9주차 결과보고서

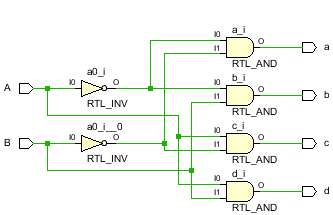
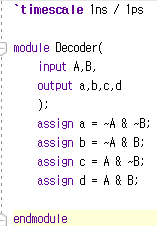
전공: 컴퓨터공학과 학년: 2학년 학번: 20191619 이름: 이동석

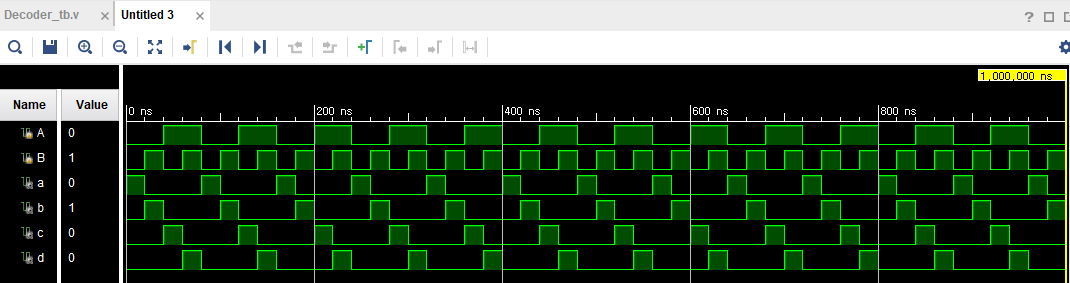
**1. Decoder**

실습에서 진행한 디코더는 2-bit의 이진코드 입력을 받고 4-bit의 출력을 하는 2 to 4 디코더이다. Active-High-level로 실습을 진행하였으며, 보고서에서는 추가적으로 AND게이트뿐 아니라 NAND게이트를 사용한 active-low 2가지 방법을 정리한다. Enable은 추가하지 않았다.

2-bit의 입력은 00, 01, 10, 11로 총 4가지의 경우가 존재한다. 이를 바탕으로 AND게이트로 구성된 active-high는 00이 들어오면 a를 1로출력, 01이 들어오면 b를 1로출력, 10은 c를 1로출력, 11은 d가 1로 출력되도록 코딩하였다. 반면, NAND게이트로 구성된 active-low는 입력은 동일하지만, 각 상황에서 a,b,c,d가 1이 아닌 0으로 출력되도록 코딩하였다.

1.1 AND





시뮬레이션을 확인해보면 의도한 대로 00일 때 a가 1을 출력하고, 01일 때 b가 1, 10일 때 c가 1을 출력, 11일 때 d가 1을 출력함을 알 수 있다. 이를 바탕으로 아래의 진리표를 작성할 수 있다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | a | b | c | d |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

작성된 진리표를 바탕으로 k-map을 작성할 수 있다.

텍스트, 낱말맞추기게임, 일이(가) 표시된 사진

자동 생성된 설명왼쪽 k-map을 바탕으로 각 불 표현식을 구할 수 있다.

a = A’B’

b = A’B

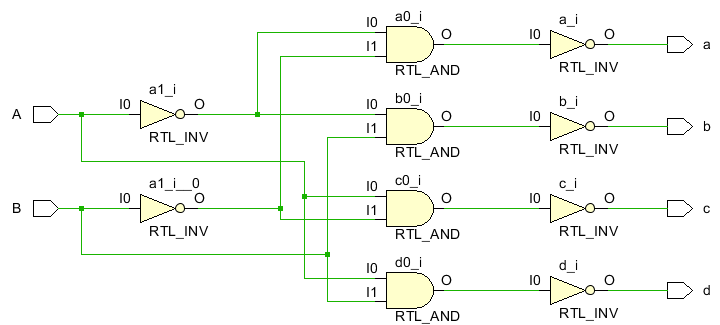
c = AB’

d = AB

간단한 형태의 카르노맵을 작성할 수 있으며, 코드와 동일하다.

1.2 NAND

텍스트이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명

시뮬레이션을 확인해보면 의도한 대로 00일 때 a가 0을 출력하고, 01일 때 b가 0, 10일 때 c가 0을 출력, 11일 때 d가 0을 출력함을 알 수 있다. 이를 바탕으로 아래의 진리표를 작성할 수 있다. 이때, AND게이트로 구성한 회로와 NOT의 관계에 있음을 알 수 있다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | a | b | c | d |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

진리표를 바탕으로 아래의 k-map을 작성할 수 있다.

텍스트, 낱말맞추기게임, 일이(가) 표시된 사진

자동 생성된 설명이때, 서로 NOT의 관계에 있음을 이용해 위에서 사용했던 k-map을 그대로 사용해 구할 수 있다. 이번에 1이 아닌, 0인 부분을 묶어 식을 작성하면 된다.

a = A + B = (A’B’)’

b = A + B’ = (A’B)’

c = A’ + B = (AB’)’

d = A’ + B’ = (AB)’

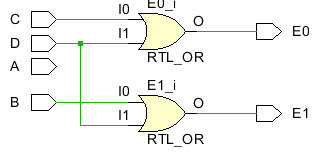
간단한 형태의 카르노맵을 작성할 수 있으며, 코드와 동일하다.

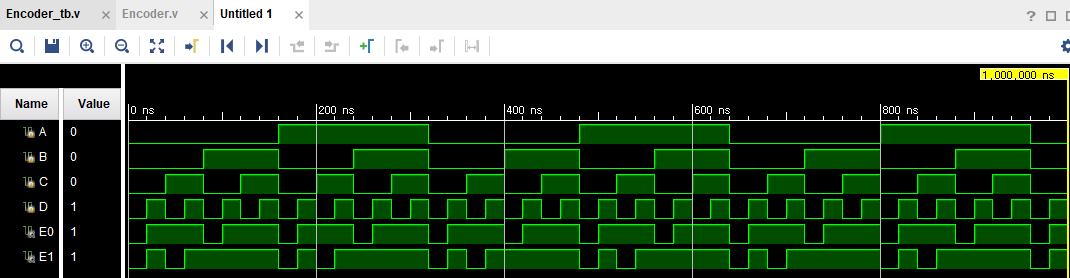
**2. Encoder**

실습에서 진행한 인코더는 2 to 4 디코더와 정반대의 기능을 하는 4 to 2 인코더이다. 4-bit의 입력을 받아 2-bit 출력을 한다. 이때 4-bit의 16개 입력을 모두 고려하는 것이 아니라, 다음 4가지 경우 1000, 0100, 0010, 0001만 생각한다. 곰곰히 생각해보면 이 4가지 경우는 앞서 AND게이트로 구현한 디코더의 출력 결과임을 알 수 있다. 이를 통해 디코더와 인코더의 관계를 쉽게 확인할 수 있다.

4가지 입력에서 출력의 결과는 AND게이트로 구현한 디코더의 입력값이다. 따라서, 1000이 들어오게 되면 출력은 00을 하며, 0100은 01, 0010은 10, 0001은 11을 출력하도록 코드를 작성하였다.

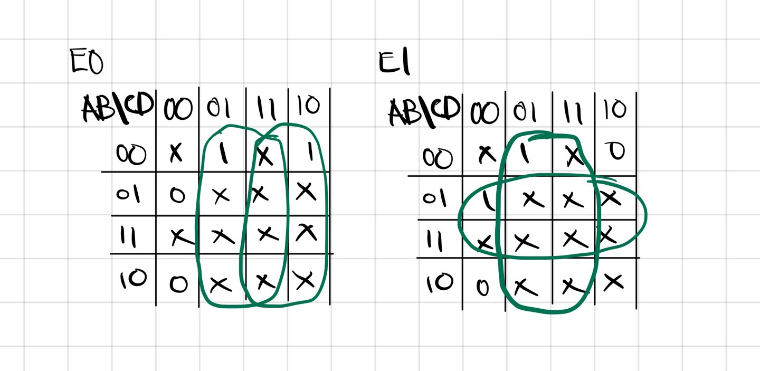
**텍스트이(가) 표시된 사진

자동 생성된 설명**

****

시뮬레이션 결과 의도한대로 결과가 출력되었다. 1000이 들어오게 되면 출력은 00을 하며, 0100은 01, 0010은 10, 0001은 11을 출력한다. 그 외의 경우도 시뮬레이션 상에서 나타나지만, 실습에선 고려하지 않았다. (입력은 다르지만, 출력이 동일한 결과가 생긴다.) 이를 바탕으로 아래의 진리표를 작성할 수 있다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E0 | E1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |

 위 진리표를 바탕으로 k-map을 작성할 수 있다.

E0 = C + D

E1 = B + D

마찬가지로 간단한 형태의 카르노맵을 작성할 수 있으며, 코드와 동일하다.

**3.**

4 to 2 인코더에서는 앞서 설명했듯 4가지의 입력만 고려한다. 이 때문에, 시뮬레이션 상에도 확인할 수 있듯이 서로 다른 입력에 대해서 같은 출력이 나타나는 상황이 생긴다. 예를 들어 입력 A와 D가 모두 1이면, 출력을 00을 해야하는지, 11을 해야하는지 애매모호한 상황이 생긴다. 주로 나머지 입력형태에 대해서 2 to 4 인코더는 출력값을 Don’t care로 처리한다. (앞서 OR게이트로 구현한 회로는 Don’t care가 아닌 값으로 직접 채울 수 있다.) 아래의 진리표는 Don’t care로 표시한 진리표이다. 이런 단점을 보완하기 위해 아래에 나올 Priority Encoder를 사용한다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E0 | E1 |
| 0 | 0 | 0 | 0 | x | x |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | x | x |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | x | x |
| 0 | 1 | 1 | 0 | x | x |
| 0 | 1 | 1 | 1 | x | x |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 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 |

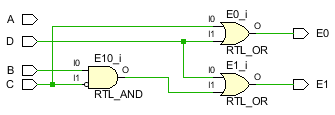
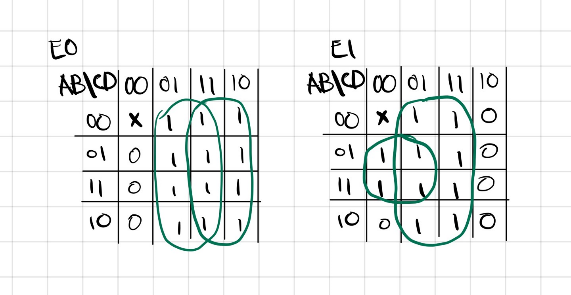
**4. Priority Encoder**

우선순위 인코더는 앞선 상황과 같이 1이 여러 개 입력될 경우가 생길 때 사용한다. 4-bit입력에 대해 우선순위를 먼저 정해준다. 예를 들어 D > A > B > C로 생각해보자. 일반적으로는 내림차순 혹은 오름차순으로 우선순위가 정해진다. 이때, 다음 진리표를 작성 할 수 있다. 주로 동작여부를 제어하는 NR(No Request)을 넣기도 한다. (0000에 대해 NR은 1의 값을 출력하고 이외의 경우 모두 0을 출력한다. 때문에, NR은 주로 NR = A + B + C + D로 표현된다.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E0 | E1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| x | 1 | 0 | 0 | 0 | 1 |
| x | x | 1 | 0 | 1 | 0 |
| x | x | x | 1 | 1 | 1 |

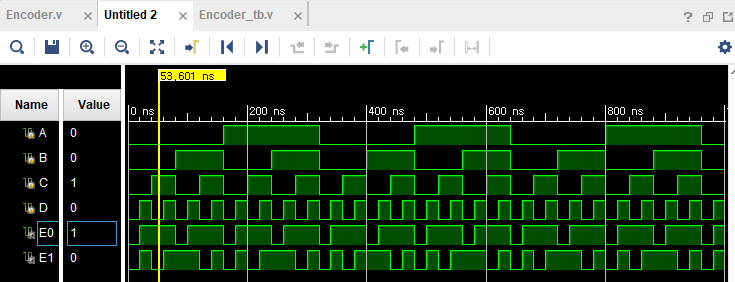
D가 가장 최우선 비트이기 때문에 D가 1이라면, 나머지 3-bit A,B,C가 어떤 값이 든 상관없이 0001과 동일한 출력을 하도록한다. 마찬가지로, C는 D가 0일 경우 다음 우선순위를 가지므로 A,B에 값과 상관없이 0010과 동일한 출력을 한다. B는 A보다 높은 우선순위를 가지므로, A에 상관없이 0100과 동일한 출력을 하며, A는 동일하다.

이를 바탕으로 k-map을 작성할 수 있다.



E0 = C + D , E1 = D + BC’

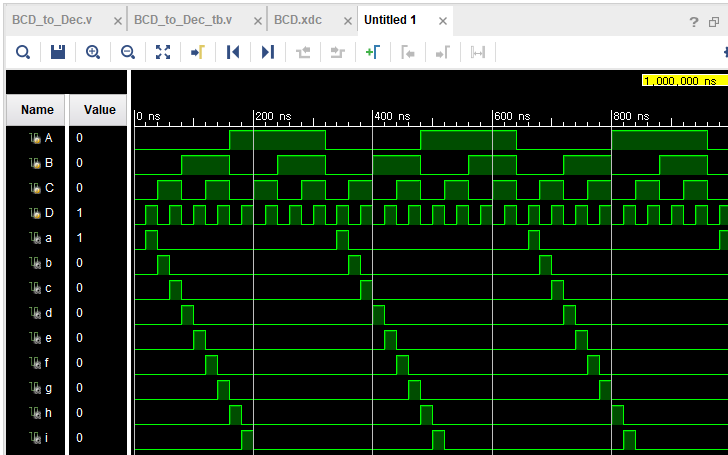
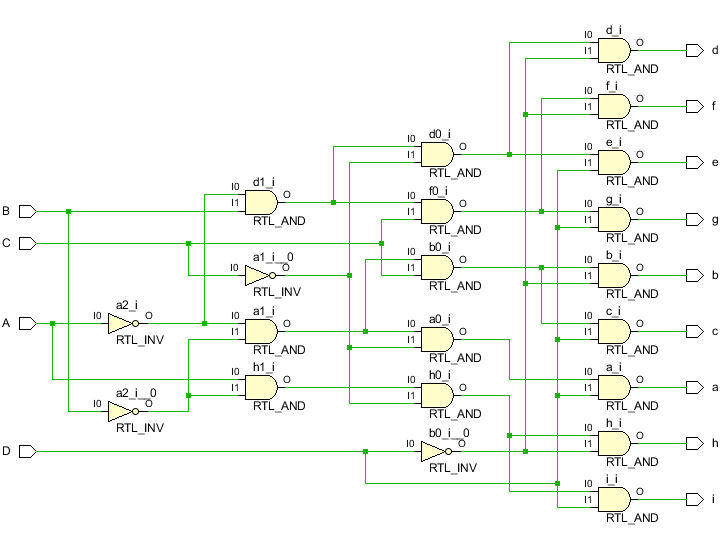
간단한 형태의 카르노맵을 작성할 수 있다. 이를 바탕으로 비바도에서 시뮬레이션을 확인하면 다음과 같이 의도한대로 출력결과가 나오는 것을 알 수 있다.



**5. BCD to Decimal Decoder**

BCD to Decimal Decoder는 4-bit 이진수를 십진수로 변환하여 출력한다. 4-bit는 0000부터 1111, 0~15까지 있다. 하지만 이번 실습에서는 1부터 9까지만 출력하는 회로를 구현했다. 각 출력 a~i는 1~9를 나태내고 각 출력이 1이 될 때 그 값을 의미한다. 예를 들어 a의 출력값이 1이라면 이는 십진수 1을 의미한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

시뮬레이션을 확인해보면, 0000~1111까지 입력이 들어오게된다. 실습에서 고려한 부분은 0001부터 1001이다. (이외의 부분은 시뮬레이션 상에서 출력되지만, 무시한다.) 먼저, 0001이 들어올 경우 a만 1로 출력된다. 0010은 십진수로 2를 의미하며, b만 1로 출력된다. c역시 0011일 때 1을 출력한다. 같은 방법으로, 0100, 0101, 0110, 0111, 1000, 1001 순서대로 십진수 4,5,6,7,8,9 값을 가지며 출력 역시 d,e,f,g,h,i만 1씩 출력됨을 확인할 수 있다. 이를 바탕으로 진리표를 작성할 수 있다. 진리표에서 1~9 이외의 출력 부분은 don’t care로 작성했다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | a | b | c | d | e | f | g | h | i |
| 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 |

이를 바탕으로 k-map을 작성하면 다음과 같다.

텍스트, 낱말맞추기게임, 영수증이(가) 표시된 사진

자동 생성된 설명

실습 때 작성한 코드보다 조금 더 최소화가 가능하다.

a = A’B’C’ , b = B’CD’, c = B’CD, d = BC’D’ or A’C’D’ , e = BC’D, f = BCD’, g = BCD, h = AD’, i = AD 로 불 표현식을 정할 수 있다.

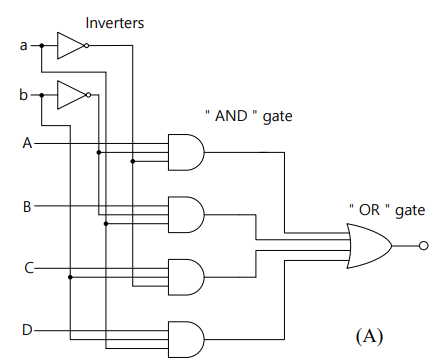
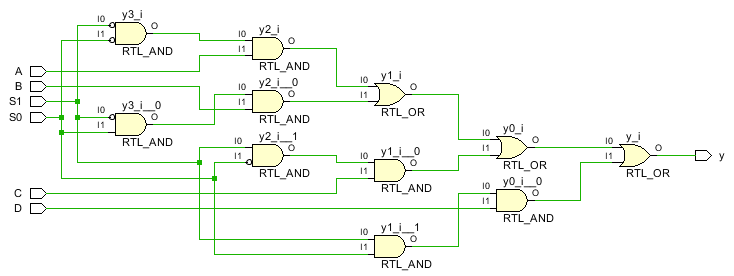
**6. Encoder와 Decoder 응용**

인코더는 주로 정보의 보안이나 파일의 압축, 표준화에 사용된다. 예로 십진수를 컴퓨터가 이해할수 있는 이진수로 변환하는 것을 말한다. 컴퓨터 공학과에서 가장 쉽게 볼 수 있는 인코딩은 바로 ASCII코드이다. 아스키 코드는 표준을 정립하기 위해 사용했다. 또한, 이 외에도 키보드의 타자기, 비밀번호와 통신에서의 보안, 동영상과 사진 압축 등 여러 분야에서 사용된다. 최근 자주 사용되는 QR코드 역시 인코딩의 일종으로 볼 수 있다.

디코딩은 인코딩과 뗄레야 뗄 수 없는 관계이다. 모든 인코딩 된 것들은 디코딩을 통해 본래의 정보로 변환하기 때문이다. 이진수를 다시 십진수로 변환하고, 아스키 코드로 인코딩 된 문자를 다시 사람이 알아 볼 수 있는 문자로의 변환, 압축된 동영상 풀기, 보안 해제 등의 분야에 응용된다. 마찬가지로 QR코드를 디코딩하는 것 등 특히, 저번시간에 배운 7-Segment도 디코더의 일종으로 엘레베이터나 디지털 시계에 사용된다.

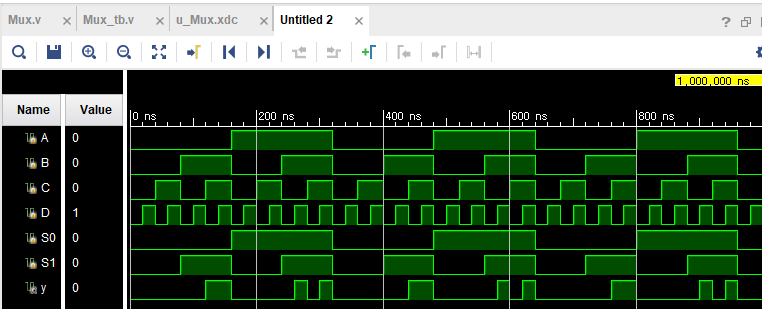
**7. 4 to 1 line Mux**

Mux는 스위치 처럼 여러 아날로그나 디지털 입력 신호들 중 하나를 선택하여 하나의 단일 출력선으로 내보낸다. 이번 실습에서는 4-bit의 디지털 입력신호를 하나의 출력선을 가지는 4 to 1 Mux를 구현하였다. Mux의 구조는 왼쪽 그림의 9주차 강의자료를 바탕으로 하여 코드를 작성하였다. 이때 y = S1’S0’A + S1’S0B + S1S0’C + S1S0D로 코드를 작성했다.

****

**텍스트이(가) 표시된 사진

자동 생성된 설명**

****

Mux는 2개의 Select Signal이 있다. 이때 시그널의 조합으로 가능한 경우는 총 4가지이다. 시그널이 각각 00,01,10,11일 때(S1S0 이다.) 출력은 순서대로 A,B,C,D가 되어야한다. 시뮬레이션에서 각 시그널의 경우에 따른 y의 출력 값이 입력 값 A,B,C,D 중 하나로 결정되는 것이 되는 것을 확인할 수 있다. 이런 특징 때문에 Data-Selector 또는 Multi-way selector swtich라고도 불린다. 시그널의 4가지 경우에 대한 진리표를 don’t care를 포함해 아래와 같이 작성할 수 있다.

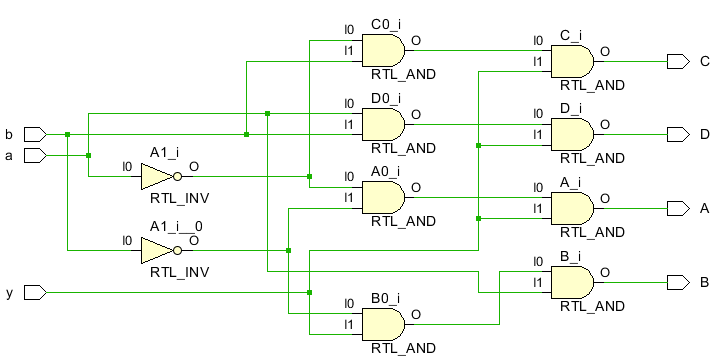
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| S1 | S0 | A | B | C | D | y |
| 0 | 0 | A | X | X | X | A |
| 0 | 1 | X | B | X | X | B |
| 1 | 0 | X | X | C | X | C |
| 1 | 1 | X | X | X | D | D |

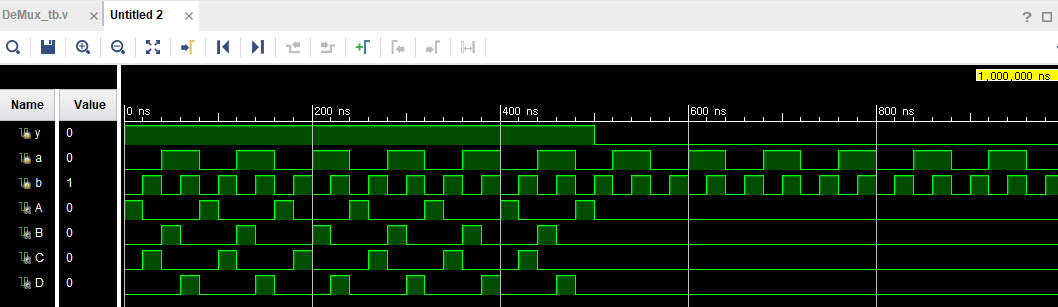
시뮬레이션 결과를 통해 작성한 진리표가 의도한 바와 같이 작성되었다. 이론상으로 생각했던 불 표현식 y가 y = S1’S0’A + S1’S0B + S1S0’C + S1S0D의 불표현식을 가진다는 것을 실험적으로 확인했다.

**8. 1 to 4 line deMux 와 4 to 16 Decoder**

DeMux는 Encoder와 Decoder의 차이처럼 Mux와 반대되는 개념이다. 한 개의 입력이 들어왔을 때 시그널에 따라 여러 개의 출력들 중 하나를 선택한다. 이런 특징으로 Data Distributor로 불린다. 실습에서는 1-bit 입력이 들어왔을 때 4개의 출력선을 가지는 1 to 4 DeMux를 구현하였다. 마찬가지로 9주차 강의자료를 참고하여 구현하였다. 이때 각 A,B,C,D는 A = a’b’y , B = b’ya, C = ‘aby, D = aby 로 코드를 작성했다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

****

DeMux 역시 2개의 Select Signal이 있다. 마찬가지로 시그널의 조합으로 가능한 경우는 총 4가지이다. 시그널이 각각 00,01,10,11일 때(ab이다.) y가 0이라는 것은, Mux에서 출력 값 A,B,C,D중 어떤 값이 0을 출력했다는 것이다. 따라서, DeMux에서는 어떤 경우가 되더라도 y가 0일 때는 항상 A,B,C,D가 0이된다. 반면, y가 1이라는 것은 Mux의 출력에 해당하는 부분이 1임을 의미한다.

Mux에서 A를 출력하는 경우엔 00이었으므로, 마찬가지로 DeMux에서는 시그널이 00이 들어오면 A를 출력한다. 마찬가지로, 01일 땐 B, 10일땐 C, 11일땐 D가 된다. 이때, 출력되는 값은 y값과 동일함을 알 수 있다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| b | a | y | A | B | C | 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 |

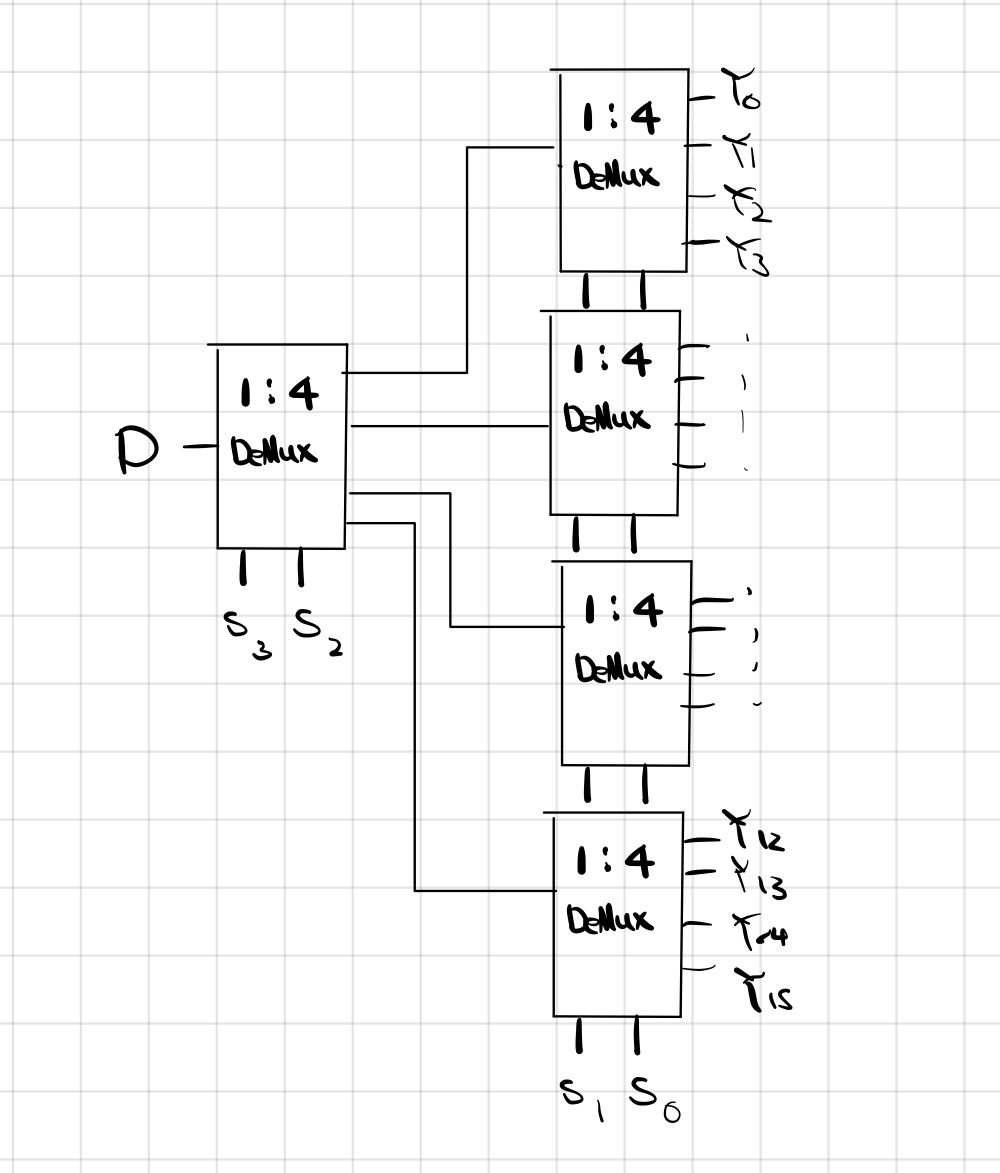
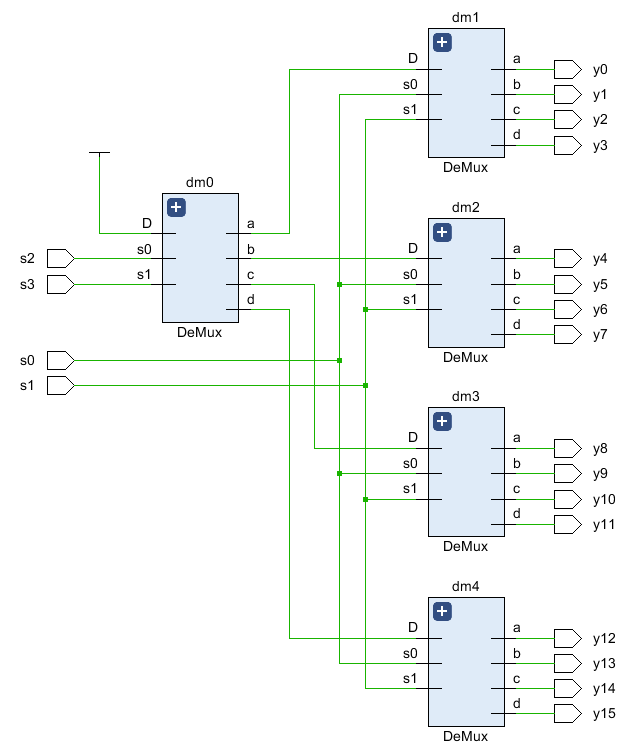
시뮬레이션 결과를 통해 작성한 진리표가 의도한 바와 같이 작성되었다. 이론상으로 생각했던 불 표현식 A,B,C,D가 모두 올바르게 작성되었음을 실험적으로 확인할 수 있다.

배운 내용을 바탕으로 DeMux를 이용해 4 to 16 Decoder를 구현할 수 있다. DeMux는 특정상황에서 Decoder와 똑같은 동작을 한다. 특정상황은 디코더에 EN이 추가되었을 경우다. EN을 DeMux의 y와 같게 생각하고, 두 시그널을 디코더의 2개의 입력으로 생각하면 된다. 따라서, 1 to 4 line DeMux로 구현할 4 to 16 디코더는 2 to 4 디코더로 구현한 것과 동일한 회로의 모습을 가진다. 또한, 이는 4 to 16 디코더가 1 to 16 line DeMux와 동일한 회로 구조를 가진다고도 해석할 수 있다.

아래그림에서 1 to 16 DeMux에서 처음 입력신호 D를 EN으로 생각하고, 출력되어지는 4개의 bit 역시 EN으로 생각하며, 4개의 시그널 비트를 디코더에서의 입력으로 생각하면, 4 to 16 디코더의 역할을 할 수 있다. 이를 바탕으로 Vivado에서 다음과 같이 코드를 작성할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명 ****

**9. 결과 및 논의**

예비보고서를 작성하면서 긴가민가 했던 부분이 직접 실습을 통해 회로를 작시키고, 결과를 보고서로 정리하면서 해소가 되었다.

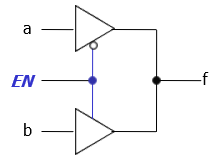
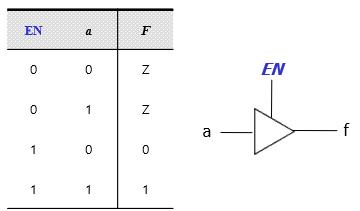
인코더의 Active-high/low 방식 두 방식 모두 회로를 구현해보았다. 이때, Active-low방식을 선호하는데, 이는 노이즈에 대한 이슈 때문이다. 그러나, 노이즈에 대한 정확한 의미를 이해하지 못했는데, 실습 중 노이즈에 대한 부분을 실험장님께서 다시한번 설명해주셔서 쉽게 이해할 수 있었다. 인코더와 디코더의 각각의 출력과 입력에 대해 정리하면서 서로의 관계를 눈으로 확인할 수 있었다. 또한, 우선순위 인코더의 사용이유에 대해서도 확인 가능했다.

BCD to Decimal Decoder는 과거 진행했던 여러 BCD 디코더와 유사했다. 따라서 실습에 큰 어려움은 없었다. 하지만, 결과보고서에서 직접 k-map을 통해 불 표현식을 구해보았을 때, 단순히 그 항에 대한 minterms로 구현한 것 보다 더 minimized가 가능했다. ( 1~9까지만 생각했을 경우)

마지막으로, 스위치 역할을 하는 Mux와 분배기 역할을 하는 DeMux는 인코더와 디코더의 역할과 조금헷갈렸지만, 이번 실습을 통해 차이점을 조금이나마 이해할 수 있었다. 특히, 추가적으로 4 to 16 디코더를 구현해보면서, DeMux와 디코더가 언제 동일한 작동을 하는지 명확히 확인할 수 있었다. 또한, 둘의 출력과 입력값을 비교해보며 결과보고서를 작성하여 관계를 눈으로 확인할 수 있었다. 또한, 비바도의 문법을 조금 공부할 수 있었다.

**10. 추가이론**

Three-state-gates (buffer)는 Z(unknown)라는 상태를 가질 수 있다. 이는 시스템 사이를 이동하는 신호에 자주사용한다. 이때 상태 Z는 출력신호선에서 전기적으로 절연된 상태를 말하며, high와 low 모두 아닌 상태이다. 이를 High-Impedance라 부른다.



그림과 표를 보면, EN이 1일 경우 F의 상태가 결정된다. (NOT과 다른 기호이다.) 이를 Mux처럼 사용할 수 있다. 오른쪽그림은 2 to 1 Mux와 비슷한 동작을 한다. EN값에 따라 f의 값이 a혹은 b / Z로 결정된다. 또한, 이렇게 Three-state-gates로 논리 값이 전달되는, 데이터가 전송되는 일련의 선을 BUS라 부른다. 다음은 BUS와 디코더를 사용해 구현한 MUX이다.

