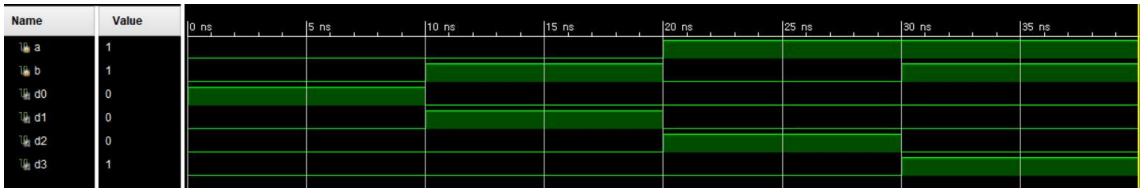
9주차 결과보고서

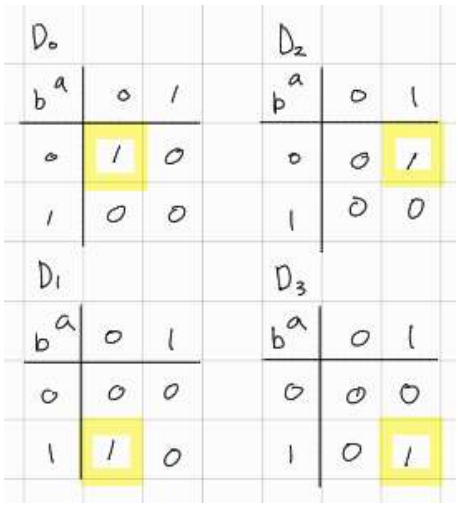
전공: 경영학과 학년: 4학년 학번: 20190808 이름: 방지혁

1.

1) 2-to-4 decoder (Active High)



Input a	Input b	Output d0	Output d1	Output d2	Output 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

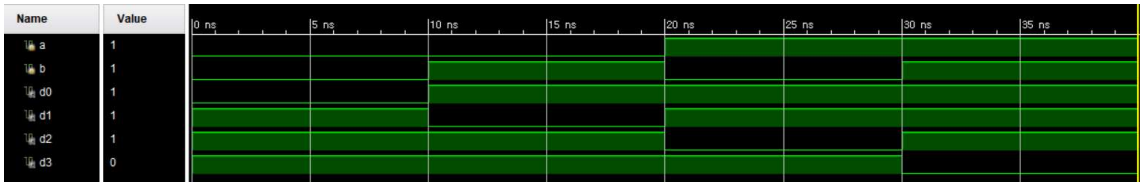



```
`timescale 1ns / 1ps

module decoder(
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
```

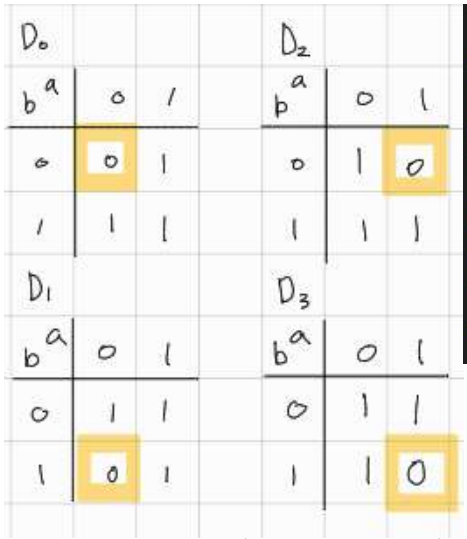
```
`timescale 1ns / 1ps

module decoder_tb;
reg a, b;
wire d0, d1, d2, d3;

decoder test(
.a(a),
.b(b),

.d0(d0),
.d1(d1),
.d2(d2),
.d3(d3)
);
initial begin
a = 1'b0;
b = 1'b0;
#40
$finish;
end

always@(a or b) begin
a <= #20 ~a;
b <= #10 ~b;
end

endmodule
```

2-to-4 decoder가 active high인 경우(단 하나의 출력만
1) 진리표를 바탕으로 카르노맵을 만들어냅니다. and 게이트를 사용 시
d0 = (~a) & (~b)
d1 = (~a) & b
d2 = a & (~b)
d3 = a & b 이렇게 식으로 표현할 수 있습니다.
이렇게 a와 b가 모두 0이면 d0의 출력만 1이고, b만 1이면 d1의 출력만 1이고, a만 1이면 d2의 출력만 1이고, 모두 1이라면 d3의 출력만 1이게 됩니다.

2) 2-to-4 decoder (Active Low)



Input a	Input b	Output d0	Output d1	Output d2	Output 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




```
`timescale 1ns / 1ps

module decoder(
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
```

```
`timescale 1ns / 1ps

module decoder_tb;
reg a, b;
wire d0, d1, d2, d3;

decoder test(
.a(a),
.b(b),

.d0(d0),
.d1(d1),
.d2(d2),
.d3(d3)
);
initial begin
a = 1'b0;
b = 1'b0;
#40
$finish;
end

always@(a or b) begin
a <= #20 ~a;
b <= #10 ~b;
end

endmodule
```

2-to-4 decoder가 active low인 경우 (단 하나의 출력만 0) 진리표를 바탕으로 카르노맵을 만들어냅니다. nand 게이트를 사용 시

d0 = ~((~a)&(~b))

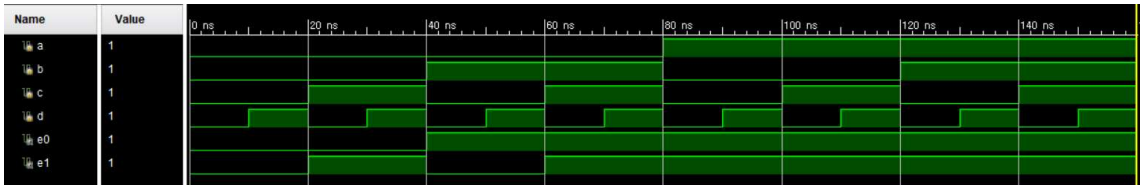
d1 = ~((~a)&b)

d2 = ~(a&(~b))

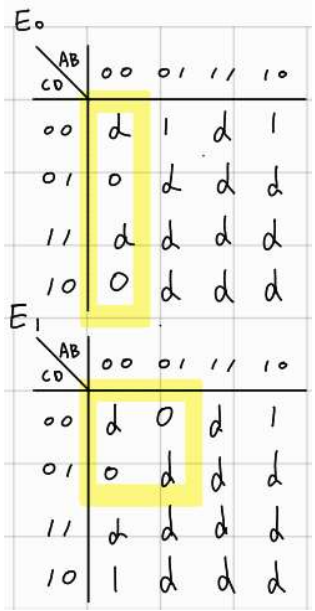
d3 = ~(a&b) 이렇게 식으로 표현할 수 있습니다.

이렇게 a와 b가 모두 0이면 d0의 출력만 0이고, b만 0이면 d1의 출력만 0이고, a만 1이면 d2의 출력만 0이고, 모두 1이라면 d3의 출력만 0이게 됩니다.

2.



Input a	Input b	Input c	Input d	Output e0	Output e1
0	0	0	0	d	d
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	d	d
0	1	0	0	1	0
0	1	0	1	d	d
0	1	1	0	d	d
0	1	1	1	d	d
1	0	0	0	1	1
1	0	0	1	d	d
1	0	1	0	d	d
1	0	1	1	d	d
1	1	0	0	d	d
1	1	0	1	d	d
1	1	1	0	d	d
1	1	1	1	d	d




```
`timescale 1ns / 1ps

module encoder(
  input a, b, c, d,
  output e0, e1
);

assign e0 = a|b;
assign e1 = a|c;

endmodule
```

```
`timescale 1ns / 1ps

module encoder_tb;
reg a, b, c, d;
wire e0, e1;

encoder test(
.a(a),
.b(b),
.c(c),
.d(d),

.e0(e0),
.e1(e1)
);

initial begin
a = 1'b0;
b = 1'b0;
c = 1'b0;
d = 1'b0;
#160
$finish;
end

always@(a or b or c or d)begin
a <= #80 ~a;
b <= #40 ~b;
c <= #20 ~c;
d <= #10 ~d;
end

endmodule
```

해당 encoder는 input인 a, b, c, d 총 4개의 input 중 단 하나의 input만 1일 경우, output인 e0과 e1이 00, 01, 10, 11이 출력되면 됩니다. 나머지 상황에 대해서는 don’t care로 두고 카르노맵 및 간소화된 식을 계산하였습니다. 그렇기에, e0 = a | b, e1 = a | c로 표현할 수 있습니다.

3.

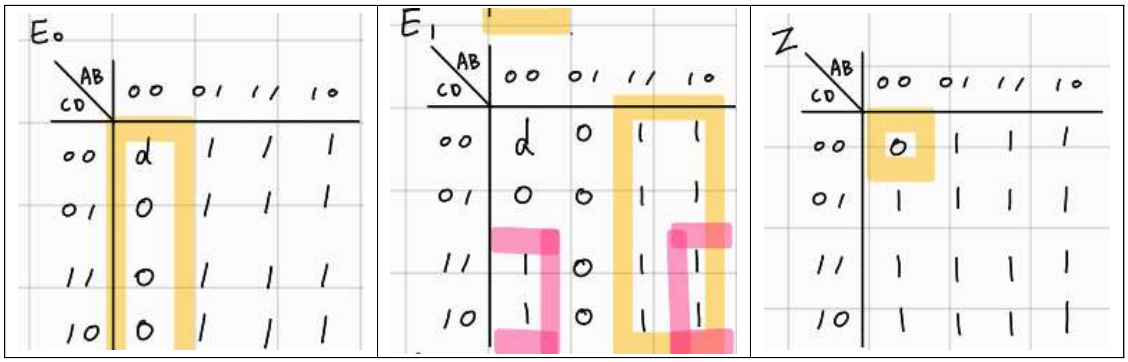
4가지 입력 경우를 제외한 경우에 대해서는 모두 don't care condition이기 때문에 출력값은 상관이 없습니다. 이러한 입력 값들일 경우 encoder를 거치고 나서 데이터를 decoder에게 전송하는 과정에서 생기는 오류라고 판단할 수 있습니다. 이러한 형태의 입력들이 나타났을 때를 대비하여 priority encoder를 사용하여 오류가 발생한 상황에서도 실제로 전달하고자 한 data와 비슷한 결과 값을 얻어낼 수 있습니다.

4.

Input a	Input b	Input c	Input d	Output e0	Output e1	Output z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	0	1	1	0	1	1
0	1	0	0	1	0	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
1	0	1	0	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1

a의 값이 제일 우선시되기 때문에 a의 값이 1이면 출력은 둘 다 1이 나옵니다.
a의 값이 0이라고 가정한다면, b의 값이 제일 우선시되기 때문에 b의 값이 1이면 e0의 출력은 1, e1의 출력은 0이 나옵니다. a와 b 모두 0이라면, c의 값이 제일 우선시되기 때문에 c의 값이 1이면 e0의 출력은 0, e1의 출력은 1이 나옵니다. d의 값만 1이라면 출력은 모두 0이 나옵니다. a, b, c, d 값이 모두 0이라면 output의 유효성을 판단하는 z가 0이 나옵니다.

다음 장에 이어서



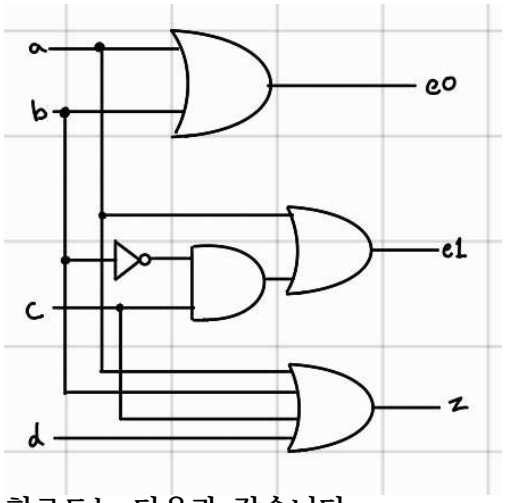
카르노 맵을 통해 간소화된 식을 구했을 때

e0 = a | b

e1 = a | ((~b)&c)

z = a + b + c + d

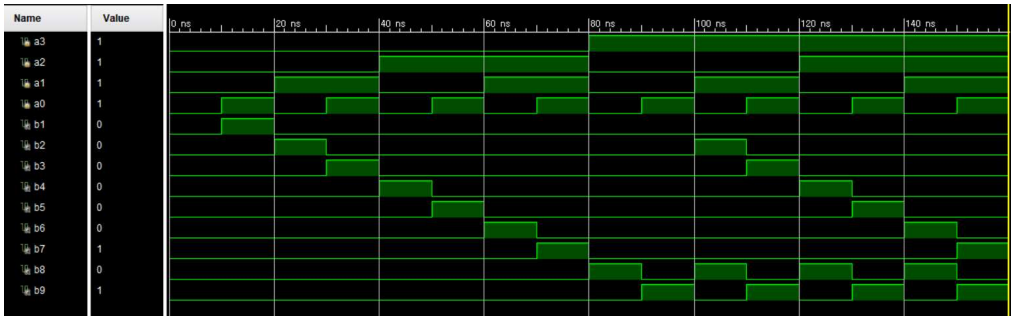
이러한 결과를 도출해낼 수 있습니다.



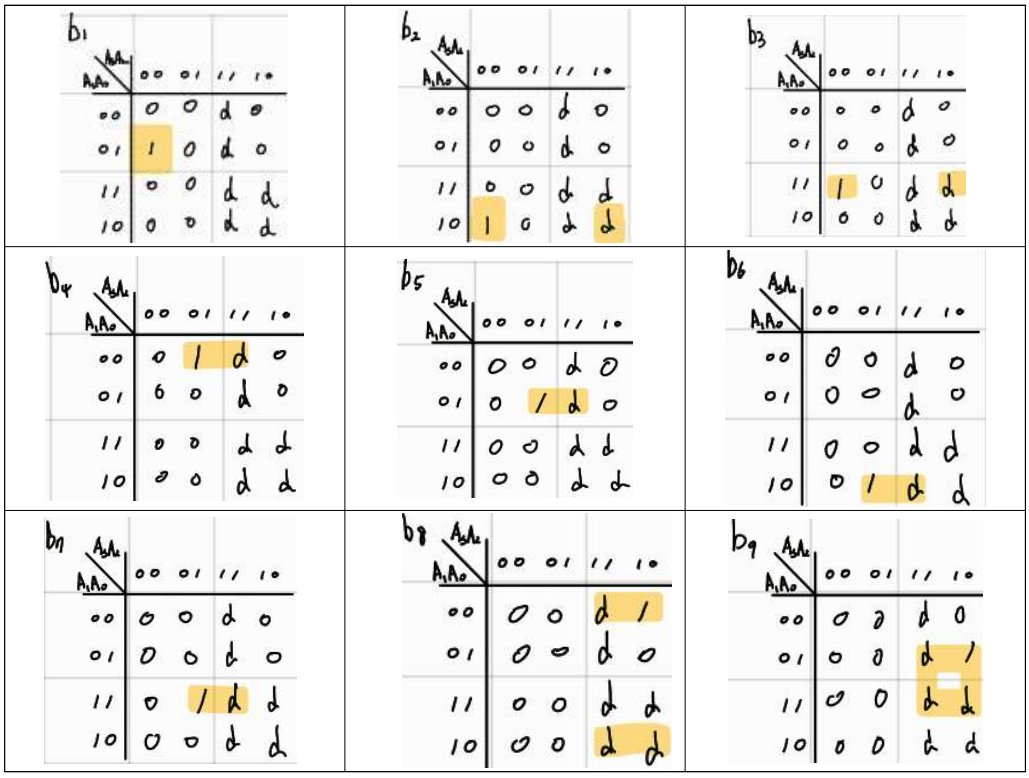
회로도는 다음과 같습니다.

다음 장에 이어서

5.



In a3	In a2	In a1	In a0	Out b1	Out b2	Out b3	Out b4	Out b5	Out b6	Out b7	Out b8	Out b9
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	0	0	1	0	0	0
0	1	1	1	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	1
1	0	1	0	d	d	d	d	d	d	d	d	d
1	0	1	1	d	d	d	d	d	d	d	d	d
1	1	0	0	d	d	d	d	d	d	d	d	d
1	1	0	1	d	d	d	d	d	d	d	d	d
1	1	1	0	d	d	d	d	d	d	d	d	d
1	1	1	1	d	d	d	d	d	d	d	d	d



```
module decoder(
input a3, a2, a1, a0,
output b1, b2, b3, b4, b5, b6, b7, b8, b9
);

assign b1 = ~a3&~a2&~a1&a0;
assign b2 = ~a2&a1&~a0;
assign b3 = ~a2&a1&a0;
assign b4 = a2&~a1&~a0;
assign b5 = a2&~a1&a0;
assign b6 = a2&a1&~a0;
assign b7 = a2&a1&a0;
assign b8 = a3&~a0;
assign b9 = a3&a0;

endmodule
```

```
`timescale 1ns / 1ps

module decoder_tb;
reg a3, a2, a1, a0;
wire b1, b2, b3, b4, b5, b6, b7, b8, b9;
decoder test(
.a0(a0),
.a1(a1),
.a2(a2),
.a3(a3),

.b1(b1),
.b2(b2),
.b3(b3),
.b4(b4),
.b5(b5),
.b6(b6),
.b7(b7),
.b8(b8),
.b9(b9)
);

initial begin
a3 = 1'b0;
a2 = 1'b0;
a1 = 1'b0;
a0 = 1'b0;
#160
$finish;
end

always@(a3 or a2 or a1 or a0)begin
a3 <= #80 ~a3;
a2 <= #40 ~a2;
a1 <= #20 ~a1;
a0 <= #10 ~a0;
end

endmodule
```

카르노 맵에 따라

b1 = ~a3&~a2&~a1&a0

b2 = ~a2&a1&~a0

b3 = ~a2&a1&a0

b4 = a2&~a1&~a0

b5 = a2&~a1&a0

b6 = a2&a1&~a0

b7 = a2&a1&a0

b8 = a3&~a0

b9 = a3&a0

이렇게 간소화된 식으로 표현할 수 있습니다.

정리하자면 출력은 총 9개로, 각 출력은 1에서부터 9까지의 10진수를 의미합니다. a3부터 a0까지 총 16개의 입력이 가능하지만, 9인 1001 이후로는 표시할 필요가 없기 때문에 don't care로 설정해줍니다.

예를 들어 1001이 입력으로 들어간다면 000000001이 나옵니다.

다음장에 이어서

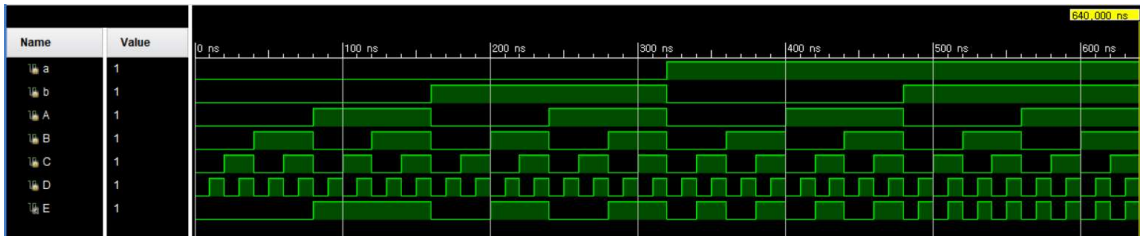
6.

Encoder와 decoder의 주요 응용에 대하여 설명하시오.

우선 encoder는 우리가 흔히 컴퓨터로 영상을 보내기 위해 용량을 줄이기 위한 코덱에서 사용되고, 보안을 강화하기 위해 신호를 암호화하는데 사용합니다. 또한 컴퓨터 내부의 프로세서에서 메모리 위치에 데이터를 전송하는 과정에서 사용됩니다.

반면 decoder는 이렇게 영상이나 소리등 압축된 데이터 또는 암호화된 정보를 이전 상태로 복구하기 위해 사용되기도 하고, 컴퓨터의 RAM에서 특정 위치의 셀을 활성화하는데 사용되기도 합니다.

7.




```
`timescale 1ns / 1ps

module mux(
input a, b, A, B, C, D,
output E
);

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

endmodule
```

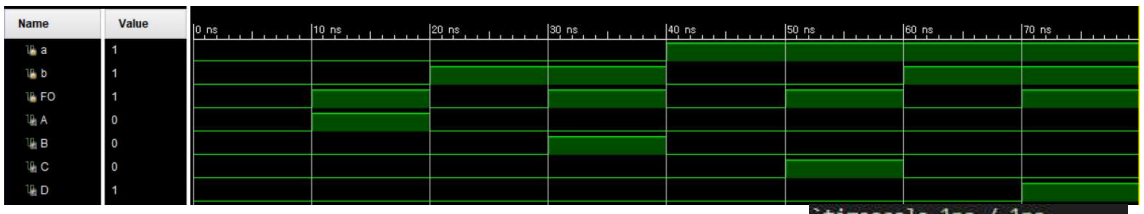
```
`timescale 1ns / 1ps

module mux_tb;
reg a,b,A,B,C,D;
wire E;


mux test(
.a(a),
.b(b),
.A(A),
.B(B),
.C(C),
.D(D),

.E(E)
);

initial begin
a = 1'b0;
b = 1'b0;
A = 1'b0;
B = 1'b0;
C = 1'b0;
D = 1'b0;
#640
$finish;
end

always@(a or b or A or B or C or D)begin
a <= #320 ~a;
b <= #160 ~b;
A <= #80 ~A;
B <= #40 ~B;
C <= #20 ~C;
D <= #10 ~D;
end
endmodule
```

Input a	Input b	Input A	Input B	Input C	Input D	Output E
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	0
0	0	1	0	0	0	1
0	0	1	0	0	1	1
0	0	1	0	1	0	1
0	0	1	0	1	1	1
0	0	1	1	0	0	1
0	0	1	1	0	1	1
0	0	1	1	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	1
0	1	0	1	0	1	1
0	1	0	1	1	0	1
0	1	0	1	1	1	1
0	1	1	0	0	0	0
0	1	1	0	0	1	0
0	1	1	0	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	1
0	1	1	1	0	1	1
0	1	1	1	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	0	1
1	0	0	0	1	1	1
1	0	0	1	0	0	0
1	0	0	1	0	1	0
1	0	0	1	1	0	1
1	0	0	1	1	1	1
1	0	1	0	0	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	1
1	0	1	0	1	1	1
1	0	1	1	0	0	0
1	0	1	1	0	1	0
1	0	1	1	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	0
1	1	0	0	1	1	1
1	1	0	1	0	0	0
1	1	0	1	0	1	1
1	1	0	1	1	0	0
1	1	0	1	1	1	1
1	1	1	0	0	0	0
1	1	1	0	0	1	1
1	1	1	0	1	0	0

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

해당 MUX는 2개의 1bit 크기의 선택 신호인 a, b와 A, B, C, D 4개의 입력, 총 6개의 입력을 가집니다. 4개의 입력 중 하나가 1로 설정되어 있고, 이 신호에 해당하는 선택 신호인 a, b가 올바르게 들어와야 해당 신호가 출력될 수 있습니다.
즉, Output E는 (~a&~b&A)|(~a&b&B)|(a&~b&C)|(a&b&D)입니다. 선택 신호인 a, b가 00일 경우, A에 들어오는 입력값이 출력되고, a, b가 0, 1일 경우에는 B에 들어오는 입력값이 출력되고, a, b가 1, 0일 경우에는 C에 들어오는 입력값이 출력되고, a, b가 1, 1일 경우에는 D에 들어오는 입력값이 출력됩니다.

8.



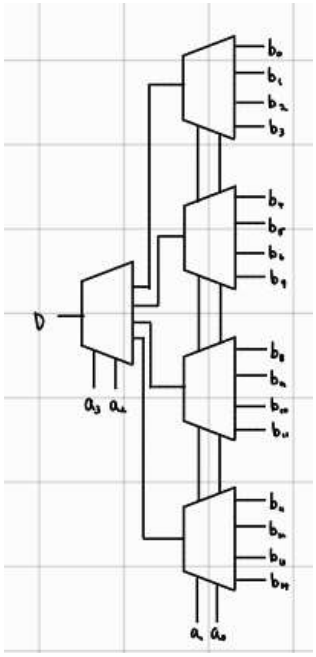

```
`timescale 1ns / 1ps

module demux(
input a, b, FO,
output A, B, C, D
);
assign A = ~a&~b&FO;
assign B = ~a&b&FO;
assign C = a&~b&FO;
assign D = a&b&FO;

endmodule
```

```
`timescale 1ns / 1ps

module demux_tb;
reg a, b, FO;
wire A, B, C, D;

demux test(
.a(a),
.b(b),
.FO(FO),

.A(A),
.B(B),
.C(C),
.D(D)
);

initial begin
a = 1'b0;
b = 1'b0;
FO = 1'b0;
#80
$finish;
end

always@(a or b or FO)begin
a <= #40 ~a;
b <= #20 ~b;
FO <= #10 ~FO;
end

endmodule
```

Input a	Input b	Input FO	Output A	Output B	Output C	Output D
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1

우선 1 to 4 demux에 대하여 설명을 해보겠습니다. FO가 0일 경우 input a. b의 값과 상관 없이 모두 0을 출력합니다. 그러나 FO가 1일 경우에는 a, b가 00일 경우 A에서 1이 출력, 01일 경우 B에서 1이 출력, 10일 경우 C에서 1이 출력, 11일 경우 D에서 1이 출력됩니다. 그렇기에 이 demux를 식으로 나타낸다면 A = ~a&~b&FO, B = ~a&b&FO, C = a&~b&FO, D = a&b&FO 이렇게 가능합니다.

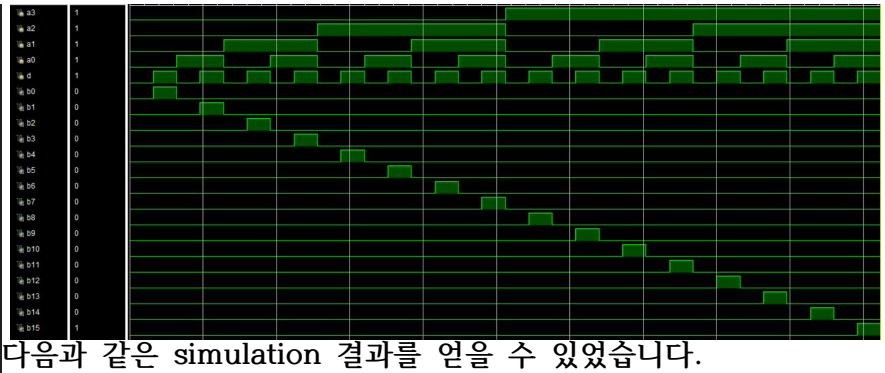
이를 이용해 4-to-16 decoder를 수행해보도록 합니다.



Input a3	Input a2	Input a1	Input a0	Output bx
0	0	0	0	b0가 d
0	0	0	1	b1가 d
0	0	1	0	b2가 d
0	0	1	1	b3가 d
0	1	0	0	b4가 d
0	1	0	1	b5가 d
0	1	1	0	b6가 d
0	1	1	1	b7가 d
1	0	0	0	b8가 d
1	0	0	1	b9가 d
1	0	1	0	b10가 d
1	0	1	1	b11가 d
1	1	0	0	b12가 d
1	1	0	1	b13가 d
1	1	1	0	b14가 d
1	1	1	1	b15가 d

진리표에서 보다시피 a3, a2, a1, a0 값에 따라 b0 ~ b15, 16개의 output 중 d를 내보낼 출력이 결정됩니다. 디자인 소스코드는 다음과 같이 만들었습니다.

```
`timescale 1ns / 1ps
module demux(
input a, b, FO,
output A, B, C, D
);
assign A = ~a&~b&FO;
assign B = ~a&b&FO;
assign C = a&~b&FO;
assign D = a&b&FO;
endmodule
module total(
input a3, a2, a1, a0, d,
output b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b13, b14, b15
);
wire w01, w02, w03, w04;
demux demux01(
  .a(a3),
  .b(a2),
  .FO(d),
  .A(w01),
  .B(w02),
  .C(w03),
  .D(w04)
);
demux demux02(
  .a(a1),
  .b(a0),
  .FO(w01),
  .A(b0),
  .B(b1),
  .C(b2),
  .D(b3)
);
demux demux03(
  .a(a1),
  .b(a0),
  .FO(w02),
  .A(b4),
  .B(b5),
  .C(b6),
  .D(b7)
);
demux demux04(
  .a(a1),
  .b(a0),
  .FO(w03),
  .A(b8),
  .B(b9),
  .C(b10),
  .D(b11)
);
demux demux05(
  .a(a1),
  .b(a0),
  .FO(w04),
  .A(b12),
  .B(b13),
  .C(b14),
  .D(b15)
);
endmodule
```



다음과 같은 simulation 결과를 얻을 수 있었습니다.

9.

2-to-4 decoder, 4-to-2 encoder, BCD-to-Decimal, 4-to-1 MUX, 1-to-4 deMUX, 4-to-16 decoder에 대해 이해하고 이론에 그치지 않고, 직접 verilog 코드로 짜서 시뮬레이션 과정을 거쳤습니다. 또한, 보드에 직접 올려 하드웨어 조작을 통해 구현이 잘 되었는지 확인할 수 있었습니다. 4-to-16 decoder를 1-to-4 demux 5개로 구현하면서 module화에 대한 이해도를 높일 수 있었고, 변수가 여러 개 추가되면서 오타를 내거나 미처 빼먹은 부분들이 증가하여 코드 작성에 어려움을 겪기도 했습니다.

10.

4-to-1 mux와 샤논 확장 이론

샤논의 확장이론은 어떠한 함수에 대하여 한 변수(예를 들어 x)를 기준으로

$f(x,y,z,\cdots) = x' \cdot f(0,y,z,\cdots) + x \cdot f(1,y,z,\cdots)$

위와 같이 분해할 수 있다는 것입니다.

이를 토대로 우리가 이번 주차에서 실험했던 4-to-1 mux도 이러한 방식으로 접근하여 식을 풀어낼 수 있습니다.

$f = a1' \cdot a0' \cdot f00 + a1' \cdot a0 \cdot f01 + a1 \cdot a0' \cdot f10 + a1 \cdot a0 \cdot f11$

TDM

multiplexing의 원리가 들어가는 통신 방식입니다. 여러 user가 하나의 channel을 사용하고 싶을 때, 이를 한 user만 점유하는 불상사를 막기 위해 만들어졌습니다. 이를 통해 bandwidth를 효율적으로 사용할 수 있게 되었습니다. 예를 들자면 각 사용자는 일정 시간의 타임슬롯을 배정받습니다. 그 다음 슬롯에서는 다음 사용자가 신호를 전송할 수 있습니다. 이렇게 수신 측에서도 time synchronization이 잘 이루어진다면 신호를 시간 슬롯별로 분리하여 복원할 수 있습니다. 실제로 2G 통신에서 이러한 방식을 사용했습니다.