Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN3030 Circuits and Systems Design

Processor Design – Project Report

T.M. Aqeel	180039C
D.N.R. Dissanayake	180153U
G.K.H. Nadeeshan	180410G
R.T.N. Rathnayake	180538F

06th July 2022

Table of Contents

Abstract

Custom processors are widely used in the electronic world. Those processors can do a specific task with better performance and good reliability. This report contains a detailed discussion of the processor designed for the given task with designing steps and the simulated outputs of the processor. The method and the theoretical approaches are included as well as Verilog code, Assembly code, and ISA are attached as reference.

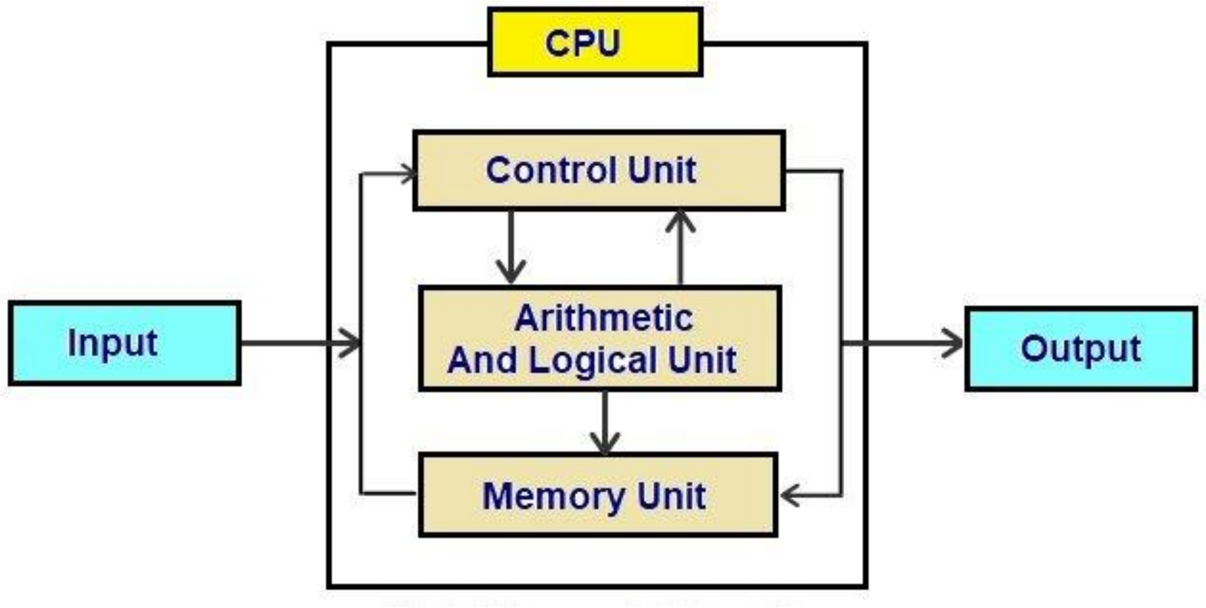
1. Introduction

1.1 Processor Design

The purpose of this project is to filter and down sample a 256*256 px image to 128*128 px image. A task specific processor can do this task with a good reliability. The designed processor can be simulated using Verilog Hardware Description Language (HDL) and the implementations are done using Vivado 2018.2. This report contains the Microprocessor and CPU design, the test code that are used to test the filtering and down sampling of a given image.

1.2 Central Processing Unit (CPU)

The central processing unit includes a control unit, arithmetic and logical unit, and memory unit to perform a program. These electronic elements need to work correctly to carry out the instructions of a computer program. 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.



1.3 Microprocessor

A microprocessor is a single integrated circuit (IC) which is a multipurpose, clock-driven, register-based, digital-integrated circuit that takes binary inputs, and processes them according to the instruction stored in its memory. And it provides results as output. Microprocessors can be designed to perform a special task not to do multiple tasks. These customizations will lead to achieve more reliable results with good efficiency.

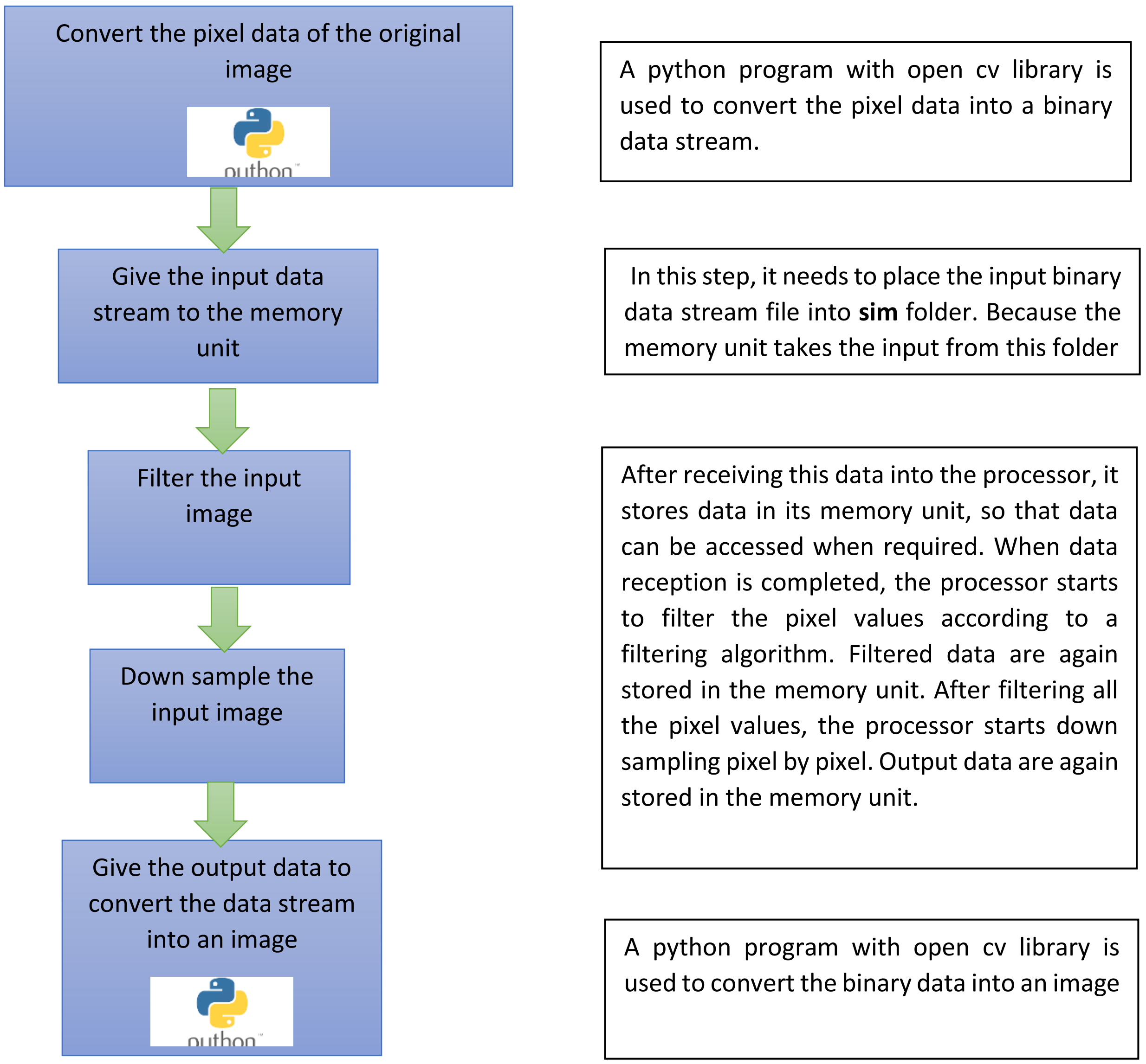
1.4 Problem Statement

The main task of the given project is to design a processor to down sample a 256*256px image. According to the given task, the main requirements for the designing processor are filtering and down sampling a given image.

The followings are the constraints that were selected for the processor design.

- Original Image - 256*256 px image
- Down Sampling factor - 2
- Image type - gray

1.5 Proposed Solution



Convert the pixel data of the original image
python
A python program with open cv library is used to convert the pixel data into a binary data stream.
Give the input data stream to the memory unit
In this step, it needs to place the input binary data stream file into sim folder. Because the memory unit takes the input from this folder
Filter the input image
After receiving this data into the processor, it stores data in its memory unit, so that data can be accessed when required. When data reception is completed, the processor starts to filter the pixel values according to a filtering algorithm. Filtered data are again stored in the memory unit. After filtering all the pixel values, the processor starts down sampling pixel by pixel. Output data are again stored in the memory unit.
Down sample the input image
Give the output data to convert the data stream into an image
python
A python program with open cv library is used to convert the binary data into an image

2. Designing the Processor

2.1 Datapath and Control Signals

- Click here

2.2 Instruction Set

There are 31 instructions that can be used by the assembly programmer. Some instructions have a 16-bit operand, and most instructions are zero operand instructions. 8 bits are used for opcode. Following table explains the operations of each instruction.

Instruction	Opcode	Operand (16 bit)	Operation
loadac	0000 0100	-	ac <= DM[ac]
movacr	0000 1000	-	r <= ac
movacr1	0000 1001	-	r1 <= ac
movacr2	0000 1010	-	r2 <= ac
movacr3	0000 1011	-	r3 <= ac
movacr4	0000 1100	-	r4 <= ac
movacr5	0000 1101	-	r5 <= ac
movacdar	0000 1110	-	dar <= ac
movrac	0000 1111	-	ac <= r
movr1ac	0001 0000	-	ac <= r1
movr2ac	0001 0001	-	ac <= r2
movr3ac	0001 0010	-	ac <= r3
movr4ac	0001 0011	-	ac <= r4
movr5ac	0001 0100	-	ac <= r5
movdarac	0001 0101	-	ac <= dar
stac	0001 0110	-	DM[ac] <= ac
add	0001 1001	-	ac <= ac + r
sub	0001 1011	-	ac <= ac – r
lshift	0001 1101	-	ac <= r << ac
rshift	0001 1111	-	ac <= r >> ac
incac	0010 0001	-	ac <= ac + 1
incdar	0010 0010	-	dar <= dar +1
incr1	0010 0011	-	r1 <= r1 +1
incr2	0010 0100	-	r2 <= r2 +1
incr3	0010 0101	-	r3 <= r3 + 1
loadim	0010 0110	imm	ac <= imm
jumpz	0010 1001	imm	if z == 1, go to instruction IM[imm]
jumpnz	0011 0000	imm	if z == 0, go to instruction IM[imm]
jump	0011 0001	imm	go to instruction IM[imm]
nop	0011 0010	-	no operation
endop	0011 0011	-	end process

2.3 Microinstructions

These microinstructions are used for finite state machine in the control unit. one microinstruction can be operated in one clock cycle. Following table explains operations of each microinstruction.

Instruction (8bits)	Opcode (decimal)	Microinstruction (6bits)	Operation
idle (internal)	-	idle	goto fetch1 if status == 1 else idle
fetch (internal)	-	fetch1	read IMEM
			write ir
			goto fetch2
		fetch2	read IMEM
			write ir
			goto fetch3
		fetch3	read IMEM
			goto ir[5:0]
loadac	4	loadac1	read ac
			write dar
			goto loadac2
		loadac2	read ac
			wrirte dar
			goto loadac3
		loadac3	read DMEM
			write ac
			goto loadac4
		loadac4	read DMEM
			write ac
			increment pc
			goto fetch1
movacr	8	movacr	read ac
			write r
			increment pc
			goto fetch1
movacr1	9	moveacr1	read ac
			write r1
			increment pc
			goto fetch1
movacr2	10	moveacr2	read ac
			write r2
			increment pc
			goto fetch1
movacr3	11	moveacr3	read ac
			write r3
			increment pc
			goto fetch1
movacr4	12	moveacr4	read ac
			write r4
			increment pc
			goto fetch1
movacr5	13	moveacr5	read ac
			write r5
			increment pc
			goto fetch1
movacdar	14	moveacdar	read ac

			write dar
			increment pc
			goto fetch1
movrac	15	moverac	read r
			write ac
			increment pc
			goto fetch1
movr1ac	16	mover1ac	read r1
			write ac
			increment pc
			goto fetch1
movr2ac	17	mover2ac	read r2
			write ac
			increment pc
			goto fetch1
movr3ac	18	mover3ac	read r3
			write ac
			increment pc
			goto fetch1
movr4ac	19	mover4ac	read r4
			write ac
			increment pc
			goto fetch1
movr5ac	20	mover5ac	read r5
			write ac
			increment pc
			goto fetch1
movdarac	21	movedarac	read dar
			write ac
			increment pc
			goto fetch1
stac	22	stac1	read ac
			goto stac2
		stac2	read ac
			write dm
			goto stac3
		stac3	read ac
			increment pc
			goto fetch1
add	25	add1	add alu
			goto add2
		add2	add alu
			write alu_to_ac
			increment pc
			goto fetch1
sub	27	sub1	sub alu
			goto sub2
		sub2	sub alu
			write alu_to_ac
			increment pc
			goto fetch1
lshift	29	lshift1	lshift alu

			goto lshift2
		lshift2	lshift alu
			write alu_to_ac
			increment pc
			goto fetch1
rshift	31	rshift1	rshift alu
			goto rshift2
		rshift2	rshift alu
			write alu_to_ac
			increment pc
			goto fetch1
incac	33	incac	increment ac
			incremtnt pc
			goto fetch1
incdar	34	incdar	increment dar
			incremtnt pc
			goto fetch1
incr1	35	incr1	increment r1
			incremtnt pc
			goto fetch1
incr2	36	incr2	increment r2
			incremtnt pc
			goto fetch1
incr3	37	incr3	increment r3
			incremtnt pc
			goto fetch1
loadim	38	loadim1	increment pc
			goto loadim2
		loadim2	read IMEM
			goto loadim3
		loadim3	read IMEM
			write ac
			increment pc
			goto fetch1
jump	49	jump	increment pc
			goto jumpz2
jumpz	41	jumpz1	increment pc
			goto jumpz2 if z == 1 else jumpz6
		jumpz2	read IMEM
			write pc
			goto jumpz3
		jumpz3	read IMEM
			write pc
			goto jumpz4
		jumpz4	read IMEM
			write ir
			goto jumpz5
		jumpz5	read IMEM
			write ir
			goto fetch3
		jumpz6	goto jumpz7
		jumpz7	increment pc

			goto jumpz4
jumpnz	48	jumpnz1	increment pc
			goto jumpz3 if z == 0 else goto jumpz7
endop	51	endop	read DMEM
			goto endop
nop	50	nop	increment pc
			goto fetch1

2.4 Finite State Machines for Microinstructions

- Click here

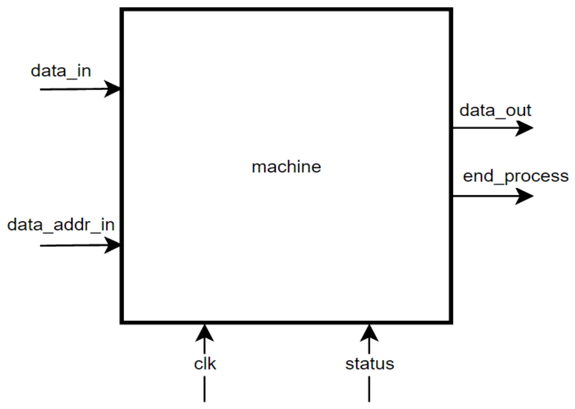
2.5 Modules and Components

2.5.1 Machine

There are three phases in this machine.

1. Load the image – Phase 1
2. Processing – Phase 2
3. Get the processed image – Phase 3

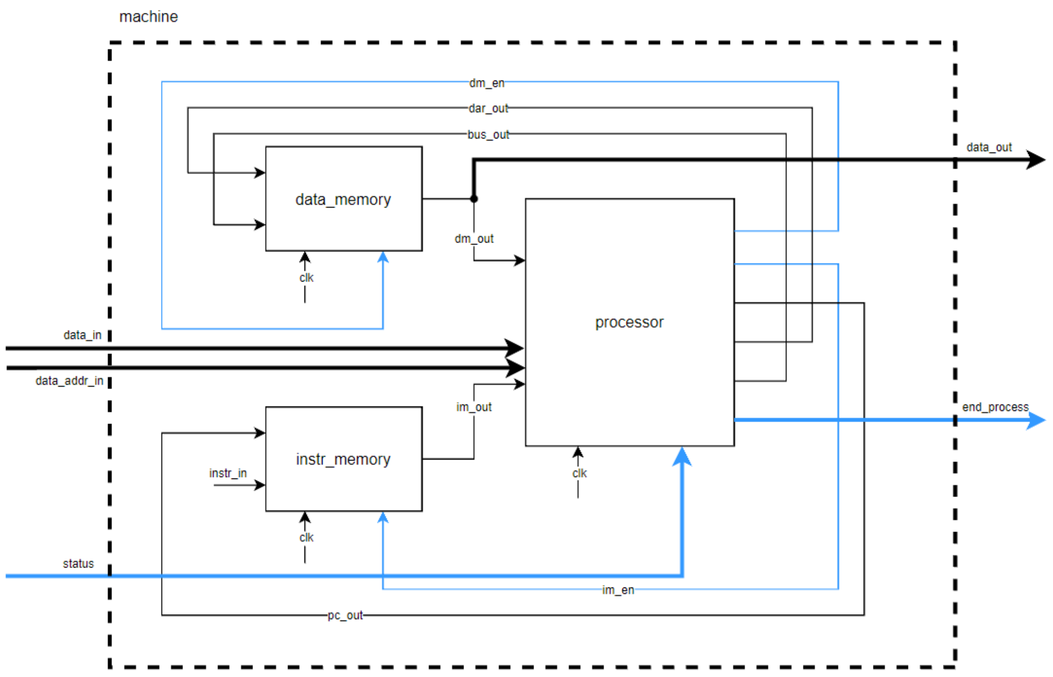
In phase 1, we have to give the image data to data_in and memory address to data_addr_in. Then processing will be done. After phase 2 is done, end_process will be asserted. In phase 3, we can get the output image data from the data_out.



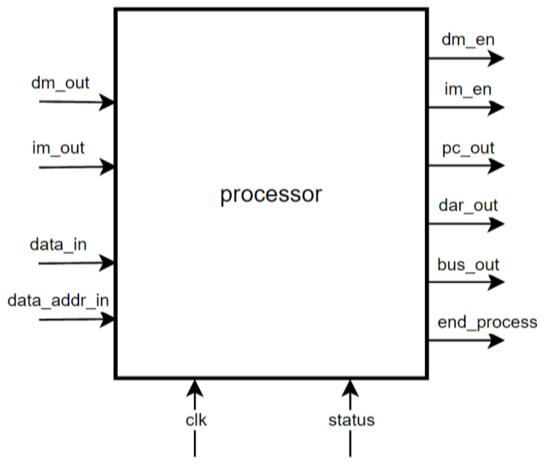
status	10	01	11
phase	1 – load image	2 – process	3 – output image

Inputs	**Outputs**
clk – clock	**end_process** – signal that indicate process has ended
status – control signal for change phases (input image data, process data, output data)	**data_out** – ouput image data
data_in – input image data	
data_addr_in – input data memory address	

2.5.2 Inside of the machine



2.5.3 Processor



Inputs	**Outputs**
clk – clock	**end_process** – signal that indicate process has ended
status – control signal for change phases (input image data, process data, output data)	**dm_en** – data memory write enable signal
dm_out – output of data memory	**im_en** – instruction memory write enable signal
im_out – output of instruction memory	**pc_out** – instruction memory address
data_in – input image data	**dar_out** – data memory address
data_addr_in – input data memory address	**bus_out** – data output for data memory

- Inside of the Processor – Click here

2.5.4 Modules Inside the Processor

Arithmetic & Logic Unit (ALU) – Do arithmetic and logic operations in the processor

Control Unit – Do all the controlling in the processor. Implemented as a finite state machine.

Special purpose registers

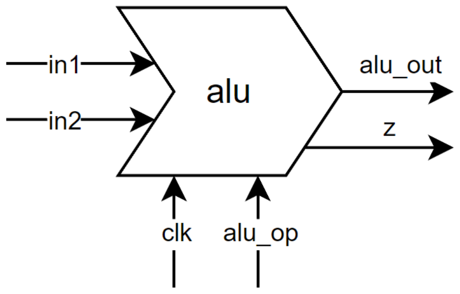
- Accumulator (ac) – input for the ALU, store ALU output
- Program counter (pc) – directed to the next instruction to be fetched
- Instruction register (ir) – store the instruction fetched from the instruction memory
- Data address register (dar) – keep the address of the store location of the data memory. Loading from data memory and storing to data memory are being done using this address
- r register – one of the inputs for the ALU

General purpose registers

- r1, r2, r3 – general purpose registers which can be incremented
- r4, r5 – general purpose registers which cannot be incremented

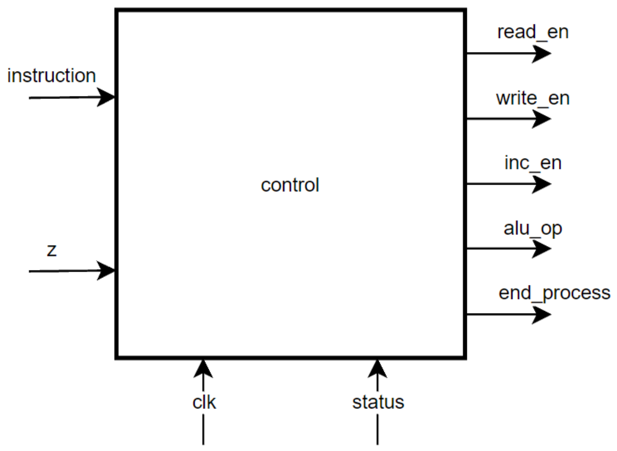
2.5.5 ALU

ALU is responsible for all the arithmetic and logic operations. Inputs are taken from ac and r registers and output is stored in ac register. We are using 3-bit alu_op even though there are only four operations since it is better to have some improvement capabilities. z will be 1 if the alu_out is zero. z will be 0 if the alu_out is nonzero.



alu_op	Operation	Description
001	Addition	Alu_out <= in1 + in2
010	Subtraction	Alu_out <= in2 – in1
011	Left shift	Alu_out <= in1 << in2
100	Right shift	Alu_out <= in1 >> in2

2.5.6 Control Unit



Control Unit is responsible for all the control signals in the processor. Here we are implemented this using a finite state machine.

Inputs	Outputs
clk – clock	**end_process** – signal that indicate process has ended
status – control signal for change phases (input image data, process data, output data)	**read_en** – 4 bit read enable signal
instruction – instruction from the instruction register	**write_en** – 16 bit write enable signal
z – z output from the alu	**inc_en** – 8 bit increment enable signal
	alu_op – ALU opcode

Read Enable – read_en

read_en is the control signal that is responsible for all the readings in the processor. Read values are written to the data bus. We cannot read from multiple registers and write them to the same bus. Hence, we are using 4 bit signal such that only one register can be read at one time.

Decimal Value	Register
0	-
1	dar
2	ac
3	r
4	r1
5	r2
6	r3
7	r4
8	r5
9	DM
10	IM

Write Enable – write_en

write_en is the control signal that is responsible for all the writings in the processor. Unlike the reading, we can write data to multiple locations at the same time. Therefore, we are using 16-bit signal and each bit has assigned to separate location. We can write those location by setting those bits to 1.

Bit 15 N/A	Bit 14 N/A	Bit 13 alu to ac	Bit 12 IM	Bit 11 DM	Bit 10 r1	Bit 9 r2	Bit 8 r3	Bit 7 r4	Bit 6 r5	Bit 5 r	Bit 4 ac	Bit 3 ir	Bit 2 dar	Bit 1 pc	Bit 0 N/A

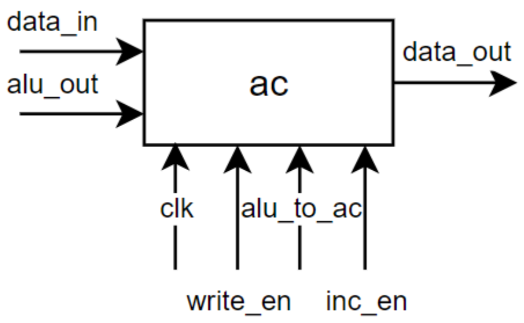
Increment Enable – inc_en

inc_en is the control signal that is responsible for all the increments in the processor. Same as the writing, we can increment multiple registers at the same time.

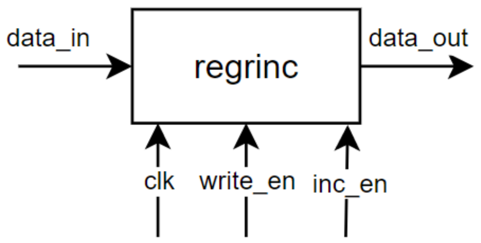
Bit 7 N/A	Bit 6 r3	Bit 5 r2	Bit 4 r1	Bit 3 dar	Bit 2 ac	Bit 1 pc	Bit 0 N/A

2.5.7 Accumulator

Accumulator is a special purpose register which has two data inputs. One is from the data bus and the other one is from the ALU output. There are two write enable signals to indicate from which input the writing should be done. This is one of the inputs to the ALU.

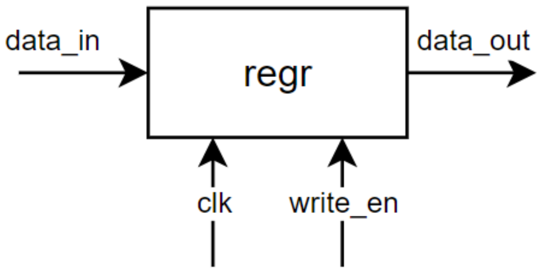


2.5.8 Register Which Can Be Incremented



pc, dar, r1, r2, r3 are registers of this type. The values in those registers can be incremented without using the ALU.

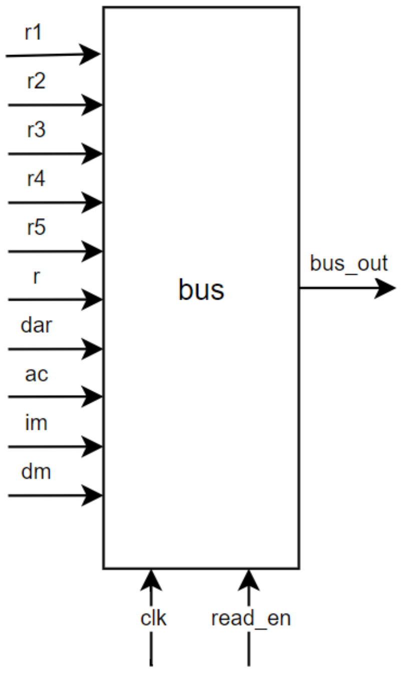
2.5.9 Register Which Cannot Be Incremented



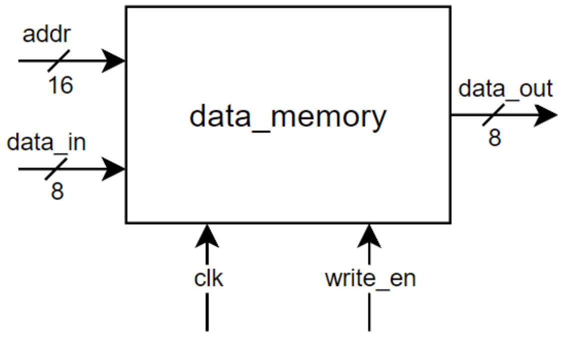
ir, r, r4, r5 are registers of this type. The values in those registers cannot be incremented without using the ALU.

2.5.10 Data Bus

This is a 16-bit path which connects all the necessary components that needed to the data flow.



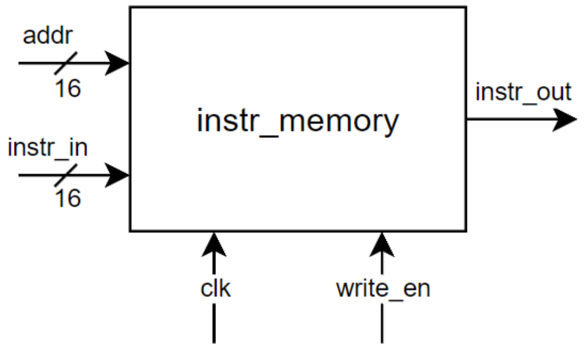
2.5.11 Data Memory



This is the main memory which data is stored. The image should be stored here before the processing begins. This memory contains 65536 locations of 8 bits long. Since we are using 256*256 image, 65536 pixel values are needed to be stored and that is why 65536 locations. Since the pixel value range is 0-255, we need 8-bit long memory to store that.

2.5.12 Instruction Memory

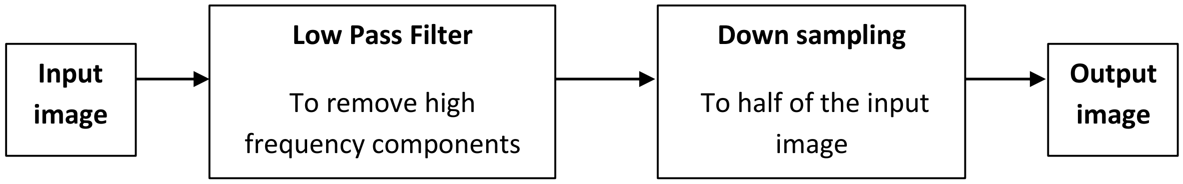
This is for storing the instructions. This is a read only memory. The processor cannot write into instruction memory during the processing. This can be written only when we need to load a new programme.



3. Algorithm and Design Considerations

3.1 Filtering & Down-sampling

Processor is designed to down sample an image. Down sampling is the reduction in spatial resolution while keeping the same two-dimensional (2D) representation. To down sample following steps are to be followed.


Input image
Low Pass Filter
To remove high frequency components
Down sampling
To half of the input image
Output image

3.2 Low Pass Filter

Prior to down sampling an image, it should be low pass filtered to reduce the effects of aliasing caused by the high frequency components in the image. For the filtering purpose, there are mainly two types of low pass filtering kernels.

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \qquad \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

Normalizing Kernel　　Gaussian Kernel

Although it is easy to perform addition using normalizing kernel, as it should be divisible by 9, we are not going to use this kernel for low pass filtering.

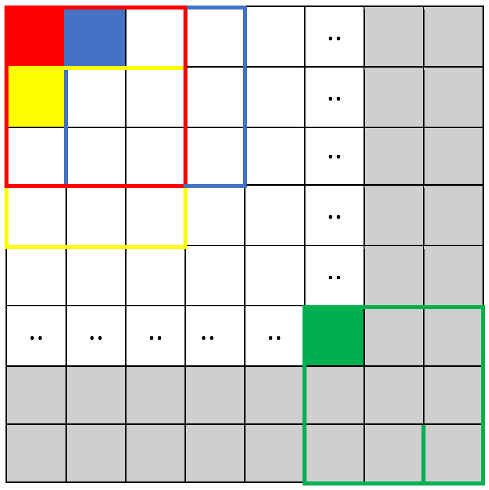
In Gaussian Kernel, shifting is needed to be done before addition. 16 is a number which is a power of 2, (16=24). So, it can be easily shifted when we use Gaussian Kernel. The filtering algorithm is based on the 3x3 Gaussian kernel where the middle pixel location is considered as (0,0) and the standard deviation is 1 in the below equation and then approximating it to discrete numbers.

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

$$\begin{bmatrix}0.075 & 0.124 & 0.075\\ 0.124 & 0.204 & 0.124\\ 0.075 & 0.124 & 0.075\end{bmatrix} \qquad 1/16\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

The Gaussian filtering can be used to blur the images and to remove the noise present. Here a 3×3 matrix is used mainly because 99% of the distribution fall within 3 standard deviations. The central pixel in Gaussian kernel has a higher weighting than the others on the sides.

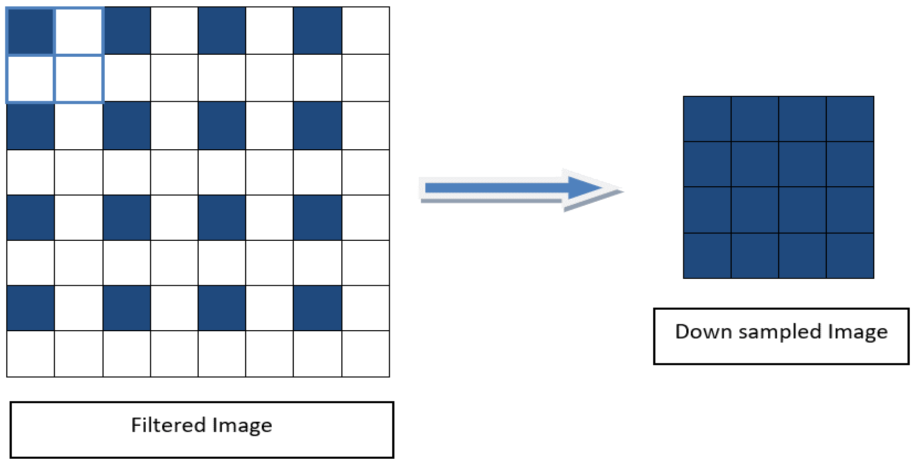
Now this center pixel is overlapped with the image pixel and a weighted summation value is calculated and it is divided by 16, which is the total weight of the kernel.



The kernel will calculate the average value first and stores it in the red colored pixel box. Then, it will move to the right to calculate the next average value. As per the figure it will move to the blue colored pixel box. After calculating all the average values in the right and storing it in that pixel, the kernel will move to the next row, which is marked in the yellow color. Same as earlier it will move right and will come to next row to filter all the pixel values. Finally, it will end up in the green colored pixel box. The gray colored pixel boxes remain unfiltered as it will not a big concern for down sampling.

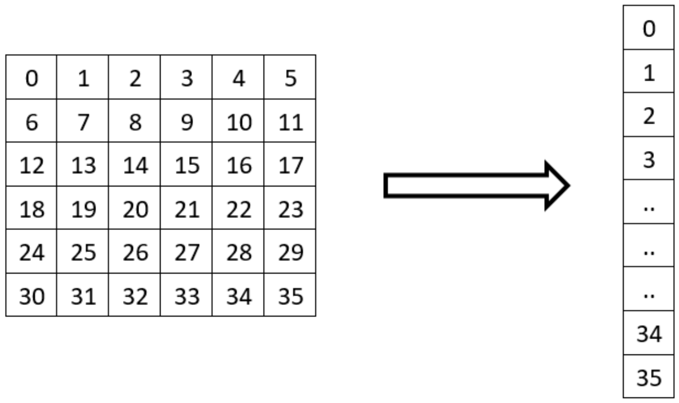
3.3 Down-sampling

Down sampling reduces the number of pixels in image by a given down sampling factor. Here, the objective is to down sample an image by a down sampling factor 2. We are going to consider an image with the size 256×256. So, the resultant image size is 128×128. For that we are going to consider one value per four pixels from the above filtered image.



3.4 Data Processing Techniques

The very first step of our task is to store the data included in the pixels of the 256x256 image into a text file. This can be done using the python code ‘input_create.py’. All the pixel values of the image are transformed to an array and the binary values of them are written in a text file. Data arrangement in the image and the text file is given below.



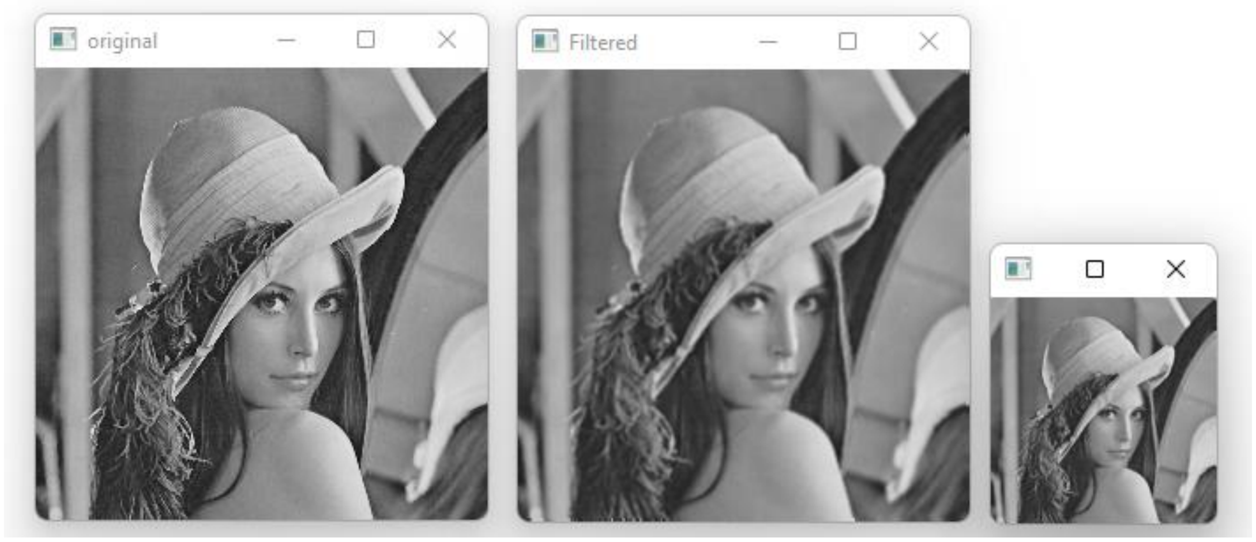
0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

0
1
2
3
..
..
..
34
35

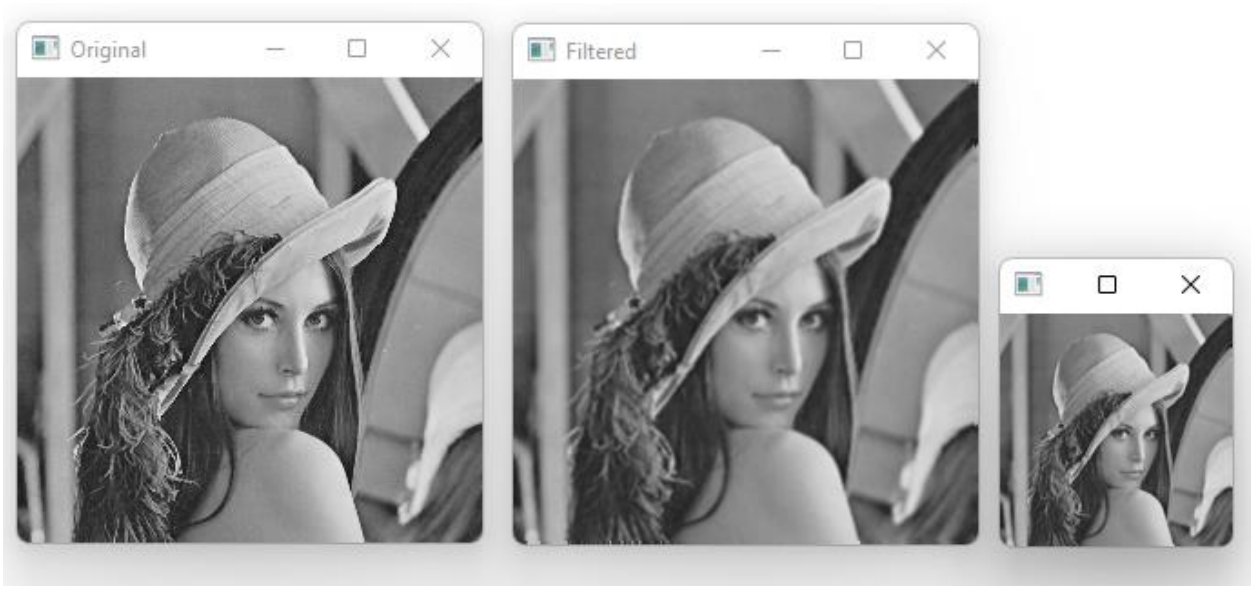
Then for the simulation of filtering and down sampling, the binary values in the text file are used as the input. From the simulation we can obtain the filtered output text file or the down sampled output text file, from considering the operation.

By using this filtered and down sampled text files as data files and by executing the ‘show.py’ python code, the binary values in the text files are stored in an array. After that this array is transformed into a matrix and then the filtered and down sampled images are obtained.

For the evaluation of the performance of our processor, a python code is used for filtering and down sampling. Here a comparison is made between the image data arrays of down sampled images from the designed processor and the python code execution by taking their difference. The error calculated from the above procedure is zero, which concludes that the designed processor is as per our expectations. Below images show the results from above mentioned methods.

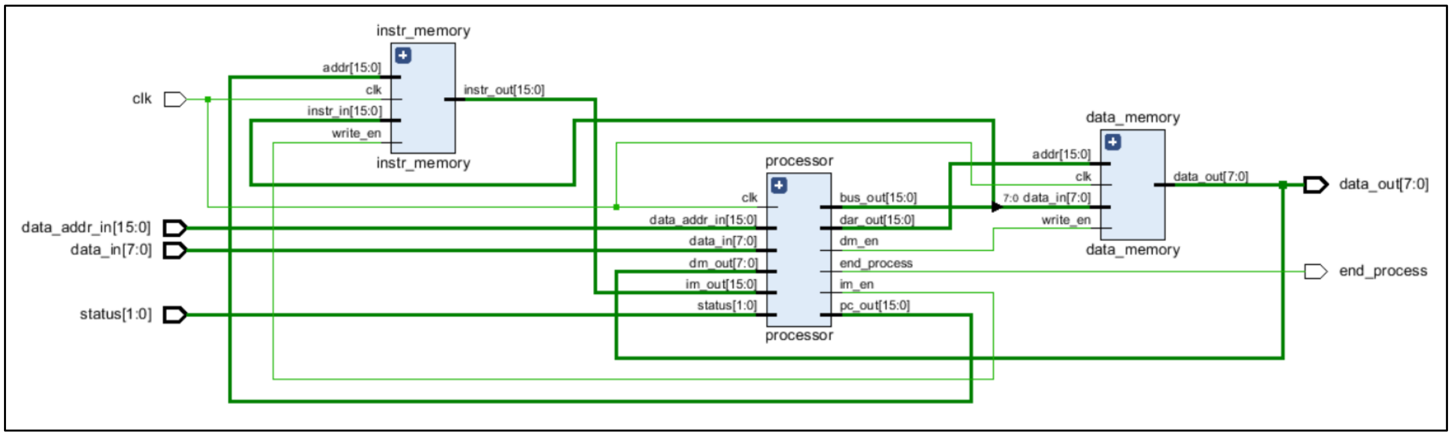


Original, filtered and down sampled images from the processor

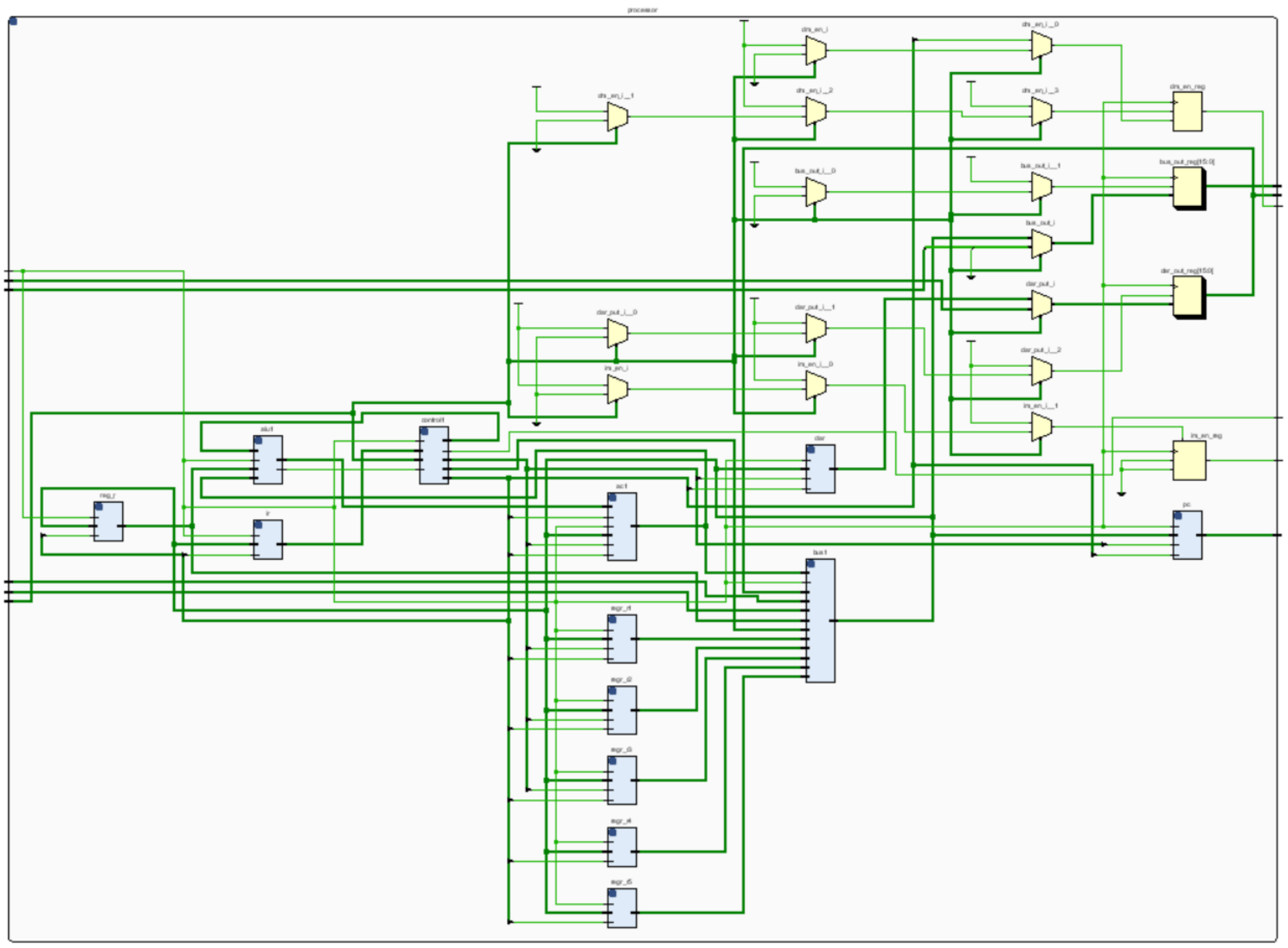


Original, filtered and down sampled images from the python code implementation

4. RTL Design Simulation



Processor with memory modules



Processor without memory modules

5. References

- https://youtube.com/playlist?list=PL5Q2soXY2Zi97Ya5DEUpMpO2bbAoaG7c6
- https://www.chipverify.com/verilog/verilog-tutorial
- https://www.geeksforgeeks.org/apply-a-gauss-filter-to-an-image-with-python/
- https://www.simplilearn.com/image-processing-article

All the source codes and other files can be found in this GitHub repository.
https://github.com/TharukaN17/Processor_Design

6. Appendix

6.1 Creating input file (Python)

```python
import cv2

image = cv2.imread('test.png', cv2.IMREAD_GRAYSCALE)
cv2.imshow('original', image)
cv2.waitKey()
image2 = image.ravel()
print(image2)
with open('input.txt', 'w') as f:
    f.write('\n'.join("{0:b}".format(int(e)) for e in image2))
```

6.2 Assembly code

```
loadim
16'd257     // Starting from 257th bit (2nd row, 2nd bit)
movacr1
movr1ac     // Start of the filtering loop
loadac
movacr      // R <- middle
loadim
8'd2        // To multiply by 4
lshift
movacr4     // R4 <- middle*4
loadim
8'd1        // To get right and left bits
movacr
movr1ac
add
loadac
movacr5     // R5 <- right
movr1ac
sub
loadac
movacr
movr5ac
add
movacr5     // R5 <- right + left
loadim
16'd256     // To get bottom bit
movacr
movr1ac
add
loadac
movacr
movr5ac
add
```

```
movacr5    // R5 <- right + left + bottom
loadim
16'd256    // To get top bit
movacr
movr1ac
sub
loadac
movacr
movr5ac
add
movacr     // R <- right + left + bottom + top
loadim
8'd1       // To multiply by 2
lshift
movacr     // R <- (right + left + bottom + top)*2
movr4ac
add
movacr4    // R4 <- middle*4 + (right + left + bottom + top)*2
loadim
16'd257    // To get bottom_right and top_left bits
movacr
movr1ac
add
loadac
movacr5    // R5 <- bottom_right
movr1ac
sub
loadac
movacr
movr5ac
add
movacr5    // R5 <- bottom_right + top_left
loadim
8'd255     // To get bottom_left bit
movacr
movr1ac
add
loadac
movacr
movr5ac
add
movacr5    // R5 <- bottom_right + top_left + bottom_left
loadim
8'd255     // To get top_right bit
movacr
movr1ac
sub
loadac
movacr
movr5ac
add
movacr     // R <- bottom_right + top_left + bottom_left + top_right
movr4ac
```

```
add
movacr      // R <- middle*4 + (right+left+bottom+top)*2 + (bottom_right+top_left+bottom_left+top_right)
loadim
8'd4        // To divide by 16
rshift
movacr4     // R4 <- (middle*4 + (right+left+bottom+top)*2 + (bottom_right+top_left+bottom_left+top_right))/16
loadim
16'd257     // To find store location
movacr
movr1ac
sub
movacdar
movr4ac
stac
loadim
16'd65278   // To check end of the image ((256*254) + (255-1))
movacr
movr1ac
sub
jumpz
8'd127        // Go to the start of downscaling
loadim
8'd253        // To check end of the row
movacr
movr2ac
sub
jumpz
8'd118
incr1
incr2
jump
8'd3
incr1
incr1
incr1
loadim
8'd0
movacr2
jump
8'd3        // Go to the start of the filtering loop
endop
loadim      // Start of downscaling
8'd0
movacr1     // R1 <- 0,   R1 = Keep track of current pixel
movacr2     // R2 <- 0,   R2 = Keep track of current row pixel
movacr3     // R3 <- 0,   R3 = Keep track of store location
movr1ac     // Start of the downscaling loop
loadac
movacr4
movr3ac
movacdar
movr4ac
stac
```

```
140.    movr1ac
141.    movacr
142.    loadim
143.    16'd65278   // To check end of the image ((256*254) + (255-1))
144.    sub
145.    jumpz
146.    8'd172      // Go to end process
147.    movr2ac
148.    movacr
149.    loadim
150.    8'd254      // To check end of the row
151.    sub
152.    jumpz
153.    8'd160
154.    incr2
155.    incr2
156.    incr1
157.    incr1
158.    incr3
159.    jump
160.    8'd132      // Go to the start of the downscaling loop
161.    loadim
162.    8'd0
163.    movacr2
164.    loadim
165.    16'd258     // To skip a row (256 + 2)
166.    movacr
167.    movr1ac
168.    add
169.    movacr1
170.    incr3
171.    jump
172.    8'd132      // Go to the start of the downscaling loop
173.    endop       // End process
```

6.2.1 Compiler to binary values (Python)

```
compDict = {
    "loadac"    : 4,
    "movacr"    : 8,
    "movacr1"   : 9,
    "movacr2"   : 10,
    "movacr3"   : 11,
    "movacr4"   : 12,
    "movacr5"   : 13,
    "movacdar"  : 14,
    "movrac"    : 15,
    "movr1ac"   : 16,
    "movr2ac"   : 17,
    "movr3ac"   : 18,
    "movr4ac"   : 19,
    "movr5ac"   : 20,
    "movdarac"  : 21,
    "stac"      : 22,
    "add"       : 25,
    "sub"       : 27,
    "lshift"    : 29,
    "rshift"    : 31,
    "incac"     : 33,
    "incdar"    : 34,
    "incr1"     : 35,
    "incr2"     : 36,
    "incr3"     : 37,
    "loadim"    : 38,
    "jumpz"     : 41,
    "jumpnz"    : 48,
    "jump"      : 49,
    "nop"       : 50,
    "endop"     : 51
}

instructionArray = open("Assembly.txt").read().splitlines()
outputFile = open("output_bin.txt", "w")
lines = []
comment = "//"

for i, instruction in enumerate(instructionArray):

    if comment in instruction:
        if comment in instruction[:3]:
            continue
        else:
            instruction = instruction.split(comment)[0]

    instruction = instruction.strip();

    if instruction in compDict:
        binaryValue = bin(compDict[instruction])[2:]
        length = 8
    else:
        val = instruction.split("'d")
        binaryValue = bin(int(val[1]))[2:]
        length = int(val[0])

    writeValue = (length-len(binaryValue))*'0'+binaryValue
    lines.append(writeValue)

outputFile.writelines("\n".join(lines))
outputFile.close()
```

6.2.2 Compiler to ram (Python)

```
compDict = {
    "loadac"    : 4,
    "movacr"    : 8,
    "movacr1"   : 9,
    "movacr2"   : 10,
    "movacr3"   : 11,
    "movacr4"   : 12,
    "movacr5"   : 13,
    "movacdar"  : 14,
    "movrac"    : 15,
    "movr1ac"   : 16,
    "movr2ac"   : 17,
    "movr3ac"   : 18,
    "movr4ac"   : 19,
    "movr5ac"   : 20,
    "movdarac"  : 21,
    "stac"      : 22,
    "add"       : 25,
    "sub"       : 27,
    "lshift"    : 29,
    "rshift"    : 31,
    "incac"     : 33,
    "incdar"    : 34,
    "incr1"     : 35,
    "incr2"     : 36,
    "incr3"     : 37,
    "loadim"    : 38,
    "jumpz"     : 41,
    "jumpnz"    : 48,
    "jump"      : 49,
    "nop"       : 50,
    "endop"     : 51
}

instructionArray = open("Assembly.txt").read().splitlines()
outputFile = open("output_ram.txt", "w")
lines = []
comment = "//"

for i, instruction in enumerate(instructionArray):

    if comment in instruction:
        if comment in instruction[:3]:
            continue
        else:
            instruction = instruction.split(comment)[0]
    instruction = instruction.strip()

    if instruction in compDict:
        value = "8'd"+str(compDict[instruction])
    else:
        value = instruction
    writeValue = 'ram['+str(len(lines))+'] = '+value+';'
    lines.append(writeValue)

outputFile.writelines("\n".join(lines))
outputFile.close()
```

6.3 Verilog codes for processor design

6.3.1 Accumulator

```
module ac(
    input                  clk, write_en, alu_to_ac,inc_en,
    input           [15:0]  data_in, alu_out,
    output reg      [15:0]  data_out = 16'd0
    );

    always @(posedge clk) begin
        if (inc_en == 1)
            data_out <= data_out + 16'd1;
        else if (write_en == 1)
            data_out <= data_in;
        else if (alu_to_ac == 1)
            data_out <= alu_out;
    end
endmodule
```

6.3.2 ALU

```
module alu (
    input                  clk,
    input           [15:0]  in1, in2,
    input           [2:0]   alu_op,
    output reg      [15:0]  alu_out,
    output reg              z
    );

    always @(posedge clk)
    begin
        case(alu_op)
            3'd1: alu_out <= in1 + in2;
            3'd2: alu_out <= in2 - in1;
            3'd3: alu_out <= in1 << in2;
            3'd4: alu_out <= in1 >> in2;
        endcase

        if (alu_out == 0)
            z <= 1;
        else
            z <= 0;
    end
endmodule
```

6.3.3 Bus

```
module bus(
    input               clk,
    input       [3:0]   read_en,
    input       [15:0]  r1,r2,r3,r4,r5,r,dar,ac,im,
    input       [7:0]   dm,
    output reg  [15:0]  bus_out
    );

    always @(*) begin
        case (read_en)
            4'd1:    bus_out <= dar;
            4'd2:    bus_out <= ac;
            4'd3:    bus_out <= r;
            4'd4:    bus_out <= r1;
            4'd5:    bus_out <= r2;
            4'd6:    bus_out <= r3;
            4'd7:    bus_out <= r4;
            4'd8:    bus_out <= r5;
            4'd9:    bus_out <= dm + 16'd0;
            4'd10:   bus_out <= im;
            default: bus_out <= 16'd0;
        endcase
    end
endmodule
```

6.3.4 State machine

```
module control(
    input               clk, z,
    input      [15:0] instruction,
    input      [1:0]  status,
    output reg [2:0]  alu_op,
    output reg [7:0]  inc_en,
    output reg [15:0] write_en,
    output reg [3:0]  read_en,
    output reg        end_process
    );

    reg [5:0] present = 6'd0;
    reg [5:0] next    = 6'd0;

    parameter
    idle       = 6'd0,
    fetch1     = 6'd1,
    fetch2     = 6'd2,
    fetch3     = 6'd3,
    loadac1    = 6'd4,
    loadac2    = 6'd5,
    loadac3    = 6'd6,
    loadac4    = 6'd7,
    movacr     = 6'd8,
    movacr1    = 6'd9,
    movacr2    = 6'd10,
    movacr3    = 6'd11,
    movacr4    = 6'd12,
    movacr5    = 6'd13,
    movacdar   = 6'd14,
    movrac     = 6'd15,
    movr1ac    = 6'd16,
    movr2ac    = 6'd17,
    movr3ac    = 6'd18,
    movr4ac    = 6'd19,
    movr5ac    = 6'd20,
    movdarac   = 6'd21,
    stac1      = 6'd22,
    stac2      = 6'd23,
    stac3      = 6'd24,
    add1       = 6'd25,
    add2       = 6'd26,
    sub1       = 6'd27,
    sub2       = 6'd28,
    lshift1    = 6'd29,
    lshift2    = 6'd30,
    rshift1    = 6'd31,
    rshift2    = 6'd32,
    incac      = 6'd33,
    incdar     = 6'd34,
    incr1      = 6'd35,
    incr2      = 6'd36,
    incr3      = 6'd37,
    loadim1    = 6'd38,
    loadim2    = 6'd39,
    loadim3    = 6'd40,
    jumpz1     = 6'd41,
    jumpz2     = 6'd42,
    jumpz3     = 6'd43,
    jumpz4     = 6'd44,
    jumpz5     = 6'd45,
    jumpz6     = 6'd46,
    jumpz7     = 6'd47,
    jumpnz1    = 6'd48,
    jump       = 6'd49,
    nop        = 6'd50,
    endop      = 6'd51;

    always @(posedge clk)
        present <= next;

    always @(posedge clk) begin
        if (present == endop)
        end_process <= 1'd1;
        else
        end_process <= 1'd0;
    end

    always @(present or z or instruction or status)
    case(present)
        idle: begin
            read_en  <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            if (status == 2'b01)
                next <= fetch1;
            else
                next <= idle;
        end

        fetch1: begin
            read_en  <=  4'd10;
            write_en <= 16'b0000000000001000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            next     <= fetch2;
        end

        fetch2: begin
            read_en  <=  4'd10;
            write_en <= 16'b0000000000001000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            next     <= fetch3;
        end
```

```
fetch3: begin
    read_en  <=  4'd10;
    write_en <= 16'b0000000000000000;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= instruction [5:0];
end

loadac1: begin
    read_en  <=  4'd2;
    write_en <= 16'b0000000000000100;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= loadac2;
end

loadac2 : begin
    read_en  <=  4'd2;
    write_en <= 16'b0000000000000100;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= loadac3;
end

loadac3 : begin
    read_en  <=  4'd9;
    write_en <= 16'b0000000000010000;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= loadac4;
end

loadac4: begin
    read_en  <=  4'd9;
    write_en <= 16'b0000000000010000;
    inc_en   <=  8'b00000010;
    alu_op   <=  3'd0;
    next     <= fetch1;
end

movacr: begin
    read_en  <=  4'd2;
    write_en <= 16'b0000000000100000;
    inc_en   <=  8'b00000010;
    alu_op   <=  3'd0;
    next     <= fetch1;
end

movacr1: begin
```

```
        read_en  <=  4'd2;
        write_en <= 16'b0000000001000000;
        inc_en   <=  8'b00000010;
        alu_op   <=  3'd0;
        next     <= fetch1;
    end

    movacr2: begin
        read_en  <=  4'd2;
        write_en <= 16'b0000000010000000;
        inc_en   <=  8'b00000010;
        alu_op   <=  3'd0;
        next     <= fetch1;
    end

    movacr3: begin
        read_en  <=  4'd2;
        write_en <= 16'b0000000100000000;
        inc_en   <=  8'b00000010;
        alu_op   <=  3'd0;
        next     <= fetch1;
    end

    movacr4: begin
        read_en  <=  4'd2;
        write_en <= 16'b0000001000000000;
        inc_en   <=  8'b00000010;
        alu_op   <=  3'd0;
        next     <= fetch1;
    end

    movacr5: begin
        read_en  <=  4'd2;
        write_en <= 16'b0000010000000000;
        inc_en   <=  8'b00000010;
        alu_op   <=  3'd0;
        next     <= fetch1;
     end

     movacdar : begin
        read_en  <=  4'd2;
        write_en <= 16'b0000000000000100;
        inc_en   <=  8'b00000010;
        alu_op   <=  3'd0;
        next     <= fetch1;
     end

     movrac: begin
        read_en  <=  4'd3;
        write_en <= 16'b0000000000010000;
```

```
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         movr1ac: begin
            read_en  <=  4'd4;
            write_en <= 16'b0000000000010000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         movr2ac: begin
            read_en  <=  4'd5;
            write_en <= 16'b0000000000010000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         movr3ac: begin
            read_en  <=  4'd6;
            write_en <= 16'b0000000000010000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         movr4ac: begin
            read_en  <=  4'd7;
            write_en <= 16'b0000000000010000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         movr5ac: begin
            read_en  <=  4'd8;
            write_en <= 16'b0000000000010000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         movdarac : begin
            read_en  <=  4'd1;
            write_en <= 16'b0000000000010000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         stac1: begin
            read_en  <=  4'd2;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            next     <= stac2;
         end

         stac2: begin
            read_en  <=  4'd2;
            write_en <= 16'b0000100000000000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            next     <= stac3;
         end

         stac3: begin
            read_en  <=  4'd2;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
         end

         add1: begin
            read_en  <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd1;
            next     <= add2;
         end

         add2: begin
            read_en  <=  4'd0;
            write_en <= 16'b0010000000000000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd1;
            next     <= fetch1;
         end

         sub1: begin
            read_en  <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd2;
            next     <= sub2;
         end
```

```
sub2: begin
   read_en  <=  4'd0;
   write_en <= 16'b0010000000000000;
   inc_en   <=  8'b00000010;
   alu_op   <=  3'd2;
   next     <= fetch1;
end

lshift1: begin
   read_en  <=  4'd0;
   write_en <= 16'b0000000000000000;
   inc_en   <=  8'b00000000;
   alu_op   <=  3'd3;
   next     <= lshift2;
end

lshift2: begin
   read_en  <=  4'd0;
   write_en <= 16'b0010000000000000;
   inc_en   <=  8'b00000010;
   alu_op   <=  3'd3;
   next     <= fetch1;
end

rshift1: begin
   read_en  <=  4'd0;
   write_en <= 16'b0000000000000000;
   inc_en   <=  8'b00000000;
   alu_op   <=  3'd4;
   next     <= rshift2;
end

rshift2: begin
   read_en  <=  4'd0;
   write_en <= 16'b0010000000000000;
   inc_en   <=  8'b00000010;
   alu_op   <=  3'd4;
   next     <= fetch1;
end

incac: begin
   read_en  <=  4'd0;
   write_en <= 16'b0000000000000000;
   inc_en   <=  8'b00000110;
   alu_op   <=  3'd0;
   next     <= fetch1;
end

incdar: begin
```

```
            read_en   <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en    <=  8'b00001010;
            alu_op    <=  3'd0;
            next      <= fetch1;
         end

         incr1: begin
            read_en   <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en    <=  8'b00010010;
            alu_op    <=  3'd0;
            next      <= fetch1;
         end

         incr2: begin
            read_en   <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en    <=  8'b00100010;
            alu_op    <=  3'd0;
            next      <= fetch1;
         end

         incr3: begin
            read_en   <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en    <=  8'b01000010;
            alu_op    <=  3'd0;
            next      <= fetch1;
         end

         loadim1: begin
            read_en   <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en    <=  8'b00000010;
            alu_op    <=  3'd0;
            next      <= loadim2;
         end

         loadim2: begin
            read_en   <=  4'd10;
            write_en <= 16'b0000000000000000;
            inc_en    <=  8'b00000000;
            alu_op    <=  3'd0;
            next      <= loadim3;
         end

         loadim3: begin
            read_en   <=  4'd10;
            write_en <= 16'b0000000000010000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= fetch1;
        end

        jump: begin
            read_en  <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            next     <= jumpz2;
        end

        jumpz1: begin
            read_en  <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000010;
            alu_op   <=  3'd0;
            if (z == 1)
                next <= jumpz2;
            else
                next <= jumpz6;
        end

        jumpnz1: begin
            read_en  <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000010;
            alu_op <=  3'd0;
            if (z == 0)
                next <= jumpz3;
            else
                next <= jumpz7;
        end

        jumpz2: begin
            read_en  <=  4'd10;
            write_en <= 16'b0000000000000010;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            next     <= jumpz3;
        end

        jumpz3: begin
            read_en  <=  4'd10;
            write_en <= 16'b0000000000000010;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            next     <= jumpz4;
        end
```

```
 jumpz4: begin
    read_en  <=  4'd10;
    write_en <= 16'b0000000000001000;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= jumpz5;
 end

 jumpz5: begin
    read_en  <=  4'd10;
    write_en <= 16'b0000000000001000;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= fetch3;
 end

 jumpz6: begin
    read_en  <=  4'd0;
    write_en <= 16'b0000000000000000;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= jumpz7;
 end

 jumpz7: begin
    read_en  <=  4'd0;
    write_en <= 16'b0000000000000000;
    inc_en  <= 16'b00000010;
    alu_op  <=  3'd0;
    next    <= jumpz4;
end

nop: begin
    read_en  <=  4'd0;
    write_en <= 16'b0000000000000000;
    inc_en   <=  8'b00000010;
    alu_op   <=  3'd0;
    next     <= fetch1;
end

endop: begin
    read_en  <=  4'd9;
    write_en <= 16'b0000000000000000;
    inc_en   <=  8'b00000000;
    alu_op   <=  3'd0;
    next     <= endop;
end

default: begin
```

```
            read_en  <=  4'd0;
            write_en <= 16'b0000000000000000;
            inc_en   <=  8'b00000000;
            alu_op   <=  3'd0;
            next     <= fetch1;
        end
    endcase
endmodule
```

6.3.5 Data memory

```
module data_memory (
    input              clk, write_en,
    input       [15:0] addr,
    input       [7:0]  data_in,
    output reg  [7:0]  data_out
    );

    reg [7:0] ram [65535:0];

    always @(posedge clk) begin
        if (write_en == 1)
            ram[addr] <= data_in;
        else
            data_out <= ram[addr];
    end
endmodule
```

6.3.6 Instruction memory

```
module instr_memory(
    input              clk, write_en,
    input       [15:0] addr, instr_in,
    output reg  [15:0] instr_out
    );

    reg [15:0] ram [180:0];

    initial begin
    ram[0] = 8'd38;
    ram[1] = 16'd257;
    ram[2] = 8'd9;
    ram[3] = 8'd16;
    ram[4] = 8'd4;
    ram[5] = 8'd8;
    ram[6] = 8'd38;
    ram[7] = 8'd2;
```

```
ram[8] = 8'd29;
ram[9] = 8'd12;
ram[10] = 8'd38;
ram[11] = 8'd1;
ram[12] = 8'd8;
ram[13] = 8'd16;
ram[14] = 8'd25;
ram[15] = 8'd4;
ram[16] = 8'd13;
ram[17] = 8'd16;
ram[18] = 8'd27;
ram[19] = 8'd4;
ram[20] = 8'd8;
ram[21] = 8'd20;
ram[22] = 8'd25;
ram[23] = 8'd13;
ram[24] = 8'd38;
ram[25] = 16'd256;
ram[26] = 8'd8;
ram[27] = 8'd16;
ram[28] = 8'd25;
ram[29] = 8'd4;
ram[30] = 8'd8;
ram[31] = 8'd20;
ram[32] = 8'd25;
ram[33] = 8'd13;
ram[34] = 8'd38;
ram[35] = 16'd256;
ram[36] = 8'd8;
ram[37] = 8'd16;
ram[38] = 8'd27;
ram[39] = 8'd4;
ram[40] = 8'd8;
ram[41] = 8'd20;
ram[42] = 8'd25;
ram[43] = 8'd8;
ram[44] = 8'd38;
ram[45] = 8'd1;
ram[46] = 8'd29;
ram[47] = 8'd8;
ram[48] = 8'd19;
ram[49] = 8'd25;
ram[50] = 8'd12;
ram[51] = 8'd38;
ram[52] = 16'd257;
ram[53] = 8'd8;
ram[54] = 8'd16;
ram[55] = 8'd25;
ram[56] = 8'd4;
ram[57] = 8'd13;
```

```
    ram[58] = 8'd16;
    ram[59] = 8'd27;
    ram[60] = 8'd4;
    ram[61] = 8'd8;
    ram[62] = 8'd20;
    ram[63] = 8'd25;
    ram[64] = 8'd13;
    ram[65] = 8'd38;
    ram[66] = 8'd255;
    ram[67] = 8'd8;
    ram[68] = 8'd16;
    ram[69] = 8'd25;
    ram[70] = 8'd4;
    ram[71] = 8'd8;
    ram[72] = 8'd20;
    ram[73] = 8'd25;
    ram[74] = 8'd13;
    ram[75] = 8'd38;
    ram[76] = 8'd255;
    ram[77] = 8'd8;
    ram[78] = 8'd16;
    ram[79] = 8'd27;
    ram[80] = 8'd4;
    ram[81] = 8'd8;
    ram[82] = 8'd20;
    ram[83] = 8'd25;
    ram[84] = 8'd8;
    ram[85] = 8'd19;
    ram[86] = 8'd25;
    ram[87] = 8'd8;
    ram[88] = 8'd38;
    ram[89] = 8'd4;
    ram[90] = 8'd31;
    ram[91] = 8'd12;
    ram[92] = 8'd38;
    ram[93] = 16'd257;
    ram[94] = 8'd8;
    ram[95] = 8'd16;
    ram[96] = 8'd27;
    ram[97] = 8'd14;
    ram[98] = 8'd19;
    ram[99] = 8'd22;
    ram[100] = 8'd38;
    ram[101] = 16'd65278;
    ram[102] = 8'd8;
    ram[103] = 8'd16;
    ram[104] = 8'd27;
    ram[105] = 8'd41;
    ram[106] = 8'd127;
    ram[107] = 8'd38;
```

```
ram[108] = 8'd253;
ram[109] = 8'd8;
ram[110] = 8'd17;
ram[111] = 8'd27;
ram[112] = 8'd41;
ram[113] = 8'd118;
ram[114] = 8'd35;
ram[115] = 8'd36;
ram[116] = 8'd49;
ram[117] = 8'd3;
ram[118] = 8'd35;
ram[119] = 8'd35;
ram[120] = 8'd35;
ram[121] = 8'd38;
ram[122] = 8'd0;
ram[123] = 8'd10;
ram[124] = 8'd49;
ram[125] = 8'd3;
ram[126] = 8'd51;
ram[127] = 8'd38;
ram[128] = 8'd0;
ram[129] = 8'd9;
ram[130] = 8'd10;
ram[131] = 8'd11;
ram[132] = 8'd16;
ram[133] = 8'd4;
ram[134] = 8'd12;
ram[135] = 8'd18;
ram[136] = 8'd14;
ram[137] = 8'd19;
ram[138] = 8'd22;
ram[139] = 8'd16;
ram[140] = 8'd8;
ram[141] = 8'd38;
ram[142] = 16'd65278;
ram[143] = 8'd27;
ram[144] = 8'd41;
ram[145] = 8'd172;
ram[146] = 8'd17;
ram[147] = 8'd8;
ram[148] = 8'd38;
ram[149] = 8'd254;
ram[150] = 8'd27;
ram[151] = 8'd41;
ram[152] = 8'd160;
ram[153] = 8'd36;
ram[154] = 8'd36;
ram[155] = 8'd35;
ram[156] = 8'd35;
ram[157] = 8'd37;
```

```
    ram[158] = 8'd49;
    ram[159] = 8'd132;
    ram[160] = 8'd38;
    ram[161] = 8'd0;
    ram[162] = 8'd10;
    ram[163] = 8'd38;
    ram[164] = 16'd258;
    ram[165] = 8'd8;
    ram[166] = 8'd16;
    ram[167] = 8'd25;
    ram[168] = 8'd9;
    ram[169] = 8'd37;
    ram[170] = 8'd49;
    ram[171] = 8'd132;
    ram[172] = 8'd51;
    end

    always @(posedge clk) begin
        if (write_en == 1)
            ram[addr] <= instr_in;
        else
            instr_out <= ram[addr];
        end
endmodule
```

6.3.7 Top module

```
module machine(
    input          clk,
    input  [1:0]   status,
    input  [7:0]   data_in,
    input  [15:0]  data_addr_in,
    output         end_process,
    output [7:0]   data_out
    );

    wire        dm_en;
    wire        im_en;
    wire [15:0] pc_out;
    wire [15:0] dar_out;
    wire [15:0] bus_out;
    wire [7:0]  dm_out;
    wire [15:0] im_out;

    assign data_out = dm_out;

    processor    processor    (.clk(clk),.dm_out(dm_out),.im_out(im_out),.data_in(data_in),.data_a
ddr_in(data_addr_in),.status(status),
                              .dm_en(dm_en),.im_en(im_en),.pc_out(pc_out),.dar_out(dar_out),.bus_
out(bus_out),.end_process(end_process));

    data_memory  data_memory  (.clk(clk),.write_en(dm_en),.addr(dar_out),.data_in(bus_out),.data_o
ut(dm_out));

    instr_memory instr_memory
(.clk(clk),.write_en(im_en),.addr(pc_out),.instr_in(bus_out),.instr_out(im_out));
endmodule
```

6.3.8 Processor

```
 module processor (
    input        clk,
    input [7:0]  dm_out, data_in,
    input [15:0] im_out, data_addr_in,
    input [1:0]  status,

    output reg        dm_en, im_en,
    output     [15:0] pc_out,
    output reg [15:0] dar_out, bus_out,
    output            end_process
    );

    wire [2:0]  alu_op;
    wire [15:0] alu_out;
    wire [15:0] dar_wire;
    wire [15:0] bus_wire;

    wire [15:0] regr_out;
    wire [15:0] regr1_out;
    wire [15:0] regr2_out;
    wire [15:0] regr3_out;
    wire [15:0] regr4_out;
    wire [15:0] regr5_out;

    wire [15:0] ac_out;

    wire [15:0] ir_out;

    wire [15:0] write_en ;
    wire [3:0]  read_en;
    wire [7:0]  inc_en;

    wire        z;

    regr    reg_r    (.clk(clk), .write_en (write_en[5]),.data_in(bus_wire),.data_out(regr_out ));

    regrinc regr_r1  (.clk(clk), .write_en (write_en[6]),.data_in(bus_wire),.data_out(regr1_out),
                     .inc_en(inc_en[4]));

    regrinc regr_r2  (.clk(clk), .write_en (write_en[7]),.data_in(bus_wire),.data_out(regr2_out),
                     .inc_en(inc_en[5]));

    regrinc regr_r3  (.clk(clk), .write_en (write_en[8]),.data_in(bus_wire),.data_out(regr3_out),
                     .inc_en(inc_en[6]));

    regr    regr_r4  (.clk(clk), .write_en (write_en[9]),.data_in(bus_wire),.data_out(regr4_out
));

    regr    regr_r5  (.clk(clk), .write_en (write_en[10]),.data_in(bus_wire),.data_out(regr5_out
));

    regrinc dar      (.clk(clk), .write_en
(write_en[2]),.data_in(bus_wire),.data_out(dar_wire),.inc_en(inc_en[3]));

    regr    ir       (.clk(clk), .write_en (write_en[3]),.data_in(bus_wire),.data_out(ir_out));

    bus     bus1     (.r1(regr1_out ),.r2(regr2_out ),.r3(regr3_out ),.r4(regr4_out
),.r5(regr5_out ),.r(regr_out ),
                     .dar(dar_out),.ac(ac_out),.dm(dm_out),.im(im_out),.bus_out(bus_wire),.read_e
n(read_en),.clk(clk));

    ac      ac1      (.clk(clk), .write_en
(write_en[4]),.data_in(bus_wire),.data_out(ac_out),.alu_out(alu_out),
                     .alu_to_ac (write_en[13]),.inc_en(inc_en[2]));

    regrinc pc       (.clk(clk), .write_en
(write_en[1]),.data_in(bus_wire),.data_out(pc_out),.inc_en(inc_en[1]));

    alu     alu1     (.clk(clk),.in1(regr_out
),.in2(ac_out),.alu_op(alu_op),.alu_out(alu_out),.z(z));

    control control1 (.clk(clk),.z(z),.instruction (ir_out),.alu_op(alu_op),.write_en (write_en ),
                     .read_en(read_en),.inc_en(inc_en),.end_process (end_process
),.status(status));

    always @ (posedge clk)
        if (status == 2'b01) begin
            dm_en   <= write_en [11];
            im_en   <= 1'b0;
            dar_out <= dar_wire;
            bus_out <= bus_wire;
        end
        else if (status == 2'b10) begin
            dm_en   <= 1'b1;
            im_en   <= 1'b0;
            bus_out <= data_in;
            dar_out <= data_addr_in;
        end
        else if (status == 2'b11) begin
            dm_en   <= 1'b0;
            im_en   <= 1'b0;
            dar_out <= data_addr_in;
        end
endmodule
```

6.3.9 Registers without increment

```
module regr (
    input              clk, write_en,
    input       [15:0] data_in,
    output reg  [15:0] data_out
    );

    always @(posedge clk) begin
        if (write_en == 1)
            data_out <= data_in;
    end
endmodule
```

6.3.10 Registers with increment

```
module regrinc (
    input              clk, write_en, inc_en,
    input       [15:0] data_in,
    output reg  [15:0] data_out = 16'd0
    );

    always @(posedge clk) begin
        if (write_en == 1)
            data_out <= data_in;
        if (inc_en == 1)
            data_out <= data_out + 16'd1;
    end
endmodule
```

6.3.11 Test bench

```
`timescale 1 ns / 1 ps

module tb_machine();
    reg        clk;
    reg [1:0]  status;
    reg [15:0] addr;
    reg [7:0]  data;

    wire       end_process;
    wire [7:0] out;

    integer    f;
    reg [7:0]  read_data [0:65535];

    machine dut(clk, status, data, addr, end_process, out);

    initial begin
        clk = 0;
        forever clk = #(50) ~clk;
    end

    initial begin
        $readmemb("image.txt", read_data);
        f = $fopen("output.txt", "w");
        status = 2'b10;
    end

    integer i;
    initial begin
        for (i=0;i<256*256;i=i+1) begin
            addr = i;
            data = read_data[i];
            #100;
        end
        status = 2'b01;
        while (end_process != 1'b1) begin
            #100;
        end
        status = 2'b11;
        for (i=0;i<128*128;i=i+1) begin
            addr = i;
            #200
            $fwrite(f, "%u\n", out);
        end
        $fclose(f);
    end
endmodule
```

6.4 Generating images

6.4.1 By processor implementation

```
import cv2
import numpy as np

# Show the original image
inputImage = open("input.txt", "r")
content = inputImage.read()
contentList = content.split("\n")
inputImage.close()
intList = [int(x, 2) for x in contentList]
uint8List = [np.uint8(x) for x in intList]
image = np.array(uint8List)
imageIn = image.reshape(256, 256)
print("Input image:\n", imageIn)
cv2.imshow('original', imageIn)
cv2.waitKey()

# Show the filtered image
inputImage = open("output_filtered.txt", "r")
content = inputImage.read()
contentList = content.split("\n")
inputImage.close()
intList = [int(x, 2) for x in contentList]
uint8List = [np.uint8(x) for x in intList]
image = np.array(uint8List)
imageFiltered = image.reshape(256, 256)
print("\nFiltered image:\n", imageFiltered)
cv2.imshow('Filtered', imageFiltered)
cv2.waitKey()

# Show the downsampled image
inputImage = open("output_downscaled.txt", "r")
content = inputImage.read()
contentList = content.split("\n")
inputImage.close()
intList = [int(x, 2) for x in contentList]
uint8List = [np.uint8(x) for x in intList]
image = np.array(uint8List)
imageDownsampled = image.reshape(128, 128)
print("\nDownsampled image:\n", imageDownsampled)
cv2.imshow('Downsampled', imageDownsampled)
cv2.waitKey()
cv2.destroyAllWindows()
```

6.4.2 By Python implementation

```python
import cv2
import numpy as np

# Show the original image
inputImage = open("input.txt", "r")
content = inputImage.read()
contentList = content.split("\n")
inputImage.close()
intList = [int(x, 2) for x in contentList]
uint8List = [np.uint8(x) for x in intList]
image = np.array(uint8List)
imageIn = image.reshape(256, 256)
print("Input image:\n", imageIn)
cv2.imshow('Original', imageIn)
cv2.waitKey()

# Do the filtering
currentPixel    = 257
currentRowPixel = 0
totalRowPixels  = 253
totalPixels     = 65278

while True:
    middle          =   np.uint16(image[currentPixel])
    right           =   np.uint16(image[currentPixel+1])
    left            =   np.uint16(image[currentPixel-1])
    bottom          =   np.uint16(image[currentPixel+256])
    top             =   np.uint16(image[currentPixel-256])
    bottomRight     =   np.uint16(image[currentPixel+257])
    topLeft         =   np.uint16(image[currentPixel-257])
    bottomLeft      =   np.uint16(image[currentPixel+255])
    topRight        =   np.uint16(image[currentPixel-255])
    storeLoc        =   currentPixel-257
    image[storeLoc] =   np.uint8((middle*4 + (right+left+bottom+top)*2 +
(bottomLeft+bottomRight+topLeft+topRight))/16)

    if currentPixel ==  totalPixels:
        break
    elif currentRowPixel == totalRowPixels:
        currentRowPixel  =  0
        currentPixel     =  currentPixel+3
    else:
        currentRowPixel  =  currentRowPixel+1
        currentPixel     =  currentPixel+1

# Show the filtered image
imageFiltered = image.reshape(256, 256)
```

```python
print("\nFiltered image:\n", imageFiltered)
cv2.imshow('Filtered', imageFiltered)
cv2.waitKey()

# Do the downsampling
currentPixelLoc = 0
currentStoreLoc = 0
currentRowPixel = 0
totalRowPixels  = 254
totalPixels     = 65278

while True:
    image[currentStoreLoc] = image[currentPixelLoc]
    if currentPixelLoc == totalPixels:
        break
    elif currentRowPixel == totalRowPixels:
        currentRowPixel = 0
        currentPixelLoc = currentPixelLoc+258
        currentStoreLoc = currentStoreLoc+1
    else:
        currentRowPixel = currentRowPixel+2
        currentPixelLoc = currentPixelLoc+2
        currentStoreLoc = currentStoreLoc+1

# Show the downsampled image
imageDownsampled = image[:16384].reshape(128, 128)
print("\nDownsampled image:\n", imageDownsampled)
cv2.imshow('Downsampled', imageDownsampled)
cv2.waitKey()

# Get the image downsampled by the processor
inputImage = open("output_downscaled.txt", "r")
content = inputImage.read()
contentList = content.split("\n")
inputImage.close()
intList = [int(x, 2) for x in contentList]
uint8List = [np.uint8(x) for x in intList]
image = np.array(uint8List)
imageDownsampledMachine = image.reshape(128, 128)

# Calculate and show the error
errorArray = imageDownsampled - imageDownsampledMachine
error = np.count_nonzero(errorArray, axis=None, keepdims=False)
print("\nError:", error)
cv2.waitKey()
cv2.destroyAllWindows()
```