4Bit Full Adder 설계

김민기

목차

```
1 | RTL code (full_adder_4bit.v)
```

RTL code (full_adder_4bit.v)

```
module half_adder (
    input  a,
    input  b,
    output sum,
    output carry
);
    // half adder
    assign sum   = a ^ b;
    assign carry = a & b;

endmodule
```

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

1 | Half adder는 a, b 두 개의 입력을 받는다.

2 | sum과 carry는 각각 xor, and gate로 표현 할 수 있다.

RTL code (full_adder_4bit.v)

```
81  module full_adder (
82      input  a,
83      input  b,
84      input  cin,
85      output sum,
86      output c
87  );
88
89      wire w_ha_sum, w_ha0_c, w_ha1_c;
90      assign c = w_ha0_c | w_ha1_c;  // to full adder output c
91
92      half_adder U_HA0 (
93          .a    (a),         // from full adder input a
94          .b    (b),
95          .sum  (w_ha_sum),
96          .carry(w_ha0_c)
97      );
98
99      half_adder U_HA1 (
100         .a    (w_ha_sum),  // from half adder0 output sum
101         .b    (cin),
102         .sum  (sum),       // to full adder output sum
103         .carry(w_ha1_c)
104     );
105 endmodule
```

a	b	C	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

1 | Full adder는 cin이 추가되고,
full adder를 만들기 위해서는 두 개의 half adder가 필요하다.
full adder의 carry는 두 half adder 각각의 carry를 입력으로 하는 or gate의 출력이다.

2 | Full adder의 sum은 HA0의 sum과 HA1의 두번째 입력(cin)을 더해주어야 하는데,
여기서 HA0의 sum이 HA1의 첫번째 입력이 된다.
이 둘을 연결시켜주기 위해서 w_ha_sum이라는 wire를 선언한다.

3 | Full adder의 carry는 두 half adder의 carry가 or gate를 거쳐 나오게 되는데,
두 half adder의 carry를 or gate와 연결시켜주기 위해
w_ha0_c와 w_ha1_c라는 wire를 선언한다.

4 | 93번째 줄에 .a(a) 이러한 방식은 이름에 의한 결합이라고도 하는데,
.은 현재 모듈 아래에 있는 하위 모듈을 가리킨다. 즉, half adder 모듈을 가리킨다.
a는 하위 모듈의 변수 이름이다.
(a)는 현재 모듈의 변수 이름이다.
이러한 방식은 순서대로 적는 방식보다 뛰어난 가독성과 실수를 줄일 수 있다는 장점이 있다.

RTL code (full_adder_4bit.v)

```
29  module adder_4bit (
30      input  a0,
31      input  a1,
32      input  a2,
33      input  a3,
34      input  b0,
35      input  b1,
36      input  b2,
37      input  b3,
38      input  cin,
39      output sum0,
40      output sum1,
41      output sum2,
42      output sum3,
43      output c
44  );
```

```
46      wire w_fa0_c, w_fa1_c, w_fa2_c;
47
48      full_adder U_FA0 (
49          .a  (a0),
50          .b  (b0),
51          .cin(cin),
52          .sum(sum0),
53          .c  (w_fa0_c)
54      );
55
56      full_adder U_FA1 (
57          .a  (a1),
58          .b  (b1),
59          .cin(w_fa0_c),
60          .sum(sum1),
61          .c  (w_fa1_c)
62      );
63
64      full_adder U_FA2 (
65          .a  (a2),
66          .b  (b2),
67          .cin(w_fa1_c),
68          .sum(sum2),
69          .c  (w_fa2_c)
70      );
71
72      full_adder U_FA3 (
73          .a  (a3),
74          .b  (b3),
75          .cin(w_fa2_c),
76          .sum(sum3),
77          .c  (c)
78      );
79  endmodule
```

1 | 4bit full adder는 full adder 4개로 만들 수 있다.
4bit들의 연산이므로 sum도 4bit로 구성된다.

2 | 여기서도 마찬가지로 이름에 의한 결합 방식을 사용하였고,
4bit의 sum은 각각의 full adder의 a, b 값의 합으로 나오게 된다.

3 | 4bit full adder는 carry값이 다른 full adder에 영향을 준다.
FA0의 carry는 FA1의 cin이 되고, FA1의 carry는 FA2의 cin이 된다.
FA1의 carry는 FA2의 cin이 되고, FA2의 carry는 FA3의 cin이 된다.
이를 연결해주기 위해 각각 w_fa0_c, w_fa1_c, w_fa2_c라는 wire로 선언한다.
4bit full adder의 최종 carry는 FA3의 carry이다.

RTL code (full_adder_4bit.v)

```v
`timescale 1ns / 1ps

module adder (
    input  [3:0] a,
    input  [3:0] b,
    output [3:0] sum,
    output       c
);

    adder_4bit U_FA_4bit (
        .a0(a[0]),
        .a1(a[1]),
        .a2(a[2]),
        .a3(a[3]),
        .b0(b[0]),
        .b1(b[1]),
        .b2(b[2]),
        .b3(b[3]),
        .cin(1'b0),
        .sum0(sum[0]),
        .sum1(sum[1]),
        .sum2(sum[2]),
        .sum3(sum[3]),
        .c(c)
    );

endmodule
```

이번 설계에서 cin은 0으로 고려하지 않는다.

1 | 이 모듈은 4개의 full adder 유닛을 하나로 통합하여 4bit 연산을 수행하는 Top 모듈이다.

2 | 입, 출력을 벡터로 표현하여 a0, a1, a2, a3 과 같은 개별 선언과 달리
코드가 간결해지고, 실수를 방지 할 수 있다.
나중에 8bit나 16bit로 변경 시, 벡터의 범위만 수정하면 되므로 유지보수에 매우 유리하다.

3 | 하위 모듈들을 하나의 Top 모듈로 통합하면 RTL schematic 적으로 이점이 있다.
단순히 4개의 full adder 유닛이 아닌 하나의 4bit full adder로 설계 의도가 명확하다.

2 | RTL code (tb_full_adder_4bit.v)

RTL code (tb_full_adder_4bit.v)

```
1  `timescale 1ns / 1ps
2
3  module tb_adder_4bit ();
4
5      reg [3:0] a, b;
6      wire [3:0] sum;
7      wire c;
8
9      integer i = 0, j = 0;  // data type 2 (0 or 1), 32bit
10
11     adder dut (
12         .a  (a),
13         .b  (b),
14         .sum(sum),
15         .c  (c)
16     );
17
18     initial begin
19         #0;
20         a = 4'b0000;
21         b = 4'b0000;
22         #10;
23
24         for (i = 0; i < 16; i = i + 1) begin
25             for (j = 0; j < 16; j = j + 1) begin
26                 a = i;
27                 b = j;
28                 #10;
29             end
30         end
31         $stop;
32         #100;
33         $finish;
34     end
35 endmodule
```

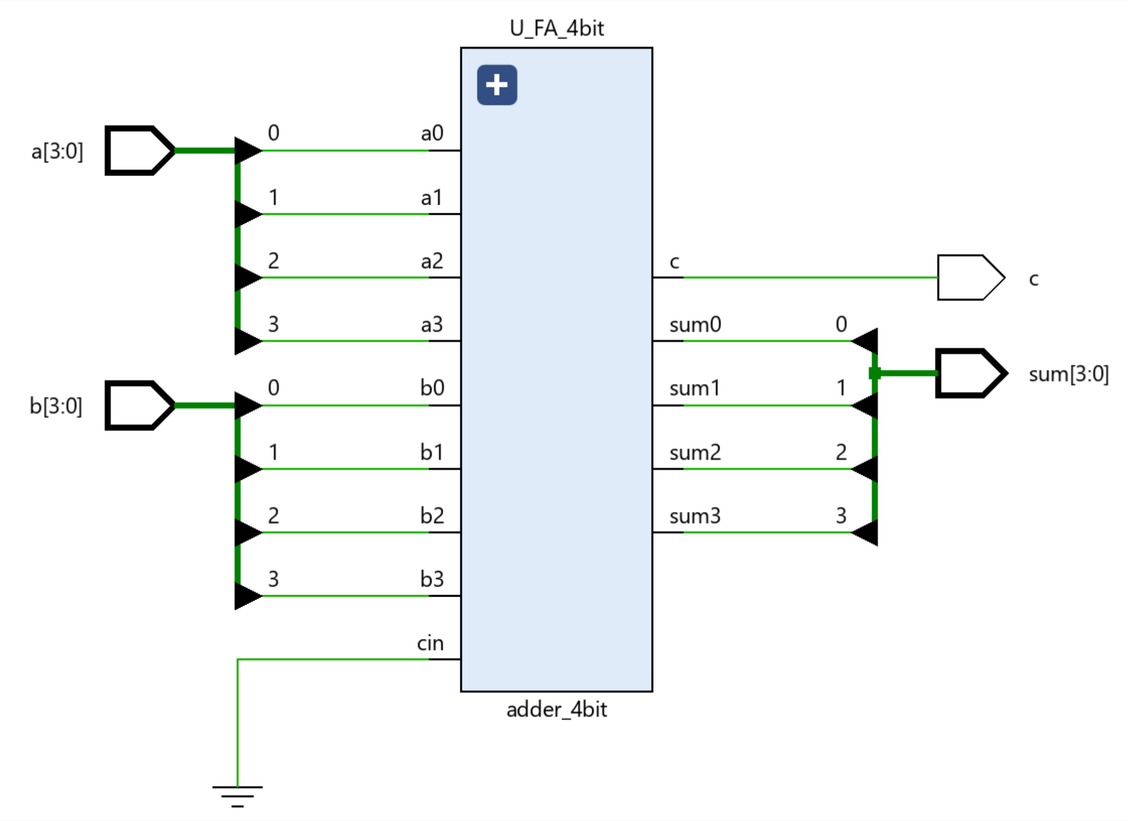
1 | Testbench 파일은 쉽게 말해 내가 설계한 회로가 설계한 대로 동작하는지
가상의 입력을 넣어 출력을 확인하는 Simulation을 위한 파일이다.

2 | DUT(Design Under Test)의 입력인 a, b를 reg로 선언하고,
출력인 sum, c를 wire로 선언한다.
a, b는 직접 값을 넣어주기 때문에 그 값을 저장하기 위해 reg로 선언하고,
sum, c는 a, b에 의한 결과 값이고 이를 연결하기 위한 wire로 선언한다.

3 | Initial begin ~ end 로 delay와 for문을 이용해 simulation을 위한 코드를 작성한다.
a, b의 값을 0으로 초기화 시켜준다. (simulation시 X 값 방지)
for문을 이용해 4bit인 a, b의 모든 경우의 수를 계산하며 그 사이에 10ns의 delay를 준다.
모든 경우가 끝나면 simulation을 멈추고, 100ns 뒤에 simulation을 완전히 끝낸다.

3 | RTL Schematic

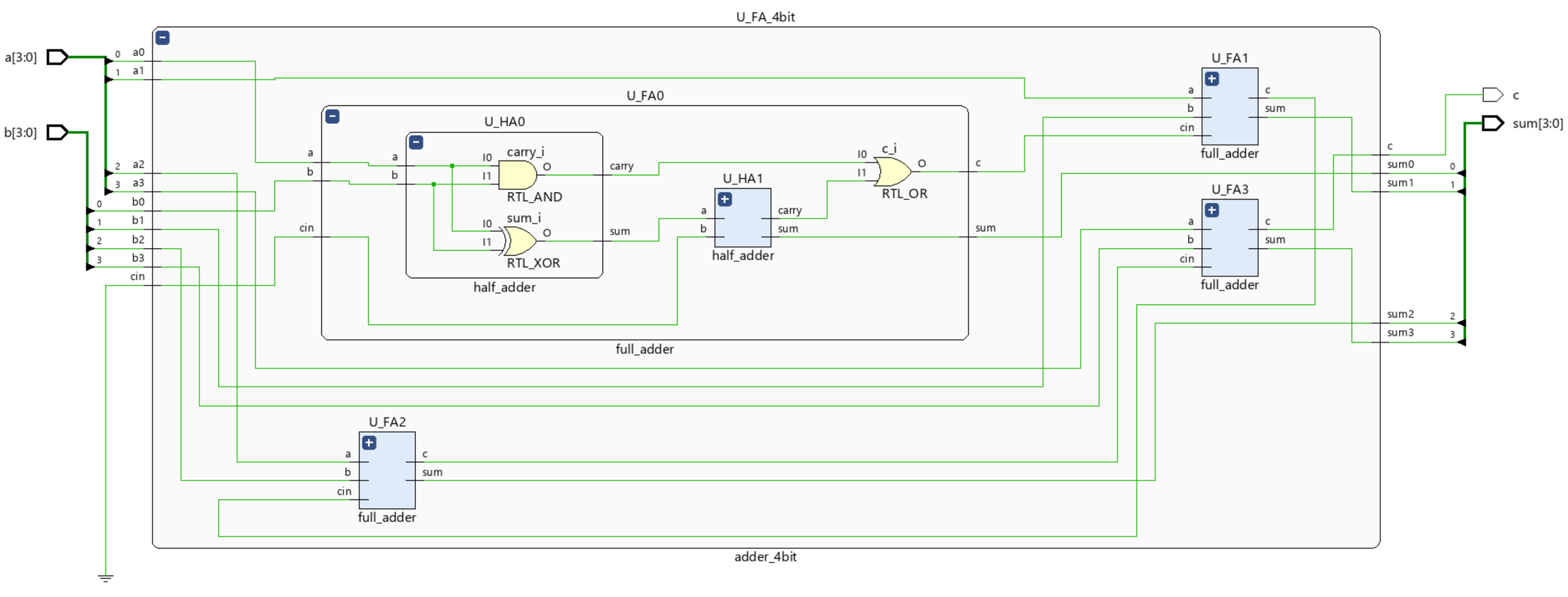
RTL Schematic



1 | 4bit a, b와 1bit cin이 input으로, 1bit c와 4bit sum이 output으로 U_FA_4bit와 알맞게 연결되어 있다.

2 | cin은 초기에 0으로 설정했기 때문에 GND와 연결되어 있다.

RTL Schematic

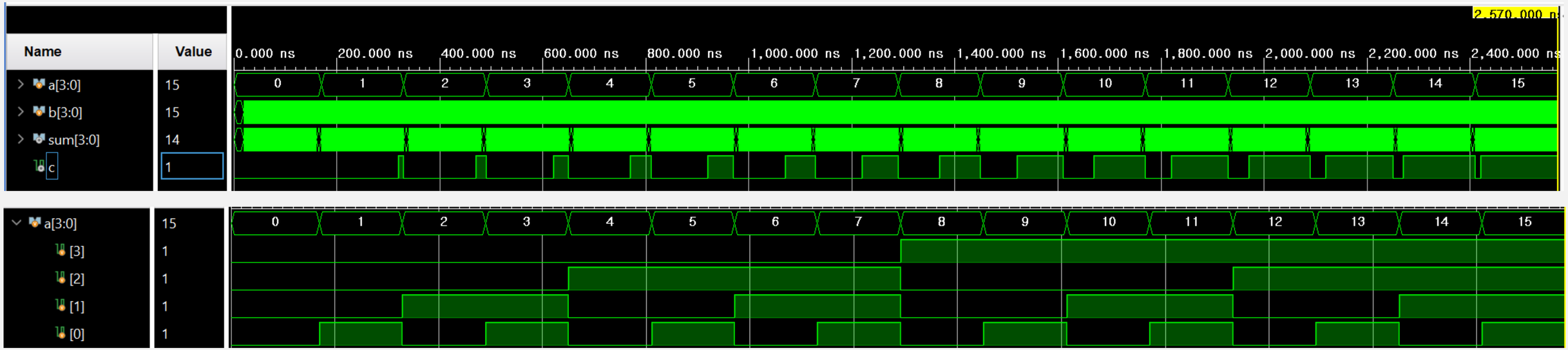


1 | U_FA_4bit Unit의 내부를 보면 full adder 4개로 구성 되어있고, 하나의 full adder 안에는 half adder 2개와 or gate로, 마지막으로 half adder는 입력의 and와 xor gate로 이루어져 있다.

2 | Input이 여러 개의 unit과 gate를 거쳐 output까지 전달되는데, 실제 신호가 제대로 의도한 대로 전달 되는지를 확인하기 위해서는 simulation을 통한 waveform을 보면 확인할 수 있다.

4 | Simulation Waveform

Simulation Waveform

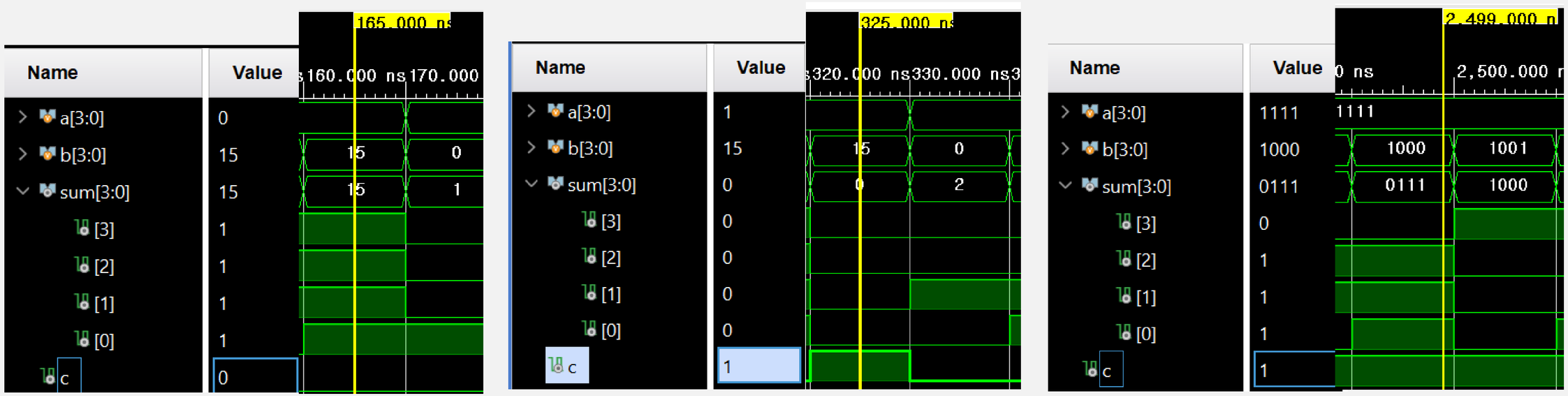


1 | 위의 사진은 simulation을 실행했을 때 나오는 전체 waveform의 모습이다.

2 | 4bit input a를 보면 10진수 a 숫자에 맞게 2진수로 표현한 4bit a값이 하나씩 증가하는 모습을 볼 수 있다.
ex) 0000 -> 0001 -> 0010 -> 0011 -> 0100 -> ~ -> 1100 -> 1101 -> 1110 -> 1111

3 | b의 값은 a와 마찬가지로 1씩 증가하는데, a의 for 문 안에 존재하므로 수가 너무 많아져 생략한다.
ex) a=0일 때, b=0~15 / a=1일 때, b=0~15

Simulation Waveform



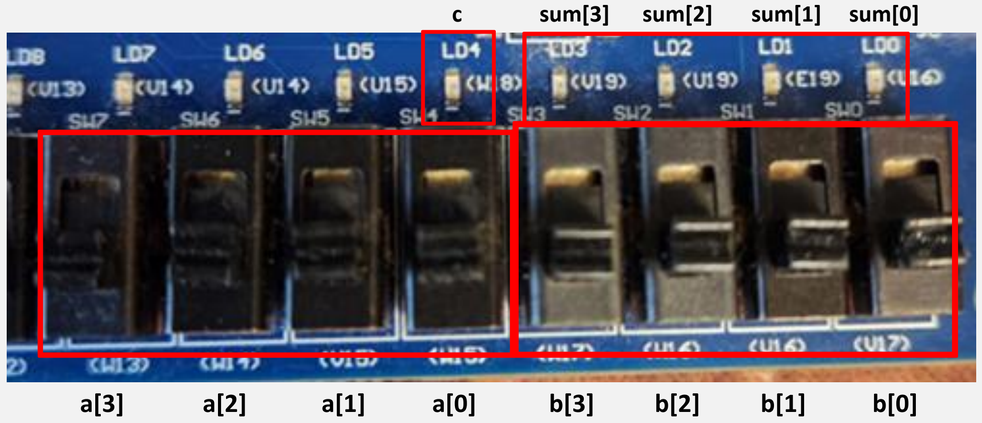
1 | 165ns 시간대를 보면 a와 b의 합이 sum 값으로 알맞게 계산되는 모습을 보인다. / a(0) + b(15) = sum(15) -> 4'b1111

2 | 325ns에서는 a = 1, b = 15로 첫번째 carry가 발생한다. 2진수로 표현하면 0001과 1111인데, 이를 더하게 되면
c = 1, sum은 0000으로 나오게 된다. 이에 대한 확인은 사진을 통해 알 수 있다.

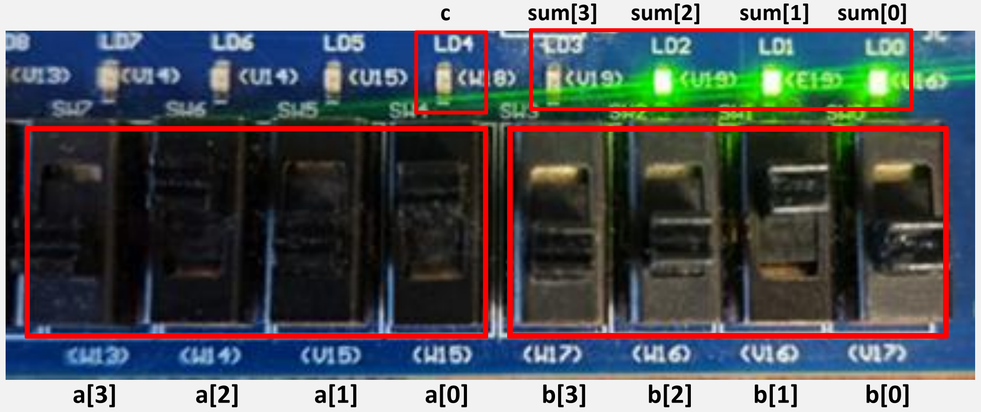
3 | a, b의 합이 16이 넘어가는 순간부터 carry는 항상 일어나므로 계속 valid한 1의 값을 가진다.
마지막 사진은 2,499ns 시간대에서 a(15) + b(8)으로 값이 16이 넘어가면서 carry가 발생하고, 16을 제외한 7의 값이 sum으로 나온다.
c = 1, sum(7) -> 4'b0111

5 | Result & Trouble Shooting

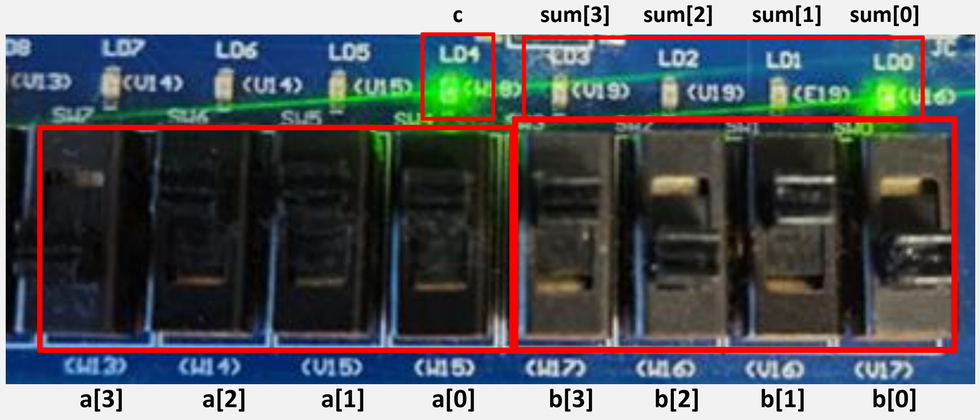
Result



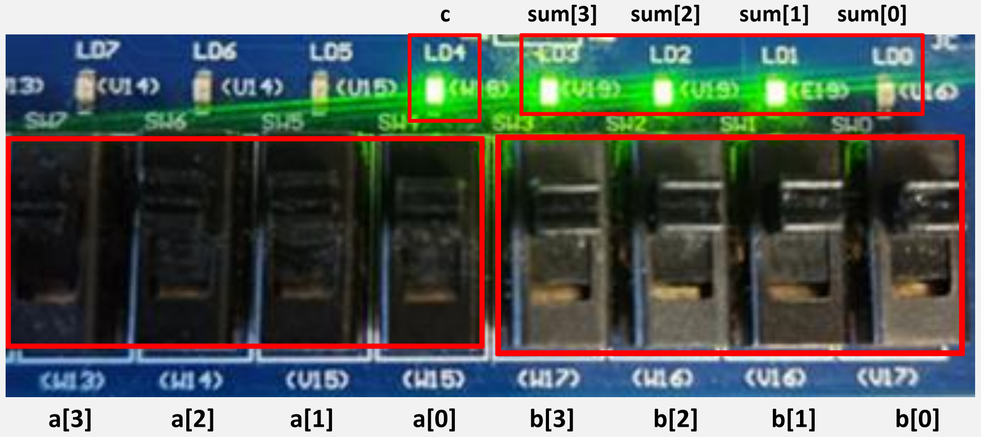
0000 + 0000 = 0000 -> carry와 sum 모든 곳에
불이 켜지지 않는다.



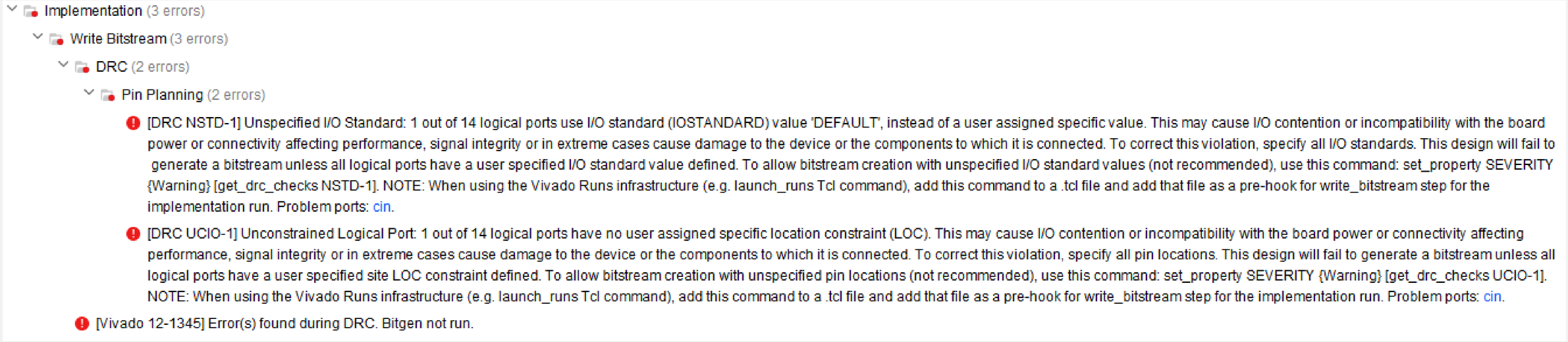
0101(5) + 0010(2) = 0111(7) -> carry는 없어 LD4는 불이 켜지지 않고,
sum은 0111이기 때문에 sum[2] ~ sum[0]까지 불이 켜진다.



0111(7) + 1010(10) = 10001(17) -> carry가 발생하여 LD4에 불이 켜지고,
sum은 0001이기 때문에 sum[0]인 LD0에만 불이 켜진다.



1111(15) + 1111(15) = 11110(30) -> carry가 발생하여 LD4에 불이 켜지고,
sum은 1110이기 때문에 sum[3] ~ sum[1]까지 불이 켜진다.



1 | [DRC NSTD-1] Unspecified I/O Standard

14개의 logical ports 중 하나가 사용자 지정 값이 아닌 default 값을 사용 중이다.

- 보드와 FPGA 간의 전압 레벨이 맞지 않아 하드웨어 손상이나, FPGA가 0과 1을 구분하지 못하는 논리적 오류가 나타날 수 있다.

2 | [DRC UCIO-1] Unconstrained Logical Port

14개의 logical ports 중 하나가 사용자가 지정한 핀 번호에 할당되어 있지 않다.

- 할당이 되어있지 않으면 bitstream을 생성할 수 없다.

Trouble Shooting

```
1   `timescale 1ns / 1ps
2
3   module adder (
4       input  [3:0] a,
5       input  [3:0] b,
6       input        cin,
7       output [3:0] sum,
8       output       c
9   );
10
11      adder_4bit U_FA_4bit (
12          .a0(a[0]),
13          .a1(a[1]),
14          .a2(a[2]),
15          .a3(a[3]),
16          .b0(b[0]),
17          .b1(b[1]),
18          .b2(b[2]),
19          .b3(b[3]),
20          .cin(cin),
21          .sum0(sum[0]),
22          .sum1(sum[1]),
23          .sum2(sum[2]),
24          .sum3(sum[3]),
25          .c(c)
26      );
27
28  endmodule
```

cin을 따로 input을 주지 않고 1'b0으로 고정

```
1   `timescale 1ns / 1ps
2
3   module adder (
4       input  [3:0] a,
5       input  [3:0] b,
6       output [3:0] sum,
7       output       c
8   );
9
10      adder_4bit U_FA_4bit (
11          .a0(a[0]),
12          .a1(a[1]),
13          .a2(a[2]),
14          .a3(a[3]),
15          .b0(b[0]),
16          .b1(b[1]),
17          .b2(b[2]),
18          .b3(b[3]),
19          .cin(1'b0),
20          .sum0(sum[0]),
21          .sum1(sum[1]),
22          .sum2(sum[2]),
23          .sum3(sum[3]),
24          .c(c)
25      );
26
27  endmodule
```

H/W와 직접 연결되는 Top module의 adder code이다.

```
## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {b[0]}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {b[1]}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports {b[2]}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {b[3]}]
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports {a[0]}]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {a[1]}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports {a[2]}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports {a[3]}]
#set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports {sw[8]}]
#set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports {sw[9]}]
#set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports {sw[10]}]
#set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports {sw[11]}]
#set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports {sw[12]}]
#set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports {sw[13]}]
#set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports {sw[14]}]
#set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports {sw[15]}]


## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {sum[0]}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {sum[1]}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {sum[2]}]
set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {sum[3]}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {c}]
#set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led[5]}]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {led[6]}]
```

FPGA 설계의 물리적 핀 배정, 타이밍 요구사항, I/O 전압 등
제약 조건을 정의하는 .xdc(Xilinx Design Constraints) 파일이다.

1 | 코드를 수정하기 전 원래는 input에 cin을 지정해주었다. 그 후에 xdc 파일에서 핀 배정을 해줄 때, cin은 sw로 직접 사용하지 않는다고 생각하여, 배정을 따로 하지 않았다. 그 결과 전과 같은 에러가 떴다.

2 | 이를 해결하려면 cin을 핀에 배정해주는 방법도 있다. 처음에는 그렇게 에러를 해결했지만,
교수님께서 이번 설계에는 cin의 값을 0으로 제약하기 때문에 input을 주지 않고, module을 instance화 할 때 cin을 1'b0로 고정하는 방식을 알려주셔서 이번 설계에서는 이 방법으로 사용하여 에러를 해결했다.

감사합니다