컴퓨터공학실험 2 9 주차 결과보고서

전공: 컴퓨터공학과 학년: 2 학번: 20191559 이름: 강상원

1. 2 to 4 Decoder의 결과 및 Simulation 과정에 대해서 설명하시오.

2 to 4 Decoder의 두 종류 중 정출력 Decoder의 진리표는 다음과 같다.

A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

위 진리표를 통해 출력값 D0 ~ D3에 대한 카르노 맵을 그릴 수 있다. 각 변수에 대한 카르노 맵은 다음과 같다.

B \ A	0	1
0	1	0
1	0	0

➔ $D_0 = A'B'$로 간소화할 수 있다.

B \ A	0	1
0	0	0
1	1	0

$D_1 = A'$B로 간소화할 수 있다.

B \ A	0	1
0	0	1
1	0	0

➔ $D_2 =$ AB'로 간소화할 수 있다.

B \ A	0	1
0	0	0
1	0	1

$D_3 =$ AB로 간소화할 수 있다.

위 논리식들을 이용해 Verilog 로 나타낼 수 있다. (후략)

보수출력 2 to 4 Decoder의 진리표는 다음과 같다.

A	B	D_0	D_1	D_2	D_3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

위 진리표를 통해 출력값 D0 ~ D3에 대한 카르노 맵을 그릴 수 있다. 각 변수에 대한 카르노 맵은 다음과 같다.

B\A	0	1
0	0	1
1	1	1

➔ $D_0 = \mathrm{A} + \mathrm{B} = (A'B')'$로 간소화할 수 있다.

B\A	0	1
0	1	1
1	0	1

$D_1 = \mathrm{A} + \mathrm{B}' = (A'\mathrm{B})'$로 간소화할 수 있다.

B\A	0	1
0	1	0
1	1	1

➔ $D_2 = \mathrm{A}' + \mathrm{B} = (AB')'$로 간소화할 수 있다.

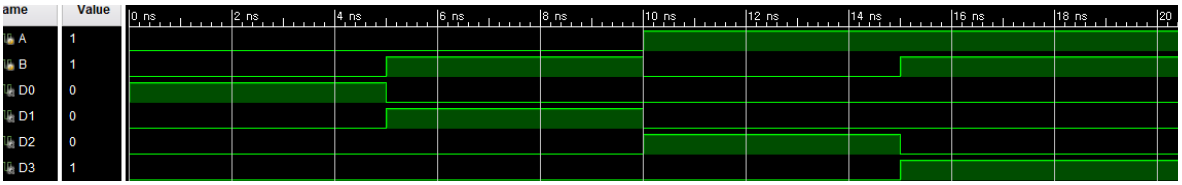
B\A	0	1
0	1	1
1	1	0

$D_3 = \mathrm{A}' + \mathrm{B}' = (\mathrm{AB})'$로 간소화할 수 있다.

이를 Verilog 로 나타낼 수 있다.

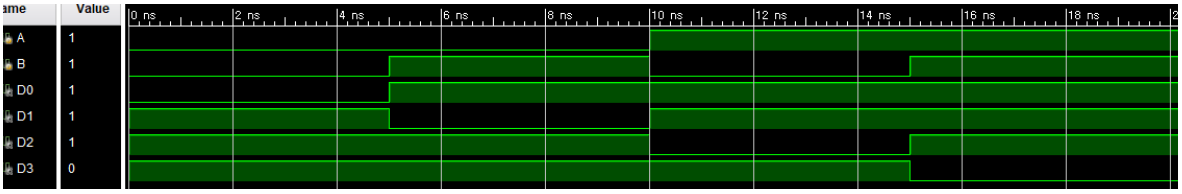
각각의 코드와 시뮬레이션 결과는 다음과 같다.

```
1  `timescale 1ns / 1ps
2
3  module two_to_four_decoder(A, B, D0, D1, D2, D3);
4      input A, B;
5      output D0, D1, D2, D3;
6
7      assign D0 = (~A) & (~B);
8      assign D1 = (~A) & B;
9      assign D2 = A & (~B);
10     assign D3 = A & B;
11
12 endmodule
13
```



AND gate 를 사용한 구조로 코딩하였다.

```
1  `timescale 1ns / 1ps
2
3  module two_to_four_decoder(A, B, D0, D1, D2, D3);
4      input A, B;
5      output D0, D1, D2, D3;
6
7      assign D0 = ~((~A) & (~B));
8      assign D1 = ~((~A) & B);
9      assign D2 = ~(A & (~B));
10     assign D3 = ~(A & B);
11
12 endmodule
13
```



NAND gate를 사용한 구조로 코딩하였다.

정출력, 보수출력 Decoder 모두 진리표와 동일한 결과값을 가짐을 확인할 수 있었다. 앞서 확인하였듯이 Decoder는 정출력, 보수출력 두 가지 형태로 구현할 수 있다. 두 Decoder의 차이점은 한 개만 1이 나오도록 하거나, 한 개만 0이 나오도록 한다는 점이다. 전체적인 기능은 다름이 없다.

2. (4 to 2 Encoder의 결과 및 Simulation 과정에 대해서 설명하시오.
(Truth table 작성 및 k-map 포함)

2 to 4 Encoder의 진리표는 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

위 진리표를 통해 E0 ~ E1에 대한 카르노 맵을 그릴 수 있다. 각 변수에 대한 카르노 맵은 다음과 같다.

CD\AB	00	01	11	10
00	X	1	X	1
01	0	X	X	X

11	X	X	X	X
10	0	D	D	D

➔ $E_0 = A + B$로 간소화할 수 있다.

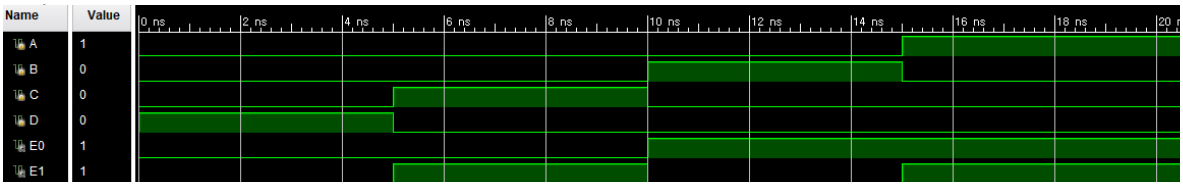
CD\AB	00	01	11	10
00	X	0	X	1
01	0	X	X	X
11	X	X	X	X

➔ $E_1 = A + C$로 간소화할 수 있다.

위 논리식들을 이용해 Verilog 로 나타낼 수 있다.

Verilog 코드와 시뮬레이션 결과는 다음과 같다.

```
1  `timescale 1ns / 1ps
2
3  module four_to_two_encoder(A, B, C, D, E0, E1);
4      input A, B, C, D;
5      output E0, E1;
6
7      assign E0 = A | B;
8      assign E1 = A | C;
9
10 endmodule
11
```



시뮬레이션을 통해 진리표와 동일한 결과값을 가짐을 확인할 수 있었다.

3. BCD to Decimal decoder의 결과 및 Simulation 과정에 대해서 설명하시오.
(Truth table 작성 및 k-map 포함)

BCD to Decimal Deocder의 진리표는 다음과 같다.

	BCD INPUT				DECIMAL OUTPUT									
No.	D	C	B	A	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	1	1
2	0	0	1	0	1	1	0	1	1	1	1	1	1	1
3	0	0	1	1	1	1	1	0	1	1	1	1	1	1
4	0	1	0	0	1	1	1	1	0	1	1	1	1	1
5	0	1	0	1	1	1	1	1	1	0	1	1	1	1
6	0	1	1	0	1	1	1	1	1	1	0	1	1	1
7	0	1	1	1	1	1	1	1	1	1	1	0	1	1
8	1	0	0	0	1	1	1	1	1	1	1	1	0	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	0
	1	0	1	0	1	1	1	1	1	1	1	1	1	1
	1	0	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	0	0	1	1	1	1	1	1	1	1	1	1
INVALID	1	1	0	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	0	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

➔ $Y_1 = A_0A_1'A_2'A_3'$

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	0	1
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

➔ $Y_2 = A_0'A_1A_2'A_3'$

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	1	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

➔ $Y_3 = A_0A_1A_2'A_3'$

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	0	0	0	0
10	0	0	0	0

➔ $Y_4 = A_0'A_1'A_2A_3'$

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	0	0
01	0	1	0	0
11	0	0	0	0
10	0	0	0	0

➔ $Y_5 = A_0A_1'A_2A_3'$

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	0
10	0	0	0	0

➔ $Y_6 = A_0'A_1A_2A_3'$

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	0	0	0
10	0	0	0	0

➔ $Y_7 = A_0A_1A_2A_3'$

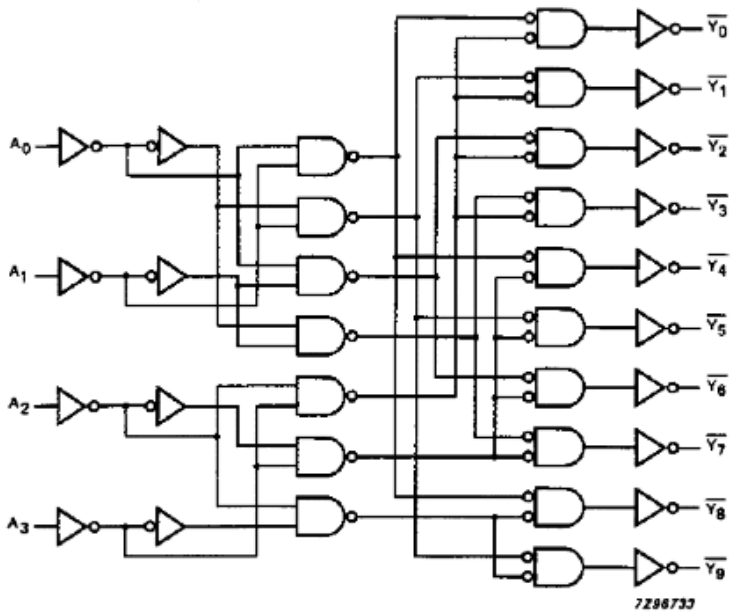
$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	0	0	0

➔ $Y_8 = A_0'A_1'A_2'A_3$

$A_3A_2\backslash A_1A_0$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	1	0	0

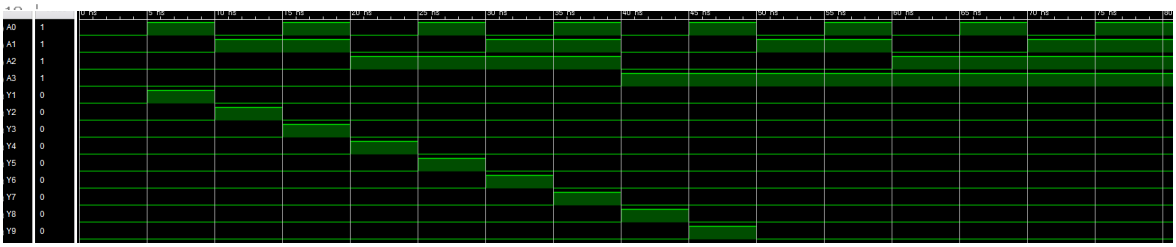
➔ $Y_9 = A_0A_1'A_2'A_3$

gate level로 나타내면 다음과 같이 나타낼 수 있다.



Verilog 코드와 시뮬레이션 결과는 다음과 같다.

```
1  `timescale 1ns / 1ps
2
3  module BCD_to_Decimal_Decoder(A0, A1, A2, A3, Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9);
4      input A0, A1, A2, A3;
5      output Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9;
6
7      assign Y1 = A0 & (~A1) & (~A2) & (~A3);
8      assign Y2 = (~A0) & A1 & (~A2) & (~A3);
9      assign Y3 = A0 & A1 & (~A2) & (~A3);
10     assign Y4 = (~A0) & (~A1) & A2 & (~A3);
11     assign Y5 = A0 & (~A1) & A2 & (~A3);
12     assign Y6 = (~A0) & A1 & A2 & (~A3);
13     assign Y7 = A0 & A1 & A2 & (~A3);
14     assign Y8 = (~A0) & (~A1) & (~A2) & A3;
15     assign Y9 = A0 & (~A1) & (~A2) & A3;
16
17 endmodule
```



시뮬레이션을 통해 진리표와 동일한 결과값을 가짐을 확인할 수 있었다.

4. 8 to 1 line MUX의 결과 및 Simulation 과정에 대해서 설명하시오.

(code, Truth table 작성)

8 to 1 MUX의 진리표는 다음과 같다.

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

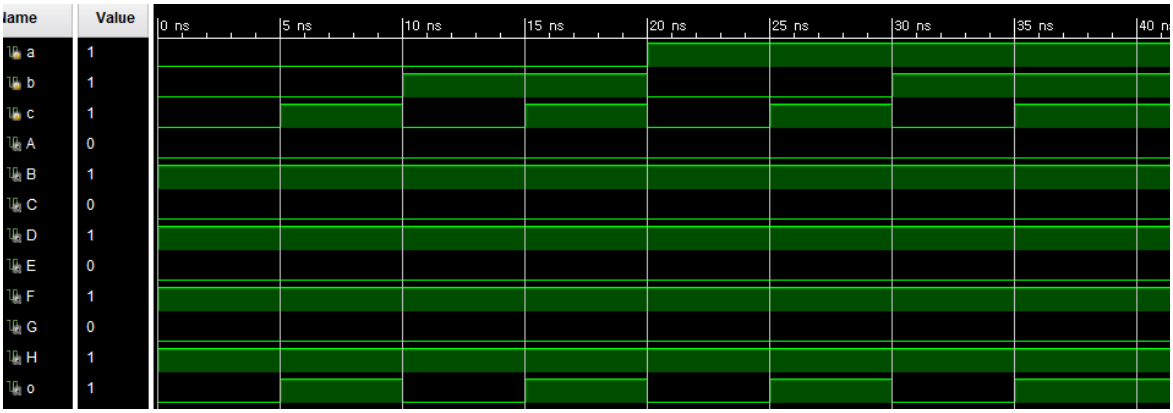
위 진리표를 이용하여 Verilog 코딩을 하면 다음과 같이 나타낼 수 있다.

A ~ H를 미리 01010101로 설정하였다.

코드와 시뮬레이션 결과는 다음과 같다.

```
`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
```



5. 1 to 4 line deMUX를 이용하여 4 to 16 decoder를 수행하고 결과를 나타내시오.

(코드, Truth table 작성)

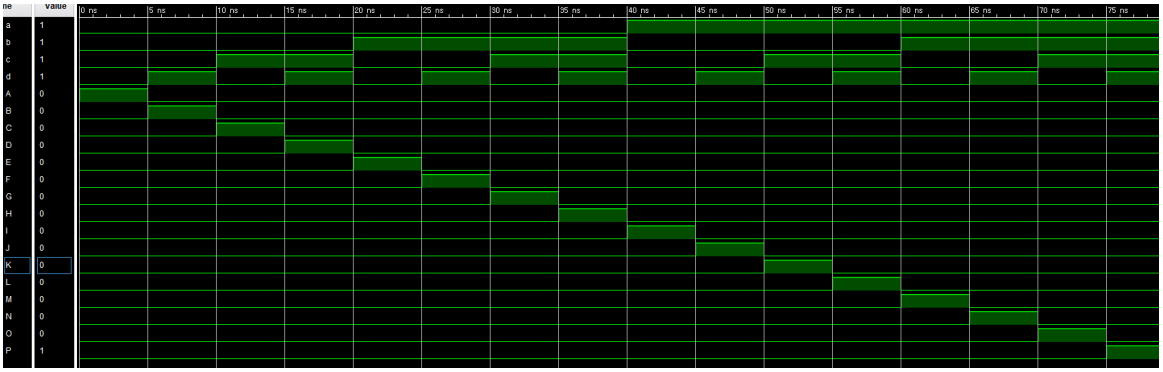
4 to 16 decoder 의 진리표는 다음과 같다.

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

```
`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
```



1 to 4 line deMUX 의 결과를 넣는다. & 0, & 1 은 각각 deMUX 의 입력으로 0, 1 을 대입했다는 의미이다. 총 16 가지 경우의 수를 시뮬레이션을 통해 나타내었다. (상단)