

# **36-bit RCA**

**Venti**

## 1. Problem statement

Ripple carry adder는 가장 간단한 가산기의 구현 방법이다. 가산기는 덧셈 연산을 수행하는 논리회로인데, 두 입력을 받아 합(Sum)과 올림수(carry)를 출력하는 반가산기(HA, Half adder)와, 이 반 가산기 두 개를 합쳐, 두 입력과 올림수를 입력 받아 합과 올림수를 출력하는 전가산기(FA, Full adder)가 기본적인 형태이다. RCA는 이러한 전가산기를 단순히 직렬 연결하여 만든 회로인데, 예를 들어 이번 과제에서 제시된 36-bit RCA는 FA 36개를 직렬로 연결한 뒤, 각각의 FA에 MSB부터 LSB까지 순서대로 입력하면 두 36-bit 이진수의 덧셈 연산을 수행할 수 있다.

논리 회로로 구현하기 쉽다는 것이 RCA의 장점이지만, RCA는 연산할 비트 수가 늘어나면 그만큼 전가산기를 더 직렬 연결하는 형태이므로, Critical path가 매우 길어지는 문제가 발생한다. 특히 전가산기는 조합 논리 회로이기 때문에, 모든 하위 비트의 FA의 연산이 끝난 후에야 정확한 carry를 받아올 수 있기 때문에, 잠시 동안 잘못된 값을 출력할 수 있다. 이렇게, carry로 인해 발생하는 문제를 해결하기 위해 Carry skip adder나 Carry select adder, 혹은 logic level 자체를 낮추기 위해 Brent-Kung adder나 Lander-Fischer adder 구조를 고려해 볼 수 있다.

## 2. Design with verilog

### 1) Descriptions for inputs and outputs

앞서 언급한 것 처럼, 36-bit RCA는 두 36비트 이진수를 입력 받아 연산한다. 이때, RCA는 36개의 FA가 직렬 연결된 구조이고, 하위 비트의 FA는 상위 비트에 Sum과 Carry만을 전달하고, 입력은 각각의 FA에 병렬적으로 이루어지게 된다. 가장 출력 단자에 가까운 쪽의 FA에 연산할 수의 MSB를 입력하고, 하위 비트의 FA에는 하나씩 낮은 자리의 값을 입력한다.

두 36비트 수의 덧셈 연산은, 오버플로우가 없다고 가정할 때 36비트로 출력되므로, 출력 또한 각각의 FA의 Sum을 병렬적으로 관찰하는 것을 통해 확인할 수 있다. 이때, 최상위 비트의 FA의 Carry out을 관찰함으로써 오버플로우 발생 여부를 알 수 있다.

### 2) Gate-level design and schematic of full adder

input		output		$BC_{in}$
A	B	$C_{in}$	S	$C_{out}$
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

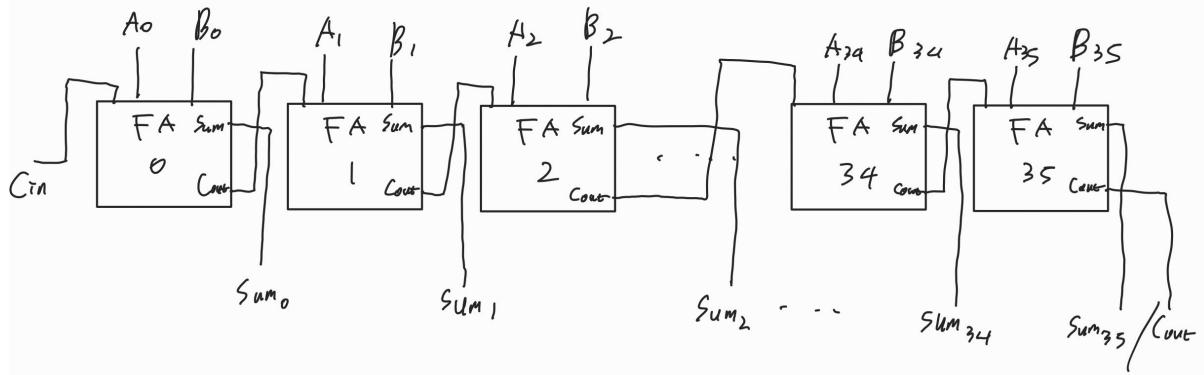
$S = A \oplus B \oplus C_{in}$

$C_{out} = AC_{in} + BC_{in} + AB$

위에서 언급한 것 처럼, FA는 두 개의 HA를 이용하여 계층적인 구조로 설계할 수 있지만, 그 자체로도 하나의 조합 논리 회로이므로, 위와 같이 Truth table을 이용하여 동작을 기술하고, 이를 Karnaugh-map을 이용해 간소화하여 회로로 나타낼 수도 있다. 하나의 FA는 세 입력  $A, B, C_{in}$ 을 받아 두 출력  $S, C_{out}$ 을 내놓는다. 이를 위의 그림과 같이 카르노 맵을 이용해 간소화 할 수 있는데,  $S$ 는 더 이상 간소화 할 수 없으므로 세 수의 XOR 연산으로 나타낸다.

진리표를 좀 더 자세히 살펴보면, FA는 1 비트 덧셈기이므로, 오버플로우가 없을 경우 그 결과는 1 비트로 표현할 수 있다. 이때, 입력 신호에 포함된 1의 갯수가 1개라면 1이 출력되지만, 두개 이상일 경우 오버플로우가 발생하고, 이는 Carry out을 통해 표현된다. 또, 1이 두개 밖에 없을 경우 출력은 100이므로, Sum은 00이 된다. 만약 세 입력이 모두 1이라면 그 결과는 11로, 오버플로우도 발생하고 합도 10이므로, Sum과 Carry out이 모두 1이 된다. XOR 연산은 입력에 포함된 1이 홀수개이면 1을 출력하고 그렇지 않으면 0을 출력하므로, Sum은 XOR 연산으로 나타낼 수 있다. Carry out은 입력에 포함된 1의 갯수가 두개 이상일 경우에 1을 출력한다고 했으므로, 위와 같이 각각의 두 입력을 AND 연산하여 1이 두개 이상 포함되었는지 확인할 수 있다.

### 3) Block diagram



36-bit RCA를 위와 같이 Block diagram으로 나타낼 수 있다. 36개의 FA를 직렬 연결하여, 하위 비트의 Carry를 상위 비트의 FA의 Carry 입력으로 사용하고, 각각의 FA에는 두 입력 A, B의 값을 한 비트씩 병렬적으로 입력한다. Sum 또한 각각의 FA의 Sum 출력을 한 비트씩 병렬적으로 참조하여 얻을 수 있고, 최종적으로 RCA의 Carry out은 오버플로우 발생 여부만을 표기하므로 한 비트로 출력된다.

#### 4) Description of operation

RCA의 동작은 FA를 단순히 필요한 비트 수 만큼 직렬 연결한 것이므로 그와 크게 다르지 않다. 각 비트를 구성하고 있는 FA는 서로 독립적인 연산을 수행하고, 단순히 Carry를 전달하는 구조이다. 이때 하위 비트에서 상위 비트로 전파되는 Carry 신호는, 위에서 살펴본 FA 구조에 의하면, 하나의 FA를 거칠 때마다 2 개의 level이 추가되므로, 상위 비트일수록 이 캐리의 전파 지연에 의해 정확한 값을 얻을 때까지 더 많은 시간이 소요된다.

이때, 전가산기는 반가산기에는 존재하지 않는 Cin이라는 입력이 존재하는데, 이는 이러한 형태의 가산기를 직렬연결하여 더 큰 비트의 숫자를 연산하는 회로를 구현하기 위해 존재하는 신호이다. 두 1비트 이진수의 합은 2비트로 표현할 수 있는데, 10진수 연산에서 했던 것처럼, 이진수의 연산에서도 이렇게 발생한 올림 수를 다음 비트로 전달하여 계산해야 한다. 따라서, 전가산기는 피연산자인 입력 신호 두 개와 하위 비트에서 전파된 올림수 까지, 총 세개의 입력을 필요로 한다.

2비트 이진수 01, 11의 덧셈을 예시로 설명하자면, 최하위 비트에서 1과 1을 더하면, 이는 10으로 원래는 오버플로우가 발생해야 하지만, 그렇게 하는 대신 현재 비트에서는 Sum으로 0을 출력하고, Carry out으로 1을 출력하여, 이를 상위 비트를 연산하는 전가산기의 Carry in 신호로 전달하는 것이다. 이렇게하여 상위 비트에서는  $0 + 1 + 1(\text{Cin}) = 10$ 을 출력하는데, 이는 해당 2비트 가산기에서 오버플로우가 발생한 것으로, 이 가산기는 00을 Sum으로 출력하고, 1을 Carry out 신호로 출력하여, 다시 이러한 2비트 가산기를 직렬 연결하여 더 큰 비트의 수를 연산할 수 있는 회로를 만들때 사용된다.

#### 5) verilog code

```

1  module half_adder(           //declaration of module
2    input A,
3    input B,
4    output S,
5    output Cout
6  );
7
8    xor G1 (S, A, B);          //the sum of half adder is 1 if when there are odd numbers of 1(true)
9    and G2 (Cout, A, B);       //the carry out of half adder is 1 if when there are odd numbers of 1(true)
10
11  endmodule                  //module declaration is complete
12

```

<반가산기의 verilog code>

위에서 설계할 때와는 다르게, 실제로 verilog를 이용하여 설계할 때 Hierarchy를 높여 모듈을 재활용 하는 것이 더 합리적인 설계인 것 같아 반가산기를 먼저 선언했다. 반가산기는 두 입력 A, B를 받아 Sum과 Carry를 출력하는 회로인데, 세개의 입력을 받아 Carry in을 포함할 수 있는 전가산기는 두 개의 반가산기를 직렬연결 함으로써 구현할 수 있다. 두 1비트 이진수를 더하면, {A, B, S, C}와 같이 나타낼 때, {0, 0, 0, 0}, {0, 1, 1, 0}, {1, 0, 1, 0}, {1, 1, 0, 1}과 같이 진리표로 나타낼 수 있다. 이를통해, 반가산기의 Sum 출력은 입력에 1이 총수개 포함되어 있을때 1이 됨을 알 수 있고, 이는 XOR 연산을 통해 구현할 수 있다. 그래서 XOR 게이트를 선언하여 Sum 신호를 얻었다. 마찬가지로 Carry out 신호는 AND 게이트를 통해 얻을 수 있다.

```
module full_adder( //module declaration
    input A,
    input B,
    input cin,
    output s,
    output cout
);
    wire w1, w2, w3; //inner circuit wire declaration, w1 and w2 connect 2 half adder, w3 was used to calculate carry out
    half_adder M1(A, B, w1, w2); //instantiation of half adder module 1
    half_adder M0(w1, cin, s, w3); //instantiation of half adder module 0
    or (cout, w2, w3); //logic for making carry out
endmodule //module declaratino end
```

#### <전가산기의 verilog code>

전가산기는 두 개의 반가산기를 연결하여 구현할 수 있다. 그래서, 두 개의 반가산기를 instance 해야 하는데, 첫번째 가산기의 carry out 신호가 두번째 가산기의 입력으로 사용되야 하므로, 이를 전달하기 위해 wire를 선언하였다. 또한, 두 반가산기 중 하나라도 carry out을 발생시켰다면 이 전가산기는 carry out이 있는 상태이므로, 이를 계산하기 위해 or 게이트를 하나 사용했는데, 마찬가지로 여기에도 반가산기에서 출력된 신호를 전달하기 위해 wire를 선언했다.

```
1  module rca_4b( //declaration of module
2      input [3:0] A,
3      input [3:0] B,
4      input cin,
5      output [3:0] s,
6      output cout
7  );
8      wire co1, co2, co3; //inner wire for propagating cout signal from each FA module
9
10     full_adder M0(A[0], B[0], cin, s[0], co1); //instantiation of full adder module
11     full_adder M1(A[1], B[1], co1, s[1], co2); //The ripple adder requires a transfer adder as many bits as it can
12     full_adder M2(A[2], B[2], co2, s[2], co3);
13     full_adder M3(A[3], B[3], s[3], cout);
14
15 endmodule
16
17
```

#### <4비트 리플 가산기의 verilog code>

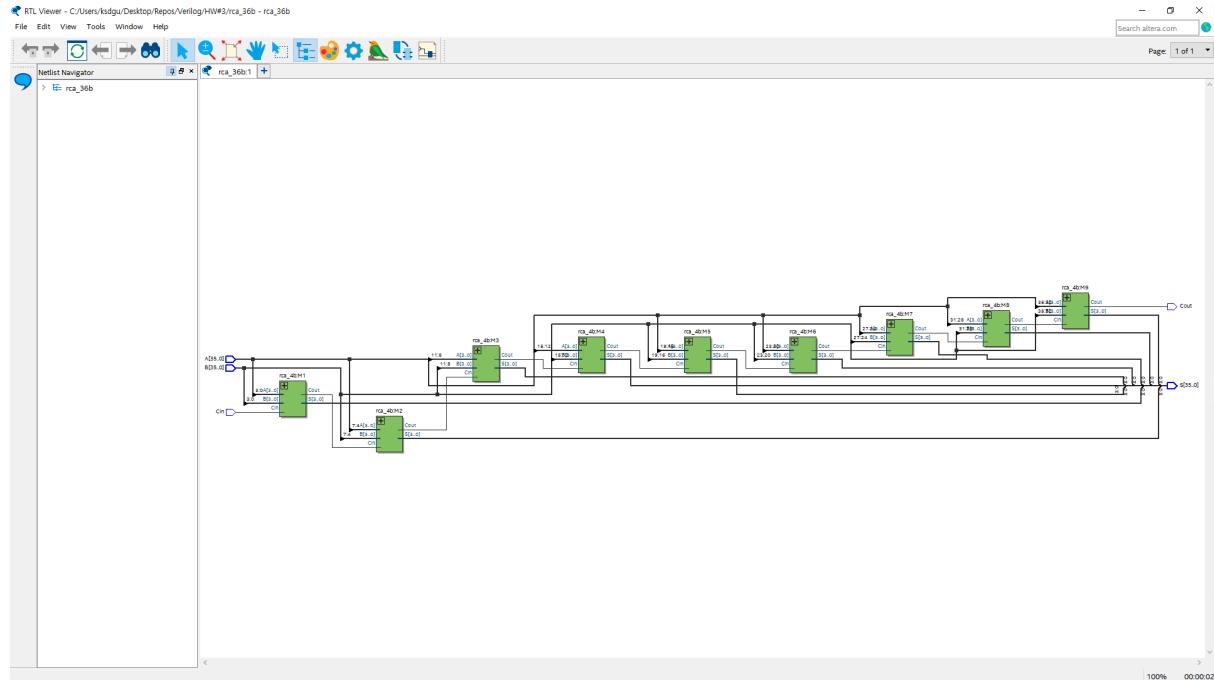
Multi bit adder는 ripple carry adder로 구현할경우, 위에서 만든 전가산기의 단순 직렬연결 형태로 구현할 수 있는데, 36개의 전가산기를 모두 선언하려면 code가 지나치게 길어지므로, 4비트 가산기 9개를 직렬연결 하는 형태로 구현하였다. 4비트 가산기는 4개의 전가산기를 직렬연결하면 만들 수 있는데, 이때, 4비트 입력과 4비트 Sum 신호는 각각의 전가산기에 1비트씩 병렬적으로 입/출력되므로, instance된 각 모듈에는 입/출력 신호중 1비트를 전달한다. 또한, carry out 신호는 다음 비트를 계산할 전가산기의 carry in 신호로 사용되어야 하므로, 이를 전달하기 위해 wire를 선언하였다.

```
1  module rca_36b (s, cout, A, B, cin); //declaration of module
2      input [35:0] A, B;
3      input cin;
4      output [35:0] s;
5      output Cout;
6
7      wire co1, co2, co3, co4, co5, co6, co7, co8; //inner wire for propagatin carry out signal from eaca rca_4b module
8
9      rca_4b M1(A[3:0], B[3:0], cin, s[3:0], co1); //instantiation of 4-bit ripple carry adder module,
10     rca_4b M2(A[7:4], B[4:1], co1, s[7:4], co2); //9 of 4-bit adders connections in series for 36-bit calculations
11     rca_4b M3(A[11:8], B[8:5], co2, s[11:8], co3);
12     rca_4b M4(A[15:12], B[12:9], co3, s[15:12], co4);
13     rca_4b M5(A[19:16], B[16:13], co4, s[19:16], co5);
14     rca_4b M6(A[23:20], B[20:17], co5, s[23:20], co6);
15     rca_4b M7(A[27:24], B[24:21], co6, s[27:24], co7);
16     rca_4b M8(A[31:28], B[28:25], co7, s[31:28], co8);
17     rca_4b M9(A[35:32], B[32:29], co8, s[35:32], Cout);
18
19 endmodule
20
```

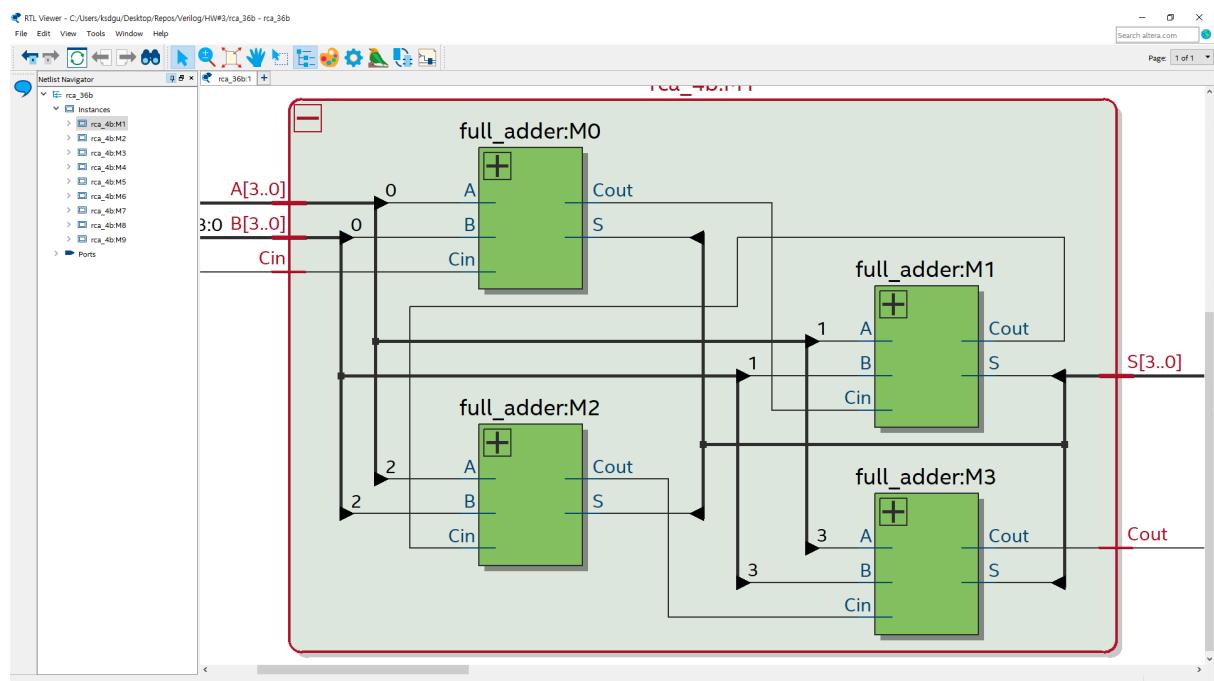
#### <36비트 리플 가산기의 verilog code>

36비트 리플 가산기는 9개의 4비트 리플 가산기를 직렬연결 하는 것으로 구현할 수 있다. 이또한 위에서 살펴본 것과 마찬가지로, 입/출력 신호는 4비트씩 병렬적으로 전달되고, 각각의 모듈 간에는 carry를 공유하기 위해 wire가 선언되어 있다.

## 6) RTL view

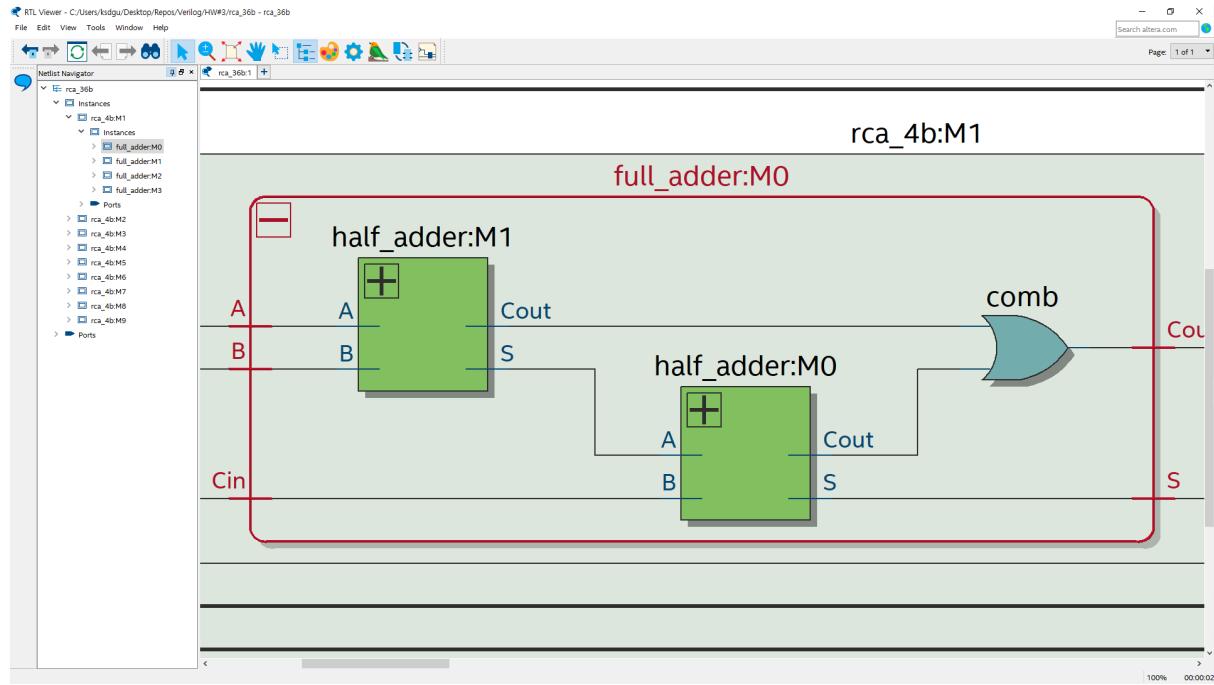


RTL view 버튼을 누르면 가장 먼저 위와 같이 9개의 4비트 리플 가산기가 직렬연결 되어있는 것을 볼 수 있다. 이때, 직렬연결 되어 가로로 길게 늘어진 회로도와 다르게, 입/출력은 병렬적으로 발생하므로, 세로로 조금 늘어진 것을 볼 수 있다. Instance된 각각의 4비트 가산기 모듈의 왼쪽 위에는 사용자가 클릭할 수 있는 + 버튼이 있는데, 이를 눌러보면 아래와 같은 회로도를 볼 수 있다.

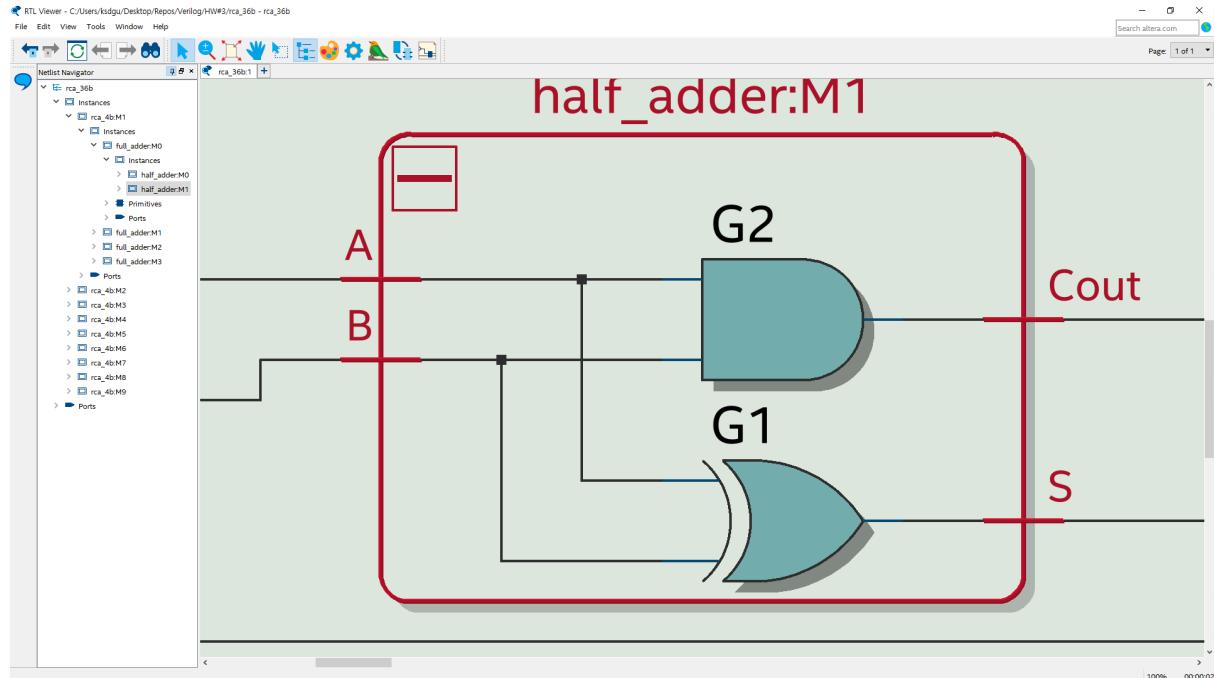


4비트 전가산기의 내부 회로가 확대되어 표시되는데, 위에서 살펴본 코드와 같이 4개의 전가산기 모듈 M0부터 M3까지 직렬연결 되어 있는 것을 확인할 수 있다. 또한, 4비트 입력

신호 A, B의 연결 상태가 wire에 0 ~ 3으로 표시되어 있으며, 각각의 S 출력 단자가 모두 S[3:0]에 연결되어 있어, 입/출력이 병렬적으로 발생한다는 것을 좀 더 명확히 확인할 수 있다.



전가산기 내부에는 두 개의 반가산기와 OR 게이트가 포함되어 있는 것을 볼 수 있다. 두 반가산기 중 하나라도 carry를 발생시키면 이 전가산기는 캐리가 발생하게 되므로, 두 Cout 출력 신호가 OR 게이트에 연결되어 있다.

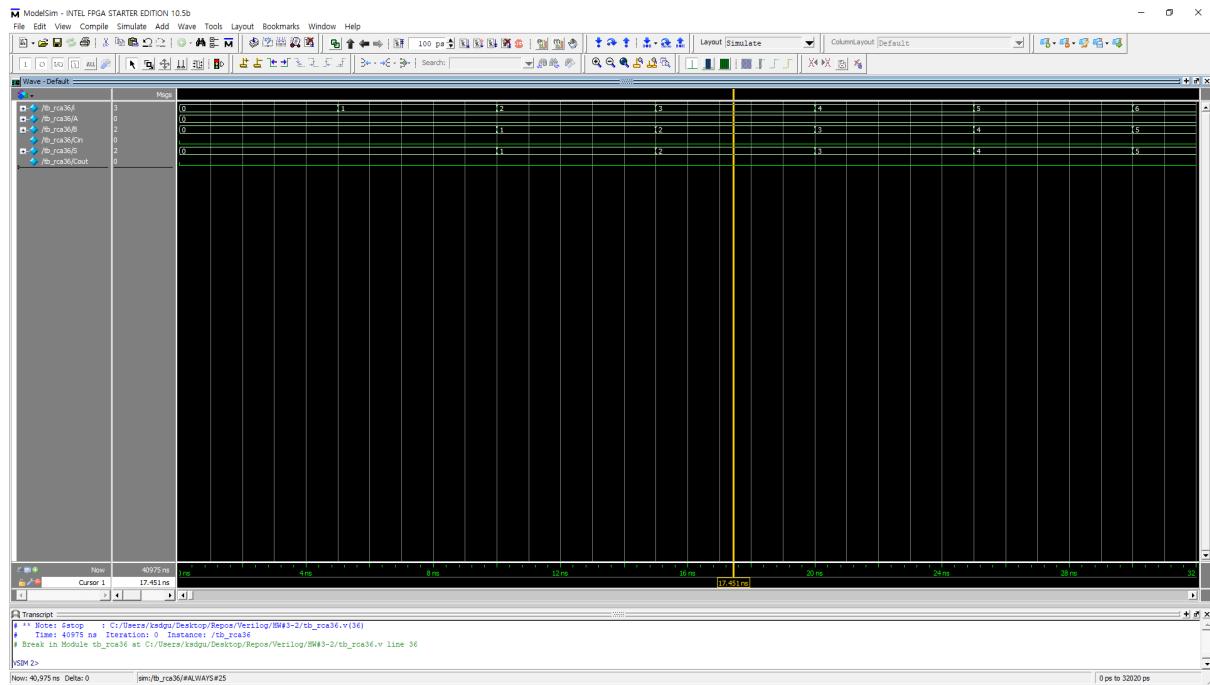


가장 작은 단위인 반가산기에는 위와 같이 두 개의 게이트가 포함되어 S와 Cout 신호가 만들어진다.

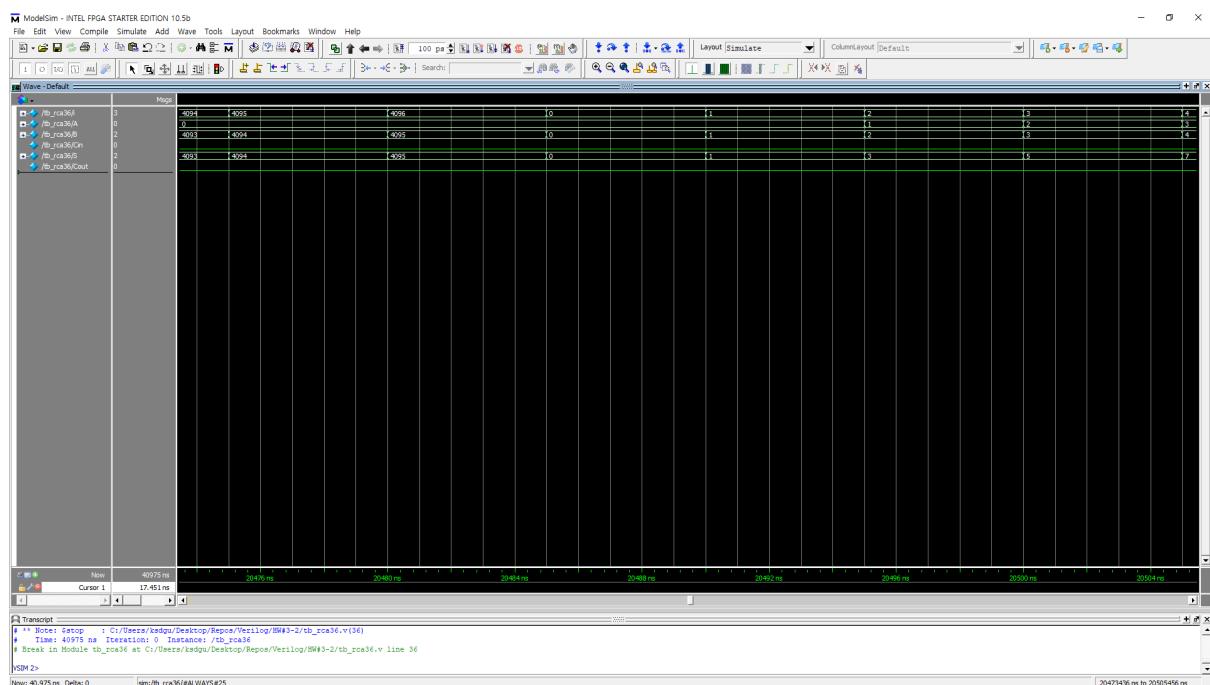
위와 같이, Verilog code로 작성된 하드웨어는 Synthesis 과정을 거쳐 자동으로 논리 회로로 합성되고, RTL(Register Transfer Level) view를 통해 이번 학기 동안 배운, 게이트 단위의 회로도로 도시할 수 있다.

### 3. Verification strategy & examples

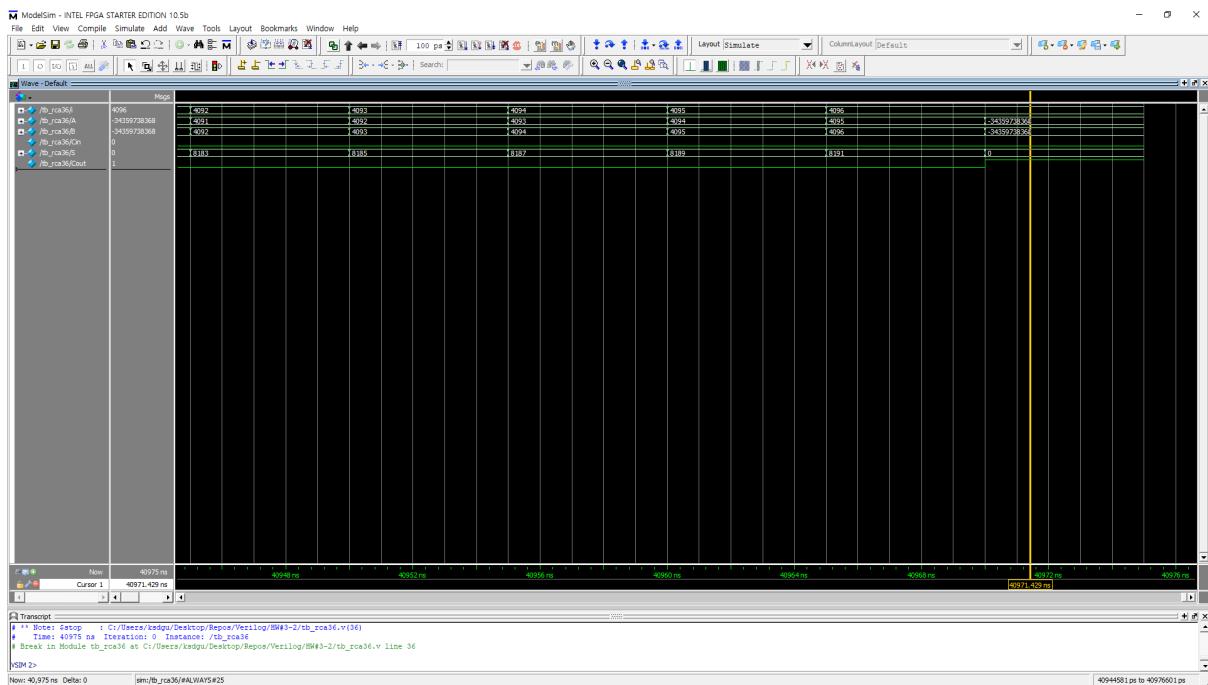
#### 1) Testbench waveform



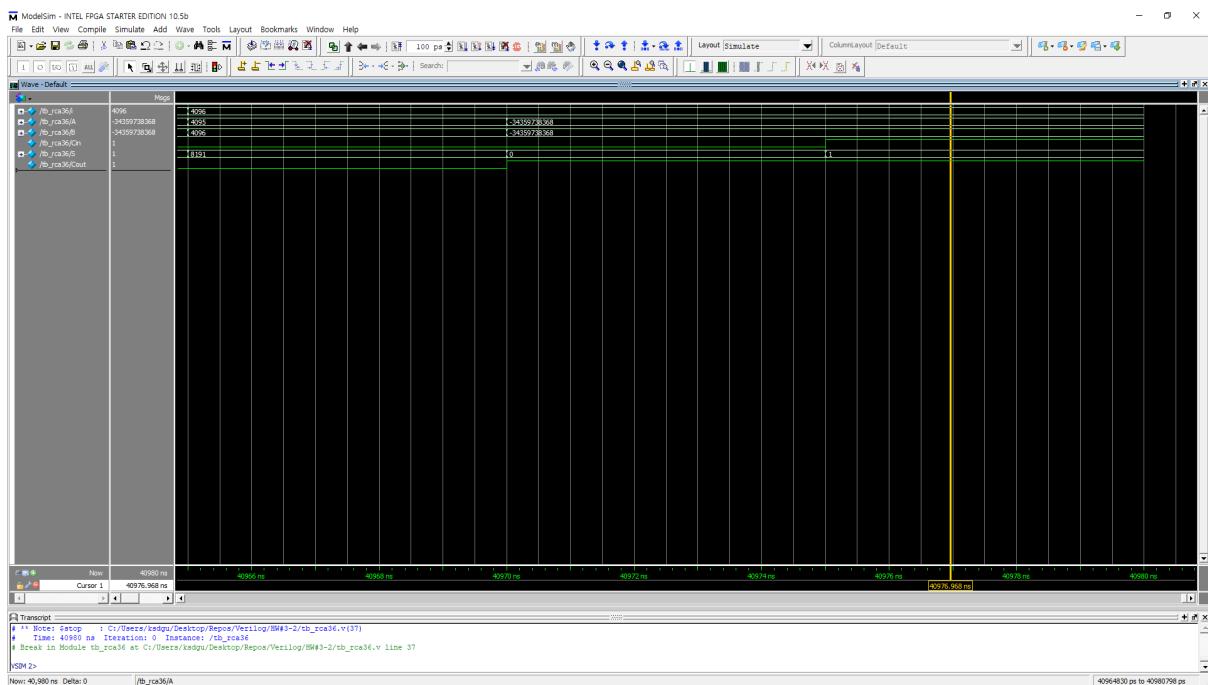
B 입력만 존재할 때에는 위와 같은 결과가 출력되는 것을 확인할 수 있었다. 출력되는 값을 좀 더 쉽게 알아보기 위하여 decimal로 결과를 표기하도록 변경했다.



이 사진은 아래에서 살펴볼 testbench code에서 입력이 바뀌는 지점을 관측한 것이다. A, B 입력이 동시에 존재할 때에는 위와 같은 파형을 얻을 수 있다.



다음으로 Cout 출력이 정상적으로 동작하는지 확인하기 위해 A와 B에 36'b1xxx를 입력했다. 이 값은 **integer**형 변수가 표시할 수 있는 범위를 아득히 초월하였으므로, **decimal**로 표기했을 때에는 위와 같이 여러 차례 오버플로우가 발생하여 -값이 입력되는 것처럼 보여졌다. 또한  $36'b1xx + 36'b1xx = 36'b0xx$ 이므로, Sum은 0이 출력되는 것을 알 수 있었고, **carry out**으로 1이 발생한 것을 확인할 수 있었다. 한편, 커서로 샘플링 된 지점의 왼쪽에서는, 아래에서 살펴볼 두번째 반복문이 막 끝난 순간인데,  $4095 + 4096 = 8191$ 로 정상적으로 연산이 이루어지고 있는 것을 확인할 수 있었다.



마지막으로, Carry in이 존재할 때에도 정상적으로 동작하는지 확인하기 위하여, 위의 케이스의 마지막에  $Cin = 1$ 을 추가하였는데, 역시 정상적으로 동작하는 것을 확인했다. 따라서, 이 회로는 정상적으로 동작하였고, 이를 **instance**하여 72비트, 108비트등의 더 큰 수의 연산이 가능한 리플 가산기를 구현할 수 있을 것이라고 추측해볼 수 있다.

## 2) Why this testbench?

```

1  `timescale 1ns/100ps
2
3  module tb_rca36();
4
5      integer i;
6
7      reg [35:0] A;
8      reg [35:0] B;
9      reg Cin;
10     wire [35:0] S;
11     wire Cout;
12
13     rca_36b m1(.A(A), .B(B), .cin(cin), .S(S), .cout(cout));
14
15     initial
16     begin
17         A = 0;
18         B = 0;
19         Cin = 0;
20     end
21
22     initial
23         $monitor("(A(%b) + B(%b)) = COUT (%b %b)", A, B, Cout, S);
24
25     always @ (A or B)
26     begin
27         for(i = 0; i < 64 * 64; i = i + 1)
28             #5 {A, B} = i;
29
30         begin A = 0; B = 0; end
31         for(i = 0; i < 64 * 64; i = i + 1)
32             #5 begin A = i; B = i + 1; end
33
34         #5 begin A = 36'b10000000000000000000000000000000; B = 36'b10000000000000000000000000000000; end
35
36         #5 Cin = 1;
37
38         #5 $stop();
39     end
40
41 endmodule

```

처음에는 적은 비트의 가산기를 시뮬레이션 할 때처럼, 단순히 반복문을 이용하여 모든 케이스에 대한 검증을 할 수 있을 것이라고 생각했다. 그러나, Unsigned 36-bit binary number의 범위는 int나 다른 자료형의 표현 가능한 범위를 아득히 초월하는 큰 수이므로, 위와 같이 하나의 변수를 이용한 수학적 연산을 통해 모든 케이스에 대한 검증을 하는 것은 불가능했다.

이때, 회로를 설계할 때에는 수학 연산을 이용할 수는 있지만, 사용하는 tool에 따라 최적화된 회로가 나오지 않을 수 있으므로, 이러한 종류의 연산을 사용하는 것을 최대한 지양해야 한다. 하지만, testbench를 설계할 때에는 수학 연산이 자주 사용되는데, testbench는 synthesis해도 회로로 합성되지 않기 때문에, 이러한 문제를 발생시키지 않기 때문이다.

그래서, 더 많은 변수를 선언하여 모든 케이스를 발생시킨 뒤 시뮬레이션을 진행하였는데, 케이스가 지나치게 많아지며 시뮬레이션 시간이 매우 길어졌고, 가정용 컴퓨터로는 정상적인 시뮬레이션이 불가능하다고 판단했다. 따라서 중요하다고 생각되는 몇몇 케이스 위주로 시뮬레이션 해야겠다고 생각했고, 이러한 검증이 필요한 케이스를 선정하는 것이 중요하다는 생각이 들었다.

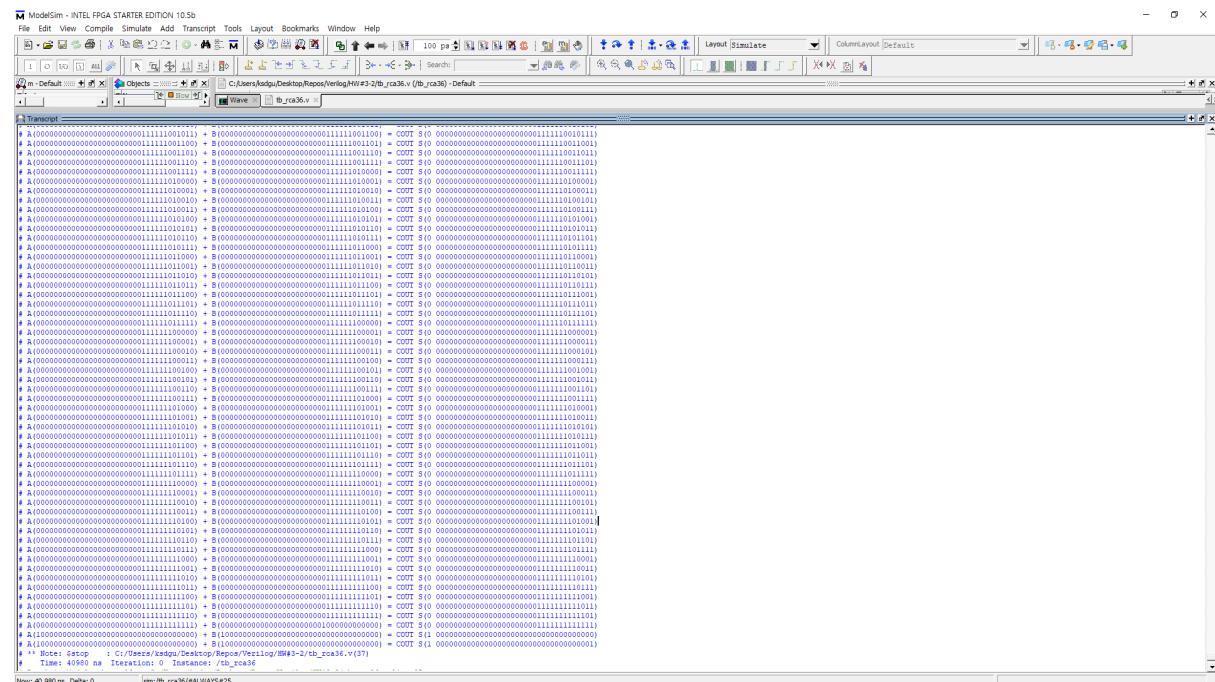
첫번째로는, 반복문을 이용하여  $64 \times 64 = 2^{12}$ 까지의 케이스를 검증하는데,  $\{A, B\} = i$ 로 선언하여, 두 레지스터형 변수 A, B에 i 값을 할당하되, 만약 B의 표현 가능한 비트 수를 초과하였으면 자동으로 A로 넘어가도록 만들었다. 물론 이는  $2^{36}$ 에 한참 미치지 못하는 수이므로, 실제로는 위의 사진과 같이 B에만 값이 입력되었다. 결국, 위 사진에서 B값과 S값이 같고, Carry out이 0이므로, 하나의 입력만 존재하는 케이스에서는 정상적으로 동작한다는 것만 확인할 수 있었다.

다음으로, 두 입력이 동시에 존재하는 케이스를 검증해보고 싶었다. 첫번째 케이스와 같이 반복문을 작성하고, 이번엔 A엔 i를, B에는  $i + 1$ 을 입력하여 서로 다른 두 수에 대해서도 정상적으로 연산이 이루어지는지 확인해 보았다. 그 결과, 두번째, 세번째 사진과 같이 A, B는 항상 1 만큼 차이나고, 두 수의 합이 Sum에 나타나면서, Carry out은 0인 것을 확인하였다. 이 케이스에 대해서도 정상적으로 동작하는 것이다.

세번째로, 위의 두 케이스에서는 Carry out이 발생하는 것을 확인할 수 없었기에, 과연 carry out이 정상적으로 발생하는지 알고싶었다. 위에서 언급한 것처럼, Carry out을 발생시킬 수 있을 만한 입력은 수학 연산으로 얻기 어려우므로, 두 입력에 36'b1xx를 직접 입력하였다. 이는 이진수의 연산으로 생각해볼 때, 오버플로우 1을 발생시키게 되고 그 값은 0이 되는 경우이다. 위의 세번째 사진에서 Carry = 1, Sum = 0인 것은 맞지만 두 입력에 이상해 보이는 값이 표시되었는데, 이는 modelsim simulation tool에서 decimal로 표기할 수 있는

범위를 넘었으므로, 프로그램 자체적으로 오버플로우가 발생하여 잘못된 값이 표기된 것이라고 생각해볼 수 있다.

마지막으로, Carry in이 존재할 때에도 정상적으로 동작하는지 알고 싶어서,  $Cin = 1$ 을 추가했다. 모든 입력은 **reg**로 선언되어 있으므로,  $Cin$ 에 1을 입력한다고 해서 두 입력 A, B가 바뀌지는 않으므로, Sum은 위에서 얻은 0에  $Cin$  1이 더해져 1이 출력되고, Carry out도 여전히 1인 것을 볼 수 있었다.



여기에 monitor 함수를 추가하여, waveform 아래의 transcript 창에서 동작 상태를 좀 더 쉽게 알아볼 수 있도록 하였다.

따라서, 모든 케이스에 대한 검증은 어려웠지만, 검증 가능한 모든 케이스에 대해서는 정상 동작하는 것을 확인할 수 있었다.

## 4 Conclusion

verilog HDL을 이용하여 실제 디지털 회로를 설계해 봄으로써, 첫 주차 쯤에 배운 Three Y(Hierarchy, Modularity, Regularity)의 개념에 대하여 좀 더 명확하게 알 수 있었다. 36비트 리플 가산기라는 매우 크고 복잡해보이는 목표라도, 설계한 것과 같이 각각의 모듈을 계층적인 구조로 배치하고, 각 모듈간의 규격화된 입/출력 신호를 잘 연결하는 것 만으로도 쉽게 구현할 수 있었다.

또한, 수십억개의 트랜지스터가 사용되는 현대의 AP/CPU등에 비해 매우 작은 36비트 가산기 정도만 되어도, 모든 케이스에 대하여 검증하는 것이 매우 어렵고 오래 걸리는 일이라는 것을 확인할 수 있었다. 그래서 검증이 필요한 케이스를 잘 선정하는 것이 중요하다는 것을 다시 한번 느끼게 되었다.