The City College of New York

# Computer Organization Lab

# CSC 34300

**(Spring 2024)**

**Final Lab**

# Dot Product


## Mahir Asef

**Instructor: Albi Arapi**


**Date: 05.13.2024**

# Table of Contents

## Objective:

The purpose of this laboratory experiment is to design a processor unit that computes dot product of two vectors in Quartus VHDL. In this experiment, we will use various components that we have built in the past such as add/sub circuit, sram, multiplier, multiplexer, and registers to build the dot product computation unit. Our goal is to design the circuit using the components listed in schematic and generate VHDL code which we will then use to write a testbench VHDL code. We will use the testbench VHDL code to verify our design by compiling it in Model Sim Simulation. Our simulation should show how the data are initialized in memory address and how the vector components get multiplied and added to show the final sum of dot product.

## Functionality and Specifications:

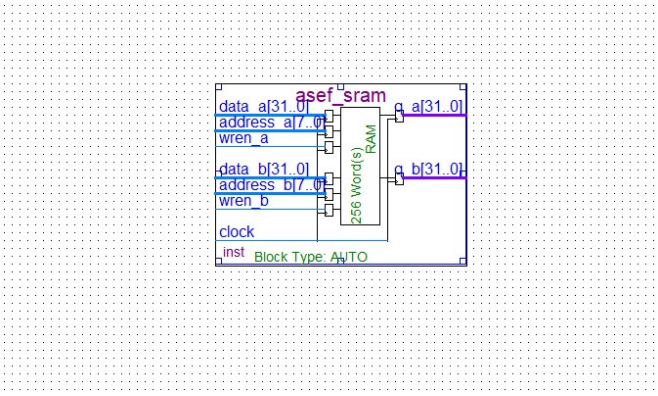
To design the processing unit of dot product computation, we need a few components. In our design we will be using a 2 port sram, a 32-bit multiplier, a 64 bit add/sub circuit, a 64 bit 2:1 mux and a 64 bit non shift register.

### SRAM:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY asef_sram IS
    PORT
    (
        address_a      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        address_b      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        clock     : IN STD_LOGIC  := '1';
        data_a      : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        data_b      : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wren_a      : IN STD_LOGIC  := '0';
        wren_b      : IN STD_LOGIC  := '0';
        q_a       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        q_b       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END asef_sram;


ARCHITECTURE SYN OF asef_sram IS

    SIGNAL sub_wire0   : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL sub_wire1   : STD_LOGIC_VECTOR (31 DOWNTO 0);

BEGIN
    q_a     <= sub_wire0(31 DOWNTO 0);
    q_b     <= sub_wire1(31 DOWNTO 0);

    altsyncram_component : altsyncram
    GENERIC MAP (
        address_reg_b => "CLOCK0",
        clock_enable_input_a => "BYPASS",
        clock_enable_input_b => "BYPASS",
        clock_enable_output_a => "BYPASS",
        clock_enable_output_b => "BYPASS",
        indata_reg_b => "CLOCK0",
        intended_device_family => "Cyclone V",
        lpm_type => "altsyncram",
        numwords_a => 256,
        numwords_b => 256,
        operation_mode => "BIDIR_DUAL_PORT",
        outdata_aclr_a => "NONE",
```

```
        outdata_aclr_b => "NONE",
        outdata_reg_a => "CLOCK0",
        outdata_reg_b => "CLOCK0",
        power_up_uninitialized => "FALSE",
        read_during_write_mode_mixed_ports => "DONT_CARE",
        read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
        read_during_write_mode_port_b => "NEW_DATA_NO_NBE_READ",
        widthad_a => 8,
        widthad_b => 8,
        width_a => 32,
        width_b => 32,
        width_byteena_a => 1,
        width_byteena_b => 1,
        wrcontrol_wraddress_reg_b => "CLOCK0"
    )
    PORT MAP (
        address_a => address_a,
        address_b => address_b,
        clock0 => clock,
        data_a => data_a,
        data_b => data_b,
        wren_a => wren_a,
        wren_b => wren_b,
        q_a => sub_wire0,
        q_b => sub_wire1
    );


END SYN;
```

The above is the VHDL code for the component 2 ports SRAM. Using the LPM module avaailable in Quartus, we have created this SRAM that can write two data input from two different memoery addresses at a time and feeds them to the output to be read further. In our experiment, the data inputs are 32 bits where as the memory space is 256 words which means this SRAM can hold 8 such 32 bit data. By reading the wren and clock pulses, the output reads the 32 bit data. This component is shown in the following:

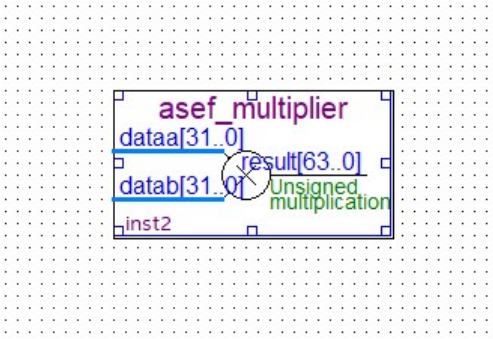



## Multiplier:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY asef_multiplier IS
    PORT
    (
        dataa    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        datab    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        result   : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
END asef_multiplier;

ARCHITECTURE SYN OF asef_multiplier IS

    SIGNAL sub_wire0  : STD_LOGIC_VECTOR (63 DOWNTO 0);


    COMPONENT lpm_mult
    GENERIC (
        lpm_hint    : STRING;
        lpm_representation    : STRING;
        lpm_type    : STRING;
        lpm_widtha    : NATURAL;
        lpm_widthb    : NATURAL;
        lpm_widthp    : NATURAL
    );
    PORT (
            dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            result   : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    result    <= sub_wire0(63 DOWNTO 0);

    lpm_mult_component : lpm_mult
    GENERIC MAP (
        lpm_hint => "MAXIMIZE_SPEED=5",
        lpm_representation => "UNSIGNED",
```

```
        lpm_type => "LPM_MULT",
        lpm_widtha => 32,
        lpm_widthb => 32,
        lpm_widthp => 64
    )
    PORT MAP (
        dataa => dataa,
        datab => datab,
        result => sub_wire0
    );


END SYN;
```

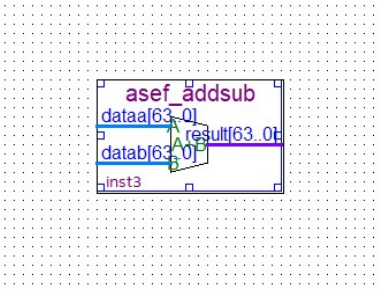
This VHDL code shown above is for the 32 bit multiplier component which we have also created from the Quartus LPM module. The 32 bit multiplier takes in two 32 bits inputs and produces a 64 bit outputs. For our design it will take the data coming from SRAM and multiply them together wjere the multiplication result is stores in the output pin for further examination. This component si shown below:




## Add/Sub:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY asef_addsub IS
    PORT
    (
        dataa    : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        datab    : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
END asef_addsub;

ARCHITECTURE SYN OF asef_addsub IS

    SIGNAL sub_wire0  : STD_LOGIC_VECTOR (63 DOWNTO 0);


    COMPONENT lpm_add_sub
    GENERIC (
        lpm_direction     : STRING;
        lpm_hint    : STRING;
        lpm_representation      : STRING;
        lpm_type    : STRING;
        lpm_width      : NATURAL
    );
    PORT (
          dataa : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
          datab : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
          result   : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    result    <= sub_wire0(63 DOWNTO 0);

    LPM_ADD_SUB_component : LPM_ADD_SUB
    GENERIC MAP (
        lpm_direction => "ADD",
        lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
        lpm_representation => "UNSIGNED",
        lpm_type => "LPM_ADD_SUB",
        lpm_width => 64
    )
    PORT MAP (
        dataa => dataa,
        datab => datab,
        result => sub_wire0
    );


END SYN;
```

This is the VHDL code for the adder/subtractor circuit we have imported from the LPM module. We have initialized this add/sub module with only add operation enabled since dot product operation only adds two elements. For our design, we have created 64-bit module which takes in two 64 bits data and outputs the result of addition as a 64 bit data. The component is shown below:



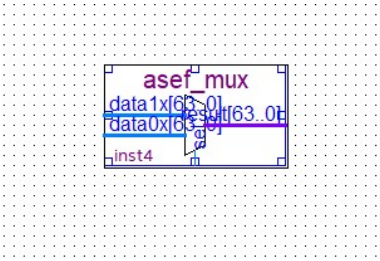

## 2x1 Multiplexer:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY asef_mux IS
   PORT
   (
      data0x      : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
      data1x      : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
      sel      : IN STD_LOGIC ;
      result      : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
   );
END asef_mux;


ARCHITECTURE SYN OF asef_mux IS

-- type STD_LOGIC_2D is array (NATURAL RANGE <>, NATURAL RANGE <>) of STD_LOGIC;

   SIGNAL sub_wire0  : STD_LOGIC_VECTOR (63 DOWNTO 0);
   SIGNAL sub_wire1  : STD_LOGIC_2D (1 DOWNTO 0, 63 DOWNTO 0);
   SIGNAL sub_wire2  : STD_LOGIC_VECTOR (63 DOWNTO 0);
   SIGNAL sub_wire3  : STD_LOGIC ;
   SIGNAL sub_wire4  : STD_LOGIC_VECTOR (0 DOWNTO 0);
   SIGNAL sub_wire5  : STD_LOGIC_VECTOR (63 DOWNTO 0);

BEGIN
   sub_wire2    <= data0x(63 DOWNTO 0);
   sub_wire0    <= data1x(63 DOWNTO 0);
   sub_wire1(1, 0)    <= sub_wire0(0);
   sub_wire1(1, 1)    <= sub_wire0(1);
   sub_wire1(1, 2)    <= sub_wire0(2);
   sub_wire1(1, 3)    <= sub_wire0(3);
   sub_wire1(1, 4)    <= sub_wire0(4);
   sub_wire1(1, 5)    <= sub_wire0(5);
   sub_wire1(1, 6)    <= sub_wire0(6);
   sub_wire1(1, 7)    <= sub_wire0(7);
   sub_wire1(1, 8)    <= sub_wire0(8);
   sub_wire1(1, 9)    <= sub_wire0(9);
   sub_wire1(1, 10)    <= sub_wire0(10);
   sub_wire1(1, 11)    <= sub_wire0(11);
   sub_wire1(1, 12)    <= sub_wire0(12);
   sub_wire1(1, 13)    <= sub_wire0(13);
```

```
 82     sub_wire1(1, 14)    <= sub_wire0(14);
 83     sub_wire1(1, 15)    <= sub_wire0(15);
 84     sub_wire1(1, 16)    <= sub_wire0(16);
 85     sub_wire1(1, 17)    <= sub_wire0(17);
 86     sub_wire1(1, 18)    <= sub_wire0(18);
 87     sub_wire1(1, 19)    <= sub_wire0(19);
 88     sub_wire1(1, 20)    <= sub_wire0(20);
 89     sub_wire1(1, 21)    <= sub_wire0(21);
 90     sub_wire1(1, 22)    <= sub_wire0(22);
 91     sub_wire1(1, 23)    <= sub_wire0(23);
 92     sub_wire1(1, 24)    <= sub_wire0(24);
 93     sub_wire1(1, 25)    <= sub_wire0(25);
 94     sub_wire1(1, 26)    <= sub_wire0(26);
 95     sub_wire1(1, 27)    <= sub_wire0(27);
 96     sub_wire1(1, 28)    <= sub_wire0(28);
 97     sub_wire1(1, 29)    <= sub_wire0(29);
 98     sub_wire1(1, 30)    <= sub_wire0(30);
 99     sub_wire1(1, 31)    <= sub_wire0(31);
100     sub_wire1(1, 32)    <= sub_wire0(32);
101     sub_wire1(1, 33)    <= sub_wire0(33);
102     sub_wire1(1, 34)    <= sub_wire0(34);
103     sub_wire1(1, 35)    <= sub_wire0(35);
104     sub_wire1(1, 36)    <= sub_wire0(36);
105     sub_wire1(1, 37)    <= sub_wire0(37);
106     sub_wire1(1, 38)    <= sub_wire0(38);
107     sub_wire1(1, 39)    <= sub_wire0(39);
108     sub_wire1(1, 40)    <= sub_wire0(40);
109     sub_wire1(1, 41)    <= sub_wire0(41);
110     sub_wire1(1, 42)    <= sub_wire0(42);
111     sub_wire1(1, 43)    <= sub_wire0(43);
112     sub_wire1(1, 44)    <= sub_wire0(44);
113     sub_wire1(1, 45)    <= sub_wire0(45);
114     sub_wire1(1, 46)    <= sub_wire0(46);
115     sub_wire1(1, 47)    <= sub_wire0(47);
116     sub_wire1(1, 48)    <= sub_wire0(48);
117     sub_wire1(1, 49)    <= sub_wire0(49);
118     sub_wire1(1, 50)    <= sub_wire0(50);
119     sub_wire1(1, 51)    <= sub_wire0(51);
120     sub_wire1(1, 52)    <= sub_wire0(52);
121     sub_wire1(1, 53)    <= sub_wire0(53);
122     sub_wire1(1, 54)    <= sub_wire0(54);
123     sub_wire1(1, 55)    <= sub_wire0(55);
124     sub_wire1(1, 56)    <= sub_wire0(56);
125     sub_wire1(1, 57)    <= sub_wire0(57);
126     sub_wire1(1, 58)    <= sub_wire0(58);
127     sub_wire1(1, 59)    <= sub_wire0(59);
128     sub_wire1(1, 60)    <= sub_wire0(60);
129     sub_wire1(1, 61)    <= sub_wire0(61);
130     sub_wire1(1, 62)    <= sub_wire0(62);
131     sub_wire1(1, 63)    <= sub_wire0(63);
132     sub_wire1(0, 0)    <= sub_wire2(0);
133     sub_wire1(0, 1)    <= sub_wire2(1);
134     sub_wire1(0, 2)    <= sub_wire2(2);
135     sub_wire1(0, 3)    <= sub_wire2(3);
136     sub_wire1(0, 4)    <= sub_wire2(4);
137     sub_wire1(0, 5)    <= sub_wire2(5);
138     sub_wire1(0, 6)    <= sub_wire2(6);
139     sub_wire1(0, 7)    <= sub_wire2(7);
140     sub_wire1(0, 8)    <= sub_wire2(8);
141     sub_wire1(0, 9)    <= sub_wire2(9);
142     sub_wire1(0, 10)    <= sub_wire2(10);
143     sub_wire1(0, 11)    <= sub_wire2(11);
144     sub_wire1(0, 12)    <= sub_wire2(12);
145     sub_wire1(0, 13)    <= sub_wire2(13);
146     sub_wire1(0, 14)    <= sub_wire2(14);
147     sub_wire1(0, 15)    <= sub_wire2(15);
148     sub_wire1(0, 16)    <= sub_wire2(16);
149     sub_wire1(0, 17)    <= sub_wire2(17);
150     sub_wire1(0, 18)    <= sub_wire2(18);
151     sub_wire1(0, 19)    <= sub_wire2(19);
152     sub_wire1(0, 20)    <= sub_wire2(20);
153     sub_wire1(0, 21)    <= sub_wire2(21);
154     sub_wire1(0, 22)    <= sub_wire2(22);
155     sub_wire1(0, 23)    <= sub_wire2(23);
156     sub_wire1(0, 24)    <= sub_wire2(24);
157     sub_wire1(0, 25)    <= sub_wire2(25);
158     sub_wire1(0, 26)    <= sub_wire2(26);
159     sub_wire1(0, 27)    <= sub_wire2(27);
160     sub_wire1(0, 28)    <= sub_wire2(28);
161     sub_wire1(0, 29)    <= sub_wire2(29);
162     sub_wire1(0, 30)    <= sub_wire2(30);
163     sub_wire1(0, 31)    <= sub_wire2(31);
164     sub_wire1(0, 32)    <= sub_wire2(32);
165     sub_wire1(0, 33)    <= sub_wire2(33);
166     sub_wire1(0, 34)    <= sub_wire2(34);
167     sub_wire1(0, 35)    <= sub_wire2(35);
168     sub_wire1(0, 36)    <= sub_wire2(36);
169     sub_wire1(0, 37)    <= sub_wire2(37);
170     sub_wire1(0, 38)    <= sub_wire2(38);
171     sub_wire1(0, 39)    <= sub_wire2(39);
```

```
    sub_wire1(0, 40)    <= sub_wire2(40);
    sub_wire1(0, 41)    <= sub_wire2(41);
    sub_wire1(0, 42)    <= sub_wire2(42);
    sub_wire1(0, 43)    <= sub_wire2(43);
    sub_wire1(0, 44)    <= sub_wire2(44);
    sub_wire1(0, 45)    <= sub_wire2(45);
    sub_wire1(0, 46)    <= sub_wire2(46);
    sub_wire1(0, 47)    <= sub_wire2(47);
    sub_wire1(0, 48)    <= sub_wire2(48);
    sub_wire1(0, 49)    <= sub_wire2(49);
    sub_wire1(0, 50)    <= sub_wire2(50);
    sub_wire1(0, 51)    <= sub_wire2(51);
    sub_wire1(0, 52)    <= sub_wire2(52);
    sub_wire1(0, 53)    <= sub_wire2(53);
    sub_wire1(0, 54)    <= sub_wire2(54);
    sub_wire1(0, 55)    <= sub_wire2(55);
    sub_wire1(0, 56)    <= sub_wire2(56);
    sub_wire1(0, 57)    <= sub_wire2(57);
    sub_wire1(0, 58)    <= sub_wire2(58);
    sub_wire1(0, 59)    <= sub_wire2(59);
    sub_wire1(0, 60)    <= sub_wire2(60);
    sub_wire1(0, 61)    <= sub_wire2(61);
    sub_wire1(0, 62)    <= sub_wire2(62);
    sub_wire1(0, 63)    <= sub_wire2(63);
    sub_wire3    <= sel;
    sub_wire4(0)    <= sub_wire3;
    result    <= sub_wire5(63 DOWNTO 0);

    LPM_MUX_component : LPM_MUX
    GENERIC MAP (
       lpm_size => 2,
       lpm_type => "LPM_MUX",
       lpm_width => 64,
       lpm_widths => 1
    )
    PORT MAP (
       data => sub_wire1,
       sel => sub_wire4,
       result => sub_wire5
    );


END SYN;
```

The component code above is for the Mux 2:1. IN our design we have imported a 64 bit mux from the lPM module which takes three inputs. The data inputs are 64 bits where the control (select) input is 1 bit. Whe the select input is 0, the output of the multiplxer is Input 1 and when the select input is 1, the output of the circuit is Input 2. The output of the mux will be 64 as wel as shown here:

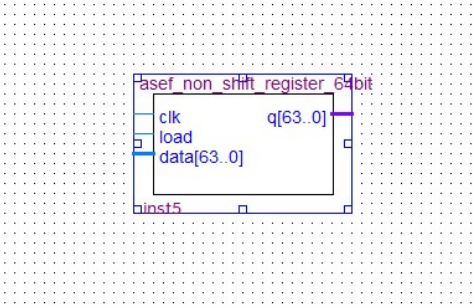



## Non Shift Register:

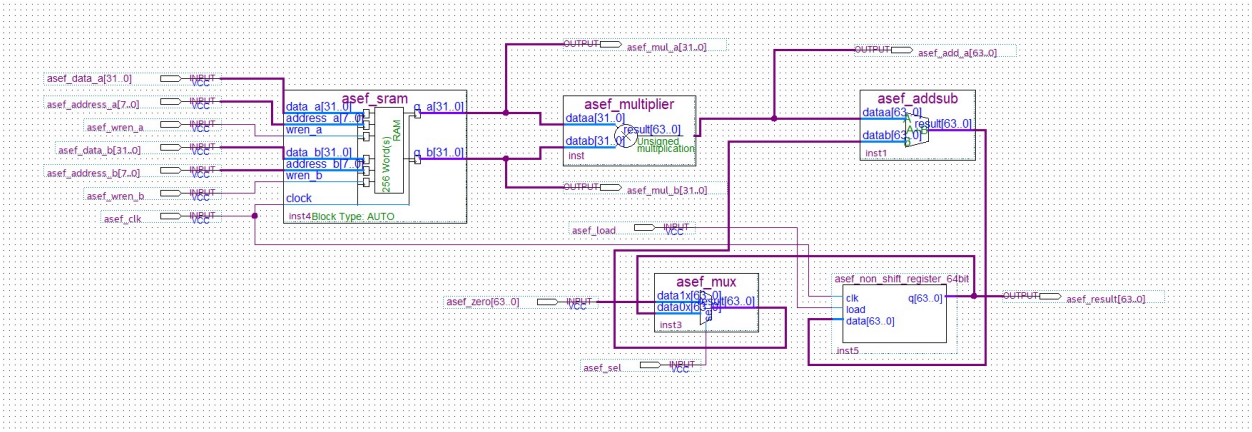
```
library ieee;
use ieee.std_logic_1164.all;

entity asef_non_shift_register_64bit is
    port (
        clk   : in  std_logic;
        load  : in  std_logic;
        data  : in  std_logic_vector(63 downto 0);
        q     : out std_logic_vector(63 downto 0)
    );
end asef_non_shift_register_64bit;

architecture Behavioral of asef_non_shift_register_64bit is
    signal reg_data : std_logic_vector(63 downto 0);
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if load = '1' then
                reg_data <= data;
            end if;
        end if;
    end process;

    q <= reg_data;
end Behavioral;

```

The final component in our design is a 64-bit non-shift register which helps store and process data for further analysis. Here, we have created the component from our own VHDL code. The register

has a clock input which will be synchronized with the clock input of the SRAM in our design. It also has load input and takes in a 64-bit input and outputs a 64-bit data as shown in the following:




## Dot Product (Block Diagram):




The above is the complete design of our dot product processing unit using all the components described above. In this design, we input two 32-bit data called “asef_data_a” and “asef_data_b” at 256 words addresses “address_a” and “address_b” respectively. With “asef_wren_a” and “asef_wren_b” being high and clock running, two 32-bit data are written simultaneously which is read in the outputs of the SRAM as “asef_mul_a” and “asef_mul_b”. These are the individual vector elements from vector 1 and vector 2 respectively. These two 32-bit data then go through a multiplier and the result is a 64-bit data stored as “asef_add_a”. This data then goes to the add/sub module as one of the 64-bit inputs. The 2$^{nd}$ 64-bit input comes from the 64-bit mux which will be explained soon below. Once the two inputs are ready, the add/sub module adds both inputs and takes the output to the input port of the 64-bit register with “asef_load” being enabled. This is done since add/sub module can’t store data, so we need to store it in the register. The register finally outputs the result of sum which is also a 64-bit data called “asef_result”. However, in the 1$^{st}$ iteration of calculating the dot product, there’s only one data available for the adder module as an input and hence the reason, we use the 64-bit mux to initialize a 0 input for “asef_zero”. With the select (“asef-sel”) input being high in our design, the mux outputs the zero input to go as the 2$^{nd}$ input of the adder module and then the process follows as above. Starting from the 2nd iteration, the sum output of the previous iteration is processed through the multiplexer with “asef_sel” being low to be added as the 2$^{nd}$ input of the add/sub module. The add/sub module then adds the previous sum

with the current multiplier result “asef-add_a” following the same process described above. The process keeps repeating until the loop is iterated through each element of the vector and the final sum output in “asef-result” is the result of the dot product computation.

## Dot Product (Generated VHDL):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY asef_dot_product IS
    PORT
    (
        asef_load :  IN  STD_LOGIC;
        asef_sel :  IN  STD_LOGIC;
        asef_wren_a :  IN  STD_LOGIC;
        asef_wren_b :  IN  STD_LOGIC;
        asef_clk :  IN  STD_LOGIC;
        asef_address_a :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        asef_address_b :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        asef_data_a :  IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        asef_data_b :  IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        asef_zero :  IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
        asef_add_a :  OUT  STD_LOGIC_VECTOR(63 DOWNTO 0);
        asef_mul_a :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
        asef_mul_b :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
        asef_result :  OUT  STD_LOGIC_VECTOR(63 DOWNTO 0)
    );
END asef_dot_product;

ARCHITECTURE bdf_type OF asef_dot_product IS

COMPONENT asef_multiplier
    PORT(dataa : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         datab : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         result : OUT STD_LOGIC_VECTOR(63 DOWNTO 0)
    );
END COMPONENT;

COMPONENT asef_addsub
    PORT(dataa : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
         datab : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
         result : OUT STD_LOGIC_VECTOR(63 DOWNTO 0)
    );
END COMPONENT;

COMPONENT asef_mux
    PORT(sel : IN STD_LOGIC;
         data0x : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
         data1x : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
         result : OUT STD_LOGIC_VECTOR(63 DOWNTO 0)
    );
END COMPONENT;

COMPONENT asef_sram
    PORT(wren_a : IN STD_LOGIC;
         wren_b : IN STD_LOGIC;
         clock : IN STD_LOGIC;
         address_a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         address_b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         data_a : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         data_b : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         q_a : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
         q_b : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END COMPONENT;

COMPONENT asef_non_shift_register_64bit
    PORT(clk : IN STD_LOGIC;
         load : IN STD_LOGIC;
         data : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
         q : OUT STD_LOGIC_VECTOR(63 DOWNTO 0)
    );
END COMPONENT;

SIGNAL   SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_1 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_2 :  STD_LOGIC_VECTOR(63 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_3 :  STD_LOGIC_VECTOR(63 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_4 :  STD_LOGIC_VECTOR(63 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_5 :  STD_LOGIC_VECTOR(63 DOWNTO 0);


BEGIN
asef_add_a <= SYNTHESIZED_WIRE_2;
asef_mul_a <= SYNTHESIZED_WIRE_0;
asef_mul_b <= SYNTHESIZED_WIRE_1;
asef_result <= SYNTHESIZED_WIRE_4;



b2v_inst : asef_multiplier
PORT MAP(dataa => SYNTHESIZED_WIRE_0,
         datab => SYNTHESIZED_WIRE_1,
         result => SYNTHESIZED_WIRE_2);
```

```
110
111
112 b2v_inst1 : asef_addsub
113 PORT MAP(dataa => SYNTHESIZED_WIRE_2,
114        datab => SYNTHESIZED_WIRE_3,
115        result => SYNTHESIZED_WIRE_5);
116
117
118 b2v_inst3 : asef_mux
119 PORT MAP(sel => asef_sel,
120        data0x => SYNTHESIZED_WIRE_4,
121        data1x => asef_zero,
122        result => SYNTHESIZED_WIRE_3);
123
124
125 b2v_inst4 : asef_sram
126 PORT MAP(wren_a => asef_wren_a,
127        wren_b => asef_wren_b,
128        clock => asef_clk,
129        address_a => asef_address_a,
130        address_b => asef_address_b,
131        data_a => asef_data_a,
132        data_b => asef_data_b,
133        q_a => SYNTHESIZED_WIRE_0,
134        q_b => SYNTHESIZED_WIRE_1);
135
136
137 b2v_inst5 : asef_non_shift_register_64bit
138 PORT MAP(clk => asef_clk,
139        load => asef_load,
140        data => SYNTHESIZED_WIRE_5,
141        q => SYNTHESIZED_WIRE_4);
142
143
144 END bdf_type;
```

This code above is the compiler generated VHDL code for the desig of dot product computer which works the exact same way as described above. We generated this code to use to create the testebench VHDL code that we need to verify our design with a simulation.

## Dot Product (Testbench Code):

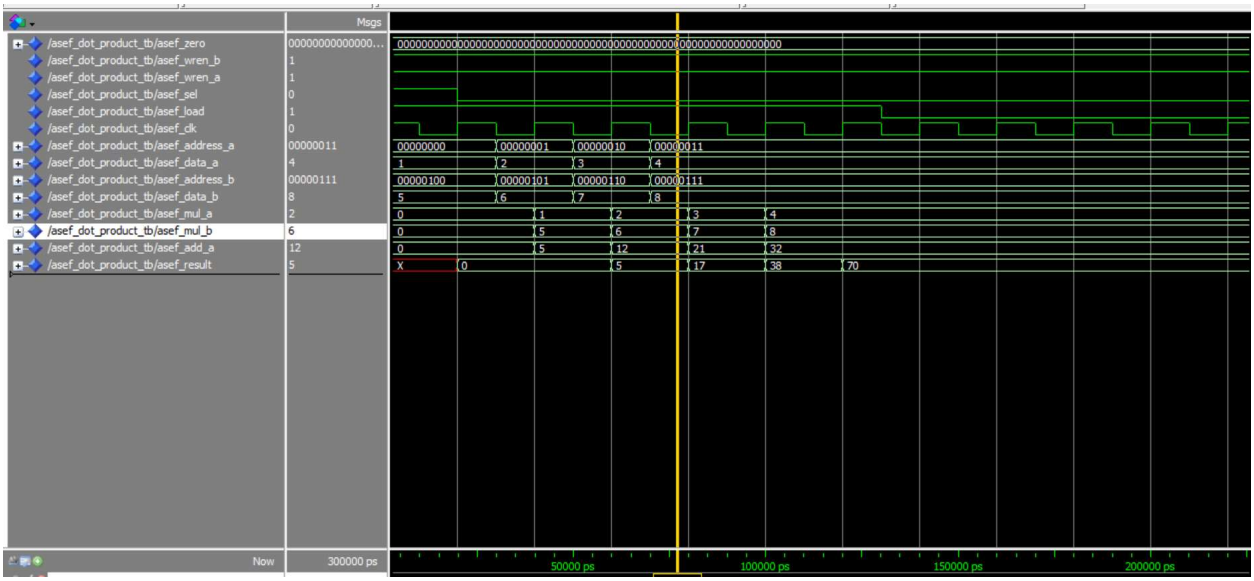
```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  ENTITY asef_dot_product_tb IS
6  END asef_dot_product_tb;
7
8  ARCHITECTURE behavior OF asef_dot_product_tb IS
9
10     -- Component declaration for the DUT (Device Under Test)
11     COMPONENT asef_dot_product
12     PORT (
13       asef_load :  IN  STD_LOGIC;
14       asef_sel :  IN  STD_LOGIC;
15       asef_wren_a :  IN  STD_LOGIC;
16       asef_wren_b :  IN  STD_LOGIC;
17       asef_clk :  IN  STD_LOGIC;
18       asef_address_a :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
19       asef_address_b :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
20       asef_data_a :  IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
21       asef_data_b :  IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
22       asef_zero :  IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
23       asef_add_a :  OUT  STD_LOGIC_VECTOR(63 DOWNTO 0);
24       asef_mul_a :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
25       asef_mul_b :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
26       asef_result :  OUT  STD_LOGIC_VECTOR(63 DOWNTO 0)
27     );
28 END COMPONENT;
29
30     -- Signals for testbench
31
32     SIGNAL asef_load : STD_LOGIC;
33     SIGNAL asef_sel : STD_LOGIC;
34     SIGNAL asef_wren_a : STD_LOGIC;
35     SIGNAL asef_wren_b : STD_LOGIC;
36     SIGNAL asef_clk : STD_LOGIC := '0';
37     SIGNAL asef_address_a : STD_LOGIC_VECTOR(7 DOWNTO 0);
38     SIGNAL asef_address_b : STD_LOGIC_VECTOR(7 DOWNTO 0);
39     SIGNAL asef_data_a : STD_LOGIC_VECTOR(31 DOWNTO 0);
40     SIGNAL asef_data_b : STD_LOGIC_VECTOR(31 DOWNTO 0);
41     SIGNAL asef_zero : STD_LOGIC_VECTOR(63 DOWNTO 0);
42     SIGNAL asef_add_a : STD_LOGIC_VECTOR(63 DOWNTO 0);
43     SIGNAL asef_mul_a : STD_LOGIC_VECTOR(31 DOWNTO 0);
44     SIGNAL asef_mul_b : STD_LOGIC_VECTOR(31 DOWNTO 0);
45     SIGNAL asef_result : STD_LOGIC_VECTOR(63 DOWNTO 0);
```

```
BEGIN

    -- Instantiate the DUT
    DUT : asef_dot_product
    PORT MAP (
    asef_load => asef_load,
    asef_sel => asef_sel,
    asef_wren_a => asef_wren_a,
    asef_wren_b => asef_wren_b,
    asef_clk => asef_clk,
    asef_address_a => asef_address_a,
    asef_address_b => asef_address_b,
    asef_data_a => asef_data_a,
    asef_data_b => asef_data_b,
    asef_zero => asef_zero,
    asef_add_a => asef_add_a,
    asef_mul_a => asef_mul_a,
    asef_mul_b => asef_mul_b,
    asef_result => asef_result
);

    -- Clock process
    PROCESS
    BEGIN
        asef_clk <= NOT asef_clk;  -- Changed from "clock" to "asef_clk"
        WAIT FOR 10 ns;  -- Half period
    END PROCESS;

    -- Memory process for writing data before dot product operation
    stimulus: PROCESS
    BEGIN
        asef_zero <= (OTHERS => '0');  -- Simplified to initialize to zero
        asef_load <= '1';
        asef_wren_a <= '1';
        asef_wren_b <= '1';


        asef_sel <= '1';
        asef_data_a <= std_logic_vector(to_unsigned(1, asef_data_a'length));
        asef_address_a <= "00000000";
        asef_data_b <= std_logic_vector(to_unsigned(5, asef_data_b'length));
        asef_address_b <= "00000100";
        WAIT FOR 20 ns;

        asef_sel <= '0';
        wait for 10 ns;
        asef_data_a <= std_logic_vector(to_unsigned(2, asef_data_a'length));
        asef_address_a <= "00000001";
        asef_data_b <= std_logic_vector(to_unsigned(6, asef_data_b'length));
        asef_address_b <= "00000101";
        WAIT FOR 20 ns;


        asef_data_a <= std_logic_vector(to_unsigned(3, asef_data_a'length));
        asef_address_a <= "00000010";
        asef_data_b <= std_logic_vector(to_unsigned(7, asef_data_b'length));
        asef_address_b <= "00000110";
        WAIT FOR 20 ns;


        asef_data_a <= std_logic_vector(to_unsigned(4, asef_data_a'length));
        asef_address_a <= "00000011";
        asef_data_b <= std_logic_vector(to_unsigned(8, asef_data_b'length));
        asef_address_b <= "00000111";
        Wait for 60 ns;
        asef_load <= '0';

        WAIT;
    END PROCESS;

END behavior;

```

The above is the testbench code written by us to test our dot product processor. In our testbench code, we have ported all the signals from the VHDL generated code and and mapped them as testbench signals properly. We initialized our clock with a low input. Next, in the process, we have provided the clock pulses before writing the test cases. For our test cases, first, we have enable writing and loading into register so that data gets written in memory as well as it also gets stored in register when the output of the the add/sub module is ready. In our 1$^{st}$ test case, we set the select input to high so that it adds zero to the result of of add/sub lpm. From 2$^{nd}$ iteration, we set the select input low so the data of previous sum calculation gets added to the current multplication result. In each case, we initialize different memory addresses with different data which ends up reading Vector 1 as <1,2,3,4> in memory location 0,1,2,3 and Vector 2 as <5,67,8> in memory loaction 4,5,6,7 respectively. Once we have iterated through each element, we unload the register to stop the computation.

## Simulation:

### ModelSim:




The above is a ModelSIm simulation that we have produced compiling all our files above with the testbench to verify our design of dot product circuit. Here we can see that “asef_zero” is intialized with zero and reamains so indefinitely. Both “asef_wren_a” and ‘asef_wren_b’ are set high to enable writing and select input is high for 1st iteration as well as the load. We also see our clock pulse is in effect as well as all the addresses are intialized properly with the specific data they were provided with in the testbench code. The, we can see when the data are written and read in “asef_mul_a” and “asef_mul_b” which then gets multiplied with result being stored in “asef_add_a”. First we multply 1*5 and then 6*2 followed by 7*3 and finally 8*4 which are stored as 5, 12, 21 and 32 respectively in “asef_add_a”. In the “asef_result” ouput we can see 5 indeed gets added with 0 for the 1st iteration and as we move to 2nd uteration, the select input toggles to enable adding 5 with 12 which is 17. 17 then gets added to 21 to produce 38 which is also added to 32 to generate the finl dot product result which is 70 which aligns with the mathematical process of calculating the dot product of two vectors shown as follows:

- ⇨ <1,2,3,4>.<5,6,7,8>
- ⇨ (1*5) + (2*6) + (3*7) + (4*8)
- ⇨ 5 + 12 + 21 + 32
- ⇨ 70

## Conclusion:

We have reached the end of our laboratory experiment of designing a processor in VHDL to compute dot product of two vectors. Throughout the lab, first we were able to identify the process of computing dot products which helped us determine the components to use to build the processor. Next, we have built all the components necessary such as 2 ports SRAM, add/sub

module, 2:1 mux, register and multiplier. We have initialized all the components to hold or input/output proper number of bits as necessary for our design which is explained above. Once we have built all our components, we have then connected them together using buses and wires and necessary in accurate manner to do the computation. We were able to successfully initialize data in memory addresses as well as able to multiply the data. We have also managed to add the multiplied data with 0 or previously multiplied data as necessary (the cases explained above) which we were able to store with register and show the final sum of the dot product. We then successfully created a testbench VHDL to provide testcases for our Model Sim Simulation which we compiled in model Sim. In the final step of our experiment, using Model Sim, we have simulated our data for a visual representation of the data and how they are processed through the design unit we have created ad we were able to explain the entire process of computation above and our final sum was indeed the accurate dot product of two vectors that we have provided. Overall, this lab has been incredibly helpful in introducing us to use singular or complex components to build a complex processing unit such as dot product computation unit. This experiment will help us to expand our knowledge further in hardware programming language and motivate us to work on building on furthermore practical processing units that we see in the computers of today.