Lab 3

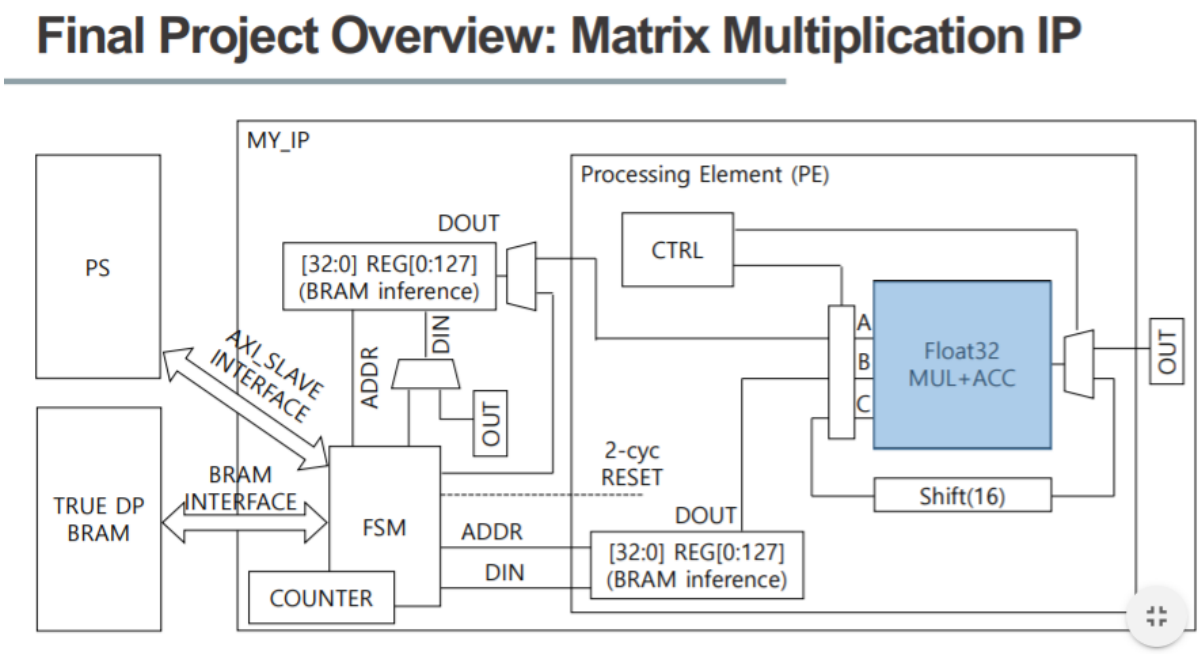
(How to use Vivado & Basic syntax)

2017-19428 서준원

0. Overview

이번 실습은 Vivado를 설치하고 옛날 논리설계 수강 때 배우고 까먹었던 Verilog 사용법을 Remind하는 실습을 진행하였다. HDL인 Verilog를 통해 가장 간단한 코드인 Adder, Multiplier, Fused Multiplier를 구현하면서 Verilog를 익혔다. 이를 통해 앞으로 수업 과정에서 다양한 프로젝트를 진행할 수 있을 것이다.

1. Introduction



위 사진은 이번 학기 프로젝트의 전체 설계이다. 지난 주 랩에서는 인터페이스를 제작했다고 한다면, 이번 주부터는 Verilog를 사용하여 실제 하드웨어를 설계하는 작업을 시작한다. 가장 간단한 연산인 Add, Multiplier, 그리고 Matrix Multiplication에 직접적으로 사용되는 Fused Multiplier를 만들었다.

2. Result & Discussion

```
module
my_add
#(
        parameter BITWIDTH = 32
    )
    (
        input [BITWIDTH-1:0] ain,
        input [BITWIDTH-1:0] bin,
        output [BITWIDTH-1:0] dout,
        output overflow
    );
    /* IMPLEMENT HERE! */
        assign {overflow, dout} = ain + bin;
    endmodule
```

Adder는 간단하게 + 연산자로 구현이 가능하다. 실제 Adder를 구현하는 회로는 더 복잡하지만, 다양한 게이트를 사용하는 방법을 명시하지 않아도 회로를 설계할 수 있다. Overflow와 out을 Concanetation하여 overflow를 처리하였다.

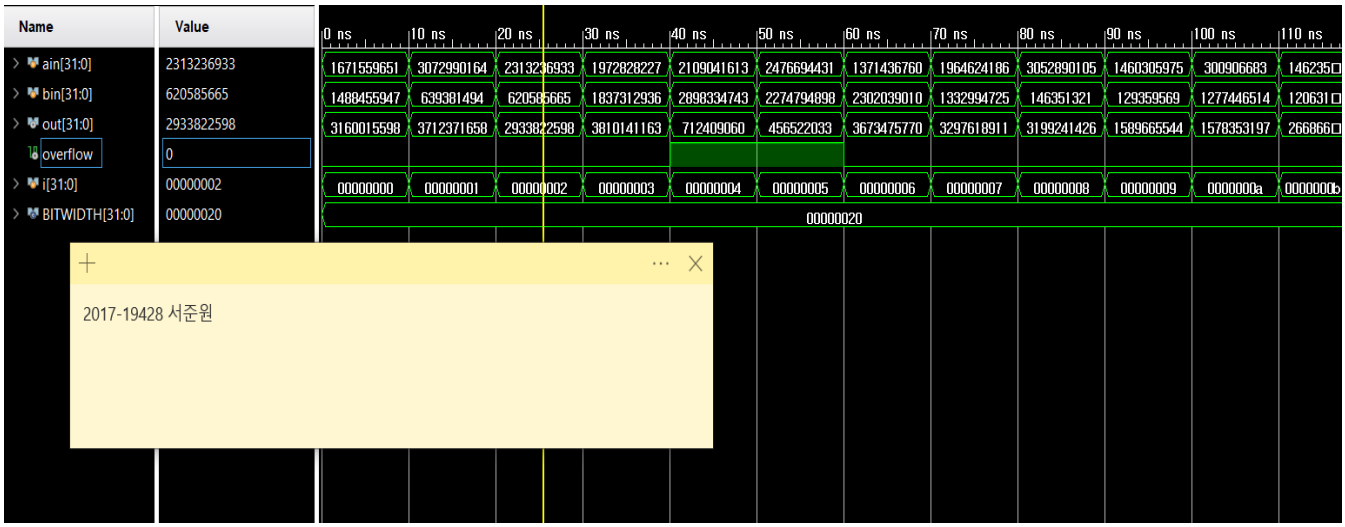


Figure 1 add

```
initial
begin
                for(i=0; i<32; i=i+1) begin
                    ain = $urandom%(2**32-1);
                    bin = $urandom%(2**32-1);
                    #10;
                end
            end
```

제공된 테스트 코드와는 조금 다르게 overflow를 실험하기 위해 랜덤한 인풋값의 범위를 늘렸다. 총 32가지 경우를 테스트했는데 한 가지 경우만 살펴보면 2313236933 + 620585665 = 2933822598로 정확한 값을 얻었다. 또한 4,5 번째 사이클에서는 overflow가 정확하게 계산된 것을 확인할 수 있다.

```
module
my_mul
#(
        parameter BITWIDTH = 32
    )
    (
        input [BITWIDTH-1:0] ain,
        input [BITWIDTH-1:0] bin,
        output [2*BITWIDTH-1:0] dout
    );
    /* IMPLEMENT HERE! */
        assign dout = ain * bin;
    endmodule
```

Multiplier또한 * 연산자를 사용하여 간단하게 구현할 수 있었다. 이를 테스트벤치를 통해 시뮬레이션하였다. 시뮬레이션 코드는 위의 Adder와 동일하다 (output의 자리수만 다르다). 10진법으로 결과를 출력해본 결과 계산이 정확하게 되는 것을 확인할 수 있다. Bandwidth는 상수로 32자리 (4Byte)의 연산을 하도록 설정했다.

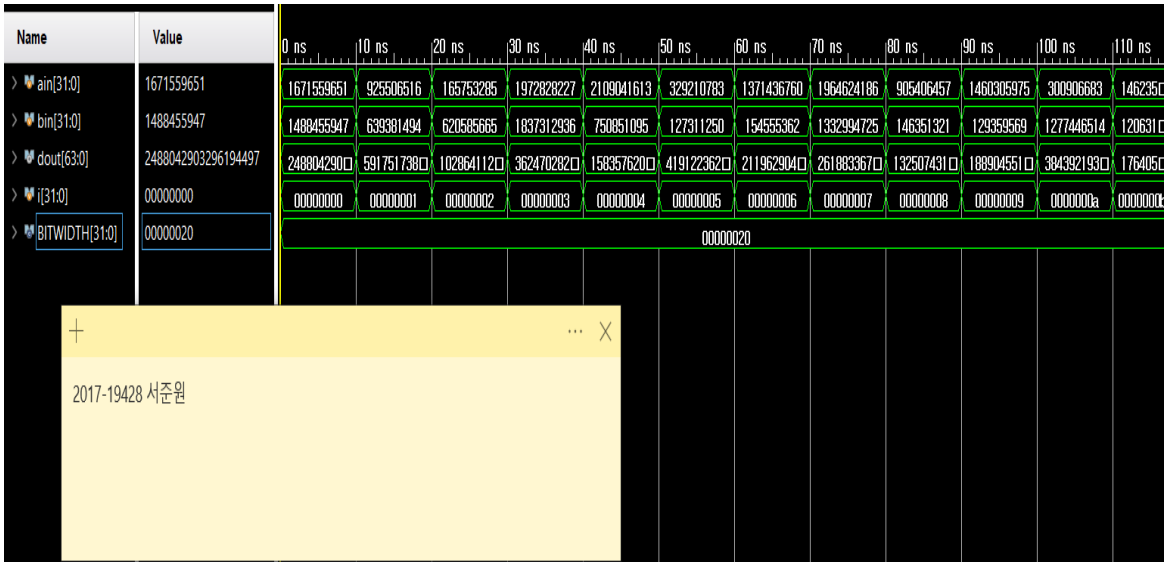


Figure 2 mul

```
module my_fusedmult #(
                            parameter BITWIDTH = 32
                        )
                        (
                            input [BITWIDTH-1:0] ain,
                            input [BITWIDTH-1:0] bin,
                            input en,
                            input clk,
                            output [2*BITWIDTH-1:0] dout
                        );
/* IMPLEMENT
HERE! */
                       reg [2*BITWIDTH-1:0] tmp;
                        always @(posedge clk) begin
                           case(en)
                               1'b0: tmp <= 32'b0;
                               1'b1: tmp <= tmp + ain * bin;
                           endcase
                       end
                       assign dout = tmp;
                   Endmodule
```

마지막으로 MV Multiplication에 가장 중요한 Fused Multiplier를 구현하였다. 이는 Enable Signal이 0이 되면 결과를 초기화하고, 나머지 경우에는 모두 지금까지의 결과값에 Multiplication 결과를 Accumulation한다.

지금까지의 다른 연산과는 다르게 이전의 State를 저장해야하기 때문에 Edge Triggered 연산을 구현하였다. always문을 사용하여 clock의 postedge에만 작동하게 하여 sequential한 회로를 구현하였다. Always문 안에는 en의 값에 따라 Accumulation을 할 지, Initialization을 할 지 결정하였다.

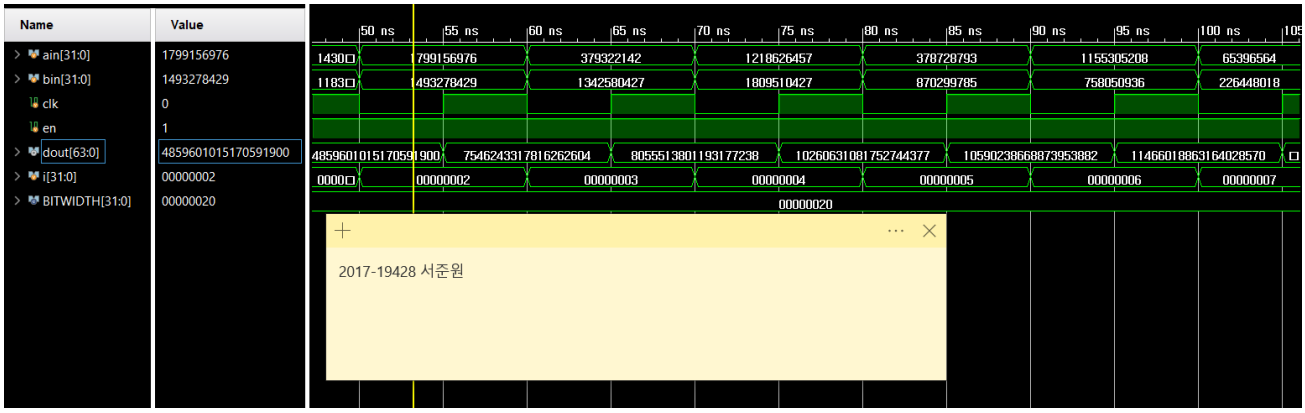


Figure 3 Fused Multiplier

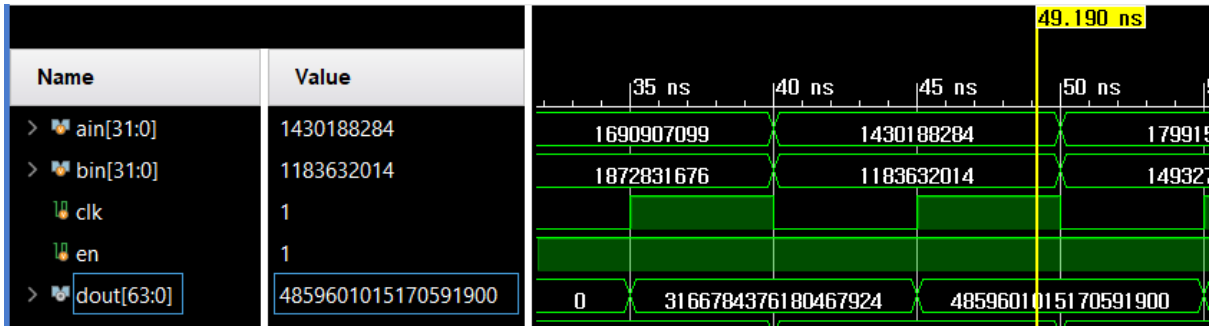


Figure 4 Fused Multiplier_2

```
module
tb_fmul();
                parameter BITWIDTH = 32;

                reg [BITWIDTH-1:0] ain;
                reg [BITWIDTH-1:0] bin;
                reg clk;
                reg en;
                wire [2*BITWIDTH-1:0] dout;

                integer i;

                always #5 clk = ~clk;

                initial begin
                    clk <= 0;
                    en <= 0;
                    #30;
                    en <= 1;
                    for(i = 0; i < 32; i = i + 1) begin
                        ain = $urandom%(2**31);
                        bin = $urandom%(2**31);
                        #10;
                    end
                end

                my_fusedmult #(BITWIDTH) MY_FMUL(
                    .ain(ain),
                    .bin(bin),
                    .clk(clk),
                    .en(en),
                    .dout(dout)
                );

            endmodule
```

위의 테스트벤치 코드를 사용하여 Fused Multiplier를 테스트한 결과이다. 정확한 결과 확인을 위해 특정 부분을 확대해보았다. 처음 1690907099와 1872831676을 곱한 결과로 3166784376180467924가 계산되었다. 이후 다음 사이클에 1430188284와 1183632014가 곱해졌고 기존 3166784376180467924에 더해져 48596010151705919000이 계산되었다.

또한 dout은 clk의 rising edge에 바뀌는 것을 확인할 수 있었다.