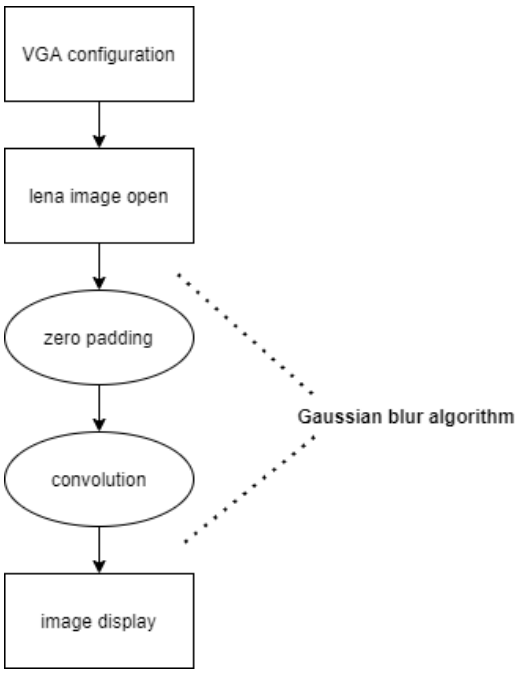
임베디드시스템 설계 및 실습 – 보고서

제출일: 2021-06-02; Lab 5
이름: 박관영(2016124099), 오한별(2018124120), 김서영(2017106007); 4조

✓ Assignment 1

개요



본 lab에서는 heterogeneous multi-processor SoC와 digital logic circuit에 대한 이해를 바탕으로 특정 application의 performance를 향상시키고자 accelerator를 구현하는 실험을 수행하였다. 지금까지의 lab은 DE1-SoC Computer라는 intel의 sample platform기반의 system에서 실험을 수행하였지만, 이번 lab의 주된 목표는 custom hardware logic을 활용하여 특정 application의 processing kernel을 가속하는 것이므로 전체적인 circuit architecture에 대한 고찰이 필요했다.

가속 대상 application으로 "Gaussian blur Algorithm"을 선택하였다. Gaussian blur algorithm이란 image processing algorithm중에 하나로써, Gaussian 형태의 weight를 가진 low pass filter kernel을 원본 image와 convolution 연산을 수행하여 고주파 성분들을 제거하는 algorithm이다. 그 결과 image는 blurring되는 효과를 얻게 된다.

Circuit architecture는 HPS와 PLL, Accelerator component로 이루어지고, Demo를 위한 intel university program의 IP인 VGA subsystem을 사용하였다.

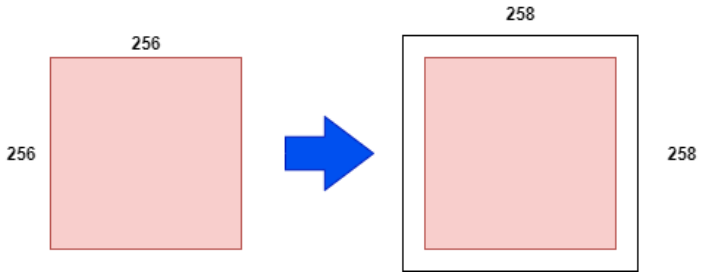
진행 내용

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/time.h>

#define LW_BRIDGE_SPAN 0x00005000
#define LW_BRIDGE_BASE 0xFF200000
```

```
#define FPGA_ONCHIP_SPAN 0x0003ffff          // pixel buffer는 on-chip memory영역에 mapping된다.

#define PIXEL_DMA_BASE 0x00000020            // pixel buffer controller의 시작 주소

#define ACC_BASE 0x00000040                  // Accelerator base address

#define SCREEN_WIDTH 320
#define SCREEN_HEIGHT 240

#define i_w 256                              // input feature spec
#define i_h 256

#define CLIP_8(a) (a>127?127:(a<-128?-128:a))   // clipping 연산, 8bit끼리의 곱셈 연산후에 다시 8bit로 clipping 하는 용도.
volatile int* pixel_ctrl_ptr;
volatile int pixel_buffer_start;
volatile int pixel_buf_virtual;

volatile int* conv_acc_base;

char image[i_w * i_h];
char output_image[i_w * i_h];            // blurring 결과 이미지
char padded_image[(i_w+2)*(i_h+2)];      // zero padding 결과

char mask[9] = {1,2,1,2,4,2,1,2,1};      // gaussian filter mask

void plot_pixel(int x, int y, char line_color){                    // 8bit grayscale pixel을 화면에 display.
        *(char*)(pixel_buf_virtual + (y<<9) + (x)) = line_color;
}

void zero_padding(void){ // zero padding  (256*256) ➔ (258*258)  convolution연산 후에도 이미지 size가 변하지 않도록 함.

    int i;int j;
    for(i=0;i<i_h * i_w;i++) {
        padded_image[i] = 0;
    }

    for(i=0;i<i_h;i++){
        for(j=0;j<i_w;j++){
            padded_image[(j+1) + i_w*(i+1)] = image[j + i_w*i];
        }
    }
}

void convolution(void){              // 전체 이미지 한 장에 대한 convolution 연산
    int i;int j;int k;int m;
        short tmp;
    for(i=0;i<i_h;i++){
```

```
        for(j=0;j<i_w;j++){

            conv_1pix(j + i_w*i);

        }
    }
}

void conv_1pix(int start_pix){      // 결과 이미지 1 pixel에 대한 conv 연산
    int k; int m; short tmp;
    for(k=0;k<3;k++){
        for(m=0;m<3;m++){
          tmp += (mask[3 * k + m] * ₩                  // mask와 image 간의 element wise 곱셈 후 전체 합
              padded_image[3 * k + m  + start_pix]);
        }
    }

    output_image[start_pix] = CLIP_8(tmp>>4);      // kernel의 값의 합으로 pixel intensity를 나눔
                                                   // ➔ gaussian filter는 모든 값이 1 이하 이기 때문
                                                   // 이후 8bit로 truncate ==> grayscale system에 display될 수 있도록 함
    tmp=0;
}

int main(void){
        int fd;
        void* lw_virtual;

        fd = open("/dev/mem", (O_RDWR | O_SYNC));
        lw_virtual = mmap(NULL, LW_BRIDGE_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, LW_BRIDGE_BASE);

        pixel_ctrl_ptr = (volatile int*)(lw_virtual + PIXEL_DMA_BASE);          // pixel control buffer base를 virtual address로 mapping
        pixel_buffer_start = *pixel_ctrl_ptr;                                   // pixel control buffer의 값을 읽는다(실제 buffer의 address)
                                                                                // 기본적으로 onchip memory에 할당되어 있음
        printf("%x", pixel_buffer_start);

        pixel_buf_virtual = mmap(NULL, FPGA_ONCHIP_SPAN, (PROT_READ | PROT_WRITE), ₩
        MAP_SHARED, fd, pixel_buffer_start);                                    // pixel buffer의 주소를 virtual address로 mapping


        FILE* fp;                                                               // blurring할 image open.
        fp = fopen("lena_256.bmp", "rb");if(fp==NULL){
                printf("cannot open file.₩n");
                exit(1);
        }
        fread(image, sizeof(char), 256*256, fp);

        int i;int j;
```

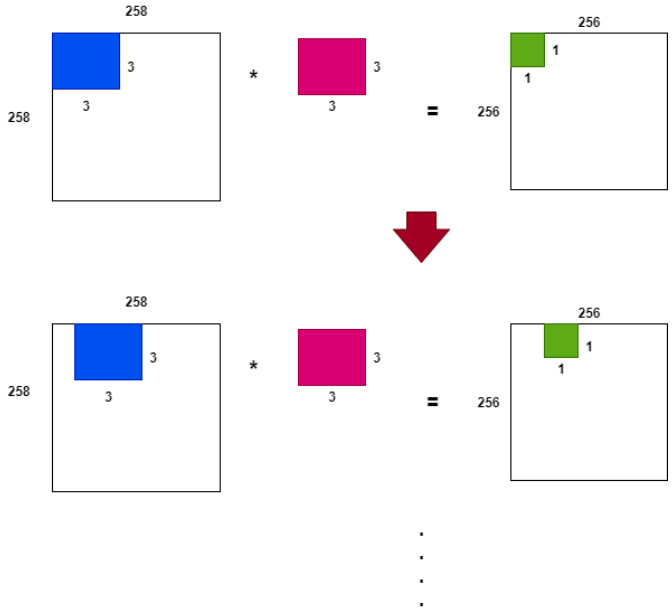
```
        for(i =100;i<220;i++){
                for(j =100;j<220;j++){
                        plot_pixel(j-100, i-100, image[256*(256-i) + j]);
                }
        }

        zero_padding();          // zero padding 256*256 ➔ 258*258

        convolution();           // convolution (3*3) * (258*258) ➔ (256*256)

        for(i =100;i<220;i++){
                for(j=100;j<220;j++){
                        plot_pixel(j+28, i-100, output_image[256*(256-i) + j]);
                }
        }

        //clear_screen();
        munmap(lw_virtual, LW_BRIDGE_BASE);
        munmap(onchip_virtual, pixel_buffer_start);
        close(fd);

        return 0;
}
```




```
void zero_padding(void){

    int i;int j;
    for(i=0;i<i_h * i_w;i++) {
        padded_image[i] = 0;
    }

    for(i=0;i<i_h;i++){
        for(j=0;j<i_w;j++){
            padded_image[(j+1) + i_w*(i+1)] = image[j + i_w*i];
        }
    }
}
```

➔ zero padding

➔ 다음과 같이 이미지 주변에 일정 간격의 0 pixel을 삽입하는 기법이다. Convolution 연산의 특성상 image의 크기를 줄이게 되므로 동일한 image size를 갖도록 하는 용도로 사용된다.




```
void conv_1pix(int start_pix){
    int k; int m; short tmp;
    for(k=0;k<3;k++){
        for(m=0;m<3;m++){
            tmp += (mask[3 * k + m] * \
                padded_image[3 * k + m  + start_pix]);
        }
    }
    output_image[start_pix] = CLIP_8(tmp>>4);

    tmp=0;
}
```

➔ image processing algorithm에서 자주 사용되는 convolution 연산이다. 수학적인 convolution과는 약간 의미가 다르지만, kernel을 한 칸 씩 옮겨가며 weighted sum을 수행한다는 관점에서 유사하다. 직관적인 그림으로 나타내면 위의 그림과 같이 image에서 kernel 만큼의 픽셀과 kernel이 element wise 곱셈을 한 후 모든 값을 합하여 결과 이미지의 한 픽셀이 된다. Convolution을 C로 구현하면 다음과 같이 이중 for loop을 통해서 구현할 수 있다. 한 픽셀이 아닌 모든 픽셀에 대해서 연산을 해야 하므로 결국엔 4중 for loop으로 구현된다.

```
#define CLIP_8(a) (a>127?127:(a<-128?-128:a))
```

➔ Cilp 연산이다. Vga IP를 8bit grayscale로 설정하였으므로 모든 값은 8bit로 연산이 진행되야 한다. 그 과정에서 overflow를 처리하기 위한 연산이다. 연산 결과가 8bit 이상이 되면 앞의 값이 truncate 되어 올바르지 못한 값이 되기 때문에 다음과 같은 quantizing 과정이 필요하다.

```
pixel_ctrl_ptr = (volatile int*)(lw_virtual + PIXEL_DMA_BASE);
pixel_buffer_start = *pixel_ctrl_ptr;
```

```
pixel_buf_virtual = mmap(NULL, FPGA_ONCHIP_SPAN, (PROT_READ | PROT_WRITE),
MAP_SHARED, fd, pixel_buffer_start);
```

➔ virtual mapping된 LW_BRIDGE를 통해 pixel buffer controller에 접근한다. pixel buffer controller에는 실제 pixel 값이 쓰여질 buffer의 주소를 저장하고 있는데, 이는 기본적으로 on-chip memory 영역으로 설정되어 있다. 따라서 on-chip memory의 주소 또한 virtual address mapping을 하여 화면에 이미지를 display할 수 있다.

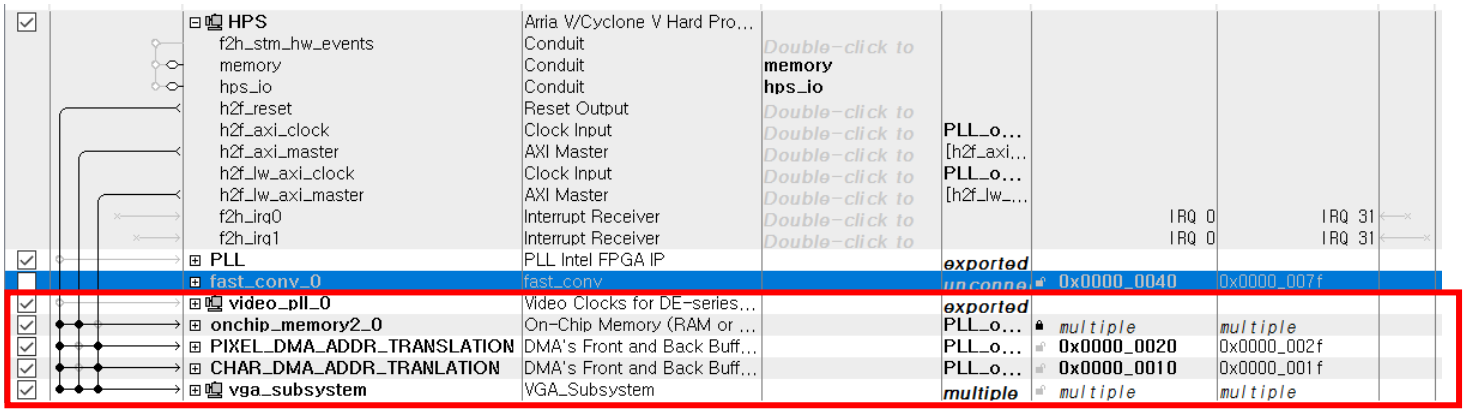
```
void plot_pixel(int x, int y, char line_color){
        *(char*)(pixel_buf_virtual + (y<<9) + (x)) = line_color;
}
```

➔ mapping된 pixel buffer의 주소에 8bit grayscale 값을 display하는 함수이다. De1-SoC에서는 9bit의 y좌표, 8bit의 x좌표를 concatenate하여 화면에 display할 위치를 정할 수 있고, pixel buffer의 주소에 이를 더해주어 실제 화면에 display를 할 수 있다.

임베디드시스템 설계 및 실습 – 보고서

진행 결과

1. circuit architecture (Processor + Demo IP)



➔ pure software program을 수행하기 위한 circuit architecture이다. Red box 부분은 Demo를 위한 intel university program의 IP이다. 다음과 같이 HPS, PLL, Demo IP로 이루어져 있는 것을 확인할 수 있다.

2. Time measure

```
gettimeofday(&start1, NULL);

zero_padding();

gettimeofday(&end1, NULL);
elapsed1 = end1.tv_usec - start1.tv_usec;
printf("\npadding exe time: %ld\n", elapsed1);


gettimeofday(&start2, NULL);

convolution();

gettimeofday(&end2, NULL);
elapsed2 = end2.tv_usec - start2.tv_usec;
printf("\nconv exe time: %ld\n", elapsed2);
```

```
padding exe time: 2472

conv exe time: 49044
```

➔ pure software 환경에서 algorithm 구성 요소들에 대한 수행시간을 측정한 결과이다. Gettimeofday 함수를 통해 us 단위로 시간을 측정하였고, 그 결과 convolution이 processing kernel임을 확인할 수 있었다.

3. pin assignment(추가적인 Demo IP)

```
.video_pll_0_ref_clk_clk          (CLOCK2_50),
.video_pll_0_ref_reset_reset      (1'b0),

.vga_CLK                          (VGA_CLK),
.vga_HS                           (VGA_HS),
.vga_VS                           (VGA_VS),
.vga_BLANK                        (VGA_BLANK_N),
.vga_SYNC                         (VGA_SYNC_N),
.vga_R                            (VGA_R),
.vga_G                            (VGA_G),
.vga_B                            (VGA_B),
```

➔ Demo IP에 대한 pin assignment

4. Program the fpga part & compile

KAU_EMBEDDED_SYS/synthesis/KAU_EMBEDDED_SYS.qip
KAU_EMBEDDED_TOP.v
KAU_EMBEDDED_TOP.sdc

```
root@de1soclinux:~/sys_test# ./program_fpga ./fast_conv.rbf
2+1 records in
2+1 records out
2870504 bytes (2.9 MB) copied, 0.140015 s, 20.5 MB/s
FPGA Programmed with ./fast_conv.rbf
```

➔ platform designer를 통해 생성된 .qip 파일과 .v 파일을 모두 compile하고 생성된 .sof 파일을 .rbf 파일로 변환하여 DE1-SoC 보드 상에 전체 system을 올릴 수 있다.

```
root@de1soclinux:~/sys_test# gcc last_main.c -o qq
last_main.c:85:6: warning: conflicting types for 'conv_1pix' [enabled by default]
last_main.c:79:13: note: previous implicit declaration of 'conv_1pix' was here
last_main.c: In function 'main':
last_main.c:111:27: warning: assignment makes integer from pointer without a cast
t]
last_main.c:117:17: warning: incompatible implicit declaration of built-in functic
by default]
root@de1soclinux:~/sys_test# ./qq
```

➔ C main 파일을 compile 후 실행한다.

5. Demo

➔ 원본 이미지와 비교했을 때 blur효과를 가진 이미지가 vga에 display되는 것을 확인할 수 있다. 이 때, 8bit pixel만 display하는 system이므로 일부 pixel이 blur 효과 이외에 pixel의 최대치로 saturation되는 현상이 발생했다. 그 결과 다음과 같이 blur효과 이외에도 전체적으로 밝아지는 현상이 발생한 것을 확인할 수 있었다.

결과 분석 및 팀원 간 토의 사항

HPS part와 FPGA part로 구성된 multi-processor 기반의 SoC인 DE1-SoC 환경에서 platform designer를 통해 custom circuit architecture를 구현하여 program을 수행해 보았다. 지금까지는 완성된 DE1-SoC Computer라는 sample architecture에서 프로그램을

구현하고, 실행했다면 이번 lab에서는 program에 필요한 IP를 직접 연결하거나 혹은 직접 만들어서 architecture를 구현하는 과정이 있었다.

Image processing algorithm 중 하나인 Gaussian blur algorithm을 pure software 환경에서 구현해 보았다. Image processing 특성상 matrix 연산이 많으므로 processing kernel이 존재할 것이라 생각하여 가속 대상으로 선정하였다. 해당 algorithm의 경우 matrix 연산인 convolution이 processing kernel이라는 것을 확인할 수 있었고, 해당 연산을 가속대상으로 정할 수 있었다. Assignment1 에서는 pure software 환경에서 program을 구현하고 실행하였으므로, HPS와 PLL IP를 사용하여 모든 program을 수행할 수 있었다.

✓ Assignment 2

개요

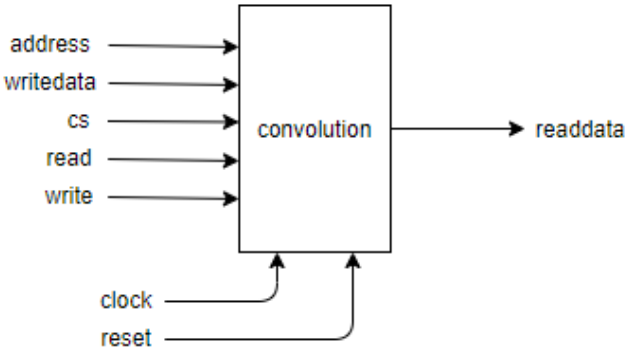
image

a	b	c
d	e	f
g	h	i

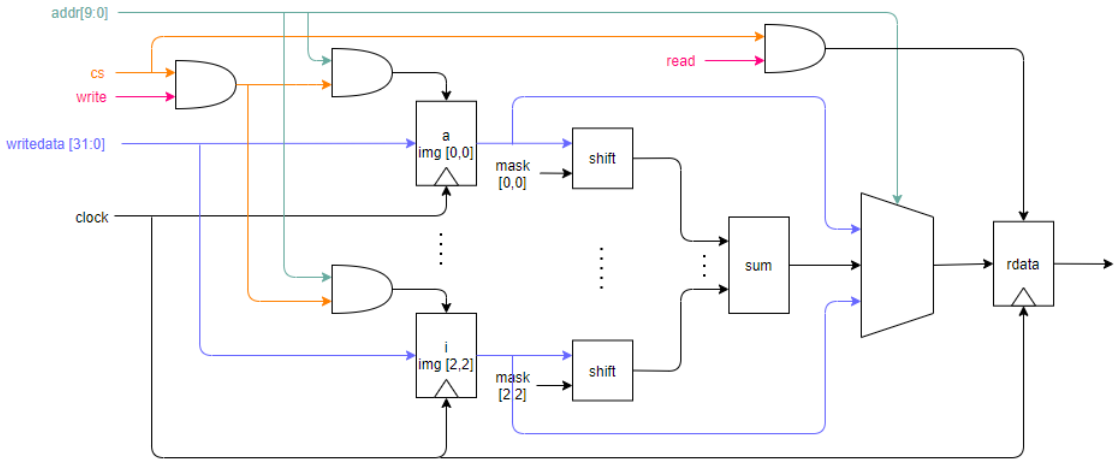
mask

[0,0]	[0,1]	[0,2]
[1,0]	[1,1]	[1,2]
[2,0]	[2,1]	[2,2]

Assignment2 에서는 image processing을 가속화하는 HW accelerator를 디자인한다. Image에 Gaussian blur 처리를 HW 가속기를 사용하여 진행하며, 해당 가속기는 3x3 Gaussian mask와 2로 padding된 영상의 convolution 연산을 수행한다. Gaussian mask는 {{1,2,1}, {2,4,2},{1,2,1}} 로 지정하고 padding 처리가 된 영상의 3x3부분 즉, 9개의 픽셀 값에 mask를 곱한 후 9개의 값을 모두 더하는 연산을 수행한다. 이러한 convolution 연산을 수행하는 HW architecture를 디자인하고 이를 Verilog HDL로 기술한다.



Convolution 연산을 수행하는 모듈의 I/O를 간략하게 나타내면 왼쪽과 같고, Verilog HDL로 architecture를 기술할 때 assignment1과 마찬가지로 Avalon MM 를 고려하여 작성하며, 각각의 signal에 따라 data를 read/write 할 수 있도록 한다. Signal의 종류는 Quartus의 Platform designer에서 지정한다. 그 후 Generate 된 platform기반의 sof 파일을 rbf파일로 변환한 뒤 main.c 에서 해당 연산이 제대로 수행되는지 확인한다.



Convolution 연산을 수행하는 모듈 내부의 구조는 위와 같다. Addr signal에 따라 image 픽셀 값을 9개의 register에 각각 저장하고 mask 값과 곱한 후에 shift 연산을 통해 지정한 gaussian mask의 총합인 16으로 나눠준다. 그 후 9개의 값을 모두 합하고 addr signal

에 따라 rdata에 저장되도록 한다.

진행 내용

```
//convolution 연산을 수행하는 module
module conv(
        //Avalon MM I/F
        input   wire    [3:0]   addr,
        output  reg     [31:0]  rdata,
        input   wire    [31:0]  wdata,
        input   wire            cs,
        input   wire            read,
        input   wire            write,

        //Avlaon clock & reset I/F
        input   wire            clk,
        input   wire            rst
);

reg [7:0] a, b, c, d, e, f, g, h, i; // img pixel values
wire [15:0] sum; // sum of pixel x mask

//Gaussian mask와 img픽셀을 convolution한 결과를 계산한 후 총 값 filter의 총합인 16으로 나눈다.
assign sum = (a + b * 2 + c + d * 2 + e * 4 + f * 2 + g + h * 2 + i) >> 4;

// mask
// 1 2 1
// 2 4 2
// 1 2 1

// assign image pixel values to each reg
always @ (posedge clk) begin
  if (rst) begin
    a <= 8'b0;   e <= 8'b0;    i <= 8'b0;
    b <= 8'b0;    f <= 8'b0;
    c <= 8'b0;    g <= 8'b0;
    d <= 8'b0;  h <= 8'b0;
  end
  else if (cs & write) begin
    if (addr == 4'd0) a <= wdata[7:0];
    else if (addr == 4'd1) b <= wdata[7:0];
    else if (addr == 4'd2) c <= wdata[7:0];
    else if (addr == 4'd3) d <= wdata[7:0];
    else if (addr == 4'd4) e <= wdata[7:0];
```

```
    else if (addr == 4'd5) f <= wdata[7:0];
    else if (addr == 4'd6) g <= wdata[7:0];
    else if (addr == 4'd7) h <= wdata[7:0];
    else if (addr == 4'd8) i <= wdata[7:0];
  end
end

always @ (posedge clk) begin
  if(cs & read) begin
    case(addr) //addr에 따라 reg에 data를 read한다.
      4'd0: rdata <= {24'd0, a}; 4'd5: rdata <= {24'd0, f};
      4'd1: rdata <= {24'd0, b}; 4'd6: rdata <= {24'd0, g};
      4'd2: rdata <= {24'd0, c}; 4'd7: rdata <= {24'd0, h};
      4'd3: rdata <= {24'd0, d}; 4'd8: rdata <= {24'd0, i};
      4'd4: rdata <= {24'd0, e}; 4'd9: rdata <= {16'd0, sum};
      default: rdata <= 32'd0;
    endcase
  end
end

endmodule
```

```
//Platform을 검증하기 위한 main.c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <time.h>

#define LW_BRIDGE_SPAN 0x00005000
#define LW_BRIDGE_BASE 0xFF200000
#define i_w 256
#define i_h 256

#define CONV_BASE 0x00000040 //avalon slave address
static volatile int* conv;

//2로 padding한 image를 생성한다.
void zero_padding(void) {

        int i; int j;
        for (i = 0; i < i_h * i_w; i++) {
                padded_image[i] = 0;
        }

        for (i = 0; i < i_h; i++) {
                for (j = 0; j < i_w; j++) {
                        padded_image[(j + 1) + i_w * (i + 1)] = image[j + i_w * i];
```

```
                }
        }
}

//padding된 image 바탕으로  convolution을 계산한 후 결과값을 반환한다.
void fast_conv(void) {
        int x, y;
        for (x = 0; x < i_h; x++) {
                for (y = 0; y < i_w; y++) {
                        *conv = padded_image[(x) * (y)];
                        *(conv + 1) = padded_image[(x) * (y + 1)];
                        *(conv + 2) = padded_image[(x) * (y + 2)];
                        *(conv + 3) = padded_image[(x + 1) * (y)];
                        *(conv + 4) = padded_image[(x + 1) * (y + 1)];
                        *(conv + 5) = padded_image[(x + 1) * (y + 2)];
                        *(conv + 6) = padded_image[(x + 2) * (y)];
                        *(conv + 7) = padded_image[(x + 2) * (y + 1)];
                        *(conv + 8) = padded_image[(x + 2) * (y + 2)];
                        outputimage2[(x+1) * (y+1)] = *(conv + 9);
                }
        }
}

int main(void) {
        int fd;
        void* lw_virtual;
        int i; int j;
        double time1, time2;

        fd = open("/dev/mem", (O_RDWR | O_SYNC));
        lw_virtual = mmap(NULL, LW_BRIDGE_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, LW_BRIDGE_BASE);
        conv = (volatile int*)(lw_virtual + CONV_BASE);

        FILE* fp;
        fp = fopen("lena_256.bmp", "rb"); if (fp == NULL) {
                printf("cannot open file.₩n");
        }
        fread(image, sizeof(char), 256 * 256, fp);

        zero_padding();
        fast_conv();

        munmap(lw_virtual, LW_BRIDGE_BASE);
        close(fd);

        return 0;
}
```

1) 각각의 signal은 다음과 같은 의미를 가진다.

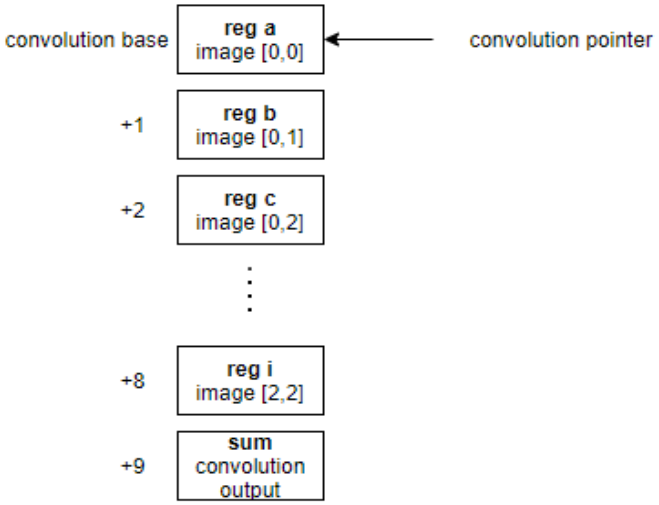
Addr	Address 타입으로 각각의 레지스터(a-i)에 image 픽셀 값을 저장할 수 있도록 한다.
Cs	Chip select 타입으로 이 signal이 assert되어 있지 않으면 다른 avalon salve의 input signal에 응답할 수 없다.
Write	Byteenable 타입으로 (a-i) register에 값을 쓸 수 있도록 addr와 and연산을 하여 enable시켜준다.
Writedata	Image 픽셀 값을 읽는다.

2) Conv.v에서의 addr signal과 main.c에서 pointer위치에 따른 값 저장의 상관 관계는 다음과 같다.

Conv.v	Main.c (fast_conv)

case(addr)
4'd0: rdata <= {24'd0, a}; 4'd5: rdata <= {24'd0, f};
4'd1: rdata <= {24'd0, b}; 4'd6: rdata <= {24'd0, g};
4'd2: rdata <= {24'd0, c}; 4'd7: rdata <= {24'd0, h};
4'd3: rdata <= {24'd0, d}; 4'd8: rdata <= {24'd0, i};
4'd4: rdata <= {24'd0, e}; 4'd9: rdata <= {16'd0, sum};
default: rdata <= 32'd0;
endcase
	*conv = padded_image[(x) * (y)];
	*(conv + 1) = padded_image[(x) * (y + 1)];
	*(conv + 2) = padded_image[(x) * (y + 2)];
	*(conv + 3) = padded_image[(x + 1) * (y)];
	*(conv + 4) = padded_image[(x + 1) * (y + 1)];
	*(conv + 5) = padded_image[(x + 1) * (y + 2)];
	*(conv + 6) = padded_image[(x + 2) * (y)];
	*(conv + 7) = padded_image[(x + 2) * (y + 1)];
	*(conv + 8) = padded_image[(x + 2) * (y + 2)];
	outputimage2[(x+1) * (y+1)] = *(conv + 9);

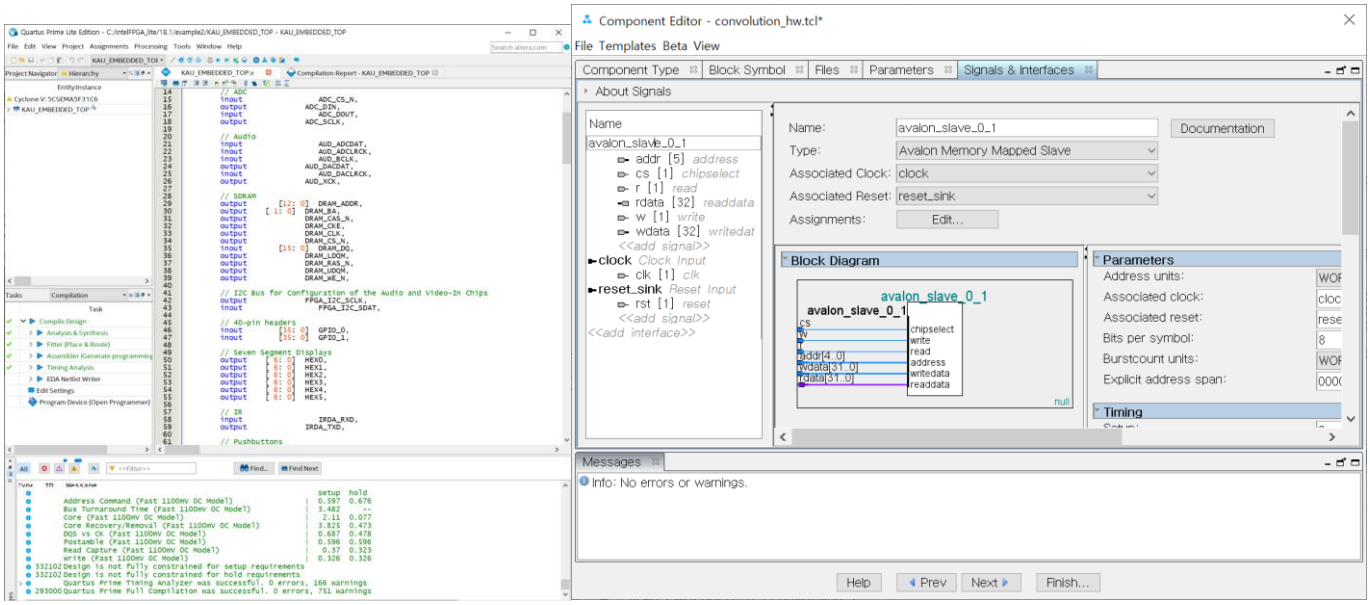
c에서의 pointer를 통해 verilog 에서 기술한 register의 값을 읽어올 수 있다.



임베디드시스템 설계 및 실습 – 보고서

진행 결과

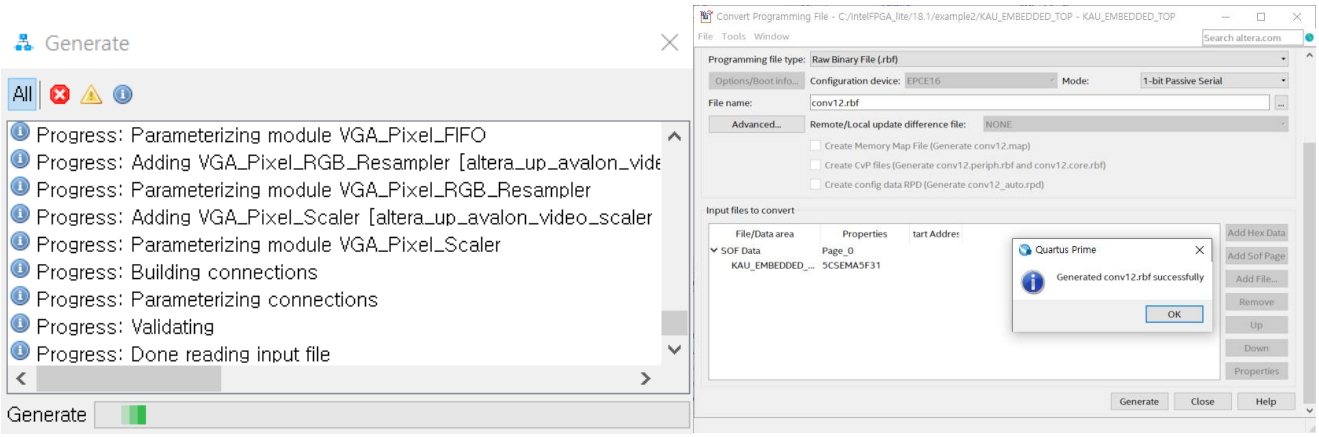
1. Generate platform



➔ 컴파일을 완료한 후 Platform designer를 통해 interface 각각의 signal type을 설정한다.

Use	Connections	Name	Description	Export	Clock	Base	End
☑		HPS	Arria V/Cyclone V Hard Pro...				
		f2h_stm_hw_events	Conduit	Double-click to			
		memory	Conduit	memory			
		hps_io	Conduit	hps_io			
		h2f_reset	Reset Output	Double-click to			
		h2f_axi_clock	Clock Input	Double-click to	PLL_o...		
		h2f_axi_master	AXI Master	Double-click to	[h2f_axi...		
		h2f_lw_axi_clock	Clock Input	Double-click to	PLL_o...		
		h2f_lw_axi_master	AXI Master	Double-click to	[h2f_lw_...		
		f2h_irq0	Interrupt Receiver	Double-click to		IRQ 0	IRQ 31
		f2h_irq1	Interrupt Receiver	Double-click to		IRQ 0	IRQ 31
☑		video_pll_0	Video Clocks for DE-series...		exported		
☑		onchip_memory2_0	On-Chip Memory (RAM or ...				
		s1	Avalon Memory Mapped Sl...	Double-click to	[clk1]	0x0800_0000	0x0803_ffff
		s2	Avalon Memory Mapped Sl...	Double-click to	[clk1]	0x0800_0000	0x0803_ffff
		clk1	Clock Input	Double-click to	PLL_o...		
		reset1	Reset Input	Double-click to	[clk1]		
☑		PLL	PLL Intel FPGA IP				
		refclk	Clock Input	system_pll_ref_clk	exported		
		reset	Reset Input	system_pll_ref_r...			
		outclk0	Clock Output	Double-click to	PLL_out...		
☑		PIXEL_DMA_ADDR_TRANSLATION	DMA's Front and Back Buff...		PLL_o...	0x0000_0020	0x0000_002f
☑		CHAR_DMA_ADDR_TRANLATION	DMA's Front and Back Buff...		PLL_o...	0x0000_0010	0x0000_001f
☑		vga_subsystem	VGA_Subsystem		multiple	multiple	multiple
☑		convolution_0	convolution				
		avalon_slave_0	Avalon Memory Mapped Sl...	Double-click to	[clock]	0x0000_0040	0x0000_007f
		clock	Clock Input	Double-click to	PLL_o...		
		reset_sink	Reset Input	Double-click to	[clock]		

➔ avalon_slave의 Base address를 0x40으로 지정한다. 이 주소를 offset으로 C file에서 pointer를 통해 register에 access할 수 있다.



2. Program the FPGA part with RBF

```
root@delsoclinux:~/lab5# ls
conv12.rbf  lab5_1.c  lena_256.bmp  program_fpga
```

➔ pc에 있던 file들을 sd카드로 복사하고, fat_partition에 저장된 conv12.rbf , lab5_1.c , lena_256.bmp file을 해당 디렉토리로 copy한다.

```
root@delsoclinux:~/lab5# cat program_fpga
#!/bin/bash

echo 0 > /sys/class/fpga-bridge/hps2fpga/enable
echo 0 > /sys/class/fpga-bridge/fpga2hps/enable
echo 0 > /sys/class/fpga-bridge/lwhps2fpga/enable
dd if=$1  of=/dev/fpga0 bs=1M
echo 1 > /sys/class/fpga-bridge/hps2fpga/enable
echo 1 > /sys/class/fpga-bridge/fpga2hps/enable
echo 1 > /sys/class/fpga-bridge/lwhps2fpga/enable
sync
echo "FPGA Programmed with $1"
```

```
root@delsoclinux:~/lab5# ./program_fpga ./conv12.rbf
2+1 records in
2+1 records out
2874488 bytes (2.9 MB) copied, 0.140007 s, 20.5 MB/s
FPGA Programmed with ./conv12.rbf
```

➔ program_fpga는 shell script로 FPGA 부분을 해당 rbf로 프로그래밍 할 수 있도록 한다. 본 설계에서는 convolution연산을 수행하는 hw인 conv12.rbf를 사용하여 FPGA 부분을 프로그래밍한다.

3. Compile / Execute

```
root@delsoclinux:~/lab5# gcc lab5_1.c -o conv
root@delsoclinux:~/lab5# ./conv
```

➔ gcc 컴파일러를 사용하여 user program을 compile & link 한 뒤 실행한다. 실제 이미지 픽셀은 input으로, register에 저장된 값은 written으로, convolution 수행 결과는 conv로 print한 결과는 다음과 같다.

```
input
5 6 7
33 34 35
61 62 63
written
5 6 7
33 34 35
61 62 63
conv: 34
```

```
input
6 7 0
34 35 36
62 63 0
written
6 7 0
34 35 36
62 63 0
conv: 30
```

image

a	b	c
d	e	f
g	h	i

mask

[0,0]	[0,1]	[0,2]
[1,0]	[1,1]	[1,2]
[2,0]	[2,1]	[2,2]

첫 번째의 경우 a=5, b=6, c=7, d=33, e=34, f=35, g=61, h=62, i=63 이고 (a*1+b*2+c*1+d*2+e*4+f*2+g*1+h*2+i*1)/16=34 로 실제 계산 결과와 동일함을 확인할 수 있다. 마찬가지로 a=6, b=7, c=0, d=34, e=35, f=36, g=62, h=63, i=0 이고 (a*1+b*2+c*1+d*2+e*4+f*2+g*1+h*2+i*1)/16=30 로 실제 계산 결과와 동일하다.

결과 분석 및 팀원 간 토의 사항

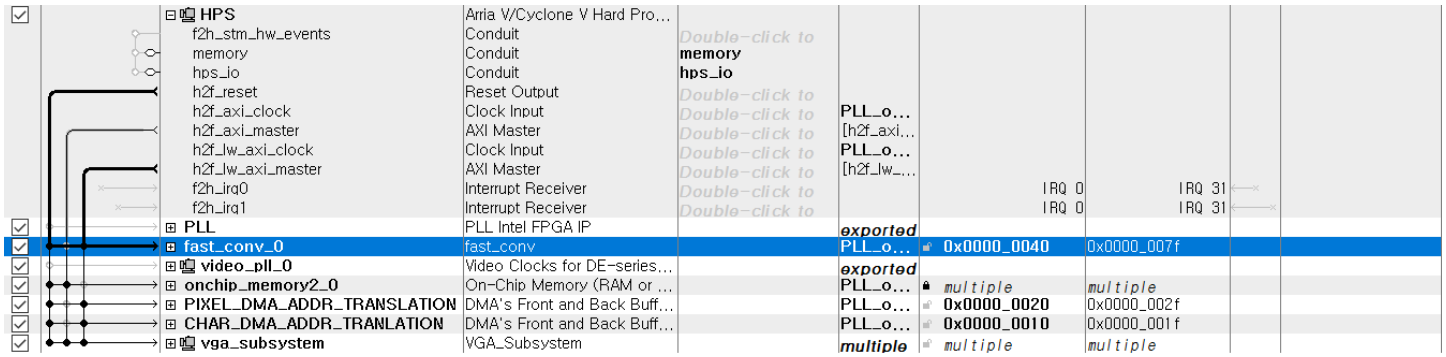
기존의 프로그램에서는 이미지와 마스크 픽셀 한 개씩 접근하여 서로 곱하고 누적하여 convolution을 계산했다. 회로에서는 3x3 mask 기준 이미지 값 3x3 = 9 개를 받아 연산된 값을 메모리 공간에 저장해두고, 사용자는 그 메모리 주소에 저장된 값을 읽음으로써 convolution 계산을 대체할 수 있다. 특정 메모리 주소지에 접근할 때 주의할 점은, 사용하는 pointer의 데이터형에 따라 base+1, base+2 ... 의 의미가 달라질 수 있다는 것이다. pointer가 char형이라면 기존 주소지에서 +1한 곳을 가리키게 되고, int형이라면 +4한 곳을 가리키게 된다. 포인터 데이터형을 올바르게 지정해주어야 의도한대로 writedata값을 원하는 메모리 주소에 저장할 수 있다.

기존 SW보다 HW로 연산하는 것이 더 빠른 이유는 회로에서는 연산의 병렬화가 가능하기 때문이다. SW에서는 for loop을 돌면서 이미지 픽셀 값과 마스크를 곱한 후에 값을 9번(3X3 mask 기준) 누적하여 convolution 연산을 한다. 그에 반해 회로에서는 이미지 값만 모두 메모리에 load되어 있는 상태라면, 반복문을 돌리면서 값을 누적할 필요 없이 한 번에 값을 계산할 수 있다. 이러한 회로의 구조가 연산의 가속화를 가능하게 한다.

특정 convolution에 사용되는 mask는 모든 이미지 픽셀에 대해 동일하기 때문에 한 가지 mask만을 쓴다고 가정했다. 따라서 본 회로 설계에서는 이미지 픽셀 값에 곱해지는 mask 값을 상수로 지정하여 convolution 값을 계산하였다. 하지만 convolution 연산은 Convolutional Neural Network(CNN)같은 neural network나 여러가지 이미지 필터링에도 사용되는 중요한 연산이다. 본 설계에서 사용한 Gaussian mask뿐만 아니라 좀 더 일반적인 사용을 위해서라면 mask도 image pixel값처럼 writedata signal을 통해 받아서 사용하도록 해야 한다. 하지만 한 가지 회로 사용을 위해 고려해야 하는 사항은 값을 메모리에 load하는 데 소요되는 시간이다. 연산을 한 번 하는 것을 기준으로 시간을 측정하면 HW 쪽이 더 빠르지만, 이는 이미지 값을 메모리에 load하는 시간을 제외하고 측정한 결과이다. 회로 사용을 위해 메모리에 write해야 하는 값들이 너무 많다면, 기존의 SW 연산보다도 소요 시간이 늘어날 수 있다.

✓ Assignment 3

개요



본 assignment에서는 Pure S/W를 위한 architecture와 accelerator component를 종합하여 최종 circuit architecture를 만드는 실험을 수행한다. S/W에서의 결과와 H/W component를 통한 결과를 비교하여 동일여부를 판단하는 것이 가장 중요하고, 그에 따른 performance 향상이 얼마나 있는 지를 검증하는 것이 이번 lab의 최종 목표이다. Avalon MM interconnect spec에 따라 component를 구현하고, platform designer를 통해 기존 system과 적절하게 연결한 후 C 기반 program에서 해당 component에 접근하여 연산을 수행하는 형태로 진행된다.

Algorithm의 processing kernel로 결정된 Convolution 연산에 대해서 가속 결과를 측정하기 위해 1 pixel 단위로 연산하는 과정을 구현하였고, 이를 기반으로 전체 image에 대한 blurring 연산을 수행하였다.

진행 내용

```
//////component 접근 함수
void img_load(int start_pix){

	int i;int j;
	for(i=0;i<3;i++){
		for(j=0;j<3;j++){
			*(conv_acc_base + 3*i + j)₩
			= (unsigned int)padded_image[start_pix + (i_w+2)*i + j];	// component에 32bit pixel value를 load.
		}
	}
}
char fast_conv(void){
	int tmp; char tmp_s;
	tmp =  *(conv_acc_base + 9);				// component의 연산 결과를 read.
	tmp_s = CLIP_8((tmp&0xffff));				// CLIP 연산 후 pixel값을 반환.
	return tmp_s;
}
```

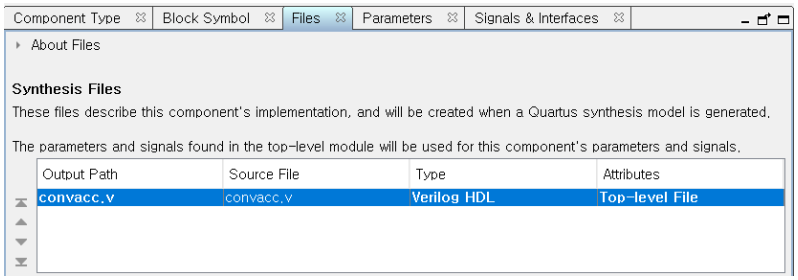
```
///////Software와 동일한 연산 수행
	 conv_acc_base = (volatile int*)(lw_virtual + ACC_BASE);		// component base addr을 virtual address mapping

	for(i=0;i<i_h;i++){
		for(j=0;j<i_w;j++){
		    img_load(j+i_w*i);				// img load
		    output_image_acc[j+i_w*i] = fast_conv(j+i_w*i);	// 연산 결과를 read 후 저장.
		}
	}
/////////HW 기반 결과 display
	for(i=100;i<220;i++){
		for(j=100;j<220;j++){
			plot_pixel(j-50, i, output_image_acc[256*(256-i) + j]);		// 연산 결과 display
		}
	}
```

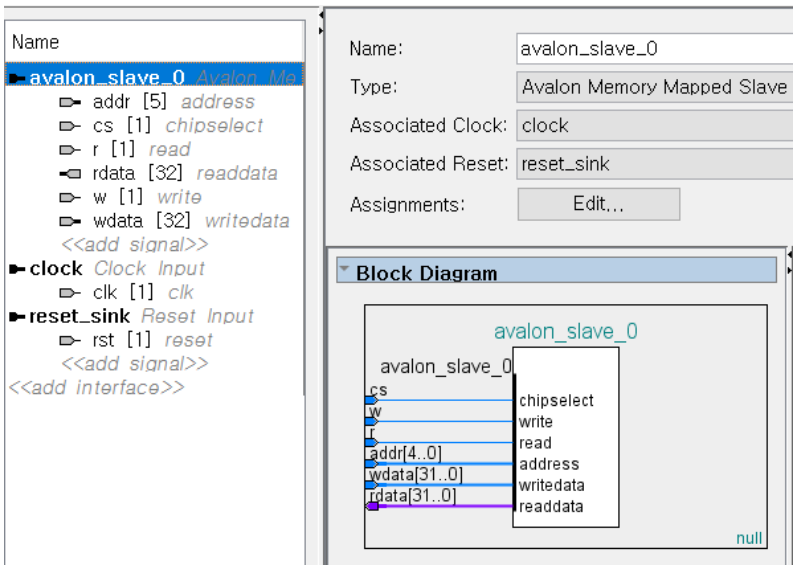
```
//////////////모든 알고리즘 구성요소 수행결과 비교
gettimeofday(&start1, NULL);

	zero_padding();			// zero padding

	gettimeofday(&end1, NULL);
	elapsed1 = end1.tv_usec - start1.tv_usec;
	printf("₩npadding exe time: %ld₩n", elapsed1);

	gettimeofday(&start2, NULL);
```

```
conv_1pix(0);                    // 1 pixel에 대한 S/W conv 수행.

gettimeofday(&end2, NULL);
elapsed2 = end2.tv_usec - start2.tv_usec;
printf("₩nconv exe time: %ld₩n", elapsed2);

gettimeofday(&start3, NULL);

fast_conv();                     // 1 pixel에 대한 H/W conv 수행(연산 결과 read)

gettimeofday(&end3, NULL);
elapsed3 = end3.tv_usec - start3.tv_usec;
printf("₩nfast conv exe time: %ld₩n", elapsed3);
```

진행 결과

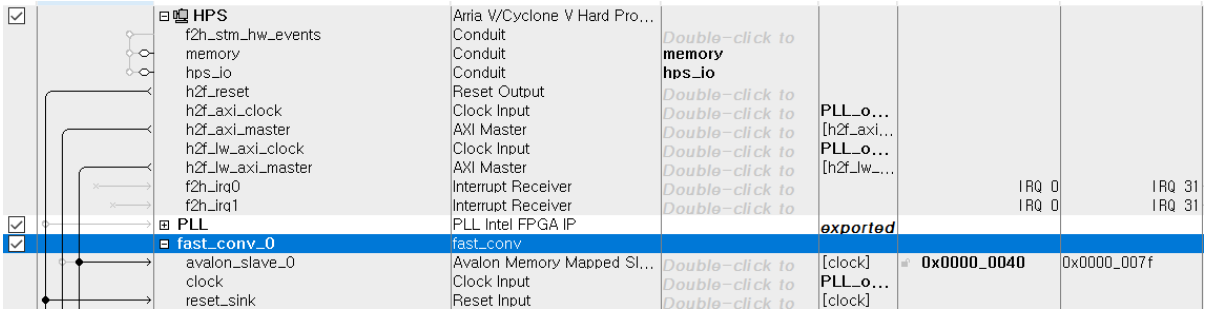
1. Make Custom component





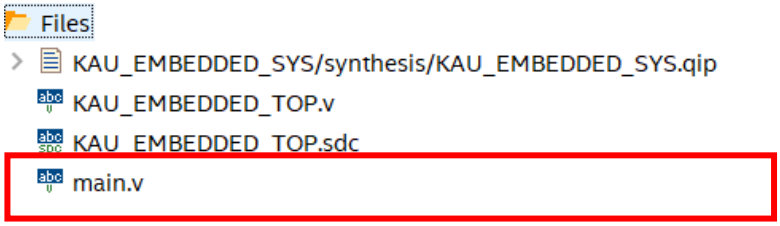
➔ 위의 assignment에서 작성한 v 파일을 통해 custom component를 만든다.

2. connect the component at base system



➔ component의 입력을 고려하여 적절히 base system과 연결한다.

3. Program the FPGA part with RBF



➔ accelerator component

```
root@de1soclinux:~/sys_test# ./program_fpga ./fast_conv.rbf
2+1 records in
2+1 records out
2870504 bytes (2.9 MB) copied, 0.140009 s, 20.5 MB/s
FPGA Programmed with ./fast_conv.rbf
```

➔ platform designer를 통해 생성된 .qip 파일과 .v 파일을 모두 compile하고 생성된 .sof 파일을 .rbf 파일로 변환하여 DE1-SoC 보드 상에 전체 system을 올릴 수 있다.

4. compile / execute

```
root@de1soclinux:~/sys_test# gcc last_main.c -o qq
last_main.c:85:6: warning: conflicting types for 'conv_1pix' [enabled by default
last_main.c:79:13: note: previous implicit declaration of 'conv_1pix' was here
last_main.c: In function 'main':
last_main.c:111:27: warning: assignment makes integer from pointer without a cas
t]
last_main.c:117:17: warning: incompatible implicit declaration of built-in funct
by default]
root@de1soclinux:~/sys_test# ./qq
```

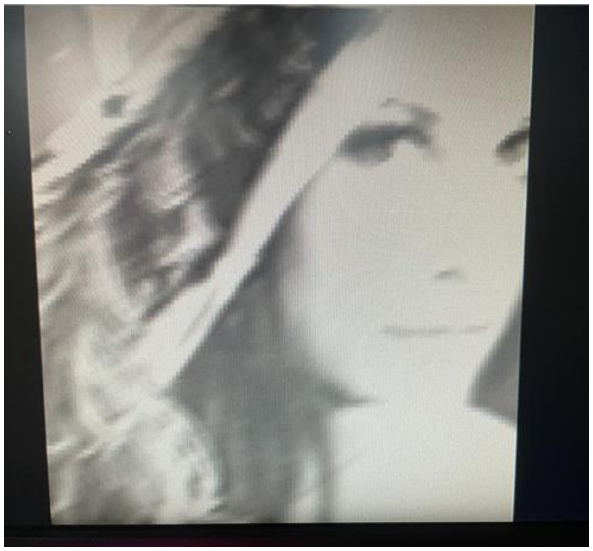
➔ C main 파일을 compile 후 실행한다.

5. Demo

➔ 왼쪽 위 : 원본 이미지, 오른쪽 위 : S/W구현 이미지, 오른쪽 아래 : H/W구현 이미지

VGA subsystem의 pixel buffer 용량 한계로 인해 모든 이미지를 표현할 수 없었고, 각 이미지의 일부를 추출하여 3장을 최대로 display 하는 형태로 demo를 진행하였다. 위와 같이 S/W와 H/W모두 blurring효과가 적용된 image를 결과로 도출하는 것을 확인할 수 있었다.

➔ S/W 결과



➔ H/W 결과

사진 촬영 시 빛에 의한 왜곡이 조금 있지만 두 이미지 모두 같은 결과를 내는 것을 확인할 수 있었다.

6. Time measure

```
gettimeofday(&start1, NULL);

zero_padding();

gettimeofday(&end1, NULL);
elapsed1 = end1.tv_usec - start1.tv_usec;
printf("\npadding exe time: %ld\n", elapsed1);

gettimeofday(&start2, NULL);

conv_1pix(0);

gettimeofday(&end2, NULL);
elapsed2 = end2.tv_usec - start2.tv_usec;
printf("\nconv exe time: %ld\n", elapsed2);

img_load(0);

gettimeofday(&start3, NULL);

fast_conv();

gettimeofday(&end3, NULL);
elapsed3 = end3.tv_usec - start3.tv_usec;
printf("\nfast conv exe time: %ld\n", elapsed3);
```

```
padding exe time: 2471

conv exe time: 2

fast conv exe time: 1
```

➔ 다음과 같이 gettimeofday 함수를 통해 각 algorithm 구성함수의 수행시간을 측정해 보았다. Convolution은 output image의 한 pixel을 만드는 연산을 기준으로 측정하였고, 그 결과 약 두 배의 가속효과가 있었다. 즉, convolution 연산에 대한 efficacy가 50%인 것을 확인할 수 있었다. S/W conv 연산은 2us, H/W conv 연산은 1us가 소요되었고, 1 pixel에 대해서 병렬적으로 곱셈과 덧셈 연산을 수행하기 때문에 S/W 연산보다 빠른 성능을 낼 수 있었다.

결과 분석 및 팀원 간 토의 사항

Multi-processor 기반 Cyclone V DE1-SoC platform에서 platform designer를 통해 custom circuit architecture를 만들고, 그 architecture를 기반으로 특정 application을 가속하는 실험을 수행해 보았다. Algorithm의 processing kernel인 convolution 연산을 가속하는 H/W component를 만들어 해당 연산을 수행한 결과, C 기반 pure S/W로만 구현한 algorithm과 동일한 결과를 내고 convolution 연산에 대한 performance 향상이 있는 것을 확인할 수 있었다.

해당 component는 data를 모두 load한 후 convolution 연산에 대해서 가속을 구현하는 component이다. 좀 더 실용적인 가속기를 만들기 위해서는 input image에 대한 data reuse나 input image에 대한 buffer를 구현하여 CPU와 component 간의 load/store 과정을 간소화하는 과정이 필요할 것임이 예상된다.

✓ 조원 별 기여 사항 및 느낀점

이름	기여도 (0 - 100%)	기여 사항	느낀점
박관영	33.3%	- S/W 설계 및 보고서 작성	Platform designer를 통해 custom architecture를 구현하는 법에 대해 학습할 수 있었고, 특정 IP를 C 언어 기반 program에서 접근하여 제어하는 방법에 대한 이해도를 높일 수 있었다.
오한별	33.3%	- 가속기 설계 및 보고서 작성	Verilog HDL을 사용하여 가속기 회로를 직접 구현한 후 bus에 연결하고, 해당 회로를 사용하기 위한 C언어 기반의 firmware를 작성하는 일련의 과정들을 통해 어떻게 연산을 더 빠르게 할 수 있는지 학습할 수 있었다.
김서영	33.3%	- 가속기 설계 및 보고서 작성	Platform design을 통해 HW accelerate design 방법론을 학습할 수 있었고, 이를 기반으로 firmware에 대한 이해도를 높일 수 있었다. HW accelerator를 사용함으로써 연산을 병렬적으로 처리하여 속도 향상이 가능하다는 점 직접 확인, 검증할 수 있었다.