**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** 3](#_Toc108112218)

[**1.** **Introduction** 4](#_Toc108112219)

[**1.1 Processor Design** 4](#_Toc108112220)

[**1.2 Central Processing Unit (CPU)** 4](#_Toc108112221)

[**1.3 Microprocessor** 4](#_Toc108112222)

[**1.4 Problem Statement** 5](#_Toc108112223)

[**1.5 Proposed Solution** 5](#_Toc108112224)

[**2. Designing the Processor** 6](#_Toc108112225)

[**2.1 Datapath and Control Signals** 6](#_Toc108112226)

[**2.2 Instruction Set** 6](#_Toc108112227)

[**2.3 Microinstructions** 6](#_Toc108112228)

[**2.4 Finite State Machines for Microinstructions** 10](#_Toc108112229)

[**2.5 Modules and Components** 10](#_Toc108112230)

[**2.5.1 Machine** 10](#_Toc108112231)

[**2.5.2 Inside of the machine** 11](#_Toc108112232)

[**2.5.3 Processor** 11](#_Toc108112233)

[**2.5.4 Modules Inside the Processor** 12](#_Toc108112234)

[**2.5.5 ALU** 12](#_Toc108112235)

[**2.5.6 Control Unit** 13](#_Toc108112236)

[**2.5.7 Accumulator** 14](#_Toc108112237)

[**2.5.8 Register Which Can Be Incremented** 15](#_Toc108112238)

[**2.5.9 Register Which Cannot Be Incremented** 15](#_Toc108112239)

[**2.5.10 Data Bus** 15](#_Toc108112240)

[**2.5.11 Data Memory** 16](#_Toc108112241)

[**2.5.12 Instruction Memory** 16](#_Toc108112242)

[**3. Algorithm and Design Considerations** 17](#_Toc108112243)

[**3.1 Filtering & Down-sampling** 17](#_Toc108112244)

[**3.2 Low Pass Filter** 17](#_Toc108112245)

[**3.3 Down-sampling** 18](#_Toc108112246)

[**3.4** **Data Processing Techniques** 19](#_Toc108112247)

[**4.** **RTL Design Simulation** 21](#_Toc108112248)

[**5.** **References** 22](#_Toc108112249)

[**6. Appendix** 23](#_Toc108112250)

[**6.1 Creating input file (Python)** 23](#_Toc108112251)

[**6.2 Assembly code** 23](#_Toc108112252)

[**6.2.1 Compiler to binary values (Python)** 26](#_Toc108112253)

[**6.2.2 Compiler to ram (Python)** 28](#_Toc108112254)

[**6.3 Verilog codes for processor design** 29](#_Toc108112255)

[**6.3.1 Accumulator** 29](#_Toc108112256)

[**6.3.2 ALU** 29](#_Toc108112257)

[**6.3.3 Bus** 30](#_Toc108112258)

[**6.3.4 State machine** 30](#_Toc108112259)

[**6.3.5 Data memory** 41](#_Toc108112260)

[**6.3.6 Instruction memory** 41](#_Toc108112261)

[**6.3.7 Top module** 45](#_Toc108112262)

[**6.3.8 Processor** 46](#_Toc108112263)

[**6.3.9 Registers without increment** 48](#_Toc108112264)

[**6.3.10 Registers with increment** 48](#_Toc108112265)

[**6.3.11 Test bench** 48](#_Toc108112266)

[**6.4 Generating images** 50](#_Toc108112267)

[**6.4.1 By processor implementation** 50](#_Toc108112268)

[**6.4.2 By Python implementation** 51](#_Toc108112269)

# **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.

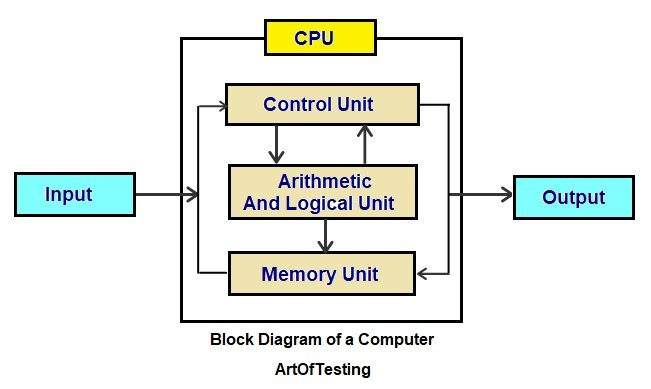
# **1.** **Introduction**

## **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**

Give the output data to convert the data stream into an image

Logo

Description automatically generated with medium confidence

Down sample the input image

Filter the input image

Give the input data stream to the memory unit

Convert the pixel data of the original image



A python program with open cv library is used to convert the pixel data into a binary data stream.

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

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.

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](https://drive.google.com/file/d/1rPPUsLKs8p134EQi3qqJmZcQUA2audfj/view?usp=sharing)

## **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](https://drive.google.com/file/d/1sTfO4wpmlyjbBeBT12DA6SesE3SfmX9s/view?usp=sharing)

## **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

Diagram, schematic

Description automatically generatedIn 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.

|  |  |
| --- | --- |
| 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 |  |

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

### **Chart, diagram Description automatically generated2.5.2 Inside of the machine**

### **Diagram Description automatically generated2.5.3 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](https://drive.google.com/file/d/19FY7MgWt2P39a2HArgFuCaK_7dW6WJbC/view?usp=sharing)

### **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

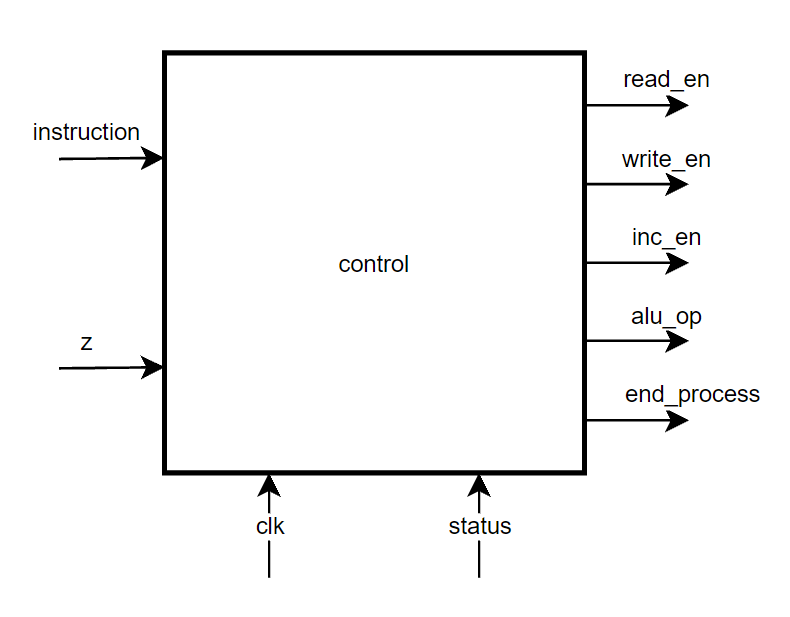
### **2.5.5 ALU**

A picture containing text, antenna

Description automatically generatedALU 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.

Diagram, schematic

Description automatically generated

### **Diagram Description automatically generated2.5.8 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.

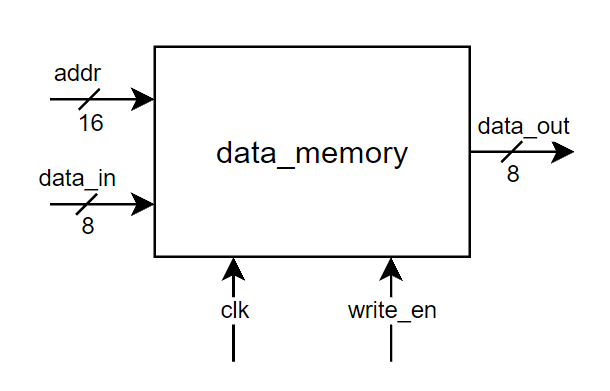
### **Diagram Description automatically generated2.5.9 Register Which Cannot Be Incremented**

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

### **Diagram Description automatically generated2.5.10 Data Bus**

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

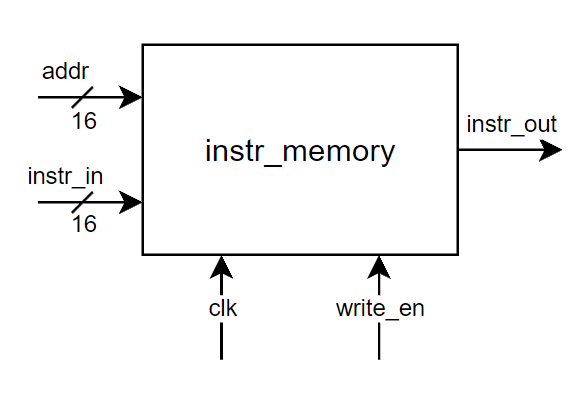
### **2.5.11 Data Memory**



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.

## **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.



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.

Table

Description automatically generatedIn 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.

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.

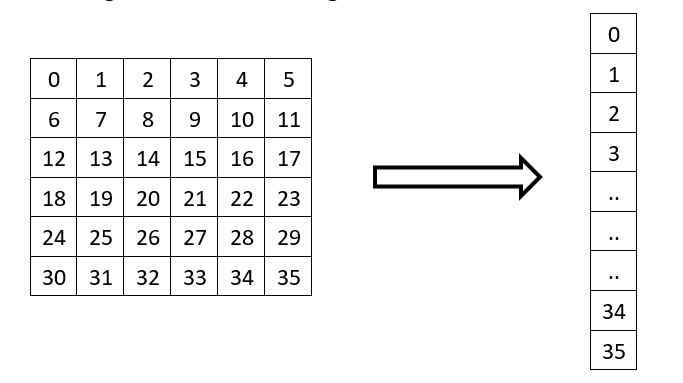
## **3.3 Down-sampling**

Diagram

Description automatically generated with low confidenceDown 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.

## **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.



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.

A screenshot of a person in a hat

Description automatically generated with low confidence

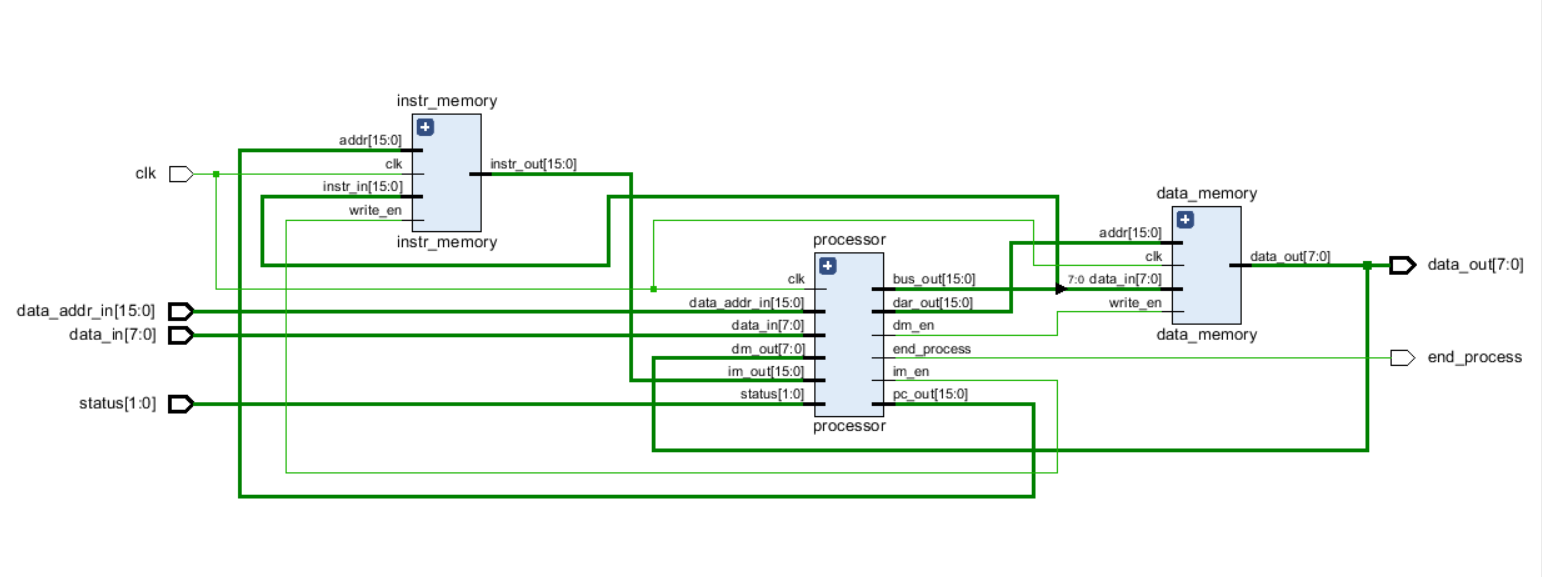
*Original, filtered and down sampled images from the processor*

A person wearing a hat

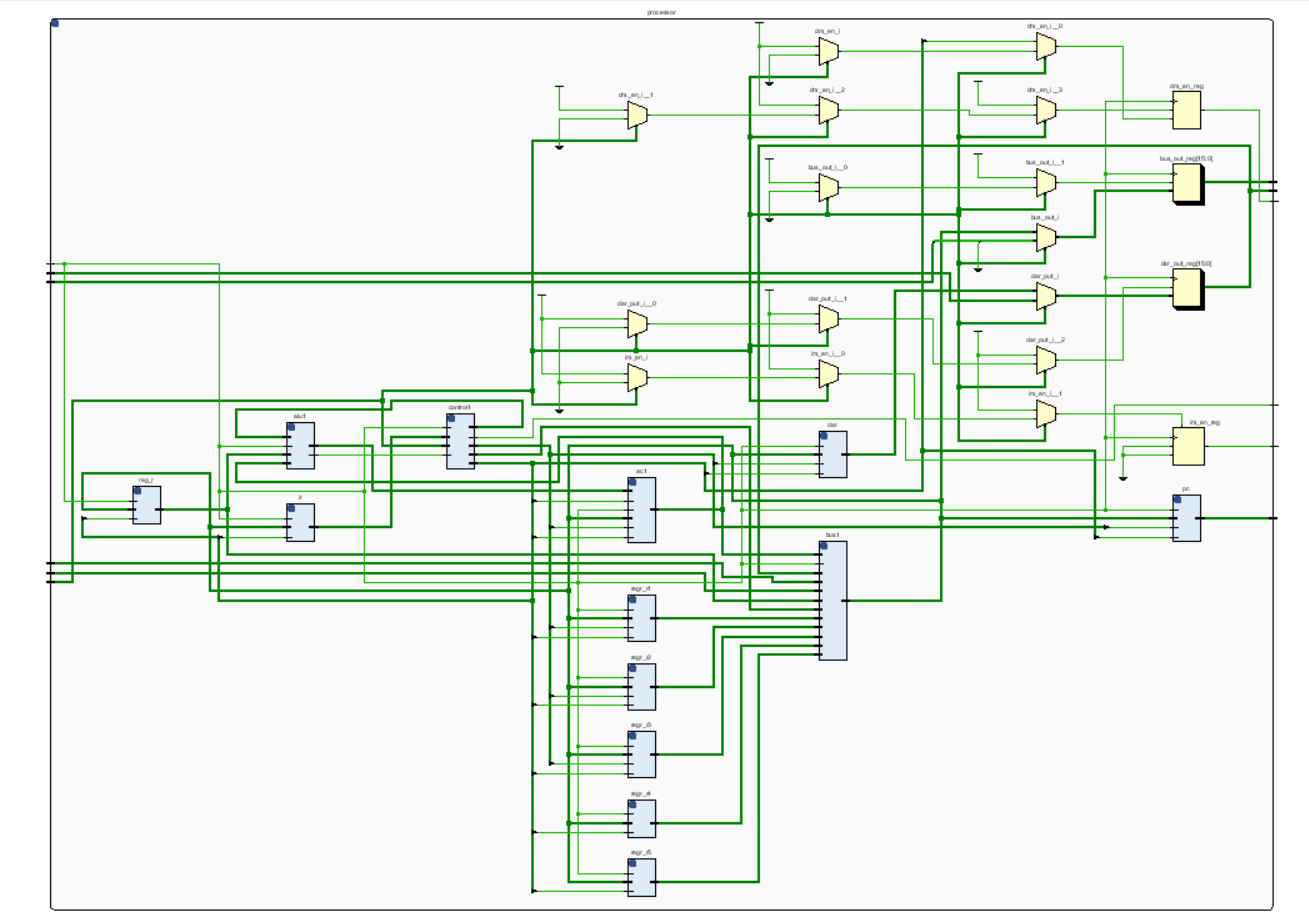
Description automatically generated with low confidence

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

# **RTL Design Simulation**



Processor with memory modules



Processor without memory modules

# **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)**

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**

1. loadim
2. 16'd257 // Starting from 257th bit (2nd row, 2nd bit)
3. movacr1
4. movr1ac // Start of the filtering loop
5. loadac
6. movacr // R <- middle
7. loadim
8. 8'd2 // To multiply by 4
9. lshift
10. movacr4 // R4 <- middle\*4
11. loadim
12. 8'd1 // To get right and left bits
13. movacr
14. movr1ac
15. add
16. loadac
17. movacr5 // R5 <- right
18. movr1ac
19. sub
20. loadac
21. movacr
22. movr5ac
23. add
24. movacr5 // R5 <- right + left
25. loadim
26. 16'd256 // To get bottom bit
27. movacr
28. movr1ac
29. add
30. loadac
31. movacr
32. movr5ac
33. add
34. movacr5 // R5 <- right + left + bottom
35. loadim
36. 16'd256 // To get top bit
37. movacr
38. movr1ac
39. sub
40. loadac
41. movacr
42. movr5ac
43. add
44. movacr // R <- right + left + bottom + top
45. loadim
46. 8'd1 // To multiply by 2
47. lshift
48. movacr // R <- (right + left + bottom + top)\*2
49. movr4ac
50. add
51. movacr4 // R4 <- middle\*4 + (right + left + bottom + top)\*2
52. loadim
53. 16'd257 // To get bottom\_right and top\_left bits
54. movacr
55. movr1ac
56. add
57. loadac
58. movacr5 // R5 <- bottom\_right
59. movr1ac
60. sub
61. loadac
62. movacr
63. movr5ac
64. add
65. movacr5 // R5 <- bottom\_right + top\_left
66. loadim
67. 8'd255 // To get bottom\_left bit
68. movacr
69. movr1ac
70. add
71. loadac
72. movacr
73. movr5ac
74. add
75. movacr5 // R5 <- bottom\_right + top\_left + bottom\_left
76. loadim
77. 8'd255 // To get top\_right bit
78. movacr
79. movr1ac
80. sub
81. loadac
82. movacr
83. movr5ac
84. add
85. movacr // R <- bottom\_right + top\_left + bottom\_left + top\_right
86. movr4ac
87. add
88. movacr // R <- middle\*4 + (right+left+bottom+top)\*2 + (bottom\_right+top\_left+bottom\_left+top\_right)
89. loadim
90. 8'd4 // To divide by 16
91. rshift
92. movacr4 // R4 <- (middle\*4 + (right+left+bottom+top)\*2 + (bottom\_right+top\_left+bottom\_left+top\_right))/16
93. loadim
94. 16'd257 // To find store location
95. movacr
96. movr1ac
97. sub
98. movacdar
99. movr4ac
100. stac
101. loadim
102. 16'd65278 // To check end of the image ((256\*254) + (255-1))
103. movacr
104. movr1ac
105. sub
106. jumpz
107. 8'd127 // Go to the start of downscaling
108. loadim
109. 8'd253 // To check end of the row
110. movacr
111. movr2ac
112. sub
113. jumpz
114. 8'd118
115. incr1
116. incr2
117. jump
118. 8'd3
119. incr1
120. incr1
121. incr1
122. loadim
123. 8'd0
124. movacr2
125. jump
126. 8'd3 // Go to the start of the filtering loop
127. endop
128. loadim // Start of downscaling
129. 8'd0
130. movacr1 // R1 <- 0, R1 = Keep track of current pixel
131. movacr2 // R2 <- 0, R2 = Keep track of current row pixel
132. movacr3 // R3 <- 0, R3 = Keep track of store location
133. movr1ac // Start of the downscaling loop
134. loadac
135. movacr4
136. movr3ac
137. movacdar
138. movr4ac
139. 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\_addr\_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\_out(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\_en(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**

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)

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()