Department of Electronic & Telecommunication Engineering

University of Moratuwa

EN 3030: Circuits and Systems Design

FPGA BASED PROCESSOR DESIGN

FINAL REPORT

Name	Index number
Ginige Y.S.	180195A
Gunasekara H.K.R.L.	180205H
Wijitharathna K.M.R.	180717E
Hewavitharana D.R.	180241M

Supervisor

Dr. Jayathu Samarawickrama

This report is submitted in partial fulfillment of the requirements for the module EN 3030: Circuits and Systems Design.

08th July 2022

ABSTRACT

This report contains the detailed discussion of implementing a processor with Field Programmable Gate Array using Vivado 2019.1. Main task is filtering the input image and then down sampling it by ½ in each dimension using the processor. Report is furnished with methods and theories used for filtering and down sampling an image. Difference between the output image of the processor is compared in the later part of the report. Verilog code, Assembly code, and ISA are attached as reference.

TABLE OF CONTENTS

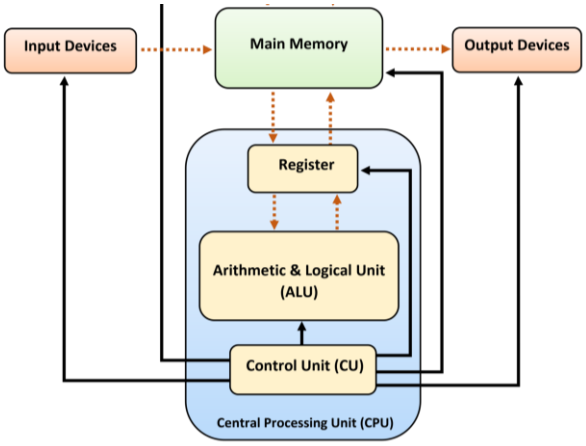
01. INTRODUCTION

1.1 PROCESSOR DESIGN

The objective of this project is to design a Microprocessor which can filter and down sample a given image. The simulations are done using Verilog Hardware Description Language (HDL) and the implementations are done using 'Vivado' version 21.0 along with Altera DE2-115 Education and Development board with Cyclone IV FPGA. This report includes the Microprocessor and CPU design and the test codes that are used.

1.2 CENTRAL PROCESSING UNIT (CPU)

To carry out the instructions of a computer program, the fundamental arithmetic, logical, and input/output (I/O) operations defined by the instructions must be accomplished. The Central Processing Unit (CPU) is the name given to the electronic circuitry in a computer that carries out the aforementioned function. The electronic circuitry within a computer which performs the above-mentioned task is called the *CPU*. The *Processing Unit* and the *Control Unit (CU)* are mainly identified as the Processor of a CPU by distinguishing these core elements of a computer from external components such as main memory and I/O circuitry.



Block diagram of a basic uniprocessor - CPU

1.3 MICROPROCESSOR

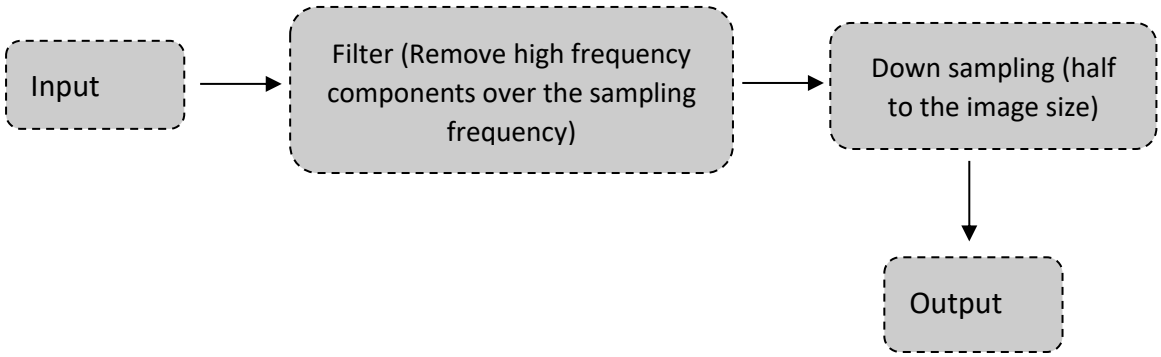
A *microprocessor* on the other hand, is a computer processor which incorporates the functions of a computer's Central Processing Unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits. It is a multipurpose, clock driven, register based, digital-integrated circuit which accepts binary data as input, processes it according to instructions stored in its memory, and provides results as output. Microprocessors contain both combinational logic and sequential digital logic.

1.4 PROBLEM STATEMENT

Design a Microprocessor which can filter high frequencies and down sample a given image to a half of its size.

1.5 PROPOSED SOLUTION BASED ON FPGA

Project is designed to realize a FPGA processor for down sampling an image.

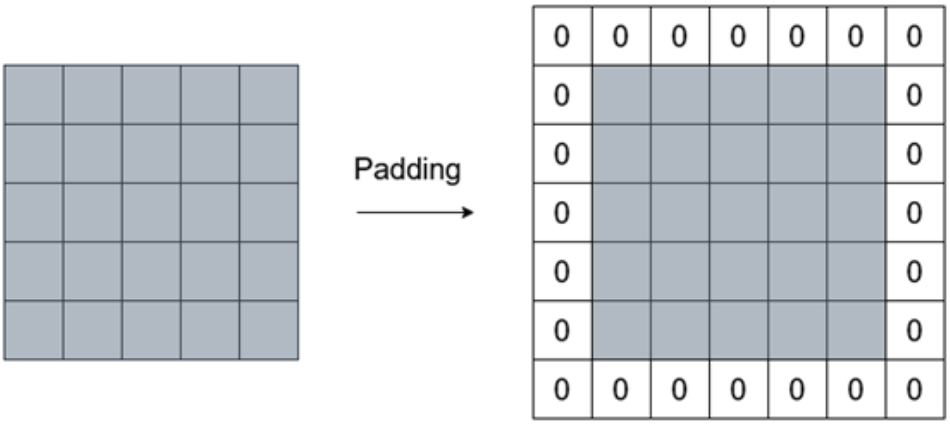


02. ALGORITHM AND DESIGN CONSIDERATIONS

2.1 DATA PROCESSING TECHNIQUES

How do we use pixel data in the 256x256 image and how we get it into DRAM?

The initial step for the given task is to convert the original RGB image into a grayscale image. The next step for the given task is to add zero padding to the image and convert the pixel data contained in the 256x256 (after zero padding) image pixels to hexadecimal values. The values are then rearranged line by line to store the data in a ".mem" file (".mem" is a file format that can be read in processor simulation). This can be done using a Python code. The code snippet used for this task is given in the Appendix '*Python Code*' (01). Data arrangement in the image and the text file is given below.



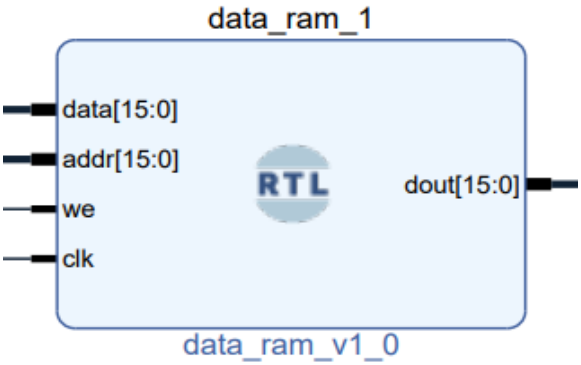
a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y

Image after padding

a
b
c
d
e
f
g
.....
x
y

".mem" file

Then this data in the ".mem" file should be loaded into DRAM. After loading, this data can be stored in the data memory (DRAM). The code snippet used to load and store the numerical data in the ".mem" file is given in Appendix '*Python Code*' (02). The block diagram of the DRAM is given below.



How do we get the final processed data for the output image?

The processed data in the DRAM is overwritten to the input ".mem" file itself. The pixel data values are then rearranged into a single line and the hexadecimal pixel values are converted to an image using Python code. The code snippet used for this task is given in the Appendix '*Python Code*' (03)

2.2 FILTERING (LOW PASS) ALGORITHM

In order to reduce the effects of aliasing caused by the high frequency components in the image, the image needs to be low pass filtered, prior to down sampling it. This part was implemented by using a Gaussian low pass filter.

The following 256×256 image is used to describe the necessity of low pass filtering the image before down sampling it.

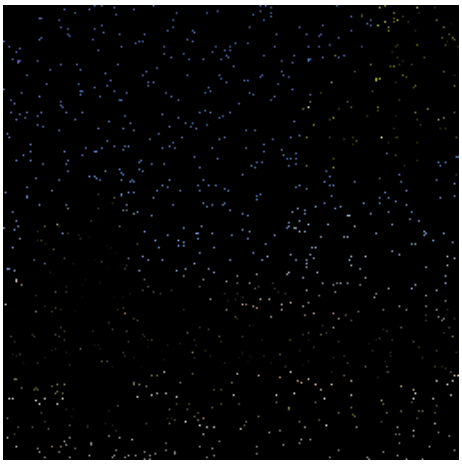


Figure 1: Original RGB Image

After converting the original RGB image in *Figure 1* into a gray scale image by using the "*rgb2gray(image)*" function, white dots which can be considered as high frequency components, are clearly visible. In order to filter this image, a 3×3 kernel that has a half width of size 1 is used. Moreover, the used sigma value that determines the extent of the smoothing is 2/3. The function which was used to create the Gaussian filter Kernel is as follows:

$$Gaussian(x, y) = \frac{1}{2\pi\sigma^2} \times e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

By doing some rounding to the values obtained using the above function for ease of implementation, the following filter kernel was obtained.

1	2	1
2	4	2
1	2	1

Using the above filter kernel, the image was filtered. After down sampling it, following 127×127 image in *Figure 2* was obtained. The image in *Figure 3* is the image which was down sampled without filtering it.

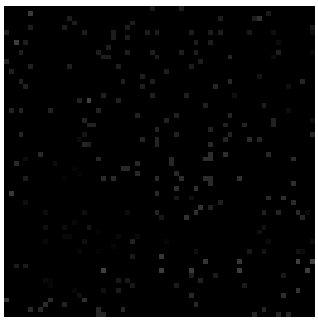


Figure 2: Down Sampled Image after Filtering

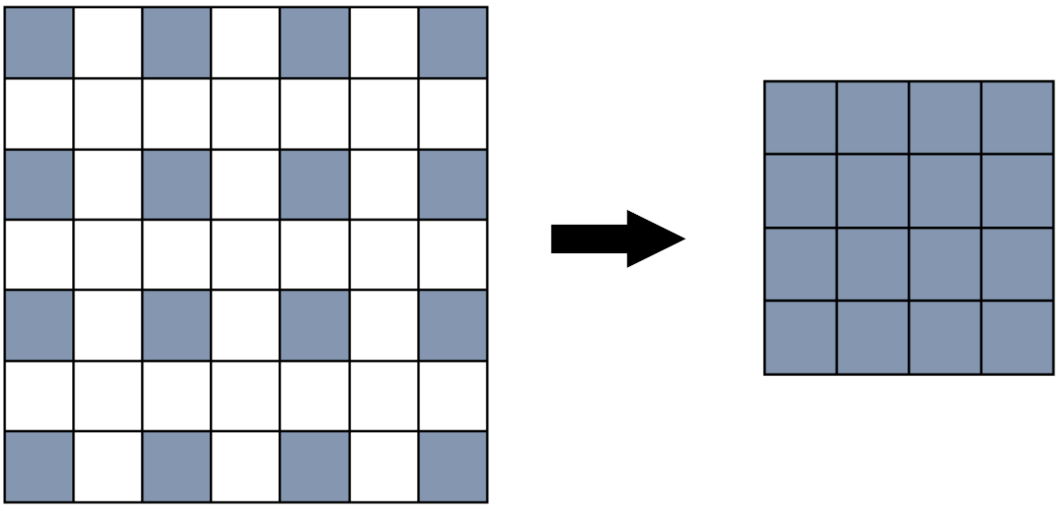
Figure 3: Down Sampled Image without Filtering

By looking at those pictures, it can be clearly seen that the high frequency components in the original image have caused aliasing effects to the image which was down sampled without filtering (*Figure 3*).

2.3 DOWNSAMPLING ALGORITHM

An 256×256 image is down sampled to an 64×64 image. The average intensity value which was computed using 9 nearby pixel values is assigned to the each pixel value in the down sampled image.

A simple down sampling algorithm is used for this processor to avoid errors. According to this algorithm, if the down sampling factor is k, 1 pixel is selected from adjacent k pixels of the filtered image while maintaining the order. This can be easily visualized as follows where the down sampling factor is 4.

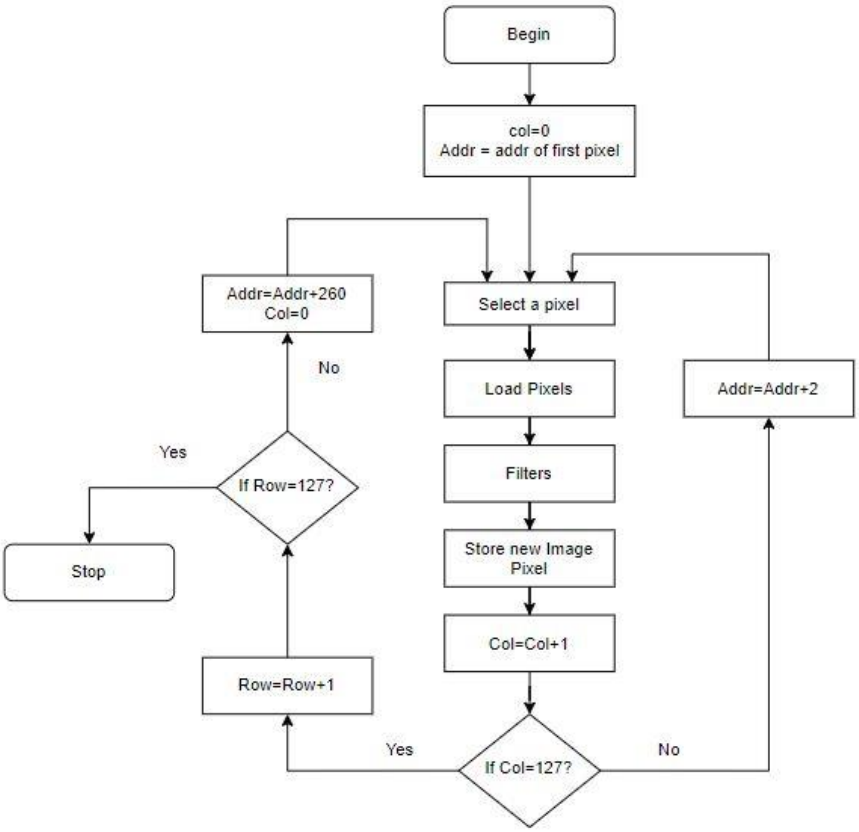


Filtered Image *Down sampled Image*

The image, which was used in this project, Gaussian low pass filtered image, and the down sampled image are provided below.

Original Image *Down sampled Image*

2.4 HIGH LEVEL ALGORITHM



03. DESIGNING A FPGA BASED CUSTOM PROCESSOR

3.1 FPGA – AN OVERVIEW

Field Programmable Logic Array, which is also known as FPGA is a very powerful device used to manipulate digital electronic circuits. It has millions of logic ICs arranged in grid pattern and the user can decide the connections between these ICs according to the requirements. The FPGA board used for this assignment has a chip manufactured by Altera with other peripheral devices. To design the connections between ICs of the FPGA, instructions need to be given from a computer.

3.2 THE ISA – INSTRUCTION SET ARCHITECTURE

3.2.1 INSTRUCTION SET

Instruction	Opcode	State	Operation
START	0x0	START1	PC←0, MAR←0, STR_Pointer←0
FETCH	0x1	FETCH 1	A Bus←PC, ALU Out=A Bus
		FETCH 2	MAR← ALU Out, PC=PC+1, MEM_Read
		FETCH 3	No_Op
LOADIM [Value,Dest Addr]	0x2	LOADIM 1	Decoder A En_op, Decoder C En_op,
		LOADIM 2	Decoder C Dis_op,Decoder A Dis_op, IMEM_Read
		LOADIM 3	Decoder A En_out, ALU_Out=A, ALU_Out=A Bus , Decode C En_out
		LOADIM 4	PC=PC+1 , FETCH
LOAD	0x3	LOAD 1	MEM_Read
		LOAD 2	FETCH
LSHIFT1 [Src Addr, Dest Addr]	0x4	LSHIFT1 1	No_op
		LSHIFT1 2	ALU LSHIFT1
		LSHIFT1 3	FETCH
LSHIFT2 [Src Addr, Dest Addr]	0x5	LSHIFT2 1	No_op
		LSHIFT2 2	ALU LSHIFT1

		LSHIFT2 3	FETCH
RSHIFT [Src Addr, Dest Addr]	0x6	RSHIFT 1	No_op
		RSHIFT 2	ALU RSHIFT
		RSHIFT 3	FETCH
ADD [Addr1 Addr2 Addr3]	0x7	ADD 1	No_op
		ADD 2	ADD signal to ALU
		ADD 3	FETCH
SUB [Addr1 Addr2 Addr3]	0x8	SUB 1	No_op
		SUB 2	SUB signal to ALU
		SUB 3	FETCH
STORE	0x9	STORE 1	MEM_Write
		STORE 2	FETCH
MOVE [Src Addr, Dest Addr]	0xa	MOVE 1	No_op
		MOVE 2	ALU_Out = A
		MOVE 3	FETCH
JUMPNZ [Addr,Value]	0xb	JUMPNZ 1	Decoder A En_op, Decoder B En_op,
		JUMPNZ 2	Decoder A Dis_op,Decoder B Dis_op, IMEM_Read
		JUMPNZ 3	SUB signal to ALU
		JUMPNZ 4	JUMP en , PC = PC+1
		JUMPNZ 5	Decoder A Dis_out,Decoder B Dis_out, IMEM_Read
		JUMPNZ 6	FETCH
MAR_INCREMENT	0xc	MAR_INC 1	en MAR_inc
		MAR_INC 2	FETCH
COL_INCREMENT	0xd	COL_INC 1	en COL_inc
		COL_INC 2	FETCH
ROW_INCREMENT	0xe	MAR_INC 1	en MAR_inc, en COL_reset

		MAR_INC 2	FETCH
END	0xf	END 1	dis_clock

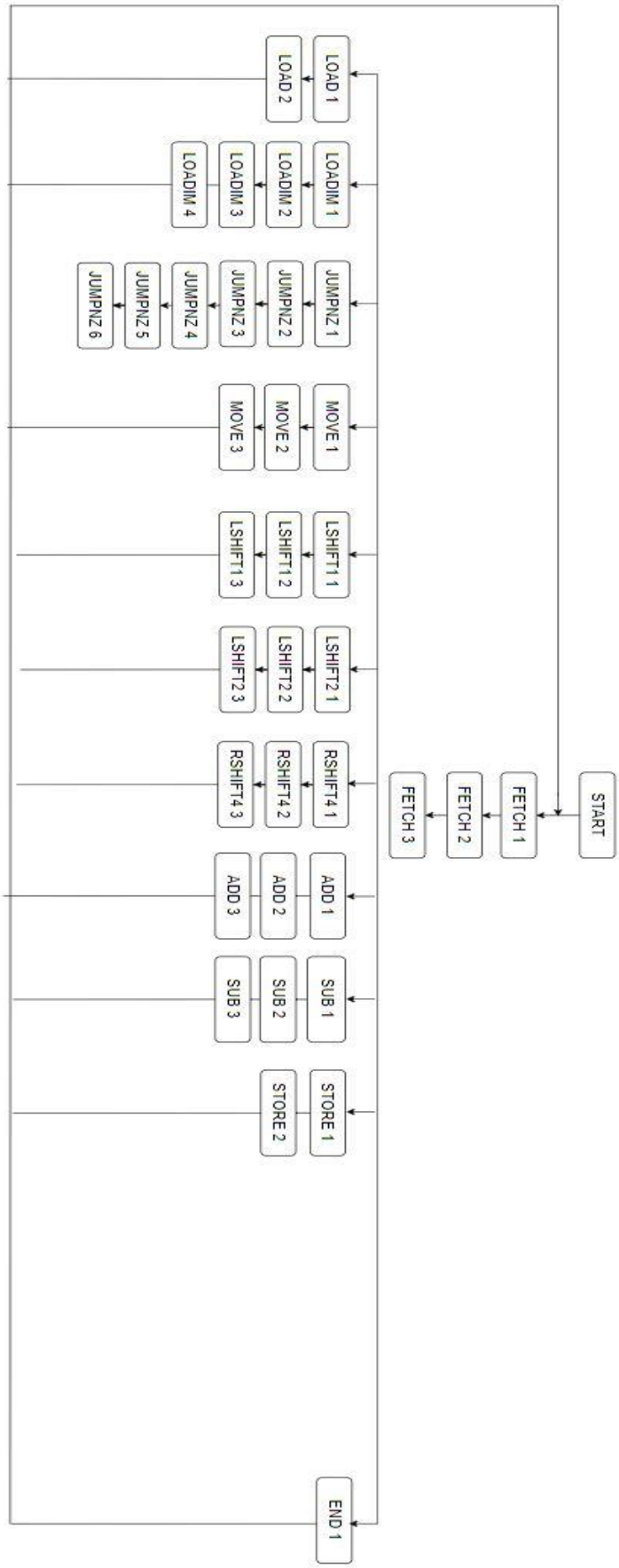
3.2.2 MICRO INSTRUCTIONS

No	Instruction	Hex Value
1	START	0x0000
2	LOADIM 258, MAR	0x2304 0x0102
3	LOAD	0x3005
4	MOVE MDR, PR1	0xA506
5	MAR_INCREMENT	0xC000
6	LOAD	0x3005
7	MOVE MDR, PR2	0xA507
8	MAR_INCREMENT	0xC000
9	LOAD	0x3005
10	MOVE MDR, PR3	0xA508
11	ADD PR1, PR3, PR1	0x7686
12	LSHIFT1 PR2, R2	0x470C
13	ADD R2, PR1, R1	0x7C6B
14	LOADIM 254, R2	0x230C 0x00FE
15	ADD MAR, R2, MAR	0x74C4
16	LOAD	0x3005
17	MOVE MDR, PR1	0xA506
18	MAR_INCREMENT	0xC000
19	LOAD	0x3005
20	MOVE MDR, PR2	0xA507
21	MAR_INCREMENT	0xC000

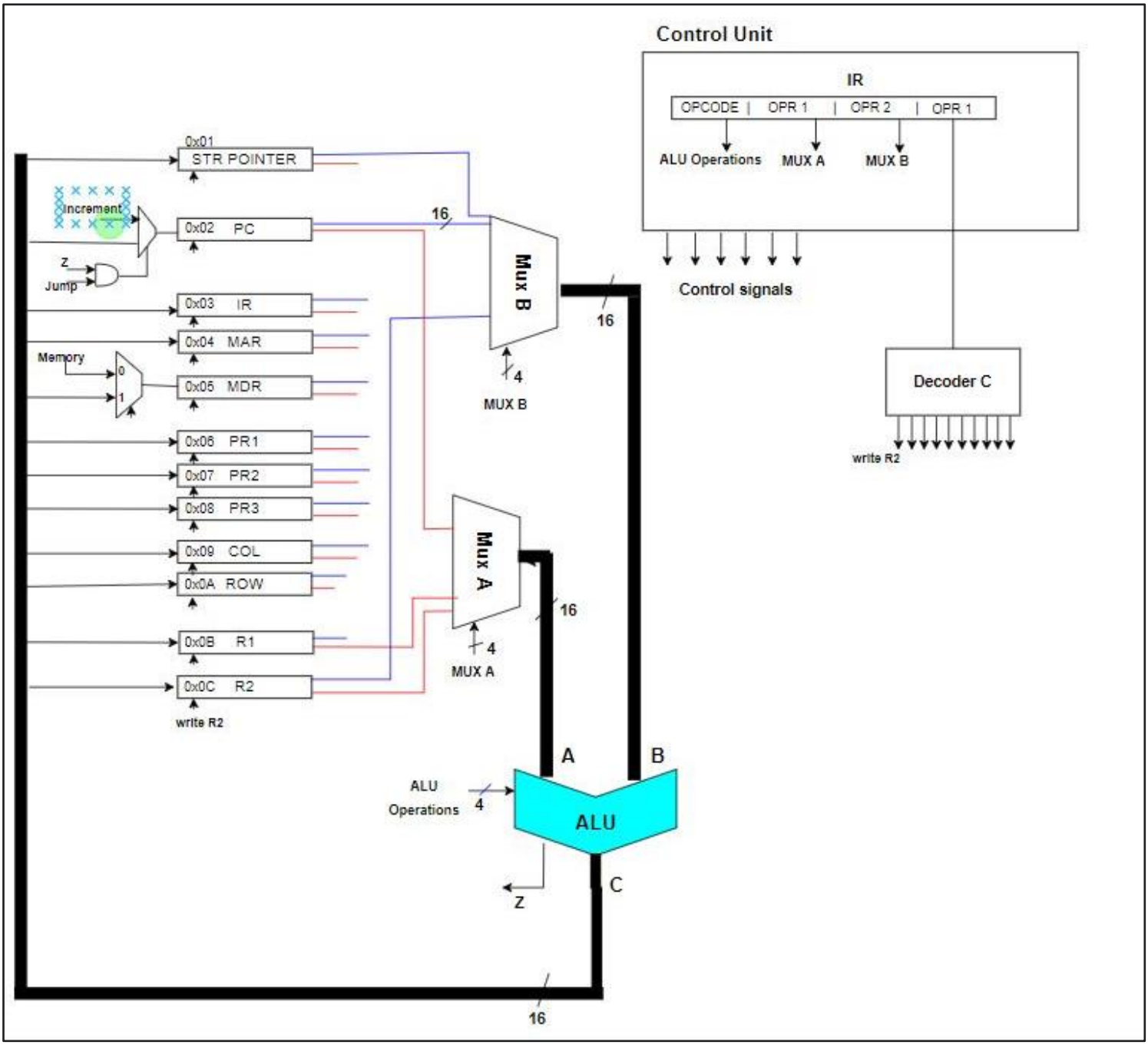
22	LOAD	0x3005
23	MOVE MDR, PR3	0xA508
24	ADD PR1, PR3, PR1	0x7686
25	LSHIFT1 PR1, PR1	0x4606
	LSHIFT2 PR2, PR2	0x5707
26	ADD PR1, PR2, R2	0x767C
27	ADD R1,R2, R1	0x7BCB
28	LOADIM 254, R2	0x230C 0x00FE
29	ADD MAR, R2, MAR	0x74C4
30	LOAD	0x3005
31	MOVE MDR, PR1	0xA506
32	MAR_INCREMENT	0xC000
33	LOAD	0x3005
34	MOVE MDR, PR3	0xA508
35	ADD PR1, PR3, PR1	0x7686
36	LSHIFT1 PR2, R2	0x470C
37	ADD R2, PR1, R2	0x7C6C
38	RSHIFT R1, R1	0x6B0B
39	MOVE R1, MDR	0xAB05
40	MOVE MAR, PR1	0xA406
41	MOVE STR, MAR	0xA104
42	STORE	0x9000
43	LOADIM 1, R1	0x230B 0x0001
44	ADD STR_POINTER, R1, STR_POINTER	0x71B1
45	COL_INCREMENT	0xd000
46	MOVE PR1, MAR	0xA604
47	LOADIM 512, R1	0x230B 0x0200

48	SUB MAR, R1, MAR	0x84B4
49	JUMPNZ COL, 126	0xB390 0x007E
50	ROW_INCREMENT	0xE000
51	LOADIM 258, R1	0x230B 0x0102
52	ADD MAR, R1, MAR	0x74B4
53	JUMPNZ ROW, 126	0xB3A0 0x007E
54	STOP	0xF000

3.2.3 STATE DIAGRAM



3.2.4 DATA PATH



3.3 MODULES AND COMPONENTS

3.3.1 ALU - ARITHMETIC AND LOGIC UNIT

ALU is capable of handling 7 operations which are listed below.

Code	**Operation**	**Code**	**Operation**
0000	dout = operand1	0100	dout = operand1 << 2
0001	dout = operand1 + operand2	0101	dout = operand1 >> 4
0010	dout = operand1 - operand2	0110	dout = operand1 + 1
0011	dout = operand1 << 1	default	dout = 16'hzzzz

3.3.2 CU - CONTROL UNIT

Control unit emits all the control signals to control register operations, memory read, write operations, decoder and multiplexer operations. Input for the Control unit is the IR register value and outputs are listed below.

Control signal	Purpose
reset	reset PC, STR_POINTER and MAR
en_decAop	Enable Mux_A operation
en_decBop	Enable Mux_B operation
en_decCop	Enable Decoder_C operation
en_decAout	Enable output of the Mux_A
en_decBout	Enable output of the Mux_B
en_decCout	Enable output of the Deocder_C
alu_ctrl [3:0]	Control signals for the ALU
dmem_read	Read data from the data memory
dmem_write	Write data to the data memory
imem_read	Read data from the instruction memory
pc_inc	Increment PC by 1

mar_inc	Increment the MAR by 1
col_zero	set COL register to 0
col_inc	Increment the COL register value by 1
row_inc	Increment the ROW register value by 1
jump	Enable branching operations at the PC
clock_en	Enable the clock

3.3.3 DATA MEMORY (DRAM)

Data memory contains 65536 (256 x 256) , 16 bit memory locations in sequential order. Address bus contains 16 bits to address all the memory locations.

3.3.4 INSTRUCTION MEMORY (IRAM)

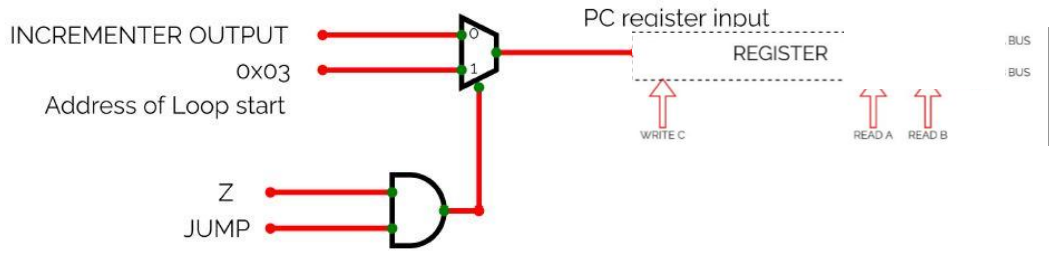
Instruction memory contains 62, 16 bit memory locations.

3.3.5 REGISTERS WITHOUT INCREMENT

R1, R2, PR1, PR2, PR3, MDR are 16 bit registers and don't have any increment options within the register. They are connected to A and B buses through MUX_A, MUX_B respectively. Values can be loaded from C bus if the writing command is high.

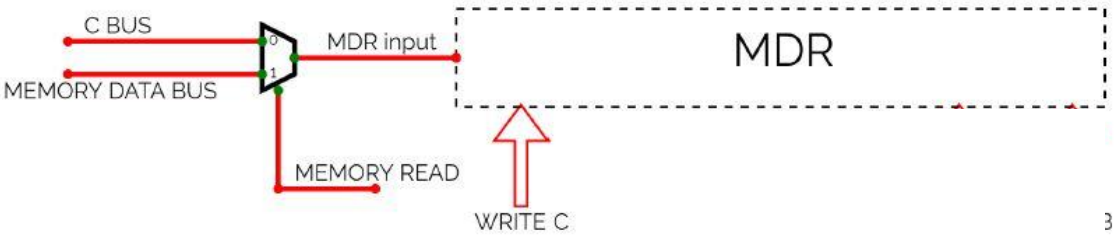
3.3.6 REGISTERS WITH INCREMENT

PC : 16 bit register



Z value will be 1 when the COL=127. When the JUMP instruction occurs Mux will output 0x03 which is the beginning of the next loop and proceed through the next row of pixels. Otherwise the PC will increment by 1 after each instruction completes.

MDR : 16 bit register



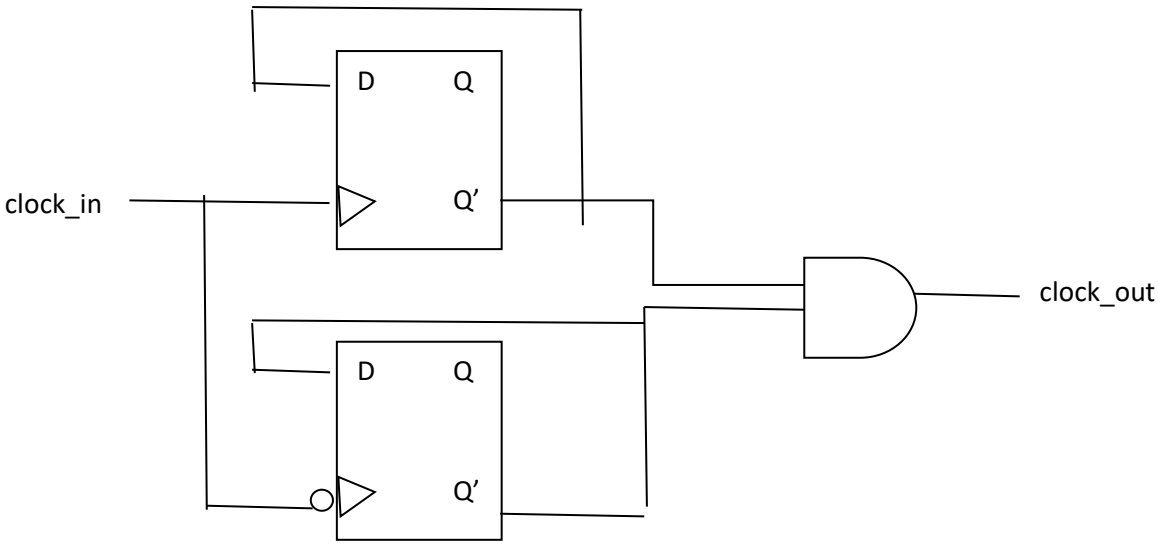
When the memory read control signal is high, data memory values will be written into the MDR. Otherwise, data in the C bus will be written into the MDR.

3.3.7 BUS

Bus consists of a set of wires or connectors which transport data. It is allowed to bundle several binary signals and address them with a single name. Code was implemented for a 16 bit data bus.

3.3.8 CLOCK DIVIDER

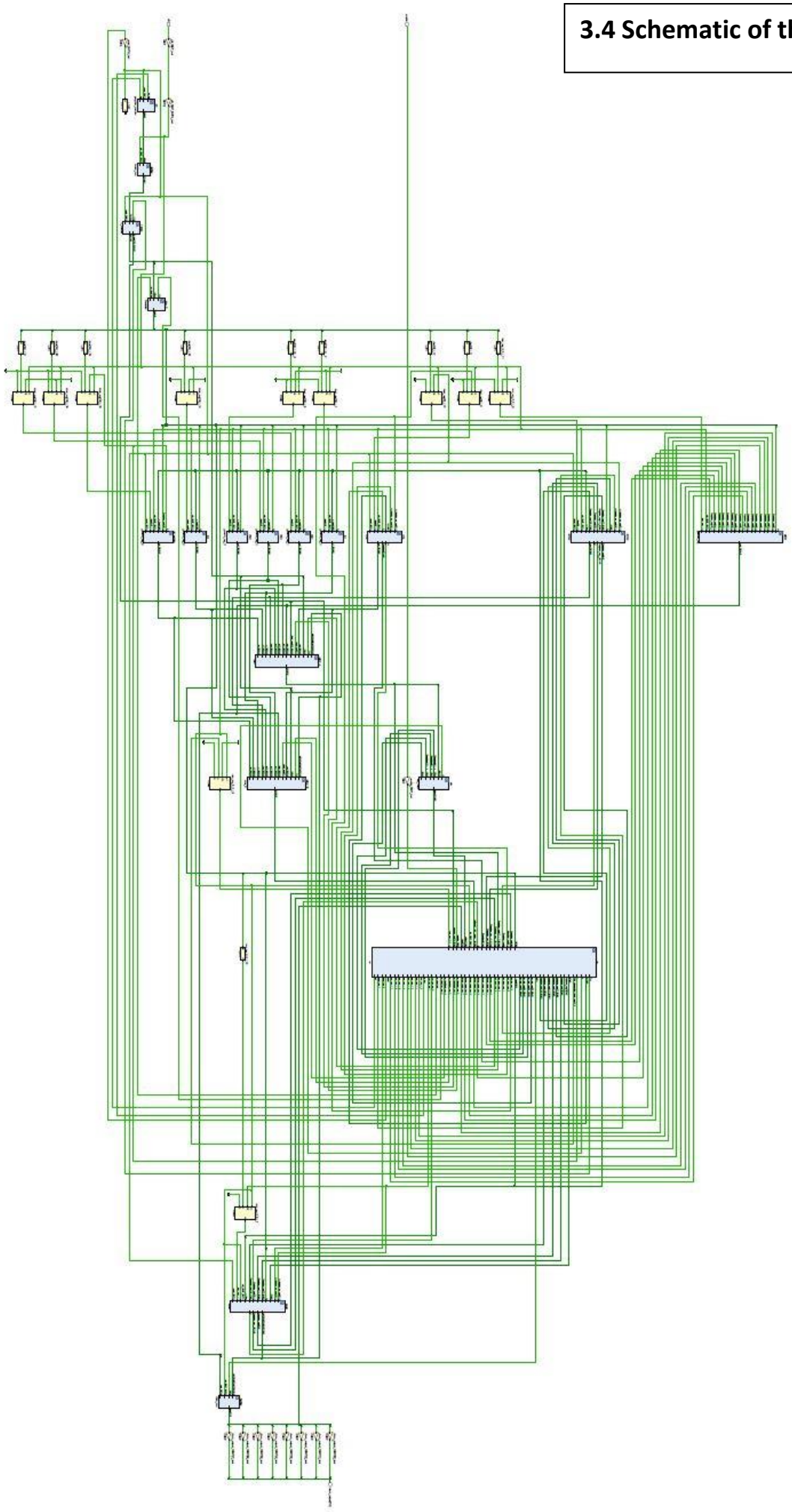
Simple logic diagram of clock divider with the division factor of 2.



Frequency of the clock_out = frequency of clock_in / divisor.

code was implemented with divisor parameter 28.

3.4 Schematic of the processor



3.5 RESOURCE UTILIZATION

Following table shows the resources utilization of the processor.

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Block RAM Tile (140)	Bonded IOB (200)	BUFGCTRL (32)
N processor_combined	391	484	15	16.5	10	2
alu (alu_16bit)	0	0	0	0	0	0
COL (COL)	2	32	0	0	0	0
cu (cu)	177	28	15	0	0	0
dec_C (Decoder)	4	24	0	0	0	0
DMEM (data_ram)	0	0	0	16	0	0
I_REG (ir_module)	0	40	0	0	0	0
IMEM (imem_ram)	0	0	0	0.5	0	0
MAR (MAR)	5	32	0	0	0	0
MDR (generic_reg)	12	32	0	0	0	0
MUX_A (mux)	66	20	0	0	0	0
MUX_B (mux_0)	64	20	0	0	0	0
PC (program_counter)	1	21	0	0	0	0
PR1 (generic_reg_1)	8	32	0	0	0	0
PR2 (generic_reg_2)	8	32	0	0	0	0
PR3 (generic_reg_3)	8	32	0	0	0	0
R1 (generic_reg_4)	8	32	0	0	0	0
R2 (generic_reg_5)	8	32	0	0	0	0
ROW (ROW)	3	32	0	0	0	0
str_pointer (generic_reg_6)	8	32	0	0	0	0

4.0 RESULTS ANALYSIS

We have compared our Vivado output with a software generated downsampled image and calculate the error. Results occurred is as follows.

```
Mean squared Error: 1.736804145483291
```

```
Mean absolute Error: 0.9189890879781759
```

This error occurs due to the lack of floating-point arithmetic support in our processor hence all the floating points are neglected. Implementing a floating-point arithmetic unit in the ALU is a hard task which is impossible to achieve within the given time.

APPENDIX

"Python code" for image pre processing

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
from skimage import color
from skimage import io
import binascii
import numpy as np

#Rescale image to 254x254 and padd in all directions
img = cv2.imread('/content/stewie.jpg', 0)
cv2_imshow( img)

BLACK = [0,0,0]

img_stretch = cv2.resize(img, (254, 254))
img_stretch = cv2.copyMakeBorder(img_stretch, 1, 1, 1,
1,cv2.BORDER_CONSTANT,value=BLACK)
print()
print("******** Original Image *********")
cv2_imshow( img_stretch)

cv2.waitKey(0)
cv2.destroyAllWindows()


#Convert image to hexadecimal format
Hex_coverted_Img=binascii.hexlify(img_stretch)
print(len(Hex_coverted_Img))

for i in range(65537):
  Hex_coverted_Img=str(Hex_coverted_Img)
  Con_part=Hex_coverted_Img[2*i:2*(i+1)]

  if i!=0:
    with open("Hex_Img.txt", "a") as f:
      f.write(Con_part+'\n')
print(Con_part)



# Read the output file of the processor and create the image
```

```
with open('/content/output.txt') as file:
    data = file.read()

print(type(data))
data=data.upper()
data=data.strip()
data=data.replace(' ', '')
data=data.split('\n')

New_array=[]

for i in data:
  New_array.append(int(i,16))

New_array=np.array(New_array[:16129])

img=np.reshape(New_array,(127,127))
cv2_imshow(img)
```

VERILOG CODES

```
module alu_16bit #(
      parameter DWIDTH = 16
   )(
   input [DWIDTH-1:0] operand1,
   input [DWIDTH-1:0] operand2,
   input [3:0] operation,
   input clk,
   output reg [DWIDTH-1:0] dout,
   output reg Z
   );



   always@(*) begin
      case(operation)
         4'b0000 : dout = operand1;
         4'b0001 : dout = operand1 + operand2;
         4'b0010 : dout = operand1 - operand2;
         4'b0011 : dout = operand1 << 1;
         4'b0100 : dout = operand1 << 2;
         4'b0101 : dout = operand1 >> 4;
         4'b0110 : dout = operand1 + 1;
         default : dout = 16'hzzzz;
```

```
        endcase
    end
    always@(dout)begin
      if(dout == 0)
        Z <= 0;
      else
        Z <= 1;
    end
endmodule
```

```
module cu #(parameter BUS_WIDTH =16,
            parameter OPCODE_LEN = 4,
            parameter ADDR_AW = 4,
            parameter ADDR_BW = 4,
            parameter DESTW = 4)(
    input [BUS_WIDTH-1:0] ir,
    input clk,
   input enable,
    output reg reset,
    output reg en_decAop,
    output reg en_decBop,
    output reg en_decCop,
    output reg en_decAout,
    output reg en_decBout,
    output reg en_decCout,
    output reg [3:0]  alu_ctrl,
    output reg dmem_read,
    output reg dmem_write,
    output reg imem_read,
    output reg pc_inc,
    output reg mar_inc,
    output reg col_zero,
    output reg col_inc,
    output reg row_inc,
    output reg jump,
    output reg clock_en,
    output reg done
);
reg [OPCODE_LEN-1:0] opcode;
reg [ADDR_AW-1:0] addr_A;
reg [ADDR_BW-1:0] addr_B;
reg [DESTW-1:0] addr_dest;


/*       4'b0000 : out = operand1;
         4'b0001 : out = operand1 + operand2;
         4'b0010 : out = operand1 - operand2;
         4'b0011 : out = operand1 << 1;
```

```
        4'b0100 : out = operand1 << 2;
        4'b0101 : out = operand1 >> 4;
        4'b0110 : out = operand1 + 1; */

integer state = 0;

always@(posedge clk)begin
    opcode <= ir[BUS_WIDTH-1:BUS_WIDTH-OPCODE_LEN];
    addr_A <= ir[BUS_WIDTH-1-OPCODE_LEN: BUS_WIDTH-OPCODE_LEN-
ADDR_AW];
    addr_B <= ir[BUS_WIDTH-OPCODE_LEN-ADDR_AW-1:BUS_WIDTH-
OPCODE_LEN-ADDR_AW-ADDR_BW];
    addr_dest <= ir[BUS_WIDTH-OPCODE_LEN-ADDR_AW-ADDR_BW-1:BUS_WIDTH-
OPCODE_LEN-ADDR_AW-ADDR_BW-DESTW];
end


always@(posedge clk)
  if (enable)  begin
    clock_en <=1;
  //begin
    case(state)
        // START
      'h0:begin
        reset <=1;        //ACTIVE HIGH RESET
        state <= state +1;
        en_decAop <=0;
        en_decBop <=0;
        en_decCop <=0;
        en_decAout <= 0;
        en_decBout <= 0;
        en_decCout <= 0;
        alu_ctrl <=0;
        dmem_read <= 0;
        dmem_write <= 0;
        imem_read <= 0;
        pc_inc <= 0;
        mar_inc <=0;
        col_zero <=0;
        col_inc <= 0;
        row_inc <= 0;
        jump <= 0;
        done <= 0;
        end

      // FETCH
      'h01:begin
//          reset <= 0;
//          en_decAop <= 1;
```

```
//              en_decAout <= 1;
//              alu_ctrl <= 4'b0000;
//              en_decCop <= 1;
//              en_decCout <=1;
//              state <= state +1;
                reset <=0;
                pc_inc <=0;
                imem_read <=1;
                state <= state +1;
              end
            'h02:begin
              pc_inc <=1;
              imem_read <=0;
              state <= state + 1;
              end
            'h03:begin
              pc_inc <= 0;
              imem_read <= 0;
              state <= state + 1;
              //state <= opcode;
              end

            //SELECTING THE INSTRUCTION STATE FROM OPCODE
            'h04:begin
              case(opcode)
                'h0:state<='h00;   //START
                'h1:state<='h01;   //FETCH
                'h2:state<='h05;   //LOADIM
                'h3:state<='h09;   //LOAD
                'h4:state<='h0b;   //LSHIFT1
                'h5:state<='h0e;   //LSHIFT2
                'h6:state<='h11;   //RSHIFT4
                'h7:state<='h14;   //ADD
                'h8:state<='h17;   //SUB
                'h9:state<='h1a;   //STORE
                'ha:state<='h1c;   //MOVE
                'hb:state<='h1f;   //JUMPNZ
                'hc:state<='h25;   //MAR INCREMENT
                'hd:state<='h27;   //COL INCREMENT
                'he:state<='h29;   //ROW INCREMENT
                'hf:state<='h2b;   //END




                endcase
              end

            // LOAD IMMEDIATE~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
'h05:begin
    en_decAop <= 1;
    en_decCop <= 1;
    //imem_read <= 1;
    state <= state + 1;
    end
'h06:begin
    imem_read <= 1;
    en_decAop <= 0;
    en_decCop <= 0;
    state <= state + 1;
    end

'h07:begin
    imem_read <=0;
    en_decAout <= 1;
    en_decCout <= 1;
    alu_ctrl <= 4'b0000;
    state <= state + 1;
    end
'h08:begin
    en_decAout <= 0;
    en_decCout <= 0;
    pc_inc <= 1;
    state <= 1;
    end

//LOAD~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h09:begin
    dmem_read <=1;
    en_decCop <=1;
    en_decCout <=1;
    state <= state +1;
    end
'h0a:begin
    dmem_read <=0;
    en_decCop <=0;
    en_decCout <=0;
    state <=1;
    end

//LSHIFT1~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h0b:begin
   state <= state +1;
   en_decAop <= 1;
   en_decCop <= 1;
   end
'h0c:begin
    alu_ctrl <= 4'b0011;
```

```
    state <= state +1;
    en_decAop <= 0;
    en_decAout <= 1;
    en_decCop <= 0;
    en_decCout <= 1;
    end
'h0d:begin
    alu_ctrl  <= 0;
    en_decAout <=0;
    en_decCout <=0;
    state <= 1;
    end
//LSHIFT2~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h0e:begin
     state <= state + 1;
     en_decAop <= 1;
     en_decCop <= 1;
     end
'h0f:begin
    alu_ctrl <= 4'b0100;
    state <= state +1;
    en_decAop <= 0;
    en_decCop <= 0;
    en_decAout  <= 1;
    en_decCout <= 1;
    end
'h10:begin
    alu_ctrl  <= 0;
    state <= 1;
    en_decAout  <= 0;
    en_decCout <= 0;
    end

//RSHIFT4~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h11:begin
    state <= state +1;
    en_decAop <=1;
    en_decCop  <= 1;
    end
'h12:begin
    alu_ctrl <= 4'b0101;
    state <= state +1;
    en_decAop <= 0;
    en_decCop <= 0;
    en_decAout <= 1;
    en_decCout <= 1;
    end
'h13:begin
    alu_ctrl <=0;
```

```
    state <= 1;
    en_decAout <= 0;
    en_decCout <= 0;
    end

//ADD~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h14:begin
    en_decAop <= 1;
    en_decBop <= 1;
    en_decCop <= 1;
    state <= state +1;
    end
'h15:begin
    alu_ctrl <= 4'b0001;
    state <= state + 1;
    en_decAop <= 0;
    en_decAout <=1;
    en_decBop <= 0;
    en_decBout<=1;
    en_decCop <= 0;
    en_decCout <=1;
    end
//'h16:state <= state+1;
'h16:begin
    alu_ctrl  <= 0;
    en_decAout <= 0;
    en_decBout <= 0;
    en_decCout <= 0;
    state <= 1;
    end

//SUB~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h17:begin
    state <= state + 1;
    en_decAop <= 1;
    en_decBop <= 1;
    en_decCop <= 1;
end
'h18:begin
    alu_ctrl <= 4'b0010;
    state <= state + 1;
    en_decAop <= 0;
    en_decAout <=1;
    en_decBop <= 0;
    en_decBout<=1;
    en_decCop <= 0;
    en_decCout <=1;
    end
'h19:begin
```

```
    alu_ctrl <= 0;
    en_decAout <= 0;
    en_decBout <= 0;
    en_decCout <= 0;
    state <= 1;
    end

//STORE~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h1a:begin
    dmem_write <= 1;
    state <= state + 1;
    end
'h1b:begin
    dmem_write <= 0;
    state <= 1;
    end

//MOVE~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h1c:begin
     state <= state + 1;
     en_decAop <=1;
     en_decCop <=1;
     end
'h1d:begin
    alu_ctrl <= 4'b0000;
    state <= state + 1;
    en_decAop <=0;
    en_decCop <=0;
    en_decAout <=1;
    en_decCout <=1;
    end
'h1e:begin
    state <= 1;
    en_decAout <=0;
    en_decCout <=0;
    end

//JUMPNZ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'h1f:begin
    en_decAop <= 1;
    en_decBop <= 1;     //Now addresses are saved in mux
    state <= state + 1;
    end
'h20:begin
    en_decAop <= 0;
    en_decBop <= 0;
    imem_read  <= 1;
    state <= state + 1;
    end
```

```
'h21:begin
   //jump <= 1;
   en_decAout <=1;
   en_decBout <=1;
   imem_read  <=0;
   alu_ctrl <= 4'b0010;
   state <= state + 1;
   end

'h22:begin
    jump <=1;
    pc_inc <=1;
    state <= state + 1;
    end
'h23:begin
    en_decAout <=0;
    en_decBout <=0;
    imem_read <= 1;
    state <= state+1;
    end
'h24:begin
    state <= 1;
    jump <=0;
    imem_read <=0;
    end

//////////////////////////
//MAR INCREMENT
'h25:begin
   mar_inc <=1;
   state <= state + 1;
   end

'h26:begin
   mar_inc <= 0;
   state <= 1;
end

//COL INCREMENT
'h27:begin
   col_inc <=1;
   state <= state + 1;
   end
'h28:begin
   col_inc <= 0;
   state <= 1;
end

//ROW INCREMENT
```

```
          'h29:begin
             row_inc <=1;
             state <= state + 1;
             col_zero <=1;
             end

          'h2a:begin
             row_inc <= 0;
             col_zero <= 0;
             state <= 1;
             state <= 1;
           end

          //END
          'h2b:begin
             clock_en <=0;
             dmem_write <=16'hzzzz;
             done <= 1;
          end

           //////////////////////////
       endcase
   //end
end
endmodule
```

```
module data_ram #(parameter DWIDTH = 16, parameter ADDR_WIDTH= 16, parameter
DEPTH = 65535)(
 input [DWIDTH-1:0] din,
 input [ADDR_WIDTH-1:0] addr,
 input done,
 input we,clk,
 output reg [DWIDTH-1:0] dout
 );

// parameter DEPTH = 1<< ADDR_WIDTH;
 reg [7:0] ram [DEPTH:0];
 reg [ADDR_WIDTH-1:0] index;
 integer f;
 //reg [ADDR_WIDTH:0] addr_reg;
 initial begin
    $readmemh("dmem.mem",ram); // read file from INFILE
    f = $fopen("C:\\Users\\menuw\\Documents\\Processor-Design\\output.txt","w");
    index = 0;
end

always@(posedge clk)begin
     if ( done  == 1 )begin
```

```
      //for (i = 0; i<16129; i=i+1) begin
        //$display("OUT %b", dram[i]);
        index <= index+1;
        $fwrite(f,"%h\n",ram[index] ); //NOT DRAM < DMEM
        //end
      end
      if(index == 16129)begin
       $fclose(f);
       $finish;
      end
end
always @ (posedge clk)
 begin
  if(we === 1)begin
   ram[addr]<=din;
   dout <= 16'hzzzz;
  end
  else
   //addr_reg<=addr;
   ram[addr] <= ram[addr];
   dout <= ram[addr];
 end

// assign dout = (~we) ? ram[addr] : 0;
Endmodule
```

```
module imem_ram #(parameter DWIDTH = 16, parameter ADDR_WIDTH= 16)(
 input [DWIDTH-1:0] data,
 input [ADDR_WIDTH-1:0] addr,

 input we,clk,
 output [DWIDTH-1:0] dout
 );

 parameter DEPTH = 1<< ADDR_WIDTH;
 reg [255 :0] ram [DEPTH:0];
 //reg [ADDR_WIDTH:0] addr_reg;

 always @ (posedge clk)
 begin
  if(we)
   ram[addr]<=data;
  else
   //addr_reg<=addr;
   ram[addr] <= ram[addr];
 end

 assign dout = (~we) ? ram[addr] : 0;
```

```
endmodule
```

```
module generic_reg(input clk, reset,writeC,
      input [15:0] D,
      output reg[15:0] A,
      output reg[15:0] B);

      reg[15:0] Register;
      always @(posedge clk, posedge reset)
         if (reset)
            Register <= 16'b0;
         else if (writeC) begin
            Register <= D;//if load is high
            A <= 16'hzzzz;
            B <= 16'hzzzz;
         end
         else begin
            A <= Register;//Read to the A Bus
            B <= Register;//Read to the B Bus
         end
endmodule
```

```
module PC(
     input     reset,
     input clk,
     input inc,
   input      jump,
   input      Z,
    output reg  [15:0]  pc_result
   );
   wire sel;
   assign sel = jump & Z;

   always @(posedge clk)
   begin
     if (reset == 1)
               pc_result <= 16'b0;
     else begin
       case(sel)
          1'b0:begin
              if(inc)
                pc_result <= pc_result + 1;
              else if (inc ==0)
                pc_result <= pc_result;
              end

          1'b1:pc_result <= 16'h0003; // LOOP Start PC number
```

```
        endcase
                end
              end
              //$display("PC=%b",PC_result);
Endmodule
```

```
module MDR(input clk, rst, readA,readB,writeC,
      input [15:0] D,
      output reg[15:0] A,
      output reg[15:0] B);

      reg[15:0] Reg_MDR;
      always @(posedge clk, posedge rst)
         if (rst)
            Reg_MDR <= 16'b0;
         else if (writeC)
            Reg_MDR <= D;//if load is high      A and C both active at once???? else if??
         else if (readA)
            A <= Reg_MDR;//Read to the A Bus
         else if (readB)
            B <= Reg_MDR;//Read to the B Bus
         else
            Reg_MDR <= Reg_MDR;//Refreshing.... Is this needed??????
Endmodule
```

```
module MDR_Mux(
      input dmem_read,
      input [15:0] C_Bus,
      input [15:0] Mem_Data_Bus,
      output /*reg*/[15:0] MDR_in);

   assign MDR_in = (dmem_read) ? Mem_Data_Bus[7:0]:C_Bus;
      /*always @(posedge dmem_read)
            MDR_in <= Mem_Data_Bus[7:0];
      always@(negedge dmem_read)
            MDR_in <= C_Bus;//if load is high
      */
Endmodule
```

```
module ir_module(
input    [15:0] din,
input    rst,
input    clk,
input    write_en,
output [3:0] addrA,
```

```
output [3:0] addrB,
output [3:0] addrC,
//output [3:0] opcode,
output reg  [15:0] A,
output reg  [15:0] B
);

reg [15:0] IR;

// Mux sel outputs , address fields of the instruction

assign addrA = IR[11:8];
assign addrB = IR[7:4];
assign addrC = IR[3:0];

// opcode output
//assign opcode = IR[15:12];

//Rest of the logic accomodates for reading immediate data

     always @(posedge rst)
      IR <= 16'b0;

   always@(posedge write_en) begin       //write at the posedge of write_en?
                                IR = din;
                                A = IR;
            B = IR;
            //$display("IR=%h",IR);
                        end
                                    //din is from instruction memory, if not in reset the

Endmodule
```

```
module Decoder(
   input clk,
   input [3:0]sel,
   input EN_OP,
   input EN_OUT,
   output str_pointer,
   output mar,
   output mdr,
   output pr1,
   output pr2,
   output pr3,
   output col,
   output row,
   output r1,
   output r2
```

```
    );

    reg [3:0]temp_sel;
    reg [11:0] temp;
    reg [11:0] out;
    assign str_pointer = out[0];//1
    assign mar = out[3];       //4
    assign mdr = out[4];       //5
    assign pr1 = out[5];       //6
    assign pr2 = out[6];       //7
    assign pr3 = out[7];       //8
    assign col = out[8];       //9
    assign row = out[9];       //a
    assign r1 = out[10];       //b
    assign r2 = out[11];       //c

    always@(posedge EN_OUT)begin
        case (temp_sel)
        4'b0001: out=12'b000000000001;
        4'b0100: out=12'b000000001000;
        4'b0101: out=12'b000000010000;
        4'b0110: out=12'b000000100000;
        4'b0111: out=12'b000001000000;
        4'b1000: out=12'b000010000000;
        4'b1001: out=12'b000100000000;
        4'b1010: out=12'b001000000000;
        4'b1011: out=12'b010000000000;
        4'b1100: out=12'b100000000000;
        default: out=12'b000000000000;
       endcase
    end
    always@(posedge EN_OP)begin
       if (EN_OP==1)
          temp_sel = sel;
       end
    //always@(posedge EN_OUT)
          //out = temp;
    always@(negedge EN_OUT)
          out = 12'h0000;
endmodule
```

```
module processor_tb(

   );
 reg enable;
 reg clk;
processor_combined downsampling_processor(.enable(enable), .clk(clk));
```

```
initial begin
    enable = 0;
    #20;
    enable =1;
end

//CLOCK GENERATION
initial begin
    clk = 1;
    forever begin
        #5 clk = ~clk;
        end
    end
```