4190.301A Hardware System Design

Spring 2020

Hardware System Design Lab3 Report

Kim Bochang

2014-16757

1. <lab3> Introduce

Lab 3의 목적은 추후 진행될 프로젝트에서 사용할 xilinx의 사용법을 익히고, verilog의 기본적인 문법을 익혀 자신만의 adder, multiplier와 fused multiplier를 구현하는 것이다.

2. Implementation

2.1 lab03_0319/lab03_0319.srcs/sim_1/new/my_add.v

먼저 구현해야하는 것은 n-bit adder를 구현하는 것이다.

이 n-bit adder는 parameter로 몇비트 짜리인지를 나타내는 bitwidth를 가지고, 다음과 같은 입력과 출력을 가진다.

input

a_in : n bit 입력 1

b_in : n bit 입력 2

output

dout : n bit 출력. a_in + b_in의 결과를 가진다.

overflow : 1 bit 출력. 계산과정중에 오버플로우가 발생하면 1, 아니면 0.

adder의 경우, 다음과 같이 간단하게 구현할 수 있다.

```
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
```

<my_add.v>

assign에서 concatenation을 이용하여 overflow와 dout을 생성하도록 하였다.

오버플로우의 경우 계산 결과가 더했을때 n-bit범위를 넘어가는지만 검출하면 되기 때문에,

즉, n+1bit가 1이 되는 경우만 검출하면 되기 때문에 위와 같이 구현하여도 괜찮다.

2.2 lab03_0319/lab03_0319.srcs/sim_1/new/my_mul.v

그 다음은 n-bit multiplier를 구현하는 것이다.

역시 n-bit adder와 같이 parameter로 몇비트짜리 multiplier인지를 나타내는 bitwidth 를 가지고,

다음과 같은 입력과 출력을 가진다.

input

a_in : n bit 입력 1

b_in : n bit 입력 2

output

dout : 2n bit 출력. a_in * b_in의 결과를 가진다.

multiplier 역시 다음과 같이 간단하게 구현이 가능하다.

```
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
```

<my_mul.v>

2.3 lab03_0319/lab03_0319.srcs/sim_1/new/my_fusedmult.v

이번 lab의 마지막 과제는 n-bit fused_multiplier를 구현하는 것이다.

fused_multiplier는, 앞에서 구현했던 adder나 multiplier와 달리,

입력만으로 결과값이 결정되는 combinational logic이 아니라,

내부에 저장된 state + 입력으로 결과값이 결정되는 sequential logic이기 때문에,

앞의 구현과는 다르게 내부에 register가 필요하다.

fuse multiplier는 다음과 같은 input과 output을 가진다.

input

a_in : n bit 입력 1

b_in : n bit 입력 2

en : 1 bit, 내부 state를 reset할지, 아니면 결과를 계산할지를 결정한다.

clk : clock. synchronize를 위해 clock이 posedge일때만 동작하도록 구현하였다.

output

dout : 2n bit 출력. clock이 posedge일때만 갱신되며, en이 1이면 dout = dout + a_in * b_in의 결과를,

en이 0이면 0으로 리셋된다.

내부 구현은 다음과 같다.

```
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] result;
//need register because of need of saving last value

assign dout = result;
// wireing register to output


initial begin
    result = 0;
end
// initialize register

always @(posedge clk) begin
    if (en) begin
        result = result + ain * bin;
    end
    else begin
        result = 0;
    end
end

endmodule
```

<my_fusedmult.v>

먼저, 내부에 result라는 2nbit register를 두어 그 전의 dout값을 저장할 수 있도록 하고,

assign문을 통해 result와 dout을 연결하였다.

내부에서 register를 사용하기 때문에 초기화가 필수적이므로 initial begin문을 이용하여 result를 초기화 해주었다.

그 후, synchronize되는 부분을 구현하기 위해 always @(posedge clk)문을 사용하여

clk의 posedge일때 en의 값에 따라 새로운 결과를 계산할지, 혹은 결과를 초기화 할지를 결정하게 하였다.

3. Result & Discussion

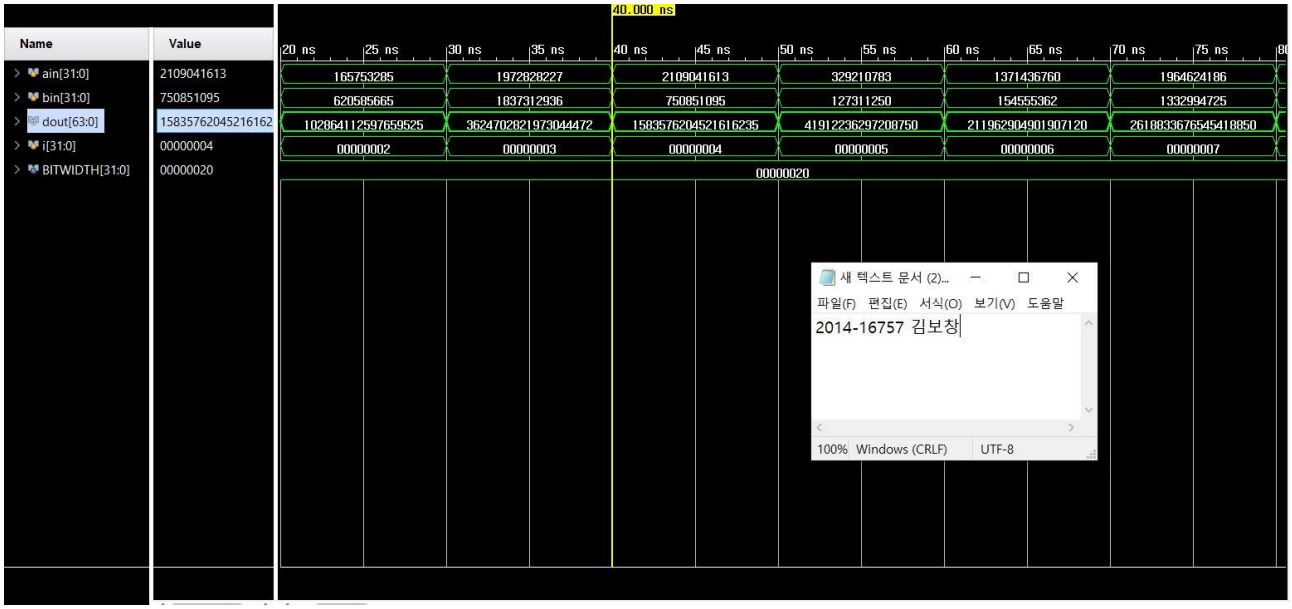
3.1 Adder

<myadder 실행 결과>

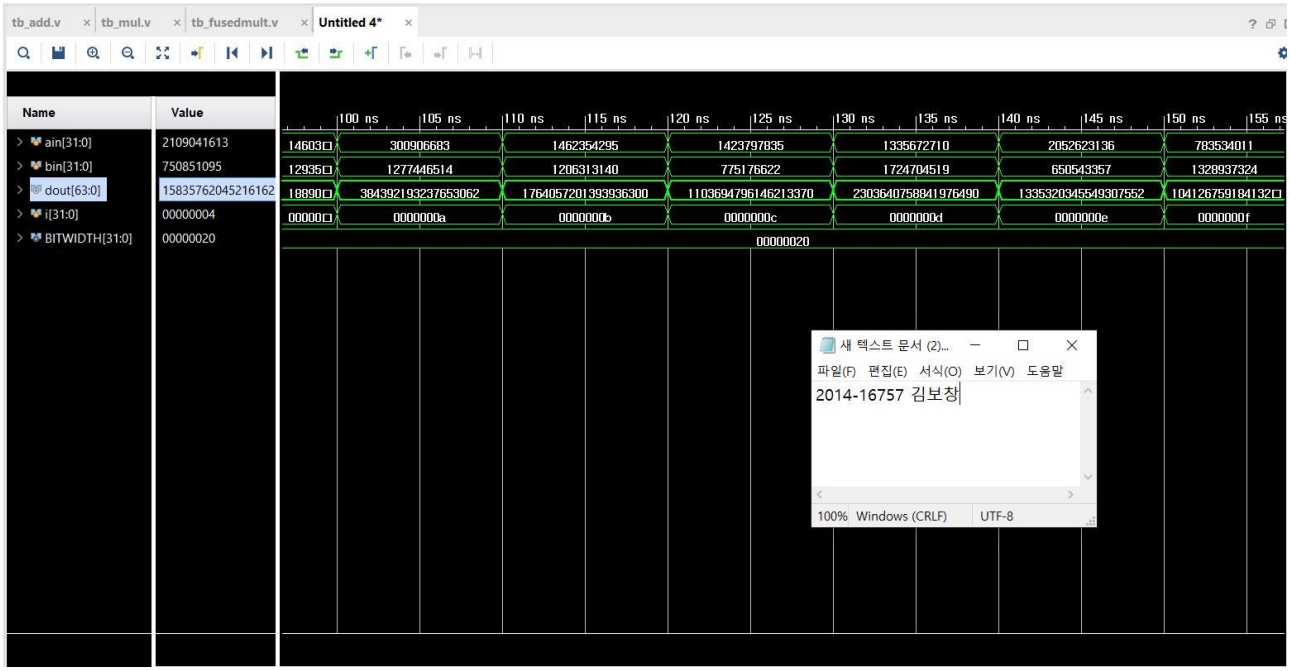
먼저, myadder의 경우 랜덤하게 생성된 32비트 a_in과 b_in의 값을 더한 값이 d_out으로 생성되어 나옴을 확인할 수 있다.

다만, testbench 코드가 32bit 값중 32번째 비트인 MSB의 값이 0인 값들만 생성하도록 되어있어 overflow 비트가 1로 검출되는 경우는 찾아볼 수 없었다.

3.2 Multiplier



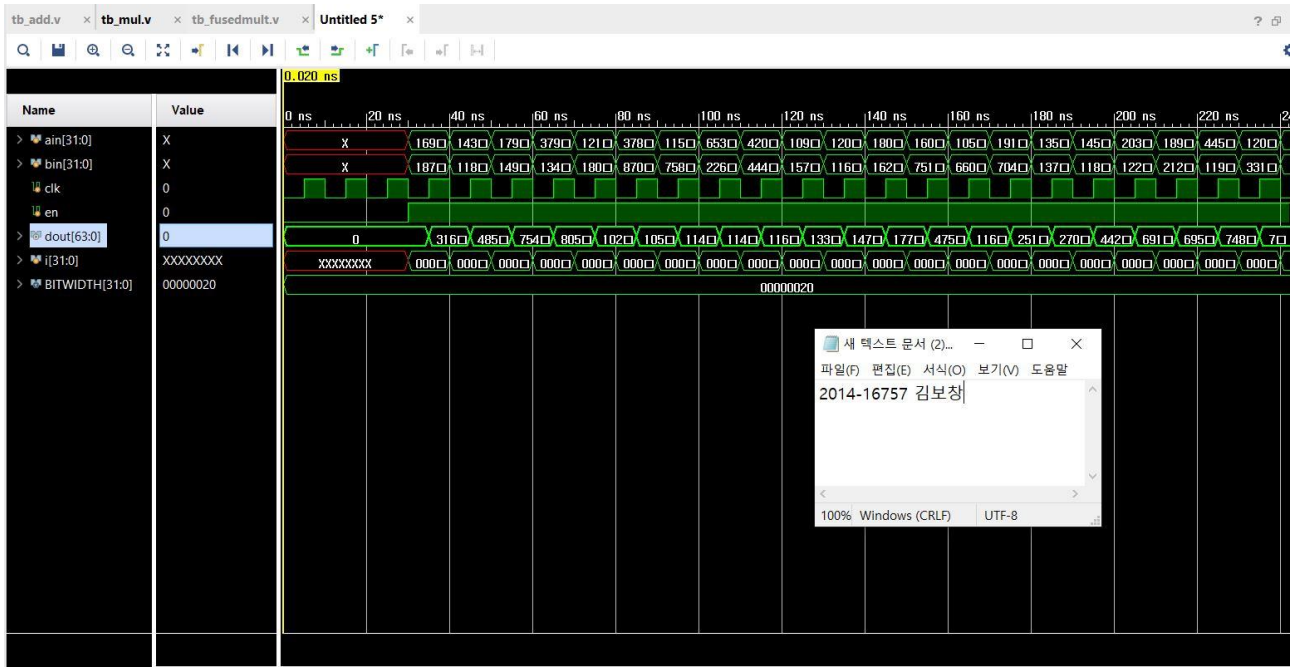
<mymultiplier 실행결과 – 1>



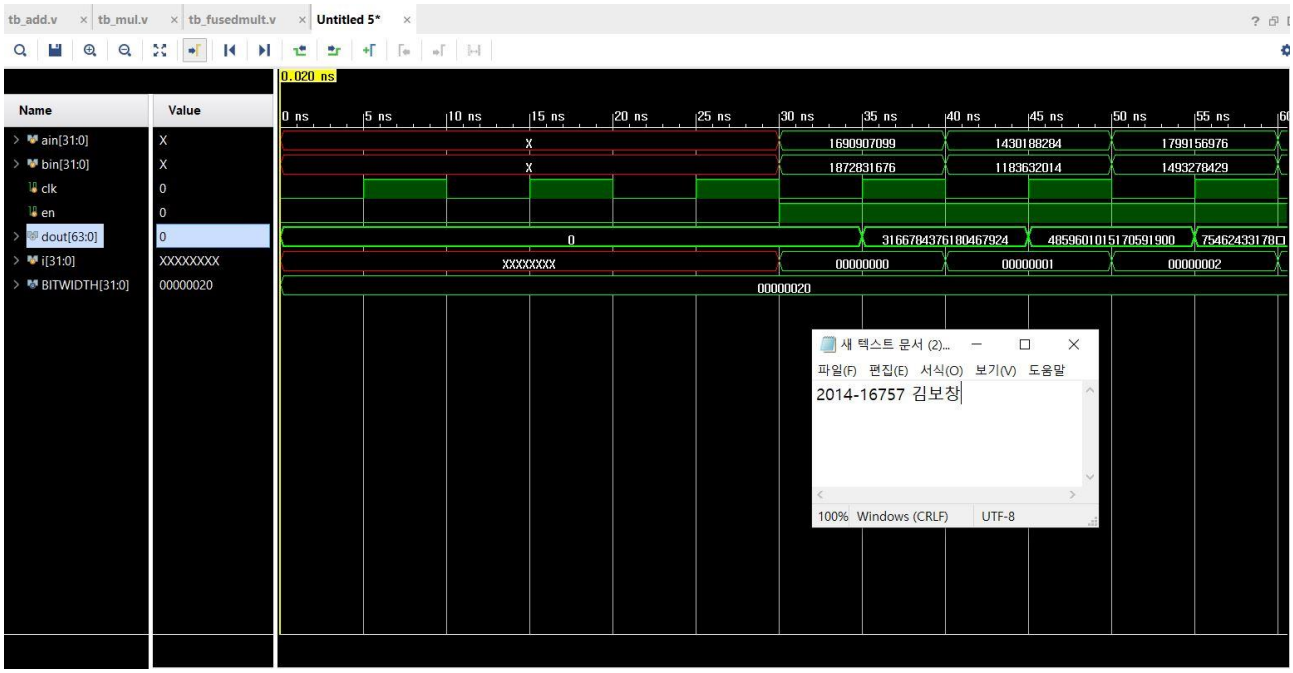
<mymultiplier 실행결과 – 2>

multiplier의 경우도, random하게 생성된 32비트 a_in과 b_in가 곱해진 값이 d_out으로 출력됨을 확인할 수 있었다.

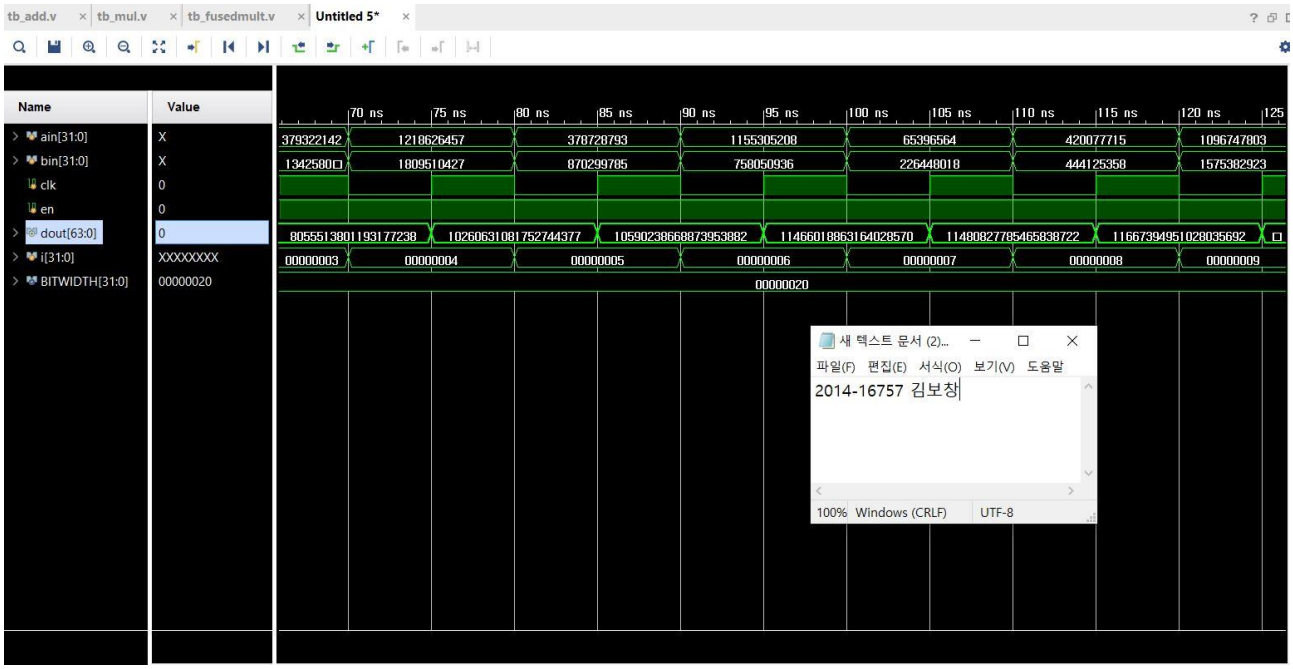
3.3 Fused-multiplier



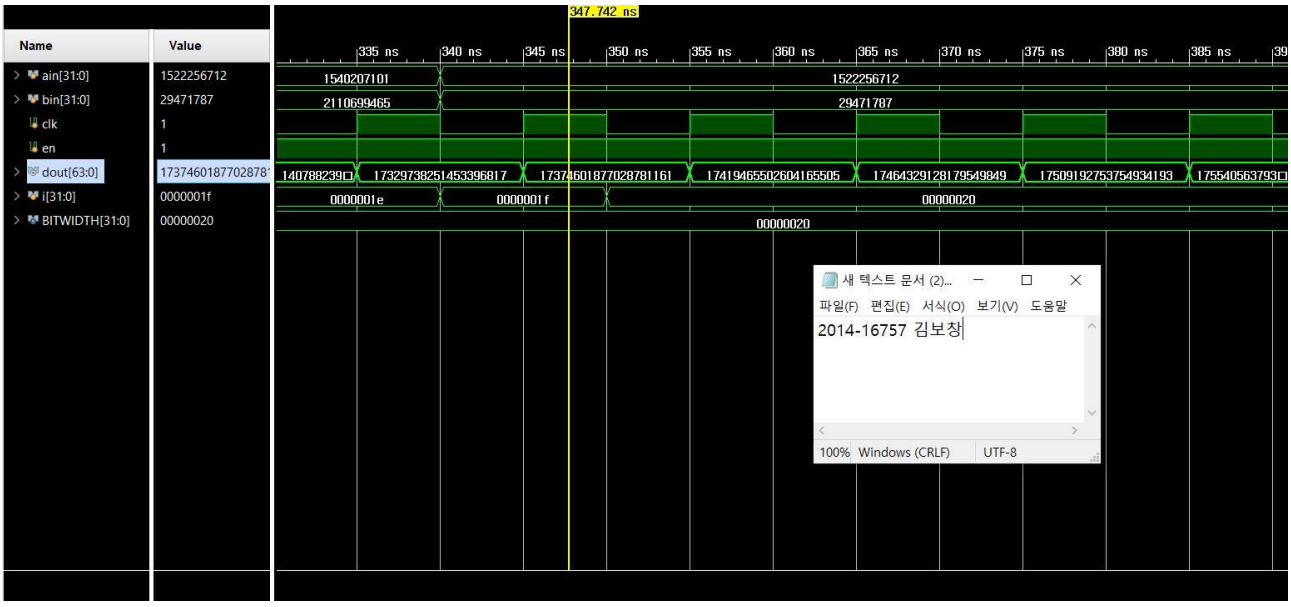
<fused_multiplier 실행결과 – 1>



<fused_multiplier 실행결과 – 2> 1번의 결과를 확대한것이다.



<fused_multiplier 실행결과 – 3> 70ns부분~



<fused_multiplier 실행결과 -4 > 340ns 이상~

결과를 확인하여 보면, en이 1로 활성화 되기 전인 30ns 전까지는, 즉, en이 0일때는

초기화된 결과값인 0이 d_out의 결과로 출력됨을 알 수 있고, en이 1로 활성화 된 30ns부터는

d_out의 값으로 그 전 d_out의 값 + a_in * b_in의 값이 출력됨을 알 수 있다.

또한, a_in과 b_in의 값이 고정된 340ns 뒤에도 계속해서 기존 d_out + a_in * b_in으로 값이 증가하는

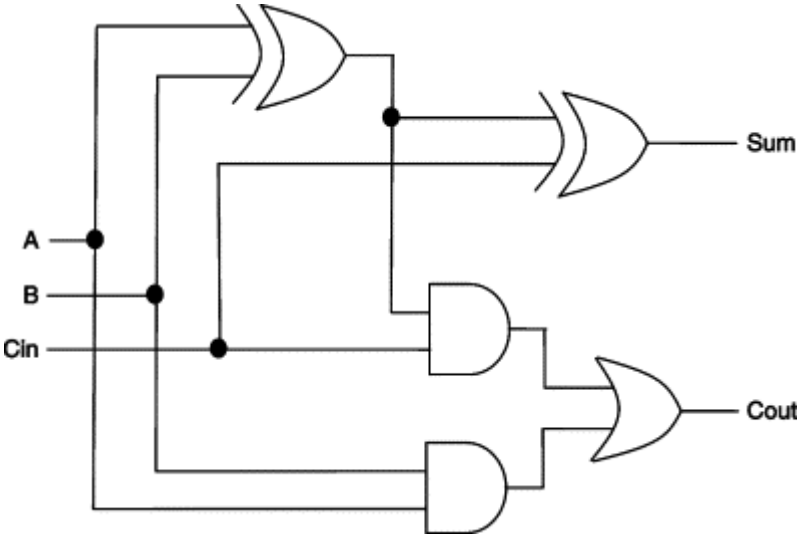
것으로 보아, 구현한 코드가 제대로 작동함을 알 수 있었다.

지금까지 구현한 것은 베릴로그의 내부 구현을 이용한 것이지만,

실제로 n-bit adder와 n-bit multiplier를 구현하기 위해서는 굉장히 많은 논리 게이트가 필요하다.

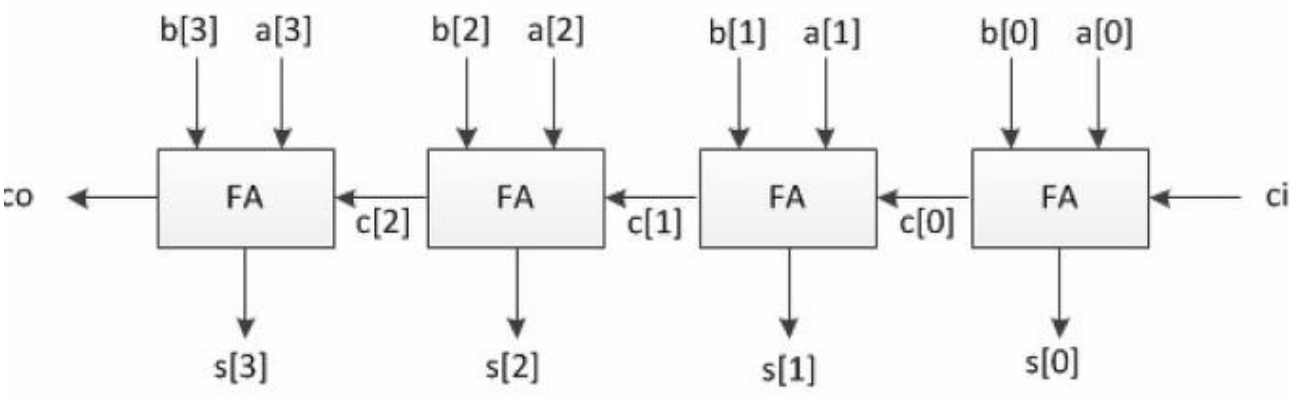
특히, 논리게이트를 사용할 때는 논리게이트마다 delay가 존재하여

전체 논리 회로의 총 delay를 최소화 하기 위해서는 게이트의 조합 순서, 조합 방법이 매우 중요하다.



<full-adder>

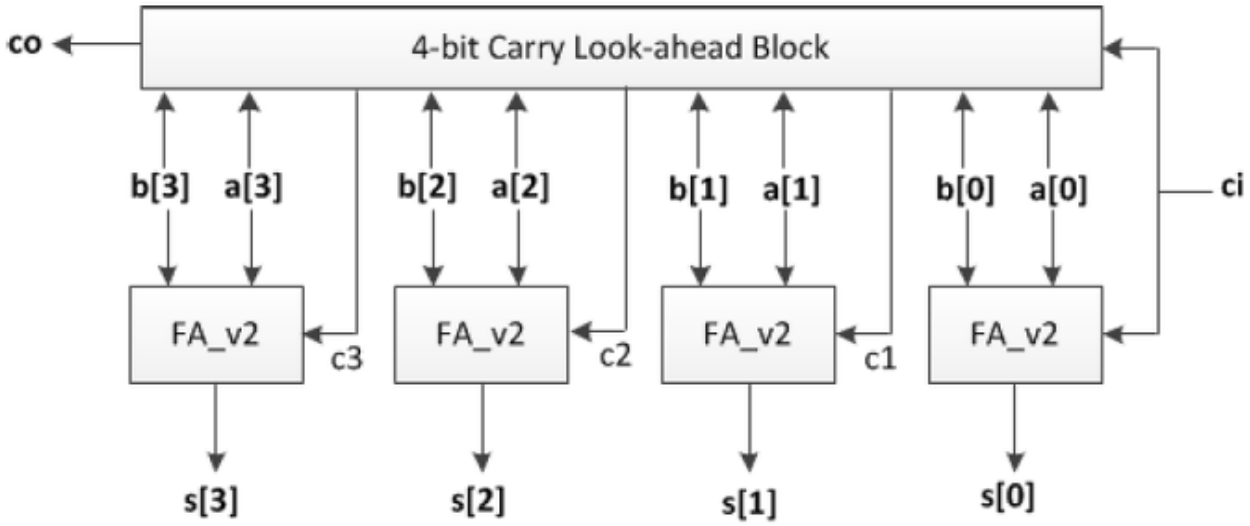
일례로, 4-bit adder의 경우만 생각을 해봐도, 다음과 같이 구현할 수 있는데



<4bit ripple carry adder>

ripple carry adder는 하위 FA의 carry를 상위 FA 계산시 사용하기 때문에, 모든 FA에서 계산이 완료될때까지 기다려야 하므로 delay가 대략 4*FA delay정도로 계산되게 된다.

이를 개선하기 위해 다음과 같이 carry-lookahead adder를 사용한다.



<4bit carry look-ahead adder>

캐리-룩어헤드 애더는 상위 비트의 캐리를 다음 식을 이용해 계산하여준다.

$$G_i = A_i B_i$$
$$P_i = A_i + B_i$$

$$C_{i+1} = A_i B_i + (A_i + B_i) C_i = G_i + P_i C_i$$

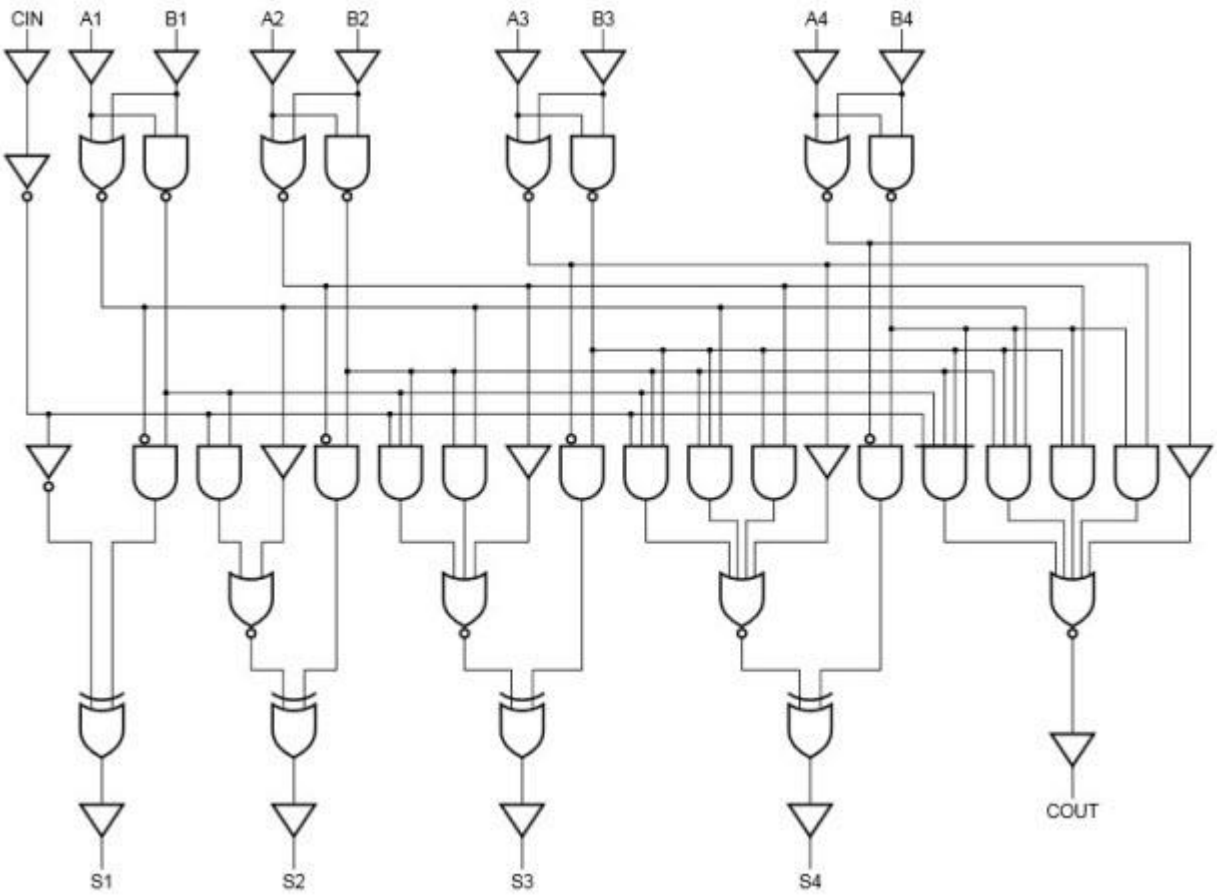
$$C_1 = G_0 + (P_0 * ci)$$

$$C_2 = G_1 + (P_1 * G_0) + (P_1 * P_0 * ci)$$

$$C_3 = G_2 + (P_2 * G_1) + (P_2 * P_1 * G_0) + (P_2 * P_1 * P_0 * ci)$$

$$Co = G_3 + (P_3 * G_2) + (P_3 * P_2 * G_1) + (P_3 * P_2 * P_1 * G_0) + (P_3 * P_2 * P_1 * P_0 * ci)$$

이를 이용하면, ripple carry adder 보다 더 많은 논리회로를 필요로 하지만, 전체 delay는 가장 상위의 Full adder의 결과가 계산되는 시점에서 결정되므로, ripple carry adder보다 delay를 작게 하며 계산을 빠르게 끝낼 수 있다는 장점이 있다.



,

<4bit carry look-ahead adder>

또한, adder 뿐만 아니라, multiplier등에서도 베릴로그를 이용하지 않고 실제로 회로를 조합해서 만들 경우, 이러한 회로의 조합에 따라 delay가 달라질 수 있다는 것을 생각해 볼 필요가 있다.

4. Conclusion

verliog를 이용하여 직접 회로를 짜려면 어려울 수 있는 adder, multiplier와 fused-multiplier를 간편하게 구현하고, 시뮬레이션까지 편리하게 할 수 있었다. HDL의 중요성을 실감 할 수 있는 보람찬 lab이었던것 같다.

ripple carry adder, carry look-ahead adder 사진 및 식 출처 :

https://catslikefish.tistory.com/entry/Carry-Lookahead-Adder?category=618664

https://www.electronicshub.org/carry-look-ahead-adder/