Hardware System Design, Lab 04

2013-11392 김지현

구현체 설명

32-bit Integer Fused Multiply-Adder

IP Catalog에서 제공하는 “Multiply Adder (3.0)” IP를 사용해 구현하였다. Multiply Adder에서 이미 정수의 fused multiply-add를 지원하고있어, 별도의 처리는 필요하지 않았다. Unsigned 32-bit 정수를 사용하였고, Latency는 0으로 설정하여, 입력이 변하면 결과가 즉시 출력에 반영되도록 했다.

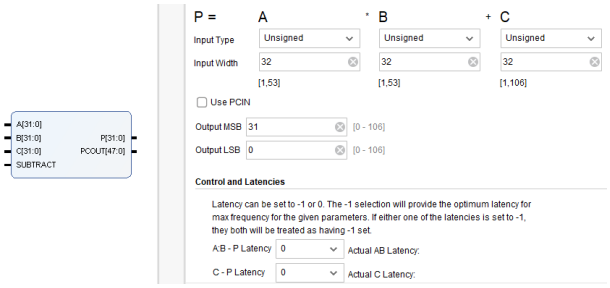

Figure 1. 과제에서 사용한 "Multiply Adder (3.0)" 설정

32-bit Floating Point Fused Multiply-Adder

실습시간에 사용한것과 동일하게 세팅하였다. IP Catalog에서 제공하는 “Floating-point (7.1)” IP의 Fused Multiply-Add 기능을 그대로 사용했다. FMA Operator option으로 덧셈으로 고정할지, 뺄셈으로 고정할지, 덧셈/뺄셈을 선택 가능하게 할것인지 선택하는 option이 있었는데 덧셈으로 고정시켜 pin 개수를 줄였다. 입출력에는 32-bit single precision floating point number를 사용하도록 설정하였고, NonBlocking 모드로 설정시켰다. 32-bit Integer Fused Multiply-Adder와는 다르게, 입력이 주어지면 출력으로 나타나기까지의 레이턴시가 존재했다.

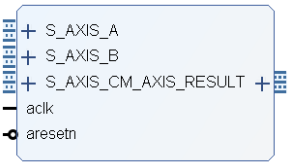

Figure 2. Configured Floating-point (7.1) IP

Adder-Array

Adder-array는 단순하게 구현해도 되지만, 과제에 요구사항이 있어 Lab03에서 구현한 my_add 모듈을 사용해 구현하였다. Packing 문법을 사용해 ain0, ain1 등의 입출력 변수들을 하나의 배열로 묶은 뒤, Generate 문법과 반복문을 사용해, 코드 반복 없이 my_add 모듈을 복수개 instantiate할 수 있었다.

```
module adder_array(
    input [2:0] cmd,
    input [31:0] ain0, ain1, ain2, ain3, bin0, bin1, bin2, bin3,
    output [31:0] dout0, dout1, dout2, dout3,
    output [3:0] overflow
);
    parameter BITWIDTH = 32;
    wire [31:0] ain[3:0], bin[3:0], dout[3:0], unmasked_dout[3:0];
    assign {ain[0], ain[1], ain[2], ain[3]} = {ain0, ain1, ain2, ain3};
    assign {bin[0], bin[1], bin[2], bin[3]} = {bin0, bin1, bin2, bin3};
    assign {dout0, dout1, dout2, dout3} = {dout[0], dout[1], dout[2], dout[3]};

    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin
            my_add #(BITWIDTH) adder(
                .ain(ain[i]),
                .bin(bin[i]),
                .dout(unmasked_dout[i]),
                .overflow(overflow[i])
            );
            assign dout[i] = (cmd == i || cmd == 4) ? unmasked_dout[i] : 0;
        end
    endgenerate
endmodule
```

Figure 3. adder_array 소스코드

실험 결과 분석

32-bit Integer Fused Multiply-Adder, 32-bit Floating Point Fused Multiply-Adder

Figure 4와 같은 방식으로 직접 Testbench를 작성하였고, Waveform을 그려볼 수 있었다. Figure 5의 Integer FMA의 경우, 입력이 변할때에 출력이 즉시 반영되지만, Figure 6의 Floating Point FMA의 경우, 입력이 변하고 15 Clock cycle 뒤에 출력이 반영되는 것을 알 수 있었다. 이는 구현에서 사용한 “Floating-point (7.1)” IP 구현체의 특징인것으로 보인다.

```
module fma_u32_test();
    // Test input
    reg [31:0] a, b, c;
    wire [31:0] d_out;
    wire [47:0] d_carry_out;

    fma_u32 UUT(
        .a(a), .b(b), .c(c), .subtract(1'b0),
        .p(d_out), .pcout(d_carry_out)
    );

    // Define test suites
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            a = $urandom;
            b = $urandom;
            c = $urandom;
            #20;
        end
        $finish;
    end
endmodule
```

Figure 4. Testbench of 32-bit Integer Fused Multiply Adder

Figure 5. Waveform of 32-bit Integer Fused Multiply-Adder Testbench

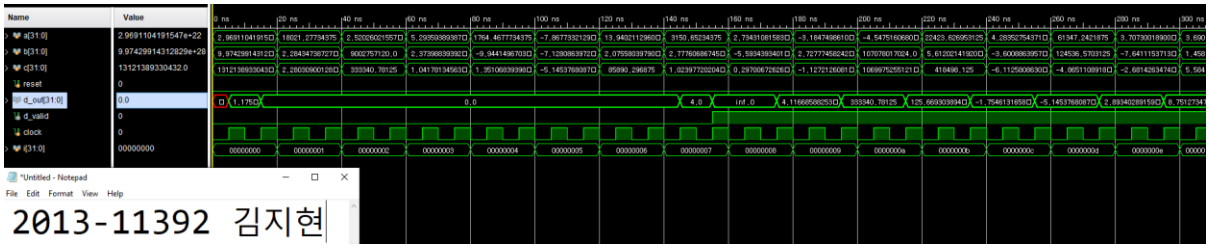

Figure 6. Waveform of 32-bit Floating Point Fused Multiply-Adder Testbench

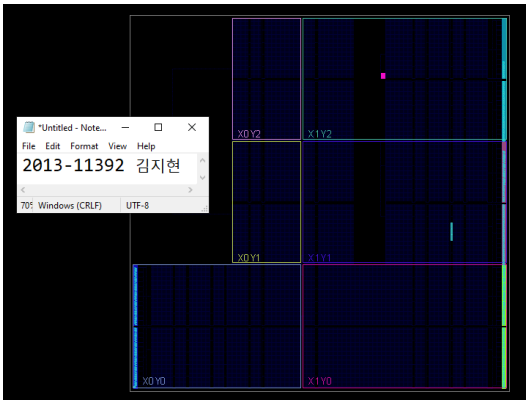

Figure 7. 32-bit Integer Fused Multiply-Adder의 implement 결과

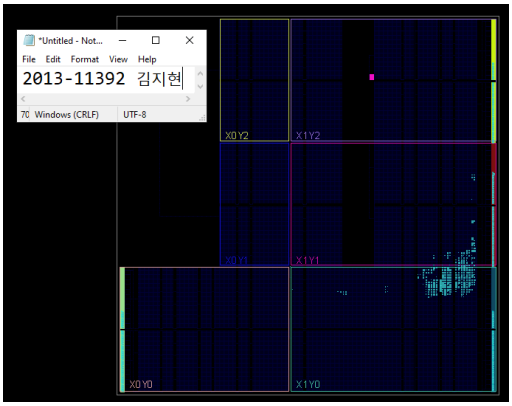

Figure 8. 32-bit Floating Point Fused Multiply-Adder의 implement 결과

두 회로 모두 성공적으로 Synthesize가 가능했고, Implement 결과는 아래와 같았다. 그림을 보면 Integer FMA가 사용하는 칩의 면적이 Floating point FMA보다 훨씬 작음을 알 수 있었는데, 이는 정수의 덧셈/곱셈이 IEEE 754 단정밀도 부동소수점의 덧셈/곱셈보다 훨씬 간단하기때문이다.

Adder-Array

32-bit Integer Fused Multiply-Adder와 유사한 방식으로 랜덤 입력을 만드는 Testbench를 작성하였다. 각 “cmd”마다 4개의 랜덤 입력을 테스트하게 만들었었고, 결과는 아래와 같았다. Cmd가 4일때엔 모든 dout이 활성화되고, cmd가 0, 1, 2, 3 중 하나일때엔 선택된 dout만 정상적으로 활성화되는 것을 확인할 수 있었다. Overflow 또한 정상적으로 동작했다.

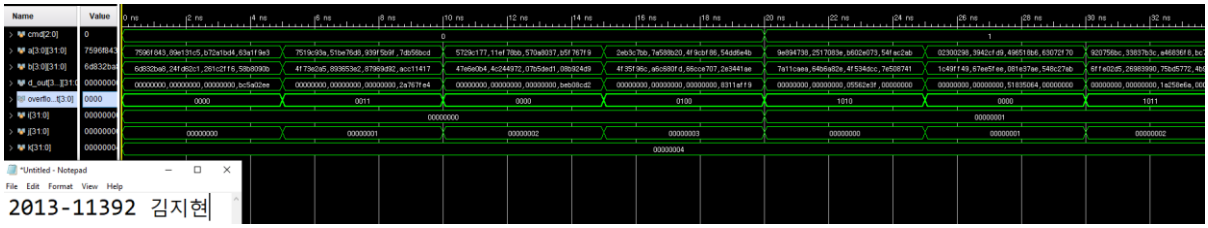

Figure 9. Waveform of Adder-Array Testbench

Adder-Array는 synthesis가 불가능했는데, TA에게 문의한 결과 Adder-Array 문제는 애초에 IO 포트의 수가 391개로, 실험환경이 허락하는 IO 포트 수보다 더 많아 synthesis가 불가능한것이었다. IO 포트의 수를 조절하는 것이 중요하다는 것을 배울 수 있었다.

결론

이번 Lab에선 IP Catalog에서 이미 만들어진 IP를 가져다 설정을 조절해 사용하는법을 배울 수 있었다. 또한 Blocking Mode, NonBlocking Mode의 차이에 대해서도 알 수 있었고, 어떤 IP는 입력의 변화가 출력에 반영되기까지 시간이 필요할 수 있음을 알 수 있었다. generate 문법을 사용해 반복되는 코드를 편리하게 축약하는것을 실습하였고, pack 문법으로 배열이 아닌 변수들을 배열로 묶는법 또한 배울 수 있었다. 작성한 코드를 시뮬레이션만 하는것이 아니라 직접 Synthesize하고 implement하여 가상의 FPGA 위에 프로그래밍 되는 모습을 관찰할 수 있었다. 마지막으로 모듈의 IO 포트 수가 제한되어있기때문에, IO 포트 갯수를 최적화해야한다는 사실 또한 알 수 있었다.

직접 작성한 Verilog 코드와 IP Catalog에서 가져다 쓴 코드들이 가상의 FPGA 위에 입력되기까지의 과정을 익힐 수 있는 유익한 실습이었다.