하드웨어 설계 실습 LAB05

이다운 2016-13919

MY_BRAM

1. CODE

코드의 핵심 구현부분에 대해서만 설명한다.

```
 initial begin
   if (INIT_FILE != "") begin
read data from INIT_FILE and store them into mem
     $readmemh(INIT_FILE, mem );
    end
   wait (done==1'b1)
write data stored in mem into OUT_FILE
   $writememh(OUT_FILE, mem,0 );
```

위는 외부 파일 입출력에 관한 코드다. initial구문을 $readmemh 로 시작함으로써 input 파일을 읽어 mem에 저장한다. 그후 done 신호 들어올 때 까지 대기하다가 신호가 들어오면 $writememh 함수를 사용해 output파일에 mem 데이터를 저장한다.

```
if(BRAM_EN ==1 && BRAM_RST != 1)begin///
  if(BRAM_WE == 4'b0000 ) begin
      buffer1<= mem[addr];
  end
  if(BRAM_WE[0] == 4'b1) begin
    buff[0] <= BRAM_WRDATA[7:0];
    addbuff[0] <= addr;
    check[0] <=1;
   // mem[addr][7:0] <= BRAM_WRDATA[7:0];
  end
  if(BRAM_WE[1] == 4'b1) begin
    buff[1] <= BRAM_WRDATA[15:8];
    addbuff[1] <= addr;
    check[1] <=1;
  //  mem[addr][15:8] <= BRAM_WRDATA[15:8];
  end
  if(BRAM_WE[2] == 4'b1) begin
    buff[2] <= BRAM_WRDATA[23:16];
    addbuff[2] <= addr;
    check[2] <= 1;
 //   mem[addr][23:16] <= BRAM_WRDATA[23:16];
  end
  if(BRAM_WE[3] == 4'b1) begin
    buff[3] <= BRAM_WRDATA[31:24];
    addbuff[3] <= addr;
    check[3] <=1;
//     mem[addr][31:24] <= BRAM_WRDATA[31:24];
  end
```

외부 파일 입출력 코드 외에는 모드 clk신호와 posedge 동기화 되어있다. 위 코드를 보면 BRAM_EN 신호가 1일때만 작동하는데 BRAM_WE==0이면 데이터를 read하는 것이므로 mem[addr]데이터를 buffer1 레지로 보내준다. 바로 out 신호로 안보내는 이유는 delay를 만들기 위한 것으로 다음 코드에서 자세히 설명한다.

BRAM_WE 신호가 0이 아닐때는 데이터를 write하는 과정이 일어나야하므로 input으로 들어온 BRAM_WRDATA와 그에 대응하는 주소 값 addr을 임시 버퍼인 buff, addbuff에 저장하고 check 변수를 1로 만들어준다. 이에 대한 설명은 다음 코드에 있다.

```
always @(posedge BRAM_CLK or posedge BRAM_RST)begin

  if(BRAM_RST==1)begin
  dout<= 0;
  buffer1<=0;
  buffer2<=0;
  end
  else begin
    buffer2<=buffer1;
    dout<=buffer2;
    if(check[0]==1) mem[addbuff[0]][7:0]   <= buff[0];
    if(check[1]==1) mem[addbuff[1]][15:8]  <= buff[1];
    if(check[2]==1) mem[addbuff[2]][23:16] <= buff[2];
    if(check[3]==1) mem[addbuff[3]][31:24] <= buff[3];
  end
```

buffer1과 buff에 저장된 임시 데이터는 다음 posedge clk 신호에 각각 buffer2와 mem 저장소로 저장된다. 이때 check 변수는 write하기전에 buff 값이 mem에 저장되는 것을 방지하기 위한 지표로, 처음 write할때만 확인해주면 되므로 다시 check값을 0으로 바꾸지는 않는다.

그리고 다음 clk 신호 때 buffer2 데이터는 dout로 이동하면서 output으로 출력된다. 이렇게 함으로써 write는 1 cycle delay를, read과정은 2 cycle delay를 가질 수 있게 된다.

reset 신호인 BRAM_RST 값이 1이 되면 OUPUT출력과 관련된 dout, buffer 값들을 0으로 초기화 시켜줬다. mem 데이터는 초기화 시켜주지 않았다.

2. Testbench $ Simulation

```
initial begin
    clk <=0;
    addr<=0;
    we<=0;
    done<=0;
    ars<=0;


    #30 we<=4'b1111;
    #10 we<=0;
    for(i=1; i<8192; i=i+1)begin
      addr<= addr+ 15'd4;
      #30;
      we<=4'b1111;
      #10 we<=0;
      #25;
    end
    done<=1'b1;
    #20
    ars<=1;
end
```

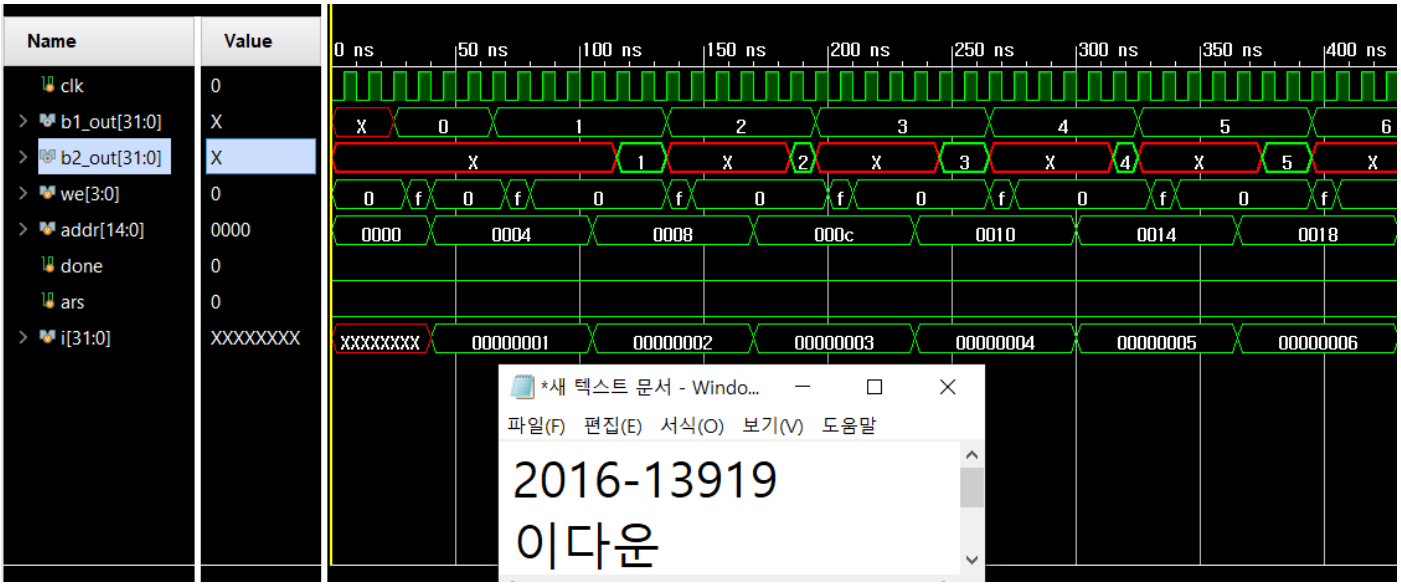
```
my_bram #(
15,"input.txt","output1.txt")
bram1(
  .BRAM_ADDR(addr),
  .BRAM_CLK(clk),
  .BRAM_WRDATA(0),
  .BRAM_RDDATA(b1_out),
  .BRAM_EN(1'b1),
  .BRAM_RST(ars),
  .BRAM_WE(0),
  .done(0)
);

my_bram #(15,"","output.txt"
)bram2(
  .BRAM_ADDR(addr),
  .BRAM_CLK(clk),
  .BRAM_WRDATA(b1_out),
  .BRAM_RDDATA(b2_out),
  .BRAM_EN(1'b1),
  .BRAM_RST(ars),
  .BRAM_WE(we),
  .done(done)
);
```

위 코드는 my_bram이 원하는 시나리오 되로 잘 작동하는지 확인하기 위한 testbench 코드이다. 먼저 클럭, 리셋, done 신호 등등을 0으로 초기화 해준다.

we신호가 0으로 시작하니 bram1에서 addr 주소에 해당하는 데이터를 read하는 과정이 일어난다. 이 때 read에는 2 cycle delay가 있으므로 충분한 시간 텀을 준다. 그 뒤 we신호를 4'b1111을 주어 bram1의 output이 bram2에서 write할 수 있도록 한다. 그런 뒤에 for문을 돌면서 addr값을 증가 시켜 bram1의 mem 데이터가 모두 bram2에 write 하게 한 뒤에 done신호를 주어 output.txt에 저장되도록 한다. 그리고 마지막으로 reset 신호를 주어 output이 모두 0이 출력 되도록 한다.

이때 addr값을 1씩 증가하지 않고 4씩 증가하는 이유는 addr의 하위 2bit는 주소 값으로 사용되지 않기 때문이다.

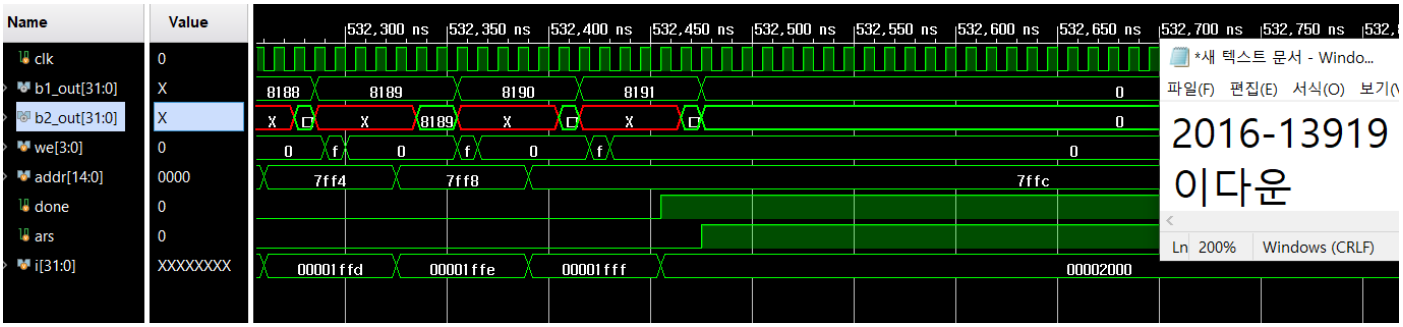
오른쪽 사진을 보면 bram1, bram2 모듈이 보이는데 bram1에는 BRAM_WE 값을 0을 줘 write하는 과정이 발생하지 않도록 했다. bram1의 output b1_out 는 bram2에서 input으로 바로 들어간다. bram1에 parameter로 input을 넣어 input.txt을 추출하는 과정을 거치지만 bram2은 공백으로 남겨 input.txt을 추출하는 과정이 일어나지 않도록 하였다.



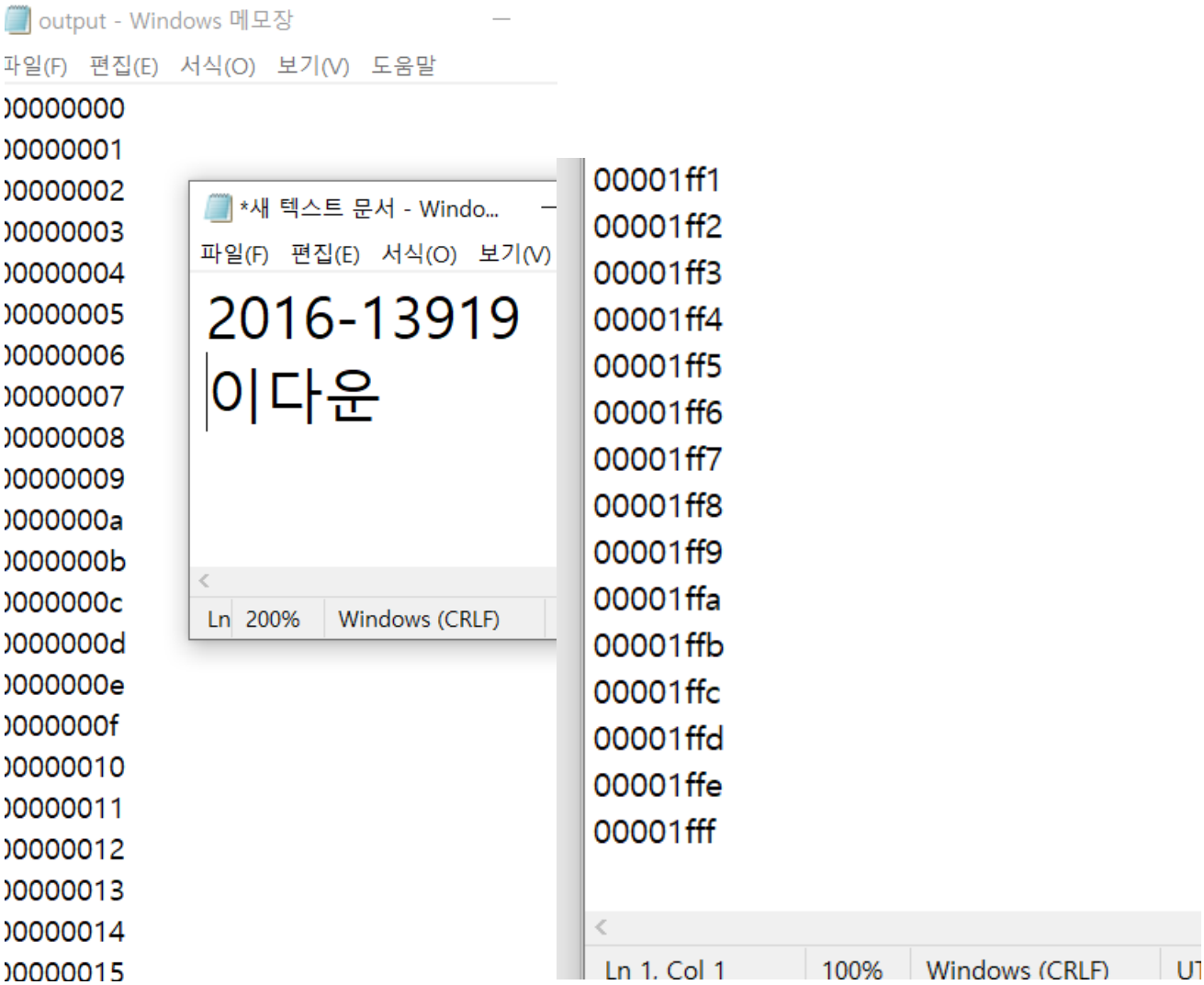
result wave form 스크린샷이다.

b1_out을 먼저 보면 맨 처음에 read 과정의 2 cycle delay 때문에 아무것도 출력되지 않는 것을 확인할 수 있다. 그리고 그 뒤에는 addr 주소 값이 들어오고 2 cycle delay 후에 순차적으로 값을 출력하는 것을 확인할 수 있다.

b2_out을 보면 X 값과 올바른 메모리 값이 번갈아 가면서 출력되는 것을 확인할 수 있다. bram2은 input.txt 을 추출하는 과정을 거치지 않으므로 mem 메모리는 비어있다. 그래서 addr 주소에 해당하는 mem 데이터를 출력할 때 처음에는 X로 비어있는 값을 출력하고 b1_out값이 bram2에 저장되면 저장된 값을 출력한다. 그 뒤에 addr값이 증가하므로 다시 비어있는 mem를 출력하게 되므로 b2_out은 X 값과 유효한 데이터를 번갈아 가며 출력하는 것이다.



마지막으로 mem의 모든 데이터 8192개를 옮기고 나서 output.txt를 write하는 신호인 done과 reset신호 ars신호가 1이 들어오는 것을 확인할 수 있다. ars 신호가 1이 된 후에 b1_out, b2_out 모두 0이 출력되는 것을 확인할 수 잇다.



simulation을 마친 뒤에 프로젝트 안에서 output.txt를 검색 후 실행시키면 위 사진과 같이 0~1fff (16진수) 모두 write된 것을 확인할 수 있다.

MY_PE

1.CODE

```
always@(posedge aclk)begin

  if(aresetn==0)begin
    psumreg<=0;
    bin<=0;
    for(i=0; i<64; i=i+1) peram[i]<=0;
  end

  if(dvalid==1)begin
    psumreg <=psum;
  end

  if(we==1)begin
    peram[addr] <= din;
  end
  else if(we==0)begin
     bin <= peram[addr];
  end

end

endmodule
```

```
assign dout = psum;


floating_point_0 fp(
.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(psumreg),
.m_axis_result_tvalid (dvalid),
.m_axis_result_tdata (psum)
    );
```

일단 we 신호부터 보면 we 신호가 1로 들어오면 peram 메모리에 input으로 들어온 din이 저장되도록 하였고 we가 0이라면 peram 메모리 안에 있는 데이터를 bin에 초기화 시켜 floating point multiply 모듈로 연결시켜 줬다.

dvalid 값이 1이라면 floating point multiply과정이 끝났다는 뜻이므로 mac 연산을 위해 psum reg 값을 psumreg 값으로 초기화 시켜준다.

reset 신호인 aresetn 신호가 0이면 내부 메모리인 peram 을 모두 0으로 초기화하고 mac 연산을 위한 psumreg, bin 또한 0으로 초기화 시켜준다.

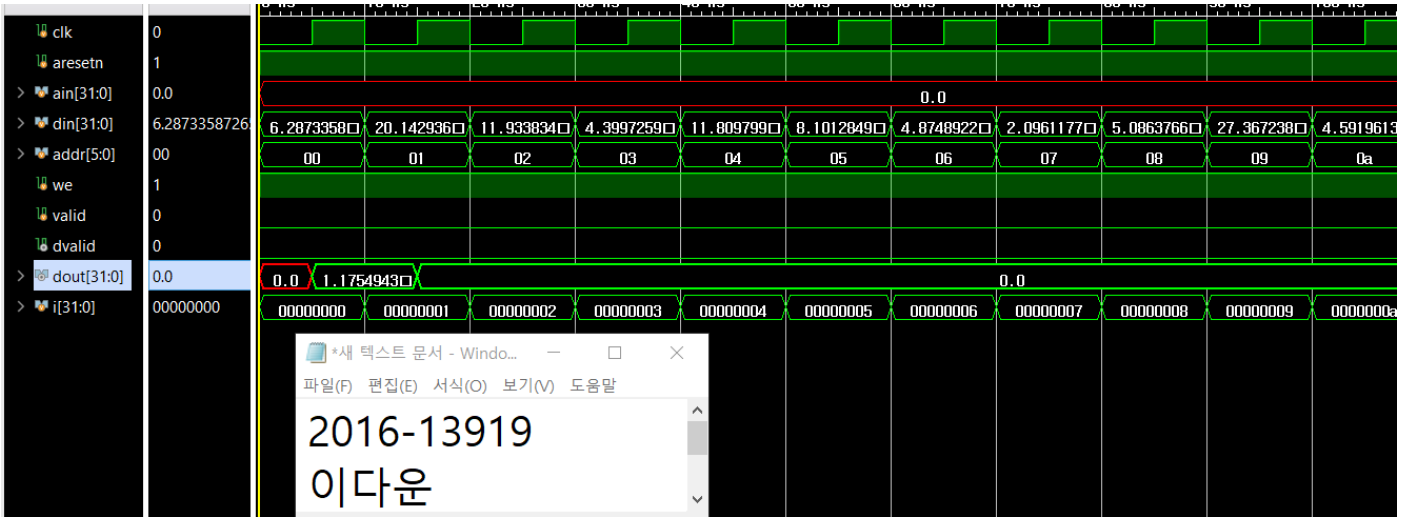
2. Testbench & Simulation

```
initial begin
  clk = 0;
  valid=0;
  addr=0;
  aresetn =1;

  we=1;
  for(i=0 ; i<16 ; i=i+1)begin
      din = $urandom%(2**31);
      din = {7'b0100000, din[24:0]};
      #10;
      addr=addr+1;
  end

  we=0;
  addr=0;
  #10;

  for(i=0; i<16; i=i+1)begin
      ain = $urandom%(2**31);
      ain = {7'b0100000, ain[24:0]};
      valid=1;
      #10 valid =0;
      wait(dvalid) addr=addr+1;
      #10;
  end
```

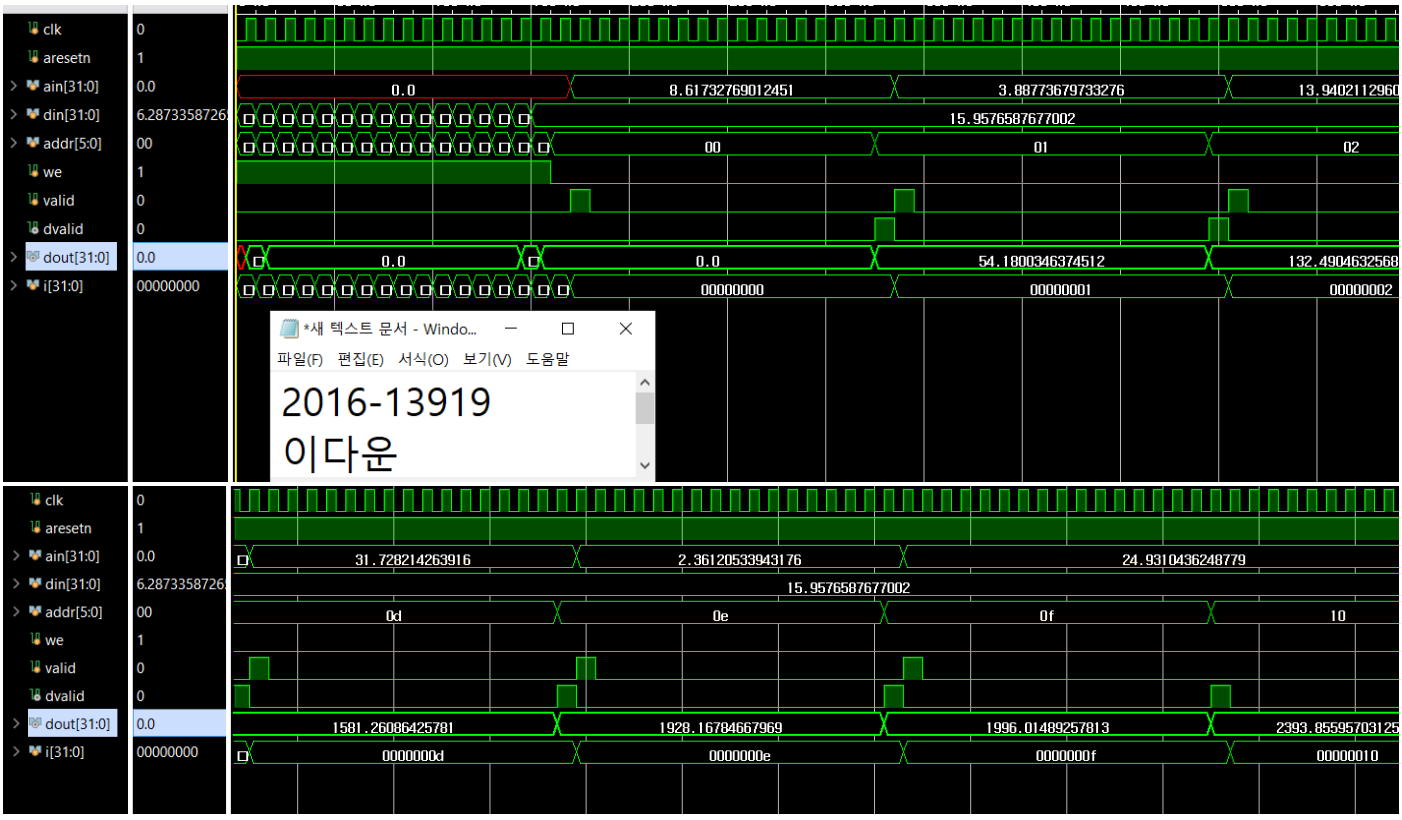
Testbench 코드를 짤 때 1순위로 고려해야하는 점이 있다. floating point multiply 연산이 16 cycle의 delay를 갖는다는 점이다. 그래서 이를 해결하기 위해 dvalid 신호가 나올 때만 코드가 진행 되도록 구현하였다.

먼저 for 문을 돌면서 addr 주소 0~16에 랜덤 값을 my_pe에 저장하였다. 이때 $urandom%($2^{31}$)만으로 랜덤 값을 생성하면 너무 큰 수가 생성되어 나중에 infinite가 빈번히 일어난다. 그래서 {7'b0100000, din[24:0]}와 같이 지수부분을 정해주었다.

그 후 16개의 새로운 랜덤 값을 받음과 동시에 peram에 저장된 값과 MAC 연산을 해주어야 한다. 그래서 마찬가지로 ain에 랜덤 값으로 초기화 하고 valid 값을 1을 주어 floaing_point_multiply 모듈이 연산을 시작하게 한다. 16 cycle delay를 고려하여 wait(dvalid) 문으로 연산이 끝난 뒤 다음 연산이 시작하게 했다.



처음 16개의 값을 peram에 저장하는 과정이다. din에 난수가 생성되는것과 addr값이 1씩 증가하는 것을 확인할 수 있다.



처음 16개의 데이터 입력이 끝난 뒤 MAC 연산을 하는 과정이다.

16개 데이터 입력이 끝난 뒤 WE 신호는 0이되고 valid 신호가 잠깐 1이 되어 연산을 시작 시킨다. 그리고 16사이클 후 dvalid 값이 잠깐 1이 되는 것을 확인할 수 있다. 이를 통해 delay가 16사이클 이라는 점도 확인 할 수 있다. 16사이클을 주기로 dvalid가 1이되고 그에 따라 결과값인 dout도 update된다. 계산기로 검산을 해보면 dout가 올바른 값을 출력한다는 것을 알 수 있다.