

기초디지털실험

Week3_Combinational Logic

결과레포트



학정번호 : EEE3313-07-00

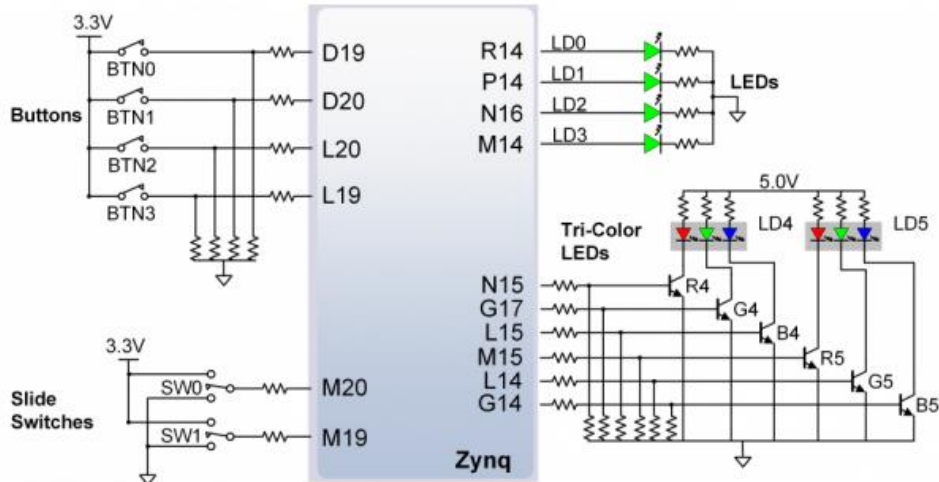
학번 : 2018142173

이름 : 이 찬

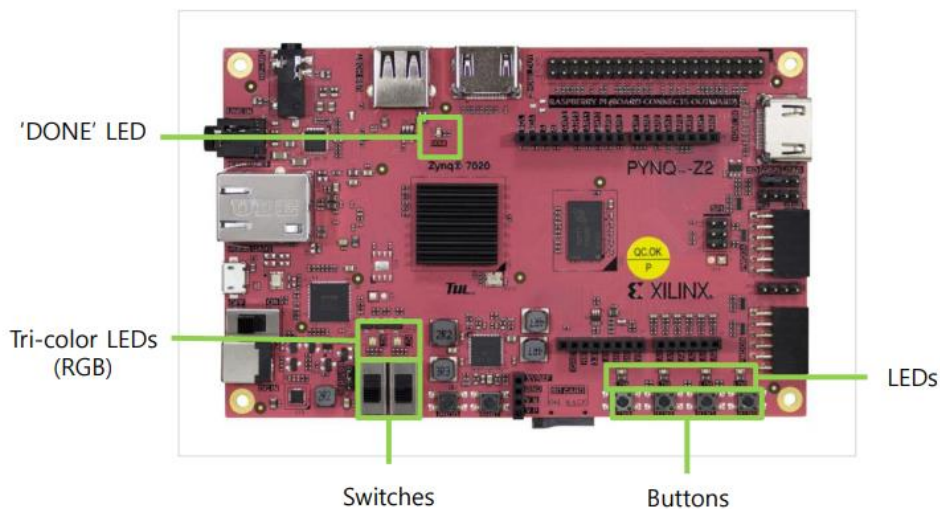
작성일자 : 2022-03-27

1. 실험 이론

- PYNQ GPIO

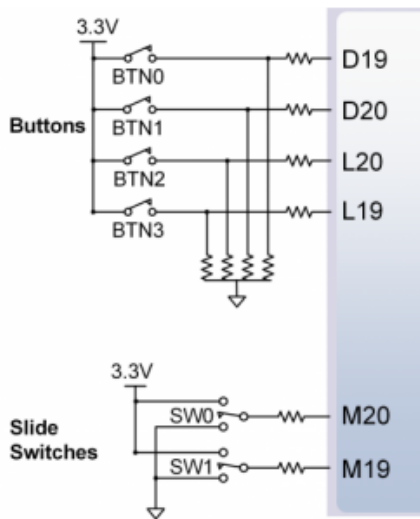


PYNQ 보드는 파이썬 생산성 향상을 위한 Zynq보드로, 위 사진은 3주차 실험에서 입력과 출력에 필요한 기본적인 소자들의 회로도이다. 버튼, LEDs, Tri-Colo LED, 스위치로 구성되어 있다.



위와 같이 'DONE' LED는 처음 vivado에 보드를 연결해서 실행이 올바르게 잘 되었을 때, 켜지는 LED 이다. Tri-color LEDs는 총 2개의 LED로 이루어져 있으며 각각 RGB, 3개의 값을 가지고 있다. 스위치도 2개의 스위치로 이루어져 있고, 버튼과 LEDs는 4개로 이루어져 있다. 각각의 소자들을 자세히 살펴보자.

1) 버튼과 스위치



Signal Name	PL PIN
BTN0	D19
BTN1	D20
BTN2	L20
BTN3	L19

Table 9 Push Button PL pin mapping

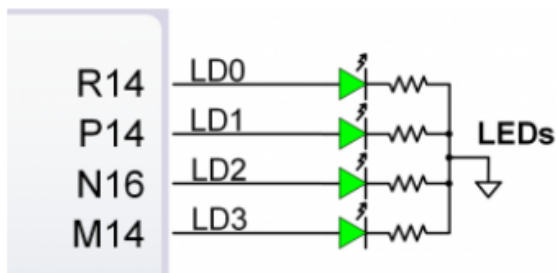
Signal Name	PL PIN
SW0	M20
SW1	M19

Table 10 Dip switch PL pin mapping

버튼은 4개로 구성되어 있으며, 각 PL 핀 마다 버튼이 할당되어 있다. 버튼을 누르지 않을 땐, zynq의 핀들이 접지의 값(0V)을 가지는 것을 알 수 있으며, 버튼을 눌렀을 때 3.3V가 흐르게 되어, 핀에 높은 값의 전류가 흐르는 것을 알 수 있다. 슬라이드 스위치는 2개로 구성되어 있으며, 각 스위치를 ON했을 경우 3.3V가 핀에 가해져서 높은 값의 전류가 흐르며, OFF했을 경우 접지와 연결되어 0V의 값을 가진다. 따라서,

{SW0(ON), SW1(OFF)}, {SW0(OFF), SW1(ON)}, {SW0(OFF), SW1(OFF)}, {SW0(ON), SW1(ON)} 이렇게 총 4가지의 경우가 나오는 것을 알 수 있다.

2) LEDS

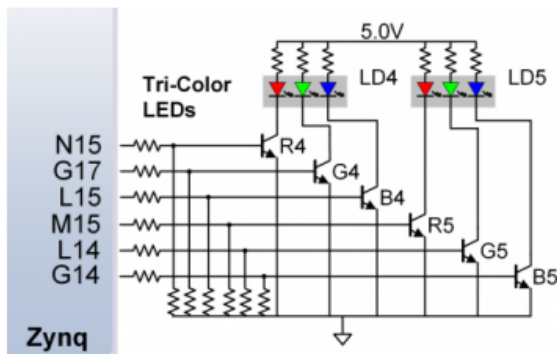


Signal Name	PL PIN
LED0	R14
LED1	P14
LED2	N16
LED3	M14

Table 11 LED PL pin mapping

LED는 4개로 구성되어 있으며, 각각의 핀들에 높은 값을 할당해주면 LED가 켜지는 방식이다.

3) Tri-Color LEDS

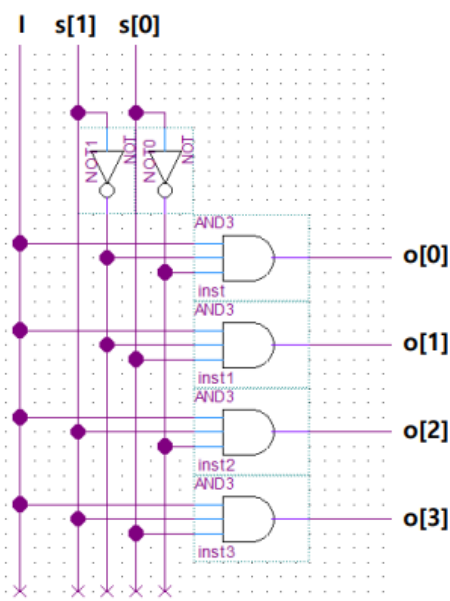


Signal Name	PL PIN
LD4 Blue	L15
LD4 Red	N15
LD4 Green	G17
LD5 Blue	G14
LD5 Red	M15
LD5 Green	L14

Table 12 LED PL pin mapping

3개의 색을 가지는 LEDS는 크게 2개로 구성되어 있으며, 각 LED마다 Red, Green, Blue 총 3개의 값을 가지기 때문에, 6개의 LED를 가지고 있다. 각각의 LED는 트랜지스터로 연결되어 있다. zynq의 핀에 전압이 가해지면, LED에 할당된 5V가 트랜지스터를 거쳐 접지로 연결되어 전류가 흐르는 방식이다. 하나의 LED에 여러 다른 색이 켜진다면 다양한 색 조합을 만들 수 있다.

- DEMUX



s[1]	s[0]	o0	o1	o2	o3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

DEMUX(디멀티플렉서)는 MUX(멀티플렉서)와 반대로, 1개의 입력 값을 여러 개의 출력값 중 1개를 선택하여 나타내는 방식이다. I(입력값)가 s[1], s[0]이 가지는 값에 따라서 o(출력값)가 가질 수 있는 경우가 달라진다. s[1]과 s[0]은 00,01,10,11 이렇게 총 4개의 경우로 나뉘며, 이를 통해 1개의 출력값만 입력값을 가지고, 나머지 3개의 출력값은 0을 가지게 된다. 따라서 출력값은 o[0], o[1], o[2], o[3] 이렇게 4개의 값을 가지며, 각각의 경우에 따라 출력값에 입력값이 할당되어 위의 진리표와 같은 결과가 나온다.

```

module DEMUX_1_to_4(in, ,control, out);
    input in;
    input [2:0] control;
    output reg [3:0] out;

    always@(*)
    begin
        if(control == 2'b00) begin
            out[0] = in; out[1] = 0; out[2] = 0; out[3] = 0; end
        else if(control == 2'b01) begin
            out[0] = 0; out[1] = in; out[2] = 0; out[3] = 0; end
        else if(control == 2'b10) begin
            out[0] = 0; out[1] = 0; out[2] = in; out[3] = 0; end
        else begin
            out[0] = 0; out[1] = 0; out[2] = 0; out[3] = in; end
        end
    end
endmodule

```

베릴로그 코드를 통해 확인하면, Demux 모듈에선 입력값 i 와 i 를 통제하는 2비트 값인 control이 할당되며, 4비트값이 출력값으로 할당되는 것을 알 수 있다. If-else문을 이용하여 각 control값이 가지는 경우 4가지를 다르게 할당하고, 그에 따라 출력값 4가지가 한 개는 in값, 나머지는 0값을 할당하였다.

- Decoder

Enable	Inputs			Outputs							
E	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Decoder(디코더)는 입력값이 n 비트가 들어오면, 그에 따른 출력값이 2^n 으로 나타나는 구조이다. 예를 들어 3비트 값이 입력으로 들어오면, 총 8가지의 수가 나오므로 출력할 수 있는 값이 $D_0 \sim D_7$ 로 구성된다. 따라서, 계단형식으로 1개의 D값만 1을 가지고 나머지가 0을 가지는 구조가 된다. 위의 진리표를 통해 확인할 수 있다. 디코더는 3비트인 입력값 뿐만 아니라 Enable이라는 값을 입력으로 받는데, Enable은 활성화 여부에 따라 0은 디코더를 비활성화, 1은 디코더를 활성화하는 구조로 구성되어 있다.

```

module decoder(in, out);
    input [2:0] in;
    output reg [7:0] out;
    always@(*)
    begin
        case(in)
            3'b000 : out <= 8'b00000001;
            3'b001 : out <= 8'b00000010;
            3'b010 : out <= 8'b00000100;
            3'b011 : out <= 8'b00001000;
            3'b100 : out <= 8'b00010000;
            3'b101 : out <= 8'b00100000;
            3'b110 : out <= 8'b01000000;
            default : out <= 8'b00000000;
        endcase
    end
endmodule

```

베릴로그 코드를 통해 확인하면, 입력값은 2비트, 출력값은 8비트로 구성되어 있음을 알 수 있다. 총 8가지의 경우의 수가 존재하므로 if-else문보다, case문을 사용하는 것이 효율적이다. 3비트값 000부터 110까지 각 입력값에 따라 출력값이 다르게 할당되고 있으며, 조건에 부합하지 않을 경우 Enable에 따라 비활성화된 값인 00000000을 기본값으로 가진다.

2. 베릴로그 코드

1) DEMUX

- DEMUX_1_to_4.v

```

module DEMUX_1_to_4( // Demux에 대한 모듈 생성
    in, control, out
);
    input in; // 1개의 input값을 받는다.
    input [2:0] control; // 2비트의 control값을 가지며, input이 출력될 output의 위치를 결정한다.
    output reg [3:0] out; // 4비트의 output값을 가지며, 한개의 element만 1을 가지고 나머지 element는 0을 가진다.

    always@(*) // always문은 bracket 안에 있는 코드를 실행하는 블록을 의미한다.
    begin // always 문의 시작
        if(control == 2'b00) begin // 입력값인 control이 00일 때
            out[0] = in; out[1] = 0; out[2] = 0; out[3] = 0; end // out[0]이 입력값인 in을 가지고 나머지는 0을 가진다.
        else if(control == 2'b01) begin // 입력값인 control이 01일 때
            out[0] = 0; out[1] = in; out[2] = 0; out[3] = 0; end // out[1]이 입력값인 in을 가지고 나머지는 0을 가진다.
        else if(control == 2'b10) begin // 입력값인 control이 10일 때
            out[0] = 0; out[1] = 0; out[2] = in; out[3] = 0; end // out[2]이 입력값인 in을 가지고 나머지는 0을 가진다.
        else begin // 입력값인 control이 11일 때
            out[0] = 0; out[1] = 0; out[2] = 0; out[3] = in; end // out[3]이 입력값인 in을 가지고 나머지는 0을 가진다.
        end // always 문의 끝

    endmodule

```

Demux_1_to_4 모듈을 생성한다. 입력값은 i와 2비트의 control 값이며, 출력값은 register를 변수로 가지는 4비트 ouput 값이다. always문은 괄호 안에 있는 값에 따라 코드를 실행한다. Always@(*)이

므로, 출력값이 입력값에 의존된 구조로 코드를 구성한다. Begin과 end를 통해 always문을 구성하고, 입력값인 control이 00,01,10,11에 따라 각 index 별로 out값이 in 값을 가지도록 한다.

-testbench.v

```
module testbench(

);

    reg inp; // inp값을 register로 구성한다.
    reg [2:0] cont; // 2비트 값으로 register를 이용하여 cont를 생성한다. 이는 입력값으로 사용된다.
    wire [3:0] ou; // 4비트 값으로 wire를 이용하여 ou를 생성한다. 이는 출력값으로 사용된다.

    initial begin // initial begin을 통해 simulation에 적용할 코드를 작성한다.
        inp <= 2'b1; // inp값은 DEMUX_1_to_4 모듈의 in과 연결되므로, 특정 값을 가진다고 가정한다.
        // cont값에 따라 ou에 할당되는 index의 값이 달라지므로 simulation에 나타날 값을 4가지 경우로 입력한다.
        cont <= 2'b00; // cont값이 00일 때
        #10 cont <= 2'b01; // 10초 딜레이 후 cont값이 01일 때
        #10 cont <= 2'b10; // 10초 딜레이 후 cont값이 10일 때
        #10 cont <= 2'b11; // 10초 딜레이 후 cont값이 11일 때
        #10
        $stop; // 종료
    end

    // module instantiation을 통해 DEMUX_1_to_4에 있는 입력값과 출력값을 testbench의 입력값과 출력값을 연결시켜 객체를 생성한다.
    DEMUX_1_to_4 DEMUX_0(
        // 순서와 상관없이 연결
        .in(inp), // 입력값 in과 연결
        .control(cont), // 입력값 control과 연결
        .out(ou) // 출력값 ou와 연결
    );
endmodule
```

Testbench 모듈을 생성하고, 모듈 생성 규칙에 따라 inp, cont를 register로, ou를 wire로 선언하여 각각의 입력값과 출력값을 구현한다. Initial begin 구문을 통해 simulation에 나타나는 값을 나타낸다. Inp 값은 고정적인 값이며, cont값이 가질 수 있는 00,01,10,11 값을 10초 딜레이를 주어 변화시키고 그에 따른 출력값인 ou값을 측정한다. DEMUX_1_to_4의 모듈을 불러와 testbench의 모듈의 변수와 인스턴스화하여 객체를 생성한다.

-LED_DEMUX.v

```
module LED_DEMUX(sw, led); // 보드에 구현할 모듈을 생성한다.
    input [1:0] sw; // 보드의 switch이며 2개이므로, constraints에 따라 2비트를 할당한다.
    output [3:0] led; // 보드의 led이며 4개이므로, constraints에 따라 4비트를 할당한다.
    wire [2:0] cont; // wire를 통해 보드와 연결할 cont값을 지정하고, 이 값에 따라 다른 led에 불이 켜진다.
    parameter DEMUX_input = 1'b1; // 기본 DEMUX의 입력값이며, 이 값이 출력값에 할당된다.
    wire [3:0] DEMUX_output; // wire를 통해 보드와 연결될 출력값이며 4개의 led로 4비트로 구성되어 있다.

    // Module Body
    // 모듈 인스턴스를 통해 DEMUX_1_to_4의 입력값과 출력값을 LED_DEMUX의 input과 output에 인스턴스화 한다.
    DEMUX_1_to_4 DEMUX_0(.in(DEMUX_input), .control(sw), .out(led));
endmodule
```

보드에 구현할 모듈인 LED_DEMUX를 생성한다. LED_DEMUX는 보드에 연결할 값과 보드에 할당할

값으로 구성되어 있다. Switch와 led를 각각 입력값과 출력값으로 구성하고, 기본 input값을 파라미터로 할당하여 DEMUX_input을 생성한다. cont와 DEMUX_output는 모듈과 보드를 연결할 용도로 wire를 통해 cont와 DEMUX_output를 지정한다. 그 후 모듈 인스턴스를 통해 DEMUX_1_to_4의 변수와 LED_DEMUX의 변수들을 인스턴스화하여 객체를 생성한다.

2) Decoder

-Decoder.v

```
module Decoder(in, out);
    input [2:0] in; //3비트의 input값을 지정한다.
    output reg [7:0] out; // 3비트이므로 총 8가지의 경우의 수가 나온다. 따라서 8비트 output을 지정한다.
    always@(*) // always문을 통해 코드를 실행한다.
    begin
        case(in) // 총 9가지 경우가 나오기 때문에 if-else문보다 case문을 이용하여 코드를 작성한다.
            3'b000 : out <= 8'b00000001; // input값이 000일 때 output은 8비트 00000001이다.
            3'b001 : out <= 8'b00000010; // input값이 001일 때 output은 8비트 00000010이다.
            3'b010 : out <= 8'b00000100; // input값이 010일 때 output은 8비트 00000100이다.
            3'b011 : out <= 8'b00001000; // input값이 011일 때 output은 8비트 00001000이다.
            3'b100 : out <= 8'b00010000; // input값이 100일 때 output은 8비트 00010000이다.
            3'b101 : out <= 8'b00100000; // input값이 101일 때 output은 8비트 00100000이다.
            3'b110 : out <= 8'b01000000; // input값이 110일 때 output은 8비트 01000000이다.
            3'b111 : out <= 8'b10000000; // input값이 111일 때 output은 8비트 10000000이다.
            default : out <= 8'b00000000; // input값이 위 case들과 다 적합하지 않을 때 enable이 0으로 설정되어 기본값인 8비트 00000000을 가진다.
        endcase
    end
endmodule
```

Simulation을 위해 기본적인 Decoder 모듈을 생성한다. Decoder는 입력값 n에 따라 2^n 만큼 출력값이 정해지므로, 3비트 입력값을 지정하고, 출력값을 register를 통해 out을 선언하고, 2^3 인 8비트로 지정하였다. always문에 따라 입력값에만 의존하여 출력되도록 설정하고, 총 9가지의 경우가 나오므로 case문을 사용하였다. input값이 000, 001, 010, 011, 100, 101, 110, 111 이렇게 총 8가지 경우를 가지고, 그에 따른 출력값을 각각의 경우에 따라 할당하였다. 만약 input값이 위 경우에 만족하지 못한다면 기본값인 00000000을 가지도록 out에 할당하였다. 이는 이론에서 설명했던 enable이 0일 경우에 대한 case이다.

- testbench.v

```
module testbench(
);
    reg [2:0] input_decoder; // register를 통해 3비트 값인 input_decoder를 선언한다.
    wire [7:0] output_decoder; // wire를 통해 8비트 값인 output_decoder를 선언한다.
    initial begin // initial begin을 통해 simulation을 구현한다.
        input_decoder <= 3'b000; // input_decoder에 3비트 값인 000부터 차례대로 할당한다.
        #10 input_decoder <= 3'b001; // 10초 딜레이를 주고, input_decoder에 3비트 값인 001을 할당한다.
        #10 input_decoder <= 3'b010; // 10초 딜레이를 주고, input_decoder에 3비트 값인 010을 할당한다.
        #10 input_decoder <= 3'b011; // 10초 딜레이를 주고, input_decoder에 3비트 값인 011을 할당한다.
        #10 input_decoder <= 3'b100; // 10초 딜레이를 주고, input_decoder에 3비트 값인 100을 할당한다.
        #10 input_decoder <= 3'b101; // 10초 딜레이를 주고, input_decoder에 3비트 값인 101을 할당한다.
        #10 input_decoder <= 3'b110; // 10초 딜레이를 주고, input_decoder에 3비트 값인 110을 할당한다.
        #10 input_decoder <= 3'b111; // 10초 딜레이를 주고, input_decoder에 3비트 값인 111을 할당한다.
        #10
        $stop; // simulation을 종료한다.
    end

    // model instantiation을 통해 Decoder.v의 모듈인 Decoder의 변수인 in과 out에 testbench의 변수인 input_decoder와 output_decoder를 인스턴스화하여 객체를 생성한다.
    Decoder decoder_0(
        .in(input_decoder),
        .out(output_decoder) // output_decoder의 객체가 생성됨에 따라 input_decoder가 가지는 값이 그대로 Decoder 모듈의 영향을 받아 값을 나타낸다.
    );
endmodule
```

Simulation 구현을 위해 testbench.v를 생성하고 testbench 모듈을 만든다. 모듈의 특성에 따라 register로 3비트 값인 input_decoder를 선언하고, wire로 8비트 값인 output_decoder를 선언한다. Simulation에 3비트의 경우인 8가지를 나타내기 위해서 10초 딜레이를 주면서 input_decoder에 000, 001, 011, 100, 101, 110, 111 순으로 값을 할당하였다. Decoder의 변수인 in, out값을 인스턴스화하여 testbench의 변수인 input_decoder, output_decoder에 대한 객체를 생성한다. output_decoder는 testbench에 정의된 input_decoder의 값에 따라서 decoder 모듈의 순서를 그대로 따른다. 따라서 output_decoder의 값이 simulation에 나타나게 된다.

-RGB_Decoder.v

```
module RGB_Decoder(in, out);
    input [2:0] in; // 3비트의 입력값을 받는다.
    output reg [5:0] out; // register를 통해 6비트의 출력값을 받는다. 6비트로 받는 이유는 LED 2개 가 각각 RGB로 구성되어 있어서 각각의 값을 나타내기 위함이다.

    // Module Body
    always@(*) // always 문을 사용하여 출력값이 입력값에 의존되게 코드를 작성한다.
    begin
        case(in)
            // 경우가 많으므로 case문을 통해 코드를 작성한다.
            3'b000 : out <= 6'b100100; // 강의안의 진리표에 따라 red 값을 나타내기 위해 6비트 100100을 out에 할당한다.
            3'b001 : out <= 6'b101101; // 강의안의 진리표에 따라 magenta 값을 나타내기 위해 6비트 101101을 out에 할당한다.
            3'b010 : out <= 6'b110110; // 강의안의 진리표에 따라 yellow 값을 나타내기 위해 6비트 100100을 out에 할당한다.
            3'b011 : out <= 6'b010010; // 강의안의 진리표에 따라 green 값을 나타내기 위해 6비트 010010을 out에 할당한다.
            3'b100 : out <= 6'b011011; // 강의안의 진리표에 따라 cyan 값을 나타내기 위해 6비트 011011을 out에 할당한다.
            3'b101 : out <= 6'b001001; // 강의안의 진리표에 따라 blue 값을 나타내기 위해 6비트 001001을 out에 할당한다.
            3'b110 : out <= 6'b111111; // 강의안의 진리표에 따라 white 값을 나타내기 위해 6비트 111111을 out에 할당한다.
            default : out <= 6'b000000; // 위의 case에 해당하는 값이 없을 경우 기본값인 LED off 값을 나타내기 위해 6비트 000000을 out에 할당한다.
        endcase
    end
endmodule
```

보드에 Decoder를 구현하기 위해서 기본 모듈인 RGB_Decoder.v를 생성한다. 입력값을 3비트로, 출력값을 register를 이용해서 6비트로 선언한다. 출력값을 6비트로 선언하는 이유는 비트 수에 따라 출력이 결정되는 것이 아니라, 강의안의 표에 따른 led의 색깔에 따라 값이 결정되기 때문이다. always 문을 사용하여 출력값이 입력값에 의존되게 코드를 작성한다. 입력값에 따른 출력값의 경우가 많으므

로 case문을 이용하여 각 색깔에 따른 비트 값을 출력값인 out에 할당하였다.

- Decoder_RGB.v

```
module Decoder_RGB(
    sw, btn, led4_r, led4_g, led4_b, led5_r, led5_g, led5_b // 보드에서 필요한 element들을 모두 선언한다.
);
    input [1:0] sw; // switch는 2개로 구성되어 있으므로 2비트로 선언한다.
    input [3:0] btn; // 버튼은 4개로 구성되어 있으므로 4비트로 선언하지만 실제 사용할 버튼은 1개이다.
    output led4_r, led4_g, led4_b, led5_r, led5_g, led5_b; // Decoder를 나타내기 위해 LED 6개를 출력값으로 선언한다.

    // Module Body
    wire [2:0] decoder_in; // wire를 통해서 switch, 버튼과 연결하기 위해 3비트 값인 decoder_in을 입력값으로 선언한다.
    wire [5:0] decoder_out; // wire를 통해서 led와 연결하기 위해 6비트 값인 decoder_out을 출력값으로 선언한다.

    assign decoder_in[2] = sw[1]; // 입력값에 스위치 값을 할당한다.
    assign decoder_in[1] = sw[0]; // 입력값에 스위치 값을 할당한다.
    assign decoder_in[0] = btn[0]; // 입력값에 버튼 값을 할당한다.

    // 강의안의 표에 따르면 led4_r가 가장 높은 비트, led5_b가 가장 낮은 비트로 색이 구현되어 있는 것을 확인할 수 있다.
    assign led4_r = decoder_out[5]; // 가장 높은 index로 출력값을 led4_r에 할당한다.
    assign led4_g = decoder_out[4]; // 그 다음 index로 출력값을 led4_g에 할당한다.
    assign led4_b = decoder_out[3]; // 그 다음 index로 출력값을 led4_b에 할당한다.
    assign led5_r = decoder_out[2]; // 그 다음 index로 출력값을 led5_r에 할당한다.
    assign led5_g = decoder_out[1]; // 그 다음 index로 출력값을 led5_g에 할당한다.
    assign led5_b = decoder_out[0]; // 마지막 index로 출력값을 led5_b에 할당한다.

    // module instantiation를 통해 RGB_Decoder 모듈에서 선언한 3비트의 입력값을 스위치(2개)와 1번짜 버튼에 할당하고 6비트의 출력값과 out을 인스턴스화하여 객체를 생성한다.
    // RGB_Decoder의 각 출력값이 led에 할당되어 서로 다른 8가지의 색을 구현할 수 있다.
    RGB_Decoder decoder_0(.in({sw,btn[0]}), .out(decoder_out));
endmodule
```

보드와 연결하기위해 Decoder_RGB.v를 생성한다. 보드에서 필요한 소자인 스위치 2개, 버튼 1개, led 6개를 선언한다. 스위치는 2개이므로 2비트로 할당하며, 버튼은 4개로 구성되어 있으므로 4비트로 할당한다. 그러나, 실제 필요한 버튼은 1개이다. led는 6개이므로 각 led를 출력값으로 선언한다. 모듈과 보드를 연결하기위해 wire로 3비트 입력값과 6비트 출력값을 선언한다. 각 입력값에 2개의 스위치 값과 1개의 버튼을 할당한다. 6비트의 출력값을 index가 하향하는 식으로 rgb 순서에 따라 led에 값을 할당한다. 하향식으로 진행하는 이유는 강의안의 표에 led4_r가 가장 높은 비트, led5_b가 가장 낮은 비트로 나와있기 때문이다. RGB_Decoder 모듈의 변수와 Decoder_RGB의 변수를 연결하여 객체를 생성한다. 따라서 RGB_Decoder 모듈에 할당되는 결과값이 led에 그대로 나타나게 되어 총 8가지의 다른 색을 구현할 수 있다.

[illegible]

-PYNQ RGB 실험



결과 분석

3비트 input_decoder가 나타나는 값에 따라서 output_decoder 값이 모두 다르게 구현되고 있는 것을 확인할 수 있다. 입력값이 n비트가 들어오면, 그에 따른 출력값이 2^n 으로 나타나는 구조인 Decoder가 올바르게 시뮬레이션 되고 있음을 알 수 있다. 이를 표로 정리하면 다음과 같다.

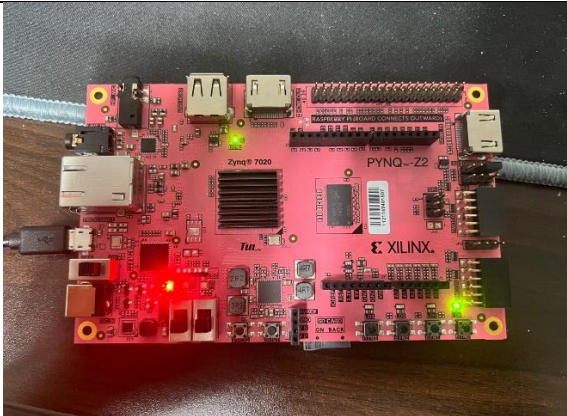
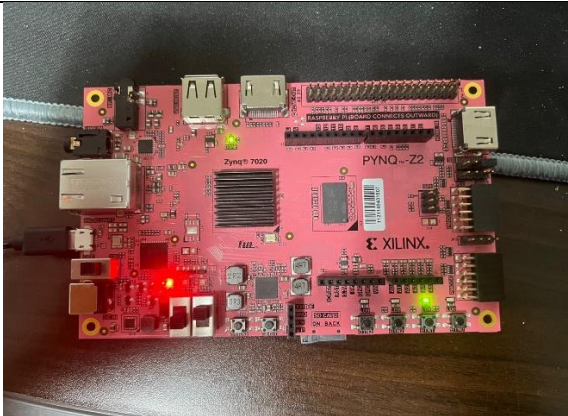
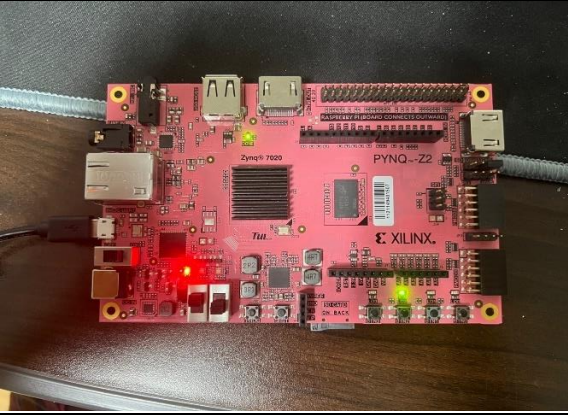
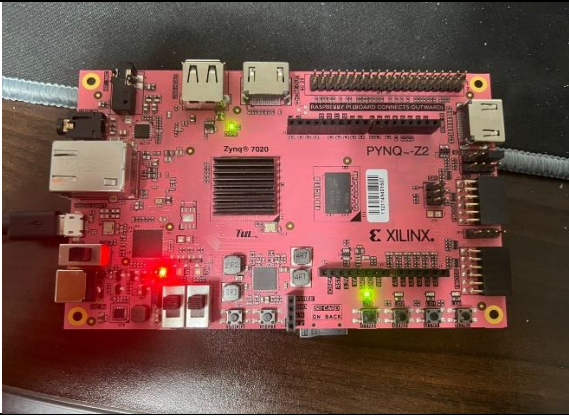
입력값(input_decoder)	출력값(output_decoder)
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

이는 이론에서 나타낸 진리표와 동일한 결과를 나타낸다. 3비트값이 가지는 모든 경우에 따라 출력값이 다 다르게 나타나는 것을 확인할 수 있다.

4. FPGA 결과

- PYNQ LED 실험

비트스트림을 생성하고 보드와 연결하였을 때 나타난 결과는 다음과 같다.

Switch 값이 00일 때 -> LED0 이 ON	Switch 값이 01일 때 -> LED1 이 ON
	
Switch 값이 10일 때 -> LED2 이 ON	Switch 값이 00일 때 -> LED0 이 ON
	

결과 분석

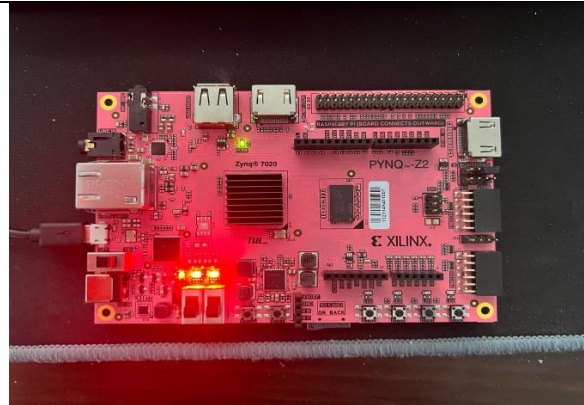
Switch	LED
00	LED0 ON
01	LED1 ON
10	LED2 ON
11	LED3 ON

이론과 Waveform 시뮬레이션에서 분석한 결과와 같이, 보드에도 동일하게 각 스위치에 따른 LED 값이 다르게 나타나는 것을 확인할 수 있다. DEMUX의 정의인 1개의 입력값과 2비트 컨트롤 값에 따른 출력값이 여러 개 나타날 수 있다는 것을 위 실험을 통해 증명하였다.

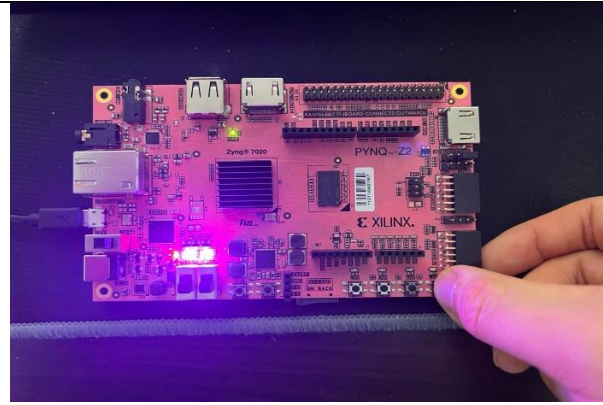
-PYNQ RGB 실험

각각의 다른 색을 보드로 구현한 결과는 다음과 같다.

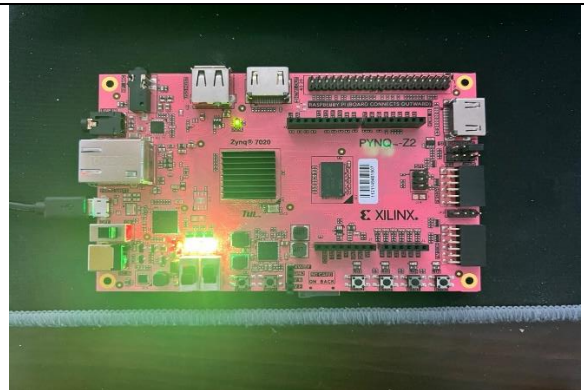
Switch값이 00, 버튼이 0일 때 -> Red



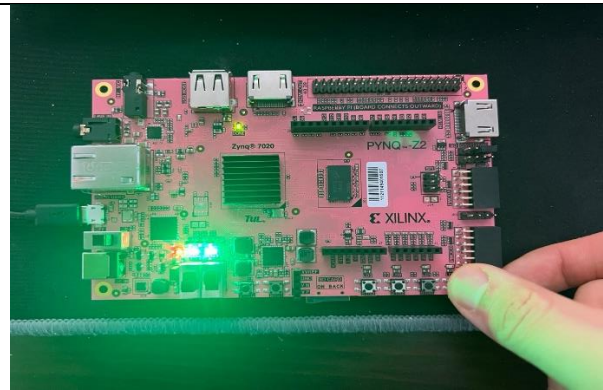
Switch값이 00, 버튼이 1일 때 -> Magenta



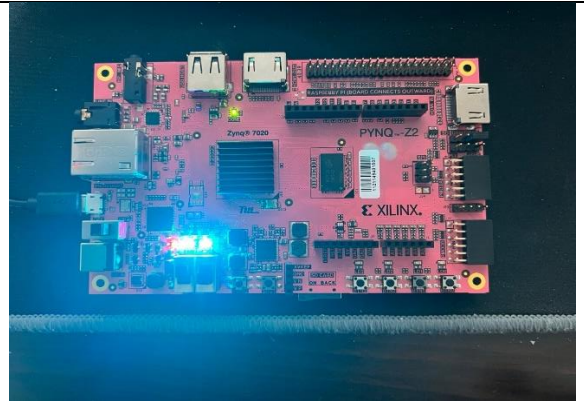
Switch값이 01, 버튼이 0일 때 -> Yellow



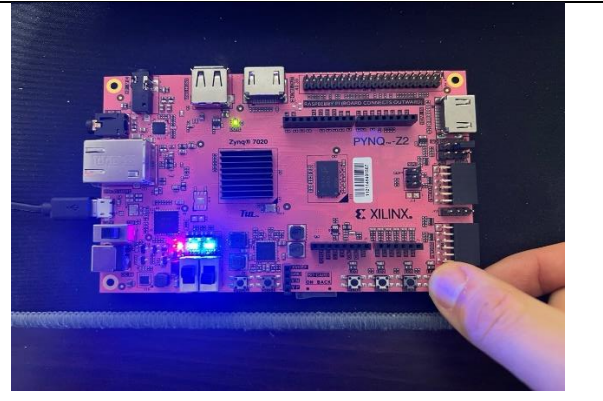
Switch값이 01, 버튼이 1일 때 -> Green



Switch값이 10, 버튼이 0일 때 -> Cyan



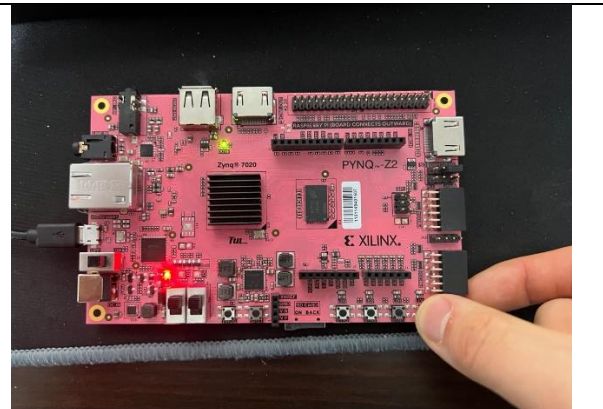
Switch값이 10, 버튼이 1일 때 -> Blue



Switch값이 11, 버튼이 0일 때 -> White



Switch값이 11, 버튼이 1일 때 -> LED off

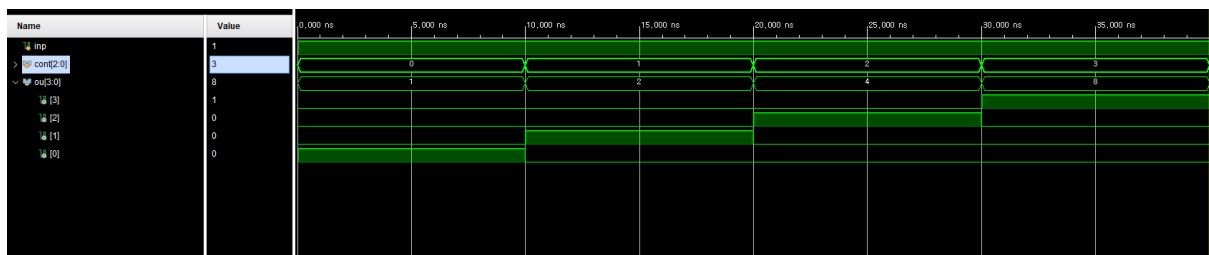


결과 분석

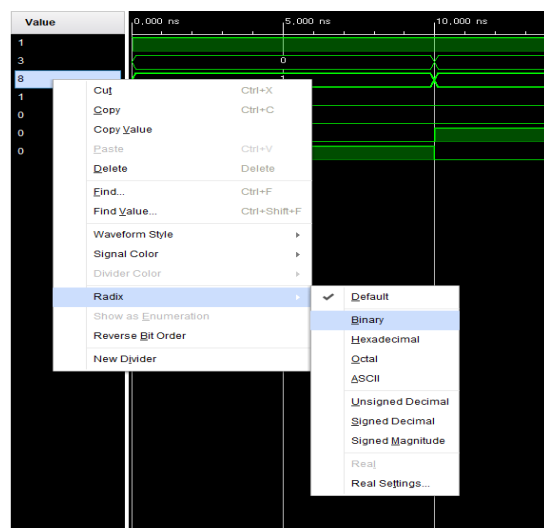
Inputs			Outputs						Color
SW1	SW2	BTN0	LED4_R	LED4_G	LED4_B	LED5_R	LED5_G	LED5_B	
0	0	0	1	0	0	1	0	0	Red
0	0	1	1	0	1	1	0	1	Magenta
0	1	0	1	1	0	1	1	0	Yellow
0	1	1	0	1	0	0	1	0	Green
1	0	0	0	1	1	0	1	1	Cyan
1	0	1	0	0	1	0	0	1	Blue
1	1	0	1	1	1	1	1	1	White
1	1	1	0	0	0	0	0	0	LED off

위 진리표에 따르면 2개의 switch와 1개의 버튼에 따라서 총 8가지의 경우가 나오며, 이를 6가지의 LED에 각각 다른 색으로 결과를 출력한다는 것을 알 수 있다. 이를 통해 3개의 입력값을 받으면 8개의 값을 출력하는 decoder를 올바르게 구현하였다.

4. 논의



Simulation을 출력할 때, 각 입력값 및 출력값을 default값으로 1,2,4,8 이렇게 나타내는 것을 알 수 있다. 그러나, 진리표와 비교했을 때, 진리표 상에서는 2진수로 구현이 되어 있기 때문에 이와 동일하게 어떻게 나타낼 수 있을까를 고민하게 되었다.



결과적으로 Value 값에 Radix part를 보면 쉽게 binary로 수정이 가능하다는 것을 알았고, 이를 통해 DEMUX와 Decoder에 대한 결과값을 2진수로 나타내어 이론과 편하게 비교할 수 있었다.