4190.301A Hardware System Design

Spring 2020

Hardware System Design Lab5 Report

Kim Bochang

2014-16757

1. <lab5> Introduce

Lab 5의 목적은 지금까지의 Lab에서 배웠던것을 이용하여, final project에서 사용할 BRAM과 Processing Element(PE)를 구현하는 것이다.

2. Implementation

2.1 BRAM (lab5₩lab5.srcs₩sources_1₩new₩my_bram.v)

우리가 구현할 BRAM은 쓰는데 1cycle, 읽는데 2cycle이 필요하다는 제약조건을 가지고 있고,

코드를 구현할때 기존에 존재하던 file 안의 파일에서 초기값을 불러오고, 계산이 끝난 값을 저장할 수 있어야 하는 기능이 필요하다.

BRAM의 구현은 다음과 같다.

```
module my_bram#(
    parameter integer BRAM_ADDR_WIDTH = 15, // 4x8192
    parameter INIT_FILE = "input.txt",
    parameter OUT_FILE = "output.txt"
    )(
    input wire [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR,
    input wire BRAM_CLK,
    input wire [31:0] BRAM_WRDATA,
    output reg [31:0] BRAM_RDDATA,
    input wire BRAM_EN,
    input wire BRAM_RST,
    input wire [3:0] BRAM_WE,
    input wire done
    );
    reg [31:0] mem[0:8191];
    wire [BRAM_ADDR_WIDTH-3:0] addr = BRAM_ADDR[BRAM_ADDR_WIDTH-1:2];
    reg [31:0] dout;

    // code for reading & writing
    initial begin
        if (INIT_FILE != "") begin
            $readmemh(INIT_FILE,mem);
            // read data from INIT_FILE and store them into mem
        end

        wait (done)
        // write data stored in mem into OUT_FILE
        $writememh(OUT_FILE, mem);
    end
```

먼저, INIT_FILE과 OUT_FILE이라는 파일 이름 파라미터를 이용하여, INIT_FILE인자가 존재하면

베릴로그의 $readmemh(filename, mem)함수를 이용하여 메모리에 파일에 저장된 값들을 모두 불러오게 하였다.

그 후, 외부에서 done 신호가 1이 되면, $writememh(filename, mem)함수를 이용하여 메모리에 저장된 값들을 해당 파일에 저장하도록 하였다.

```
    always @ (posedge BRAM_CLK) begin
        if (BRAM_EN) begin
            if(|BRAM_WE) begin // reduction to one bit, use reduction or,
                // do write to BRAM
                mem[addr] <= {(BRAM_WRDATA[8*4-1:8*3] & {(8){BRAM_WE[3]}}),
                              (BRAM_WRDATA[8*3-1:8*2] & {(8){BRAM_WE[2]}}),
                              (BRAM_WRDATA[8*2-1:8*1] & {(8){BRAM_WE[1]}}),
                              (BRAM_WRDATA[8*1-1:8*0] & {(8){BRAM_WE[0]}})};
            end
            else begin
                // do read from BRAM
                dout <= mem[addr];
            end
        end
    end

    always @ (posedge BRAM_CLK) begin   // dout comes to BRAM_RDDATA, it makes read value after two cycle, reseted value also appear two cycle,
       BRAM_RDDATA <= (dout & {(32){!BRAM_RST}});
       /*
       if(BRAM_RST) begin  //reset, set data to 0
                mem[addr] <= {(32){2'b0}};
                // dout <= {(32){2'b0}};
            end
            else
       */
    end

endmodule
```

BRAM의 상시 동작은 위와 같이 구현하였다.

먼저, dout이라는 레지스터가 따로 존재하는데, 이 레지스터는 bram의 읽기동작이 2cycle이 걸리도록,

동작을 지연시키는 버퍼 역할을 하는 레지스터다. bram의 ouptut은 dout을 거쳐 BRAM_RDDATA로 간접적으로

출력되게 된다. (즉, 1사이클의 딜레이가 추가되어 총 2사이클이 걸리게 됨)

CLK가 posedge일때 동작하도록 하였고, (즉, clk이 posedge일때만 읽기/쓰기 작동)

BRAM_EN 시그널이 1일때만 내부 동작을 하도록 하였다.

시그널이 1인경우,

BRAM_WE에 reduction or operator를 사용하여 1인 비트가 하나라도 있는지 체크한다.

만약 BRAM_WE에 1인 비트가 하나라도 있는 경우, BRAM의 해당 address에 쓰기 작업을 진행한다.

이때, BRAM_WE의 각 자리수 비트는 마스크로 작동하게 구현하였다.

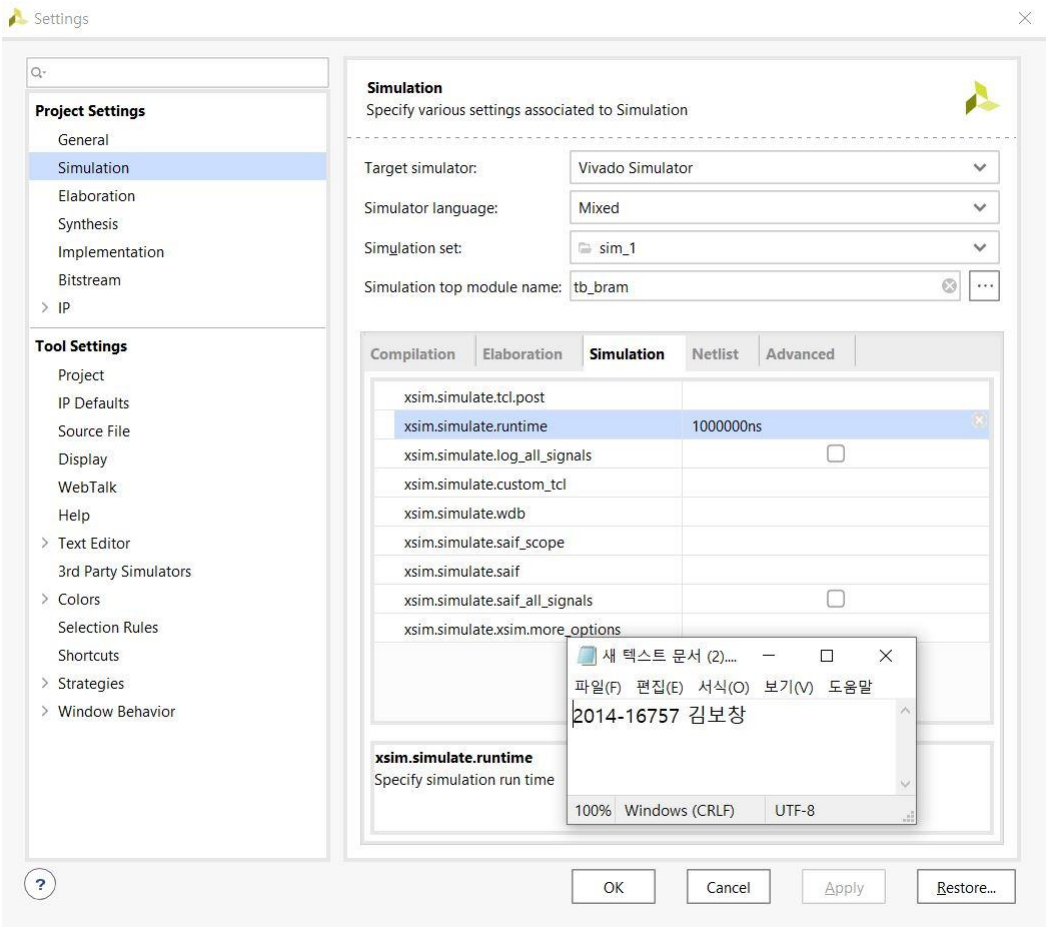
1인 비트가 하나도 없는 경우, dout에 해당 어드레스의 데이터를 저장 ,읽기 작업을 하도록 하였다.

또한, 매 사이클마다 BRAM_RDDATA에 dout을 넣어 문제에서 요구하는 read delay를 구현하였다.

BRAM_RST이 1로 설정된 경우, 문제의 스펙대로라면 0을 출력해야 하므로 BRAM_RDDATA에

!BRAM_RST을 mask로 사용하여 BRAM_RDDATA를 출력하게 하였다.

위를 테스트 하기 전에, 시나리오상 모든 BRAM에 저장된 모든 데이터에 접근해야 하므로, 꽤 긴시간이 걸리게 된다. 따라서 다음과 같이 simulator 설정을 추가로 진행해 주었다.



runtime을 적당히 늘려주었다.

이제, BRAM의 작동을 테스트 해야하는데, 그 전에 한쪽 BRAM을 initialize 하기 위해서 input.txt 파일이 필요하다. 이 input.txt파일은 베릴로그 코드를 이용하여 생성해주었다. (lab5\lab5.srcs\sources_1\new\input.txt)

```
module tb_gen_input();

integer i;
reg [31:0] mem [0:8191];

    initial begin
    for (i=0; i<8192; i = i+1) begin
           mem[i] <= i;
    end
    #10;
    $writememh("input.txt", mem);
    end


endmodule
```

(lab5\lab5.srcs\sim_1\new\tb_gen_input.v)

위 BRAM의 작동을 테스트하기 위한 코드는 다음과 같다.

```
module tb_bram();

  integer i;

  reg clk;
  reg [14:0] BRAM_ADDR;
  reg [14:0] BRAM_ADDR_delayed_1; // this is used for delay one clock cycle
  reg [14:0] BRAM_ADDR_delayed_2; // this is used for delay two clock cycle
  reg done;
  wire [31:0] BRAM_RDDATA;
  wire [31:0] BRAM_RDDUMMY; // it should return trash value.
  wire [31:0] BRAM_WRDUMMY;  // there is no use for this.

  initial begin
    clk <= 0;
    BRAM_ADDR <= {(15){1'b0}};
    for(i=0; i<8192;i=i+1) begin
        #10;
        BRAM_ADDR <= BRAM_ADDR + 3'b100;
    end
    #20;
    done <= 1'b1;
  end

  always #5 clk <= ~clk;

  always @ (posedge clk) begin
     BRAM_ADDR_delayed_1 <= BRAM_ADDR;
     BRAM_ADDR_delayed_2 <= BRAM_ADDR_delayed_1;    //delaying for two clock cycle.
  end

  my_bram # (15, "input.txt", "output1.txt") bram1 (
    .BRAM_ADDR(BRAM_ADDR),
    .BRAM_CLK(clk),
    .BRAM_WRDATA(BRAM_WRDUMMY),
    .BRAM_RDDATA(BRAM_RDDATA),
    .BRAM_EN(1'b1),
    .BRAM_RST(1'b0),
    .BRAM_WE(4'b0000),
    .done(done)
  );

 my_bram # (15, "", "output2.txt") bram2 (
    .BRAM_ADDR(BRAM_ADDR_delayed_2),
    .BRAM_CLK(clk),
    .BRAM_WRDATA(BRAM_RDDATA),
    .BRAM_RDDATA(BRAM_RDDUMMY),
    .BRAM_EN(1'b1),
    .BRAM_RST(1'b0),
    .BRAM_WE(4'b1111),
    .done(done)
  );


endmodule
```

(lab5\lab5.srcs\sim_1\new\tb_bram.v)

간단하게 동작을 설멍하면, address를 4씩 (bram 내부 address로 1) 증가시키면서

bram1에 저장된 값을 bram2로 옮기는 역할을 한다.

이때 bram1에서 address x에서 값을 꺼내서 bram2의 address x에서 저장할때, bram1에서 값을 꺼내는 시간만큼이 필요하므로, BRAM_ADDR_delayed_n 레지스터를 이용해 하나의 어드레스값을 내부에서 지연시켜서 사용하도록 한다.

이렇게 값을 모두 옮긴 뒤에는, done을 1로 설정해 내부에 무슨값이 저장되었는지 파일로 출력하도록 하였다.

출력된 output1, output2 파일은 베릴로그 프로젝트의 simulation 폴더에도 들어있고,

제출된 폴더에도 동봉되어있다.

2.2 PE (lab5₩lab5.srcs₩sources_1₩new₩my_pe.v)

우리가 구현해야 하는 PE는 LAB4에서 IP catalog를 이용하여 만든 floating point fused-multiplier를 사용하여, 내부 메모리에 저장된 값(bin)과 외부에서 주어지는 값(ain)을 곱하고, 내부 메모리에 저장된 기존 결과에 이 두 값을 곱한값을 더하여 blocked-vector multiply를 구현하게 된다.

내부 구현은 다음과 같다.

```
module my_pe #(parameter L_RAM_SIZE = 6)
(
    // clk /reset
    input aclk,
    input aresetn,
    // port A
    input [31:0] ain,
    //peram --> port B
    input [31:0] din,
    input [L_RAM_SIZE-1:0] addr,
    input we,
    // integrated valid signal
    input valid,
    // computation result
    output dvalid,
    output [31:0] dout
);

(* ram_style = "block" *) reg [31:0] peram [0:2**L_RAM_SIZE - 1]; // local register

reg [31:0] r_dout;
reg [31:0] bin;
wire [31:0] temp_dout;


initial begin
    r_dout <= 0;
    bin <= 0;
end
```

먼저, 내부에서 사용할 r_dout과 bin이라는 레지스터와 temp_dout이라는 와이어를 선언한다.

r_dout은 fused multiplier에서 나온 값을 저장할 레지스터,

bin은 peram에서 address에 맞게 꺼내져 나온값을 임시로 저장할 레지스터다.

문제에서 we==0일때 address의 값을 fused_multiplier에 먹여야 하기 때문에 필요하다.

```
always @ (posedge aclk) begin
    if (we) begin
        peram[addr] <= din;
    end
    else begin
        bin <= peram[addr];
    end
end

always @ (posedge aclk) begin
    if (!aresetn) begin
        r_dout <= {(32){2'b0}};
    end
    else if(dvalid) begin
        r_dout <= temp_dout;
    end
    else begin
        r_dout <= r_dout;
    end
end

assign dout = temp_dout & {(32){dvalid}};
```

그 후, 클락이 posedge일때 we가 1이라면 외부에서 들어온 값(din)을 peram의 해당 어드레스에 저장하고,

we가 0이면 peram에서 값을 꺼내 fused multiplier의 input으로 들어가도록 준비시킨다.

또한, aresetn이 0일때 리셋이 진행되어야 하므로, 내부에 저장되어있는 r_dout을 0으로 초기화 하고,

그렇지 않다면 dvalid가 1일때 fused-multiplier에서 나온 결과를 r_dout에 저장한다.

아래 dout에 assign하는 코드는, dvalid가 0일때는 fused_multiplier의 값을 출력하면 안되므로 dvalid를 마스크로 이용하였다.

```
    fused_mult UUT(
            .aclk(aclk),
            .aresetn(aresetn),
            .s_axis_a_tvalid(valid),
            .s_axis_b_tvalid(valid),
            .s_axis_c_tvalid(valid),
            .s_axis_a_tdata(ain),
            .s_axis_b_tdata(bin),
            .s_axis_c_tdata(r_dout),
            .m_axis_result_tvalid(dvalid),
            .m_axis_result_tdata(temp_dout)
        );

endmodule
```

연결은 위와 같다.

이를 테스트하기 위한 코드는 다음과 같다. (lab5\lab5.srcs\sim_1\new\tb_module.v)

```
module tb_module();

reg [32-1:0] ain;
reg [32-1:0] din;
reg [31:0] ain_mem [0:15];
reg [31:0] din_mem [0:15];
reg [6-1:0] addr;
reg we;
reg clk;
reg valid;
reg areset;

wire dvalid;
wire [31:0] dout;
//for test
integer i;
//random test vector generation
    initial begin
        clk<=0;
        addr = 0;
        valid=0;    //internal MAC sholdn't working
        we=1;       // data goes to register
```

din은 PE의 내부 메모리에 저장될 값, ain은 PE의 외부에서 직접 입력되는 값이다.

ain_mem과 din_mem은 테스트 벤치 용도로, 각각 ain,din에 들어갈 값을 미리 저장해 놓은 것이다.

```
        ain_mem[0] = 32'b00111111100000000000000000000000;
        din_mem[1] = 32'b01000000000000000000000000000000;  //2
        ain_mem[1] = 32'b01000000000000000000000000000000;
        din_mem[2] = 32'b01000000010000000000000000000000;  //3
        ain_mem[2] = 32'b01000000010000000000000000000000;
        din_mem[3] = 32'b01000000100000000000000000000000;  //4
        ain_mem[3] = 32'b01000000100000000000000000000000;
        din_mem[4] = 32'b01000000101000000000000000000000;  //5
        ain_mem[4] = 32'b01000000101000000000000000000000;
        din_mem[5] = 32'b01000000110000000000000000000000;  //6
        ain_mem[5] = 32'b01000000110000000000000000000000;
        din_mem[6] = 32'b01000000111000000000000000000000;  //7
        ain_mem[6] = 32'b01000000111000000000000000000000;
        din_mem[7] = 32'b01000001000000000000000000000000;  //8
        ain_mem[7] = 32'b01000001000000000000000000000000;
        din_mem[8] = 32'b01000001000100000000000000000000;  //9
        ain_mem[8] = 32'b01000001000100000000000000000000;
        din_mem[9] = 32'b01000001001000000000000000000000;  //10
        ain_mem[9] = 32'b01000001001000000000000000000000;
        din_mem[10] = 32'b01000001001100000000000000000000;  //11
        ain_mem[10] = 32'b01000001001100000000000000000000;
        din_mem[11] = 32'b01000001010000000000000000000000;  //12
        ain_mem[11] = 32'b01000001010000000000000000000000;
        din_mem[12] = 32'b01000001010100000000000000000000;  //13
        ain_mem[12] = 32'b01000001010100000000000000000000;
        din_mem[13] = 32'b01000001011000000000000000000000;  //14
        ain_mem[13] = 32'b01000001011000000000000000000000;
        din_mem[14] = 32'b01000001011100000000000000000000;  //15
        ain_mem[14] = 32'b01000001011100000000000000000000;
```

ain_mem과 din_mem은 floating point(single precision)에 해당하는 값으로 1~16까지의 값이 들어가 있다.

원래는 랜덤생성을 하려 했으나, 그렇게 하면 inf와 같은 값이 발생하는 경우가 생겨 동작을 확인하기 편하도록 위와 같은 값을 이용하였다.

```
areset <= 0;    //reset

#20;

areset <= 1;

for(i=0; i<16; i=i+1) begin     //random generate
    din = din_mem[i];
    #10;
    addr = addr + 2'b1;
end

#10;

addr = 0;
we=0;

for(i=0; i<16; i=i+1) begin
    ain = ain_mem[i];
    #10;    //load inside register value to b_in, and calcuated d_out value into r_dout. now calcualtaion ready.
    valid = 1;   //now internal mac working
    #10;
    valid = 0; // input inserted, so we should wait
    wait(dvalid == 2'b1)
    #5;     //clock edge sync
    addr = addr + 2'b1;    //address change
end
#10;
addr <= 0;
```

그 후, areset 신호를 0으로 주어 내부 회로를 모두 초기화 한뒤, areset을 다시 1로 준다.

이때 2클록사이클에 해당하는 #20을 이용한것은, fused multiply-adder의 documentation의 요구에 따른것이다.

(최소 2클록사이클동안 0으로 유지되어야함)

그 후, we=1일때 address를 1씩 높여가면서 PE 내부 메모리의 각 address에 1,2,...16의 값이 저장되도록 하였다.

그 후, we=0으로 만들고, address를 1씩 높여가면서 외부 입력 (ain)과 안에 있는 내부 메모리의 address에 해당하는 값을 계산하도록 하였다.

이때 중요한것은 address를 지정한 후에 1사이클 쉬어 준 후, valid를 1로 설정하는 것인데,

address 를 지정한다음 내부 메모리에서 PE 의 연산기로 값을 불러오기까지 1사이클이 필요하기 때문이다.
valid 를 1로 설정하고, 1사이클 뒤에 다시 0으로 바꾸어 쓸데없는 값이 들어가지 않도록 하였다.

그 후 dvalid 가 1이 될때까지 wait 문을 이용하여 기다리고, dvalid 가 true 가 되었을때
반사이클에 해당하는 #5를 이용하여, 클록 타이밍을 맞춰주었다.

반 사이클을 기다리는 이 과정을 진행하지 않으면, PE 내부의 multiply-adder 의 결과가 PE 의 내부 레지스터에 저장되는 시간을 확보되지 못해, 이전에 계산되었던 dout 값을 사용하는 경우가 생긴다.

그 후, 어드레스를 바꾸고, ain 에 새로운 값이 들어오고 한사이클을 기다려 다시 PE 내부에서 바뀐 어드레스에 해당하는 값을 불러오도록 한다... 를 16 address 까지 반복한다.

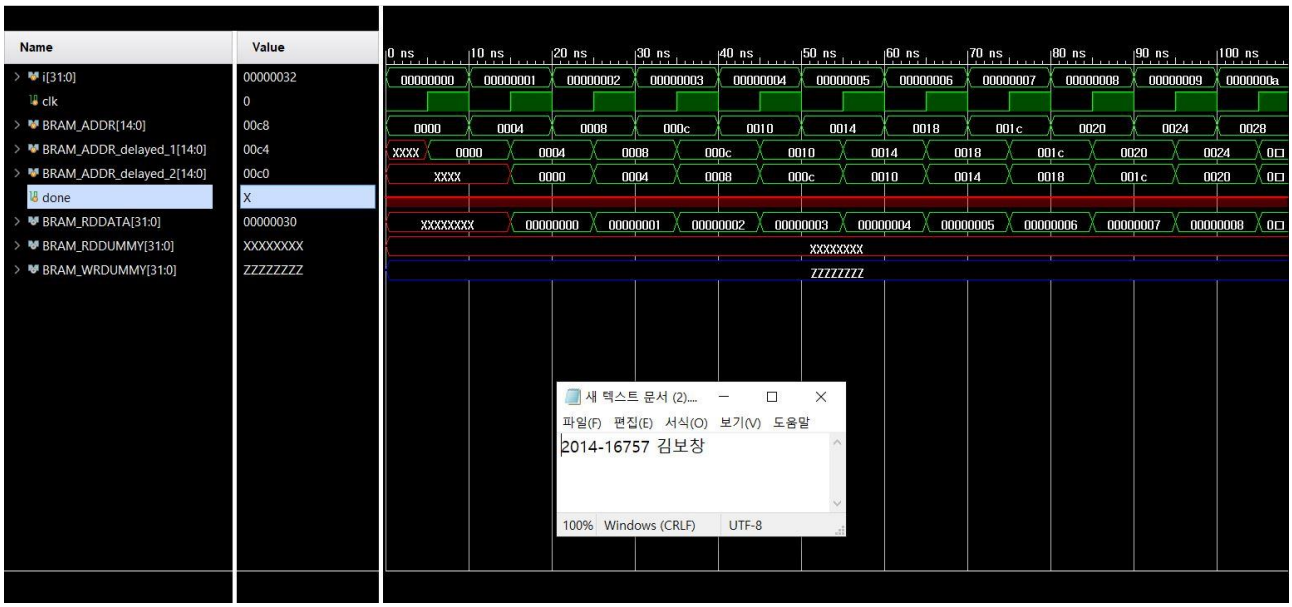
```
    end

    always #5 clk <= ~clk;


    my_pe #(6) PE (
      .aclk(clk),
      .aresetn(areset),
      .ain(ain),
      .din(din),
      .addr(addr),
      .we(we),
      .valid(valid),
      .dvalid(dvalid),
      .dout(dout)
    );


endmodule
```

clock은 5ns 주기로 바뀌고, PE에 대한 할당은 위와 같이 하였다.

3. Result & Discussion

3.1 BRAM

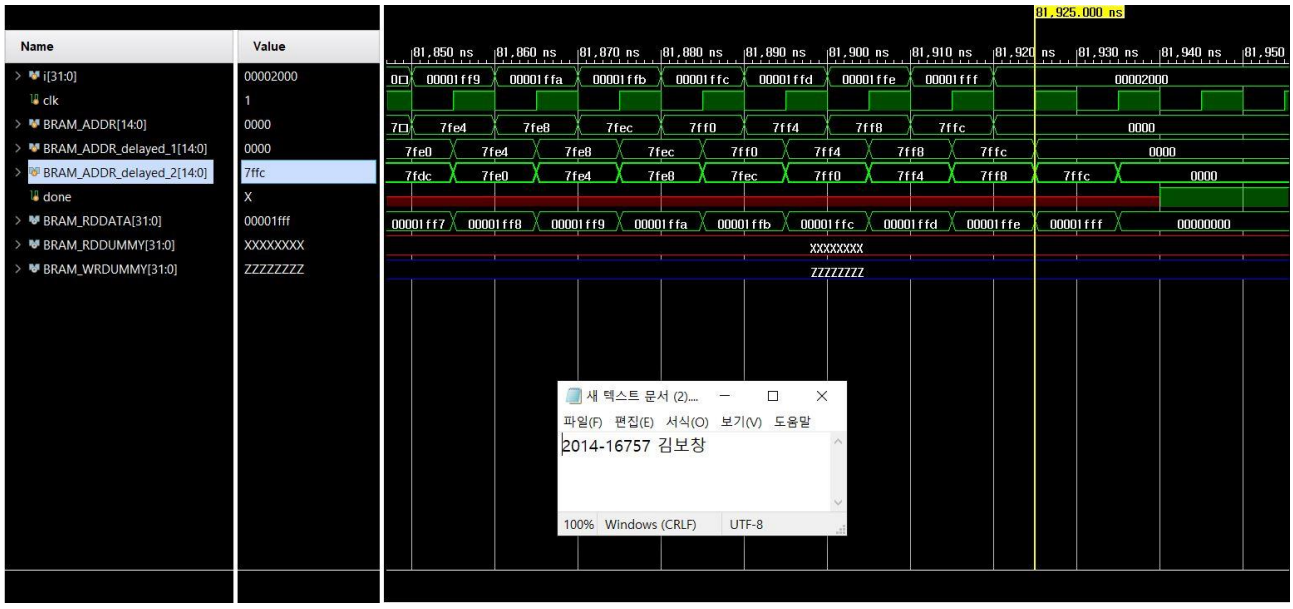
<전체 waveform>



처음 address 0부터 시작해서, 1사이클이 지났을때 bram1의 해당 address위치에 저장된 값이

계속해서 BRAM_RDDATA의 값으로 출력됨을 알 수 있다.

또한, BRAM_RDDATA로 출력된 값은 bram2로 들어가게 된다.



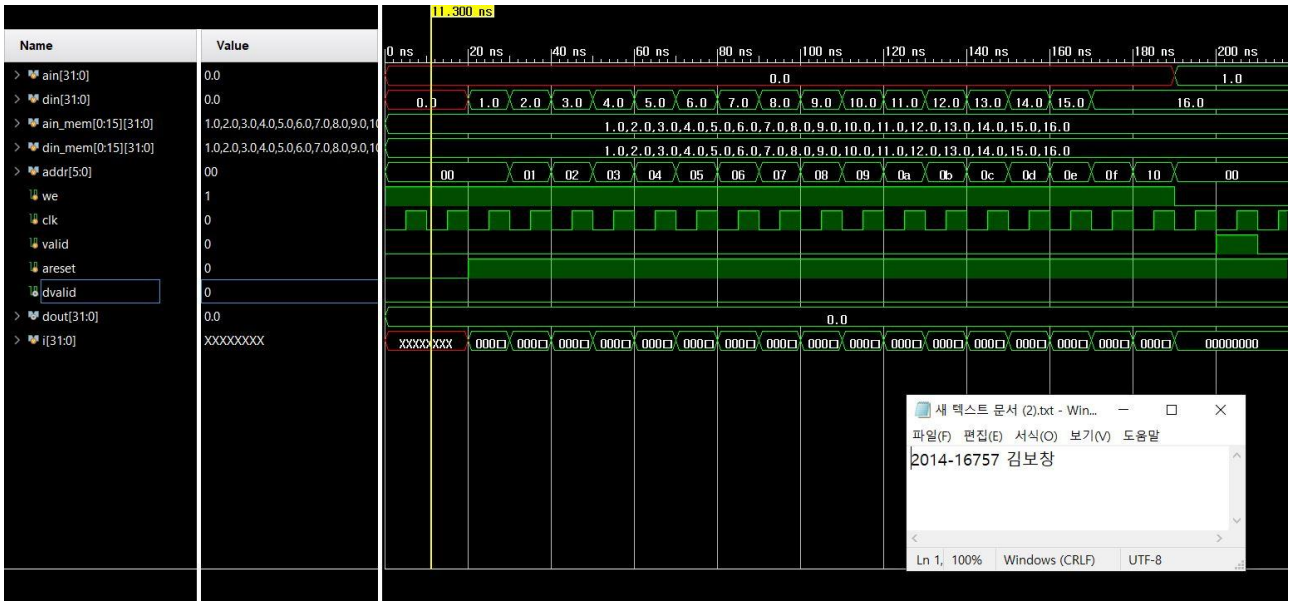
마지막 부분이다. 역시 값이 잘 나오는것을 확인할 수 있다.

이제 bram2에 값이 어떻게 저장되었는지 output2.txt를 열어보면

처음과 끝부분을 보면, 모두 값이 제대로 들어갔음을 알 수 있다.

3.2 PE

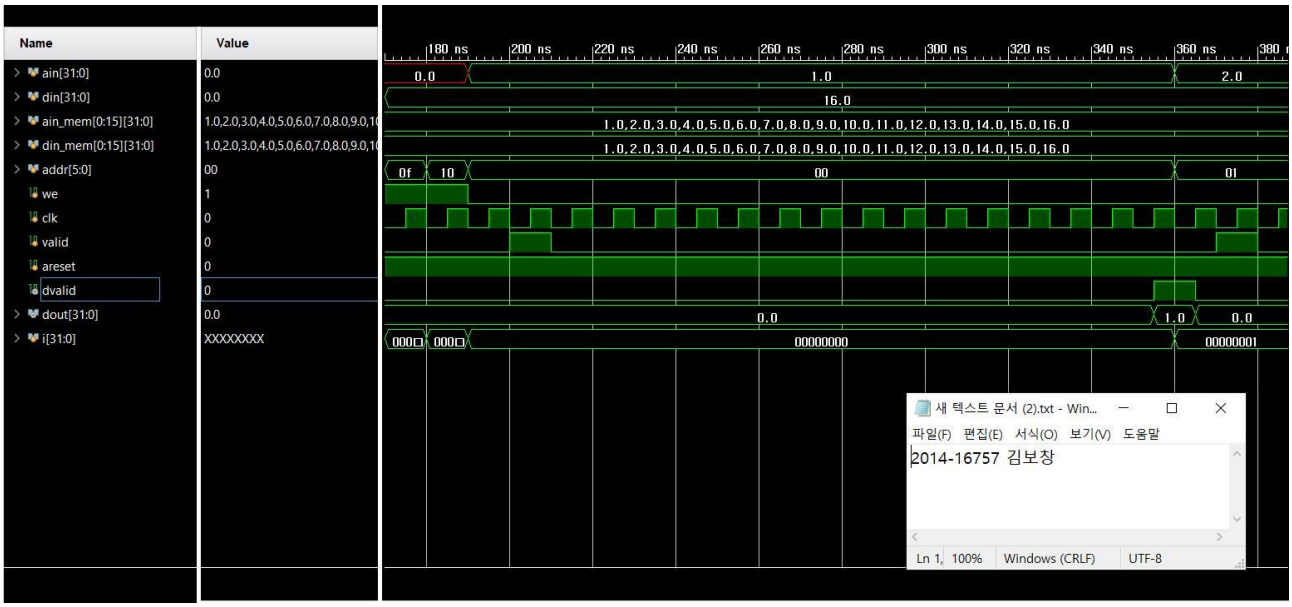
<전체 wave form.>



<waveform 초반부.>

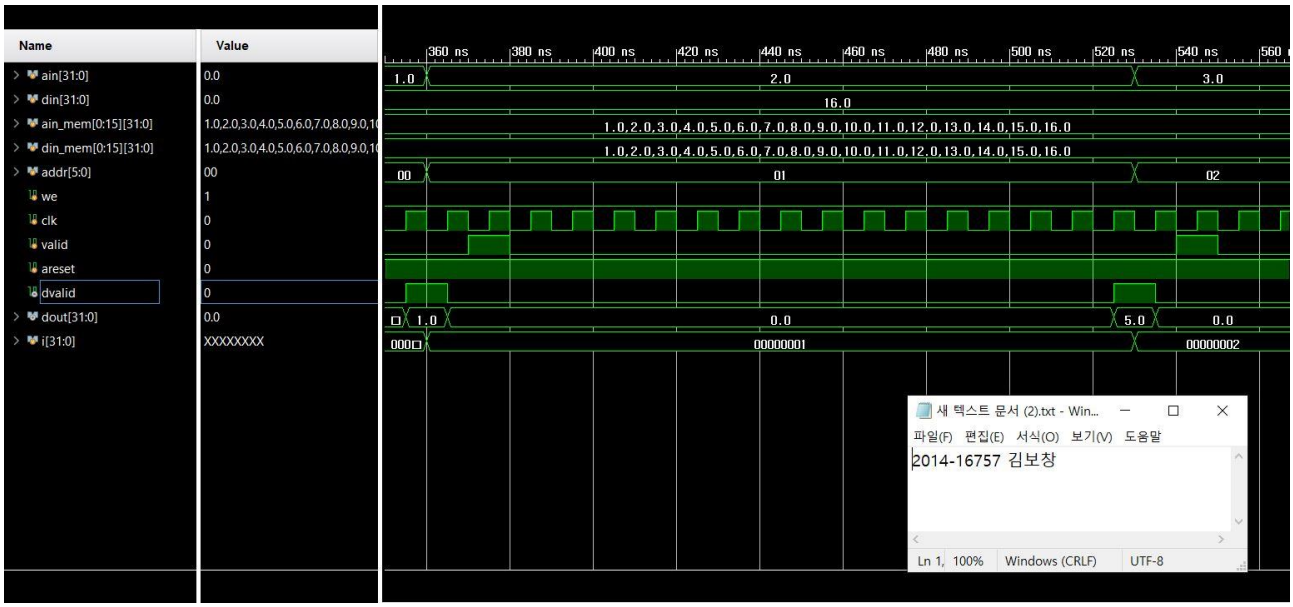
처음에 areset을 0으로 설정하여 초기화를 해주고, din의 값으로 각각 1.0, 2.0... 16.0의 값을 집어넣어

PE내부의 memory의 address 0~15까지에 저장되게 한다.

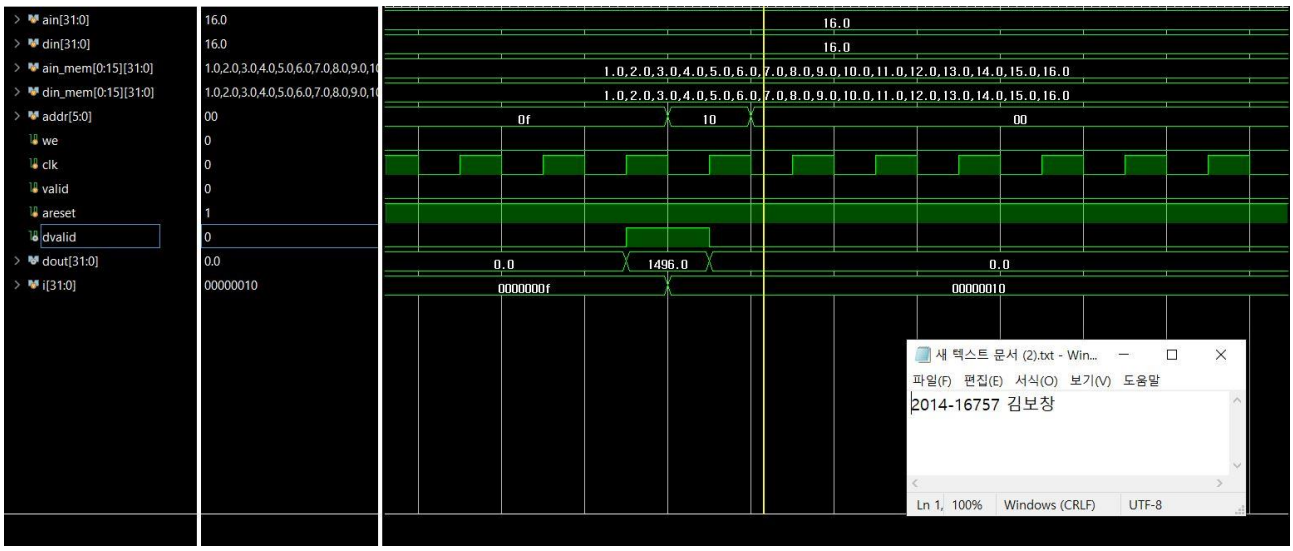


<waveform 초반부 바로 뒤>

그 후, ain에 1.0, 2.0...의 값을 집어넣는데, valid가 1인 시점에서 ain = 1, din = 1 (내부 메모리 address 0 에 저장)된 값이 들어가고, dvalid가 1일때 dout의 값으로 1.0이 출력되는 것을 알 수 있다.



그 후, ain = 2, din = 2에 해당하는 값이 들어가서 dout의 값으로 2.0 * 2.0 + 1.0 = 5.0의 값이 나오는것을 확인할 수 있다.



최종적으로, 1496의 값이 나오는데, n = 1 부터 16까지 n^2의 합이 1496이므로, 구현한 회로가 정상적으로 동작함을 알 수 있다.

결과적으로, 내가 구현한 코드가 정상적으로 작동함을 알 수 있었다.

4. Conclusion

이번 프로젝트를 할 때는 유난히 시간이 많이 걸렸는데, 일단 베릴로그 자체의 버그때문에 코드를 바꿔도 시뮬레이션에는 적용되지 않는 현상이 발생하여, 이를 해결하는데 많은 시간이 걸렸었다.

또한, 타이밍을 맞추는 것이 꽤 까다로운 편이었는데, 적절한 타이밍을 맞추지 않으면 이상한 결과가 나오는 일이 잦았다.

이는 PE과제를 할 때 특히 두드러졌는데, PE 내부 메모리의 address에 저장된 값을 불러오는 타이밍을 기다리지 않고 계산을 진행하여 의도한것과는 다른 address의 값을 들고 오거나,
내부의 총 결과값을 저장하는 레지스터에 fused multiply-adder에서 계산된 값이 저장되기 전에,
기존 갱신되기 전의 총 결과값을 fused multilpy-adder의 입력으로 넣는 일이 생겨서 디버깅하는데 애를 많이 먹었었다.

이를 해결하기 위해 타이밍에 굉장히 많이 신경을 써야했고, 때문에 여러가지 테크닉을 익힐 수 있었던 좋은 랩이었다.