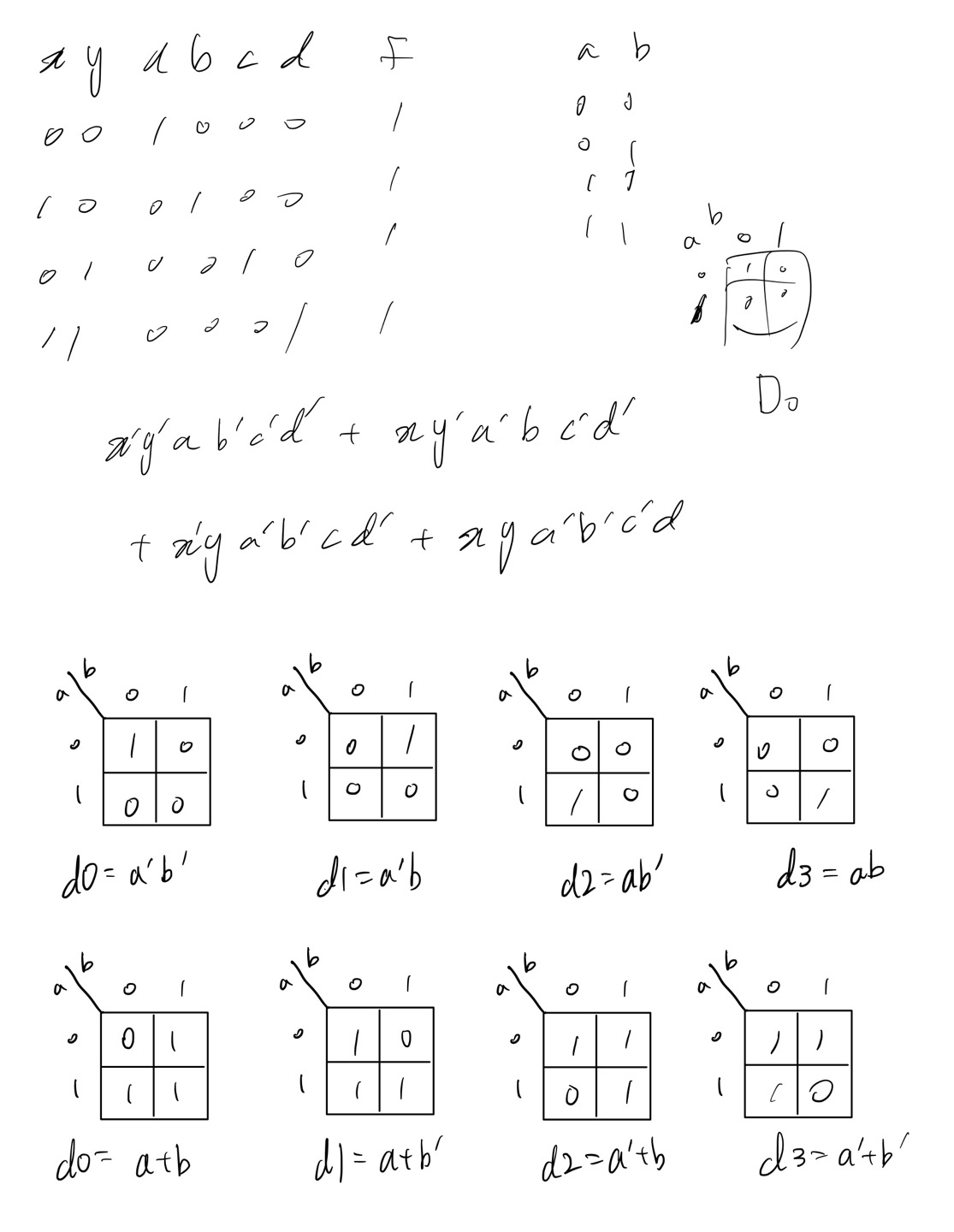
**9주차 결과보고서**

**전공: 경영학과 학년: 3학년 학번: 20190963 이름: 한다현**

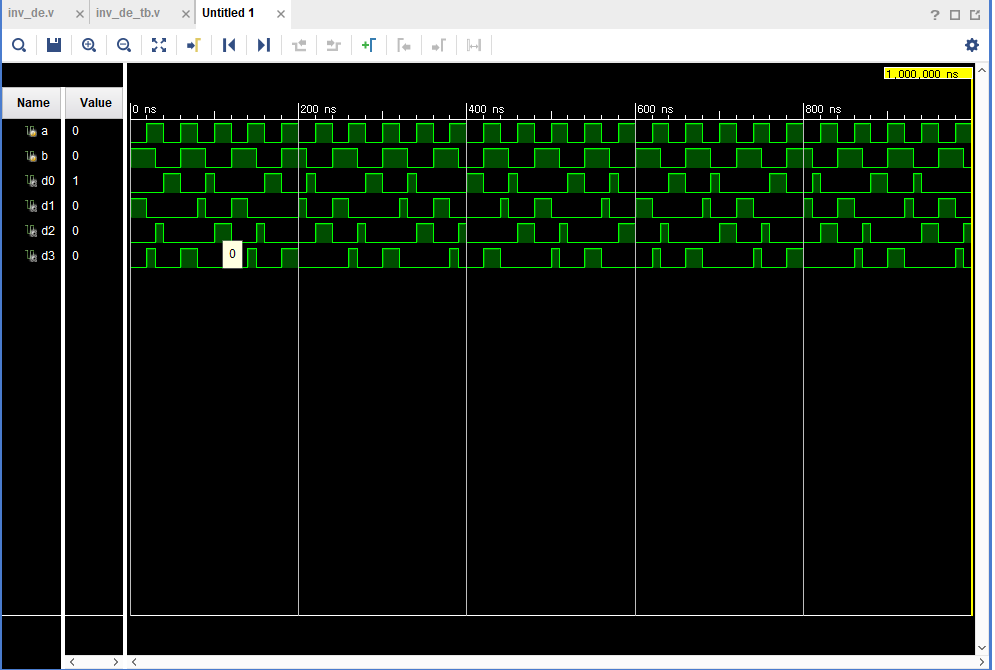
**1.**

**2 to 4 decoder는 2개의 입력의 조합으로 4개의 출력 중 하나를 결정하는 장치이다. 2개의 입력 a, b와 4개의 출력 d0, d1, d2, d3가 있는데, a, b의 값에 따라 d0~d3 중 어떤 게 출력될지 결정된다. 2 to 4 decoder는 AND 게이트로 설계되기도 하고 NAND 게이트로 설계되기도 한다. 우선 AND 게이트로 설계한 디코더는 active high decoder이며 4개의 출력 중 하나의 값만 1이 되고, 나머지 3개의 출력은 0이 된다. 이는 1의 값을 갖는 아웃풋이 출력된다는 것을 의미한다. AND 게이트로 설계된 2 to 4 디코더의 진리표와 카르노 맵은 다음과 같다.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **In a** | **In b** | **Out d0** | **Out d1** | **Out d2** | **Out 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** |



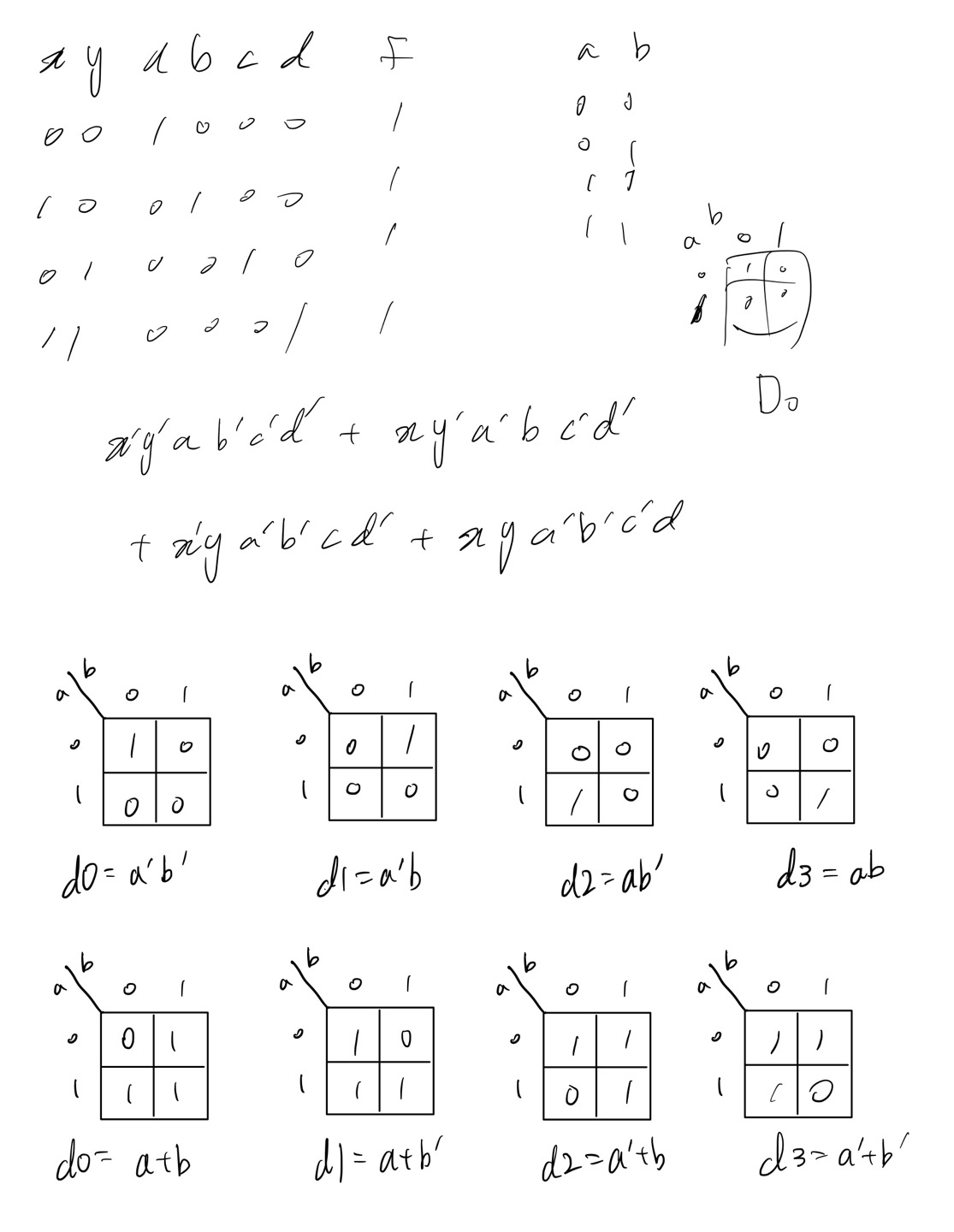
**a, b가 0 모두 0일 때는 d0가 출력되고, a는 0이고 b가 1일 때는 d1이 출력되고, a가 1이고 b가 0일 때는 d2가 출력되고, a, b 모두 1일 때는 d3가 출력된다. 이를 verilog의 디자인 코드로 설계하여 a, b가 일정한 간격으로 변화할 때의 결과를 시뮬레이션으로 살펴보면 다음과 같다.**



**시뮬레이션 결과를 보면 a, b가 모두 0일 때는 d0가 출력되는 것을 볼 수 있으며, 다른 경우에도 모두 진리표와 같은 결과를 출력하는 것을 볼 수 있다.**

**다음으로, NAND 게이트를 사용하여 설계한 2 to 4 decoder는 active low decoder이며 앞서 설계한 active high decoder와 반대의 결과를 출력한다. 즉, a, b가 모두 0일 때는 d0가 0이 되고, 나머지 3개의 아웃풋은 1이되며, a가 0이고 b가 1인 경우에는 d1이 0이 되고, 나머지 3개의 아웃풋은 1이 되는 방식이다. NAND 게이트를 사용하여 설계한 디코더의 진리표와 카르노 맵은 다음과 같다.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **In a** | **In b** | **Out d0** | **Out d1** | **Out d2** | **Out 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** |



**진리표와 카르노 맵을 바탕으로 NAND 게이트를 사용하여 2 to 4 decoder의 디자인 코드를 설계한 뒤 시뮬레이션 결과를 살펴보면 다음과 같다.**

텍스트, 스크린샷, 전자기기, 컴퓨터이(가) 표시된 사진

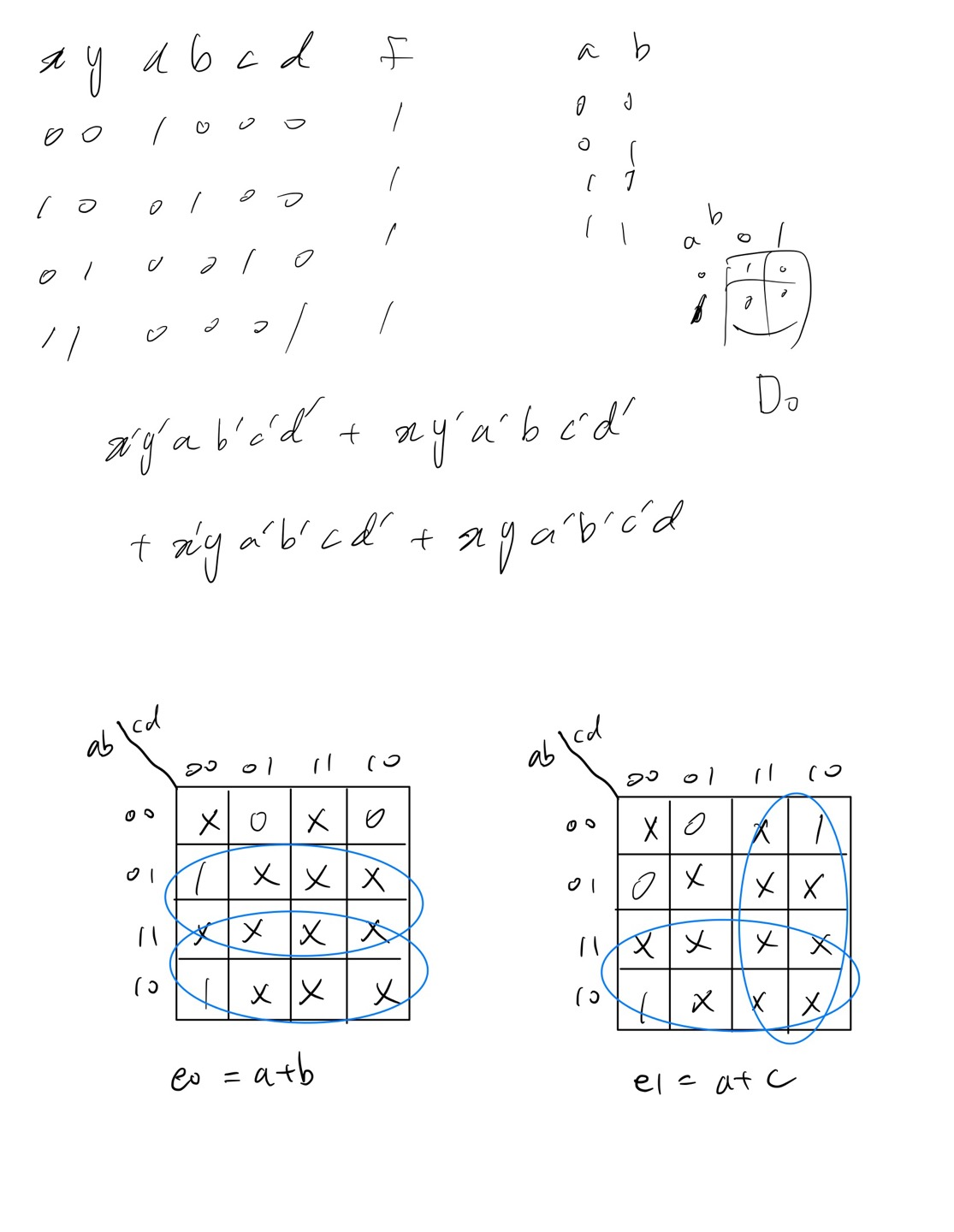
자동 생성된 설명

**시뮬레이션 결과 이전에 AND 게이트로 설계했던 디코더와 반대로 NAND 게이트를 사용하여 디코더를 설계하면 두 개의 입력의 조합에 따라 특정 아웃풋의 값이 0이 되는 것을 알 수 있다.**

**2.**

**인코더는 디코더와 반대의 기능을 수행한다. 인코더는 2n개의 인풋과 n개의 아웃풋으로 이루어져 있으며, 2n개의 인풋 중 하나가 1이 되었을 때 그에 해당하는 결과를 출력한다. 4\*2 인코더에는 a, b, c, d 4개의 인풋과 e0, e1 2개의 아웃풋이 있고 a, b, c, d 중 하나의 값이 1이 될 때 그에 해당하는 e0와 e1 값이 출력된다. 다음은 4\*2 인코더의 진리표와 카르노 맵이다.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **In a** | **In b** | **In c** | **In d** | **Out e0** | **Out e1** |
| **0** | **0** | **0** | **1** | **0** | **0** |
| **0** | **0** | **1** | **0** | **0** | **1** |
| **0** | **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **0** | **0** | **1** | **1** |

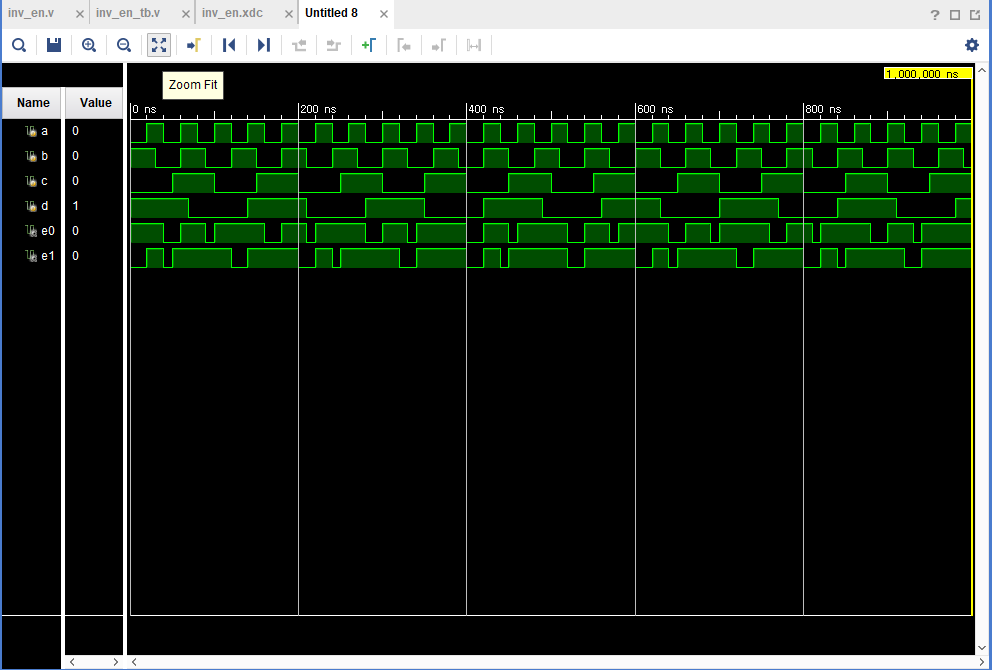


**인코더에서는 위의 진리표와 같이 4개의 인풋 중 하나의 값만 1이 되어야 결과가 정상적으로 출력된다. 인코더는 OR 게이트로 설계되며, e0와 e1의 디자인 코드는 다음과 같다.**

**assign e0 = a | b;**

**assign e1 = a | c;**

**이 디자인 코드를 바탕으로 a, b, c, d 값이 일정한 간격에 따라 변화하도록 시뮬레이션 코드를 작성하면 그 결과는 다음과 같다.**



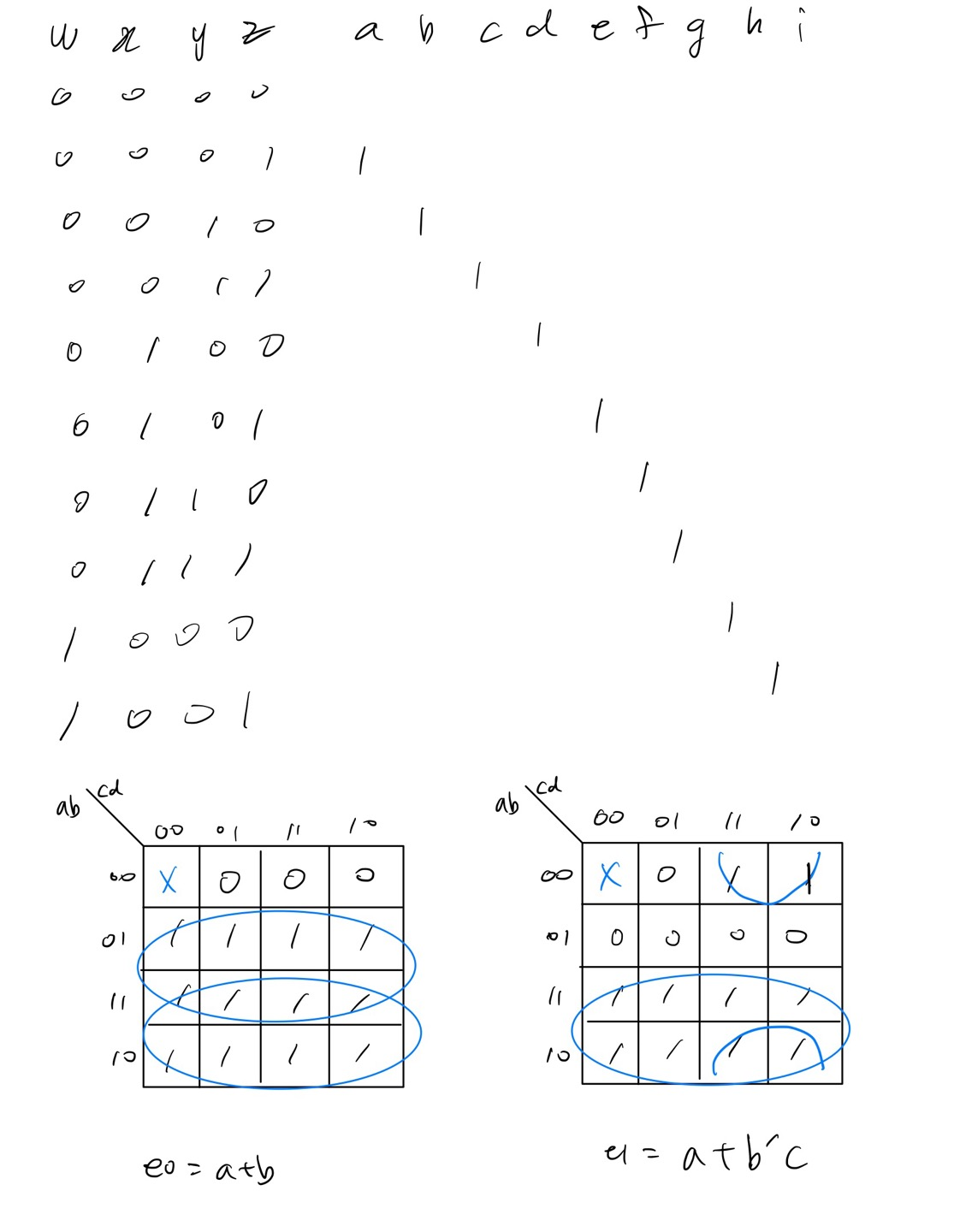
**시뮬레이션 결과를 보면 a, b, c가 0이고, d가 1일 때 e0와 e1의 값은 진리표와 동일하게 0, 0이 되는 것을 알 수 있으며, a가 1이고 나머지 3개의 값이 0이면 e0와 e1의 값은 진리표와 동일하게 1, 1이 되는 것을 알 수 있다.**

**3.**

**인코더는 여러 개의 입력 중에서 단 하나의 입력만이 1이 될 수 있다고 가정한다. 그렇지 않고 여러 개의 입력이 동시에 1이 된다면 회로의 의미가 없어지기 때문이다. 8개의 입력을 가지고 있는 인코더를 생각해봤을 때, 8개의 입력으로 만들어지는 조합은 28 = 256개이다. 하지만 8\*3 인코더는 256개의 조합 중 단 8개의 조합만을 사용한다. 즉, 256개의 조합 중 8개의 조합만 인코더에서 의미를 갖는다. 나머지 248개의 조합은 1의 값을 갖는 입력이 두 개 이상이거나 하나도 없는 조합이기 때문에 여러 개의 입력 중 1이 단 하나만 존재해야 하는 인코더에서는 의미가 없는 조합이다. 따라서 이러한 나머지 조합들은 don’t care 조건으로 처리된다. 이를 4 to 2 encoder에 적용해도 마찬가지이다. 4개의 입력으로 얻을 수 있는 조합의 수는 24 = 16개이다. 하지만 앞선 실험에서 확인했던 것과 같이 16개의 조합에서 4개의 입력 중 하나의 값만 1이 되는 조합은 단 4개이다. 나머지 12개의 조합은 4 to 2 encoder에서 의미가 없는 조합이다. 따라서 앞서 확인한 4개의 입력 형태를 제외한 나머지 12개의 입력 형태는 don’t care를 의미한다.**

**4.**

**기본적인 인코더는 입력 형태가 제한되어 있다. 예를 들어 4\*2 인코더는 입력 중 하나만이 1이 되어야 하기 때문에 인풋의 조합이 (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0) 네 가지만 가능하다. 이러한 문제를 극복하기 위해 값이 1인 입력이 두 개 이상이거나 하나도 없을 경우에도 정상적으로 동작하는 Priority encoder를 사용하고 이는 4가지의 조합이 아니라 16가지의 모든 입력 형태에 대해 동작한다. Priority encoder는 우선순위를 설정해야 하고 우선 순위가 높은 입력의 값이 1이 된다면 그보다 낮은 우선순위를 갖는 입력의 값은 don’t care가 된다. 4개의 입력 a, b, c, d와 두 개의 출력 e0, e1을 갖는 priority encoder의 16가지 입력형태에 대해 카르노 맵으로 표현하면 다음과 같다.**



**Priority encoder의 카르노 맵에서 볼 수 있듯이, 기본적인 encoder는 대부분의 조합에 대해 don’t care 상태를 가졌는데 priority encoder는 대부분 1과 0으로 구분된다. 이를 verilog로 디자인 코드를 작성한 뒤 schematic을 보면 다음과 같다.**

`timescale 1ns / 1ps

module inv(

input a, b, c, d,

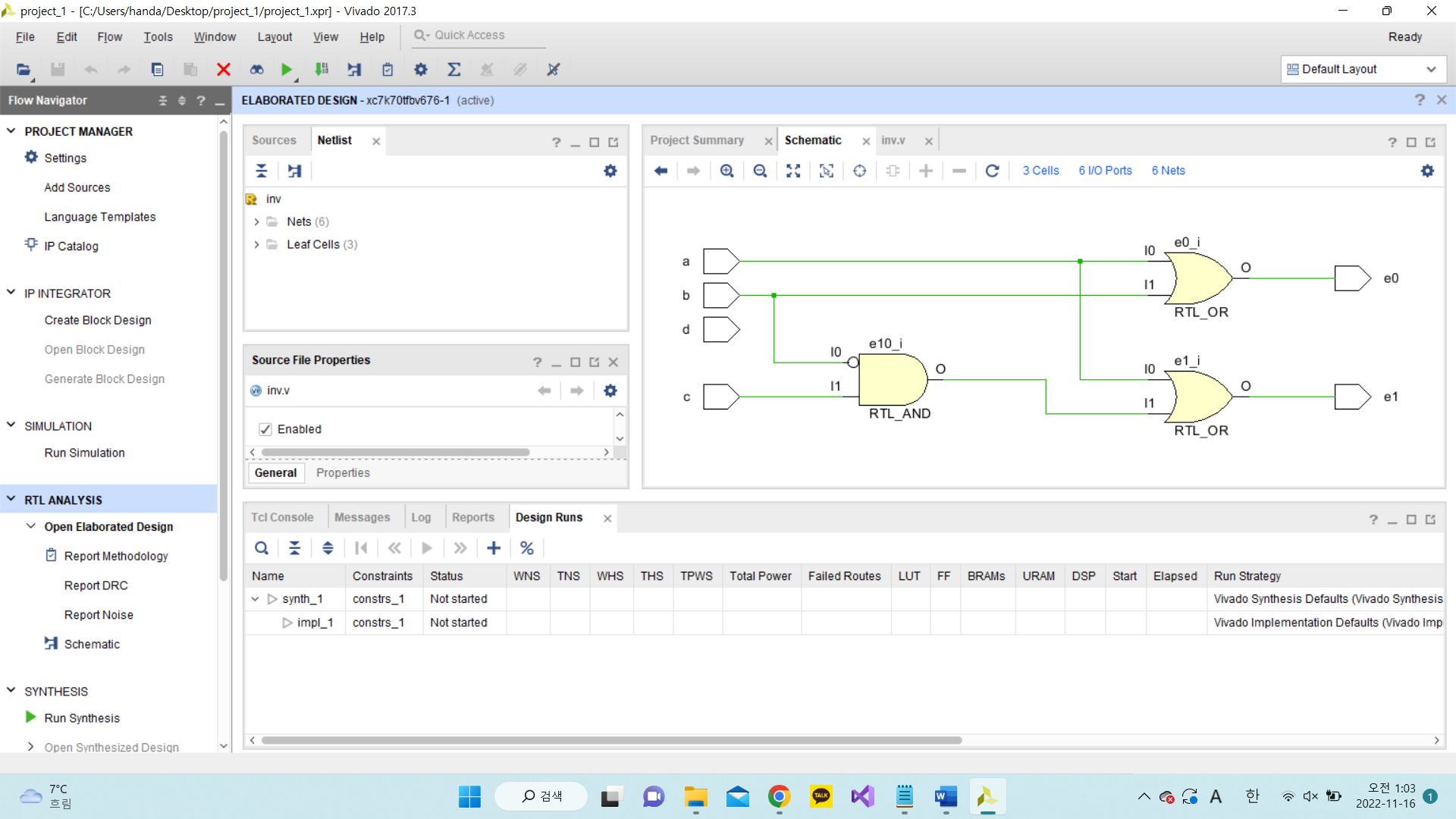
output e0, e1

);

assign e0 = a|b;

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

endmodule

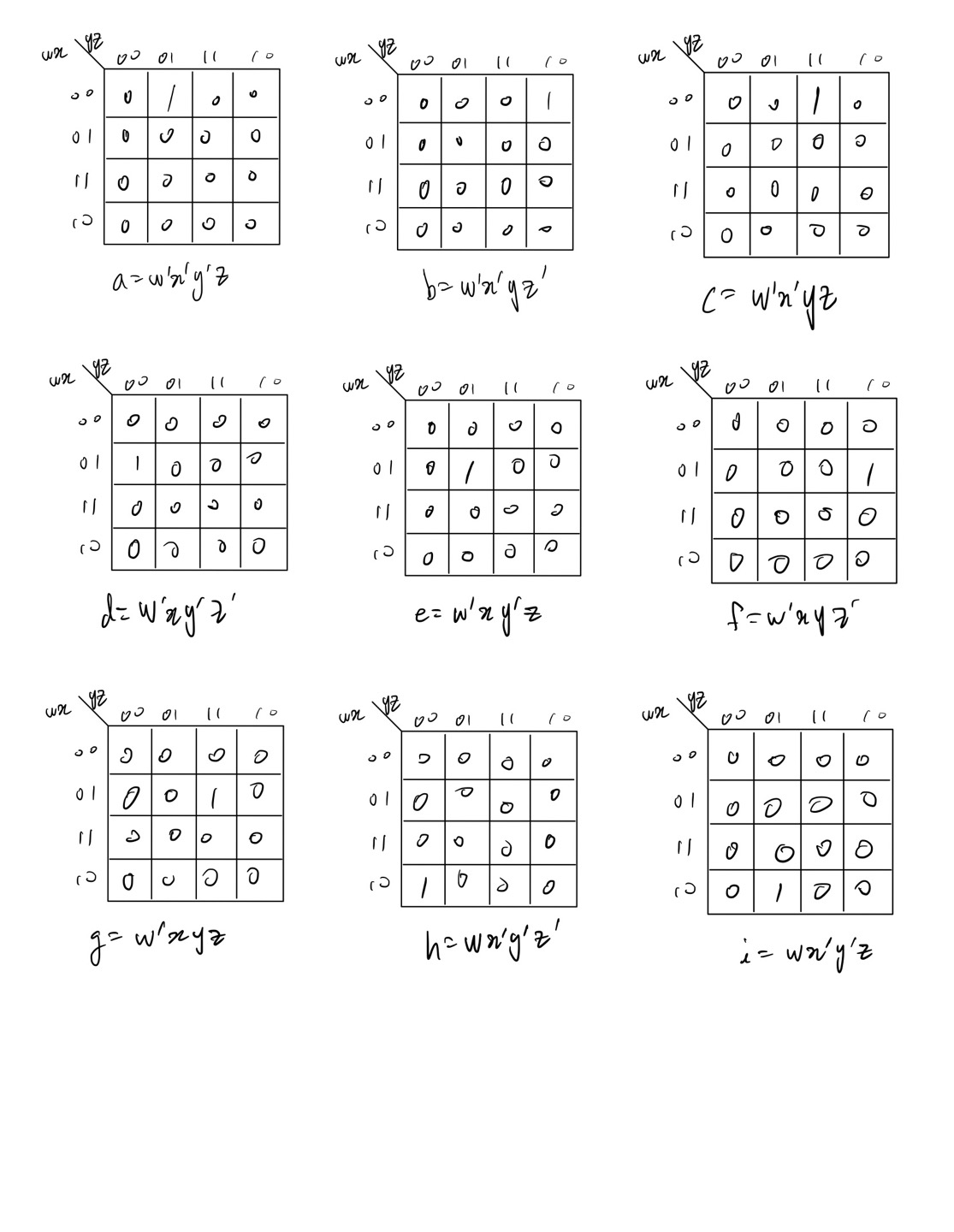


**schematic에서도 확인할 수 있듯이 e0와 e1이 카르노 맵으로 도출한 것과 동일하게 적절히 설계되었다는 것을 알 수 있다.**

**5.**

**이번 실험에서 실습한 BCD to decimal decoder는 1~9에 해당하는 BCD코드를 10진수로 표현하는 장치이다. 이번 실험에서는 8421 BCD 코드를 사용하였다. 9개의 출력 a, b, c, d, e, f, g, h, i를 각각 10진수 1~9에 대응시킨 후 1~9에 해당하는 8421 BCD 코드와 각 숫자에 해당하는 출력에 대해 진리표와 카르노 맵을 작성하면 다음과 같다.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Deci** | **In w** | **In x** | **In y** | **In z** | **Out a** | **Out b** | **Out c** | **Out d** | **Out e** | **Out f** | **Out g** | **Out h** | **Out i** |
| **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **2** | **0** | **0** | **1** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **3** | **0** | **0** | **1** | **1** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| **4** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** |
| **5** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** |
| **6** | **0** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** |
| **7** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** |
| **8** | **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** | **0** | **1** |



**이를 verilog에서 디자인 코드로 나타내면**

**assign a = ~w & ~x & ~y & z;**

**assign b = ~w & ~x & y & ~z;**

**assign c = ~w & ~x & y & z;**

**assign d = ~w & x & ~y & ~z;**

**assign e = ~w & x & ~y & z;**

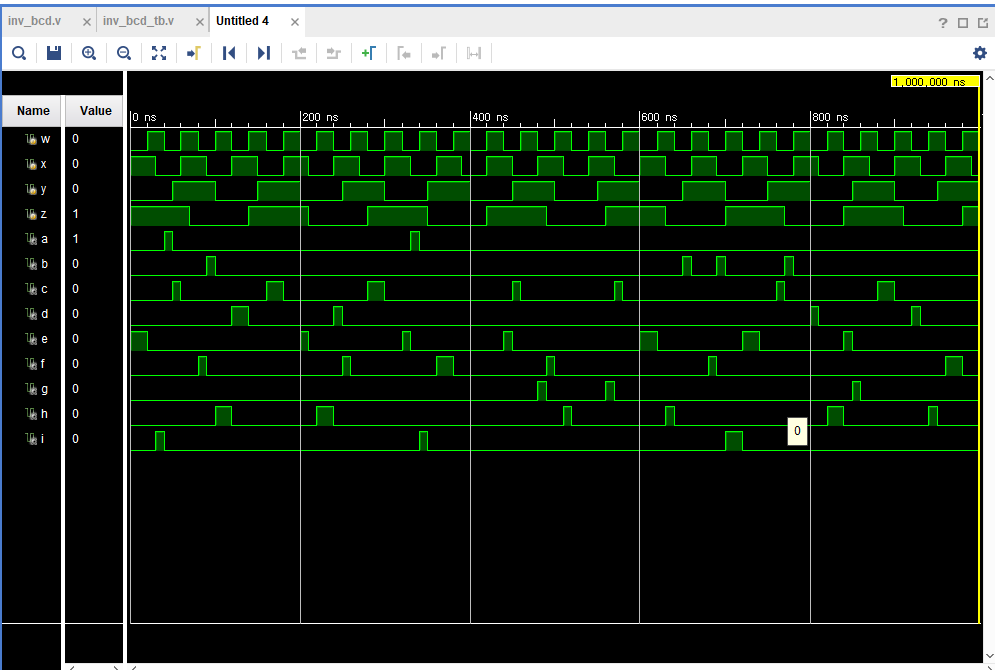
**assign f = ~w & x & y & ~z;**

**assign g = ~w & x & y & z;**

**assign h = w & ~x & ~y & ~z;**

**assign i = w & ~x & ~y & z;**

**이다. 위에 코드는 1~9에 해당하는 BCD 코드가 1~9에 해당하는 출력으로 표시되도록 한다. 시뮬레이션 코드에서 네 개의 인풋 w, x, y, z가 일정한 간격에 따라 변화하도록 설계한 뒤 a~i에 해당하는 결과를 살펴보면 다음과 같다.**



**위 사진은 시뮬레이션 결과이다. 입력 w, x, y가 0이고 z가 1인 경우는 8421 BCD 코드 0001을 의미하고 이는 10진수 1을 뜻한다. 따라서 1에 해당하는 출력 a의 값만 1이 되고 나머지 출력의 값은 0이 되어 올바르게 출력된 것을 알 수 있다. 추가로, w가 1이고 x, y, z가 0인 경우는 8421 BCD 코드 1000을 의미하고 이는 10진수 8을 뜻하기 때문에 8에 해당하는 출력인 g의 값만 1이 되고 나머지 출력의 값은 0이 되어 BCD to decimal decoder가 정상적으로 동작하는 것을 확인할 수 있다.**

**6.**

**Decoder는 여러 장치 중에서 하나를 선택하는 용도로 많이 사용된다. 각각의 장치는 고유한 주소를 갖는데, 이 주소가 디코더에 입력되고, 디코더에서 하나의 출력을 활성화시키면 그 출력에 해당하는 주소를 가진 장치가 사용된다. 이러한 대표적인 장치가 7-segment display이다. Seven segment display는 7개의 LED로 구분되어 있는데 decoder를 사용하여 입력에 따라 그에 해당하는 LED를 활성화시켜 원하는 출력을 display에 보여줄 수 있다. 인코더는 디코더의 반대 기능을 수행한다. 즉, 2n개의 입력으로 n개의 출력을 나타낸다. 따라서 인코더는 10진수를 2진수로 변환하는 장치, 정보 전송을 일정한 규칙에 따라 암호로 변환하는 장치 등에서 사용된다. 10진수를 2진수로 변환하는 장치에서는 0에서 9까지의 10진수를 0000부터 1001까지의 2진수로 변환한다. 이러한 장치에는 10개의 입력과 4개의 출력이 필요하다. 10개의 입력 중 하나가 활성화되면 그 10진수에 해당하는 2진수가 출력되도록 4개의 출력을 결정한다.**

**7.**

**Multiplexer는 여러 개의 입력선 중에서 하나를 선택하여 출력하는 조합 논리회로이다. 4 to 1 line MUX는 4개의 입력선과 2개의 선택선, 그리고 하나의 출력선으로 이루어져 있다. 2개의 선택선 x, y의 조합으로 4개의 입력선 a, b, c, d 중 하나를 출력선에 출력하는 기능을 한다. 즉, x, y의 값에 따라 출력되는 입력선이 다르다는 것을 의미하고 4 to 1 line MUX는 4개의 AND 게이트와 하나의 OR 게이트로 설계되어 있다. 각각의 AND 게이트에는 a, b, c, d 중 하나의 입력선과 x, y의 조합이 입력되며, OR 게이트에는 AND 게이트의 출력이 입력된다. 선택선 x, y의 조합에 따른 출력 값을 진리표로 표현하면 다음과 같다.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **In x** | **In y** | **In a** | **In b** | **In c** | **In d** | **Out f** |
| **0** | **0** | **1** | **X** | **X** | **X** | **1** |
| **1** | **0** | **X** | **1** | **X** | **X** | **1** |
| **0** | **1** | **X** | **X** | **1** | **X** | **1** |
| **1** | **1** | **X** | **X** | **X** | **1** | **1** |

**진리표에서 볼 수 있듯이 x가 0이고 y가 0일 때 a의 값이 1이면 1이 출력된다. 이는 4개의 입력선 중 a가 선택되어 출력된 것이라고 할 수 있다. 선택선 x, y가 모두 0이고 a가 1일 때 다른 입력선의 값은 중요하지 않다. 두 선택선의 조합에 대응하는 입력선의 값이 1이어야지만 출력 값이 1이 되기 때문이다. 선택선의 조합에 대응하지 않는 입력선의 값은 1이 되더라도 출력 값에 영향을 미치지 않는다.**

**`timescale 1ns / 1ps**

**module inv\_mux(**

**input x, y, a, b, c, d,**

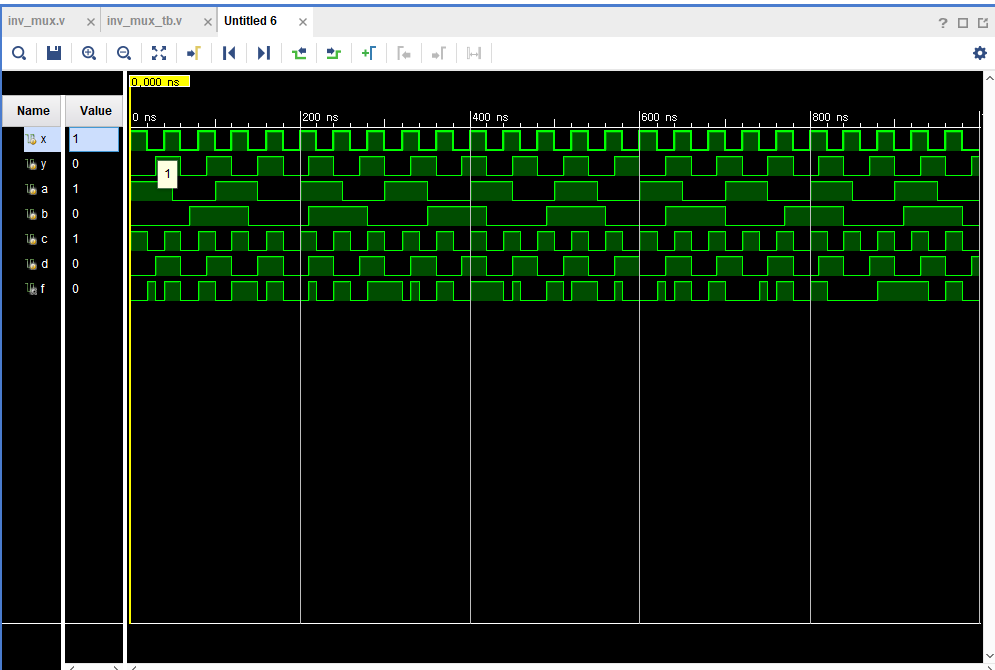
**output f**

**);**

**assign f = (~x & ~y & a) | (x & ~y & b) | (~x & y & c) | (x & y & d);**

**endmodule**

**위 코드는 4 to 1 line MUX를 설계한 디자인 코드이다. 코드에서 볼 수 있듯이 4 to 1 line MUX에는 4개의 AND 게이트가 있는데, 선택선의 조합이 (0,0)일 때는 a의 값이, (1,0)일 때는 b의 값이, (0,1)일 때는 c의 값이, (1,1)일 때는 d의 값이 1이 되어야 출력 f의 값이 1이 되는 것을 알 수 있다. 위 코드에 따르면 선택선의 조합이 (1,1)일 경우에는 다른 입력선의 값이 1이더라도 d의 값이 0이면 출력 f의 값은 0이 되고, d의 값이 1이면 출력 f의 값은 1이 된다. 이제 이 디자인 코드를 바탕으로 입력선 a, b, c, d와 선택선 x, y의 값이 일정한 간격으로 변화하도록 시뮬레이션 코드를 설계하면 그 결과는 다음과 같다.**



**시뮬레이션의 결과를 보면 x가 1이고 y가 0일 때 출력 f가 1이 되려면 진리표에 따라 b의 값이 1이 되어야 한다. 하지만 사진에 나타난 경우에는 b가 아닌 a와 c가 1이고, b는 0이기 때문에 출력 f는 0이 된다. 반면, x, y가 모두 0이고 a가 1인 경우에는 출력 f 값이 1이 되는 것을 확인할 수 있으므로 정상적으로 작동하는 것을 알 수 있다.**

**8.**

**1 to 4 deMUX를 이용하여 4 to 16 decoder를 설계하려면 5개의 1 to 4 deMUX가 필요하다. 두 개의 선택선 s0, s1와 4개의 출력 x0, x1, x2, x3, 그리고 입력 D로 구성된 첫 번째 1 to 4 deMUX에서 두 선택선 s0, s1의 조합으로 4개의 출력 중 하나가 정해지고 각각의 출력은 나머지 4개의 1 to 4 deMUX의 입력이 된다. 나머지 4개의 1 to 4 deMUX는 두 선택선 s2, s3와 y0부터 y15까지 총 16개의 출력을 4개씩 가지고 있다. 예를 들어, 두 번째 deMUX를 살펴보면, 이 deMUX는 첫 번째 deMUX로부터 x0를 입력 받으며, 두 선택선 s2와 s3의 조합에 따라 각 조합에 해당하는 결과인 y0, y1, y2, y3 중 하나가 출력되는 방식이다. 이를 진리표로 나타내면 다음과 같다.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S0** | **S1** | **S2** | **S3** | **Y0** | **Y1** | **Y2** | **Y3** | **Y4** | **Y5** | **Y6** | **Y7** | **Y8** | **Y9** | **Y10** | **Y11** | **Y12** | **Y13** | **Y14** | **Y15** |
| **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** |

**이렇게 완성된 진리표를 바탕으로 verilog에서 작성한 디자인 코드는 다음과 같다.**

**`timescale 1ns / 1ps**

**module inv (**

**input s0,s1,s2,s3,**

**output y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15**

**);**

**assign y0 = ~s0 & ~s1 & ~s2 & ~s3;**

**assign y1 = ~s0 & ~s1 & ~s2 & s3;**

**assign y2 = ~s0 & ~s1 & s2 & ~s3;**

**assign y3 = ~s0 & ~s1 & s2 & s3;**

**assign y4 = ~s0 & s1 & ~s2 & ~s3;**

**assign y5 = ~s0 & s1 & ~s2 & s3;**

**assign y6 = ~s0 & s1 & s2 & ~s3;**

**assign y7 = ~s0 & s1 & s2 & s3;**

**assign y8 = s0 & ~s1 & ~s2 & ~s3;**

**assign y9 = s0 & ~s1 & ~s2 & s3;**

**assign y10 = s0 & ~s1 & s2 & ~s3;**

**assign y11 = s0 & ~s1 & s2 & s3;**

**assign y12 = s0 & s1 & ~s2 & ~s3;**

**assign y13 = s0 & s1 & ~s2 & s3;**

**assign y14 = s0 & s1 & s2 & ~s3;**

**assign y15 = s0 & s1 & s2 & s3;**

**endmodule**

**16개의 출력 y0~y15에 대해 그에 대응하는 4개의 선택선의 조합을 할당하였다. 다음으로, 4개의 선택선이 일정한 간격으로 변화하도록 시뮬레이션 코드를 작성하면 다음과 같다.**

**`timescale 1ns / 1ps**

**module inv\_tb();**

**reg s0,s1,s2,s3;**

**wire y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15;**

**inv u\_inv(**

**.s0(s0),**

**.s1(s1),**

**.s2(s2),**

**.s3(s3),**

**.y0(y0),**

**.y1(y1),**

**.y2(y2),**

**.y3(y3),**

**.y4(y4),**

**.y5(y5),**

**.y6(y6),**

**.y7(y7),**

**.y8(y8),**

**.y9(y9),**

**.y10(y10),**

**.y11(y11),**

**.y12(y12),**

**.y13(y13),**

**.y14(y14),**

**.y15(y15));**

**initial begin**

**s0 = 1'b1;**

**s1 = 1'b0;**

**s2 = 1'b1;**

**s3 = 1'b0;**

**end**

**always @(s0 or s1 or s2 or s3) begin**

**s0 <= #20 ~ s0;**

**s1 <= #30 ~ s1;**

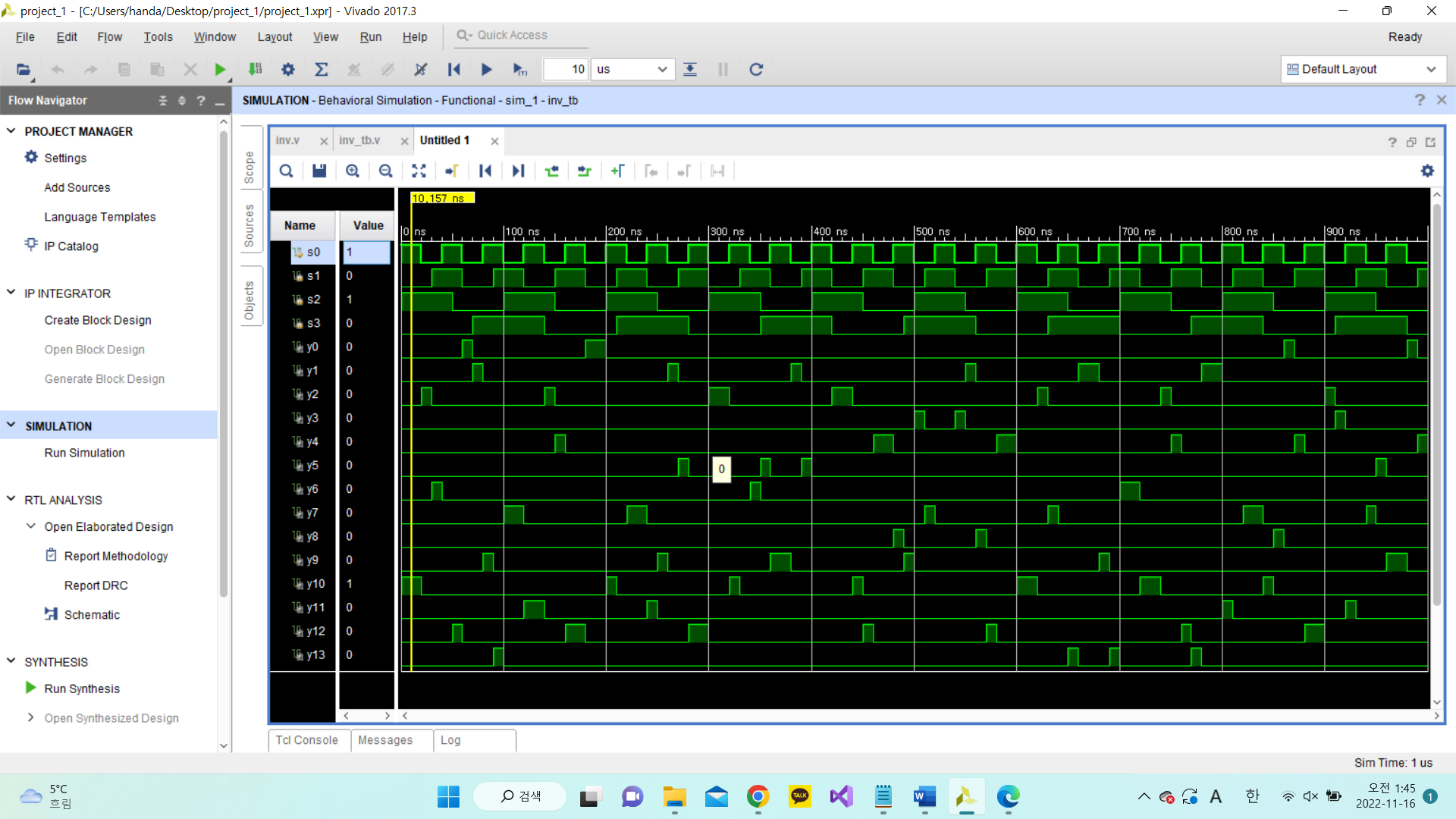
**s2 <= #50 ~ s2;**

**s3 <= #70 ~ s3;**

**end**

**endmodule**

**이 시뮬레이션 코드를 통해 우리는 다음과 같은 시뮬레이션 결과를 얻을 수 있다.**



**위 시뮬레이션에서 볼 수 있듯이 s0 =1, s1 = 0, s2 = 1, s3 = 0인 상황에서 y10의 값이 1이 되고 나머지 모든 출력들의 값은 0이 된다. 즉, 4\*16 디코더에서 y10이 출력되었다고 할 수 있다. 이는 앞서 작성한 진리표와 동일한 결과이므로 1 to 4 deMUX를 통해 4\*16 디코더가 올바르게 설계되었음을 확인할 수 있다.**

**9.**

**9주차 실험에서는 encoder, decoder, multiplexer, demultiplexer, BCD to decimal decoder의 개념에 대해 배우고 이를 직접 설계하였다. Encoder는 2n개의 입력을 받고 n개의 출력 값을 결정한다. 이번 실험에서는 4\*2 encoder를 실험하였다. 하지만 4\*2 encoder는 16가지의 입력형태 중 단 4개의 입력형태만 유효하다는 한계가 있었기 때문에 이를 보완하기 위해 4\*2 priority encoder도 추가적으로 설계하였다. Priority encoder는 기존 인코더와 다르게 입력들에 우선순위가 부여되어 4\*2 우선순위 인코더의 경우 16가지의 입력형태를 모두 사용할 수 있었다. Decoder는 인코더와 반대의 기능을 수행하는 장치로, n개의 입력을 받아 2n개의 출력 중 하나를 활성화하는 장치이다. 디코더에는 active-high 디코더와 active-low 디코더가 있는데, active-high decoder는 입력의 조합에 따라 특정 출력의 값이 1이 되고 나머지 출력 값은 0이 되는 디코더이고, active-low decoder는 입력의 조합에 따라 특정 출력의 값이 0이 되고 나머지 출력 값이 1이 되는 디코더를 의미한다. Decoder의 한 종류로 BCD to decimal decoder를 설계하는 실습도 진행하였는데, 이는 BCD 코드를 10진수로 변환하는 장치이다. 실험에서는 1~9에 대해서 9개의 출력을 할당하였고, BCD 코드를 디코더에서 입력 받으면 그 코드에 해당하는 출력의 값이 1이 되어 10진수를 표현하도록 하였다. Multiplexer는 입력선, 출력선, 선택선으로 구성되어 있는데, 이 실험에서는 4개의 입력선과 두개의 선택선, 그리고 하나의 출력선으로 이루어진 4 to 1 line MUX를 사용하였다. 두 선택선의 조합에 따라 그에 대응하는 입력선의 값이 활성화되면 출력의 값도 활성화된다. 즉, 선택선의 조합에 따라 입력이 선택된다고 할 수 있다. 마지막으로 deMultiplexer는 Multiplexer와 반대의 기능을 수행한다. 이번 실험에서는 하나의 입력선과 두 개의 선택선, 그리고 4개의 출력선으로 이루어진 1 to 4 line deMUX를 사용하였는데, 두 선택선의 조합에 따라 입력 받은 데이터가 어떤 출력선을 통해 출력될 것인지 결정된다. 이는 2\*4 디코더와 유사한 기능을 수행한다는 것을 알 수 있다.**

**10.**

**9주차 예비보고서에서 조사한 것처럼 디코더를 활용하여 ROM을 설계할 수 있었다. 하지만 ROM으로 조합 회로를 실현할 경우 조합 회로에서 사용하지 않는 don’t care 상태가 발생하고, 이는 이용할 수 있는 ROM의 기억 영역이 적어지므로 소자의 낭비로 이어진다. 이러한 단점을 보완하기 위해 사용할 수 있는 게 programmable logic array(PLA)이다. PLA는 ROM과 유사하지만 변수들 전체를 디코딩 하지도 않고, ROM처럼 모든 최소항을 만들지 않는다. PLA는 디코더가 AND 게이트의 집합으로 이루어져 있고, 입력 변수들의 곱의 항을 만들 수 있다. PLA 내부의 AND 게이트와 OR 게이트는 초기에는 퓨즈들로 연결되어 있는데, 사용자가 필요로 하지 않는 퓨즈는 끊고 사용자가 필요로 하는 퓨즈는 남겨둘 수 있기 때문에, 많은 입출력을 포함하는 조합회로를 설계할 때 PLA가 주로 사용된다.**