9주차 결과보고서

전공: 컴퓨터공학과 학년: 3학년 학번: 20191599 이름: 송경호

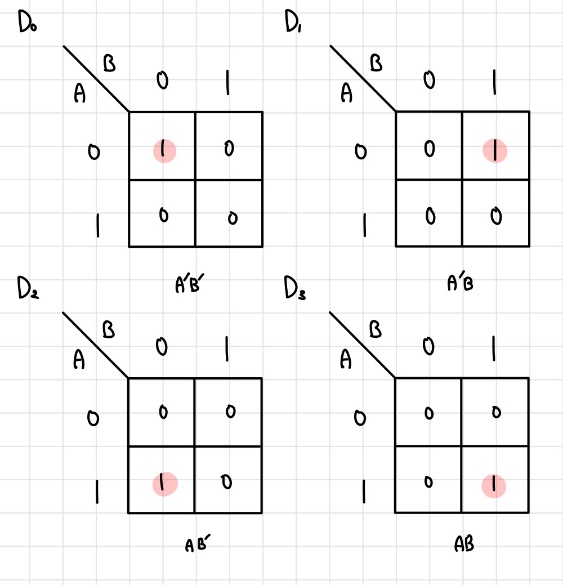
**1.**

**1-1) Active High Decoder**

카르노맵을 바탕으로 2 to 4 Decoder의 Boolean function을 이끌어 내기 위해선 먼저 진리표의 작성이 선행되어야 한다. 2 to 4 Decoder는 2 bit 입력에 대해 22 bit 출력, 즉 4 bit 출력을 하는 회로이다. 2 bit 입력은 총 4개의 경우의 수를 발생시키는데 각 경우의 수마다 하나의 출력 비트만이 1이고 나머지 비트는 0이 되는 것이 Active High Decoder이다. 따라서 진리표는 다음과 같이 작성된다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | **Output** | | | |
| **A** | **B** | **D0** | **D1** | **D2** | **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 |

예를 들어, 입력 값 01에 대하여 0100과 같이 D1 비트만 1이 들어오고 나머지 비트는 0이된다. 진리표를 바탕으로 각 입력 경우의 수에 대한 각 출력 비트 D­0 ~ D3에 대해 4개의 K-Map을 아래와 같이 작성한다.



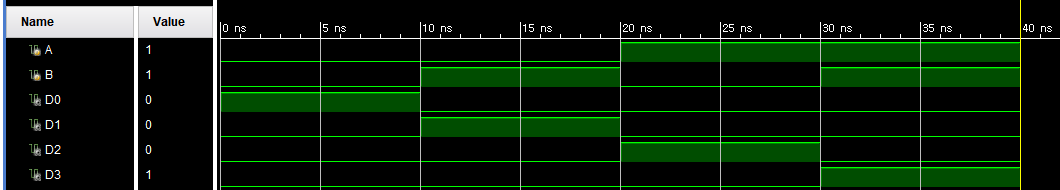
그려진 K-Map을 바탕으로 Verilog code를 작성한다. 작성된 디자인 코드와 테스트 밴치 코드는 다음과 같다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

K-Map을 통해 구한 Boolean function을 그대로 적용하여 Verilog code를 작성한다. 또한 입력 값 A, B를 20, 10ms 마다 변경하여 총 40ms의 시간동안 입력의 모든 경우의 수를 확인하고 동시에 진리표의 입력 값 변화와 같은 순서로 변화를 주어 최대한 같은 형식의 시뮬레이션 결과를 얻는다. 시뮬레이션 결과는 다음과 같다.



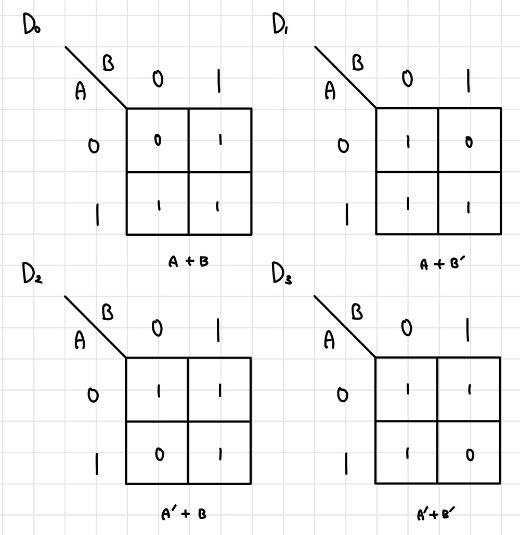
진리표와 동일하게 한 입력에 대하여 출력 비트 중 하나의 비트에만 1이 들어온 것을 확인할 수 있다.

**1-2) Active Low Decoder**

Low Decoder도 High Decoder와 마찬가지로 카르노맵을 바탕으로 Boolean function을 이끌어 내기 위해선 먼저 진리표의 작성이 선행되어야 한다. 이때 High Decoder와 달리 2 bit 입력은 총 4개의 경우의 수를 발생시키는데 각 경우의 수마다 하나의 출력 비트만이 0이고 나머지 비트는 1이 된다. 즉 High Decoder의 결과값을 부정한 형태이며 진리표는 다음과 같이 작성된다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | **Output** | | | |
| **A** | **B** | **D0** | **D1** | **D2** | **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 |

예를 들어, 입력 값 01에 대하여 1011과 같이 D1 비트만 0이 들어오고 나머지 비트는 1이된다. 진리표를 바탕으로 각 입력 경우의 수에 대한 각 출력 비트 D­0 ~ D3에 대해 4개의 K-Map을 아래와 같이 작성한다.



그려진 K-Map을 바탕으로 Verilog code를 작성한다. 작성된 디자인 코드와 테스트 밴치 코드는 다음과 같다.

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

K-Map을 통해 구한 Boolean function을 그대로 적용하여 Verilog code를 작성한다. 테스트 밴치 코드의 경우 high decoder와 동일하게 입력 값 A, B를 20, 10ms 마다 변경하여 총 40ms의 시간동안 입력의 모든 경우의 수를 확인하고 동시에 진리표의 입력 값 변화와 같은 순서로 변화를 주어 최대한 같은 형식의 시뮬레이션 결과를 얻는다. 시뮬레이션 결과는 다음과 같다.

스크린샷, 라인, 다채로움이(가) 표시된 사진

자동 생성된 설명

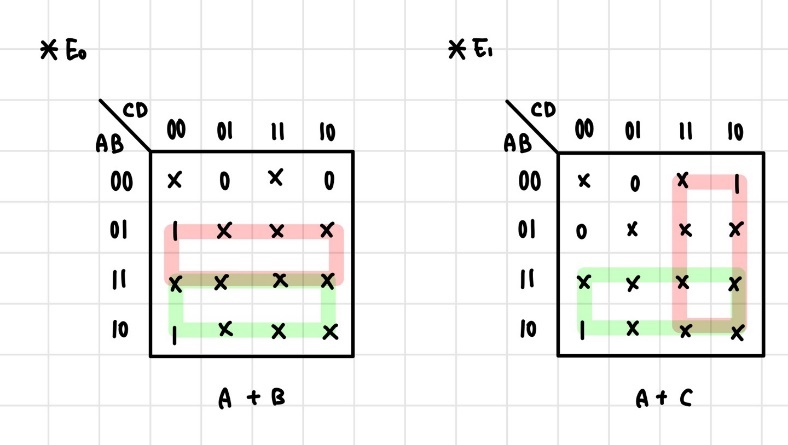
진리표와 동일하게 한 입력에 대하여 출력 비트 중 하나의 비트에만 0이 들어온 것을 확인할 수 있다.

**2.**

4 to 2 Encoder는 카르노맵을 바탕으로 Boolean function을 이끌어 내기 위해선 먼저 진리표의 작성이 선행되어야 한다. 4 to 2 Encoder는 22 bit 입력에 대하여 2 bit 출력을 발생시키는 회로이다. 또한 Encoder는 Decoder의 반대 역할을 수행하므로 Decoder의 출력에 존재하는 Encoder의 입력만이 의미를 갖게 된다. 진리표는 다음과 같이 작성된다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 |

예를 들어, 입력 값 0100에 대하여 10과 같이 High Decdoer의 반대 역할을 수행하며, High Deocder의 출력값인 0001, 0010, 0100, 1000을 제외한 입력은 결과값이 모두 Don’t Care 비트가 발생한다. 진리표를 바탕으로 각 입력 경우의 수에 대한 각 출력 비트 E0, E1에 대해 2개의 K-Map을 아래와 같이 작성한다.



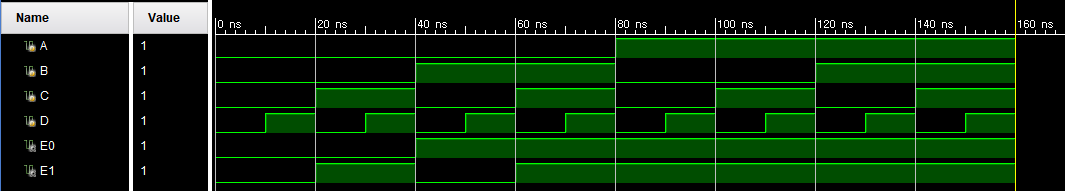
그려진 K-Map을 바탕으로 Verilog code를 작성한다. 이때 Don’t Care Bit는 1로 취급하여 Boolean Function을 계산했다. 작성된 디자인 코드와 테스트 밴치 코드는 다음과 같다.

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

K-Map을 통해 구한 Boolean function을 그대로 적용하여 Verilog code를 작성한다. 테스트 밴치 코드의 경우 A, B, C, D를 80, 40, 20, 10ms 마다 변경하여 총 160ms의 시간동안 입력의 모든 경우의 수를 확인하고 동시에 진리표의 입력 값 변화와 같은 순서로 변화를 주어 최대한 같은 형식의 시뮬레이션 결과를 얻는다. 시뮬레이션 결과는 다음과 같다.



코드에서는 따로 Don’t Care Bit를 표현할 수 없어서 모든 입력에 대한 결과값이 출력되었다. 이때 입력 비트 중 하나의 비트에만 1이 있는 0001, 0010, 0100, 1000에 대한 출력 비트만을 확인 했을 때 진리표와 동일한 결과를 확인할 수 있다.

**3.**

앞서 말했듯이, 4 to 2 Encoder의 truth table에서 입력 비트 0001, 0010, 0100, 1000에 대한 출력 비트만이 인정되고 나머지 결과값은 don’t care 비트가 된다. 이는 사실 단순히 결과값을 해석하지 않는 것이 아니라 priority encoder에 대한 진리표를 작성한 것으로, 특정 priority를 가진 입력 비트가 들어왔을 때, 더 낮은 priority를 가진 입력 비트들의 값은 무시하는 특성을 담고 있다.

위의 4 to 2 Encoder를 예시로 들자면 A, B, C, D입력 비트 순으로 높은 priority를 갖고 있다. 따라서 A = 1 인 경우 그 하위 비트인 B, C, D의 입력은 모두 무시되어 항상 11의 출력이 발생하게 된다.

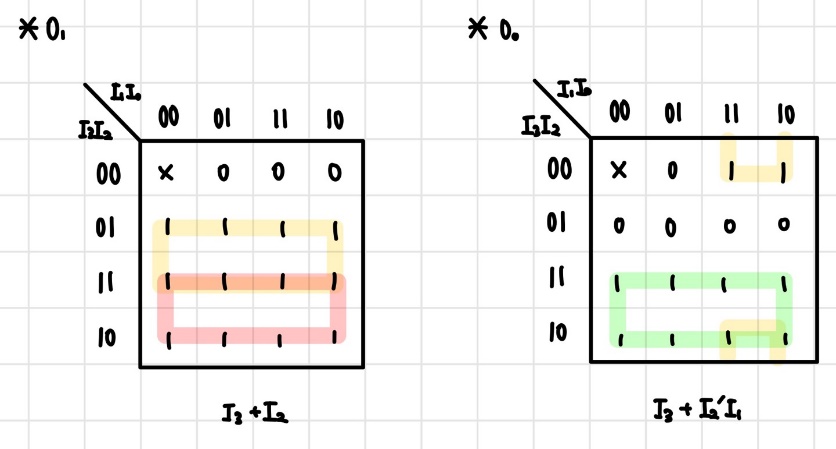
그러나 이러한 priority encoder의 특성은 simulation에서는 나타나고 있지 않는다. 예를 들어, 입력이 0111일 경우 B가 가장 높은 우선순위 비트가 되어 출력은 10이 되어야 하지만 11이 되는 것을 알 수 있다. 이는, 회로 구성 시 4가지 입력만을 고려했기 때문이다.

**4.**

16가지 입력에 대한 4 to 2 priority encoder의 진리표를 다음과 같이 구성한다. Priority는 I3, I2, I1, I0 순으로 높으며 특정 비트가 1일 때 이보다 낮은 priority bit는 모두 don’t care bit가 된다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | | | **Output** | |
| **I3** | **I2** | **I1** | **I0** | **O1** | **O0** |
| 0 | 0 | 0 | 0 | x | x |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | x | 0 | 1 |
| 0 | 1 | x | x | 1 | 0 |
| 1 | x | x | x | 1 | 1 |

진리표를 바탕으로 나타낸 카르노맵은 아래와 같다.



해당 카르노맵을 바탕으로 구성한 논리 회로이다.

텍스트, 스크린샷, 소프트웨어, 번호이(가) 표시된 사진

자동 생성된 설명

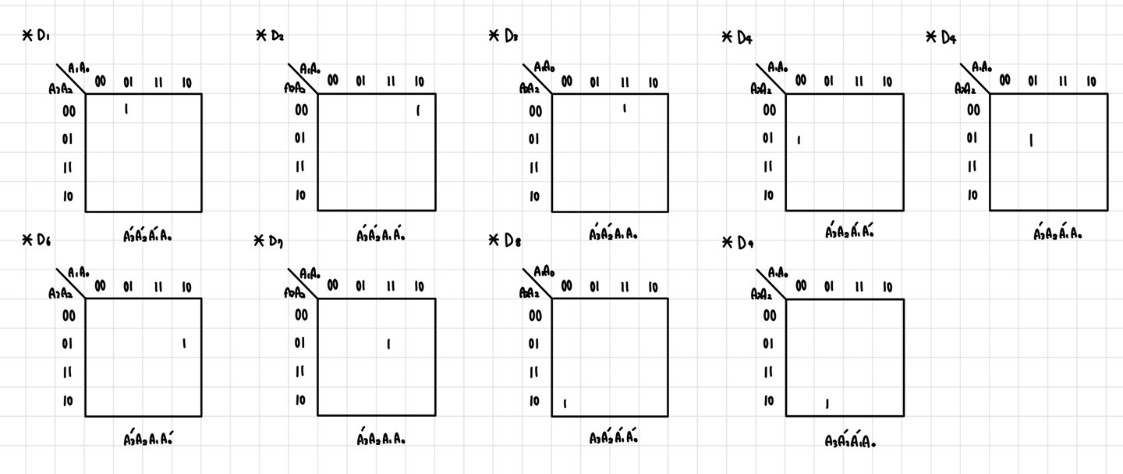
K-map을 통해 구한 Boolean Function이 그대로 적용되어 있으며 추가적으로 v bit가 존재하는데 이는 입력이 0000일 때와 0001일 때의 출력이 모두 00으로 같으므로 이를 비교하기 위한 비트이다.

**5.**

BCD to Decimal Decoder는 카르노맵을 바탕으로 Boolean function을 이끌어 내기 위해선 먼저 진리표의 작성이 선행되어야 한다. BCD to Decimal Decoder는 4 bit의 입력에 대하여 1 부터 9까지의 출력을 대응시키는 회로이다. 따라서 9개의 출력 비트를 갖게 되며 high decoder이기 때문에 특정 출력 비트가 1이 되는 경우 나머지 출력 비트는 모두 0이된다. 네 개의 입력 A3, A2, A1, A0, 가 각 각 8,4,2,1에 대응되므로 2진수에서의 10진수로의 치환을 생각하고 진리표를 다음과 같이 구성한다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | | | | **Output** | | | | | | | | |
| **A3** | **A2** | **A1** | **A0** | **D1** | **D2** | **D3** | **D4** | **D5** | **D6** | **D7** | **D8** | **D9** |
| 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

진리표에서 파란색으로 표시한 칸이 유효한 출력이며 high decoder의 형태를 띄는 것을 알 수 있다. 또한 BCD to Decimal Decoder인 만큼 나머지 입력에 대한 모든 출력은 0이된다. 진리표를 바탕으로 구성한 K-Map은 다음과 같다.



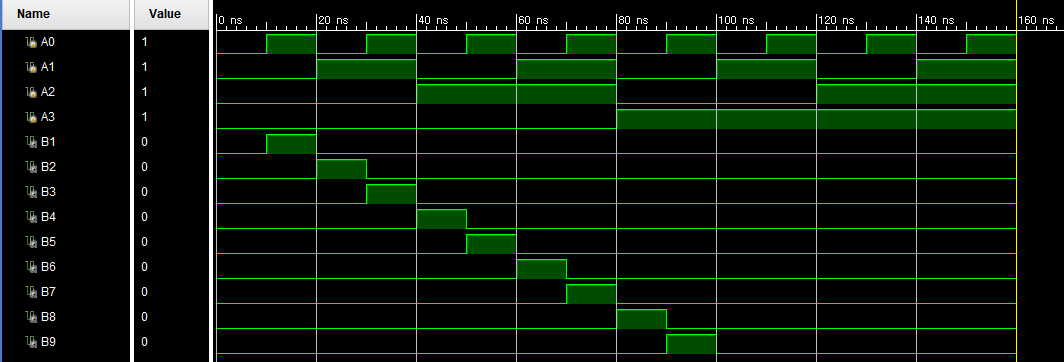
편의상 1을 제외한 칸은 모두 0이므로 표시를 하지 않았다. 위 카르노맵을 통한 Boolean Function을 바탕으로 디자인 코드와 테스트 밴치코드를 아래와 같이 작성한다.

텍스트, 폰트, 스크린샷, 문서이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

카르노맵의 Boolean Function을 그대로 적용시켰으며 위와 동일하게 입력값에 대해 80, 40, 20, 10ns마다 변화를 주어 160ns에 모든 입력 경우의 수에 대한 출력을 최대한 진리표와 동일하게 출력할 수 있도록 테스트 밴치코드를 설계했다. 아래는 시뮬레이션 결과이다.



시뮬레이션 결과와 진리표가 동일한 결과를 출력하는 것을 확인할 수 있다.

**6.**

Encoding과 Decoding은 암호화와 복호화라는 의미에 걸맞게 특정 입력, 다시 말해 정보를 다른 형태로 변환하는 처리와 변환된 정보를 다시 원래 형태로 되돌리는 처리를 말한다. Encoding의 목적은 표준화, 보안, 처리 속도 향상, 저장 공간 절약 등의 다양한 이유에서 일어난다. 따라서 Encoding과 Decoding 모두 변환의 과정이라 그 개념이 조금 애매할 수 있으나, 어떤 대상을 특정 목적을 위해 변환한다면 Encoding, Encoding된 대상을 다시 원래 형태로 변환한다면 Decoding을 의미한다.

|  |  |  |
| --- | --- | --- |
|  | **인코딩** | **디코딩** |
| 통신보안 | 암호화 | 암호 해제로 원래의 신호로 변환 |
| 아파트 층을 숫자화 | 십진수화 | 해당 층을 지목 |
| 음성 신호 처리 | 전기신호로 변환 | 물리적 신호로 변환 |
| 디지털화 | 이진화 | 아날로그로 변환 |
| 통신 라인의 특성에 맞춤 | 라인코딩 | 원래 직렬데이터로 변환 |

**7.**

4 to 1 line mux는 카르노맵을 바탕으로 Boolean function을 이끌어 내기 위해선 먼저 진리표의 작성이 선행되어야 한다. 4 to 1 line mux는 4 bit의 입력에 대하여 하나의 bit를 선정하여 출력하는 회로이다. 따라서 4 bit의 입력과 1 bit의 출력을 갖는데 이에 더해, 어떤 입력 비트를 선택할 지에 대한 결정을 해주는 select bit 2 bit가 추가되어 결과적으로 6 bit의 입력을 갖게 된다. Select bit에 따라 4 개의 입력 비트 중 어떤 비트가 전달될지 결정되며 출력 비트는 선택된 입력 비트의 값을 그대로 갖는다. 진리표는 다음과 같다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Input** | | | | **Selector** | | **Output** |
| **A** | **B** | **C** | **D** | **a** | **b** | **O** |
| 0 | x | x | x | 0 | 0 | 0(=A) |
| 1 | x | x | x | 0 | 0 | 1(=A) |
| x | 0 | x | x | 1 | 0 | 0(=B) |
| x | 1 | x | x | 1 | 0 | 1(=B) |
| x | x | 0 | x | 0 | 1 | 0(=C) |
| x | x | 1 | x | 0 | 1 | 1(=C) |
| x | x | x | 0 | 1 | 1 | 0(=D) |
| x | x | x | 1 | 1 | 1 | 1(=D) |

Selector bit에 따라 어떤 입력을 출력으로 전달할지 결정한다. 예를 들어 select bit가 00 일 경우 A를 선택하여 출력하며 이때 A bit가 0일 경우 0을 1일 경우 1을 그대로 출력하는 것을 확인할 수 있다.

4 to 1 line mux의 경우 카르노맵을 구하지 않더라도 직관적으로 Boolean function 유도가 가능하다. 우선 출력의 경우 A, B, C, D의 값 중 하나를 갖게 되므로 기본적으로 다음과 같이 구성할 수 있다.

O = A | B | C | D

그러나 이때 select bit a, b에 대해 A, B, C, D의 값이 결정 되므로 select bit와의 AND 연산을 통해 입력을 조절한다. 결과적으로 출력을 다음과 같이 유도할 수 있다.

O = (A&(~a)&(~b)) | (B&a&(~b)) | (C&(~a)&b) | (D&a&b)

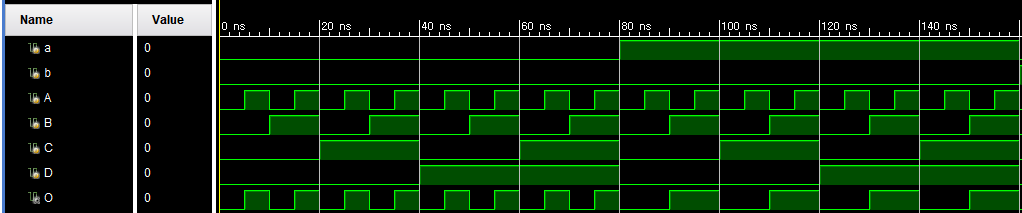
위 Boolean Function을 바탕으로 디자인 코드와 테스트 밴치코드를 아래와 같이 작성한다.

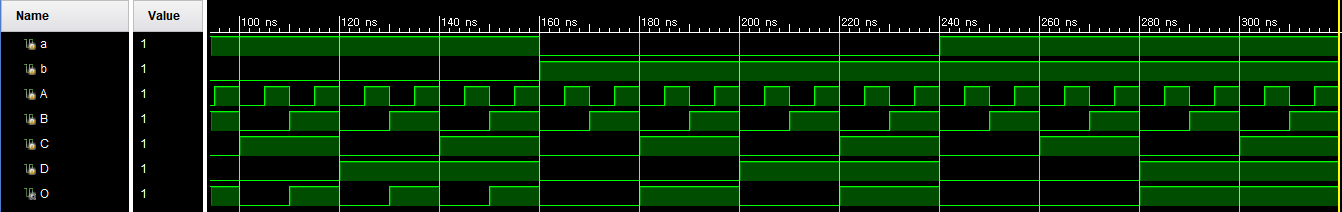
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Boolean Function을 그대로 적용시켰으며 select bit의 변화에 따른 입력값의 변화가 더 잘 드러나도록 select bit의 변화를 80, 160ms마다 입력 비트 A, B, C, D의 변화를 5, 10, 20, 40ms 마다 주어 총 320ms동안 결과를 측정했다. 시뮬레이션 결과는 아래와 같다.





변화를 더욱 잘 보이게 하고자 두 사진에 나누어 캡쳐했다.

**8.**

1 to 4 line deMux의 Boolean function을 이끌어 내기 위해선 먼저 진리표의 작성이 선행되어야 한다. 1 to 4 line mux는 1 bit의 입력에 대하여 4 bit를 출력하는 회로이다. 이때 mux의 역할을 완전히 반대로 수행하므로 4bit 출력 중 하나의 비트를 선택하여 1을 출력하고 나머지 출력 비트는 0이된다. 이때 입력 비트 외에 어떤 출력 비트를 선택할 지에 대한 결정을 해주는 select bit 2 bit가 추가되어 결과적으로 3 bit의 입력을 갖게 된다. Select bit에 따라 입력 비트가 4 개의 출력 비트 중 어떤 비트로 전달될지 결정되며 출력 비트는 선택된 입력 비트의 값을 그대로 갖는다. 진리표는 다음과 같다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Selector** | | **Output** | | | |
| **F** | **a** | **b** | **A** | **B** | **C** | **D** |
| 0 | x | x | 0 | | | |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

Selector bit에 따라 입력을 어떤 출력으로 전달할지 결정한다. 예를 들어 select bit가 00 일 경우 A 출력 비트를 선택하여 입력 비트를 그대로 전달한다. 따라서 입력 비트가 0일 경우 selector bit와 상관없이 모든 출력 비트는 0이된다.

1 to 4 line deMux의 역시 직관적으로 Boolean function 유도가 가능하다. Mux와 반대로 입력 비트 F와 selector bit간의 AND 연산으로 출력 비트를 결정할 수 있다. 유도 결과는 다음과 같다.

A = F&~a&~b

B = F&~a&b

C = F&a&~b

D = F&a&b

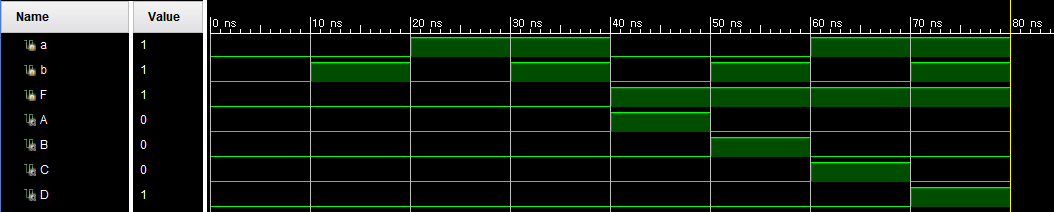
위 Boolean Function을 바탕으로 디자인 코드와 테스트 밴치코드를 아래와 같이 작성한다.

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명텍스트, 폰트, 스크린샷, 대수학이(가) 표시된 사진

자동 생성된 설명

Boolean Function을 그대로 적용시켰으며 select bit의 변화에 따른 출력값의 변화가 더 잘 드러나도록 입력 비트를 40ns마다 select bit를 변화를 20, 10ns마다 변화시켜 총 80ns동안 결과를 측정했다. 시뮬레이션 결과는 아래와 같다.



위에서 구한 1 to 4 line demux를 이용하여 4 to 16 decoder를 구현한다.1 to 4 line demux를 2중으로 겹쳐 총 5개를 통해 구현한다. 첫 단계의 1 to 4 line demux를 통해 4개의 출력을 생성하고 각 출력을 입력으로 한 1 to 4 line demux를 만들게 되면 총 16개의 출력을 가진 4 to 16 decoder를 생성할 수 있다.

디자인 코드와 테스트 벤치 코드는 아래와 같다.

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

시뮬레이션 결과는 아래와 같다.

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

진리표는 아래와 같다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Select** | | | | **Input** | **Output** | | | | | | | | | | | | | | | |
| **InA** | **InB** | **InC** | **InD** | **F** | **Out1** | **Out2** | **Out3** | **Out4** | **Out5** | **Out6** | **Out7** | **Out8** | **Out9** | **Out10** | **Out11** | **Out12** | **Out13** | **Out14** | **Out15** | **Out16** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 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 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 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 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 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 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 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 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**9.**

Decoder, Encoder, BCD to Decimal Decoder, mux, demux, 4to16decoder(using mux)를 verilog 코드로 구현하고 시뮬레이션 결과와 진리표를 비교했다. Verilog 코드에 사용된 Boolean function의 경우 K-map을 통해 구한 경우와 진리표를 바탕으로 직관적으로 코드를 얻어낸 두 경우 모두 진리표의 결과와 시뮬레이션 결과가 완정히 동일했다.

또한 구현 시, 항상 진리표와 비슷한 형태로 입력 값을 변환시켜 최대한 진리표와 동일한 형태로 시뮬레이션을 뽑아낼 수 있도록 구현했다.

결과에 대한 자세한 분석은 앞선 항목(1~8)에서 자세히 언급하였으니 생략한다.

**10.**

**10-1) mux**

mux에는 다양한 종류의 mux가 존재한다. Inverse mux가 그중 하나로, inverse mux는 고밀도의 데이터를 저밀도의 데이터로 나누어 전달하는 mux를 의미한다.

Inverse mux의 장점은 잘못된 전송의 발생에 대해 강하다는 점인데, 기존 데이터에서 하나의 값이라도 바뀔 경우 큰 혼동이 올 수 있는 고밀도의 데이터를 다양한 출력으로 저밀도의 데이터로 전송함으로써 이를 가능하게 한다.

**10-2) 데이터 통신 분야의 mux와 demux**

Network Protocol Layers 중 Transport Layer에서 데이터 통신이 일어나는 과정에서 mux와 demux가 사용된다. mux의 경우 application layer의 소켓으로부터 얻은 데이터에 헤더를 추가하여 새그먼트를 생성하고 이를 다시 network layer로 전달하는 과정을 포함한다. demux의 경우 네트워크로 받은 세그먼트를 세그먼트 내에 포함된 헤더 정보를 바탕으로 올바른 소켓에 전달한다.

mux와 demux를 통해 여러 응용 프로그램에서 네트워크를 공유하는 동시에 각각의 데이터를 정확하게 수신할 수 있도록 한다.