9주차 결과보고서

전공: 컴퓨터공학 학년: 2학년 학번: 20191629 이름: 이주헌

1. 2 to 4 decoder의 결과 및 simulation 과정에 대해서 설명하시오.

2 to 4 디코더를 구현하려면 먼저 각 입력에 대하여 어떤 출력 핀이 참이 되어야 하는지 구해야 한다. 이를 구하려면 진리표와 카르노 맵을 이용할 수 있다. 먼저 정출력 디코더의 진리표는 다음과 같다.

A	B	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

D0		A	
		1	0
B	0	1	0
	1	0	0

D0 = A’B’

D1		A	
		1	0
B	0	0	0
	1	1	0

D1 = A’B

D2		A	
		1	0
B	0	0	1
	1	0	0

D2 = AB’

D3		A	
		1	0
B	0	0	0
	1	0	1

D3 = AB

다음으로 보수를 출력하는 디코더는 다음과 같이 작성할 수 있다. 이 진리표의 카르노 맵을 최소화할 때는 maxterm을 사용해 최소화해야 한다.

A	B	D0	D1	D2	D3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

D0		A	
		1	0
B	0	0	1
	1	1	1

D0 = A + B = (A'B')'

D1		A	
		1	0
B	0	1	1
	1	0	1

D1 = A + B' = (A'B)'

D2		A	
		1	0
B	0	1	0
	1	1	1

D2 = A' + B = (AB')'

D3		A	
		1	0
B	0	1	1
	1	1	0

D3 = A' + B' = (AB)'

여기서 구한 디코더의 식을 Verilog로 작성하면 다음과 같다.

```
`timesclae 1ns / 1ps

module and_four_to_two_decoder(A, B, D0, D1, D2, D3);
    input A, B;
    output D0, D1, D2, D3;

    assign D0 = ~A & ~B;
```

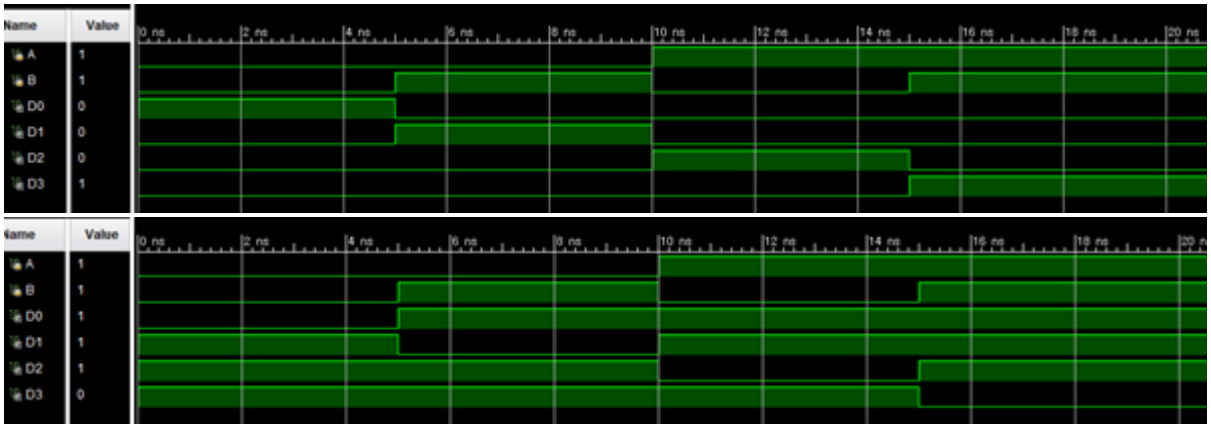
```
    assign D1 = ~A & B;
    assign D2 = A & ~B;
    assign D3 = A & B;
endmodule
```

```
`timesclae 1ns / 1ps

module and_four_to_two_decoder(A, B, D0, D1, D2, D3);
    input A, B;
    output D0, D1, D2, D3;

    assign D0 = ~A & ~B;
    assign D1 = ~A & B;
    assign D2 = A & ~B;
    assign D3 = A & B;
endmodule
```

위 코드를 시뮬레이션하면 다음과 같은 결과를 볼 수 있다.



두 코드의 시뮬레이션 모두 우리가 진리표를 작성한 것과 같은 결과를 낸다는 사실을 알 수 있다. 특히 AND를 사용한 회로는 해당하는 데이터 출력 핀이 1, 그 외는 0이 되므로 active high 회로를 조작할 때유용하게 사용할 수 있고, 반대로 NAND를 사용한 회로는 데이터 출력 핀이 0, 그 외는 1이 되므로 active low 회로를 조작할 때 유용하게 사용할 수 있다.

2. 4 to 2 encoder의 결과 및 simulation 과정에 대해서 설명하시오.

디코더를 설계할 때와마찬가지로 우선 진리표를 그려야 한다. 4 to 2 인코더의 진리표를 작성하면 다음과 같다.

A	B	C	D	E_0	E_1
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

다음 출력에 대해 카르노 맵 2개를 구성하여 E0과 E1에 대한 논리 식을 구할 수 있다.

CD/AB	00	01	11	10
00	D	1	D	1
01	0	D	D	D
11	D	D	D	D
10	0	D	D	D

$E_0=A+B$

CD/AB	00	01	11	10
00	D	0	D	1
01	0	D	D	D
11	D	D	D	D
10	1	D	D	D

$E_1=A+C$

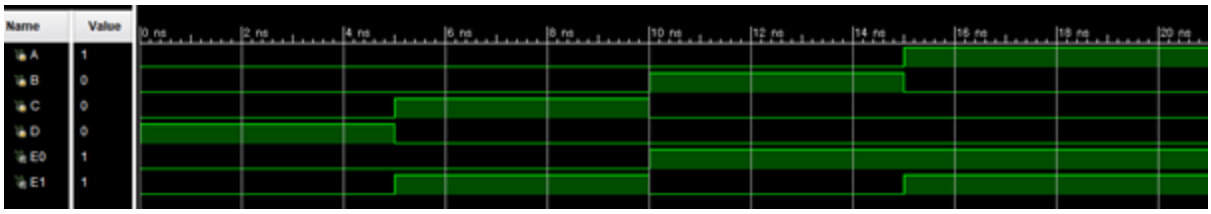
이를 이용해서 Verilog로 2 to 4 encoder를 구현할 수 있다.

```
`timesclae 1ns / 1ps

module two_to_four_encoder(A, B, C, D, E0, E1);
    input A, B, C, D;
    output E0, E1;

    assign E0 = A | B;
    assign E1 = A | C;
endmodule
```

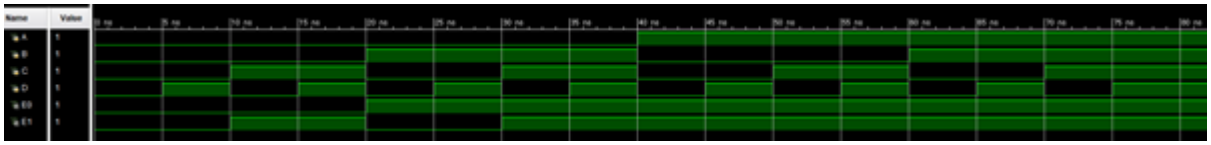
위 코드를 바탕으로 시뮬레이션을 시행하면 다음과 같은 결과를 얻을 수 있다.



이 결과를 방금 전에 구한 진리표와 대조해보면 올바른 결과가 나온다는 사실을 확인할 수 있다.

3. 4 to 2 encoder에서 입력 형태 4가지를 제외한 나머지 입력 형태는 무엇을 뜻하는지 설명하시오.

이전 실험에서는 한 번에 하나의 핀만 논리적 참으로 하는 방식으로 시뮬레이션을 구현하였다. 그런데 만약 모든 16가지 경우의 수에 대하여 출력 형태를 관찰해 보면 다음과 같다.



시뮬레이션 결과에서도 알 수 있듯이, don’t care 비트는 출력에 큰 영향을 주지 못하지만, ABCD 입력이 0110보다 커지게 되면 E0과 E1이 모두 1이 되는 것을 확인할 수 있다. 즉, 1이 두 개 이상 들어오게 되면 실제 입력된 4비트의 입력값과 관계없이 모든 출력 비트가 1이 되는 경우가 생기는 경우가 많다는 점을 알 수 있다. 이는 결과값의 오류로 이어지기 쉽다. 이를 해결하기 위해서는 1) 입력값을 단 하나만으로 한정하는 회로를

추가하거나, 2) 입력에 우선순위를 부여하여 우선순위가 높은 입력이 다른 입력을 무시하도록 설정하는 것이다.

4. 4 to 2 encoder의 4가지 형태가 아닌 모든 입력 형태에 대하여 동작되는 priority encoder의 논리 회로를 구성하여라.

이전 문제에서 알아본 것과 같이, 일반적인 인코더에 하나 이상의 입력값을 넣으면 원하는 출력과 다른 결과가 나타나게 된다. 이를 해결하기 위해 각 입력에 우선순위를 부여하는 priority encoder를 구현해야 한다. 우선순위 인코더의 진리표는 다음과 같다.

A	B	C	D	E_0	E_1
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

해당 식으로부터 E0과 E1의 식을 구하는 카르노 맵은 생략하고, E0과 E1의 식을 구하면 다음과 같이 나타낼 수 있다.

E0 = A + B
E1 = A + B'C

여기서 구한 식을 Verilog로 구현하면 다음과 같다.

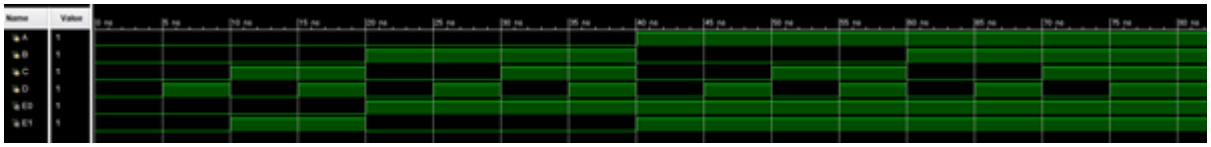
```
`timesclae 1ns / 1ps

module two_to_four_priority_encoder(A, B, C, D, E0, E1);
    input A, B, C, D;
    output E0, E1;

    assign E0 = A | B;
    assign E1 = A | (~B & C);
```

```
endmodule
```

이 코드를 시뮬레이션하면 다음과 같은 결과를 얻을 수 있다.



이 결과와 지난 문제에서의 시뮬레이션 결과를 살펴보면, A에 1이 입력되는 부분부터 출력으로 11이 나오는 것을 볼 수 있다. 따라서, 입력 핀의 우선순위가 잘 적용되었다고 볼 수 있다.

5. BCD to decimal decoder의 결과 및 simulation 과정에 대해서 설명하시오.

BCD 디코더 역시 진리표로부터 각 출력 핀에 대한 논리식을 구해야 한다. Active-low BCD 디코더의 진리표는 다음 사진과 같다.

INPUTS				OUTPUTS									
A_3	A_2	A_1	A_0	$\overline{Y_0}$	$\overline{Y_1}$	$\overline{Y_2}$	$\overline{Y_3}$	$\overline{Y_4}$	$\overline{Y_5}$	$\overline{Y_6}$	$\overline{Y_7}$	$\overline{Y_8}$	$\overline{Y_9}$
L	L	L	L	L	H	H	H	H	H	H	H	H	H
L	L	L	H	H	L	H	H	H	H	H	H	H	H
L	L	H	L	H	H	L	H	H	H	H	H	H	H
L	L	H	H	H	H	H	L	H	H	H	H	H	H
L	H	L	L	H	H	H	H	L	H	H	H	H	H
L	H	L	H	H	H	H	H	H	L	H	H	H	H
L	H	H	L	H	H	H	H	H	H	L	H	H	H
L	H	H	H	H	H	H	H	H	H	H	L	H	H
H	L	L	L	H	H	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L
H	L	H	L	H	H	H	H	H	H	H	H	H	H
H	L	H	H	H	H	H	H	H	H	H	H	H	H
H	H	L	L	H	H	H	H	H	H	H	H	H	H
H	H	L	H	H	H	H	H	H	H	H	H	H	H
H	H	H	L	H	H	H	H	H	H	H	H	H	H
H	H	H	H	H	H	H	H	H	H	H	H	H	H

이 진리표로부터 각 핀에 대하여 카르노 맵을 그리면 다음과 같다. 여기서 수업시간에 언급이 없었던 Y0 핀에 대한 출력은 무시하였다.

A_3A_2/A_1A_0	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

$Y_1=A_0A_1'A_2'A_3'$

A_3A_2/A_1A_0	00	01	11	10
00	0	0	0	1
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

$Y_2=A_0'A_1A_2'A_3'$

A_3A_2/A_1A_0	00	01	11	10
00	0	0	1	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

$Y_3=A_0A_1A_2'A_3'$

A_3A_2/A_1A_0	00	01	11	10

00	0	0	0	0
01	1	0	0	0
11	0	0	0	0
10	0	0	0	0

$Y_4=A_0'A_1'A_2A_3'$

A_3A_2/A_1A_0	00	01	11	10
00	0	0	0	0
01	0	1	0	0
11	0	0	0	0
10	0	0	0	0

$Y_5=A_0A_1'A_2A_3'$

A_3A_2/A_1A_0	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	0
10	0	0	0	0

$Y_6=A_0'A_1A_2A_3'$

A_3A_2/A_1A_0	00	01	11	10
00	0	0	0	0

01	0	0	1	0
11	0	0	0	0
10	0	0	0	0

$Y_7=A_0A_1A_2A_3'$

A_3A_2/A_1A_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	0	0	0

$Y_8=A_0'A_1'A_2'A_3$

A_3A_2/A_1A_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	1	0	0

$Y_9=A_0A_1'A_2'A_3$

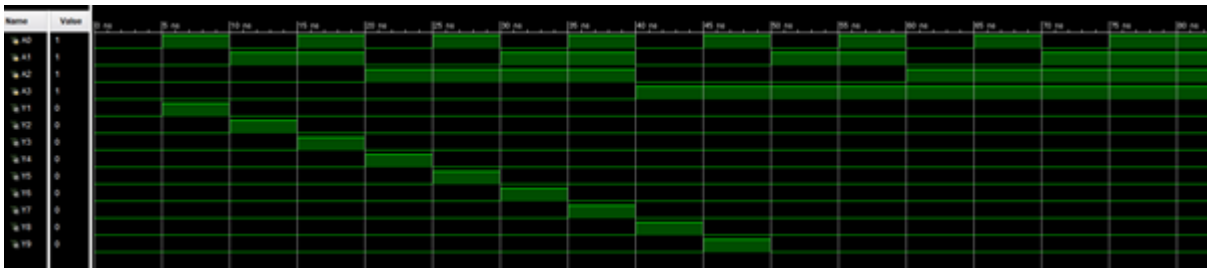
여기서 구한 식을 verilog로 옮기면 다음과 같은 형태로 나타낼 수 있다.

```
`timescale 1ns / 1ps
module bcddecoder(A0,A1,A2,A3,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9);
    input A0,A1,A2,A3;
    output Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9;

    assign Y1=A0&(~A1)&(~A2)&(~A3);
    assign Y2=(~A0)&A1&(~A2)&(~A3);
    assign Y3=A0&A1&(~A2)&(~A3);
    assign Y4=(~A0)&(~A1)&A2&(~A3);
    assign Y5=A0&(~A1)&A2&(~A3);
    assign Y6=(~A0)&A1&A2&(~A3);
    assign Y7=A0&A1&A2&(~A3);
    assign Y8=(~A0)&(~A1)&(~A2)&A3;
    assign Y9=A0&(~A1)&(~A2)&A3;

endmodule
```

그리고 이 코드를 바탕으로 16가지 입력에 대해서 시뮬레이션을 해주면 다음과 같다.



위 시뮬레이션에서 알 수 있듯이 Don’t care 경우를 포함해서 올바른 출력이 나온다는 사실을 알 수 있다.

6. Encoder와 decoder의 주요 응용에 대하여 설명하시오.

Encoder는 전기 신호를 압축하기 위해 주로 사용되고, 뿐만 아니라 정보를 암호화할 때도 사용된다. 반대로 디코더는 암호화된 정보를 복호화하거나, 압축된 정보를 해제하여 원래 정보로 복구할 때 사용된다. 이러한 인코더와 디코더는 많은 정보를 한정된 공간 안에 저장해야 하는 영상이나 사진, 음악과 같은 미디어 관련 작업과, 원래 데이터 그대로 보이면 곤란한 보안 등의 분야에서 활용된다.

7. 8 to 1 line mux의 결과 및 simulation 과정에 대해서 설명하시오.

8 to 1 line mux도 이전 회로와 마찬가지로 진리표를 그려야 한다. 진리표를 그리면 다음과 같이 나타낼 수 있다.

a	b	c	o
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E
1	0	1	F
1	1	0	G
1	1	1	H

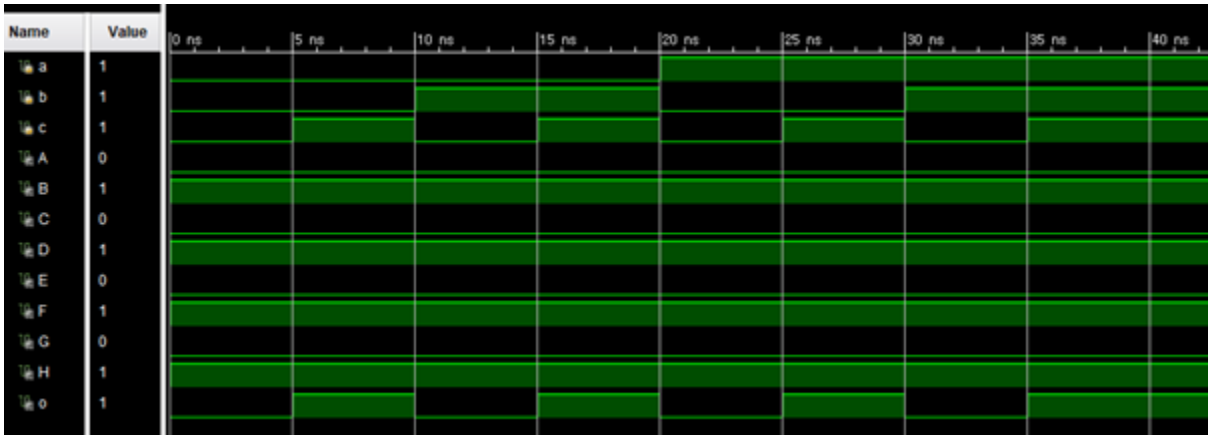
여기에서 A, B, C, D, E, F, G, H는 기본적으로 pass-through되는 입력이다. 따라서, 출력의 결과에 큰 영향을 주지 않으므로 적어 둘 필요는 없다고 판단하였다. 여기서 나온 식을 Verilog로 구현하면 다음과 같은 코드를 얻을 수 있다.

```
`timescale 1ns / 1ps
module MUX(a,b,c,A,B,C,D,E,F,G,H,o);
    input a,b,c;
    output A,B,C,D,E,F,G,H,o;

    assign A=0;
    assign B=1;
    assign C=0;
    assign D=1;
    assign E=0;
    assign F=1;
    assign G=0;
    assign H=1;
    assign o=((~a)&(~b)&(~c)&A)|((~a)&(~b)&c&B)|((~a)&b&(~c)&C)|((~a)&b&c&D)|(a&(~b)&(~c)&E)|(a&(~b)&c&F)|(a&b&(~c)&G)|(a&b&c&H);

endmodule
```

pass-through되는 입력까지 다르게 하여 구현할 필요는 없기 때문에, A부터 H까지의 passthrough 입력은 임의로 01010101으로 설정한 뒤 코드를 작성하였다. 이 코드를 시뮬레이션하면 다음과 같은 결과를 얻을 수 있다.



o 출력이 a, b, c 입력에 따라 01010101으로 바뀌는 것을 알 수 있다. 이는 A, B, C, D, E, F, G, H 입력이 순서대로 passthrough 되는 것으로 이 입력의 값을 바꾸게 되면 시뮬레이션에서도 바뀐 결과가 보이게 된다.

8. 1 to 4 line demux를 이용하여 4 to 16 decoder를 수행하고 결과를 나타내시오.

demux를 사용한 구현체를 만들기 전에 우선 decoder의 진리표를 그려야 한다.

a	b	c	d	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

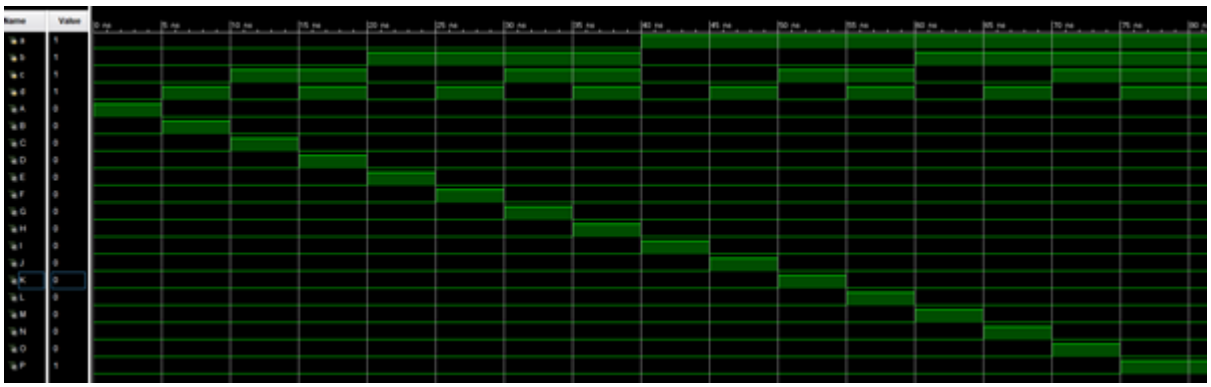
여기에서 4 to 1 demux를 이용하여 하나의 모듈 안에 구현하면 다음과 같은 소스코드를 얻을 수 있다.

```
`timescale 1ns / 1ps
module decoder(a,b,c,d,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P);
    input a,b,c,d;
    output A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P;

    assign A=((~a)&(~b)&1)&(~c)&(~d);
    assign B=((~a)&(~b)&1)&(~c)&d;
    assign C=((~a)&(~b)&1)&c&(~d);
    assign D=((~a)&(~b)&1)&c&d;
    assign E=((~a)&b&1)&(~c)&(~d);
    assign F=((~a)&b&1)&(~c)&d;
    assign G=((~a)&b&1)&c&(~d);
    assign H=((~a)&b&1)&c&d;
    assign I=(a&(~b)&1)&(~c)&(~d);
    assign J=(a&(~b)&1)&(~c)&d;
    assign K=(a&(~b)&1)&c&(~d);
    assign L=(a&(~b)&1)&c&d;
    assign M=(a&b&1)&(~c)&(~d);
    assign N=(a&b&1)&(~c)&d;
    assign O=(a&b&1)&c&(~d);
    assign P=(a&b&1)&c&d;

endmodule
```

여기서 괄호 안에 들어간 내용이 4 to 1 demux의 구현체이다. 이 코드를 시뮬레이션하면 다음과 같은 결과를 얻을 수 있다.



위 결과를 보면 진리표와 같은 결과가 도출된다는 것을 알 수 있다.

9. 결과 검토 및 논의 사항

이번 실습에서는 디코더와 인코더를 구현하였다. 이번에 구현한 디코더는 2비트 이진수 입력에 따라 4개의 서로 다른 핀으로 출력을 분산하는 역할을 맡았고, 인코더는 서로 다른 4개의 핀으로 들어오는 입력을 하나의 2비트 출력으로 압축하는 역할을 하였다. 그러나

여기에서 하나 이상의 핀이 입력되는 경우에 대응하기 위하여 각 입력 핀에 우선순위를 부여한 우선순위 인코더도 구현하였다.
또, BCD to decimal 디코더도 구현하였는데, 각 십진수 값을 10개의 출력으로 분할하는 이 회로는 이전 실습때 진행했던 7-segment display를 구현할 때 유용하게 사용할 수 있다.
마지막으로는 multiplexer와 demultiplexer를 사용했는데, multiplexer는 여러 입력 중 하나를 선택하는 회로이다. 마지막으로 demultiplexer를 이용해서 4 to 16 decoder를 만들어 보면서 회로를 설계하는 능력을 키울 수 있었다.

10. 추가 이론 조사 및 작성

작성한 Verilog 코드를 실제 FPGA에서 실행하기 위해서는 verilog에서 작성한 핀을 실제 FPGA에서 연결하기 위한 방법이 필요하다.
이렇게 실제 FPGA와 verilog에서 정의한 핀 사이의 관계를 설명하는 파일을 constraints 파일이라고 하고, 보통 .xdc 확장자를 가진다.
이 파일은 Tcl이라는 프로그래밍 언어의 코드로 구성되어 있으며, FPGA에 업로드할 비트스트림을 생성할 때 필요한 정보로 구성되어 있다.

FPGA를 연결할 때 핀을 연결하려면 두 개의 커맨드를 입력해야 한다. 먼저 핀이 사용할 인터페이스 방식을 선택하고, Verilog에서 정의한 포트와 실제 포트를 연결하는 커맨드이다.

```
set_property IOSTANDARD LVCMOS18 [get_ports {port}]
set_property PACKAGE_PIN {FPGA pin} [get_ports {port}]
```

여기서 각 입력과 출력 포트에 대하여 이 두 커맨드를 입력해야 한다. 위 커맨드에서 {port}는 Verilog에서 정의한 포트 이름을 입력해야 한다. 또, {FPGA pin}은 FPGA의 핀 이름을 입력해야 하는데, 여기서 주의해야 할 점은 이 핀 이름과 FPGA에 쓰여 있는 핀 이름과 다르다는 점이다. 내부 핀 이름을 입력해야 하기 때문에 FPGA의 데이터시트를 참고해서 올바른 핀 입력할 수 있도록 주의해야 한다.