

Digital Signal Processor

Image processing with three filtering modes



2016135011 Min-Young Shin

Contents

I. Introduction

- Subject of the project
- Goal of the project

II. Hardware

- Block diagram
- BRAM port module
- Register set module
- 7-segment module
- Clock cycle IP
- TFTLCD module
- Filter module

III. Firmware

- main function
- Interrupt handler

IV. Results

- Operations
- Consumed clock cycles
- Filtered images

V. Discussion

VI. Reference

Introduction

1. Subject of the Project

The subject of the project is to design a digital signal processor using a Xilinx FPGA. The FPGA consists of Processing System(PS) and Programmable Logic(PL). Thus, it needs to design hardware and firmware that process images.

2. Goal of the Project

The digital signal processor has to meet the following requirements. First of all, it has three filtering modes to process an image: edge, sharp, and blur. Image filtering is implemented through 1-D convolution. Each filtering mode has their convolution coefficient as follow.

$$\text{Edge Filter's 1D coefficient} = [-1, -2, 6, -2, -1]$$

$$\text{Sharp Filter's 1D coefficient} = [-1, -2, 7, -2, -1]$$

$$\text{Blur Filter's 1D} = [0.1, 0.2, 0.4, 0.2, 0.1]$$

The size of the image we need to process is 480x272 with 16 bits per one pixel. And the edge of the image will be zero-padded for ease of calculation. It means the value of *Reference Image*[*x*] will be zero if *x* is less than zero or larger than 480*272-1 in the following equation.

$$\begin{aligned} \text{Filtered Image}[i] = & (\text{coeff}[0] * \text{Reference Image}[i - 2]) \\ & + (\text{coeff}[1] * \text{Referecnce Image}[i - 1]) \\ & + (\text{coeff}[2] * \text{Reference Image}[i]) \\ & + (\text{coeff}[3] * \text{Referece Image}[i + 1]) \\ & + (\text{coeff}[4] * \text{Referece Image}[i + 2]) \end{aligned}$$

Each RGB value has to calculate separately, and it is handled as zero when it occurs overflow.

The next goal of the design is to record how many clock cycles are taken. It is necessary to analyze and compare the DSP's performance.

Moreover, the design has to handle interrupt. The firmware should catch the hardware interrupt signal and read the cycle register's value that is storing the consumed clock cycle during DSP calculation. Then, it is shown by 7-segment. The whole process including read the consumed cycle and write the 7-segment displaying value would be handled by firmware logic.

Finally, the TFTLCD will display the reference image and filtered image for a second alternately while the 7-segment shows the consumed clock cycle for calculation.

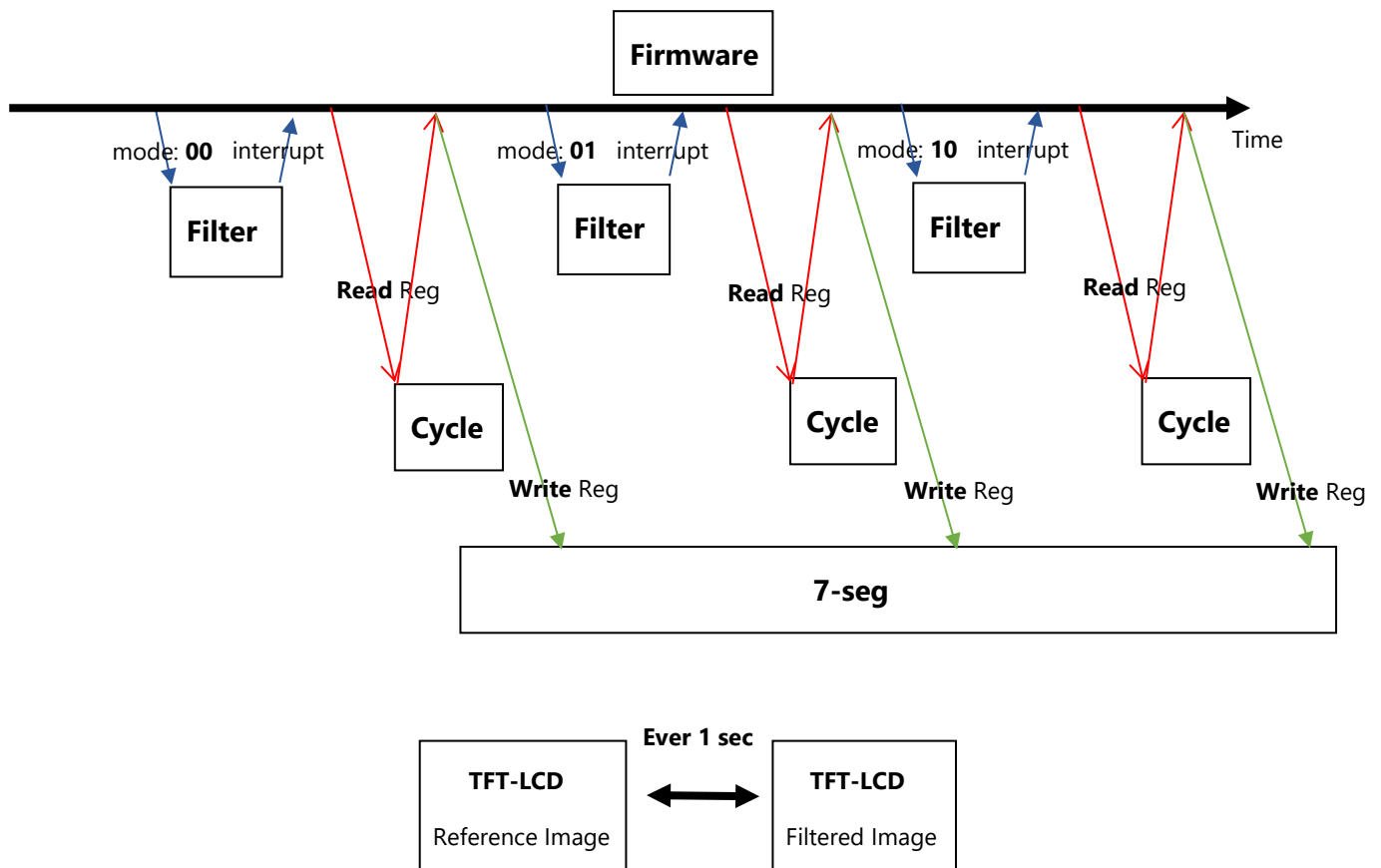


Figure 1. Control Logic of the image filtering DSP

Hardware

1. Block diagram

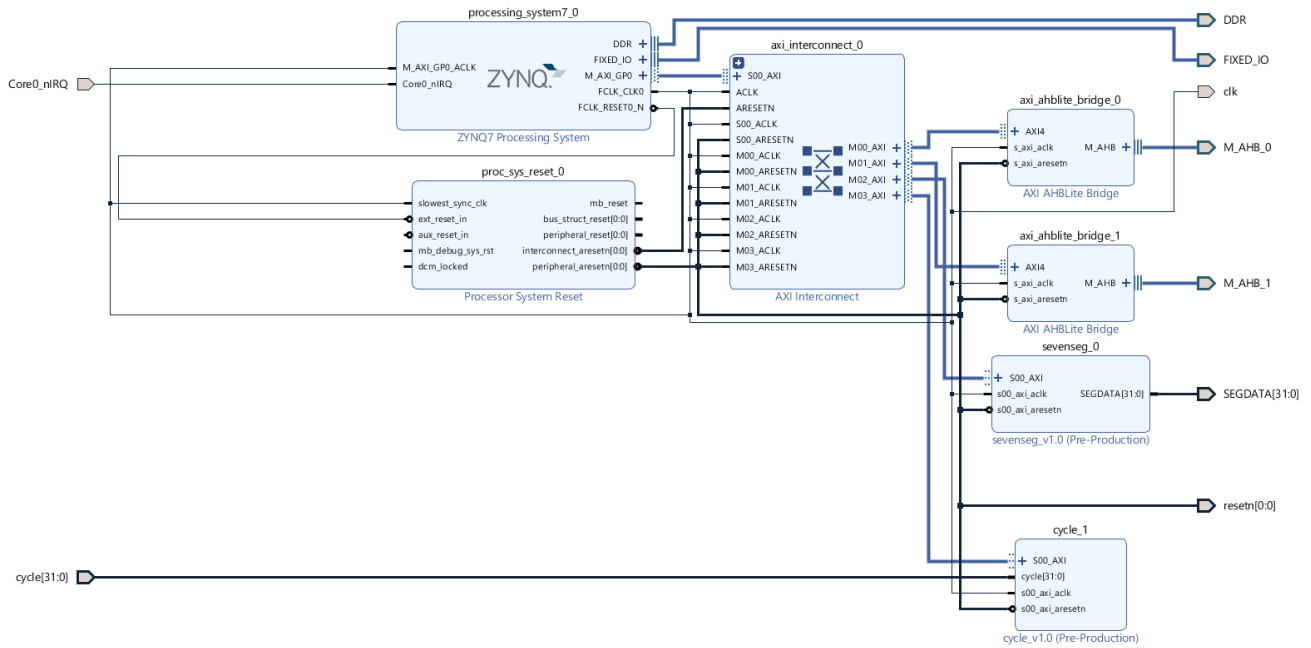


figure 2. Block diagram for system wrapper

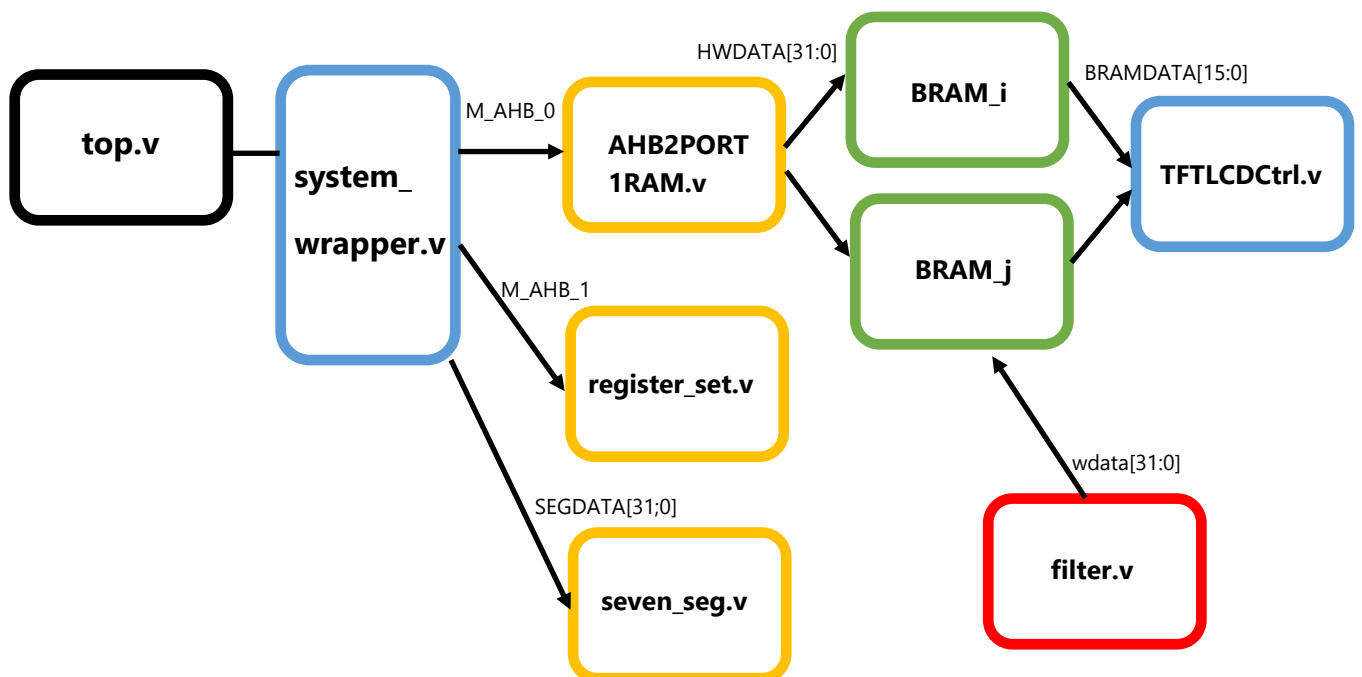


figure 3. hardware architecture for whole system

2. BRAM port module

AHB2PORT1RAM.v in figure 3 is the module for handling BRAM data flow. We applied two 256KB BRAM which have 16 bits read width and 32 bits write width. This is because TLFTLCD read 16 bits width and AHB bus protocol write 32 bits width as well as a filter module. Thus, the key implementation code is as follows.

```
1. module AHB2PORT1RAM ;
2. .
3. omit non-key part
4. .
5. /* To write data from the filter */
6. wire en_BRAM_i;
7. wire en_BRAM_j;
8. wire [15:0] dout_BRAM_i;
9. wire [15:0] dout_BRAM_j;
10.
11. /* Write data from the filter when fwrite signal comes in */
12. assign en_BRAM_i = (sel_rbram == 0) ? 1'b1 : 0;
13. assign en_BRAM_j = (sel_rbram == 1) ? 1'b1 : 0;
14. assign port2do = (sel_rbram == 0) ? dout_BRAM_i : dout_BRAM_j;
15.
16. /*original image store and load */
17. bufferram bufferram_i (
18.     .clka ( PORT2HCLK ), // read clock
19.     .ena ( en_BRAM_i ), // read enable
20.     .wea ( port2bwe ), // write enable -> always 0
21.     .addra ( port2addr[16:0] ), // read address
22.     .dina ( port2di ), // write data -> always 0
23.     .douta( dout_BRAM_i ), // read data
24.     .clkb ( PORT1HCLK ), // write clock from processor system
25.     .enb ( REQ[1] ), // write enable from processor system
26.     .web ( BWE1 ), // write enable from processor system
27.     .addrb( P1HADDRMUX[17:2] ), // write address from processor system
28.     .dinb(PORT1HWDATA), // write data from processor system
29.     .doutb(PORT1HRDATA) // read data to processor system
30.);
31.
32. /* filtered Image store and load */
33. bufferram bufferram_j (
34.     .clka ( PORT2HCLK ), // read clock
35.     .ena ( en_BRAM_j ), // read enable
36.     .wea ( 1'b0 ), // write enable -> always 0
37.     .addra ( port2addr[16:0] ), // read address
38.     .dina ( 1'b0 ), // write data -> always 0
39.     .douta( dout_BRAM_j ), // read data
40.     .clkb ( PORT2HCLK ), // write clock from filter.v
41.     .enb ( fwrite ), // write enable from filter.v
42.     .web ( fwrite ), // write enable from filter.v
43.     .addrb( fwaddr ), // write address from filter.v
44.     .dinb( fwdata ), // write data from filter.v
45.     .doutb(1'b0) // read data to filter.v -> always 0
46.);
```

Line 6 to 14, needs to decide which BRAM would be shown in TFTLCD. Since the TFTLCD should change displaying image every one second, enable signal and read data from BRAM is decided by selection signal(sel_bram). Line 17, the bufferram_i is used to store reference images from the processor

system(firmware) and send data to the TFTLCD. Line 33, bufferram_j is used to store filtered images from the filter.v module and send data to the TFTLCD same as bufferram_i.

3. Register set module

The register set IP is needed since the firmware gives the filtering modes to the hardware. We can get values from firmware through the register set IP. Furthermore, the designed filter we made starts its calculation when the START signals come and are initialized when the INIT signals come. Thus, we need four registers that can control hardware. So, we made those few additional lines to the baseline code.

```
1. /* four register set module */
2.
3. module register_set ;
4. .
5. /* omit non-key part */
6. .
7.     reg [31:0] curr_reg0; //mode[0]
8.     reg [31:0] next_reg0;
9.     reg [31:0] curr_reg1; //mode[1]
10.    reg [31:0] next_reg1;
11.    reg [31:0] curr_reg2; //START
12.    reg [31:0] next_reg2;
13.    reg [31:0] curr_reg3; //INIT
14.    reg [31:0] next_reg3;
15.    .
16.    /* omit non-key part */
17.    .
18.    always @ (curr_reg0 or curr_reg1 or curr_reg2 or curr_reg3 or
curr_hsel or curr_hwrite or curr_haddr or sig_hwdata)
19.    begin
20.        next_reg0 <= curr_reg0;
21.        next_reg1 <= curr_reg1;
22.        next_reg2 <= curr_reg2;
23.        next_reg3 <= curr_reg3;
24.        next_ext <= curr_ext;
25.        sig_hrdata <= 32'd0;
26.        if (curr_hsel)
27.        begin
28.            case (curr_haddr[5:2])
29.                4'b0000: sig_hrdata <= curr_reg0;
30.                4'b0001: sig_hrdata <= curr_reg1;
31.                4'b0010: sig_hrdata <= curr_reg2;
32.                4'b0011: sig_hrdata <= curr_reg3;
33.                default: sig_hrdata <= 32'd0;
34.            endcase
```

Line 11 to line 14 are additional lines on the baseline code which is provided in previous weeks. We added two 32 bits registers for START and INIT signals. Line 18 to 34 are code for receiving AHB write signal came from firmware. The write signal is handled as a read signal to the slave, and the slave stores the value to its registers according to the address offset; offset of 0x00 means mode[0], 0x04 mean mode[1], 0x08 means START, and 0x1c means INIT.

4. 7-segment module

The seven-segment module should show the cycle value that came from firmware. The firmware reads a value from cycle IP, then the firmware wants to access the seven_seg IP to store the value of the cycle. Accessing to the seven_seg IP is processed by bus protocol AHB, and this can work without any additional setting since we built the system wrapper to handle the system to IP communication. Therefore, the firmware uses functions, register to write, in seven segments IP's base address so that firmware can modify segdata[31:0]. The only thing we needed to do was to declare the 7-segment module which is provided in the previous week and gives it segdata[31:0] that came from IP in the system wrapper.

```
1. module top;
2.
3. seven_seg seven_seg_inst (
4.   .clk(TFTLCD_CLK),
5.   .resetn(TFTLCD_nRESET),
6.   .data(SEGDATA), //SEGDATA = system_wrapper's seven_seg IP output
7.   .segout(segout),
8.   .segcom(segcom)
9. );
```

5. Clock cycle IP

The consumed clock cycle is recorded by the filter module and the module sends the cycle value to the top module. Then, the cycle goes to clock IP in the system wrapper. If we do not modify the basic IP structure, the cycle value is just flown away which means none of the registers in clock IP stores cycle value. Therefore, we modified the cycle IP as follows.

```
1. module cycle_v1_0_S00_AXI #
2. .
3. /* omit non-key part */
4. .
5.     // Users to add ports here
6.     input wire[31:0] cycle,
7.     // User ports ends
8.
9. always @( posedge S_AXI_ACLK )
10.     begin
11.         if ( S_AXI_ARESETN == 1'b0 )
12.             begin
```



```

13.         slv_reg0 <= cycle;
14.         slv_reg1 <= 0;
15.         slv_reg2 <= 0;
16.         slv_reg3 <= 0;
17.     end
18.     else begin
19.         .
20.         /* omit non-key part */
21.         .
22.
23.     default : begin
24.         slv_reg0 <= cycle;
25.         slv_reg1 <= slv_reg1;
26.         slv_reg2 <= slv_reg2;
27.         slv_reg3 <= slv_reg3;
28.     end

```

Line 6, we declared a wire which is the input port that we send to the system wrapper. To store the value of the cycle, we connect it to a slave register named *slv_reg0*. Line 13 and line 24 is the only part we make additional code to the baseline code so that we can access the cycle register via the *base address of the Cycle IP*.

6. TFTLCD module

The TFTLCD should display the reference image and filtered image, alternately. However, the image data is stored in BRAM, and the BRAM selection is controlled by sel_bram as we discussed right before. Therefore, most of the content of the code is not different from what was done in previous weeks. The only thing we need to add was the content of the selection signal. It should be changed every second. This content is included in BRAMCtrl.v inside TFTLCDCtrl.v.

```

1. module BRAMCtrl;
2. .
3. omit non-key part
4. .
5. //generate selection signal and change it every one second
6. always @ (posedge CLK or posedge RESET)
7. begin
8.     if (RESET)
9.     begin
10.         sel <= 0;
11.         cnt <= 33'd0;
12.         START_reg <= 0;
13.     end
14.     else
15.     begin
16.         /* if START signal changes to one, sel register becomes zero */
17.         if((START_reg != START) && (START == 1'b1)) begin
18.             sel <= 0;
19.             cnt <= 0;
20.         end

```

```

21.     START_reg <= START;
22.     if (cnt < 33'd25000000)
23.         cnt <= cnt + 33'd1;
24.     else
25.         cnt <= 33'd0;
26.     /* reverse sel register every one second */
27.     if (cnt == 33'd25000000)
28.         sel <= ~sel;
29. end
30.end

```

Line 8 to 13 is reset condition. Line 17, when START signal, came from firmware, is changed to 1, sel register would be set as 0. It means the TFTLCD shows the reference(original) image. Line 22 to 25, counts 25000000 cycles which is one second in 25MHz clock frequency. If it counts 25000000, sel register changes its value. If it is changed to 1, it means the TFTLCD shows the filtered image.

7. Filter module

It is the key module of this project. Thus, we will introduce the code as many as possible. It receives read address, read data, filtering mode, calculation start signal, and initialization signal as input, then it sends write enable, write address, write data, interrupt signal, consumed cycle, and calculation end signal as output.

```

1. /*
2.  * key module for DSP hardware
3.  * read a image from BRAM memory
4.  * filtering the image according to the filtering mode
5.  * 00: edge mode, 01: sharp mode, 10: blur mode
6.  * generate interrupt signal after filtering done
7.  * record consumed cycle for processing
8.  */
9. module filter(
10.    input clk,
11.    input resetn,
12.    input [16:0] raddr, //read address
13.    input [15:0] rdata, //read data
14.    input [1:0] mode, //filtering mode
15.    output reg wen, //write enable
16.    output reg [15:0] waddr, //write address
17.    output reg [31:0] wdata, //write data
18.    output reg Intr, //interrupt signal
19.    output reg [31:0] cycle, //consumed cycle
20.    input START, //start signal
21.    output reg END, //end flag
22.    input INIT //initialize signal
23.);
24./* skip some declaration parts */
25.
26. //edge mode
27.assign edge_R = im1_R * (-1) + im2_R * (-2) + im3_R * 6 + im4_R * (-2) +
    im5_R * (-1) < 32 ?
28.    im1_R * (-1) + im2_R * (-2) + im3_R * 6 + im4_R * (-2) +
    im5_R * (-1) : 0;

```

```

29.assign edge_G = im1_G * (-1) + im2_G * (-2) + im3_G * 6 + im4_G * (-2) +
   im5_G * (-1) < 64 ?
30.      im1_G * (-1) + im2_G * (-2) + im3_G * 6 + im4_G * (-2) +
   im5_G * (-1) : 0;
31.assign edge_B = im1_B * (-1) + im2_B * (-2) + im3_B * 6 + im4_B * (-2) +
   im5_B * (-1) < 32 ?
32.      im1_B * (-1) + im2_B * (-2) + im3_B * 6 + im4_B * (-2) +
   im5_B * (-1) : 0;

```

Before line 24, declare RGB wires for all filtering mode cases. Line 27 to 32, assign convolution calculation to RGB wires so that it calculates convolution in parallel. Also, it is written the same for sharp mode and blur mode.

```

33.assign last_1_edge_R = im2_R * (-1) + im3_R * (-2) + im4_R * 6 + im5_R * (-
   2) < 32 ?
34.      im2_R * (-1) + im3_R * (-2) + im4_R * 6 + im5_R * (-2) :
   0;
35.assign last_1_edge_G = im2_G * (-1) + im3_G * (-2) + im4_G * 6 + im5_G * (-
   2) < 64 ?
36.      im2_G * (-1) + im3_G * (-2) + im4_G * 6 + im5_G * (-2) :
   0;
37.assign last_1_edge_B = im2_B * (-1) + im3_B * (-2) + im4_B * 6 + im5_B * (-
   2) < 32 ?
38.      im2_B * (-1) + im3_B * (-2) + im4_B * 6 + im5_B * (-2) :
   0;

```

Line 33 to 38, is needed for the last to second pixel's edge filtering mode calculations. As we discuss in the introduction, the last two pixels are exceptional cases, thus, it has to pad zero. Since the calculations are different from the general case, we made additional wire assignments. There are codes for blur, sharp, and last pixel but we omitted them due to the limitations of the paper.

```

39.//Sync with loading image
40.always@(negedge resetn or posedge clk)
41.begin
42.    if(!resetn)
43.        Sync <= 0;
44.    else if(START == 1 && END == 1 && INIT == 0)
45.        Sync <= 0;
46.    else if(START == 1 && END == 0 && INIT == 0 && raddr==17'd130559)
47.        Sync <= 1;
48.end

```

Line 39 to 48 is needed for Synchronization. The BRAM read address(raddr) comes from BRAMCtrl.v and its range is 0 to 130599. However, we cannot guarantee it starts from the base address when we want to start the calculation. Thus, we put the Sync register which is set as one when the raddr is initialized.

```

49./* handling interrupt and cycle count */
50.always@(negedge resetn or posedge clk)
51.begin
52.    if(!resetn) begin
53.        Intr <= 1;
54.        cycle <=0;
55.    end
56.    /* increase cycle during calculation */
57.    else if(START == 1 && END == 0 && INIT == 0 && Sync == 1) begin
58.        Intr <= 1;
59.        if(cycle != 32'hFFFFFFFF)
60.            cycle <= cycle +1;
61.    end
62.    /* calculation end & interrupt start condition */
63.    else if(START == 1 && END == 1 && INIT==0)
64.        Intr <= 0;
65.    /* interrupt end condition */
66.    else if(INIT == 1) begin
67.        Intr <= 1;
68.        cycle <=0;
69.    end
70.end

```

Line 49 to 66 is a block that has two roles. The first role is to record the consumed clock cycle and the second role is to generate an interrupt signal when the calculations are finished. Line 52 to 55 is a simple reset condition. Line 57 to 61 deal with the calculation situation. Interrupt should remain its value as 1(non-interrupt), and the cycle increases its value every cycle. Line 62 to 69, it is the initialization condition that is ordered by the firmware.

```

71./* calculation block */
72.always@(negedge resetn or posedge clk)
73.begin
74.    if(!resetn || ((START==0) && (INIT == 1)))
75.        begin
76.            wen <= 0; waddr <=0; wdata <=0;
77.            END <= 0; lastFlag <= 0;
78.            im1_R <= 0; im1_G <= 0; im1_B <= 0;
79.            im2_R <= 0; im2_G <= 0; im2_B <= 0;
80.            im3_R <= 0; im3_G <= 0; im3_B <= 0;
81.            im4_R <= 0; im4_G <= 0; im4_B <= 0;
82.            im5_R <= 0; im5_G <= 0; im5_B <= 0;
83.        end
84.    else if(START == 1 && END == 1 && INIT == 0)
85.        wen <= 0;
86.    else if(START == 0 && INIT == 1) begin
87.        END <= 0;
88.        wen<= 0;
89.    end

```

This is the main block of the filter module that serves as sending a signal to BRAM. Line 74 to 83 is simple a reset condition. In lines 84 to 85, if the calculations are finished which means END is set as 1, write enable is deactivated. Line 86 is a condition that the initialization signal has arrived

from the firmware's interrupt handler. Then, the END register is initialized, and write enable is waiting until calculation starts.

```

90.    //start calculation if a mode is changed or start is on
91.    else if(START == 1 && END == 0 && INIT== 0 && Sync==1)
92.    begin
93.        //handling last, last-1 cell
94.        im5_R <= rdata[15:11]; im5_G <= rdata[10:5]; im5_B <= rdata[4:0];
95.        im4_R <= im5_R; im4_G <= im5_G; im4_B <= im5_B;
96.        im3_R <= im4_R; im3_G <= im4_G; im3_B <= im4_B;
97.        im2_R <= im3_R; im2_G <= im3_G; im2_B <= im3_B;
98.        im1_R <= im2_R; im1_G <= im2_G; im1_B <= im2_B;
99.
100.        //store last two cell
101.        if (raddr == 1 && lastFlag==1) begin
102.            wdata <= lastcell;
103.            wen <= 1;
104.            waddr <= 16'd65279;
105.            END <= 1;
106.            lastFlag <= 0;
107.        end
108.        //write data when address is even number
109.        else if(raddr[0] == 0 && mode == 2'b00 && (raddr >=3 ||
lastFlag == 1)) begin
110.            if(lastFlag)
111.                waddr <= 16'd65278;
112.            else
113.                waddr <= raddr[16:1]-2;
114.            wdata[31:16] <= {edge_R[4:0], edge_G[5:0], edge_B[4:0]};
115.            wen <= 1;
116.        end
117.        //store half of data when address is odd number
118.        else if(raddr[0] == 1 && mode == 2'b00 && raddr >=3) begin
119.            wdata[15:0] <= {edge_R[4:0], edge_G[5:0], edge_B[4:0]};
120.            wen <= 0;
121.        end
122.        //last image address
123.        if(raddr == 17'd130559)
124.            lastFlag <= 1;
125.
126.        //calculation last pixel information rather than making zero
padding
127.        if(raddr == 17'd0 && lastFlag == 1 && mode == 2'b00 )
128.            lastcell <= {last_2_edge_R[4:0], last_2_edge_G[5:0],
last_2_edge_B[4:0],
129.                last_1_edge_R[4:0], last_1_edge_G[5:0],
last_1_edge_B[4:0]};
130.        end
131.
132.    endmodule
133.

```

The calculation starts when proper START and INIT signal is coming from firmware, END is initialized, and Sync register is set as 1 which means read address starts increasing from ground address. The new reference image's RGB values are stored in fifth image registers and their values are shifted from higher to lower image registers. Using this manner, we do not need to make an exceptional case for the first pixel

and the second pixel since the initial register's value is zero. It means we do not need to pad zero for the first and second pixels. Line 101 to 107, is needed to store the last two pixels. If the last flag register is set as 1 and read BRAM address is one, it is time to store the last two-pixel RGB value to BRAM. Line 109 to 116 is needed for writing data to BRAM in the general case. If it is last to the third, fourth case, the last flag will be set as 1 so that the write address must be the last to a second address of the BRAM. If not, the write address would be $raddr / 2 - 2$. It is divided by two because the write width of the BRAM is four but the read width of the BRAM is two. And it is subtracted by two because it is delayed one cycle to store BRAM and the address starts from zero, not one. Line 118 to 121 are logic for storing values in the write data register if the reading address is odd. It also stems from the fact that the write width of the BRAM is twice as large as the read width of the BRAM. If the read address is the last address of the image, the last flag is set as one. Line 126 to 130 are the logic for the last pixel case.

Firmware

1. main function

```
1. int mode = 0;
2.
3. int main(void)
4. {
5.
6.     printf("hello world\n");
7.
8.     int x, y;
9.
10.    //Store a reference image
11.    for(y=0;y<272;y++)
12.    {
13.        for(x=0;x<240;x++)
14.        {
15.            Xil_Out32(XPAR_M_AHB_0_BASEADDR+y*960+x*4, hex[x*2 + 1 +
y*480]<<16 | hex[x*2 + y*480]);
16.        }
17.    }
18.
19.    //Start calculation with mode 0
20.    Xil_Out32(XPAR_M_AHB_1_BASEADDR, 0x00); //MODE[0]
21.    Xil_Out32(XPAR_M_AHB_1_BASEADDR+0x04, 0x00); //MODE[1]
22.    Xil_Out32(XPAR_M_AHB_1_BASEADDR+0x08, 0x01); //START
23.    Xil_Out32(XPAR_M_AHB_1_BASEADDR+0x0c, 0x00); //INIT
24.
25.
26.
27.    int Status;
28.
29.    Status = ScuGicExample(INTC_DEVICE_ID);
30.    if (Status != XST_SUCCESS) {
31.        //xil_printf("GIC Example Test Failed\r\n");
32.        return XST_FAILURE;
33.    }
```

```

34.
35.     //xil_printf("Successfully ran GIC Example Test\r\n");
36.     return XST_SUCCESS;
37.}

```

Line 11 to line 17 are codes for storing reference images to the BRAM. This process is done by the AHB2PORT1BRAM module, thus, we write the value of the image to the M_AHB_0 port which is connected to the module.

Line 20 to line 23 is the first calculation start signals. It gives mode of edge filter, START is activated and INIT is deactivated.

After this logic, the interrupt handler is set and the interrupt handler is waiting until the interrupt occurs.

2. Interrupt handler

When the interrupt occurs, the firmware enters the interrupt handler routine to handle the interrupt situations.

```

1. void DeviceDriverHandler(void *CallbackRef)
2. {
3.
4.     //Read a consumed clock cycle register
5.     int cycle = Xil_In32LE(XPAR_CYCLE_0_S00_AXI_BASEADDR);
6.
7.     //Change filtering mode
8.     if(mode != 2)
9.         mode++;
10.    else
11.        mode = 0;
12.    uint segdata = 0 ;
13.
14.    //Convert decimal to 7-segment value
15.    int i;
16.    for(i = 0; i < 8 ; i++){
17.        int temp = cycle % 10;
18.        temp = temp << (4*i);
19.        cycle = cycle / 10;
20.        segdata = segdata + temp;
21.    }
22.
23.    //Write 7-segment register
24.    SEVENSEG_mWriteReg (XPAR_SEVENSEG_0_S00_AXI_BASEADDR, 0, segdata);
25.
26.
27.
28.    //Initialize System
29.    Xil_Out32(XPAR_M_AHB_1_BASEADDR+0X08, 0x00); //START
30.    Xil_Out32(XPAR_M_AHB_1_BASEADDR+0X0c, 0x01); //INIT
31.
32.    int t;
33.
34.    //Displaying for 1249999900 CPU cycle
35.    for(t=0;t<1249999900;t++);
36.

```

```

37.      //Put changed mode
38.      if(mode == 0){
39.          Xil_Out32(XPAR_M_AHB_1_BASEADDR, 0x00);
40.          Xil_Out32(XPAR_M_AHB_1_BASEADDR+0X04, 0x00);
41.      }
42.      else if (mode == 1){
43.          Xil_Out32(XPAR_M_AHB_1_BASEADDR, 0x01);
44.          Xil_Out32(XPAR_M_AHB_1_BASEADDR+0X04, 0x00);
45.      }
46.      else{
47.          Xil_Out32(XPAR_M_AHB_1_BASEADDR, 0x00);
48.          Xil_Out32(XPAR_M_AHB_1_BASEADDR+0X04, 0x01);
49.      }
50.
51.      //Put start signal
52.      Xil_Out32(XPAR_M_AHB_1_BASEADDR+0X08, 0x01); //START
53.      Xil_Out32(XPAR_M_AHB_1_BASEADDR+0X0c, 0x00); //INIT
54.
55.      InterruptProcessed = TRUE;

```

If the interrupt occurs, it means the filter has done its filtering calculation and stored the results of the calculation in the BRAM. Therefore, the first thing the firmware has to do is reading the cycle register.

Line 5 is the code for reading a cycle value in cycle IP. Give the parameter of address as cycle IP's base address.

Line 8 to line 11 are needed to change the filtering mode. The mode variable is a global variable, thus it keeps its value after the interrupt handler's routine is done. The value of mode is increase as one if it is not two. If its value is two, it turns back to the value of zero.

Line 12 to line 21 are contents of converting integer numbers to seven-segment numbers. 32 bits of integer number should be divided into eight pieces. Then, it combines the 32 bits again. By doing so, we convert the cycle integer to the seven segment values.

Line 24 is a function to write the value to seven segments. By writing base address, offset, and value in order, we could write the cycle value to the seven-segment display.

Then, it is needed to initialize the filter module. Line 29 is to initialize the filter module. START signal goes to zero and the INIT signal goes to the one.

In line 32 to 35, the interrupt handler rest enough for 1249999900 cycles during TFTLCD shows the reference image and the filtered image alternately





Then, the interrupt handler sends the changed filtering mode in lines 38 to 49.

With giving changed mode, line 52 to line 53 order to star calculation. START signal set as 1 again and the INIT signal is deactivated.

Results

1. Operations

Table 1. operations according to the filtering mode

Reference Image	Edge filter(mode: 00)
	
Sharp filter(mode: 01)	Blur filter(mode: 10)
	

2. Consumed clock cycles

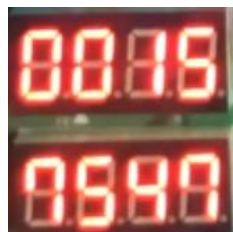






Figure 4. consumed clock cycle

The consumed clock cycles for convolutions calculations are all same at **157547**, regardless of filtering mode.

3. Filtered images

Table 2. filtered images result

Reference Image	Edge filter(mode: 00)
	
Sharp filter(mode: 01)	Blur filter(mode: 10)
	

Discussion

1. Verification

To verify the result of the calculation, we needed to make a testbench and C code for convolution. It would be hard work if we want to compare all values of the calculation. Thus, we checked the only first few values and the last few values.

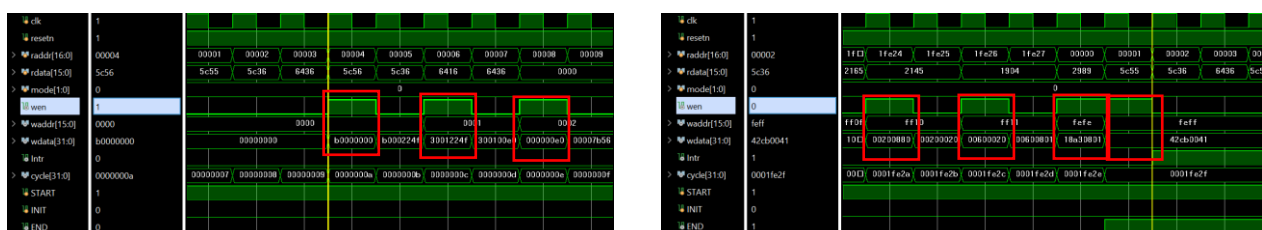


Figure 5. first three values generated by testbench(left), last four values(right)

A comparison of the directly computed convolution and the test bench shows that it is the same. Therefore, we proceeded with verification.

2. Clock cycle

The clock cycle shows 157547, regardless of filtering mode. This is because we introduced the Sync register to start calculation at the same timing. Therefore, the scale of the calculation is similar to the scale of image size which is $408 \times 272 = 130600$ and there is no difference among the filtering mode

3. Improve performance

Our strategy was reading reference image data when the TFTLCD accesses the BRAM. However, this manner is slower twice since the TFTLCD uses an up-cycled clock. Furthermore, we could not be able to read data according to the HSync and VSync.

If we introduce a new BRAM for the filter to access reference images, we can improve the performance of the DSP. We could access data regardless TFTLCD's behavior and the read width of the BRAM can be set larger than 16 bits.

Another performance-improving method is declaring huge registers to store reference images in the filter module. However, it costs a lot more than others, thus, we can not make the same size of register with the reference image, 256KB.

To sum up, in order to increase the performance of the DSP, we need to introduce the new BRAM or registers to be freely accessed by the filter module. As a result, we knew the important thing to improve the DSP's performance is reducing memory latency rather than improving the calculation ability.

Reference

- [1] *Lecture note*