고급 Project 최종보고서

HW based convolution 연산 및 동영상 filter 설계

3 조

20202962 김무성

20202975 송승규

20203023 우상욱

2025. 12. 14

숭실대학교
Soongsil University

목차

1. 서론.....3
1.1 개요.....3
1.2 팀원 별 역할 분배.....3
1.3 세부 설계 계획 및 진행 내용.....3
2. 본론.....4
2.1 예상 설계 구조.....4
2.2 최종 설계 구조.....6
2.3 최종 설계 구조 세부 모듈 설명.....7
2.4 결과.....20
2.5 설계 이전 예상 구조 와 최종 설계 구조의 차이점 및 원인.....33
3. 본론.....34
3.1 Futher_Work.....34
3.2 느낀점.....35

1. 서론

1.1 개요

본 프로젝트는 Ultra96 v2 (Zynq UltraScale+ MPSoC) FPGA를 기반으로 Camera 입력 영상을 실시간으로 처리하여 TFT-LCD에 출력하는 HW/SW Co-design 시스템 구현을 목표로 한다. 고성능의 CPU를 사용하더라도 반복적인 픽셀 연산(Convolution 등)을 소프트웨어로만 처리하는 것은 CPU 리소스를 과도하게 점유하여 비효율적이다. 이에 따라 연산량이 많은 데이터 변환 및 필터링 연산 작업을 하드웨어로 가속하고 CPU는 시스템 제어에 집중하도록 역할을 분담하여 전체적인 처리 속도와 효율성을 극대화하는 것을 목표로 한다.

1.2 팀원 별 역할 분배

팀원	역할
김무성	InBuf_Ctrl, Convolution, ReLU, Pixel Conversion 설계 및 검증, Register Map 및 AXI 모듈 설계, Top module 검증
송승규	OutBuf, TFT-LCD Control module 설계 및 검증, Top module instantiation 및 PS, PL 결합
우상욱	Window3x3_pixel 설계 및 검증, Lcd_OutBuf control logic 설계 및 검증, PS C-code 작성, Top module 검증

1.3 세부 설계 계획 및 진행 내용

날짜	진행 내용
2025.12.1	설계 구조 논의 및 역할 배분
2025.12.3	이 전 설계 구조(PL_CNN)에 Camera 결합 (결과: 화면 출렁거림, 상하 반전, 색상반전)
2025.12.5	화면 상하 반전 및 색상 반전 해결, 화면 출렁거림을 해결을 위해 Line Buffer 구조로 변경
2025.12.8	InBuf dual port 2개로 변경 및 OufBuf dual port로 변경, 하지만 OufBuf가 계속 덮어쓰여지고 FPS가 떨어지는 문제 계속 발생
2025.12.10	카메라 입력, LCD 출력 싱크 조절
2025.12.12	실시간 카메라 입력, LCD 출력 정상 동작 확인
2025.12.13	TB에서 카메라 구현 및 전체 Top 모듈 검증
2025.12.14	최종 보고서 완성

2. 본론

2.1 예상 설계 구조

예상 설계 구조는 **Figure5**와 같다. 본 시스템은 크게 제어 명령을 처리하는 PS (Processing System)와 영상 데이터의 연산을 담당하는 PL (Programmable Logic)으로 구분되어 동작한다.

먼저 PS 영역에 위치한 ARM Cortex-A53 프로세서는 시스템 전반의 초기화 및 필터 모드의 설정을 관리한다. PS는 영상 데이터 경로에 직접 관여하지 않고 I2C(SCCB) 인터페이스를 사용하여 카메라 센서의 해상도와 포맷을 초기화한다. 또한 UART 인터페이스를 통해 PC 터미널과 연결되어 사용자 입력을 수신하며 AXI Master 인터페이스를 통해 PL 내부 레지스터에 접근한다. 이때 PS는 사용자의 선택에 따라 필터 모드 변경 값 혹은 직접 입력 받은 필터 계수(Coefficient)를 레지스터로 전송하여 필터 계수을 실시간으로 제어한다.

실제 영상 데이터 처리는 PL 영역의 최상위 모듈인 Cam_top.v 내부에서 순차적으로 수행된다. 카메라로부터 입력된 신호는 InBuf에 저장된 후 Window3x3 모듈을 거쳐 연산에 필요한 픽셀 단위로 정렬된다. 정렬된 데이터는 Conv3x3 & ReLU 모듈로 전달되는데 이 모듈은 PS가 설정한 레지스터의 필터 모드와 필터 계수 값을 참조하여 Convolution 연산 및 활성화 함수를 적용한다. 처리된 결과는 RGB888toRGB565 모듈을 통해 RGB565 포맷으로 변경된 후 OutBuf와 LcdCtrl 모듈을 거쳐 TFT-LCD 타이밍에 맞춰 출력된다. 위 과정을 순차적으로 처리하여 한장의 이미지를 처리하고자 한다. 이 구조에서 가장 특징적인 구조는 Resource를 최소화한 구조이라는 것이다. 다음 두 모듈을 보면 이야기하고자 하는 바를 알 수 있다.

Window3x3 function

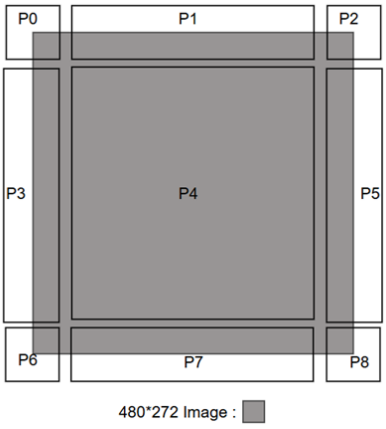


Figure 1<State 구역 설명>

다음과 같이 zero padding이 되는 구역을 구분하여 state를 설정하여 P0, P1, P2 구역에서는 첫번째 line을 가져온다. P0일때는 4개 P1일 때는 2개를 input buffer에서 가져오고 P2일 때는 가져올 필요가 없다. P3, P4, P5도 마찬가지로 각각 6개, 3개, 0개를 가져온다. P6, P7, P8은 마지막 line으로 첫번째 line과 같다. **이렇게 Data Reuse를 통해** 이미지를 input buffer에서 읽어와 제로 패딩을 포함한 3x3 총 9개의 픽셀 정보를 convolution 연산기에 전달한다.

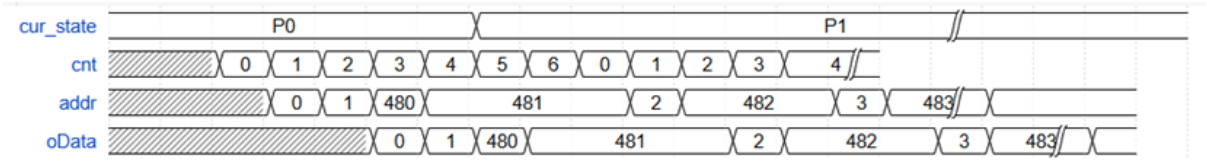


Figure 2 <address timing diagram>

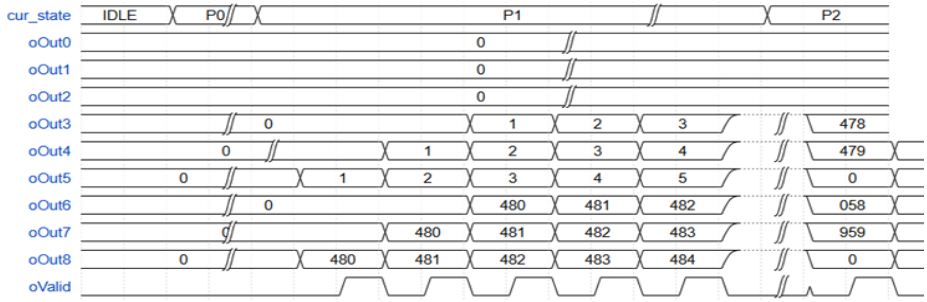


Figure 3 <window3x3 모듈 timing diagram>

Convolution & RELU

Window 3x3 module에서 만들어준 24bit의 9개의 pixel 데이터로 연산을 수행한다. Window3x3 모듈의 enable 신호에 반응하여 연산을 시작하고, 연산 후 Pixel Conversion에 valid 값을 전달한다. 다음과 같이 리소스를 최소한으로 사용하여 MAC 9개와 ReLU 연산기 하나를 재사용하는 방식으로 Timing Diagram을 구성했다.

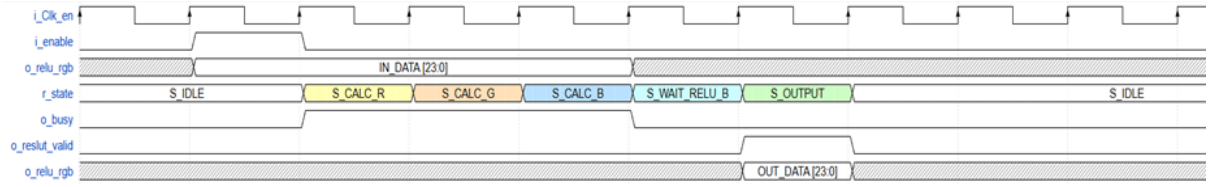


Figure 4 <Convolution 모듈 timing diagram>

이렇게 최소한의 Resource 를 사용하여 Convolution 연산을 수행하여 Output_Buffer 에 Data 를 쌓는 방식으로 예정하였다. 다음은 전체 아키텍처의 구조를 나타낸 것이다.

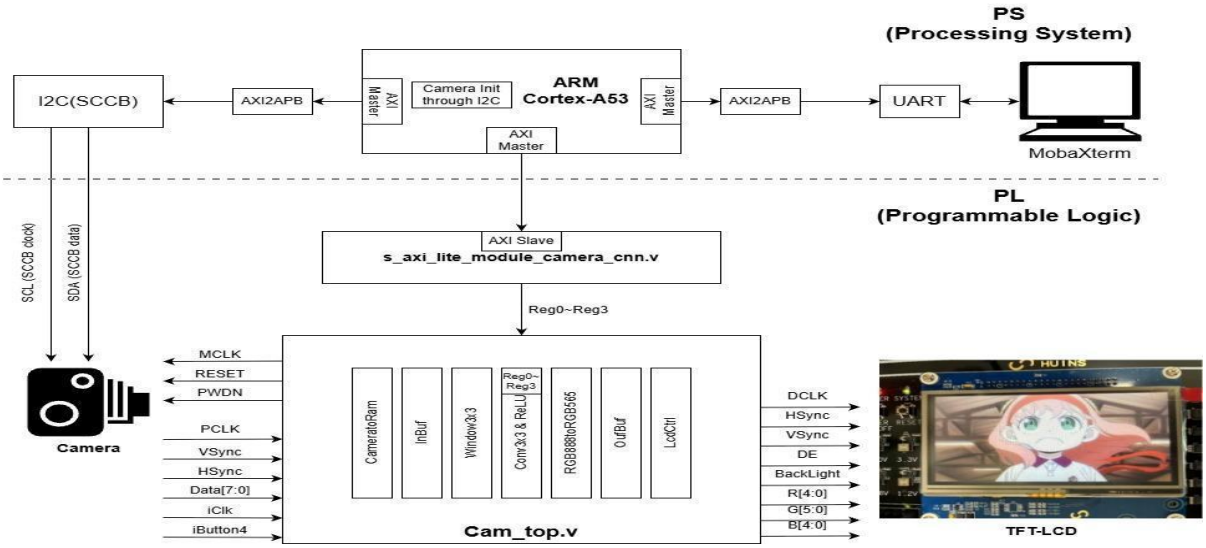


Figure 5 <예상 설계 구조>

2.2 최종 설계 구조

최종 설계 Block diagram은 **Figure6**와 같다. 먼저 PS 영역에서 제어하는 부분을 살펴보면, AXI4-Lite 인터페이스를 통해 PL 영역의 iReg0~iReg3 레지스터에 접근하여 하드웨어를 제어한다. 구체적으로 사용자가 UART 통신을 통해 필터링 모드(Sharpening, Bypass 등)을 선택하거나 임의의 필터 계수를 입력하면 PS는 해당 값을 레지스터에 기록하여 Conv3x3_RGB888.v 모듈의 연산 커널을 실시간으로 재구성하도록 설계하였다.

실질적인 영상 데이터 처리 흐름은 PL 영역의 최상위 모듈인 Cam_top.v 내부의 하드웨어 파이프라인을 통해 이루어진다. 카메라 인터페이스(camera_to_ram.v)를 통해 수신된 8-bit 두 쌍의 데이터는 가장 먼저 Camer_to_ram을 거쳐서 in_buf_ctrl.v 모듈로 전달된다. 이 모듈은 실시간 처리시 발생하는 입출력 속도 차이를 극복하기 위해 두 개의 메모리 공간을 교차 사용하는 이중 버퍼(Double Buffering) 구조를 채택하였으며 VSync 신호를 기준으로 Write/Read 동작을 제어하고 또한 전체 System의 Sync를 맞춰주기 위해 TFT-LCD의 동작을 확인해 만약 동작 중이라면 새로운 이미지 data를 읽지 않는 Frame-Drop 또한 수행한다.

이후 버퍼로부터 읽힌 데이터는 Window3x3_RGB888.v 모듈로 입력되어 내부 라인 버퍼(Line Buffer)와 Shift register를 통해 매 클럭마다 3x3 크기의 픽셀 윈도우로 재구성된다. 생성된 3x3 윈도우 데이터는 핵심 연산 모듈인 Conv3x3_RGB888.v로 전달되며 이곳에서 총 27개의 MAC 연산기를 통해 PS가 설정한 필터 계수와 연산이 수행된다. 연산 결과는 ReLU 활성화 함수를 거쳐 유효한 픽셀 값으로 확정되며, 이후 RGB888ToRGB565.v 모듈을 통해 TFT-LCD 출력 포맷인 RGB565 포맷으로 변환된다. 최종 처리된 영상 데이터는 출력 버퍼에 저장된 후, LcdCtrl_RGB565.v 모듈이 생성하는 VSync, HSync 및 DE 신호에 동기화되어 TFT-LCD 화면에 실시간으로 출력되는 구조이다.

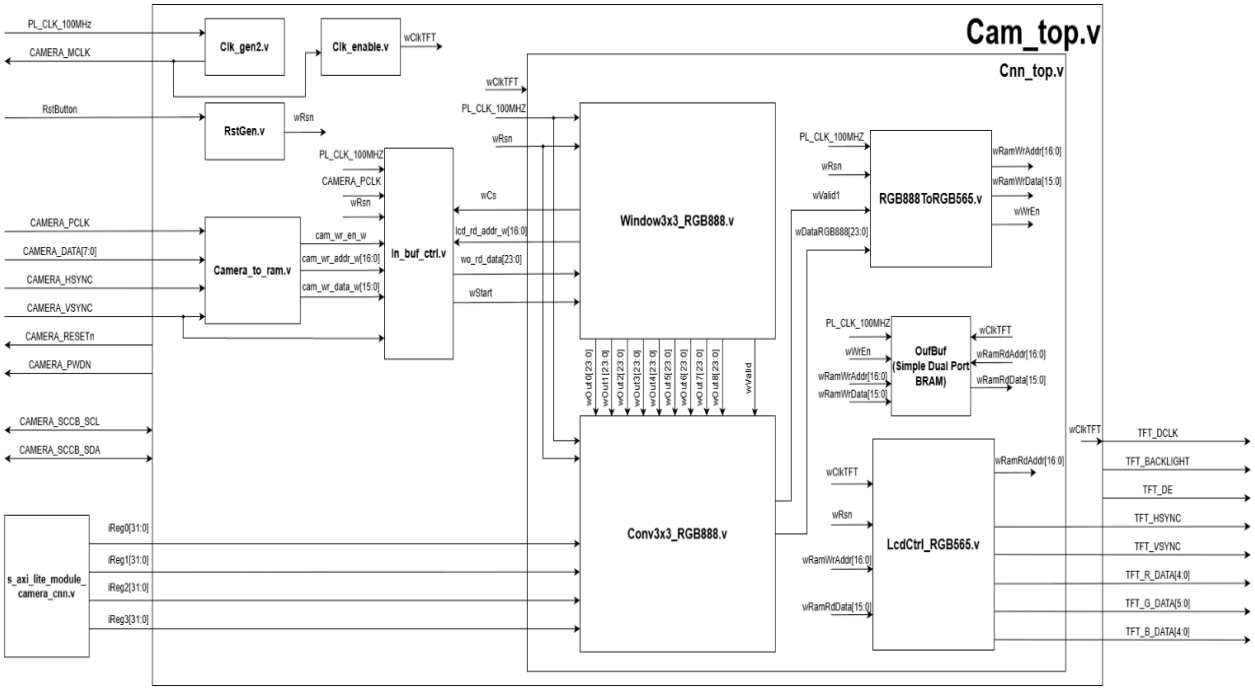


Figure 6 <HW Block Diagram>

2.3 최종 설계 구조 세부 모듈 설명

최종 설계 구조의 세부 모듈의 동작과 원리를 살펴보면 다음과 같이 구체적으로 나타낼 수 있다.

s_axi_lite_module_camera_cnn.v

PS와 PL 간의 통신 인터페이스 모듈이다. PS로부터 전달된 필터 모드 선택 값(iReg0)과 사용자 정의 커널 계수(iReg1~iReg3)를 수신하여 이를 PL 내부 레지스터에 저장함으로써 SW에 의한 실시간 하드웨어 제어를 가능하게 한다.

clk_gen2.v / clk_enable.v

시스템의 동기화를 위한 클럭 생성기이다. FPGA의 메인 PLL로부터 입력 받은 100MHz 클럭을 기반으로 카운터 로직을 통해 분주(Division)을 수행하여 Camera에 입력되는 25Mhz Clk인 MClk을 생성할 뿐만 아니라 TFT-LCD 타이밍 규격에 적합한 12.5MHz의 픽셀 클럭(DCLK)을 생성한다.

camera_to_ram.v

OV7670 카메라 센서로부터 입력되는 PCLK, VSync, HSync 신호에 동기화하여 8-bit 데이터 쌍을 구성하여 각 픽셀의 정보를 올바른 Address와 함께 뒷단의 inbuf로 전달하는 모듈이다.

in_buf_ctrl.v

InBuf_ctrl 모듈은 카메라 입력부와 시스템 처리 부 사이에서 영상 데이터를 안정적으로 전달하기 위한 실시간 프레임 버퍼 컨트롤러 역할을 수행한다. 현재 시스템의 병목인 TFT-LCD의 출력 타이밍에 가리워지기 위해 더블 버퍼링을 도입하여 DATA를 저장하는 BRAM, DATA를 내보내는 BRAM 서로 따로 두어 Convolution 연산기에 DELAY 없이 바로바로 내보내 Throughput을 높이고자 하였다. 이를 통해 한 쪽 메모리 뱅크가 카메라로부터 새로운 프레임을 수신하는 동안, 동시에 다른

쪽 뱅크는 이전 프레임 데이터를 시스템으로 송신함으로써 끊김 없는 데이터 파이프라인 처리를 보장한다.

제어 로직을 간단하게 살펴보면 다음과 같다. 카메라의 수직 동기 신호(VSYNC)를 기준으로 한 뱅크 스위칭(Bank Switching) 메커니즘이다. 프레임이 종료되는 시점에 맞춰 쓰기 및 읽기 권한을 가진 메모리 뱅크를 상호 전환하며, 새로운 data를 읽고 기존 data를 내보낸다 이때 시스템의 병목 지점인 TFT-LCD가 만약 데이터를 출력하고 있다면 OutPut Buffer에 새로운 값이 덮어씌워지는 것을 방지하기 위해 LCD 출력 주소가 초기화된 시점에만 전환이 이루어지도록 설계하여 프레임 드랍(Frame Drop)을 구현하여 전체 System의 Sync를 맞추었다. 특히, 현재 모듈은 48Mhz와 100Mhz로 동작하는 도메인 사이에 존재하므로 신호 전달 안정성을 확보하기 위해 뱅크 선택 신호에 2-stage Synchronizer를 적용하여 CDC로 인한 Metastability 문제를 해결하였다.

데이터 저장 및 출력 단계에서는 FPGA 내부 메모리 자원(Block RAM)를 이용하여 RGB565(16-bit) 포맷으로 저장한다. 이후 후단 영상 처리 모듈로 데이터를 전달하는 출력 과정에서는, 저장된 5~6비트 색상 데이터를 RGB888(24-bit) 포맷으로 실시간 복원한다. Timing Diagram은 다음과 같이 나타낼 수 있다.

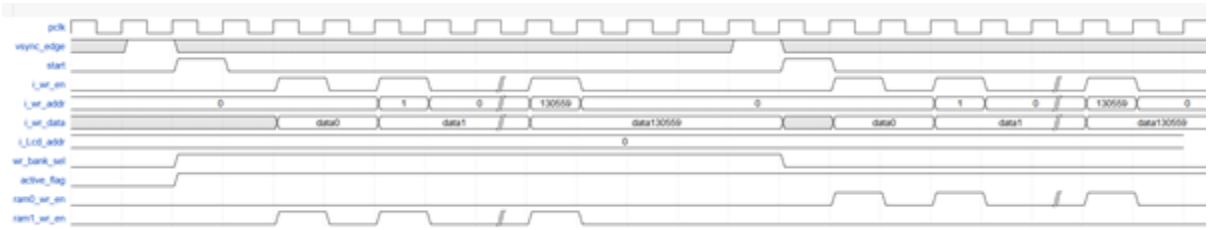


Figure 7 <InBuf_ctrl Write Timing Diagram>

다음은 쓰기 동작을 수행할 때의 InBuf_ctrl Module의 Timing Diagram이다. Camera의 Vsync로 인해 발생하는 Vsync_edge의 발생 여부에 따라 wr_bank_sel register가 0과 1을 바꾸게 되고 이는 현재 들어오는 image Data를 쓸 RAM을 결정하게 된다 앞부분에서는 wr_bank_sel이 1이므로 i_wr_en이 정확하게 ram1_wr_en로 들어가게 된다. 이후 전체 1-Frame의 전송이 완료된 후 Vsync_edge가 다시 발생하게 되고 CNN 연산을 시작하라는 Start 신호뿐만 아니라 wr_bank_sel이 다시 0으로 바뀌게 되면서 i_wr_en 신호가 ram0로 들어가게 되어 이미지 data가 이번에는 ram0에 쓰이게 된다. 이때 위 신호에서 active_flag가 존재하게 되는데 이는 시스템의 전체 Sync를 맞춰주기 위해 존재한다. 구체적인 역할은 전체 TestBench를 통해 설명하고자 한다.

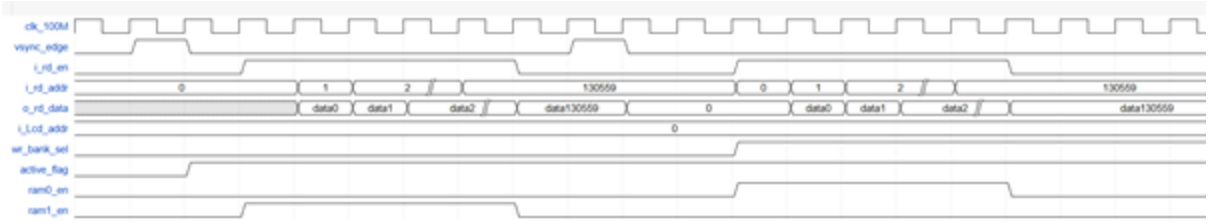


Figure 8 <InBuf_ctrl Read Timing Diagram>

Read 동작은 Write랑 다르게 wr_bank_sel 값에 반대되는 RAM이 읽히게 된다. 위 Timing Diagram을 보면 알 수 있듯이 Vsync_edge가 발생하여 wr_bank_sel이 0일 때 i_rd_en 신호는 ram1_en로 들어가 RAM1에 적혀있는 값을 읽어오는 것을 확인할 수 있다. 이러한 방식으로 이후 Vsync_edge가 발생하여 wr_bank_sel이 바뀌면 Read 동작을 수행하는 Ram도 바뀌게 된다.

in_buf_ctrl.v 중 일부

```
// 2. Control Signal Demux (신호 분배)
wire valid_wr_en = i_wr_en && active_flag;

// [RAM 0 제어 신호]
// Write Enable: 카메라(Write)가 켜져있고 + 현재 타겟 뱅크가 0 일 때
wire ram0_wea = valid_wr_en && (wr_bank_sel == 1'b0);

// Read Enable: 윈도우(Read)가 켜져있고 + 현재 타겟 뱅크가 1 일 때 (즉, 0 은 읽기 가능)
wire ram0_enb = i_rd_en && (wr_bank_sel_100M == 1'b1);
// [RAM 1 제어 신호]
// Write Enable: 카메라(Write)가 켜져있고 + 현재 타겟 뱅크가 1 일 때
wire ram1_wea = valid_wr_en && (wr_bank_sel == 1'b1);

// Read Enable: 윈도우(Read)가 켜져있고 + 현재 타겟 뱅크가 0 일 때 (즉, 1 은 읽기 가능)
wire ram1_enb = i_rd_en && (wr_bank_sel_100M == 1'b0);
```

다음은 TestBench를 통한 시뮬레이션 로직 검증이다.

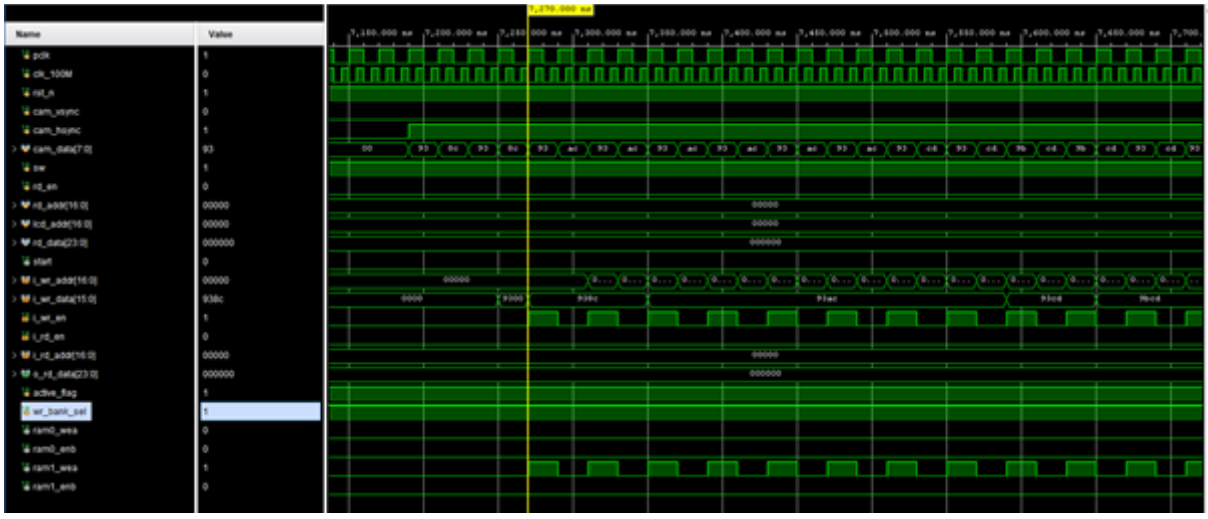


Figure 9<InBuf_ctrl TestBench(1)>

위 시뮬레이션과 코드를 보면 알 수 있듯이 Vsync를 기반으로 RAM0와 RAM1의 역할이 Swithching 됨을 알 수 있다. 위 시뮬레이션 상황에서는 Camera를 통해 현재 RAM1에 DATA가 적히고 있음을 알

수 있다. 이후 130560 Pixel에 대한 모든 정보가 다 들어오게 되면 Camera는 Vsync를 발생시키고 다음과 같이 RAM의 역할이 스위칭 된다.

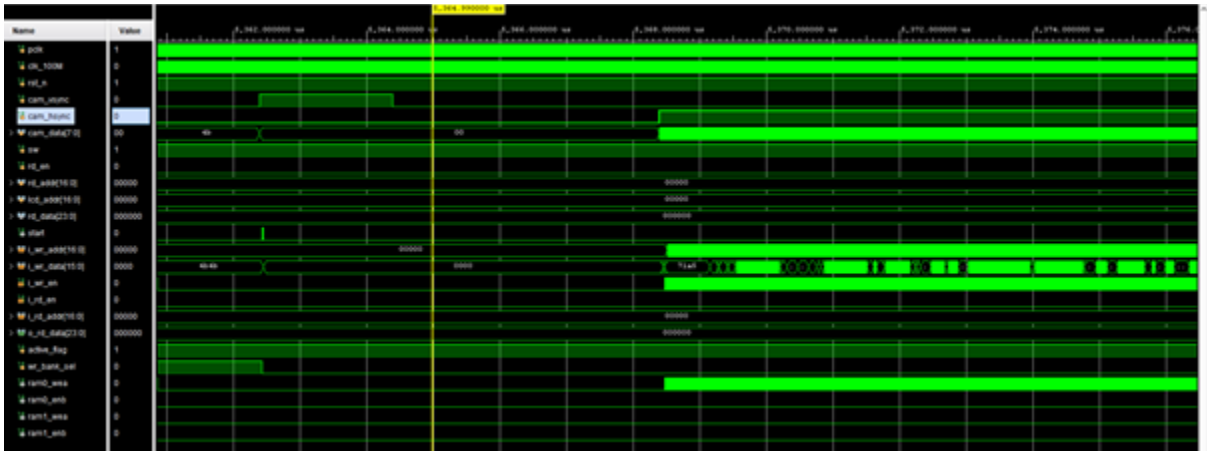


Figure 10 <InBuf_ctrl TestBench(2)>

위 사진을 보면 알 수 있듯이 Vsync 발생 이후에는 RAM0가 Write_enable 신호를 통해 Data를 받아들이는 것을 확인할 수 있다. 이러한 방식으로 두 RAM에 두 장의 이미지 DATA를 전송하고 동일한 방식으로 다시 RAM에 적혀 있는 값들을 읽어와 Golden 값과 비교한 결과 다음과 같이 모든 픽셀이 동일함을 알 수 있고. 이는 로직이 올바르게 작동함을 확인할 수 있다.

InBuf_ctrl Testbench 중 일부

```
for (i = 0; i < FRAME_PIXELS; i = i + 1) begin
    // ~~생략 주소 setting 및 bram latency 대기
  if (which == 0)
    pix16 = img1_mem[i];
  else
    pix16 = img2_mem[i];
  expected_rgb888 = rgb565_to_rgb888(pix16);
  // 5) 실제 값과 비교
  if (rd_data !== expected_rgb888) begin
    $display("[%0t] ERROR: image%0d mismatch at pixel %0d: got=%h, expected=%h",
         $time, which+1, i, rd_data, expected_rgb888);
    $stop;
  end
end
rd_en = 1'b0;
$display("[%0t] INFO: image%0d check PASSED (all %0d pixels matched).",$time, which+1, FRAME_PIXELS);
```

```
[TB] Send Frame 0 from image1.txt
[TB] Send Frame 1 from image2.txt
[12014186000] INFO: image1 check PASSED (all 130560 pixels matched).
[13330766000] INFO: image2 check PASSED (all 130560 pixels matched).
[TB] Simulation done.
```

Figure 11 <InBuf_ctrl Testbench Result>

Window3x3_RGB888.v

100MHz 클럭 기반으로 동작하며 3x3 Convolution 연산에 필요한 3x3 Window를 생성하는 모듈이다. 내부의 Line Buffer와 Shift Register를 사용한다. 동작 방식은 Figure12와 같이 구성된다. 먼저 첫번째 줄의 픽셀 값들을 모두 가져온다. 그 후에는 값이 최소 두개만 있으면 윈도우가 구성됨으로 첫번째 줄과 다음 두개의 값 이후에는 매 Clk 3X3윈도우를 만들어 보내줄 수 있다.

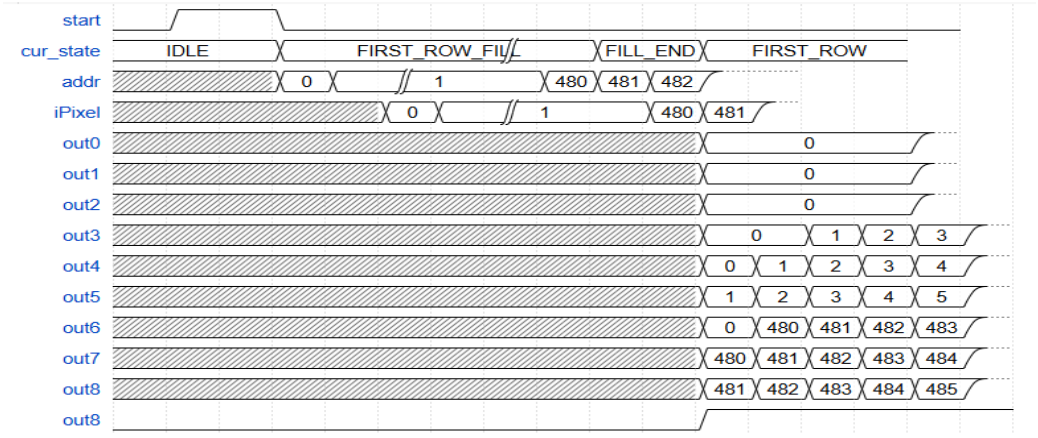


Figure 12 <LineBuf Window Logic Timing diagram>

실제 설계는 다음과 같은 Figure13과 같은 FSM으로 구성되었다. FIRST_ROW_FILL_END State가 끝나면 앞서 말했던 첫 줄과 최소 필요한 다음 두개의 픽셀 값이 레지스터에 저장되고 FIRST_ROW state부터 값을 뽑아주게 된다.

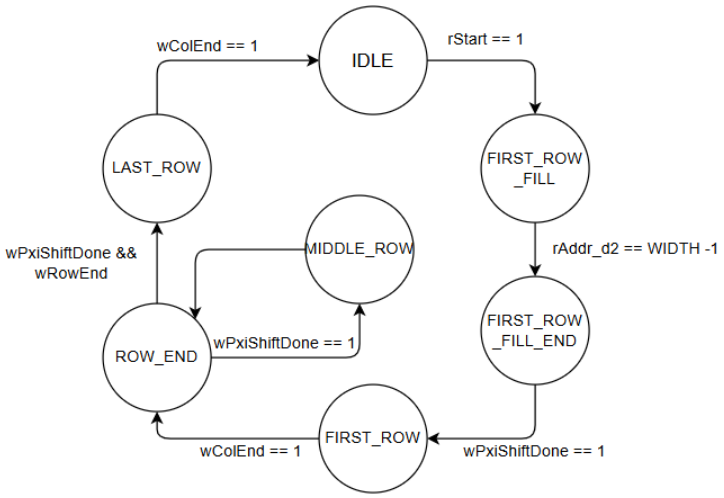


Figure 13 <Window logic FSM>

값을 뽑아주게 된다. Figure14는 Window 모듈의 개략적인 구성도이다. Linebuf0, Linebuf1, register 4개는 각각 out0~out2, out3~out5, out6~out8로 표현하는 줄이 고정되어 있고 counter에 맞춰 muxing되며 zero padding mux를 통해 다음 모듈로 9개의 픽셀 값이 전달된다.

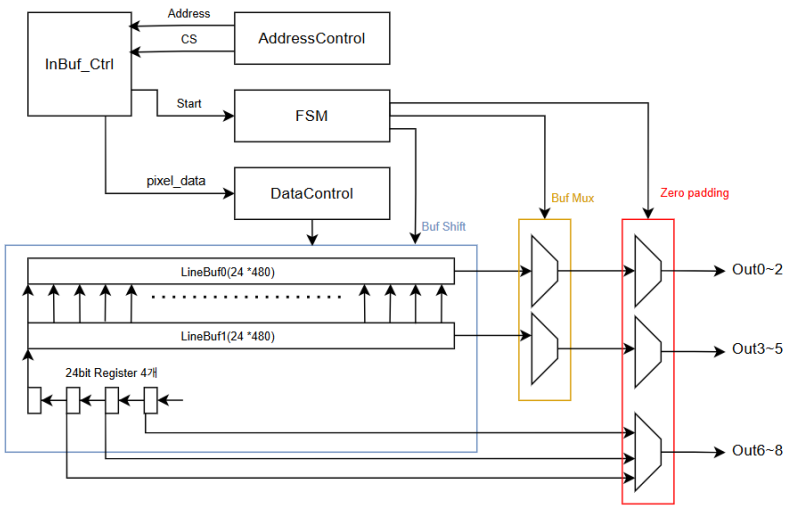


Figure 14 <Window 모듈 구성도>

모듈로 9개의 픽셀 값이 전달된다. 다음은 Window 모듈의 TestBench 결과이다. 초기 모듈 설계시에는 SpSram에 coe파일을 주소 번지에 맞게 0부터 130559까지 넣어놓고 값을 잘 읽어오는지 테스트를 했고 Figure15와 Figure16은 그 결과이다.

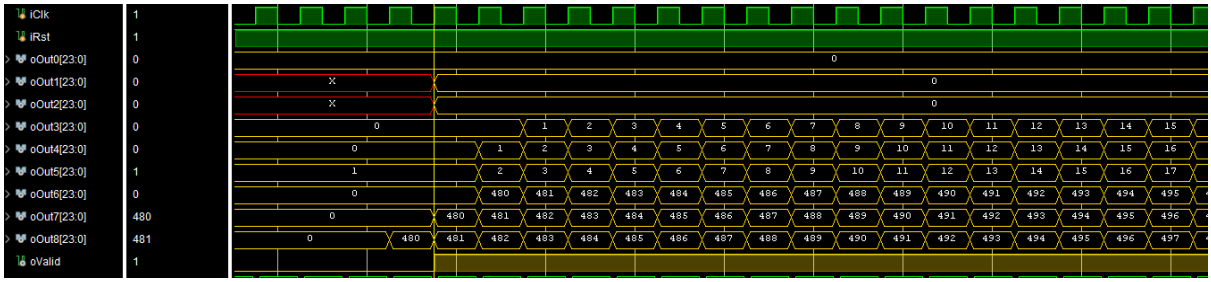


Figure 15 <Window 모듈 testbench(1)>

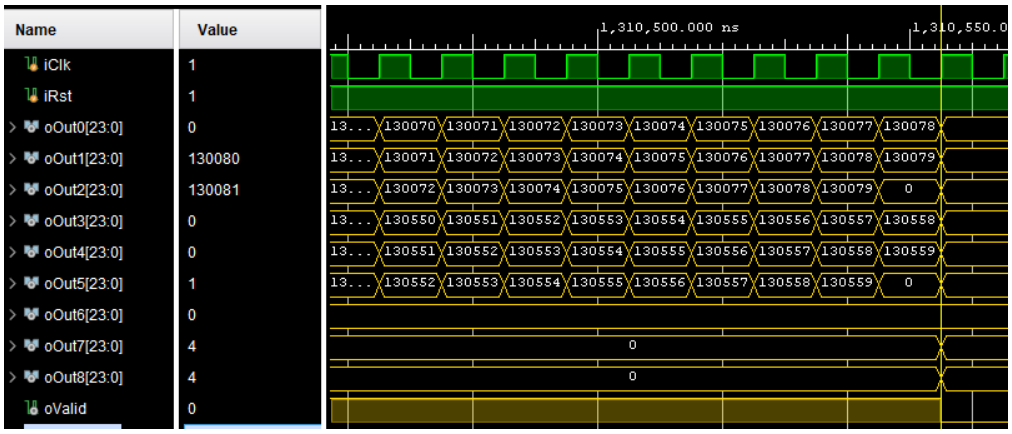


Figure 16 <Window 모듈 testbench(2)>

후에 연산기와 같이 검증하기 위해 모듈 뒤에 연산기를 붙여 검증하였다. 내부 모듈에서 outBuf에 한 이미지가 모두 연산되어 저장된 순간 done_valid 신호가 1이 되기 때문에 이때를 기준으로 outbuf에 모든 값을 읽어와서 input대비 출력과 golden값을 비교하였고 모든 픽셀 통과하였다. 테스트를 용이하게 하기 위해 outbuf는 register array로 구성하여 테스트했다.

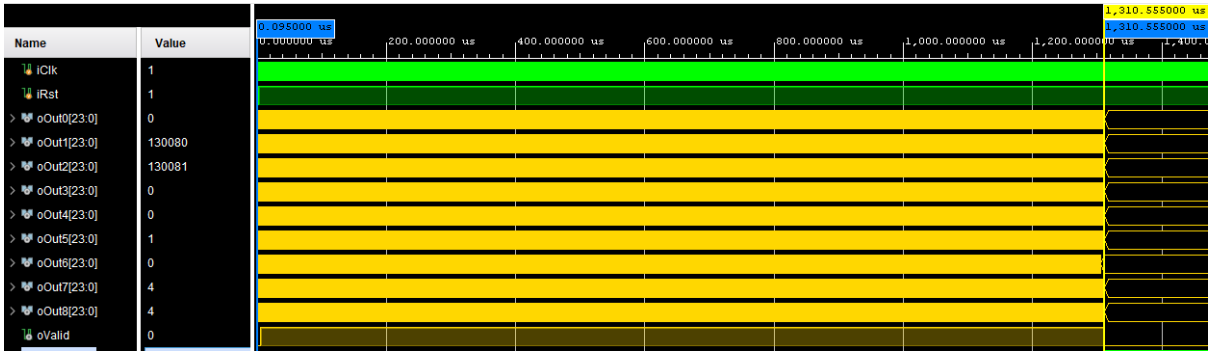


Figure 17 <Window 모듈 + 연산기 testbench>

Figure17로 한 이미지를 처리하는데 얼마정도 걸리는지 알 수 있다. 한 프레임 당 1.3ms로 초기 설계에 비해 매우 빠른 속도로 연산 가능함을 알 수 있다.

TestBench코드 중 일부

```
reg [15:0] expected_rgb565;
    always @(posedge top.u_cnn_top.u_RGB888ToRGB565.done_valid_reg)begin
        for(i = 0; i<DEPTH;i=i+1) begin
            expected_rgb565 = expected_memory[i];
            if(top.u_cnn_top.u_OufBuf_DPSram_RGB565.rOufBuf[i] != expected_rgb565) begin
                flag = 0;
                $display("ERROR!!!: %d expected: %h , outbuf: %h
       \n",i,expected_rgb565,top.u_cnn_top.u_OufBuf_DPSram_RGB565.rOufBuf[i]);
                $finish;
            end
        end
        if(flag == 1) begin
            $display("All Completed\n");
        end
        else begin
            $display("Failed\n");
        End
   $finish;
    end
```

Conv3x3_RGB888.v

Conv3x3_RGB888 모듈은 입력된 3x3 픽셀 윈도우에 대해 실시간 컨볼루션 연산을 수행하는 가속기의 핵심이다. 기존 Resouce를 줄이기 위해 선정한 순차적 처리 방식 LCD의 최대 FPS를 뽑아내기 위해 R, G, B 각 채널의 9개 픽셀에 대해 총 27개의 곱셈기와 3개의 ReLu 연산기가 동시에 동작하는 병렬 아키텍처를 적용하였다, 이를 통해 매 클럭마다 9개의 유효한 픽셀 DATA를 전달하는 line Buffer에 동기화 하여 매 Clk 마다 유효한 결과값을 산출하는 1-Cycle Throughput 성능을 확보하였다. 이뿐만 아니라, 외부 AXI 인터페이스와 연동되는 레지스터 설정을 통해 Sharpen, Strong Sharpen, Bypass, 사용자 정의 필터 등 다양한 필터로 교체할 수 있도록 설계하였다.

최종 출력 단계에서는 하드웨어 ReLU 및 Clamping 로직을 적용하여, 연산 결과가 0 미만이거나 255를 초과할 경우 이를 8-bit 유효 범위(0~255) 내로 자동 보정함으로써 색상 왜곡 없는 안정적인 영상 데이터를 출력한다. 위 아키텍처를 나타내면 다음과 같이 그림을 그릴 수 있다.

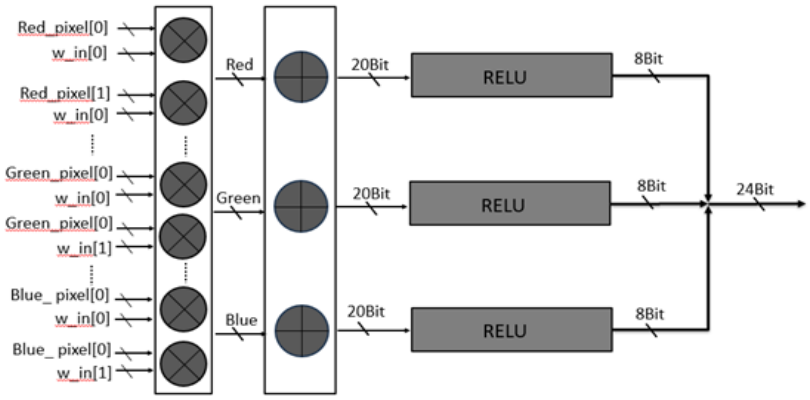


Figure 18 <conv 3x3 구조>

RGB 각각의 채널에 9 개의 MAC 연산기와 1 개의 RELU 연산기가 할당 되어있음을 알 수 있다.
다음은 Timing Diagram 및 Simulation 결과이다.

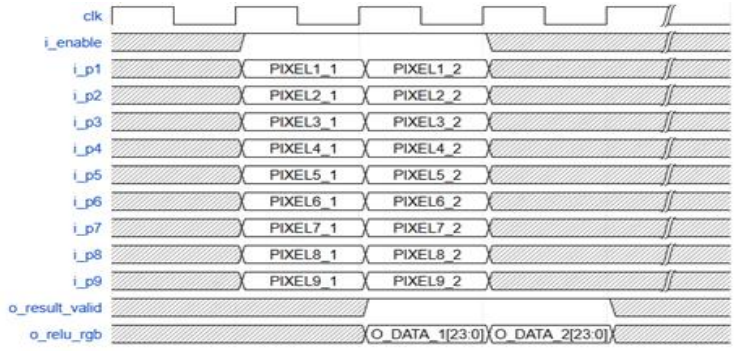


Figure 19 Conv3X3_RGB888 Timing Diagram>

앞서 설명하였듯 1-Cycle 만에 유효한 값이 Convolution 연산과 Activation(RELU)를 거쳐서 바로 출력됨을 알 수 있다.

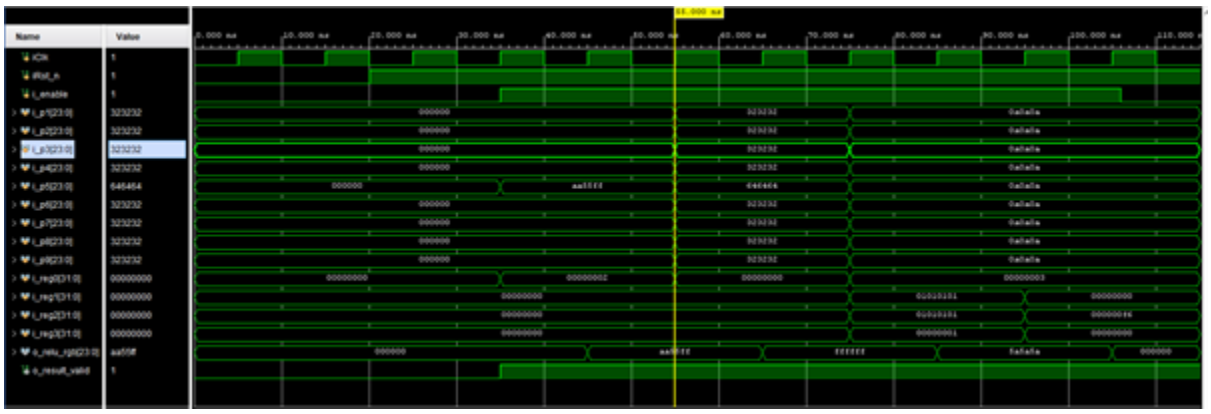


Figure 20 <Conv3x3_RGB888 Simulation>

위 Simulation 을 보면 알 수 있듯이 각 모드에 맞는 Filter 와 Input 에 대하여 기대하는 값이 o_enable 신호와 함께 출력됨을 알 수 있다. 예를 들어 MODE 3 번은 사용자 정의 Filter 이다. 그래서 위 Simulaiton 과 같이 입력을 주었을 때 예상되는 값은 **0x5a5a5a** 이다.

P1= 0x0a0a0a, P2= 0x0a0a0a, P3= 0x0a0a0a, P4= 0x0a0a0a,

P5= 0x0a0a0a, P6= 0x0a0a0a, P7= 0x0a0a0a, P8= 0x0a0a0a, P9= 0x0a0a0a

9 개의 Filter 값 =0x01

Simulation 결과를 보면 알 수 있듯이 o_result_valid 신호와 함께 0x5a5a5a 가 출력되고 있음을 알 수 있다.

RGB888ToRGB565.v

위 모듈은 앞서 계산된 24비트 RGB888 형식의 픽셀 데이터를 입력받아 16비트 RGB565 형식으로 실시간 변환하며, 동시에 이를 레지스터 파일로 정의된 외부 메모리에 순차적으로 기록하는 메모리 쓰기 컨트롤러입니다. 이 모듈의 핵심 기능은 크게 두 가지로 나눌 수 있다. 데이터 변환과 메모리 주소 생성 및 쓰기 제어이다. 각각의 기능을 살펴보면 다음과 같다.

RGB888에서 RGB565로 Conversio을 수행할 때 에는 다음과 같은 공식을 거치게 된다.

$$\mathbf{RED\_PIXEL[5BIT]} = \mathbf{RED\_PIXEL[8BIT]} * \mathbf{31/255}$$

하지만 좀 더 하드웨어 친화적이게 접근하기 위해 약간의 근사를 진행하게 된다

$$\mathbf{RED\_PIXEL[5BIT]} = \mathbf{RED\_PIXEL[8BIT]} * \mathbf{31/256}$$

위 식으로 바꾸게 되면 단순하게 Shift 연산으로 구현할 수 있게 된다. 위 유도를 통해 RED, BLUE_PIXEL은 SHIFT_RIGHT를 3bit를 진행하면 되고 GREEN_PIXEL은 2bit을 진행하면 된다. 이렇게 조합회로를 통해 Conversion을 진행하였다면 Register File로 구현된 Width 16bit Depth 130560짜리 Output_Buffer에 써주게 된다. 이때 앞단에서 들어오는 Valid신호를 기반으로 address 포인터 역할을 수행하는 addr_cnt를 증가시켜주게 된다. 이러한 방식으로 130560의 Depth를 다 채우게 되면 0으로 리셋해주고 꽉 찼다는 o_done_valid Signal을 내보내게 된다. 다음은 Timing Diagram 및 Simulation이다.

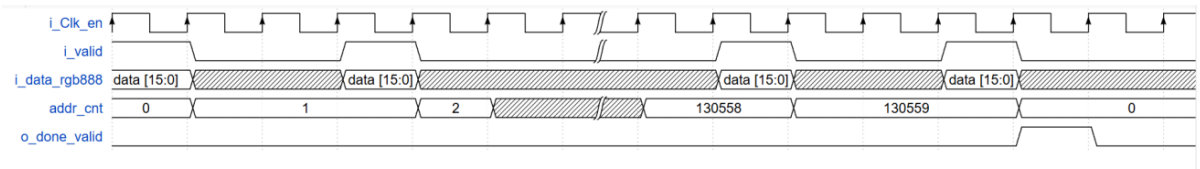


Figure 21 <Pixel conversion timing diagram>

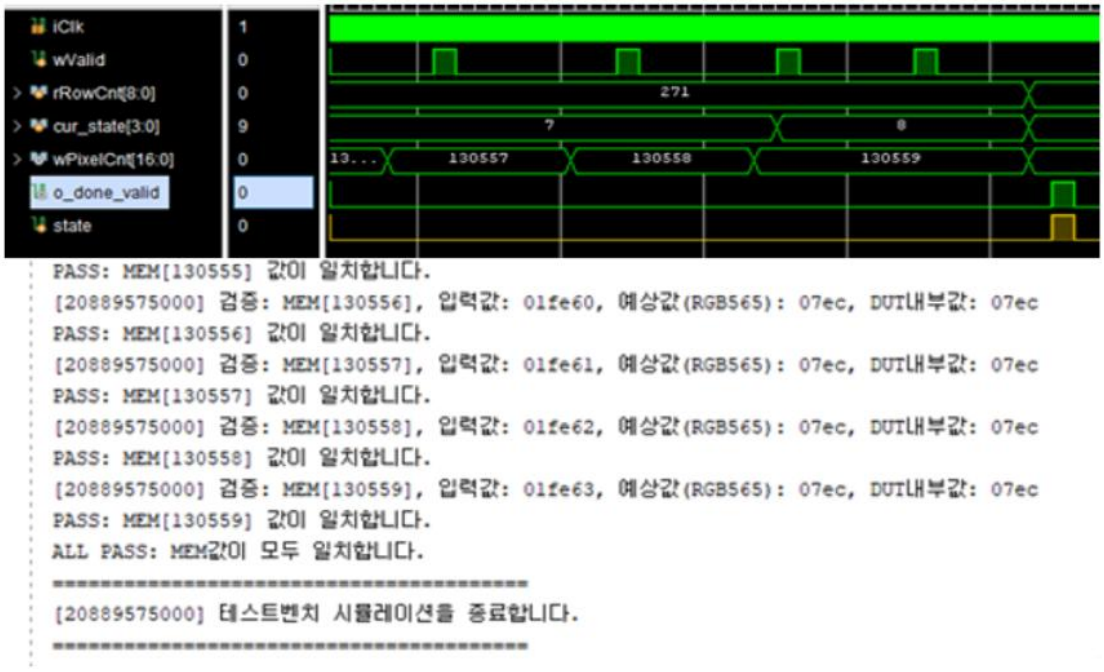


Figure 22 <RGB888ToRGB565 Simulation>

TestBench는 가상의 Input_Vector를 만들어 Register_File에 예상되는 Golden값과 비교하는 방식으로 진행하였다. 위 그림과 같이 모두 예상된 값이 잘 쌓임을 알 수 있다.

LcdCtrl_RGB565.v

12.5MHz 픽셀 클럭에 맞춰 TFT-LCD 구동에 수평 동기(HSync), 수직 동기(VSync), Data Enable(DE) 신호를 생성하고 OufBuf로부터 데이터를 읽어와 TFT-LCD 화면에 출력한다. 이번 프로젝트의 병목은 LCD이다. 따라서 카메라와의 싱크를 맞추기 위한 컨트롤이 필요하다. 기존 LCD control은 계속 해서 outbuf에 있는 값을 읽어 화면에 출력하는 방식이었다. 하지만 앞 단의 clk도 높고 연산도 빠르기 때문에 값이 덮어 씌어지는 문제가 발생한다. 이를 해결하기 위해 최종 모듈에서는 output buffer는 dual port ram을 사용하였고 simultaneous access control을 해주었다. Outbuf에 쓰이는 write address가 0이 아닐 때만 read 동작을 시작한다. write address의 증가는 매우 빠르기 때문에 LCD 입장에서는 0이 아닌 경우에만 동작하면 된다. 앞서 설명했던 inbuf_control은 lcdCtrl의 read address를 보고 프레임 드랍을 통해 싱크를 맞춰준다.

이를 구현하기 위한 FSM은 간단히 2개의 State로 구현 가능하다.

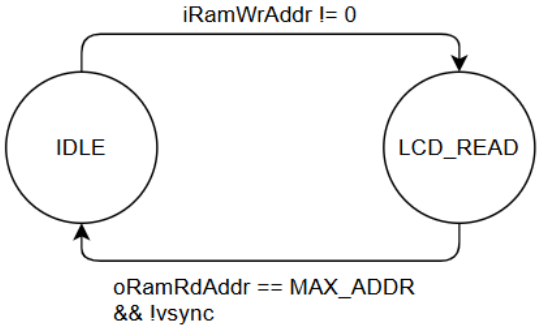


Figure 23 <LcdCtrl FSM>

Testbench를 통해 iRamWrAddr가 0이 아닐 때만 LCD_READ State로 가 oRamRdAddr가 증가하는 것을 확인하였고 iRamWrAddr가 0일때는 IDLE에서 대기하는 동작이 잘 구현되었음을 확인했다.

iRamWrAddr : 연산기에서 OutBuf에 쓰는 Address
oRamRdAddr : LCD에 출력할 픽셀 값을 OutBuf에서 읽어오는 Address

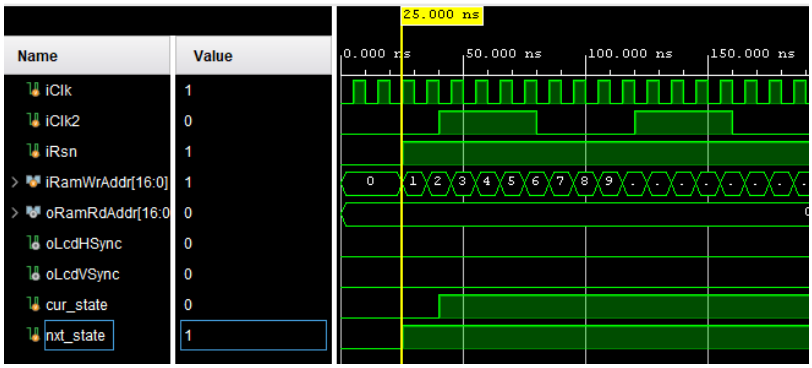


Figure 24 <LcdCtrl testbench 결과 (1)>

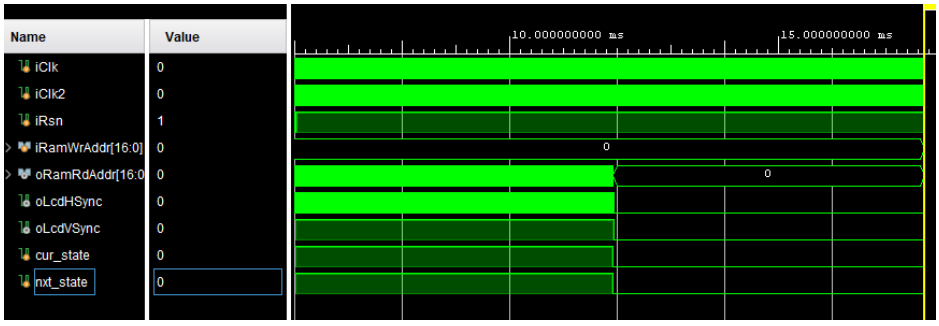


Figure 25 <LcdCtrl testbench 결과 (2)>

I/O List

다음 표는 본 프로젝트의 타겟 보드인 Ultra96 v2 (Zynq UltraScale+ MPSoC)의 물리적 핀 할당(Physical Pin Assignment) 내역을 나타낸 것이다. Vivado 프로젝트의 Constraints 파일(XDC) 설정을 기반으로 작성되었으며 신호의 기능에 따라 **System Control**, **Camera Interface**, **TFT-LCD Interface** 로 구분하여 정리하였다.

Signal	Direction	Pin	Description
PL_CLK_100MHZ	Input	D7	System Clock (100MHz)
RstButton	Input	F8	Asynchronous Active Low Reset
CAMERA_MCLK	Output	F6	Camera Master Clock
CAMERA_PCLK	Input	G5	Camera Pixel Clock
CAMERA_VSYNC	Input	G7	Camera Vertical Sync
CAMERA_HSYNC	Input	F7	Camera Horizontal Sync
CAMERA_DATA[7]	Input	D8	Video Data[7]
CAMERA_DATA[6]	Input	E8	Video Data[6]
CAMERA_DATA[5]	Input	C5	Video Data[5]
CAMERA_DATA[4]	Input	B6	Video Data[4]
CAMERA_DATA[3]	Input	C7	Video Data[3]
CAMERA_DATA[2]	Input	D5	Video Data[2]
CAMERA_DATA[1]	Input	D6	Video Data[1]
CAMERA_DATA[0]	Input	E5	Video Data[0]
CAMERA_SCCB_SCL	Inout	E6	I2C Serial Clock
CAMERA_SCCB_SDA	Inout	G6	I2C Serial Data
CAMERA_PWDN	Output	A7	Power Down Control
CAMERA_RESETn	Output	A6	Camera Reset Control
TFT_DCLK	Output	G1	LCD Clock (12.5MHz)
TFT_VSYNC	Output	F1	LCD Vertical Sync
TFT_HSYNC	Output	E4	LCD Horizontal Sync
TFT_DE	Output	E3	Data Enable
TFT_BACKLIGHT	Output	E1	Backlight On/Off
TFT_R_DATA[4]	Output	R3	LCD Red Data[4]
TFT_R_DATA[3]	Output	U2	LCD Red Data[3]
TFT_R_DATA[2]	Output	U1	LCD Red Data[2]
TFT_R_DATA[1]	Output	T3	LCD Red Data[1]
TFT_R_DATA[0]	Output	T2	LCD Red Data[0]
TFT_G_DATA[5]	Output	M1	LCD Green Data[5]
TFT_G_DATA[4]	Output	M5	LCD Green Data[4]
TFT_G_DATA[3]	Output	M4	LCD Green Data[3]
TFT_G_DATA[2]	Output	L2	LCD Green Data[2]
TFT_G_DATA[1]	Output	L1	LCD Green Data[1]
TFT_G_DATA[0]	Output	P3	LCD Green Data[0]
TFT_B_DATA[4]	Output	N2	LCD Blue Data[4]
TFT_B_DATA[3]	Output	P1	LCD Blue Data[3]
TFT_B_DATA[2]	Output	N5	LCD Blue Data[2]
TFT_B_DATA[1]	Output	N4	LCD Blue Data[1]
TFT_B_DATA[0]	Output	M2	LCD Blue Data[0]

2.4 결과

전체 Top 모듈의 Testbench의 검증 결과를 서술한다. 또한 Vivado SW을 이용한 합성(Synthesis) 및 구현(Implementation) 결과를 제시하고, 최종적으로 FPGA 보드 상에서의 동작을 검증한다.

Top 모듈의 input과 output

```
module top(
input  wire            PL_CLK_100MHZ,
input  wire            RstButton,
inout  wire            CAMERA_SCCB_SCL,
inout  wire            CAMERA_SCCB_SDA,
output wire  [ 4:0]    TFT_B_DATA,
output wire  [ 5:0]    TFT_G_DATA,
output wire  [ 4:0]    TFT_R_DATA,
output wire            TFT_DCLK,
output wire            TFT_BACKLIGHT,
output wire            TFT_DE,
output wire            TFT_HSYNC,
output wire            TFT_VSYNC,
input  wire            CAMERA_PCLK,
input  wire  [ 7:0]    CAMERA_DATA,
output wire            CAMERA_RESETn,
input  wire            CAMERA_HSYNC,
input  wire            CAMERA_VSYNC,
output wire            CAMERA_PWDN,
output wire            CAMERA_MCLK,
// axi lite interface
input wire [31:0] iReg0,
input wire [31:0] iReg1,
input wire [31:0] iReg2,
input wire [31:0] iReg3
);
```

Top 모듈의 input은 카메라이고 ouput의 LCD 출력이므로 우리의 Structure을 엄밀히 검증하기 위해서는 시뮬레이션상에서 카메라의 동작을 구현해야 한다. 이를 위해 Camera의 DataSheet를 파악하여 다음 Task를 통해 Camera의 Signal들을 재현하였다.

figure 6-7 DVP timing diagram

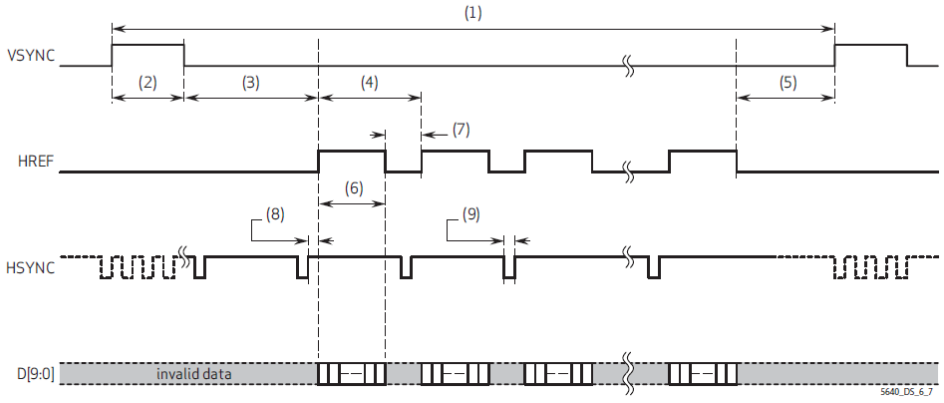


Figure 26<OV5640 DataSheet Timing Diagram>

Testbench상 카메라 구현 코드

```
reg [15:0] pix;
task send_one_frame_from_mem(input integer frame_id);
integer x, y, idx;
begin
    // 1) 프레임 시작 전 VSYNC=1 로 충분히 유지 (카운터 리셋용)
    cam_vsync <= 1;
    cam_hsync <= 0;
    cam_data  <= 8'd0;
    repeat(100) @(posedge pclk);  // 100clk 정도 여유
    // 2) VSYNC=0 내려서 "프레임 시작"
    cam_vsync <= 0;
    repeat(200) @(posedge pclk); // front porch
    idx = 0;
    // 3) 272 라인 전송
    for (y = 0; y < 272; y = y + 1) begin
        // 라인 유효 구간: HSYNC=1
        cam_hsync <= 1;

        for (x = 0; x < 480; x = x + 1) begin
            if (frame_id == 0)
                pix = img1_mem[idx];
            else
                pix = img2_mem[idx];
            // 상위 바이트
```

```
            cam_data <= pix[15:8];
            @(posedge pclk);
            // 하위 바이트
            cam_data <= pix[7:0];
            @(posedge pclk);
            idx = idx + 1;
        end
        cam_hsync <= 0; // 4) 라인 끝: HSYNC 내려서 v_count++ 발생
        repeat(20) @(posedge pclk);  // 라인 블랭크
    end
    repeat(200) @(posedge pclk); // 프레임 사이 블랭크 추가 (선택)
end
endtask
```

위 코드를 보면 알 수 있듯이 send_one_frame_from_mem() Task 를 통해 카메라의 DataSheet 를 기반으로 480x272 해상도의 카메라 모듈이 FPGA 로 영상 데이터를 전송하는 과정을 동일하게 수행할 수 있도록 카메라를 모델링 하였다, 수직 동기 신호(VSYNC)와 수평 동기 신호(HSYNC)의 타이밍을 제어하여 프레임 및 라인 유효 구간을 정의하고. 미리 로드된 2 장의 이미지의 정보를 담고있는 메모리(img_mem)에서 16-bit 픽셀 데이터를 순차적으로 꺼내 와, 8-bit 데이터 버스 대역폭의 제약에 맞춰 상위 바이트와 하위 바이트로 분할하여 2 클럭 사이클에 걸쳐 전송하는 역할을 수행하여 카메라와 동일하게 수행한다.

전체 모듈에 대한 시뮬레이션은 크게 두가지 시나리오가 발생할 수 있다. 첫번째는 프레임이 버려지는 상황, 두번째는 프레임이 버려지지 않고 잘 들어오는 상황이다. 따라서 이 두가지 경우에 대하여 테스트를 진행하였다.

시나리오1

시나리오 1은 프레임 드랍이 발생하는 상황을 구현했다. 프레임 드랍은 LCD쪽에서 OutBuf에 쓰여 있는 값을 읽어오는 상황에서 새로운 프레임이 들어왔을 때 input buffer에 저장하지 않고 버리는 상황이다. LCD control의 READ State일 때 시뮬레이션상에서 새로운 프레임을 입력했고 그 프레임이 거절됨을 확인하였다.

시나리오1 testbench 코드 중 일부

```
task case1;
    begin
        task2_en = 0;
        $display("Case1 Start");
        $display("send frame1");
        send_one_frame_from_mem(0);
        fork : WAIT_OR_TIMEOUT
```

```
            begin : WAIT_BLOCK
                wait (u_top.u_cnn_top.u_LcdCtrl_RGB565.cur_state == 1);
                $display("[OK] LCD_READ entered at %t", $time);
                disable WAIT_OR_TIMEOUT;
            end
            begin : TIMEOUT_BLOCK
                //~timeout waiting
            end
        join
        $display("Send Frame2");
        send_one_frame_from_mem(1);
        //~wait enough time
        $stop;
    end
    endtask
```

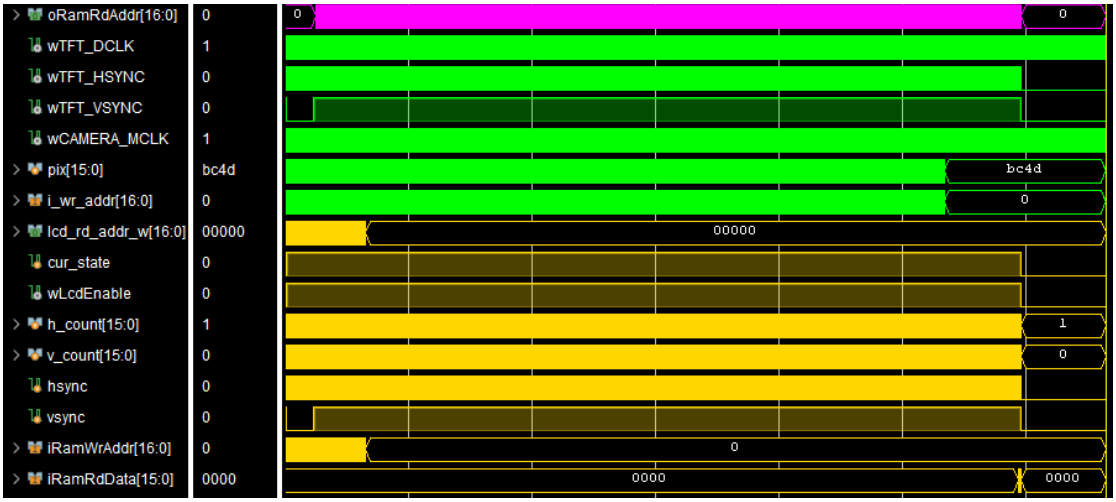


Figure 27 <시나리오 1 결과>

위의 testbench 를 보면 프레임 1 번을 보내고 아직 LCD 가 OutBuf 에 값을 읽고 있는 상태에서 새로운 프레임을 보내는 동작이다. 기대되는 동작은 두번째 프레임을 쓸 때 i_wr_addr 값이 증가 하지 않고 프레임을 버리는 동작이다. LCD 가 READ 상태일 때 들어온 프레임은 써지지 않고 i_wr_addr 값도 증가하지 않음을 확인할 수 있다.

시나리오2

시나리오2는 프레임이 LCD쪽에 READ state가 아닌 IDLE state에 들어오는 상황이다. 이때는 프레임 드랍이 발생하지 않고 LCD쪽에 RGB output이 정상적으로 출력된다. 이때 TFT-LCD를 통해 출력되는 rgb값을 주어진 image data를 바탕으로 동일한 변환을 거친 Phython을 통해 생성한 golden 값과 비교하여 모든 픽셀 값을 비교하였다.

시나리오2 testbench 주요 코드1

```
task case2;
    begin
        task2_en = 1;
        rhsync_delay = 0;
        $display("Case2 Start");
        $display("send frame1");
        send_one_frame_from_mem(0);
        $display("Send Done");
        fork : WAIT_OR_TIMEOUT
            begin : WAIT_BLOCK
                wait (u_top.u_cnn_top.u_LcdCtrl_RGB565.cur_state == 1);
                wait (u_top.u_cnn_top.u_LcdCtrl_RGB565.cur_state == 0);
                $display("[OK] LCD_READ entered at %t", $time);
                disable WAIT_OR_TIMEOUT;
            end
            begin : TIMEOUT_BLOCK
                //~~timeout
            end
        join
        rComNum = 1; rCnt = 2;
        $display("send frame2");
        send_one_frame_from_mem(1);
        $display("Send Done");
        fork : WAIT_OR_TIMEOUT1
            begin : WAIT_BLOCK1
                wait (u_top.u_cnn_top.u_LcdCtrl_RGB565.cur_state == 1);
                wait (u_top.u_cnn_top.u_LcdCtrl_RGB565.cur_state == 0);
                $display("[OK] LCD_READ entered at %t", $time);
                disable WAIT_OR_TIMEOUT1;
            end
            begin : TIMEOUT_BLOCK1
                //~~timeout
            end
        join
        toggle_vsync_only();
        rComNum = 2; rCnt = 2;
        wait(rCnt == 0);
        if(flag==0) $display("TB Succeed");
```

```
        $stop;
    end
    endtask
```

위의 코드는 시나리오 2 상황을 구현해 놓은 task 의 일부이다. 프레임 1 과 프레임 2 가 LCD 와 타이밍이 맞게 들어오는 상황을 구현하였다.

시나리오2 testbench 주요 코드2

```
always @(posedge wTFT_DCLK) begin //cnn result compare
        if(task2_en) begin //only task2 logic
            if(!(wTFT_HSYNC && u_top.u_cnn_top.u_LcdCtrl_RGB565.hsync)) begin
                rhsync_delay <= 0;
            end
            else if(rhsync_delay >=3) begin
                rhsync_delay <= rhsync_delay;
            end
            else if(wTFT_HSYNC) begin
                rhsync_delay <= rhsync_delay + 1;
            end
            case (rComNum)
                1 : begin
                  if(wTFT_HSYNC && rhsync_delay >=3 && u_top.u_cnn_top.u_LcdCtrl_RGB565.hsync) begin
         //bram latency 2 + reg 1
                      if(img1_out[u_top.u_cnn_top.u_LcdCtrl_RGB565.oRamRdAddr-3] != oRGBCom) begin
                            flag = 1;// ~~compare failed
                            $stop;
                        end
                    end
                    else begin
                        if(u_top.u_cnn_top.u_LcdCtrl_RGB565.oRamRdAddr == 130560) begin
                            if(rCnt >= 1) begin // left 2 compare
                                if(img1_out[u_top.u_cnn_top.u_LcdCtrl_RGB565.oRamRdAddr-rCnt]
                != oRGBCom) begin
                                    flag = 1;// ~~compare failed
                                    $stop;
                                end
                                rCnt <= rCnt - 1;
                            end
                        end
                    end
                end
```

```
                2 : begin
                  if(wTFT_HSYNC && rhsync_delay >=3 && u_top.u_cnn_top.u_LcdCtrl_RGB565.hsync) begin
          //bram latency 2 + reg 1
                      if(img2_out[u_top.u_cnn_top.u_LcdCtrl_RGB565.oRamRdAddr-3] != oRGBCom) begin
                            flag = 1;// ~~compare failed
                            $stop;
                        end
                    end
                    else begin // left 2 compare
                        if(u_top.u_cnn_top.u_LcdCtrl_RGB565.oRamRdAddr == 130560) begin
                            if(rCnt >= 1) begin
                                if(img2_out[u_top.u_cnn_top.u_LcdCtrl_RGB565.oRamRdAddr-rCnt]
                != oRGBCom) begin
                                    flag = 1;// ~~compare failed
                                    $stop;
                                end
                                rCnt <= rCnt - 1;
                            end
                        end
                    end
                end
            endcase
        end
    end
```

위의 코드는 시나리오 2 를 진행하면서 나오는 LCD 의 RGB 값들의 실제 golden 값과 맞는 체크하기 위한 block 이다. Hsync 에 맞춰서 모든 픽셀 값들을 비교하고 정상 동작을 확인하였다.

```
TB Succeed
$stop called at time : 35747585 ns : File "C:/Users/user/Desktop/AdvancedProject/hw16_realfinal/cam_simul/testbench/tb_camera_inbuf.v" Line 247
run: Time (s): cpu = 00:00:42 ; elapsed = 00:00:40 . Memory (MB): peak = 1400.176 ; gain = 0.000
```

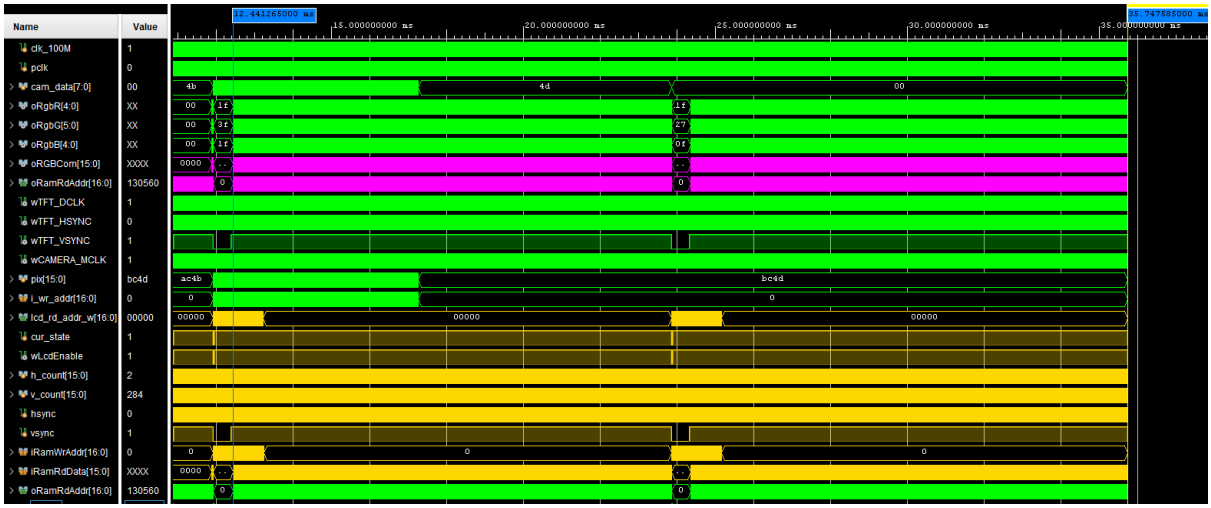


Figure 28 <시나리오 2 결과>

Vivado Block Design

Vivado IP Integrator를 통해 구성한 전체 시스템의 블록 다이어그램은 아래 그림11과 같다. Zynq UltraScale+ MPSoC를 중심으로 AXI Interconnect, AXI GPIO, AXI IIC 등의 주변 장치 IP가 연결되었으며, 커스텀 IP로 제작된 top 모듈이 시스템에 통합되었다.

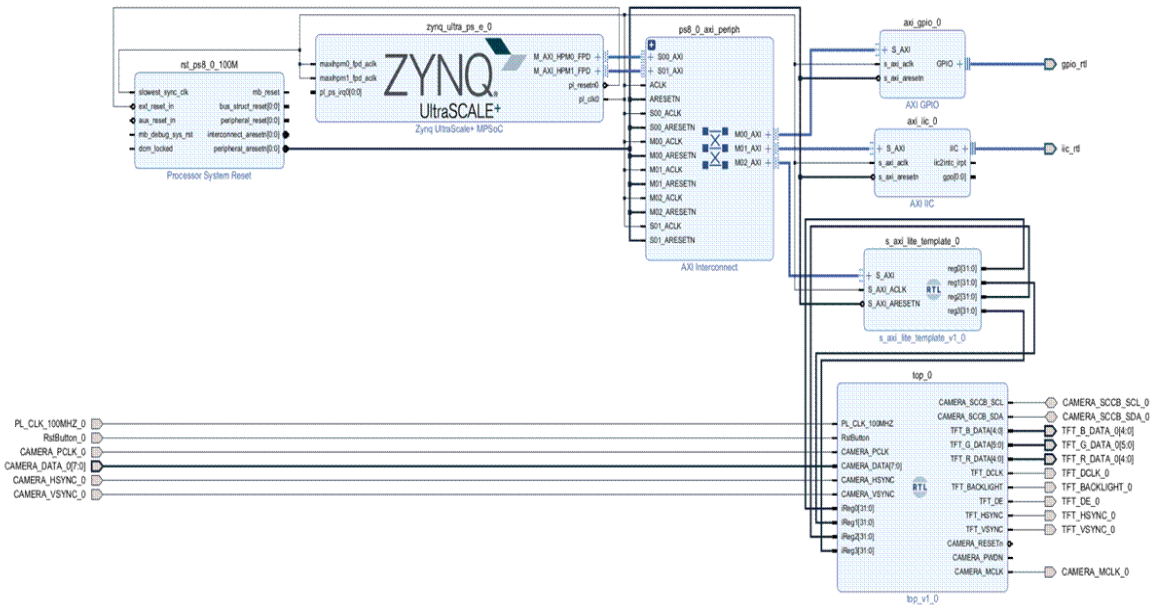


Figure 29 <Vivado Block Design>

Resource Utilization

합성 및 구현 완료 후 FPGA 리소스 점유율을 분석한 결과는 다음 **그림 12** 와 같다. 본 설계에서 가장 핵심적인 리소스는 Block RAM (BRAM)으로, 영상 데이터 저장을 위한 이중 버퍼(Double Buffer)와 라인 버퍼(Line Buffer) 구현으로 인해 전체 가용량의 **83.33%**를 사용하였다. 반면, 로직 연산을 담당하는 LUT 는 **31.04%**, Flip-Flop 은 **13.87%**를 사용하였다.

Resource	Utilization	Available	Utilization %
LUT	21900	70560	31.04
LUTRAM	189	28800	0.66
FF	19570	141120	13.87
BRAM	180	216	83.33
IO	39	82	47.56

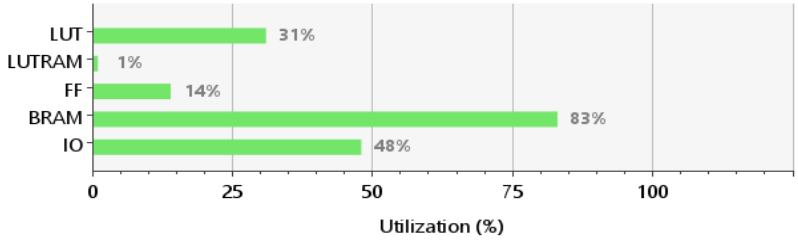


Figure 30 <Post-Implementation Utilization>

Timing

시스템 메인 Clock 인 100MHz (10ns) 제약 조건 하에서 타이밍 분석을 수행하였다. 분석 결과, Setup Time 의 WNS(Worst Negative Slack)는 **0.417ns** 로 양수(+) 값을 기록하였다. 이는 데이터가 클럭 엣지보다 약 0.4ns 먼저 도착하여 안정적으로 래치됨을 의미하며, TNS(Total Negative Slack) 또한 **0.000ns** 로 모든 타이밍 경로(Failing Endpoints: 0)에서 위배 사항(Violation) 없이 타이밍 제약 조건이 충족되었음을 확인하였다.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.417 ns	Worst Hold Slack (WHS):	0.016 ns	Worst Pulse Width Slack (WPWS):	3.500 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:	45249	Total Number of Endpoints:	45249	Total Number of Endpoints:	19829

All user specified timing constraints are met.

Figure 31 <Timing Report>

Power

구현된 디자인의 예상 소비 전력(On-Chip Power) 분석 결과는 총 **2.204W** 이다. 전력 소비의 대부분은 PS(Processing System) 영역에서 **1.620W(약 85%)**를 차지하였으며, CNN 가속기가 포함된 PL(Programmable Logic) 영역은 **0.218W** 로 **전체 전력의 약 10%** 수준으로 매우 효율적으로 동작함을 확인하였다. Dynamic Power 는 **1.889W**, Static Power 는 **0.315W** 로 분석되었다.

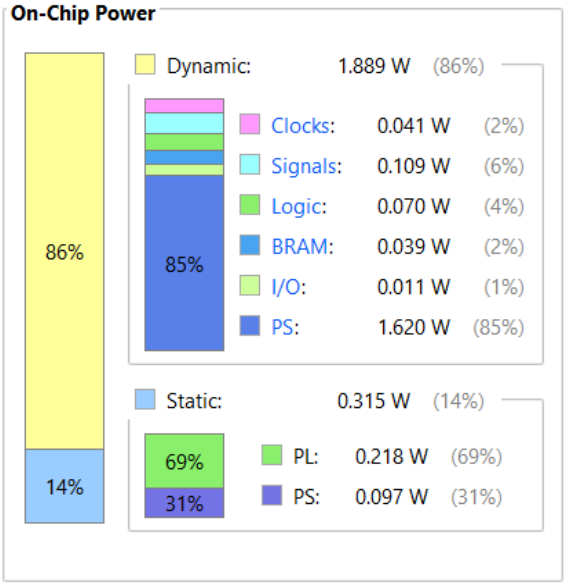


Figure 32 <On-Chip Power Report>

Register map

	NAME	Offset	Range	Field	Reset	Access	Description
B	CNN(Cipher only)	0xA002_0000	0xFFFF				
S	Filter control						
R	filterMode	0x0000	[1:0]		0x0	R/W	Filter mode(0,1,2,3)
F			[1:0]	iReg0	0x0		(0,1,2) : already define. 3: user define mode
R	iiReg1	0x0004	[31:0]		0x0	R/W	Filter value(3,2,1,0)
F			[31:0]	iReg1	0x0		
R	iReg2	0x0008	[31:0]		0x0	R/W	Filter value(7,6,5,4)
F			[31:0]	iReg2	0x0		
R	iReg3	0x000C	[31:0]		0x0	R/W	Filter value(8)
F			[31:0]	iReg3	0x0		

Figure 33 <Register Map>

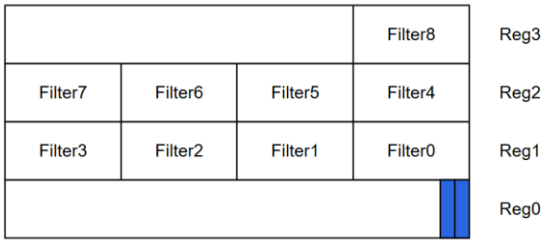


Figure 34 <AXI Register 설정>

Register Map 은 다음과 같이 구성하였다. Reg0 에 2 비트를 활용하여 mode 를 구분하여 전달한다. 추후 모드가 늘어날 수 있도록 나머지 공간을 예약했다. Reg1 ~ Reg3 은 mode3 인 사용자 정의 모드의 값을 전달하기 위해 사용되며 3*3 의 각각 8 비트 총 72 비트가 전달된다.

FPGA 동작 결과

PS(SW)에서 입력한 제어 명령이 PL(HW)에 실시간으로 반영되는지를 검증한다. 터미널(UART)을 통해 필터 커널을 설정하고, 그 즉시 FPGA 보드에 연결된 LCD 화면이 어떻게 변화하는지 비교 분석하였다. 전체 동작 영상은 첨부한 동영상을 통해 확인할 수 있다.

System Initialization

시스템 전원 인가 시 FSBL 부팅이 완료되고, I2C 인터페이스를 통해 OV5640 카메라 센서 초기화(111 개 레지스터 설정)가 정상적으로 수행됨을 터미널 로그를 통해 확인하였다.

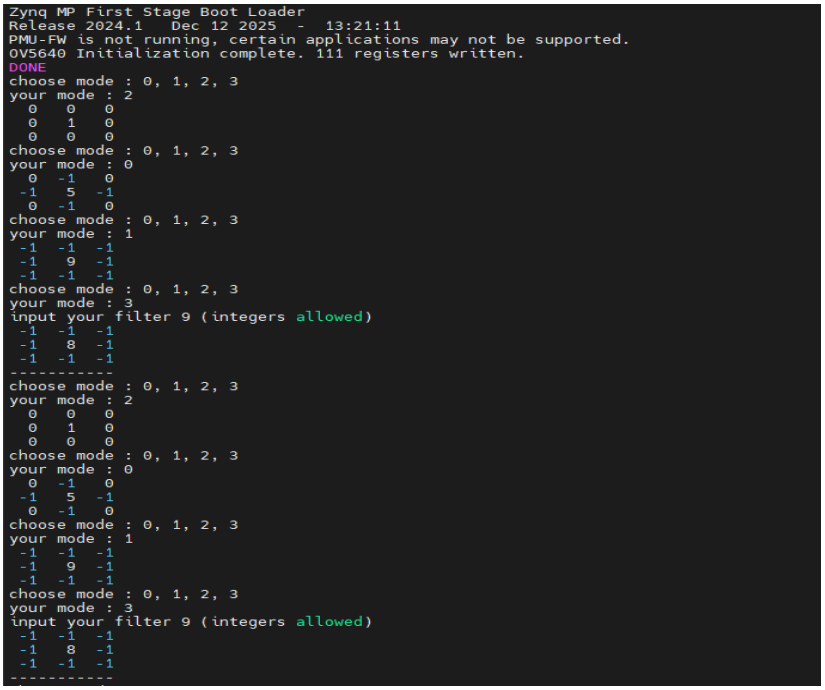


Figure 35 <시스템 부팅 로그 및 필터모드 선택 메뉴>

C-code Driver

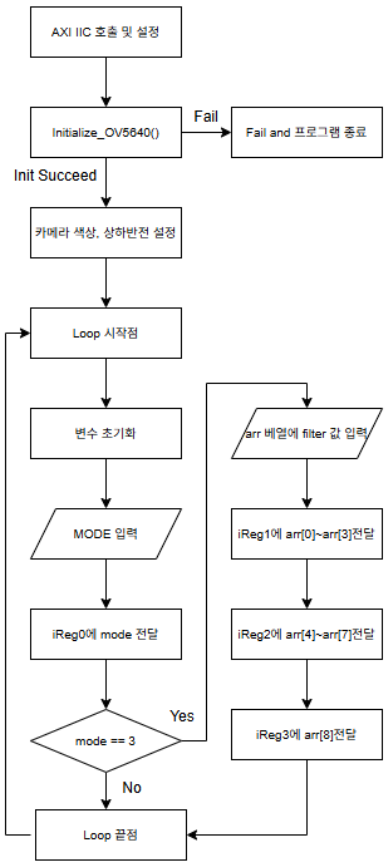


Figure 36<Code Diagram>

다음은 C-Driver의 순서도를 나타낸 것이다. Vivado에서 제공하는 AXI I2C를 통해 카메라를 초기화를 우선 진행한다. 만약 초기화에 실패하면 프로그램이 에러메시지와 함께 종료되고 초기화에 성공하면 카메라의 설정을 마치고 Loop에 진입한다. Loop는 uart를 통해 mode를 입력받고 만약 mode3인 사용자 정의 모드이면 추가로 필터값 9개를 정수로 받고 PL에 전달하게 된다.

Preset Filter Mode Switching Verification

시스템 부팅 후 기본 제공되는 필터 모드(Mode 0~2)를 변경하며 하드웨어 가속기의 동작을 확인했다. 또한 단순한 모드 선택을 넘어, 사용자가 직접 필터 계수를 입력하여 하드웨어를 실시간으로 재구성(Reconfiguration)할 수 있는지 검증하였다.

```
choose mode : 0, 1, 2, 3
your mode : 0
  0  -1   0
 -1   5  -1
  0  -1   0
```

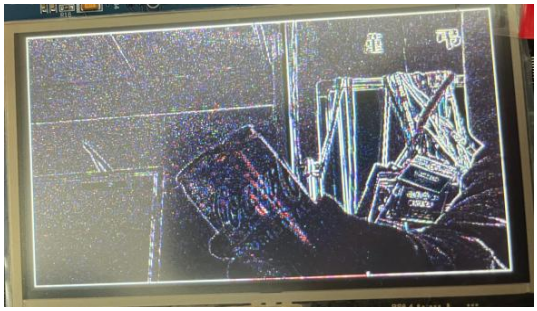
Figure37 <mode0 Sharpen filter 적용>

```
choose mode : 0, 1, 2, 3
your mode : 1
 -1  -1  -1
 -1   9  -1
 -1  -1  -1
```

Figure38 <mode1 Very Sharpen filter 적용>

```
choose mode : 0, 1, 2, 3
your mode : 2
  0   0   0
  0   1   0
  0   0   0
```

Figure39 <mode2 Bypass filter 적용>



```
choose mode : 0, 1, 2, 3
your mode : 3
input your filter 9 (integers allowed)
 -1  -1  -1
 -1   8  -1
 -1  -1  -1
```

Figure40 <mode3 사용자 정의 edge detect filter 적용>

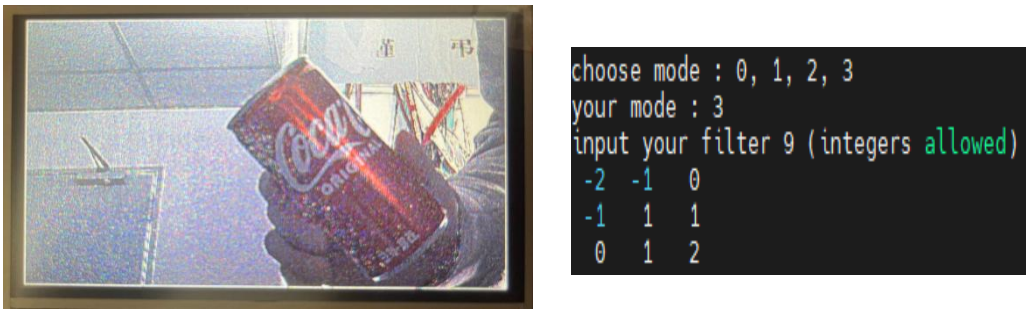


Figure41 <mode3 사용자 정의 emboss filter 적용>

2.5 설계 이전 예상 구조 와 최종 설계 구조의 차이점 및 원인

아래의 두 표는 설계 이전 예상 구조의 Operation Time 을 비교하여 어느 지점에서 병목이 발생하고 전체 System 의 성능 즉, Performance 가 어떻게 되는지 나타낸 표이다.

SW task name	Function	Operation time	Portion
Camera interface	1) Camera initialization through I2C(SCCB) 2) Camera data(RGB565, 16bit) to IBuf write	**6.00ms**	4.95%
IBuf Read	1) RGB565 & 16bit read 2) RGB888 & 24bit data conversion	**80.0ms**	66.01%
Convolution	1) 3x3 & 8bit integer kernel 2) RGB888 & 24bit & 3-channel data 3) 0 ~ 255 ReLU	**13.0ms**	10.73%
OutBuf Write	1) RGB888 & 24bit to RGB565 & 16bit data conversion 2) OBuf write	**10ms**	8.25%
LCD Interface	1) OBuf read 2) TFT-LCD interface	**12.20ms**	10.07%

<표 1 설계 전 예상 구조 Operation Time>

위 표를 보면 알 수 있듯이 Ibuf_Read 부분이 병목임을 알 수 있다. 이는 당연하게도 설계 예상 전 구조에서는 전체 시스템을 12.5Mhz 로 구동 시켰을 뿐만 아니라 최소한의 Resource 를 사용하기 위해 9 개의 MAC 과 1 개의 RELU 연산기가 사용되었고 이는 한 Window 에 대하여 Convolution 연산과 Activation 연산이 완료되기 전까지는 새로운 값을 읽어오지 못하는 현상을 만들어냈다 즉, 다시 말해 BRAM 에서 Pixel 의 값을 가져오는 시간을 비약적이게 증가시켰다. 이는 완벽한 Pipelining 을 구현하였더라도 8.25 FPS 의 성능 밖에 되지 못하게 만드는 원인이 되었다. 낮은 성능은 Fpga 상에서 동영상의 끊김을 발생시켰다. 따라서 실시간 카메라 시스템에서는 자원 절약보다 처리 성능 확보가 우선되어야 한다고 판단하였고. 이에 대한 해결책으로 최종 설계

구조에는 Line Buffer 구조와 27 개의 MAC 배치를 도입하여 1-Cycle Throughput 을 만들었고 이에 그치지 않고 Camera-Interface 의 Operation Time 을 없애기 위해 Double Buffering 을 사용하여 연산기가 쉴 새 없이 바로바로 연산이 되도록 하였다. 마지막으로는 시스템의 병목인 TFT-LCD 가 읽는 속도보다 써지는 속도가 훨씬 빠르다는 점을 활용해 연산 시간을 숨기는 Latency Hiding(연산 중첩) 기법을 적용하여 시스템 성능을 개선하였다.

SW task name	Function	Operation time	Portion
Camera interface	1) Camera initialization through I2C(SCCB) 2) Camera data(RGB565, 16bit) to IBuf write	**6.00ms -> 0ms**	0%
IBuf Read	1) RGB565 & 16bit read 2) RGB888 & 24bit data conversion	**1.04ms -> 0ms**	0%
Convolution	1) 3x3 & 8bit integer kernel 2) RGB888 & 24bit & 3-channel data 3) 0 ~ 255 ReLU	**1.04ms -> 0ms**	0%
OutBuf Write	1) RGB888 & 24bit to RGB565 & 16bit data conversion 2) OBuf write	**1.04ms -> 0ms**	0%
LCD Interface	1) OBuf read 2) TFT-LCD interface	**12.20ms**	100%

<표2 최종 설계 구조 Operation Time>

결론적으로 위 표에서 확인할 수 있듯이, 전체 연산과 데이터 전송(Data Transfer) 과정을 LCD 인터페이스 동작 시간 내에 Overlap 시켜 처리 시간을 효과적으로 줄일 수 있었다. 이를 통해 시스템은 81.97 FPS라는 기존 대비 9.93배 라는 탁월한 성능을 얻을 수 있었다.

3. 본론

3.1 Futher_Work

SW 개선

현재 구현된 시스템은 소프트웨어(SW)를 통해 단순히 필터링 기능을 활성화하거나 비활성화하는 기초적인 제어 수준에 머물러 있다. 향후에는 FPGA 내부의 레지스터 맵(Register Map)을 확장하여, 소프트웨어가 하드웨어의 동작을 보다 세밀하고 유연하게 제어할 수 있도록 개선할 것이다. 특히 화면 분할과 같은 고도화된 기능을 구현하여, 원본 영상과 신경망을 거친 처리 영상(Processed Image)을 좌우 혹은 4 분할로 동시에 출력하는 등 카메라의 Interface 를 SW 로 처리할 수 있게 만드는 것이 효율적일 것이다.

실시간 INTERFACE 도입

현재 설계된 시스템은 FPGA 내부의 BRAM(Block RAM)을 프레임 버퍼로 사용하여 480x272 해상도를 처리하고 있다. 이는 데이터 접근 속도 측면에서는 유리하지만, FPGA 내부 자원의 물리적 한계로 인해 고해상도 영상 처리로 확장하는 데에는 구조적 제약이 따른다.

따라서 향후에는 외부 DDR 메모리(DDR SDRAM)를 메인 프레임 버퍼로 활용하는 아키텍처로 전환할 것이다. 이를 위해 MIG(Memory Interface Generator) 및 DMA 컨트롤러를 도입하여 내부 메모리 용량의 한계를 극복하고, 다양한 해상도에 유연하게 대응할 수 있는 확장성 높은 시스템을 구축할 수 있을 것이다.

3.2 느낀점

이번 프로젝트에서 Camera 로부터 480×272 해상도의 실시간 이미지를 입력받아, CNN 의 핵심 연산인 3×3 Convolution 과 ReLU 활성화 함수를 수행한 뒤 RGB565 포맷으로 변환하여 출력하는 전용 하드웨어 파이프라인을 설계하고 검증하였다.

특히 기존 설계에서 가장 큰 한계였던 낮은 FPS 문제를 인지하고, 이를 구조적 개선을 통해 해결하는 과정을 통해 입력 영상의 중복 데이터를 효율적으로 재사용하고 Throughput 을 늘리기 위해 LINE BUFFER 구조를 도입하고, 병목 구간이었던 Convolution 연산부에서 MAC 개수를 늘려 병렬 처리를 수행함으로써, 동일한 해상도에서 기존 대비 향상된 FPS 를 확보할 수 있었다. 이 과정에서 카메라 입력, CNN 연산, LCD 출력 각각의 처리 시간과 지연을 분석하여 시스템의 실제 병목 구간을 파악하고, 이를 기준으로 전체 SYSTEM 의 SYNC 를 맞추는 작업이 매우 까다로웠지만, 하드웨어 설계에서는 타이밍과 동기화가 먼저 기반이 되고 이를 바탕으로 리소스 최적화를 진행할 수 있다는 중요한 점 또한 알 수 있었다.

종합적으로 본 프로젝트를 통해 단순히 CNN 알고리즘을 이해하는 수준을 넘어, 제한된 자원과 타이밍 제약 하에서 어떻게 구조를 바꾸고 병렬화를 적용해야 실제 성능이 개선되는지를 경험적으로 학습할 수 있었고 또한 프레임 단위의 데이터 흐름을 고려하며 설계를 반복 수정하는 과정에서 시스템 관점에서 문제를 정의하고 해결책을 찾아가는 능력을 함양할 수 있었다.