컴퓨터공학실험2 9주차 결과 보고서

전공: 컴퓨터공학 학년: 2학년 학번: 20201635 이름: 전찬

**0. 목차**

1. 2 to 4 Decoder의 결과 및 simulation 과정 설명

2. 4 to 2 Encoder의 결과 및 simulation 과정 설명

3. 4 to 2 Encoder에서 입력 형태 4가지를 제외한 나머지 입력 형태의 의미

4. 4 to 2 Encoder의 모든 입력 형태에서 동작하는 priority encoder의 논리 회로

5. BCD to Decimal decoder의 결과 및 Simulation 과정 설명

6. Encoder와 decoder의 주요 응용 설명

7. 4 to 1 line MUX의 결과 및 simulation 과정 설명

8. 1 to 4 line deMUX를 이용한 4 to 16 decoder 구현

9. 결과 검토 및 논의 사항

**1. 2 to 4 Decoder의 결과 및 simulation 과정**

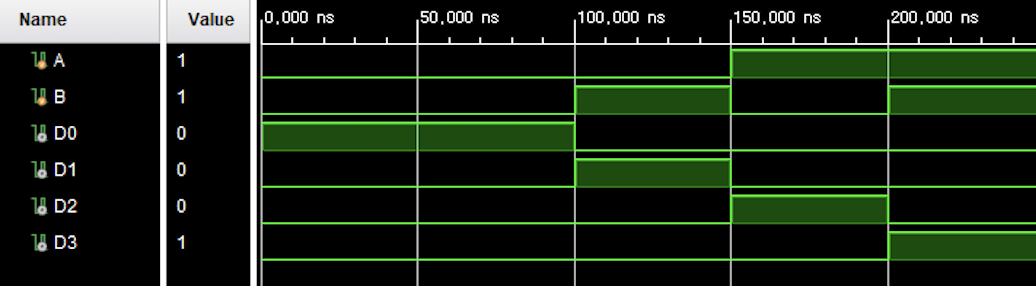
2 to 4 decoder란, 2 bit input(00, 01, 10, 11)에 대해서 각 binary input의 decimal 표현이 무엇인지 판단하는 decoder 이다. 만약 input = 10 이라면 output은 2를 의미하는 output이 High가 되며 나머지 3개의 output은 Low가 되는 형태이다. 또한 2 to 4 Decoder은 AND gate, NAND gate 두 가지 방법으로 구현할 수 있는데, AND gate를 사용하는 경우에는 active High output, NAND gate의 경우에는 active Low output(output이 Low가 되면 그 숫자임을 판단하는 형태)을 사용하게 된다.

우선 AND gate로 구현한 decoder의 truth table은 아래와 같다.(k-map은 2x2 형태에서 단순히 한 block만 색칠하게 되어, 크게 중요하지 않아 생략한다..)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 to 4 Decoder | | | | | |
| input | | output | | | |
| in1 | in2 | out0 | out1 | out2 | out3 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

<2 to 4 Decoder(AND gate)의 truth table>

또한 simulation 결과는 다음과 같다.



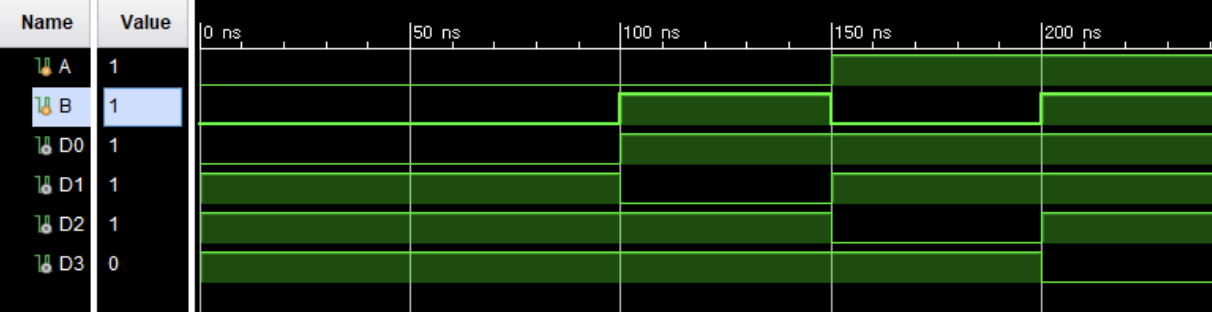
<2 to 4 decoder(AND)의 truth table>

또한 NAND gate로도 decoder을 구현할 수 있는데, 이 경우에는 위 simulation의 output에서 NOT gate를 추가해주면 된다. 예를 들어 out1 = ~in1&~in2 인데, 여기에 다시 NOT 을 추가해주어 out1 = ~(~in1&~in2) 와 같은 형태가 된다. 이에 대한 truth table은 아래와 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 to 4 Decoder | | | | | |
| input | | output | | | |
| in1 | in2 | out0 | out1 | out2 | out3 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

<2 to 4 Decoder(NAND gate)의 truth table>

또한 simulation 결과는 다음과 같다.



<2 to 4 decoder(NAND)의 truth table>

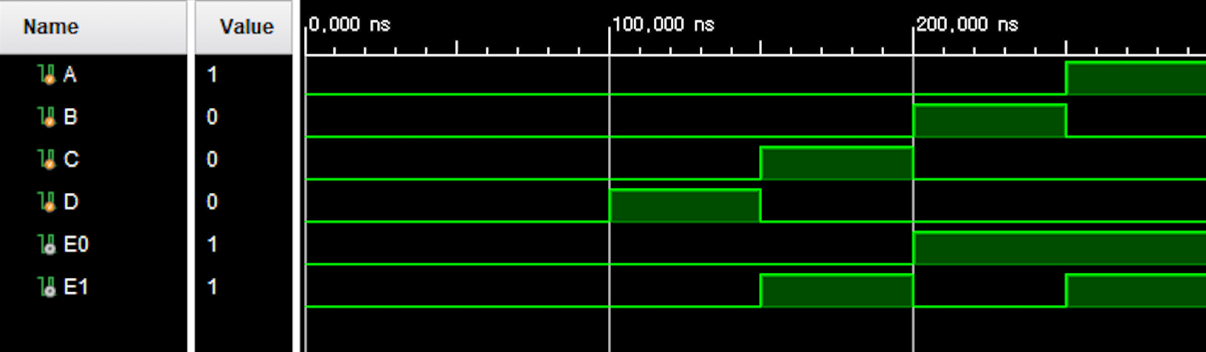
**2. 4 to 2 Encoder의 결과 및 simulation 과정**

4 to 2 encoder은 위의 decoder와 다르게, 0~3의 decimal input에 대해서 output을 binary 형태로 표현해주는 것이다. 따라서 input = 3이라면 output out1 = High, out2 = High, input = 1이라면 output out1 = Low와 같은 형태를 취하게 된다. 이에 대한 truth table과 k-map은 아래와 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 to 2 Encoder | | | | | |
| input | | | | output | |
| in1 | in2 | in3 | in4 | out1 | out2 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 |

<4 to 2 Encoder의 truth table>

simulation 결과는 아래와 같다.



<4 to 2 encoder의 truth table>

**3. 4 to 2 Encoder의 나머지 입력 형태의 의미**

4 to 2 encoder에서 입력 형태는 0000, 0010, 0100, 1000 4가지 이다. 각 입력은 input의 decimal 숫자가 0, 1, 2, 3을 의미한다. 하지만 실제 4 bit는 총 24, 16가지의 input을 표현해낼 수 있는데, 나머지 12가지의 입력 형태는 하나의 오류라고 볼 수 있다.(k-map에서 don’t care term과 같이 input으로 고려하지 않는 형태) 따라서 이를 보정하기 위한(error correction) 기술이 존재하는데, 아래에서 설명할 priority encoder 이다.

**4. 4 to 2 Priority Encoder**

위에서 설명한 것처럼, 4 to 2 encoder을 priority encoder로 만들어낸다면 16가지의 모든 input에 대해서 output을 고려할 수 있다. 또한 오류 정정의 효과도 가질 수 있다. 이를 제작하기 위해서는 encoder의 이름처럼 우선순위를 정해야 하는데, 가장 쉬운 방법은 input = 1xxx 이라면 1000으로 판단하는 것처럼 MSB를 우선으로 두는 방법이다. 이 경우에서 truth table은 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 to 2 Priority Encoder | | | | | |
| input | | | | output | |
| in1 | in2 | in3 | in4 | out1 | out2 |
| 1 | x | x | x | 1 | 1 |
| 0 | 1 | x | x | 1 | 0 |
| 0 | 0 | 1 | x | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 |

<4 to 2 Priority Encoder의 truth table>

이와 같은 형태로 논리 회로를 구성할 경우 모든 16가지 input에 대해서 적절한 4가지 output을 출력해내는 회로를 만들어낼 수 있다.

**5. BCD to Decimal decoder의 결과 및 simulation 과정**

BCD to Decimal은 위에서 구현한 2 to 4 decoder와 비슷하게, 4bit input을 받아 10개의 output중 일치하는 수에 출력을 주는 형태이다. 이는 AND gate로 구현할 수 있는데, 각 output은 아래와 같다.

assign out0 = ~in1 & ~in2 & ~in3 & ~in4;

assign out1 = ~in1 & ~in2 & ~in3 & in4;

assign out2 = ~in1 & ~in2 & in3 & ~in4;

assign out3 = ~in1 & ~in2 & in3 & in4;

assign out4 = ~in1 & in2 & ~in3 & ~in4;

assign out5 = ~in1 & in2 & ~in3 & in4;

assign out6 = ~in1 & in2 & in3 & ~in4;

assign out7 = ~in1 & in2 & in3 & in4;

assign out8 = in1 & ~in2 & ~in3 & ~in4;

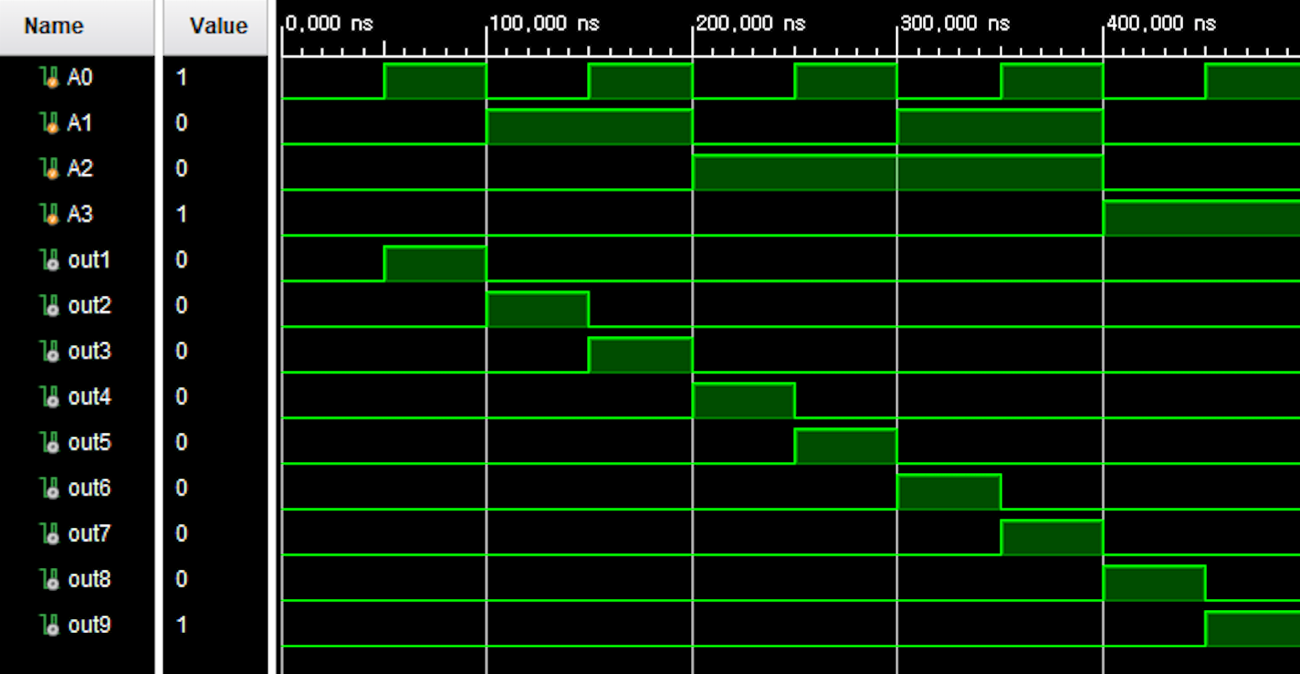
assign out9 = in1 & ~in2 & ~in3 & in4;

위 형태로 구현을 수행한다면, BCD의 k-map에서는 don’t care term인 input 1011, 1111등에 대해서 모든 출력이 0이 되는 회로를 구현해낼 수 있다. 이에 대한 truth table과 k-map은 아래와 같다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BCD to Decimal decoder | | | | | | | | | | | | | |
| input | | | | output | | | | | | | | | |
| in1 | in2 | in3 | in4 | out0 | out1 | out2 | out3 | out4 | out5 | out6 | out7 | out8 | out9 |
| 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 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 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 | 1 |

<BCD to Decimal decoder의 truth table>

또한 simulation 결과는 아래와 같다.



<BCD to Decimal decoder의 simulation 결과>

**6. Encoder와 Decoder의 주요 응용**

encoder은 위에서 본 것처럼 2n input을 n input으로 바꿔주는 효과를 가진다. 따라서 데이터 압축에서 큰 효과를 가질 수 있다. 이렇게 압축된 형태의 data는 전송, 저장 등에서 큰 이점을 가질 것임을 쉽게 파악할 수 있다. 실제로 아주 큰 data인 자연적인 data(analog) 형태를 기계적인 data(digital) 형태로 바꿀 때 encoding을 사용하는데, 소리를 변환하는 마이크, 카메라를 통해 찍은 사진을 png 등의 형태로 저장하는 것 등이 존재한다. 이와 반대로 decoder은 압출을 푸는 효과라고 생각할 수 있다. 또한 위에서 encoder을 설명한 것과 반대로 저장해 놓았던 data를 다시 자연 상태로 보내주는 형태라고 볼 수 있다. 이 경우에는 data를 받아 다시 화면에 보여주는 tv 스크린, 소리 data를 실제 소리로 만들어주는 스피커 등이 존재한다.

**7. 4 to 1 line MUX의 결과 및 simulation 과정**

4 to 1 line MUX는 4개의 input 중 1가지를 선택할 수 있도록 만들어주는 회로이다. 여기에서 n binary bit가 존재한다고 해 보자. n binary bit로는 총 2n 경우의 수를 만들어낼 수 있다. 따라서 n to 1 line MUX에서는 log2n 만큼의 추가적인 input(select를 수행할 input)이 필요하다는 것을 알 수 있다. 이에 대한 코드는 아래와 같다.

`timescale 1ns / 1ps

module mux(

input in1, in2, in3, in4, a, b,

output out1

);

assign out1 = (in1&~a&~b) | (in2&a&~b) | (in3&~a&b) | (in4&a&b);

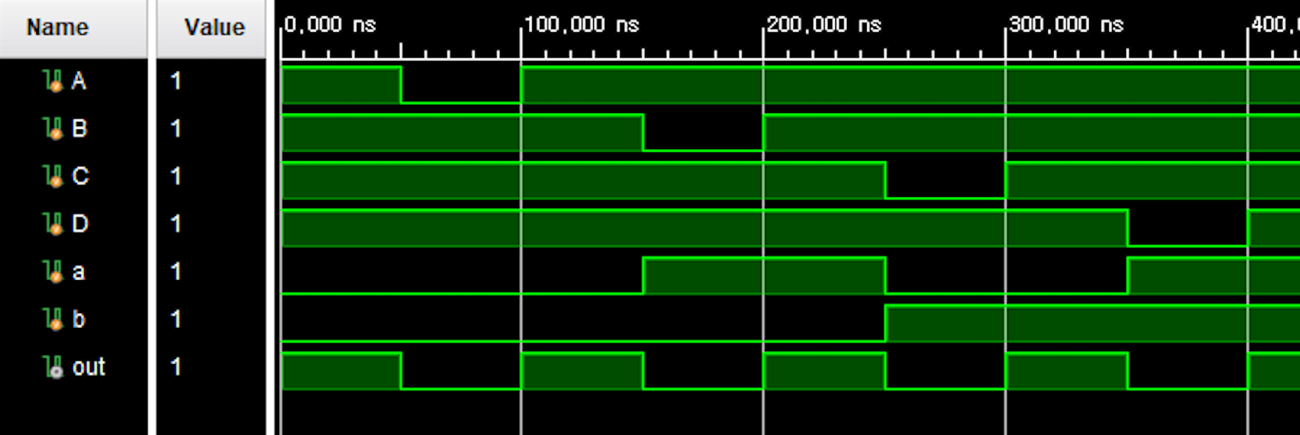
endmodule

<mux의 verilog 코드>

또한 truth table과 simulation 결과는 아래와 같다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 to 1 line MUX | | | | | | |
| input | | | | | | output |
| A | B | C | D | a | b | out |
| 0 | x | x | x | 0 | 0 | 0 |
| 1 | x | x | x | 0 | 0 | 1 |
| x | x | 0 | x | 0 | 1 | 0 |
| x | x | 1 | x | 0 | 1 | 1 |
| x | 0 | x | x | 1 | 0 | 0 |
| x | 1 | x | x | 1 | 0 | 1 |
| x | x | x | 0 | 1 | 1 | 0 |
| x | x | x | 1 | 1 | 1 | 1 |

<4 to 1 line MUX의 truth table>



<4 to 1 line MUX의 simulation 결과>

**8. 1 to 4 line deMUX를 이용한 4 to 16 decoder의 구현**

MUX와 반대로, deMUX는 회로에서 일정 작업을 수행하고 생긴 output에 대해, output을 내보내줄 line을 정하는 것이다. 이 또한 1 to n line deMUX라면 log2n 만큼의 추가적인 input이 필요함을 알 수 있다. 결과적으로 deMUX는 1개의 input(High / Low)를 받아서 deMUX를 거쳐 4가지의 output에 전달해줄 것인데, 만약 input이 항상 high라면, 2개의 추가적인 select input에 따라서 1000, 0100, 0010, 0001 4가지 형태로 나타내질 수 있다. 따라서 결과적으로 이는 select input이 input으로 들어가는 2 to 4 decoder와 동일하다. 또한 2 to 4 decoder 여러 개를 활용해 4 to 16 decoder을 구현해낼 수 있으며, 이는 아래와 같다.

(1) select input이 총 4개가 존재할 것이며, 이를 S3, S2, S1, S0 이라 하자.

(2) S3, S2의 4가지 경우에 대해서, 이는 모두 S1, S0에 대한 2 to 4 decoder 이다.

(3) S3, S2의 4가지 경우 또한 2 to 4 decoder로 표현 가능하다.

(4) 결과적으로 4 to 16 decoder은 총 5개의 1 to 4 line deMUX로 구현해낼 수 있다.

위와 같이 코드를 작성했는데, verilog에서 제대로 실행되지 않아, 코드만 아래 첨부한다.

`timescale 1ns / 1ps

module mux(

input S1, S0, in1,

output out3, out2, out1, out0

);

assign out3 = S1&S0&in1;

assign out2 = S1&~S0&in1;

assign out1 = ~S1&S0&in1;

assign out0 = ~S1&~S0&in1;

endmodule

module decoder(

input S3, S2, S1, S0, in1,

output out15, out14, out13, out12, out11, out10, out9, out8, out7, out6, out5, out4, out3, out2, out1, out0

);

wire select3, select2, select1, select0;

mux mux\_select(.S1(S3), .S0(S2), .in1(in1), .out3(select3), .out2(select2), .out1(select1), .out0(select0));

mux mux\_3(.S1(S1), .S0(S0), .in1(selelct3), .out3(out15), .out2(out14), .out1(out13), .out0(out12));

mux mux\_2(.S1(S1), .S0(S0), .in1(selelct2), .out3(out11), .out2(out10), .out1(out9), .out0(out8));

mux mux\_1(.S1(S1), .S0(S0), .in1(selelct1), .out3(out7), .out2(out6), .out1(out5), .out0(out4));

mux mux\_0(.S1(S1), .S0(S0), .in1(selelct0), .out3(out3), .out2(out2), .out1(out1), .out0(out0));

endmodule

<4 to 16 decoder의 verilog 코드>

**9. 결과 검토 및 논의 사항**

이번 실험에서는 decoder, encoder, mux, demux의 여러 형태를 직접 구현했다. 또한 단순히 n to 2n decoder 뿐만이 아닌 4 to 10 decoder(BCD to Decimal)과 같은 형태도 구현했다. 또한 priority encoder라는 개념으로 잘못된 input이 들어왔을 때에 처리방법에 대해서도 알아볼 수 있었다. 마지막으로는 demux를 통해서 Decoder을 구현하는 방법에 대해서 알아보았는데, 위에서 본 것 같이 개인적으로 코드는 작성했으나 작동하지 않았다. 이에 대해 원인을 알아보는 것도 중요할 것 같다.