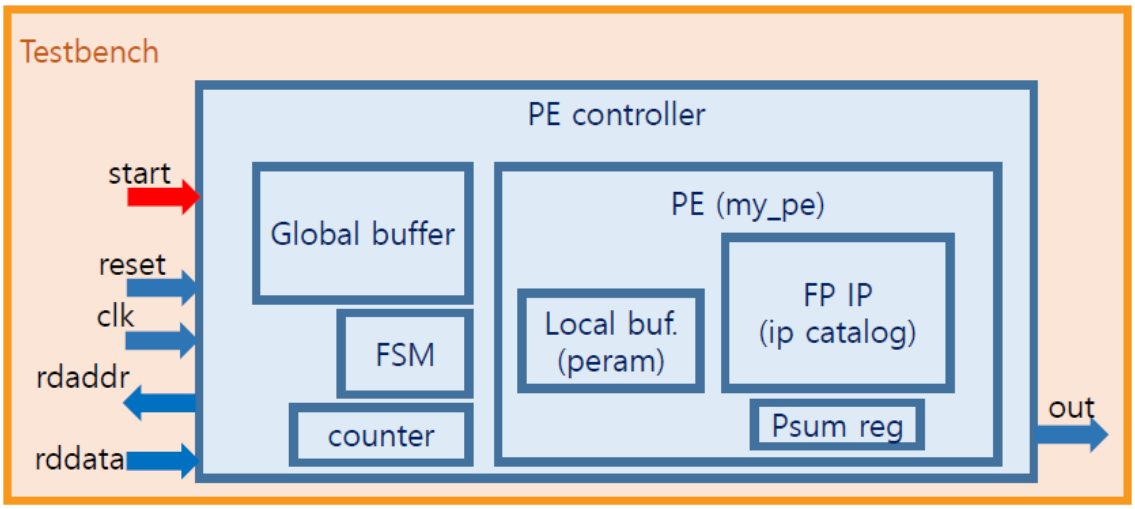
Hardware System Design

LAB06

이다운 2016-13919



LAB05 에서 구현한 my_pe module을 말그대로 control 하는 PE_controller를 구현하는 것이 이번 LAB의 목표다. PE_controller 에는 input 값을 저장하는 Global buffer, state에 따라 module이 동작할 수 있도록 하는 FSM 그리고 PE_controller와 my_pe 동기화 및 입력을 위한 counter가 있다.

PE_controller 는 IDLE, LOAD, CALC, DONE 이라는 4가지 state를 가진다. IDLE은 대기 상태를 의미하고 start 신호를 받으면 LOAD로 넘어간다. LOAD 단계에서는 input data를 받아 buffer에 저장하는 과정으로 앞의 절반은 global buffer에 뒤의 절반은 my_pe의 local buffer에 저장한다. LOAD 과정이 끝나면 자동으로 CALC 단계로 넘어가 floating point fused multiply 과정이 시작된다. floating point fused multiplier 모듈은 delay를 가지는데, 이를 고려하여 pe_controller는 delay cycle 수에 알아서 맞춰 연산을 한다. 연산이 끝난 후 state는 DONE으로 넘어간다. DONE state에서는 5사이클 동안 done이라는 output port를 1로 출력하여 연산이 끝났음을 외부에 알린다. 5사이클이 지난 후 state는 다시 IDLE로 돌아와 start 신호가 들어오기 전까지 대기상태를 유지한다.

1. Code

```
module pe_control#(
    parameter DATASIZE = 16
)
(
input start,
input areset,
input clk,
input [31:0] rddata,
output reg [5:0] raddr,
output reg done,
output  [31:0] wrdata,
output dvalid,
output [2:0] state
);
```

pe_controller는 DATASIZE라는 parameter를 받는다. 이번 랩에서는 총 32개의 data, 즉 16번의 연산을 하지만 data 개수가 다른 상황에서도 동작하기 위해 존재하는 것이다. input port로는 start, areset, clk, rddata가 있는데 rddata는 32bit 입력 port이다. output 으로는 raddr, done, wrdata를 가지며 dvalid 와 state는 testbench waveform에서 결과를 직관적으로 확인하기 위해 임시로 추가하였다.

```
reg [2:0] p_state,n_state;          //state
reg [4:0] counterdata, countercal;// fsm counter
reg [2:0] counter5;                // fsm counter
reg resetdata, reset5, resetcal;   // counter reset
reg  we;                           // my_pe enable
reg valid;                         // my_pe dvalid  출력용

wire [31:0] dout;                 // my_pe output
reg [31:0] ain,din;               // my_pe input
reg [5:0] addrin;
reg calflag, calflag2;            // buffer for delay
wire reset;
reg pe_reset;
wire pe_areset;
reg pe_reset_flag;
assign state = p_state;
assign pe_areset = ~pe_reset;

parameter IDLE = 3'd0, LOAD = 3'd1, CALC = 3'd2, DONE = 3'd3;

(* ram_style = "block" *) reg [31:0] globalram [0:2**6 - 1]; // global register
reg  [5:0] global_addr, local_addr;
```

```
always @(*)begin
  case(p_state)
    IDLE: if(start) n_state = LOAD; else n_state = p_state;
    LOAD: if(counterdata==31) n_state = CALC; else n_state = p_state;
    CALC: if(countercal ==16) n_state = DONE; else n_state = p_state;
    DONE: if(counter5 ==4) n_state = IDLE; else n_state = p_state;
  endcase
end
```

start 및 counter를 항시 체크하여 next state가 바뀌어야 하는지 아니면 유지되어야 하는지 판단하는 블록이다.

```
always@(*)begin
    case(p_state)
        LOAD: resetdata = 0;
        default: resetdata =1;
     endcase
end

always@(*)begin
    case(p_state)
       DONE: reset5 = 0;
       default: reset5 = 1;
    endcase
end

always@(*)begin
    case(p_state)
        CALC: resetcal = 0;
        default: resetcal =1;
     endcase
end
```

현재 state에 따라 counter의 reset값을 1로 할지 0으로 할지 판단하는 블록들이다. 현재 state에 해당하는 counter는 동작해야 하므로 0으로 두고 그 외 state에서는 동작하지 않도록 reset 신호에 1을 둔다.

```
always @(posedge clk or posedge resetdata)begin
  if(resetdata) counterdata <= 0;
  else begin
    if(counterdata ==31) counterdata <= 0; else counterdata <= counterdata + 1;
  end
end


always@(posedge clk or posedge reset5)begin
    if(reset5)begin counter5 <= 0; done <=0; end
    else begin
        if(counter5 < 4) begin
        done <= 1;
        counter5 <= counter5 + 1;
        end
        else begin
            done <=0;
        end
    end
end
```

위에서 확인한 reset신호에 따라 counter가 어떻게 동작하는지 나타내는 블록이다. 코드에서 확인할 수 있듯이 reset ==1 이면 counter는 0으로 유지하고 아니면 매 clk마다 1씩 더해지게 하였다. 이때 DONE의 counter가 작동할 때는 done 신호가 1을 가지도록 하였다.

```
always @(posedge clk or posedge resetcal)begin
    if(resetcal)begin countercal <= 0; calflag <=1; calflag2<=0; end
    else begin
        if(calflag ==1)begin calflag<=0; calflag2 <=1; end
        if(calflag2 ==1)begin valid<=1; calflag2<= 0; end
        if(valid ==1) valid <=0;
        if(dvalid ==1)begin
            countercal <= countercal +1;
            calflag<=1;
        end
    end
end
```

CALC의 counter는 앞의 counter와 약간 다르다. 앞의 counter는 clk 신호에 맞춰 값을 증가한 반면에 CALC은 my_pe의 delay에 맞춰 카운터가 1씩 증가해야 하므로 dvalid 신호가 1일 일 때 counter에 1씩 더하였다. 이때 dvalid가 한번에 2클럭 이상 나와 연속 2번 증가하는 것을 방지하기 위해 calflag라는 buffer를 사용하여 valid값이 1클럭 동안만 1의 값을 가지도록 하였다.

```
always@(posedge clk) begin
    case(p_state)

    LOAD: begin
        if(counterdata <= 15)begin
            globalram[counterdata] <= rddata;
            raddr <= counterdata;
        end
        if(counterdata > 15)begin
            we <= 1;
            raddr <= counterdata- 16;
            local_addr<= counterdata - 16;
            din <= rddata;
        end
    end

    CALC: begin
        we <= 0;
        ain <= globalram[countercal];
        local_addr <= countercal;
        raddr <= countercal;

    end

    default: begin
        raddr <= 0;

    end
```

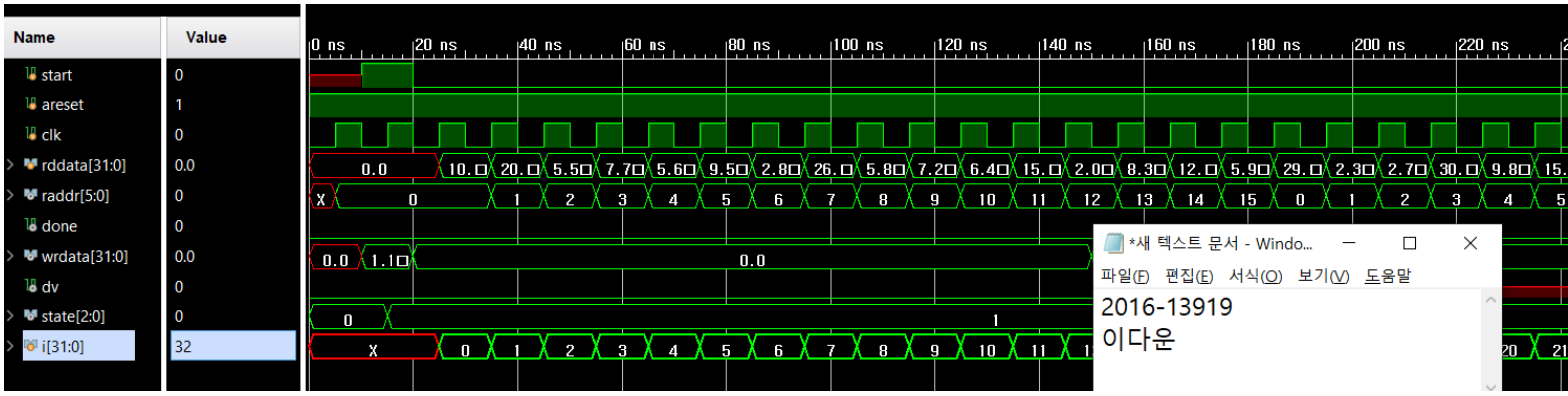
각 state에서 counter 값에 따라 input data와 그에 해당하는 주소 값을 지정하는 블록이다. global, local buffer에 들어갈 주소는 곧 counter 값과 동일하면 되므로 counter값을 raddr에, 매 clk마다 초기화 시켜주었고 그 raddr 주소에 해당하는 ram에 input data를 집어 넣어 주었다. LOAD, CALC state 외에는 raddr은 0 이 출력되게 하였다.

2. Testbench & result

```
initial begin

clk =0;
areset = 1;

#10;

start =1;
#10 start= 0;
#5;


for(i=0; i<32; i=i+1)begin
     rddata = $urandom%(2**31);
     rddata = {7'b0100000, rddata[24:0]};
     #10;

end
rddata = 0;
```

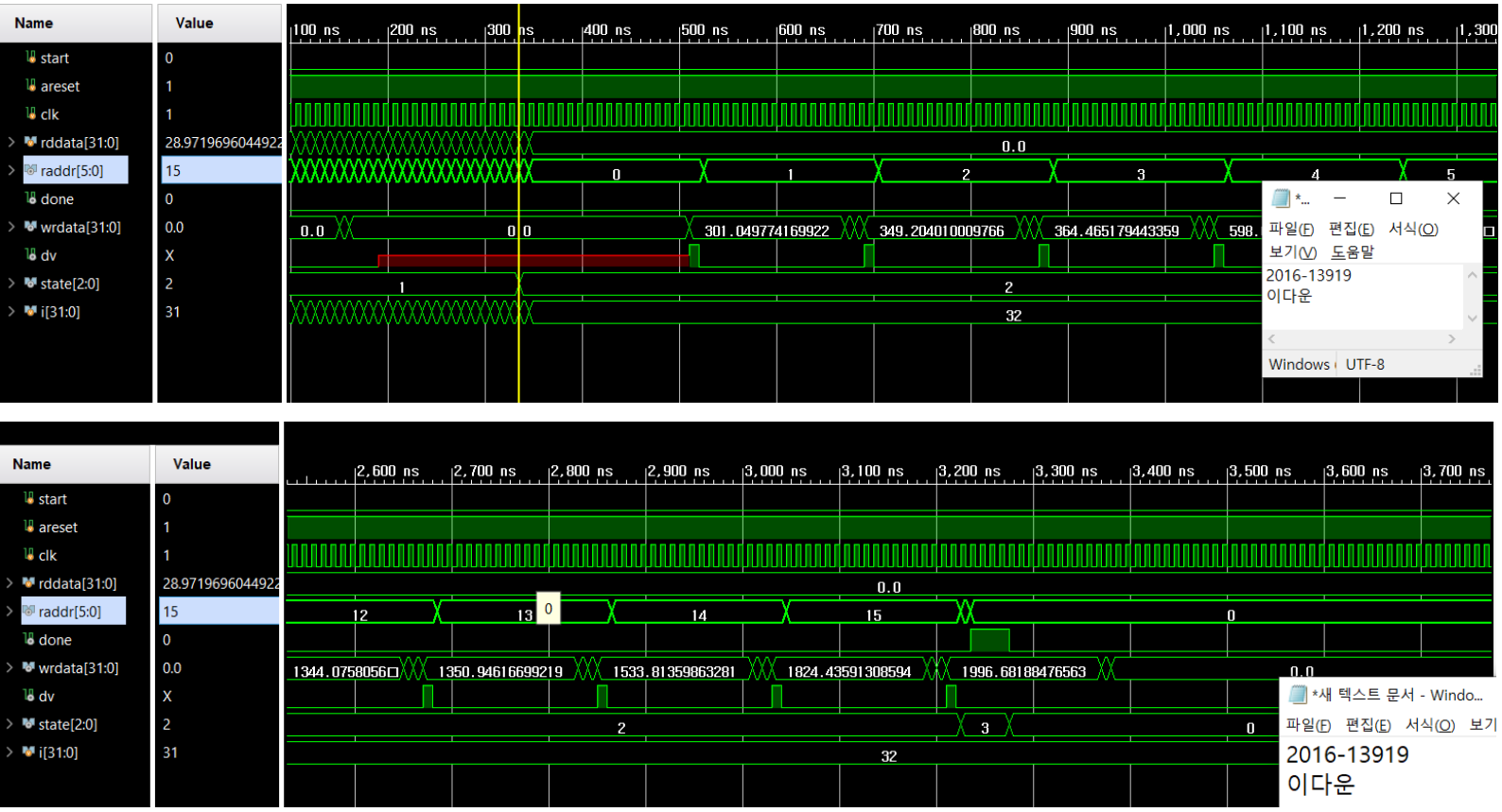
Testbench 코드는 비교적 간단하다. 처음에 start신호를 1 준다음에 LOAD state에 맞춰 input 32개 date를 urandom 함수와 for문으로 주었다. uramdon함수로 너무 큰 수가 생성되어 결과를 제대로 확인 못하는 경우를 막기위해 지수부분만 고정시켜 주는 연산도 하였다.

IDLE & LOAD



start에 1 값이 들어가고 해당 사이클에 state 값이 IDLE 에서 LOAD로 (0->1) 바뀌는 것을 확인할 수 있다. state가 바뀌고 바로 다음 사이클부터 input rddata값을 입력 받는데 이 입력값이 ram의 어느 주소에 저장되는지 raddr에 출력되는 것을 확인할 수 있다.

CALC & DONE



먼저 위의 스크린샷의 노란색 커서 줄을 보면 LOAD의 마지막 data가 입력됨과 동시에 state가 LOAD 에서 CALC로 바뀌는 것을 확인 가능하다. (1->2) 그리고 그 뒤 raddr은 몇 번째 연산을 진행 중인지 출력하게 되는데 각 단계의 연산을 끝났음은 my_pe에 연결되어 있는 dvalid가 나타낸다. dvalid가 1이 될 때마다 raddr 또한 1씩 증가함을 확인할 수 있다. 이렇게 총 16개의 곱연산이 끝난 뒤 state는 CALC에서 DONE으로 바뀌게 된다. (2->3) state가 바뀜과 동시에 done output port가 1을 출력하여 이를 외부에 알린다.

DONE state에서 5 사이클동안 대기한 뒤에 state는 다시 IDLE로 돌아가 대기상태에 놓이게 되는 것을 확인할 수 있다.

3. Discussion

state가 CALC일 때 wrdata가 중간중간 2사이클 씩 쓰레기 값이 출력됨을 waveform을 통해 확인할 수 있다. 하지만 pe_controller에서 실질적 정보는 done==1 때의 wrdata 값이므로 중간 쓰레기 값들은 무시해도 된다.

my_pe의 연산은 16사이클의 delay를 가진다. 하지만 pe_contorller에서 하나의 연산은 18사이클이 걸리는 것을 확인할 수 있다. 2사이클이 더 걸리는 이유는, dvalid 값이 바뀌는 것을 확인하고 counter 값을 올리는데 1사이클, counter 값이 올라가면 올라간 값을 address에 초기화하는데 1사이클씩 소모되기 때문이다. 이 소모되는 사이클을 줄이기 위해 always@(*) 블록을 사용하여 보았는데 pe_controller와 my_pe의 동기화가 어긋나 dvalid가 제대로 출력되지 않는 경우가 생길 수 있어 안정성을 위해 위처럼 두었다.

이번 LAB에서 구현하기 가장 어려웠던 부분은 CALC state 부분이었다. 다른 state의 counter는 clk 신호에 맞춰 증가하지만 CALC의 경우에는 delay에 맞춰 증가해야 하기 때문에 이를 구현하기 시행착오가 많았다. 결국에는 임시 buffer를 사용하여 위 문제를 해결하였다.