	0	1

Figure 3: Stages counter



Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t_0	S_0	00	000000000	0	0	0	0	0	0	0
t_1	S_0	00	000000000	1	0	0	1	0	0	0
t_2	S_1	00	000001011	0	1	0	0	1	0	0
t_3	S_2	00	011011011	0	1	0	0	0	1	0
t_4	S_1	01	001101101	0	1	0	0	1	0	0
t_5	S_2	01	100111101	0	1	0	0	0	1	0
t_6	S_1	10	010011110	0	0	0	0	0	1	0
t_7	S_1	11	001001111	0	1	1	0	1	0	0
t_8	S_2	11	100011111	0	1	1	0	0	1	0
t_9	S_3	00	010001111	0	1	0	0	0	0	1

Figure 4: System Outputs



M	A	Q	CNT	State	
110	0000	101	0	INIT	Multiplicand->M, 0->A, Multiplier->Q, CNT=0
	+ 110			ADD	(Since Q0=1) A = A+M
	0110	101	0		
	0011	010	1	SHIFT	Shift A:Q, CNT+1=1 (CNT not 3 yet)
					(skip ADD, since Q0 = 0)
	0001	101	2	SHIFT	Shift A:Q, CNT+1=2 (CNT not 3 yet)
	+ 110			ADD	(Since Q0 = 1) A = A+M
	0111	101	2		
	0011	110	3	SHIFT	Shift A:Q, CNT+1=2 (CNT= 3)
	0011	110	3	HALT	Done = 1

P = 30

Figure 5: Stages overview

2 Software Implementation (VHDL)

The multiplier was designed using Structural VHDL. The multiplier unit consists of a control unit, 3 registers, and an adder. The top level module in section 2.1 highlights the full system, The main objective of this module is to connect the multiplier designed to the 7 segment code.

The seven segment code in section 2.2 works using a clock divider that cycles through the anode activations to display one number at a time. The number displayed is then sent to the appropriate location on the display based on the anodes and the position in the binary representation of the number to be displayed. A decoder is then used to translate the binary representation to the appropriate output for the seven segment display representation.

The multiplier unit takes two 4-line inputs and sends out an 8-line output. The multiplier's algorithm starts when the start signal switches from low to high. The algorithm is complete when done signal is high. The multiplier code in section 2.3 connects the control unit, ALU(adder), and the registers together. The functionality of the code in the section below is described using comments.

2.1 Top-Level Module

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

entity TopLevel is
---Setting the top level inputs and outputs
Port ( SW  :  in STD_LOGIC_VECTOR (15 downto 0);
           LED : out STD_LOGIC_VECTOR (15 downto 0);
           Anodes : out STD_LOGIC_VECTOR (7 downto 0);
           sevenOut : out STD_LOGIC_VECTOR (6 downto 0);
           clock : in STD_LOGIC;
           BTNC : in STD_LOGIC
           );

end TopLevel;

architecture Behavioral of TopLevel is
---Inserting the seven segment component
component sevenSegDriver is
         Port (
           displayed_number: in STD_LOGIC_VECTOR (31 downto 0);
               clock_100Mhz : in STD_LOGIC;-- 100Mhz clock on Basys 3 FPGA board
           reset : in STD_LOGIC; -- reset
           Anode_Activate : out STD_LOGIC_VECTOR (7 downto 0);-- 8 Anode signals
           LED_out : out STD_LOGIC_VECTOR (6 downto 0));
end component;
---Inserting the multiplier component
component multUnit is
port (
    MPLIER: in std_logic_vector(3 downto 0);
    MCAND: in std_logic_vector(3 downto 0);
    result: out std_logic_vector(7 downto 0);
    startSignal: in std_logic;
    clockSignal: in std_logic;
    doneSignal: out std_logic);
end component;
---Signals to be used
signal multout: std_logic_vector(31 downto 0);

begin

    multout(27 downto 24) <= SW(7 downto 4); ---Set switch 4,5,6,and 7 to Multiplier
    multout(19 downto 16) <= SW(3 downto 0); ---Set switch 1,2,3,and 4 to Multiplican

    multiplier: multUnit ---Connecting signals to the multiplier Unit
    port map(SW(7 downto 4), SW(3 downto 0), multout(7 downto 0), BTNC, clock, LED(0));

    sevenSeg: sevenSegDriver ---Connecting signals to the 7 segment
    port map(multout, clock, '0', Anodes, sevenOut);

end Behavioral;
```

2.2 7-Segment Display

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

entity sevenSegDriver is
         Port (
 displayed_number: in STD_LOGIC_VECTOR (31 downto 0);

         clock_100Mhz : in STD_LOGIC;-- 100Mhz clock
           reset : in STD_LOGIC; -- reset
           Anode_Activate : out STD_LOGIC_VECTOR (7 downto 0);-- 8 Anode signals
           LED_out : out STD_LOGIC_VECTOR (6 downto 0));
end sevenSegDriver;

architecture Behavioral of sevenSegDriver is
signal LED_BCD: STD_LOGIC_VECTOR (3 downto 0);
signal refresh_counter: STD_LOGIC_VECTOR (19 downto 0);
signal LED_activating_counter: std_logic_vector(2 downto 0);

begin

process(LED_BCD)
begin
    case LED_BCD is
    when "0000" => LED_out <= "0000001"; -- "0"
    when "0001" => LED_out <= "1001111"; -- "1"
    when "0010" => LED_out <= "0010010"; -- "2"
    when "0011" => LED_out <= "0000110"; -- "3"
    when "0100" => LED_out <= "1001100"; -- "4"
    when "0101" => LED_out <= "0100100"; -- "5"
    when "0110" => LED_out <= "0100000"; -- "6"
    when "0111" => LED_out <= "0001111"; -- "7"
    when "1000" => LED_out <= "0000000"; -- "8"
    when "1001" => LED_out <= "0000100"; -- "9"
    when "1010" => LED_out <= "0001000"; -- a
    when "1011" => LED_out <= "1100000"; -- b
    when "1100" => LED_out <= "0110001"; -- C
    when "1101" => LED_out <= "1000010"; -- d
    when "1110" => LED_out <= "0110000"; -- E
    when "1111" => LED_out <= "0111000"; -- F
         when others => LED_out <= "0000001";
    end case;
end process;

process(clock_100Mhz,reset)
begin
    if(reset='1') then
        refresh_counter <= (others => '0');
    elsif(rising_edge(clock_100Mhz)) then
        refresh_counter <= refresh_counter + 1;
    end if;
end process;
 LED_activating_counter <= refresh_counter(19 downto 17);

process(LED_activating_counter, displayed_number)
begin
    case LED_activating_counter is
    when "000" =>
```

```
        Anode_Activate <= "01111111";
                  LED_BCD <= displayed_number(3 downto 0);
    when "001" =>
        Anode_Activate <= "10111111";
        LED_BCD <= displayed_number(7 downto 4);
    when "010" =>
        Anode_Activate <= "11011111";
        LED_BCD <= displayed_number(11 downto 8);
    when "011" =>
        Anode_Activate <= "11101111";
        LED_BCD <= displayed_number(15 downto 12);
    when "100" =>
        Anode_Activate <= "11110111";
        LED_BCD <= displayed_number(19 downto 16);
    when "101" =>
        Anode_Activate <= "11111011";
        LED_BCD <= displayed_number(23 downto 20);
    when "110" =>
        Anode_Activate <= "11111101";
        LED_BCD <= displayed_number(27 downto 24);
    when "111" =>
        Anode_Activate <= "11111110";
        LED_BCD <= displayed_number(31 downto 28);
    when others => Anode_Activate <="11101111";
    end case;
end process;

end Behavioral;
```

2.3 Multiplier

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library xil_defaultlib;
---Declaring Entity of Multiplier Unit
entity multUnit is
port (
    MPLIER: in std_logic_vector(3 downto 0);
    MCAND: in std_logic_vector(3 downto 0);
    result: out std_logic_vector(7 downto 0);
    startSignal: in std_logic;
    clockSignal: in std_logic;
    doneSignal: out std_logic);
end multUnit;
---Declaring Architecture of Multiplication Unit
architecture Behavioral of multUnit is
    ---Using Component Package
    use xil_defaultlib.componentPackage.all;
    ---Signals used to connect components
    signal regM,regQ: std_logic_vector (3 downto 0); ---Carries output of Register M and Q
    signal regD,regProduct: std_logic_vector (4 downto 0);  ---Last bit is a carry bit
    signal loadSignal,shiftSignal,addSignal: std_logic; ---State signals

begin
    ---Porting components
    Control: controlUnit generic map (2)
    port map (clockSignal,regQ(0),startSignal,loadSignal,shiftSignal,addSignal,doneSignal);

    Adder: ALU generic map (4)
    port map (regProduct(3 downto 0),regM,regD);

    MPCReg: multRegister generic map (4)
    port map (MCAND,regM,clockSignal,loadSignal,'0','0','0');

    MPLReg: multRegister generic map (4)
    port map (MPLIER,regQ,clockSignal,loadSignal,shiftSignal,'0',regProduct(0));
    ---Set N to 5 due to carry included in input
    ProdReg: multRegister generic map (5)
    port map (regD,regProduct,clockSignal,addSignal,shiftSignal,loadSignal,'0');
    ---Set result ignore carry
    result <= regProduct(3 downto 0) & regQ;
    ---END OF COMPONENT CONNECTION
end Behavioral;
```

2.4 Control Unit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
---Setting Control Unit inputs and outputs
entity controlUnit is
    generic (N: integer := 2);
    port (
        Clock: in std_logic; ---Clock connection
        LSB: in std_logic; ---Value that contains LSB
        startPulse: in std_logic; ---Start signal
        loadSignal: out std_logic; ---Load signal
        shiftSignal: out std_logic; ---Shift signal
        SignalToA: out std_logic; ---Signal to Product Regg
        Complete: out std_logic --- Done Signal
    );
end controlUnit;

    architecture Behavioral of controlUnit is
    type states is (waitStage,initStage,sendStage,addStage,shiftStage);
    signal state: states := waitStage;
    signal iterationSignal: unsigned(N-1 downto 0);

begin
    Complete <= '1' when state = waitStage else '0'; ---Terminate the Algorithm
    loadSignal <= '1' when state = initStage else '0'; ---Set load signal if init
    SignalToA <= '1' when state = addStage else '0'; ---Set ALU signal if add
    shiftSignal <= '1' when state = shiftStage else '0'; ----Set shift signal

    process(Clock) ---Run as clock ticks
    begin
        if rising_edge(Clock) then ---At rising edge of clock, set signals
            case state is ---Check State
                when waitStage => if startPulse = '1' then
                    state <= initStage; ---set state to initialization
                end if;

                when initStage => state <= sendStage;
                when sendStage => if (LSB = '1') then
                        state <= addStage; ---set state to addition
                    else
                        state <= shiftStage;
                    end if;
                when addStage => state <= shiftStage;
                when shiftStage => if (iterationSignal = 2**N - 1) then
                        state <= waitStage; ---Set state to idle
                    else
                        state <= sendStage; ---Set state to send signals
                    end if;
            end case;
        end if;
    end process;

    process(Clock) ---Count itterations
    begin
        if rising_edge(Clock) then
            if state = initStage then
                iterationSignal <= to_unsigned(0,N);
            elsif state = shiftStage then
                iterationSignal <= iterationSignal + 1; ---Increment counter
            end if;
        end if;
    end process;

end Behavioral;
```

2.5 Registers

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
---Registers used in the multiplication design. The same register implementation
---is used to keep design simple and easy to understand
entity multRegister is
    generic (N: integer := 4);
    port (
        regIn: in std_logic_vector(N-1 downto 0);  ---Value to be shifted
        regOut: out std_logic_vector(N-1 downto 0); ---Output of value after shifting
        clock: in std_logic; ---Running the clock to process the register
        LoadEnable: in std_logic; ---Load signal that allows the register to read the regIn value
        shiftEnable: in std_logic; ---Signal that tells register to shift input
        reset: in std_logic; ---reset signal
        SerialInput: in std_logic ---Serial input if register is not in idle mode
    );
    end multRegister;

architecture Behavioral of multRegister is
    signal regState: std_logic_vector(N-1 downto 0);
    begin
        process (clock) ---Runs the code below as clock ticks
        begin
            if (rising_edge(clock)) then ---On high edge set the registerState Value
                if (reset = '1') then
                    regState <= (others => '0'); ---If reset is high reset the value
                elsif (LoadEnable = '1') then
                    regState <= regIn; --- Load regIn
                elsif (shiftEnable = '1') then
                    regState <= SerialInput & regState(N-1 downto 1); ---Shift register state and
                end if;
            end if;
        end process;
regOut <= regState; ---Set register output to regState

end Behavioral;
```

2.6 Component Package

```
library ieee;
use ieee.std_logic_1164.all;

package componentPackage is
    ---Declaring Control Unit
    component controlUnit
    generic (N: integer := 2);
    port (
        Clock: in std_logic;
        LSB: in std_logic;
        startPulse: in std_logic;
        loadSignal: out std_logic;
        shiftSignal: out std_logic;
        SignalToA: out std_logic;
        Complete: out std_logic
    );
    end component;
    ---Declaring ALU
    component ALU
    generic (N: integer := 4);
    port(
        in1: in std_logic_vector(N-1 downto 0);
        in2: in std_logic_vector(N-1 downto 0);
        output: out std_logic_vector(N downto 0)
    );
    end component;
    ---Declaring Registers
    component multRegister is
    generic (N: integer := 4);
    port (
        regIn: in std_logic_vector(N-1 downto 0);
        regOut: out std_logic_vector(N-1 downto 0);
        clock: in std_logic;
        LoadEnable: in std_logic;
        shiftEnable: in std_logic;
        reset: in std_logic;
        SerialInput: in std_logic
    );
    end component;
    ---END OF PACKAGE
end package;
```

2.7 Bench Test

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

entity testMultTop is
end testMultTop;

architecture Behavioral of testMultTop is
    component multUnit is
port (
    MPLIER: in std_logic_vector(3 downto 0);
    MCAND: in std_logic_vector(3 downto 0);
    result: out std_logic_vector(7 downto 0);
    startSignal: in std_logic;
    ClockSignal: in std_logic;
    doneSignal: out std_logic);
end component;

    signal MPLIER: std_logic_vector(3 downto 0);
    signal MCAND: std_logic_vector(3 downto 0);
    signal result: std_logic_vector(7 downto 0);
    signal startSignal: std_logic:= '0';
    signal ClockSignal: std_logic;
    signal doneSignal: std_logic;


    component binary_multiplier
    port(
        CLK, RESET, G, LOADB, LOADQ: in std_logic;
        MULT_IN: in std_logic_vector(3 downto 0);
        MULT_OUT: out std_logic_vector(7 downto 0));
    end component;

    signal CLK, RESET, G, LOADB, LOADQ: std_logic;
    signal MULT_IN: std_logic_vector(3 downto 0);
    signal MULT_OUT: std_logic_vector(7 downto 0);


begin
uut: multUnit
    port map(
       MPLIER => MPLIER,
       MCAND => MCAND,
       result => result,
       startSignal => startSignal,
       ClockSignal => ClockSignal,
       doneSignal => doneSignal
    );

uut1: binary_multiplier
    port map(
        CLK => Clk,
        RESET => RESET,
        G => G,
        LOADB => LOADB,
        LOADQ => LOADQ,
        MULT_IN => MULT_IN,
        MULT_OUT => MULT_OUT
    );

   clk_process :process
   begin
```

```
        ClockSignal <= '0';
        CLK <= '0';
        wait for 10ns;
        ClockSignal <= '1';
        CLK <= '1';
        wait for 10ns;
   end process;

   stim_proc:
    process
    begin

    MPLIER <= std_logic_vector(to_unsigned(4,4));
    MCAND <=  std_logic_vector(to_unsigned(6,4));

    startSignal <= '0', '1' after 5 ns, '0' after 40 ns;

    wait until doneSignal = '1';

    wait for 5ns;


    LOADB <= '0';
    LOADQ <= '0';
    wait for 50 ns;
    MULT_IN <= std_logic_vector(to_unsigned(6,4));
    LOADB <= '1';
    wait for 50 ns;
    LOADB <='0';
    MULT_IN <= std_logic_vector(to_unsigned(4,4));
    LOADQ <='1';
    wait for 50ns;
    LOADQ <= '0';
    G <= '1';

    end process;
end Behavioral;
```

2.8 Top ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
---Setting the entity of the nBit ALU
entity nBitAlu is
    generic (
                N : integer := 8   ---Building a 4 Bit ALU
        );
        port (
                in1, in2 : in std_logic_vector(N - 1 downto 0);  ---4 inputs and 3 outputs
                output : out std_logic_vector(N - 1 downto 0)
        );
end nBitAlu;
---Setting Architecture of N bit ALU
architecture Structural of nBitAlu is
---Using the ALU designed previously
component ALU
    Port ( in1 : in  std_logic;
           in2 : in  std_logic;
           sel : in  STD_LOGIC_vector(1 downto 0);
           cin : in  STD_LOGIC;
           Alu_out : out std_logic;
           cout : out  STD_LOGIC);
end component;
---Setting a 5 signal lines to connect the carrys
signal tempFour: std_logic_vector(N downto 0);
---Always run this code below. Code will generate n ALUs
begin
    tempFour(0) <= Cin; ---Setting up the first carry to the first signal line

        GEN_ALU: ---Generating Function
        for i in 0 to N - 1 generate
                nALUDesign : ALU ---4 ALUs are generated here
                port map (in1(i), in2(i), sel, tempFour(i),  Alu_out(i) ,tempFour(i+1));
                ---Connecting the ALUs together
        end generate GEN_ALU;

        OVF <= tempFour(N-2) xor tempFour(N-1); ---Generating overflow signal
        process (in1,in2)
        begin
```

```
                if in1 = in2 then ---Generating a zero signal
                        zero <= '1';
                else
                        zero <= '0';
                end if;
        end process; ---End Process
end Structural;
```

2.9 ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
---Setting inputs and outputs
entity ALU is
    Port ( in1 : in  std_logic;
           in2 : in  std_logic;
           sel : in  STD_LOGIC_vector(1 downto 0);
           cin : in  STD_LOGIC;
           Alu_out : out std_logic;
           cout : out  STD_LOGIC);
end ALU;

architecture Structural of ALU is
---Declaring Component BitAdder to use in ALU
component bitAdder
    port (
                in1, in2 : in std_logic;
                mode : in std_logic;
                adder_out : out std_logic;
                carryOutAdder : out std_logic
        );
end component;
---Signals used to connect inputs and outputs to gates
signal a: std_logic;
signal b: std_logic;
signal carryIn: STD_LOGIC := '0';
---If sel,In1,In2,or Carryin is high run the ALU
begin
   process(sel,in1,in2,cin)
    begin
        case sel is
```

```
                        when "00" => ---When sel is 00 carry takes care of both cases
                            a <= in1;
                        b <= in2;
                        carryIn <=cin; ---Carry = 0 (Add A+B)||Carry = 1 (A + comp(B) + 1)
                        when "01" =>
                            if(cin = '0') then --comp(A) + B
                                a <= not(in1);---Compliment in1
                                b <= in2;
                                carryIn <=cin;
                            else
                                a <= not(in1); ---comp(A) + comp(-B) + 1
                                b <= not(in2); ---comp(B)
                                carryIn <=cin; ------Add A-B
                            end if;
                        when "10" =>
                            if(cin = '0') then
                                a <= in1; ---Sets A to as is
                                b <= '1'; ---Sets B to 1
                                carryIn <='1';---Add A-B
                            else
                                a <= in1;---Send A as it is
                                b <= '1'; ---sets B to 1
                                carryIn <='0'; ---Add A+B
                            end if;
                        when "11" =>
                            if(cin = '0') then
                                a <= not(in1); ---Finds compliment of A
                                b <= '0'; ---Set B = 0
                                carryIn <='0'; ---Add A+B
                            else
                                a <= not(in1); ---Finds compliment of A
                                b <= '1'; ---Set B to 1
                                carryIn <='0'; ---Add A+B
                            end if;
                         when others => ---Testing/Debugging
                             a <= '0';
                             b <= '0';
                             carryIn <= '0';
                end case;
    end process;
    BitALU: bitAdder
```

```
   port map(a,b,carryIn,Alu_out,cout);

end Structural;
```

2.10 Full Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
---Setting inputs and outputs
entity fullAdder is
    Port ( CarryIn : in STD_LOGIC;
           P : in STD_LOGIC;
           Q : in STD_LOGIC;
           SUM : out STD_LOGIC;
           CarryOut : out STD_LOGIC);
end fullAdder;

architecture Structural of fullAdder is
---Using the half adder to create a full adder
Component halfAdder
    Port (  Ain : in STD_LOGIC;
            Bin : in STD_LOGIC;
            AdderOut : out STD_LOGIC;
            carry : out STD_LOGIC);
end component;
---Setting signals to from carryout of half adder to an or gate
signal CarryToOr1: STD_LOGIC;
signal CarryToOr2: STD_LOGIC;
signal SumToQ: STD_LOGIC;
---Connecting signals and inputs to 2 half adders
begin
    HA1: halfAdder
    port map(CarryIn, SumToQ, SUM, CarryToOr1);

    HA2: halfAdder
    port map(P,Q,SumToQ,CarryToOr2);

    CarryOut <= CarryToOr1 or CarryToOr2;

end Structural;
```

2.11 Half Adder

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

entity halfAdder is
    --- Input to half adder
    Port (  Ain : in STD_LOGIC;
            Bin : in STD_LOGIC;
            AdderOut : out STD_LOGIC;
            carry : out STD_LOGIC);
end halfAdder;

architecture Behavioral of halfAdder is
---Design of half Adder
begin
    AdderOut <= Ain xor Bin; ---Input A is XOed with Input B.
    carry <= Ain and Bin;    ---Input A is anded with Input B.

end Behavioral;
```

2.12 XDC

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clock }];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock}];


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }];
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }];
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }];
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }];
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }];
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }];
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }];
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }];
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }];
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }];
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }];
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }];
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }];
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }];
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }];

## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }];
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }];
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }];
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { LED[4] }];
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }];
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { LED[6] }];
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { LED[7] }];
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { LED[8] }];
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { LED[9] }];
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { LED[10] }];
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { LED[11] }];
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { LED[12] }];
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { LED[13] }];
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { LED[14] }];
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }];
##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { sevenOut[6] }];
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { sevenOut[5] }];
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { sevenOut[4]}];
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { sevenOut[3] }];
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { sevenOut[2] }];
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { sevenOut[1] }];
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { sevenOut[0] }];
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { Anodes[7] }];
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { Anodes[6] }];
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { Anodes[5] }];
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { Anodes[4] }];
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { Anodes[3] }];
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { Anodes[2] }];
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { Anodes[1] }];
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { Anodes[0] }];
##Buttons
#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }];
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }];
```

```
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTNU }];
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { BTNL }];
#set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { BTNR }];
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { BTND }];
```

3 Hardware Overview

3.1 Hardware Implementation

Multiplier hardware consists of a control unit designed using sequential decoders and registers to keep track of what stage the algorithm is currently in. The structural ALU designed in Lab 2 was used in the design. The structural ALU consists of a full adder cascaded to form a 4 bit adder. The hardware schematic is described in section 3.2. Figure 1 describes the hardware schematic of the top level program. The top level consists of the multiplier unit and the seven segment. Figure 2 describes the hardware schematic of the seven segment module. Figure 3 describes the overview of the multiplier unit. The multiplier unit consists of the control unit, registers and the ALU.

3.2 Hardware Schematic

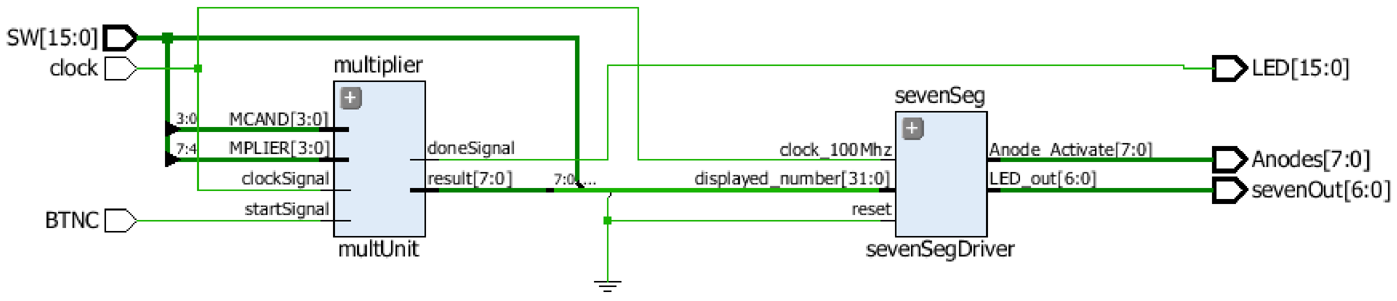


Figure 6: Schematic of the top level module

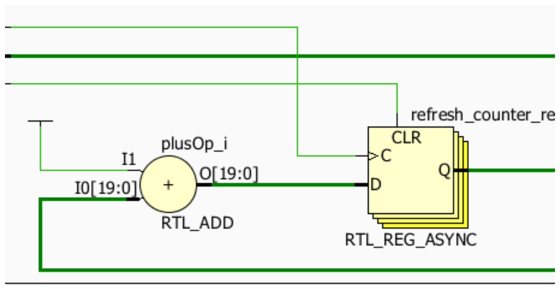


Figure 7: Schematic of the seven segment driver-1

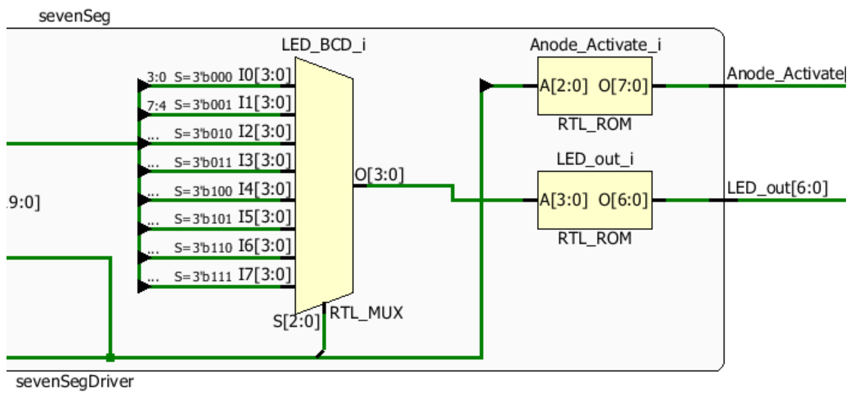


Figure 8: Schematic of the seven segment driver-2

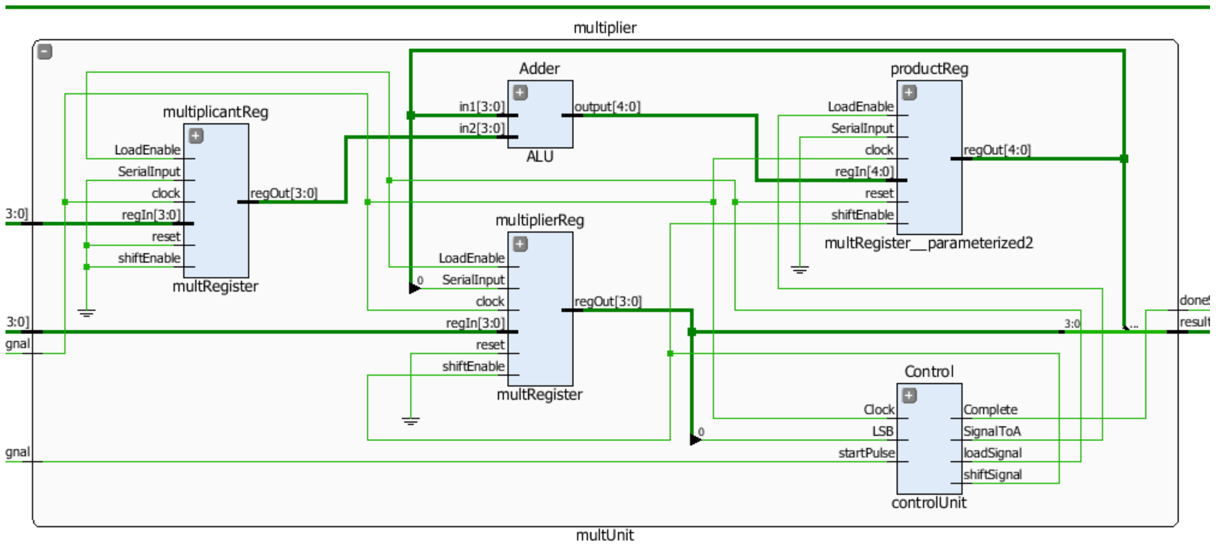


Figure 9: Multiplier Hardware Overview

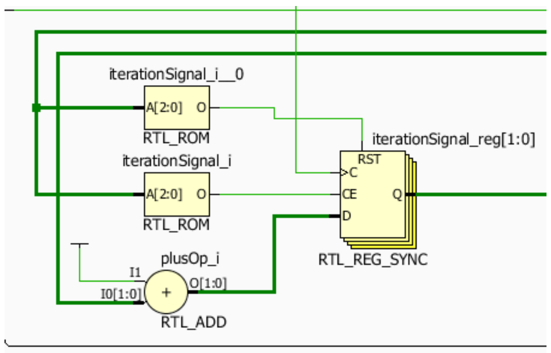


Figure 10: Control Unit Implementation-1

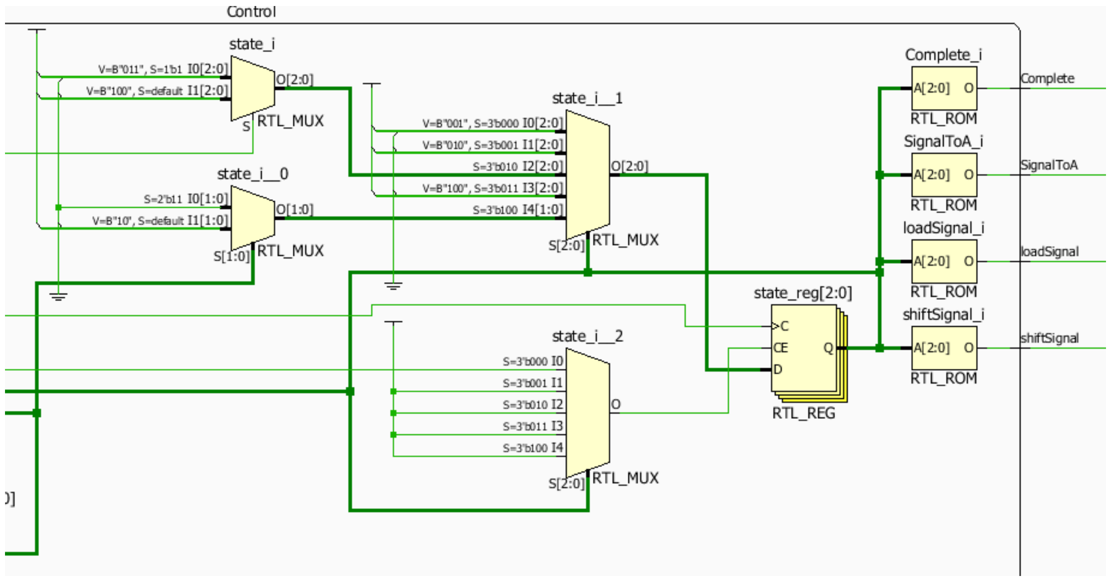


Figure 11: Control Unit Implementation-2

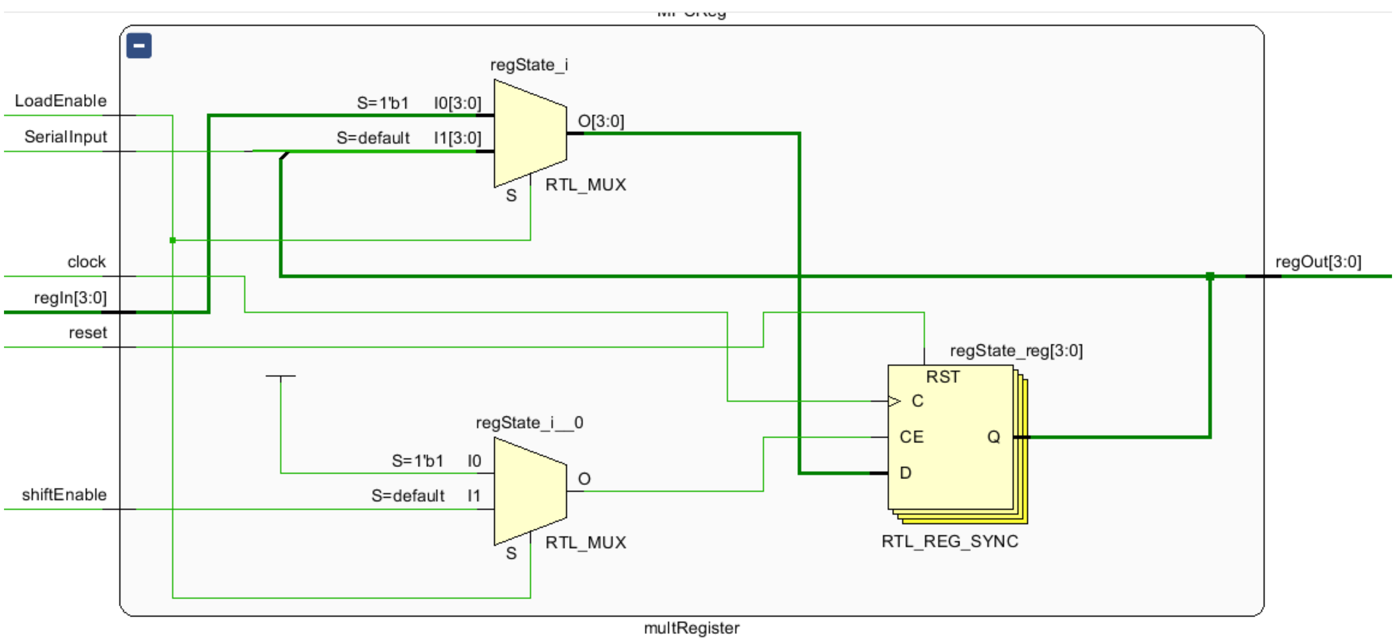
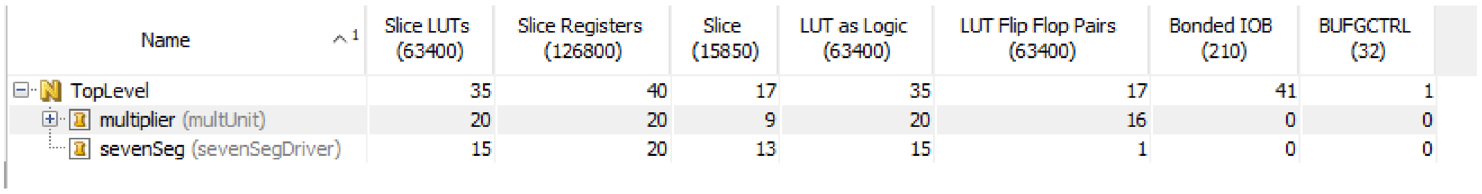


Figure 12: Registers Implementation

3.3 Resources Used



Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT Flip Flop Pairs (63400)	Bonded IOB (210)	BUFGCTRL (32)
TopLevel	35	40	17	35	17	41	1
multiplier (multUnit)	20	20	9	20	16	0	0
sevenSeg (sevenSegDriver)	15	20	13	15	1	0	0

Figure 13: Resourses used summary

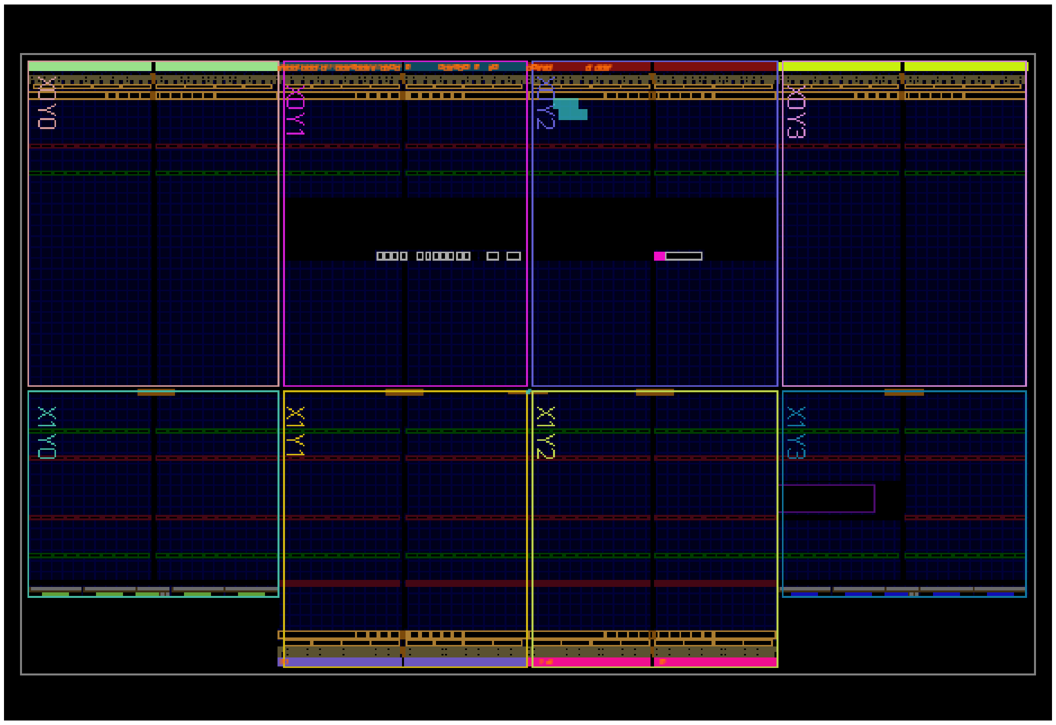


Figure 14: Resources Used from FPGA Board

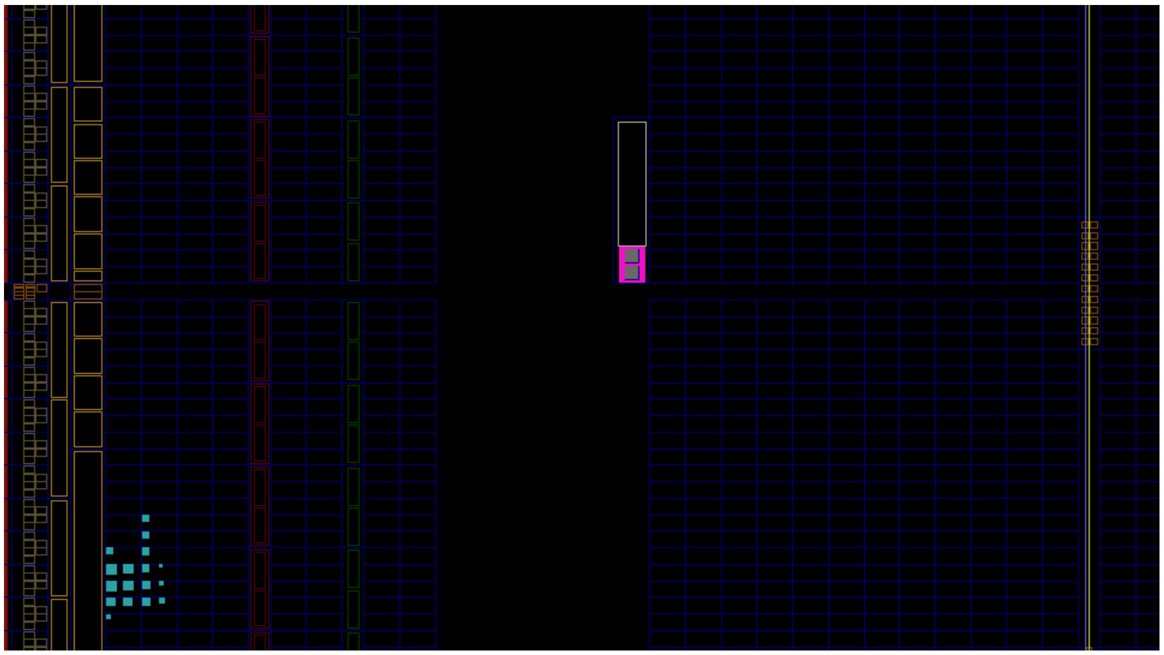


Figure 15: Hardware Usage(Detailed View)

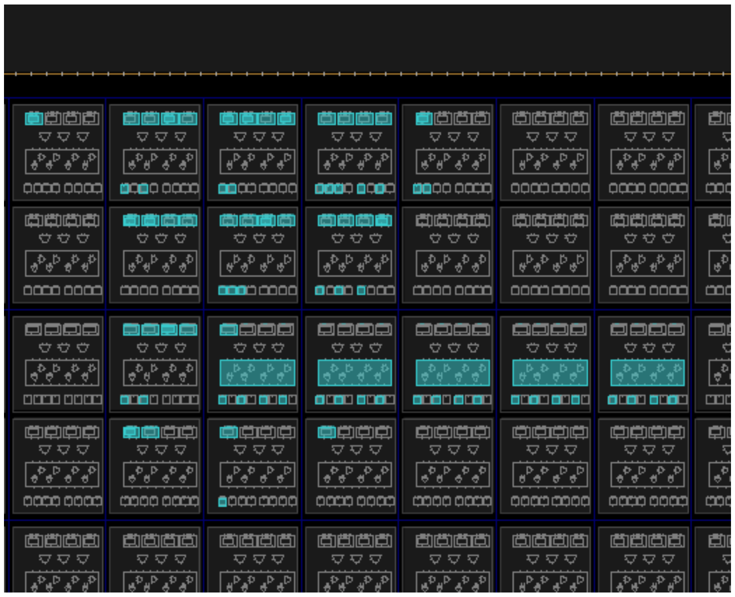


Figure 16: Hardware Usage(Zoomed In View)

4 Project Development Process

4.1 Error Analysis

When designing the system the first time, a lot of errors where encountered as no clock signal was inputted. The design process had to be restarted. A project development plan was created (Highlighted in section 4). After gaining a deep understanding of the system, the system was easier to re-design. No error in functionality was encountered. Design was successful after second design attempt. By reviewing the simulation wave forms after the design, it was safe to assume that the multiplier produces the expected results. There were no errors in the design or output of the multiplier.

4.2 Simulation Analysis

4.2.1 Results Analysis

The results obtained from our design match the expected output exactly, including the shifting of the register from 4 to 2 to 1 to 61 etc. The program uses very little resources as it is extremely small and simple in comparison to the board that is being used. Figure 13 summarizes the resources used.

The simulations below show the functionality of the multipliers in both structural and behavioural designs. Figure 19 displays outputs from both designs for comparison. The speed of the behavioural multiplier is 195ns and the speed of the structural multiplier is 200 ns. The behavioural is faster due to it making use of background processes that were not utilized in the structural implementation. The structural implementation has a doneSignal that is triggered when the multiplication is complete. The value that goes into the shift register starts at 4 and is shifted to 2, 1, then 61. 61 is displayed due to the combination of the highest bits of the product with the lowest bits register Q.

4.2.2 Simulation Figures

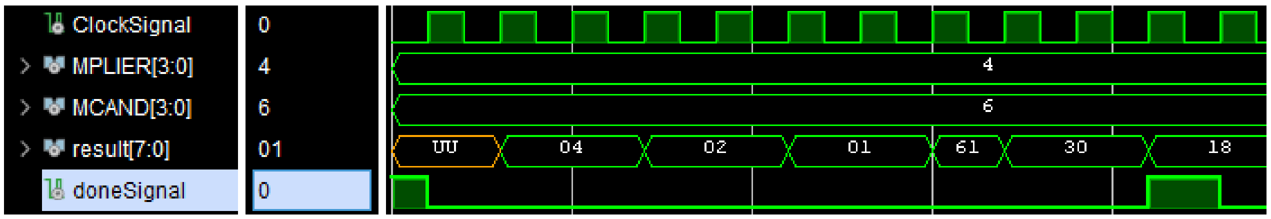


Figure 17: Structural Implementation Results

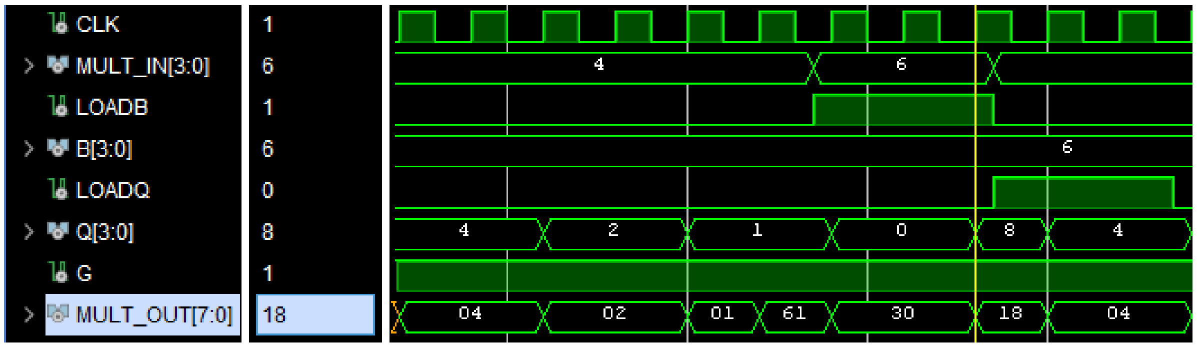


Figure 18: Behavioral Implementation Results

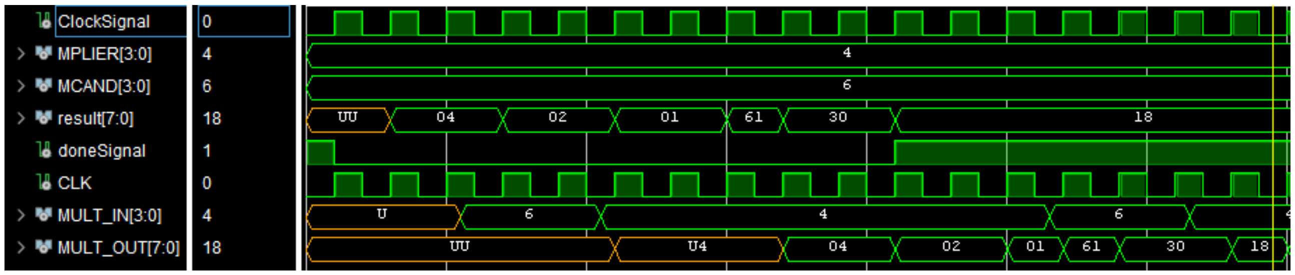


Figure 19: Structural vs Behavioral Comparison

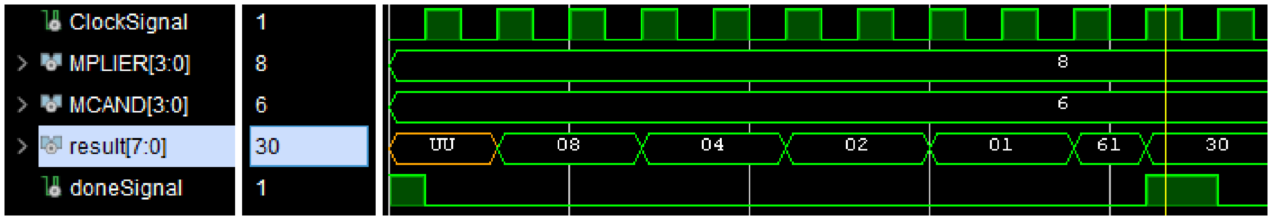


Figure 20: Structural Implementation Test 1

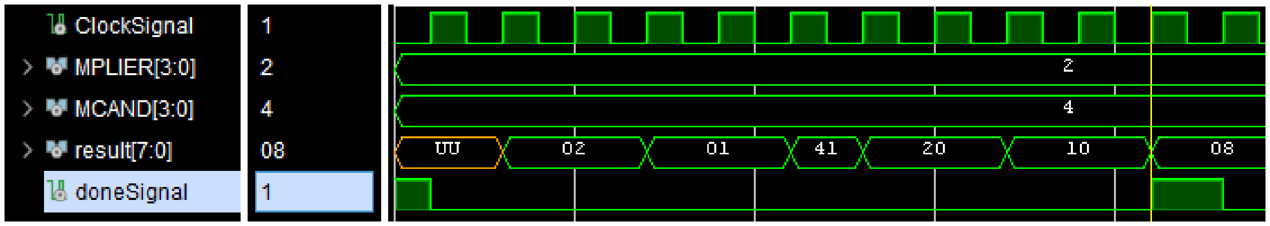


Figure 21: Structural Implementation Test 2