전공: 컴퓨터공학 학년: 2학년 학번: 20161603 이름: 신민준

1. 2 to 4 Decoder의 결과 및 Simulation 과정에 대해서 설명하시오.

2 to 4 Decoder는 2-bit의 입력에 대해 4개의 각기 다른 출력 중 하나에만 신호를 주어야 하므로, 이를 표현하는 truth table은 다음과 같이 작성할 수 있었다. 디코더에는 active-high 방식과 active-low 방식이 존재하는데, active-high로 구현하면 다음과 같은 진리표가 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 |

위와 같은 decoder를 active-low 방식으로 구현한 경우의 진리표는 위 진리표의 출력값을 invert한 결과로 나타낼 수 있다.

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

진리표의 결과가 너무 간단하고 직관적이기 때문에, 위 두 진리표를 기반으로 2 to 4 decoder들의 함수식을 바로 구할 수 있다. Active high의 경우에는 각 digit에 해당하는 값이 다음과 같다.

Active low 방식으로 구현한 경우, 위 식에 간단히 inverter를 추가한 방식이기 때문에, 다음처럼 나타내어진다.

구한 Boolean 함수 식을 기반으로 Verilog code를 다음처럼 작성할 수 있었다.

|  |
| --- |
| `timescale 1ns / 1ps  module ah\_decoder(  input in1, in2,  output out1, out2, out3, out4  );  assign out1 = ~in1&~in2;  assign out2 = ~in1&in2;  assign out3 = in1&~in2;  assign out4 = in1&in2;  endmodule  module al\_decoder(  input in1, in2,  output out1, out2, out3, out4  );  assign out1 = ~(~in1&~in2);  assign out2 = ~(~in1&in2);  assign out3 = ~(in1&~in2);  assign out4 = ~(in1&in2);  endmodule |

식에서 볼 수 있듯이, active high 방식은 AND gate를 사용해 구현되며, active low 방식은 반대로 NAND gate를 사용해 구현함을 확인할 수 있고, 이는 RTL Schematic을 통해 다음 도식처럼 나타내어졌음을 볼 수 있다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

위 도식은 AND gate를 사용한 active high 방식이다.

텍스트, 지도이(가) 표시된 사진

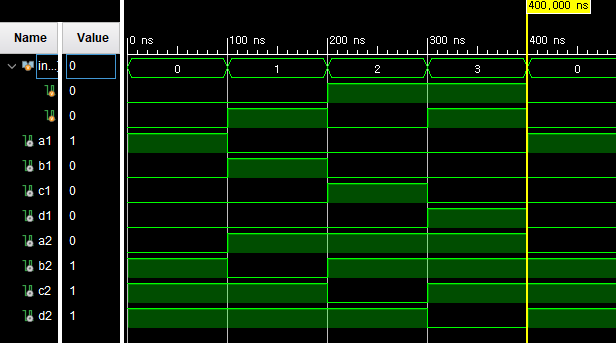
자동 생성된 설명

이 도식은 active-low 방식의 decoder를 도식화한 결과이다.

구현한 decoder가 정확하게 의도한 결과를 나타내는지를 확인하기 위해 Simulation을 실행했다. Active high와 low, 두 가지의 방식으로 작성한 decoder를 동시에 simulation 하기 위해 다음과 같은 simulation source를 작성했다.

|  |
| --- |
| `timescale 1ns / 1ps  module decodersim;  reg [1:0] in;  wire a1, b1, c1, d1, a2, b2, c2, d2;    ah\_decoder ah(in[1], in[0], a1, b1, c1, d1);  al\_decoder al(in[1], in[0], a2, b2, c2, d2);    initial in = 2'b00;  always in = #100 in+1;    initial begin  #1600  $finish;  end  endmodule |

위 simulation의 결과는 다음의 simulation 결과에서 확인할 수 있다.



각 decoder의 결과가 의도했던 진리표의 결과와 일치하는 것을 확인할 수 있고, 이를 통해 구현한 decoder가 올바르게 작동함을 검증할 수 있었다.

1. 4 to 2 Encoder의 결과 및 Simulation 과정에 대해서 설명하시오.

4 to 2 encoder의 경우, 1. 에서 작성한 2 to 4 decoder의 정 반대의 역할을 수행하므로, 사실상 동일한 진리표를 공유한다. 따라서 4 to 2 decoder의 truth table은 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | | | **Output** | |
| A | B | C | D | E0 | 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 |

위 truth table로부터, 다음과 같은 Boolean function 식을 간단하게 곧바로 구할 수 있다.

구한 식을 Verilog code로 구현해 다음과 같이 작성했다.

|  |
| --- |
| `timescale 1ns / 1ps  module encoder (  input a, b, c, d,  output e0, e1  );  assign e0 = a|b;  assign e1 = a|c;  endmodule |

위 Verilog code의 RTL schematic을 확인했을 때 결과는 다음과 같았다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

구현한 decoder가 정상적으로 작동하는지 확인하기 위해 다음과 같이 simulation source를 작성했다.

|  |
| --- |
| `timescale 1ns / 1ps  module encodersim;  reg [3:0] in;  wire e0, e1;    encoder en(in[3], in[2], in[1], in[0], e0, e1);    initial in = 4'b0001;  always in = #100 in\*2;    initial begin  #1600  $finish;  end  endmodule |

이 때, 0001, 0010, 0100, 1000을 제외한 다른 입력은 고려 대상이 아니므로, 이 네 입력만이 들어오도록 simulation code를 작성해야 결과가 깔끔하게 도출될 수 있었다.

이렇게 작성한 simulation의 결과는 다음과 같았다.

앉아있는이(가) 표시된 사진

자동 생성된 설명

의도했던 결과가 정확하게 simulation에서 나타남을 확인할 수 있었고, 따라서 구현한 Verilog code가 정상적으로 동작함을 보일 수 있다.

1. 4 to 2 Encoder에서 입력 형태 4가지를 제외한 나머지 입력 형태는 무엇을 뜻하는지 설명하시오.

위에서 구현한 4 to 2 encoder의 경우, 입력 형태 0001, 0010, 0100, 1000을 제외한 다른 입력 형태는 애초에 고려 대상이 아니기 때문에, 구현 과정에서 “don’t care” 조건으로서 구현된다. 따라서 다른 입력 형태가 입력으로 들어온 경우, 결과 값은 예측할 수 없는 값이 출력되므로, 잘못된 입력 값으로 인해 에러가 발생한다고 볼 수 있다.

따라서, 만일 여러 가지의 high 상태를 갖는 input을 사용하는 encoder를 구현해야 할 경우, 위 문제를 해결하기 위해 priority encoder를 사용하는 것이 바람직할 수 있다.

1. 4 to 2 Encoder의 4가지 형태가 아닌 모든 입력 형태(16가지)에 대하여 동작되는 priority encoder의 논리 회로를 구성하여라.

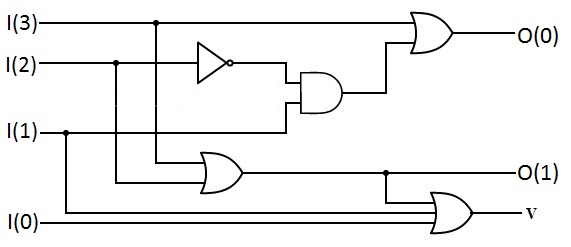
Priority encoder는 입력된 값에서 가장 높은 priority를 갖는 비트의 값에 대해서만 작동하는 encoder이다. 이 논리를 진리표로 나타내면 다음과 같이 don’t care를 사용해 나타낼 수 있다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | | | **Output** | |
| A | B | C | D | E0 | E1 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | x | 0 | 1 |
| 0 | 1 | x | x | 1 | 0 |
| 1 | x | x | x | 1 | 1 |

위 진리표를 다음과 같이 K-map으로 나타내고, 이를 통해 식을 작성하면 다음과 같다.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| E0 | | | | |  | E1 | | | | |
| ab cd | 00 | 01 | 11 | 10 | ab cd | 00 | 01 | 11 | 10 |
| 00 | 0 | 0 | 0 | 0 | 00 | 0 | 0 | 1 | 1 |
| 01 | 1 | 1 | 1 | 1 | 01 | 0 | 0 | 0 | 0 |
| 11 | 1 | 1 | 1 | 1 | 11 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 | 10 | 1 | 1 | 1 | 1 |

따라서 각 output line에 AND gate 1개, OR gate 1개를 사용해 구현할 수 있다. 이 때, 입력으로 0000이 들어오는 경우는 예외 처리를 해주어야 하기 때문에, 에러를 확인하는 출력을 포함해서 이 함수를 회로도로 표현하면 다음과 같다.



1. BCD to Decimal decoder의 결과 및 Simulation 과정에 대해서 설명하시오.

BCD to Decimal decoder의 경우, 4개의 입력에 대해 10개의 출력이 존재한다. 따라서 다음과 같이 진리표가 작성된다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **a3** | **a2** | **a1** | **a0** | **e0** | **e1** | **e2** | **e3** | **e4** | **e5** | **e6** | **e7** | **e8** | **e9** |
| 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 |
| 1 | 0 | 1 | x | x | x | x | x | x | x | x | x | x | x |
| 1 | x | x | x | x | x | x | x | x | x | x | x | x | x |

위 진리표에 대한 Boolean 식은 각 가 한 가지 case의 입력에 대해서만 high 값을 가지므로, 바로 다음과 같이 자명하게 각 식을 표현한 Verilog code를 작성할 수 있었다. 이 때, 1010, 1011, 1100, 1101, 1110, 1111의 입력은 고려 대상이 아니므로 don’t care로 처리했다.

|  |
| --- |
| `timescale 1ns / 1ps  module bcd2dec(  input a3, a2, a1, a0,  output o0, o1, o2, o3, o4, o5, o6, o7, o8, o9  );  assign o0 = ~a3&~a2&~a1&~a0;  assign o1 = ~a3&~a2&~a1&a0;  assign o2 = ~a3&~a2&a1&~a0;  assign o3 = ~a3&~a2&a1&a0;  assign o4 = ~a3&a2&~a1&~a0;  assign o5 = ~a3&a2&~a1&a0;  assign o6 = ~a3&a2&a1&~a0;  assign o7 = ~a3&a2&a1&a0;  assign o8 = a3&~a2&~a1&~a0;  assign o9 = a3&~a2&~a1&a0;  endmodule |

작성한 Verilog code의 구조는 RTL Schematic을 통해 확인할 수 있다.

텍스트, 지도이(가) 표시된 사진

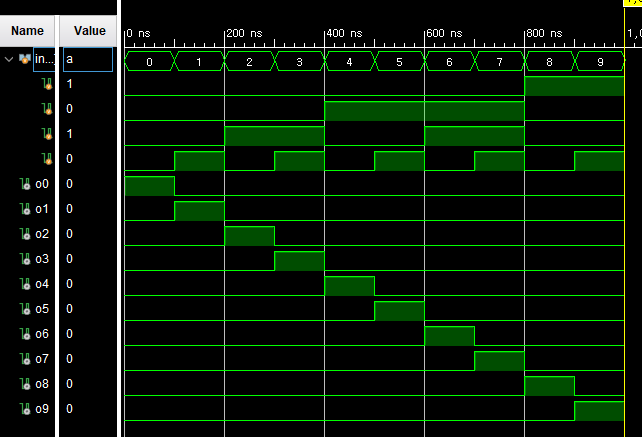
자동 생성된 설명

각 output마다 3개씩의 AND gate를 거치는 모습을 볼 수 있다.

구현한 Verilog code가 정상적으로 작동하는지 확인하기 위해 다음과 같이 simulation source를 작성해 simulation을 수행했다.

|  |
| --- |
| `timescale 1ns / 1ps  module b2dsim;  reg [3:0] in;  wire o0, o1, o2, o3, o4, o5, o6, o7, o8, o9;    bcd2dec b2d(in[3], in[2], in[1], in[0], o0, o1, o2, o3, o4, o5, o6, o7, o8, o9);    initial in = 4'b0000;  always in = #100 in+1;    initial begin  #1600  $finish;  end  endmodule |

Simulation 수행 결과는 다음과 같았다.



Simulation 결과로부터, 0000 ~ 1001 까지의 input에 대해 올바른 출력을 나타내는 것을 확인할 수 있었다.

1. Encoder와 decoder의 주요 응용에 대하여 설명하시오.

Encoder는 decoder와 함께 사용되어, 모뎀처럼 데이터를 올바른 목적지로 보내는 역할을 할 수 있다. 또한, 모터와 같은 다양한 부품들이 서로 동기화되어 작동할 수 있는 데에도 쓰이고 있다.

여기서 Decoder의 경우, n개의 binary 정보를 받아 이를 개의 고유한 출력으로 변환해주는 기능을 하는데, 이 특성을 활용해 데이터 멀티플렉싱/demultiplexing에 사용되기도 하고, 숫자를 표시하는 7-segment display에 사용되기도 한다. 또한, 컴퓨터 내의 메모리의 위치를 받아 해당 위치의 값을 출력하는 memory addressing 작업에도 사용되는 등 다양한 응용 분야가 존재한다.

1. 8 to 1 line MUX의 결과 및 simulation 과정에 대해서 설명하시오(code, truth table 작성).

8 to 1 line multiplexer는 select signal에 따라 입력 line을 골라 출력 line으로 그 값을 전달해주는 역할을 한다. 이를 진리표로 나타내면 다음과 같다(a, b, c, …, h는 입력 line)

|  |  |  |  |
| --- | --- | --- | --- |
| **s2** | **s1** | **s0** | **q** |
| 0 | 0 | 0 | a |
| 0 | 0 | 1 | b |
| 0 | 1 | 0 | c |
| 0 | 1 | 1 | d |
| 1 | 0 | 0 | e |
| 1 | 0 | 1 | f |
| 1 | 1 | 0 | g |
| 1 | 1 | 1 | h |

각 line에 해당하는 select signal의 경우가 단 하나밖에 없으므로, 다음과 같이 간단하게 Boolean function으로 나타낸 Verilog code를 작성할 수 있었다.

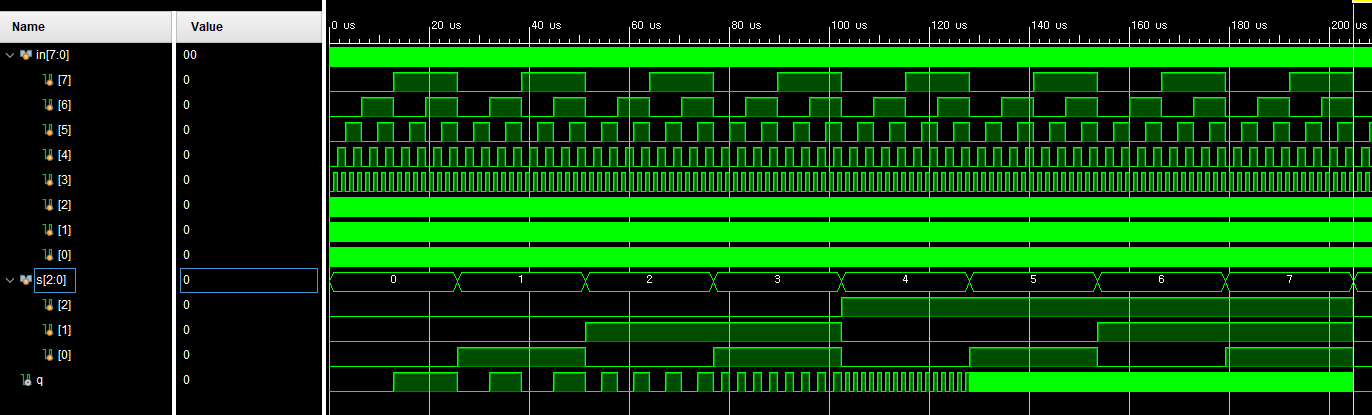
|  |
| --- |
| `timescale 1ns / 1ps  module mux81(  input a, b, c, d, e, f, g, h, s2, s1, s0,  output q  );  assign q = (a&~s0&~s1&~s2)|(b&s0&~s1&~s2)|(c&~s0&s1&~s2)|(d&s0&s1&~s2)|(e&~s0&~s1&s2)|(f&s0&~s1&s2)|(g&~s0&s1&s2)|(h&s0&s1&s2);  endmodule |

이 때, 작성한 코드가 정상적으로 동작하는지 확인하기 위해 다음의 simulation source를 작성해서 테스트 해보았다.

|  |
| --- |
| `timescale 1ns / 1ps  module mux8sim;  reg [7:0] in;  reg [2:0] s;  wire q;    mux81 mux(in[7], in[6], in[5], in[4], in[3], in[2], in[1], in[0], s[2], s[1], s[0], q);    initial in = 8'b00000000;  initial s = 3'b000;  always in = #100 in+1;  always s = #25600 s+1;    initial begin  #1600  $finish;  end  endmodule |

이 때, in 배열은 1byte 짜리 배열이므로, in의 값이 256번째 바뀔 때 마다 select signal을 바꿔주도록 구현했다.

Simulation 결과는 다음과 같다. 예상했던 8-1 mux의 결과값과 일치하는 것으로 보아 정상적으로 구현되었음을 확인할 수 있다.



1. 1 to 4 line deMUX를 이용하여 4 to 16 decoder를 수행하고 결과를 나타내시오.

1 to 4 line demux는 다음과 같은 진리표를 갖는다

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select** | | **Data** | **Output** | | | |
| S0 | S1 | F | Out 0 | Out 1 | Out２ | Out３ |
| 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 |

Demux의 특징에 맞게, Data의 값을 Select 값이 나타내는 output line으로 연결해주는 것을 확인할 수 있다. 이 진리표의 결과가 매우 간단하기 때문에, 다음과 같이 각 output line에 대한 Boolean function을 작성할 수 있었다.

이 식을 바탕으로 다음과 같이 1 to 4 deMUX를 Verilog code로 작성할 수 있었다.

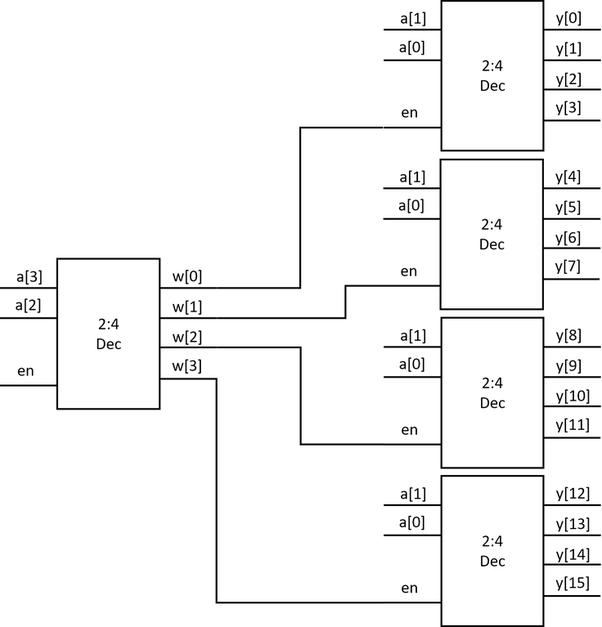
|  |
| --- |
| `timescale 1ns / 1ps  module demux14(  input s1, s0, f,  output a, b, c, d  );  assign a = f&(~s1&~s0);  assign b = f&(~s1&s0);  assign c = f&(s1&~s0);  assign d = f&(s1&s0);  endmodule |

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

위 회로도는 해당 verilog code의 Schematic을 확인한 결과이다.

Demux의 입력 데이터를 1로 고정해두면 해당 demux는 decoder와 같은 작업을 수행한다. 따라서, 1-to-4 deMux로 4-to-16 Decoder를 구현하는 방법은 2-to-4 decoder로 구현하는 방법인 다음 사진과 동일하다.



이 때, demux의 입력 데이터(F)는 위 그림에서 쓰인 decoder에서 받는 enable 입력(en)과 동일한 역할을 수행하므로, 그대로 사용하면 된다.

따라서 구현한 4-to-16 decoder의 Verilog code와 RTL Schematic은 다음과 같다.

|  |
| --- |
| module decoder416(  input s3, s2, s1, s0,  output o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14, o15  );  wire en0, en1, en2, en3;  demux14 dm(s3, s2, 1, en0, en1, en2, en3);  demux14 dm0(s1, s0, en0, o0, o1, o2, o3);  demux14 dm1(s1, s0, en1, o4, o5, o6, o7);  demux14 dm2(s1, s0, en2, o8, o9, o10, o11);  demux14 dm3(s1, s0, en3, o12, o13, o14, o15);  endmodule |

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

구현한 4 to 16 decoder가 정상적으로 작동하는지 확인하기 위해, 다음과 같이 simulation source를 작성했다.

|  |
| --- |
| `timescale 1ns / 1ps  module dc416sim;  reg [3:0] in;  wire o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14, o15;    decoder416 dc(in[3], in[2], in[1], in[0], o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14, o15);    initial in = 4'b0000;  always in = #100 in+1;    initial begin  #1600  $finish;  end  endmodule |

위 simulation source를 사용해 수행한 결과는 다음과 같다. 정상적인 4 to 16 decoder의 작동과 동일함을 확인할 수 있다.

전자기기이(가) 표시된 사진

자동 생성된 설명

1. 결과 검토 및 논의 사항

* 수행했던 모든 실험의 simulation 결과가 예상했던 결과값과 일치하는 것으로부터 구현이 정확하게 이루어졌음을 확인할 수 있었다.
* 그러나, 구현과는 별개로 simulation 하는 과정에서 8-1 mux와 같이 input과 output의 갯수가 많아지면 한번의 simulation으로 전체 결과를 확인하기 어려워졌다. 4x16 decoder를 작성할 때 모듈을 활용해 더 작은 과정으로 작업을 나누는 방식을 쓴 것처럼, 큰 input/output 갯수에 대해서도 비슷한 접근을 해야 오류가 나는 빈도를 줄일 수 있을 것이다.

1. 추가 이론 조사 및 작성

* MUX와 deMUX를 동시에 사용해 한 개의 line에 다수의 line들의 정보를 교환하는 것이 가능하다. 이는 전송하는 쪽에서는 MUX를, 수신하는 쪽에서는 deMUX를 사용해 같은 select signal로 line을 맞춰 놓고 서로 정보를 교환하는 방식으로 이루어진다.