Hardware System Design 4190.309A
Seoul National University

Practice #6. BRAM to PE controller
Jiwon Lee, Sangjun Son

Goal

- Implement PE controller based on PE made in Practice #5.
  - PE controller consisted of PE and FSM
  - Inner product made with MAC operation of PE
- FSM controls PE to calculate inner product with several states.
  (e.g., `data1[0]*data2[0] + data1[1]*data2[1] + ··· + data1[15]*data2[15]`)

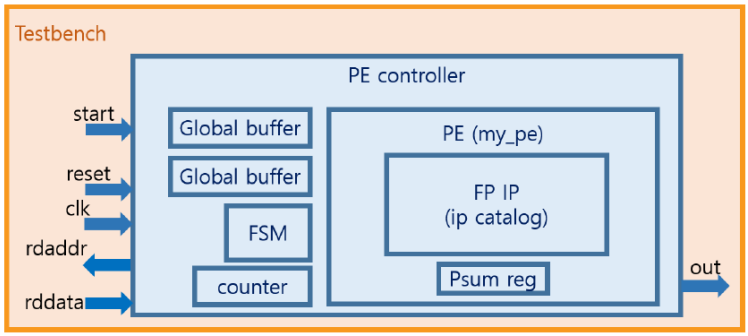

Figure 1: PE와 FSM이 결합된 2개의 벡터의 내적을 연산하는 PE Controller module이다 [2].

1 Implementation

이번 프로젝트는 지난 프로젝트에서 구현한 Processing Element와 이를 실행을 시키기 위해 사용되었던 테스트 벤치 코드를 결합한 새로운 모듈을 구현하는 것을 목적으로 한다. PE Control 모듈은 추후 구현할 Matrix-Matrix Multiplication을 수행하기 위해 존재하는 하위 모듈이다. 아래는 코드 구현과 함께 아이디어 및 기능에 대한 설명이 진행된다.

1.1 PE Controller

Figure 1을 참고하면 Practice #5에서 구현한 벡터의 내적을 계산할 수 있는 `my_pe` 모듈과 그것을 구성하는 Floating-point fused multiplier IP catalog가 존재한다. PE controller에서는 내적을 연산할 벡터의 모음을 Global buffer에 저장한 뒤 PE에 전달, 연산을 수행하게 된다. 상기된 Sequential logic 모듈이 상황에 따라 어떤 기능을 수행할 지 다르게 명시해주어야 하기 때문에 State를 정의하여 FSM의 논리를 모두 구현해준다.

PE controller는 IDLE, LOAD, CALC, DONE이라는 총 4가지 state를 가진다. IDLE은 대기 상태를 의미하며 이 때 `start` 신호가 입력되면 LOAD state로 전이된다. 이 상태에서는 `rdaddr` 주소를 Testbench에 전달하여 `rddata` 값을 하나씩 입력받아 Global buffer에 저장한다.

모든 LOAD 과정이 마치면 CALC 단계에서 FP IP catalog를 수행하게 된다. Global buffer에 저장된 16개의 실수 벡터 2개를 추출하여 Multiply-Accumulator를 이용해 내적 값을 구하게 된다. 연산이 모두 끝난 후 state는 DONE으로 넘어가게 되고 5 사이클의 Latency를 가지고 `done` signal과 함께 결과 값을 반환한다. 5 사이클 이후에는 다시 state를 IDLE 상태로 돌려 놓고 `start` 신호를 기다린다.

Hardware System Design 4190.309A
Seoul National University

Practice #6. BRAM to PE controller
Jiwon Lee, Sangjun Son

- 아래에 첨부된 코드 중 3-30 라인은 PE controller의 변수 선언에 관한 내용이다.
  - `parameter`에 포함된 (1) `L_RAM_SIZE`는 입력되는 벡터의 크기를 표현하는 상수이고 (2) `BITWIDTH`는 연산을 하기 위한 실수 자료형의 크기를 나타낸다. 나머지 입력과 출력에 관한 Wire는 모두 Figure 1에서 표기한 바와 같다.
  - FSM과 관련된 변수는 현재 상태와 앞으로 업데이트해야하는 상태 변수 `present_state`와 `next_state`가 있고 그 외에 Counter가 있다. 상태를 나타내는 상수는 총 네 가지로 0 ~ 3의 값을 `S_IDLE`, `S_LOAD`, `S_CALC`과 `S_DONE`에 해당하도록 설정하였다.
  - 내장 모듈인 MY_PE의 입출력 변수 `ain`, `bin`, `valid`, `dout`, `dvalid`를 함께 선언하고 지정시켜주어야 한다.
- 32-23 라인과 42-78 라인은 강의시간에 배운 FSM의 format을 그대로 차용하여 구현하였다. 초기 상태를 `reset` 신호와 함께 IDLE 상태로 초기화 한다. 현재 상태에서 어떤 조건이 중족되었을 때 다음 상태를 지정해주는 논리를 구현해준다.
  - IDLE 상태에서 `start` 신호가 들어올 경우 LOAD로 전환. (44 라인)
  - LOAD 상태에서 테스트 벤치에서 모든 데이터 (`cnt_load` 갯수 만큼의 데이터)를 전달받아 연산에 쓰일 값들이 Global buffer에 전부 저장이 되었을 경우 CALC로 전환. (45 라인)
  - CALC 상태에서 내장 모듈 MY_PE에서 2개의 벡터의 크기만큼의 연산을 모두 마쳤을 경우 (`cnt_calc` 갯수 만큼의 Fused multiplication 수행) DONE로 전환. (46 라인)
  - DONE 상태에서 Latency 만큼 딜레이가 진행된 후에 (Clock이 `cnt_done` 만큼 경과된 후) IDLE로 전환. (47 라인)
- 50-60라인은 각 상태에 위치했을 때 어떤 Counter를 활성화 시킬지 지정해주는 부분이다. 62-65라인에서는 현재 상태가 LOAD에 이르렀을 때 테스트벤치로 `rdaddr`을 전달해주고 해당하는 실수형 자료 `rddata`를 받아와 Global buffer에 저장한다.

MY_PE_CONTROLLER

```
1  `timescale 1ns / 1ps
2  module my_pe_controller #(
3      parameter L_RAM_SIZE = 6,
4      parameter BITWIDTH = 32
5  )(
6      input start,
7      input reset,
8      input clk,
9      output [L_RAM_SIZE:0] rdaddr,
10     input [BITWIDTH-1:0] rddata,
11     output [BITWIDTH-1:0] out,
12     output done
13 );
14     parameter DONE_LATENCY = 5;
15     parameter S_IDLE = 2'd0, S_LOAD = 2'd1, S_CALC = 2'd2, S_DONE = 2'd3;
16     reg [1:0] present_state, next_state;
17
18     reg [L_RAM_SIZE:0] cnt_load, cnt_calc;
19     reg [2:0] cnt_done;
20     reg rst_cnt_load, rst_cnt_calc, rst_cnt_done;
21
22     reg [BITWIDTH-1:0] gb1[0:2**L_RAM_SIZE-1];
23     reg [BITWIDTH-1:0] gb2[0:2**L_RAM_SIZE-1];
24
25     reg [BITWIDTH-1:0] ain;
26     reg [BITWIDTH-1:0] bin;
27     reg valid = 0;
28
29     wire [BITWIDTH-1:0] dout;
30     wire dvalid;
31
32     always @(posedge clk or posedge reset)
```

```
33          if (reset) present_state <= S_IDLE; else present_state <= next_state;
34
35      always @(posedge clk or posedge rst_cnt_load)
36          if (rst_cnt_load) cnt_load <= 0; else cnt_load <= cnt_load + 1;
37      always @(posedge clk or posedge rst_cnt_calc)
38          if (rst_cnt_calc) cnt_calc <= 0;
39      always @(posedge clk or posedge rst_cnt_done)
40          if (rst_cnt_done) cnt_done <= 0; else cnt_done <= cnt_done + 1;
41
42      always @(*)
43          case (present_state)
44              S_IDLE: if (start)                              next_state = S_LOAD; else next_state = present_state;
45              S_LOAD: if (cnt_load == 2**(L_RAM_SIZE+1)-1)    next_state = S_CALC; else next_state = present_state;
46              S_CALC: if (cnt_calc == 2**L_RAM_SIZE)          next_state = S_DONE; else next_state = present_state;
47              S_DONE: if (cnt_done == DONE_LATENCY-1)         next_state = S_IDLE; else next_state = present_state;
48          endcase
49
50      always @(*)
51          case (present_state)
52              S_LOAD: rst_cnt_load <= 0;
53              S_CALC: rst_cnt_calc <= 0;
54              S_DONE: rst_cnt_done <= 0;
55              default: begin
56                  rst_cnt_load <= 1;
57                  rst_cnt_calc <= 1;
58                  rst_cnt_done <= 1;
59              end
60          endcase
61
62      always @(rddata or present_state)
63          if (present_state == S_LOAD) begin
64              if (cnt_load < 2**L_RAM_SIZE) gb1[cnt_load] = rddata; else gb2[cnt_load-2**L_RAM_SIZE] = rddata;
65          end
66
67      always @(dvalid or present_state)
68          if (present_state == S_CALC) begin
69              if (dvalid) begin
70                  cnt_calc <= cnt_calc + 1;
71                  valid <= 0;
72              end
73              else begin
74                  ain <= gb1[cnt_calc];
75                  bin <= gb2[cnt_calc];
76                  valid <= 1;
77              end
78          end
79
80      assign rdaddr = present_state == S_LOAD ? cnt_load : 0;
81      assign out = present_state == S_DONE ? dout : 0;
82      assign done = present_state == S_DONE ? 1 : 0;
83
84      my_pe #(L_RAM_SIZE, BITWIDTH) MY_PE(
85          .aclk(clk),
86          .aresetn(~reset),
87          .ain(ain),
88          .bin(bin),
89          .valid(valid),
90          .dvalid(dvalid),
91          .dout(dout)
92      );
93  endmodule
```

코드의 하단부에는 지난 실습에서 구현한 Floating point MAC모듈의 호출 부분이다. Parameter에 들어가는 값은 모두 변수명과 동일하게 매칭을 해주었다 [1, 3].

2 Result

2.1 Testbench Implementation

아래의 코드는 MY_PE_CONTROLLER의 구현의 Validity를 확인하기 위해 검증 시나리오를 설정하고 그에 맞게 구현한 것이다. MY_PE_CONTROLLER 인스턴스를 입력 벡터의 크기, $2^{\texttt{L_RAM_SIZE}} = 2^4 = 16$와 저장되는 실수의 자료형 크기, $\texttt{BITWIDTH} = 32$를 사용하여 선언하였다.
전체적인 testbench의 수행순서는 다음과 같다. 1. 두 개의 벡터 데이터의 초기화 및 설정. 2. floating_point_MAC의 정상적인 동작을 위한 기본설정. 3. MY_PE_CONTROLLER의 동작 과정에서 data load요청이 들어왔을 때, 요청한 주솟값의 데이터 전달.

아래는 작성된 코드의 자세한 설명이다.

- 내적 연산을 위해서는 두 개의 벡터가 필요하고, 하나의 벡터당 크기가 16이므로, 총 32개의 데이터가 초기화되는 셈이다. 따라서, 먼저 `gb1`(첫번째 벡터), `gb2`(두번째 벡터)의 값들을 초기화 시켜주었다. 초기화된 데이터의 전체 주소는 0x00-0x1f 이고, `gb1`가 0x00-0x0f, `gb2`가 0x10-0x1f의 주소를 갖는다 (16-19 라인).

- testbench에서 가장 먼저 수행한 작업은 floating_point_MAC을 초기화한 것이다. reset 값을 1로 설정한 후, 적절한 clock cycle만큼 기다려주었다. 그 이후, start 값을 1로, reset 값을 0으로 설정하여 MY_PE_CONTROLLER가 동작하도록 하였다 (21-26 라인).

- `always` 구문을 사용하여, rdaddr(주솟값의 데이터 요청)이 들어왔을 때, 해당 주소의 벡터 데이터를 제공해준다 (30-31 라인).

TB_MY_PE_CONTROLLER

```
1  `timescale 1ns / 1ps
2  module tb_my_pe_controller #(
3      parameter L_RAM_SIZE = 4,
4      parameter BITWIDTH = 32
5  )();
6      reg [BITWIDTH-1:0] gb[0:2**(L_RAM_SIZE+1)-1];
7      reg [BITWIDTH-1:0] rddata;
8      wire [BITWIDTH-1:0] out;
9      wire [L_RAM_SIZE:0] rdaddr;
10     wire done;
11
12     reg start, clk, reset;
13     integer i;
14
15     initial begin
16         for(i = 0; i < 2**L_RAM_SIZE; i = i+1) begin
17             gb[i]                 = $urandom_range(2**30, 2**30+2**24);
18             gb[2**L_RAM_SIZE + i] = $urandom_range(2**30, 2**30+2**24);
19         end
20
21         clk <= 0;
22         start <= 0;
23         reset <= 1; #20;
24
25         start <= 1;
26         reset <= 0; #20;
27     end
28
29     always #5 clk = ~clk;
30     always @(rdaddr) begin
31         rddata = gb[rdaddr];
32     end
33
34     my_pe_controller #(L_RAM_SIZE, BITWIDTH) MY_PE_CONTROLLER(
```

```
35          .start(start),
36          .reset(reset),
37          .clk(clk),
38          .rdaddr(rdaddr),
39          .rddata(rddata),
40          .out(out),
41          .done(done)
42      );
43  endmodule
```

2.2 Simulation Results

Section 2.1에서 구현된 테스트 벤치를 실행하면 아래 Figure 2와 같은 Waveform을 확인할 수 있다. 결과를 자세히 보면, `start`가 high일 때 MY_PE_CONTROLLER가 동작을 시작하고, LOAD_STATE일 때, 주솟값을 요청하여 데이터를 받고, CALC_STATE일 때, floating_point_MAC의 동작으로 출력값을 계산하는 것을 볼 수 있다. 모든 계산이 완료되면, `done`이 high가 되며, 최종 결과값을 `out`으로 출력한다.

아래의 Table 1은 `TB_MY_PE_CONTROLLER` 17-18 라인에서 생성된 테스트 데이터이며 벡터 `ain`과 `bin`의 내적 결과값을 구하기 위해 각 원소별로 곱셈 결과값을 3열에 연산하였고 이를 누적하여 마지막에 약 323.0319 라는 결과값을 가지는 것을 확인할 수 있다. 아래의 Figure 3의 `CALC` 상태와 `DONE` Waveform을 확인하면 같은 값이 출력됨을 알 수 있다.

index	ain	bin	dout
1	7.80244779586791	5.81660604476929	45.3837650134422
2	4.89655828475952	4.23463726043701	66.1189131739864
3	3.40448760986328	2.32309341430664	74.0278559195483
4	2.79091644287109	3.30574178695679	83.2539050286521
5	6.46279287338257	6.47845220565796	125.122799773928
6	6.84624338150024	2.63594484329224	143.169119711317
7	7.12470054626465	2.21409893035889	158.943911569929
8	3.94174790382385	5.58761501312256	180.96888133528
9	6.90677261352539	3.36496090888977	204.209901186383
10	6.85968828201294	6.03147268295288	245.583923672916
11	3.57905983924866	3.37159180641174	257.651052501584
12	3.80960464477539	3.03244590759277	269.20347251618
13	5.47310543060303	3.23944854736328	286.933315952913
14	2.04824304580688	2.31567335128784	291.676377791048
15	5.22220277786255	3.8579089641571	311.82316070041
16	3.34818053245544	3.34771490097046	323.031914560051

Table 1: Testbench dataset

Practice #6. BRAM to PE controller
Jiwon Lee, Sangjun Son

Figure 2: `IDLE`, `LOAD`, `CALC` 전환이 되는 동안의 Waveform이다. 중간에 `state`를 출력하여 Controller 모듈의 FSM이 제대로 된 기능을 하는지 볼 수 있다. 또한 `CALC`에서 하위 모듈 `MY_PE`에 올바른 입력과 출력이 이뤄지는지 확인을 위해 `ain`, `bin`, `dout`을 함께 비교하였다.

2.3 Design Implementation

Figure 4는 구현된 MY_PE_CONTROLLER를 synthesis, implementation을 하였을 때 나오는 implementation design 결과이다.

Figure 3: `CALC`, `DONE`, 다음 `IDLE` 전환이 되는 동안의 Waveform이다. 벡터 내적 결과값이 `out`에 `done` 신호와 함께 5 Cycle 동안 지속되고 `start` 신호가 활성화 되어 같은 작업을 기존 `out`에 누적하여 연산을 반복하는 것을 볼 수 있다.

3 Conclusion

이번 과제는 고려해야 할 것들이 많았다. counter를 이용하여 LOAD 데이터 갯수 설정, floating_point_MAC의 계산 횟수, DONE_STATE에서의 latency 설정을 제어해 주었다.

CALC_STATE에서 latency에 관계없이 올바른 결과값이 출력되도록 하는 부분의 구현이 까다로웠는데, 먼저 MY_PE에서 dvalid값을 clk에 동기화 시켜주었다. 이후에, MY_PE_CONTROLLER에서 dvalid가 high이고, 모든 계산을 마쳤을 때 state를 바꿔줌으로써 done state일 때 올바른 결과값을 출력해내었다.

이번 과제를 수행하면서 올바른 시뮬레이션 결과를 얻었지만, synthesis와 implementation이 정상적으로 되지 않는 경우가 있었다. 이를 고쳐나가면서, 어느 구문들이 implementation이 되지 않는지, net이 동시에 업데이트 되어 충돌되는 부분은 없는지 등을 확인하며 구현해야겠다고 생각하였다.

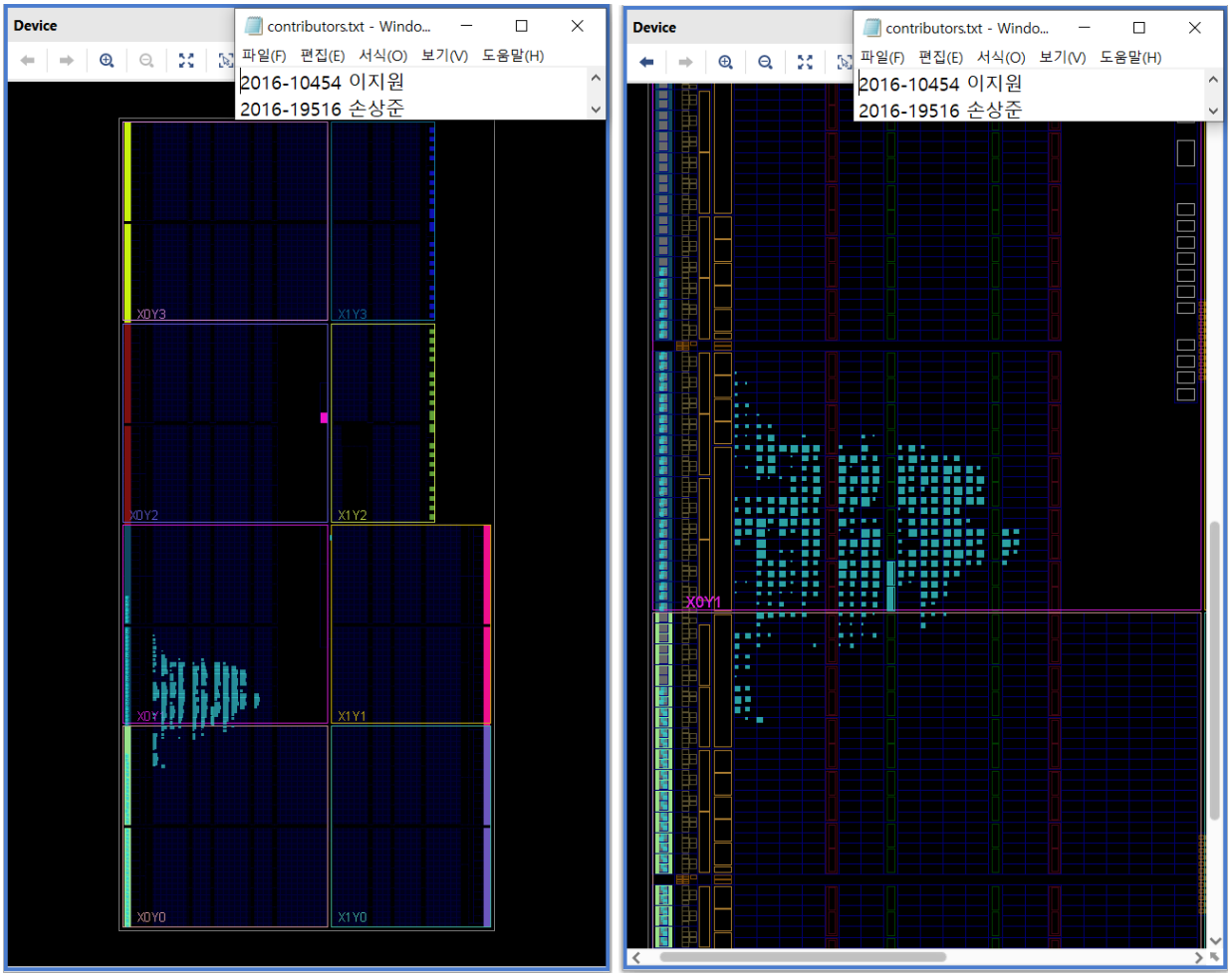

Figure 4: Implementation design, `MY_PE_CONTROLLER` 모듈을 Synthesis와 Implementation을 진행하고 보드에 어느 부분을 사용하는 지 시각화 시킨 것이다. 좌측에 전반적인 보드와 특히 많이 사용된 `X0Y1`를 확대한 결과이다.

References

[1] Computing Memory Architecture Lab. *Practice 5: PE implementation and BRAM modeling*. Hardware System Design, April 2021.

[2] Computing Memory Architecture Lab. *Practice 6: BRAM to PE controller*. Hardware System Design, April 2021.

[3] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.