컴퓨터공학실험2 10주차 결과 보고서

전공: 컴퓨터공학 학년: 2학년 학번: 20201635 이름: 전찬

**0. 목차**

1. 4bit Binary Parallel Adder의 결과 및 simulation 과정 설명

2. 4bit Binary Parallel Subtractor의 결과 및 simulation 과정 설명

3. BCD Adder의 결과 및 simulation 과정 설명

4. 결과 검토 및 논의 사항

**1. 4bit Binary Parallel Adder의 결과 및 simulation 과정**

4bit Binary Parallel Adder은 이전 실험에서 진행했던 full Adder을 병렬적으로 연결해서, 많은 bit 단위의 덧셈 연산을 수행할 수 있도록 구성된 회로이다. 각 형태에서 회로를 거치며 나온 output을 다음 full Adder의 input Carry로 넣는 형태로 구현된다. 이의 verilog source는 아래와 같다.

module full\_adder(

input A, B, Cin,

output S, Cout

);

assign S = A^B^Cin;

assign Cout = A&B | A&Cin | B&Cin;

endmodule

module parallel\_adder(

input A4, A3, A2, A1, B4, B3, B2, B1,

output S4, S3, S2, S1, Cout

);

reg Cin;

initial Cin = 0;

wire C1, C2, C3;

full\_adder adder1(.A(A1), .B(B1), .Cin(Cin), .S(S1), .Cout(C1));

full\_adder adder2(.A(A2), .B(B2), .Cin(C1), .S(S2), .Cout(C2));

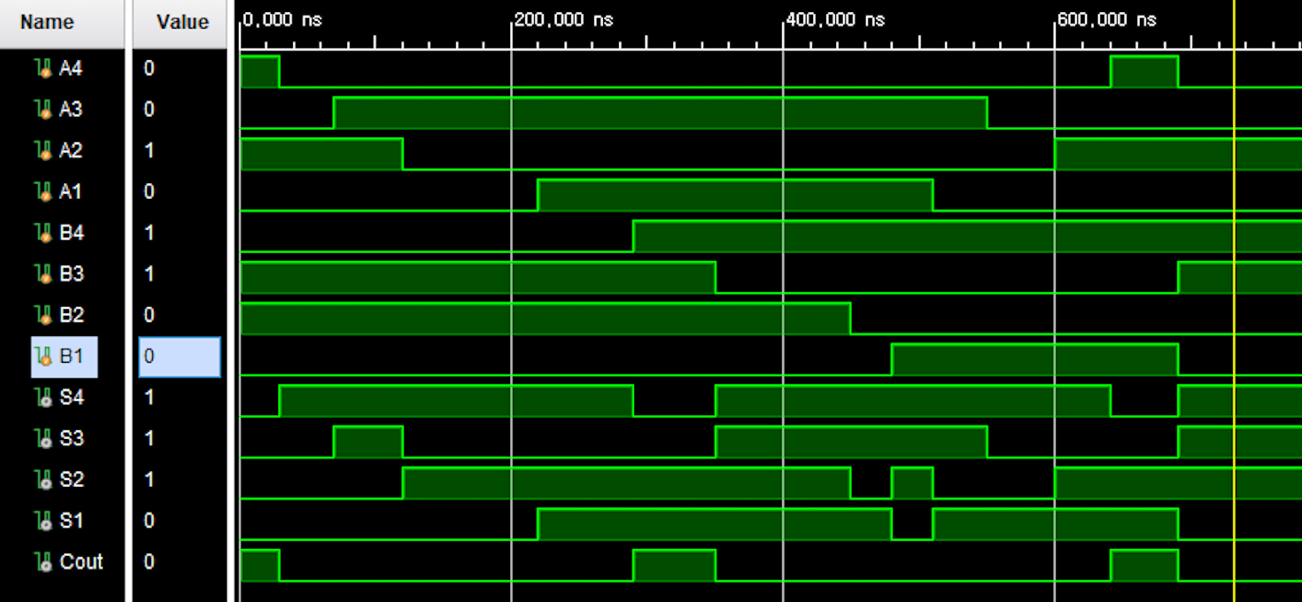
full\_adder adder3(.A(A3), .B(B3), .Cin(C2), .S(S3), .Cout(C3));

full\_adder adder4(.A(A4), .B(B4), .Cin(C3), .S(S4), .Cout(Cout));

endmodule

<4bit Binary Parallel adder의 Verilog 코드>

코드를 통해 4bit Parallel adder에는 4개의 full adder이 사용된다는 것을 알 수 있다. 위 코드에서 adder1의 output인 C1이 adder2의 input으로 들어가며, 이 형태가 반복되어 Parallel adder이 구현된다. 또한 이번 실습에서는 4bit Binary Parallel adder에서 input Cin이 사용되지 않아서 위의 변수 reg Cin, initial Cin = 0; 으로 설정하도록 코드를 작성했다. 하지만 실제로 사용되는 회로 74HC283N을 보면 input에 Cin이 존재하는 것을 알 수 있다. Cin이 존재하는 이유는, 4bit Parallel adder을 구현할 때 4개의 full adder을 병렬적으로 사용한 것처럼, 4bit Parallel adder을 병렬적으로 사용해서(이전 Cout을 Cin으로 연결하는 형식으로) 8bit, 12bit, 16bit, …까지 계속해서 확장할 수 있기 때문이다. 이에 대한 simulation 결과는 아래와 같다.



<4bit Binary Parallel adder의 simulation 결과>

**2. 4bit Binary Parallel Subtractor의 결과 및 simulation 과정**

4bit Binary Parallel Subtractor 또한 adder와 동일하게 4개의 full subtractor을 병렬적으로 연결해서 구현한다. 이 또한 위에서 설명한 것처럼 8, 12, 16, … 등의 큰 bit에서의 뺄셈 연산으로 확장하기 위해서 input Brin이 존재한다. 이에 대한 코드는 아래와 같다.

module full\_subtractor(

input A, B, Brin,

output S, Brout

);

assign S = Brin^A^B; // out1은 D의 역할을 수행

assign Brout = ~A&B | Brin&~(A^B); // Br의 역할을 수행

endmodule

module parallel\_subtractor(

input A4, A3, A2, A1, B4, B3, B2, B1,

output S4, S3, S2, S1, Brout

);

reg Brin;

initial Brin = 0;

wire Br1, Br2, Br3;

full\_subtractor sub1(.A(A1), .B(B1), .Brin(Brin), .S(S1), .Brout(Br1));

full\_subtractor sub2(.A(A2), .B(B2), .Brin(Br1), .S(S2), .Brout(Br2));

full\_subtractor sub3(.A(A3), .B(B3), .Brin(Br2), .S(S3), .Brout(Br3));

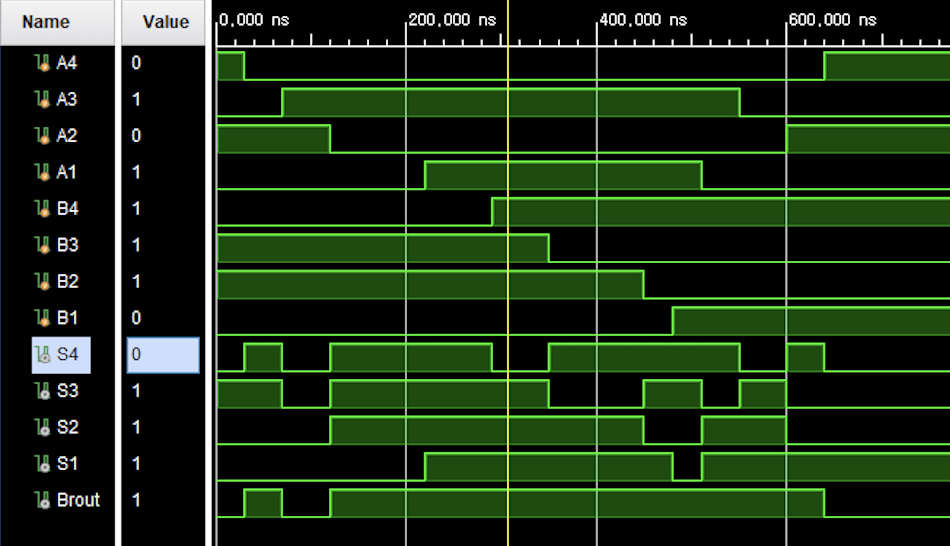
full\_subtractor sub4(.A(A4), .B(B4), .Brin(Br3), .S(S4), .Brout(Brout));

endmodule

<4bit Binary Parallel subtractor의 Verilog 코드>

full subtractor 또한 위에서 설명한 adder과 변수에서 약간 차이가 존재할 뿐 동일한 형식으로 진행된다. sub1의 output Br1이 sub2의 input으로, Br2는 sub3의 input으로, Br3은 sub4의 input으로 연결되며, 마지막 output Brout이 위 순서를 거치며 나온다.

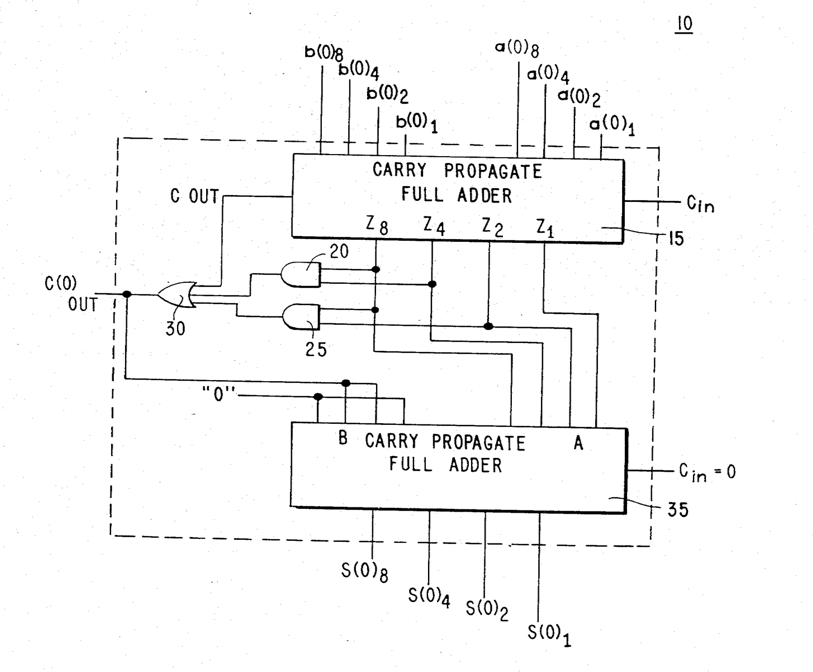
이에 대한 simulation 결과는 아래와 같다.



<4bit Binary Parallel subtractor의 simulation 결과>

**3. BCD Adder의 결과 및 simulation 과정**

10진수의 각 자릿수를 4bit단위로 저장하는 방법인 BCD 또한 덧셈 연산이 가능하다. 예를 들어, 0010 + 0001은 4bit Binary Parallel Adder와 동일하게 output = 0011이 될 것이다. 여기에서 문제는 4bit BCD A, B가 존재할 때, A + B의 결과가 BCD의 4bit 범위를 벗어나는 10 ~ 18일 때 발생한다. 예를 들어 A+B의 결과가 1111인 경우에는, Carry output = 1로 만들어준 이후, S4S3S2S1 = 0101로 만들어주어야 한다. 이 과정을 수행하는 형태는 아래와 같다.



<BCD adder 회로의 형태>1

위의 코드를 살펴보면, 우선 A, B에 대한 Parallel adder로 output C, Z를 만들어낸 것을 볼 수 있다.(위 회로에서는 ripple carry가 아닌 Carry propagate 형태의 덧셈기를 사용했다.) 여기에서 C는 A + B의 연산 결과가 16~18일 때 = 1 이 된다. 하지만 BCD는 4bit 단위로 0~9까지 표현할 수 있으므로, Z에서 표현하고 있는 수가 10~15인지를 확인해야 한다. 이를 확인하는 것이 가운데에 있는 AND gate로, Z4&Z3 , Z4&Z2 연산을 통해서 Z가 10~15인지를 파악해낼 수 있다. 마지막으로 OR gate를 통해서 Carry\_out = C out | Z4&Z3 | Z4&Z2 형태로 구현하면 실제 회로에서의 output인 Carry\_out을 구해낼 수 있다.

이후 과정은 첫 번째 adder에서 도출해낸 Z와 0110 / 0000을 더해주는 것인데, 만약 Carry\_out이 1이라면 0110을 더해주며, 0이라면 0000을 더해주며 S4S3S2S1을 구한다.

이 과정을 수행하는 Verilog 코드는 아래와 같다.

`timescale 1ns / 1ps

module full\_adder(

input A, B, Cin,

output S, Cout

);

assign S = A^B^Cin;

assign Cout = A&B | A&Cin | B&Cin;

endmodule

module parallel\_adder(

input A4, A3, A2, A1, B4, B3, B2, B1,

output S4, S3, S2, S1, Cout

);

reg Cin;

initial Cin = 0;

wire C1, C2, C3;

full\_adder adder1(.A(A1), .B(B1), .Cin(Cin), .S(S1), .Cout(C1));

full\_adder adder2(.A(A2), .B(B2), .Cin(C1), .S(S2), .Cout(C2));

full\_adder adder3(.A(A3), .B(B3), .Cin(C2), .S(S3), .Cout(C3));

full\_adder adder4(.A(A4), .B(B4), .Cin(C3), .S(S4), .Cout(Cout));

endmodule

module BCD\_adder(

input A4, A3, A2, A1, B4, B3, B2, B1,

output S4, S3, S2, S1, Cout

);

reg sub\_input;

initial sub\_input = 0;

wire sum4, sum3, sum2, sum1, carry\_out, sub\_out;

parallel\_adder add1(.A4(A4), .A3(A3), .A2(A2), .A1(A1),

.B4(B4), .B3(B3), .B2(B2), .B1(B1),

.S4(sum4), .S3(sum3), .S2(sum2), .S1(sum1), .Cout(carry\_out)

);

assign Cout = carry\_out | sum4&sum2 | sum4&sum3;

parallel\_adder add2(.A4(sub\_input), .A3(Cout), .A2(Cout), .A1(sub\_input),

.B4(sum4), .B3(sum3), .B2(sum2), .B1(sum1),

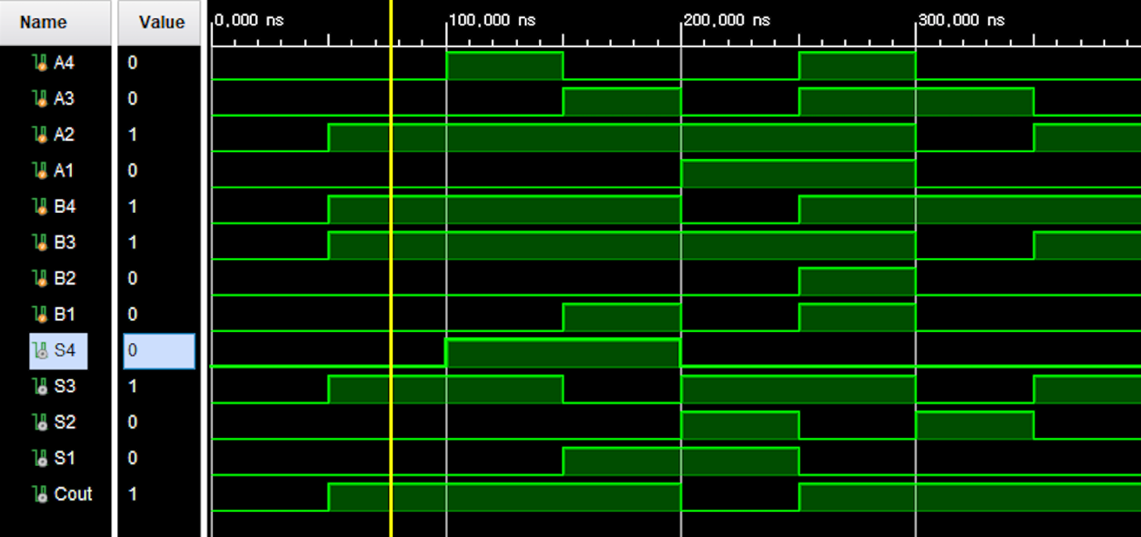
.S4(S4), .S3(S3), .S2(S2), .S1(S1), .Cout(sub\_out)

);

endmodule

<BCD adder의 Verilog 코드>

또한 simulation 결과는 아래와 같다.



<BCD adder의 simulation 결과>

**4. 결과 검토 및 논의 사항**

이번 실험에서는 4bit binary parallel adder/subtractor, BCD Adder을 구현하여 c에서 한 함수에서 다른 함수를 호출하는 것처럼, 한 module에서 다른 module을 내부적으로 사용할 수 있는 형태를 구현했다. parallel adder/subtractor은 full adder/subtractor의 병렬적 호출의 결과로 구성되었으며, BCD Adder은 2개의 4bit binary parallel adder을 사용한 결과로 구현할 수 있었다. 추가로 이전 실습에서 맨 처음 input에 항상 1을 주고 싶은데 어떻게 구현해야 하는지 잘 몰라 구현하지 못했는데, reg A; initial A = 1; 형태로 정의한 이후 A를 사용하면 된다는 것을 파악할 수 있었다.

추가로 이번 실습에서 BCD Adder을 구현할 때, 4bit binary parallel adder의 첫 번째 4bit input을 A4 A3 A2 A1 로 생각했다. 여기에서 BCD\_Adder module의 첫 번째 input 또한 A4 A3 A2 A1이라는 변수를 사용해서 구현했는데, 각 port를 FPGA기기에 맞추어 연결한 이후 수행했을 때는 개인적으로 생각한 결과(BCD adder로 인한 적절한 output)와 동일한 결과가 나왔다. 추가로 Output 또한 겹치는 변수 S4 S3 S2 S1을 사용했는데 생각한 결과가 나왔다. 따라서 동일한 변수를 사용했을 때, module 간에 우선순위가 존재하는지, 또한 존재하면 어떻게 존재하는지 논의해보면 좋을 것 같다.

**5. 출처**

1. BCD adder circuit, scienceon(kisti),

<https://scienceon.kisti.re.kr/srch/selectPORSrchPatent.do?cn=USP1989024805131>