SoC설계 Final Project			
Major	Student ID	Grade	Name
융합전자공학부	2022099425	4	최훈서
융합전자공학부	2022085241	4	서석현

[Project 1: Sobel Filter]

1. Project 1 로직 설계 아이디어
   - Top level 설계

     Input Image는 ROM에서 한 클럭에 한 개의 픽셀씩만 불러올 수 있으므로, 3×3 커널과 곱해질 데이터를 순차적으로 불러와야 했다.
     이를 위해 Counter Logic과 Decoder Logic을 통합한 Control 모듈을 설계하였다. 이후, 9개의 데이터를 순차적으로 받아들이며 윈도우를 구성하기 위해 SIPO Shift Register 방식으로 구현하였다.

     Shift Register 는 9 클럭마다 데이터 수신이 완료되었음을 알리는 start_conv 신호와 함께 저장된 데이터를 다음 모듈로 전달한다.
     start_conv 신호를 받은 Conv_9 모듈은 9 개의 데이터에 동시에 접근하여, 로컬 파라미터로 정의된 커널 값들과 Convolution 연산을 수행한다.
     이때, SobelX 와 SobelY 를 모두 하나의 모듈 내에서 계산하며, 연산이 완료되면 두 결과를 결합하여 출력한다. convolution 연산이 완료되었음을 나타내는 신호도 함께 전송한다.

     -모듈1: control

     counter logic이 counter phase9개 만들어서, 9개 phase 돌리면서 각 phase에 해당하는 counter 값을 증가시킨다. 9bit로 선언된 Ring이 clk마다 돌며 Phase를 생성하고 Phase number와 일치하는 counter 값이 증가한다. 이때, 각 Counter의 초기값을 ImageData의 첫 Window(00,01,02;10,11,12;20,21,22;) 주소를 가리키게 한 다음 매 클락 증가시키며 window를 9clk마다 오른쪽으로 슬라이딩 했다. 이때, phase8(window[8]= 2행 2열 pixel)이 우측 끝에 도달하면 모든 counter에 3을 더해 다음 행 주소를 가리키게 한다.
     nstate에 다음 phase 값을 받아온 뒤 FSM처럼 clk마다 다음 phase를 넘기도록 하였고 이때 phase에 해당하는 coutner 값이 address에 출력되도록 하였다. 이러한 Logic을 통해 순차적으로 Window의 주소를 반환하며 ROM이 한 clk당 한 Pixel의 이미지 data 값을 출력할 수 있는 제한 사항을 만족시켰다.

     -모듈2: ImageROM

     control에서 출력하는 addr(address)을 입력받아 input.mem에 저장된 값을 한 개씩 읽어드린다. ImageROM 모듈은 8비트 grayscale 이미지 데이터를 저장한 ROM으로,

입력된 주소(addr)에 해당하는 데이터를 출력하는 역할을 한다. 클럭 신호와 enable 신호를 기반으로 동작하며, 외부 파일(input.mem)로부터 초기화된 데이터를 순차적으로 읽어 필터 연산 등의 전처리 단계에 활용된다.

-모듈3: Shift_Reg_9

Shift_Reg_9 모듈은 입력으로 들어오는 8비트 데이터를 클럭마다 하나씩 받아들이며, 이를 순차적으로 밀어 넣어 3×3 윈도우 형태로 구성하는 역할을 한다. 총 9개의 데이터가 입력되면 윈도우가 완성되며, 이때 start_conv 신호를 통해 convolution 연산이 가능함을 알려준다.

- -모듈4: conv_9

conv_9 모듈은 Shift_Reg_9로부터 입력받은 3×3 윈도우 데이터를 기반으로 Sobel 필터 연산을 수행하는 회로로, 경계(edge) 검출을 위한 기본 연산인 Sobel X(Gx)와 Sobel Y(Gy) 방향의 convolution을 동시에 수행한다. 각각의 결과를 16비트로 계산한 뒤, 두 결과를 하나의 32비트 데이터로 결합하여 출력한다.

2. Code 분석

-top

top

```
`timescale 1ns / 1ps

module top(
    input clk,
    input rst_n,
    input en,
    output conv_fin,
    output [31:0] output_word
    );

wire [9:0] addr;
wire [7:0] image_data;

control contt(
    .clk(clk),
    .rst_n(rst_n),
    .addr(addr),
```

```
        .en(en),
        .conv_fin(conv_fin)
);


ImageROM IR(
        .clk(clk),
        .rst_n(rst_n),
        .en(en),
        .addr(addr),
        .data_out(image_data)
);

wire start_conv;
wire [7:0] window00,window01,window02;
wire [7:0] window10,window11,window12;
wire [7:0] window20,window21,window22;

Shift_Reg_9 sr9(
        .clk(clk),
        .rst_n(rst_n),
        .din(image_data),
        .en(en),
        .start_conv(start_conv),
        .window00(window00), .window01(window01), .window02(window02),
        .window10(window10), .window11(window11), .window12(window12),
        .window20(window20), .window21(window21), .window22(window22)
);

conv_9 cv(
        .clk(clk),
        .rst_n(rst_n),
        .start_conv(start_conv),
        .window00(window00), .window01(window01), .window02(window02),
        .window10(window10), .window11(window11), .window12(window12),
        .window20(window20), .window21(window21), .window22(window22),
        .output_word(output_word)
);
```

```
endmodule
```

top 모듈은 전체 Sobel 필터 시스템의 상위 구조로서, 하위 모듈들을 인스턴스화하고 각 모듈 간의 신호를 연결해 전체 시스템을 통합하는 역할을 한다. 입력으로는 클럭(clk), 비동기 리셋(rst_n), enable 신호(en)를 받고, 출력으로는 convolution 연산이 끝났음을 알리는 신호(conv_fin)와 최종 연산 결과(output_word)를 제공한다.
코드의 상단에서 선언된 wire [9:0] addr과 wire [7:0] image_data는 하위 모듈 간의 연결을 위한 내부 신호선이다. 먼저 control 모듈 인스턴스를 통해 이미지 데이터에 접근할 주소(addr)를 생성하고, convolution이 끝났는지를 나타내는 conv_fin 신호를 출력으로 받는다. 이 주소는 다음 단계인 ImageROM 모듈의 입력으로 연결되며, ImageROM은 해당 주소의 데이터를 image_data라는 8비트 신호로 출력한다.
이렇게 불러온 이미지 데이터는 Shift_Reg_9 모듈의 입력 din으로 전달되며, 클럭과 enable 신호에 따라 내부 시프트 레지스터 구조를 통해 3×3 윈도우로 구성된다. 이 모듈의 출력은 3×3 윈도우를 나타내는 window00부터 window22까지 총 9개의 신호이며, 윈도우가 완전히 구성되었을 때 이를 알리는 start_conv 신호도 함께 출력된다.
최종적으로 conv_9 모듈은 윈도우의 9개 값을 입력받아 Sobel 필터 연산을 수행하고, 그 결과를 32비트 신호 output_word로 출력한다. 이 출력은 top 모듈의 최종 출력으로 연결되어 외부로 전달된다.

-모듈1: control

Control

```
`timescale 1ns / 1ps

module control(
    input   wire            clk,
    input   wire            rst_n,
    input   wire            en,
    output reg   [9:0]   addr,      // ROM 접근 주소 (0 ~ 1023)
    output reg conv_fin
);

    // ----------------
    // 1) Phase ring (one-hot, 9-bit)
    // ----------------
    reg [8:0] ring;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            ring <= 9'b100000000;           // phase0부터 시작
```

```
        else if (en)
            ring <= { ring[7:0], ring[8] };   // rotate right
    end

    // ----------------
    // 2) 9개의 카운터와 init 플래그
    // ----------------
    reg [9:0] count0, count1, count2,
                count3, count4, count5,
                count6, count7, count8;
    reg init0, init1, init2,
        init3, init4, init5,
        init6, init7, init8;

    // offset 값을 parameter로 정의 (00,01,02,32,33,34,64,65,66)
    localparam [9:0] OFF0 = 10'd00,
                        OFF1 = 10'd32,
                        OFF2 = 10'd64,
                        OFF3 = 10'd01,
                        OFF4 = 10'd33,
                        OFF5 = 10'd65,
                        OFF6 = 10'd02,
                        OFF7 = 10'd34,
                        OFF8 = 10'd66;

    // 마지막 주소가 1023에 도달시 fin
    assign addr_fin = (count8 == 10'd1023);

    reg d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12;
    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            d1          <= 1'b0;
            d2          <= 1'b0;
            d3          <= 1'b0;
            d4          <= 1'b0;
            d5          <= 1'b0;
            d6          <= 1'b0;
            d7          <= 1'b0;
            d8          <= 1'b0;
```

```
            d9          <= 1'b0;
            d10         <= 1'b0;
            d11         <= 1'b0;
            d12         <= 1'b0;

            conv_fin <= 1'b0;
        end
        else begin
            d1          <= addr_fin;
            d2          <= d1;
            d3          <= d2;
            d4          <= d3;
            d5          <= d4;
            d6          <= d5;
            d7          <= d6;
            d8          <= d7;
            d9          <= d8;
            d10         <= d9;
            d11         <= d10;
            d12         <= d11;
            conv_fin <= d12;
        end
    end

    // 각 phase별 count 블록 (예: phase0)
    always @(posedge ring[0] or negedge rst_n) begin
        if (!rst_n) begin
            count0 <= OFF0;
            init0   <= 1'b0;
        end else if (en) begin
            if (!init0) init0 <= 1'b1;
            else
                if(count8 % 32 == 31) count0 <= count0+3;
                else count0 <= count0 + 1;
        end
    end

    // phase1
    always @(posedge ring[1] or negedge rst_n) begin
```

```
        if (!rst_n) begin
            count1 <= OFF1;
            init1   <= 1'b0;
        end else if (en) begin
            if (!init1) init1 <= 1'b1;
            else
                if(count8 % 32 == 31) count1 <= count1+3;
                else count1 <= count1 + 1;
        end
    end

    // phase2
    always @(posedge ring[2] or negedge rst_n) begin
        if (!rst_n) begin
            count2 <= OFF2;
            init2   <= 1'b0;
        end else if (en) begin
            if (!init2) init2 <= 1'b1;
            else
                if(count8 % 32 == 31) count2 <= count2+3;
                else count2 <= count2 + 1;
        end
    end

    // phase3
    always @(posedge ring[3] or negedge rst_n) begin
        if (!rst_n) begin
            count3 <= OFF3;
            init3   <= 1'b0;
        end else if (en) begin
            if (!init3) init3 <= 1'b1;
            else
                if(count8 % 32 == 31) count3 <= count3 + 3;
                else count3 <= count3 + 1;
        end
    end

    // phase4
    always @(posedge ring[4] or negedge rst_n) begin
```

```
        if (!rst_n) begin
            count4 <= OFF4;
            init4   <= 1'b0;
        end else if (en) begin
            if (!init4) init4 <= 1'b1;
            else
                if(count8 % 32 == 31) count4 <= count4 + 3;
                else count4 <= count4 + 1;
        end
    end

    // phase5
    always @(posedge ring[5] or negedge rst_n) begin
        if (!rst_n) begin
            count5 <= OFF5;
            init5   <= 1'b0;
        end else if (en) begin
            if (!init5) init5 <= 1'b1;
            else
                if(count8 % 32 == 31) count5 <= count5+3;
                else count5 <= count5 + 1;
        end
    end

    // phase6
    always @(posedge ring[6] or negedge rst_n) begin
        if (!rst_n) begin
            count6 <= OFF6;
            init6   <= 1'b0;
        end else if (en) begin
            if (!init6) init6 <= 1'b1;
            else
                if(count8 % 32 == 31) count6 <= count6 + 3;
                else count6 <= count6 + 1;
        end
    end

    // phase7
    always @(posedge ring[7] or negedge rst_n) begin
```

```
        if (!rst_n) begin
            count7 <= OFF7;
            init7   <= 1'b0;
        end else if (en) begin
            if (!init7) init7 <= 1'b1;
            else
                if(count8 % 32 == 31) count7 <= count7 + 3;
                else count7 <= count7 + 1;
        end
    end

    // phase8
    always @(posedge ring[8] or negedge rst_n) begin
        if (!rst_n) begin
            count8 <= OFF8;
            init8   <= 1'b0;
        end else if (en) begin
            if (!init8) init8 <= 1'b1;
            else
                if(count8 % 32 == 31) count8 <= count8 + 3;
                else count8 <= count8 + 1;
        end
    end

    // ----------------
    // 3) 현재 phase에 대응하는 addr 선택
    // ----------------
    reg [9:0] addr_next;
    always @(*) begin
        case (ring)
            9'b000000001: addr_next = init0 ? count0 : 10'b0;
            9'b000000010: addr_next = init1 ? count1 : 10'b0;
            9'b000000100: addr_next = init2 ? count2 : 10'b0;
            9'b000001000: addr_next = init3 ? count3 : 10'b0;
            9'b000010000: addr_next = init4 ? count4 : 10'b0;
            9'b000100000: addr_next = init5 ? count5 : 10'b0;
            9'b001000000: addr_next = init6 ? count6 : 10'b0;
            9'b010000000: addr_next = init7 ? count7 : 10'b0;
            9'b100000000: addr_next = init8 ? count8 : 10'b0;
```

```
            default:        addr_next = 10'b0;
        endcase
    end

    // ----------------
    // 4) addr 출력 레지스터
    // ----------------
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            addr <= 10'b0;
        else if (en)
            addr <= addr_next;
    end

endmodule
```

로직에서 설명했듯이, Ring으로 Phase를 생성하고 Phase에 해당하는 Counter 변수가 count되는 방식이다.

입력 포트: clk, rst_n, en

출력 포트: reg [9:0] addr, reg conv_fin

포트부터 설명하자면, Asynchronous 형식으로 디자인하였으며, 외부에서 clk과 reset signal을 받는다. Rst_n 신호 이후 En을 받아 Logic이 시작된다.

출력으로는 매 클럭 주소를 반환하며, Image Pixel이 끝에 도달 시 Convolution 연산이 완료되었다는 신호를 출력한다.

1. Phase Ring

   총 9개의 주소에 매 클럭 접근해야 하기 때문에 9개의 phase를 갖는 ring을 제작해야 한다. 이를 위해 9bit reg를 선언하였으며 rst_n시 Phase 0부터 시작하기 위해 9'b100000000으로 초기화 한다. En 신호가 켜지면 ring의 MSB를 LSB로 이동시키며 순환하는 phase generator를 생성한다.

   Window를 3행 3열로 표현했을 때,

   phase0이 window[0][0]의 주소를,

   phase1이 window[1][0]의 주소를,

   phase2이 window[2][0]의 주소를,

   phase3이 window[0][1]의 주소를,

   phase4이 window[1][1]의 주소를,

   phase5이 window[2][1]의 주소를,

   pahse6이 window[0][2]의 주소를,

   phase7이 window[1][2]의 주소를,

phase8이 window[2][2]의 주소를 담당하는 형식이다. 그렇기에 아래처럼 초기값을 선언하여 이미지 pixel을 32x32라고 바라봤을 때 우측 맨 위의 Window로 초기화해주었다.

2. Local param 각 counter들이 시작할 count 값을 localparameter로 선언하였다.

OFF0 = 10'd00,	Window[0][0]의 초기 주소
OFF1 = 10'd32,	Window[1][0]의 초기 주소
OFF2 = 10'd64,	Window[2][0]의 초기 주소
OFF3 = 10'd01,	Window[0][1]의 초기 주소
OFF4 = 10'd33,	Window[1][1]의 초기 주소
OFF5 = 10'd65,	Window[2][1]의 초기 주소
OFF6 = 10'd02,	Window[0][2]의 초기 주소
OFF7 = 10'd34,	Window[1][2]의 초기 주소
OFF8 = 10'd66;	Window[2][2]의 초기 주소

3. Counting Logic always block x9
   각 Phase 별로 Count 블록을 선언했다. 각 블록은 자신의 phase의 상승 에지 또는 rst_n의 하강 에지에 실행된다. 위에 할당된 Local param으로 초기화 되며 enable 이후 init0~8신호가 각 블록에서 한 phase 상승에지 만큼 지연되어 출력된다. 각 블록 내부에서 init이 켜진 후부터 각 phase에 할당된 counter0~8을 증가시킨다. 이 counter 값은 각자 자신의 window가 가리켜야 하는 주소를 항상 가리키도록 설정되어 있으며 window의 2행 2열을 의미하는 count8이 우측 끝 pixel에 도달하면 count 값을 3개 늘려 왼쪽으로 이동시켜 줬다. Init을 넣은 이유는 초기 offset을 유지시키기 위함이다. 또한 addr가 항상 주소를 가질 수 있도록 해준다.
4. Next_address logic and address transition logic
   Ring이 변할 때 마다 자신의 변화를 감지하여 해당 count를 다음 주소에 할당한다. FSM처럼 다음 주소를 두고 clk마다 다음 주소를 현재 주소에 넘기며 끊김 없이 일정한 간격으로 순차적인 주소를 반환할 수 있도록 선언하였다.
5. Assign addr
   Init을 조건으로 Condition Operator를 사용하여 안정적인 출력을 유지했다.
6. 주소가 끝에 도달하면 Conv_fin을 반환하며, 이 신호는 내부에서 주소가 반환된 후 최종 pixel 값이 반환될 때까지 지연된 후 출력된다. 해당 buffer가 선언 되어있다.

ImageROM

```
`timescale 1ns / 1ps

module ImageROM (
    input clk,
    input rst_n,
    input en,
```

```
    input [9:0] addr,
    output reg [7:0] data_out
);
    reg en_d;
    reg post_en;

    always @(posedge clk) begin
        if (!rst_n) begin
            en_d      <= 0;
            post_en   <= 0;
        end else begin
            en_d      <= en;              // 1클럭 전의 en 저장
            post_en   <= en_d;            // 1클럭 뒤에 반영
        end
    end

    reg [7:0] rom [0:1023];

    initial begin
        $readmemh("input.mem", rom);
    end

    always @(posedge clk) begin
        if(!rst_n) begin
            data_out <= 0;
        end
        else
            if (post_en) data_out <= rom[addr]; //두 클럭 지연 후 출력

    end

endmodule
```

해당 모듈은 입력으로 들어온 주소에 해당하는 픽셀 값을 .mem 파일로부터 읽어와 출력하는 역할을 하며, 클럭 기반의 동기 동작과 enable 신호를 기반으로 동작 흐름을 제어한다.

이 모듈은 총 1024개의 8비트 데이터를 ROM에 저장하고(grayscale image 한 픽셀), 입력된 주소 addr에 해당하는 값을 data_out으로 출력한다.

입력 포트로는 클럭(clk), 리셋(rst_n), enable(en), 주소(addr[9:0])가 있으며, 출력은 ROM으로부터 읽어낸 8비트 데이터(data_out)이다. ROM은 reg [7:0] rom [0:1023]로 정의되어 있으며, 시뮬레이션 시작 시 $readmemh("input.mem", rom);을 통해 외부 파일로부터 초기화된다.

enable 신호는 바로 사용되지 않고 en_d, post_en을 통해 두 클럭 지연된다. 이는 다른 모듈과 타이밍을 맞추기 위해 설정하였다,

Shift_Reg_9

```
`timescale 1ns / 1ps

module Shift_Reg_9(
    input [7:0] din,
    input rst_n,
    input clk,
    input en,
    output [7:0] window00,window01,window02,
    output [7:0] window10,window11,window12,
    output [7:0] window20,window21,window22,
    output start_conv
    );
    reg [7:0] window [2:0][2:0]; //3x3 윈도우
    reg [3:0] count; //윈도우가 형성되었는지 판단하기 위한 장치
    reg post_en;  //타이밍 제어를 위한 딜레이

    assign window00 = window[0][0]; assign window01 = window[0][1]; assign window02 = window[0][2];
    assign window10 = window[1][0]; assign window11 = window[1][1]; assign window12 = window[1][2];
    assign window20 = window[2][0]; assign window21 = window[2][1]; assign window22 = window[2][2];

     //이미지 데이터가 준비되었다는 신호 출력
    assign start_conv = (count == 4'd9);

    reg en_d1, en_d2;

    always @(posedge clk) begin
        if (!rst_n) begin
            en_d1 <= 0;
            en_d2 <= 0;
        end else begin
            en_d1 <= en;          // 1클럭 지연
            en_d2 <= en_d1;
```

```
            post_en <= en_d2;      // 2클럭 지연
        end
    end


    // 윈도우가 형성되었는지 확인하기 위한 count
    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            count <= 4'd0;
        end
        else begin
            if(post_en) begin
                if(count == 4'd9) count <= 4'd1;
                else count <= count+1;
            end
        end
    end

//window를 만들기 위해 입력되는 픽셀을 shift시켜 윈도우를 완성
    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            window[0][0] <= 8'b0; window[0][1] <= 8'b0; window[0][2] <= 8'b0;
            window[1][0] <= 8'b0; window[1][1] <= 8'b0; window[1][2] <= 8'b0;
            window[2][0] <= 8'b0; window[2][1] <= 8'b0; window[2][2] <= 8'b0;
        end
        else begin
            if(post_en) begin
                window[2][2] <= din;
                window[1][2] <= window[2][2];
                window[0][2] <= window[1][2];
                window[2][1] <= window[0][2];
                window[1][1] <= window[2][1];
                window[0][1] <= window[1][1];
                window[2][0] <= window[0][1];
                window[1][0] <= window[2][0];
                window[0][0] <= window[1][0];
            end
        end
    end
```

```
endmodule
```

Shift_Reg_9 모듈은 입력으로 들어오는 8비트 단위의 픽셀 데이터를 클럭 신호에 따라 순차적으로 저장하며, 이를 통해 3×3 윈도우를 형성하는 역할을 한다. 이 윈도우는 후속 모듈인 conv_9 모듈에서 Sobel 필터 등의 convolution 연산을 수행할 수 있도록 데이터를 제공하는 핵심 구성 요소이다. 이 모듈은 Shift Register 구조로 설계되어 있으며, 한 픽셀씩 입력받으며 내부 3x3의 윈도우를 구성한다.

입력 포트로는 din (8비트 단일 픽셀), clk, rst_n (비동기 리셋), 그리고 동작 제어를 위한 en이 존재하고, 출력 포트는 3×3 형태로 윈도우 데이터(window00~window22)와 convolution 수행 조건이 만족되었음을 나타내는 start_conv 신호가 있다.

```
always @(posedge clk) begin
    if (!rst_n) begin
        en_d1 <= 0;
        en_d2 <= 0;
    end else begin
        en_d1 <= en;          // 1클럭 지연
        en_d2 <= en_d1;
        post_en <= en_d2;    // 2클럭 지연
    end
end
```

앞에 모듈과 마찬가지로 enable 신호 en을 2클럭 지연시키는 구조로, 실제 데이터 입력 및 시프트 연산이 이루어지는 시점을 조절하기 위한 딜레이 처리이다.

```
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        count <= 4'd0;
    end
    else begin
        if(post_en) begin
            if(count == 4'd9) count <= 4'd1;
            else count <= count+1;
        end
    end
end

assign start_conv = (count == 4'd9);
```

count가 9가 되면, 즉 9개의 연속된 픽셀이 입력되었을 때 start_conv 신호가 high로 설정되어 convolution 연산이 가능한 상태가 되었음을 알린다.

```
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        window[0][0] <= 8'b0; window[0][1] <= 8'b0; window[0][2] <= 8'b0;
        window[1][0] <= 8'b0; window[1][1] <= 8'b0; window[1][2] <= 8'b0;
        window[2][0] <= 8'b0; window[2][1] <= 8'b0; window[2][2] <= 8'b0;
    end
    else begin
        if(post_en) begin
            window[2][2] <= din;
            window[1][2] <= window[2][2];
            window[0][2] <= window[1][2];
            window[2][1] <= window[0][2];
            window[1][1] <= window[2][1];
            window[0][1] <= window[1][1];
            window[2][0] <= window[0][1];
            window[1][0] <= window[2][0];
            window[0][0] <= window[1][0];
        end
    end
end
```

데이터를 한 칸씩 밀어주는 구조로, 이러한 방식은 1열 → 2열 → 3열 순서로 픽셀이 입력될 때 3×3 윈도우를 올바르게 구성할 수 있도록 설계된 것이다. 즉, din이 들어올 때마다 세로 방향으로 쌓이게 되고, 한 열이 다 차면 다음 열로 이동하도록 시프트된다.
결과적으로 단일 픽셀 입력을 시간의 흐름에 따라 공간적으로 3×3 윈도우로 재구성하여 출력하는 역할을 하며, convolution 처리를 위한 필수 전처리 블록이라 할 수 있다.

conv_9

```
`timescale 1ns / 1ps

module conv_9(
    input clk,
    input rst_n,
    input start_conv,
    input [7:0] window00,window01,window02,
    input [7:0] window10,window11,window12,
    input [7:0] window20,window21,window22,
```

```
    output reg [31:0] output_word
    );
    reg signed [17:0] conv_temp_x, conv_temp_y;

    // Sobel Gx Kernel (signed 8-bit)
    localparam signed [7:0] k00 = -1, k01 = 0,   k02 = 1;
    localparam signed [7:0] k10 = -2, k11 = 0,   k12 = 2;
    localparam signed [7:0] k20 = -1, k21 = 0,   k22 = 1;

    // Sobel Gy Kernel (signed 8-bit)
    localparam signed [7:0] kGx00 = -1, kGx01 = -2, kGx02 = -1;
    localparam signed [7:0] kGx10 = 0,   kGx11 = 0,   kGx12 = 0;
    localparam signed [7:0] kGx20 = 1,   kGx21 = 2,   kGx22 = 1;



    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            output_word <= 32'd0;
            conv_temp_x <= 18'd0;
            conv_temp_y <= 18'd0;
        end
        else begin
            if (start_conv) begin
                // 1. Sobel X (Gx) direction convolution
                conv_temp_x <=
                        k00 * $signed({1'b0, window00}) + k01 * $signed({1'b0, window01}) + k02
* $signed({1'b0, window02})
                    + k10 * $signed({1'b0, window10}) + k11 * $signed({1'b0, window11}) + k12
* $signed({1'b0, window12})
                    + k20 * $signed({1'b0, window20}) + k21 * $signed({1'b0, window21}) + k22
* $signed({1'b0, window22});

                // 2. Sobel Y (Gy) direction convolution
                conv_temp_y <=
                        kGx00 * $signed({1'b0, window00}) + kGx01 * $signed({1'b0, window01})
+ kGx02 * $signed({1'b0, window02})
                    + kGx10 * $signed({1'b0, window10}) + kGx11 * $signed({1'b0, window11})
+ kGx12 * $signed({1'b0, window12})
```

```
                        + kGx20 * $signed({1'b0, window20}) + kGx21 * $signed({1'b0, window21})
+ kGx22 * $signed({1'b0, window22});

                    // 3. Concatenate Sobel X and Sobel Y to form 32-bit output
                    output_word <= {conv_temp_x[15:0], conv_temp_y[15:0]}; // {sobel_x[15:0],
sobel_y[15:0]}
                end
            end
        end

endmodule
```

conv_9 모듈은 입력된 3×3 윈도우 이미지 데이터를 기반으로 Sobel 필터를 적용하여 X 방향(Gx)과 Y 방향(Gy)의 에지 성분을 각각 계산한 뒤, 두 결과를 하나의 32비트 출력으로 합쳐 출력하는 convolution 연산 모듈이다. 이 모듈은 에지 검출을 위한 핵심적인 연산 유닛이며, 실제 영상 처리에서 물체의 edge을 검출하는 데 사용된다.
모듈의 입력은 클럭(clk), 리셋(rst_n), convolution 시작을 알리는 start_conv, 그리고 총 9개의 8비트 윈도우 입력(window00 ~ window22)이다. 이 윈도우는 상위 모듈에서 Shift Register를 통해 구성된 3×3 픽셀 블록으로, convolution 연산에 필요한 입력이다. 출력은 Gx와 Gy 결과를 결합한(concat) 32비트 레지스터 output_word이다.

```
// Sobel Gx Kernel (signed 8-bit)
    localparam signed [7:0] k00 = -1, k01 = 0,   k02 = 1;
    localparam signed [7:0] k10 = -2, k11 = 0,   k12 = 2;
    localparam signed [7:0] k20 = -1, k21 = 0,   k22 = 1;

    // Sobel Gy Kernel (signed 8-bit)
    localparam signed [7:0] kGx00 = -1, kGx01 = -2, kGx02 = -1;
    localparam signed [7:0] kGx10 = 0,   kGx11 = 0,   kGx12 = 0;
    localparam signed [7:0] kGx20 = 1,   kGx21 = 2,   kGx22 = 1;
```

위 부분은 Gx, Gy필터의 값들을 localparam으로 저장한 것이다,

```
conv_temp_x <=
                        k00 * $signed({1'b0, window00}) + k01 * $signed({1'b0, window01}) + k02
* $signed({1'b0, window02})
                    + k10 * $signed({1'b0, window10}) + k11 * $signed({1'b0, window11}) + k12
* $signed({1'b0, window12})
                    + k20 * $signed({1'b0, window20}) + k21 * $signed({1'b0, window21}) + k22
```

```
* $signed({1'b0, window22});

                    // 2. Sobel Y (Gy) direction convolution
                    conv_temp_y <=
                            kGx00 * $signed({1'b0, window00}) + kGx01 * $signed({1'b0, window01})
+ kGx02 * $signed({1'b0, window02})
                        + kGx10 * $signed({1'b0, window10}) + kGx11 * $signed({1'b0, window11})
+ kGx12 * $signed({1'b0, window12})
                        + kGx20 * $signed({1'b0, window20}) + kGx21 * $signed({1'b0, window21})
+ kGx22 * $signed({1'b0, window22});
```

위 코드는 convolution을 실제로 수행하는 부분이다. 입력 받은 윈도우를 미리 저장해둔 커널 값들과 곱한 것으로 x방향 y방향 각각 conv_temp_x, conv_temp_y에 결과를 저장하고 output_word <= {conv_temp_x[15:0], conv_temp_y[15:0]}; 를 통해 출력한다.

tb_top

```
`timescale 1ns / 1ps

module tb_top;

    // Inputs
    reg             clk;
    reg             rst_n;
    reg             en;
    wire [31:0]   output_word;
    wire            conv_fin;

    // ---- 추가된 신호 선언 ----
    // 1. 내부 start_conv를 hierarchical 참조할 레지스터
    reg             start_conv_d;

    // 2. 저장용 메모리와 인덱스
    localparam    MEM_DEPTH = 1025;
    reg   [31:0]   out_mem [0:MEM_DEPTH-1];

    // -------------------------

    // Instantiate the Unit Under Test (UUT)
```

```
top uut(
    .clk            (clk),
    .rst_n          (rst_n),
    .en              (en),
    .output_word (output_word),
    .conv_fin (conv_fin)
);

// Clock generation: 50MHz
initial clk = 0;
always #10 clk = ~clk;   // 20ns period

// ----------------------------------------
// start_conv 딜레이 + 첫 클럭 스킵 + 캡처
// ----------------------------------------
reg             seen_first;  // 첫 번째 이벤트를 봤는지
integer         idx;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        start_conv_d <= 1'b0;
        seen_first    <= 1'b0;
        idx             <= 0;
    end
    else begin
        // 1clk 딜레이된 start_conv
        start_conv_d <= uut.start_conv;

        // 딜레이된 신호가 올라올 때만 처리
        if (start_conv_d) begin
            if (!seen_first) begin
                // 첫 번째 start_conv_d: 플래그만 세우고 캡처 안 함
                seen_first <= 1'b1;
            end
            else begin
                // 그 다음부터는 output_word 캡처
                out_mem[idx] = output_word;
                idx = idx + 1;
            end
```

```
            end
        end
    end

    // ---- fin 감지시 덤프 & 종료 로직 추가 ----
    always @(posedge clk) begin
        if (conv_fin) begin
            $writememh("output_dump.mem", out_mem, 0, idx-1);
            $display("[%0t] fin detected! Dumped %0d words to output_dump.mem", $time, idx);
            $finish;
        end
    end
    // ----------------------------------------

    initial begin
        // Initialize inputs
        rst_n = 1;
        en      = 0;

        // Reset sequence
        #5   rst_n = 0;
        #2   rst_n = 1;

        // Enable convolution
        #23 en = 1;

        // (기존의 time-based 종료/덤프는 fin이 없으면 동작)
        #200_000;
        $writememh("output_dump.mem", out_mem, 0, idx-1);
        $display("Timeout reached. Dumped %0d words to output_dump.mem", idx);

        $display("Simulation finished.");
        $finish;
    end

endmodule
```

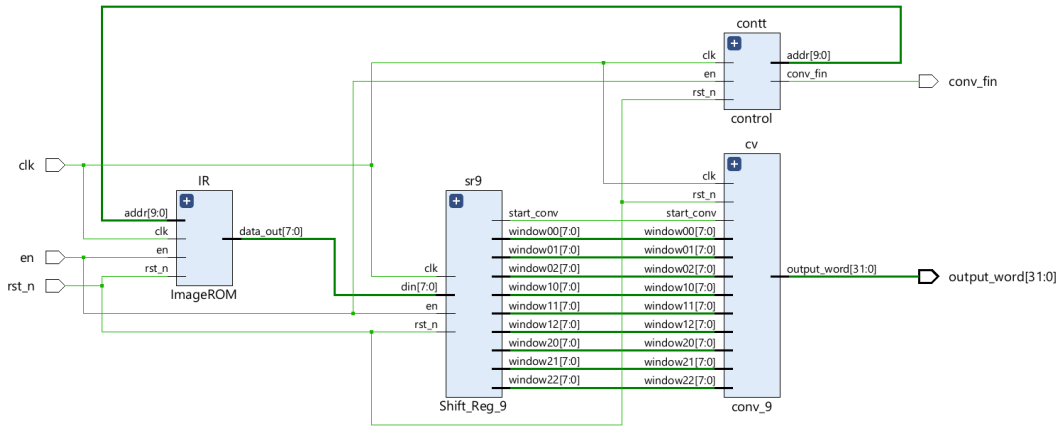
이 tb_top 모듈은 top 모듈에 대한 테스트벤치로, 실제 합성된 시스템을 시뮬레이션하고 그 출력을 output_dump.mem 파일로 저장하기 위한 용도로 설계되었다. 특히 이 테스트벤치는 Convolution 연산의 타이밍에 맞춰 결과를 정확히 수집할 수 있도록 세밀하게 동작을 제어하며,

start_conv 신호에 맞춰 데이터를 메모리에 저장하고, conv_fin 신호에 따라 시뮬레이션을 종료한다.

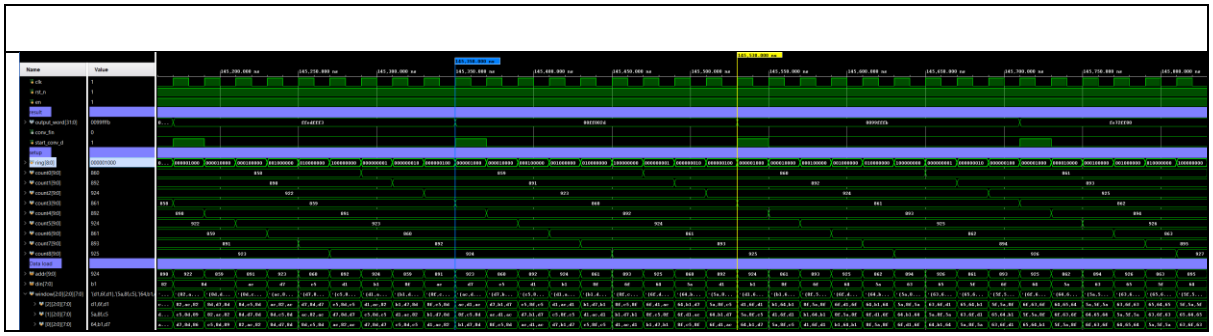
우선 클럭은 50MHz (20ns 주기)로 생성되며, initial 블록에서 리셋을 걸고 난 뒤 일정 시점에서 en 신호를 1로 올려 convolution 연산을 시작하도록 한다. 내부 모듈인 top.uut.start_conv 신호를 관찰하고, 이를 한 클럭 지연된 start_conv_d로 저장한 뒤, 그 상승 에지에 맞춰 결과를 output_word에서 추출한다.

또한 conv_fin 신호가 high가 되는 순간을 감지하여, 지금까지 수집한 데이터를 output_dump.mem 파일에 저장하고 시뮬레이션을 종료한다.

3. 시뮬레이션 분석



-모듈1: control



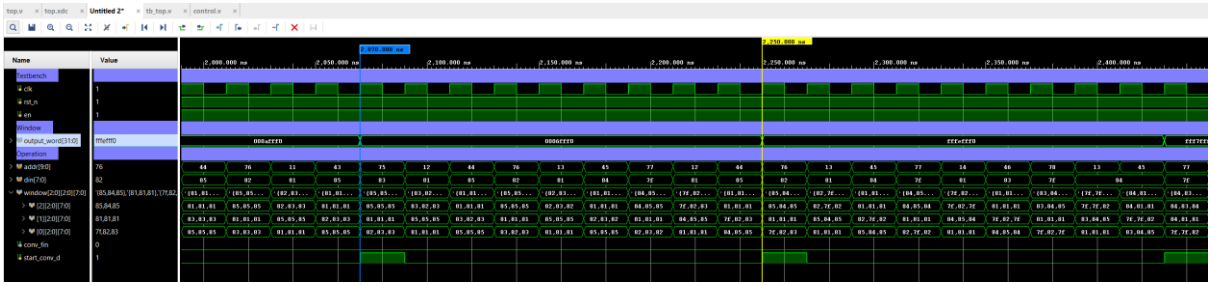
Control Waveform 분석

Rst_n이 들어오면 count0~count8이 모두 앞서 설명한 localParam 값으로 초기화된다. 이후, Ring이 clk마다 phase를 올바르게 생성하는 것을 확인할 수 있으며 각 Phase에 해당하는 counter

값이 증가하는 것을 확인할 수 있다. 각 counter 값이 9clk씩 유지되는 것이 그 증명이며 한 클럭씩 밀려서 변하는 것이 Phase generator인 Ring counter가 잘 작동한다는 의미이다. 이후 address는 count0~count8이 보유하고 있는 값이 그 다음 클럭에 잘 넘어가게 되어 올바른 count0->count1->count2-> … ->count8->count0 값을 순차적으로 출력하고 있다. 이 값은 window[0][0] ->window[1][0]-> … ->window[2][2]까지의 주소를 순차적으로 출력하는 것이다. 이렇게 Ring과 주소 반환이 올바르게 되고 있음을 확인할 수 있다.

ImageROM 분석

control에서 출력된 일정한 주기로 변하는 addr 신호를 입력으로 받아 din[7:0] 이 그 와 같은 주기로 출력이 나오는 것을 알 수 있다. Input.mem파일과 비교했을 때 addr에 해당하는 값들이 잘 출력되고 있음을 확인하였고 이는 모듈이 의도대로 동작함을 알 수 있다.



Shift_Register_9 모듈 분석

ImageROM으로 부터 데이터를 받아와(din) 윈도우를 형성해야한다. Din의 입력에 따른 window를 보면 입력되는 값들이 순차적으로 shift되면서 저장되는 것을 확인 할 수 있다. 또 start_conv의 딜레이 된 신호가 제 시간마다(9개의 데이터 입력시) 발생하는 것으로 보면 기대하던 동작을 수행함을 확인하였다.

conv_9모듈 분석

이전 모듈에서 입력받은 window의 값들과 start_conv_d 신호를 보면 start_conv신호에 따라 output_word가 업데이트 되는 모습을 볼 수있다. 이 때 타이밍을 확인하였고, convolution 연산 결과를 제공된 output.mem파일과 비교하였을 때 모든 데이터가 일치하였을 확인하였다. 따라서 정상 작동하는 모듈임을 확인하였다.

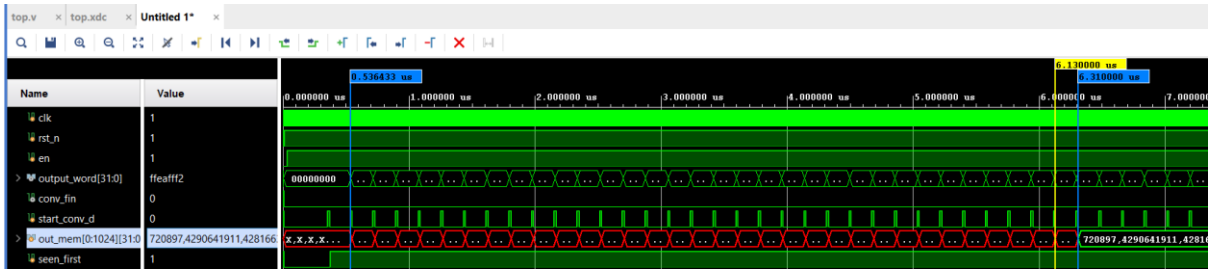
Timing ×

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.742 ns	Worst Hold Slack (WHS):	0.137 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	312	Total Number of Endpoints:	312	Total Number of Endpoints:	171

All user specified timing constraints are met.

Implementation이 오류없이 잘 이루어 졌고 slack들이 모두 음수가 아닌 것을 확인하였다.



Post-implementation timing simulation 결과

Start=0.536443us

End=6.13us으로 총 걸린 시간은 **5.593557us**이다. 이는 out_mem에 데이터가 들어오기 전에 마지막 출력이 이루어지기 때문에 위와 같은 방식으로 측정하였다.

4. 합성 결과 분석

```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |  243 |     0 |          0 |     47200 |  0.51 |
|   LUT as Logic          |  243 |     0 |          0 |     47200 |  0.51 |
|   LUT as Memory         |    0 |     0 |          0 |     19000 |  0.00 |
| Slice Registers         |  262 |     0 |          0 |     94400 |  0.28 |
|   Register as Flip Flop |  262 |     0 |          0 |     94400 |  0.28 |
|   Register as Latch     |    0 |     0 |          0 |     94400 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |     15850 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

총 243개의 LUT가 사용되었으며, 모두 조합 논리(LUT as Logic)로 활용되었고 LUT 메모리는 사용되지 않았다. 레지스터는 총 262개가 사용되었으며, 전부 Flip-Flop으로 구성되었고 latch는 단 하나도 사용되지 않았다. 이는 비의도적인 래치가 형성되지 않았음을 의미한다. 체 자원 대비 사용률은 LUT 기준 0.51%, 레지스터 기준 0.28%로, 낮은 자원 소모를 보인다.

2. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	0.5	0	0	105	0.48
RAMB36/FIFO*	0	0	0	105	0.00
RAMB18	1	0	0	210	0.48
RAMB18E1 only	1				

Block RAM 자원은 RAMB18 블록을 1 개 사용하여 총 용량의 약 0.48%를 소모하였다. RAMB36 또는 FIFO 형태는 사용되지 않았고, RAMB18E1 단일 블록으로만 구성되었다. 이는 입력 이미지 데이터를 저장하거나 연산을 위한 line buffer 등에 활용된 것으로 보이며, Gaussian 필터와 달리 RAMB18 타입이 선택된 점이 구조상의 차이를 나타낸다. 다만 사용량은 여전히 매우 적다.

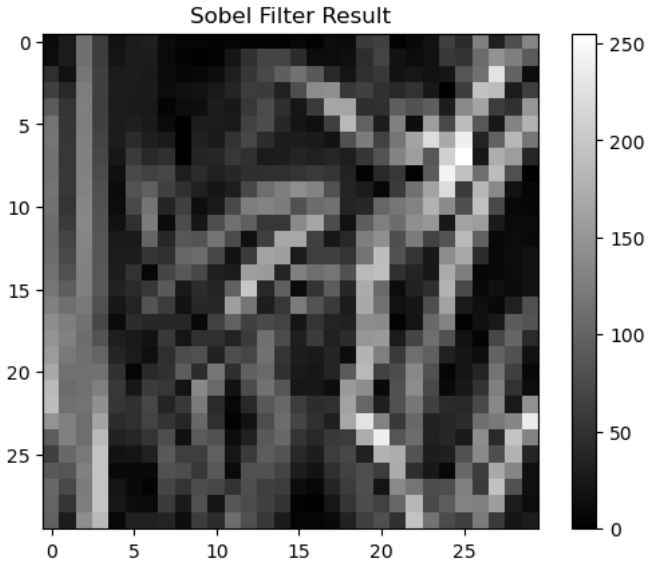
3. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	0	0	0	180	0.00

DSP 블록은 전혀 사용되지 않았다. Sobel 필터에서도 모든 곱셈 및 덧셈 연산이 LUT 기반으로 처리되었음을 의미하며, 이는 Sobel 커널 역시 계수가 단순(-1, 0, +1, +2 등)하기 때문에 DSP 없이도 계산이 가능하다는 것을 보여준다. 따라서 필터링 연산의 복잡도에 비해 연산 자원 요구가 매우 낮은 구조다.

5.이미지 결과


Figure 1
Original Image
Sobel Combined


Sobel Filter Result
0
5
10
15
20
25
250
200
150
100
50
0

[Project 2: Gaussian Blur]

1. Project 2 로직 설계 아이디어

-Top level 설계

. Input Image는 ROM에서 한 클럭당 한 개의 픽셀씩 불러올 수 있으므로, 3×3 커널과 곱해질 데이터를 순차적으로 받아와야 했다. 이를 위해 Counter Logic과 Decoder Logic을 통합한 control 모듈을 설계하였다. 이후 9개의 데이터를 순차적으로 수신하며 윈도우를 구성해야 했기 때문에, 이를 SIPO Shift Register 방식으로 구현하였다.

Sobel Filter와는 달리 Gaussian Blur는 R, G, B 세 개의 채널에 각각 적용되어야 하므로, 입력 데이터를 8비트 단위로 분리하여 각 채널을 구분하였다. Shift Register는 9클럭마다 데이터가 충분히 수집되었음을 알리는 start_conv 신호와 함께 저장한 데이터를 다음 모듈로 전달한다.

start_conv 신호를 받은 conv_9 모듈은 9개의 데이터를 한 번에 읽어, 로컬 파라미터로 정의된 커널 값들과 convolution 연산을 수행한다. 이 연산은 R, G, B 세 채널 각각에 대해 병렬로 진행되며, 연산이 완료되면 각 채널의 결과를 16으로 나눈 후 하나로 합쳐 출력한다. convolution 연산이 완료되었음을 나타내는 신호도 함께 전송한다.

-모듈1: control

Project 1의 control과 동일.

-모듈2: ImageROM

Project1의 ImageROM과 같음. 단, input.mem에 저장된 pixel이 24bit이므로 24bit 데이터를 읽어 출력한다.

-모듈3: Shift_Reg_9

Project 1의 Shift_Reg_9와 같음. 단, R, G, B채널을 구분하여 3개의 window를 만들어 값들을 분리시켰음.

-모듈4: conv_9

Project 1의 conv_9을 변형한 것으로, Sobel Filter에 해당하는 커널 대신 R, G, B 각 윈도우에 Gaussian Blur에 해당하는 커널을 적용하여 Convolution을 수행한다.

출력은 48비트의 output_word로, 각 채널의 결과를 {R[15:0], G[15:0], B[15:0]} 형태로 연결하여 출력한다.

마지막 단계에서는 각 채널의 Convolution 결과를 16으로 나눈 값을 출력하기 위해, 각 결과를 우측으로 4비트 시프트한 값을 연결(concatenate)하여 구성하였다.

2. Code 분석

top

```
`timescale 1ns / 1ps

module top(
    input clk,
    input rst_n,
    input en,
    output conv_fin,
    output [47:0] output_word
    );

wire [9:0] addr;
wire [23:0] image_data;

control contt(
    .clk(clk),
    .rst_n(rst_n),
    .addr(addr),
    .en(en),
    .conv_fin(conv_fin)
);


ImageROM IR(
    .clk(clk),
    .rst_n(rst_n),
    .en(en),
    .addr(addr),
    .data_out(image_data)
);

wire start_conv;
wire [7:0] R_window00,R_window01,R_window02;
wire [7:0] R_window10,R_window11,R_window12;
wire [7:0] R_window20,R_window21,R_window22;

wire [7:0] G_window00,G_window01,G_window02;
wire [7:0] G_window10,G_window11,G_window12;
```

```
wire [7:0] G_window20,G_window21,G_window22;

wire [7:0] B_window00,B_window01,B_window02;
wire [7:0] B_window10,B_window11,B_window12;
wire [7:0] B_window20,B_window21,B_window22;

Shift_Reg_9 sr9(
	.clk(clk),
	.rst_n(rst_n),
	.din(image_data),
	.en(en),
	.start_conv(start_conv),
	.R_window00(R_window00), .R_window01(R_window01), .R_window02(R_window02),
	.R_window10(R_window10), .R_window11(R_window11), .R_window12(R_window12),
	.R_window20(R_window20), .R_window21(R_window21), .R_window22(R_window22),
	.G_window00(G_window00), .G_window01(G_window01), .G_window02(G_window02),
	.G_window10(G_window10), .G_window11(G_window11), .G_window12(G_window12),
	.G_window20(G_window20), .G_window21(G_window21), .G_window22(G_window22),
	.B_window00(B_window00), .B_window01(B_window01), .B_window02(B_window02),
	.B_window10(B_window10), .B_window11(B_window11), .B_window12(B_window12),
	.B_window20(B_window20), .B_window21(B_window21), .B_window22(B_window22)


);

conv_9 cv(
	.clk(clk),
	.rst_n(rst_n),
	.start_conv(start_conv),
	.R_window00(R_window00), .R_window01(R_window01), .R_window02(R_window02),
	.R_window10(R_window10), .R_window11(R_window11), .R_window12(R_window12),
	.R_window20(R_window20), .R_window21(R_window21), .R_window22(R_window22),
	.G_window00(G_window00), .G_window01(G_window01), .G_window02(G_window02),
	.G_window10(G_window10), .G_window11(G_window11), .G_window12(G_window12),
	.G_window20(G_window20), .G_window21(G_window21), .G_window22(G_window22),
	.B_window00(B_window00), .B_window01(B_window01), .B_window02(B_window02),
	.B_window10(B_window10), .B_window11(B_window11), .B_window12(B_window12),
	.B_window20(B_window20), .B_window21(B_window21), .B_window22(B_window22),
	.output_word(output_word)
```

```
);


endmodule
```

이 top 모듈은 24비트 RGB 이미지를 입력받아 각 채널(R, G, B)에 대해 3×3 가우시안 커널을 적용한 후, 블러 처리된 결과를 48비트(output_word)로 출력하는 시스템의 상위 모듈이다. 구성된 하위 모듈들은 각각 메모리에서 이미지 데이터를 읽어오고, 윈도우 데이터를 구성하며, 최종적으로 convolution 연산을 수행하는 역할을 담당한다.
우선 control 모듈은 클럭과 리셋 신호를 기반으로 주소 생성 및 유효 신호(conv_fin) 출력을 제어한다. 여기서 생성된 주소는 ImageROM에 전달되어 해당 위치의 RGB 이미지 데이터를 가져오며, 이 데이터는 24비트(8비트씩 R, G, B)로 구성되어 있다.
ImageROM에서 전달된 24비트 data는 Shift_Reg_9 모듈로 입력되며, 이 모듈은 3×3 윈도우를 형성하기 위해 시리얼하게 입력되는 픽셀들을 레지스터 배열로 순차적으로 저장한다. 동시에 이 데이터를 RGB 채널별로 분리하여 R_window00부터 B_window22까지 총 27개의 윈도우 출력을 형성한다. 이때 윈도우의 구성 순서는 위쪽에서 아래쪽, 왼쪽에서 오른쪽 순으로 픽셀을 정렬해 convolution 연산에 적합하도록 구성되어 있다. 또한, 일정 수의 유효한 입력이 누적되었을 때 start_conv 신호가 high가 되어 convolution이 수행될 수 있는 시점을 알려준다.
conv_9 모듈은 start_conv 신호가 high가 될 때 윈도우 내의 9개 픽셀 값에 가우시안 커널을 적용하여 각각 R, G, B 채널에 대해 컨볼루션 결과를 계산하고, 이를 각 채널당 16비트씩 총 48비트로 합쳐 output_word로 출력한다. 이때 계산은 고정된 가우시안 커널 값들을 기반으로 이루어지며, 결과는 채널별 블러 효과가 적용된 픽셀 값이다.
결과적으로 이 top 모듈은 컨트롤 신호에 따라 시리얼하게 입력된 이미지 데이터를 처리하여, 각 픽셀마다 3×3 윈도우 기반의 가우시안 블러를 수행하고 최종 출력하는 전체 시스템을 구성한다.

control

```
`timescale 1ns / 1ps

module control(
    input   wire            clk,
    input   wire            rst_n,
    input   wire            en,
    output reg   [9:0]   addr,      // ROM 접근 주소 (0 ~ 1023)
    output reg conv_fin
);

    // ----------------
    // 1) Phase ring (one-hot, 9-bit)
```

```
// ----------------
reg [8:0] ring;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        ring <= 9'b100000000;          // phase0부터 시작
    else if (en)
        ring <= { ring[7:0], ring[8] };   // rotate right
end

// ----------------
// 2) 9개의 카운터와 init 플래그
// ----------------
reg [9:0] count0, count1, count2,
          count3, count4, count5,
          count6, count7, count8;
reg init0, init1, init2,
    init3, init4, init5,
    init6, init7, init8;

// offset 값을 parameter로 정의 (00,01,02,32,33,34,64,65,66)
localparam [9:0] OFF0 = 10'd00,
                 OFF1 = 10'd32,
                 OFF2 = 10'd64,
                 OFF3 = 10'd01,
                 OFF4 = 10'd33,
                 OFF5 = 10'd65,
                 OFF6 = 10'd02,
                 OFF7 = 10'd34,
                 OFF8 = 10'd66;

// 마지막 주소가 1023에 도달시 fin
assign addr_fin = (count8 == 10'd1023);

reg d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12;
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        d1      <= 1'b0;
        d2      <= 1'b0;
        d3      <= 1'b0;
```

```
            d4          <= 1'b0;
            d5          <= 1'b0;
            d6          <= 1'b0;
            d7          <= 1'b0;
            d8          <= 1'b0;
            d9          <= 1'b0;
            d10         <= 1'b0;
            d11         <= 1'b0;
            d12         <= 1'b0;

            conv_fin <= 1'b0;
        end
        else begin
            d1          <= addr_fin;
            d2          <= d1;
            d3          <= d2;
            d4          <= d3;
            d5          <= d4;
            d6          <= d5;
            d7          <= d6;
            d8          <= d7;
            d9          <= d8;
            d10         <= d9;
            d11         <= d10;
            d12         <= d11;
            conv_fin <= d12;
        end
    end

    // 각 phase별 count 블록 (예: phase0)
    always @(posedge ring[0] or negedge rst_n) begin
        if (!rst_n) begin
            count0 <= OFF0;
            init0   <= 1'b0;
        end else if (en) begin
            if (!init0) init0 <= 1'b1;
            else
                if(count8 % 32 == 31) count0 <= count0+3;
                else count0 <= count0 + 1;
```

```
        end
    end

    // phase1
    always @(posedge ring[1] or negedge rst_n) begin
        if (!rst_n) begin
            count1 <= OFF1;
            init1   <= 1'b0;
        end else if (en) begin
            if (!init1) init1 <= 1'b1;
            else
                if(count8 % 32 == 31) count1 <= count1+3;
                else count1 <= count1 + 1;
        end
    end

    // phase2
    always @(posedge ring[2] or negedge rst_n) begin
        if (!rst_n) begin
            count2 <= OFF2;
            init2   <= 1'b0;
        end else if (en) begin
            if (!init2) init2 <= 1'b1;
            else
                if(count8 % 32 == 31) count2 <= count2+3;
                else count2 <= count2 + 1;
        end
    end

    // phase3
    always @(posedge ring[3] or negedge rst_n) begin
        if (!rst_n) begin
            count3 <= OFF3;
            init3   <= 1'b0;
        end else if (en) begin
            if (!init3) init3 <= 1'b1;
            else
                if(count8 % 32 == 31) count3 <= count3 + 3;
                else count3 <= count3 + 1;
```

```
        end
    end

    // phase4
    always @(posedge ring[4] or negedge rst_n) begin
        if (!rst_n) begin
            count4 <= OFF4;
            init4   <= 1'b0;
        end else if (en) begin
            if (!init4) init4 <= 1'b1;
            else
                if(count8 % 32 == 31) count4 <= count4 + 3;
                else count4 <= count4 + 1;
        end
    end

    // phase5
    always @(posedge ring[5] or negedge rst_n) begin
        if (!rst_n) begin
            count5 <= OFF5;
            init5   <= 1'b0;
        end else if (en) begin
            if (!init5) init5 <= 1'b1;
            else
                if(count8 % 32 == 31) count5 <= count5+3;
                else count5 <= count5 + 1;
        end
    end

    // phase6
    always @(posedge ring[6] or negedge rst_n) begin
        if (!rst_n) begin
            count6 <= OFF6;
            init6   <= 1'b0;
        end else if (en) begin
            if (!init6) init6 <= 1'b1;
            else
                if(count8 % 32 == 31) count6 <= count6 + 3;
                else count6 <= count6 + 1;
```

```
        end
    end

    // phase7
    always @(posedge ring[7] or negedge rst_n) begin
        if (!rst_n) begin
            count7 <= OFF7;
            init7   <= 1'b0;
        end else if (en) begin
            if (!init7) init7 <= 1'b1;
            else
                if(count8 % 32 == 31) count7 <= count7 + 3;
                else count7 <= count7 + 1;
        end
    end

    // phase8
    always @(posedge ring[8] or negedge rst_n) begin
        if (!rst_n) begin
            count8 <= OFF8;
            init8   <= 1'b0;
        end else if (en) begin
            if (!init8) init8 <= 1'b1;
            else
                if(count8 % 32 == 31) count8 <= count8 + 3;
                else count8 <= count8 + 1;
        end
    end

    // ----------------
    // 3) 현재 phase에 대응하는 addr 선택
    // ----------------
    reg [9:0] addr_next;
    always @(*) begin
        case (ring)
            9'b000000001: addr_next = init0 ? count0 : 10'b0;
            9'b000000010: addr_next = init1 ? count1 : 10'b0;
            9'b000000100: addr_next = init2 ? count2 : 10'b0;
            9'b000001000: addr_next = init3 ? count3 : 10'b0;
```

```
                9'b000010000: addr_next = init4 ? count4 : 10'b0;
                9'b000100000: addr_next = init5 ? count5 : 10'b0;
                9'b001000000: addr_next = init6 ? count6 : 10'b0;
                9'b010000000: addr_next = init7 ? count7 : 10'b0;
                9'b100000000: addr_next = init8 ? count8 : 10'b0;
                default:          addr_next = 10'b0;
            endcase
        end

        // ----------------
        // 4) addr 출력 레지스터
        // ----------------
        always @(posedge clk or negedge rst_n) begin
            if (!rst_n)
                addr <= 10'b0;
            else if (en)
                addr <= addr_next;
        end

endmodule
```

Project 1의 control과 동일하다.

ImageROM

```
`timescale 1ns / 1ps

module ImageROM (
        input clk,
        input rst_n,
        input en,
        input [9:0] addr,
        output reg [23:0] data_out
);
        reg en_d;
        reg post_en;

        always @(posedge clk) begin
            if (!rst_n) begin
```

```
            en_d        <= 0;
            post_en     <= 0;
        end else begin
            en_d        <= en;              // 1클럭 전의 en 저장
            post_en     <= en_d;            // 1클럭 뒤에 반영
        end
    end

    reg [23:0] rom [0:1023]; //1024개의 24비트 RGB 데이터 저장

    initial begin
        $readmemh("input.mem", rom);
    end

    always @(posedge clk) begin
        if(!rst_n) begin
            data_out <= 0;
        end
        else
            if (post_en) data_out <= rom[addr]; // 2클럭 지연된 en 신호 기준으로 ROM 데이터
                                                출력


    end
endmodule
```

ImageROM 모듈은 입력으로 들어온 주소 addr을 기준으로, 내부 ROM에 저장된 이미지 데이터를 읽어 출력하는 역할을 한다. 이 ROM에는 .mem 파일을 통해 초기화된 1024개의 24비트 RGB 픽셀 값이 저장되어 있으며, 각 픽셀은 8비트씩 R, G, B 세 채널로 구성된다.
입력 포트로는 클럭(clk), 비동기 리셋(rst_n), 데이터 접근을 제어하는 enable 신호(en), 그리고 읽고자 하는 데이터의 주소를 나타내는 10비트 주소 입력(addr)이 있다. 출력 포트인 data_out은 24비트 RGB 한 픽셀 데이터를 제공한다.
외부에서 주소와 enable 신호가 입력되면, 내부적으로 enable 신호는 두 클럭에 걸쳐 순차적으로 지연된다. 이는 다른 모듈과의 타이밍을 맞추기 위해 설정한 것이다. 이렇게 딜레이된 enable 신호가 활성화되는 시점에 ROM은 addr에 해당하는 주소의 데이터를 읽고, 그 결과를 data_out으로 출력한다. 최종적으로 출력된 data_out에는 요청된 주소에 해당하는 RGB 픽셀 값이 담기게 된다.

Shift_Reg_9

```
`timescale 1ns / 1ps
```

```
module Shift_Reg_9(
    input [23:0] din,
    input rst_n,
    input clk,
    input en,

 // R 채널 3x3 윈도우 출력
    output [7:0] R_window00,R_window01,R_window02,
    output [7:0] R_window10,R_window11,R_window12,
    output [7:0] R_window20,R_window21,R_window22,

// G 채널 3x3 윈도우 출력
    output [7:0] G_window00,G_window01,G_window02,
    output [7:0] G_window10,G_window11,G_window12,
    output [7:0] G_window20,G_window21,G_window22,
 // B 채널 3x3 윈도우 출력
    output [7:0] B_window00,B_window01,B_window02,
    output [7:0] B_window10,B_window11,B_window12,
    output [7:0] B_window20,B_window21,B_window22,



    output start_conv // 윈도우가 완성되었음을 알리는 신호
    );

// R/G/B 윈도우를 2차원 배열로 선언
    reg [7:0] R_window [2:0][2:0];
    reg [7:0] G_window [2:0][2:0];
    reg [7:0] B_window [2:0][2:0];
    reg [3:0] count; // 현재까지 입력된 픽셀 수를 카운트 (0~9)
    reg post_en;



    assign R_window00 = R_window[0][0]; assign R_window01 = R_window[0][1]; assign R_window02 = R_window[0][2];
    assign R_window10 = R_window[1][0]; assign R_window11 = R_window[1][1]; assign R_window12 = R_window[1][2];
    assign R_window20 = R_window[2][0]; assign R_window21 = R_window[2][1]; assign R_window22 = R_window[2][2];
```

```
    assign G_window00 = G_window[0][0]; assign G_window01 = G_window[0][1]; assign G_window02 = G_window[0][2];
    assign G_window10 = G_window[1][0]; assign G_window11 = G_window[1][1]; assign G_window12 = G_window[1][2];
    assign G_window20 = G_window[2][0]; assign G_window21 = G_window[2][1]; assign G_window22 = G_window[2][2];

    assign B_window00 = B_window[0][0]; assign B_window01 = B_window[0][1]; assign B_window02 = B_window[0][2];
    assign B_window10 = B_window[1][0]; assign B_window11 = B_window[1][1]; assign B_window12 = B_window[1][2];
    assign B_window20 = B_window[2][0]; assign B_window21 = B_window[2][1]; assign B_window22 = B_window[2][2];

    assign start_conv = (count == 4'd9);

    reg en_d1, en_d2;

    always @(posedge clk) begin
        if (!rst_n) begin
            en_d1 <= 0;
            en_d2 <= 0;
        end else begin
            en_d1 <= en;            // 1클럭 지연
            en_d2 <= en_d1;
            post_en <= en_d2;       // 2클럭 지연
        end
    end



    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            count <= 4'd0;
        end
        else begin
            if(post_en) begin
                if(count == 4'd9) count <= 4'd1;
                else count <= count+1;
```

```
            end
        end
    end

// R 채널 시프트 레지스터 로직
    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            R_window[0][0] <= 8'b0; R_window[0][1] <= 8'b0; R_window[0][2] <= 8'b0;
            R_window[1][0] <= 8'b0; R_window[1][1] <= 8'b0; R_window[1][2] <= 8'b0;
            R_window[2][0] <= 8'b0; R_window[2][1] <= 8'b0; R_window[2][2] <= 8'b0;

        end
        else begin
            if(post_en) begin
                R_window[2][2] <= din[23:16];
                R_window[1][2] <= R_window[2][2];
                R_window[0][2] <= R_window[1][2];
                R_window[2][1] <= R_window[0][2];
                R_window[1][1] <= R_window[2][1];
                R_window[0][1] <= R_window[1][1];
                R_window[2][0] <= R_window[0][1];
                R_window[1][0] <= R_window[2][0];
                R_window[0][0] <= R_window[1][0];
            end
        end
    end

// G 채널 시프트 레지스터 로직
      always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin

            G_window[0][0] <= 8'b0; G_window[0][1] <= 8'b0; G_window[0][2] <= 8'b0;
            G_window[1][0] <= 8'b0; G_window[1][1] <= 8'b0; G_window[1][2] <= 8'b0;
            G_window[2][0] <= 8'b0; G_window[2][1] <= 8'b0; G_window[2][2] <= 8'b0;



        end
        else begin
```

```
            if(post_en) begin
                G_window[2][2] <= din[15:8];
                G_window[1][2] <= G_window[2][2];
                G_window[0][2] <= G_window[1][2];
                G_window[2][1] <= G_window[0][2];
                G_window[1][1] <= G_window[2][1];
                G_window[0][1] <= G_window[1][1];
                G_window[2][0] <= G_window[0][1];
                G_window[1][0] <= G_window[2][0];
                G_window[0][0] <= G_window[1][0];
            end
        end
    end

// B 채널 시프트 레지스터 로직
        always @(posedge clk or negedge rst_n) begin
         if(!rst_n) begin

            B_window[0][0] <= 8'b0; B_window[0][1] <= 8'b0; B_window[0][2] <= 8'b0;
            B_window[1][0] <= 8'b0; B_window[1][1] <= 8'b0; B_window[1][2] <= 8'b0;
            B_window[2][0] <= 8'b0; B_window[2][1] <= 8'b0; B_window[2][2] <= 8'b0;




        end
        else begin
            if(post_en) begin
                B_window[2][2] <= din[7:0];
                B_window[1][2] <= B_window[2][2];
                B_window[0][2] <= B_window[1][2];
                B_window[2][1] <= B_window[0][2];
                B_window[1][1] <= B_window[2][1];
                B_window[0][1] <= B_window[1][1];
                B_window[2][0] <= B_window[0][1];
                B_window[1][0] <= B_window[2][0];
                B_window[0][0] <= B_window[1][0];
            end
        end
    end
```

```
endmodule
```

Convolution을 수행하기 위해서는 입력 영상에서 3×3 크기의 윈도우를 순차적으로 형성할 수 있어야 한다. 이 모듈은 해당 3×3 윈도우를 형성하기 위한 구조로, 데이터를 {px1, px4, px7}, {px2, px5, px8}, {px3, px6, px9} 형태로 배치하여 1열 → 2열 → 3열 순서로 데이터를 받아들이도록 설계되었다. 즉, px1 → px2 → px3 → ... → px9 순으로 9개의 픽셀이 순차적으로 입력될 때, 이들이 3열 3행의 윈도우를 구성하게 된다.

이 구조는 shift register 기반으로 구현되며, 매 클럭마다 새로운 RGB 픽셀이 들어오면 기존의 데이터를 한 칸씩 밀어내며 윈도우를 갱신한다.

입력 데이터는 ImageROM으로부터 24비트(RGB) 단위로 들어오며, 하나의 픽셀은 R, G, B 세 개의 8비트 채널로 구성된다. 이 데이터를 채널별로 분리하기 위해, R은 din[23:16], G는 din[15:8], B는 din[7:0]으로 추출하여 각각 R_window, G_window, B_window 배열에 저장된다. 각 채널에 대해 동일한 시프트 로직이 적용되며, 이를 통해 R, G, B 각각에 대해 독립적인 3×3 윈도우가 형성된다.

이러한 구조를 통해, 입력으로부터 9개의 픽셀이 주어졌을 때 각 채널별 3×3 윈도우를 완성할 수 있고, 이를 기반으로 convolution 연산을 수행할 수 있다. 윈도우가 완전히 채워졌다는 것은 내부 count가 9에 도달했을 때 start_conv 신호로 표시되며, 이후 conv_9 모듈이 이 신호를 받을 때 해당 윈도우를 사용해 필터 연산을 수행하게 된다.

conv_9

```
`timescale 1ns / 1ps

module conv_9(
    input clk,
    input rst_n,
    input start_conv,      // convolution 연산 시작 신호

    input [7:0] R_window00,R_window01,R_window02,
    input [7:0] R_window10,R_window11,R_window12,
    input [7:0] R_window20,R_window21,R_window22,

    input [7:0] G_window00,G_window01,G_window02,
    input [7:0] G_window10,G_window11,G_window12,
    input [7:0] G_window20,G_window21,G_window22,

    input [7:0] B_window00,B_window01,B_window02,
    input [7:0] B_window10,B_window11,B_window12,
    input [7:0] B_window20,B_window21,B_window22,
```

```
    output reg [47:0] output_word
    );
    reg signed [15:0] conv_temp_r, conv_temp_g, conv_temp_b;

// Gaussian 필터의 커널 계수 (3x3 고정)
localparam   [7:0] k00 = 1, k01 = 2, k02 = 1;
localparam    [7:0] k10 = 2, k11 = 4, k12 = 2;
localparam    [7:0] k20 = 1, k21 = 2, k22 = 1;

    // convolution 및 출력 로직
    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            output_word <= 48'd0;
            conv_temp_r <=8'd0;
        end
        else begin
            if (start_conv) begin
                 // R 채널 convolution 연산
                conv_temp_r <=
                        k00 * ({1'b0, R_window00}) + k01 * ({1'b0, R_window01}) + k02 * ({1'b0,
R_window02})
                    + k10 * ({1'b0, R_window10}) + k11 * ({1'b0, R_window11}) + k12 * ({1'b0,
R_window12})
                    + k20 * ({1'b0, R_window20}) + k21 * ({1'b0, R_window21}) + k22 * ({1'b0,
R_window22});
     // G 채널 convolution 연산
              conv_temp_g <=
                        k00 * ({1'b0, G_window00}) + k01 * ({1'b0, G_window01}) + k02 * ({1'b0,
G_window02})
                    + k10 * ({1'b0, G_window10}) + k11 * ({1'b0, G_window11}) + k12 * ({1'b0,
G_window12})
                    + k20 * ({1'b0, G_window20}) + k21 * ({1'b0, G_window21}) + k22 * ({1'b0,
G_window22});
     // B 채널 convolution 연산
    conv_temp_b <=
                        k00 * ({1'b0, B_window00}) + k01 * ({1'b0, B_window01}) + k02 * ({1'b0,
B_window02})
                    + k10 * ({1'b0, B_window10}) + k11 * ({1'b0, B_window11}) + k12 * ({1'b0,
B_window12})
```

```
                        + k20 * ({1'b0, B_window20}) + k21 * ({1'b0, B_window21}) + k22 * ({1'b0,
B_window22});

                // 각 채널의 결과를 16으로 나누어 RGB 순으로 결합하여 출력
                    output_word <= {(conv_temp_r>>4), (conv_temp_g>>4), (conv_temp_b>>4)};
                end
            end
        end

endmodule
```

conv_9 모듈은 입력으로 주어진 R, G, B 각각의 3×3 윈도우에 대해 Gaussian Blur 필터를 적용하여 convolution 연산을 수행하고, 그 결과를 하나의 48비트 출력으로 내보내는 기능을 한다. 입력 포트로는 클럭, 리셋, convolution 시작을 알리는 start_conv 신호, 그리고 각각 R, G, B 채널의 3×3 윈도우 데이터가 존재한다. 윈도우 데이터는 R_window00부터 R_window22, G_window00부터 G_window22, 그리고 B_window00부터 B_window22까지 총 27개의 8비트 값으로 구성된다. 이 값들은 앞서 shift register 모듈에서 형성된 3×3 영역을 그대로 전달받는다.
내부적으로는 각 채널에 대해 convolution 연산 결과를 저장할 임시 변수인 conv_temp_r, conv_temp_g, conv_temp_b가 정의되어 있다. Gaussian Blur 필터는 고정된 커널로, 각 위치에 대응하는 계수들은 localparam으로 정의되어 있다. 커널의 구조는 중심이 4, 인접이 2, 대각선이 1로 구성된 대표적인 Gaussian 3×3 필터이며, 전체 가중치의 합은 16이 된다.
연산은 클럭 상승 에지에서 이루어지며, start_conv 신호가 high일 때 동작한다. 이때 각 채널별 윈도우의 9개 값에 커널 계수를 곱해 모두 더한 값을 convolution 결과로 저장한다. 이 과정에서 8비트 입력값은 부호 없는 9비트 값으로 확장되어 곱셈 연산의 정확도를 높인다. convolution 결과는 16으로 나누기 위해 비트 쉬프트 연산을 통해 4비트 우측 이동시키고, 그 결과들을 R, G, B 순서로 16비트씩 이어붙여 48비트 출력 output_word에 저장한다.
최종적으로 conv_9 모듈은 start_conv가 1이 되는 순간 각 채널에 대해 convolution을 수행하고, Gaussian 필터 결과를 평균 처리한 값을 R, G, B 순으로 결합하여 출력하는 역할을 한다.

tb_top

```
`timescale 1ns / 1ps

module tb_top;

    // Inputs
    reg             clk;
    reg             rst_n;
    reg             en;
    wire [47:0]   output_word;
    wire            conv_fin;
```

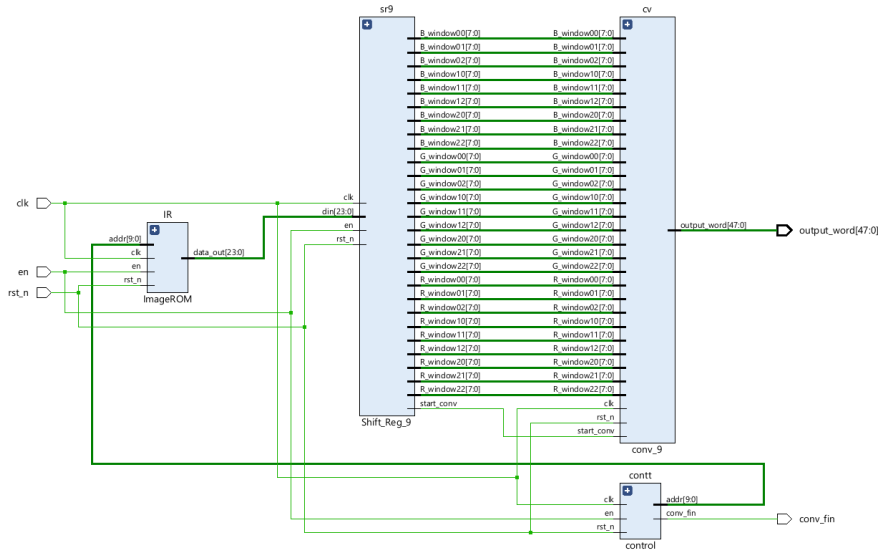
```
// ---- 추가된 신호 선언 ----
// 1. 내부 start_conv를 hierarchical 참조할 레지스터
reg              start_conv_d;

// 2. 저장용 메모리와 인덱스
localparam     MEM_DEPTH = 1025;
reg   [47:0]   out_mem [0:MEM_DEPTH-1];

// --------------------------

// Instantiate the Unit Under Test (UUT)
top uut(
      .clk              (clk),
      .rst_n            (rst_n),
      .en                (en),
      .output_word (output_word),
      .conv_fin (conv_fin)
);

// Clock generation: 50MHz
initial clk = 0;
always #10 clk = ~clk;   // 20ns period

// ----------------------------------------
// start_conv 딜레이 + 첫 클럭 스킵 + 캡처
// ----------------------------------------
reg              seen_first;   // 첫 번째 이벤트를 봤는지
integer         idx;

always @(posedge clk or negedge rst_n) begin
      if (!rst_n) begin
            start_conv_d <= 1'b0;
            seen_first     <= 1'b0;
            idx               <= 0;
      end
      else begin
            // 1clk 딜레이된 start_conv
            start_conv_d <= uut.start_conv;
```

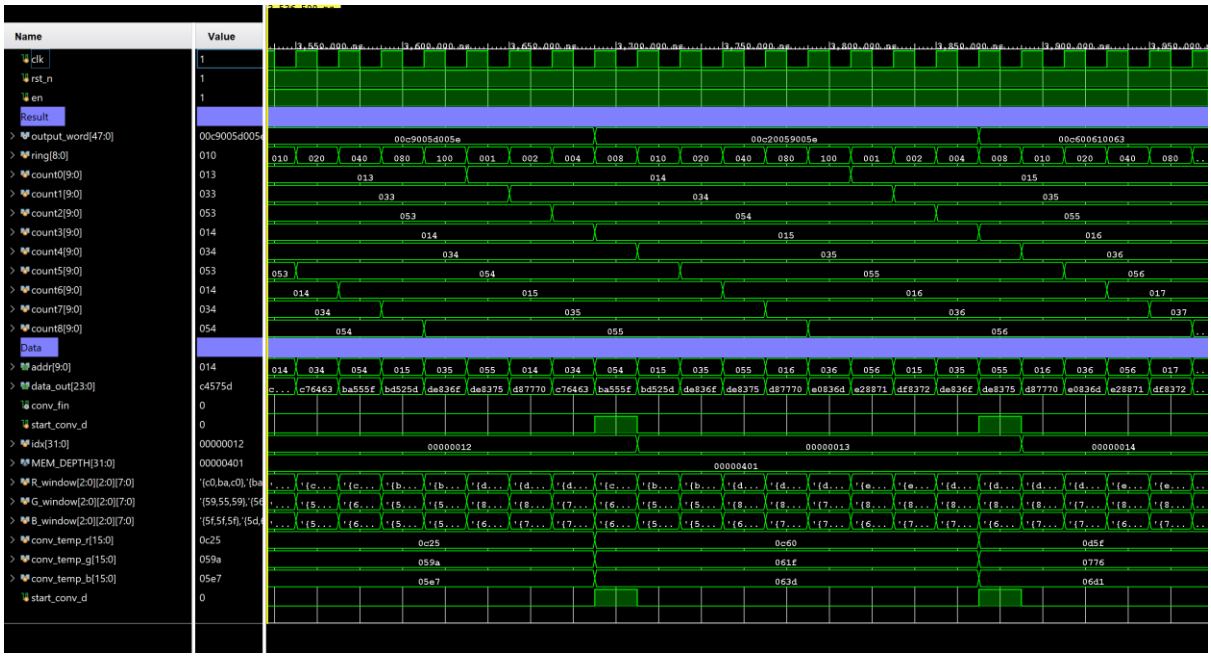
```
            // 딜레이된 신호가 올라올 때만 처리
            if (start_conv_d) begin
                if (!seen_first) begin
                    // 첫 번째 start_conv_d: 플래그만 세우고 캡처 안 함
                    seen_first <= 1'b1;
                end
                else begin
                    // 그 다음부터는 output_word 캡처
                    out_mem[idx] = output_word;
                    idx = idx + 1;
                end
            end
        end
    end

    // ---- fin 감지시 덤프 & 종료 로직 추가 ----
    always @(posedge clk) begin
        if (conv_fin) begin
            $writememh("output_dump.mem", out_mem, 0, idx-1);
            $display("[%0t] fin detected! Dumped %0d words to output_dump.mem", $time, idx);
            $finish;
        end
    end
    // ----------------------------------------

    initial begin
        // Initialize inputs
        rst_n = 1;
        en    = 0;

        // Reset sequence
        #5   rst_n = 0;
        #2   rst_n = 1;

        // Enable convolution
        #23 en = 1;

        // (기존의 time-based 종료/덤프는 fin이 없으면 동작)
```

```
        #200_000;
        $writememh("output_dump.mem", out_mem, 0, idx-1);
        $display("Timeout reached. Dumped %0d words to output_dump.mem", idx);

        $display("Simulation finished.");
        $finish;
    end

endmodule
```

Sobel filter때와 동일하고 다만, 48비트 출력을 위해 output_word와 out_mem의 크기만 수정하였다.

3. 시뮬레이션 분석





-control

Project1과 동일하게 동작

-ImageROM

Project1과 마찬가지로 addr에 따른 data_out 매칭 확인

-Shift_Register_9

Project1과 동일하지만 윈도우가 3개(R,G,B,)

-conv_9

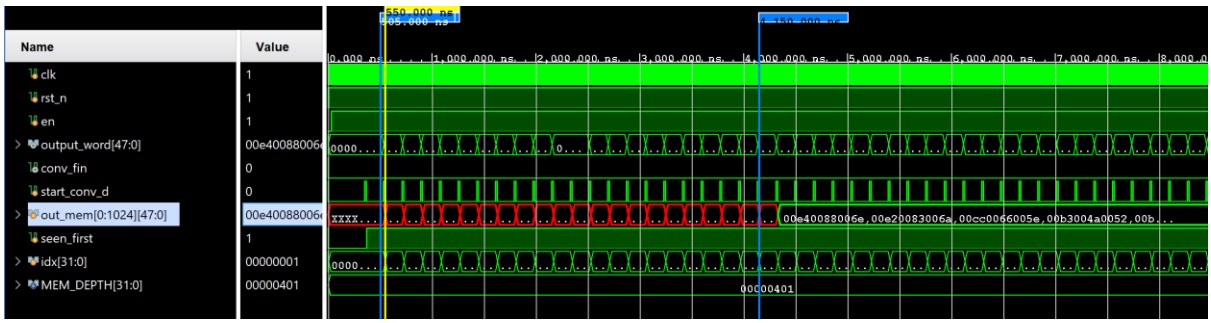
Project1과 동일하게 start_conv 신호에 맞추어 conv_temp에 각 채널의 convolution값을 계산해 저장하고 output_word에 16을 나누어 concat하여 출력. 타이밍도 proj1과 동일. → 정상작동.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.687 ns	Worst Hold Slack (WHS):	0.080 ns	Worst Pulse Width Slack (WPWS):	5.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	616	Total Number of Endpoints:	616	Total Number of Endpoints:	323

All user specified timing constraints are met.

Slack이 음수가 되는 것을 막기 위해 xdc에 넣은 clock값의 period를 10에서 11로 수정하여 clk을 낮추어 implementation을 진행하였다.



Project 1과 같은 방법으로

Start=550ns

End=4150ns

총 걸린 시간은 3600ns =3.6us

4. 합성 결과 분석

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	301	0	0	47200	0.64
LUT as Logic	301	0	0	47200	0.64
LUT as Memory	0	0	0	19000	0.00
Slice Registers	420	0	0	94400	0.44
Register as Flip Flop	420	0	0	94400	0.44
Register as Latch	0	0	0	94400	0.00
F7 Muxes	0	0	0	31700	0.00
F8 Muxes	0	0	0	15850	0.00

전체 LUT는 47,200개 중 301개(0.64%), 레지스터는 94,400개 중 420개(0.44%)가 사용되었으며, LUT는 모두 조합 논리로만 활용되었고 LUT 기반 메모리는 사용되지 않았다. 모든 레지스터는 Flip-Flop으로만 사용되었으며, Latch는 0으로 나타나 latch가 불필요하게 형성되지 않았음을 확인할 수 있다. 또한 F7, F8 Mux 자원도 전혀 사용되지 않아 비교적 단순하고 효율적인 로직 구조로 구현되었음을 보여준다.

2. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	1	0	0	105	0.95
RAMB36/FIFO*	1	0	0	105	0.95
RAMB36E1 only	1				
RAMB18	0	0	0	210	0.00

Block RAM RAMB36 블록 하나만 사용되었다. 이는 이미지 데이터를 저장한 ImageROM이 Block RAM에 매핑되었음을 의미한다. 전체 메모리 자원 중 매우 적은 비율만 사용되었다.

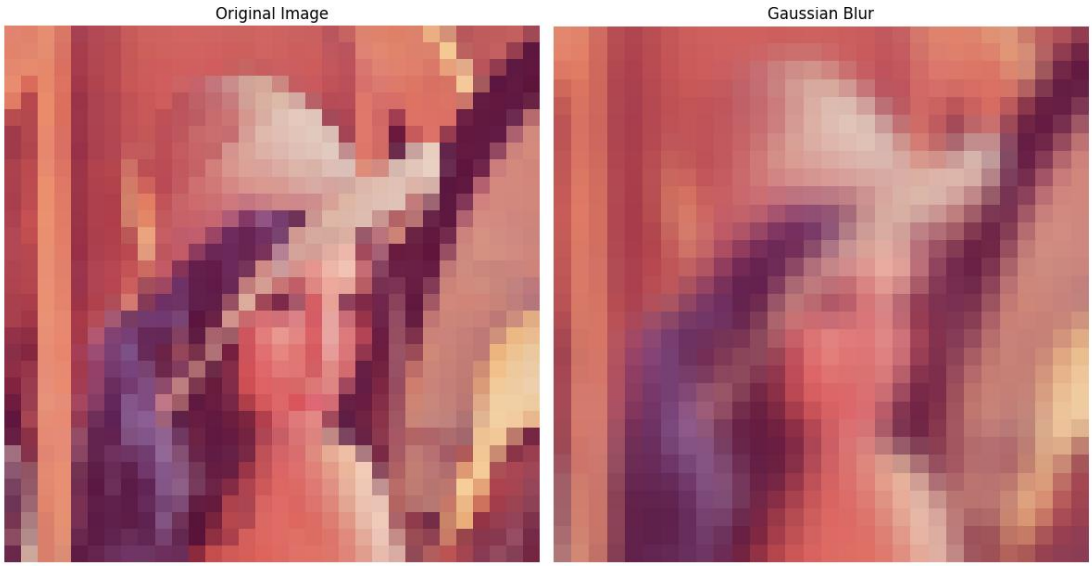
3. DSP

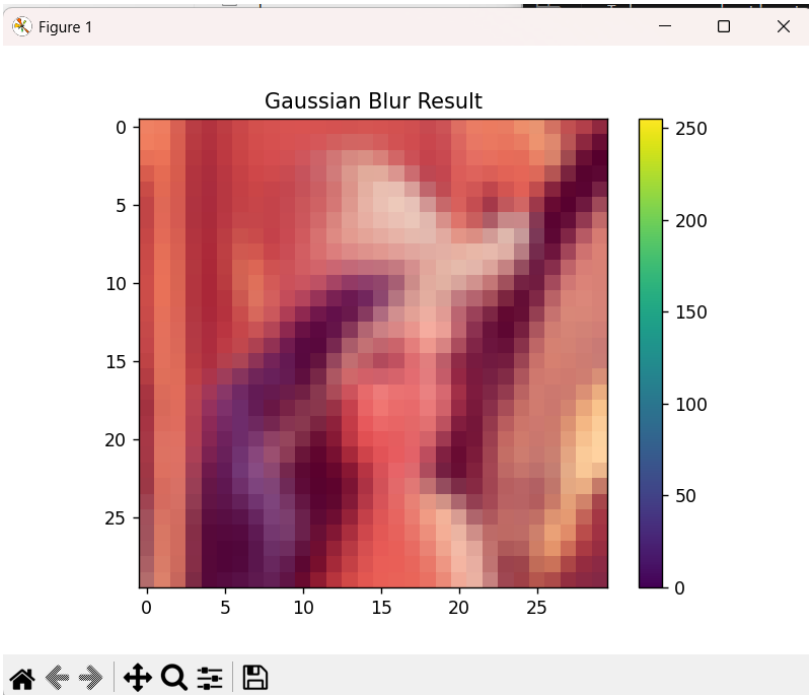
Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	0	0	0	180	0.00

DSP 블록은 사용되지 않았다. 즉, 모든 convolution 연산은 LUT 기반의 연산으로 처리되었음을

의미한다. 이는 Gaussian 필터 계수가 고정된 소수(1,2,4 등)로 단순하여 곱셈이 복잡하지 않고 DSP 없이도 가능하기 때문이다.

5. 결과 이미지





[팀원 역할]

최훈서:
+Sobel Filter
+모듈구성 아이디어
+디버깅
+타이밍 제어

서석현:
+Gaussian Blur
+모듈구성 아이디어
+결과분석