

LAB 7 ACCUMULATOR MULTIPLIER DOT PRODUCT

CSC 343 FALL 2017

NOVEMBER, 22 2017

Jeter Gutierrez

TABLE OF CONTENTS**1 OBJECTIVE pp 4****2 16 BIT ACCUMULATOR pp 4-7****2.1 FUNCTIONALITY AND SPECIFICATIONS FOR 16 BIT ACCUMULATOR pp 4-5****2.2 SIMULATION FOR 16 BIT ACCUMULATOR pp 5-6****2.3 TESTBENCH FOR 16 BIT ACCUMULATOR pp 6-7****3 4 BIT MULTIPLIER pp 7-10****3.1 FUNCTIONALITY AND SPECIFICATIONS FOR 4 BIT MULTIPLIER pp 7-8****3.2 SIMULATION FOR 4 BIT MULTIPLIER pp 8-9****3.3 TESTBENCH FOR 4 BIT MULTIPLIER pp 9-10****4 8 BIT MULTIPLIER USING N-BIT ADDER pp 10-12****4.1 FUNCTIONALITY AND SPECIFICATIONS FOR 8 BIT MULTIPLIER USING
N-BIT ADDER pp 10-11****4.2 SIMULATION FOR 8 BIT MULTIPLIER USING N-BIT ADDER pp 11****4.3 TESTBENCH FOR 8 BIT MULTIPLIER USING N-BIT ADDER pp 11-12****5 8 BIT MULTIPLIER USING TREE STRUCTURE pp 12-16****5.1 FUNCTIONALITY AND SPECIFICATIONS FOR 8 BIT MULTIPLIER USING
TREE STRUCTURE pp 12-15****5.2 TESTBENCH FOR 8 BIT MULTIPLIER USING TREE STRUCTURE pp 15-16****6 DOT PRODUCT pp 16-19****6.1 FUNCTIONALITY AND SPECIFICATIONS FOR DOT PRODUCT pp 16-17****6.2 SIMULATION FOR DOT PRODUCT pp 17**

6.3 TEST BENCH FOR DOT PRODUCT. Pp 17-19

7 DEMONSTRATION OF DOT PRODUCT ON DE2-115 BOARD. Pp 19-21

8 CONCLUSION pp 21

9 APPENDIX pp 22

1. OBJECTIVE: In this lab we will be designing a dot product function that uses integrated memory to store vector values. We will be using the DE2-115 board to test our design and code. We will also be writing testbench code to test the functionality of our design. In order to design the final product of a component that can perform dot product while also using SSRAM on an FPGA board we will have to design other components before hand that will help make the design of the overall component more efficient. We will be testing different methods for the implementation of some of the components to show that some implementations are more efficient than other and will be more beneficial to us in the long run. After we are done designing, simulation and testing the components necessary for this project we will test only the dot product on the FPGA programmable DE2-115.

The components that we will be designing in this lab are:

1. 16 BIT ACCUMULATOR
2. 4 BIT MULTIPLIER
3. 8 BIT MULTIPLIER USING N-BIT ADDER
4. 8 BIT MULTIPLIER USING TREE STRUCTURE
5. DOT PRODUCT

2 16 BIT ACCUMULATOR

2.1 FUNCTIONALITY AND SPECIFICATIONS FOR 16 BIT ACCUMULATOR

The purpose of this circuit is to add 16-bit numbers to themselves recursively. We want to be able to store a value and add to the result of a sum which is why we will be using an accumulator that will keep track of the previous result and add a new value to it instead of just having to add two different independent values. There will only be one input value, and a clock signal, the

same value will be added to itself and then the output will be sent to a hexadecimal display in the final product when being tested on the board. There is also a reset value that resets the accumulator value to 0 when it is a true value of 1.

```

library IEEE;
use IEEE.std_logic_1164.all;
entity GUTI_16bit_Accumulator is
    port(a: in std_logic_vector(15 downto 0);
         clk,reset: in std_logic;
         sum: out std_logic_vector(15 downto 0);
         hex1, hex2, hex3, hex4: out std_logic_vector(6 downto 0);
         overflow: out std_logic);
end GUTI_16bit_Accumulator;
architecture arch of GUTI_16bit_Accumulator is
    component Subtractor_16 IS
        PORT
        (
            add_sub      : IN STD_LOGIC ;
            dataa         : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            datab         : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            overflow      : OUT STD_LOGIC ;
            result        : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
    end component;
    component DFF
        port(d, clk, clrn: in std_logic;
             q: out std_logic);
    end component;
    component GUTI_hex
        port( bi : in std_logic_vector(3 downto 0) := X"0";
             seg : out std_logic_vector(6 downto 0) := "000000");
    end component;
    signal over: std_logic;
    signal input: std_logic_vector(15 downto 0);
    signal b, output: std_logic_vector(15 downto 0);
    begin
        DFF8: for i in 0 to 15 generate
            DFF_in: DFF port map (a(i), clk, reset, input(i));
            DFF_out: DFF port map (output(i), clk, reset, b(i));
        end generate;
        Adder: Subtractor_16 port map ('1', b, input, over, output);
        Overf: DFF port map (over, clk, '1', overflow);
        result: sum <= b;
        Display1: GUTI_hex port map (b(3 downto 0),hex1);
        Display2: GUTI_hex port map (b(7 downto 4),hex2);
        Display3: GUTI_hex port map (b(11 downto 8),hex3);
        Display4: GUTI_hex port map (b(15 downto 12),hex4);
    end arch;

```

Figure 1: VHDL code for 16 bit accumulator.

2.2 SIMULATION FOR 16 BIT ACCUMULATOR

In this simulation we will be giving different values to a in order to add it to the accumulator value, the accumulator will have the total value after each clock signal of the previously stored value and the new value. We will consider the functionality of the design.

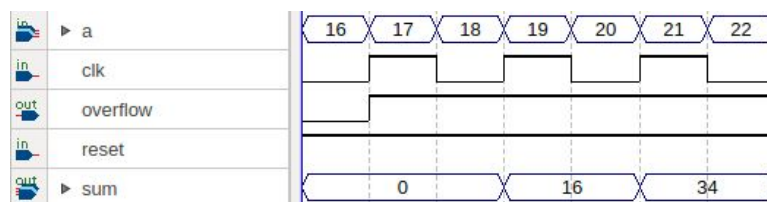


Figure 2: Vector waveform simulation for 16 bit accumulator. As we can see in the sum we can see the result of adding 16 and 18. That is because it stores what is on the rising edge of the clock there is a delay and that is expected because we are using d flip flops for memory. Our simulation did not give us any errors which means we were able to successfully design a 16 bit accumulator in VHDL code that works in simulation.

2.3 TESTBENCH FOR 16 BIT ACCUMULATOR

In this section we will be testing in more detail the correctness of the 16 bit accumulator. We will be using modelsim to simulate the design using specific values to make sure that it is working correctly in a more advanced manner.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
entity GUTI_ACCUMULATOR_TEST is
end entity;
architecture arch_test of GUTI_ACCUMULATOR_TEST is
  COMPONENT accumulator is
  port(a : in std_logic_vector(15 downto 0));
  clk, reset : in std_logic;
  sum : out std_logic_vector(15 downto 0);
  hex1, hex2, hex3, hex4: out std_logic_vector(6 downto 0);
  overflow: out std_logic;
  END COMPONENT;
  signal X,S :STD_LOGIC_VECTOR(15 DOWNT0 0);
  SIGNAL H1,H2,H3,H4: STD_LOGIC_VECTOR(6 DOWNT0 0);
  SIGNAL CLK1,RESET1,OVERFLOW1: STD_LOGIC;
begin
  uut: ACCUMULATOR
  port map (A => X,CLK => CLK1,RESET =>RESET1,SUM =>S,HEX1 =>H1,HEX2 =>H2,HEX3 =>H3,HEX4 =>H4,OVERFLOW=>OVERFLOW1);
  tb : process
  begin
    wait for 100 ns;
    report "Testing Accumulator";
    X<="0000000000000001";
    RESET1<='1';
    WAIT FOR 30 NS;
    RESET1<='0';
    for I in 0 to 256 loop
      assert (S = S+X-"0000000000000001") report "The total accumulator sum is " & integer'image(to_integer(unsigned((S)))) &
        " while the expected value is|" & integer'image(to_integer(unsigned((S+X-"0000000000000001")))) severity ERROR;
      wait for 100 ns;
      CLK1<='0';
      WAIT FOR 30 NS;
      CLK1<='1';
    end loop;
    report "Test completed";
    wait;
  end process;
end arch_test;

```

Figure 3: VHDL code for testbench of 16 bit accumulator.

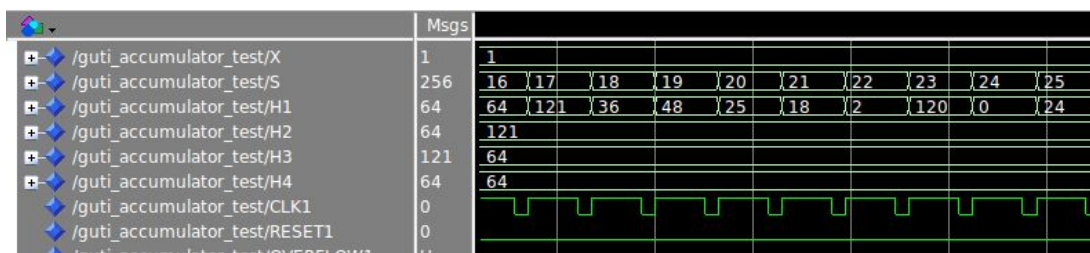


Figure 4: Testbench output for 16 bit accumulator Test Bench code. As we can see in the output for the 16 bit accumulator testbench we did not get any errors again, that means we were able to successfully design a 16 bit accumulator and the values are all correct the values for the total sum to the accumulator are calculated without errors.

```
VSIM 5> run -all
# ** Note: Testing Accumulator
#   Time: 100 ns  Iteration: 0  Instance: /guti_accumulator_test
# ** Note: Test completed
#   Time: 33540 ns  Iteration: 0  Instance: /guti_accumulator_test
```

Figure 5: Result of 16 bit accumulator Test bench. As we can see it took 33540 nanoseconds to perform each of the total iterations using the 16 bit accumulator and we did not get any errors.

3 4 BIT MULTIPLIER

3.1 FUNCTIONALITY AND SPECIFICATIONS FOR 4 BIT MULTIPLIER

The purpose of this circuit is to multiply 4 bit numbers. We need to do that because in calculators and for other computer programs that are necessary and important there are more operations besides just addition or subtraction that are necessary. That is why we will be multiplying numbers together and sending the product to an 8-bit number that is P. the inputs that will be multiplied are a and b but both of those numbers are 4-bit numbers. The result of their product will be 8 bits. If we were to be multiplying 5 bit numbers their result would be 10 bits maximum value.

```

library ieee;--JETER GUTIERREZ NOVEMBER 22, 2017
use ieee.std_logic_1164.all;--JETER GUTIERREZ NOVEMBER 22, 2017
entity GUTI_multi_4bit is--JETER GUTIERREZ NOVEMBER 22, 2017
    port(a, b : in std_logic_vector (3 downto 0);--JETER GUTIERREZ NOVEMB
        hex1, hex2, hex3, hex4: out std_logic_vector(6 downto 0);--JETER
        p: out std_logic_vector (7 downto 0)); --JETER GUTIERREZ NOVEMB
end GUTI_multi_4bit; --JETER GUTIERREZ NOVEMBER 22, 2017
architecture arch of GUTI_multi_4bit is--JETER GUTIERREZ NOVEMBER 22, 20:
    component LPM_4bitAdd is --JETER GUTIERREZ NOVEMBER 22, 2017
        PORT( cin
            : IN STD_LOGIC ;--JETER GUTIERREZ NOVEMBER 22, 2017
            dataa
            : IN STD_LOGIC_VECTOR (3 DOWNT0 0);--JETER GUTIERREZ N
            datab
            : IN STD_LOGIC_VECTOR (3 DOWNT0 0);--JETER GUTIERREZ N
            cout
            : OUT STD_LOGIC ;--JETER GUTIERREZ NOVEMBER 22, 2017
            result
            : OUT STD_LOGIC_VECTOR (3 DOWNT0 0));--JETER GUTIER
    end component;--JETER GUTIERREZ NOVEMBER 22, 2017
    component GUTI_hex--JETER GUTIERREZ NOVEMBER 22, 2017
    port( bi : in std_logic_vector(3 downto 0) := X"0";--JETER GUTIERREZ
        seg : out std_logic_vector(6 downto 0) := "0000000");--JETER GU
    end component;--JETER GUTIERREZ NOVEMBER 22, 2017
    signal c1, c2: std_logic;--JETER GUTIERREZ NOVEMBER 22, 2017
    signal product: std_logic_vector(7 downto 0);--JETER GUTIERREZ NOVEMB
    signal s1,s2,s3, a1,b1, a2,b2, a3, b3 :std_logic_vector(3 downto 0);-
begin--JETER GUTIERREZ NOVEMBER 22, 2017
    product(0)<= a(0) and b(0);--JETER GUTIERREZ NOVEMBER 22, 2017
    product(1)<=s1(0);--JETER GUTIERREZ NOVEMBER 22, 2017
    product(2)<=s2(0);--JETER GUTIERREZ NOVEMBER 22, 2017 |
    product(6 downto 3)<=s3;--JETER GUTIERREZ NOVEMBER 22, 2017
    a1(3)<='0';--JETER GUTIERREZ NOVEMBER 22, 2017
    a2(3)<=c1;--JETER GUTIERREZ NOVEMBER 22, 2017
    a3(3)<=c2;--JETER GUTIERREZ NOVEMBER 22, 2017
    p<=product;--JETER GUTIERREZ NOVEMBER 22, 2017
    Adder4bit1: for i in 0 to 2 generate--JETER GUTIERREZ NOVEMBER 2:
        a1(i) <= b(0) and a(i+1);--JETER GUTIERREZ NOVEMBER 22, 2017
        a2(i) <= s1(i+1);--JETER GUTIERREZ NOVEMBER 22, 2017
        a3(i) <= s2(i+1);--JETER GUTIERREZ NOVEMBER 22, 2017
    end generate;--JETER GUTIERREZ NOVEMBER 22, 2017
    Adder4bit2: for i in 0 to 3 generate
        b1(i) <= b(1) and a(i);--JETER GUTIERREZ NOVEMBER 22, 2017
        b2(i) <= b(2) and a(i);--JETER GUTIERREZ NOVEMBER 22, 2017
        b3(i) <= b(3) and a(i);--JETER GUTIERREZ NOVEMBER 22, 2017
    end generate;--JETER GUTIERREZ NOVEMBER 22, 2017
    Sum1 : LPM_4bitAdd port map('0',a1, b1, c1, s1);--JETER GUTIERREZ
    Sum2 : LPM_4bitAdd port map('0',a2, b2, c2, s2);--JETER GUTIERREZ
    Sum3 : LPM_4bitAdd port map('0',a3, b3, product(7), s3);--JETER GU
    Display1: GUTI_hex port map (a(3 downto 0),hex1);--JETER GUTIERREZ
    Display2: GUTI_hex port map (b(3 downto 0),hex2);--JETER GUTIERREZ
    Display3: GUTI_hex port map (product(3 downto 0),hex3);--JETER GUT
    Display4: GUTI_hex port map (product(7 downto 4),hex4);--JETER GUT
end arch;--JETER GUTIERREZ NOVEMBER 22, 2017

```

Figure 6: VHDL code for 4 bit multiplier.

3.2 SIMULATION FOR 4 BIT MULTIPLIER

In this simulation we will be giving the words a and b different values and comparing their product in unsigned decimal format, we are doing this to test the correctness of our design.

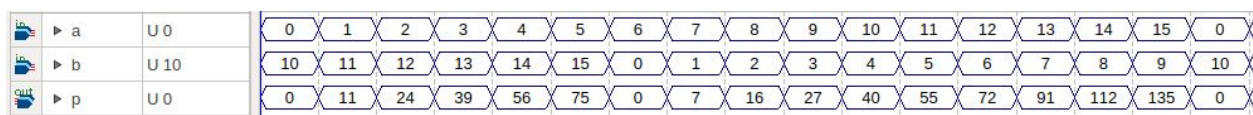


Figure 7: Vector waveform simulation for 4 bit multiplier.

As we can see the products in every case are correct, we do not have any errors this means we were able to correctly design a 4 bit multiplier.

3.3 TESTBENCH FOR 4 BIT MULTIPLIER

The purpose of this section is to test the correctness of the 4 bit multiplier in more specific detail.

We will be using modelsim to simulate the design using predefined values in order to make sure that it is working correctly in a more advanced manner.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
entity GUTI_MULTIPLI_TEST4 is
end entity;
architecture arch_test of GUTI_MULTIPLI_TEST4 is
component GUTI_multi_4bit is--JETER GUTIERREZ NOVEMBER 22, 2017
port(a, b : in std_logic_vector (3 downto 0);--JETER GUTIERREZ NOVEMBER 22, 2017
      hex1, hex2, hex3, hex4: out std_logic_vector (6 downto 0);--JETER GUTIERREZ NOVEMBER 22, 2017
      p: out std_logic_vector (7 downto 0)); --JETER GUTIERREZ NOVEMBER 22, 2017
end component; --JETER GUTIERREZ NOVEMBER 22, 2017
signal X,Y:STD_LOGIC_VECTOR (3 DOWNT0 0);
signal P1 :STD_LOGIC_VECTOR (7 DOWNT0 0);
SIGNAL H1,H2,H3,H4: STD_LOGIC_VECTOR (6 DOWNT0 0);
begin
uut: GUTI_Multi_4bit
port map (A => X,B => Y, hex1 =>H1,hex2 =>H2,hex3 =>H3,hex4 =>H4,P => P1);
tb : process
begin
wait for 100 ns;
report "Hello Simulator";
X<="0000";
Y<="0000";
for I in 0 to 256 loop
for J in 0 to 256 loop
wait for 100 ns;
assert (P1 = X*Y) report "The Product from 4 IS " & integer'image(to_integer(unsigned((P1)))) &
" EXPECTED " & integer'image(to_integer(unsigned((X*Y)))) severity ERROR;
Y<=Y+"0001";
end loop;
X<=X+"0001";
end loop;
report "Test completed";
wait;
end process;
end arch_test;
```

Figure 8: VHDL code for 4 bit multiplier test bench.

+ /guti_multipli_test4/X	6	6																		
+ /guti_multipli_test4/Y	14	5	6	7	8	9	10	11	12	13	14	15	0	1	2					
+ /guti_multipli_test4/P1	84	30	36	42	48	54	60	66	72	78	84	90	0	6	12					

Figure 9: Testbench simulation output for 4 bit multiplier. As we can see the output for the 4 bit multiplier in the test bench we did not get any errors that means we were able to successfully design the 4 bit multiplier. We see only correct values at each instance for the product. The products were calculated without errors.

```
# ** Note: Test completed
# Time: 6605 us Iteration: 0 Instance: /guti_multipli_test4
```

Figure 10: Test bench result for 4 bit multiplier. As we can see it took 6605 us to perform all of the total iterations for the 4 bit multiplier and we did not get any errors.

4 8 BIT MULTIPLIER USING N-BIT ADDER

4.1 FUNCTIONALITY AND SPECIFICATIONS FOR 8 BIT MULTIPLIER USING N-BIT ADDER

The purpose of this circuit is to multiply 8 bit numbers. We need to do that because in calculators and for other computer programs that are necessary and important there are more operations besides just addition or subtraction that are necessary. That is why we will be multiplying numbers together and sending the product to an 16-bit number that is P. the inputs that will be multiplied are a and b but both of those numbers are 4-bit numbers. The result of their product will be 16 bits. If we were to be multiplying 5 bit numbers their result would be 10 bits maximum value before risking an overflow.

```

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

entity GUTI_8BIT_MULTIPLICATION is
port ( ONE,TWO : in std_logic_vector (7 downto 0);
      RESULTS : out STD_LOGIC_VECTOR (15 DOWNT0 0);
      HEX11, HEX22, HEX33, HEX44, HEX55, HEX66, HEX77, HEX88 : out std_logic_vector (6 downto 0));
end GUTI_8BIT_MULTIPLICATION ;
ARCHITECTURE MULTIPLYING OF GUTI_8BIT_MULTIPLICATION IS
COMPONENT GUTI_8BITMULTIPLIER IS
port ( INPUT_X,INPUT_Y : in std_logic_vector (7 downto 0);
      PRODUCT : out std_logic_vector (15 downto 0));
end COMPONENT;
component dec_to_hex is port (
  hex_digit: in std_logic_vector (3 downto 0);
  seg_a: out std_logic;
  seg_b: out std_logic;
  seg_c: out std_logic;
  seg_d: out std_logic;
  seg_e: out std_logic;
  seg_f: out std_logic;
  seg_g: out std_logic);
end component;
SIGNAL RESULT: STD_LOGIC_VECTOR (15 DOWNT0 0);

BEGIN
MULTIPLICATION: GUTI_8BITMULTIPLIER PORT MAP(ONE,TWO,RESULT);
RESULTS<=RESULT;

Hx1: dec_to_hex port map(ONE(3 downto 0), HEX11(0), HEX11(1), HEX11(2), HEX11(3),HEX11(4), HEX11(5), HEX11(6));
Hx2: dec_to_hex port map (ONE(7 downto 4), HEX22(0), HEX22(1), HEX22(2), HEX22(3),HEX22(4), HEX22(5), HEX22(6));
Hx3: dec_to_hex port map (TWO(3 downto 0), HEX33(0), HEX33(1), HEX33(2), HEX33(3),HEX33(4), HEX33(5), HEX33(6));
Hx4: dec_to_hex port map (TWO(7 downto 4), HEX44(0), HEX44(1), HEX44(2), HEX44(3),HEX44(4), HEX44(5), HEX44(6));
Hx5: dec_to_hex port map (RESULT(3 downto 0), HEX55(0), HEX55(1), HEX55(2), HEX55(3),HEX55(4), HEX55(5), HEX55(6));
Hx6: dec_to_hex port map (RESULT(7 downto 4), HEX66(0), HEX66(1), HEX66(2), HEX66(3),HEX66(4), HEX66(5), HEX66(6));
Hx7: dec_to_hex port map (RESULT(11 downto 8), HEX77(0), HEX77(1), HEX77(2), HEX77(3),HEX77(4), HEX77(5), HEX77(6));
Hx8: dec_to_hex port map (RESULT(15 downto 12), HEX88(0), HEX88(1), HEX88(2), HEX88(3),HEX88(4), HEX88(5), HEX88(6));
END MULTIPLYING;

```

Figure 11: VHDL code for 8 bit multiplier using an 8 bit adder.

4.2 SIMULATION FOR 8 BIT MULTIPLIER USING N-BIT ADDER

In this simulation we will be giving words one and two different values in order to find the product of the two 8 bit numbers into a 16 bit number using 8 bit adders and linear shift registers. Then we will observe the design in order to determine its correctness depending on if we receive errors and what kind of errors they are.

▶ ONE	129	177	206	69	183
▶ TWO	173	38	47	39	53
▶ RESULTS	22317	6726	9682	2691	9699

Figure 12: Vector waveform simulation for 8 bit multiplier using 8 bit adder.

As we can see the products in every case is correct, we do not have any errors this means we were able to successfully design a 8 bit multiplier using 8 bit adders.

4.3 TESTBENCH FOR 8 BIT MULTIPLIER USING N-BIT ADDER

The purpose of this section is to test the correctness of the 8 bit multiplier using n bit adders in more detail. We will be using modelsim to simulate our test bench code using predetermined values in order to make sure that our design is working correctly in a more advanced professional testing environment.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
entity GUTI_MULTIPLI_TEST is
end entity;
architecture arch_test of GUTI_MULTIPLI_TEST is
component GUTI_8BITMULTIPLIER is
port (INPUT_X, INPUT_Y : in std_logic_vector (7 downto 0);
      PRODUCT: out std_logic_vector (15 downto 0));
end component;
signal X,Y:STD_LOGIC_VECTOR (7 DOWNTO 0);
signal P :STD_LOGIC_VECTOR (15 DOWNTO 0);
begin
uut: GUTI_8BITMULTIPLIER
port map (INPUT_X => X, INPUT_Y => Y, PRODUCT => P);
tb : process
begin
wait for 100 ns;
report "TESTING 8 BIT MULTIPLIER" ;
X<="00000000";
Y<="00000000";
for I in 0 to 256 loop
for J in 0 to 256 loop
wait for 100 ns;
assert (P = X*Y) report "RESULT " & integer'image(to_integer(unsigned((P)))) &
"EXPECTED" & integer'image(to_integer(unsigned((X*Y)))) severity ERROR;
Y<=Y+"00000001";
end loop;
X<=X+"00000001";
end loop;
report "Test completed";
wait;
end process;
end arch_test;

```

Figure 13: VHDL code of test bench for 8 bit multiplier using 8 bit adders as component.

	Msgs	
/guti_multipli_test/X	9	9
/guti_multipli_test/Y	33	31 32 33 34 35 36 37 38 39 40 41
/guti_multipli_test/P	297	279 288 297 306 315 324 333 342 351 360 369

Figure 14: Testbench output simulation for 8 bit multiplier using 8 bit adders as a component. As we can see the output for the 8 bit multiplier in this test bench code is correct we did not get any errors, the values of the products between X and Y are correct at each instance. This means we were calculating products of 8 bit numbers without errors.

```
# ** Note: TESTING 8 BIT MULTIPLIER
# Time: 100 ns Iteration: 0 Instance: /guti_multipli_test
# ** Note: Test completed
# Time: 6605 us Iteration: 0 Instance: /guti_multipli_test
```

Figure 15: Testbench result for 8 bit multiplier using 8 bit adder as a component. As we can see the time it took to complete the operations are 6605 us for the 8 bit multiplier and we did not get any errors.

5 8 BIT MULTIPLIER USING TREE STRUCTURE

5.1 FUNCTIONALITY AND SPECIFICATIONS FOR 8 BIT MULTIPLIER USING TREE STRUCTURE

The purpose of this design is to make another 8 bit multiplier using a tree structure that will add different values in parallel in order to make the multiplication process take less time than if we were just using an 8 bit adder like we did in the previous design.

```

library ieee;
use ieee.std_logic_1164.all;
entity GUTI_TREEMULTI is
    port(a, b : in std_logic_vector(7 downto 0);
          product : out std_logic_vector(15 downto 0));
end GUTI_TREEMULTI;
architecture arch of GUTI_TREEMULTI is
    signal p0, p1, p2, p3, p4, p5, p6, p7 : std_logic_vector(7 downto 0);
    signal s101, s102, s103, s104, s105, s106, s107, s108, s109, s110, s111, s112, s113, s114,
           s202, s203, s204, s205, s206, s207, s208, s209, s210, s211, s212, s213,
           s303, s304, s305, s306, s307, s308, s309, s310, s311, s312,
           s404, s405, s406, s407, s408, s409, s410, s411,
           s505, s506, s507, s508, s509, s510,
           s606, s607, s608, s609,
           s707, s708 : std_logic;
    signal c101, c102, c103, c104, c105, c106, c107, c108, c109, c110, c111, c112, c113, c114,
           c202, c203, c204, c205, c206, c207, c208, c209, c210, c211, c212, c213,
           c303, c304, c305, c306, c307, c308, c309, c310, c311, c312,
           c404, c405, c406, c407, c408, c409, c410, c411,
           c505, c506, c507, c508, c509, c510,
           c606, c607, c608, c609,
           c707, c708 : std_logic;
    component GUTI_FA is --JETER GUTIERREZ MAY 10 2016
        port(a, b : in std_logic;--JETER GUTIERREZ MAY 10 2016
              cin : in std_logic;--JETER GUTIERREZ MAY 10 2016
              cout : out std_logic;--JETER GUTIERREZ MAY 10 2016
              sum : out std_logic;--JETER GUTIERREZ MAY 10 2016
    end component; --JETER GUTIERREZ MAY 10 2016
    component GUTI_HA is
        port(a, b : in std_logic;
              carry, sum : out std_logic);
    end component;
    begin
        product(0) <= p0(0);
        product(1) <= s101;
        product(2) <= s202;
        product(3) <= s303;
        product(4) <= s404;
        product(5) <= s505;
        product(6) <= s606;
        product(7) <= s707;
        product(8) <= s708;
        product(9) <= s609;
        product(10) <= s510;
        product(11) <= s411;
        product(12) <= s312;
        product(13) <= s213;
        product(14) <= c114;
        product(15) <= s114;
        process(a, b)
            begin
                for i in 0 to 7 loop
                    p0(i) <= A(i) and B(0);
                    p1(i) <= A(i) and B(1);
                    p2(i) <= A(i) and B(2);
                    p3(i) <= A(i) and B(3);

```

Figure 16: PART 1 VHDL code for 8 bit multiplier using tree structure.


```

        p4(i) <= A(i) and B(4);
        p5(i) <= A(i) and B(5);
        p6(i) <= A(i) and B(6);
        p7(i) <= A(i) and B(7);
    end loop;
end process;
HA101 : GUTI_HA
    port map (p0(1), p1(0), c101, s101);
FA102 : GUTI_FA
    port map (p0(2), p1(1), c101, c102, s102);
FA103 : GUTI_FA
    port map (p0(3), p1(2), c102, c103, s103);
FA104 : GUTI_FA
    port map (p0(4), p1(3), c103, c104, s104);
FA105 : GUTI_FA
    port map (p0(5), p1(4), c104, c105, s105);
FA106 : GUTI_FA
    port map (p0(6), p1(5), c105, c106, s106);
FA107 : GUTI_FA
    port map (p0(7), p1(6), c106, c107, s107);
FA108 : GUTI_FA
    port map (p1(7), p2(6), c107, c108, s108);
FA109 : GUTI_FA
    port map (p2(7), p3(6), c108, c109, s109);
FA110 : GUTI_FA
    port map (p3(7), p4(6), c109, c110, s110);
FA111 : GUTI_FA
    port map (p4(7), p5(6), c110, c111, s111);
FA112 : GUTI_FA
    port map (p5(7), p6(6), c111, c112, s112);
FA113 : GUTI_FA
    port map (p6(7), p7(6), c112, c113, s113);
FA114 : GUTI_FA
    port map (p7(7), c113, c213, c114, s114);
HA201 : GUTI_HA
    port map (s102, p2(0), c202, s202);
FA202 : GUTI_FA
    port map (s103, p2(1), c202, c203, s203);
FA203 : GUTI_FA
    port map (s104, p2(2), c203, c204, s204);
FA204 : GUTI_FA
    port map (s105, p2(3), c204, c205, s205);
FA205 : GUTI_FA
    port map (s106, p2(4), c205, c206, s206);
FA206 : GUTI_FA
    port map (s107, p2(5), c206, c207, s207);
FA207 : GUTI_FA
    port map (s108, p3(5), c207, c208, s208);
FA208 : GUTI_FA
    port map (s109, p4(5), c208, c209, s209);
FA209 : GUTI_FA
    port map (s110, p5(5), c209, c210, s210);
FA210 : GUTI_FA
    port map (s111, p6(5), c210, c211, s211);
FA211 : GUTI_FA
    port map (s112, p7(5), c211, c212, s212);
FA212 : GUTI_FA
    port map (s113, c212, c312, c213, s213);
HA301 : GUTI_HA
    port map (s203, p3(0), c303, s303);
FA302 : GUTI_FA
    port map (s204, p3(1), c303, c304, s304);
FA303 : GUTI_FA
    port map (s205, p3(2), c304, c305, s305);
FA304 : GUTI_FA
    port map (s206, p3(3), c305, c306, s306);
FA305 : GUTI_FA
    port map (s207, p3(4), c306, c307, s307);
FA306 : GUTI_FA
    port map (s208, p4(4), c307, c308, s308);
FA307 : GUTI_FA
    port map (s209, p5(4), c308, c309, s309);
FA308 : GUTI_FA
    port map (s210, p6(4), c309, c310, s310);
FA309 : GUTI_FA
    port map (s211, p7(4), c310, c311, s311);
FA310 : GUTI_FA
    port map (s212, c311, c411, c312, s312);
HA401 : GUTI_HA
    port map (s304, p4(0), c404, s404);
FA402 : GUTI_FA
    port map (s305, p4(1), c404, c405, s405);
FA403 : GUTI_FA
    port map (s306, p4(2), c405, c406, s406);
FA404 : GUTI_FA
    port map (s307, p4(3), c406, c407, s407);
FA405 : GUTI_FA
    port map (s308, p5(3), c407, c408, s408);
FA406 : GUTI_FA
    port map (s309, p6(3), c408, c409, s409);
FA407 : GUTI_FA
    port map (s310, p7(3), c409, c410, s410);
FA408 : GUTI_FA
    port map (s311, c410, c510, c411, s411);
HA501 : GUTI_HA
    port map (s405, p5(0), c505, s505);
FA502 : GUTI_FA
    port map (s406, p5(1), c505, c506, s506);
FA503 : GUTI_FA
    port map (s407, p5(2), c506, c507, s507);
FA504 : GUTI_FA
    port map (s408, p6(2), c507, c508, s508);
FA505 : GUTI_FA
    port map (s409, p7(2), c508, c509, s509);
FA506 : GUTI_FA
    port map (s410, c509, c609, c510, s510);
HA601 : GUTI_HA
    port map (S506, p6(0), c606, s606);
FA602 : GUTI_FA
    port map (s507, p6(1), c606, c607, s607);
FA603 : GUTI_FA
    port map (s508, p7(1), c607, c608, s608);
FA604 : GUTI_FA
    port map (s509, c608, c708, c609, s609);
HA701 : GUTI_HA
    port map (s607, p7(0), c707, s707);
HA702 : GUTI_HA
    port map (s608, c707, c708, s708);
end arch;

```

Figure 17: PART 2 VHDL code for 8 bit multiplier using tree structure.

5.2 TESTBENCH FOR 8 BIT MULTIPLIER USING TREE STRUCTURE

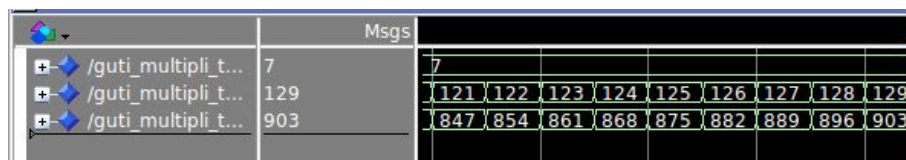
The purpose of this design is to make a test bench code to test the speed and correctness of multiplying 8 bit numbers using a tree design. We will see if we were able to design the product correctly and if it is faster than using an 8 bit adder.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
entity GUTI_MULTIPLI_TEST_TREE is
end entity;
architecture arch_test of GUTI_MULTIPLI_TEST_TREE is
component GUTI_TREEMULTI is
port(A,B : in std_logic_vector (7 downto 0));
PRODUCT: out std_logic_vector (15 downto 0));
end component;
Signal X,Y:STD_LOGIC_VECTOR (7 DOWNTO 0);
signal P :STD_LOGIC_VECTOR (15 DOWNTO 0);
begin
uut: GUTI_TREEMULTI
port map (A => X,B => Y,PRODUCT => P);
tb : process
begin
wait for 100 ns;
report "TESTING 8BIT MULTIPLIER WITH TREE" ;
X<="00000000";
Y<="00000000";
for I in 0 to 16 loop
for J in 0 to 16 loop
wait for 100 ns;
assert (P = X*Y) report "RESULT " & integer'image(to_integer(unsigned((P)))) &
" EXPECTED " & integer'image(to_integer(unsigned((X*Y)))) severity ERROR;
Y<=Y+"00000001";
end loop;
X<=X+"00000001";
end loop;
report "Test completed";
wait;
end process;|
end arch_test;

```

Figure 18: VHDL code for 8 bit multiplier using tree structure test bench.



Msgs	
/guti_multipli_t...	7
/guti_multipli_t...	129
/guti_multipli_t...	903

Figure 19: Testbench output simulation for 8 bit multiplier using tree structure. As we can see the products sent to the product variable are correct this means we were able to correctly design an 8 bit multiplier using a tree design instead of nested 8 bit adders.

```

# ** Note: TESTING 8BIT MULTIPLIER WITH TREE
#   Time: 100 ns Iteration: 0 Instance: /guti_multipli_test_tree
# ** Note: Test completed
#   Time: 29 us Iteration: 0 Instance: /guti_multipli_test_tree

```

Figure 20: Test bench results for testing 8 bit multiplier using tree structure. As we can see it took 29 us instead of 6605 us to complete the products operations for the predetermined cases we were testing, this means that we were able to successfully design an 8 bit multiplier with a tree structure for adding instead of an 8 bit multiplier, it also is significantly quicker and more efficient than the previous design. The products were calculated without errors and we only see correct values for each instance of the product.

6 DOT PRODUCT

6.1 FUNCTIONALITY AND SPECIFICATIONS FOR DOT PRODUCT

The purpose of the dot product is so that when we are making the video for this lab we can store 2 arrays each of 12 elements on the DE2-115 board and, where each element will be 8 bits and then we will store that value onto an accumulator for each position in order to perform the dot product correctly.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity GUTI_DOT_PRODUCT is
port(ONE,TWO : in std_logic_vector (7 downto 0);
      CLOCK,R: IN STD_LOGIC;
      RESULTING:OUT STD_LOGIC_VECTOR (15 DOWNT0 0);
      HEX11,HEX22,HEX33,HEX44 : out std_logic_vector (6 downto 0)
);
end GUTI_DOT_PRODUCT;
ARCHITECTURE ARCH OF GUTI_DOT_PRODUCT IS
COMPONENT accumulator is
  port(a: in std_logic_vector (15 downto 0);
        clk,reset: in std_logic;
        sum: out std_logic_vector (15 downto 0);
        hex1, hex2, hex3, hex4: out std_logic_vector (6 downto 0);
        overflow: out std_logic);
end COMPONENT;
COMPONENT GUTI_8BITMULTIPLIER is
port ( INPUT_X,INPUT_Y : in std_logic_vector (7 downto 0);
        PRODUCT : out std_logic_vector (15 downto 0));
end COMPONENT;

SIGNAL RESULTS: STD_LOGIC_VECTOR (15 DOWNT0 0);
SIGNAL H1,H2,H3,H4:STD_LOGIC_VECTOR (6 DOWNT0 0);
BEGIN
MULTIPLY: GUTI_8BITMULTIPLIER PORT MAP(ONE,TWO,RESULTS);

STORE: accumulator PORT MAP(RESULTS,NOT CLOCK, not R,RESULTING,HEX11,HEX22,HEX33,HEX44);
END ARCH;

```

Figure 21: VHDL code for Dot product.

6.2 SIMULATION FOR DOT PRODUCT

In this section we will be giving different values to the clock in order to have a rising edge and we will set both of the multiplier values to 2 for simplicity to show that the design will add using the product instead of adding just using a single value, this is so that we can show that the accumulator and 8 bit multiplier are working correctly together in order to perform the dot product.

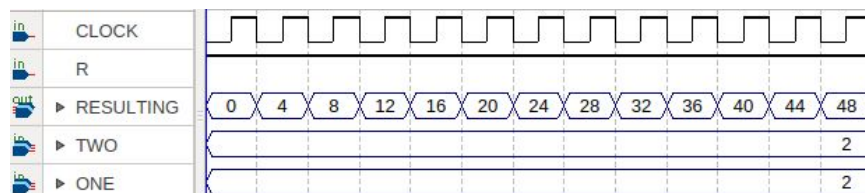


Figure 22: Vector waveform simulation for dot product.

6.3 TEST BENCH FOR DOT PRODUCT.

The purpose of this section is to test the correctness of the Dot product component in detail before we test it on the FPGA board. This way we will know if our design is functioning correctly for the values that we want to test to make sure it will also work on the board.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
entity GUTI_DOT_TEST is
end entity;
architecture arch_test of GUTI_DOT_TEST is
  COMPONENT GUTI_DOT_PRODUCT is
  port(ONE,TWO : in std_logic_vector (7 downto 0);
        CLOCK,R: IN STD_LOGIC;
        RESULTING:OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        HEX11,HEX22,HEX33,HEX44 : out std_logic_vector (6 downto 0));
  end COMPONENT;
  signal X,Y :STD_LOGIC_VECTOR (7 DOWNTO 0);
  SIGNAL H1,H2,H3,H4: STD_LOGIC_VECTOR (6 DOWNTO 0);
  SIGNAL CLK1,RESET1: STD_LOGIC;
  SIGNAL DONE:STD_LOGIC_VECTOR (15 DOWNTO 0);
begin
  uut: GUTI_DOT_PRODUCT
  port map (ONE => X,TWO => Y,CLOCK => CLK1,R =>RESET1,RESULTING =>DONE,HEX11 =>H1,HEX22 =>H2,HEX33 =>H3,HEX44 =>H4);
  tb : process
  begin
    wait for 100 ns;
    report "TESTING DOT PRODUCT↑";
    X<="00000001";
    Y<="00000010";
    RESET1<='0';
    WAIT FOR 30 NS;
    RESET1<='1';
    for I in 0 to 256 loop
      assert (DONE = DONE+X*Y) report "EXPECTED " & integer'image(to_integer(unsigned((DONE)))) &
        " RESULT " & integer'image(to_integer(unsigned((DONE+(X*Y))))) severity ERROR;
      wait for 100 ns;
      CLK1<='0';
      WAIT FOR 30 NS;
      CLK1<='1';
    end loop;
    report "Test completed";
    wait;
  end process;
end arch_test;

```

Figure 23: VHDL code for test bench of Dot product.

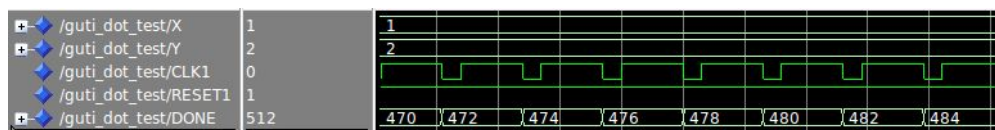


Figure 24: Testbench output simulation for dot product. As we can see the value in the current instance of the test bench code is adding each instance by 2, which is correct and expected because we currently see a value stored in done and the two values being multiplied together are 1 and 2 so we should see an increase of 2 which is what we see and is correct.

```

# ** Note: TESTING DOT PRODUCT
#   Time: 100 ns   Iteration: 0   Instance: /guti_dot_test
# ** Note: Test completed
#   Time: 33540 ns   Iteration: 0   Instance: /guti_dot_test

```

Figure 25: Testbench result for Dot product. As we can see we were able to design the dot product component without any errors it is working correctly in simulation and in our test bench code which means it should work properly in the FPGA DE2-115 board too.

7 DEMONSTRATION OF DOT PRODUCT ON DE2-115 BOARD.

The inputs and outputs assigned to the DE2-115 board are:

INPUT_DATA[0] is assigned to PIN_AB28	INPUT_ADDR[7] is assigned to PIN_AA22	RESULTING[15] is assigned to PIN_G15
INPUT_DATA[1] is assigned to PIN_AC28	R is assigned to PIN_M21	SRAM_ADDR[0] is assigned to PIN_AB7
INPUT_DATA[2] is assigned to PIN_AC27	CLOCK is assigned to PIN_M23	SRAM_ADDR[1] is assigned to PIN_AD7
INPUT_DATA[3] is assigned to PIN_AD27	INPUT_WE is assigned to PIN_Y23	SRAM_ADDR[2] is assigned to PIN_AE7
INPUT_DATA[4] is assigned to PIN_AB27	CLOCK1 is assigned to PIN_N21	SRAM_ADDR[3] is assigned to PIN_AC7
INPUT_DATA[5] is assigned to PIN_AC26	CLOCK2 is assigned to PIN_R24	SRAM_ADDR[4] is assigned to PIN_AB6
INPUT_DATA[6] is assigned to PIN_AD26	STORE is assigned to PIN_Y24	SRAM_ADDR[5] is assigned to PIN_AE6
INPUT_DATA[7] is assigned to PIN_AB26	-----	SRAM_ADDR[6] is assigned to PIN_AB5
INPUT_ADDR[0] is assigned to PIN_AC25	RESULTING[0] is assigned to PIN_G19	SRAM_ADDR[7] is assigned to PIN_AC5
INPUT_ADDR[1] is assigned to PIN_AB25	RESULTING[1] is assigned to PIN_F19	SRAM_ADDR[8] is assigned to PIN_AF5
INPUT_ADDR[2] is assigned to PIN_AC24	RESULTING[2] is assigned to PIN_E19	SRAM_ADDR[9] is assigned to PIN_T7
INPUT_ADDR[3] is assigned to PIN_AB24	RESULTING[3] is assigned to PIN_F21	SRAM_ADDR[10] is assigned to PIN_AF2
INPUT_ADDR[4] is assigned to PIN_AB23	RESULTING[4] is assigned to PIN_F18	SRAM_ADDR[11] is assigned to PIN_AD3
INPUT_ADDR[5] is assigned to PIN_AA24	RESULTING[5] is assigned to PIN_E18	SRAM_ADDR[12] is assigned to PIN_AB4
INPUT_ADDR[6] is assigned to PIN_AA23	RESULTING[6] is assigned to PIN_J19	SRAM_ADDR[13] is assigned to
	RESULTING[7] is assigned to PIN_H19	
	RESULTING[8] is assigned to PIN_J17	
	RESULTING[9] is assigned to PIN_G17	
	RESULTING[10] is assigned to PIN_J15	
	RESULTING[11] is assigned to PIN_H16	
	RESULTING[12] is assigned to PIN_J16	
	RESULTING[13] is assigned to PIN_H17	
	RESULTING[14] is assigned to PIN_F15	

SRAM_ADDR[14] is assigned to	SRAM_DQ[13] is assigned to PIN_AE4	HEX44[6] is assigned to PIN_Y19
PIN_AA4	SRAM_DQ[14] is assigned to PIN_AF3	HEX55[0] is assigned to PIN_AB19
SRAM_ADDR[15] is assigned to	SRAM_DQ[15] is assigned to PIN_AG3	HEX55[1] is assigned to PIN_AA19
PIN_AB11	HEX11[0] is assigned to PIN_G18	HEX55[2] is assigned to PIN_AG21
SRAM_ADDR[16] is assigned to	HEX11[1] is assigned to PIN_F22	HEX55[3] is assigned to PIN_AH21
PIN_AC11	HEX11[2] is assigned to PIN_E17	HEX55[4] is assigned to PIN_AE19
SRAM_ADDR[17] is assigned to	HEX11[3] is assigned to PIN_L26	HEX55[5] is assigned to PIN_AF19
PIN_AB9	HEX11[4] is assigned to PIN_L25	HEX55[6] is assigned to PIN_AE18
SRAM_ADDR[18] is assigned to	HEX11[5] is assigned to PIN_J22	HEX66[0] is assigned to PIN_AD18
PIN_AB8	HEX11[6] is assigned to PIN_H22	HEX66[1] is assigned to PIN_AC18
SRAM_ADDR[19] is assigned to	HEX22[0] is assigned to PIN_M24	HEX66[2] is assigned to PIN_AB18
PIN_T8	HEX22[1] is assigned to PIN_Y22	HEX66[3] is assigned to PIN_AH19
SRAM_CE_N is assigned to PIN_AF8	HEX22[2] is assigned to PIN_W21	HEX66[4] is assigned to PIN_AG19
SRAM_OE_N is assigned to PIN_AD5	HEX22[3] is assigned to PIN_W22	HEX66[5] is assigned to PIN_AF18
SRAM_WE_N is assigned to PIN_AE8	HEX22[4] is assigned to PIN_W25	HEX66[6] is assigned to PIN_AH18
SRAM_UB_N is assigned to PIN_AC4	HEX22[5] is assigned to PIN_U23	HEX77[0] is assigned to PIN_AA17
SRAM_LB_N is assigned to PIN_AD4	HEX22[6] is assigned to PIN_U24	HEX77[1] is assigned to PIN_AB16
SRAM_DQ[0] is assigned to PIN_AH3	HEX33[0] is assigned to PIN_AA25	HEX77[2] is assigned to PIN_AA16
SRAM_DQ[1] is assigned to PIN_AF4	HEX33[1] is assigned to PIN_AA26	HEX77[3] is assigned to PIN_AB17
SRAM_DQ[2] is assigned to PIN_AG4	HEX33[2] is assigned to PIN_Y25	HEX77[4] is assigned to PIN_AB15
SRAM_DQ[3] is assigned to PIN_AH4	HEX33[3] is assigned to PIN_W26	HEX77[5] is assigned to PIN_AA15
SRAM_DQ[4] is assigned to PIN_AF6	HEX33[4] is assigned to PIN_Y26	HEX77[6] is assigned to PIN_AC17
SRAM_DQ[5] is assigned to PIN_AG6	HEX33[5] is assigned to PIN_W27	HEX88[0] is assigned to PIN_AD17
SRAM_DQ[6] is assigned to PIN_AH6	HEX33[6] is assigned to PIN_W28	HEX88[1] is assigned to PIN_AE17
SRAM_DQ[7] is assigned to PIN_AF7	HEX44[0] is assigned to PIN_V21	HEX88[2] is assigned to PIN_AG17
SRAM_DQ[8] is assigned to PIN_AD1	HEX44[1] is assigned to PIN_U21	HEX88[3] is assigned to PIN_AH17
SRAM_DQ[9] is assigned to PIN_AD2	HEX44[2] is assigned to PIN_AB20	HEX88[4] is assigned to PIN_AF17
SRAM_DQ[10] is assigned to PIN_AE2	HEX44[3] is assigned to PIN_AA21	HEX88[5] is assigned to PIN_AG18
SRAM_DQ[11] is assigned to PIN_AE1	HEX44[4] is assigned to PIN_AD24	HEX88[6] is assigned to PIN_AA14
SRAM_DQ[12] is assigned to PIN_AE3	HEX44[5] is assigned to PIN_AF23	

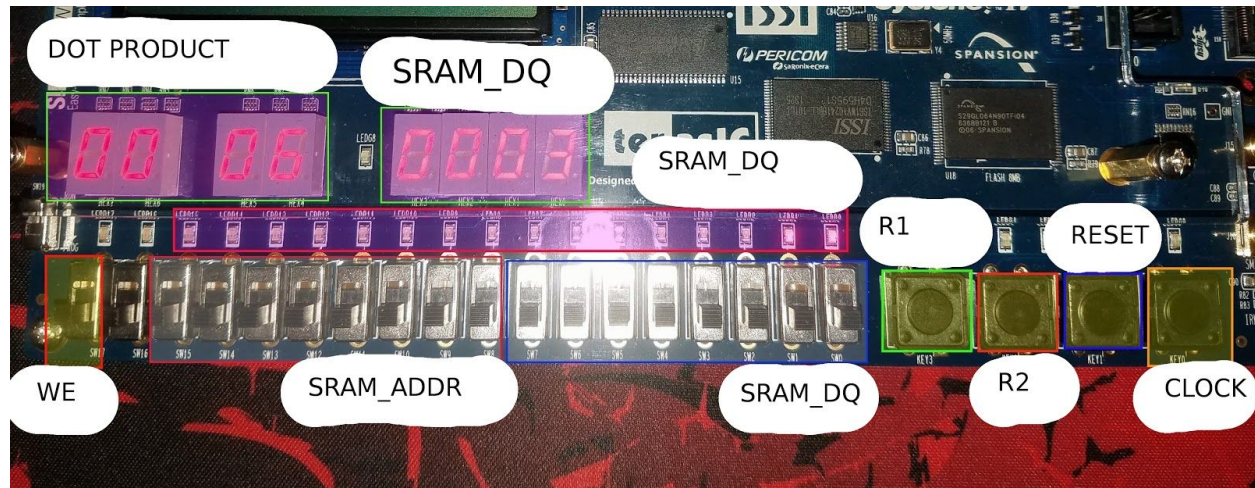


Figure 26: Digital circuit design of dot product project we have 8 switches to specify an address in memory, we have 8 to specify the data being stored at the given address, we have a clock for the value being stored in register 1, one for register 2 and one for the clock. We also have a write enable switch and a reset button. Currently on display we can see what is stored at address 0x0 and we can see that value is 3, the value in the register for the dot product is 6 because we had multiplied 3 and 2 and stored it on the accumulator register dot product location.

8 CONCLUSION

As it turns out designing the dot product is a design that takes time to compute. It takes several operation in order to finalize and it is time consuming. We have to store the variables and values that we will be using because we need to have the two complete arrays before we can perform the operations, and then we need to update the registers each one at a time, before performing the product and storing the product to the total sum at each instance, this happens for each index of the elements in the array and then once it reaches the end of the last element in the array it will finally have officially performed the dot product, and each instance before that it will be calculating partial dot products.

9 APPENDIX

DOTPRODUCT.txt	
1 To, Location	
2	
3 INPUT_DATA[0], PIN_AB28	68 SRAM_LB_N, PIN_AD4
4 INPUT_DATA[1], PIN_AC28	69
5 INPUT_DATA[2], PIN_AC27	70 SRAM_DQ[0], PIN_AH3
6 INPUT_DATA[3], PIN_AD27	71 SRAM_DQ[1], PIN_AF4
7 INPUT_DATA[4], PIN_AB27	72 SRAM_DQ[2], PIN_AG4
8 INPUT_DATA[5], PIN_AC26	73 SRAM_DQ[3], PIN_AH4
9 INPUT_DATA[6], PIN_AD26	74 SRAM_DQ[4], PIN_AF6
10 INPUT_DATA[7], PIN_AB26	75 SRAM_DQ[5], PIN_AG6
11 INPUT_ADDR[0], PIN_AC25	76 SRAM_DQ[6], PIN_AH6
12 INPUT_ADDR[1], PIN_AB25	77 SRAM_DQ[7], PIN_AF7
13 INPUT_ADDR[2], PIN_AC24	78 SRAM_DQ[8], PIN_AD1
14 INPUT_ADDR[3], PIN_AB24	79 SRAM_DQ[9], PIN_AD2
15 INPUT_ADDR[4], PIN_AB23	80 SRAM_DQ[10], PIN_AE2
16 INPUT_ADDR[5], PIN_AA24	81 SRAM_DQ[11], PIN_AE1
17 INPUT_ADDR[6], PIN_AA23	82 SRAM_DQ[12], PIN_AE3
18 INPUT_ADDR[7], PIN_AA22	83 SRAM_DQ[13], PIN_AE4
19 R, PIN_M21	84 SRAM_DQ[14], PIN_AF3
20 CLOCK, PIN_M23	85 SRAM_DQ[15], PIN_AG3
21 INPUT_WE, PIN_Y23	86
22 CLOCK1, PIN_M21	87 HEX11[0], PIN_G18
23 CLOCK2, PIN_R24	88 HEX11[1], PIN_F22
24 STORE, PIN_Y24	89 HEX11[2], PIN_E17
25	90 HEX11[3], PIN_L26
26 RESULTING[0], PIN_G19	91 HEX11[4], PIN_L25
27 RESULTING[1], PIN_F19	92 HEX11[5], PIN_J22
28 RESULTING[2], PIN_E19	93 HEX11[6], PIN_H22
29 RESULTING[3], PIN_F21	94
30 RESULTING[4], PIN_F18	95
31 RESULTING[5], PIN_E18	96 HEX22[0], PIN_M24
32 RESULTING[6], PIN_J19	97 HEX22[1], PIN_Y22
33 RESULTING[7], PIN_H19	98 HEX22[2], PIN_W21
34 RESULTING[8], PIN_J17	99 HEX22[3], PIN_W22
35 RESULTING[9], PIN_G17	00 HEX22[4], PIN_W25
36 RESULTING[10], PIN_J15	01 HEX22[5], PIN_U23
37 RESULTING[11], PIN_H16	02 HEX22[6], PIN_U24
38 RESULTING[12], PIN_J16	03
39 RESULTING[13], PIN_H17	04 HEX33[0], PIN_AA25
40 RESULTING[14], PIN_F15	05 HEX33[1], PIN_AA26
41 RESULTING[15], PIN_G15	06 HEX33[2], PIN_Y25
42	07 HEX33[3], PIN_W26
43 SRAM_ADDR[0], PIN_AB7	08 HEX33[4], PIN_Y26
44 SRAM_ADDR[1], PIN_AD7	09 HEX33[5], PIN_W27
45 SRAM_ADDR[2], PIN_AE7	10 HEX33[6], PIN_W28
46 SRAM_ADDR[3], PIN_AC7	11
47 SRAM_ADDR[4], PIN_AB6	12 HEX44[0], PIN_V21
48 SRAM_ADDR[5], PIN_AE6	13 HEX44[1], PIN_U21
49 SRAM_ADDR[6], PIN_AB5	14 HEX44[2], PIN_AB20
50 SRAM_ADDR[7], PIN_AC5	15 HEX44[3], PIN_AA21
51 SRAM_ADDR[8], PIN_AF5	16 HEX44[4], PIN_AD24
52 SRAM_ADDR[9], PIN_T7	17 HEX44[5], PIN_AF23
53 SRAM_ADDR[10], PIN_AF2	18 HEX44[6], PIN_Y19
54 SRAM_ADDR[11], PIN_AD3	19
55 SRAM_ADDR[12], PIN_AB4	20 HEX55[0], PIN_AB19
56 SRAM_ADDR[13], PIN_AC3	21 HEX55[1], PIN_AA19
57 SRAM_ADDR[14], PIN_AA4	22 HEX55[2], PIN_AG21
58 SRAM_ADDR[15], PIN_AB11	23 HEX55[3], PIN_AH21
59 SRAM_ADDR[16], PIN_AC11	24 HEX55[4], PIN_AE19
60 SRAM_ADDR[17], PIN_AB9	25 HEX55[5], PIN_AF19
61 SRAM_ADDR[18], PIN_AB8	26 HEX55[6], PIN_AE18
62 SRAM_ADDR[19], PIN_T8	27
63	28 HEX66[0], PIN_AD18
64 SRAM_CE_N, PIN_AF8	29 HEX66[1], PIN_AC18
65 SRAM_OE_N, PIN_AD5	30 HEX66[2], PIN_AB18
66 SRAM_WE_N, PIN_AE8	31 HEX66[3], PIN_AH19
67 SRAM_UB_N, PIN_AC4	32 HEX66[4], PIN_AG19
	33 HEX66[5], PIN_AF18
	134 HEX66[6], PIN_AH18
	135
	136 HEX77[0], PIN_AA17
	137 HEX77[1], PIN_AB16
	138 HEX77[2], PIN_AA16
	139 HEX77[3], PIN_AB17
	140 HEX77[4], PIN_AB15
	141 HEX77[5], PIN_AA15
	142 HEX77[6], PIN_AC17
	143
	144 HEX88[0], PIN_AD17
	145 HEX88[1], PIN_AE17
	146 HEX88[2], PIN_AG17
	147 HEX88[3], PIN_AH17
	148 HEX88[4], PIN_AF17
	149 HEX88[5], PIN_AG18
	150 HEX88[6], PIN_AA14

Figure 27: Pin assignments for DOT PRODUCT final design.