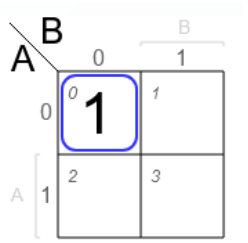
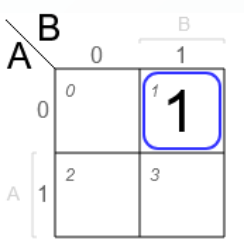
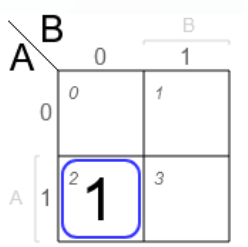
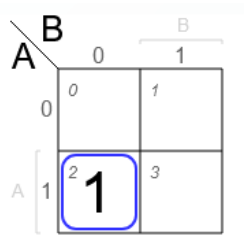
9주차 결과보고서

전공: 신문방송학과 학년: 4학년 학번: 20191150 이름: 전현길

1. 2 to 4 Decoder의 결과 및 Simulation 과정에 대해서 설명하시오. (Truth table 작성 및 k-map 포함, [Active high, Active low 두 경우 모두])

1-a. active high 2-4 decoder

input		output			
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

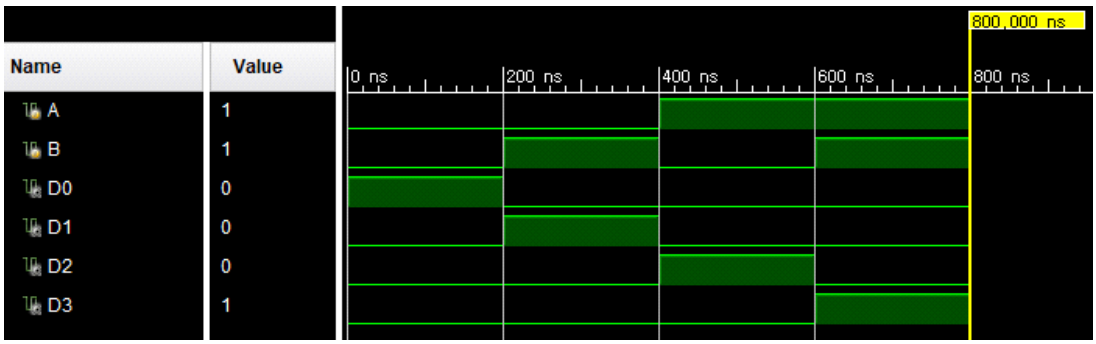
Karnaugh Map			
D_0	D_1	D_2	D_3
			

SOP form (by K-map)	
D_0	A'B'
D_1	A’B
D_2	AB’
D_3	AB

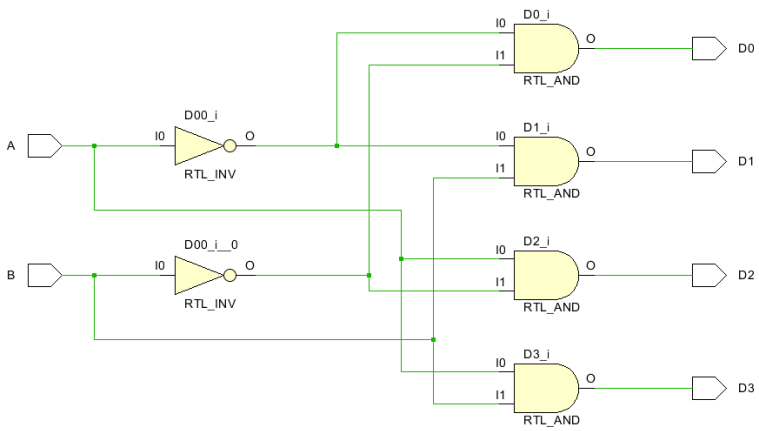
source code	testbench code

`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 u_test (
	  .A (A), .B (B),
	  .D0 (D0), .D1 (D1), .D2 (D2), .D3 (D3)
	);
	
	initial begin
	  A = 1'b0;
	  B = 1'b0;
	end
	
	always@ (A or B) begin
	  A <= #400 ~A;
	  B <= #200 ~B;
	end
	
	initial begin
	  #800
	  $finish;
	end
	
	endmodule

active-high 2 to 4 decoder simulation



active-high 2 to 4 decoder circuit design



active-high 2-4 decoder의 경우 2bit 입력을 받아서 정해진 출력 bit sequence로 복호화하는 작업을 수행한다. active-high encoder의 경우, 입력 bit의 이진수 값에 대응되는 출력 라인을 High 상태로 두고(=1), 그 외의 출력 라인을 Low 상태로 둔다(=0).

simulation을 통해 입력 bit sequence 00, 01, 10, 11에 따라 정상적으로 D_0, D_1, D_2, D_3이 High 상태가 되고, 나머지 출력 라인이 Low 상태가 되므로, 정상적으로 구현되었음을 확인할 수 있다.

1-b. active low 2-4 decoder

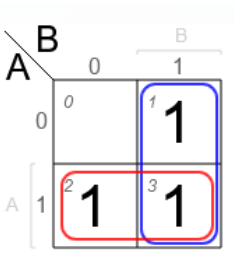
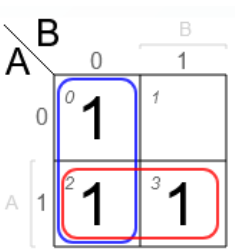
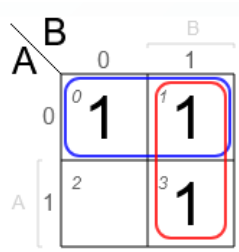
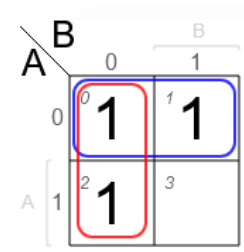
input		output			
A	B	D_0	D_1	D_2	D_3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

source code

```
`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
```

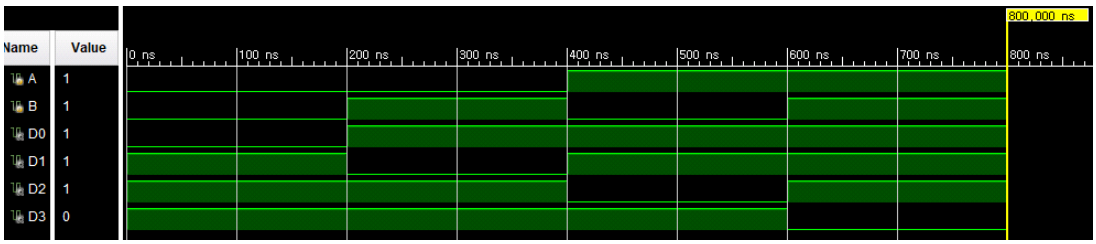
testbench code

```
`timescale 1ns / 1ps

module decoder_tb;

reg A, B;
wire D0, D1, D2, D3;

decoder u_test (
    .A (A), .B (B),
    .D0 (D0), .D1 (D1), .D2 (D2), .D3 (D3)
);

initial begin
    A = 1'b0;
    B = 1'b0;
end

always@ (A or B) begin
    A <= #400 ~A;
    B <= #200 ~B;
end

initial begin
    #800
    $finish;
end

endmodule
```

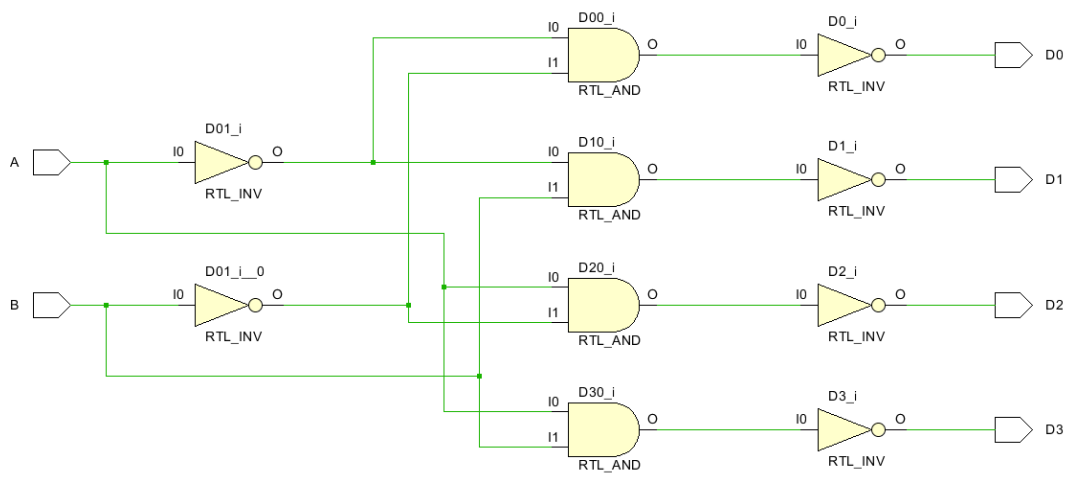
Karnaugh Map			
D_0	D_1	D_2	D_3
			

SOP form (by K-map)	
D_0	A + B
D_1	A + B’
D_2	A’ + B
D_3	A' + B'

active-low 2 to 4 decoder simulation



active-low 2 to 4 decoder circuit design



active-low 2-4 decoder의 경우 2bit 입력을 받아서 정해진 출력 bit sequence로 복호화하는 작업을 수행한다. active-low encoder의 경우, 입력 bit의 이진수 값에 대응되는 출력 라인을 Low 상태로 두고(=0), 그 외의 출력 라인을 High 상태로 둔다(=1).

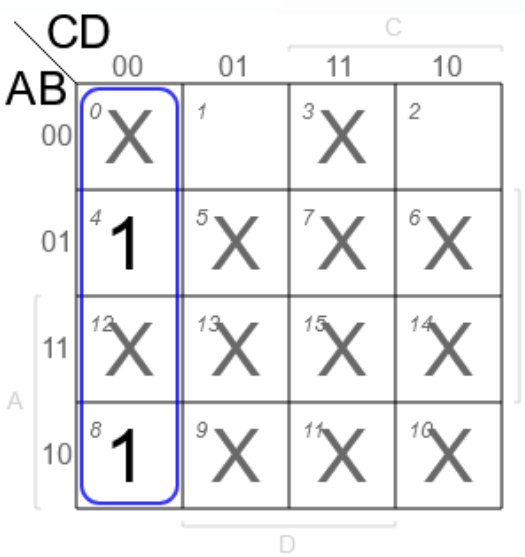
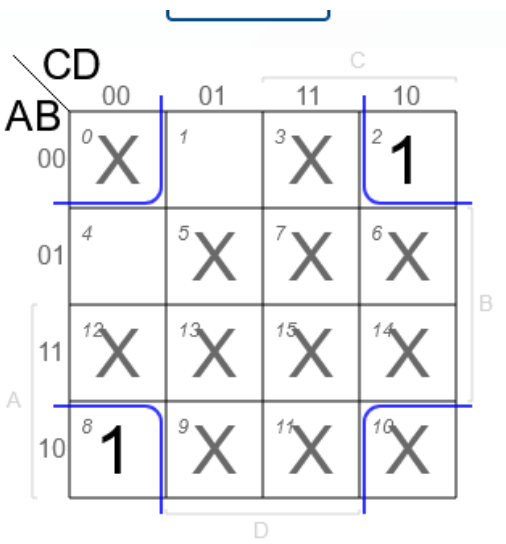
simulation을 통해 입력 bit sequence 00, 01, 10, 11에 따라 정상적으로 D_0, D_1, D_2, D_3이 Low 상태가 되고, 나머지 출력 라인이 High 상태가 되므로, 정상적으로 구현되었음을 확인할 수 있다.

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

input				output	
A	B	C	D	E0	E1
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	X	X
0	1	0	0	1	0
0	1	0	1	X	X
0	1	1	0	X	X
0	1	1	1	X	X
1	0	0	0	1	1
1	0	0	1	X	X
1	0	1	0	X	X
1	0	1	1	X	X
1	1	0	0	X	X
1	1	0	1	X	X
1	1	1	0	X	X
1	1	1	1	X	X

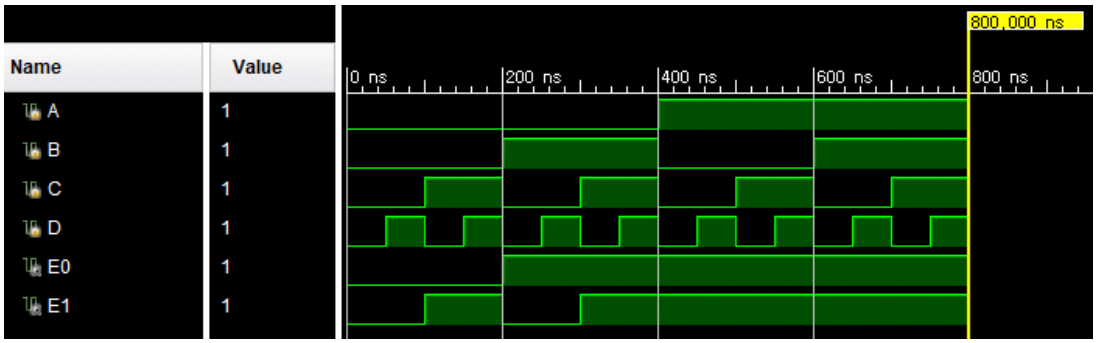
source code	testbench code

`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 u_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;
	end
	
	always@ (A or B or C or D) begin
	  A <= #400 ~A;
	  B <= #200 ~B;
	  C <= #100 ~C;
	  D <= #50 ~D;
	end
	
	initial begin
	  #800
	  $finish;
	end
	
	endmodule

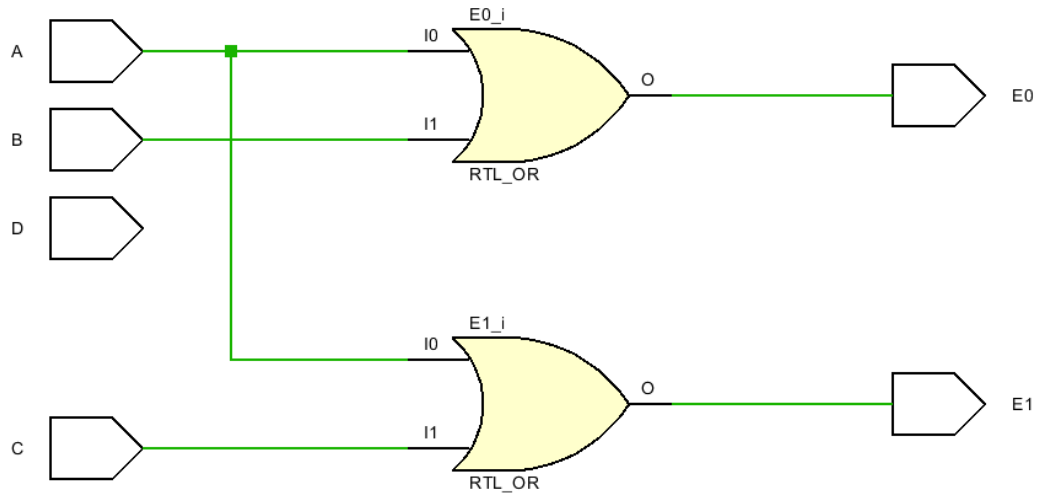
Karnaugh Map	
E_0	E_1
	

SOP form (by K-map)	
E_0	C'D' // A + B
E_1	B'D' // A + C

4 to 2 encoder simulation



4 to 2 encoder circuit design



active-high 4 to 2 encoder의 경우 미리 정해진 0001, 0010, 0100, 1000의 4가지 입력에 대해 00, 01, 10, 11을 출력한다. active-high 2 to 4 decoder와 정확히 정반대의 작업을 수행한다고 볼 수 있다.

simulation을 확인했을 때, 0001, 0010, 0100, 1000이라는 4가지 입력에 대해 00, 01, 10, 11을 출력하고 있으므로, 정상적으로 구현되었음을 확인할 수 있다.

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

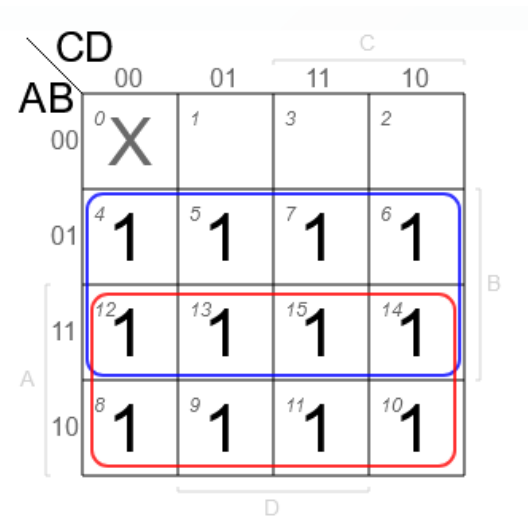
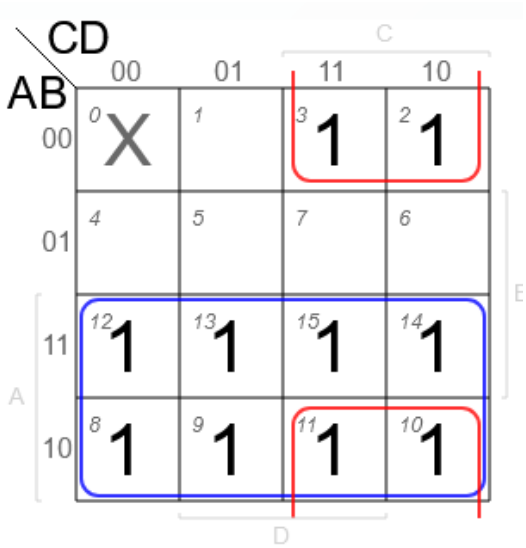
4 to 2 encoder에서 예상되는 입력은 0001, 0010, 0100, 1000의 4가지이다. 이를 제외한 입력은 인코더에 있어서 **결과값이 보장되지 않은 입력**이므로, **잘못된 입력이나 중복된 입력, 또는 오류에 의한 입력**이라고 볼 수 있다. 하지만 이와 같은 입력은 현실 세계에서 쉽게 일어날 수 있고, 그 때마다 결과값이 보장되지 않는다면 시스템에 심각한 오류를 끼칠 수 있다.

이러한 문제를 개선할 수 있는 회로가 priority encoder로, binary encoder와 달리 입력 비트에 우선순위를 설정하고 우선순위가 높은 bit가 set되었다면 그보다 우선순위가 낮은 bit의 활성화 여부와는 상관없이 미리 결정된 출력을 제공한다.

위의 4 to 2 binary encoder의 경우 '0110'이 입력되었을 때 출력 결과값은 '11'이 되지만, 4 to 2 priority encoder의 경우 우선순위가 더 우선순위가 높은 bit를 기준으로 출력 결과값이 '01' 또는 '10'이 된다.

4. 4 to 2 encoder의 4가지 형태가 아닌 모든 입력 형태(16가지)에 대하여 동작되는 priority encoder의 논리 회로를 구성하여라.

input				output	
A	B	C	D	E0	E1
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

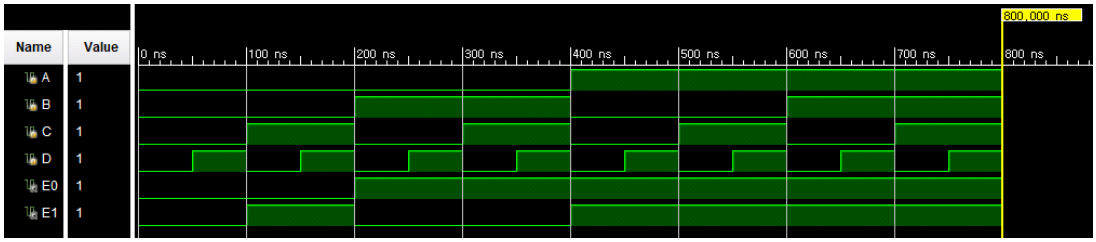
Karnaugh Map	
E_0	E_1
	

SOP form (by K-map)	
E_0	A + B
E_1	A + B'C

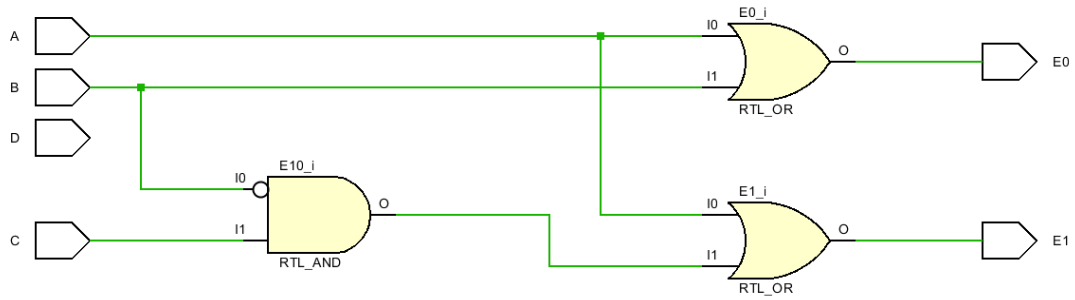
source code	testbench code

`timescale 1ns / 1ps
 	
module encoder (
input A, B, C, D,
output E0, E1
);
 	
assign E0 = A | B;
assign E1 = A | (~B&C);
 	
endmodule
	`timescale 1ns / 1ps
	
	module encoder_tb;
	
	reg A, B, C, D;
	wire E0, E1;
	
	encoder u_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;
	end
	
	always@ (A or B or C or D) begin
	A <= #400 ~A; B <= #200 ~B;
	C <= #100 ~C; D <= #50 ~D;
	end
	
	initial begin
	#800
	$finish;
	end
	
	endmodule

4 to 2 priority encoder simulation



4 to 2 priority encoder circuit design



보이는 것처럼 E_0의 경우엔 원래의 SOP 식과 동일하게 처리할 수 있지만, E_1에는 차이가 발생하는 것을 볼 수 있다. 위와 같이 SOP 식을 바꿈으로써 A > B > C > D 순으로 우선순위를 설정할 수 있고, 우선순위가 더 높은 bit가 활성화된 경우 그 아래의 bit와 무관하게 인코딩되도록 할 수 있다.

priority encoder에 따라 **NR 출력을 추가**함으로써, 0000이 입력되었을 경우 NR 출력을 High 또는 Low 상태로 두어 인코더가 활성화되지 않았음을 알리는 경우도 있지만, 이 경우에는 구현하지 않았다.

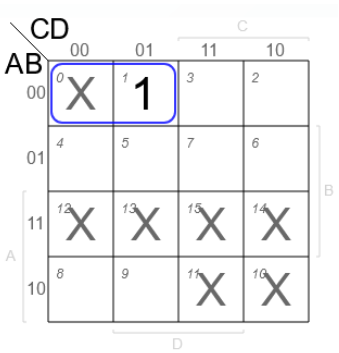
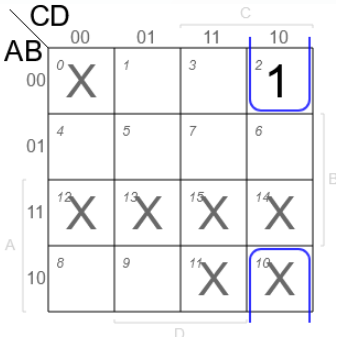
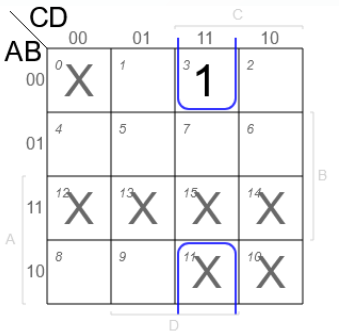
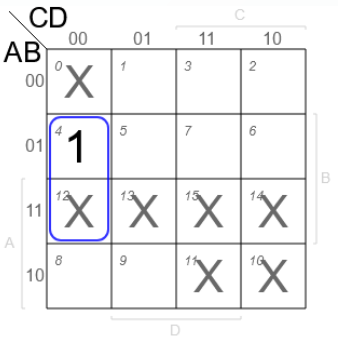
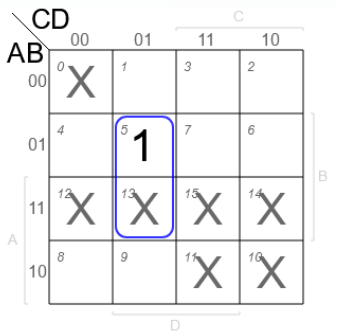
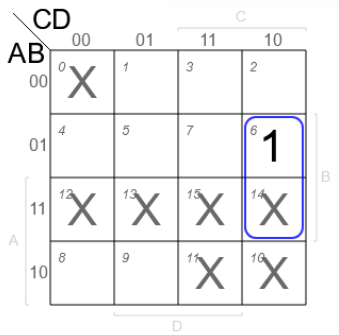
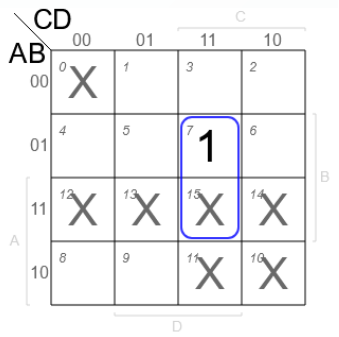
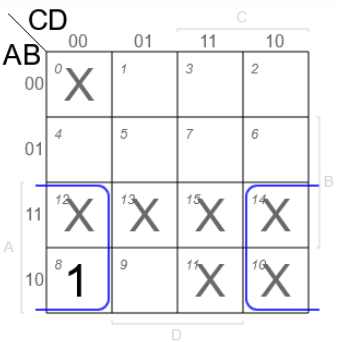
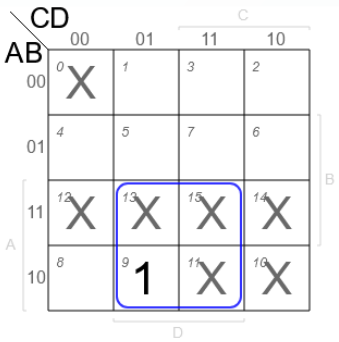
기대했던 것과 동일하게 구현된 모습을 simulation을 통해 다시 한 번 확인할 수 있다.

5. BCD to Decimal decoder 의 결과 및 Simulation 과정에 대해서 설명하시오. (Truth table 작성 및 k-map 포함)

input				output								
A_8	A_4	A_2	A_1	out1	out2	out3	out4	out5	out6	out7	out8	out9
0	0	0	0	X	X	X	X	X	X	X	X	X
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	X	X	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X

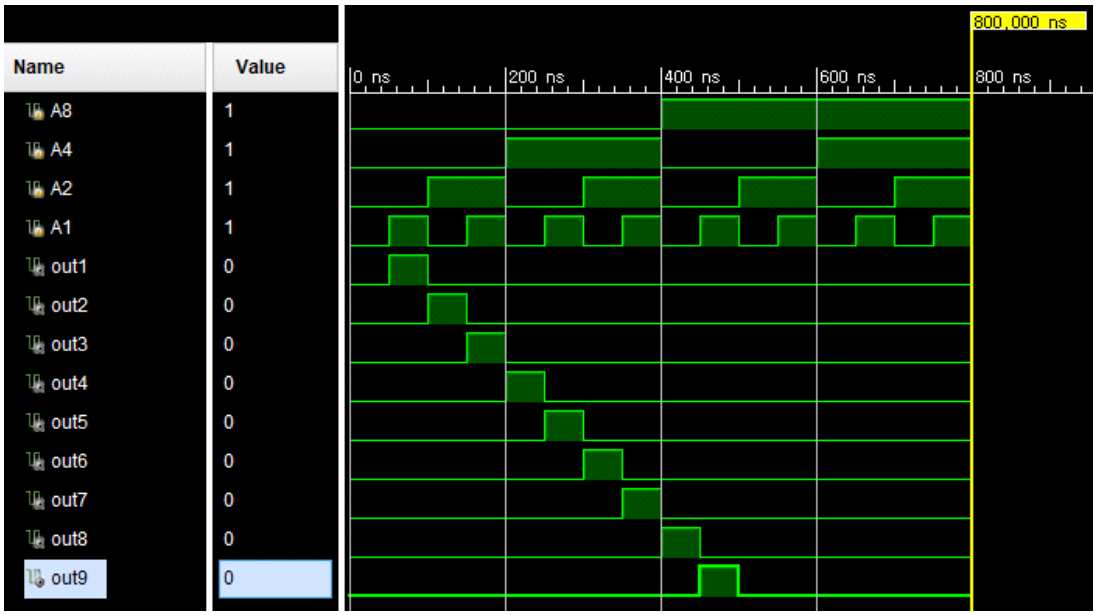
source code	testbench code

`timescale 1ns / 1ps
 	
module BCD_decoder(
input A8, A4, A2, A1,
output out1, out2, out3, out4, out5, out6, out7, out8, out9
);
 	
assign out1 = ~A8 & ~A4 & ~A2 & A1;
assign out2 = ~A8 & ~A4 & A2 & ~A1;
assign out3 = ~A8 & ~A4 & A2 & A1;
assign out4 = ~A8 & A4 & ~A2 & ~A1;
assign out5 = ~A8 & A4 & ~A2 & A1;
assign out6 = ~A8 & A4 & A2 & ~A1;
assign out7 = ~A8 & A4 & A2 & A1;
assign out8 = A8 & ~A4 & ~A2 & ~A1;
assign out9 = A8 & ~A4 & ~A2 & A1;
 	
endmodule
	`timescale 1ns / 1ps;
	
	module BCD_decoder_tb;
	
	reg A8, A4, A2, A1;
	wire out1, out2, out3, out4, out5, out6, out7, out8, out9;
	
	BCD_decoder u_test (
	.A8 (A8), .A4 (A4), .A2 (A2), .A1 (A1),
	.out1 (out1), .out2 (out2), .out3 (out3), .out4 (out4), .out5 (out5), .out6 (out6), .out7 (out7), .out8 (out8), .out9 (out9)
	);
	
	initial begin
	A8 = 1'b0;
	A4 = 1'b0;
	A2 = 1'b0;
	A1 = 1'b0;
	end
	
	always@ (A8 or A4 or A2 or A1) begin
	A8 <= #400 ~A8;
	A4 <= #200 ~A4;
	A2 <= #100 ~A2;
	A1 <= #50 ~A1;
	end
	
	initial begin
	#800
	$finish;
	end
	
	endmodule

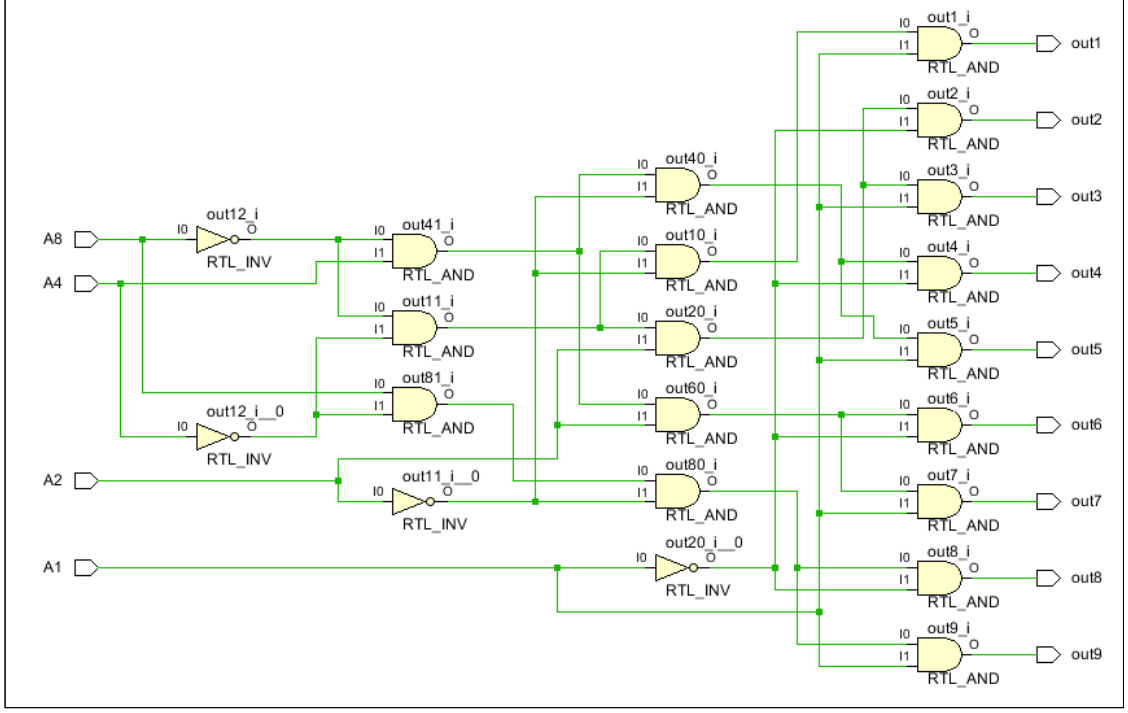
Karnaugh Map		
out1	out2	out3
		
out4	out5	out6
		
out7	out8	out9
		

SOP form (by K-map)	
out1	A'B'C
out2	B'CD'
out3	B'CD
out4	BC'D'
out5	BC'D
out6	BCD'
out7	BCD
out8	AD'
out9	AD

BCD to decimal decoder simulation



BCD to decimal decoder circuit design



BCD to decimal decoder는 BCD 방식으로 표현된 bit sequence를 10진법 숫자로 복호화하는 회로이다. 구현한 회로의 경우 0001~1001까지의 이진수 bit sequence를 십진수 1~9로 바꾸어 출력하는데, 이 때문에 0000, 1010, 1011, 1100, 1101, 1110, 1111에 해당하는 7가지 bit 입력들은 don't care 항으로 둘 수 있다.

단 don't care 항을 사용해 회로를 최적화하면 simulation이나 FPGA를 통한 test가 불편해지기 때문에 각 10진수에 대해 유일한 bit sequence를 대응시키도록 구현했다.

simulation을 확인하면, 0001~1001의 입력 bit sequence에 대해 십진수 1~9(out1~out9) 출력 라인이 정상적으로 대응된 것을 확인할 수 있다.

6. encoder와 decoder의 주요 응용에 대하여 설명하시오.

encoder와 decoder가 응용되는 분야는 네트워크 통신 분야가 대표적이다. 실습의 경우 간단한 인코딩, 디코딩 체계만을 다루었지만 데이터를 송신하고 수신하는 모든 분야에 인코더와 디코더의 개념을 적용할 수 있다. 아날로그 음성을 디지털 신호로 변환하고, 디지털 신호를 아날로그 신호로 변환하는 ADC(Analog to Digital Converter)와 DAC(Digital to Analog Converter)도 디

코더와 인코더의 응용이다.

뿐만 아니라, 컴퓨터에서 10진수 문자열과 ASCII code bit sequence로 변환하는 것, ASCII 코드로 표현된 숫자와 실제 이진수를 변환하는 것 등도 역시 encoder과 decoder의 작업이다. 컴퓨터는 모든 데이터를 이진수로 변환해 처리하므로, 컴퓨터에서 처리되는 거의 모든 데이터에 encoder와 decoder가 응용되고 있다고 보아도 무리가 없다.

특히, priority encoder는 마이크로프로세서 어플리케이션에서 프로그램 실행 중 interrupt를 감지하는 데에 사용되기도 한다.

7. 4 to 1 line MUX의 결과 및 Simulation 과정에 대해서 설명하시오. (code, Truth table 작성)

input				selector		output
A	B	C	D	a	b	out1
X	X	X	X	0	0	A
X	X	X	X	0	1	B
X	X	X	X	1	0	C
X	X	X	X	1	1	D

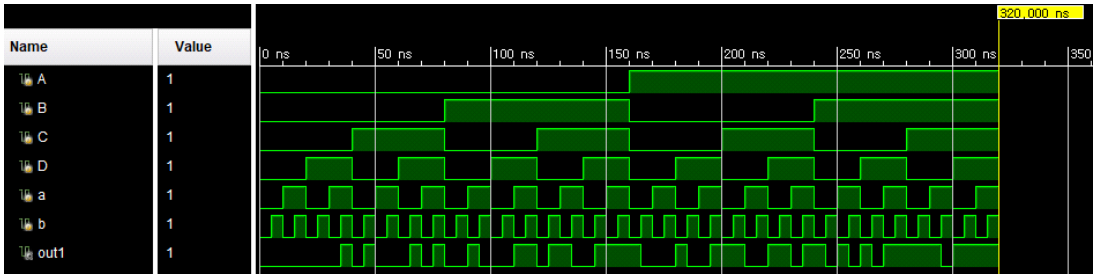
source code

```
`timescale 1ns / 1ps

module multiplexer(
    input A, B, C, D, a, b, // selector1, selector2
    output out1
);

assign AND_A = A & ~a & ~b;
assign AND_B = B & a & ~b;
assign AND_C = C & ~a & b;
assign AND_D = D & a & b;

assign out1 = AND_A | AND_B | AND_C | AND_D;

endmodule
```

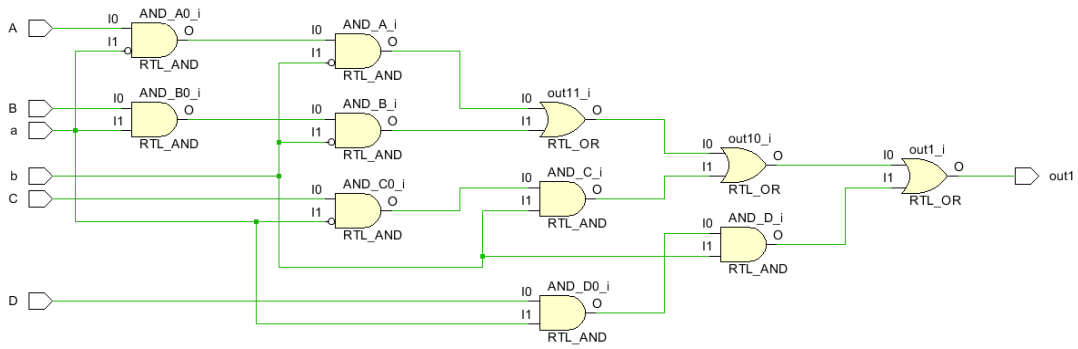
testbench code

```
`timescale 1ns / 1ps

module multiplexer_tb;

reg A, B, C, D, a, b;
wire out1;

multiplexer u_test (
    .A (A), .B (B), .C (C), .D (D), .a (a), .b (b),
    .out1 (out1)
);

initial begin
    A = 1'b0; B = 1'b0; C = 1'b0; D = 1'b0;
    a = 1'b0; b = 1'b0;
end

always@ (A or B or C or D or a or b) begin
    A <= #160 ~A;
    B <= #80 ~B;
    C <= #40 ~C;
    D <= #20 ~D;
    a <= #10 ~a;
    b <= #5 ~b;
end

initial begin
    #320
    $finish;
end

endmodule
```

multiplexer simulation



multiplexer circuit design



4-1 multiplexer의 경우 **4개의 데이터 입력, 2개의 선택기 입력**을 받아 **선택기 입력에 따라 데이터 중 하나를 선택해 출력**으로 내보낸다. 구현한 multiplexer의 경우 선택 입력 00, 01, 10, 11 각각에 따라 데이터 입력 A, B, C, D를 출력한다.

simulation을 확인했을 때, ab가 00일 때 A, ab가 01일 때 B, ab가 10일 때 C, ab가 11일 때 D가 출력되고 있으므로, 정상적으로 구현되었음을 확인할 수 있다.

8. 1 to 4 line deMUX 를 이용하여 4 to 16 decoder를 수행하고 결과를 나타내시오. (code, truth table 작성)

a. 1 to 4 line demultiplexer

data	selector		output			
F	a	b	A	B	C	D
X	0	0	F	X	X	X
X	0	1	X	F	X	X
X	1	0	X	X	F	X
X	1	1	X	X	X	F

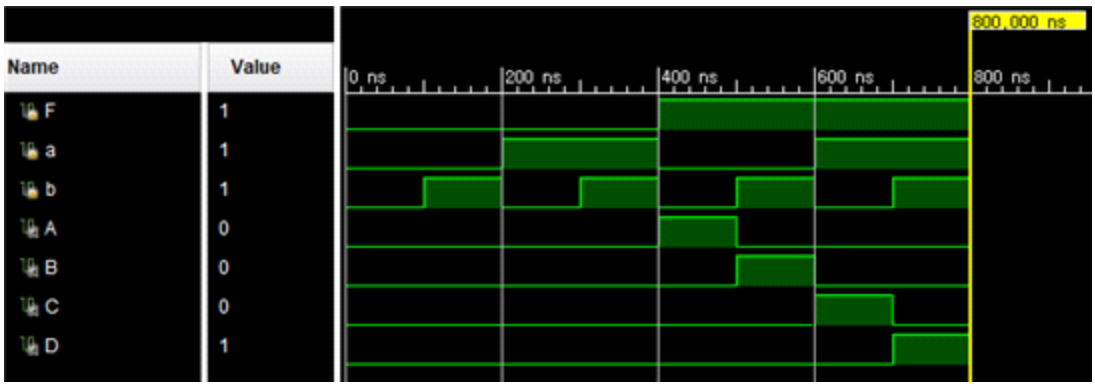
source code

```
`timescale 1ns / 1ps

module demultiplexer(
    input F, a, b,
    output A, B, C, D
);

assign A = F & ~a & ~b;
assign B = F & ~a & b;
assign C = F & a & ~b;
assign D = F & a & b;

endmodule
```

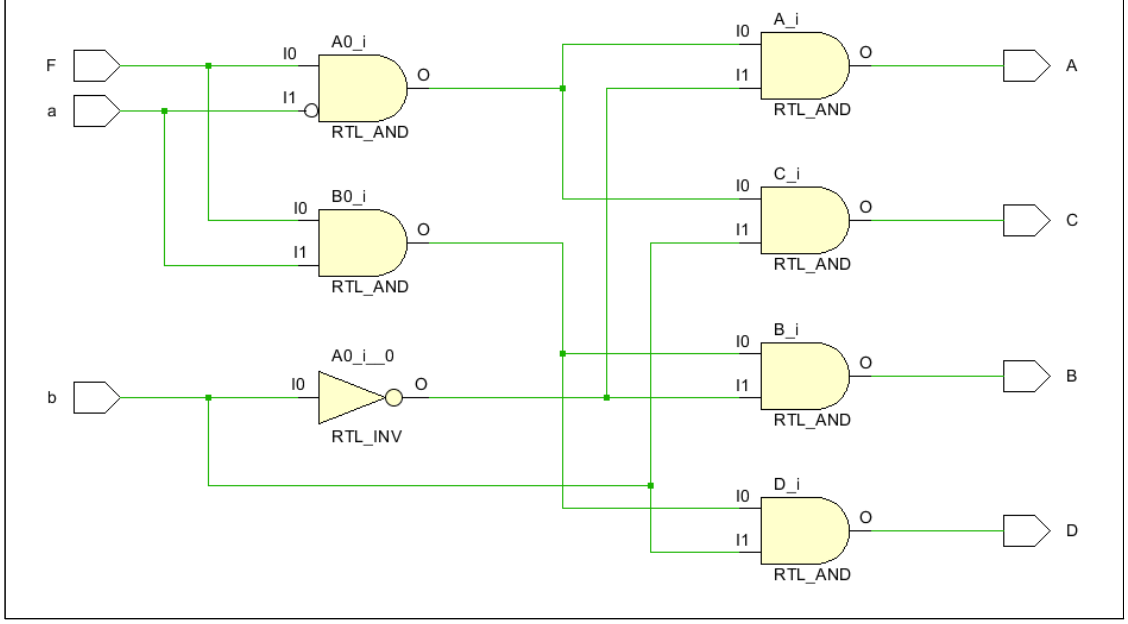
testbench code

```
`timescale 1ns / 1ps

module demultiplexer_tb;

reg F, a, b;
wire A, B, C, D;

demultiplexer u_test (
    .F (F), .a (a), .b (b),
    .A (A), .B (B), .C (C), .D (D)
);

initial begin
    F = 1'b0;
    a = 1'b0;
    b = 1'b0;
end

always@ (F or a or b) begin
    F <= #400 ~F;
    a <= #200 ~a;
    b <= #100 ~b;
end

endmodule
```

demultiplexer simulation



demultiplexer circuit design



1-4 demultiplexer의 경우 **1개의 데이터 입력, 2개의 선택기 입력**을 받아 **선택기 입력에 따라 출력 라인 4개 중 하나를 선택해 데이터 입력을 출력**한다. 구현한 demultiplexer의 경우 선택 입력(=ab) 00, 01, 10, 11 각각에 따라 출력 라인 A, B, C, D에 데이터 입력값이 출력된다.

simulation을 확인했을 때, ab가 00일 때 A, ab가 01일 때 B, ab가 10일 때 C, ab가 11일 때 D 라인에 데이터가 출력되고 있으므로, 정상적으로 구현되었음을 확인할 수 있다.

다음은 1-4 demultiplexer 5개를 이용해 4 to 16 decoder를 구현한 내용이다.

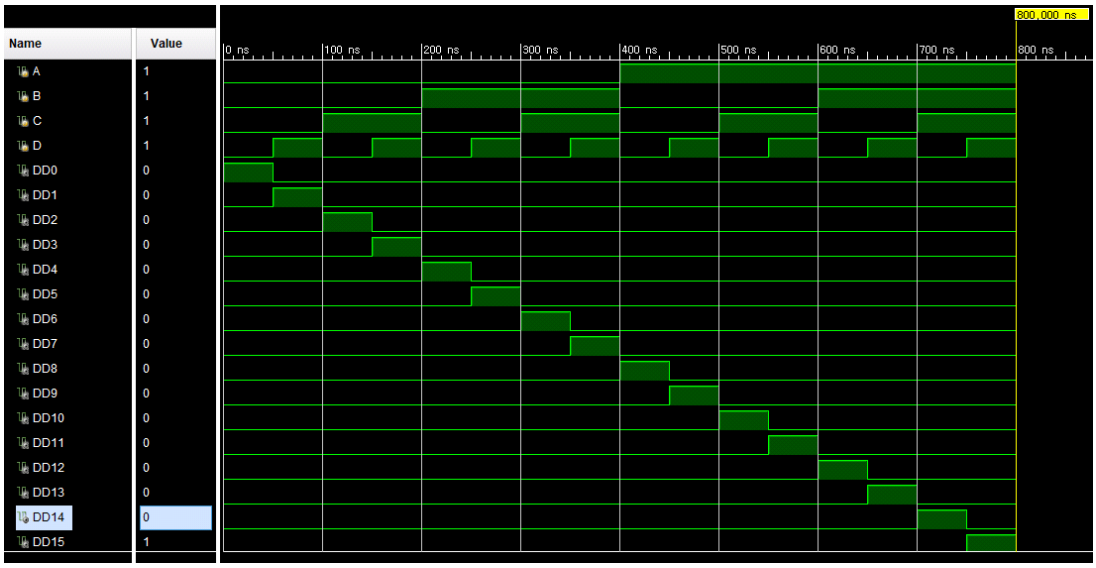
b. 4 to 16 decoder by 1 to 4 line demultiplexer

input				output															
a	b	c	d	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}
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

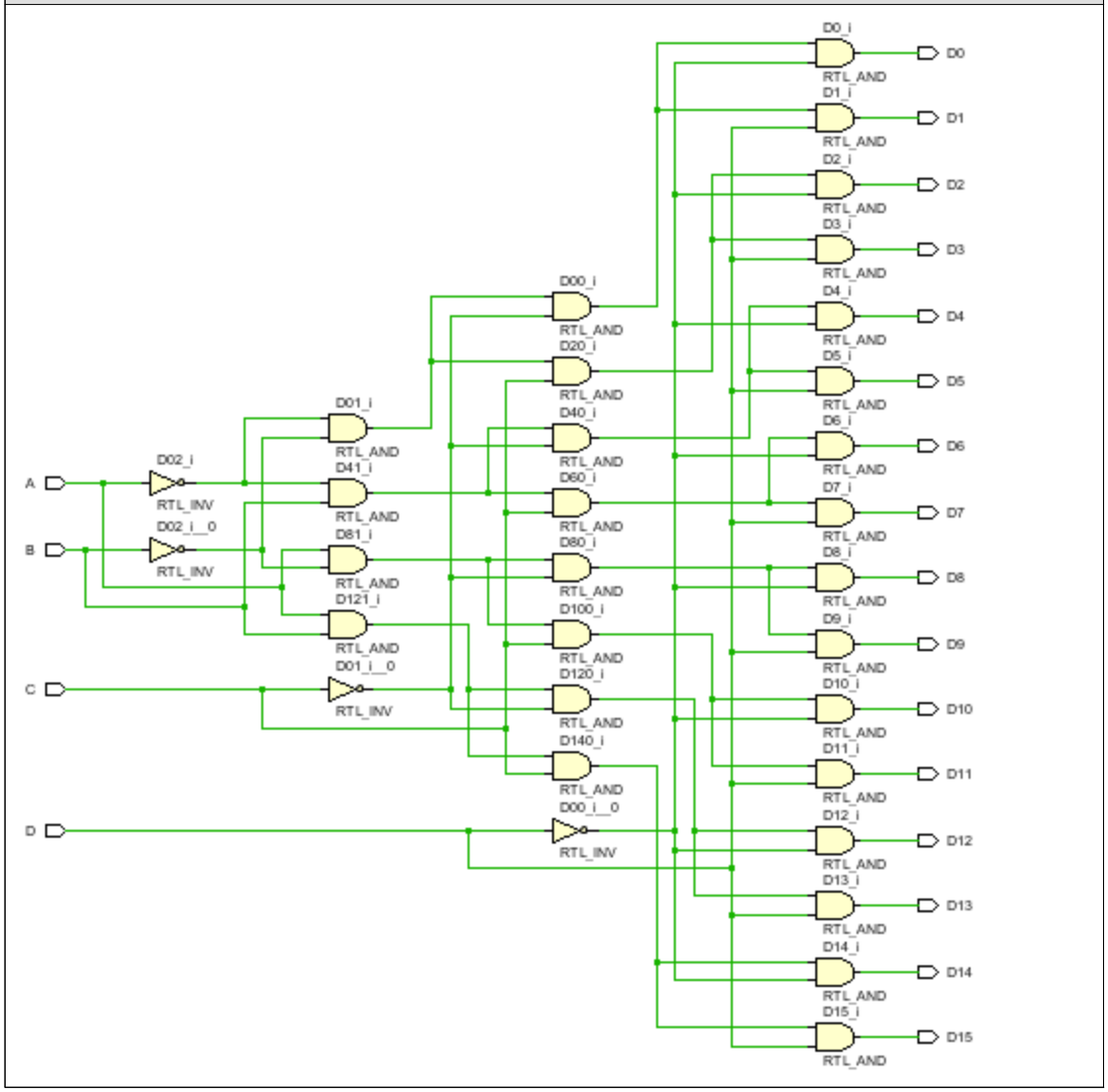
source code	testbench code

module demultiplexer_4x16(
  input a, b, c, d,
  output [15:0] D
);
  assign e0 = ~a&~b;
  assign e1 = ~a&b;
  assign e2 = a&~b;
  assign e3 = a&b;
  demultiplexer_1x4 (e0, c, d, D[0], D[1], D[2], D[3]);
  demultiplexer_1x4 (e0, c, d, D[4], D[5], D[6], D[7]);
  demultiplexer_1x4 (e0, c, d, D[8], D[9], D[10], D[11]);
  demultiplexer_1x4 (e0, c, d, D[12], D[13], D[14], D[15]);
endmodule
	`timescale 1ns / 1ps
	
	module decoder_4x16_tb;
	
	reg A, B, C, D;
	wire DD0, DD1, DD2, DD3, DD4, DD5, DD6, DD7, DD8, DD9, DD10, DD11, DD12, DD13, DD14, DD15;
	
	decoder_4x16 u_test(
	  .A (A), .B (B), .C (C), .D (D),
	  .D0 (DD0), .D1 (DD1), .D2 (DD2), .D3 (DD3),
	  .D4 (DD4), .D5 (DD5), .D6 (DD6), .D7 (DD7),
	  .D8 (DD8), .D9 (DD9), .D10 (DD10), .D11 (DD11),
	  .D12 (DD12), .D13 (DD13), .D14 (DD14), .D15 (DD15)
	);
	
	initial begin
	  A = 1'b0;
	  B = 1'b0;
	  C = 1'b0;
	  D = 1'b0;
	end
	
	always@ (A or B or C or D) begin
	  A <= #400 ~A;
	  B <= #200 ~B;
	  C <= #100 ~C;
	  D <= #50 ~D;
	end
	
	initial begin
	  #800
	  $finish;
	end
	
	endmodule

demultiplexer simulation

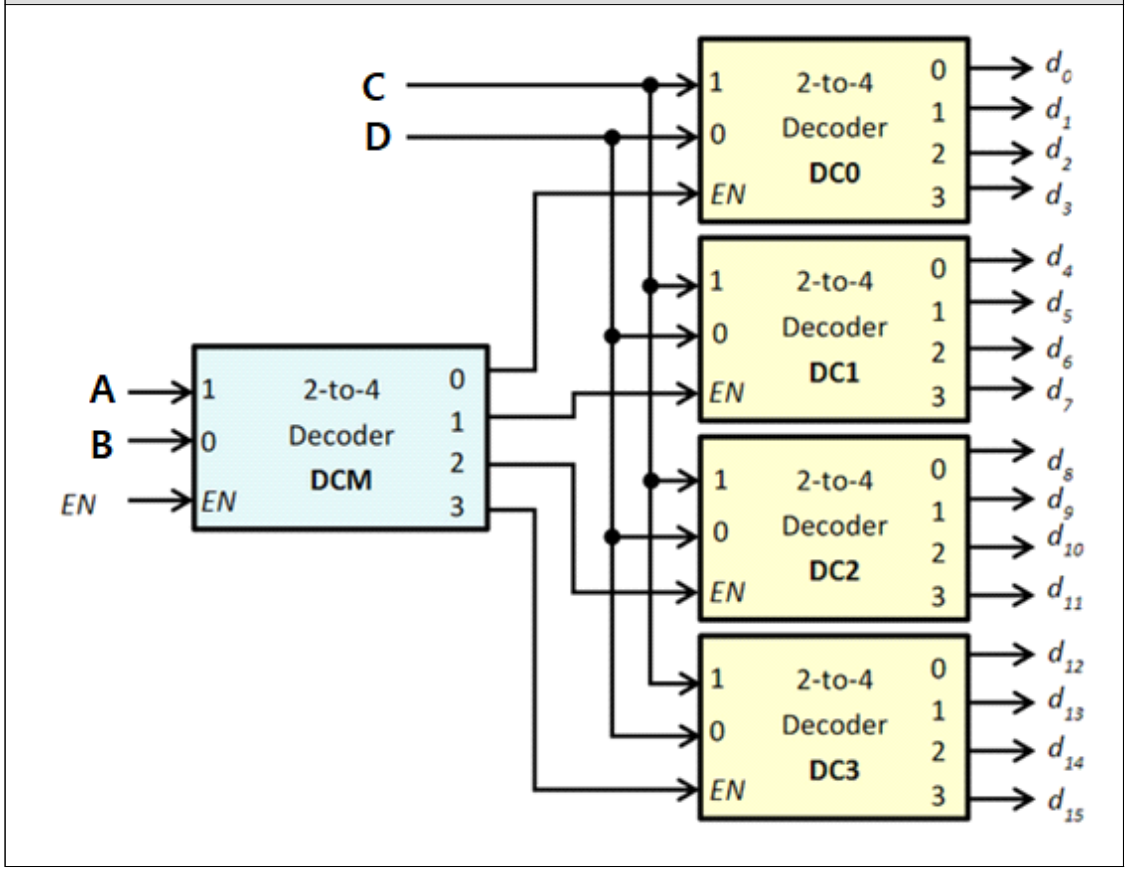


demultiplexer circuit design



예비보고서에서 이미 보았다시피, 1 to 4 demultiplexer와 EN 입력이 있는 2-4 decoder는 구조가 실질적으로 동일하다. 4 to 16 decoder를 구현하기 위해 2 to 4 decoder를 사용한다면 2 to 4 decoder 5개가 필요한데, 이를 1 to 4 multiplexer 5개로 대체할 수 있다. 구현되는 실제 회로의 logic level diagram은 다음과 같은 형태이다.

4 to 16 decoder logic level diagram



DCM은 **항상 활성화된 EN 입력**과 **2개의 선택 입력(A, B)**을 받으며, **A, B에 따라 몇 번째 2-to-4 decoder(1-4 demultiplexer)를 활성화할지를 결정**한다. **활성화되지 않은 decoder은 자동으로 '0000'을 출력**한다. 다음으로 C, D는 각 디코더에서 출력될 값을 결정한다. CD가 00, 01, 10, 11일 때 순서대로 '1000', '0100', '0010', '0001'이 각 디코더에서 출력된다.

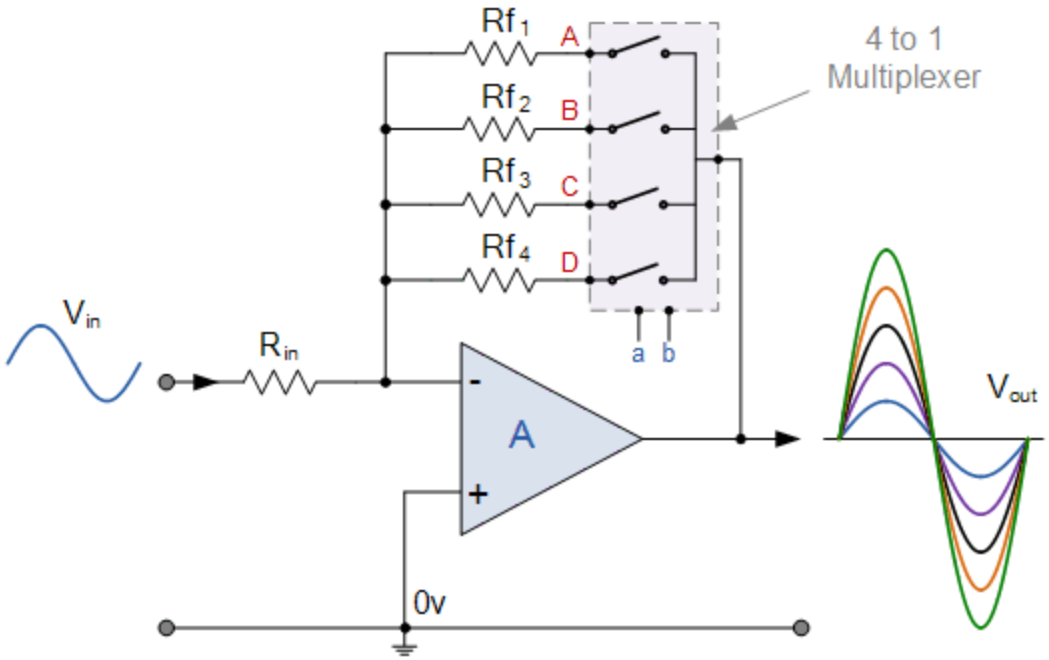
4 to 16 decoder의 진리표의 값에 따라 결과값이 출력되었으므로 성공적으로 구현된 것을 확인할 수 있다.

3. 결과 검토 및 논의 사항

이번 실험에서는 2x4 active low decoder, 2x4 active high decoder, 4x2 encoder, 4x2 priority encoder, 4x1 multiplexer, 1x4 multiplexer, 1x4 multiplexer를 이용한 4x16 decoder를 각각 verilog 상에서 구현하여 시뮬레이션해 보았으며, 4x2 priority encoder, 4x16 decoder를 제외한 회로를 FPGA 보드를 활용하여 실제 출력 결과를 확인해 보았다. 각 회로의 논리적 동작

에 따라 진리표를 그리고, Karnaugh map을 그림으로써 SOP 식을 도출했다. 실험 결과, 진리표를 바탕으로 기대되었던 동작이 정상적으로 이루어졌음을 확인했다.

4. 추가 이론 조사 및 작성



멀티플렉서는 신호를 전환하거나 라우팅할 수 있는 스위칭 조합 회로로서, 신호 라우팅, 데이터 통신, 데이터 버스 제어 등 다양한 응용 분야에 사용된다. 뿐만 아니라, 병렬 데이터를 광섬유 케이블 및 전화선 등의 단일 데이터 라인을 통과할 수 있도록 직렬 데이터로 변환할 수 있다. 이 때, 직렬 데이터를 다시 병렬 데이터로 전환하는 작업은 **디멀티플렉서**가 담당한다.

멀티플렉서는 아날로그 신호, 디지털 신호, 비디오 신호 등 다양한 신호를 상호 전환할 수 있고, 이 경우 회선이 과열되는 것을 방지하기 위해 교류 전류를 채널당 10mA-20mA 내외로 제한한다.