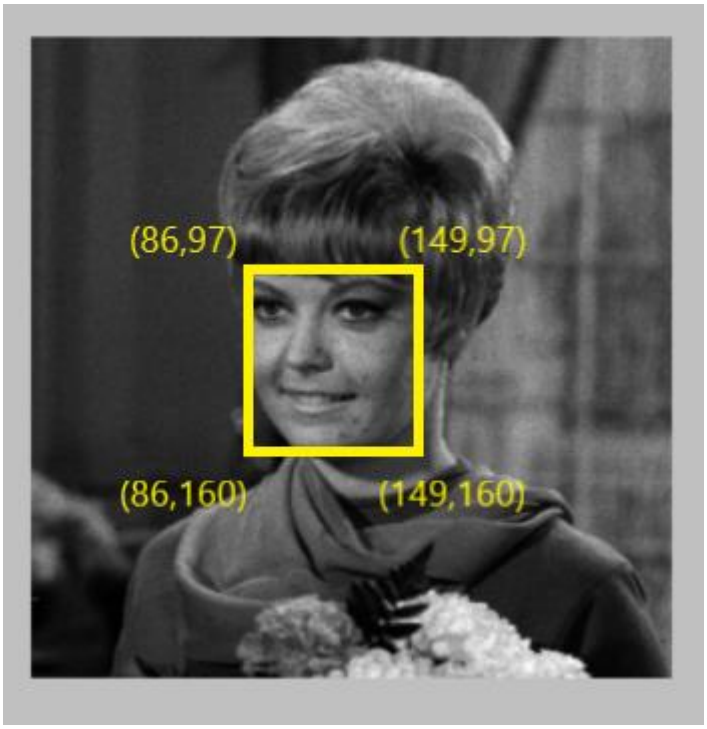
FACE ENCRYPTION, DECRYPTION

1. 준비

- encryption 할 얼굴 영역 좌표 정하기



<그림1> 얼굴영역의 좌표 설정

먼저 영역 좌표를 설정해주기 전, 각 픽셀의 주소값은 256*256이므로 0~65535까지의 값을 할당 받게 된다. 이후 얼굴 영역 좌표를 설정해주었는데 먼저 input 주소값을 받아 LFSR을 통해 random 난수를 만들고 키 값이 주소의 비트에 사용되므로 output 주소값은 input 주소값의 비트 수로 표현된다. 따라서 주소가 16비트의 수로 표현된다고 해서 encryption 모듈에 주소를 16비트로 준다면, output 주소는 0부터 65535 사이의 주소값을 갖게 된다. 이는 설정 영역밖의 주소를 가지고 올 수도 있다. 이는 encryption 에서 문제를 일으키진 않지만, decryption 과정에서 문제를

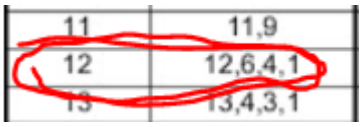
처음에 이를 고려하지 않고 내 맘대로 얼굴영역을 설정해서 주소를 16bit로 받는 random 난수를 만들어보았는데 결국 output 주소값이 0~65535의 값을 가지게 되므로 나중에 decryption에서 원래의 주소가 돌아오지 않았다. 따라서 나는 얼굴 영역의 크기를 64*64로 잡고 주소의 비트수를 2^12인 12비트로 설정해주었다. 이러면 인풋주소와 아웃풋주소가 이 얼굴영역의 범위를 벗어나지 않게 된다.

2. Encryption.v

주어진 링크를 참고하여 이 이미지를 어떻게 encryption 할 지를 생각해보았다. 먼저 우리는 주소에 저장 되어있는 8비트짜리 픽셀값이 아닌 주소값을 통해 얼굴 영역을 encryption 해야 한다. 따라서 LFSR을 통해 random 난수를 생성할 때, key값은 픽셀이 아닌 주소의 비트에 적용을 시켜야 한다.

먼저 LFSR을 분석해보았다. 이는 어떠한 입력을 받은 값의 비트들 중 몇 개를 골라 서로 XOR시킨 후 입력 값을 Left Shift 시킨 뒤 그 XOR한 결과를 LSB에 넣어주고 이를 그 비트 수만큼 반복시키는 것이다. 결과적으로 비트 수만큼 SHIFT가 발생하고 원래의 값이 아닌 새로운 값이 발생하게 된다.

나는 모듈로 넘길 주소의 비트수를 12bit라고 설정했으므로 비트의 12, 6, 4, 1자리가 key값을 받아 XOR 시킨다.



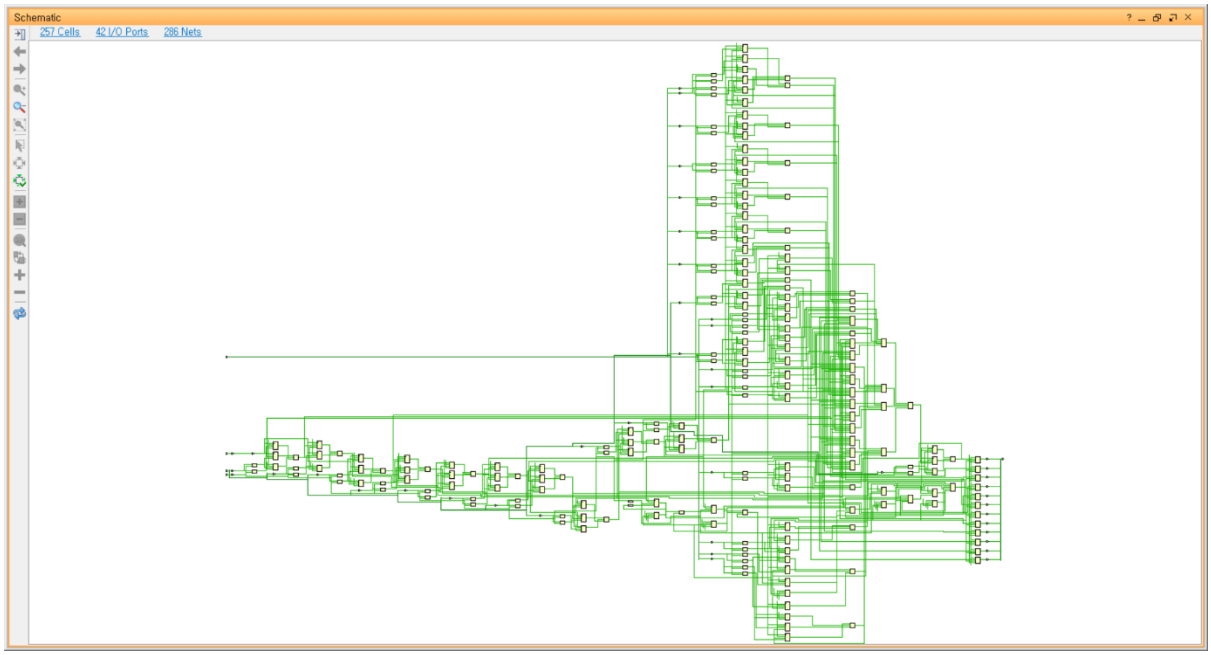
<그림2>: 비트수에 따른 XOR시켜야할 자릿수

```
module scramble(in_addr, key, clk, reset, out_addr);
input [11:0]in_addr;
input [15:0]key;
input clk, reset;
output [11:0]out_addr;

reg [11:0] random, random_next, random_done;
reg [4:0] count, count_next;
reg [3:0] k1, k2, k3, k4;
wire feedback = (random[k4] ^ random[k3] ^ random[k2] ^ random[k1]);

always@(posedge clk or negedge reset) begin
    if(!reset) begin
        {k4,k3,k2,k1}=key;
        random <= in_addr;
        count <= 0;
    end
    else begin
         random <= random_next;
         count <= count_next;
    end
end

always@(*) begin
    random_next = random;
    count_next = count;
```

```
        random_next = {random[10:0],feedback};
        count_next = count+1;
        if(count == 12)  begin
            count = 0;
            random_done = random;
        end
    end

    assign out_addr=random_done;

    endmodule
```

<그림3> encryption 모듈

Key 값은 testbench에서 한번에 입력 받으므로 입력 받은 key값을 각 자리를 표현할 수 있는 비트로 나눈다. 가장 큰 자리수인 12는 4비트로 표현이 가능하므로 key값은 16비트로 입력받아 각각 4비트로 된 k4, k3, k2, k1으로 나눠준다. 이렇게 키 값을 통해 각 자리를 XOR 시킬 수 있다.



<그림4> Synthesizer schematic about encryption module

3. testbench_encryption.v

```
module tb_scramble();
reg clk, reset;
reg [11:0]in_addr;
reg [15:0]key;
wire [11:0] out_addr;
reg [7:0] RegisterA[65535:0];
reg [7:0] RegisterB[65535:0];
reg [7:0] Face[4095:0];
integer i,j,k,l,f;

scramble my_scramble(in_addr, key, clk, reset, out_addr);
initial begin
    $readmemh("C:\\gray\\GIRL.hex",RegisterA);
    clk=1;
    key=16'b1011_0101_0011_0000;
    f=0;
    for (k=97;k<161;k=k+1)begin
        for(l=86;l<150;l=l+1)begin
            Face[f]=RegisterA[256*k+l];
            f=f+1;
        end
    end

    #10
    in_addr=0;
    for(i=0;i<256;i=i+1)begin //세로
        for(j=0;j<256;j=j+1)begin //가로
            if((i>96 && i<161) && (j>85 && j<150)) begin
            reset=0;
            #1 reset=1;
            #24 RegisterB[256*i+j]=Face[out_addr];
            in_addr=in_addr+1;
            end
            else
            RegisterB[256*i+j]=RegisterA[256*i+j];
        end
    end

    #10 $writememh("C:\\gray\\GIRLSCRAMBLE.hex",RegisterB);
end

always #1 clk = ~clk;

endmodule
```

그림5> tb_encryption

원래의 gray이미지를 불러올 RegisterA와 얼굴영역부분만을 따로 저장할 2^12 bit 크기를 가진 Face Register, 그리고 encryption된 이미지를 저장할 RegisterB를 선언해주었다.

RegisterA에 readmemh을 사용해 gray 이미지를 저장시켜주었다. 이후 설정한 얼굴 영역의 좌표만을 Face Register에 담기 위해 이중 포문을 사용하였는데 여기서 integer k는 세로, l은 가로를 의미한다. 그리고 input 주소를 0부터 for문을 통해 차례로 넣어주었다. 이 때, 주의해야 할 것은 이

중 for문안에 조건문을 달아줘서 RegisterB의 encryption될 영역을 설정한 뒤, encoding 모듈에서 주소를 넘겨받으려면 negedge reset이 작동되어야 하므로 주소를 넣어줄 때마다 reset을 해주었다. 그리고 12비트의 주소를 output 주소로 반환 받기 위해서는 클락이 12번 돌아야하므로 24초의 delay를 설정해준 뒤, RegisterB에 Face안의 output 주소를 저장시켜주었다. 설정한 영역이외에는 원래의 원본이 RegisterB에 저장된다.

<그림6> encryption 된 girl

4. decryption.v

LFSR을 통해 난수를 만든 주소를 다시 원래의 주소로 되돌려야 한다. 가장 먼저 든 생각은 Left shift를 사용했으니 Right shift를 사용하면 다시 돌아올 것 같았다. 또, Left shift를 하였을 때 feedback을 LSB에 저장했으니 Right shift에서는 MSB에 저장하면 될 것 같았다. 그러나 4비트의 수로 직접 해보니 원래의 값이 돌아오지 않았다. 그렇게 고민하다가 encryption에서 각 비트를 XOR 시키는 자릿수를 보니 MSB는 꼭 XOR을 시키는 것에 포함이 되어있었다. 그래서 이번에는 Right shift를 시킬 때 버려지는 LSB를 어떻게 활용하면 좋을까라고 생각해보았는데 먼저 사라지는 LSB를 Right shift한 값의 MSB에 대입을 시킨 후 키 값을 넘겨받는 자리 수를 XOR한 feedback을 MSB에 넣어주었다. 신기하게도 원래의 주소값이 돌아왔다.

아래는 예시이다. (4비트는 비트의 4, 3자리를 key값으로 넘겨받음)

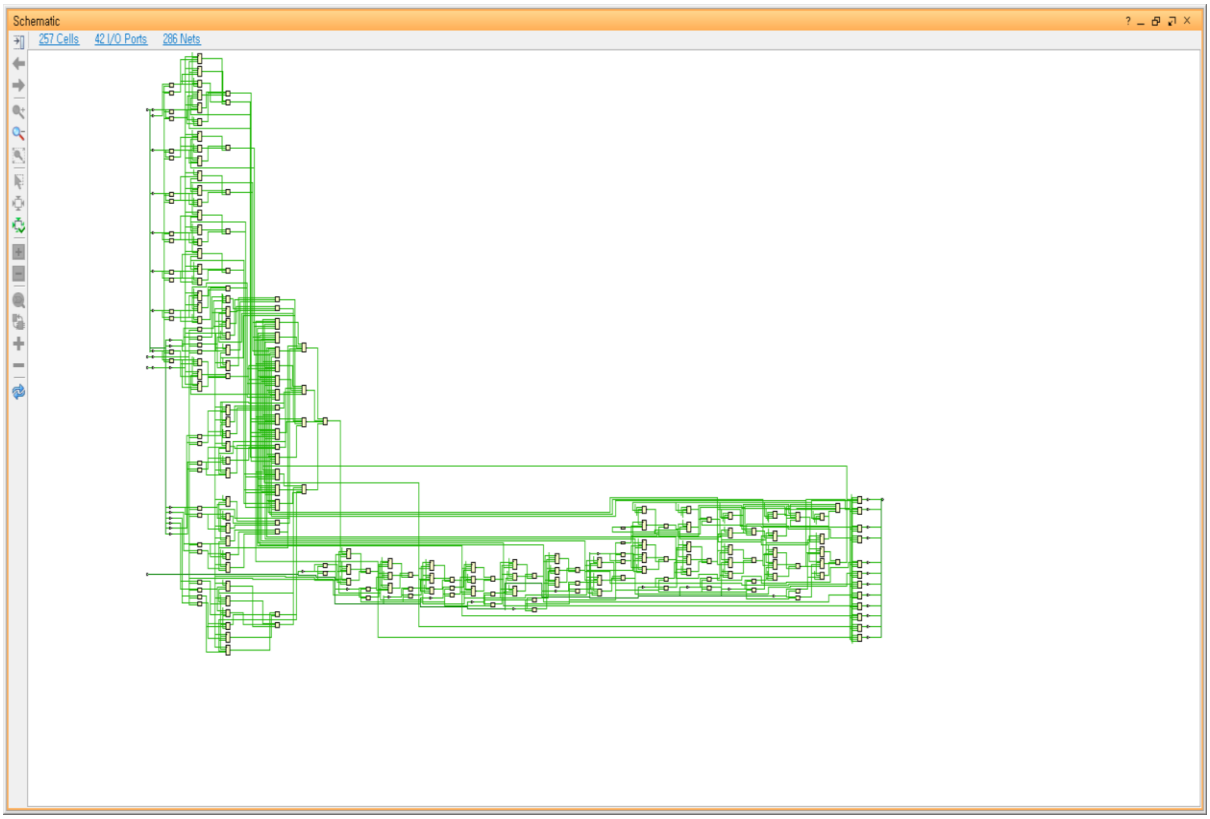
encryption에서의 난수 생성) 1011->0111->1111->1110->1100 return 1100

decryption에서복원)1100(start)->0110->1110(count=1)->0111->1111(count=2)->1111->0111(count=3)->1011->1011(count=4) return 1011

직접 다른 값들에 대해서도 수행한 결과 이 알고리즘이 성립했음을 확인할 수 있었다.

```
module descramble(in_addr, key, clk, reset, out_addr);
input [11:0]in_addr;
input [15:0]key;
input clk, reset;
output [11:0]out_addr;

reg [11:0] random, random_next, random_done, random_r;
reg [4:0] count, count_next;
reg [3:0] k1, k2, k3, k4;
wire feedback = (random_r[k4] ^ random_r[k3] ^ random_r[k2] ^ random_r[k1]);

always@(posedge clk or negedge reset) begin
    if(!reset) begin
        {k4,k3,k2,k1}=key;
        random <= in_addr;
        count <= 0;
    end
    else begin
         random <= random_next;
         count <= count_next;
    end
end
always@(*) begin
    random_next = random; //default
    count_next = count;//default
    random_r = random;

    random_r={random[0],random[11:1]};
    random_next = {feedback,random_r[10:0]};
    count_next = count+1;
    if(count == 12)  begin
        count = 0;
        random_done = random;
    end
end

assign out_addr=random_done;

endmodule
```

<그림7>decryption.v



<그림8> Synthesizer schematic about decryption module

5. testbench_decryption.v

```
module tb_descramble();
reg clk, reset;
reg [11:0]in_addr;
reg [15:0]key;
wire [11:0] out_addr;
reg [7:0] RegisterA[65535:0];
reg [7:0] RegisterB[65535:0];
reg [7:0] Face[4095:0];
integer i,j,k,l,f;

descramble my_descramble(in_addr, key, clk, reset, out_addr);

initial begin
    $readmemh("C:₩₩gray₩₩GIRLSCRAMBLE.hex",RegisterA);
    clk=1;
    key=16'b1011_0101_0011_0000;
    f=0;
    for (k=97;k<161;k=k+1)begin
        for(l=86;l<150;l=l+1)begin
            Face[f]=RegisterA[256*k+l];
            f=f+1;
        end
    end
```

```
    #10
    in_addr=0;
    for(i=0;i<256;i=i+1)begin //세로
        for(j=0;j<256;j=j+1)begin //가로
            if((i>96 && i<161) && (j>85 && j<150)) begin
            reset=0;
            #1 reset=1;
            #24 RegisterB[256*i+j]=Face[out_addr];
            in_addr=in_addr+1;
            end
            else
            RegisterB[256*i+j]=RegisterA[256*i+j];
        end
    end

    #10 $writememh("C:\\gray\\GIRLDESCRAMBLE.hex",RegisterB);
end

always #1 clk = ~clk;

endmodule
```

<그림8>

decryption의 테스트벤치는 encryption의 테스트벤치 구성과 똑같다고 보면 된다. 다만, 다른점은 readmemh으로 encryption된 파일을 불러오는 점과 writememh으로 다른 decryption된 파일을 Dumpling시키는 것뿐이다.


<그림9>원상복구된 gray 파일

6. 느낀 점 및 고찰

이번 final project를 진행하면서 이전에 했던 mid project가 도움이 된 것 같다. mid project를 통해 많은 시행착오를 겪으면서 주소를 어떻게 넘겨주어야 하는지, 파일을 어떻게 불러오고 Dump 시킬 수 있는지 알았고 Register에 저장된 주소들을 for문을 통해 어떻게 표현해야 하는지를 알았다.

LFSR을 통해 난수를 생성했는데 입력 값의 비트 수만큼 count변수를 선언해 shift를 시켜준다면 정말 새로운 난수가 나와 이미지들을 잘 섞어주어 이미지가 모자이크 처리가 잘 됐다. 그리고 참고자료에 나온 설명들이 워낙 잘 되어있고 친절하게 모듈까지 나와있어서 encryption 단계에서는 별 어려움없이 구현할 수 있었다. 그러나 decryption 과정에서 알고리즘을 찾느라 시간을 투자해야 했다.

가장 어려웠던 곳은 테스트 벤치 단계였다. 테스트 벤치에서는 주소값을 모듈이 입력 받을 수 있도록 reset도 매번 설정했고 이중 포문을 통해 영역을 따로 정해주고 Face라는 Register를 따로

할당해 얼굴영역의 주소들을 새로 배정받은 주소값을 넘겨주는 형식으로 구현했다. 이렇게 한 이유는 RegisterA만 가지고는 encryption은 가능하지만 decryption 단계에서 어려움을 느껴 Face라는 Register를 추가해주었다. 테스트벤치의 목적은 모듈이 잘 작동하는지 확인을 하기 위함으로 모듈이 잘 동작함을 보여준다면 큰 문제가 없다고 생각한다. 프로젝트를 진행하면서 한 학기동안 verilog의 사용법과 Chip에 관한 내용을 배워 앞으로 도움이 될 것이라 생각된다.