임베디드 시스템 종합설계
프로젝트 보고서
주제: Verilog를 통한 Image Processing System 설계 및 검증

학과: 전자전기공학부
학번: B615088
이름: 박현석
담당 교수님: 김영민

제출일자: 2021.12.19.

1. 프로젝트 주제 및 목적

a. 프로젝트 주제

Verilog를 통한 Image Processing System 설계 및 검증

b. 프로젝트 목적

Verilog를 이용하여 수업시간에 배운 내용을 토대로 이미지를 처리하는 시스템을 설계한다. 이미지를 처리하기 위해 사용되는 Convolution을 이용한 각종 Filter에 대한 개념을 정확히 이해하고, 이에 필요한 Line Memory와 Memory 입출력을 담당하는 Module을 설계하는 것이 본 프로젝트의 목적이다. 또한 대부분의 Hardware는 clock에 동기화되어 작동하는 경우가 많지만, 앞서 수업의 Lab에서 MUX2와 ALU와 같은 설계를 할 때 clock을 고려하지 않았기 때문에 clock에 동기화된 시스템을 설계하고자 한다.

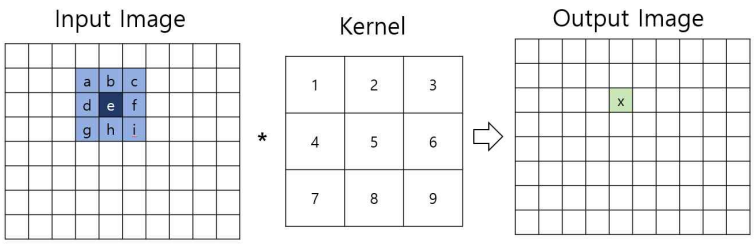
c. 주제 선정 과정

평소에 배워보고 싶고 구현해보고 싶었던 것은 암호화 알고리즘이기 때문에 프로젝트 제안서에서는 AES 암호화 및 복호화와 관한 프로젝트를 기획하였다. 하지만, AES 암호화를 이해하는데 필수적인 Galois Field와 같은 수학적 지식이 부족하기 때문에 학기중에 이에 대한 지식을 습득하고 프로젝트를 진행하기에는 시간이 부족하다고 생각하였다. 알고리즘에 대한 이해를 하지 않고 인터넷에서 쉽게 구할 수 있는 pseudo code를 참고하여 프로젝트를 진행하는 것은 의미가 없을 것이라고 판단하였기 때문에, 알고리즘을 완벽히 이해하고 설계할 수 있는 학습한 적이 있던 전공과 관련된 프로젝트 주제를 찾게되었다. 또한, 실제 Embedded System에서 사용되고 적용할 수 있는 주제를 선택하는 것에도 우선순위를 두었다.

이미지 처리 시스템은 Display를 사용하고 있는 많은 Embedded System에서 사용된다. 주변에서 쉽게 접할 수 있는 대부분의 Monitor, TV와 같은 Device에도 Display의 밝기와 선명도를 조절할 수 있는 기능이 탑재되어있는 경우가 많다. 이미지 처리 시스템은 Filter 적용 시에 Convolution을 사용한다. Convolution은 신호처리에 대한 과목 및 선형대수학에서 학습하였으므로 이미지 처리 관련 Filter의 원리에 대한 이해를 충분히 할 수 있을 것이라고 생각하였기 때문에 이미지 처리 시스템을 주제로 정하였다. 또한, Convolution을 하기 위해서는 Memory에 대한 설계도 필요하므로 프로젝트 주제로 적합하다고 생각하였다.

2. 원리 및 배경지식

a. Filter

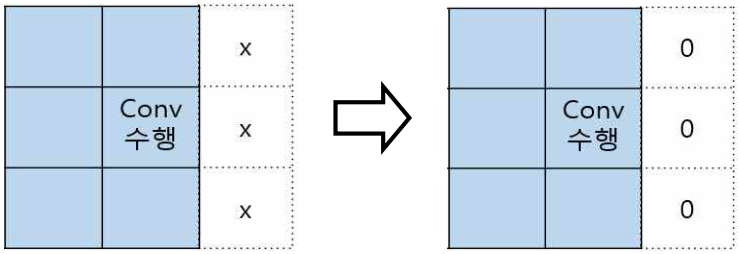


[그림 2-1] 이미지에 대한 Convolution 적용

$$x = \left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right)[2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9)$$

[수식 2-1] Convolution [그림 2-1]에 대한 계산 결과

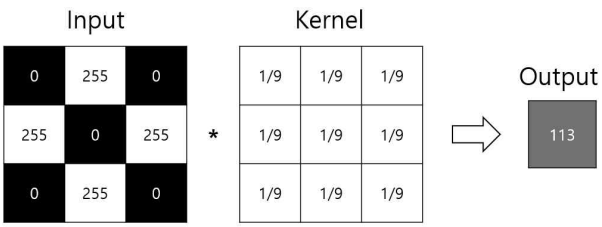
입력 이미지를 스캔하면서 현재 위치와 주변 픽셀과 Kernel에 대해 Convolution을 하여 결과 이미지의 현재 위치값으로 결정한다. Kernel 행렬의 값에 따라 각 픽셀에 대한 가중치 값이 결정되므로, Kernel의 값에 따라 여러 가지 형태로 이미지를 처리할 수 있다.



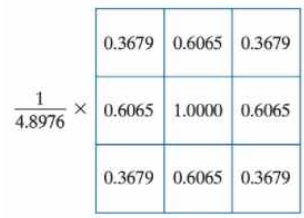
[그림 2-2] Convolution 문제점 및 Zero-padding을 통한 해결

Convolution을 진행할 때, Input Image의 모서리 부분 픽셀은 인접 픽셀의 정보가 Unknown 값이므로, 이 부분은 Convolution을 수행할 수 없기때문에 출력 이미지의 크기가 입력 이미지에 비해 작아지는 현상이 발생한다. 하지만, 입력 이미지의 크기와 출력 이미지의 크기가 동일해야 하므로 [그림 2-2]와 같이 Convolution을 수행할 수 없는 Unknown 영역을 0으로 확장하는 Zero-padding 방법을 사용하여 이 문제를 해결하였다. [그림 2-2]의 좌측은 Zero-padding을 적용하지 않았기 때문에 Convoluion의 결과 또한 Unknown이 되어 출력 이미지에 문제가 생기지만, Zero-padding을 적용하면 문제 없이 Convolution을 수행할 수 있다.

b. Box Blur: Smoothing Linear Filter



[그림 2-3] Box Blur Filter 예시

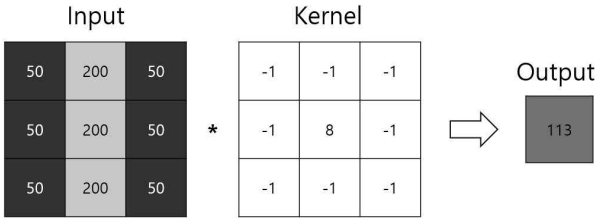


[그림 2-4] Gaussian Kernel

Smoothing Linear Filter는 입력 이미지에서 Kernel 영역에 포함되는 픽셀들의 평균이 출력 이미지가 된다. Kernel의 모든 원소가 동일한 가중치를 가지고 있고, 모든 원소의 합이 1이 되기 때문에 출력값이 입력에 대한 평균이 되는 것을 쉽게 알 수 있다. 출력이 입력에 대한 평균값이 되기 때문에 이미지의 Noise를 감소시켜 주는 Filter임을 알 수 있다.

Smoothing Linear Filter의 종류에는 Box Filter와 Gaussian Filter가 존재한다. [그림 2-4]과 같이 Gaussian Filter는 중앙을 기준으로 한 픽셀의 거리를 기준으로 가중치를 계산하므로 실수값에 대한 연산을 수행하게 되어 연산에 대한 연산에 대한 비용이 큰 것을 알 수 있다. 반면에, Box Filter는 모든 입력 픽셀에 대한 합을 구한 뒤, 나눗셈을 한번 진행하는 것으로 간단하게 연산이 가능하므로 본 프로젝트에서는 상대적으로 비용이 적은 Box Filter로 구현을 하였다.

c. Edge Detection: Laplacian Filter



[그림 2-5] Laplacian 필터 예시

$$\nabla^2 f = \frac{\partial^2 f}{dx^2} + \frac{\partial^2 f}{\partial y^2}$$

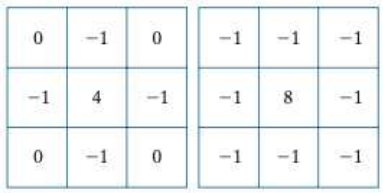
[수식 2-2] Laplacian 수식

$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y)$$

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) - 2f(x, y)$$

$$\nabla^2 f = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

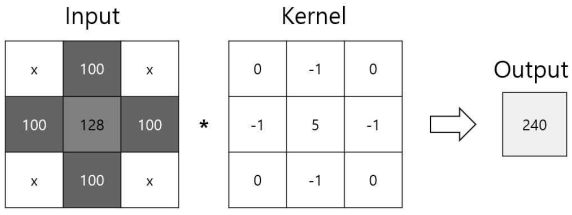
[수식 2-3] 이산 데이터에 대한 Laplacian



[그림 2-6] Laplacian Kernel

Laplacian Filter는 입력 이미지에 2차 미분을 적용한 결과가 출력 이미지가 된다. [수식 2-2]는 Laplacian 기본 수식이다. 이미지는 2차원의 이산 데이터이므로 [수식 2-3]과 같이 표현될 수 있다. 이미지에 Laplacian을 적용하여 급격하게 값이 변화하는 지점을 검출해 낼 수 있으므로, 이미지에서 Edge를 검출해낼 수 있다. 이를 Kernel로 표현하면 [그림 2-6]와 같은 결과를 얻을 수 있다. [그림 2-6]의 좌측 Kernel은 대각선 방향을 고려하지 않은 것이고, 우측 Kernel은 대각선 방향까지 고려한 것이다. 우측 Kernel을 적용한 것이 더욱 선명하게 Edge를 검출할 수 있기 때문에 본 프로젝트에서는 우측 Kernel을 사용하였다.

d. Sharpen: Sharpening Spatial Filter



[그림 2-7] Sharpen Filter 예시

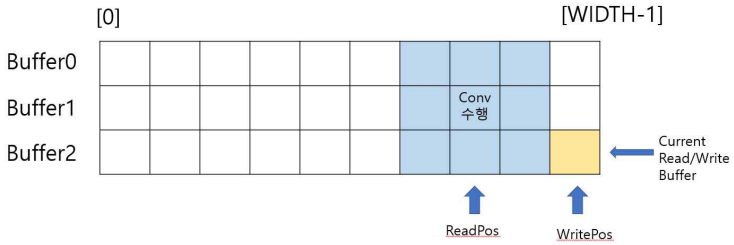
$$g(x, y) = f(x, y) + \nabla^2 f(x, y)$$

[수식 2-4] Sharpen Filter 수식

이미지를 선명하게 처리하는 Sharpen Filter는 앞서 다룬 Laplacian Filter에서 검출한 Edge 값을 원본 이미지에 더해서 원본 이미지의 Edge를 더욱 선명하게 하는 방식을 적용하였다. 앞서 보인 [수식 2-3]에 현재 Image Pixel인 $f(x,y)$를 더하는 것으로 간단하게 [수식 2-4]와 같은 Sharpen Filter의 수식을 구할 수 있다. Sharpen Filter를 적용한 이미지가 $g(x,y)$가 된다.

Base가 되는 Kernel로는 [그림 2-6]의 우측 Kernel을 적용하면 Edge가 너무 부각되어 보이기 때문에 출력 이미지가 부자연스러운 형태로 나타나서 좌측 Kernel을 적용하였다.

e. Memory for Convolution

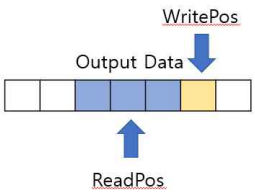


[그림 2-8] Pixel Data를 저장하기 위한 Buffer

이미지의 입출력은 Pixel 단위로 이루어진다. 하지만, Convolution을 수행하려면 현재 Pixel Data 뿐만 아니라 최소 3x3의 인접 Pixel의 Data도 필요하기 때문에 메모리를 설계하는 것이 필수적이다. 이미지의 가로 크기를 WIDTH라고 하다면, Grayscale의 1픽셀 데이터, 즉 1바이트를 저장할 수 있는 WIDTH 만큼의 길이를 가지는 Buffer로 쓰이는 Memory를 설계하였다. 해당 Buffer를 3개 사용하면 총 3줄에 해당하는 Pixel Data를 저장할 수 있으므로 Convolution을 수행하는 데 문제가 발생하지 않는 것을 알 수 있다.

3. RTL Design

a. Buffer 설계



[그림 3-1] One Line Buffer

```
if(writeSignal) begin
        mem[writePos] <= inputData; //store data
        if(writePos == `WIDTH-1)
                writePos <=0;
        else
                writePos <= writePos +1; //increase pos
end
if(readSignal) begin
        if(readPos == `WIDTH-1)
                readPos <=0;
        else
                readPos <= readPos +1; //increase pos
end
```

[코드 3-1] Buffer Module 일부; always @(posedge clk)

앞서 설명한 것과 같이 Convolution을 수행하기 위해서는 3줄의 Buffer가 필요하다. Buffer를 3개 만들어야 하므로 Buffer 모듈을 설계하였다. Buffer는 수업 강의자료의 Memory model LAB을 기반으로 설계하였다. Buffer의 총 길이는 이미지의 가로 한줄을 담아야 하므로, Compiler Directive인 define을 이용하여 이미지의 가로 픽셀 수 만큼의 값을 WIDTH로 정의하였다. Buffer의 어느 index를 읽고 쓸지 결정하는 readPos와 writePos는 0~WIDTH-1의 값을 표현할 수 있어야 하므로 $\lceil \log_2 WIDTH \rceil$ 만큼의 크기를 가지도록 설정한다. 만약, WIDTH가 640이면, $\lceil \log_2 WIDTH \rceil = \lceil 9.322 \rceil = 10$와 같이 readPos와 writePos의 크기를 정할 수 있다.

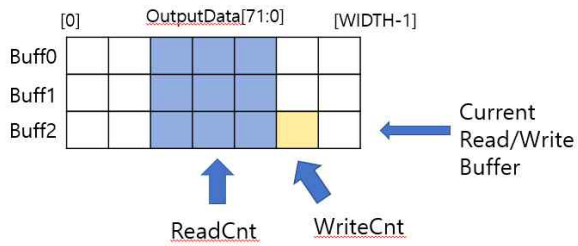
[코드 3-1]은 Buffer의 데이터 입력 및 read/write에 대한 index인 readPos와 writePos의 값을 조절하는 역할을 수행한다. Input Data가 있다는 Signal인 writeSignal의 값이 1이면 현재의 writePos에 inputData의 값을 assign하고, writePos를 1 증가시킨다. 이 때, writePos의 값이 WIDTH-1이면 1을 증가할 경우 Buffer의 길이를 벗어나기 때문에 0으로 assign한다. readPos의 경우에도 같은 과정을 반복한다.

```
always @(*) begin
        if(readPos ==0)
                outputData <= {8'b0,mem[readPos],mem[readPos+1]};
        else if(readPos == `WIDTH-1)
                outputData <= {mem[readPos-1],mem[readPos],8'b0};
        else
                outputData <= {mem[readPos-1],mem[readPos],mem[readPos+1]};
end
```

[코드 3-2] Buffer Module 일부; always @(*)

[코드 3-2]는 출력 Data를 assign하는 과정이다. 해당 코드는 readPos 값의 변화에 대한 timing 문제로 always @(*)를 이용해 실시간으로 outputData를 assign하게 하였다. readPos가 0과 WIDTH-1인 경우에는 [그림 2-2]에서 확인한 것처럼 인접 픽셀 값이 Unknown이 되는 문제가 발생하기 때문에 이 부분을 예외처리하여 Zero-padding을 적용하였다.

b. Memory Control Unit 설계



[그림 3-2] Memory Control Unit

Convolution을 하기 위해 3개의 Buffer를 가지고 3x3의 9개의 Pixel Data를 출력하는 것이 Memory Control Module이다. Buffer 3개를 순차적으로 Read/Write 해야하므로 Read/Write Count가 WIDTH-1이 되면 Current Read/Write Buffer라는 Reg의 값을 변화시켜 다음 Buffer를 가리키게 하고, Read/Write의 값은 0이 되어 다음 Buffer의 첫 번째 Data부터 Access 한다. Read/Write Count를 증가시키고, Current Read/Write Buffer의 값을 바꾸는 것은 앞서 [코드 3-1]에서 본 Buffer의 Read/Write Posistion을 증가시키는 방법과 동일하므로 설명은 생략한다.

```
always @(*) begin
        case(currentReadBuffer)
        0: begin
                outputData <= {buffer1Data, buffer2Data, buffer0Data};
        end
        1: begin
                outputData <= {buffer2Data, buffer0Data, buffer1Data};
        end
        2: begin
                outputData <= {buffer0Data, buffer1Data, buffer2Data};
        end
        endcase
end
```

[코드 3-3] Memory Control Module: Output Data Assign; always @(*)

CurrentReadBuffer	0	1	2
현재 Line	0	1	2
이전 Line	2	0	1
전전 Line	1	2	0

[표 3-1] Buffer에서 Data를 읽는 순서

[표 3-1]과 같이 이전 Line의 ReadPos의 Data를 기준으로 3x3의 Pixel Data를 읽어와서 Convolution을 수행하는 Module에 전달한다. 이를 Code로 표현한 것이 [코드 3-3]과 같다. bufferData는 현재 readPos에 대한 좌우 인접 픽셀의 값을 포함한 3byte의 output이다. 이러한 3줄의 Buffer의 Output을 입력 순서의 역순에 맞게 concatenation operator를 사용해 컨볼루션에 사용할 9바이트의 Data를 outputData로 배정하였다.

이를 위해서는 3줄 이상의 Data가 Memory에 있어야 하므로, Memory에 3줄 이상 Data를 쓰게 되면 Read를 시작한다. 각 Buffer 당 ReadPos-1, ReadPos, ReadPos+1의 Pixel Data를 Output으로 전달하기 때문에, 이는 총 9개의 Pixel Data이기 때문에 Convolution을 수행하는 Filter Module의 Input으로 전달하게 된다.

c. Filter 설계

```
initial begin
        kernel[0][0] =0; kernel[1][0] =1; kernel[2][0] =0; kernel[3][0] =-1;
        kernel[0][1] =0; kernel[1][1] =1; kernel[2][1] =-1; kernel[3][1] =-1;
        kernel[0][2] =0; kernel[1][2] =1; kernel[2][2] =0; kernel[3][2] =-1;
        kernel[0][3] =0; kernel[1][3] =1; kernel[2][3] =-1; kernel[3][3] =-1;
        kernel[0][4] =1; kernel[1][4] =1; kernel[2][4] =5; kernel[3][4] =8;
        kernel[0][5] =0; kernel[1][5] =1; kernel[2][5] =-1; kernel[3][5] =-1;
        kernel[0][6] =0; kernel[1][6] =1; kernel[2][6] =0; kernel[3][6] =-1;
        kernel[0][7] =0; kernel[1][7] =1; kernel[2][7] =-1; kernel[3][7] =-1;
        kernel[0][8] =0; kernel[1][8] =1; kernel[2][8] =0; kernel[3][8] =-1;
end
```

[코드 3-4] Filter Module: Kernel Data 초기화

parameter FILTER_MODE	Kernel
0	None {0, 0, 0, 0, -1, 0, 0, 0, 0}
1	Blur $\frac{1}{9}\times${1, 1, 1, 1, 1, 1, 1, 1, 1}
2	Sharpen {0, -1, 0, -1, 5, -1, 0, -1, 0}
3	Edge Detection {-1, -1, -1, -1, 8, -1, -1, -1, -1}

[표 3-2] parameter FILTER_MODE에 따른 적용되는 Kernel 값

Filter Module은 parameter FILTER_MODE를 통해 어떠한 Kernel을 통해 Convolution을 수행할지 결정한다. 각 Kernel의 값은 앞서 이론에서 설명한 Kernel의 값과 동일하다. Blur의 경우에는 Kernel의 가중치가 $\frac{1}{9}$가 되어야 하지만, Verilog에서 나눗셈 연산을 매번 수행하면 비용이 크므로 일단 Kernel의 모든 가중치를 1로 두고, Convolution 수행 결과를 Output Data에 배정할 때 9로 나누어준다.

```
//convolution
always @(*) begin
        sum =0;
        for(i=0;i<9;i=i+1) begin
                sum = sum + kernel[FILTER_MODE][i]*$signed({1'b0,inputData[i*8+:8]});
        end
        if(FILTER_MODE ==1) //for blur
                processedData <= sum /9 + BRIGHTNESS;
        else
                processedData <= sum + BRIGHTNESS;
        processedValid <= inputValid;
end

//assign output
always @(posedge clk)
begin
        if(processedData >255)
                outputData <=255;
        else if(processedData <0)
                outputData <=0;
        else
                outputData <= processedData;
        outputValid <= processedValid;
end
```

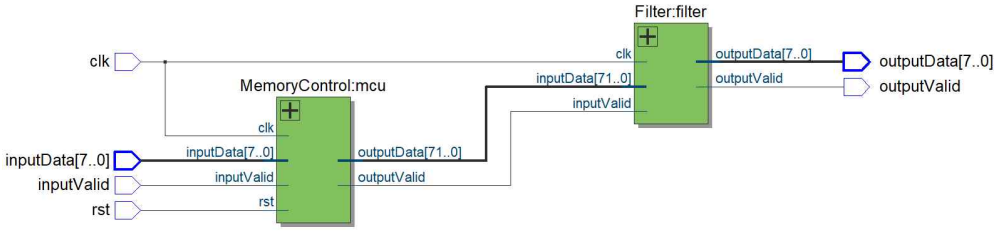
[코드 3-5] Filter Module: Convolution 수행 및 연산 결과 assign

각 Kernel의 값과 Input Data에 대한 곱에 대한 합이 Convolution의 결과가 된다. 9바이트의 Input Data에서 한 Pixel에 대하여 연산을 수행하야 하므로 '+:'라는 indexed part-select operator를 사용하였다. 또한, Kernel은 음수 값을 가지므로 signed 이지만, Input Data는 Pixel Data이므로 unsigned이다. 따라서 Convolution을 수행하기 위해서는 Input Data를 signed로 casting 해주는 과정이 필요하다. 이렇게 Convolution을 수행이 완료되면 앞서 설명한 것과 같이 Blur Filter의 경우에는 연산 결과를 9로 나누어준다. parameter는 FILTER_MODE 이외에도 Convolution 결과의 밝기를 조절할 수 있는 BRIGHTNESS라는 parameter가 존재하기 때문에 이 값을 더해준다.

clk이 들어오면 Convolution 결과를 Output Data에 assign 해주는데, processed Data는 signed이고, Output Data는 unsigned 이므로 만약 Convolution 결과가 음수가 나오면 Output Data에 0을 assign 해주고, Output Data의 최대값인 255(0xFF)를 초과하면 255를

assign 해준다. 이러한 과정을 거쳐 Convolution 수행이 완료된다.

d. RTL Viewer



[그림 3-3] RTL Viewer 결과

설계한 Module을 RTL Viewer로 확인한 결과는 [그림 3-3]과 같다. 1바이트의 Input Data가 Memory Control Unit에 Write되고, Convolution을 위한 9바이트의 Data가 Filter를 거쳐 Output Data로 출력 되는 것을 확인할 수 있다.

4. Testbench 작성 및 시뮬레이션

a. Grayscale Image

```
ImageProcessing tb_IP(clk, rst, inputDataValid, inputData, outputDataValid, outputData);
...
//input
initial begin
        rst =1;
        inputDataValid =0;
        @(posedge clk);
        rst =0;
        @(posedge clk);
        file_in = $fopen(INPUT_FILE, "rb"); //read binary
        file_out = $fopen(OUTPUT_FILE, "wb"); //write binary
        for(i=0; i<`WIDTH*`HEIGHT; i=i+1) begin
                @(posedge clk);
                $fscanf(file_in, "%c", inputData); //read 1 byte pixel data
                inputDataValid<=1;
        end
        @(posedge clk);
        inputDataValid<=0;
        $fclose(file_in);
end
//output
always @(posedge clk) begin
        if(outputDataValid) begin
                $fwrite(file_out,"%c",outputData); //wirte 1 byte pixel data
                writeSize = writeSize +1;
        end
        //done
        if(writeSize == `WIDTH * `HEIGHT) begin
                $fclose(file_out);
                $stop; //stop tb
        end
end
```

[코드 4-1] Grayscale Image Testbench 일부

먼저, 한 Pixel에 1바이트의 Grayscale의 Pixel Data를 가지는 Image를 fopen을 사용해 불러

오고, fread를 통해 1바이트씩 읽어와서 RTL Module에 전달한다. 이 과정을 이미지의 가로와 세로의 곱만큼 반복해 모든 픽셀을 전송하고, 만약 RTL Module에서 Output Signal인 outputDataValid의 값이 참이면 fwrite를 이용해 Output Data를 파일에 출력한다. 모든 Pixel Data가 출력되면 Testbench를 종료한다.

	
Input Image	Blur Filter
	
Sharpen Filter	Edge Detection

[표 4-1] GrayScale Image Testbench 시뮬레이션 결과

[표 4-1]과 같이 입력 이미지에 대한 출력이 설계한 대로 잘 이루어지는 것을 확인할 수 있다. Blur의 경우에는 입력 이미지와 큰 차이가 없어 보이지만, 이는 보고서에 결과 사진을 넣으면서 크기를 축소하였기 때문이다.

b. Application: RGB Image

```
ImageProcessing tb_R(clk, rst, inputDataValid, inputR, outputDataValid, outputR);
ImageProcessing tb_G(clk, rst, inputDataValid, inputG, outputDataValid, outputG);
ImageProcessing tb_B(clk, rst, inputDataValid, inputB, outputDataValid, outputB);

...

//input
for(i=0; i<`WIDTH*`HEIGHT; i=i+1) begin
        @(posedge clk);
        $fscanf(file_in, "%c", inputR);
        $fscanf(file_in, "%c", inputG);
        $fscanf(file_in, "%c", inputB);
        inputDataValid<=1;
end

...
```

```
//output
if(outputDataValid) begin
        $fwrite(file_out,"%c",outputR); //wirte 1 byte pixel data
        $fwrite(file_out,"%c",outputG);
        $fwrite(file_out,"%c",outputB);
        writeSize = writeSize +1;
end
```

[코드 4-2] RGB Image Testbench 일부

	
Input Image	Sharpen Filter

[표 4-2] RGB Image Testbench 시뮬레이션 결과

Grayscale의 Testbench를 약간 수정하는 것으로 RGB Image에도 Filter 적용을 할 수 있다. 설계한 Module을 R, G, B Pixel 각각에 대해 적용하는 것으로 Filter가 적용된 각각의 R, G, B Pixel Data를 얻을 수 있다. [표 4-2]를 보면 R, G, B 각각의 Layer에 대해 Sharpen Filter가 적용되어 선명해진 결과를 확인할 수 있다.

5. 고찰

a. Hardware Description Language vs. High Level Language

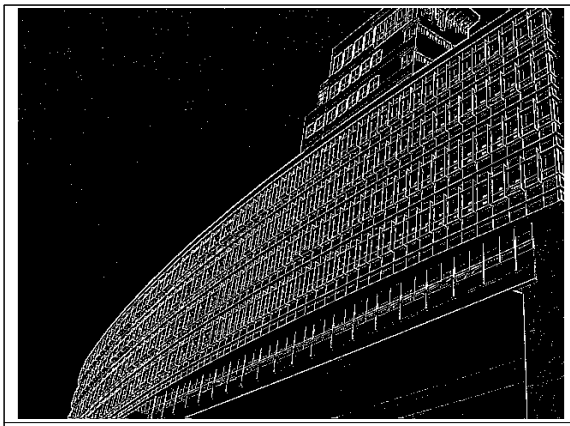
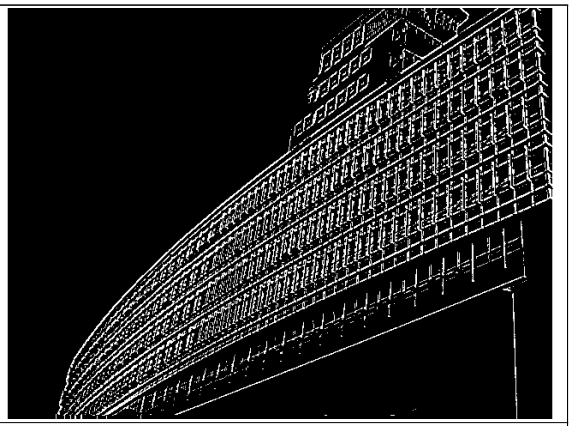
이번 수업을 통해 처음으로 HDL에 대해 배우고, 수업 때 배운 Verilog HDL을 이용하여 프로젝트를 진행하게 되었다. 여태까지 C와 Java와 같은 High Level Language로 프로젝트를 진행한 경험은 있었지만, HDL이라고 여태까지 High Level Language로 해오던 프로젝트와 크게 다르지 않을 것이라 생각했다.

하지만 HDL을 이용해 시스템을 설계할 때는 Software 관점에서 보면 전혀 상관없는 것들을 신경써야만 했다. Software에서는 어떤 변수가 어떤 배열에 접근하는지 신경쓰지 않아도 되지만, HDL에서는 어떠한 레지스터가 메모리에 접근하면 Wire가 생기고, 메모리의 값을 레지스터에 배정하려고 하면 MUX와 F/F, Latch 등이 발생할 수 있게 된다. 디지털 시스템을 설계할 때는 메모리의 설계를 효율적으로 하여 불필요한 F/F나 Latch가 발생하지 않게 하는 것이 중요하다고 생각했다.

실제로 이번 프로젝트를 진행할 때, Buffer를 사용하지 않고 2차원 배열을 이용하여 메모리를 구현했었는데, 합성시에 Latch와 MUX와 같은 소자들이 메모리 Module 합성 시에 대량 발생하여 자원이 부족하여 합성에 오류가 생겼다. Buffer를 사용하여 Module을 세분화하면 Module 사이의 Input과 Output 사이에 Wire와 Latch, MUX가 생기기 때문에 사용하는 자원이 크게 줄어들기 때문에 해당 문제를 해결할 수 있었다. 이번 프로젝트를 통해 디지털 시스템 설계에서 알고리즘을 구현할 때, 디지털 시스템 관점으로 사고하고 설계하는 법을 배울 수 있었다고 생각

한다.

b. Laplacian Edge Detection의 문제점

	
2진화 Edge Detection	Blur 처리후 2진화 Edge Detection

[표 5-1] Laplacian Filter의 Noise 문제 해결

Edge Detection의 경우에는 Sobel Edge Detection을 사용하는 경우가 많지만, Sobel Edge Detection은 1차 미분을 사용해 Input Image의 x, y축 각각에 대해 Convolution을 진행하므로 channel이 두 개 필요하기 때문에 1개의 channel으로 사용이 가능한 Laplacian Kernel을 사용하였다. 하지만, 이러한 Laplacian Filter는 2차 미분을 사용하기 때문에 Image의 Noise에 매우 취약하다는 단점이 있다. 이러한 문제의 해결 방법으로는 1차적으로 Blur Filter를 적용해 Noise를 감소시키고, 2차적으로 Laplacian Filter를 적용하면 Noise 문제를 해결할 수 있다.

[표 4-1]의 Edge Detection은 이진화 되지 않은 결과이고, 여기에 Threshold를 적용하여 Pixel 값이 40보다 큰 경우에는 값을 255로 하고, 작은 경우에는 0으로 해 이진화를 한 결과과 [표 5-1]의 좌측 사진과 같다. 보이는 것과 같이 Noise가 많이 발생한 것을 확인할 수 있다. 하지만, 먼저 Blur 처리를 하고 Laplacian Filter를 적용하면 [표 5-1]의 우측 사진과 같이 Noise가 제거된 결과를 얻을 수 있다. 하지만 Filter 적용을 두 번 하게되므로 이미지 처리에 대한 Latancy가 증가할 것이다.

6. 참고문헌

[1] Phillip A. Mlsna, Jeffrey J. Rodríguez, "Gradient and Laplacian Edge Detection", The Essential Guide to Image Processing, pp. 495-524, 2009.

[2] Rafael C. Gonzalez, Digital Image Processing 4th edition, Pearson, pp. 239-259, 2018.

[2] Digital System and Verilog HDL for AI SoC, !DEA LAB

[3] Overview of Verilog -Sequential Circuit Modeling-, !DEA LAB