5주차 결과보고서

학번 : 20211531

이름 : 나호영

1.

이번 실험의 목적은 다음과 같다.

1. 드모르간의 제 1법칙, 제 2법칙을 베릴로그로 코드를 작성하여 구현
2. Boolean Function (A’+B’)\*C’, ((A\*B)+C)’를 베릴로그로 코드를 작성하여 구현하고 비교
3. Boolean Function (A’\*B’)+C’, ((A+B)\*C)’를 베릴로그로 코드를 작성하여 구현하고 비교
4. 1Bit 비교기를 베릴로그로 코드를 작성하여 구현  
     
   각 과정을 테이블 벤치를 작성하여 시뮬레이션하고 합성을 거쳐 FPGA상에서 작동하게 한다.

2.

드모르간의 제 1법칙과 제 2법칙의 각각의 2개의 논리식에서 각각 사용되는 변수의 종류가 같으므로 아래와 같은 테이블 벤치 코드를 똑같이 사용하였다.

`timescale 1ns / 1ps

module inv\_tb;

reg A;

reg B;

wire C;

inv u\_inv(

.A ( A ),

.B ( B ),

.C ( C )

);

initial begin

A=1'b1;

B=1'b1;

end

always begin

A=#20 ~A;

B=#30 ~B;

end

initial begin

#1000

$finish;

end

endmodule

아래는 각각 드모르간 제 1법칙의 a와 b의 코드이다.

`timescale 1ns / 1ps

module inv(

input A,

input B,

output C

);

assign C = ~(A||B);

endmodule

`timescale 1ns / 1ps

module inv(

input A,

input B,

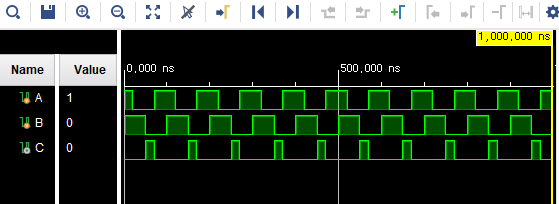
output C

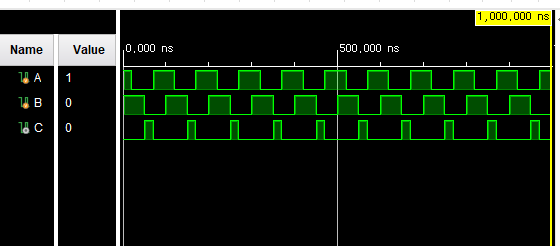
);

assign C = (~A)&&(~B);

endmodule

시뮬레이션 결과는 각각 아래와 같다.





해당 시뮬레이션을 바탕으로 드모르간 제 1법칙의 a와 b는 모든 경우에서 같은 결과값을 도출한다는 것을 알 수 있다.

아래는 각각 드모르간 제 2법칙의 a와 b의 코드이다.

`timescale 1ns / 1ps

module inv(

input A,

input B,

output C

);

assign C = ~(A&&B);

endmodule

`timescale 1ns / 1ps

module inv(

input A,

input B,

output C

);

assign C = (~A)||(~B);

endmodule

시뮬레이션 결과는 각각 아래와 같다.

테이블이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명

해당 시뮬레이션을 바탕으로 드모르간 제 2법칙의 a와 b는 모든 경우에서 같은 결과값을 도출한다는 것을 알 수 있다.

결과를 바탕으로 작성한 진리표이다.

| In A | In B | A+B | A\*B | A’ | B’ | (A+B)’ | A’\*B’ | (A\*B)’ | A’+B’ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

제 1법칙은 NOR 게이트의 결과와 동일하고 제 2법칙은 NAND 게이트의 결과와 동일한 것을 확인해 볼 수 있다.

3.

Boolean Function의 테이블 벤치 코드는 각각의 논리식들을 구성하는 변수들이 동일하기 때문에 똑같은 테이블 벤치 코드를 사용하였다. 해당 테이블 벤치 코드는 아래와 같다.

`timescale 1ns / 1ps

module inv\_tb;

reg A;

reg B;

reg C;

wire D;

inv u\_inv(

.A ( A ),

.B ( B ),

.C ( C ),

.D ( D )

);

initial begin

A=1'b1;

B=1'b1;

C=1'b1;

end

always begin

A=#20 ~A;

B=#30 ~B;

C=#40 ~C;

end

initial begin

#1000

$finish;

end

endmodule

아래는 각각 a = (A’+B’)\*C’, b = ((A\*B)+C)’의 모듈 코드이다. assgin을 이용한 연속 할당으로 output을 정의하였다.

`timescale 1ns / 1ps

module inv(

input A,

input B,

input C,

output D

);

assign D = ( (~A)|(~B) ) & (~C);

endmodule

`timescale 1ns / 1ps

module inv(

input A,

input B,

input C,

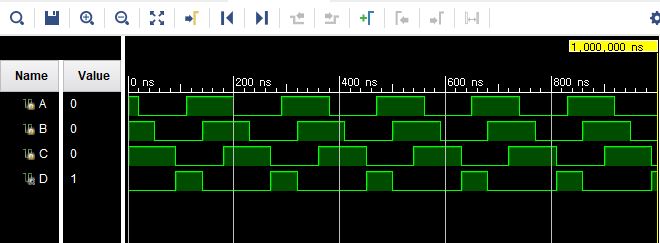
output D

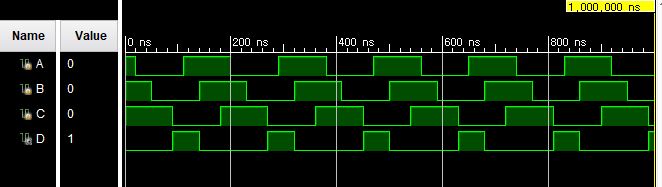
);

assign D = ~( ( (A)&(B) ) | C );

endmodule

그 후, 위의 테이블 벤치 코드를 작성하여 시뮬레이션하였다. 각각의 시뮬레이션 결과는 아래와 같다.





위의 시뮬레이션에서 확인해보면 논리식 a와 b의 시뮬레이션 결과는 동일하다. 그 이유는 드모르간 법칙을 사용하면 a와 b의 논리식은 같은 논리식이기 때문이다.

아래는 실험 결과를 바탕으로 작성한 진리표이다.

| In A | In B | In C | Out D |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

아래는 각각 a = (A’\*B’)+C’, b = ((A+B)\*C)’의 모듈 코드이다. assgin을 이용한 연속 할당으로 output을 정의하였다.

`timescale 1ns / 1ps

module inv(

input A,

input B,

input C,

output D

);

assign D = ( (~A)&(~B) ) | (~C);

endmodule

`timescale 1ns / 1ps

module inv(

input A,

input B,

input C,

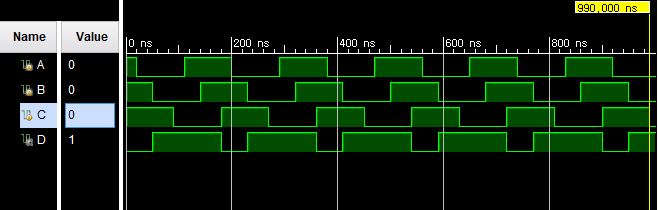
output D

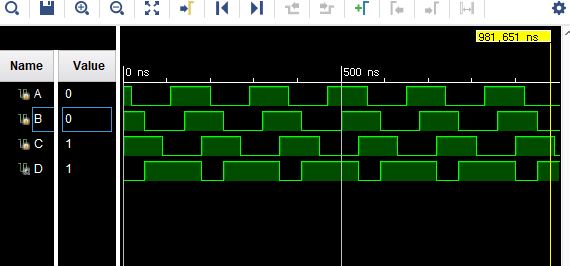
);

assign D = ~( ( (A)|(B) ) & C );

endmodule

그 후, 위의 테이블 벤치 코드를 작성하여 시뮬레이션하였다. 각각의 시뮬레이션 결과는 아래와 같다.





위의 시뮬레이션에서 확인해보면 논리식 a와 b의 시뮬레이션 결과는 동일하다. 그 이유는 위의 경우와 마찬가지로 드모르간 법칙을 사용하면 a와 b의 논리식은 같은 논리식이기 때문이다.

아래는 실험 결과를 바탕으로 작성한 진리표이다.

| In A | In B | In C | Out D |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

4.

아래는 1Bit 비교기의 베릴로그 코드이다. 과정 중에 필요한 ~x와 ~y는 wire으로 코드상에 구현하였고 아웃풋은 assign문을 이용하여 할당하였다.

`timescale 1ns / 1ps

module bit(

input x,

input y,

output E,

output not\_E,

output G,

output L

);

wire not\_x, not\_y;

assign not\_x = ~x;

assign not\_y = ~y;

assign not\_E = x^y;

assign E = ~not\_E;

assign G = x&(~y);

assign L = (~x)&y;

endmodule

아래는 1Bit 비교기의 테이블 벤치 코드이다.

`timescale 1ns / 1ps

module bit\_tb;

reg x;

reg y;

wire E;

wire not\_E;

wire G;

wire L;

bit u\_bit(

.x ( x ),

.y ( y ),

.E ( E ),

.not\_E ( not\_E ),

.G ( G ),

.L ( L )

);

initial begin

x=1'b1;

y=1'b1;

end

always begin

x=#20 ~x;

y=#30 ~y;

end

initial begin

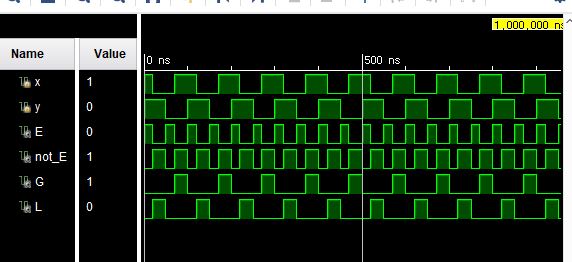
#1000

$finish;

end

endmodule

위의 테이블 벤치 코드를 이용하여 시뮬레이션한 결과는 아래와 같다.



각각의 output은 다음과 같은 결과를 도출한다.

E : x, y가 동일하면 1, 아니면 0

not\_E : x, y가 동일하지 않으면 1, 아니면 0

G : x>y일 시 1, 아니면 0

L : x<y 일 시 1, 아니면 0

결과를 바탕으로 작성한 진리표는 아래와 같다.

| In X | In Y | X=Y | X!=Y | X>Y | X<Y |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

5.

드모르간 제 1법칙과 제 2법칙을 시뮬레이션 결과와 진리표를 봤을 때, 제 1법칙의 논리식 a, b은 동일하고 제 2법칙 a, b도 마찬가지로 동일한 것을 확인해볼 수 있었다. 제 1법칙은 NOR 게이트의 결과와 동일하고 제 2법칙은 NAND 게이트의 결과와 동일하다.

Boolean Function의 논리식인 (A’+B’)\*C’, (A’\*B’)+C’ 각각은 시뮬레이션 결과와 진리표를 바탕으로 A, B, C가 각각001, 110일 때 결과가 다른 것을 알 수 있다.

1Bit 비교기는 총 4개의 output을 가지는데 각각의 논리식을 구현하여 ouput으로 정의해서 시뮬레이션했다. 논의해볼 사항은 ~x, ~y는 output으로 사용되는 것이 아니라 과정 중에 필요한 값인데 베릴로그 코드에서 wire으로 구현하였다. 하지만 wire로 구현한 not\_x와 not\_y는 실제 시뮬레이션을 하기 위해 작성하는 테이블벤치에는 필요없기 때문에 테이블 벤치에 따로 구현하지 않아도 된다는 점을 알게되었다.

6.

++4Bit 비교기

실험 중 구현했던 1Bit 비교기를 확장하여 4Bit 비교기에 대하여 생각해볼 수 있다. 2개의 4Bit 이진수를 비교할 수 있는데 마찬가지로 값이 같은지, 다른지, 큰지, 작은지로 output을 출력한다. 각각의 4Bit 이진수를 a와 b라 하자. A와 b가 최상단 자리수부터 1Bit 비교기를 이용하여 비교한다. 만약 서로 다른 자리수가 있으면 1을 가지고 있는 이진수가 더 큰 이진수이고 두 이진수가 다른 것이라고 결과를 확정지을 수 있다. 끝까지 모두 1Bit 비교기에서 같은 결과가 나온다면 같은 값으로 결과를 확정지을 수 있다. input으로 4Bit 값을 각각 A, B에 넣고 1Bit 비교기를 최상단부터 4개를 연결하고 logic을 구성한다면 4Bit 비교기를 만들 수 있을 것이다.