4190.301A Hardware System Design

Spring 2020

Hardware System Design Lab6 Report

Kim Bochang

2014-16757

1. <lab6> Introduce

Lab 6의 목적은 final project에서 사용할 Processing Element controller를 구현하는 것이다.

2. Implementation

2.1 PE controller (lab6₩lab6.srcs₩sources_1₩new₩my_pe_controller.v)

PE controller는 다음과 같은 state를 가지고 있고, 각 state마다 동작이 달라지게 된다.

즉, FSM의 형태를 띄고 있다.

S_INIT : 초기상태

S_LOAD : 외부에서 벡터를 받아와 pe 내부의 local buffer와 controller 내부의 global buffer에 벡터 곱을 계산할 수 있도록 값을 저장하는 state.

S_CALC : 저장된 값들을 이용해 벡터곱을 계산하는 state.

S_DONE : 계산이 끝났을때의 state.

```
module my_pe_controller #(
      parameter G_BUF_SIZE = 6, // global buffer size
      parameter L_RAM_SIZE = 6,  // local buffer size
      parameter VECTOR_NUM = 4, // vector number. 2**vector_num.
      parameter integer BRAM_ADDR_WIDTH = 15
    )
    (
        input start,
        input aclk,
        input aresetn,
        input [31:0] rddata,
        output [BRAM_ADDR_WIDTH-1:0] rdaddr,
        output reg [31:0] wrdata,
        output done
    );
```

포트는 위와 같다. G_BUF_SIZE와 L_RAM_SIZE는 각각 내부 global buffer와 local buffer size,

vector_num은 vector 내부의 원소 개수, BRAM_ADDR_WIDTH는 외부 BRAM address의 너비다.

간단하게 전체 구조를 설명하면 다음과 같다.

PE controller는 다음 4부분으로 나누어진다.

1. FSM

FSM의 역할은 state의 변화가 일어나야할때 state를 바꾸어 주는것이다.

S_INIT에서는 입력으로 start가 들어오면,

S_LOAD에서는 입력을 모두 내부 메모리에 저장했을 때 ,

S_CALC에서는 저장된 값들을 이용한 벡터곱이 완료되었을 때,

S_DONE에서는 5사이클이 지나면 다음 상태로 가게 된다.

코드는 다음과 같다.

```
always @(posedge aclk) begin
    if (!aresetn) begin
            state <= S_IDLE;
    end
    else begin
        case (state)
            S_IDLE:
                state <= (start) ? S_LOAD : S_IDLE;
            S_LOAD:
                state <= (load_done) ? S_CALC : S_LOAD;
            S_CALC:
                state <= (calc_done) ? S_DONE : S_CALC;
            S_DONE:
                state <= (done_done) ? S_IDLE : S_DONE;
            default:
                state <= S_IDLE;
        endcase
    end
end
```

단순히 reset 신호가 들어오면 state 를 초기화하고, state 가 바뀌어야 할 때를 인식하여 다음 state 로 바꾸어주는 역할을 한다.

2. Global buffer

외부에서 들어온 값을 저장하고, 벡터 곱 계산에 필요할 때 불러오는 역할을 한다.

```
// global buffer part
always @(posedge aclk) begin
        if (we_global) // write enable of global is 1, then write data into global buffer
            global_buffer[addr] <= rddata;
        else
            global_reg <= global_buffer[addr];
end
```

we_global (write_enable) 의 값에 따라, 이 값이 1이면 global buffer에 쓰기 작업을 진행하고,

이 값이 0이면 읽기를 수행한다.

global_buffer는

(* ram_style = "block" *) reg [31:0] global_buffer [0:2**G_BUF_SIZE - 1];

와 같이 선언되어 있다.

3. counter

값이 저장될 address를 지정하거나, 지금까지 얼마나 많은 값이 계산되었는가 등을 관측해야 할 때 counter는 필수불가결 하므로, 이러한 counter를 구현해서 사용한다.

여기서 구현한 counter는 down-counter로, 초기값은 1로 초기화되며, S_LOAD, S_CALC, S_DONE state에서 모두 사용하게 된다.

통상적인 up-counter를 사용하지 않고 down-counter를 사용하는 이유는 다음과 같다.

각 state마다 counter가 세어 줘야하는 개수가 다른데, up-counter를 사용하면 각 state마다 어떤 값에서 계산을 끝마칠지 각각 체크해 줘야 하므로 구현이 복잡해 지지만,
down-counter를 사용하면 각 state마다 초기값만 다르게 설정해주고, counter가 0이 되는 순간 계산을 끝마치면 되므로 상대적으로 구현이 간편해진다.

또한, 사용하지 않을 때 초기값을 1로 두는 이유도, 0으로 초기해놓고 사용하면, 계산이 끝마치는 순간을 체크해야할때 잘못 체크되어 state 변환이 일어나는 경우가 생길 수 있으므로, 이러한 경우를 막기위해 초기값을 1로 설정한다.

```
assign counter_reset = (!aresetn) | (done_done);
assign counter_load_signal = (load_trigger) | (calc_trigger) | (done_trigger);
assign counter_load_value = (load_trigger) ? LOAD_CVAL :
                            (calc_trigger) ? CALC_CVAL :
                            (done_trigger) ? DONE_CVAL : 32'd0;
assign counter_enable = (state == S_LOAD) | (state == S_CALC && dvalid == 2'b1) | (state == S_DONE);


always @(posedge aclk) begin
    if(counter_reset) begin
        counter <= 32'd1; // I initialize counter to 1 because detecting current state at end whens counter is zero,
        //initializing counter to 1 prevent false-detecting current state at end.
    end
    else if(counter_load_signal) begin
        counter <= counter_load_value;
    end
    else if(counter_enable) begin
        counter <= counter - 32'd1;
    end
end
```

counter 부분 코드는 위와 같다.

counter_reset은 카운터가 리셋되어야 하는경우를 지정하고,

coutner_load_signal은 카운터에 특정 값을 불러와야 할 때 사용한다.

counter_enable은 counter가 동작해 야할 때를 지정하기 위해 사용하게 된다.

이 카운터 하나를 여러 state에서 사용하는데, 각 state에서의 정확한 동작은 다음 파트에서 서술한다.

4. state_relative block

각 state마다 실행되는 부분을 구현한 부분이다.

S_INIT

이 단계에서는 할 것이 없다.

S_LOAD

카운터 초기값 : 4 * vector_size – 1 (vector_size = 2**vector_num)

카운터 작동상황 : clk posedge마다

vector – vector multiplication을 할때, vector간의 곱셈을 하기 위해서는 2 * vector size개수만큼의 원소를 불러와야만 한다. 때문에, 처음에 카운터를 4*vector_size – 1로 초기화한다.

2 * vector_size – 1이 아닌, 4 * vector_size – 1로 초기화 하는 이유는, 각 데이터를 불러오는데 2사이클씩을 소모해야하기 때문에, counter를 불러와야하는 원소의 개수 * 2정도로 초기화해두고, 각 clock마다 카운터의 값을 감소시키게 된다.

```
// LOAD part.
assign rdaddr = (state == S_LOAD) ? counter[BRAM_ADDR_WIDTH:1] : 'd0;
assign we_local = ((state == S_LOAD) && (counter[VECTOR_NUM+1])) ? 1'b1 : 1'b0;
assign we_global = ((state == S_LOAD) && (!counter[VECTOR_NUM+1])) ? 1'b1 : 1'b0;
assign addr = (state == S_LOAD) ? {{(L_RAM_SIZE - VECTOR_NUM){2'b0}} , counter[VECTOR_NUM:1]} :
              (state == S_CALC) ? counter : 'd0;
assign load_done = (state == S_LOAD && counter == 32'd0);
```

위와 같이, 카운터의 LSB를 제외한 상위 비트들을 통해서 외부에서 값을 불러올 address와, 어떤 위치에 값을 저장할지 (local buffer or global buffer)를 결정하게 된다.

그리고, counter가 0이되면 load가 끝났음을 load_done을 이용하여 FSM에 알려주게 된다.

이때, down counter 특성상 데이터를 저장하는 순서는 주소의 역순이 된다.

S_CALC

카운터 초기값 : vector_size – 1 (vector_size = 2**vector_num)

카운터 작동상황 : PE에서 계산이 한번 끝날 때마다. (dvalid = 1일때)

이제, 각 값들이 로컬 버퍼와 글로벌 버퍼에 정상적으로 저장되었다면, PE를 이용하여 이 값들을 이용, 벡터곱을 계산 해야한다. 따라서 counter를 벡터 크기-1로 초기화하고, 각 원소의 계산이 끝날 때마다 counter의 값을 하나씩 내려주어, 모든 값이 계산되었을 때 counter가 0이되는 순간 계산이 끝났음을 알게된다.

counter의 각 값은 각각 local buffer, global buffer에 저장되어 있는 값의 주소로 사용되어 해당 값들을 불러오게된다.

```
always @(posedge aclk) begin
    if(!aresetn) begin
        calc_input_need = 1'b0;
    end
    else if(calc_trigger_delay1) begin
        calc_input_need = 1'b1;
    end
    else if(dvalid_delay_1 && state == S_CALC) begin
        calc_input_need = 1'b1;
    end
    else begin
        calc_input_need = 1'b0;
    end
end

assign valid = (state == S_CALC) && calc_input_need;
assign calc_done = (state == S_CALC && counter == 32'd0 && dvalid == 1'b1);

always @(posedge aclk) begin
        if (!aresetn) begin
                wrdata <= 'd0;
        end
        else begin
            if (calc_done) begin
                    wrdata <= dout;
            end
            else begin
                    wrdata <= wrdata;
            end
```

xxx_delay1 레지스터는 기존 값 에다 1사이클 딜레이를 준 register이다.

calc_input_need라는 값을 이용하여, PE에 공급되는 값이 현재 계산에 사용될 수 있는지를 통보해준다.

즉, PE의 valid의 역할을 하는 값이다.

calc_trigger는 S_LOAD에서 S_CALC state로 전환되었을 때 1이 되는 값이다.

먼저 state가 전환되고 나서 카운터가 정상적으로 초기화 된 상태에서, counter 값을 주소로 이용하여 계산할 값을 PE에 공급하게 되고, 그 후 PE 내부에서 계산이 완료되고, 계산이 끝난 값이 정상적으로 저장될 수 있도록 한 사이클 여유를 주기 위해 dvalid_delay_1을 사용한다. 이 값이 true가 되었을때, S_CALC라면 다시 값을 공급하여 다음 계산을 진행하게 되고, 그 외에는 값을 사용하지 않게 한다.

이렇게 계산이 끝났다면, calc_done이라는 시그널이 1이되어 FSM에서 state 변화가 일어나게 되고, 외부로 출력될 결과값인 wrdata에 값을 저장하게 된다.

이때, down counter의 특성상 주어진 벡터의 앞부터 multiplication 하는것이 아니라, 뒤에서 앞으로, 역순으로 multiplication하게 된다.

S_DONE

카운터 초기값 : 5 - 1

카운터 작동상황 : clk의 posedge마다

```
// DONE part.
assign done_done = (state == S_DONE && counter == 32'd0);
assign done = (state == S_DONE);
```

계산이 끝난 후에는, 5사이클동안 기다리면서 S_INIT으로 넘어갈 준비를 하게 된다.

2.2 PE controller test code (lab6₩lab6.srcs₩sim_1₩new₩tb_my_pe_controller.v)

위를 테스트 하는 코드는 다음과 같다.

```
module tb_my_pe_controller();

parameter BRAM_ADDR_WIDTH = 5;

reg clk;
reg aresetn;
reg start;
reg [31:0] din;
wire done;
wire [BRAM_ADDR_WIDTH-1:0] rdaddr;
wire [31:0] wrdata;

reg [31:0] din_mem [0:2**BRAM_ADDR_WIDTH-1];

    initial begin
        clk<=0;
        start <= 0;
          din <= 0;

        din_mem[0] = 32'b00111111100000000000000000000000;  //1
        din_mem[1] = 32'b01000000000000000000000000000000;  //2
        din_mem[2] = 32'b01000000010000000000000000000000;  //3
        din_mem[3] = 32'b01000000100000000000000000000000;  //4
        din_mem[4] = 32'b01000000101000000000000000000000;  //5
        din_mem[5] = 32'b01000000110000000000000000000000;  //6
        din_mem[6] = 32'b01000000111000000000000000000000;  //7
        din_mem[7] = 32'b01000001000000000000000000000000;  //8
        din_mem[8] = 32'b01000001000100000000000000000000;  //9
        din_mem[9] = 32'b01000001001000000000000000000000;  //10
        din_mem[10] = 32'b01000001001100000000000000000000;  //11
```

먼저, 메모리 역할을 할 부분에 각각 0~31번지까지,

0~15번지에 1~16, 16~31번지에 1~16에 해당하는 single precision floating point 값을 저장한다.

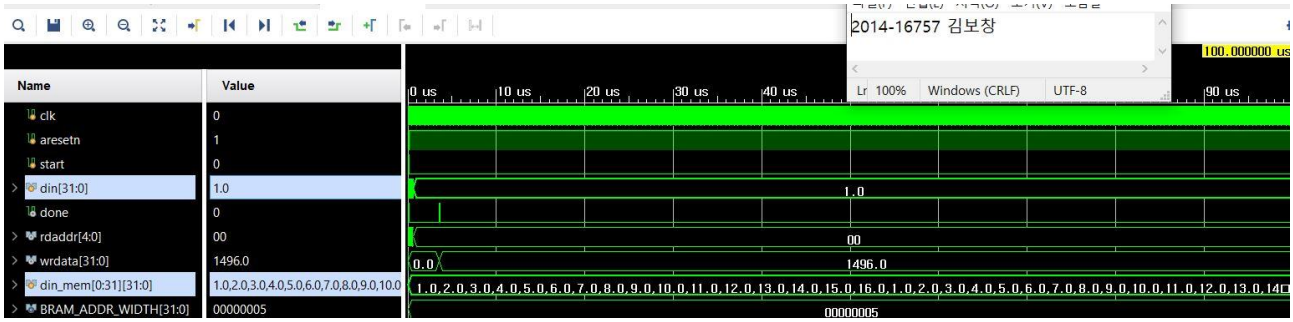
```
    din_mem[30] = 32'b01000001011100000000000000000000;  //15
    din_mem[31] = 32'b01000001100000000000000000000000;  //16

    aresetn <= 0;    //reset

    #20;

    aresetn <= 1;
    start <= 1;

    #30;

    start <= 0;
    #10;

end
```

그 후, 20 시간단위만큼 aresetn을 0으로 주어 리셋을 시키고, 그 후 start를 1로 주고, 계산이 끝날때까지 대기하게 된다.

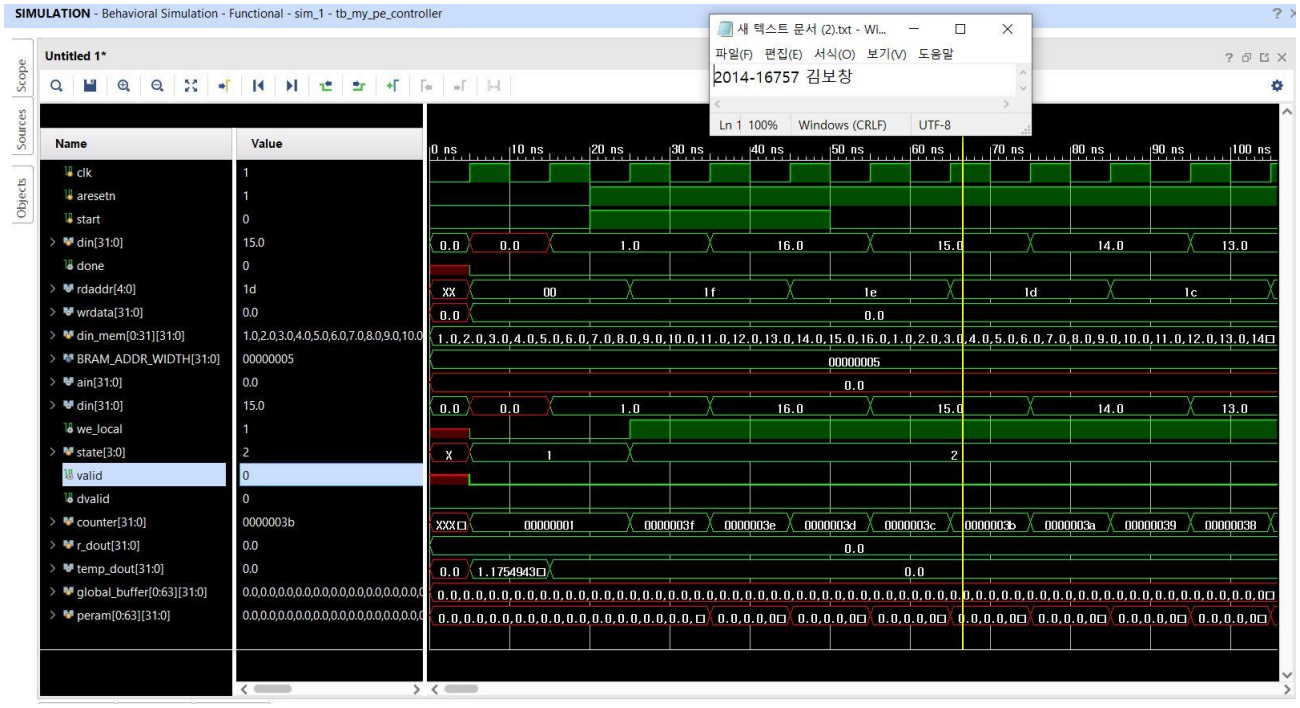
```
always #5 clk <= ~clk;

always @(posedge clk)
din <= din_mem[rdaddr];


my_pe_controller #(6, 6, 4 , BRAM_ADDR_WIDTH) PE_CTR (
  .start(start),
  .aclk(clk),
  .aresetn(aresetn),
  .rddata(din),
  .rdaddr(rdaddr),
  .wrdata(wrdata),
  .done(done)
);
```

클록 사이클은 10 시간단위가 된다.

3. Result & Discussion



<전체 waveform>



<S_INIT -> S_LOAD 부분>

먼저, 처음에 start가 1이 들어 왔을 때 S_INIT에서 S_LOAD로 바뀌는 부분이다.

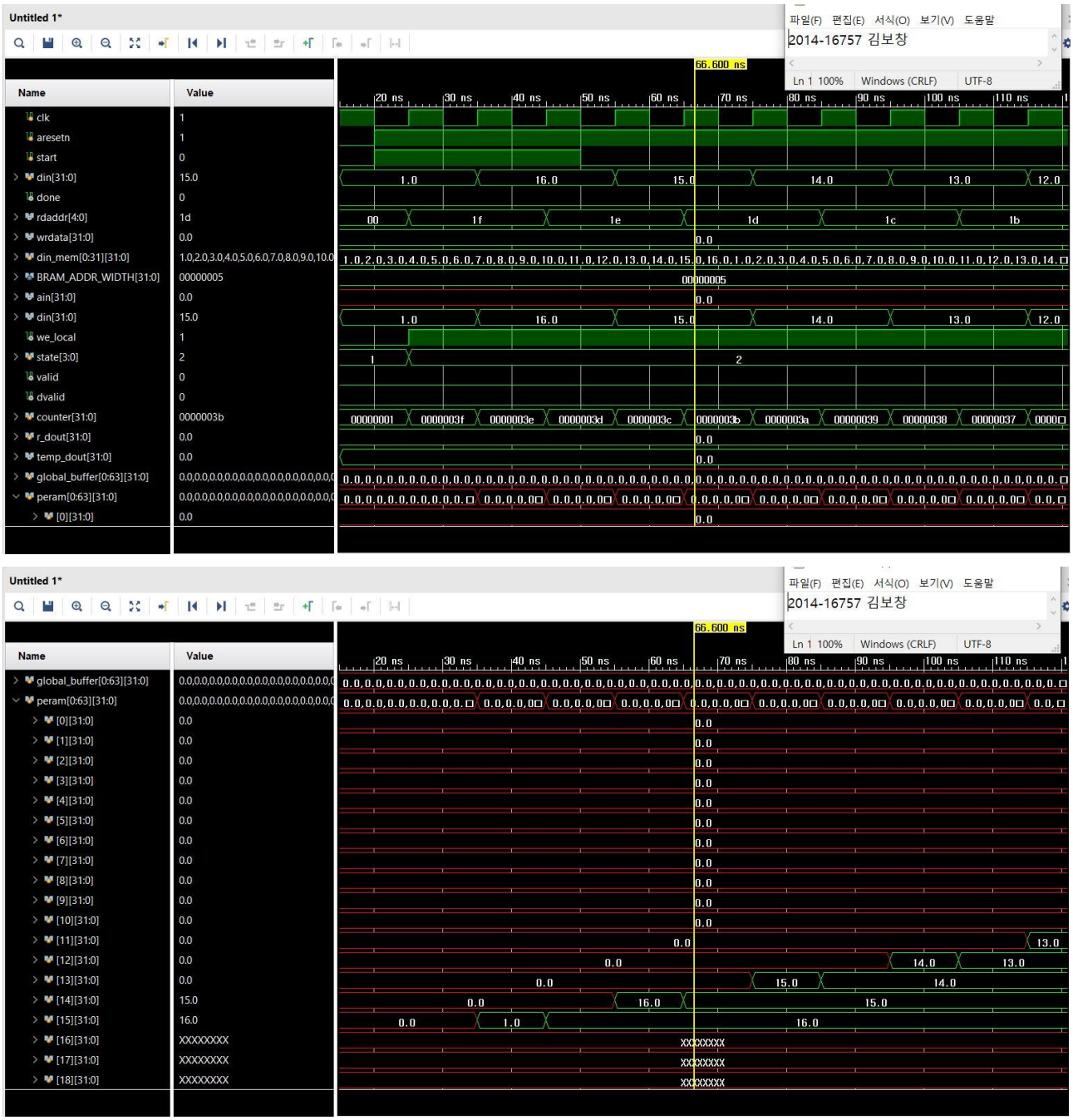
start가 1일때, clk의 posedge에서 state가 1 -> 2로 바뀌는 모습을 볼 수 있다. (25ns 부분)

state는 0001 : S_INIT, 0010 : S_LOAD, 0100 : S_CALC, 1000 : S_DONE

이므로, state change가 잘 작동했음을 알 수 있다.

동시에 local buffer에 값이 들어가야함을 나타내는 we_local도 1로 변하게 되고,

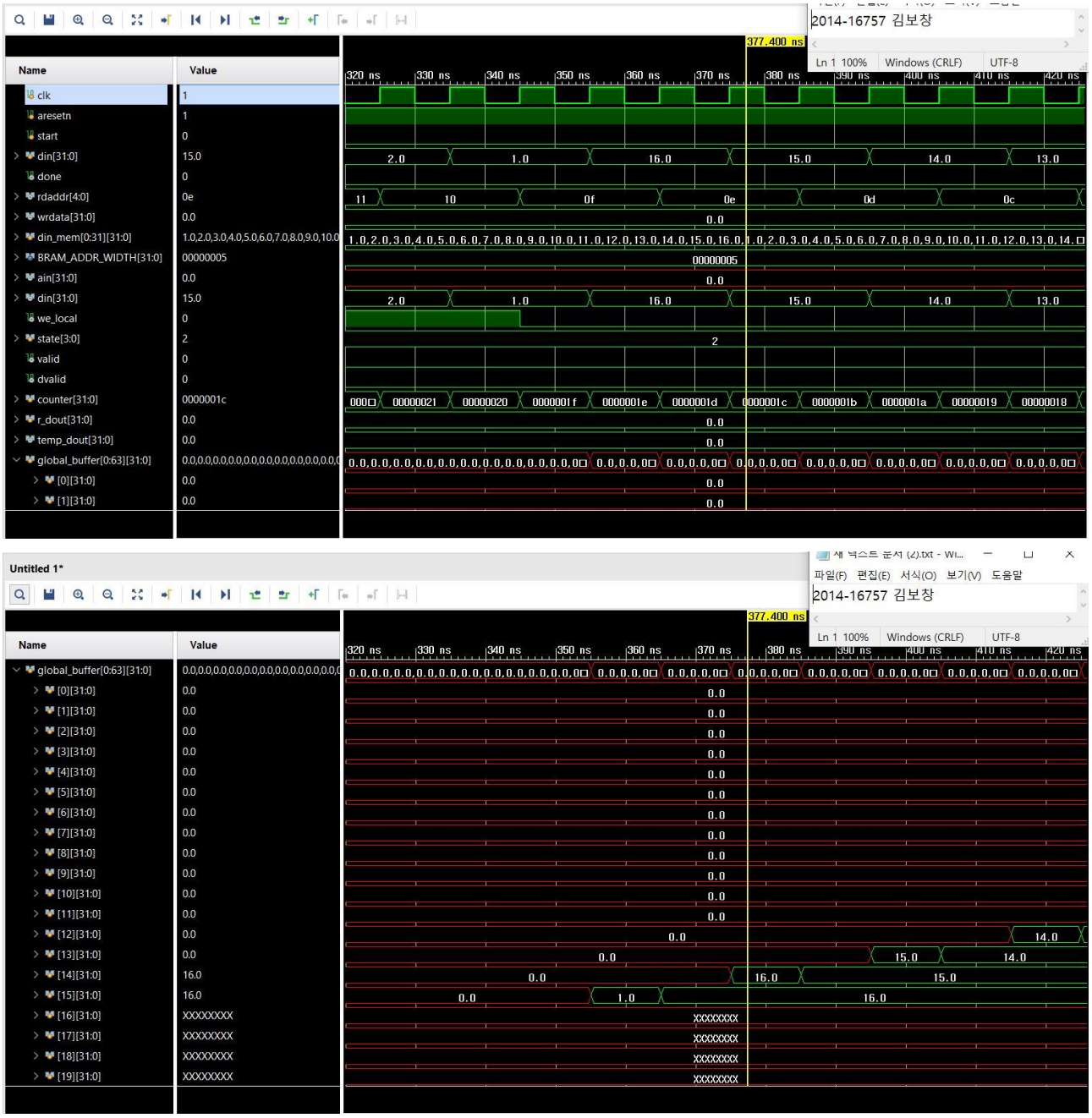
counter가 2 사이클마다 1씩 감소하는 모습을 볼 수 있다.



<local buffer에 값이 저장되는 모습>

address 31의 값이 localbuffer의 15, address 30의 값이 localbuffer의 14... 에 저장되는것을 알 수 있다.

즉, 의도한대로 local buffer에 값이 저장되는것을 확인할 수 있다.

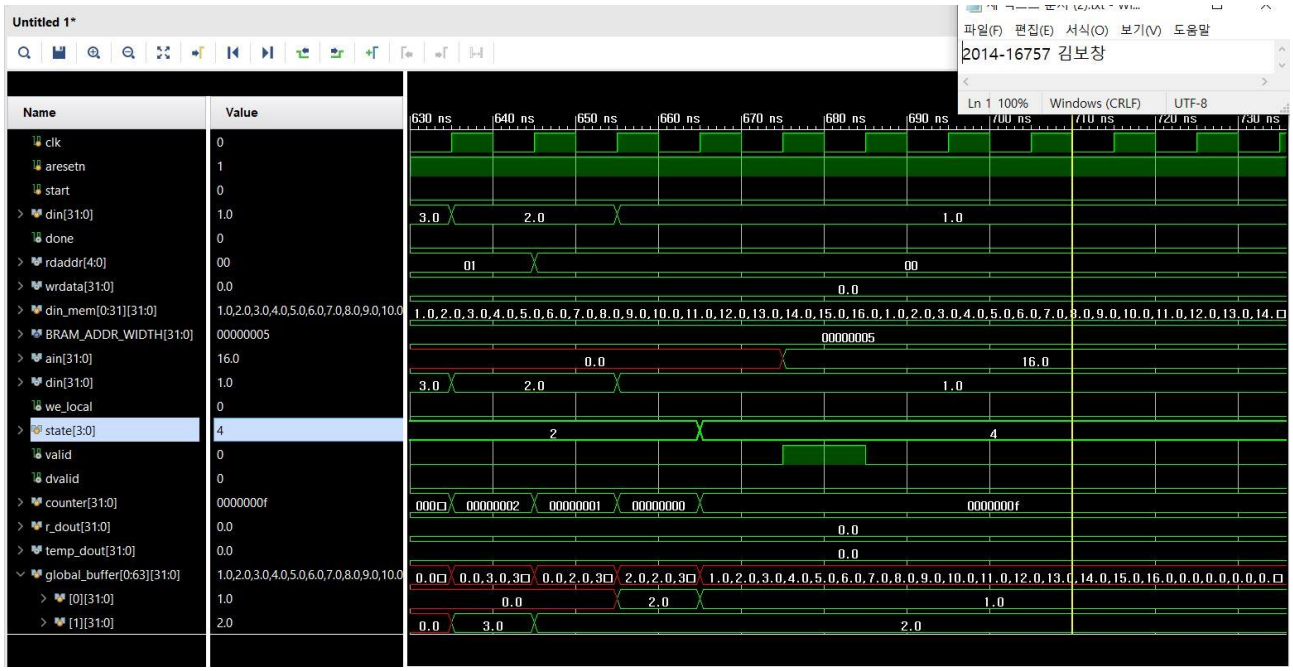


<global buffer에 값이 저장되는 모습>

local buffer에 값이 모두 저장된 후에는, we_local의 값이 0으로 떨어지고, 스크린샷에는 누락됐지만 we_global의 값이 1이되어 global buffer에 값이 저장되게 된다. (345ns부분)

실제로, global buffer의 15, 14... 번지에 각각 16.0, 15.0... 의 값이 저장되는것을 확인할 수 있다.

즉, global buffer에도 값이 정상적으로 저장됨을 알 수 있다.

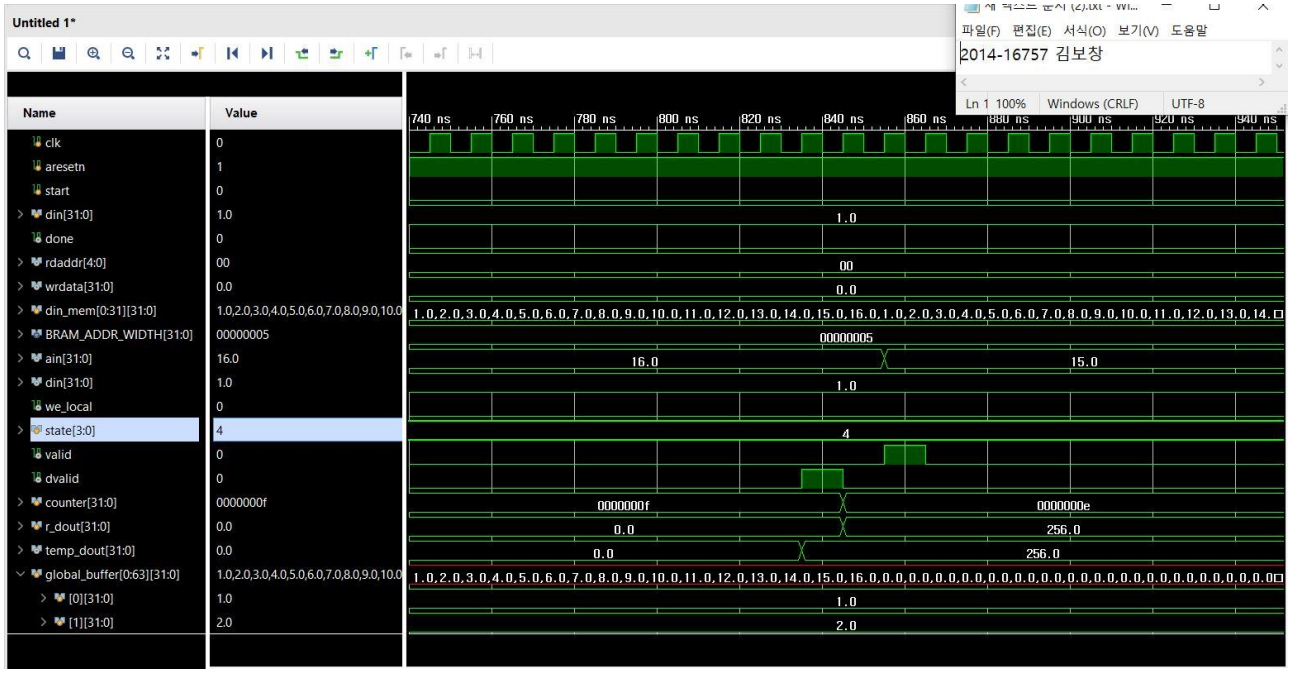


<load가 끝난 후>

load가 끝나고 난 후, 665ns부분에서 state가 S_CALC로 변환되게 된다.

그 후, counter를 초기화 하고, local, global buffer의 15번지에 저장된 16.0, 16.0을 가지고 곱셈을 진행하게 된다.

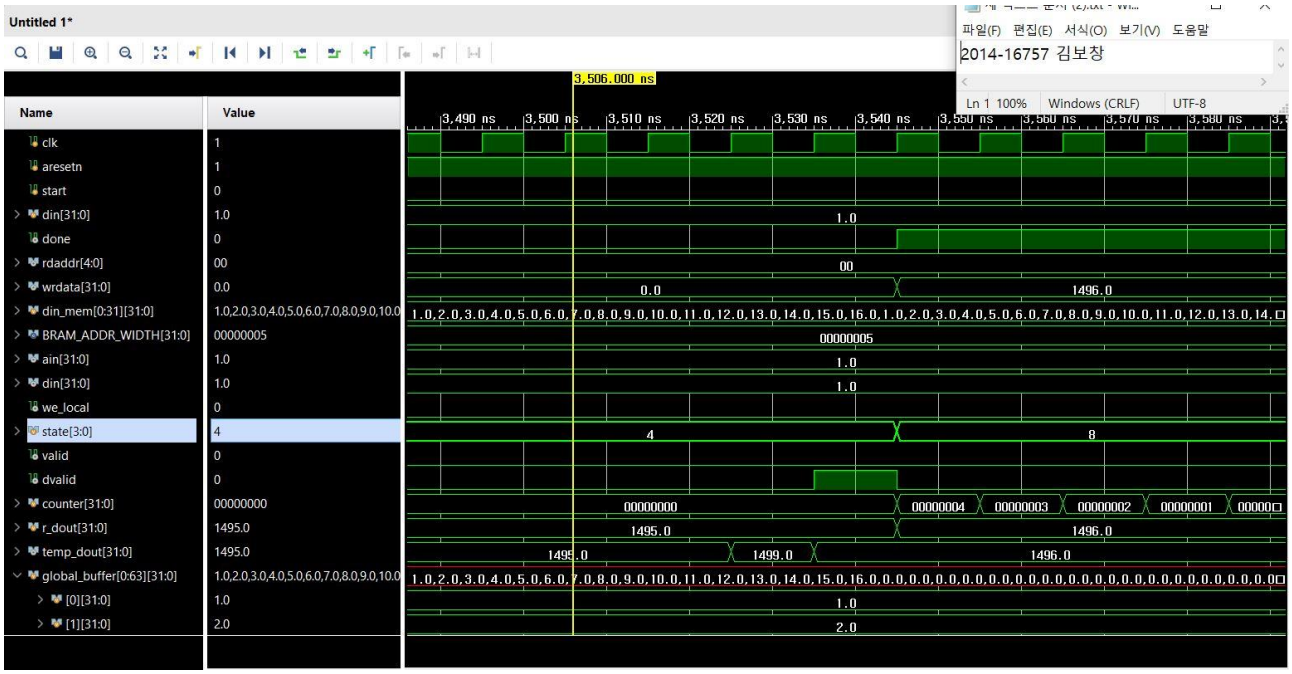
이때 valid가 1사이클동안 1이 됨을 확인할 수 있다.



<첫번째 계산 후>

첫번째 계산이 끝난 후, dvalid가 1일때 계산이 끝났음을 알 수 있고, 1사이클의 딜레이를 주어 내부 PE 모듈에 계산될 결과값이 성공적으로 누적되도록 한다. (r_dout)

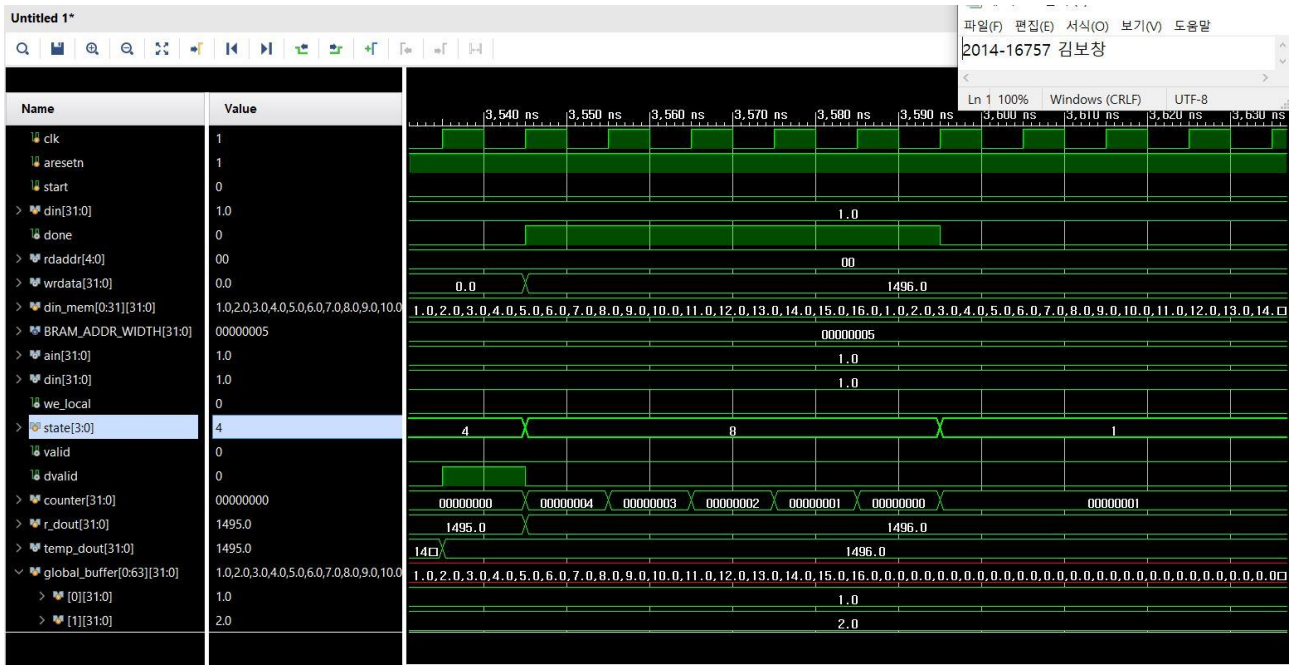
그 후, 카운터를 하나 내리고 valid 값을 다시 1로 주어 global buffer, local buffer의 14번지에 있는 15.0, 15.0 의 값을 fused multiply adder에 전달하여 계산하게 된다.



<모든 계산이 끝난 후>

모든 계산이 끝난 후에, state를 S_DONE으로 변환하고, 외부로 출력되는 done을 1로 만들게 된다.

결과인 wrdata를 보면, sum(k = 1 to 16) k^2 에 해당하는 1496의 값이 정상적으로 출력됨을 알 수 있다.



<S_DONE -> S_INIT>

그 후, 5사이클이 경과하고 S_DONE에서 S_INIT으로 다시 state가 변화하게 된다.

counter가 정상적으로 초기화되고, 작동함을 확인할 수 있다.

결과적으로, 내가 구현한 코드가 정상적으로 작동함을 확인할 수 있다.

이 코드를 synthesis, implement 한 결과는 같이 제출한 폴더에 사진으로 찍어두었다.

이번 과제를 구현하면서, 최적화에 대해서도 신경을 썼다.

우리 과제의 경우, state가 변화할 때 counter를 초기화 해야하므로, state의 변화를 체크하기 위해 처음에는 다음과 같은 방법으로 코드를 구현하였었다.

present_state와 next_state라는, 현재 상태와 다음 상태를 저장하는 레지스터 2개를 이용하여,

present_state에는 clk의 posedge마다 next_state의 값을 저장하게 하였다.

그리고 present_state와 next_state가 다를 때 load_trigger, calc_trigger, done_trigger와 같은 시그널이 1이 되게 한 후,

이 시그널을 이용하여 counter를 초기화 하였는데, 이렇게 카운터를 초기화 할 경우,

현재 state에서 다음 state로 넘어가는 상황을 체크할때 1cycle의 delay가 생기기 때문에 state가 바뀔 때마다 1cycle씩의 손해를 보게 된다.

따라서, 이러한 점을 제거하기 위해 state 레지스터를 하나로 줄이고, 각 state에서 다음 state로 바뀌는 조건 (xxx_done)을 이용하여 state가 변화하는 시점을 기존방법보다 1cycle 일찍 예측하게 하여, state를 바꿀 때 일어나는 delay를 줄였다.

또한, 각 state마다 counter를 두지 않고, counter 하나를 여러 state에서 돌려쓰게 하여 들어가는 회로의 개수를 많이 줄였다. 이를 구현하기 위해 일반적인 up-counter가 아닌 down-counter를 사용하였는데,

앞에서도 설명하였지만, state마다 counter를 돌려써야 하기 때문에 counter의 종료 조건이 각각 다른데,

up-counter의 경우 counter의 값이 종료조건에 맞는지를 비교하는 로직을 state마다 사용해야 하므로 드는 회로가 많아지게 된다.

반면, down_counter의 경우 counter의 초기화만 처음에 다르게 해주면, (이는 메모리 블록 하나로 처리가 됨)

종료조건에 맞는지는 단순히 counter가 0인지만 체크하면 되므로 up-counter 구현보다 효율적이다.

또한, 앞으로의 project에서 vector size가 바뀌어도 계산을 편하게 수행할 수 있도록 구현한 모듈이 VECTOR_NUM이라는 파라미터를 가지게 했다. 이 VECTOR_NUM이라는 파라미터를 바꿔주면,

예를들어 현재는 크기 16인 벡터곱을 계산하므로 VECTOR_NUM이 4이지만,

이를 6으로 바꾸어 주면 바로 크기 64인 벡터 곱을 계산하도록 할 수 있어 모듈의 재사용성에도 신경을 썼다.

4. Conclusion

비교적 커다란 모듈을 구현하면서, 어떻게 하드웨어를 설계해야 하는지 많이 생각해 볼 수 있었던 프로젝트였다.

지금까지의 프로젝트중에 가장 시간이 많이 걸렸는데, 특히 처음에 어떻게 시작해야할지 막막해서 꽤 많은 시간을 쓴 것 같다.

모듈이 비교적 크다 보니 여러가지를 동시에 생각해야하는 경우가 생겼고, 때문에 무작정 처음부터 효율적인 코드를 짜려 하다 꽤 많은 시간을 허비하게 되었다.

그 후 그러한 욕심을 버리고, 일단 돌아가게 구현하고 그 후 최적화를 하자는 마음가짐으로 코드를 구현하니 전보다 수월하게 구현할 수 있었다.

일단 돌아가게 구현하고, 그 후 최적화를 하는 것이 굉장히 좋은 방법이라는 것을 다시 확인할 수 있었던 좋은 랩이었다.