**10주차 결과보고서**

**전공: 생명과학과 학년: 4학년 학번: 20182186 이름: 김승원**

**1. 4bit Binary Parallel adder 결과 및 simulation (Verilog source code, 출력 예시, 과정 상세히)**

실내이(가) 표시된 사진

자동 생성된 설명

**<Fig 1. 병렬 가감산기>**

4 bit Binary Parallel adder는 1bit full adder를 4개를 연결하여 구성한다. 이전 bit의 full adder의 Carry가 다음 bit의 Carry in으로 들어가도록 구현했다.

Full adder를 이용해서 subtractor 또한 구현이 가능했다. 2’s complement를 통해 음수를 표현할 수 있음을 이용해서 A-B를 A+B’+1으로 계산했다. 임의의 bit과 1을 XOR gate를 통과시키면 그 bit는 반전이 일어난다. 따라서 빼고 싶은 수를 1과 함께 XOR gate를 통과시켜서 빼어질 수에 더하기를 해주고 +1을 해주면 뺄셈이 가능했다. 일반적으로 가감산기의 경우 이때 Sign이 0일 경우 덧셈을 수행하고, 1일 경우 뺄셈을 수행한다. 본 실험에서는 carry in을 통해 sign까지 역할을 할 수 있도록 사용하여 구현했다. LSB에 들어오는 carry in bit가 0일 경우 덧셈을 진행하도록 구현했고, 1이 입력될 경우 뺄셈이 진행되도록 구현했다. 즉, Carry in bit와 B의 모든 bit를 XOR gate로 구현했고, 0과 1입력에 따라 덧셈 뺄셈을 할 수 있도록 구현했다.

1, 2번 adder와 subtractor은 사실상 같은 source code로 구현할 수 있으며, 각각 Cin, Bin을 각각0과 1로 고정시켜서 adder와 subtractor을 구현했다.

**1) Verilog source code**

`timescale 1ns / 1ps

module adder4bit(//9input, 5output

input [3:0] a,b,

input cin,

output [3:0] s,

output c4

);

wire c1, c2, c3;

assign s[0] = (a[0]^(b[0]^cin))^cin,

c1 = ((a[0]^(b[0]^cin))&cin) | (a[0]&(b[0]^cin)),

s[1] = (a[1]^(cin^b[1]))^c1,

c2 = ((a[1]^(b[1]^cin))&c1) | (a[1]&(b[1]^cin)),

s[2] = (a[2]^(cin^b[2]))^c2,

c3 = ((a[2]^(b[2]^cin))&c2) | (a[2]&(b[2]^cin)),

s[3] = (a[3]^(b[3]^cin))^c3,

c4 = ((a[3]^(b[3]^cin))&c3) | (a[3]&(b[3]^cin));

endmodule

배열을 사용해서 input a, b와 s를 input으로 할당했다. Cin 입력을 LSB의 Carry in bit로 사용했다. 그리고 연산하기 위해서만 사용되고, 의미있게 남겨져야 하는 값들이거나 결과로 직접 확인해야 할 값들도 아니기 때문에, 나머지 각 adder 사이에 있는 carry bit들인 carry1, 2, 3는 wire를 통해 따로 구현했다.

**2) Test bench code(simulation code)**

`timescale 1ns / 1ps

module tb\_adder4bit;

reg[3:0] A, B;

reg Cin;

wire[3:0] S;

wire C4;

adder4bit mygate(.a(A), .b(B), .cin(Cin), .s(S), .c4(C4));

initial begin

A = 4'b0000;

B = 4'b0000;

Cin = 1'b0;

end

always @(A or B or Cin) begin

A <= #7 A+4'b0001;

B <= #11 B-4'b0001;

Cin <= #1000 ~Cin;

end

initial begin

#1000

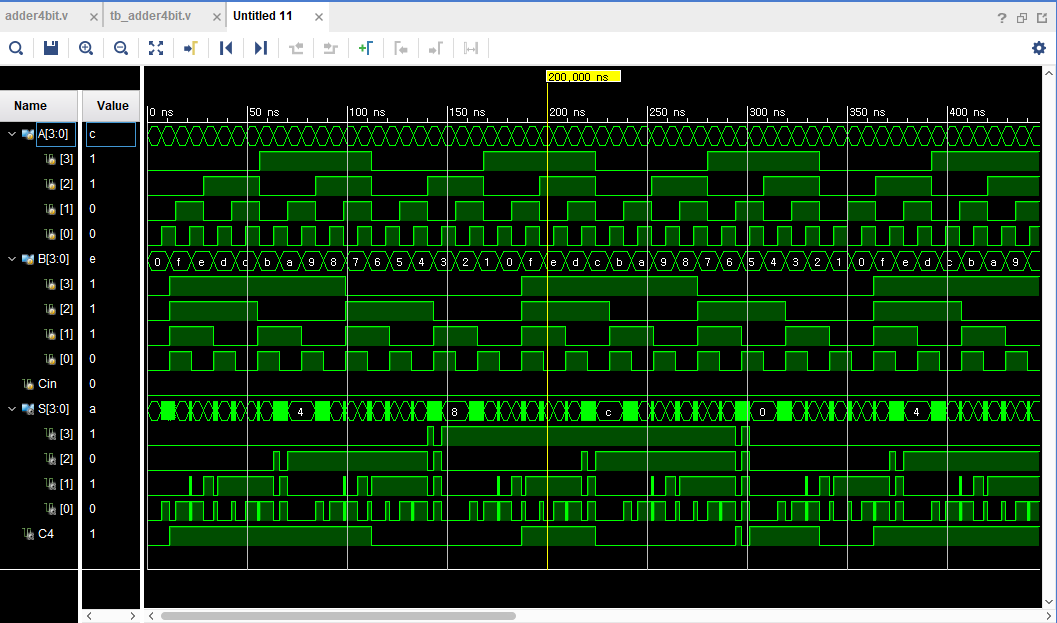
$finish;

end

endmodule

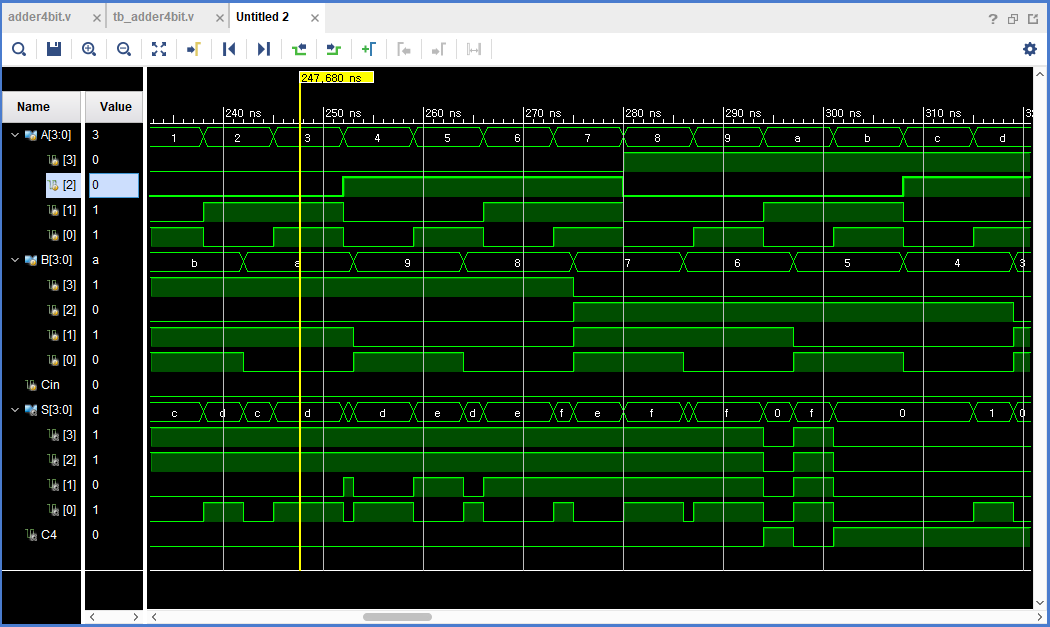
test bench code를 통해 simulation을 진행했다. Non-blocking으로 구현했고, Cin에 값 변화를 주지 않고 0으로 고정하여 덧셈 결과만을 확인했다.

**3) 출력예시 및 Simulation 결과**



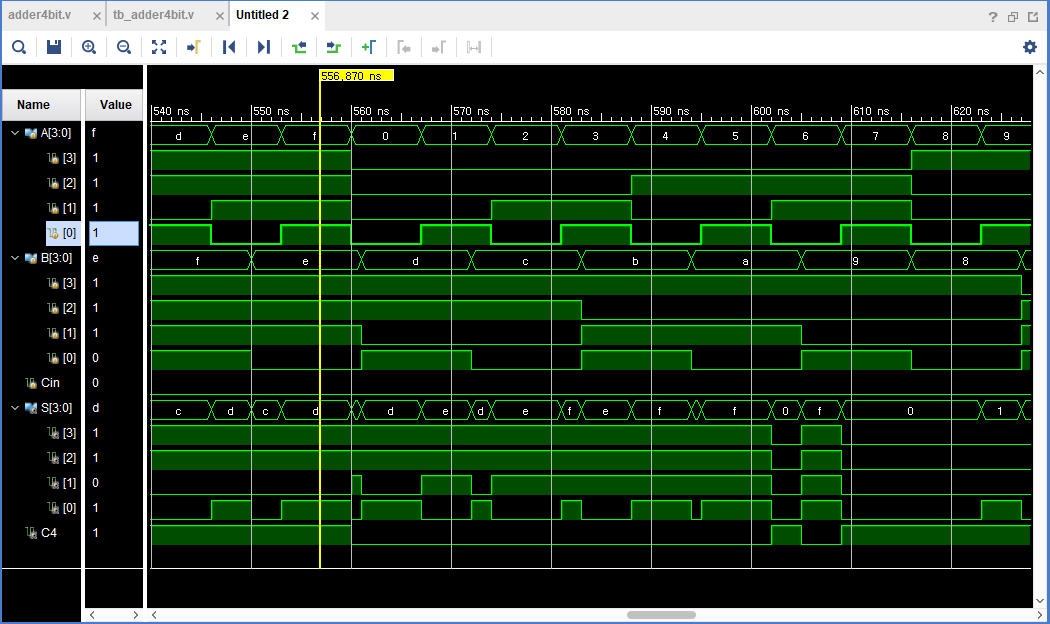
**<Figure 2. 4bit adder Simulation 결과1>**

총 확인 가능한 모든 경우의 수는 256개이고, 육안으로 모든 결과를 관찰하기에는 쉽지 않았다. 따라서 하단의 몇 가지 case만 확인했다.



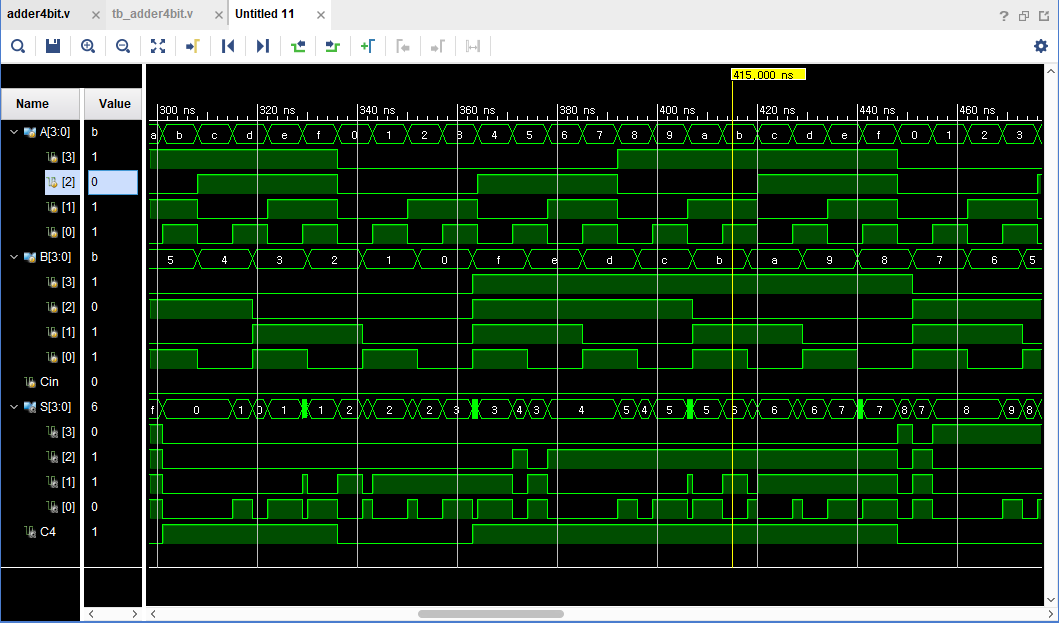
**<Figure 3. 4bit adder Simulation 결과2>**

Fig 1에서 확인해보면, 우선 A bit가 0011, B bit가 1010에 Cin이 0일경우 덧셈을 진행하는 것이다. 따라서 S bit 결과로 1101이 출력됐고, Carry out인 C4는 0으로 잘 출력됐다.



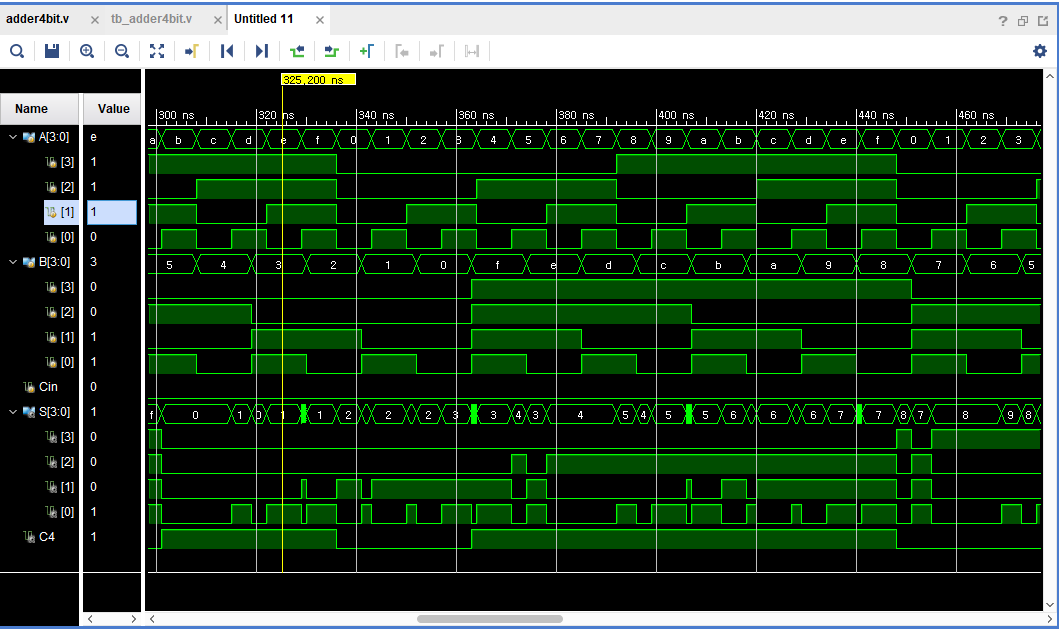
**<Figure 4. 4bit adder simulation 결과3>**

Fig 3에서 확인해보면, 우선 A bit가 1111, B bit가 1110에 Cin이 0일경우 덧셈을 진행하는 것이다. 따라서 S bit 결과로 1101이 출력됐고, Carry out인 C4는 1으로 잘 출력됐다. 즉, 1111 + 1110을 덧셈을 진행하면, 11101이 되고, MSB에 있는 1은 C4로 출력되므로, 나머지 부분은 S bit로 1101로 출력된 것이다.



**<Figure 5. 4bit adder simulation 결과4>**

Fig 4에서 확인해보면, 우선 A bit가 1011, B bit가 1011이다. 따라서 S bit 결과로 0110이 출력됐고, Carry out인 C4는 1으로 잘 출력됐다. 즉, 1011 + 1011을 덧셈을 진행하면, 10110이 되고, MSB에 있는 1은 C4로 출력되므로, 나머지 부분은 S bit로 0110로 출력된 것이다.



**<Figure 6. 4bit adder simulation 결과5>**

Fig 5에서 확인해보면, 우선 A bit가 1110, B bit가 0011이다. 따라서 S bit 결과로 0001이 출력됐고, Carry out인 C4는 1으로 잘 출력됐다. 즉, 1110 + 0011을 덧셈을 진행하면, 10001되고, MSB에 있는 1은 C4로 출력되므로, 나머지 부분은 S bit로 0001로 출력된 것이다.

이로써 Fig2~5까지 4가지 case를 통해 잘 구현된 것을 확인했다.

**2. 4bit Binary Parallel Subtractor의 결과 및 simulation(Verilog source code, 출력 예시, 과정 상세히)**

4bit Binary Parallel Subtractor 또한 1번과 같이 adder 회로를 구현한 뒤 Bin bit를 1로 고정시켜서 subtractor의 역할을 할 수 있도록 하였다. Bin을 1로 고정시켜서 A-B의 뺄셈을 A+B’+1로 계산하여 오로지 뺄셈만을 할 수 있는 subtractor로 구현을 진행했다.

**1) Verilog source code**

`timescale 1ns / 1ps

//9input, 5output

module subtractor4bit(

input [3:0] a,b,

input bin,

output [3:0] d,

output b4

);

wire b1, b2, b3;

assign d[0] = (a[0]^(bin^b[0]))^bin,

b1 = ((a[0]^(b[0]^bin))&bin) | (a[0]&(b[0]^bin)),

d[1] = (a[1]^(bin^b[1]))^b1,

b2 = ((a[1]^(bin^b[1]))&b1) | (a[1]&(b[1]^bin)),

d[2] = (a[2]^(bin^b[2]))^b2,

b3 = ((a[2]^(b[2]^bin))&b2) | (a[2]&(bin^b[2])),

d[3] = (a[3]^(bin^b[3]))^b3,

b4 = ((a[3]^(bin^b[3]))&b3) | (a[3]&(bin^b[3]));

endmodule

배열을 사용해서 input a, b와 d를 input으로 할당했다. 의미있게 남겨져야 하는 값들이거나 결과로 직접 확인해야 할 값이 아닌 것들은 wire를 통해 따로 처리했다.

**2) Test bench code (simulation code)**

`timescale 1ns / 1ps

module tb\_subtractor4bit;

reg[3:0] A, B;

reg Bin;

wire[3:0] D;

wire B4;

subtractor4bit mygate(.a(A), .b(B), .bin(Bin), .d(D), .b4(B4));

initial begin

A = 4'b0000;

B = 4'b0000;

Bin = 1'b1;

end

always @(A or B or Bin) begin

A <= #7 A+4'b0001;

B <= #11 B-4'b0001;

end

initial begin

#1000

$finish;

end

endmodule

test bench code를 통해 simulation을 진행했다. Non-blocking으로 구현했고, Bin에 값 변화를 주지 않고 1로 고정시켜 뺄셈 결과만을 확인했다. 총 확인 가능한 모든 경우의 수는 256개이고, 육안으로 모든 결과를 관찰하기에는 쉽지 않았다. 몇 가지 case만 확인했다.

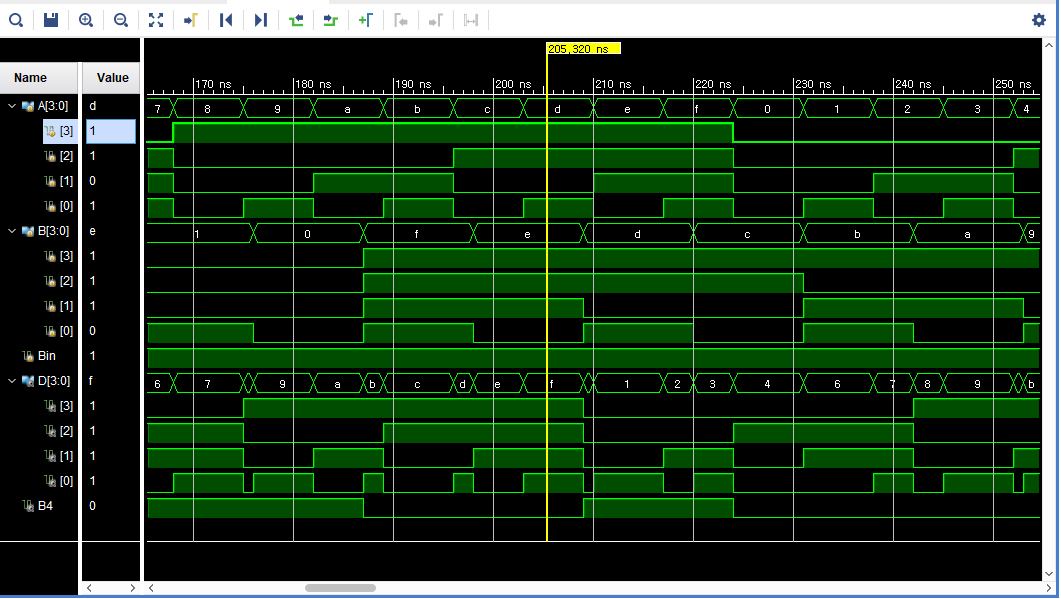
**3) 출력 예시 및 Simulation 결과**

텍스트, 전자기기, 디스플레이이(가) 표시된 사진

자동 생성된 설명

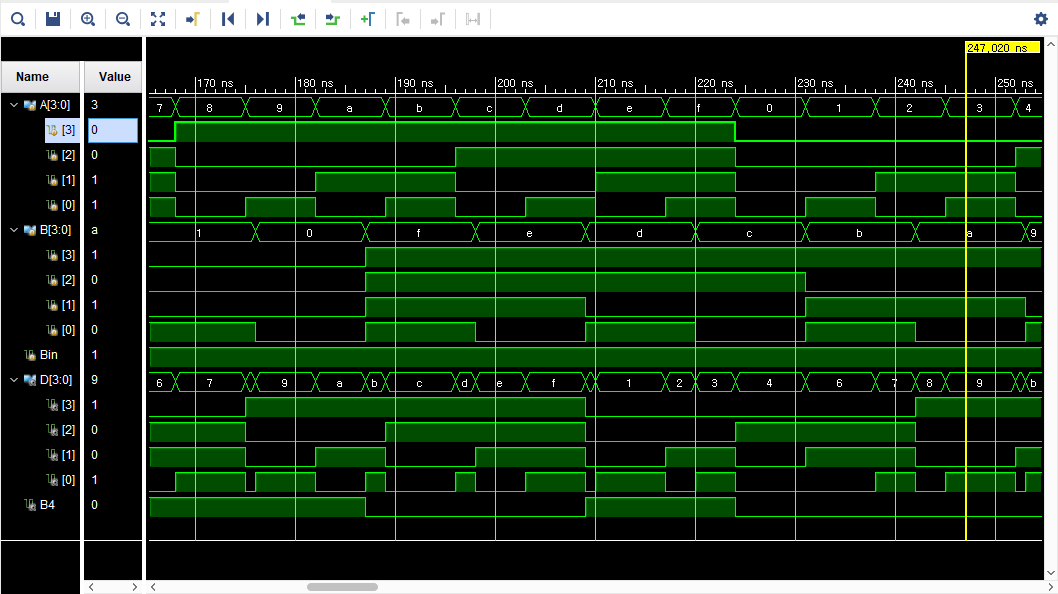
**<Figure 7. 4bit subtractor simulation 결과1>**

총 확인 가능한 모든 경우의 수는 256개이고, 육안으로 모든 결과를 관찰하기에는 쉽지 않았다. 따라서 하단의 몇 가지 case만 확인했다.



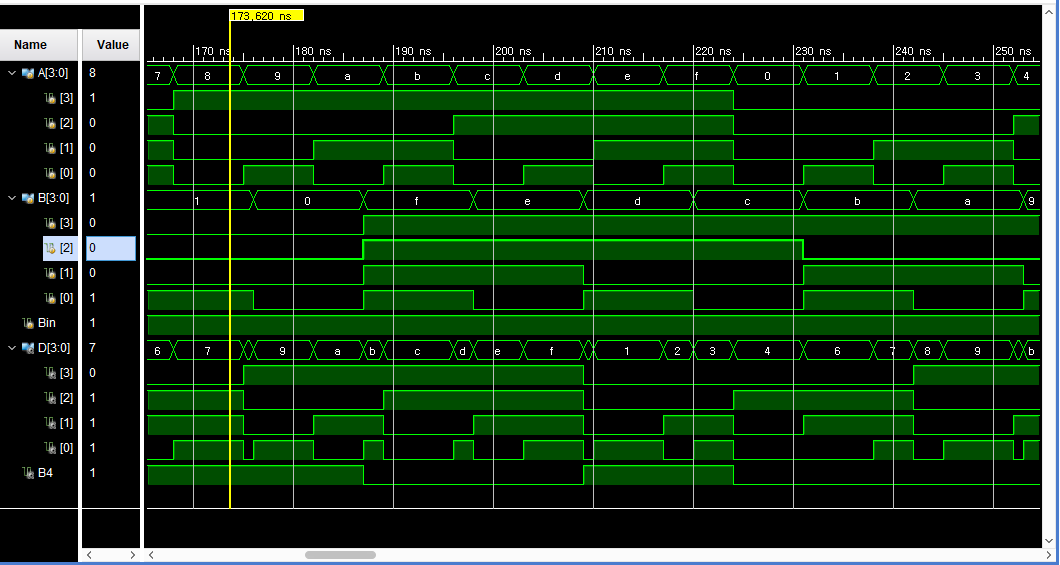
**<Figure 8. 4bit subtractor simulation 결과2>**

Fig 7의 역할을 확인해보면, A bit는 1101, B는 1110으로 1101 + 0001 + 0001이므로, D bit 결과로 1111이 나왔으므로, 정확한 결과값을 얻었다.



**<Figure 9. 4bit subtractor simulation 결과3>**

Fig 8의 결과를 확인해보면, A bit는 0011, B는 1010으로 0011 + 0101 + 0001이므로, D bit 결과로 1001이 나왔으므로, 정확한 결과값을 얻었다.



**<Figure 10. 4bit subtractor simulation 결과4>**

Fig 9를 확인해보면, A bit는 1000, B는 0001으로 1000 + 1110 + 0001이므로, 10001이다. 따라서 B4로 1이 출력되었고, D bit 결과로 1001이 나왔으므로, 정확한 결과값을 얻었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**<Figure 11. 4bit subtractor simulation 결과5>**

Fig 10의 결과를 확인해보면, A bit는 1001, B는 1100으로 1001 + 0011 + 0001이므로, 1101이다. D bit 결과로 1001이 나왔으므로, 정확한 결과값을 얻었다.

**3. BCD adder 및 Simulation 과정 설명** **(verilog source code, 출력 예시, 과정 상세히)**

BCD adder는 Decimal adder라고도 부른다. 2개의 십진수를 받고 carry를 받아서 덧셈을 진행하는 adder이다. 출력으로는 더한 값인 하나의 십진수와 캐리를 출력한다. 즉, 이러한 시스템은 두 일의 자리만을 갖는 십진수와 carry out 총 5개의 output을 가지고 출력하였고, input으로는 carry in과 A, B 4자리씩 하여 9개를 입력으로 받았다. 캐리 발생기준은 합이 9보다 큰 경우 발생시켰고, 우리는 이에 합에 6을 더하여 계산결과를 출력했다. 그 이유는 8421 code에서 사용되지 않는 6개의 bit 조합들을 보충하기 위함이다.

**1) Verilog source code**

`timescale 1ns / 1ps

module bcdadder(

input [3:0] a,b,

input cin,

output [3:0] sum,

output cout

);

wire c1, c2, c3, c4, s0, s1, s2, s3;

assign s0 = a[0]^b[0]^cin,

c1 = (cin&(a[0]^b[0])) | (a[0]&b[0]),

s1 = a[1]^b[1]^c1,

c2 = (c1&(a[1]^b[1])) | (a[1]&b[1]),

s2 = a[2]^b[2]^c2,

c3 = (c2&(a[2]^b[2])) | (a[2]&b[2]),

s3 = a[3]^b[3]^c3,

c4 = (c3&(a[3]^b[3])) | (a[3]&b[3]);

assign cout = (s1 & s3) | (s2 & s3) | c4,

sum[0] = s0,

sum[1] = s1^cout,

sum[2] = s2^cout^(cout&s1),

sum[3] = (((cout&s1) & (cout^s2)) | (cout&s2)) ^ s3;

endmodule

배열을 사용해서 input a, b를 input으로 할당했다. 그리고 배열을 사용해 sum을 output으로 할당했다. 중간에 결과로 출력할 필요 없는 변수(c1~4, s0~3)들은 wire로 처리해주었다. 회로를 확인하여 위와 같이 source code를 작성하였다.

2) Test bench code (simulation code)

`timescale 1ns / 1ps

module tb\_bcdadder;

reg [3:0] A,B;

reg Cin;

wire [3:0] Sum;

wire Cout;

bcdadder mygate(.a(A), .b(B), .cin(Cin), .sum(Sum), .cout(Cout));

initial begin

A = 4'b0000;

B = 4'b0000;

Cin = 1'b0;

end

always @(A or B or Cin) begin

A <= #7 A+4'b0001;

B <= #11 B-4'b0001;

Cin <= #17 ~Cin;

end

initial begin

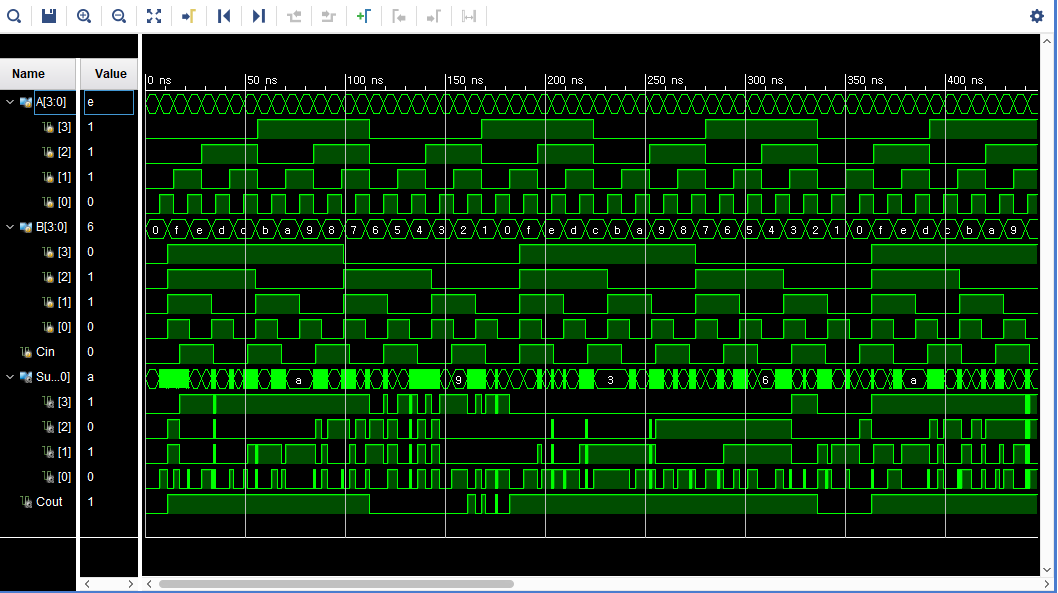
#1000

$finish;

end

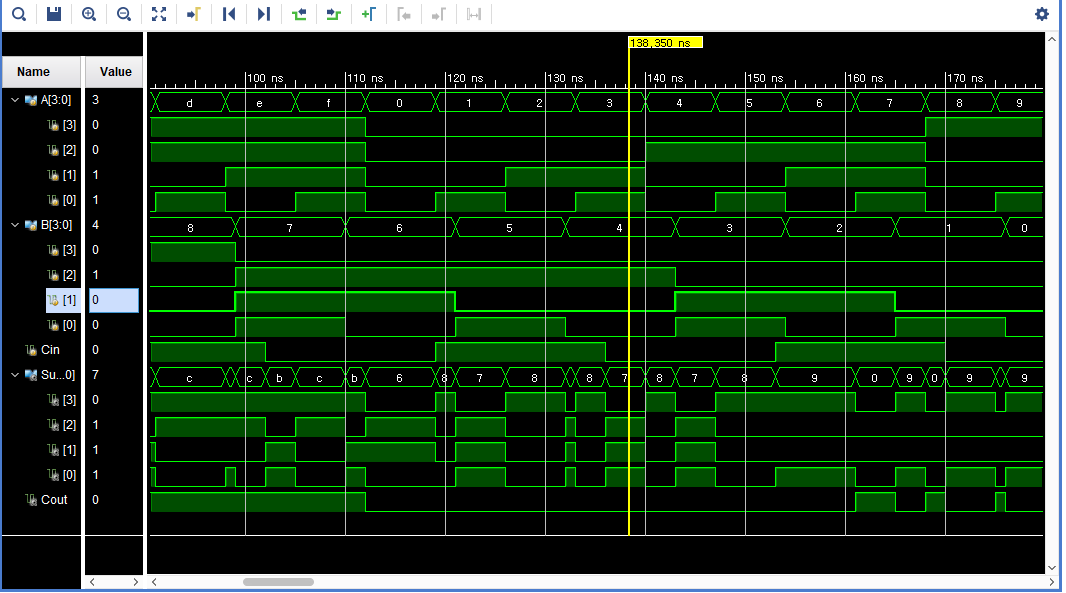
endmodule

**3) 출력 예시 및 simulation 결과**



**<Figure 12. BCD adder simulation 결과1>**

육안으로 모든 결과를 관찰하기에는 쉽지 않았다. 따라서 하단의 몇 가지 case만 확인했다. 이때 Cin이 0인 case만 확인한다. 본 실험은 많은 수의 BCD adder를 연결하여 여러 자리의 십진수를 나타내는 것이 목적이 아니기 때문에 Carry input이 크게 의미를 가지지 않기 때문이다.



**<Figure 13. BCD adder simulation 결과2>**

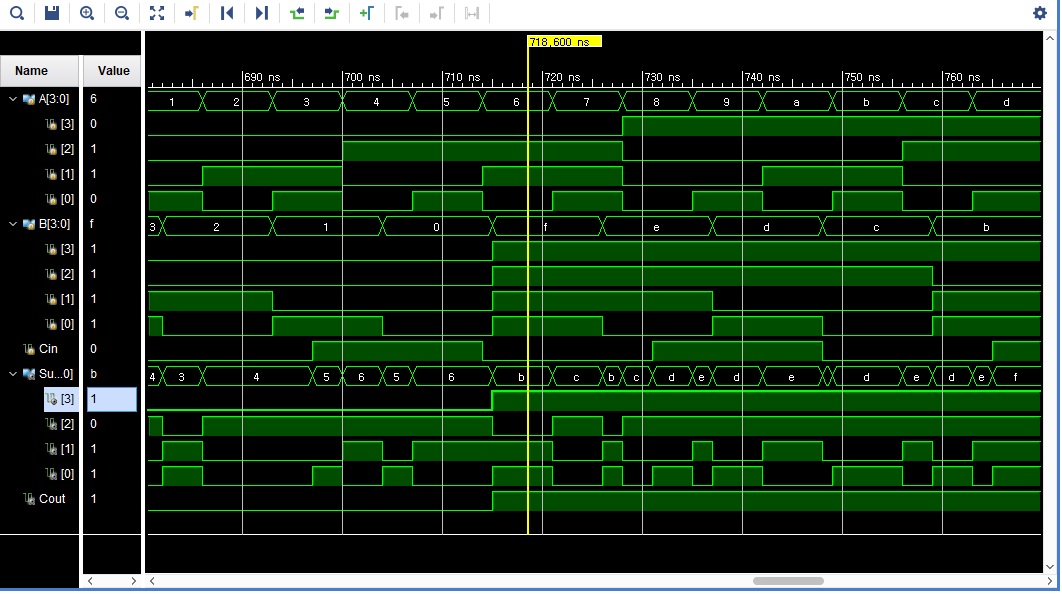
Fig 12의 simulation 결과를 확인해보면, A는 0011를 입력하고, B는 0100을 입력받을 경우 Sum은 0111이나온다. 3+4 = 7로 정확한 결과값이 나왔다.

텍스트이(가) 표시된 사진

자동 생성된 설명

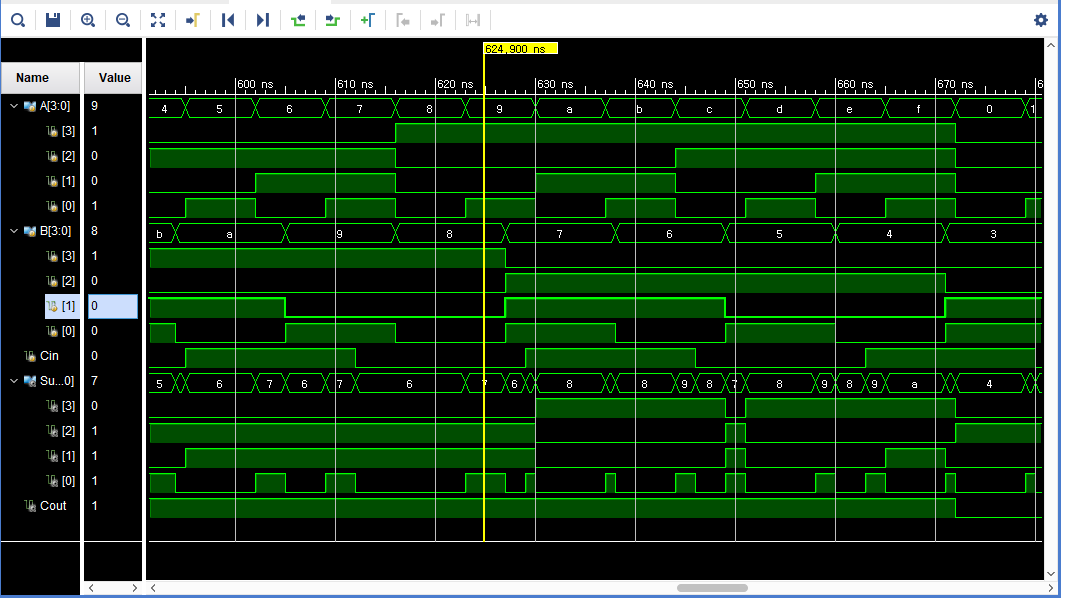
**<Figure 14. BCD adder simulation 결과3>**

Fig 13의 simulation 결과를 확인해보면, A는 0111를 입력하고, B는 1001을 입력 받을 경우 Sum은 0110에 Cout이 1로 출력되어 나왔다. 7+9 = 16로 정확한 결과값이 나왔다.



**<Figure 15. BCD adder simulation 결과4>**

Fig 14의 simulation 결과를 확인해보면, A는 0110를 입력하고, B는 1111을 입력 받을 경우 Sum은 1011에 Cout이 1로 출력되어 나왔다. 6+15 = 21로 정확한 결과값이 나왔다.



**<Figure 16. BCD adder simulation 결과5>**

Fig 15의 simulation 결과를 확인해보면, A는 1001를 입력하고, B는 1000을 입력 받을 경우 Sum은 0111에 Cout이 1로 출력되어 나왔다. 9+8 = 17로 정확한 결과값이 나왔다.

**4. 결과 검토 및 논의 사항**

우선 4-bit Binary Parallel adder는 full adder 4개를 연결하여 구현한 adder였다. Fig 1에서 확인할 수 있듯이 Sign 회로를 추가해서 Sign이 0일 경우에 덧셈을 진행한다. 본 실험에서는 Cin을 이용하여 Sign역할까지 할 수 있도록 하였다. 어차피 Cin과 Sign은 같은 입력으로 들어가기에 큰 상관이 없는 것을 확인했다. simulation결과 Cin이 0으로 고정시킨 후 각각의 A, B input을 더한 결과로 S를 출력했다. 모든 경우의 수에서 잘 출력되는 것이 확인됐다.

4-bit Binary Parallel Subtractor는 위의 4-bit Binary Parallel adder회로에서 Cin 입력 값을 1로 입력하여 구현했다. XOR gate로 1을 입력하여 B의 경우 보수들이 adder에 입력으로 사용되도록 했다. 따라서 2’s complement를 통해 연산이 되도록 했고, 1을 더해주어 뺄셈이 아닌 덧셈으로 진행되도록 구현했다. simulation결과 Cin이 1으로 고정시킨 후 A와 B의 complement에 1을 더한 값을 결과로 S를 출력했다. 모든 경우의 수에서 잘 출력되는 것이 확인됐다.

모두 올바른 결과를 얻었기에 구현이 잘 된 것으로 생각된다. Full adder와 subtractor를 구현하면서 각각 따로 구현하였는데, 단순히 Carry input(Sign)값을 조절해서 한 회로에 모두 구현할 수 있음을 확인했다. XOR gate를 잘 활용하여 Carry input을 0을 입력 받아서 덧셈을 수행하는 adder를 구현할 수 있고, 1을 입력 받으면 뺄셈을 할 수 있도록 구현하는 것이 가능함을 확인했다.

BCD adder는 덧셈한 결과를 8421 code로 표현 불가능한 경우 0110을 더해주어서 사용되지 않는 6개의 bit 조합을 보충해주었다. 따라서 carry output까지 활용해 십진수를 잘 표현할 수 있는 것을 확인했다.

**5. 추가이론 조사 및 작성**

BCD code의 특징

<장점>

- 우리가 사용하는 10진수 시스템과 유사하여 사용하기 편함

- 0-9에 해당하는 binary 값만 알면된다.

- 하드웨어 algorithm 구현이 간단해서 편한 장점이 있다.

- 사용자로부터 입력받고 보여줄 때 효율적이다.

- ASCll code보다 크기가 반이나 작다.

<단점>

- BCD code의 경우 덧셈과 뺄셈 방법이 다르다(0110 더해줌)

- BCD 연산은 ALU가 복잡한 단점이 있다.

- BCD는 10진수를 표현하기 위하여 Binary보다 더 많은 bit를 사용하는 단점이 있다.