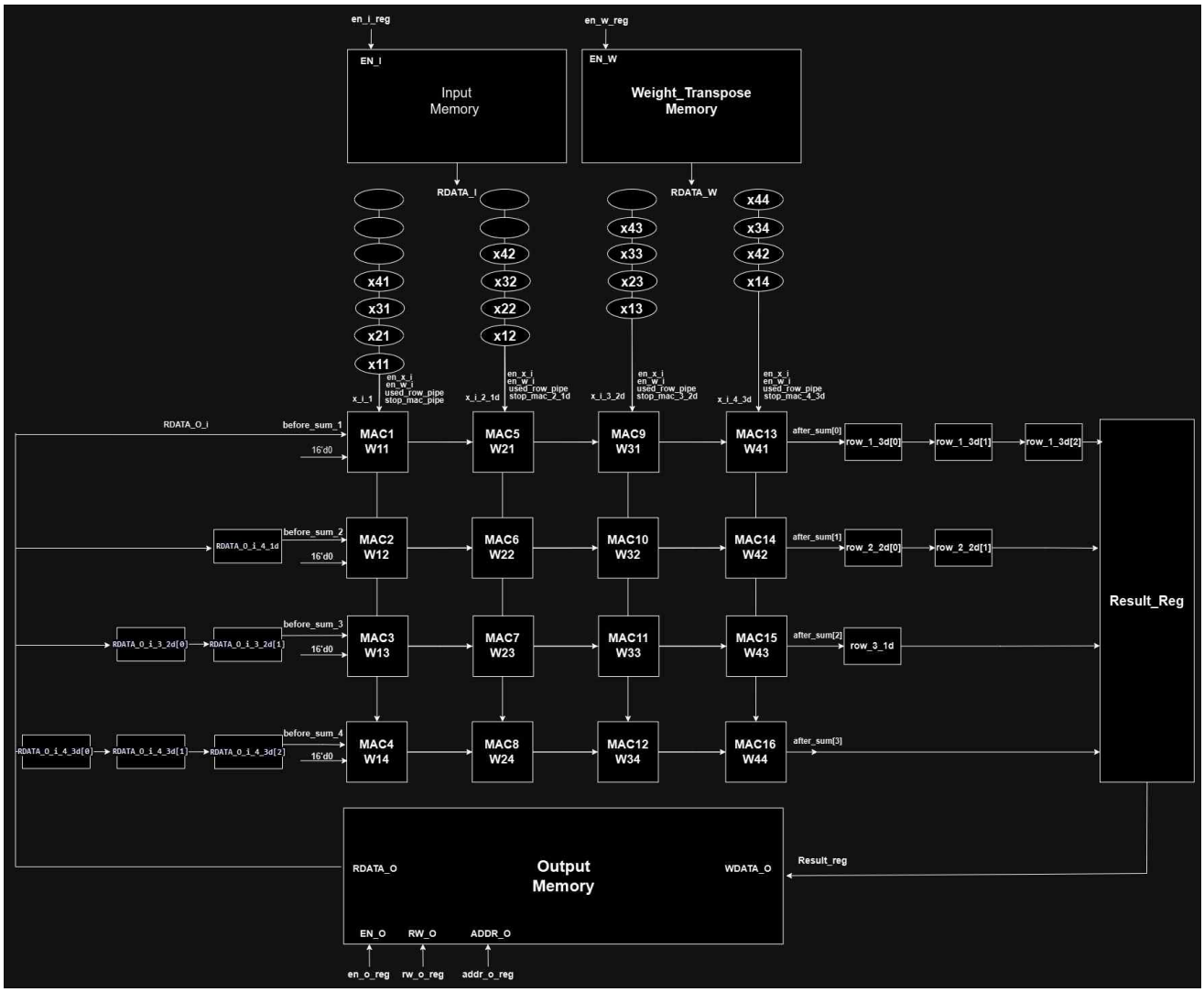
Project 1 - Matrix Multiplication Unit

8조 2020105400 김지원
2020105420 인선우

1. Schematic



16개의 MAC unit으로 구성된 Weight stationary 4x4 systolic MAC array를 구현했다. 이러한 구조에서 Input은 열마다 1-cycle delay를 주고 입력되며, Output은 행마다 1-cycle delay를 주고 출력되어야하기 때문에 MAC array의 입력단과 출력단에 Delay를 위한 Register block들이 존재한다. Delay를 받고 최종적으로 출력되는 64비트 Result는 Result_reg에 저장되고, ADDR_O에 의해 output memory에 저장된다.

2. Design direction

해당 프로젝트에서 주어진 4X4 MAC Array에 대한 최대 8x8까지의 크기를 가질 수 있는 Input과 Weight_transpose를 고려해주기 위해 8x8 Data를 0, 1, 2, 3사분면으로 구분하고 이를 where_w, where_i로 나타낸다. Weight stationary 4x4 systolic MAC array로 구현하면 고정된 4x4 Weight에 대해 Input은 유동적으로 1~8개를 한 번에 넣어서 연산할 수 있다는 장점이 있다. 이때 최초로 연산이 이루어지는 위치는 T<4, N<4, M<4인 부분으로 하고, 한 번의 Weight Loading을 통해 T행 Input을 모두 연산하여 T행의 Output을 얻을 수 있다. 그림의 회색 영역으로 확인할 수 있다. Transpose된 Weight에 대해 N≤4에 해당하는 Weight는 N≤4에 Input과 연산이 가능하다. 그러므로 Input은 크게 N≤4과 N>4로 2개의 Block으로 나누고, Weight는 한 번에 4x4로 Loading되므로 N과 M에 대해서 4개의 사분면으로 나눈다. 이를 바탕으로 M, N, T가 4보다 큰지 작은지에 대한 변수인 Stage로 나누어서 MNT input에 대한 type을 Stage로 정의하고, FSM의 next state를 결정하는 식으로 구현했다.

```
wire       [2:0]   stage;
assign  stage   =   {(M>4'd4),(N>4'd4),(T>4'd4)};
/*
  stage 0 | 3'b000 | M≤4, N≤4, T≤4일 때 1CYCLE                | 사분면 (input, weight_transpoe)
                                                              | (1,1)
        1 | 3'b001 | M≤4, N≤4, T>4일 때 1CYCLE                | (1,1), (2,1)
        2 | 3'b010 | M≤4, N>4, T≤4일 때 1CYCLE + LOAD_W       | (1,1), (0,0)
        3 | 3'b011 | M≤4, N>4, T>4일 때 1CYCLE + LOAD_W       | (1,1), (2,1), (1,0), (2,0)
        4 | 3'b100 | M>4, N≤4, T≤4일 때 1CYCLE + LOAD_W       | (1,1), (1,2)
        5 | 3'b101 | M>4, N≤4, T>4일 때 1CYCLE + LOAD_W       | (1,1), (2,1), (1,2), (2,2)
        6 | 3'b110 | M>4, N>4, T≤4일 때 1CYCLE + LOAD_W*3     | (1,1), (1,2), 0(,0), (0,3)
        7 | 3'b111 | M>4, N>4, T>4일 때 1CYCLE + LOAD_W*3     | (1,1), (2,1), (1,2), (2,2), (0,0), (3,0), (0,3), (3,3)
```

그림 2 (MNT에 따른 Stage 경우의 수)

1) M>4

M>4이면, 4X4 MAC Array를 벗어나는 Weight가 필요하므로 필연적으로 Weight loading을 2회 이상 진행해야한다.

2) N>4

N>4는 Input과 Weight 모두 4X4 MAC Array를 벗어나므로, M과 마찬가지로 Weight loading을 2회 이상 진행해야 한다. 또한 행렬 곱에서 N≤4인 경우와, N>4인 경우가 더해져서 하나의 Output 값을 결정하므로, N>4를 저장할 때 기존에 output memory에 저장된 N≤4에 대한 output를 read해온 다음, N>4의 연산 결과와 더하고 같은 주소에 다시 저장해야한다. (다음과 같이 회색 영역이 먼저 곱해져 Output Mem에 저장되고, 노란색 영역 연산 시 회색 영역 값을 Read하고 더해서 저장해야한다.)

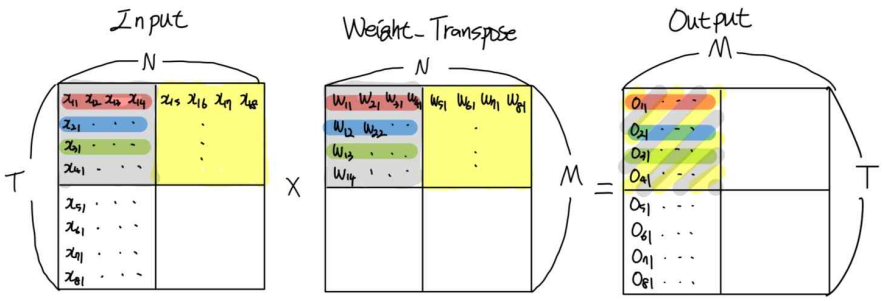


그림 3 (N>4인 경우)

3) T>4

T>4일 때, loading된 Weight에 대해 T행의 Input을 연속하여 연산한다.

앞서 M과 N에 따라 Stage가 결정되는 것을 언급했다. 이때. N≤4인 경우에는 64비트 Input, Weight data에 대해 상위 32비트만 Loading해야 한다. N>4인 경우에는 하위 32비트만 Loading해야 한다. stage를 통해 고려하면 Memory로부터 불러온 64-bits Weight, Input data에 대해서 상위 32-bits를 사용할지, 하위 32-bits를 사용할지 알 수 있다. 따라서 후에 기술하겠지만, STORE_O state에서 지정된 where_w, where_i를 바탕으로 아래와 같이 MAC Array에서 사용한 Input과 Weight를 upper_I와 lower_I 중에, upper_W와 lower_W 중에 사용할 값을 결정할 수 있다.

```
reg     [63:0]  I;
wire    [31:0]  x_i;
wire    [31:0]  upper_I, lower_I;

always @(*) begin
    case (N)
        4'd1: I =   {RDATA_I[63:56], {56{1'b0}}};
        4'd2: I =   {RDATA_I[63:48], {48{1'b0}}};
        4'd3: I =   {RDATA_I[63:40], {40{1'b0}}};
        4'd4: I =   {RDATA_I[63:32], {32{1'b0}}};
        4'd5: I =   {RDATA_I[63:24], {24{1'b0}}};
        4'd6: I =   {RDATA_I[63:16], {16{1'b0}}};
        4'd7: I =   {RDATA_I[63:8],  {8{1'b0}}};
        4'd8: I =   RDATA_I;
        default: I = 64'd0; // Handles invalid N
    endcase
end

assign  upper_I =   I[63:32];
assign  lower_I =   I[31:0];
assign  x_i     =   (where_i_reg==2'd1 || where_i_reg==2'd2) ? upper_I : lower_I;
```

그림 4 (x_i 결정 조건)

```
reg     [63:0]  W;
wire    [31:0]  w_i;
wire    [31:0]  upper_W, lower_W;

always @(*) begin
    case (N)
        4'd1: W =   {RDATA_W[63:56], {56{1'b0}}};
        4'd2: W =   {RDATA_W[63:48], {48{1'b0}}};
        4'd3: W =   {RDATA_W[63:40], {40{1'b0}}};
        4'd4: W =   {RDATA_W[63:32], {32{1'b0}}};
        4'd5: W =   {RDATA_W[63:24], {24{1'b0}}};
        4'd6: W =   {RDATA_W[63:16], {16{1'b0}}};
        4'd7: W =   {RDATA_W[63:8],  {8{1'b0}}};
        4'd8: W =   RDATA_W;
        default: W = 64'd0; // Handles invalid N
    endcase
end

assign  upper_W =   W[63:32];
assign  lower_W =   W[31:0];
assign  w_i     =   (where_w_reg==2'd1 || where_w_reg==2'd2) ? upper_W : lower_W;
```

그림 5 (w_i 결정 조건)

```
reg     [2:0]   load_w_times, load_i_times;
reg     [3:0]   compute_cycle;
always @(*) begin
    if((N>=4)) begin
        if(where_w_reg==2'd0||where_w_reg==2'd3) begin
            compute_cycle   =   (N-4'd4) + 4'd5;
        end else begin
            compute_cycle   =   4'd9;
        end
    end else begin
        compute_cycle   =   N + 4'd5;
    end
end
always @(*) begin
    case (state)
        IDLE:       next_state  =   (START)                 ? READY     :   IDLE;
        STOP:       next_state  =   (START && ~prev_START)  ? READY     :   STOP;
        READY:      next_state  =   (START)                 ? LOAD_W    :   READY;
        LOAD_W:     next_state  =   (addr_w_reg[1:0]==2'b00) ? LOAD_I    :   LOAD_W;
        LOAD_I:     next_state  =   (cycle_count==compute_cycle)         ? STORE_O   :   LOAD_I;
```

그림 6 (IDLE, STOP, READY, LOAD_W, LOAD_I State 전환)

3. State Design, Datapath

우리는 State를 크게 **IDLE, READY, LOAD_W, LOAD_I, STORE_O, STOP(6단계)**로 나누었다.

1) IDLE

RSTN에 따라 시스템이 초기화되고 대기 중, START Signal이 들어오면 READY State로 이동한다.

2) READY

Weight Load 준비 단계로 초기 제어 신호와 Weight 주소를 설정한다. START Signal이 들어오고 최초 연산은 Input과 Weight 모두 1번에 해당하는 사분면 값을 가져오고 다음 연산부터는 STORE_O 단계에서 정해진 Weight,

Input 값을 사용한다. LOAD_W State를 몇 번 거쳤는지 확인하는 load_w_times를 증가한다.

3) LOAD_W

load_w state가 되면 MNT에서 특히 N의 값에 따라 MAC Weight_Transpose Memory로부터 전치된 Weight를 불러오고, Weight를 위에서 아래 방향으로 MAC에 Load한다. 위에서 아래로 고정하므로 ADDR_W는 3b'x11부터 1씩 줄어들며 3b'x00까지 불러온다.

따라서 addr_w_reg[1:0]==2'b00임을 확인해 LOAD_I State로 전환을 결정할 수 있다.

4) LOAD_I

T행에 해당하는 Input을 불러옴과 동시에 연산을 진행한다. 따라서 LOAD_I State가 유지될 때 마다 Count (cycle_count)를 하여 T개의 Input을 전부 불러오고 RESULT가 생성되면 STORE_O로 전환한다. T개의 Input을 load 하므로 cycle_count==T일 때 STORE_O로 전환하는 것이 기본적이나 후에 기술하겠지만 N<4이거나 4<N<8일 경우 연산결과가 더 일찍 나오므로 compute_cycle로 고려하여 state을 진행한다.

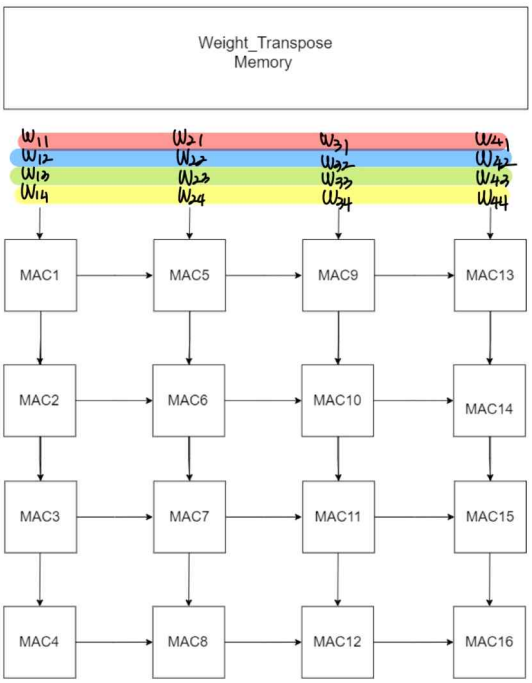


그림 7 (Weight Load)

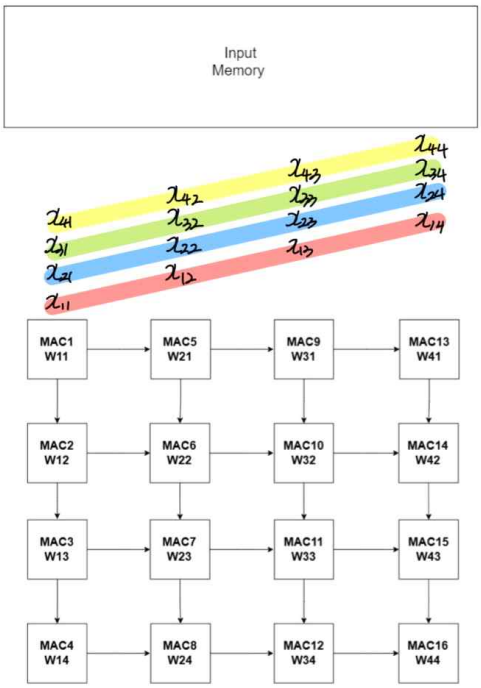


그림 8 (Input Load)

5) STORE_O

```
3'd7: begin //M>4, N>4, T>4
    if(store_count==T) begin
        if(load_w_times==1) begin
            next_state  =  READY;
            where_w     =  2'd2;
            where_i     =  2'd1;
        end else if(load_w_times==2) begin
            next_state  =  READY;
            where_w     =  2'd0;
            where_i     =  2'd0;
        end else if(load_w_times==3) begin
            next_state  =  READY;
            where_w     =  2'd3;
            where_i     =  2'd0;
        end else begin
            next_state  =  STOP;
            where_w     =  2'd1;
            where_i     =  2'd1;
        end
    end else begin
        next_state      =  STORE_O;
    end
end
```

그림 9 (stage==3'd7일 때 STORE_O state 전환)

4X4 MAC 연산 결과를 Output memory에 저장하는 단계로, T개의 Input에 대응하여 T개의 Ountput이 생성되므로 STORE_O State가 진행될 때마다 store_count를 증가시키고 T와 같아지면 다음 State로 이동한다. 이전에 Stage에 따른 연산 과정을 알고 있으므로 load_w_times를 비교하여 다음 연산 시 사용할 Weight와 Input 값을 (where_w, where_i)를 통해 전달한다. 모든 연산이 종료되면 최종적으로 STOP State로 이동하여 추가적인 연산을 배제한다.

6) STOP

모든 연산이 종료되면 추가 Cycle이 발생하지 않게 STOP State로 이동.

output memory은 8x8 input weight에 대해 총 16개의 64-bits data를 가지고 있다.

이때 STORE_O 동작이 끝나고도 output memory에 저장되지 않은 부분에 32'd0을 저장한다.

4. Control Unit Design

MNT에 따른 각 State에서 고려해야 할 부분을 중점적으로 생각하여 Control Unit을 구축했다.

1) READY

LOAD_W로 넘어가기 전에 변수를 초기화한다. 초기 ADDR_W를 설정.

2) LOAD_W

Weight Load 시 고려할 사항은 M, N 값을 바탕으로 연산하지 않는 MAC을 결정해야하므로 다음과 같이 구성했다.

I) used_row_reg: MAC Array에 Weight Load 시 처음 Load되는 Weight와 같이 입력되어 MAC에서 used_row_reg에 저장된다. M이 4보다 작거나 4<M<8에서 2번째 Load되는 Weight에서 연산을 하지 않는 행을 배제하기 위한 control이다. used_row_reg가 0이면 MAC 연산이 이루어지지 않고 RESULT로 0을 출력한다.

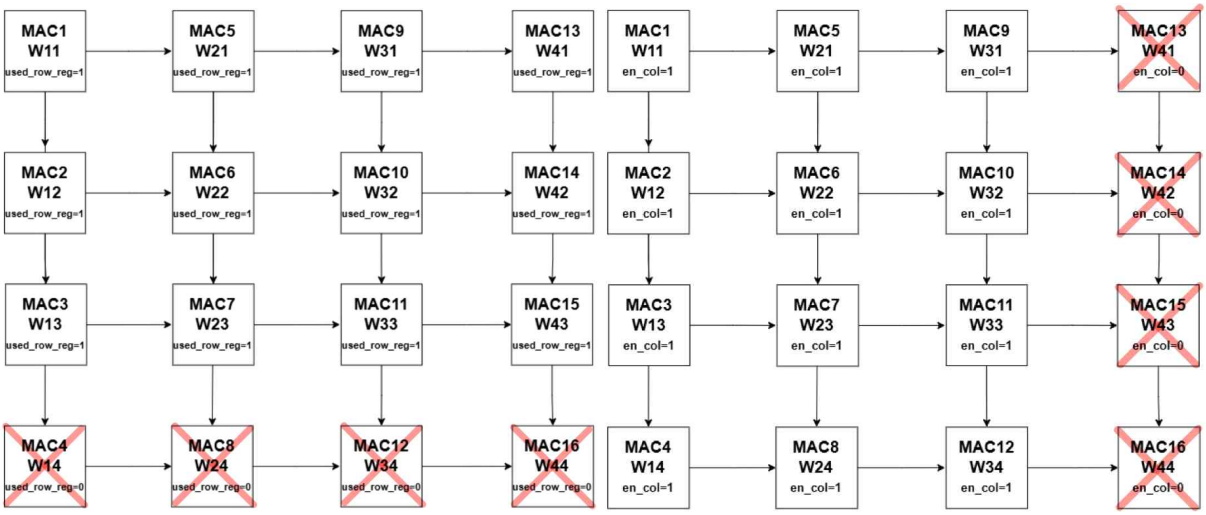


그림 10 used_row_reg에 따른 array

그림 11 en_col에 따른 array

```
if(M>4) begin
    used_row_reg    <=  (addr_w_reg[1:0] <= M[1:0]-2'd1) ? 1'b1 : ~addr_w_reg[2];
end else begin
    used_row_reg    <=  (addr_w_reg[1:0] <= M[1:0]-2'd1) ? 1'b1 : 1'b0;
end
```

그림 12 used_row_reg

ii)en_col: used_row_reg와 비슷하게, N값을 참조하여 MAC Array에서 연산을 하지않는 열을 지정할 수 있다. Weight Load 시 Weight와 같이 MAC에 고정한다.

```
reg     [3:0]   en_col;
wire    [3:0]   shift;
assign  shift =  (where_w_reg==2'd0||where_w_reg==2'd3) ? (4'd8 - N) : 4'd0;
always @(*) begin
    if(EN_W) begin
        if(N>4) begin
            en_col  =   4'b1111 << shift;
        end else begin
            en_col  =   4'b1111 << (4'd4-N);
        end
    end else begin
        en_col = 4'd0;
    end
end
```

그림 13 en_col

4) LOAD_I

i) compute_cycle, cycle count: compute_cycle, cycle count은 STORE_O로 state를 변경하기 위해 사용한다. cycle count은 input memory에서 input data를 불러오기 위한 EN_I가 켜질 때부터의 clock cycle 개수이다. compute_cycle은 MAC array에서 RESULT가 출력될 때까지 걸리는 clock cycle이며, cycle count가 이와 같을 때 input 값을 array에 넣는 en_x_i를 0으로 만들고 STORE_O로 state가 바뀐다. 이때, Result가 나올 때까지 걸리는 clock cycle은 N과 관련이 있다. RESULT는 en_col에 의해 비활성화되는 MAC array 열은 제외하고 활성화된 열을 기준으로 출력하기 때문에 mac array에서 input이 들어온 출력까지 (N-1) cycle이 발생한다. 이때 EN_I가 1이 되면서 input memory에서 ADDR_I에 대한 input data가 나오는 1-cycle, RDATA_I가 mac array의 I 레지스터에 저장하는 1-cycle, 출력단의 Delay 레지스터 블록에 의해 3-cycle이 발생하므로, compute_cycle은 (N+4)가 된다.

```
reg     [3:0]   compute_cycle;
always @(*) begin
    if((N>=4)) begin
        if(where_w_reg==2'd0||where_w_reg==2'd3) begin
            compute_cycle   =   (N-4'd4) + 4'd5;
        end else begin
            compute_cycle   =   4'd9;
        end
    end else begin
        compute_cycle       =   N + 4'd5;
    end
end
```

그림 14 (compute_cycle 조건)

```
end else if (cycle_count==T) begin
    stop_mac_reg    <=  1'b1;
    en_i_reg        <=  1'b0;
    addr_i_reg      <=  3'd0;
    cycle_count     <=  cycle_count + 4'd1;
    en_o_reg        <=  1'b0;
    addr_o_reg      <=  {3'd0,(where_w_reg==2'd2||where_w_reg==2'd3)};
end else if (cycle_count==compute_cycle) begin
    load_i_times    <=  load_i_times + 2'd1;
    stop_mac_reg    <=  1'b1;
    en_i_reg        <=  1'b0;
    addr_i_reg      <=  3'd0;
    cycle_count     <=  4'd0;
```

그림 15 (cycle_count, compute_cycle)

ii) stop_mac_reg: 모든 Input이 입력된 다음에 MAC unit의 불필요한 연산을 막기 위해 사용된다. MAC array에 입력되는 마지막 INPUT과 같이 타이밍에 입력하여 MAC unit에서 다음 cycle에 레지스터에 이를 저장한다. 따라서 input과 같이 열마다 delay를 주고 입력한다. 그 다음 input이 입력되어도 MAC unit 안에서 stop_mac_reg가 1이면 연산이 이루어지지 않고, After_sum으로 0을 출력한다. stop_mac 자체는 cycle마다 mac array에 계속해서 들어가므로, 다시 input이 입력되기 전 cycle에 0을 넣어주어 다시 RESULT를 출력할 수 있도록 구현했다.

```
stop_mac_4_3d[0] <=  stop_mac_pipe;
stop_mac_4_3d[1] <=  stop_mac_4_3d[0];
stop_mac_4_3d[2] <=  stop_mac_4_3d[1];

stop_mac_3_2d[0] <=  stop_mac_pipe;
stop_mac_3_2d[1] <=  stop_mac_3_2d[0];

stop_mac_2_1d    <=  stop_mac_pipe;
```

그림 16 (stop_mac_reg 입력 시 delay)

iii) overwrite_sig: 앞서 언급했듯이 N>4일 경우에 N≤4인 Weight와 input 연산하고, N>4인 Weight와 input에 대해서 연산을 하는데 이 과정에서 N≤4에 해당하는 output memory를 read하여 RESULT와 더해준 다음 같은 주소에 다시 저장해야한다. 아래와 같이 stage와 load_w_times 값을 기준으로 N>4인 연산을 실행하는 flag 신호이다. en_o_reg에 overwrite_sig를 할당하여 1일 경우, RW_O=0로 Output Memory에서 RDATA_O를 불러온다. 그 다음 mac array에 입력되는 input과 같은 타이밍에 RDATA_O를 MAC array 1열의 before sum으로 넣어주어 RESULT가 이를 반영하도록 구현했다. 이때 MAC unit에서 Input이 cycle마다 행 단위로 내려오므로 Delay 레지스터 블록을 두어 타이밍을 맞춰주었다.

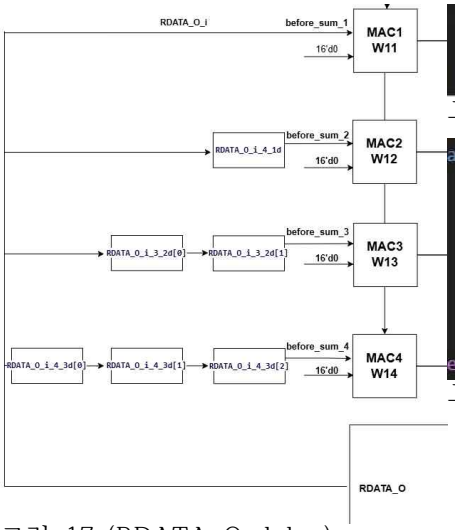


그림 17 (RDATA_O delay)

```
en_o_reg        <=  overwrite_sig;
rw_o_reg        <=  1'b0; // Read Mode
addr_o_reg      <=  {3'd0,(where_w_reg==2'd2||where_w_reg==2'd3)};
```

그림 18 (en_o_reg에 overwrite_sig 할당)

```
always @(*) begin
    if((stage==3'd2)||(stage==3'd3)) begin
        overwrite_sig   =   (load_w_times>=3'd2);
    end else if((stage==3'd6)||(stage==3'd7)) begin
        overwrite_sig   =   (load_w_times>=3'd3);
    end else begin
        overwrite_sig   =   1'b0;
    end
end
```

그림 19 (overwrite_sig 조건)

5) STORE_O

i) where_w_reg, where_i_reg: stage에 따라 다른 Weight를 Loading하여 연산을 다시 시행하는 경우에 Loading할 input과 weight의 위치를 정해주기 위한 signal이다. stage에 따라 Loading해야 하는 Weight가 정해져 있기 때문에 load_w_times를 바탕으로 다음 weight loading을 결정한다. 이 신호는 input, weight에서 상위 32비트를 쓸지, 하위 32비트를 쓸 지에도 사용된다.

ii) store_count: STORE_O operation에 걸리는 clock cycle로 timing을 조절하기 위한 신호이다. 한 번의 연산에서 T개의 Input에 따라 T개의 Output이 저장되므로 store_count==T가 되면 저장을 종료하고 다음 state로 넘어감을 알려준다. EN_O가 켜진 시점부터 cycle마다 1씩 증가하므로 store_count가 T가 되면 STORE_O가 종료된다,

```
3'd7: begin
    if(store_count==T) begin
        if(load_w_times==1) begin
            where_w_reg     <=  2'd2;
            where_i_reg     <=  2'd1;
        end else if(load_w_times==2) begin
            where_w_reg     <=  2'd0;
            where_i_reg     <=  2'd0;
        end else if(load_w_times==3) begin
            where_w_reg     <=  2'd3;
            where_i_reg     <=  2'd0;
        end else begin
            where_w_reg     <=  2'd1;
            where_i_reg     <=  2'd1;
        end
    end else begin
        where_w_reg     <=  where_w_reg;
        where_i_reg     <=  where_i_reg;
    end
end
```

그림 20 (stage==3'd7 where_reg

```
if(store_count==4'd0) begin
    en_o_reg    <=  1'b1;
    rw_o_reg    <=  1'b1; // Write Mode
    wdata_o_reg <=  RESULT_reg;
    addr_o_reg  <=  {3'd0,(where_w_reg==2'd2||where_w_reg==2'd3)};
    store_count <=  4'd1;
end else if(store_count < T) begin
    addr_o_reg  <=  addr_o_reg + 4'd2;
    store_count <=  store_count + 4'd1;
    wdata_o_reg <=  RESULT_reg;
end else if(store_count == T) begin
    en_o_reg    <=  1'b0;
    rw_o_reg    <=  1'b0; // Write 종료
    addr_o_reg  <=  addr_o_reg + 4'd2;
    store_count <=  4'd0;
    wdata_o_reg <=  RESULT_reg;
end else begin
    store_count <=  4'd0;
    en_o_reg    <=  1'b0;
    rw_o_reg    <=  1'b0;
end
```

그림 21 (store_count를 통해 저장 주소와 저장 종료 시점 update)

6) STOP

```
3'd4, 3'd6: begin
    // T 1 & 2
    if(save) begin
        en_o_reg    <= 1'b1;
        rw_o_reg    <= 1'b1;
        wdata_o_reg <= 64'd0;
        addr_o_reg  <= T * 4'd2;
        save        <= 1'b0;
    end else if(addr_o_reg == 4'd14)begin
        addr_o_reg  <= 4'd2*T + 4'd1;
    end else if(addr_o_reg == 4'd15)begin
        en_o_reg    <= 1'b0;
    end else begin
        addr_o_reg  <= addr_o_reg  +  4'd2;
    end
end
```

그림 22 stage 4, 6

```
3'd5, 3'd7: begin
    // T 1 & 2
    if(T<4'd8) begin
        if(save) begin
            en_o_reg    <= 1'b1;
            rw_o_reg    <= 1'b1;
            wdata_o_reg <= 64'd0;
            addr_o_reg  <= T * 4'd2;
            save        <= 1'b0;
        end else if(addr_o_reg == 4'd14)begin
            addr_o_reg  <= 4'd2 * T + 4'd1;
        end else if(addr_o_reg == 4'd15)begin
            en_o_reg    <= 1'b0;
        end else begin
            addr_o_reg  <= addr_o_reg  +  4'd2;
        end
    end else begin
        en_o_reg    <= 1'b0;
        rw_o_reg    <= 1'b0;
        addr_o_reg  <= 4'd0;
    end
end
```

그림 23 stage 5, 7

T행의 input에 대해서 연산이 끝나고 나면 stage에 따라 저장되지 않은 output memory address가 발생한다. M≤4인 부분은 address가 0, 2, 4, 8, 10, 12, 14, M>4인 부분은 address가 1, 3, 5, 7, 9, 11, 13, 15이다.

stage 0, 2, 4, 6 : 짝수 부분 T개만큼 저장, 홀수 부분은 저장 X

stage 1, 3, 5, 7 : 짝수, 홀수 부분 모두 T개만큼 저장

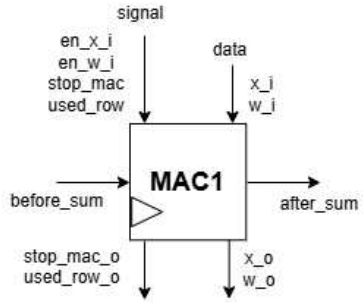
stage 4, 6 : 짝수 부분 T개만큼 저장, 홀수 부분은 저장 X

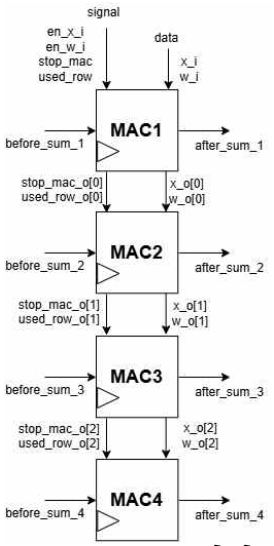
stage 5, 7 : 짝수, 홀수 부분 모두 T개만큼 저장

addr_o_reg를 짝수인 2*T부터 cycle마다 2씩 증가시켜 64'd0을 저장하고 14에 도달하면, addr_o_reg를 홀수 2*T+1로 지정하고 cycle마다 2씩 증가시켜 마지막인 15가 될 때까지 저장하는 식으로 구현했다.

```
always @(*) begin
    if(state==STOP) begin
        WDATA_O_reg =   64'd0;
    end else begin
        WDATA_O_reg =   ((T >= compute_cycle)) ? wdata_o_reg : RESULT_reg;
    end
end
```

5. MAC array



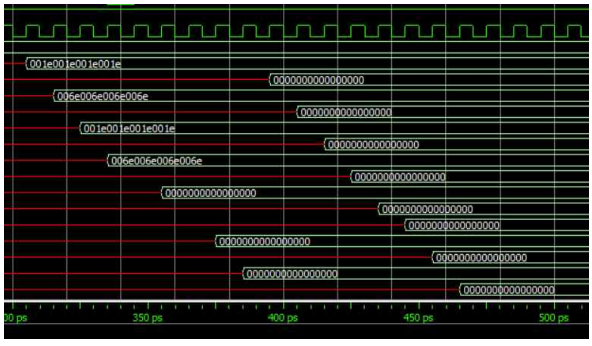



```
always @(*) begin
    case(en_w_i_reg)
        4'b1111 : begin
            after_sum[0] = after_sum_4[0];
            after_sum[1] = after_sum_4[1];
            after_sum[2] = after_sum_4[2];
            after_sum[3] = after_sum_4[3];
        end
        4'b1110 : begin
            after_sum[0] = after_sum_3[0];
            after_sum[1] = after_sum_3[1];
            after_sum[2] = after_sum_3[2];
            after_sum[3] = after_sum_3[3];
        end
        4'b1100 : begin
            after_sum[0] = after_sum_2[0];
            after_sum[1] = after_sum_2[1];
            after_sum[2] = after_sum_2[2];
            after_sum[3] = after_sum_2[3];
        end
        4'b1000 : begin
            after_sum[0] = after_sum_1[0];
            after_sum[1] = after_sum_1[1];
            after_sum[2] = after_sum_1[2];
            after_sum[3] = after_sum_1[3];
        end
        default : begin
            after_sum[0] = after_sum_4[0];
            after_sum[1] = after_sum_4[1];
            after_sum[2] = after_sum_4[2];
            after_sum[3] = after_sum_4[3];
        end
    endcase
end
```

en col에 따라 output을 출력하는 열을 다르게 해주는 식으로 구현했다.

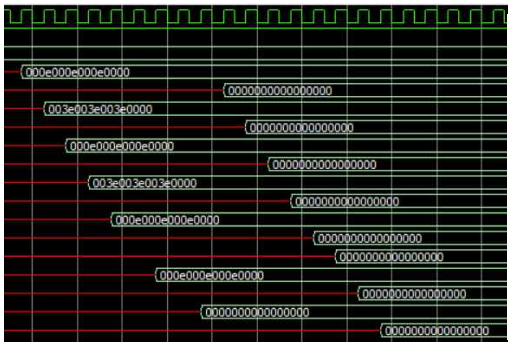
6. Testbench 결과

stage 0 : MNT=12'h444




```
Selected Matrix 1 (T x N):
[[ 1  2  3  4]
 [ 9 10 11 12]
 [ 1  2  3  4]
 [ 9 10 11 12]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]
 [4 4 4 4]]

Resulting Matrix in Hexadecimal:
0x1e 0x1e 0x1e 0x1e
0x6e 0x6e 0x6e 0x6e
0x1e 0x1e 0x1e 0x1e
0x6e 0x6e 0x6e 0x6e
```

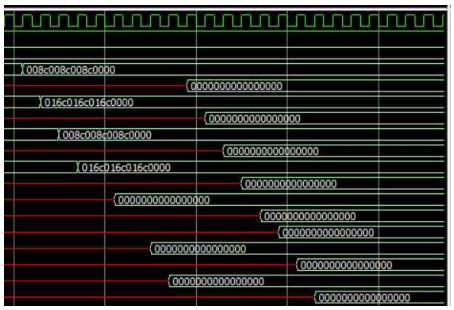
stage 1 : MNT=12'h337



```
Selected Matrix 1 (T x N):
[[ 1  2  3]
 [ 9 10 11]
 [ 1  2  3]
 [ 9 10 11]
 [ 1  2  3]
 [ 9 10 11]
 [ 1  2  3]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

```
Resulting Matrix in Hexadecimal:
0xe 0xe 0xe
0x3e 0x3e 0x3e
0xe 0xe 0xe
0x3e 0x3e 0x3e
0xe 0xe 0xe
0x3e 0x3e 0x3e
0xe 0xe 0xe
```

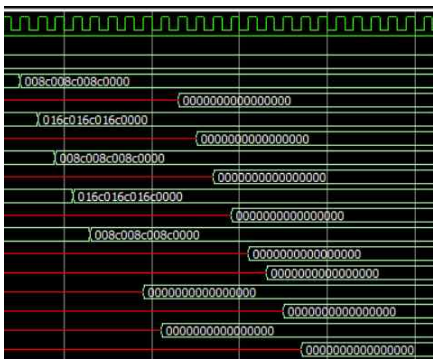
stage 2 : MNT=12'h374



```
Selected Matrix 1 (T x N):
[[ 1  2  3  4  5  6  7]
 [ 9 10 11 12 13 14 15]
 [ 1  2  3  4  5  6  7]
 [ 9 10 11 12 13 14 15]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1]
 [2 2 2]
 [3 3 3]
 [4 4 4]
 [5 5 5]
 [6 6 6]
 [7 7 7]]
```

```
Resulting Matrix in Hexadecimal:
0x8c 0x8c 0x8c
0x16c 0x16c 0x16c
0x8c 0x8c 0x8c
0x16c 0x16c 0x16c
```

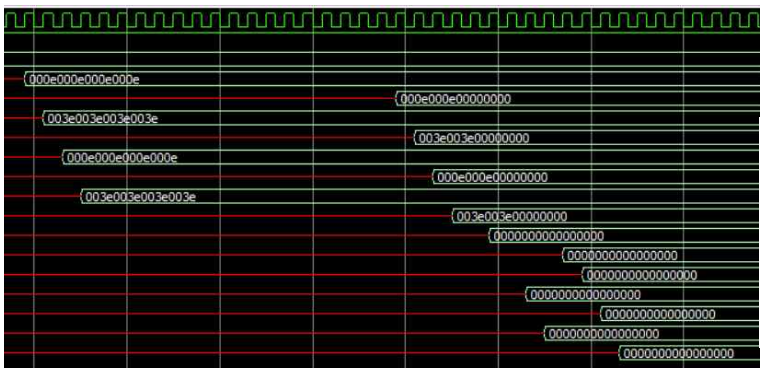
stage 3 : MNT=12'h376



```
Selected Matrix 1 (T x N):
[[ 1  2  3  4  5  6  7]
 [ 9 10 11 12 13 14 15]
 [ 1  2  3  4  5  6  7]
 [ 9 10 11 12 13 14 15]
 [ 1  2  3  4  5  6  7]
 [ 9 10 11 12 13 14 15]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1]
 [2 2 2]
 [3 3 3]
 [4 4 4]
 [5 5 5]
 [6 6 6]
 [7 7 7]]
```

```
Resulting Matrix in Hexadecimal:
0x8c 0x8c 0x8c
0x16c 0x16c 0x16c
0x8c 0x8c 0x8c
0x16c 0x16c 0x16c
0x8c 0x8c 0x8c
0x16c 0x16c 0x16c
```

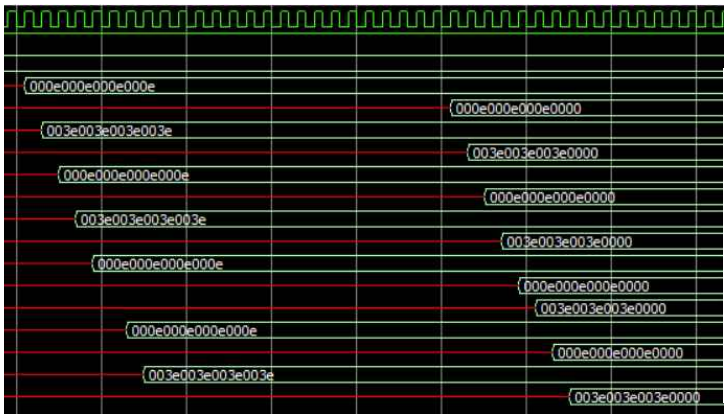
stage 4 : MNT=12'h634



```
Selected Matrix 1 (T x N):
[[ 1  2  3]
 [ 9 10 11]
 [ 1  2  3]
 [ 9 10 11]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1 1 1 1]
 [2 2 2 2 2 2]
 [3 3 3 3 3 3]]
```

```
Resulting Matrix in Hexadecimal:
0xe 0xe 0xe 0xe 0xe 0xe
0x3e 0x3e 0x3e 0x3e 0x3e 0x3e
0xe 0xe 0xe 0xe 0xe 0xe
0x3e 0x3e 0x3e 0x3e 0x3e 0x3e
```

stage 5 : MNT=12'h738

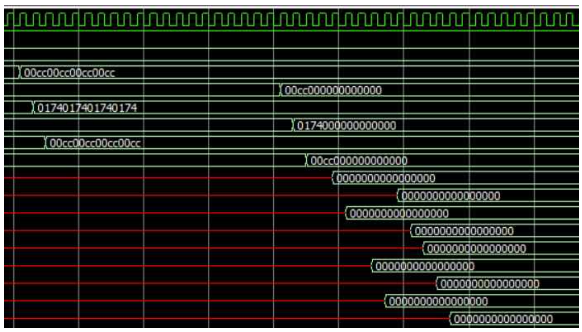


```
Selected Matrix 1 (T x N):
[[ 1  2  3]
 [ 9 10 11]
 [ 1  2  3]
 [ 9 10 11]
 [ 1  2  3]
 [ 9 10 11]
 [ 1  2  3]
 [ 9 10 11]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1 1 1 1 1]
 [2 2 2 2 2 2 2]
 [3 3 3 3 3 3 3]]
```

```
Resulting Matrix in Hexadecimal:
0xe 0xe 0xe 0xe 0xe 0xe 0xe
0x3e 0x3e 0x3e 0x3e 0x3e 0x3e 0x3e
0xe 0xe 0xe 0xe 0xe 0xe 0xe
0x3e 0x3e 0x3e 0x3e 0x3e 0x3e 0x3e
0xe 0xe 0xe 0xe 0xe 0xe 0xe
0x3e 0x3e 0x3e 0x3e 0x3e 0x3e 0x3e
0xe 0xe 0xe 0xe 0xe 0xe 0xe
0x3e 0x3e 0x3e 0x3e 0x3e 0x3e 0x3e
```

---

stage 6 : MNT=12'h583

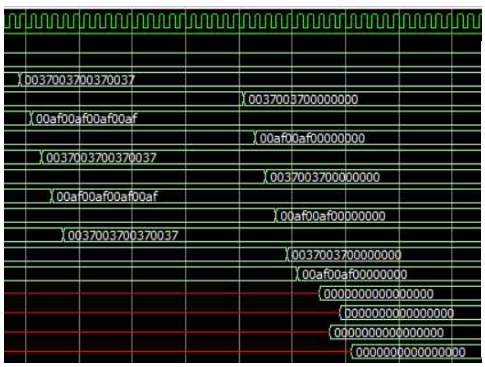


```
Selected Matrix 1 (T x N):
[[ 1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15  1]
 [ 1  2  3  4  5  6  7  8]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]
 [5 5 5 5 5]
 [6 6 6 6 6]
 [7 7 7 7 7]
 [8 8 8 8 8]]
```

```
Resulting Matrix in Hexadecimal:
0xcc 0xcc 0xcc 0xcc 0xcc
0x174 0x174 0x174 0x174 0x174
0xcc 0xcc 0xcc 0xcc 0xcc
```

---

stage 7 : MNT=12'h656



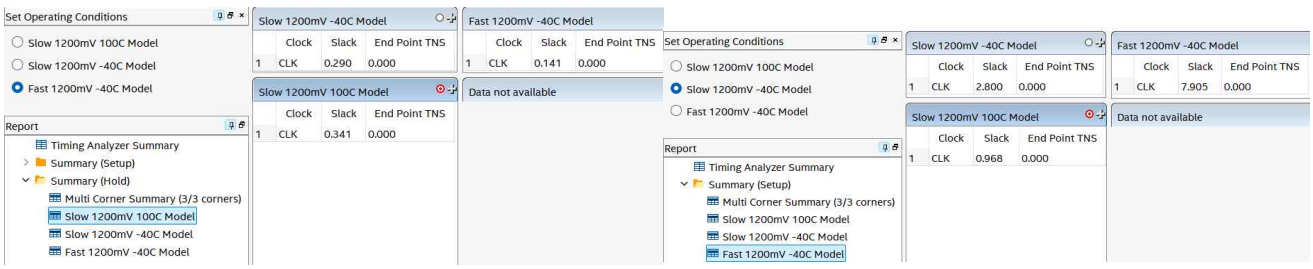
```
Selected Matrix 1 (T x N):
[[ 1  2  3  4  5]
 [ 9 10 11 12 13]
 [ 1  2  3  4  5]
 [ 9 10 11 12 13]
 [ 1  2  3  4  5]
 [ 9 10 11 12 13]]

Selected Matrix 2 (transposed, N x M):
[[1 1 1 1 1 1]
 [2 2 2 2 2 2]
 [3 3 3 3 3 3]
 [4 4 4 4 4 4]
 [5 5 5 5 5 5]]
```

```
Resulting Matrix in Hexadecimal:
0x37 0x37 0x37 0x37 0x37 0x37
0xaf 0xaf 0xaf 0xaf 0xaf 0xaf
0x37 0x37 0x37 0x37 0x37 0x37
0xaf 0xaf 0xaf 0xaf 0xaf 0xaf
0x37 0x37 0x37 0x37 0x37 0x37
0xaf 0xaf 0xaf 0xaf 0xaf 0xaf
```

stage에 대한 모든 경우에 전체적으로 정상적으로 동작하는 것을 확인할 수 있다.

1. 결과가 성공적으로 저장되었고,
2. 연산되지 않는 부분에 모두 0이 저장되어있다.

7. Quartus 결과



Quartus를 이용해 Simulation을 돌린 결과, 모든 경우에 Slack이 양수로 나타나 Timing Violation이 발생하지 않는 것을 확인할 수 있다.