Practice 6 - BRAM to PE controller

2017-19428

컴퓨터공학부 서준원

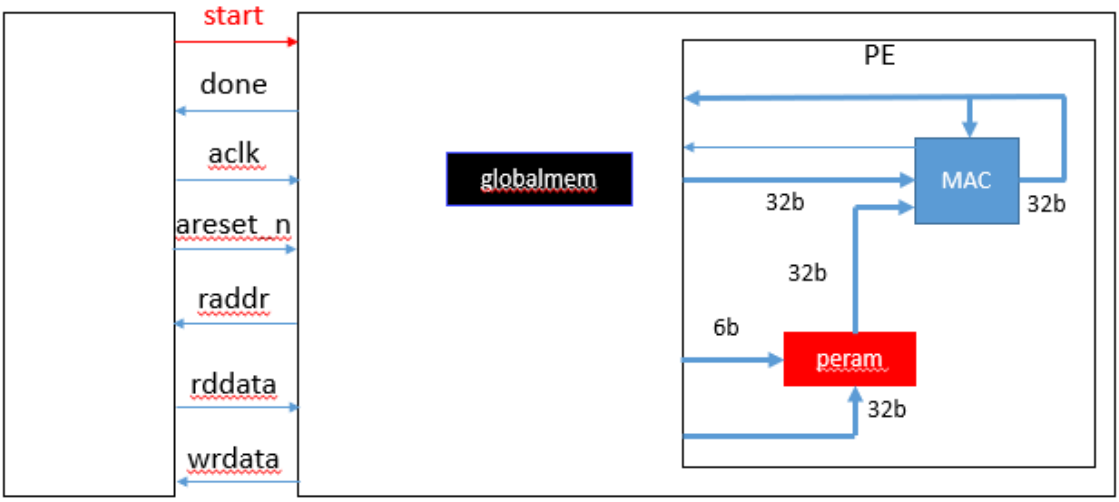
1. Introduction

이번 실습은 전 실습에서 만든 Processing Element (PE)를 테스트 벤치의 난잡한 조정 없이 안정적으로 사용할 수 있는 PE Controller를 만드는 것이었다. 우리의 최종 목표는 Matrix Multiplication을 위한 IP를 만드는 것이다. Matrix Multiplication의 핵심 계산을 수행하는 모듈이 PE이고 이를 위해서는 데이터를 읽고 이를 메모리에 저장하는 것뿐만 아니라 적절한 타이밍에 계산을 위해 데이터를 넣어주는 등의 작업을 해주어야 한다. 이를 위해 PE에 데이터를 넣어주는 등의 작업을 하는 Controller가 필요하다.

PE Controller는 다양한 State를 가지는 Finite State Machine으로 설계해야 한다. 총 4개의 State를 가진다. 데이터가 들어오는 것을 기다리는 S_IDLE, 데이터를 읽어오는 S_LOAD, Fused Multiplication을 이용하는 S_CALC, 마지막으로 계산이 모두 완료된 후 결과를 반환하는 S_DONE이 있다. 각각의 State에 대한 기술은 아래와 같다.

1) IDLE

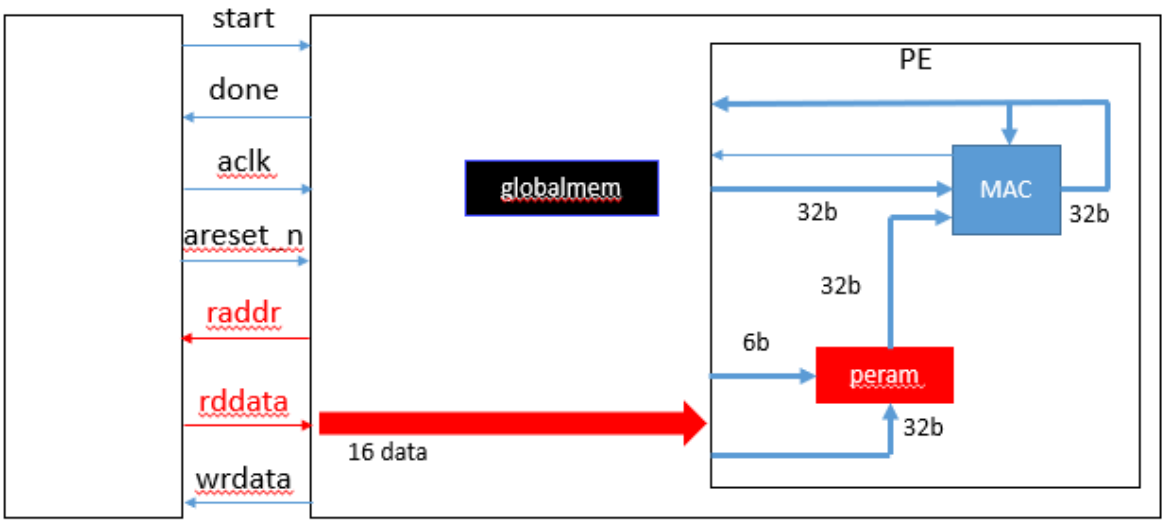
계산을 하고 있지 않은 상태이다. Start가 들어오면 Load 로 state가 바뀐다. Start는 synchronous한 input으로 구현되었다.



2) LOAD

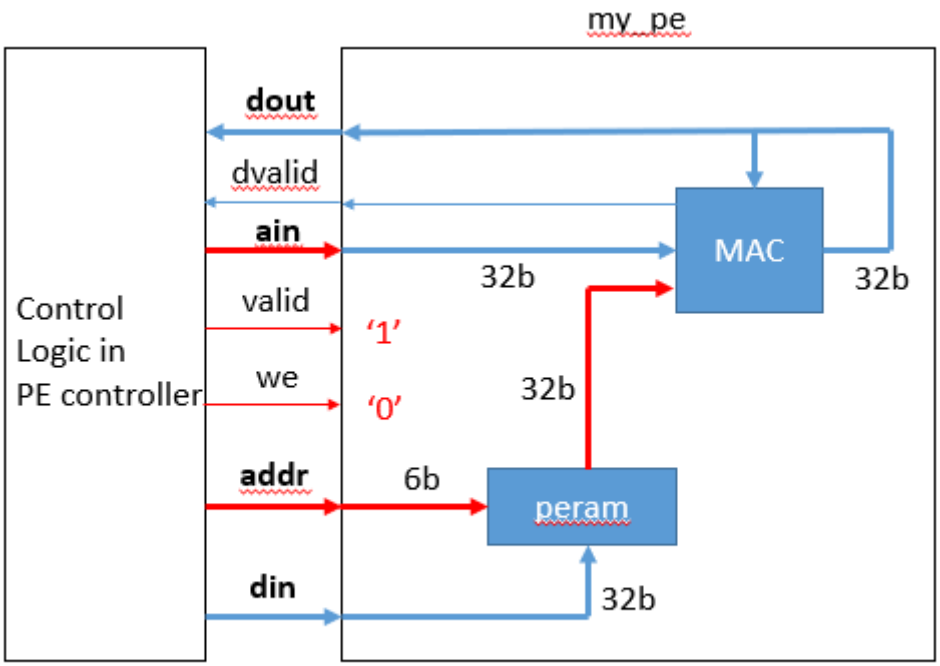
Start 이후 load state에서는 총 2 * vector size의 input이 rddata로부터 들어온다. 이는

BRAM으로부터 Matrix와 Input vector의 데이터가 들어오는 것으로, 여기서는 vector size의 크기는 16이었다. 추후의 이 사이즈는 변할 수 있으므로 parameter로 만들어야만 한다. 첫 16 input은 pe controller의 global mem에다가 저장한다. 이는 matrix의 데이터에 해당한다. 이후 나머지 데이터들은 pe의 내부 메모리에다가 저장한다. 이는 input vector이다.



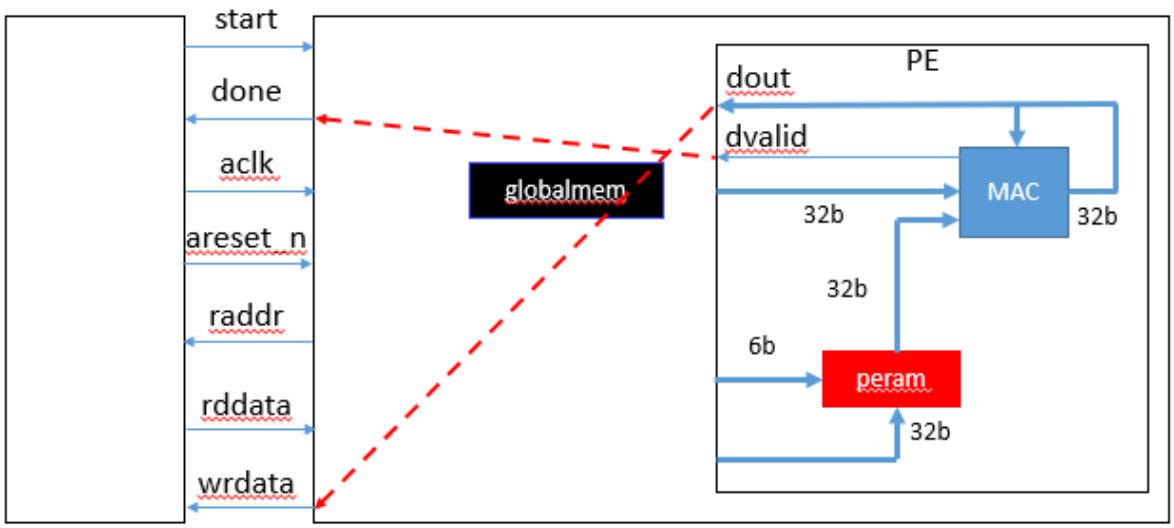
저장된 데이터에 해당하는 주소를 raddr에 반환해준다. 여기서 pe에 데이터를 저장할 때는 pe에 write_enable 시그널을 줘야만 한다.

3) CALC



계산하는 부분은 pe module에 적당한 input을 넣어줘야 한다. 또한 valid 값을 1로 세팅하여 결과를 계산해줘야 한다. 한 데이터 당 16사이클의 계산이 필요하기 때문에 (unblocking IP) 이 또한 고려해줘야 한다. 즉 16사이클의 계산을 vector_size(16) 번 만큼 수행한 후 calc는 끝나고 pe에서 dvalid를 1로 세팅하며 결과가 나온다.

4) DONE



계산이 모두 완료된 후에는 done을 1로 출력해준다. 이를 몇 사이클 정도 유지한 후 다시 idle로 돌아간다.

2. Implementation

A. Finite State Machine

FSM의 State를 parameter로 설정해 두었다. 그리고 present state와 next state를 지정해 clock의 rising edge마다 state를 바꾸어주었다. 만약 state가 바뀌지 않아야 한다면 next_state와 present_state가 같도록 설정했다.

```
parameter IDLE = 3'd0, INIT = 3'd1, LOAD = 3'd2, CALC = 3'd3, DONE = 3'd4;

    wire load_done;

    wire calc_done;

    wire done_done;

    reg [2:0] present_state, next_state;
```

// state shift

always @(posedge aclk)

if(!areset_n) present_state <= IDLE; else present_state <= next_state;

매 클럭의 rising edge마다 state를 바꾸어주는 부분이다. Present state에 next state를 넣어준다.

// state update

always @(*)

case(present_state)

IDLE : if(start) next_state = LOAD; else next_state = present_state;

LOAD : if(load_done) next_state = CALC; else next_state = present_state;

CALC : if(calc_done) next_state = DONE ; else next_state = present_state;

DONE : if(done_done) next_state = IDLE; else next_state = present_state;

endcase

State가 끝나야 하는 가에 대한 신호가 필요했다. 이는 후에 counter를 활용하는 것이다. 이를 기점으로 state를 바꾸게 된다. 이는 asyncronous하게 수행된다.

B. Counter

State를 변화시켜주기 위해 counter가 필요하다. 여기서는 Load, Calc, Done에서 모두 필요한 사이클의 횟수가 다르다. 나는 이를 별개의 counter로 만들지 않고 하나의 counter로 구현하였다. State가 변하게 될 때 counter의 initial value를 알맞게 세팅해준 후, 감소시키는 형태이다. 따라서 counter의 값이 0이 되면 state를 변화시켜줘야 한다는 시그널을 세팅해줘야 한다.

*localparam count_load = 2 * 2 * VECTOR_SIZE - 1; // 2 for memory load, 2 for global & local ram*

*localparam count_cal = VECTOR_SIZE * 16 - 1; // 16*

localparam count_done = 5; // 5

각각의 state마다 counter 값을 설정해주었다. 파라미터를 사용했다.

reg [31:0] counter;

```
wire [31:0] counter_val = (next_state == LOAD) ? count_load
: (next_state == CALC) ?  count_cal
: (next_state == DONE) ? count_done
: 0;

wire counter_start =  start || load_done || calc_done;
wire counter_down = (present_state == LOAD) || (present_state == CALC) || (present_state == DONE);
```

counter를 시작하거나, 감소시킬 여부를 판단한다.

```
always @(posedge aclk) begin
    if(!areset_n) counter <= 'd0;
    else
        if(counter_start) counter <= counter_val;
        else if(counter_down) counter <= counter-1;
end
assign load_done = (present_state == LOAD) && (counter == 'd0);
assign calc_done = (present_state == CALC) && (counter == 'd0) && dvalid;
assign done_done = (present_state == DONE) && (counter == 'd0);
assign done = (present_state == DONE);
```

counter의 값을 활용하여 state가 끝났는지 판단하여 시그널을 준다.

C. LOAD

Load는 처음에 global memory에, 후에는 local memory에 저장해준다. 주소는

counter를 활용해주어 적당한 counter 값을 설정해주었다. 또한 pe에 전달되는 we 시그널은 counter의 MSB를 이용했다. 처음에만 local에 저장된다.

assign we_local = ((present_state == LOAD) && counter[L_RAM_SIZE+1]) ? 1'd1 : 1'd0;

assign we_global = ((present_state == LOAD) && !counter[L_RAM_SIZE+1]) ? 1'd1 : 1'd0;

*// ** IMPORTANT : LOAD : one cycle for read data , one cycle for write data.*

// if counter[0] == 0 -> read && counter[0] == 1 -> write

assign addr = (present_state == LOAD) ? counter[L_RAM_SIZE:1] : // applicable for both global, local RAM

(present_state == CALC) ? counter / VECTOR_SIZE : 1'd0; //counter[31:4] : 1'd0;

// raddr : address for 2 RAMs

assign raddr = (present_state == LOAD) ? counter[L_RAM_SIZE+1:1] : 1'd0;

// din

assign din = (present_state == LOAD) ? rddata : 1'd0;

D. CALC

Calc는 pe에 적당한 값을 할당해주기만 하면 된다. Ain과 valid, 그리고 addr을 세팅해주었다.

assign ain = (present_state == CALC) ? peram[addr] : 1'd0;

assign valid = (present_state == CALC) ? 1'd1 : 1'd0;

E. DONE

Done인 경우 pe의 dout을 연결해주었다.

assign wrdata = (done) ? dout : 'd0;

3. Result

테스트를 위해 적당한 input 데이터를 만들어주었다. c언어를 사용하여 데이터를 hexa byte로 저장해준 후 파일을 만들었다. 간단한 테스트를 위해 matrix는 모두 1, vector는 모두 2로 했다. 32가 결과로 나와야 한다.

데이터를 변형하여 다양한 인풋에 대해서도 테스트해봤다. 1, 2의 경우에만 32가 나와도 일반성을 잃지 않고 충분히 값이 잘 나왔다고 볼 수 있어 보고서에는 이 경우의 결과만 기술했다.

```c
FILE *fp = fopen("input.txt", "w");

for(int i=0; i<16; i++){
    union a temp = {.y = 1.0};
    fprintf(fp, "%x\n", temp.x);
}
for(int i=0 ; i<16; i++){
    union a temp = {.y = 2.0};
    fprintf(fp, "%x\n", temp.x);
}

fclose(fp);
```

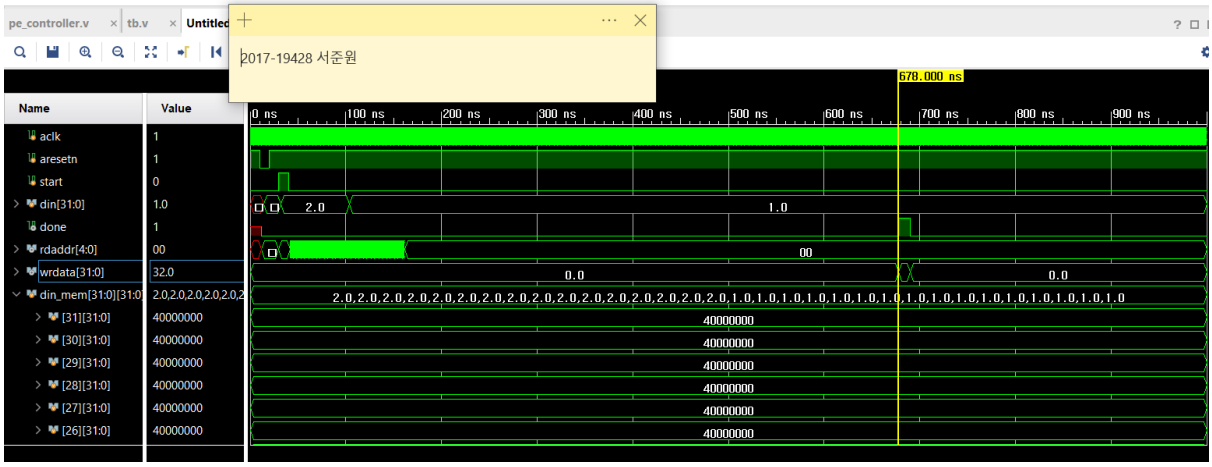
테스트벤치에는 이와 같이 데이터를 읽은 후 값을 넣어주어 간단하게 실행했다. 매 사이클마다 din에만 데이터를 다르게 해주었다. 이는 pe controller의 rddata에 연결된다.

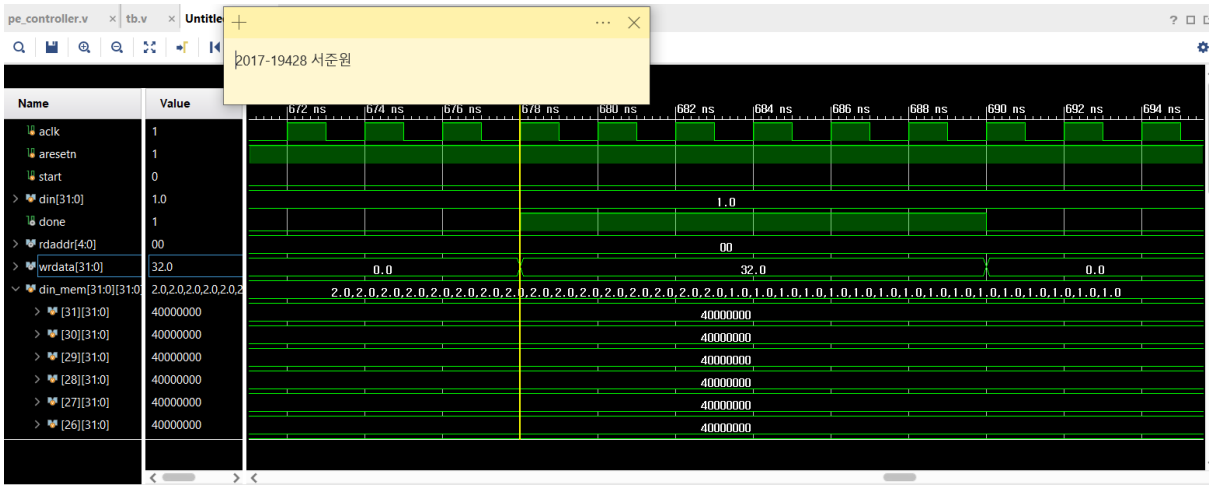
initial

$readmemh("input.txt", din_mem);

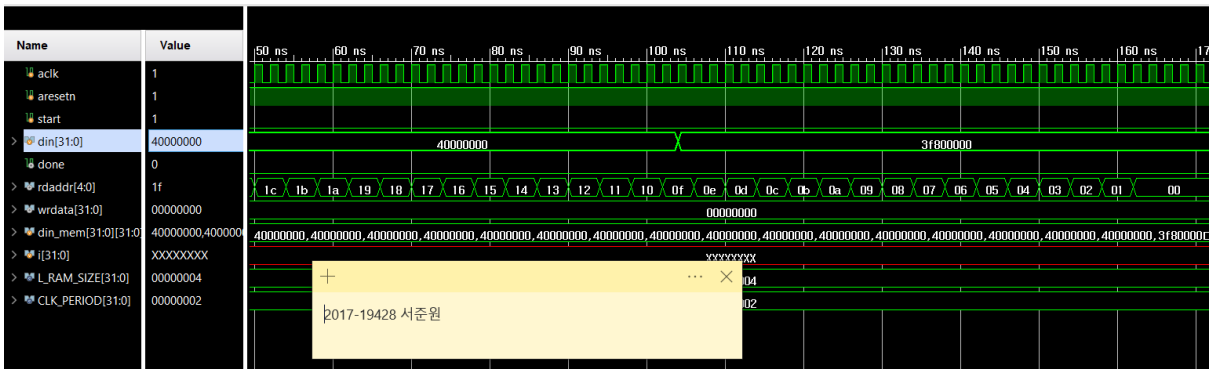
always @(posedge aclk)

din <= din_mem[rdaddr];





결과는 위 사진같이 나왔다. Done이 나옴과 동시에 결과가 32로 출력되었다. Done이 끝나면 다시 wrdata는 0으로 세팅되도록 구현이 잘 되었다.



또한 Load시에 rdaddr이 잘 변하는 것도 확인했다.