ETIN 35

IC project 1-2

Final Report

Hanyu Liu

Proofread: Xingda Li

January 11, 2023

Abstract

In this report, we designed a hardware accelerator that performs matrix multiplication and identifies the result with maximum value. The finite state machine was redesigned to extend the previous design to accommodate a larger matrix. To finish the design, four applications will be used during the project, Vivado for RTL design, Cadence Genus for synthesis, SoC Encounter for place and route, and ModelSim for netlist verification.

Contents

1. Introduction

The project aims to develop a 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 coefficients and results are each stored in a separate pair of SRAMs. The multiplier is constructed by using VHDL language. Then, synthesis and, placed and routed (PnR) will be completed by Cadence EDA tools.

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

where X is the element of the input matrix while A is the element of the coefficient matrix. The matrices of input and coefficient are specified in the following format.

$$Input = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} & x_{1,6} & x_{1,7} & x_{1,8} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} & x_{2,6} & x_{2,7} & x_{2,8} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} & x_{3,5} & x_{3,6} & x_{3,7} & x_{3,8} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} & x_{4,5} & x_{4,6} & x_{4,7} & x_{4,8} \\ x_{5,1} & x_{5,2} & x_{5,3} & x_{5,4} & x_{5,5} & x_{5,6} & x_{5,7} & x_{5,8} \\ x_{6,1} & x_{6,2} & x_{6,3} & x_{6,4} & x_{6,5} & x_{6,6} & x_{6,7} & x_{6,8} \\ x_{7,1} & x_{7,2} & x_{7,3} & x_{7,4} & x_{7,5} & x_{7,6} & x_{7,7} & x_{7,8} \\ x_{8,1} & x_{8,2} & x_{8,3} & x_{8,4} & x_{8,5} & x_{8,6} & x_{8,7} & x_{8,8} \\ x_{9,1} & x_{9,2} & x_{9,3} & x_{9,4} & x_{9,5} & x_{9,6} & x_{9,7} & x_{9,8} \\ x_{10,1} & x_{10,2} & x_{10,3} & x_{10,4} & x_{10,5} & x_{10,6} & x_{10,7} & x_{10,8} \\ x_{11,1} & x_{11,2} & x_{11,3} & x_{11,4} & x_{11,5} & x_{11,6} & x_{11,7} & x_{11,8} \\ x_{12,1} & x_{12,2} & x_{12,3} & x_{12,4} & x_{12,5} & x_{12,6} & x_{12,7} & x_{12,8} \\ x_{13,1} & x_{13,2} & x_{13,3} & x_{13,4} & x_{13,5} & x_{13,6} & x_{13,7} & x_{13,8} \\ x_{14,1} & x_{14,2} & x_{14,3} & x_{14,4} & x_{14,5} & x_{14,6} & x_{14,7} & x_{14,8} \end{bmatrix}$$

$$Coeff = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} & a_{1,9} & a_{1,10} & a_{1,11} & a_{1,12} & a_{1,13} & a_{1,14} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} & a_{2,9} & a_{2,10} & a_{2,11} & a_{2,12} & a_{2,13} & a_{2,14} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} & a_{3,9} & a_{3,10} & a_{3,11} & a_{3,12} & a_{3,13} & a_{3,14} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} & a_{4,9} & a_{4,10} & a_{4,11} & a_{4,12} & a_{4,13} & a_{4,14} \\ a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} & a_{5,11} & a_{5,12} & a_{5,13} & a_{5,14} \\ a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} & a_{6,11} & a_{6,12} & a_{6,13} & a_{6,14} \\ a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} & a_{7,11} & a_{7,12} & a_{7,13} & a_{7,14} \\ a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} & a_{8,11} & a_{8,12} & a_{8,13} & a_{8,14} \end{bmatrix}$$

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.

2. Improvements

The task for this project is very similar to the one for the first IC project but aims to engage with much larger matrixes. At the same time, the design also should be optimized in terms of area and power efficiency. In the previous project, the coefficient matrix is stored in an SRAM beforehand and then both the coefficient matrix and input matrix were loaded to registers, which is quite costly in terms of area as it needs 32 8-bit registers and 32 7-bit registers to hold these values.

In the new design, only 2 7-bit registers and 8 8-bit registers are assigned to the loading of coefficients and inputs, respectively. Therefore, the number of registers can be significantly reduced, which helps save area as they are quite expensive in terms of area.

The data path of the design has also been optimized to reduce the area further ahead. In the previous design, some registers were located in the module “controller” to temporarily hold the results before storing them in the SRAM. This is totally unnecessary as the results can be passed directly to the SRAM with the help of the module “store” which can generate the corresponding addresses.

3. 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. The principle of the matrix multiplier can be understood in this section.

3.1 Block diagram

The block diagram of the top module is shown in Figure 3.1. This matrix multiplier can be divided into 8 modules, which are “load_coeff”, “load_input”, “multiply”, “store”, “max”, “controller”, and two SRAMs. There are three inputs and three outputs for the top module, “input”, “start”, “restart”, “start_ld_coeff”, “start_ld_input” and “max_out”. The functionality of each module and port is given as follows.

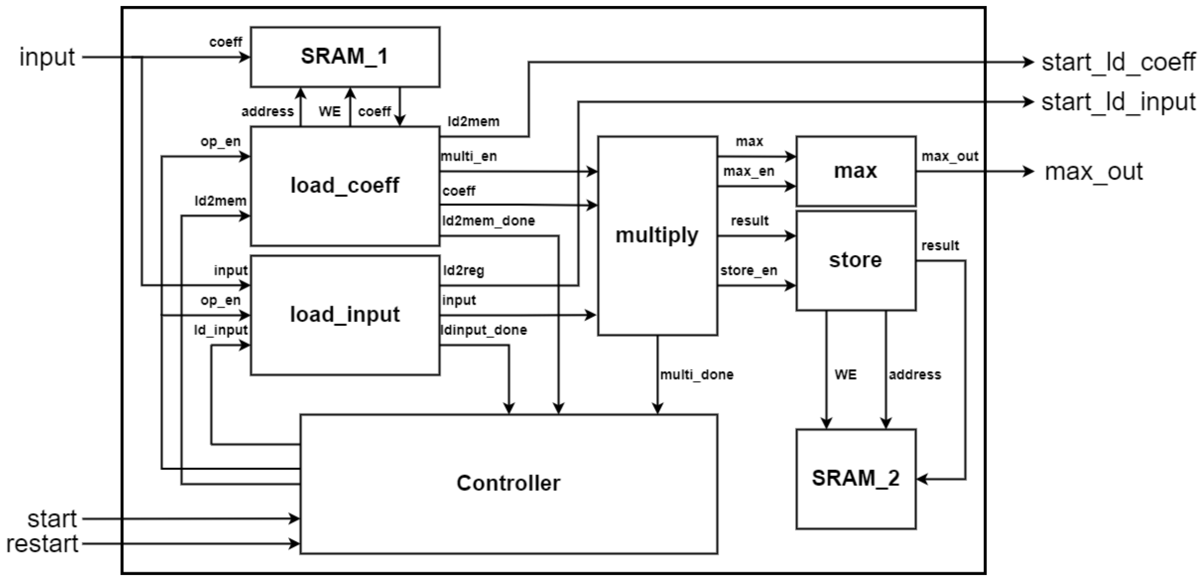


Figure 3.1: The block diagram of the matrix multiplier

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_1” is a storage for the coefficient matrix, while “load_input” can be a temporary storage for the input matrix. Both of them get data through “input” which is a 16-bit input port that can read either 2 coefficients or 2 inputs in one clock cycle. “load_coeff” is similar to “load_input” but it generates an address for SRAM_1 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 to “SRAM_2” with the help of “store” and the maximum result found for the time being is placed in “max”.

3.2 ASMD charts

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

3.2.1 Controller

The ASMD chart of the “controller” is shown in Figure 3.2.1. There are four states in this module.

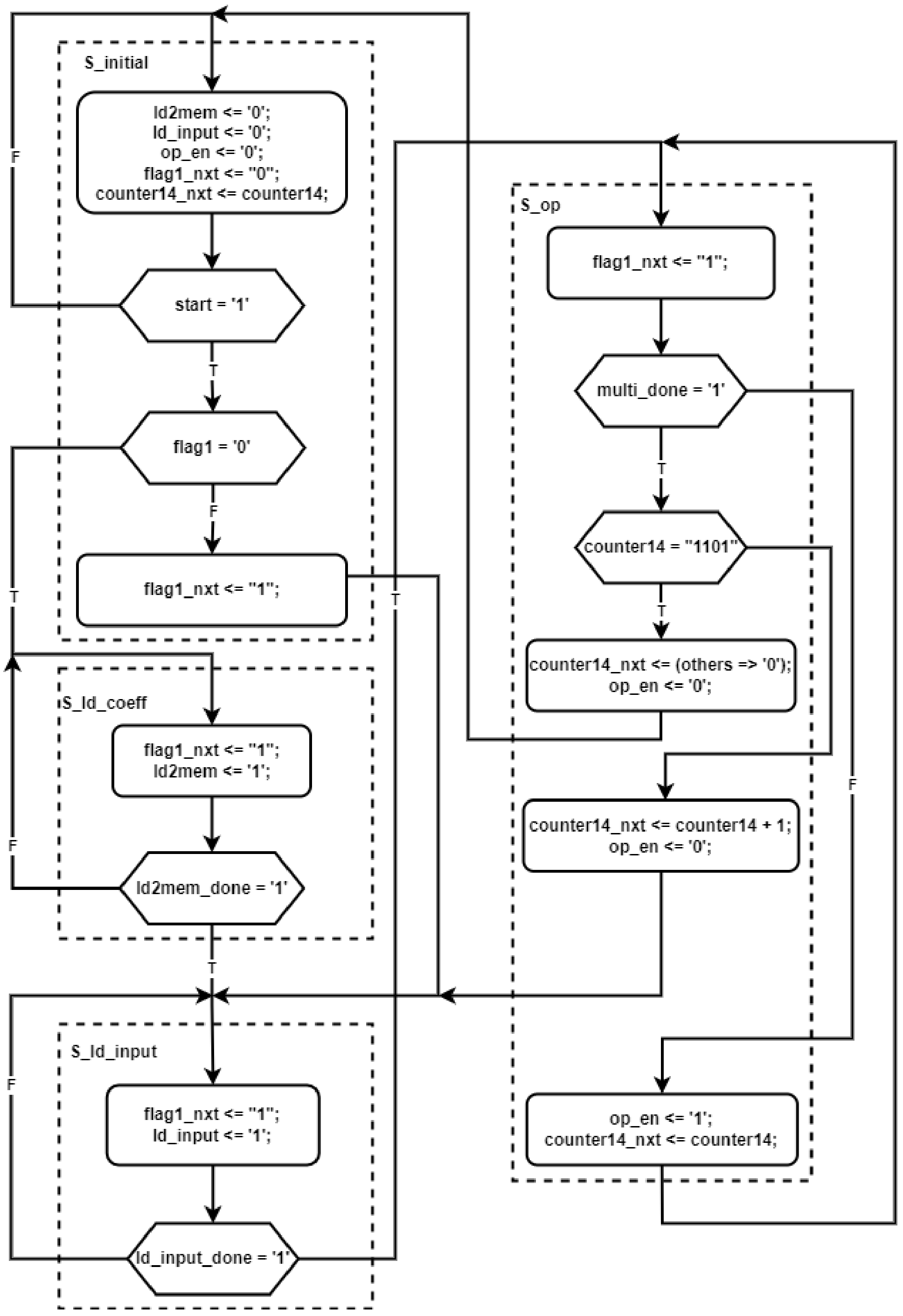


Figure 3.2.1: The ASMD chart of the “controller”

S_initial:

In this state, all signals are set to default values. Signal “flag1” is a 1-bit signal which indicates whether the coefficients have been placed in SRAM_1. For instance, when all coefficients are written to SRAM_1, “flag_1” will then be set to HIGH, during which state “s_ld_coeff” will be skipped. However, flag1 will be forced to LOW when the signal “restart” is asserted, which allows us to overwrite the coefficients.

S_ld_coeff:

The coefficients are fed to the SRAM in this state, where two coefficients share one address from a txt file. To store 114 coefficients, this state will iterate 56 times. The signal “ld2mem_done” will be set to HIGH after all coefficients are in place.

S_ld_input:

In this state, 8 inputs will be loaded from a txt file and stored in registers. After that, the signal “ldinput_done” will be set to HIGH, which indicates that the system is ready to produce one result. To complete a whole matrix multiplication, this process will be repeated for 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.

3.2.2 Load coefficient

The ASMD chart of "load_coeff" is shown in Figure 3.2.2. There are four states in this module.

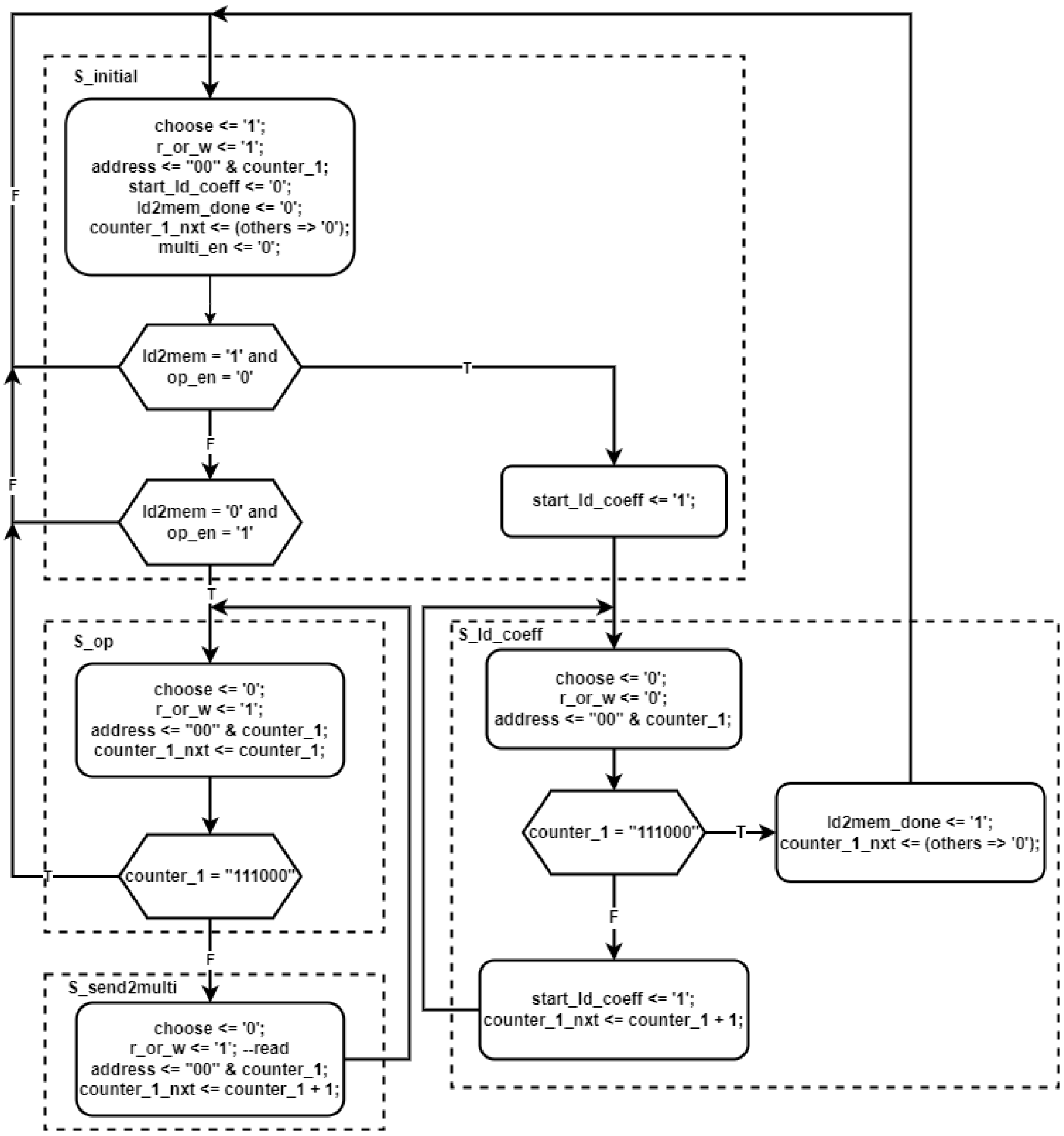

S_initial
choose <= '1';
r_or_w <= '1';
address <= "00" & counter_1;
start_ld_coeff <= '0';
ld2mem_done <= '0';
counter_1_nxt <= (others => '0');
multi_en <= '0';
ld2mem = '1' and op_en = '0'
ld2mem = '0' and op_en = '1'
start_ld_coeff <= '1';
S_op
choose <= '0';
r_or_w <= '1';
address <= "00" & counter_1;
counter_1_nxt <= counter_1;
counter_1 = "111000"
S_send2multi
choose <= '0';
r_or_w <= '1'; --read
address <= "00" & counter_1;
counter_1_nxt <= counter_1 + 1;
S_ld_coeff
choose <= '0';
r_or_w <= '0';
address <= "00" & counter_1;
counter_1 = "111000"
ld2mem_done <= '1';
counter_1_nxt <= (others => '0');
start_ld_coeff <= '1';
counter_1_nxt <= counter_1 + 1;
T
F

Figure 3.2.2: The ASMD chart of "load_coefficient"

S_initial:

In this state, all variables will be set to default values. To avoid unnecessary coefficient overwriting, the state "s_ld_coeff" will only be accessed when signal "ld2mem" is HIGH and signal "op_en" is LOW. Otherwise, the state machine will enter the state "s_op" when signal "ld2mem" is LOW, and signal "op_en" is HIGH.

S_ld_coeff:

In this state, 112 coefficients will be loaded into SRAM. As each entry of the SRAM can store two coefficients, the state will be iterated 56 times to store all coefficients. Signal “ld2mem_done” is set to HIGH after all coefficients have been written into SRAM.

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.

3.2.3 Load input

The ASMD chart of "load_input" is shown in Figure 3.2.3. There are five states in this module.

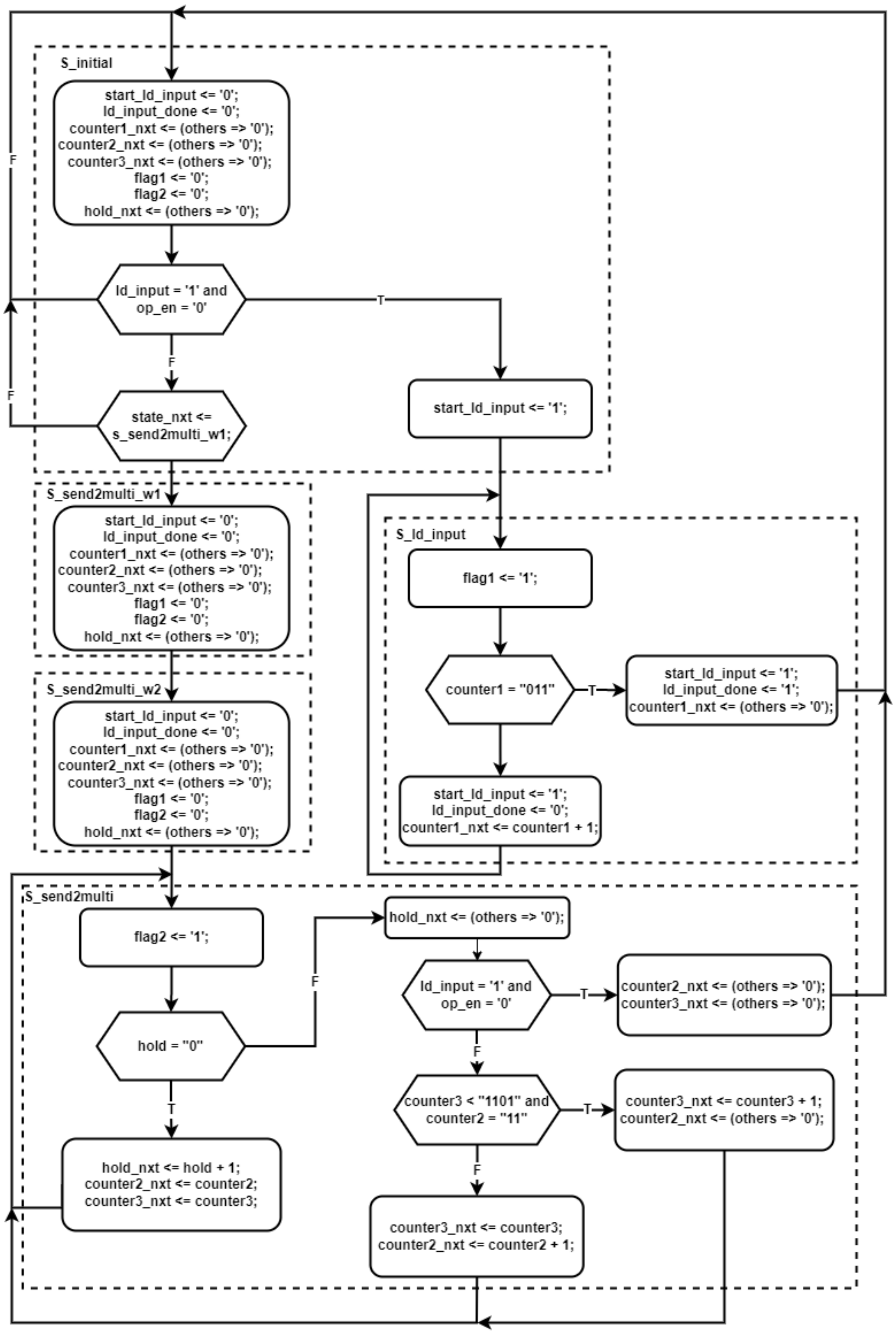

S_initial
start_ld_input <= '0';
ld_input_done <= '0';
counter1_nxt <= (others => '0');
counter2_nxt <= (others => '0');
counter3_nxt <= (others => '0');
flag1 <= '0';
flag2 <= '0';
hold_nxt <= (others => '0');
ld_input = '1' and op_en = '0'
T
F
start_ld_input <= '1';
state_nxt <= s_send2multi_w1;
F
S_send2multi_w1
S_send2multi_w2
S_ld_input
flag1 <= '1';
counter1 = "011"
start_ld_input <= '1'; ld_input_done <= '1'; counter1_nxt <= (others => '0');
start_ld_input <= '1'; ld_input_done <= '0'; counter1_nxt <= counter1 + 1;
S_send2multi
flag2 <= '1';
hold = "0"
hold_nxt <= hold + 1; counter2_nxt <= counter2; counter3_nxt <= counter3;
hold_nxt <= (others => '0');
counter2_nxt <= (others => '0'); counter3_nxt <= (others => '0');
counter3 < "1101" and counter2 = "11"
counter3_nxt <= counter3 + 1; counter2_nxt <= (others => '0');
counter3_nxt <= counter3; counter2_nxt <= counter2 + 1;

Figure 3.2.3: 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" 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_send2multi_w1, S_send2multi_w2:

These two states will wait for 2 clock cycles to ensure that the coefficients have been received.

S_ld_input:

In this state, 8 inputs will be loaded from a txt file and stored in registers. Signal “ld_input_done” will be set to HIGH after finishing storing.

S_send2multi:

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

3.2.4 Multiply

The ASMD chart of “multiply” is shown in Figure 3.2.4. There are four states in this module.

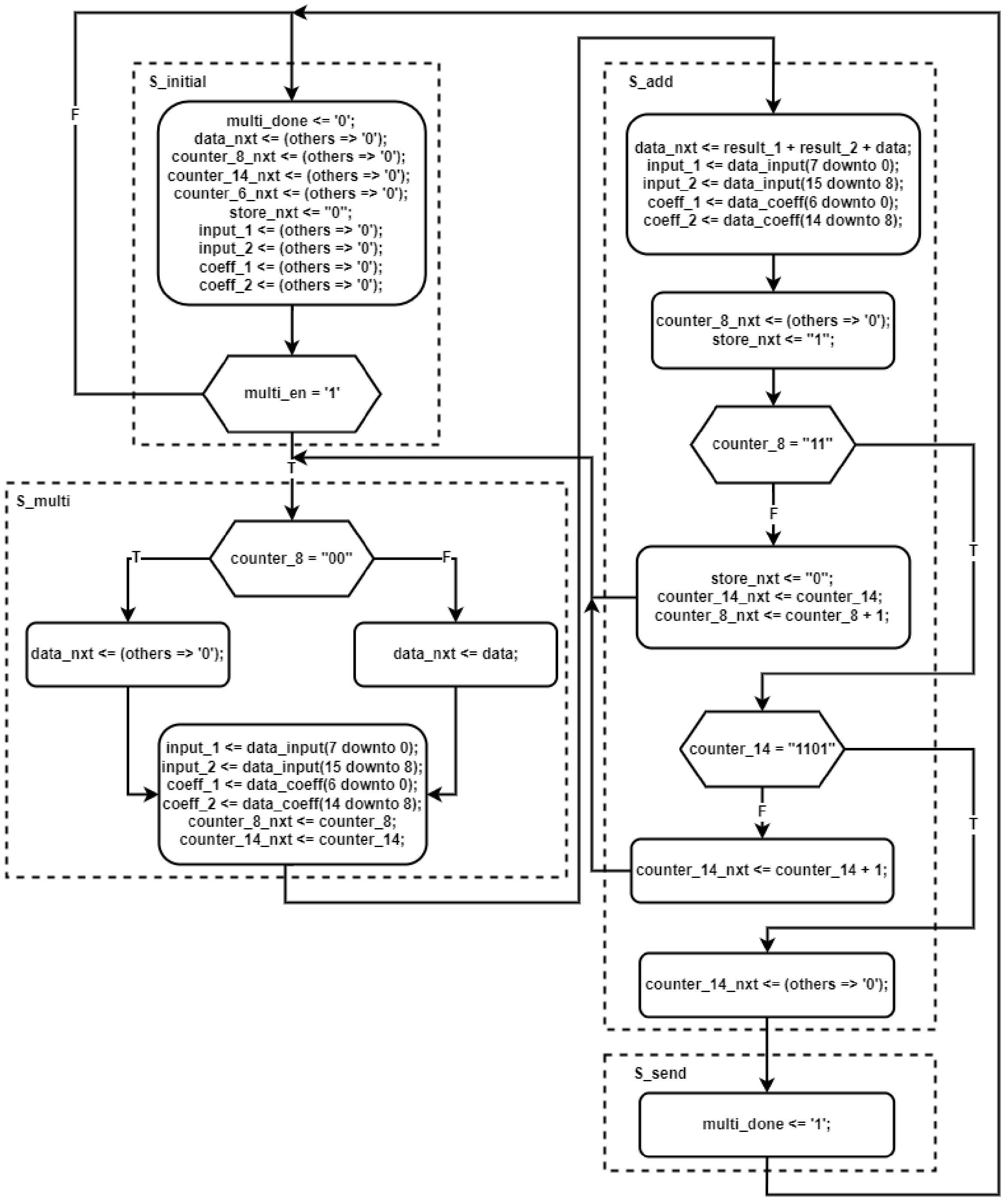

S_initial
multi_done <= '0';
data_nxt <= (others => '0');
counter_8_nxt <= (others => '0');
counter_14_nxt <= (others => '0');
counter_6_nxt <= (others => '0');
store_nxt <= "0";
input_1 <= (others => '0');
input_2 <= (others => '0');
coeff_1 <= (others => '0');
coeff_2 <= (others => '0');
multi_en = '1'
F
T
S_multi
counter_8 = "00"
T
F
data_nxt <= (others => '0');
data_nxt <= data;
input_1 <= data_input(7 downto 0);
input_2 <= data_input(15 downto 8);
coeff_1 <= data_coeff(6 downto 0);
coeff_2 <= data_coeff(14 downto 8);
counter_8_nxt <= counter_8;
counter_14_nxt <= counter_14;
S_add
data_nxt <= result_1 + result_2 + data;
input_1 <= data_input(7 downto 0);
input_2 <= data_input(15 downto 8);
coeff_1 <= data_coeff(6 downto 0);
coeff_2 <= data_coeff(14 downto 8);
counter_8_nxt <= (others => '0');
store_nxt <= "1";
counter_8 = "11"
F
T
store_nxt <= "0";
counter_14_nxt <= counter_14;
counter_8_nxt <= counter_8 + 1;
counter_14 = "1101"
F
T
counter_14_nxt <= counter_14 + 1;
counter_14_nxt <= (others => '0');
S_send
multi_done <= '1';

Figure 3.2.4: 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.

4. Synthesis

To reduce power consumption, 65LPHVT libraries are used because high Vt can reduce leakage current to reduce static power consumption. The clock is defined as shown in table 4.1. And the timing report and area report is shown in Tables 4.2 and 4.3, respectively.

Clock Period	10ns (100MHz)
Clock Skew	500ps
Clock Rise Time	100ps
Clock Fall Time	110ps
Clock Source Latency	500ps
Clock Network Latency	500ps

Table 4.1: Clock parameters and constraints

Slack	6659ps
Critical Path	3341ps

Table 4.2: Timing report

Total area(um^2)	204023

Table 4.3: Area report

Considering the slack is 6659ps (Table 4.2), which occupies 66.6% of the clock period, there is no need to reduce the clock frequency even if the slack will decrease when doing place and route.

5. Verification

The design was verified in Vivado, and the simulation results are shown in Figure 5.1 and Figure 5.2. Post-synthesis simulation was performed in ModelSim to examine the design further, and the result is shown in Figure 5.3. Figure 5.3 shows the results 60, 140 and 80 in decimal can be stored in the memory. Figure 5.4 shows 14 operations after the "start" is asserted. The evidence above indicates that the design can complete the whole matrix multiplication without a problem.

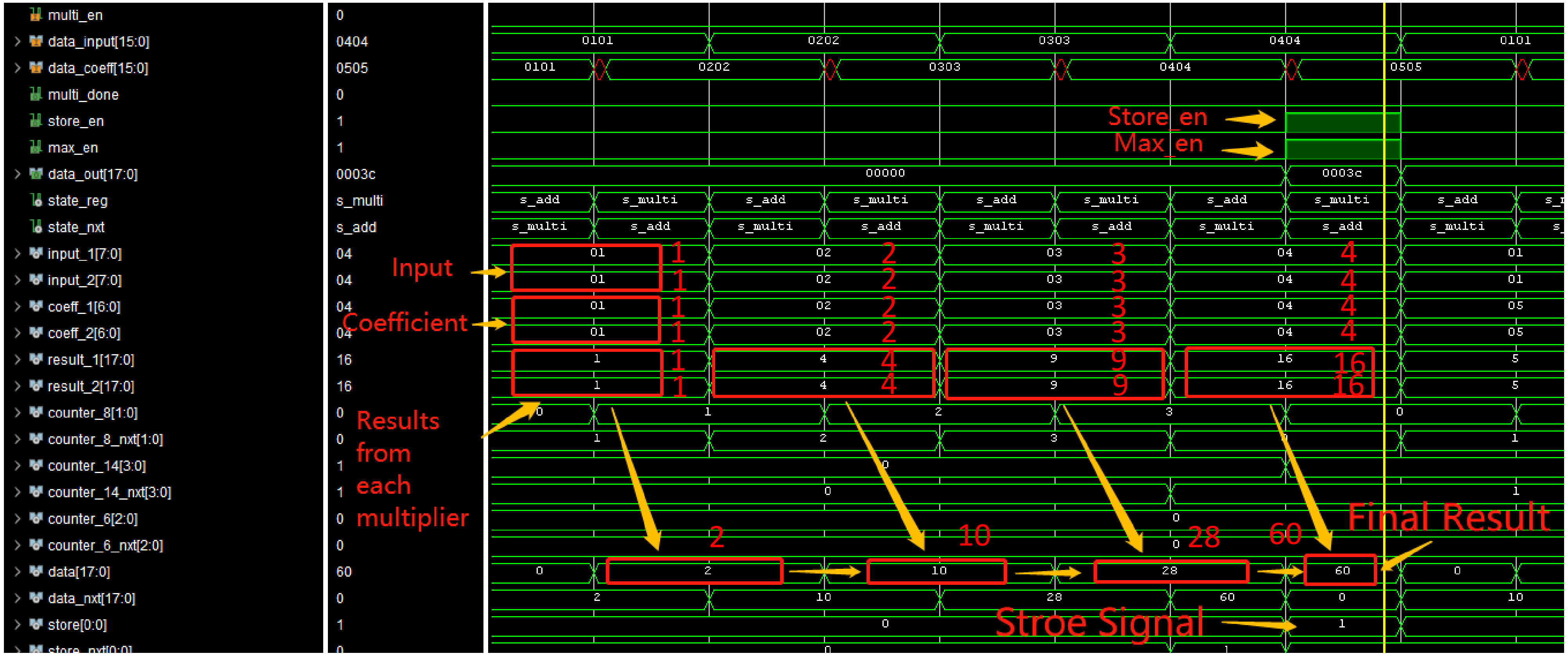


Figure 5.1: Simulation result from Vivado

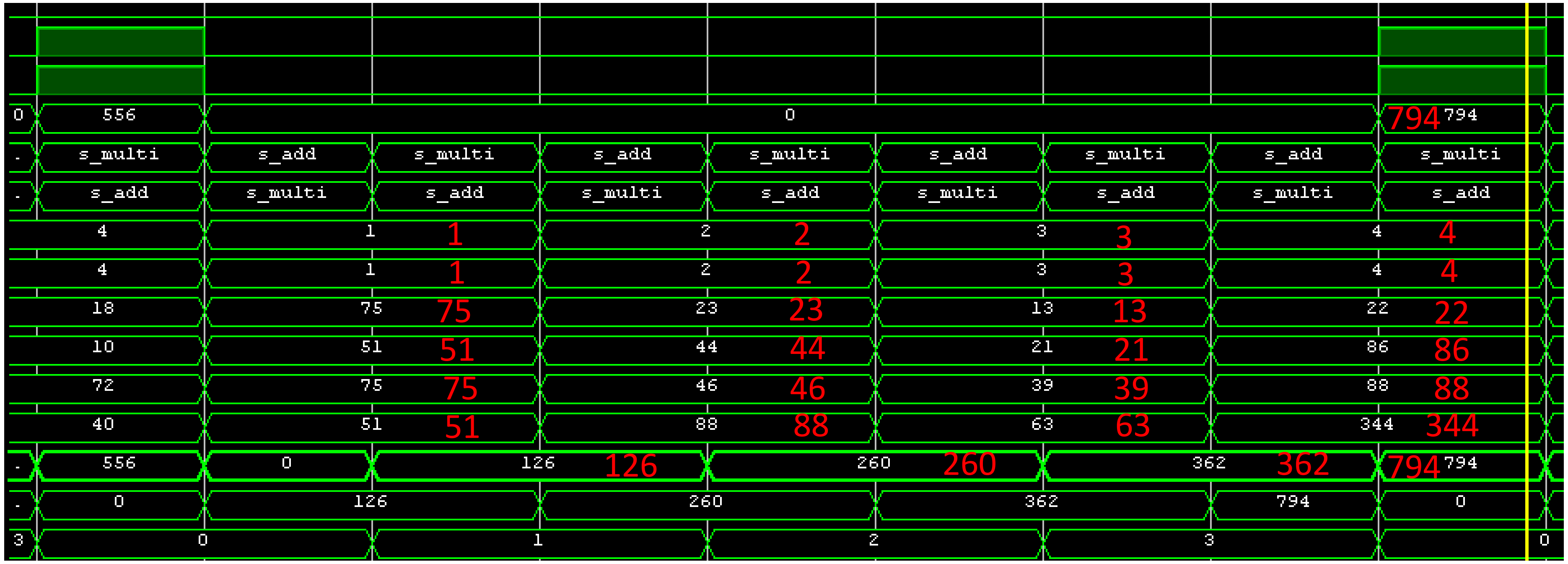


Figure 5.2: Another group of simulation result from Vivado

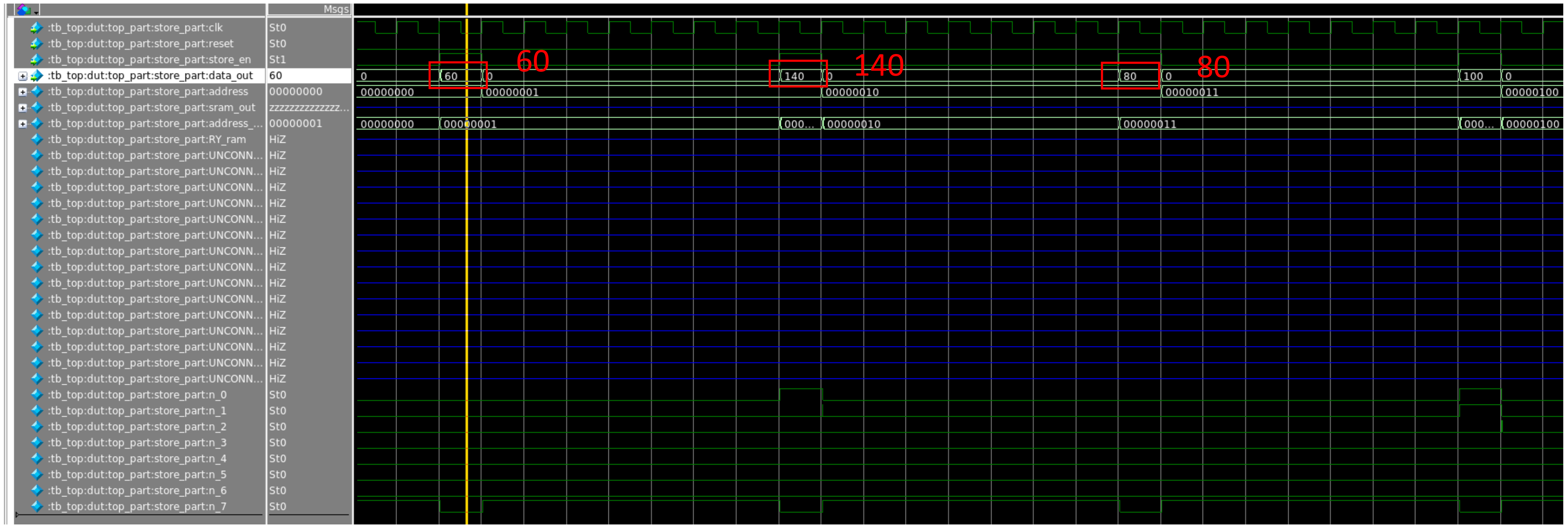

Msgs
:tb_top:dut:top_part:store_part:clk St0
:tb_top:dut:top_part:store_part:reset St0
:tb_top:dut:top_part:store_part:store_en St1
:tb_top:dut:top_part:store_part:data_out 60
:tb_top:dut:top_part:store_part:address 00000000
:tb_top:dut:top_part:store_part:sram_out zzzzzzzzzzzzzz...
:tb_top:dut:top_part:store_part:address_... 00000001
:tb_top:dut:top_part:store_part:RY_ram HiZ
:tb_top:dut:top_part:store_part:UNCONN... HiZ
:tb_top:dut:top_part:store_part:n_0 St0
:tb_top:dut:top_part:store_part:n_1 St0
:tb_top:dut:top_part:store_part:n_2 St0
:tb_top:dut:top_part:store_part:n_3 St0
:tb_top:dut:top_part:store_part:n_4 St0
:tb_top:dut:top_part:store_part:n_5 St0
:tb_top:dut:top_part:store_part:n_6 St0
:tb_top:dut:top_part:store_part:n_7 St0
60
140
80
100
00000001
00000010
00000011
00000100

Figure 5.3: The results in decimal stored in memory successfully during post-synthesis.

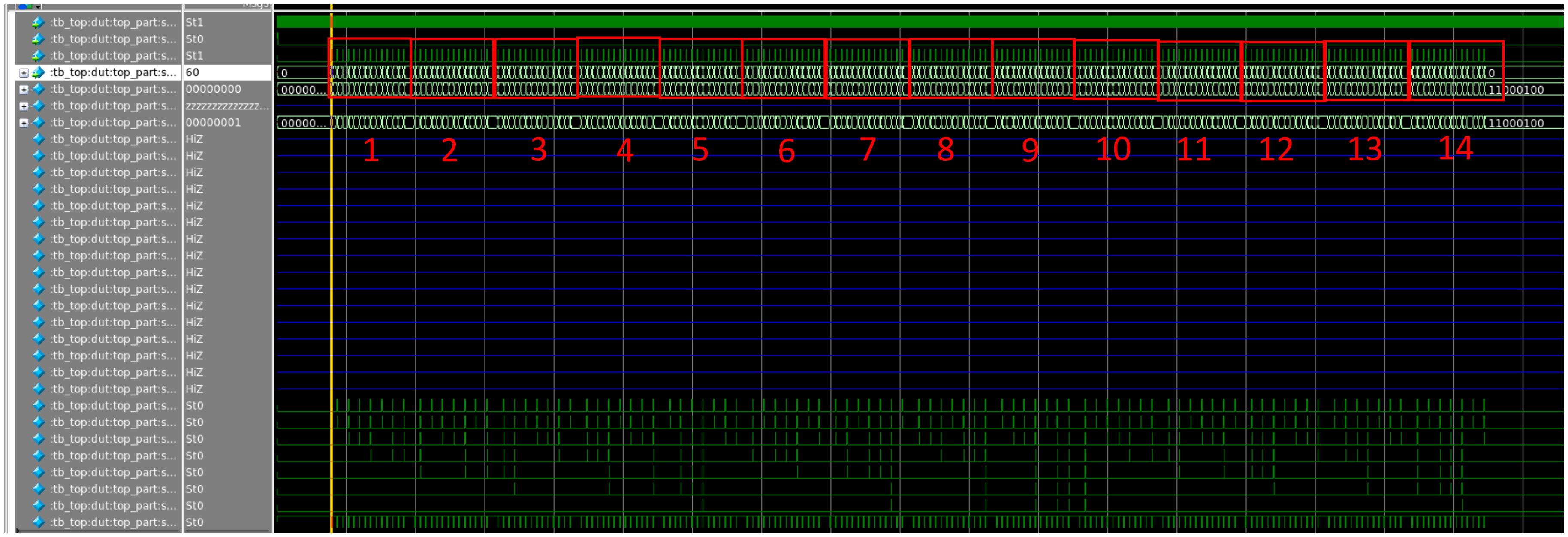

:tb_top:dut:top_part:s... St1
:tb_top:dut:top_part:s... St0
:tb_top:dut:top_part:s... St1
:tb_top:dut:top_part:s... 60
:tb_top:dut:top_part:s... 00000000
:tb_top:dut:top_part:s... zzzzzzzzzzzzzz...
:tb_top:dut:top_part:s... 00000001
:tb_top:dut:top_part:s... HiZ
:tb_top:dut:top_part:s... St0
0
00000...
11000100
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Figure 5.4: The operation has been executed for 14 times.

6. Place and route

In this section, the layout of matrix multiplier design needs to be used by Encounter and some data are needed.

LEF: Library Exchange Format

• header.lef

• standardCell.lef: Cell Library

• IO.lef: Pad Library

• memory.lef: custom

lib/tlf: libraries that contain timing information

sdc: Synopsys Design Constraint (generated during synthesis). Optional

Memory: memory.lib

Design (netlist): top.v

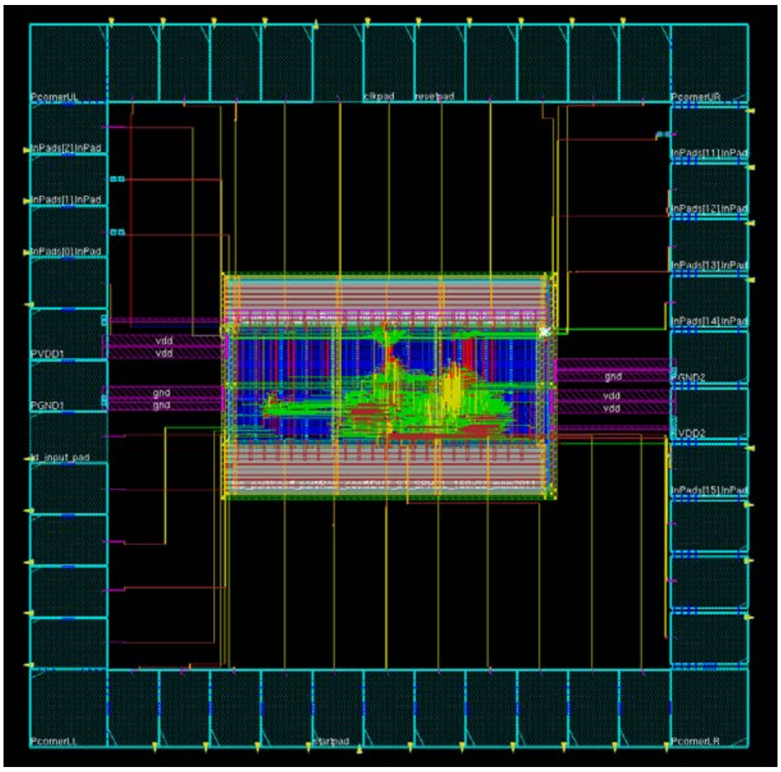


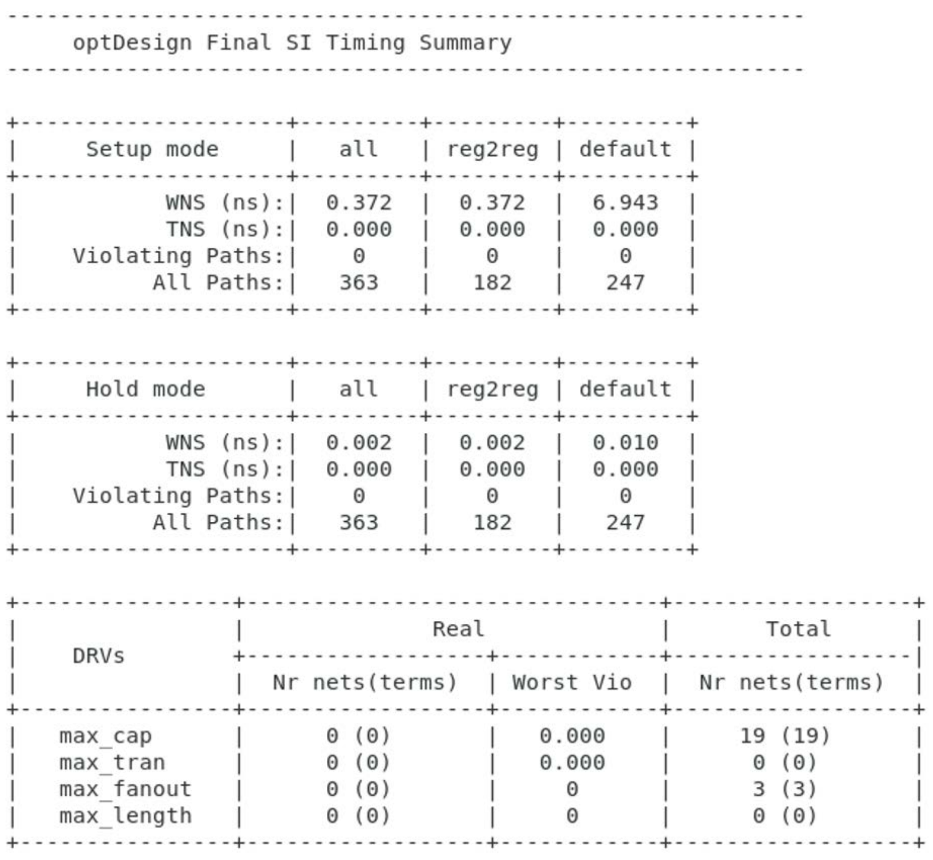
Figure 6.1: The layout of the matrix multiplier

The scripts need to be added before the place and route. The layout of the matrix multiplier is shown in figure 6.1. And the total area of the design after PnR is shown in table 6.1.

Total Area (um^2)	26671

Table 6.1: Area report after optimization

After the place and route, we need to optimize our design and fulfil the setup time and hold time requirements to avoid timing issues. As shown in Figure 6.2, the setup time is 0.372ns and the hold time is 0.002ns, which meets the requirements.



```
------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.372  |  0.372  |  6.943  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   363   |   182   |   247   |
+--------------------+---------+---------+---------+

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.002  |  0.002  |  0.010  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   363   |   182   |   247   |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |     19 (19)      |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      3 (3)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+
```

Figure 6.2: Timing report after optimization

7. Appendix

7.1 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;
            ld2mem_done : in std_logic;
            ld_input_done   : in std_logic;
            multi_done  : in std_logic;
            ld2mem      : out 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_coeff, s_ld_input, s_op);
signal state_reg, state_nxt : state_type;
signal flag1, flag1_nxt : std_logic_vector(0 downto 0) := (others => '0');
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, flag1, ld2mem_done, ld_input_done, multi_done,
counter14)
begin

    ld2mem <= '0';
    ld_input <= '0';
    op_en <= '0';
    flag1_nxt <= "0";
    counter14_nxt <= counter14;
    case state_reg is
        when s_initial =>
        counter14_nxt <= (others => '0');
            if start = '1' and flag1(0) = '0' then
               state_nxt <= s_ld_coeff;
            elsif start = '0' and flag1(0) = '0' then
                state_nxt <= s_initial;
            elsif start = '1' and flag1(0) = '1' then
               flag1_nxt <= "1";
               state_nxt <= s_ld_input;
            else
               flag1_nxt <= "1";
               state_nxt <= s_initial;
            end if;

        when s_ld_coeff =>
            flag1_nxt <= "1";
            ld2mem <= '1';
            if ld2mem_done = '1' then
                state_nxt <= s_ld_input;
            else
                state_nxt <= s_ld_coeff;
            end if;


        when s_ld_input =>
            flag1_nxt <= "1";
            ld_input <= '1';
            if ld_input_done = '1' then
                state_nxt <= s_op;
            else
                state_nxt <= s_ld_input;
            end if;

        when s_op =>
            flag1_nxt <= "1";
            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;

    flag_01: FF
        generic map(N => 1)
        port map(   D     =>flag1_nxt,
                    Q     =>flag1,
            clk    =>clk,
            reset  =>reset
      );

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

7.2 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;
            --load coeff part ------------------------------------
            --signal from controller
            ld2mem      : in std_logic;
            --read data
            coeff       : in std_logic_vector(15 downto 0);
            --control signal
            start_ld_coeff    : out std_logic;
            --feedback to controller
            ld2mem_done : out std_logic;
            --coeff to memory
            --coeff2mem   : out std_logic_vector(15 downto 0);
            ---------------------------------------------------------
            --op part ---------------------------------------------
            --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_SP_WRAPPER
  port (
    ClkxCI  : in  std_logic;
    CSxSI   : in  std_logic;            -- Active Low
    WExSI   : in  std_logic;            --Active Low
    AddrxDI : in  std_logic_vector (7 downto 0);
    RYxSO   : out std_logic;
    DataxDI : in  std_logic_vector (31 downto 0);
```

```
   DataxDO : 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; -- Active Low (reand & write) --write '0' --
read '1'
signal address      : std_logic_vector(7 downto 0);
signal RY_ram       : std_logic;
---------------------------------------------------


type state_type is (s_initial, s_ld_coeff, 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(31 downto 0);
signal data_coeff_32 : std_logic_vector(31 downto 0);


begin
--SRAM bits transfer----------------------
coeff_32 <= "0000000000000000" & coeff;
data_coeff <= data_coeff_32(15 downto 0);
-----------------------------------------


Ram_coeff: SRAM_SP_WRAPPER
port map(
   ClkxCI          => clk          ,
   CSxSI           => choose       , -- Active Low
   WExSI           => r_or_w       , -- Active Low
   AddrxDI         => address      ,
   RYxSO           => RY_ram       ,
   DataxDI         => coeff_32     ,
    DataxDO          => 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, ld2mem, op_en, counter_1)
begin

    --SRAM-----------------------------
    choose <= '1';
    r_or_w <= '1';--read
    address <= "00" & counter_1;
------------------------------------

start_ld_coeff <= '0';
ld2mem_done <= '0';

counter_1_nxt <= (others => '0');
multi_en <= '0';

case state_reg is

    when s_initial =>
        if ld2mem = '1' and op_en = '0' then
            start_ld_coeff <= '1';
            state_nxt <= s_ld_coeff;
        elsif ld2mem = '0' and op_en = '1' then
            state_nxt <= s_op;
        else
            state_nxt <= s_initial;
        end if;

    when s_ld_coeff =>
        choose <= '0';
    r_or_w <= '0'; --write
    address <= "00" & counter_1;
    if counter_1 = "111000" then
        ld2mem_done <= '1';
        counter_1_nxt <= (others => '0');
        state_nxt <= s_initial;
    else
        start_ld_coeff <= '1';
        counter_1_nxt <= counter_1 + 1;
        state_nxt <= s_ld_coeff;
    end if;

when s_op =>
    choose <= '0';
    r_or_w <= '1'; --read
    address <= "00" & 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 <= '1'; --read
    address <= "00" & 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;
```

7.3 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;
            input       : in std_logic_vector(15 downto 0);
            ld_input_done   : out std_logic;--feedback to controller

            --op part ------------------------------------------
            --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 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_send2multi, s_send2multi_w1,
s_send2multi_w2);
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');

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, ld_input, op_en, counter1, counter2, counter3, hold)
begin
   start_ld_input <= '0';
   ld_input_done <= '0';
   counter1_nxt <= (others => '0');
   counter2_nxt <= (others => '0');
   counter3_nxt <= (others => '0');
   flag1 <= '0';
   flag2 <= '0';
   hold_nxt <= (others => '0');

   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;
    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 =>


    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 + 1;
        state_nxt <= s_ld_input;
    end if;

when s_send2multi_w1 =>
    state_nxt <= s_send2multi_w2;

when s_send2multi_w2 =>
    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
      );

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

7.4 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;
```

7.5 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_SP_WRAPPER
  port (
    ClkxCI  : in  std_logic;
    CSxSI   : in  std_logic;            -- Active Low
    WExSI   : in  std_logic;            --Active Low
    AddrxDI : in  std_logic_vector (7 downto 0);
    RYxSO   : out std_logic;
    DataxDI : in  std_logic_vector (31 downto 0);
    DataxDO : 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; -- Active Low (reand & write) --write '0' --
read '1'
signal address      : std_logic_vector(7 downto 0) := (others => '0');
signal RY_ram       : std_logic;
signal sram_out     : std_logic_vector(31 downto 0);
---------------------------------------------------
signal address_nxt  : std_logic_vector(7 downto 0);
--signal address_sel  : std_logic_vector(7 downto 0);
signal store_data_32 : std_logic_vector(31 downto 0);

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


Ram_coeff: SRAM_SP_WRAPPER
port map(
   ClkxCI          => clk          ,
   CSxSI           => choose        , -- Active Low
   WExSI            => r_or_w        , -- Active Low --only write in this
module
   AddrxDI          => address       ,
   RYxSO            => RY_ram         ,
   DataxDI          => store_data_32   ,
   DataxDO           => sram_out
   );

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

data_address: FF
  generic map(N => 8)
  port map(   D     =>address_nxt,
              Q     =>address,
            clk     =>clk,
            reset   =>reset
      );

end Behavioral;
```

7.6 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)
        ----------------------------------------------------
  );
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

process(max_en)
begin
    if max_en = '1' then
        max_out <= data_max;
    else
```

```
        max_out <= (others => '0');
    end if;

end process;

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;
```

7.7 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;
        inputi          : in std_logic_vector(15 downto 0);
        --output
        start_ld_inputo : out std_logic;
        start_ld_coeffo : out std_logic;
        max_outo        : out std_logic_vector(17 downto 0)
  );
end top;

architecture Behavioral of top is
```

```
component controller is
  Port (
           clk, reset  : in std_logic;
           start       : in std_logic;
           ld2mem_done : in std_logic;
           ld_input_done   : in std_logic;
           multi_done  : in std_logic;
           ld2mem      : out std_logic;
           ld_input    : out std_logic;
           op_en       : out std_logic
  );
end component;

component load_coeff is
  Port (
           clk, reset          : in std_logic;
           ld2mem               : in std_logic;
           coeff               : in std_logic_vector(15 downto 0);
           start_ld_coeff      : out std_logic;
           ld2mem_done         : out 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;
            input       : in std_logic_vector(15 downto 0);
            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)
  );
end component;

--controller
signal ld2mem_done      : std_logic;
signal ld_input_done    : std_logic;
signal multi_done       : std_logic;
signal ld2mem           : std_logic;
signal ld_input         : std_logic;
signal op_en            : std_logic;

--ld_coeff
--signal coeff              : std_logic_vector(15 downto 0);
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


controller_part: controller
port map(
            clk           => clki         ,
            reset         => reseti        ,
            start         => starti        ,
            ld2mem_done   => ld2mem_done  ,
            ld_input_done => ld_input_done  ,
            multi_done    => multi_done    ,
            ld2mem        => ld2mem        ,
            ld_input      => ld_input      ,
            op_en         => op_en
);

coeff_part: load_coeff
```

```
port map(
        clk           => clki         ,
        reset         => reseti        ,
        ld2mem         => ld2mem       ,
        coeff          => inputi       ,
        start_ld_coeff  => start_ld_coeffo,
        ld2mem_done     => ld2mem_done  ,
        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     ,
        input          => inputi       ,
        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
);

end Behavioral;
```

7.8 VHDL code for top_top

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

entity top_top is
Port (
        --input
        clk          : in std_logic;
        reset        : in std_logic;
        start        : in std_logic;
        input        : in std_logic_vector(15 downto 0);
        --output
        start_ld_input : out std_logic;
        start_ld_coeff : out std_logic;
        max_out       : out std_logic_vector(17 downto 0)
  );

end top_top;

architecture Behavioral of top_top is

component CPAD_S_74x50u_IN          --input PAD
    port (
      COREIO : out std_logic;
      PADIO  : in  std_logic);
  end component;

  component CPAD_S_74x50u_OUT          --output PAD
    port (
      COREIO : in  std_logic;
      PADIO  : out std_logic);
  end component;

component top is
  Port (
        --input
        clki          : in std_logic;
        reseti        : in std_logic;
        starti        : in std_logic;
        inputi        : in std_logic_vector(15 downto 0);
        --output
        start_ld_inputo : out std_logic;
        start_ld_coeffo : out std_logic;
        max_outo       : out std_logic_vector(17 downto 0)
  );
end component;

signal clki             : std_logic;
signal reseti           : std_logic;
signal starti           : std_logic;
signal inputi           : std_logic_vector(15 downto 0);
signal start_ld_inputo  : std_logic;
signal start_ld_coeffo  : std_logic;
signal max_outo         : std_logic_vector(17 downto 0);

begin

top_part: top
port map (
    clki            => clki  ,
    reseti          => reseti     ,
    starti          => starti  ,
    inputi          => inputi     ,
    start_ld_inputo => start_ld_inputo,
    start_ld_coeffo => start_ld_coeffo,
    max_outo        => max_outo
```

```
);

clkpad : CPAD_S_74x50u_IN
  port map (
    COREIO => clki,
    PADIO  => clk
    );
resetpad : CPAD_S_74x50u_IN
  port map (
    COREIO => reseti,
    PADIO  => reset
    );
 startpad : CPAD_S_74x50u_IN
  port map (
    COREIO => starti,
    PADIO  => start
    );
InPads : for i in 0 to 15 generate
  InPad : CPAD_S_74x50u_IN
    port map (
      COREIO => inputi(i),
      PADIO  => input(i)
      );
end generate InPads;

OutPads : for i in 0 to 17 generate
  OutPad : CPAD_S_74x50u_OUT
    port map (
      COREIO => max_outo(i),
      PADIO  => max_out(i)
      );
end generate OutPads;

ld_coeff_pad : CPAD_S_74x50u_OUT
 port map (
   COREIO => start_ld_coeffo,
   PADIO  => start_ld_coeff
   );
ld_input_pad : CPAD_S_74x50u_OUT
 port map (
   COREIO => start_ld_inputo,
   PADIO  => start_ld_input
   );

end Behavioral;
```