논리회로설계실험 Term Project

Group3 박준석 최장헌 노원우 이승우

1. Objective

Verilog-HDL을 사용하여 CNN 모델을 설계한다. Module을 조정하는 Main Controller, RAM, 이를 이용해 계산하는 Computation module, 계산 결과를 반영하는 Display module을 설계한다. 각 module을 설계하고 Simulation을 통해 검증한다. synthesis에 성공하여 FPGA를 통해 결과를 확인한다.

2. Main Controller

다음은 Main controller 의 state machine diagram 이다. Moore machine 으로 설계되었다.

큰 틀에서 위와 같이 진행되며, 각 state 에서 세부적으로 많은 단계가 존재한다.

Input write 단계는 입력을 원하는 행렬 값들을 RAM 에 저장하는 단계이다.

Input load 단계는 RAM 에서 행렬 값들을 가져오는 단계이다.

PE, SA, CT 단계는 행렬 값으로 single PE 에서 연산을 하는 단계이다. 연산과 동시에 RAM 에 결과를 저장한다.

Output load 단계는 RAM 에 저장된 연산 결과를 가져오는 단계이다.

DP 단계는 연산 결과를 display 에 표시하는 단계이다.

```
default: begin next_state <= INITIAL; end

INITIAL:
begin
    if (switch)
        next_state <= a_1_1_write;
    else
        next_state <= INITIAL;
end

// PE
pe_start:
begin
    next_state <= pe_wait;
end

pe_wait:
begin
    if (pe_en_result)
        next_state <= pe_c_1_1_load;
    else
        next_state <= pe_wait;
end

pe_c_1_1_load:
begin
    if (pe_en_result)
        next_state <= pe_c_1_2_load;
    else
        next_state <= pe_c_1_1_load;
end
```

최초 switch를 눌러야 본격적으로 작동하기 시작한다. 그 후 Input write, load 단계에서는 posedge 마다 다음 state로 이동한다.

PE, SA, CT 단계에서는 결과를 확인하는 control bit 인 en_result 신호마다 다음 state 로 넘어간다.

DP 는 마지막 단계로 무한 반복한다.

```
// DP
dp_start:
begin
    next_state <= dp_start;
end
```

```
pe_start:      begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end
pe_wait:       begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end
pe_c_1_1_load: begin data <= pe_result; addr <= 6'd32; we <= 1'b1; end
pe_c_1_2_load: begin data <= pe_result; addr <= 6'd33; we <= 1'b1; end
pe_c_2_1_load: begin data <= pe_result; addr <= 6'd34; we <= 1'b1; end
pe_c_2_2_load: begin data <= pe_result; addr <= 6'd35; we <= 1'b1; end
pe_end:        begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end

sa_start:      begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end
sa_c_2_1_load: begin data <= sa_result; addr <= 6'd39; we <= 1'b1; end
sa_c_1_1_load: begin data <= sa_result; addr <= 6'd37; we <= 1'b1; end
sa_c_2_2_load: begin data <= sa_result; addr <= 6'd40; we <= 1'b1; end
sa_c_1_2_load: begin data <= sa_result; addr <= 6'd38; we <= 1'b1; end
sa_end:        begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end

ct_start:      begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end
ct_wait:       begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end
ct_c_1_1_load: begin data <= ct_result; addr <= 6'd42; we <= 1'b1; end
ct_c_1_2_load: begin data <= ct_result; addr <= 6'd43; we <= 1'b1; end
ct_c_2_1_load: begin data <= ct_result; addr <= 6'd44; we <= 1'b1; end
ct_c_2_2_load: begin data <= ct_result; addr <= 6'd45; we <= 1'b1; end
ct_end:        begin data <= 8'd000; addr <= 6'd25; we <= 1'b0; end
```

값을 저장하는 블록이다.

Data, adder, we 값을 결정하여 원하는 값을 RAM 에 저장한다.

```
a_1_1_load_1: begin a_1_1_out <= 8'd233; end
a_1_2_load_1: begin a_1_2_out <= 8'd123; end
a_1_3_load_1: begin a_1_3_out <= 8'd012; end
a_1_4_load_1: begin a_1_4_out <= 8'd003; end
a_2_1_load_1: begin a_2_1_out <= 8'd005; end
a_2_2_load_1: begin a_2_2_out <= 8'd002; end
a_2_3_load_1: begin a_2_3_out <= 8'd003; end
a_2_4_load_1: begin a_2_4_out <= 8'd003; end
a_3_1_load_1: begin a_3_1_out <= 8'd009; end
a_3_2_load_1: begin a_3_2_out <= 8'd255; end
a_3_3_load_1: begin a_3_3_out <= 8'd001; end
a_3_4_load_1: begin a_3_4_out <= 8'd012; end
a_4_1_load_1: begin a_4_1_out <= 8'd013; end
a_4_2_load_1: begin a_4_2_out <= 8'd064; end
a_4_3_load_1: begin a_4_3_out <= 8'd055; end
a_4_4_load_1: begin a_4_4_out <= 8'd027; end
b_1_1_load_1: begin b_1_1_out <= 8'd013; end
b_1_2_load_1: begin b_1_2_out <= 8'd002; end
b_1_3_load_1: begin b_1_3_out <= 8'd003; end
b_2_1_load_1: begin b_2_1_out <= 8'd002; end
b_2_2_load_1: begin b_2_2_out <= 8'd001; end
b_2_3_load_1: begin b_2_3_out <= 8'd050; end
b_3_1_load_1: begin b_3_1_out <= 8'd051; end
b_3_2_load_1: begin b_3_2_out <= 8'd052; end
b_3_3_load_1: begin b_3_3_out <= 8'd001; end
```

왼쪽은 값을 불러오는 블록으로 제대로 구현하지 못해 임의로 값을 불러오게 하였다.

아래는 각종 control bit 을 결정하는 블록이다.

```
pe_start: begin en_pe <= 1'b1; en_sa <= 1'b0; en_ct <= 1'b0; all_done = 1'b0; end
pe_wait: begin en_pe <= 1'b0; en_sa <= 1'b0; en_ct <= 1'b0; all_done = 1'b0; end
pe_c_1_1_load: begin en_pe <= 1'b0; en_sa <= 1'b0; en_ct <= 1'b0; all_done = 1'b0; end
pe_c_1_2_load: begin en_pe <= 1'b0; en_sa <= 1'b0; en_ct <= 1'b0; all_done = 1'b0; end
pe_c_2_1_load: begin en_pe <= 1'b0; en_sa <= 1'b0; en_ct <= 1'b0; all_done = 1'b0; end
pe_c_2_2_load: begin en_pe <= 1'b0; en_sa <= 1'b0; en_ct <= 1'b0; all_done = 1'b0; end
pe_end: begin en_pe <= 1'b0; en_sa <= 1'b0; en_ct <= 1'b0; all_done = 1'b0; end
```

3. Memory

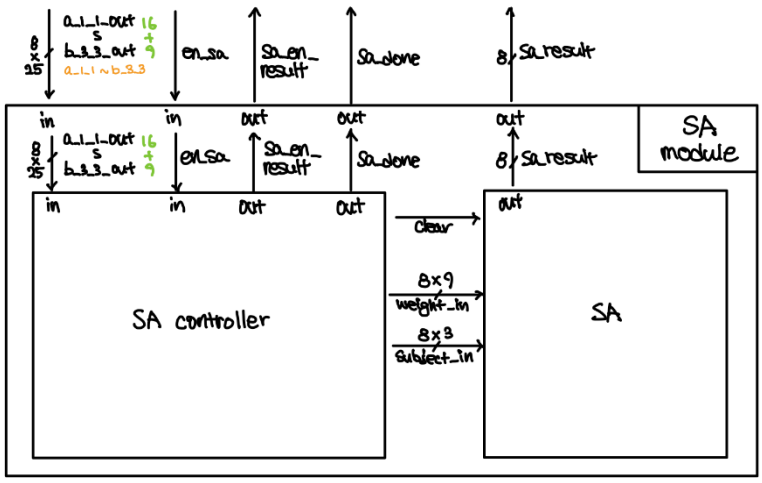
메모리는 미리 주어진 Single port ram 파일을 이용했다. 메모리는 한 Clock에 한 개의 값만 꺼내 올 수 있기 때문에, Systolic Array와 같이 3개씩 꺼내는 상황에서 Timing 문제를 해결하기 어렵다고 생각했다. 따라서 미리 input 16개, filter 값 9개 총 25개의 값을 미리 Controller에 옮겨 둔 후, 이를 필요에 따라 동시에 꺼내 쓸 수 있게 설계했다.

4. SA Module

우리가 설계한 PE의 모든 기능은 SA mode에서 사용되고, Single PE mode에서는 제한된다. 따라서 PE를 설명하며 미리 SA mode를 설명한 후 Single PE mode로 넘어가겠다.

4.1 Theoretical Design

다음은 우리가 설계한 SA module의 블록 다이어그램이다.



검은색은 input, output 신호,

초록색은 wire를 의미한다.

Single SA module은 SA_controller와

SA로 구성되어 있다.

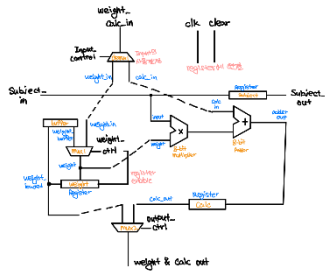
SA module은 Main controller로부터 연산에 사용될 8-bit 신호 25개(**a_1_1~a_4_4, b_1_1~b_3_3**)와 SA module의 작동 시작을 알리는 control bit인 **en_sa**를 **input**으로 받는다. 그리고 연산 결과인 8-bit 신호 **sa_result**와 연산 결과가 유효함을 알려주는 control bit인 **sa_en_result**(result enable), 마지막으로 SA module의 동작이 완전히 끝났음을 알리는 control bit인 **sa_done** 신호를 output으로 내보낸다. SA module에서 사용된 모든 control bit은 active high로 동작한다.

SA module의 내부에는 연산을 담당하는 SA와 그것을 control할 SA_controller가 있다.

SA는 주요 기능인 연산을 담당한다. 연산에 사용될 8-bit 신호인 9개의 **weight**와 3개의 **subject**, 그리고 초기화에 필요한 control bit인 clear 신호를 SA_controller로부터 input으로 받고, 연산 결과를 8-bit 신호인 **sa_result** output으로 내보낸다.

SA_controller는 SA가 연산을 수행할 수 있게, SA에 들어가는 input을 조절해주는 SA module의 controller이다. Main controller로부터 8-bit 신호인 25개(**a_1_1~a_4_4, b_1_1~b_3_3**)의 모든 데이터와 SA module의 작동 시작을 알리는 control bit인 **en_sa**를 **input**으로 받는다. 그리고 연산의 과정을 Main controller에게 전달하기 위해 2개의 control bit인 **sa_en_result**, **sa_done**를 output으로 내보낸다.

SA 는 조사한 바에 따르면, weight stationary 와 output stationary 두가지가 있다. 이 프로젝트는 개발의 지향점을 weight stationary systolic array(SA)로 잡고 시작하였다. 그 이유는 프로젝트를 최초 설명할 때 교수님이 설명해 주신 SA 의 형태와 유사하고, convolution 연산에 사용할 때 output stationary 와 비교하여 장점이 많기 때문이다. 다음은 최초 설계했던 PE 이다.

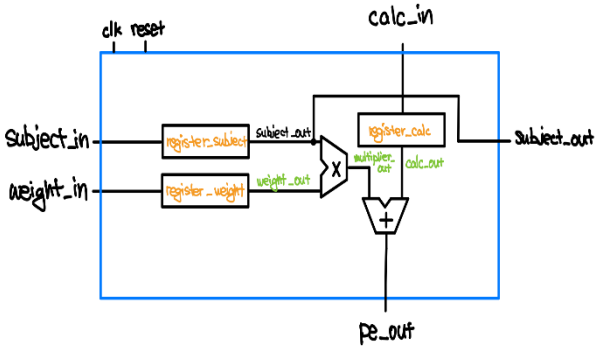


SA 에 사용할 PE 를 설계하던 중 이와 같은 PE 를 완성하였다. Weight 도 subject 처럼 밑으로 전달해주다 flipflop 의 enable bit 를 사용해 원하는 타이밍에 값을 저장하는 방식이었다. 이 PE 는 최종적으로 사용되지 못했는데, SA 만 synthesis simulation 했을때는 정상적으로 작동했으나, SA controller 와 결합하여 synthesis simulation 을 진행했을 때, 유효한 weight 를 register 와 buffer 에 저장하는데 timing 문제가 있어, 결국 그 문제를 해결하지 못해 사용하지 못했다.

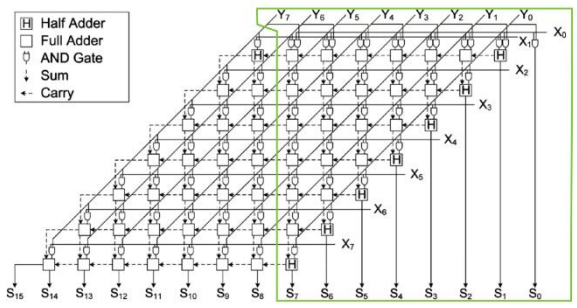
weight stationary 를 지향했으나, 구현하는데 실패하여 그 차선책으로 모든 PE 에 weight 신호를 계속 주어 weight stationary 처럼 작동하게 구현하였다. 최종적으로 PE 는 3 개의 8-bit register 와 8-bit multiplier, 8-bit adder 로 간단하게 구성하였고, controller 를 통해 weight stationary 처럼 동작할 수 있게 구현하였다.

최종적으로 완성된 PE 는 **clk**, **reset** 신호와 연산에 사용할 8-bit **subject_in**, **weight_in**, **calc_in** 을 input 으로 받으며, output 으로 8-bit **subject_out** 과 **pe_out** 을 내보낸다.

clk, reset 신호는 register 를 control 하는데 사용하며, subject_in 은 계산의 대상이 되는 행렬의 값, weight_in 은 filter 행렬의 값을 받게 설계되었다. clac_in 은 이전 pe 의 결과값을 받을 수 있게 되어 있고, 최종적으로 subject_out 으로 이전에 사용했던 subject_in 값을 내보내고, subject_in * weight_in + calc_in 의 결과를 pe_out 으로 내보낸다.



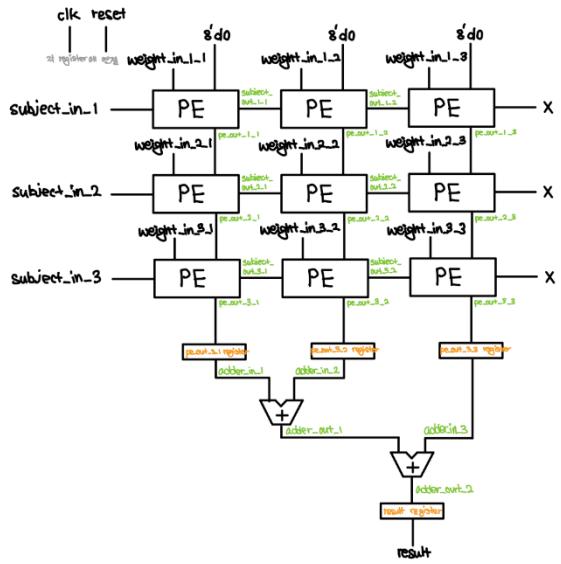
프로젝트에 전체적으로 사용된 8-bit 곱셈기는 다음과 같다.



자료조사를 통해 찾은 8-bit 곱셈기를 필요한 부분만 남기고 잘라내 사용했다. Synthesis 후 신호가 조각나 debugging 이 안되는 것을 방지하기 위해 필요 없는 부분을 꼼꼼하게 잘랐다. 그를 위해 기존의 half_adder 와 full_adder 에서 carry_out 없이 sum 연산만을 수행하는 모듈을 따로 제작하여 활용하였다.

8-bit adder 는 4-bit half adder 와 4-bit full adder 를 합쳐 사용했다.

위의 모듈들을 활용하여 만든 SA 는 다음과 같다

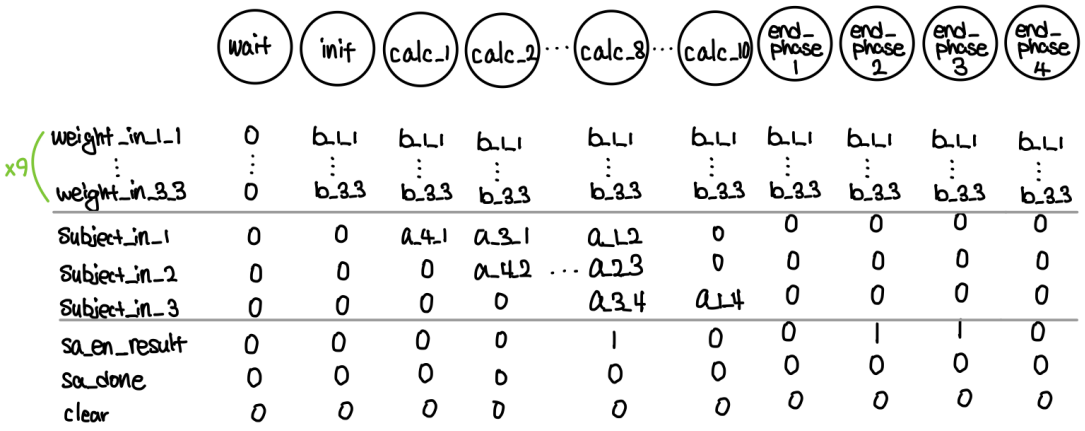


9 개의 PE, 4 개의 8-bit register, 2 개의 8-bit adder 로 구성되어 있다.

Weight stationary 처럼 작동하기 위해 모든 PE 에 weight 신호가 연결되어 있으며, subject 는 왼쪽의 3 개의 선에서 들어와 오른쪽으로 차례로 전달된다. 모든 register 는 clk 와 reset 신호를 받으며, 9 개의 PE 에서 연산한 결과를 하단부에서 더해 result 로서 output 으로 내보낸다. 하단부의 register 4 개는 timing 이 어긋나는 것을 막기 위해 추가하였다.

다음은 SA controller 의 state machine diagram 이다.

Moore machine 으로 설계되었으며, 각 state 에 따른 output 표이다. 회색선은 다른 블록으로 처리되었음을 의미하며 output 의 성격에 따라 3 개의 블록으로 나누었다. 블록마다 따로 처리되어 delay 에 의한 문제가 없게 하기 위함이다.



sa_en_result는 calc_8, calc_9, end_phase_3, end_phase_4 에 1, 나머지 0

	calc_1	calc_2	calc_3	calc_4	calc_5	calc_6	calc_7	calc_8	calc_9	calc_10
Subject_in_1	a_4_1	a_3_1	a_2_1	a_1_1	a_1_2	a_2_2	a_3_2	a_1_2	0	0
Subject_in_2	0	a_4_2	a_3_2	a_2_2	a_1_2	a_1_3	a_3_3	a_2_3	a_1_3	0
Subject_in_3	0	0	a_4_3	a_3_3	a_2_3	a_1_3	a_1_4	a_3_4	a_2_4	a_1_4

Subject_in_1 은 순서대로 SA 의 1-1, 첫번째 PE 에 들어가는 입력이다. Calc_1 부터 Calc_10 까지 총 10 번에 걸쳐 인풋이 들어가며, a_4_1 부터 들어간다. input 이 들어가는 순서를 거꾸로 정한 근거는 다음과 같다.

a_4_1 a_3_1 a_2_1

a_4_2 a_3_2 a_2_2

a_4_3 a_3_3 a_2_3 을 입력 받으면 c_2_1 이 나오고,

a_3_1 a_2_1 a_1_1

a_3_2 a_2_2 a_1_2

a_3_3 a_2_3 a_1_3 을 입력 받으면 c_1_1 이 나오는데,

a_4_1 a_3_1 a_2_1 a_1_1

a_4_2 a_3_2 a_2_2 a_1_2

a_4_3 a_3_3 a_2_3 a_1_3 을 입력 받으면,

c_2_1 의 결과가 나온 직후 c_1_1 의 결과도 얻을 수 있다.

위 방식은 프로젝트의 예시인 4 by 4 행렬을 3 by 3 filter 로 convolution 하는데도 눈에 띄게 빠르다는 장점이 있고, input 행렬이 n by n 일 경우 n 이 커짐에 따라 연산에 걸리는 시간이 O(n)만큼만 증가한다는 매우 큰 장점이 있다.

위와 같은 방식으로 연산을 수행하면 state 가 calc_8, calc_9, end_phase_2, end_phase_3 에서 각각 c_2_1, c_1_1, c_2_2, c_1_2 의 값이 연산 되고, 이를 main controller 에 알리기 위해 sa_en_result 신호가 켜진다.

4.2 Verilog Implementation

Controller를 제외한 다른 module들은 전부 structural modeling 기법으로 위의 Theoretical Design을 바탕으로 하여 input, output만 적절하게 연결했다.

이 부분에서는 Behavioral modeling 기법으로 설계된 controller를 자세히 다루겠다.

```
parameter
wait_state = 0,
init = 26,

delay_1 = 1, delay_2 = 2 ,
//    delay_3 = 3, delay_4 = 4, delay_5 = 5,
//    delay_6 = 6, delay_7 = 7 , delay_8 = 8, delay_9 = 9, delay_10 = 10,

calc_1 = 16, calc_2 = 17, calc_3 = 18, calc_4 = 19, calc_5  = 20,
calc_6 = 21, calc_7 = 22, calc_8 = 23, calc_9 = 24, calc_10 = 25,

end_phase_1 = 28, end_phase_2 = 29, end_phase_3 = 30, end_phase_4 = 31;
```

state를 선언한 부분이다.

대기 상태를 의미하는 wait_state

시작을 의미하는 init(initial)

계산 과정, 정확히는 subject input이 들어가는 과정, calc_1~10

계산이 끝나고 원하는 output이 나올때까지 기다리는 과정 end_phase_1~4 으로 선언하였다.

post-synthesis timing simulation에서 state가 3->4, 7->8, 15->16 등 2진법에서 한자릿수가 오르는 동작에서 알 수 없는 이유로 레지스터의 값이 0으로 초기화되는 문제가 있어서 의미있는 동작은 하지 않는 delay state 를 추가하였다.

```
// decide output control
always @ (state)
begin
    case (state)
        wait_state: // wait
        begin
            sa_en_result <= 1'b0;
            clear <= 1'b1;
            sa_done <= 1'b0;
        end

        init:   // init
        begin
            sa_en_result <= 1'b0;
            clear <= 1'b0;
            sa_done <= 1'b0;
        end

        end_phase_4:
        begin
            sa_en_result <= 1'b0;
            clear <= 1'b0;
            sa_done <= 1'b1;
        end

        default:
        begin
            sa_en_result <= 1'b0;
            clear <= 1'b0;
            sa_done <= 1'b0;
        end
    endcase
end
```

Control bit에 해당하는 output을 결정하는 블록이다. 혹시 모를 오류를 대비하여 모든 state에 대해서 output을 결정해주었고, default도 따로 설정해주었다.

```
// decide output value weight
always @ (state)
begin
    case (state)
        wait_state: // wait
        begin
            weight_in_1_1 <= 8'd0;
            weight_in_1_2 <= 8'd0;
            weight_in_1_3 <= 8'd0;
            weight_in_2_1 <= 8'd0;
            weight_in_2_2 <= 8'd0;
            weight_in_2_3 <= 8'd0;
            weight_in_3_1 <= 8'd0;
            weight_in_3_2 <= 8'd0;
            weight_in_3_3 <= 8'd0;
        end

        init:
        begin
            weight_in_1_1 <= b_1_1;
            weight_in_1_2 <= b_1_2;
            weight_in_1_3 <= b_1_3;
            weight_in_2_1 <= b_2_1;
            weight_in_2_2 <= b_2_2;
            weight_in_2_3 <= b_2_3;
            weight_in_3_1 <= b_3_1;
            weight_in_3_2 <= b_3_2;
            weight_in_3_3 <= b_3_3;
        end
```

```
// decide output subject
always @ (state)
begin
    case (state)
        wait_state: // wait
        begin
            subject_in_1 <= 8'd0;
            subject_in_2 <= 8'd0;
            subject_in_3 <= 8'd0;
        end

        init: // init
        begin
            subject_in_1 <= 8'd0;
            subject_in_2 <= 8'd0;
            subject_in_3 <= 8'd0;
        end

        calc_1:
        begin
            subject_in_1 <= a_4_1;
            subject_in_2 <= 8'd0;
            subject_in_3 <= 8'd0;
        end
        calc_2:
        begin
            subject_in_1 <= a_3_1;
            subject_in_2 <= a_4_2;
            subject_in_3 <= 8'd0;
        end
```

```
//Update the current state
always @ (posedge clk)
begin
    if (reset)
    begin
        state <= wait_state;
    end
    else
    begin
        state <= next_state;
    end
end
```

```
//Determine the next state
always @ (*)
begin
    case(state)
        default:
        begin
            next_state <= wait_state;
        end

        wait_state: // wait state
        begin
            if (en_sa)
                next_state <= init;
            else
                next_state <= wait_state;
        end

        init:
        begin
        end_phase_4:
        begin
            next_state <= end_phase_4;
        end
    endcase
```

왼쪽의 자료 2개는 SA에 입력할 weight와 subject를 결정하는 블록 2개이다.

위의 블록과 마찬가지로 모든 state에 대해서 output을 결정해주었고, default도 따로 설정해주었다.

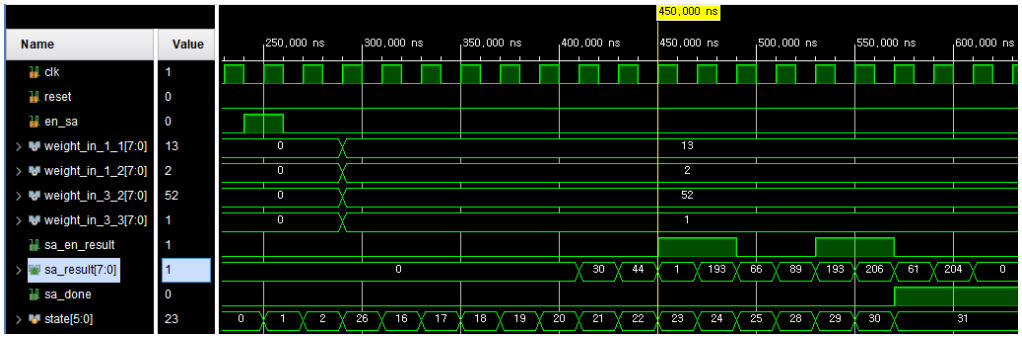
오른쪽의 자료 2개는 차례로 state를 update하는 블록과 next_state를 결정하는 블록이다.

최초 동작시 wait_state로 시작하게 설계하였으며, Reset 신호를 받아도 wait_state가 된다.

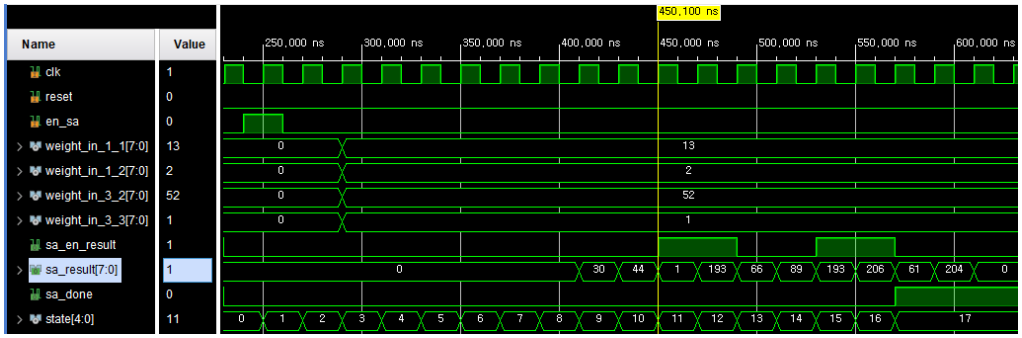
Wait_state일 때, en_sa 신호를 받으면 다음 clk에서 init state로 넘어가게 되고, 그 이후로는 clk마다 다음 state로 넘어가게 설계했다.

4.3 simulation

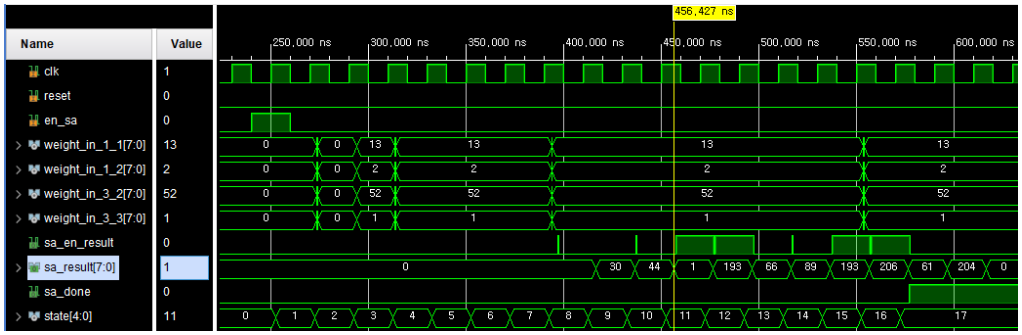
1. Behavioral simulation



2. post-synthesis functional simulation



3. post-synthesis timing simulation



각 simulation의 첫번째 결과가 나오는 지점의 시간을 보면, (1) 450ns, (2) 450.1ns, (3) 456.427ns로 조금씩 차이가 난다. 프로젝트를 진행하면서 (1)과 (2)의 결과는 매우 잘 나와도 (3) post-synthesis timing simulation에서 오류가 많이 생겨 (3)을 중심으로 수많은 디버그를 진행하였다.

결과적으로 모든 오류를 잘 고쳐 3가지 simulation에서 정상적으로 작동하게 하는데 성공하였다.

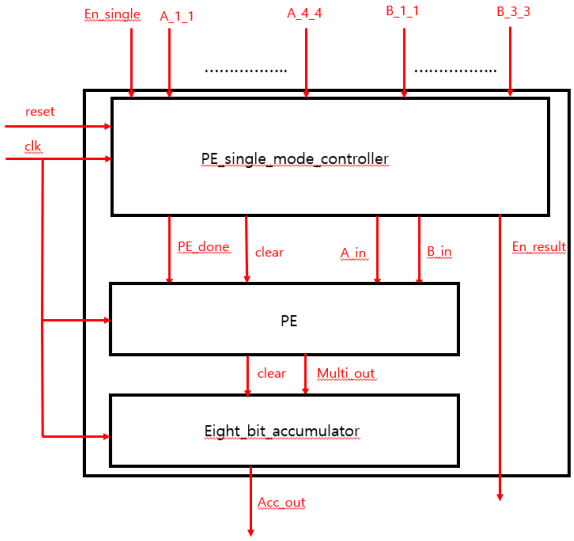
en_sa 신호를 받은 뒤, state가 1에서부터 마지막인 17번 state까지 매 clk마다 변하는 것을 확인할 수 있고, 원하는 결과 1, 193, 193, 206이 나올 때, sa_en_result 신호가 켜지는 것도 확인할 수 있다.

210ns에 en_sa가 켜져 계산을 시작한 후, 550ns의 rising edge에서 마지막 결과 값이 Main controller로 보내진다. 따라서 SA module의 계산 시간은 340ns이다.

5. Single PE Mode

5.1 Theoretical Design

다음은 우리가 설계한 Single PE module의 블록 다이어그램이다.



Single PE module은 PE_Single_mode_controller 와 PE, Eight_bit_accumulator로 구성되어 있다.

여기서 PE는 single PE mode이기 때문에 단순한 곱셈기의 역할을 한다. 후의 SA에서 쓰일 PE와 같은 모듈이므로, input과 output단이 더 존재하나 필요하지 않는 부분들을 표시하지 않았다. PE_Single_mode_controller에서는 먼저 Main controller로부터 input 행렬의 값 16개와 필터 값 9개를 받는다. 또한 En_single bit를 받는데, 이는 PE_single mode를 시작하라는 신호이다. 이 컨트롤러는 en_single이 켜진 후부터는 자동적으로 계산에 필요한 A와 B의 값을 PE로 보낸다. PE는 이 값을 곱하여 Eight_bit_accumulator로 보낸다. Eight_bit accumulator는 이 곱셈 값을 축적시켜서 매번 Acc_out으로 내보낸다. 하나의 C가 계산이 끝나면, Accumulator는 초기화된다. 유효한 C 값을 판단하는 기준은 En_result이며, Single_mode가 끝났을 때 PE_done을 1로 켜준다. En_result에 따라 값을 디스플레이부로 옮기는 것은 Main controller에서 SA와 마찬가지로 담당할 것이다.

5.2 Verilog Implementation

5.2.1 PE single mode controller

반복되는 코드가 많기 때문에, 필요한 부분만 첨부하여 설명하겠다. 각 C는 9개의 항으로 이루어져 있으므로, 각 항에서 이루어질 곱셈을 위한 a, b값을 a_in, b_in에 넣는 state이 c11_1 ~c11_9, c12_1 ~ c12_9, c21_1 ~ c21_9, c22_1 ~ c22_9 state이다. 이렇게 a_in, b_in을 정하는 순간 뒤의 PE에서 곱셈을 진행할 것이다. 따라서 c11_9가 끝나면 c11_done state로 보내고, 그 다음 c11_result에서 en_result를 1로 켜준다. 이는 main controller가 이 module의 결과 값을 언제 저장해야 하는지 timing을 알려주는 신호이다. C11_result -> c11_end state -> c11_reset으로 가 reset을 1로 켜주면 accumulator가 초기화되며 다음 C 값을 계산할 준비가 완료된다. 이를 반복하여 각 C값을 계산하고, 모든 C값이 끝나면 c22_reset에서는 pe_done 신호를 1로 켜주며 Single_PE_mode 계산이 끝났음을 알려준다.

```
module PE_single_mode_controller(clk, reset, en_single,
a_1_1, a_1_2, a_1_3, a_1_4,
a_2_1, a_2_2, a_2_3, a_2_4,
a_3_1, a_3_2, a_3_3, a_3_4,
a_4_1, a_4_2, a_4_3, a_4_4,
b_1_1, b_1_2, b_1_3,
b_2_1, b_2_2, b_2_3,
b_3_1, b_3_2, b_3_3,
a_in, b_in,
en_result,pe_done, clear);

    input clk, reset, en_single;
    input [7:0] a_1_1, a_1_2, a_1_3, a_1_4;
    input [7:0] a_2_1, a_2_2, a_2_3, a_2_4;
    input [7:0] a_3_1, a_3_2, a_3_3, a_3_4;
    input [7:0] a_4_1, a_4_2, a_4_3, a_4_4;

    input [7:0] b_1_1, b_1_2, b_1_3;
    input [7:0] b_2_1, b_2_2, b_2_3;
    input [7:0] b_3_1, b_3_2, b_3_3;


    output reg en_result;
    output reg clear;

    output reg pe_done;
    output reg [7:0]a_in, b_in;

    reg [6:0]state, next_state;

    parameter wait_state = 0, init = 1, c11_1 = 2, c11_2 = 3, c11_3 = 4, c11_4 = 5, c11_5 = 6, c11_6 = 7, c11_7 = 8, c11_8 = 9, c11_9 = 10, c11_done = 12,
    c11_result = 13,c11_end = 14, c11_reset = 15,
    c12_1 = 16, c12_2 = 17, c12_3 = 18, c12_4 = 19, c12_5 = 20, c12_6 = 21, c12_7 = 22, c12_8 = 23, c12_9 = 24, c12_done = 25, c12_result = 26, c12_end = 27, c12_reset = 28,
    c21_1 = 29, c21_2 = 30, c21_3 = 31, c21_4 = 32, c21_5 = 33, c21_6 = 34, c21_7 = 35, c21_8 = 36, c21_9 = 37, c21_done = 38, c21_result = 39, c21_end = 40, c21_reset = 41,
    c22_1 = 42, c22_2 = 43, c22_3 = 44, c22_4 = 45, c22_5 = 46, c22_6 = 47, c22_7 = 48, c22_8 = 49, c22_9 = 50, c22_done = 51, c22_result = 52, c22_end = 53, c22_reset = 54,
    end_phase = 55;
```

```
always @ (posedge clk)
begin
   if (reset)
   begin
      state <= wait_state;
    end
   else
   begin
      state <= next_state;
    end
end
```

Module의 input과 output, state들을 설정한 그림이다. 오른쪽 그림은 Reset의 동작 코드이다.

```
always @ (*)
begin
   if (reset) // action
     begin
         next_state <= wait_state;
     end
     else if (en_single)
     begin
         next_state <= init;
     end
     else
     begin
         case (state)
             init:
             begin
                 next_state <= c11_1;
             end
             c11_1:
             begin
                 next_state <= c11_2;
             end
             c11_2:
             begin
                 next_state <= c11_3;
             end
             c11_3:
             begin
                 next_state <= c11_4;
             end
             c11_4:
             begin
                 next_state <= c11_5;
             end
```

```
c11_9:
begin
    next_state <= c11_done;
end
c11_done:
begin
    next_state <= c11_result;
end
c11_result:
begin
    next_state <= c11_end;
end
c11_end:
begin
    next_state <= c11_reset;
end
c11_reset:
begin
    next_state <= c12_1;
end
```

```
always @ (state)
begin
  case (state)
     wait_state: // wait
     begin
        a_in <= 8'b0;
        b_in <= 8'b0;
     end

     init:
     begin
        a_in <= 8'b0;
        b_in <= 8'b0;
     end

     c11_1:
     begin
        a_in <= a_1_1;
        b_in <= b_1_1;
     end
     c11_2:
     begin
        a_in <= a_1_2;
        b_in <= b_2_1;
     end
     c11_3:
     begin
        a_in <= a_1_3;
        b_in <= b_3_1;
     end
```

다음은 next_state을 결정해주는 코드이다. Reset이 0이고 en_single이 켜졌다면, 코드에 따라 계산이 진행된다. C11_9-> c11_done_->c11_result -> c11_end -> c11_reset -> 다음 c의 state으로 가며 진행된다. C22까지 반복되므로 생략하겠다. 오른쪽 코드는 c의 각 항이 계산될 때 PE로 넣어줄 a_in과 b_in을 결정하는 코드이다. 이 또한 각 C별로 9번, 총 36번 반복되었다.

```
always @ (state)
 begin
   case (state)
      wait_state: // wait
      begin
            en_result <= 1'b0;
            clear <= 1'b1;
            pe_done <= 1'b0;
      end

      init:
      begin
            en_result <= 1'b0;
            clear <= 1'b0;
            pe_done <= 1'b0;
      end
                  output reg
      c11_1:
      begin
            en_result <= 1'b0;
            clear <= 1'b0;
            pe_done <= 1'b0;
      end
      c11_2:
      begin
            en_result <= 1'b0;
            clear <= 1'b0;
            pe_done <= 1'b0;
      end
      c11_3:
      begin
            en_result <= 1'b0;
            clear <= 1'b0;
            pe_done <= 1'b0;
      end
```

```
c22_reset:
begin
       en_result <=1'b0;
       clear <= 1'b1;
       pe_done <= 1'b1;
end

  c11_done:
  begin
        en_result <= 1'b0;
        clear <= 1'b0;
        pe_done <= 1'b0;
  end
  c11_result:
  begin
        en_result <=1'b1;
        clear <= 1'b0;
        pe_done <= 1'b0;
  end
  c11_end:
  begin
        en_result <=1'b0;
        clear <= 1'b0;
        pe_done <= 1'b0;
  end
  c11_reset:
  begin
        en_result <=1'b0;
        clear <= 1'b1;
        pe_done <= 1'b0;
  end
```

컨트롤러의 마지막 역할로 state별 en_result, clear, pe_done을 정해주었다. 굳이 한 코드로 묶지 않고 위의 코드들과 쪼갠 이유는, 이렇게 구문을 나눠 놓으면 개별적으로 동작하기 때문에 속도 측면에서도 좋고, 가독성이 좋기 때문이다. 사용하지 않는 신호들까지 모드 선언해서, 예상치 못한 오류를 없앴다. En_result는 c11_result에서 켜지고, 이 신호는 main controller로 가서 결과 값을 저장할 타이밍을 알려준다. C11_9에서 input을 전달한 후, 실질적으로 마지막 항의 계산은 c11_done에서 끝나기 때문에, 그 다음 state에서 en_result를 인가했다. C11_reset에서 clear 신호를 통해 accumulator를 초기화하며 다음 C 계산으로 넘어간다. 이를 4번 반복하였고, C22_reset에서는 Single_mode의 끝을 알리기 위해 pe_done을 1로 인가한다.

5.2.2 PE

PE는 이미 설명하였기 때문에 여기서는 생략한다.

5.2.3 eight_bit_accumulator

```
module eight_bit_accumulator(clk, clear, acc_in, acc_out);

    input clk;
    input clear;

    input [7:0] acc_in;

    wire [7:0] acc_in_register_out; // for calc register
    wire [7:0] acc_register_out; // for multiplier

    output [7:0] acc_out; // for adder

    eight_bit_register      acc_in_register (.in(acc_in), .clk(clk), .rst(clear), .en(1'b1), .out(acc_in_register_out));
    eight_bit_register      acc_register    (.in(acc_out), .clk(clk), .rst(clear), .en(1'b1), .out(acc_register_out));

    eight_bit_half_adder    eight_bit_adder (.a(acc_in_register_out), .b(acc_register_out), .out(acc_out));
endmodule
```

Eight_bit accumulator는 register와 half adder로 이루어져 있다. Half adder는 Cin과 Cout가 필요 없기 때문에 제거한 adder이다. Register를 통해 들어온 값이 저장되며, Adder를 통해 새로 들어온 값과 더해져 축적되는 구조이다.

5.2.4 PE_single_module

```
module PE_single_module(clk, reset, en_single,
a_1_1, a_1_2, a_1_3, a_1_4, a_2_1, a_2_2, a_2_3, a_2_4, a_3_1, a_3_2, a_3_3, a_3_4, a_4_1, a_4_2, a_4_3, a_4_4,
b_1_1, b_1_2, b_1_3, b_2_1, b_2_2, b_2_3, b_3_1, b_3_2, b_3_3, acc_out, pe_done);

input clk, reset, en_single;
input [7:0]a_1_1, a_1_2, a_1_3, a_1_4, a_2_1, a_2_2, a_2_3, a_2_4, a_3_1, a_3_2, a_3_3, a_3_4, a_4_1, a_4_2, a_4_3, a_4_4;
input [7:0]b_1_1, b_1_2, b_1_3, b_2_1, b_2_2, b_2_3, b_3_1, b_3_2, b_3_3;

wire [7:0]a_in, b_in;
wire en_result, clear;
wire [7:0]multi_out;

output [7:0]acc_out;
output pe_done;

PE_single_mode_controller PE_single_mode_controller_1(.clk(clk), .reset(reset), .en_single(en_single),
.a_1_1(a_1_1), .a_1_2(a_1_2), .a_1_3(a_1_3), .a_1_4(a_1_4),
.a_2_1(a_2_1), .a_2_2(a_2_2), .a_2_3(a_2_3), .a_2_4(a_2_4),
.a_3_1(a_3_1), .a_3_2(a_3_2), .a_3_3(a_3_3), .a_3_4(a_3_4),
.a_4_1(a_4_1), .a_4_2(a_4_2), .a_4_3(a_4_3), .a_4_4(a_4_4),
.b_1_1(b_1_1), .b_1_2(b_1_2), .b_1_3(b_1_3),
.b_2_1(b_2_1), .b_2_2(b_2_2), .b_2_3(b_2_3),
.b_3_1(b_3_1), .b_3_2(b_3_2), .b_3_3(b_3_3),
.a_in(a_in), .b_in(b_in),
.en_result(en_result), .pe_done(pe_done), .clear(clear));

PE PE_1(.clk(clk), .clear(clear), .weight_control(1'b1), .weight_in(b_in), .subject_in(a_in), .calc_in(8'd0), .weight_out(), .subject_out(), .adder_out(multi_out));
eight_bit_accumulator acuum_1(.clk(clk), .clear(clear), .acc_in(multi_out), .acc_out(acc_out));
```

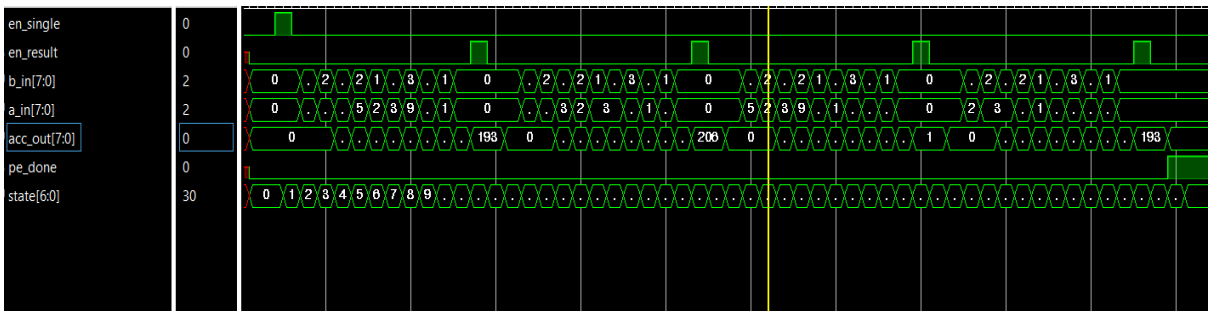
위의 controller, PE, accumulator를 통합하여 만든 Single module이다. 결론적으로 이 module의 output은 accumulator의 acc_out이 된다.

5.2.5 Simulation

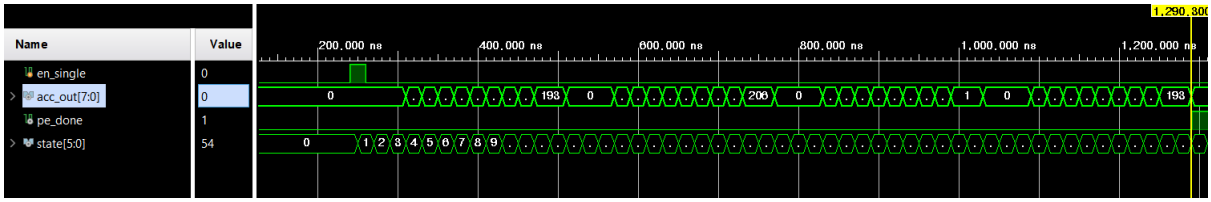
후에 Top module에서 전체 simulation testbench를 설명할 것이므로, 여기서는 결과만 설명하겠다.

```
a_1_1 = 8'd233;   a_1_2 = 8'd123;   a_1_3 = 8'd12;   a_1_4 = 8'd3;
a_2_1 = 8'd5;   a_2_2 = 8'd2;   a_2_3 = 8'd3;   a_2_4 = 8'd3;
a_3_1 = 8'd9;   a_3_2 = 8'd255;   a_3_3 = 8'd1;   a_3_4 = 8'd12;
a_4_1 = 8'd13;   a_4_2 = 8'd64;   a_4_3 = 8'd55;   a_4_4 = 8'd27;

b_1_1 = 8'd13;   b_1_2 = 8'd2;   b_1_3 = 8'd3;
b_2_1 = 8'd2;   b_2_2 = 8'd1;   b_2_3 = 8'd50;
b_3_1 = 8'd51;   b_3_2 = 8'd52;   b_3_3 = 8'd1;
```

설정해준 input과 filter 값은 다음과 같고, 예상결과는 C11= 193, C12 = 206, C21 = 1, C22 = 193이다.



Simulation 결과 en_single이 켜진 후 계산이 시작되었다. acc_out이 각 C값을 반영하여 193, 206, 1, 193이 나왔으며, 이때 en_result가 1이되어 main_controller로 결과가 전달되었음을 확인할 수 있다. C22까지 계산이 끝난후 PE_done이 1이 되며 Single mode가 성공적으로 설계되었음을 확인했다.

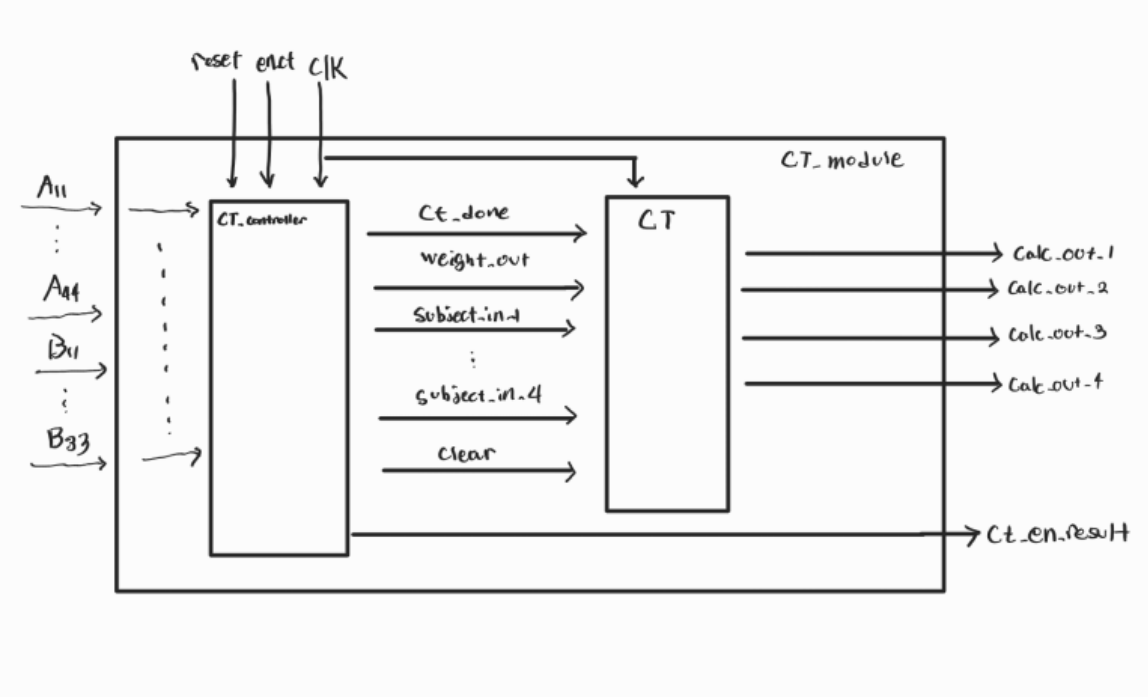


다음은 post-synthesis Functional simulation 결과이다. 240ns에 en_single bit가 켜져서 계산을 시작한 후에, 순서대로 193, 206, 1, 193이 나온다. 결과적으로 1250ns에서 마지막 결과 값이 메인 컨트롤러로 전달된다. Single PE mode에 걸린 총 시간은 1010ns이다.

6. Custom Mode

6.1 Theoretical Design

다음은 우리가 설계한 Custom module이다.



Custom module은 CT_controller와 CT로 구성되어 있다.

여기서 CT는 우리가 설계한 Custom module의 계산을 하는 역할을 하고 CT_controller는 CT에 input을 조절해주기 위한 장치이다.

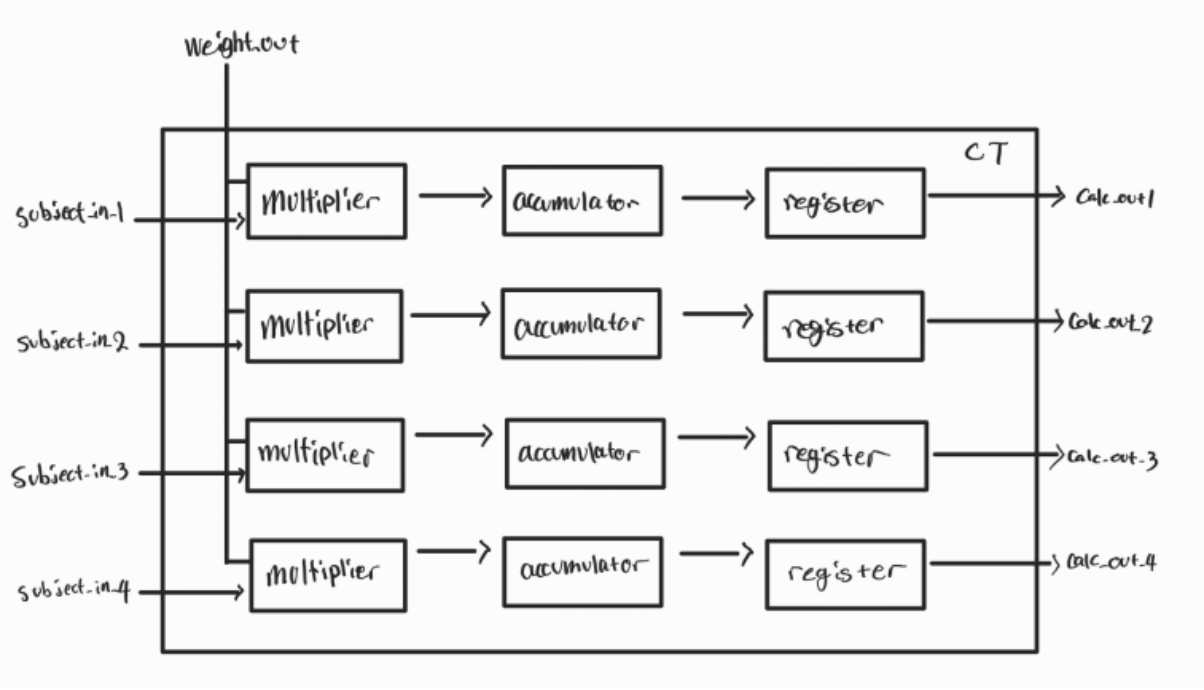
Custom module은 Main controller로부터 input으로 행렬값 16개와 필터값 9개를 받는다. 또한 en_ct를 받는데 이는 Custom module의 동작 시작을 결정하는 변수이다. 이후 설정한 State에 따라 CT에 들어가야 할 input인 weight_out과 subject_in_1,2,3,4, ct_done, clear을 조절해준다. CT에서는 주어진 input값을 이용하여 우리가 설계한 방식으로 계산을 한다. 그런다음 CT controller 4개의 값을 순차적으로 ct_result으로 출력한다.

ct_done이라는 변수는 Main controller에 연산이 끝났음을 알려주는 신호이다.

6.2 Verilog Implementation

6.2.1 CT

우선 우리가 설계한 Custom module의 방식은 다음과 같다. 1개의 필터값은 행렬 연산을 할 때 4번씩 쓰인다는 것에 초점을 맞춰 필터값을 input으로 넣고 한번에 4개의 값을 계산하는 방식을 택했다. 예를 들면 b11의 경우 c11에서는 a11과 곱해지고 c12에서는 a12와 c21에서는 a21, c22에서는 a22와 곱해진다. 이와 같은 방식으로 모든 필터값에 대해 계산을 진행한 뒤 모든 값을 더하면 9번의 연산으로 c11, c12, c21, c22의 결과값이 나온다.



CT의 연산 방식

이를 코드로 나타내면 다음과 같다.

```
//CT calculater
eight_bit_multiplier    eight_bit_multiplier1       (.a(subject_in_1), .b(weight_out), .out(multiplier_out_1));
eight_bit_multiplier    eight_bit_multiplier2       (.a(subject_in_2), .b(weight_out), .out(multiplier_out_2));
eight_bit_multiplier    eight_bit_multiplier3       (.a(subject_in_3), .b(weight_out), .out(multiplier_out_3));
eight_bit_multiplier    eight_bit_multiplier4       (.a(subject_in_4), .b(weight_out), .out(multiplier_out_4));

eight_bit_accumulator   eight_bit_accumulator1      (.clk(clk), .clear(clear), .acc_in(multiplier_out_1), .acc_out(acc_out_1));
eight_bit_accumulator   eight_bit_accumulator2      (.clk(clk), .clear(clear), .acc_in(multiplier_out_2), .acc_out(acc_out_2));
eight_bit_accumulator   eight_bit_accumulator3      (.clk(clk), .clear(clear), .acc_in(multiplier_out_3), .acc_out(acc_out_3));
eight_bit_accumulator   eight_bit_accumulator4      (.clk(clk), .clear(clear), .acc_in(multiplier_out_4), .acc_out(acc_out_4));

eight_bit_register      eight_bit_register_calc1    (.in(acc_out_1), .clk(clk), .reset(clear), .out(calc_out_1));
eight_bit_register      eight_bit_register_calc2    (.in(acc_out_2), .clk(clk), .reset(clear), .out(calc_out_2));
eight_bit_register      eight_bit_register_calc3    (.in(acc_out_3), .clk(clk), .reset(clear), .out(calc_out_3));
eight_bit_register      eight_bit_register_calc4    (.in(acc_out_4), .clk(clk), .reset(clear), .out(calc_out_4));
```

연산 방식은 input으로 받은 subject_in_x 값과 weight_out 값을 multiplier로 곱한 뒤 accumulator로 계산 결과값들을 덧셈한다. 그런 다음 register을 이용하여 output으로 내보내는 형식이다.

6.2.2 CT controller

CT controller의 목표는 CT의 input에 적절한 값을 넣어주는 것이다. 9번의 연산을 위해서는 9개의 연산 State와 앞뒤로 연산의 시작과 종료를 정해주는 State가 필요하다. 그리고 ct_result에 출력값이 순차적으로 나와야하기 때문에 9개의 state를 추가로 설정해주었다.

wait_state : 초기화 단계　　init : 연산 시작 전 대기 단계

cal1 : b11이 input인 경우　　cal2 : b21이 input인 경우　　cal3: b31이 input인 경우

cal4 : b12이 input인 경우　　cal5 : b22이 input인 경우　　cal6: b32이 input인 경우

cal7 : b13이 input인 경우　　cal8 : b23이 input인 경우　　cal9: b33이 input인 경우

delay : 계산을 기다리는 state

end_phase 1~9 : 연산이 끝난 뒤 ct_result에 순차적으로 출력값이 나오게 하기 위한 state

연산 단계에서 CT에 들어갈 input값은 다음과 같이 설정했다. cal1부터 cal9까지 진행한 뒤 그 값들을 각각 더하면 결과값인 c11, c12, c21, c22가 나온다.

```
always @ (state)
begin
    case (state)
        wait_state: // wait
        begin
            subject_in_1 <= 8'd0;
            subject_in_2 <= 8'd0;
            subject_in_3 <= 8'd0;
            subject_in_4 <= 8'd0;
            weight_out <= 8'd0;
        end

        init: // init
        begin
            subject_in_1 <= 8'd0;
            subject_in_2 <= 8'd0;
            subject_in_3 <= 8'd0;
            subject_in_4 <= 8'd0;
            weight_out <= 8'd0;
        end
        calc_1:
        begin
            subject_in_1 <= a_1_1;
            subject_in_2 <= a_1_2;
            subject_in_3 <= a_2_1;
            subject_in_4 <= a_2_2;
            weight_out <= b_1_1;
        end
        calc_2:
        begin
            subject_in_1 <= a_1_2;
            subject_in_2 <= a_1_3;
            subject_in_3 <= a_2_2;
            subject_in_4 <= a_2_3;
            weight_out <= b_2_1;
        end
        calc_3:
        begin
            subject_in_1 <= a_1_3;
            subject_in_2 <= a_1_4;
            subject_in_3 <= a_2_3;
            subject_in_4 <= a_2_4;
            weight_out <= b_3_1;
        end
        calc_4:
        begin
            subject_in_1 <= a_2_1;
            subject_in_2 <= a_2_2;
            subject_in_3 <= a_3_1;
            subject_in_4 <= a_3_2;
            weight_out <= b_1_2;
        end
        calc_5:
        begin
            subject_in_1 <= a_2_2;
            subject_in_2 <= a_2_3;
            subject_in_3 <= a_3_2;
            subject_in_4 <= a_3_3;
            weight_out <= b_2_2;
        end
        calc_6:
        begin
            subject_in_1 <= a_2_3;
            subject_in_2 <= a_2_4;
            subject_in_3 <= a_3_3;
            subject_in_4 <= a_3_4;
            weight_out <= b_3_2;
        end
        calc_7:
        begin
            subject_in_1 <= a_3_1;
            subject_in_2 <= a_3_2;
            subject_in_3 <= a_4_1;
            subject_in_4 <= a_4_2;
            weight_out <= b_1_3;
        end
        calc_8:
        begin
            subject_in_1 <= a_3_2;
            subject_in_2 <= a_3_3;
            subject_in_3 <= a_4_2;
            subject_in_4 <= a_4_3;
            weight_out <= b_2_3;
        end
        calc_9:
        begin
            subject_in_1 <= a_3_3;
            subject_in_2 <= a_3_4;
            subject_in_3 <= a_4_3;
            subject_in_4 <= a_4_4;
            weight_out <= b_3_3;
        end
        end_phase_1:
        begin
            subject_in_1 <= 8'd0;
            subject_in_2 <= 8'd0;
            subject_in_3 <= 8'd0;
            subject_in_4 <= 8'd0;
        end
        default:
        begin
            subject_in_1 <= 8'd0;
            subject_in_2 <= 8'd0;
            subject_in_3 <= 8'd0;
            subject_in_4 <= 8'd0;
        end
    endcase
```

```
always @ (state)
begin
    case (state)
        wait_state: // wait
        begin
            ct_en_result <= 1'b0;
            clear <= 1'b1;
        end
        init:  // init
        begin
            ct_en_result <= 1'b0;
            clear <= 1'b0;
        end
        end_phase_1:
        begin
            ct_en_result <= 1'b1;
            ct_done <= 1'b1;
        end
        default:
        begin
            ct_en_result <= 1'b0;
            clear <= 1'b0;
        end
    endcase
```

ct_en_result는 연산이 끝났다는 신호이다. ct_done은 CT에서의 accumulator 값이 output으로 나갈 수 있게 해주는 신호이다. 연산이 끝났을 때 ct_done을 1로 바꿔줌으로써 Main_controller에서 연산이 완료된 값들을 제어할 수 있게 해준다.

end_phase에서는 ct_en_result와 4-1 MUX를 컨트롤하는 sel 값을 조절해준다.

```
always @ (state)
begin
    case (state)
        wait_state: // wait
        begin
            ct_en_result <= 1'b0;
            clear <= 1'b1;
        end
        init:   // init
        begin
            ct_en_result <= 1'b0;
            clear <= 1'b0;
        end
        end_phase_1:
        begin
            ct_en_result <= 1'b0;
            sel <= 2'b00;
        end
        end_phase_2:
        begin
            ct_en_result <= 1'b1;
            sel <= 2'b00;
        end
        end_phase_3:
        begin
            ct_en_result <= 1'b0;
            sel <= 2'b01;
        end
        end_phase_4:
        begin
            ct_en_result <= 1'b1;
            sel <= 2'b01;
        end
        end_phase_5:
        begin
            ct_en_result <= 1'b0;
            sel <= 2'b10;
        end
        end_phase_6:
        begin
            ct_en_result <= 1'b1;
            sel <= 2'b10;
        end
        end_phase_7:
        begin
            ct_en_result <= 1'b0;
            sel <= 2'b11;
        end
        end_phase_8:
        begin
            ct_en_result <= 1'b1;
            sel <= 2'b11;
        end
        end_phase_9:
        begin
            ct_en_result <= 1'b0;
            ct_done <= 1'b1;
        end
        default:
        begin
            ct_en_result <= 1'b0;
            clear <= 1'b0;
        end
```

6.2.3 Simulation

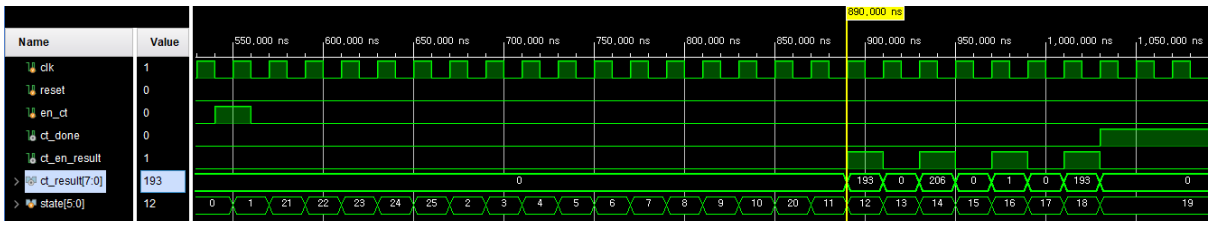
CT_module이 제대로 작동하는지 살펴보기 위해 간단한 testbench를 만들어 시뮬레이션을 돌려보았다. input 값은 다음과 같이 설정하였다. 이에 대한 예상 결과값은 C11 = 193, C12 = 206, C21 = 1, C22 = 193이다.

```
initial
begin
    clk = 1'b0;
    reset = 1'b0;
    en_ct = 1'b0;

    a_1_1 = 8'd233;   a_1_2 = 8'd123;   a_1_3 = 8'd12;   a_1_4 = 8'd3;
    a_2_1 = 8'd5;   a_2_2 = 8'd2;   a_2_3 = 8'd3;   a_2_4 = 8'd3;
    a_3_1 = 8'd9;   a_3_2 = 8'd255;   a_3_3 = 8'd1;   a_3_4 = 8'd12;
    a_4_1 = 8'd13;   a_4_2 = 8'd64;   a_4_3 = 8'd55;   a_4_4 = 8'd27;

    b_1_1 = 8'd13;   b_1_2 = 8'd2;   b_1_3 = 8'd3;
    b_2_1 = 8'd2;   b_2_2 = 8'd1;   b_2_3 = 8'd50;
    b_3_1 = 8'd51;   b_3_2 = 8'd52;   b_3_3 = 8'd1;

end
```

아래는 시뮬레이션 결과이다. reset이 0이 되고 한 clk이 지난 뒤부터 연산이 시작된다. 9 번의 연산 뒤 C11, C12, C21, C22의 연산값이 순서대로 출력되도록 하였다. 이 때 C값은 193, 206, 1, 193으로 Custom module이 정상 작동함을 알 수 있다.

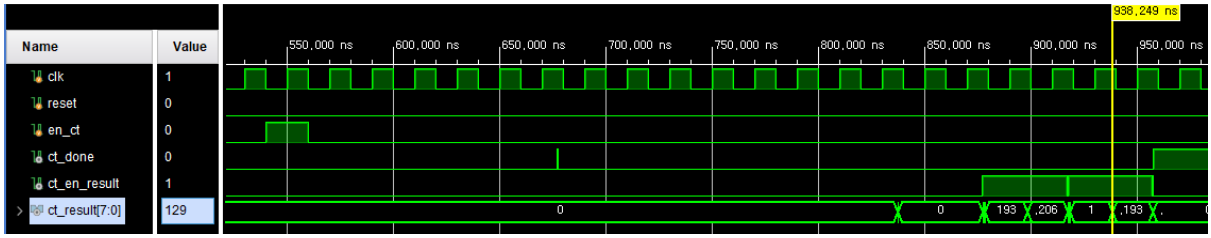
아래의 3개의 사진은 각각 Behavioral simulation, post-synthesis functional simulation, post-synthesis timing simulation이다. 모두 같은 결과를 출력하고 있다.

1. Run Behavioral Simulation



2. Run Post-Synthesis Functional Simulation

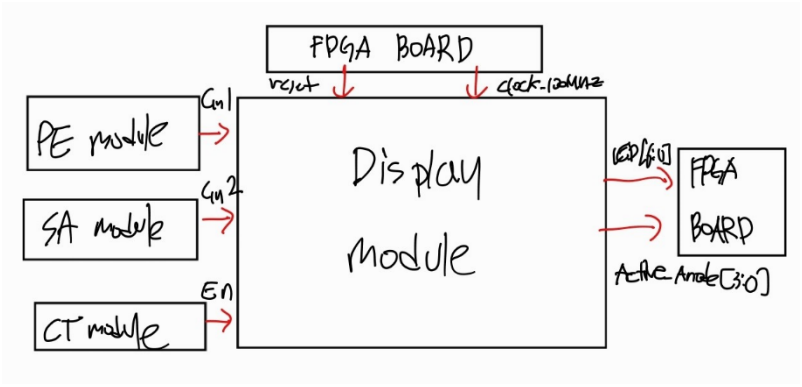
3. Run Post-Synthesis Timing Simulation



7. Display module

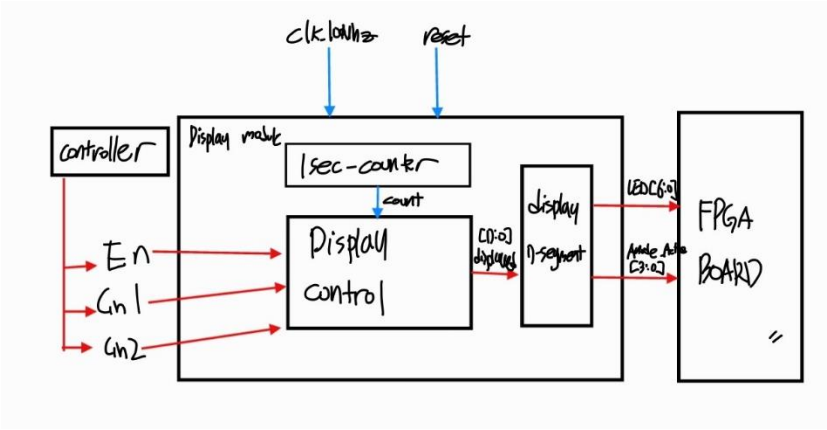
7.1 Theoretical Design

Display module은 SA와 PE의 계산결과를 input으로 받아, 1초 간격으로 FPGA Board에 출력하는 역할을 하는 module이다. 동작원리를 간단하게 설명하면,



1) Control module이 Display Module이 값을 저장해야 함을 알려주는 enable 신호를 보낸다
2) Enable 신호가 1~4회 들어오면 PE MODULE의 출력(결과값) Cin1을 4번 저장한다.
3) Enable 신호가 5~8회 들어오면 SA MODULE의 출력(결과값) Cin2을 4번 저장한다.
4) Enable 신호가 (1~8) 모두 들어온 직후부터 1초씩 count하기 시작한다.
5) 매 초마다 FPGA의 7-segment로 2~3과정을 통해 저장된 Cin1, Cin2를 출력한다.

6) Board 출력을 위해 Cin1,2를 Anode_Activate와 LED_out의 신호로 변환시켜 출력한다.,



이를 바탕으로 Display module을 세분화 해 보면, 매 1초를 세는 역할을 하는 1 second counter와 1초마다 출력 값을 변경해주는 display control, 7-segment의 형식에 맞게 입력 값을 변형해 출력해주는 display 7-segment를 structure modeling으로 작성할 예정이다.

7.2 Verilog Implementation

7.2-1) Display Control

Display machine은 Mealy machine을 사용해 구현했다. Cin 입력, 즉 PE와 SA의 연산 결과값을 input으로 총 8회 받은 뒤, 8회 저장이 완료되었음을 알리는 신호를 받으면, 저장이 완료되었음을 알리는 신호를 보내 1-sec counter가 동작하기 시작하고 reset 에서 벗어나 동작이 시작된다. 각 state는 1-sec counter에서 신호가 올 때마다 해당 값을 출력하고 다음 state로 넘어가게 된다. 이때 출력된 값은 display 7-segment module로 넘어가게 된다. Input된 Cin 값 8개가 모두 출력 된 후에는, 마지막 state에서 대기한다. (Behavioral Modeling으로 설계하였다.)

7.2-2) 1 second counter

100MHz clock을 사용하기 때문에 10^8번째의 clock들이 각각 1초를 의미한다. 10^8번째 clock마다 Display control에 1을 전달하게 된다. (Behavioral Modeling으로 설계하였다.)

7.2-3) 7-segment Conversion

Display module에서 출력되는 displayed값을 LED_OUT[6:0], Anode_Activate[3:0]에 알맞은 신호로 변환해 7-segment로 이루어진 십진수가 표시될 수 있도록 출력하는 역할을 한다. (LED에 숫자를 구현하는 코드의 경우는 교수님께서 제공해 주신 source code를 사용하였다)

7.2-4) Total_display

위에서 구현한 3개의 모듈을 연결한 top module이다. Clock과 reset, 그리고 pe11~ct22까지의 값을 input 신호로 받은 뒤 [3:0] Anode_Activate, [6:0] LED_out, [7:0] displayed 값을 output으로 내보낸다.

7.3 Test bench & Simulation

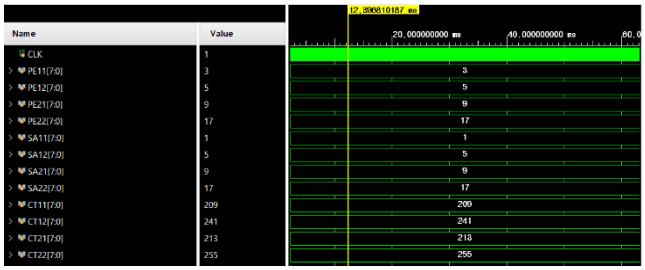
```
16      begin
17          CLK = 1'b0;
18              RESET = 1'b1;
19          PE11 = 0;   PE12 = 0;   PE21 = 0;   PE22 = 0;
20          SA11 = 0;   SA12 = 0;   SA21 = 0;   SA22 = 0;
21          CT11 = 0;   CT12 = 0;   CT21 = 0;   CT22 = 0;
22      end
```

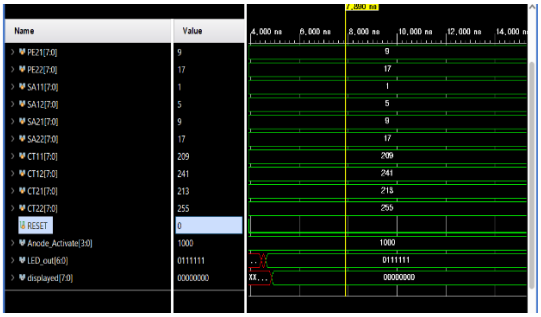
임의로 설정한 값들을 통해 display module이 올바르게 작동하는지 확인하는 test bench code를 작성하였다. 위의 코드는 초기에 reset에 1을 부여해 모든 값들을 초기화 시키고, 12개의 input을 입력해 일정 시간 간격에 따라 출력되도록 만들어 주는 test bench 코드이다.

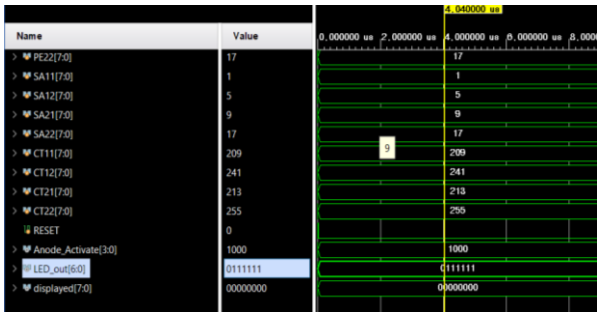
```
        begin
            #10 CLK = !CLK;
        end
    end

    initial
    begin
        //test patterns
        //Create a test pattern for the counter properly

        #20
        RESET = 0;
        PE11 = 8'b00000011; PE12 = 8'b00000101; PE21 = 8'b00001001; PE22 = 8'b00010001;
        SA11 = 8'b00000001; SA12 = 8'b00000101; SA21 = 8'b00001001; SA22 = 8'b00010001;
        CT11 = 8'b11010001; CT12 = 8'b11110001; CT21 = 8'b11010101; CT22 = 8'b11111111;
    end
```

아래의 시뮬레이션 결과는 순서대로 Behavioral, post-synthesis timing simulation, post-synthesis functional simulation을 진행한 결과이다.

입력 받은 PE와 SA값이(임의로 설정) 7 segment 십진수의 형태로 올바르게 표현되는 것을 확인할 수 있다. 또한 always문에 의해 특정 주기에 따라 1000,0100,0010,0001 자리 숫자가 반복되며 출력되는 것 또한 확인할 수 있었다. 따라서 Display module이 정상적으로 작동함을 확인할 수 있다.



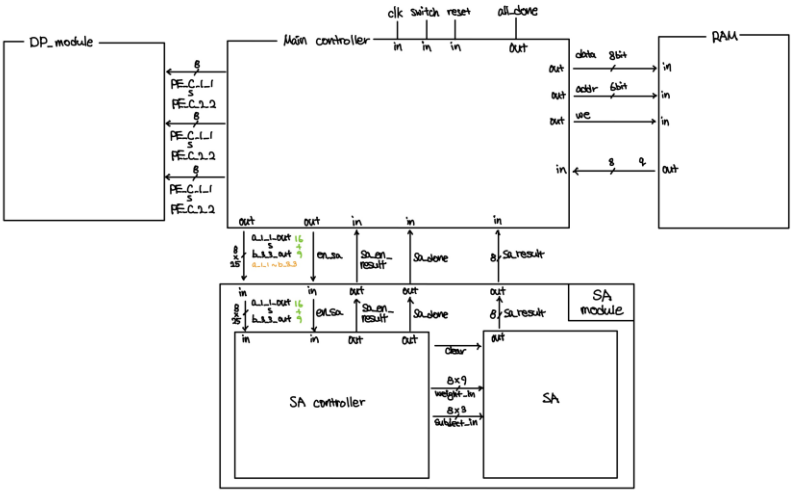




8. Top Module

8.1 Theoretical Design

Top module의 Block Diagram은 다음과 같다. 설계한 각 sub block들을 연결하였다. Single PE module 또한 PE controller, PE로 이루어져 SA module과 input과 output 신호가 동일하다. Custom module 또한 동일한 in/output을 지켜서 설계하였으므로, Diagram에서 동일하게 구성되어 있다고 생각한다.



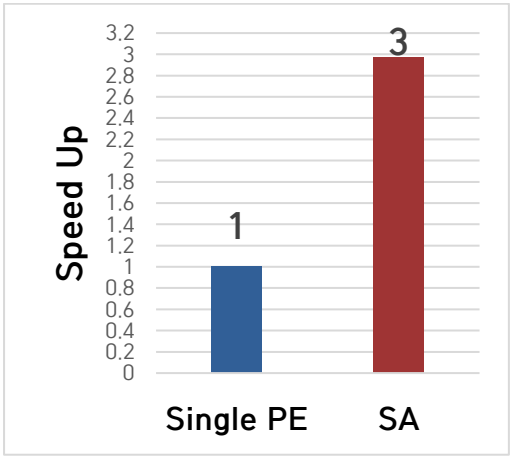
8.2 Verilog Implementation

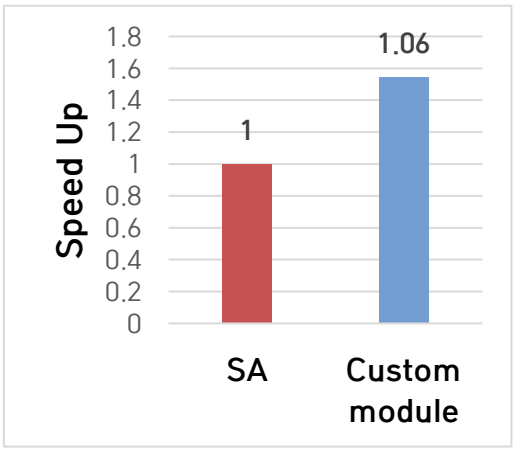
작성한 모든 모듈을 적절하게 불러왔다.

```
//SA module
SA_module SA_module(.clk(clk), .reset(reset), .en_sa(en_sa),
.a_1_1(a_1_1), .a_1_2(a_1_2), .a_1_3(a_1_3), .a_1_4(a_1_4),
.a_2_1(a_2_1), .a_2_2(a_2_2), .a_2_3(a_2_3), .a_2_4(a_2_4),
.a_3_1(a_3_1), .a_3_2(a_3_2), .a_3_3(a_3_3), .a_3_4(a_3_4),
.a_4_1(a_4_1), .a_4_2(a_4_2), .a_4_3(a_4_3), .a_4_4(a_4_4),
.b_1_1(b_1_1), .b_1_2(b_1_2), .b_1_3(b_1_3),
.b_2_1(b_2_1), .b_2_2(b_2_2), .b_2_3(b_2_3),
.b_3_1(b_3_1), .b_3_2(b_3_2), .b_3_3(b_3_3),
.sa_en_result(sa_en_result), .sa_result(sa_result), .sa_done(sa_done));

//CT module
CT_module CT_module(.clk(clk), .reset(reset), .en_ct(en_ct),
.a_1_1(a_1_1), .a_1_2(a_1_2), .a_1_3(a_1_3), .a_1_4(a_1_4),
.a_2_1(a_2_1), .a_2_2(a_2_2), .a_2_3(a_2_3), .a_2_4(a_2_4),
.a_3_1(a_3_1), .a_3_2(a_3_2), .a_3_3(a_3_3), .a_3_4(a_3_4),
.a_4_1(a_4_1), .a_4_2(a_4_2), .a_4_3(a_4_3), .a_4_4(a_4_4),
.b_1_1(b_1_1), .b_1_2(b_1_2), .b_1_3(b_1_3),
.b_2_1(b_2_1), .b_2_2(b_2_2), .b_2_3(b_2_3),
.b_3_1(b_3_1), .b_3_2(b_3_2), .b_3_3(b_3_3),
.ct_done(ct_done), .ct_en_result(ct_en_result), .ct_result(ct_result));

//DP module
display_module display_module(.clock_100Mhz(clk), .reset(reset),
.pe11(pe11), .pe12(pe12), .pe21(pe21), .pe22(pe22),
.sa11(sa11), .sa12(sa12), .sa21(sa21), .sa22(sa22),
.ct11(ct11), .ct12(ct12), .ct21(ct21), .ct22(ct22),
.Anode_Activate(Anode_Activate), .LED_out(LED_out), .displayed(displayed));
```

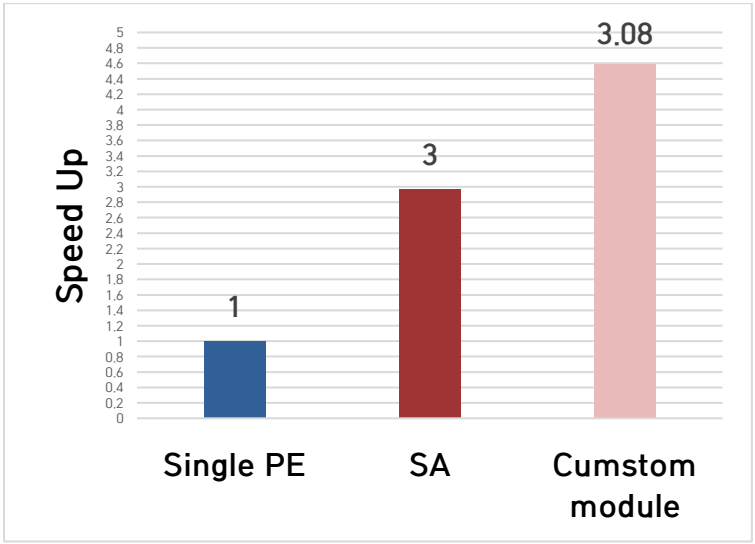
8.3 Simulation and Waveform

우리가 설계한 바와 같이 enable 신호를 이용해 각 모듈의 동작을 시작하였다. En_pe가 켜진 3260ns에 5420ns 때 Single PE mode의 계산이 종료되었다. 총 소요된 계산 시간은 2160ns이다. 마찬가지로 계산 결과 SA는 5460~6180, 즉 720ns, Custom module은 6220~7140ns, 920ns 가 소요되었다. 그러나 Custom module 에서 결과를 확실하게 반영하기 위한 delay 220ns를 넣었기 때문에, 이를 제외한 시간을 비교하면 700ns가 소요되었다.





Single PE를 1로 잡고 비교했을 때의 결과는 다음과 같다. 위의 계산 시간을 이용해 Speed up을 구해보았다. SA는 Single PE mode보다 3배가 빠르며, Custom module은 SA보다 1.06배 빠르다. 실제 계산에 사용된 Clock 은 Custom module이 9Clk, SA가 10Clk이 사용되었다. 만약 공간을 더 극단적으로 사용하여, 동시에 각 C 결과 행의 9 항을 동시에 진행하였다면 속도 측면에서 훨씬 향상되었을 것이다.



9. Collaboration

조원 명	설계한 Module
박준석	Systolic Array, Main Controller, Custom module, Top module
최장헌	Single PE module, Main Controller, Top module
노원우	Display Module, Custom module
이승우	Display Module, Top module

이번 Term project에서 가장 중요했던 부분은 각 module의 interface에 대해 이해하고 맞추는 것이었다. 특히 port명과 동작 원리 등을 맞춰야 했기 때문에 서로 연결되는 모듈들을 이해하고, 결과적으로 모든 module의 기능과 원리를 이해하는 것이 필요했다. 위에 제시한 것들을 이해하고 있어야, Top module에서 각 module이 원하는 시간에 원하는 조건으로 동작하고, 결과를 내보내게 설계할 수 있었다. 우리는 이론적인 설계가 제일 중요하다고 생각하여 초기에 다 같이 모여서 각 모듈의 역할과 동작원리를 이해하는 데에 집중했다. Single PE Mode와 SA를 위한 PE 설계를 먼저 진행하였고, 이 후 동시에 박준석 팀원은 SA module, 최장헌 팀원은 Single PE mode를 진행했다. Main controller는 가장 마지막에, 모든 동작이 확실히 정해졌을 때 의미가 있다고 판단하였기에, 노원우 팀원과 이승우 팀원은 Display module을 설계를 동시에 진행하였다. 각자 맡은 모듈을 설계하면서, Custom module에 대한 아이디어를 자유롭게 제시하고 피드백을 받을 수 있었다.

결론적으로 Top module에서 동작하기 위한 Main controller를 설계할 때의 쟁점은 메모리에서 한번에 하나의 값 만을 불러올 수 있는데, 계산 시 동시에 두 개 이상의 input을 받아야 하는 상황을 어떻게 해결할 지에 관한 문제였다. 첫번째 아이디어는 counter를 이용하여 원하는 input 개수를 확인한 후 계산을 진행하는 방식과, 두번째 아이디어는 미리 Controller에 메모리에서 필요한 값들을 옮겨 놓는 방식이었다. 우리는 두번째 방식을 채택했다. 회의 중 이런 방식은 RAM을 사용하는 의미를 퇴색시킨다는 의견도 제시되었지만, 결과적으로 post synthesis에서 timing 문제를 더 깔끔하게 해결할 수 있을 것이란 예측과 input을 동시에 더 많이 받는 Custom module에서 Clock 수와 계산 시간을 줄여줄 수 있을 것이란 점에서 결정했다. 미리 각 module의 동작신호를 Enable bit로 선언하고, 다른 모듈의 동작 순서와 과정을 자세히 이해하고 있었기에, Timing 문제는 크게 발생하지는 않았다. 그러나 각자의 module을 Top module로 통합하는 과정에서 같은 Adder, Register, F/F명이어도 안의 코드가 다른 경우로 인한 문제가 발생했다. 각자의 module에서는 자신의 하위 module을 이용하여 문제가 없었지만, Top module 단계에서는 같은 module 명인데도 다르게 움직여야 하는 것으로부터 에러가 발생했다. 이렇게 하위 module의 기능을 바꾸는 이유는 대부분 module 상 에러와 코드 간소화였다. 예를 들어 multiplier나 adder의 경우에, 필요하지 않은 Cin이나 Cout 관련 코드를 제거했는데, 이를 필요로 하는 다른 module을 함께 사용하려고 하면서 오류가 발생하는 것과 같은 문제였다. 이러한 Error를 해결하기 위해, Top module 단계에서 팀원들은 다 같이 Main controller를 이용한 전체적인 Module 동작을 하나하나 되짚어가며, 그 타이밍의 각 module과 코드가 어떻게 동작하는지 체크하였다. 이런 교차 검증을 통해 협력하여 에러를 해결할 수 있었다.

10. 결론

이번 Term project를 진행하며 활발한 협력과 의사소통, 하위 module부터 쌓아올리는 것의 중요성을 느꼈다. 여러 사람이 하나의 module을 설계하는 Term project다. 각자가 맡은 기능을 구현하는 것 뿐만이 아닌, 구현하면서 지켜야하는 제한, 동작 조건, 그리고 구현을 위해 사용하는 하위 module들과 port 명 등의 통일이 얼마나 중요한지에 대해 깨달을 수 있었다. 이는 이론 설계 단계에서 서로 합의되고 다른 block들의 동작까지 이해하고 있는 것이 기반되어야 하며, 설계 중 예상치 못한 오류가 발생하거나, 이를 해결하기 위해 기존에 합의된 규칙을 어쩔 수 없이 바꿔야할 경우, 이에 관련된 모든 팀원이 이를 인지하고 있음이 중요하다. 이론 설계 단계에서 많은 시간과 규칙을 정했음에도, 각자 맡아 설계한 module을 합치는 과정에서 디버깅 시간이 매우 크게 소모되었다. 또한 module명이 같음에도 코드의 동작이 조금씩 다른 것은 파일명으로는 구별이 되지 않기 때문에 치명적이었다. 그러나 이런 디버깅과 서로 활발한 논의를 통해 문제를 해결하면서 각각의 module, 나아가서는 전체적인 모듈에 대해 더 깊이 이해하고, 설계의 효율을 향상시킬 수 있었다.