LUND UNIVERSITY

LTH

FACULTY OF ENGINEERING

ETIN 35

IC project 2

Final Report

Hanyu Liu

February 26, 2023

Abstract

In this report, we verified the hardware accelerator which was designed in ICP 1. The SRAM which was used in ICP 1 was instead by distributed memory generator with Xilinx IP. The input data and coefficients will be embedded into the memory as coe file. Therefore, the finite state machine needs to be re-designed to fulfil the new requirements. The verification will be completed by Vivado and logic analyzer.

Contents

1. Introduction

The project aims to verify the IC project 1 hardware accelerator that realizes the multiplication of an input matrix with a size of 14x8 and a coefficient matrix with a size of 8x14. The product of the matrix multiplication P(n) is specified as equation (1). Each matrix contains 112 elements, and the result will be posted as a matrix with 196 elements. The input data and coefficients are pre-stored in SRAMs with Xilinx IP, respectively. And the results will be stored in another SRAM after calculation. The accelerator is constructed by using VHDL language and verified by a logic analyzer.

$$P(n) = X(n)A \qquad (1)$$

where X is the element of the input matrix while A is the element of the coefficient matrix.

The element in result (P) is the sum of 8 products. An example of the operation is given in equation (2).

$$p_{1,1} = x_{1,1}a_{1,1} + x_{1,2}a_{2,1} + x_{1,3}a_{3,1} + x_{1,4}a_{4,1} + x_{1,5}a_{5,1} + x_{1,6}a_{6,1} + x_{1,7}a_{7,1} + x_{1,8}a_{8,1} \qquad (2)$$

Finally, the result matrix with a size of 14x14 will be placed in an SRAM. There is an extra module called “max” which can show the maximum value found in the result matrix. Each input is an 8-bit unsigned number whilst each coefficient is a 7-bit unsigned number. When a result is obtained, it will be stored in the SRAM as an 18-bit unsigned number.

To verify the design in IC project 2, the state machine has been changed in modules “controller”, “load_coeff” and “load_input”. The block diagram has also been changed accordingly which has shown in the following paragraph. All the verifications are based on the FPGA of Xilinx “xc7a100tcsg324-1”.

2. Implementation

In this part, the block diagram and ASMD chart will be introduced respectively. The connection between each module will be clarified. ASMD chart will show how the module works. Comparing the ASMD charts in IC project 1, some states were deleted to adapt to the new requirements in IC project 2.

2.1 Block diagram

The block diagram of the top module is shown in Figure 2.1. This matrix multiplier can be divided into 8 modules, which are “load_coeff”, “load_input”, “multiply”, “store”, “max”, “controller”, and three SRAMs. There are two inputs and one output for the top module, “start”, “restart” and “max_out”. The functionality of each module and port is given as follows.

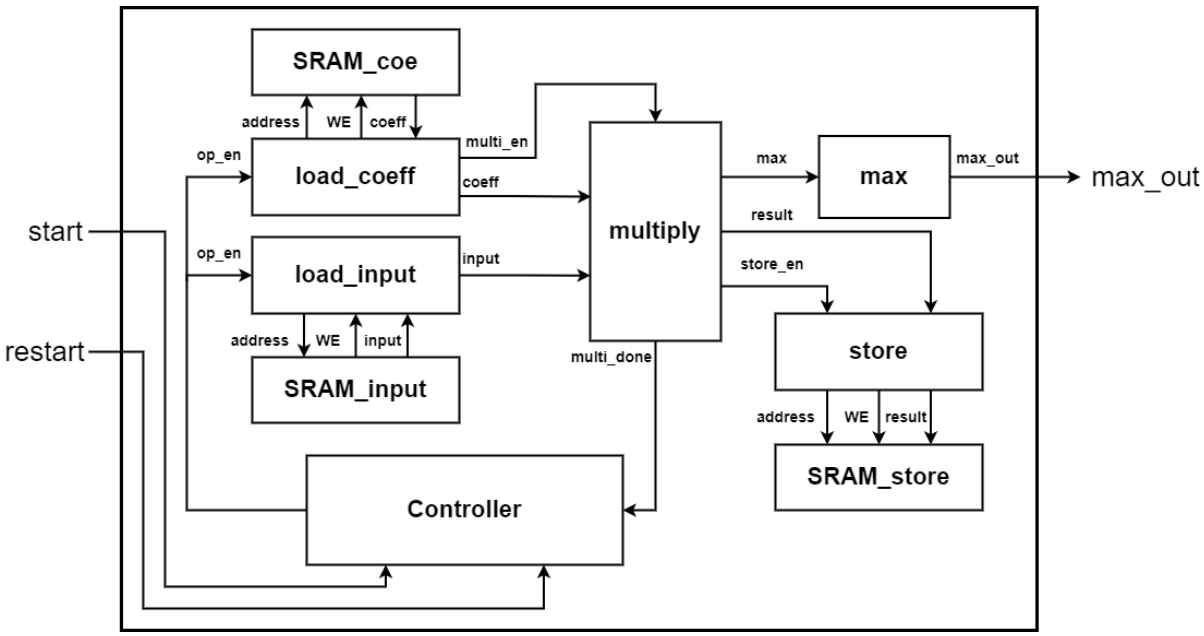

Figure 2.1: The block diagram of the matrix multiplier

Once the “start” signal is asserted, the system is activated and the finite state machines embedded in modules “load_coeff”, “load_input”, “multiply” and “controller” start to work. To make the system work in order, the “controller” sends control signals to tell when each operation should take place and when the result is available. “SRAM_coe” and “SRAM_input”, two SRAMs keep the coefficient matrix and input matrix respectively. Module “load_input” can be a temporary storage for a row of input matrix by four 16-bit registers. Module “load_coeff” is similar to “load_input” but it generates an address for “SRAM_coe” and gets the corresponding data from it. When the system is ready for multiplication operation, coefficients and inputs are brought to “multiply” from “load_coeff” and “load_input”, respectively. Then, the result produced by “multiply” is stored in “SRAM_store” with the help of module “store” and the maximum result found for the time being is placed in “max”.

2.2 ASMD charts

There are four ASMD charts shown in this part, “controller”, “load_coeff”, “load_input” and “multiply”. Each ASMD chart shows the condition and states. Each state in the state machine is explained, which will help to understand the design of the matrix multiplier.

2.2.1 Controller

The ASMD chart of the “controller” is shown in Figure 2.2.1. There are four states in this module.

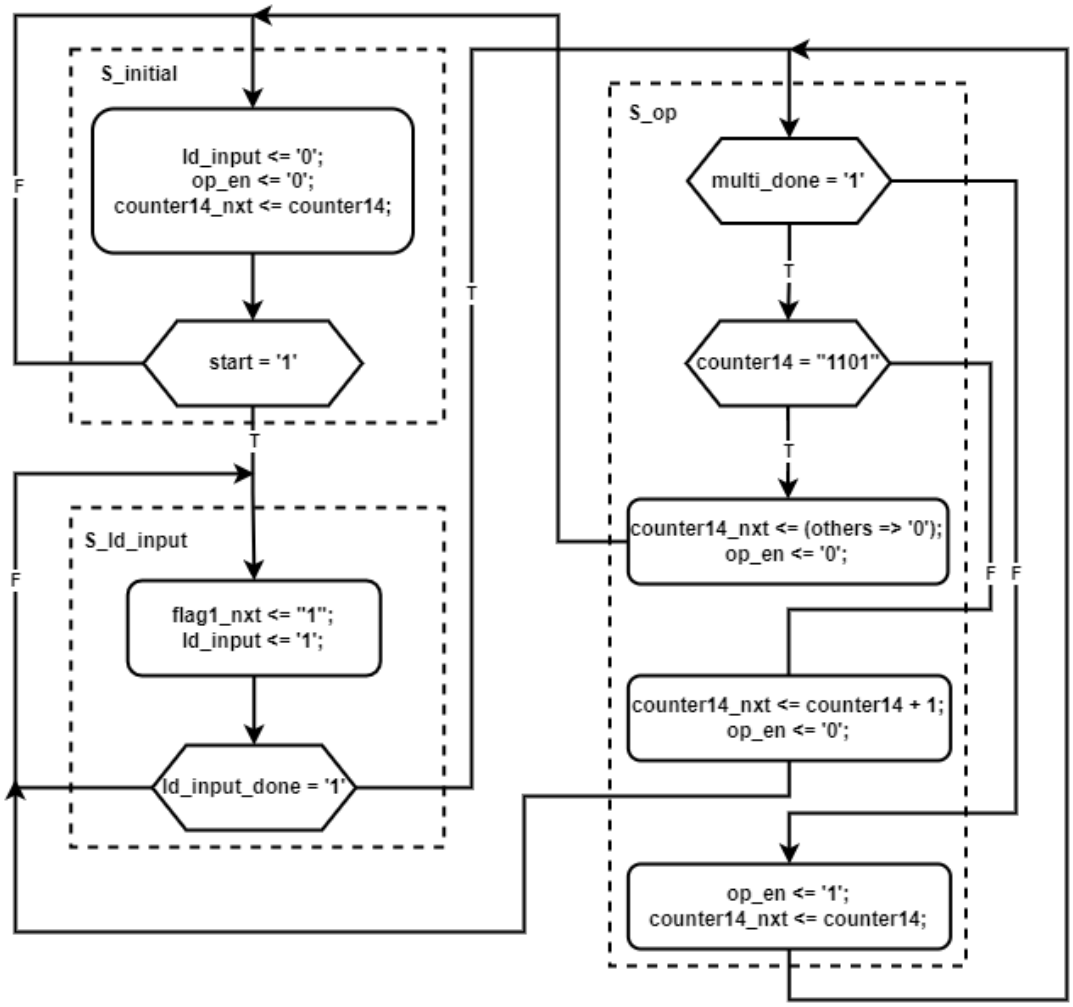

Figure 2.2.1: The ASMD chart of the “controller”

S_initial:

In this state, all signals are set to default values. Signal “flag1” in IC project 1 has been deleted due to all the data has been stored in SRAMs. There is no need to get the data from the outside. Thus, once the signal “start” is HIGH, the calculation will start immediately.

S_ld_input:

In this state, 8 inputs will be loaded from “SRAM_input” and stored in four 16-bit registers. After that, the signal “ldinput_done” will be set to HIGH, which indicates that the system is ready to produce a column of results. To complete a whole matrix multiplication, this process will be repeated 14 times.

S_op:

The state “s_op” is where the multiplication and accumulation take place. The loaded inputs and coefficients are retrieved from the register and SRAM, respectively, and one sum of the product can be obtained. Similarly, this process will be repeated 14 times to get 14 results.

2.2.2 Load coefficient

The ASMD chart of “load_coeff” is shown in Figure 2.2.2. There are three states in this module.

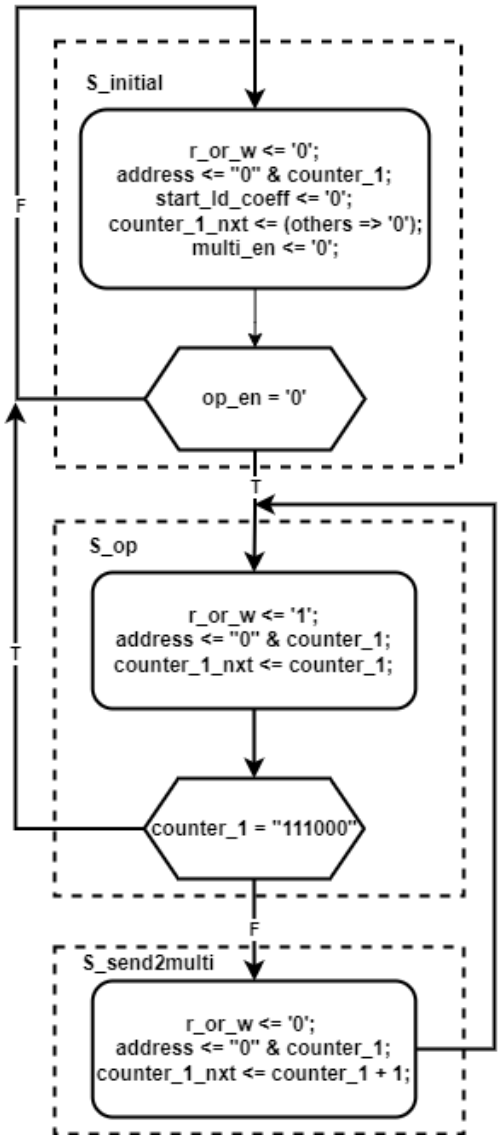

Figure 2.2.2: The ASMD chart of “load_coefficient”

S_initial:

In this state, all variables will be set to default values. The system will start immediately after signal “op_en” HIGH.

S_op:

In this state, an address for the SRAM is generated, and the corresponding data become available in the next clock cycle.

S_send2multi:

In this state, the data received from SRAM will be sent to module “multiply”. To move all the coefficients to “multiply”, the state machine will switch between “s_op” and “s_send2multi” 56 times.

2.2.3 Load input

The ASMD chart of “load_input” is shown in Figure 2.2.3. There are five states in this module.

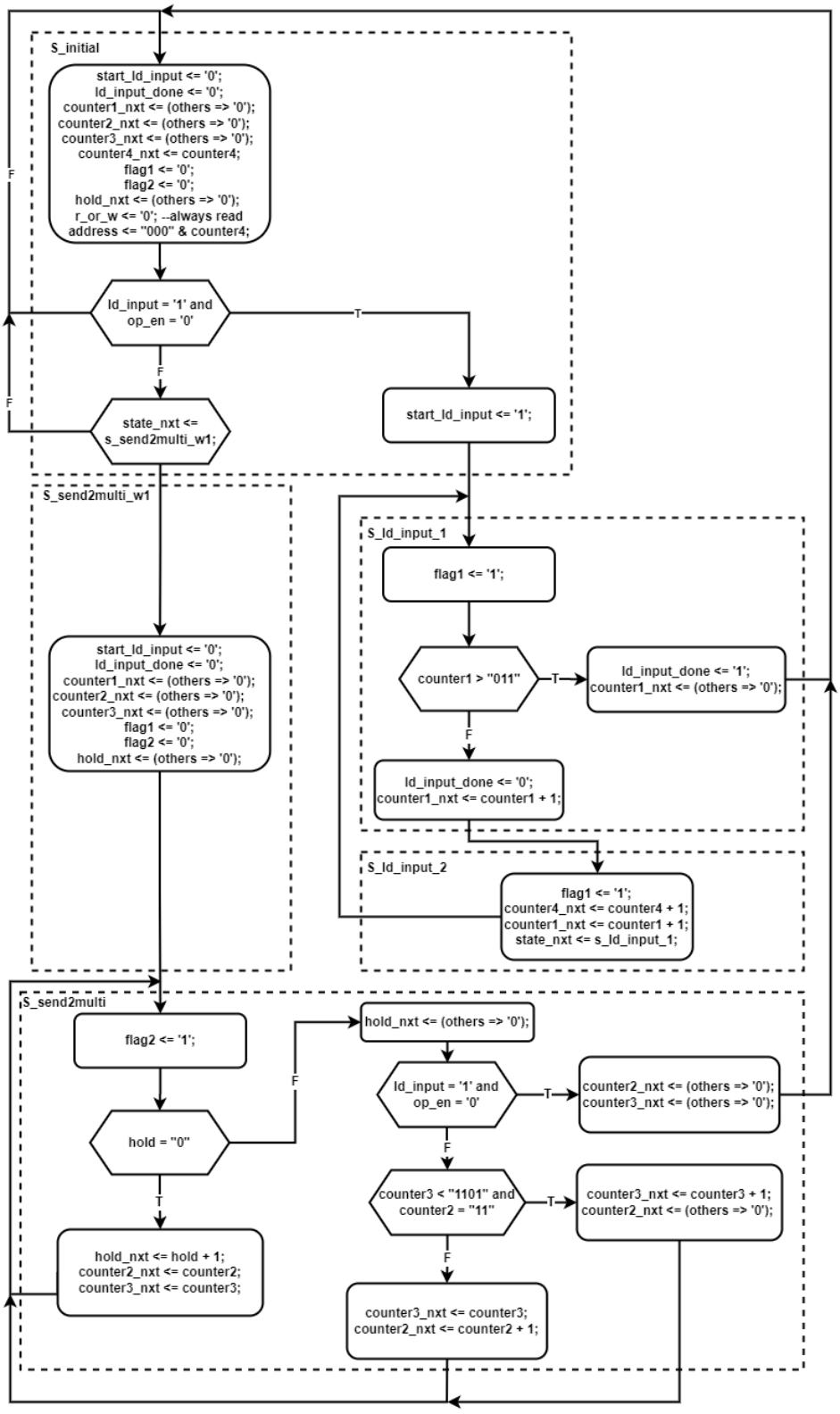

Figure 2.2.3: The ASMD chart of “load_input”

S_initial:

In this state, all variables will be set to default values. If signal “ld_input” is HIGH and signal “op_en” is LOW, the state machine will enter the state “s_ld_input_1” in the next clock cycle. When signal “ld_input” is LOW, and signal “op_en” is HIGH, the state machine will enter the state “s_send2multi_w1” in the next clock cycle.

S_ld_input_1:

In this state, the module will send the address to the “SRAM_input”. After receiving one raw input data, the state machine will return to the initial state.

S_ld_input_2:

In this state, the system will receive the input data from the SRAM and store them in registers.

S_send2multi_w1:

This state will wait for a clock cycle to ensure that the coefficients have been received.

S_send2multi:

In this state, 8 inputs will be sent to the module “multiply” two by two.

2.2.4 Multiply

The ASMD chart of “multiply” is shown in Figure 2.2.4. There are four states in this module. This part has no changes compared to the design in IC project 1.

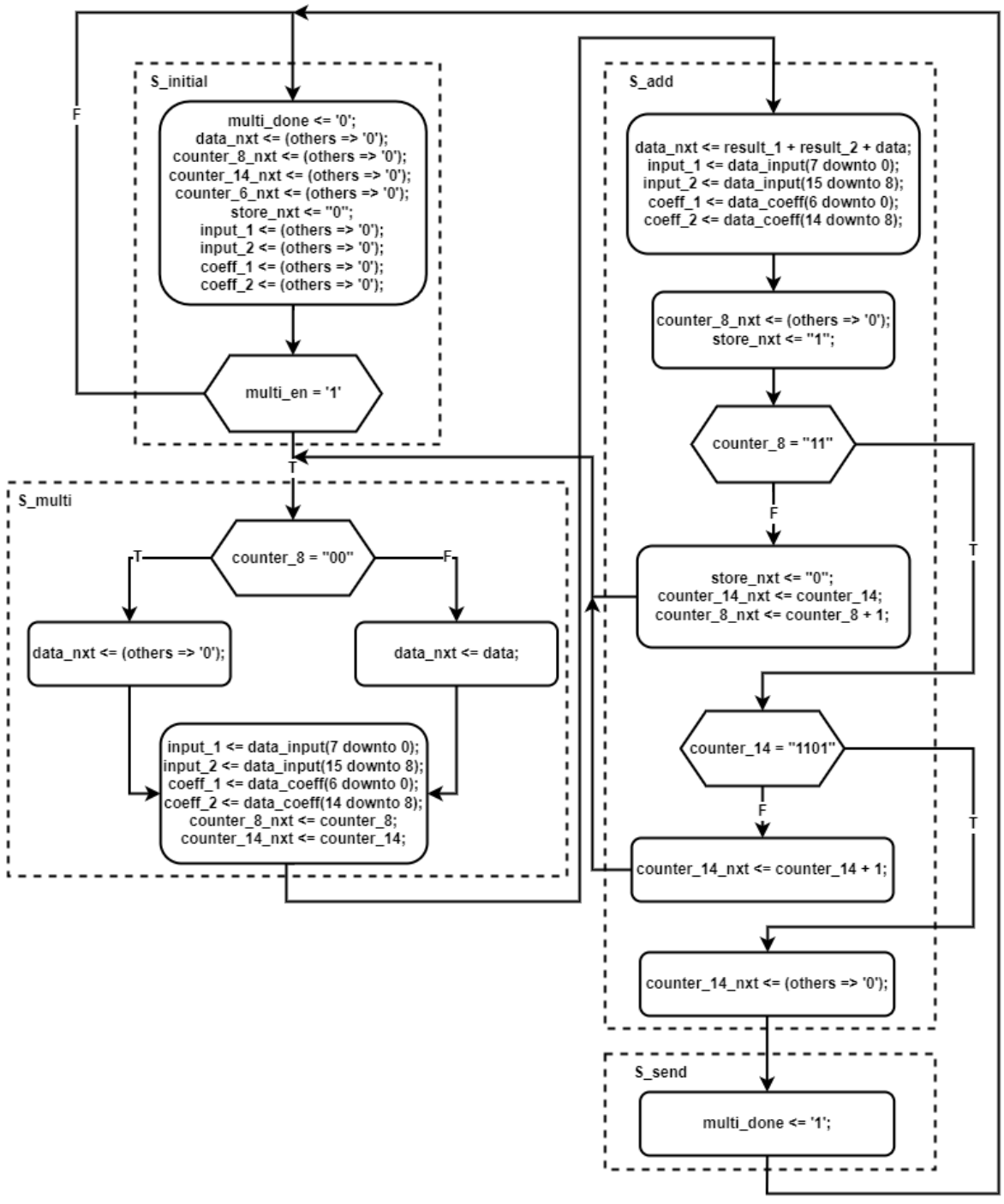

Figure 2.2.4: The ASMD chart of “operation”

S_initial:

In this state, all the variables are set to default values. The state machine will enter state “s_multi” when signal “multi_en” is HIGH.

S_multi:

There are two multipliers in the module “multiply”. In this state, 2 inputs from “load_input” and 2 coefficients from “load_coefficient” will be received and stored in registers for further processing. And the state machine will enter the state “s_add” in the next clock cycle.

S_add:

In this state, two multiplication operations will be performed on the inputs and the coefficients obtained in the state “S_multi”. Then, the intermediate value, which is the sum of these two products, is stored in the register for accumulation. As in equation (2), a result can be obtained after six more multiplication and accumulation operations take place.

S_send:

In this state, one of the results has been calculated. The result will be sent to the module “store” and module “max” within one clock cycle. The signal “multi_done” will be set as HIGH at the same time. The state machine will enter the initial state in the next clock cycle.

3. Verification

The multiplication was verified in Vivado, and the simulation results are shown in Figures 3.1 and 3.2. The results are stored in SRAM in order successfully which has been shown in figure 3.3. Figure 3.4 shows the system iterated 14 times due to 14 rows in the input matrix. Figure 3.5 shows the data are the same in both the simulation and logic analyzer.

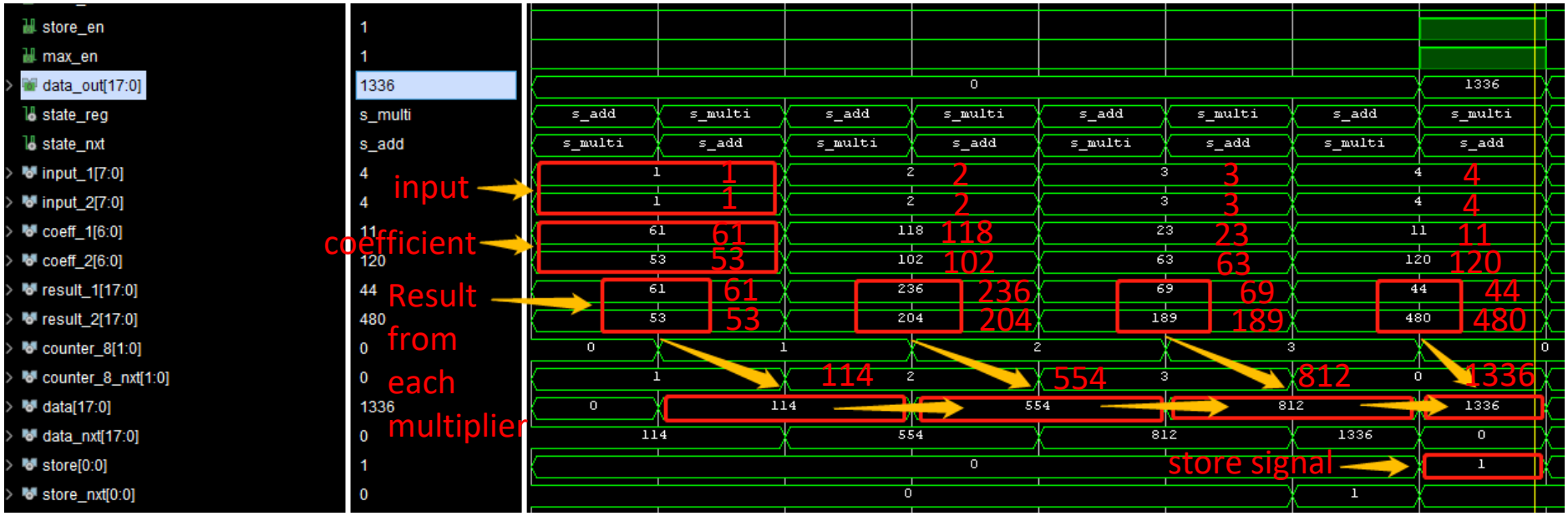

Figure 3.1: Simulation result from Vivado

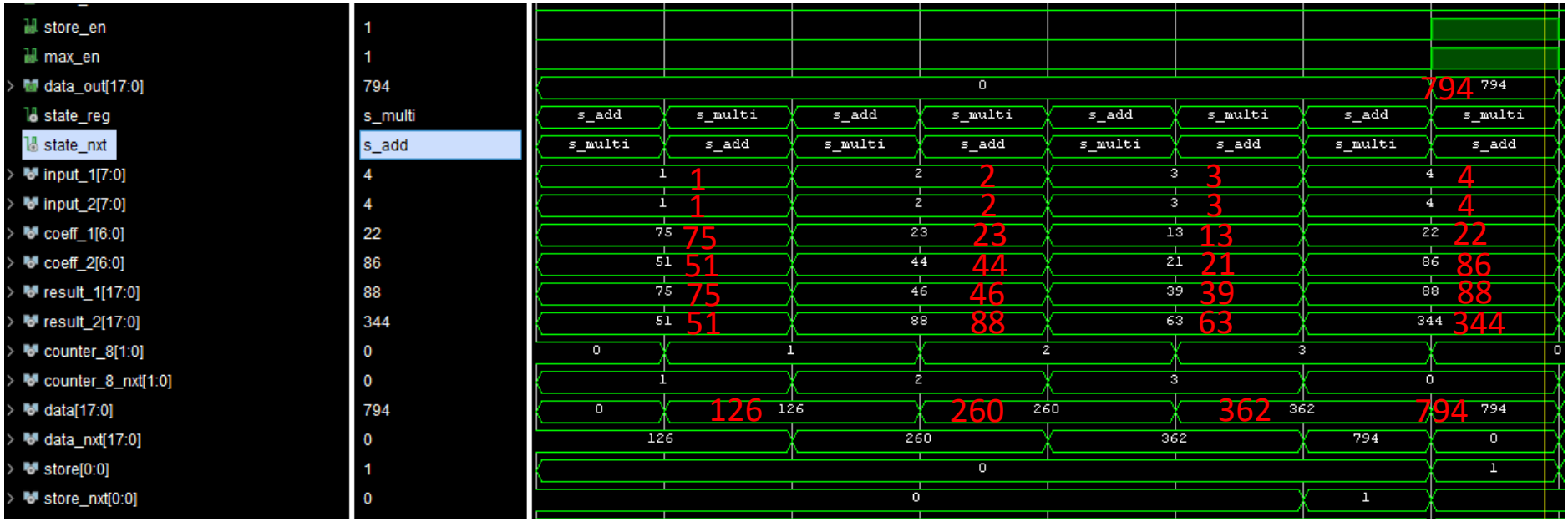

Figure 3.2: Another group of simulation results from Vivado

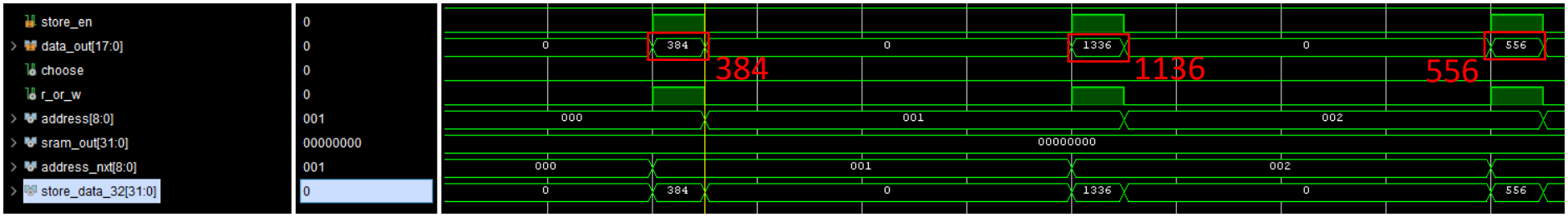

Figure 3.3: The results in decimal stored in memory successfully.

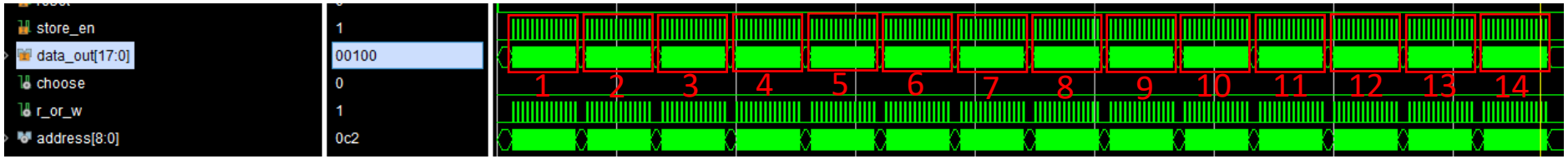

Figure 3.4: The calculation will iterate 14 times due to 14 rows in the input matrix.

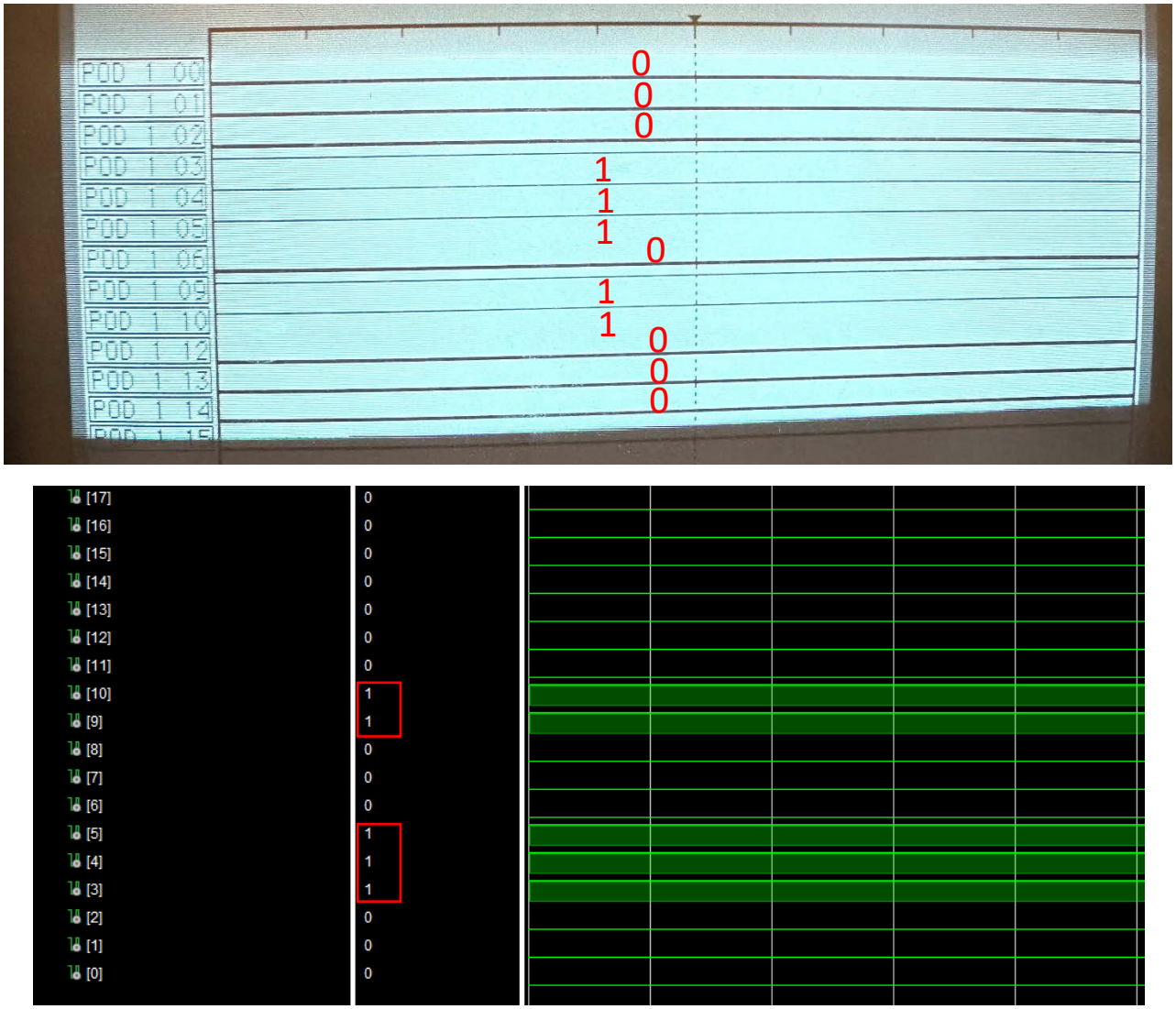

Figure 3.5: Compare the data detected in the logic analyzer with the simulation result.

4. Appendix

4.1 VHDL code for controller

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

entity controller is
  Port (
        clk, reset  : in std_logic;
        start       : in std_logic;
        ld_input_done   : in std_logic;
        multi_done  : in std_logic;
        ld_input    : out std_logic;
        op_en       : out std_logic

  );
end controller;

architecture Behavioral of controller is

component ff is
  generic(N:integer:=1);
  port(   D  :  in std_logic_vector(N-1 downto 0);
          Q  :  out std_logic_vector(N-1 downto 0);
        clk  :  in std_logic;
        reset:  in std_logic
      );
end component;

type state_type is (s_initial, s_ld_input, s_op);
signal state_reg, state_nxt : state_type;
signal counter14, counter14_nxt : std_logic_vector(3 downto 0) := (others
=> '0');

begin
--state contrl-------------------------------
process(clk, reset)
begin
    if reset = '1' then
        state_reg <= s_initial;
    elsif (clk'event and clk = '1') then
        state_reg <= state_nxt;
```

```
    end if;

end process;

--state machine-----------------------------------------
process(state_reg, start, ld_input_done, multi_done, counter14)
begin

    ld_input <= '0';
    op_en <= '0';
    counter14_nxt <= counter14;

    case state_reg is

        when s_initial =>
            counter14_nxt <= (others => '0');
            if start = '1' then
               state_nxt <= s_ld_input;
            else
               state_nxt <= s_initial;
            end if;

when s_ld_input =>
    ld_input <= '1';
    if ld_input_done = '1' then
       state_nxt <= s_op;
    else
       state_nxt <= s_ld_input;
    end if;

when s_op =>
    if multi_done = '1' then
       if counter14 = "1101" then
          counter14_nxt <= (others => '0');
          state_nxt <= s_initial;
          op_en <= '0';
       else
          counter14_nxt <= counter14 + 1;
          state_nxt <= s_ld_input;
          op_en <= '0';
       end if;
    else
       op_en <= '1';
       counter14_nxt <= counter14;
```

```
            state_nxt <= s_op;
        end if;
    end case;
end process;

counter_14: FF
  generic map(N => 4)
  port map(   D     =>counter14_nxt,
              Q     =>counter14,
            clk     =>clk,
            reset   =>reset
      );

end Behavioral;
```

4.2 VHDL code for load_coeff

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

entity load_coeff is
  Port (
        clk, reset  : in std_logic;
        --signal from controller
        op_en       : in std_logic;
        --control signal to multiply
        multi_en    : out std_logic;
        --coeff to multiply
        data_coeff  : out std_logic_vector(15 downto 0)
  );
end load_coeff;

architecture Behavioral of load_coeff is

component SRAM_coe
  port (
    clk     : in  std_logic;
    we      : in std_logic;
    a       : in  std_logic_vector (6 downto 0);
    d       : in  std_logic_vector (15 downto 0);
    qspo    : out std_logic_vector (15 downto 0)
    );
end component;

component ff is
  generic(N:integer:=1);
  port(   D  :  in std_logic_vector(N-1 downto 0);
          Q  :  out std_logic_vector(N-1 downto 0);
        clk  :  in std_logic;
        reset:  in std_logic
      );
end component;
--SRAM-------------------------------------------
signal choose       : std_logic;
signal r_or_w       : std_logic;
signal address      : std_logic_vector(6 downto 0);
-------------------------------------------------
```

```
type state_type is (s_initial, s_op, s_send2multi);
signal state_reg, state_nxt : state_type;

signal counter_1, counter_1_nxt : std_logic_vector(5 downto 0) := (others
=> '0');

signal coeff_32 : std_logic_vector(15 downto 0);
signal data_coeff_32 : std_logic_vector(15 downto 0);
signal coeff       : std_logic_vector(15 downto 0);

begin
--SRAM bits transfer----------------------
coeff_32 <= coeff;
data_coeff <= data_coeff_32(15 downto 0);
-----------------------------------------

Ram_coeff: SRAM_coe
  port map(
    clk     => clk,
    we      => r_or_w,
    a       => address,
    d       => coeff_32,
    qspo    => data_coeff_32
    );

--state contrl--------------------------------
process(clk, reset)
begin
    if reset = '1' then
        state_reg <= s_initial;
    elsif (clk'event and clk = '1') then
        state_reg <= state_nxt;
    end if;

end process;
--state machine-----------------------------------------------
process(state_reg, op_en, counter_1)
begin
    --SRAM-----------------------------
    choose <= '1';
    r_or_w <= '0';--read
    address <= "0" & counter_1;
    ----------------------------------
```

```
counter_1_nxt <= (others => '0');
multi_en <= '0';
case state_reg is

   when s_initial =>
      if op_en = '1' then
         state_nxt <= s_op;
      else
         state_nxt <= s_initial;
      end if;

   when s_op =>
      choose <= '0';
      r_or_w <= '0'; --read
      address <= "0" & counter_1;
      counter_1_nxt <= counter_1;
      if counter_1 = "111000" then --counter = 56 (address = 0 -55)
         state_nxt <= s_initial;
      elsif counter_1 = "000000" then
         multi_en <= '1';
         state_nxt <= s_send2multi;
      else
         multi_en <= '0';
         state_nxt <= s_send2multi;
      end if;

      when s_send2multi =>
         choose <= '0';
         r_or_w <= '0'; --read
         address <= "0" & counter_1;
         counter_1_nxt <= counter_1 + 1;
         state_nxt <= s_op;
   end case;
end process;
counter1: FF
  generic map(N => 6)
  port map(   D     =>counter_1_nxt,
              Q     =>counter_1,
              clk     =>clk,
              reset   =>reset
      );
end Behavioral;
```

4.3 VHDL code for load_input

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

entity load_input is
  Port (
        clk, reset  : in std_logic;
        ----------------------------------------------------
        ld_input    : in std_logic;
        ld_input_done   : out std_logic;--feedback to controller
        --signal from controller
        op_en       : in std_logic;
        start_ld_input  : out std_logic;
        data_input  : out std_logic_vector(15 downto 0)
  );
end load_input;

architecture Behavioral of load_input is

component SRAM_input
  port (
    clk     : in  std_logic;          --Active Low
    we      : in std_logic;
    a       : in  std_logic_vector (6 downto 0);
    d       : in  std_logic_vector (15 downto 0);
    qspo    : out std_logic_vector (15 downto 0)
    );
end component;

component ff is
  generic(N:integer:=1);
  port(   D  :  in std_logic_vector(N-1 downto 0);
          Q  :  out std_logic_vector(N-1 downto 0);
        clk  :  in std_logic;
        reset:  in std_logic
      );
end component;

--SRAM-----------------------------------------
signal choose       : std_logic;
```

```
signal r_or_w       : std_logic; -- Active Low (reand & write) --write '0' --
read '1'

signal address      : std_logic_vector(6 downto 0);

--------------------------------------------------


type state_type is (s_initial, s_ld_input_1, s_ld_input_2, s_send2multi,
s_send2multi_w1);

signal state_reg, state_nxt : state_type;


signal reg_1, reg_1_nxt : std_logic_vector(15 downto 0);

signal reg_2, reg_2_nxt : std_logic_vector(15 downto 0);

signal reg_3, reg_3_nxt : std_logic_vector(15 downto 0);

signal reg_4, reg_4_nxt : std_logic_vector(15 downto 0);


--enable write in reg

signal flag1 : std_logic;

--enable send data

signal flag2 : std_logic;


signal hold, hold_nxt : std_logic_vector(0 downto 0) := (others => '0');


--control load

signal counter1, counter1_nxt : std_logic_vector(2 downto 0) := (others =>
'0');

--control send

signal counter2, counter2_nxt : std_logic_vector(1 downto 0) := (others =>
'0');

--control the loop will be executed 14 times

signal counter3, counter3_nxt : std_logic_vector(3 downto 0) := (others =>
'0');


signal counter4, counter4_nxt : std_logic_vector(3 downto 0) := (others =>
'0');


signal input_32 : std_logic_vector(15 downto 0);

signal input    : std_logic_vector(15 downto 0);


begin


Ram_input: SRAM_input
  port map(
    clk     => clk,
    we      => r_or_w,
    a       => address,
    d       => input_32,
```

```
    qspo    => input
    );


--state contrl-----------------------------------------------
process(clk, reset)
begin
    if reset = '1' then
        state_reg <= s_initial;
    elsif (clk'event and clk = '1') then
        state_reg <= state_nxt;
    end if;
end process;


--state machine -----------------------------------------
process(state_reg, ld_input, op_en, counter1, counter2, counter3, counter4,
hold)
begin
    start_ld_input <= '0';
    ld_input_done <= '0';
    counter1_nxt <= (others => '0');
    counter2_nxt <= (others => '0');
    counter3_nxt <= (others => '0');
    counter4_nxt <= counter4;
    flag1 <= '0';
    flag2 <= '0';
    hold_nxt <= (others => '0');
    r_or_w <= '0'; --always read
    address <= "000" & counter4;


    case state_reg is


      when s_initial =>
        if ld_input = '1' and op_en = '0' then
            start_ld_input <= '1';--give signal to outside
            state_nxt <= s_ld_input_1;
        elsif ld_input = '0' and op_en = '1' then
            state_nxt <= s_send2multi_w1;
        else
            state_nxt <= s_initial;
        end if;


      when s_ld_input_1 =>
        flag1 <= '1';
        if counter1 > "011" then
```

```
            start_ld_input <= '1';
            ld_input_done <= '1';
            counter1_nxt <= (others => '0');
            state_nxt <= s_initial;
        else
            start_ld_input <= '1';
            ld_input_done <= '0';
            counter1_nxt <= counter1;
            state_nxt <= s_ld_input_2;
        end if;

    when s_ld_input_2 =>
        flag1 <= '1';
        counter4_nxt <= counter4 + 1;
        counter1_nxt <= counter1 + 1;
        state_nxt <= s_ld_input_1;

    when s_send2multi_w1 =>
        state_nxt <= s_send2multi;

    when s_send2multi =>
        flag2 <= '1';
        if hold = "0" then
            hold_nxt <= hold + 1;
            state_nxt <= s_send2multi;
            counter2_nxt <= counter2;
            counter3_nxt <= counter3;
        else
            hold_nxt <= (others => '0');
            if counter3 = "1101" and counter2 = "11" then
                counter2_nxt <= (others => '0');
                counter3_nxt <= (others => '0');
                state_nxt <= s_initial;
            elsif counter3 < "1101" and counter2 = "11" then
                counter3_nxt <= counter3 + 1;
                counter2_nxt <= (others => '0');
                state_nxt <= s_send2multi;
            else
                counter3_nxt <= counter3;
                counter2_nxt <= counter2 + 1;
                state_nxt <= s_send2multi;
            end if;
        end if;
    end case;
```

```
end process;
reg_1_nxt <= input when counter1 = "000" and flag1 = '1' else reg_1;
reg_2_nxt <= input when counter1 = "001" and flag1 = '1' else reg_2;
reg_3_nxt <= input when counter1 = "010" and flag1 = '1' else reg_3;
reg_4_nxt <= input when counter1 = "011" and flag1 = '1' else reg_4;
--Send the data ------------------------------------------
data_input <=   reg_1 when counter2 = "00" and flag2 = '1' else
              reg_2 when counter2 = "01" and flag2 = '1' else
              reg_3 when counter2 = "10" and flag2 = '1' else
              reg_4 when counter2 = "11" and flag2 = '1' else
              (others => '0');

--Flip Flop ----------------------------------------------
reg_01: FF
  generic map(N => 16)
  port map(   D     =>reg_1_nxt,
            Q     =>reg_1,
           clk     =>clk,
           reset   =>reset
      );

reg_02: FF
  generic map(N => 16)
  port map(   D     =>reg_2_nxt,
            Q     =>reg_2,
           clk     =>clk,
           reset   =>reset
      );

reg_03: FF
  generic map(N => 16)
  port map(   D     =>reg_3_nxt,
            Q     =>reg_3,
           clk     =>clk,
           reset   =>reset
      );

reg_04: FF
  generic map(N => 16)
  port map(   D     =>reg_4_nxt,
            Q     =>reg_4,
           clk     =>clk,
           reset   =>reset
      );
```

```
counter_01: FF
  generic map(N => 3)
  port map(   D     =>counter1_nxt,
              Q     =>counter1,
             clk    =>clk,
             reset  =>reset
      );

counter_02: FF
  generic map(N => 2)
  port map(   D     =>counter2_nxt,
              Q     =>counter2,
             clk    =>clk,
             reset  =>reset
      );

counter_03: FF
  generic map(N => 4)
  port map(   D     =>counter3_nxt,
              Q     =>counter3,
             clk    =>clk,
             reset  =>reset
      );

counter_04: FF
  generic map(N => 4)
  port map(   D     =>counter4_nxt,
              Q     =>counter4,
             clk    =>clk,
             reset  =>reset
      );

hold_time : FF
  generic map(N => 1)
  port map(   D     =>hold_nxt,
              Q     =>hold,
             clk    =>clk,
             reset  =>reset
      );

end Behavioral;
```

4.4 VHDL code for multiply

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

entity multiply is
  Port (
        clk, reset  : in std_logic;
        ----------------------------------------------------
        --signal from load_coeff
        multi_en    : in std_logic;
        --data in
        data_input  : in std_logic_vector(15 downto 0);
        data_coeff  : in std_logic_vector(15 downto 0);

        ----------------------------------------------------
        --ctrl out
        multi_done  : out std_logic;
        store_en    : out std_logic;
        max_en      : out std_logic;
        --data out
        data_out    : out std_logic_vector(17 downto 0)
        ----------------------------------------------------
    );
end multiply;

architecture Behavioral of multiply is

component ff is
  generic(N:integer:=1);
  port(   D  :  in std_logic_vector(N-1 downto 0);
          Q  :  out std_logic_vector(N-1 downto 0);
        clk  :  in std_logic;
        reset:  in std_logic
      );
end component;

type state_type is (s_initial, s_multi, s_add, s_send);--, s_wait);
signal state_reg, state_nxt : state_type;

signal input_1, input_2 : std_logic_vector(7 downto 0);
```

```
signal coeff_1, coeff_2 : std_logic_vector(6 downto 0);


signal result_1, result_2   : std_logic_vector(17 downto 0);


--the matrix is [14*8]*[8*14], when counting 111 in binary means one
number is done.
signal counter_8, counter_8_nxt     : std_logic_vector(1 downto 0) :=
(others => '0');
signal counter_14, counter_14_nxt   : std_logic_vector(3 downto 0) :=
(others => '0');
signal counter_6, counter_6_nxt     : std_logic_vector(2 downto 0) :=
(others => '0');
--------------------------------------------------------------------------------
signal data, data_nxt   : std_logic_vector(17 downto 0);


signal store, store_nxt : std_logic_vector(0 downto 0);


begin


--state contrl--------------------------------
process(clk, reset)
begin
    if reset = '1' then
        state_reg <= s_initial;
    elsif (clk'event and clk = '1') then
        state_reg <= state_nxt;
    end if;


end process;


--state machine-----------------------------------------
process(state_reg, multi_en, counter_8, data, result_1, result_2,
counter_14, counter_6)
begin
    multi_done <= '0';
    data_nxt <= (others => '0');
    counter_8_nxt <= (others => '0');
    counter_14_nxt <= (others => '0');
    counter_6_nxt <= (others => '0');
    store_nxt <= "0";
     input_1 <= (others => '0');
     input_2 <= (others => '0');
     coeff_1 <= (others => '0');
     coeff_2 <= (others => '0');
```

```
case state_reg is

   when s_initial =>
      if multi_en = '1' then
         state_nxt <= s_multi;
      else
         state_nxt <= s_initial;
      end if;

   when s_multi =>
      input_1 <= data_input(7 downto 0);--the input has 8 bits
      input_2 <= data_input(15 downto 8);
      coeff_1 <= data_coeff(6 downto 0);--the coeff only has 7 bits
      coeff_2 <= data_coeff(14 downto 8);

      if counter_8 = "00" then
         data_nxt <= (others => '0');
      else
         data_nxt <= data;
      end if;
      counter_8_nxt <= counter_8;
      counter_14_nxt <= counter_14;
      state_nxt <= s_add;


when s_add =>
   data_nxt <= result_1 + result_2 + data;
   input_1 <= data_input(7 downto 0);--the input has 8 bits
   input_2 <= data_input(15 downto 8);
   coeff_1 <= data_coeff(6 downto 0);--the coeff only has 7 bits
   coeff_2 <= data_coeff(14 downto 8);

   if counter_8 = "11" then
      counter_8_nxt <= (others => '0');
      store_nxt <= "1";--signal to store
      if counter_14 = "1101" then
         counter_14_nxt <= (others => '0');
         state_nxt <= s_send;
      else
         counter_14_nxt <= counter_14 + 1;
         state_nxt <= s_multi;
      end if;

   else
```

```
            store_nxt <= "0";--signal to store
            counter_14_nxt <= counter_14;
            counter_8_nxt <= counter_8 + 1;
            state_nxt <= s_multi;
          end if;

        when s_send =>
          multi_done <= '1';--feedback to controller & average
          state_nxt <= s_initial;

    end case;

end process;

--Computing part with two multipliers----------------------
result_1 <= input_1 * coeff_1 + "000000000000000000"; --to make they have the same digits
result_2 <= input_2 * coeff_2 + "000000000000000000";
--Send data-----------------------------------------------
store_en <= store(0);
max_en <= store(0);
data_out <= data when store = "1" else (others => '0');

--Flip Flop-----------------------------------------------
counter1: FF
  generic map(N => 2)
  port map(   D     =>counter_8_nxt,
              Q     =>counter_8,
             clk    =>clk,
             reset  =>reset
      );

counter2: FF
  generic map(N => 4)
  port map(   D     =>counter_14_nxt,
              Q     =>counter_14,
             clk    =>clk,
             reset  =>reset
      );

counter3: FF
  generic map(N => 3)
  port map(   D     =>counter_6_nxt,
              Q     =>counter_6,
```

```
            clk     =>clk,
            reset   =>reset
        );

add: FF
  generic map(N => 18)
  port map(   D     =>data_nxt,
              Q     =>data,
            clk     =>clk,
            reset   =>reset
        );

send_data: FF
  generic map(N => 1)
  port map(   D     =>store_nxt,
              Q     =>store,
            clk     =>clk,
            reset   =>reset
        );

end Behavioral;
```

4.5 VHDL code for store

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

entity store is
  Port (
        clk, reset  : in std_logic;
        store_en    : in std_logic;
        data_out    : in std_logic_vector(17 downto 0)
  );
end store;

architecture Behavioral of store is
component SRAM_store
  port (
    clk     : in  std_logic;          --Active Low
    we      : in std_logic;
    a       : in  std_logic_vector (8 downto 0);
    d       : in  std_logic_vector (31 downto 0);
    qspo    : out std_logic_vector (31 downto 0)
    );
end component;

component ff is
  generic(N:integer:=1);
  port(   D  :  in std_logic_vector(N-1 downto 0);
          Q  :  out std_logic_vector(N-1 downto 0);
        clk  :  in std_logic;
        reset:  in std_logic
      );
end component;

--SRAM-----------------------------------------
signal choose       : std_logic;
signal r_or_w       : std_logic;
signal address      : std_logic_vector(8 downto 0) := (others => '0');
signal RY_ram       : std_logic;
signal sram_out     : std_logic_vector(31 downto 0);
------------------------------------------------
signal address_nxt  : std_logic_vector(8 downto 0);
```

```
signal store_data_32 : std_logic_vector(31 downto 0);

begin
--SRAM bits transfer----------------------
store_data_32 <= "00000000000000" & data_out;
-----------------------------------------

Ram_store: SRAM_store
  port map(
    clk     => clk,
    we      => r_or_w,
    a       => address,
    d       => store_data_32,
    qspo    => sram_out
    );

address_nxt <= address + 1 when store_en = '1' else address;
r_or_w <= '1' when store_en = '1' else '0';
choose <= '0';

data_address: FF
  generic map(N => 9)
  port map(   D     =>address_nxt,
              Q     =>address,
             clk    =>clk,
             reset  =>reset
      );

end Behavioral;
```

4.6 VHDL code for max

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

entity max is
  Port (
        clk, reset  : in std_logic;
        ----------------------------------------------------
        --control signal from multiply
        max_en      : in std_logic;
        --data from multiply
        data_out    : in std_logic_vector(17 downto 0);
        ----------------------------------------------------
        --find the maximum data
        max_out     : out std_logic_vector(17 downto 0);
        clk_out     : out std_logic
        ----------------------------------------------------
  );
end max;

architecture Behavioral of max is
component ff is
  generic(N:integer:=1);
  port(   D  :  in std_logic_vector(N-1 downto 0);
          Q  :  out std_logic_vector(N-1 downto 0);
        clk  :  in std_logic;
        reset:  in std_logic
      );
end component;

signal count, count_nxt : std_logic_vector(6 downto 0);
signal data_max, data_max_nxt   : std_logic_vector(17 downto 0) :=
(others => '0');

begin
max_out <= data_max;
clk_out <= clk;
data_max_nxt <= data_out when data_max < data_out else data_max;

--Flip Flop----------------------------------------------
compare_data: FF
  generic map(N => 18)
```

```
  port map(   D     =>data_max_nxt,
              Q     =>data_max,
             clk    =>clk,
             reset   =>reset
      );

end Behavioral;
```

4.7 VHDL code for top

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

entity top is
  Port (
       --input
       clki           : in std_logic;
       reseti         : in std_logic;
       starti         : in std_logic;
       --output
       start_ld_inputo : out std_logic;
       start_ld_coeffo : out std_logic;
       max_outo        : out std_logic_vector(17 downto 0);
       clk_out         : out std_logic
  );
end top;

architecture Behavioral of top is

component controller is
  Port (
          clk, reset  : in std_logic;
          start       : in std_logic;
          ld_input_done   : in std_logic;
          multi_done  : in std_logic;
          ld_input    : out std_logic;
          op_en       : out std_logic
  );
end component;

component load_coeff is
  Port (
          clk, reset          : in std_logic;
          op_en               : in std_logic;
          multi_en            : out std_logic;
          data_coeff          : out std_logic_vector(15 downto 0)
  );
end component;

component load_input is
  Port (
```

```
        clk, reset  : in std_logic;
        ld_input    : in std_logic;
        ld_input_done   : out std_logic;--feedback to controller
        op_en       : in std_logic;
        start_ld_input  : out std_logic;
        data_input  : out std_logic_vector(15 downto 0)
  );
end component;

component multiply is
  Port (
        clk, reset  : in std_logic;
        multi_en    : in std_logic;
        data_input  : in std_logic_vector(15 downto 0);
        data_coeff  : in std_logic_vector(15 downto 0);
        multi_done  : out std_logic;
        store_en    : out std_logic;
        max_en      : out std_logic;
        data_out    : out std_logic_vector(17 downto 0)
    );
end component;

component store is
  Port (
        clk, reset  : in std_logic;
        store_en    : in std_logic;
        data_out    : in std_logic_vector(17 downto 0)
  );
end component;

component max is
  Port (
        clk, reset  : in std_logic;
        max_en      : in std_logic;
        data_out    : in std_logic_vector(17 downto 0);
        max_out     : out std_logic_vector(17 downto 0);
        clk_out     : out std_logic
  );
end component;

--controller
signal ld_input_done    : std_logic;
signal multi_done       : std_logic;
signal ld_input         : std_logic;
```

```
signal op_en           : std_logic;
--ld_coeff
signal multi_en          :  std_logic;
signal data_coeff        :  std_logic_vector(15 downto 0);
--ld_input
signal data_input     :  std_logic_vector(15 downto 0);
--multiply
signal store_en    :  std_logic;
signal data_out    :  std_logic_vector(17 downto 0);
--store
--max
signal max_en      : std_logic;


-----------------------------------------------------------
begin
--clk_out <= clki;
controller_part: controller
port map(
        clk          => clki         ,
        reset        => reseti       ,
        start        => starti       ,
        ld_input_done => ld_input_done  ,
        multi_done    => multi_done    ,
        ld_input      => ld_input      ,
        op_en         => op_en
);

coeff_part: load_coeff
port map(
        clk           => clki         ,
        reset         => reseti       ,
        op_en         => op_en        ,
        multi_en      => multi_en     ,
        data_coeff    => data_coeff
);

input_part: load_input
port map(
        clk           => clki         ,
        reset         => reseti       ,
        ld_input      => ld_input     ,
        ld_input_done  => ld_input_done  ,
        op_en         => op_en        ,
        start_ld_input  => start_ld_inputo    ,
```

```
        data_input      => data_input
);

multiply_part: multiply
port map(
        clk         => clki,
        reset       => reseti,
        multi_en    => multi_en,
        data_input => data_input,
        data_coeff => data_coeff,
        multi_done => multi_done,
        store_en => store_en,
        max_en    => max_en,
        data_out    => data_out
);

store_part: store
port map(
    clk         => clki,
    reset       => reseti,
    store_en    => store_en,
    data_out    => data_out
);

max_part: max
port map(
        clk      => clki,
        reset    => reseti,
        max_en   => max_en,
        data_out => data_out,
        max_out  => max_outo,
        clk_out  => clk_out
);

end Behavioral;
```