Hardware System Design 4190.309A
Seoul National University

Practice #5. PE implementation & BRAM modeling
Jiwon Lee, Sangjun Son

Goal

- Implement BRAM model & test bench according to scenarios.
  - Make test bench that instantiates two BRAMs and initialize one BRAM to store address as data.
  - Copy every data from the initialized BRAM to the other BRAM.
- Implement PE with floating point fused multiply adder.

1 Implementation

이번 프로젝트는 Block Random Access Memory와 Processing Element를 각각 구현함으로써 추후 구현할 Matrix-Matrix Multiplication을 수행하기 위한 기본 모듈을 구성하는 것을 목적으로 한다. 아래는 코드 구현과 함께 간략한 아이디어 및 기능에 대한 설명이 (1) BRAM, (2) PE 순으로 진행된다.

1.1 Block Random Access Memory, BRAM

BRAM의 경우 크게 두 부분으로 이뤄져 있다. (1) 모듈이 실행이 되기 전 INIT_FILE에서 내부 메모리 `mem`를 읽는 부분과 done 신호가 주어졌을 때, OUT_FILE에 `mem`의 상태를 출력하는 부분과, (2) EN, RST, WE 신호가 들어왔을 때 경우에 따라 입력되는 데이터를 `mem`에 읽고 쓰는 역할을 한다.

- 아래에 첨부된 코드 중 21-28 라인의 외부 파일 입출력에 관한 구현 보면, `initial` 구문을 이용하여 모듈의 생성과 동시에 파일 입출력에 대한 실행 구문에 대한 scope를 지정한다. `$readmemh` 로 시작함으로써 INIT_FILE 파일을 읽어 `mem`에 저장한다. 그 후 `done` 신호 들어올 때 까지 대기하다가 신호가 들어오면 `$writememh` 함수를 사용해 OUT_FILE에 `mem` 데이터를 저장한다. 이 때 함수에 붙어있는 `$readmem`**h** 와 `$writemem`**h**의 **h**는 hexadecimal로 파일에 저장하는 값을 16진수 형태로 저장하는 옵션을 의미한다 [3].

- 30-46 라인은 BRAM의 input으로 주어지는 신호에 따라 모듈로써의 기능을 구현하는 부분이다. BRAM_CLK와 BRAM_RST 그리고 BRAM_EN, BRAM_WE의 신호에 따라 읽기, 쓰기, 초기화, 파일 출력을 위한 기능을 수행하게 된다. BRAM_RST은 BRAM_CLK에 Async로, BRAM_EN, BRAM_WE는 Sync로 구현하였다.
  - BRAM_RST이 posedge일 경우 BRAM_RDDATA에는 0을 할당한다. (31 라인)
  - BRAM_EN이 활성화되어 있고 BRAM_WE 또한 활성화되어 있다면, True를 가지는 bit에 해당하는 영역을 BRAM_WRDATA에서 `mem`으로 복사한다. (35-38 라인)

$$\texttt{mem}[\texttt{addr}][8*(i+1)-1:8*i] \leftarrow \text{BRAM\_WRDATA}[8*(i+1)-1:8*i] \tag{1}$$

  - BRAM_EN이 활성화되어 있고 BRAM_WE이 활성화되어 있다면, 메모리로부터 데이터를 읽어오는 기능을 수행한다. Read에 걸리는 싸이클이 2 cycle이 걸리도록 구현을 해야하기 때문에 `dout`을 버퍼로 사용해 1 cycle이 추가되도록 한다. (41-42 라인)

MY_BRAM

```
`timescale 1ns / 1ps
module my_bram #(
    parameter integer BRAM_ADDR_WIDTH = 15,
    parameter INIT_FILE = "input.txt",
    parameter OUT_FILE = "output.txt"
)(
    input wire [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR,
    input wire BRAM_CLK,
    input wire [31:0] BRAM_WRDATA,
    output reg [31:0] BRAM_RDDATA,
    input wire BRAM_EN,
    input wire BRAM_RST,
    input wire [3:0] BRAM_WE,
    input wire done
);
    reg [31:0] mem[0:8191];
    wire [BRAM_ADDR_WIDTH-3:0] addr = BRAM_ADDR[BRAM_ADDR_WIDTH-1:2];
    reg [31:0] dout;

    initial begin
        if (INIT_FILE != "") begin
            $readmemh(INIT_FILE, mem);
        end
        wait (done) begin
            $writememh(OUT_FILE, mem);
        end
    end

    always @(posedge BRAM_CLK or posedge BRAM_RST) begin
        if (BRAM_RST) begin
            BRAM_RDDATA <= 0;
        end
        if (BRAM_EN) begin
            if (BRAM_WE) begin
                if (BRAM_WE[0]) mem[addr][7:0] <= BRAM_WRDATA[7:0];
                if (BRAM_WE[1]) mem[addr][15:8] <= BRAM_WRDATA[15:8];
                if (BRAM_WE[2]) mem[addr][23:16] <= BRAM_WRDATA[23:16];
                if (BRAM_WE[3]) mem[addr][31:24] <= BRAM_WRDATA[31:24];
            end
            else begin
                dout <= mem[addr];
                BRAM_RDDATA <= dout;
            end
        end
    end
endmodule
```

1.2 Processing Element, PE

Processing Element (이하 PE)는 구현의 최종목표인 CNN에서 벡터의 내적연산을 계산하는 역할을 한다. PE의 구현방식은 다음과 같다.

1. 외부 컨트롤러 (현재는 testbench로 대체)에서 들어온 input값들을 받는다.
2. ain, bin, cin로 floating point fused multiply-add(이하 MAC)를 수행한다.
3. (2)에서 나온 결과값을 psum에 더해준다.
4. (1),(2),(3)을 모든 입력값에 대해 반복해서 수행한다.

위의 구현을 수식으로 나타내면 다음과 같다.

$$\texttt{psum} = \texttt{ain}_i \times \texttt{bin}_i + \texttt{psum} \quad (\texttt{i} : \texttt{input number})$$

Practice #5. PE implementation & BRAM modeling

Jiwon Lee, Sangjun Son

MAC모듈의 parameter중 dvalid bit는 연산이 끝난 후, m_axis_result_tdata가 실제 연산의 결과값임을 보장해준다. 따라서, always구문을 사용하여 결과값을 다시 psum register에 저장해 주었다. 코드를 보면, s_axis_c_tdata과 m_axis_result_tdata가 다름을 확인 할 수 있는데, 이는 MAC에서 parallel 하게 cin과 result값을 parameter로 줄 수 없기 때문이다. 따라서, 임시 res register를 이용하여 결과값을 받았고, 이후 그 값을 psum에 할당해주었다.

MY_PE

```
`timescale 1ns / 1ps

module my_pe #(
    parameter L_RAM_SIZE = 6,
    parameter BITWIDTH = 32
)
(
    input aclk,
    input aresetn,
    input [BITWIDTH-1:0] ain,
    input [BITWIDTH-1:0] bin,
    input valid,
    output dvalid,
    output [BITWIDTH-1:0] dout
);

    // local reg ( can make overflow )
    reg [BITWIDTH-1:0] psum = 0;
    wire [BITWIDTH-1:0] res;

    floating_point_MAC UUT (
        .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(psum),
        .m_axis_result_tvalid(dvalid),
        .m_axis_result_tdata(res)
    );

    always @(dvalid) begin
        if(dvalid == 1) begin
            psum = res;
        end
    end

    assign dout = dvalid == 1 ? psum : 0;
endmodule
```

2 Result

2.1 Block Random Access Memory, BRAM

아래의 코드는 BRAM의 구현의 Validity를 확인하기 위해 시나리오에 맞게 구현한 것이다. BRAM 인스턴스 두 개를 만들고 각각을 MY_BRAM1과 MY_BRAM2로 명명하였다.

MY_BRAM1은 `input.txt`에 저장된 `mem`에 있는 값들을 호출하여 저장하는 역할을 하고 또한 `mem`에 있는 값들을 MY_ADDR을 변화하면서 BRAM_RDDATA1로 읽어온다. MY_BRAM2의 경우 이렇게 읽어온 BRAM_RDDATA1을 BRAM_WRDATA2로 사용하여 `mem`에 저장하게 되고 완료가 되면 `done` 신호를 주어 `output.txt`에 저장하게 된다. 아래 Figure 1은 상기된 설명을 도식화한 것이다.

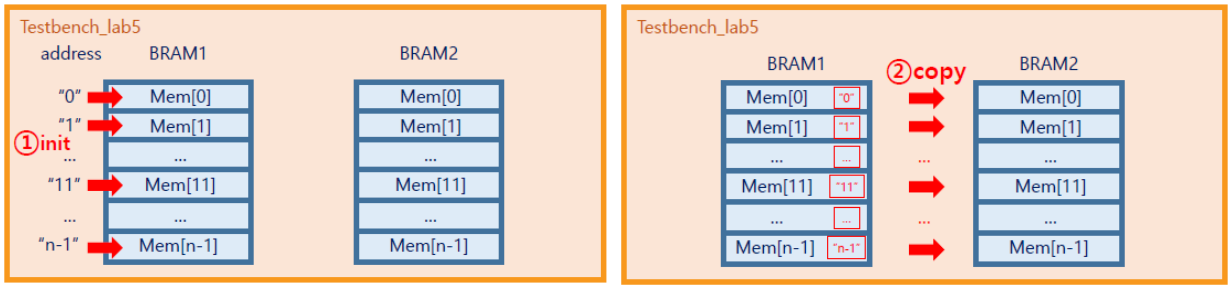

Figure 1: Testbench Scenarios: 2개의 BRAM을 인스턴스화 하고 BRAM1에서 `input.txt`을 읽어 메모리에 저장하고 다른 BRAM2에서 BRAM1의 데이터를 전달받아 `output.txt`에 저장한다 [1].

- 모든 테스트를 시작하기 전에 `input.txt`를 초기화하기 위한 과정을 거친다. 주소에 해당하는 인덱스를 값으로 가질 수 있도록 for문을 통해 대입시킨 후 `$writememh`를 통해 파일에 저장을 한다 (23-24 라인).
- 테스트벤치에서 BRAM_EN 신호는 항상 활성화 하고 (45, 55 라인) BRAM_RST와 `done` 신호는 모든 데이터 전송이 끝나고 입력과 출력이 완료되었을 때 True를 대입할 것이다 (33-34 라인).
- BRAM_ADDR의 경우 i번째 entry의 주소값은 BRAM에서 2개의 LSB를 사용하지 않으므로 주소값 또한 4의 배수로 증가시켜 대입해 주어야 한다. BRAM_WE 신호를 주소값이 유지되는 한 구간을 5 CLK 싸이클과 1 CLK 싸이클로 나누어 DISABLE과 ENABLE을 번갈아 대입해준다 (29-31 라인).

TB_MY_BRAM

```
1   `timescale 1ns / 1ps
2
3   module tb_my_bram #(
4       parameter integer BRAM_ADDR_WIDTH = 15,
5       parameter INIT_FILE = "input.txt"
6   )();
7       reg [31:0] BRAM_INIT[0:8191];
8       reg [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR;
9       reg BRAM_CLK;
10      reg BRAM_RST;
11      reg [3:0] BRAM_WE;
12      reg done;
13      wire [31:0] BRAM_WRDATA1, BRAM_RDDATA1;
14      wire [31:0] BRAM_WRDATA2, BRAM_RDDATA2;
15      integer i;
16
17      initial begin
```

```
18          BRAM_ADDR <= 0;
19          BRAM_CLK <= 1;
20          BRAM_RST <= 0;
21          BRAM_WE <= 0;
22          done <= 0;
23          for (i = 0; i < 8192; i = i + 1) begin
24              BRAM_INIT[i][31:0] <= i;
25          end
26          #10 $writememh(INIT_FILE, BRAM_INIT);
27
28          for (i = 0; i <= 8192; i = i + 1) begin
29              BRAM_ADDR <= i << 2; #20;
30              BRAM_WE <= 4'b1111; #10;
31              BRAM_WE <= 0; #30;
32          end
33          done <= 1'b1; #30;
34          BRAM_RST <= 1'b1;
35      end
36
37      always #5 BRAM_CLK = ~BRAM_CLK;
38      assign BRAM_WRDATA2 = BRAM_RDDATA1;
39
40      my_bram MY_BRAM1 (
41          .BRAM_ADDR(BRAM_ADDR),
42          .BRAM_CLK(BRAM_CLK),
43          .BRAM_WRDATA(BRAM_WRDATA1),
44          .BRAM_RDDATA(BRAM_RDDATA1),
45          .BRAM_EN(1'b1),
46          .BRAM_RST(BRAM_RST),
47          .BRAM_WE(0),
48          .done(0)
49      );
50      my_bram #(.INIT_FILE("")) MY_BRAM2 (
51          .BRAM_ADDR(BRAM_ADDR),
52          .BRAM_CLK(BRAM_CLK),
53          .BRAM_WRDATA(BRAM_WRDATA2),
54          .BRAM_RDDATA(BRAM_RDDATA2),
55          .BRAM_EN(1'b1),
56          .BRAM_RST(BRAM_RST),
57          .BRAM_WE(BRAM_WE),
58          .done(done)
59      );
60  endmodule
```

위 Testbench 코드를 수행하면 아래의 Figure 2와 같은 Waveform을 확인할 수 있다. 결과를 자세히 보면 `mem`에 해당하는 BRAM_ADDR이 4씩 증가하는 것을 확인할 수 있고 이에 따라 BRAM_RDDATA1 또한 BRAM_WE가 비활성화 되어 있을 때 2 cycle을 delay로 읽게 되는 것을 확인할 수 있다. BRAM_RDDATA1가 곧 BRAM2의 BRAM_WRDATA2이므로 같은 Waveform을 관찰하였다.

`mem`에 write가 될 때 1 cycle delay가 되는 것을 눈으로 확인할 수는 없지만 BRAM_WE가 ENABLE 되었다가 DISABLE 되었을 때 read하는 데이터 BRAM_RDDATA2의 delay가 총 3 cycle가 걸렸다는 것을 확인할 수 있었다. 이 사실로 미뤄 보아 읽기에 걸리는 시간이 2 CLK 싸이클, 쓰기에 걸리는 시간이 1 CLK 싸이클이 걸린다는 것을 유추할 수 있었고 스펙에 맞는 올바른 구현이 되었다는 것을 짐작할 수 있었다.

Figure 4은 생성된 입력과 출력파일로 각각 `input.txt`는 MY_BRAM1이 입력을 받기 위한 파일, `output.txt`는 MY_BRAM2이 MY_BRAM1으로부터 데이터를 전달받아 출력을 하기 위해 생성된 파일이다. 모든 데이터가 입력을 받은 대로 정상적으로 출력이 되었음을 확인하였다.

Practice #5. PE implementation & BRAM modeling
Jiwon Lee, Sangjun Son

Figure 2: `TB_MY_BRAM` Waveform

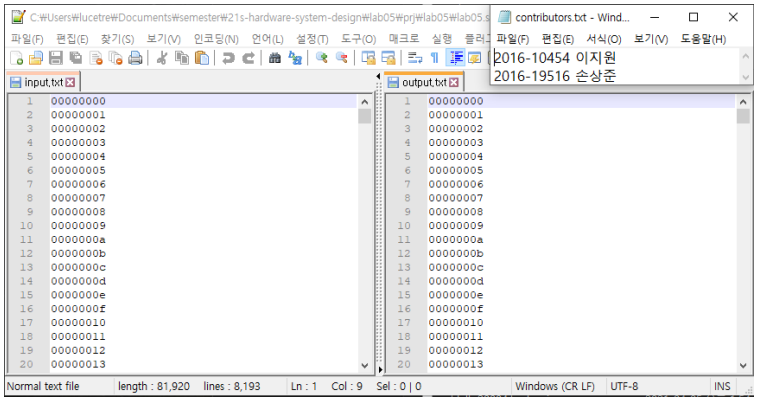

Figure 3: `TB_MY_BRAM` I/O files. `input.txt`와 `output.txt`의 8192개의 `mem` 값이 서로 같다는 결과를 토대로 데이터 전달 및 입력/출력이 잘 이뤄진 것을 짐작할 수 있다.

2.2 Processing Element, PE

PE의 컨트롤러 역할을 하는 testbench는 valid bit를 조절하여, MAC연산이 끝난 타이밍에 새로운 input(ain, bin)을 PE에 제공해야 한다. 따라서, MAC연산이 수행됐음을 알리는 dvalid bit가 high가 되면, negedge이후에 valid bit를 high로 설정하여 다음 연산을 수행하도록 하였다.

또한, IP catalog document에서 나와 있듯이, aresetn을 low로 2 clock cycle동안 할당하여, 초기 값을 초기화해주었고, wait구문을 이용하여 dvalid가 high에서 low로 바뀌는 타이밍에 valid bit를 high로 설정해주었다.

하지만, dvalid bit가 negedge이후에 바로 valid bit를 high로 설정하였을 때는 이전 input값이 MAC연산의 input이 되는 것을 확인, 1 clock cycle delay를 주어 올바른 input이 연산의 input으로 사용되도록 조절하였다.

위 Testbench 코드를 수행하면 아래의 Figure 4와 같은 Waveform을 확인할 수 있다. Waveform의 ain, bin은 미리 random initialize된 2개의 global buffer에서 가져온 값들이다. 결과를 확인해 보면, 총 16개의 ain, bin vector에 대해 dout에 올바른 결과값이 누적되며 출력되는 것을 확인 할 수 있다. 아래의 Waveform은 결과값의 확인을 편하게 하기 위함이고, 코드에서 주석처리된 floating point에서 나올 수 있는 모든 값들의 균일한 입력에 대해서도 올바른 출력값이 출력됨을 확인하였다.

TB_MY_PE

```
1  `timescale 1ns / 1ps
2
3  module tb_my_pe #(
4      parameter L_RAM_SIZE = 6,
5      parameter BITWIDTH = 32
6  );
7
8      reg [BITWIDTH-1:0] gb1 [0:2**L_RAM_SIZE-1];
9      reg [BITWIDTH-1:0] gb2 [0:2**L_RAM_SIZE-1];
10     reg [BITWIDTH-1:0] ain;
11     reg [BITWIDTH-1:0] bin;
12     reg rst;
13     reg clk;
14     wire [BITWIDTH-1:0] dout;
15     reg valid;
16     wire dvalid;
17
18     integer i;
19     initial begin
20         clk <= 0;
21         rst <= 1;
22         #20;
23         rst <= 0;
24
25         // value setting in global buffers
26         for(i = 0; i < 16; i = i+1) begin
27             gb1[i] = $urandom_range(2**30,2**30+2**26);
28             gb2[i] = $urandom_range(2**30,2**30+2**26);
29 //            gb1[i] <= ($urandom%2 << 31) + ($urandom%(2**8) << 23) + $urandom%(2**23);
30 //            gb2[i] <= ($urandom%2 << 31) + ($urandom%(2**8) << 23) + $urandom%(2**23);
31         end
32
33         // execute PE
34         for(i = 0; i < 16; i = i+1) begin
35             if(i != 0) begin
36                 wait (dvalid == 1);
37                 wait (dvalid == 0);
38             end
39             ain = gb1[i];
40             bin = gb2[i];
41             #15;
42             valid <= 1;
43             #10;
44             valid <= 0;
45         end
46     end
47
48     always #5 clk = ~clk;
49
50     my_pe MY_PE (
51         .aclk(clk),
52         .aresetn(~rst),
53         .ain(ain),
54         .bin(bin),
55         .valid(valid),
```

```
56          .dvalid(dvalid),
57          .dout(dout)
58      );
59
60  endmodule
```

위 Testbench 코드를 수행하면 아래의 Figure 4와 같은 Waveform을 확인할 수 있다.

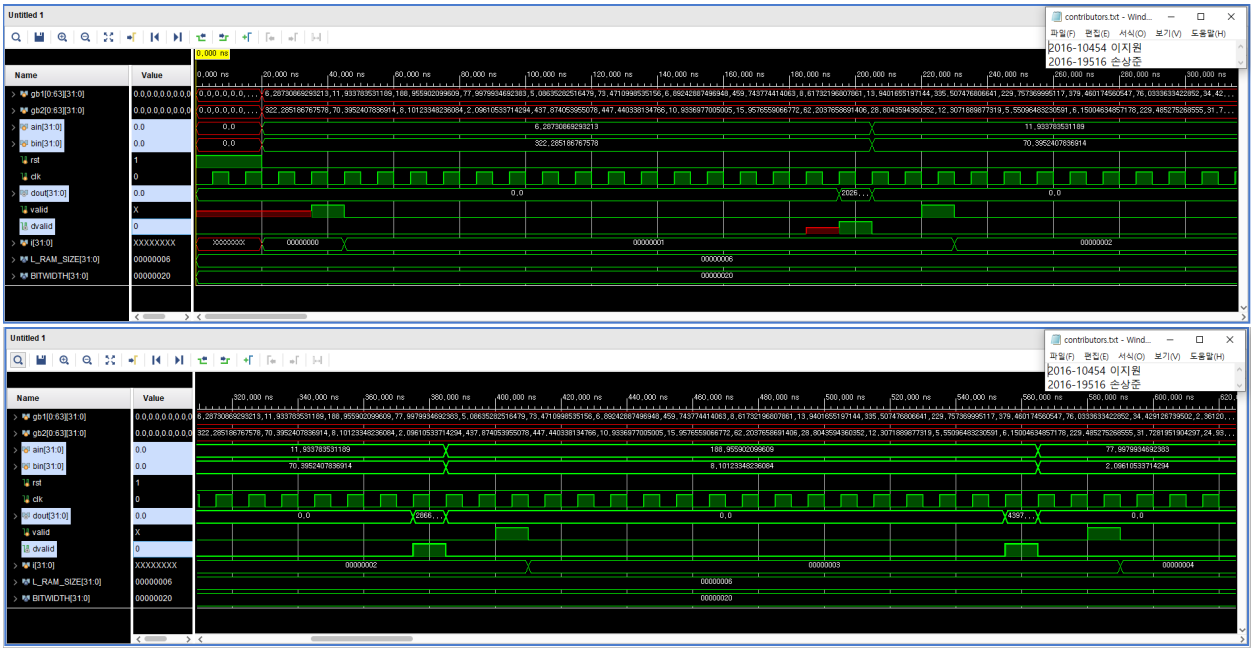

Figure 4: `TB_MY_PE` Waveform

3 Conclusion

이후 프로젝트에서 어떤 모듈을 구현해야 하는지 Bottom-up으로 구현하다 보니 무슨 기능을 위한 구현인지는 아직 잘 모르겠지만 반대로 이전 lab 세션에서 구현을 진행한 모듈에 대해서는 연계성을 확인할 수 있었다. 지금 구현한 모듈이 앞으로도 쓰일 수 있기 때문에 가독성을 높이면서 최대한 임의 구현 방식을 최대한 피하기 위해 노력하였다.

MY_BRAM 모듈을 구현하면서 WE signal에 따라 `mem`에 저장하는 statement를 for-generate로 구현해보려 했으나 이런 저런 오류가 나면서 나열형 방식으로 구현해 코드의 효율성이 떨어진다는 나름의 판단을 하였다. 추후 프로젝트를 진행하기 전에 always 구문 안에서 block assignment를 for-generate로 구현하는 방식을 익혀야 겠다는 필요성을 제고하였다 [2].

MY_PE 모듈 자체의 구현은 그리 어렵지 않았지만, IP catalog의 floating point MAC모듈의 내부 구조를 모르기 때문에, testbench를 작성하는데 어려움을 겪었는데, 이는 valid bit의 high 설정 후 dvalid bit가 high가 될 때까지의 정확한 clock cycle delay, valid bit와 input parameter가 동시에 할당되었을 때의 비정상적인 출력이 그리 직관적이지는 않았기 때문이라고 생각한다. MY_PE 모듈도 이후 구현에서 중요한 부분을 차지하기 때문에, 최대한 이번 과제에만 국한되지 않게 구현하려 하였다.

References

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

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

[3] WillFlux. *Initialize Memory in Verilog*. Project F - FPGA Development, April 2020.