Department of Electronic & Telecommunication Engineering

University of Moratuwa

EN 3030 – Circuits and Systems Design

Processor Design Project

Final Report

Name	***Index Number***
Rathnayaka R.G.H.V.	*180529E*
Sewwandi B.L.P.N.	*180589K*
Thilakarathna G.D.O.L.	*180642T*
Udara A.W.T.	*180650P*

Supervisor

Dr. Jayathu Samarawickrama

July 14, 2022

This Report is submitted as a partial fulfillment of the requirements for the module EN 3030 – Circuits and Systems Design

Department of Electronic and Telecommunication Engineering,

University of Moratuwa.

Abstract

Object or task specified microprocessors plays a major role in consumer electronics world today increasing their popularity day by day. Low power consumption, economic feasibility, reliability, efficiency and ability to be enclosed in a very tiny space are the major factors that tempts to make a pivotal trend for them in the industry.

This report was compiled as a partial fulfillment of the requirements for the module EN3030 – Circuits and System Design under the supervision of Dr. Jayathu Samarawickrama. Report contains a detailed discussion on the design of a custom processor which was purposely designed for filtering an input image and down sampling the image by a specific factor. You will find with theoretical concepts and methodologies used in filtering and down sampling an image.

1 Table of Contents

1 Introduction

The main objective of this project is to design a microprocessor and a central processing unit (CPU) which can filter and down sample the provided 256-pixel x 256-pixel image. Out of the available Hardware Descriptive Languages (HDL), Verilog was used in this processor design simulations and **Python 3.9** with OpenCV functions on **Jupyter Notebook** was used for the simulations and testing purposes. All the Verilog designs and simulations were done with **Vivado 2018.2** version. The algorithm for filtering and down sampling, Instruction Set Architecture, Complete design process, software simulations and results can be found in this report.

1.1 Central Processing Unit (CPU)

The elementary circuitry in a computer which carries out the instructions of a computer program by means of performing arithmetic, logical, input/output and control operations which are specified by instructions is known as Central Processing Unit (CPU). CPU specifically refers to the Control Unit (CU) and the Processing Unit which can distinguish these core elements of a computer through external components namely input and output circuitry and main memory.

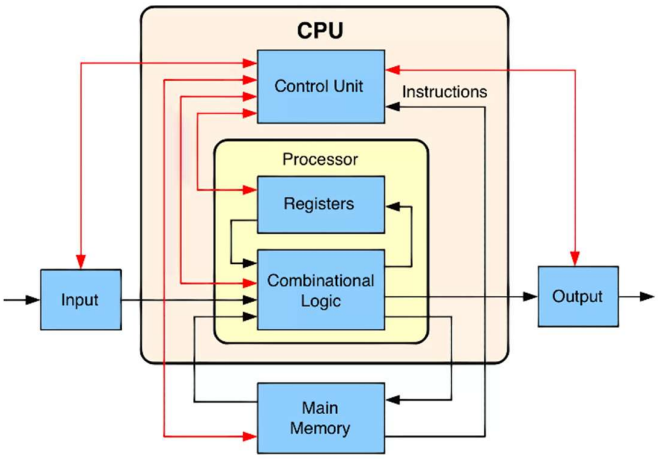


Figure 1.1 – Block diagram of a Central Processing Unit

1.2 Microprocessor

Microprocessor can be defined as the computer processor which can incorporate the functions of a Central Processing Unit (CPU) of a computer on a single integrated circuit or few integrated circuits. This can be identified as a multipurpose, clock driven, register based programmable

digital electronic device which can accept digital or binary data as inputs, process the input data as of the instructions stored in the memory and retrieve the results of processing as outputs. Both combinational logic and sequential logic incorporate with microprocessors, and they can operate on numbers and symbols represented in binary numerical system. Most of the electronic devices rather than computers are using microprocessors. For an example, cars, toys, dimmers, household appliances, electrical circuit breakers, smoke alarms, and battery packs can be pointed out. All they need is a unique featured microprocessors for its own task. So, microprocessors play a major role in modern world electronic and electrical appliances. In order to achieve the specific task, need to be performed such microprocessors should be designed and implemented as suit to the specified task.

1.3 Problem Statement

As the problem statement, the specified task that should be performed by the microprocessor is considered. The main objective of design a CPU and a microprocessor is to filter and down sample a given image. The resolution of the input image, the down sampling factor and the image type were selected as constraints in designing the processor. Here we used an image of 256-pixel x 256-pixel as the input image. We selected the down sampling factor as 2 where we can obtain an image of 128-pixel x 128-pixel as the output image. We used grayscale image as the original image. For the verification purposes, we simultaneously down sampled the same input image using Python 3.9 demonstrations and compared with the output image that was retrieved from the designed processor. The main criteria for choosing pixels for down sampling of the image is selecting a pixel while leaving a pixel which is adjacent to it. There can be a possibility of retrieving a totally different image as a down sampled image if there were many high frequency components in the original image due to aliasing. Here, aliasing means possibility of adjacent pixels with a significant pixel difference.

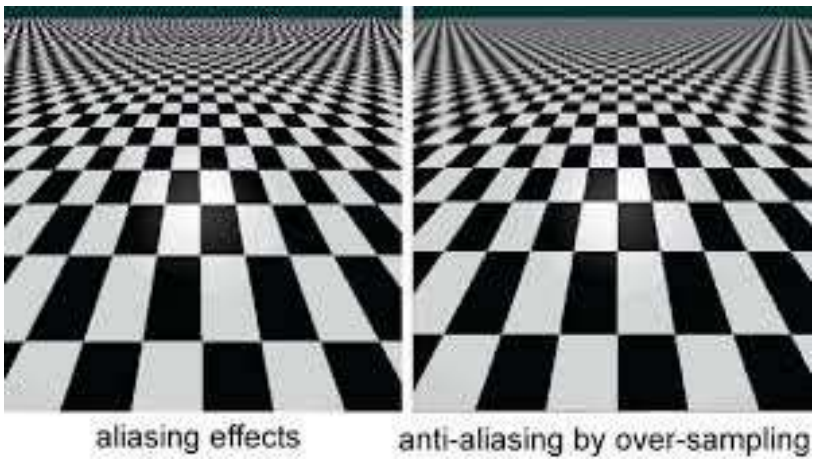


Figure 1.2 – Image with aliasing and anti-aliasing

As a solution for the aliasing effect, filtering the initial original image with a gaussian kernel help to avoid this aliasing effect as it normalize the high frequency components from the image. So, it is important to have a filtering stage prior to down sampling using the mechanism of selecting a pixel and leaving the adjacent pixel, retains the basic composition of the input image.

The Central Processing Unit should get the data from the serial input and store the pixel data of the original image in the main memory and later need to do required processing for the down sampling. After all CPU need to retrieve the processed data and visualize the results. The whole process can be sub divided into mini tasks as below.

- Control input data signal and store the data in the main memory of designed CPU.

As the processor design is planning only for a software implementation, we are planning to provide the pixel values of the input image as a text file which is taken as a result of a Python 3.9 implementation. When the input data stream is received to the processor it stores the data in its main memory as it can be easily accessed whenever needed.

- Filter the input image.

Whenever the data reception is completed, the processor starts to filter the pixel values as of the provided algorithm. The main aim in this section is to reduce the effect of high frequency components in the image as high frequency components can result a totally different output whenever aliasing is there as discussed above. This process can be performed using a Gaussian filter implementation. After the required processing, processed data is stored in the same main memory. Here, new pixel values are assigned to the existing values of the original image. Overwriting is happened here.

- Down sample the filtered image.

After the image get filtered, as in the requirements, the filtered image is down sampled into half of the size. The pixel by pixel down sampled data is stored in the main memory again.

- Transmit the processed data in the main memory to the computer as a data stream.

For the visualization purpose, the pixel data stored in the main memory of the designed processor should be transmitted out of the processor. The output pixel data values output as a text file containing all the pixels.

- Convert the data stream to an image and display the image.

Finally, the pixel values inside the output text file are converted to an image using the Python 3.9 implementation and displayed as an image.

1.4 Solution for the problem

Considering the task flow of the process, design requirements can be decided. We need to implement Python code for preparing the text files containing the pixel values of the input image in order to transmit the data from the computer to the implemented processor and receive data to the computer back.

For storing the data into main memory (RAM), as we are using a 256-pixel x 256-pixel image as the input RAM need to have a minimum capacity of 65536 bytes to store 65536-pixel data in the primary memory. The processed image is overwritten over the original image in order to save the memory. After that, the processing part is programmed as of the designed Instruction Set Architecture (ISA) which should be capable in handling the filtering part and down sampling part. Proper control of these instructions and data might result in the proper execution of the required processing task.

After the expected processing part, the processed data is delivered to the computer as a set of pixel values and Python 3.9 code is employed for displaying the down sampled image. Then, it is compared with the image filtered and down sampled using Python's inbuilt functions and calculated the square mean error.

2 Algorithm

The main objective of the project is to design a processor for down sampling an image. Algorithm for the processing part of the task is developed based on two parts as,

I. Filtering Algorithm

II. Down Sampling Algorithm

In the algorithm, it first takes the array of the pixel values of the image and then it is filtered using a low pass filter in order to remove high frequency components over the sampling frequency. In order to do that a Gaussian filter is used. Then the filtered image is down sampled using the down sampling algorithm with a down sampling factor of 2. All the testing regarding these two algorithms was implemented with **Python 3.9** under **OpenCV** environment in **Jupyter Notebook.**

2.1 Filtering Algorithm

As the very first part of the task, we had to remove the high pass frequency components in the image. For that, we used a Gaussian low pass filter which can remove the high pass frequency components. Gaussian kernel is a non-uniform low pass filter which is mostly used to blur images and remove noise. For this low pass filtering we employed a Gaussian kernel of 3x3 and used the standard deviation (Sigma) of 0.6. The main consideration for selecting this value is to balance the blur effect and aliasing effect. Increasing the standard deviation will make the image more blur and more robust for aliasing as we learned under the module EN2550 – Fundamentals of Image Processing and Machine Vision. Initially we set the middle pixel of the Gaussian kernel window coordinates as, (x, y) as (0, 0) and other pixels according to the middle pixel as shown in the *Figure 2.1*.

(-1, -1)	(-1, 0)	(-1, 1)
(0, -1)	(0, 0)	(0, 1)
(1, -1)	(1, 0)	(1, 1)

Figure 2.1 - Coordinates of 3x3 Gaussian Kernel

The probability distribution for Gaussian function that was employed in calculating the pixel values of the kernel is as of below.

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

According to the above function, Gaussian kernel can be taken as,

0.0622	0.2494	0.0622
0.2494	1	0.2494
0.0622	0.2494	0.0622

Figure 2.2 - 3x3 Gaussian Kernel

As we are using a discrete Gaussian kernel to convolve with the image, 99% of the distribution falls under standard deviation 3. So, we can limit the kernel size to contain only the three standard deviations of the mean. Central pixels contain higher weightage than the outer pixels. This is the reason that we are employing a 3x3 kernel for this convolution. The values in the above kernel showing decimal values. For easy calculation purposes these values are transformed to integer values as of the total kernel value sum makes a power of 2. Multiplying the kernel values by 16 we could be able to get the kernel matrix as,

0.9952	3.9904	0.9952
3.9904	16	3.9904
0.9952	3.9904	0.9952

Figure 2.3 - 3x3 Gaussian Kernel

Now, the sum of the kernel values gets approximate to 36, which is not a power of 2. In order to approximate the kernel sum to a power of 2, the 3.9904 is floored to 3 instead of approximating to 4. But this error needs to be considered during the evaluation. This will result a kernel which can satisfy the integer values and their sum as a power of 2 as below.

1	3	1
3	16	3
1	3	1

Figure 2.4 - 3x3 Gaussian Kernel

The selected image of N x N pixels was convoluted with this 3 x 3 Gaussian kernel which outputs an image where low pass component are there. In this convolution process, overlapping the center pixel of the kernel with an image pixel provides a weighted summation value. This result is divided by the total weight of the kernel (here it is 32) to normalize. Then, this average value is stored at the top left corner pixel location in the RAM. After the initial convolution operation, the kernel is moving forward and the average value of each set of convolutions is stored at the top left corner pixel. After finishing the forward move kernel move downwards and again the procedure is continuing until all the set of pixels are covered. Filtered image is completely stored in the RAM in this sequential order. As we neglected the effect surrounding marginal pixels' effect. So, no padding was done for the 3 x 3 kernel.

2.2 Down Sampling Algorithm

Since it is needed to down sample the image from both height and width with a down sampling factor of 2, a one value is taken per four pixels from the image obtained through convolution with the 3 x 3 kernel. Then, that value is kept stored in the memory for getting the output. Values taken from a 6 x 6 filtered image is shown in the *Figure 2.5* below.

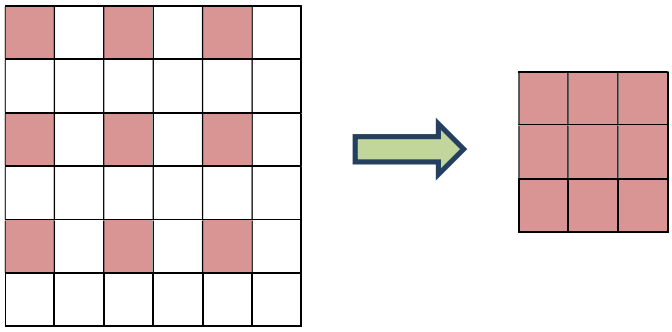


Figure 2.5 – Graphical Illustration of Down Sampling

Every other pixel from the filtered image is used in constructing the output image. No interpolation is used as the image is already down sampled. Python implementation done with **Jupyter Notebook 6.0** to filter the image to avoid high frequency components and down sample can be found under **Appendix**.

3 Processor Design

3.1 Instruction Set Architecture (ISA)

3.1.1 General Architecture

The main objective associated with this processor is down sampling the image with selected constraints. As we are down sampling a 256 x 256 image, there should be 65536 memory locations in order to store data related to 65536 pixels. Processor architecture was designed basically based on below considerations.

- *DRAM* – Data RAM which is for storing the pixel data of 65536 pixels (256-pixel x 256-pixel) in its 65536 of memory locations and a word size of 8 bits (1 Byte) where there the pixel values of the image can be stored. The minimum size expected from the data memory can be evaluated as,

  Total number of pixels in the input image----------256 x 256.

  Intensity value width per pixel---------- 8 bits.

  Memory size needed to store input image----------256 x 256 x 8 bits.

- *IRAM* – Instruction RAM which contains the assembly code of algorithm where we used for filtering and down sampling the image. This is comprised of 256 memory locations where instructions that need to be executed stored with a width of 8 bits.

- *Program Counter (PC)* – Need 16 bits to store the address of the next instruction which needs to be executed in the instruction RAM.

- *Accumulator (AC)* – Need 16 bits accumulator which has a direct access to Arithmetic and Logical Unit (ALU). Data stored in the main memory is first loaded to the accumulator and then processed from the ALU and after that stored in the main memory via accumulator.

- *Address Register (AR)* – Need a 16 bit register in order to point the address in DRAM.

- *Instruction Register (IR)* – Need an 8 bit register in order to store the instructions from IRAM.

- *Arithmetic and Logical Unit (ALU)* – Need 16 bits to performs seven different operations.

- *R, R1, R2, TR* – 16-bit general purpose registers.

- *Control Unit* – A controller (State machine) which can generate all the control signals for the processor make the decisions based on the provided instructions from the IR.

- *Bus* – 16-bit wire which can carry the data from registers, DRAM and IRAM.

3.1.2 Data Path

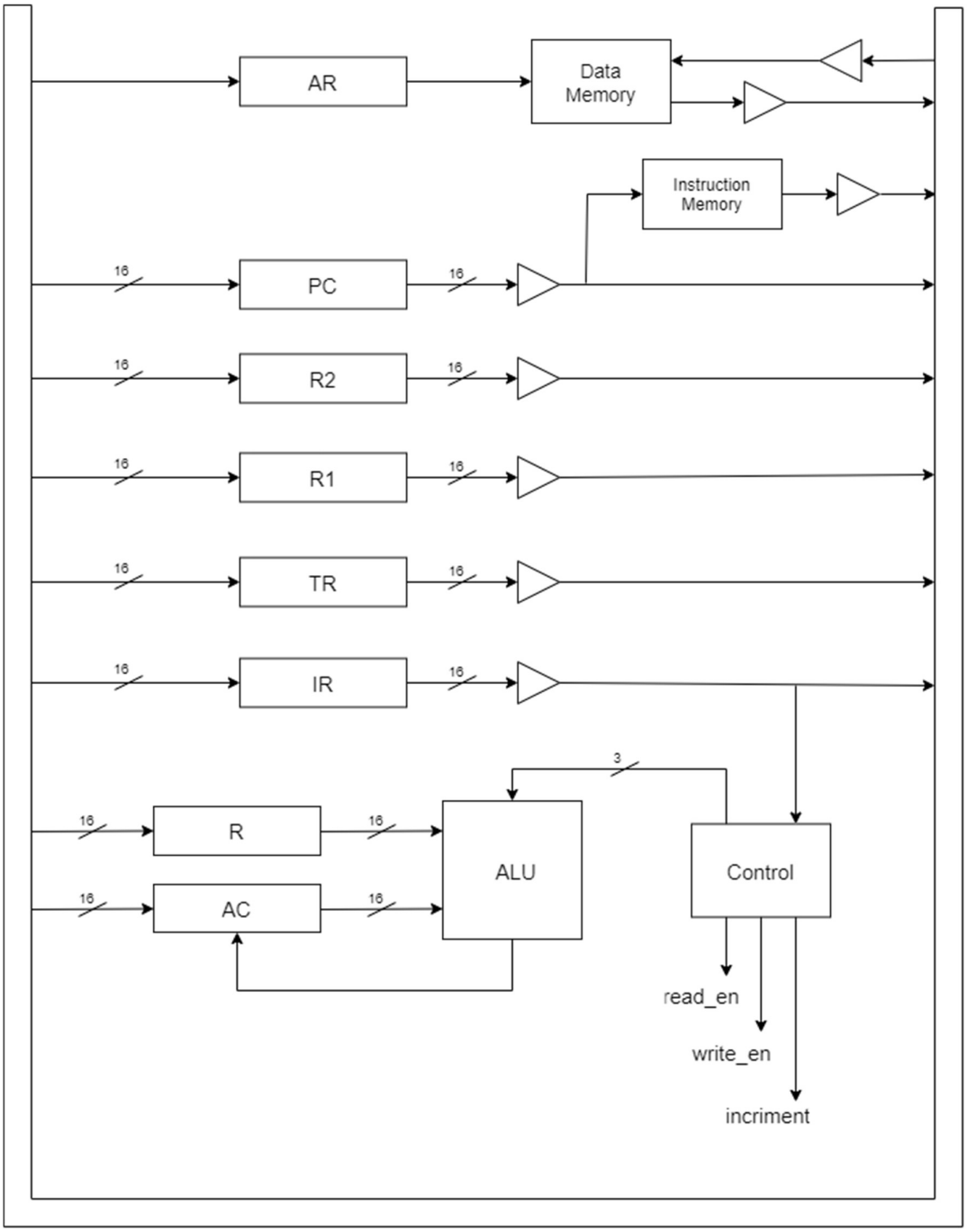


Figure 3.1 – Data Path of the Processor

3.1.3 Instruction Set

For the purpose of performing our desired task of filtering and down sampling the input image from the custom processor, below instruction set was defined for the processor. 32 instructions are used in Instruction Set Architecture.

Instruction	Opcode	Instruction Code	Instruction Operation	Requirement Reason
NOP	5	00000000	No Operation	When idling
CLAC	6	00000001	AC←0, Z←1	Reset Accumulator
LDAC	7	00000010	AC←M[AR]	Load values from data memory. AR is the address to load from.
STAC	8	00000011	M[AR]←AC	Store values to AC from memory. AR is the address to store.
MVACR	10	00000100	R ←AC	Move values from Accumulator to register R
MVACR1	11	00000101	R1←AC	Move values from Accumulator to register R1
MVACR2	12	00000110	R2←AC	Move values from Accumulator to register R2
MVACTR	13	00000111	TR←AC	Move values from Accumulator to register TR
MVACAR	14	00001000	AR←AC	Move values from Accumulator to register AR
MVR	15	00001001	AC←R	Move values from register R to Accumulator
MVR1	16	00001010	AC←R1	Move values from register R1 to Accumulator
MVR2	17	00001011	AC←R2	Move values from register R2 to Accumulator
MVTR	18	00001100	AC←TR	Move values from register TR to Accumulator
INCAR	19	00001101	AR←AR+1	Enable quick increments in R1 without using ALU
INCR1	20	00001110	R1←R1+1	Enable quick increments in AR without using ALU
INCR2	21	00001111	R2←R2+1	Enable quick increments in R2 without using ALU
JPNZ $\mathcal{T}$	22	00010000	IF Z=0 THEN GOTO $\mathcal{T}$	Control the order of instruction execution to implement while loops and if condition. τ is the address to jump to
ADD	27	00010001	AC←AC+R	Arithmetic Operations
SUB	28	00010010	AC←AC-R If AC-R=0 THEN Z←1 ELSE Z←0	Arithmetic Operations
MUL4	29	00010011	AC←AC*4	Arithmetic Operations
DIV2	30	00010100	AC←AC/4	Arithmetic Operations
ADDM α	31	00010101	AC←AC+α	Arithmetic Operations
END	32	00010110	End Of Operation	Ending the process

Table 1 - Instruction Set (IS)

3.1.4 Micro Instruction Sequence

The processor operation is basically depended on the state machine or controller which executes a single task at an instance. For this purpose, control unit follows a three-component cycle which is executed repeatedly whereas the current state is a direct result of the previous state. The figure shown below is a graphical representation of the state machine of the expected processor.

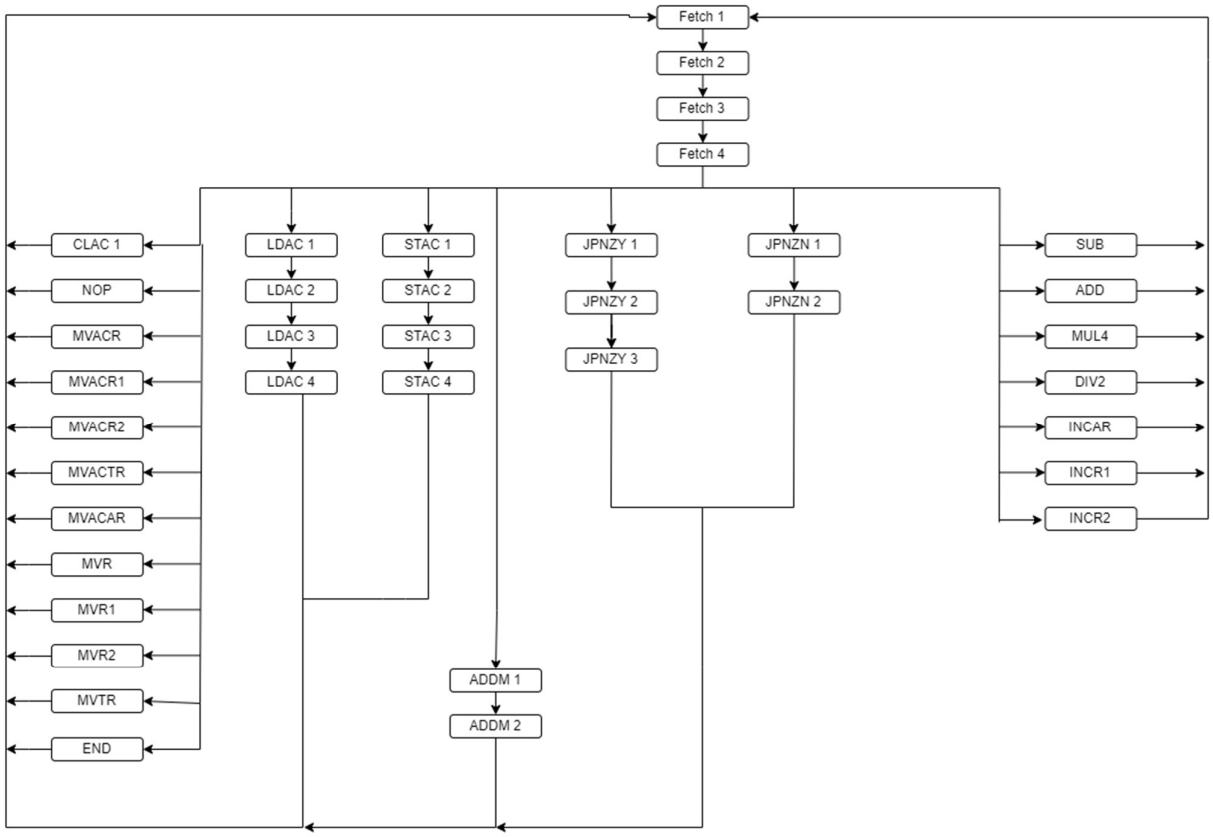


Figure 3.2 – Graphical Representation of State Machine

3.1.5 FETCH, DECODE, EXECUTION Cycle

Let's see the hardwired controlling in the CPU design one by one according to the FETCH, DECODE, EXECUTION cycle.

3.1.5.1 FETCH

In this architecture The FETCH instruction comprises of four states which results in fetching the next instruction from the Instruction memory (IRAM) which need to be loaded into the Instruction Register (IR).

- FETCH 1 – initial fetch
- FETCH 2 – IR ← IRAM
- FETCH 3 – PC ← PC + 1
- FETCH 4 – Next State ← IR

3.1.5.2 DECODE

After the fetching the instructions from the Instruction memory, CPU need to know the relevant instruction fetched thereby invoking the correct execution cycle. This process is performed by the control unit. Then, the Instruction Register (IR) supplies the necessary fetched instruction to the control unit and control unit execute the relevant state followed by the next states of the instruction. Unless it returns to the FETCH cycle if the instruction contains only one state.

3.1.5.3 EXECUTION

- **Clear Instructions**

  This instruction CLAC is for clearing the contents in the accumulator and setting the Z value as 1 in order fetch the next instruction from the IRAM. This is executed by one state.

  - CLAC - AC←0, Z←1

- **Load Instructions**

  This instruction LDAC is for loading the data stored in the memory which is pointed by the Address Register (AR) to accumulator. This is executed by one state.

  - LDAC - AC←M[AR]

- **Store Instructions**

  This instruction STAC is for copying the data in the accumulator to store it in the memory which is pointed by Address Register (AR). This is executed by two states as below.

  - STAC 1 - M[AR]←AC

- STAC 2 – Write

- **Move Instructions**

  These instructions are for copying the data in the accumulator to respective registers and copying the data in relevant registers to accumulator. These are executed by single state as below.

  - MVACR - R←AC
  - MVACR1 – R1←AC
  - MVACR2 – R2←AC
  - MVACTR - TR←AC
  - MVACAR - AR←AC
  - MVR - AC←R
  - MVR1 - AC←R1
  - MVR2 - AC←R2
  - MVTR - AC←TR

- **Increment Instructions**

  These instructions are for incrementing the data in the register from 1 and get written into the same register. These are executed by single state as below.

  - INCAR - AR←AR + 1
  - INCR1 – R1←R1 + 1
  - INCR2 – R2←R2 + 1

- **Arithmetic Operation Instructions**

  These instructions are for performing required arithmetic operations by taking the values from the accumulator and the register R and writing the processed value to the accumulator again. These are executed by single state as below.

  - ADD - AC←AC + R
  - SUB – AC←AC - R
  - MUL4 – AC←AC * 4
  - DIV2 - AC←AC / 2

- **ADDM Instructions**

  These instructions are for adding The Instruction RAM value to accumulator and write back to the accumulator. These are executed by two states as below.

- ADDM 1 – FETCH, PC←PC + 1
- ADDM 2 –AC←AC + α

➢ **JPNZ Instructions**

In these instructions if Z=1, Program Counter (PC) is incremented by 1 and CPU move to the fetch routine and start to fetch the next instruction need to be executed.

- JPNZY1 – PC←PC + 1

If Z=0, the three states get involved. During the second instruction in the memory address is loaded into accumulator. After that, the value of the accumulator is copied to Program Counter (PC). Finally, the CPU move back to fetch routine.

- JPNZN1 – FETCH
- JPNZN2 – AC← $\mathcal{T}$
- JPNZN3 – PC← AC

➢ **No Operation Instructions**

These instructions are for backing to fetch operation.

- NOP – no operation

➢ **End of Operation Instructions**

These instructions are for ending the processing operation.

- END – ending operation

3.1.6 Assembly Language

The compiled code using Python 3.9 Using Jupyter Notebook code was done with the use of some assembly level codes. The code segment used in implementation using multiple instructions from ISA is given below.

[This code is to filter and down sample a grayscale image of size 256 x 256 pixel to an image of 128 x 128 pixels]

Filtering Part

1	CLAC (Initializing two registers for calculating Address)
2	MVACR1
3	MVACR2
4	MVR2 (Starting the Loop)
5	MUL4 (Calculating Address)
6	MUL4
7	MUL4
8	MUL4
9	MVACR
10	MVR1
11	ADD
12	MVACAR
13	LDAC (Filtering part)
14	MVACR
15	INCAR
16	LDAC
17	MUL4
18	DIV2
19	ADD
20	MVACR
21	INCAR
22	LDAC
23	ADD
24	MVACR
25	MVAR
26	ADDM
27	“254”
28	MVACAR
29	LDAC
30	MUL4
31	DIV2
32	ADD
33	MVACR
34	INCAR
35	MVAR
36	MVACTR

37	LDAC
38	MUL4
39	ADD
40	MVACR
41	INCAR
42	LDAC
43	MUL4
44	DIV2
45	ADD
46	MVACR
47	MVAR
48	ADDM
49	“254”
50	MVACAR
51	LDAC
52	ADD
53	MVACR
54	INCAR
55	LDAC
56	MUL4
57	DIV2
58	ADD
59	MVACR
60	INCAR
61	LDAC
62	ADD
63	DIV2
64	DIV2
65	DIV2
66	DIV2
67	MVACR
68	MVTR
69	MVACAR
70	MOVR
71	STAC
72	INCR1
73	MVR1
74	MVACR
75	CLAC (Checking the loop condition for R1)
76	ADDM
77	“254”
78	SUB
79	JPNZ
80	“4”
81	MVACR1
82	INCR2
83	MVR2
84	MVACR
85	CLAC (Checking the loop condition for R2)

86	ADDM
87	“254”
88	SUB
89	JPNZ
90	“4”

Down-sampling Part

91	CLAC (Initializing two registers for calculating address)
92	MVACR1
93	MVACR2
94	MVR2 (Starting the loop)
95	MUL4 (Calculating Address)
96	MUL4
97	MUL4
98	MUL4
99	MVACR
100	MVR1
101	ADD
102	MVACAR
103	DIV2
104	MVACTR
105	LDAC
106	MVACR
107	INCAR
108	LDAC
109	ADD
110	MVACR
111	MVAR
112	ADDM
113	“255”
114	LDAC
115	ADD
116	MVACR
117	INCAR
118	LDAC
119	ADD
120	DIV2
121	DIV2
122	MVACR
123	MVTR
124	MVACAR
125	STAC
126	INCR1
127	INCR1
128	MVR1
129	MVACR
130	CLAC (Checking the loop condition for R1)
131	ADDM
132	“255”

133	INCAC
134	SUB
135	JPNZ
136	“94”
137	MVACR1
138	INCR2
139	MVR2
140	MVAC
141	CLAC (Checking the loop condition for R2)
142	ADDM
143	“255”
144	INCAC
145	SUB
146	JPNZ
147	“94”
148	END

4 Modules & Components

4.1 Registers

4.1.1 Registers without increment

Registers can be defined as the modules that are used to store the data temporarily for a short period of time during the processing cycle. General purpose registers excluding Instruction Register (IR) can store up to 16 bits of data. Instruction Registers can store 8 bits of data. Data stored in the register always stores at the **Out** and it is connected to the demultiplexer to select the data which need to be read from the bus. As there is not any increment flag in these registers, increment is done through an ALU increment operation and get write back to the register. A register has a 16-bit wide input pin and a 16-bit wide output pin as it is required to handle address locations in the memory which has a maximum length of 16 bits. The **Load** pin is connected to each register module along with a clock pin. If the value of **Load** pin is 1, it can write the data available in **Data** to the register at the positive edge of the clock cycle. A prototype of a register without increment is shown in the figure below.

Figure 4.1 – Block diagram of a register without increment

4.1.2 Registers with increment

Here, the only difference is as there is an increment flag **Increment** with the register. So, for an increment of 1, it is no requirement of go through an ALU operation as it is already performed by the increment flag. Register gets incremented if the increment flag is at its high value in the positive edge of the clock. The main purpose of these registers is keeping the track of loop iterations without the process getting much slower. The block diagram of the register can be depicted as below.

Figure 4.2 – Block diagram of a register with increment

4.1.3 Arithmetic and Logical Unit (ALU)

All the arithmetic and logical operations related to the processor are done through the ALU. The two inputs associated to this module are A bus and B bus both with 16 bits in each. A bus is the output received from the AC register as accumulator is directly connected to the ALU. The output of the ALU is also a 16-bit bus called C bus. C bus is connected to the input of AC register. This transmits the control signal from the control unit that control the operation performed by the ALU. So, any output from the ALU is only written to AC register. In an ALU operation, one operand is stored in R register and other it taken into accumulator in order to do the ALU operation. So, the operand in the AC register is the input for the ALU from bus A while the operand stored in the R register is the input for the ALU from bus B. Afterwards, according to the available ALU operation flag in the module operation will take place. These operations are decided through the control unit according to each instruction.

Operation	Operation flag	Description
ADD	000	C ← A+B
SUB	001	C ← A-B
PASS	010	C ← B
ZERO	011	C ← 0
MUL4	100	C ← A*4
DIV2	101	C ← A/2

Table 1 - ALU Operations

In ALU, z flag is used in indicating whether the output is zero or not. If the output of the ALU is 0, then the flag z is set to 1. So, thereby, z flag is used in identifying the end of a loop. Shown below is a block diagram of an ALU.

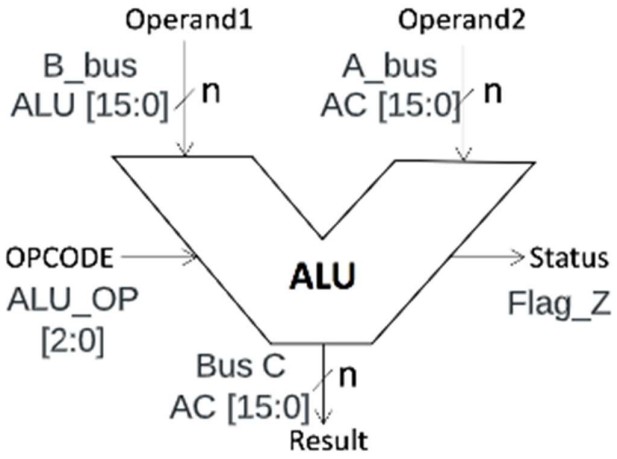


Figure 4.3 – Block Diagram of ALU

4.1.4 Multiplexer (MUX)

The bus B is able only to read one data at an instance from the available 9 input registers directing towards the multiplexer. The block diagram of the multiplexer is shown as of below.

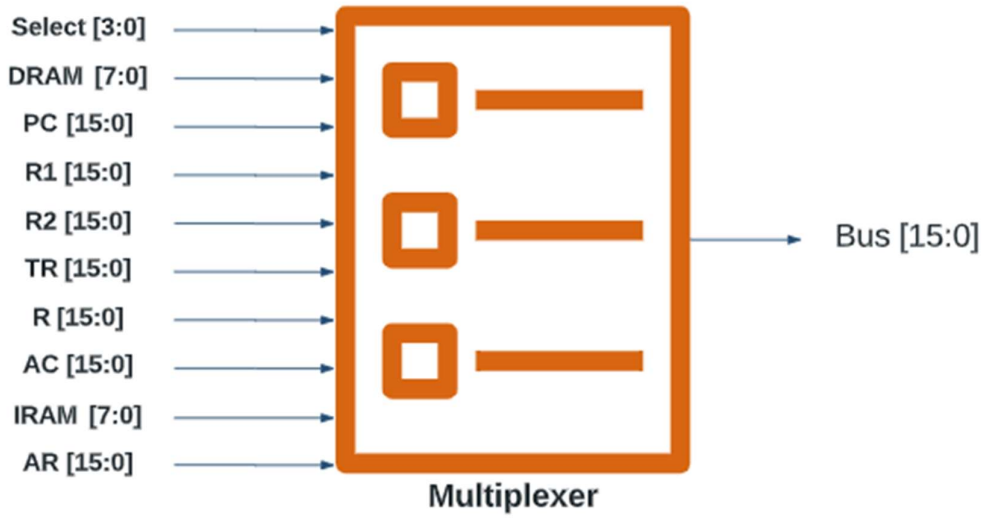


Figure 4.4 – Block Diagram of Multiplexer

In the architecture there are 7 registers to read data which have been connected to the bus as **PC**, **R**, **R1**, **R2**, **TR**, **AR**, and **AC**. Additionally, **DRAM** and **IRAM** are also connected to the bus for the purpose of reading data from RAM to the bus. So, we have to employ 16-bit bus to facilitate the data flow of 7 registers although the two RAMs are in 8-bit word size. So, 8 zeros need to be added in front of both RAMs when reading the data to the bus. As there are totally 9 inputs, we

had to employ 4-bit flag in order to select the register or RAM that needs to be read to the bus. Flag is set as shown in the below table.

Selection Flag	Register / RAM
0000	DRAM
0001	PC
0010	R1
0011	R2
0100	TR
0101	R
0110	AC
0111	IRAM
1000	AR

Table 2 – Selection Flag Index for Registers and RAMs

4.1.5 Control Unit (CU) / State Machine

This is the heart of the processor where all the controlling and signal generation is done. It directs the operation of the processor or guides the way of responding to the instructions that flow to memory, ALU and input output circuitry. The flow of instructions can be basically demonstrated as below.

1. Fetch the instruction from main memory.
2. Decode the instruction into Commands.
3. Execute the commands.
4. Store the results in main memory.

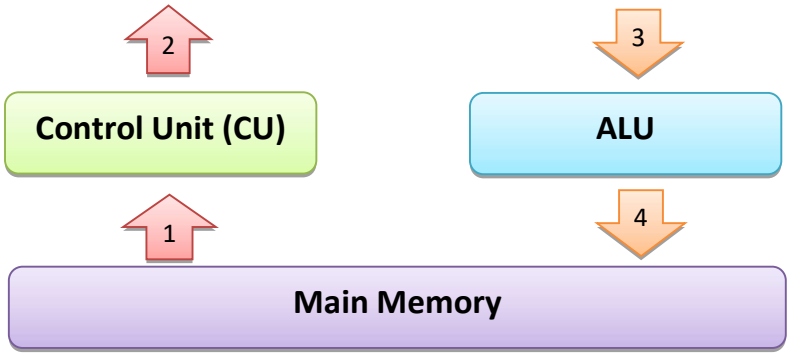


Figure 4.5 – Flow of Instructions

There are different functions performed by the control unit. It can control the data flow inside the processor. Control Unit receives the external instructions and converts them to sequence of signals. After that, control unit applies register transfer level operations for those

instructions. It can decode individual instruction into several sequential steps as fetching addresses and data from registers, memory management during executions and storing data back to the memory or registers. It can schedule the micro instructions between the selected execution units in the Arithmetic and Logical Unit or other available functions. Thereby, control unit is sub divided into three units as, instruction unit, scheduling unit and retirement unit. Retirement unit can handle the results coming out of instruction pipeline. For this control unit, there are three inputs as, **Clock**, **Flag_Z**, and **IR** (Instruction Register) which is 8 bits wide. This module generates mainly 6 outputs as **Fetch**, **Finish**, **ALU_OP**, **Reg_B bus**, **CMD** and **Selectors**. A block diagram of a control unit is shown in the figure below.

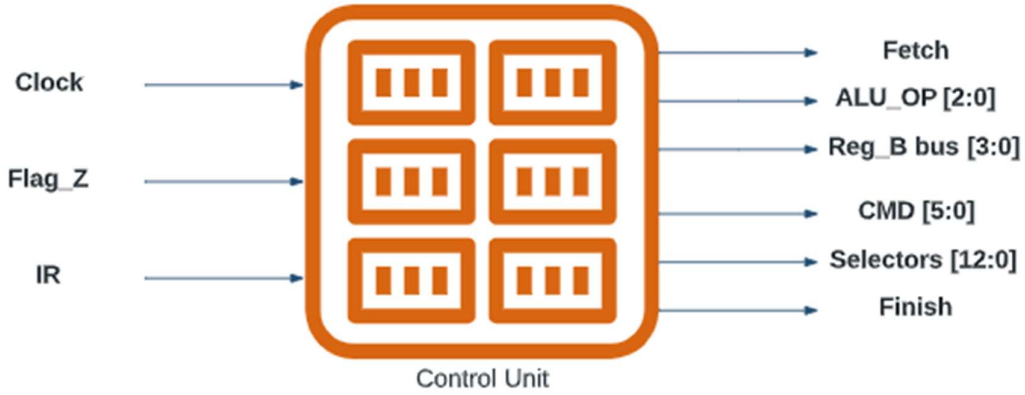


Figure 4.6 – Block Diagram of a Control Unit

Let's concern the outputs one by one.

- **Fetch** – Used in enabling the data write from bus to Instruction Register (IR).
- **Finish** – Used in indicating the end of the process.
- **ALU_OP** – This is a three-bit data value used in controlling the Arithmetic and Logic operations. ALU is operating as of the control signals provided. This flag is selecting the operation which needs to be performed by ALU. The ***Table 1*** shown above clearly depicts the operation with its 3-bit control flag.
- **Reg_B bus** – This is a four-bit data path which is directly connected to multiplexer. This flag control which data has sent from registers, DRAM and IRAM. As the four bits can represent 16 different combinations, this flag can control one data path from 16 different data paths which come out of registers or RAMs as shown in the ***Table 4*** above.
- **Selectors –** Selectors are of two types. One is for loading purpose and other is for incrementing purposes.

✓ **Load** – This control signal has 8 bits to control the registers in the processor. This is the signal which enables the writing to the register from the bus. Whenever the data needs to be stored in a register, writing is enabled through this control signal as an access permission. These 8-bit flow relates to the **Load** pin of each register. Whenever the **Load** pin of a register get high, the value of the bus gets to its memory at the positive clock edge. Multiple writes to several registers at once too can be performed using this control signal.

LD AR	LD PC	LD R1	LD R2	LD TR	LD R	LD AC	LD M

Figure 4.7 – Control signal of write_en flag

✓ **Increment –** This control signal path is connected to the increment pins of registers with increment. If the increment enable pin value gets high, the register is incrementing the current value by one.

INC PC	INC R1	INC R2	INC TR	INC AR

Figure 4.8 – Control signal of inc_en flag

When considering the input control signals **IR** input is the main input which provides the instructions on the next instruction to be executed.

4.1.6 CPU

This is the module which control the instructions and the instances of other processing modules. This doesn't include any communication or memory related components. The main inputs for the processor module which act as the CPU are,

- **Main Clock** – Providing a clock pulse for synchronization.
- **Data_from_RAM[7:0]** – Output from the data memory to processor module (8 bit)
- **Instruction[7:0]** – Output from the instruction memory to the processor module (8 bit)

Mainly there are 6 output data paths coming out of the processor as,

- **CPU_clock –** Providing a clock pulse for synchronization.
- **CPU_write_en** – Enabling the data memory.
- **Instruction_address[7:0]** – Output the instruction address.
- **Process_finished** – Notifying the end of the processing.
- **CPU_data[7:0]** - Carrying the processed data.
- **CPU_address[15:0]** – Output the data address register value.
- **CMD[5:0]** – Output command Instruction ID
- **Reg_AC[15:0]** – Output for accumulator register.

- **Reg_1[15:0]** – Output for Register R1.
- **Reg_2[15:0]** – Output for Register R2.

There are many modules which have been integrated in the processor module as, clock divider, control unit, multiplexer, registers with increment, registers with increment, accumulators and arithmetic and logical unit. The block diagram for the processor module is shown as below.

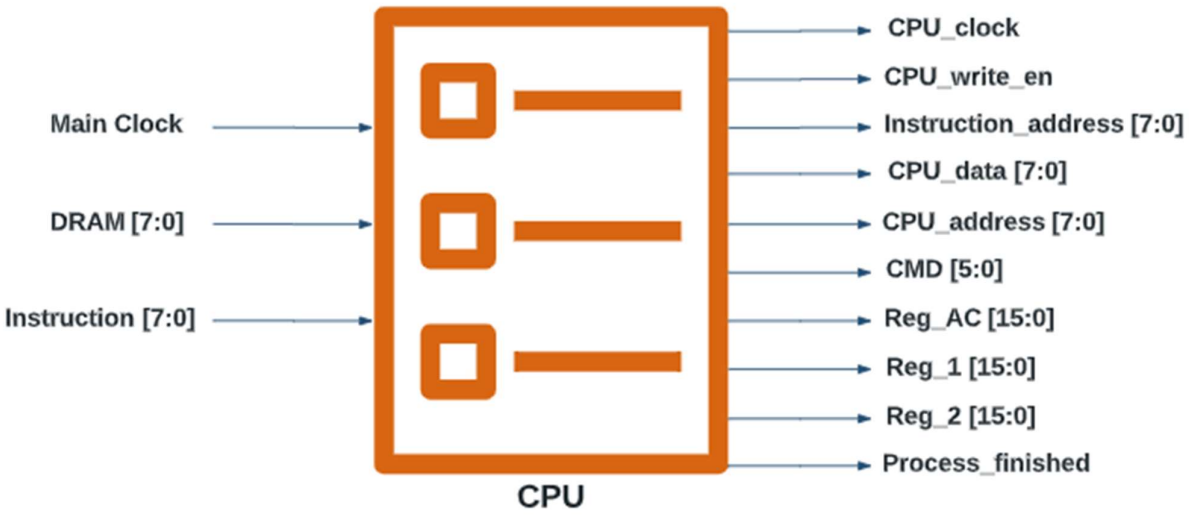


Figure 4.9 – Block diagram of CPU

4.1.7 DRAM (Data RAM)

Data RAM is the primary data memory which is used in storing the image data for processing purpose. After receiving the byte data of the input image, it is stored in the DRAM. This module should have 65536 memory locations with each having an 8-bit word size. Here, 8-bit word size is used as to range each pixel values from 0 to 255. As the inputs for the DRAM. We selected,

- **clock** – For synchronizing the input bits a clock pulse is used.
- **wren** – One bit that give the control signal for the DRAM when there is a need to write data to the memory from the input data path.
- **data –** The pixel values that need to be written into the memory. This is an 8-bit data stream.

- **address –** The memory location which the DRAM must read or write data. This is also an 8-bit data stream.

As the output of this module,

- **dataout** – Outputting the data in the memory when a read pulse is provided to the DRAM.

The block diagram for the DRAM with its inputs and outputs is as of below.

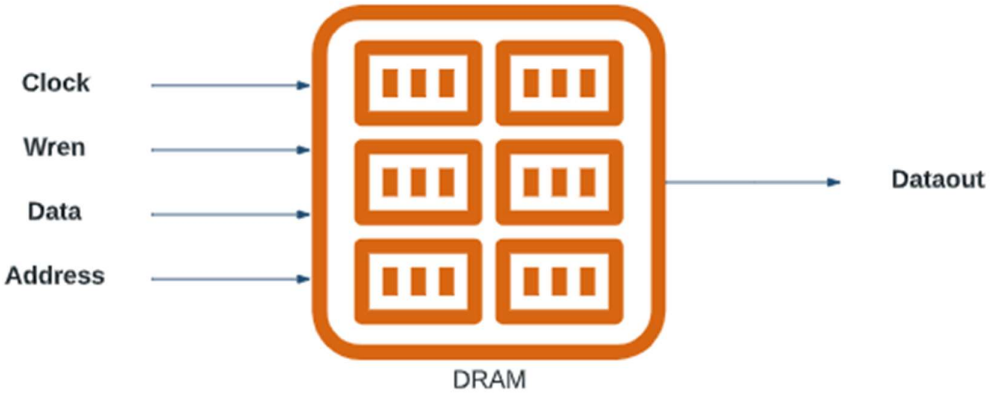


Figure 4.10 – Block Diagram of Data Memory (DRAM)

4.1.8 IRAM (Instruction RAM)

IRAM is specially used in storing instructions in the memory. Here, all instructions coded in the assembly language are stored. Whenever processor calls for the instructions, instructions are fetched at the Instruction RAM. This is implemented as a ROM where data can be only read where no permission to write as they are embedded as a read only memory in the processor. All the instructions are saved in this module in order to coded assembly level instructions. Module is basically consisted of 256 memory locations with a word size of 8 bits. All the assembly level codes of the algorithm for the basic task of the processor of filtering and down sampling the image is there. The instructions which need to be executed are stored in the IRAM memory locations and execution is done as one after other. When considering the inputs of this module,

- **clock** – Use in synchronization.
- **address** – Use in specifying the address to store the instructions. This is a 16-bit address.

The one and only output comes out of the instruction memory is,

- **q** – Output the instructions when called by the processing cycle.

In this processor design task, we have 90 instructions. So, instruction memory is designed with 8 bits. The block diagram for the instruction memory is shown as of below.

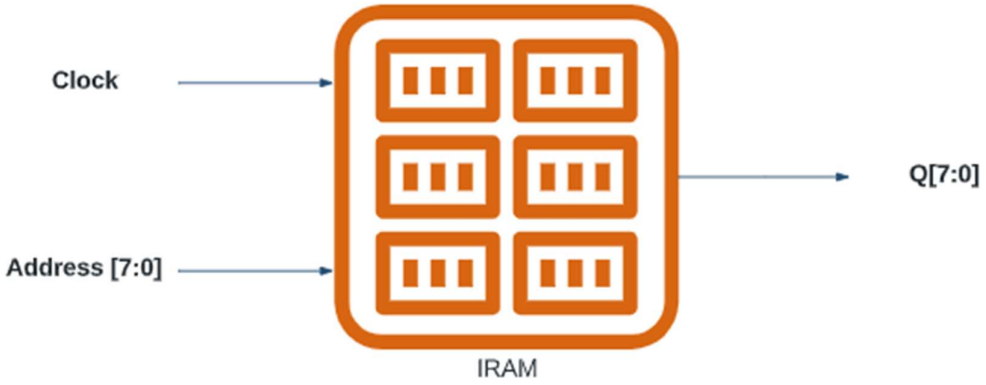


Figure 4.11 – Block Diagram of Instruction memory (IRAM)

4.1.9 Processor – Top Module

This is the module that combines all the sub modules present in the processor design which shows the outlook of the processor by showing only the inputs and outputs only to the outwards. The only input for this module is,

- **Main Clock –** Providing a clock pulse for synchronization.

The outputs that come out of the processor module are,

- **Current_address [15:0]**
- **Reg_AC [15:0]**
- **Reg_1 [15:0]**
- **Reg_2 [15:0]**
- **Instruction_address [7:0]**
- **Current_instruction [7:0]**
- **Output_from_RAM [7:0]**
- **CMD [5:0]**
- **Process_done**

So, the block diagram for the processor can be shown as below.

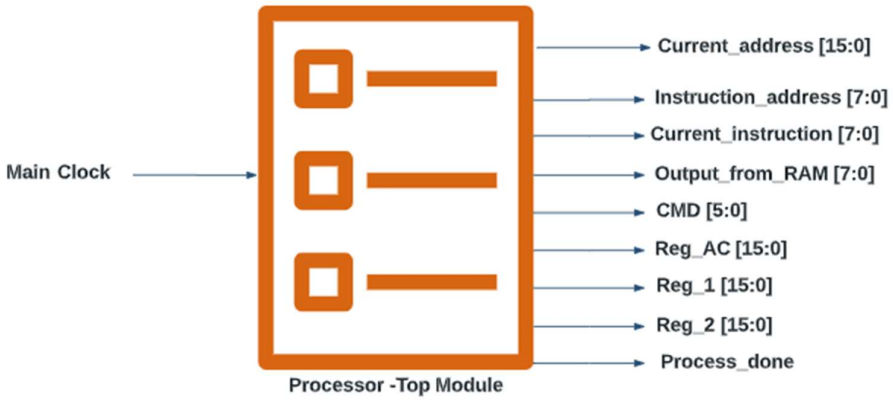


Figure 4.12 – Block Diagram of the Processor (Top Module)

5 RTL Design Simulation

RTL design view of the designed processor including all the modules was resulted as in the *Figure 5.1* shown below.

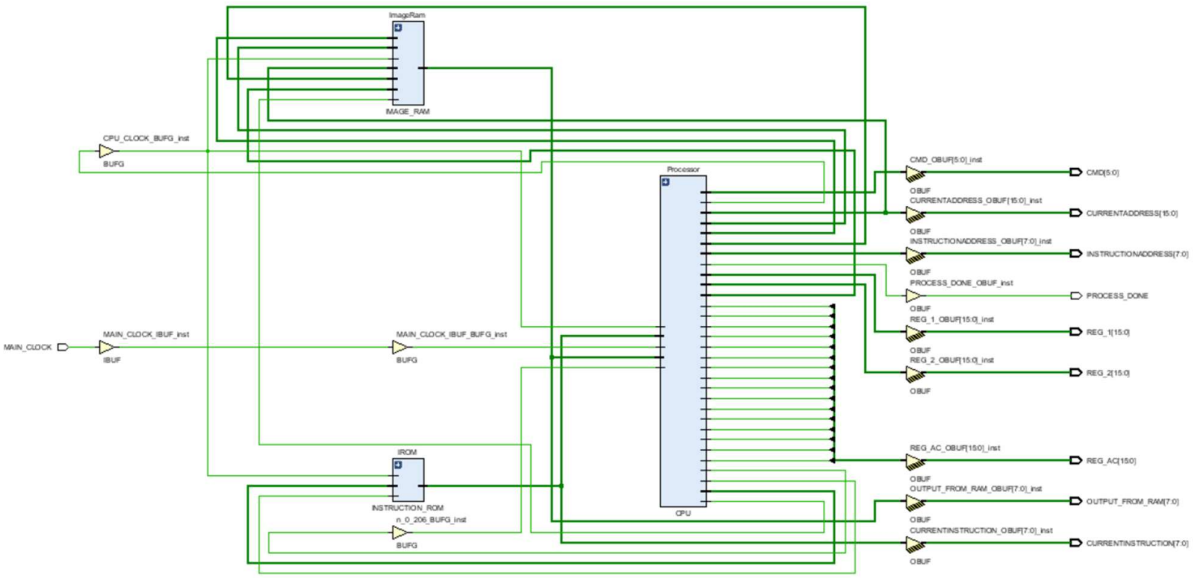


Figure 5.1 – RTL Design View of the Designed Processor

6 Performance Evaluation

6.1 Verification

The pixel data of the original image of size 256-pixel x 256-pixel is initially stored to the text file through the **Python 3.9** implementation with OpenCV using **Jupyter Notebook 6.** The python implementation procedure can be found under **Appendix 7.1.1.** Afterwards, this input.txt file is used as an input for our simulated processor and output pixel data of the image after down sampling process followed the filtering process is outputted from the designed processor. Then from this output.txt file, the image is displayed using the pixel data through python implementations and checked the success of the filtering and down sampling process.

Simultaneously, the original image of 256 x 256 pixel is filtered and down sampled using the OpenCV functions with Python and the down sampled image of 128 x 128 pixel is taken. The two images retrieved from the designed processor and the python implementation are compared using a python demonstration in means of Square Mean Error (SME). The required Python 3.9 implementations for these procedures are found under **Appendix 7.1.2**.

6.2 Results

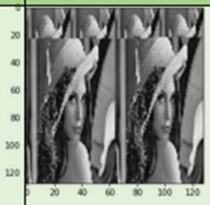
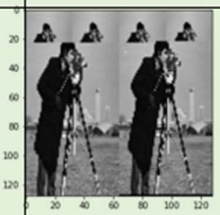
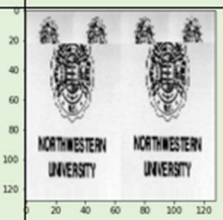
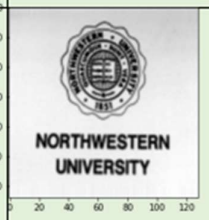
	A	B	C	D
1	Original Image used as Input	Down Sampled Image - Processor	Down Sampled Image - Python with OpenCV	Mean Square Error
2				5454.004517
3				6944.982422
4				2512.528809

Figure 5.2 – Output Comparisons through Processor implementation and Python Implementation.

6.3 Discussion

According to the above results obtained through comparing the two output images obtained through Python 3.9 implementation and Processor Design simulation, we experienced an abnormal Mean Square Error (MSE). We realized that it was due to an error resulting due to inability of stopping the loop of down sampling algorithm. Hence, new entry for the pixel values gets replaced the existing values till the

stimulation time. That is the main cause that has made the Mean Square Error to a greater extent. We experienced that the image is somewhat close to the expected output by appearance.

7 Appendix

7.1 Python Implementation – Python 3.9 with OpenCV (Jupyter Notebook)

7.1.1 Input text File Formation

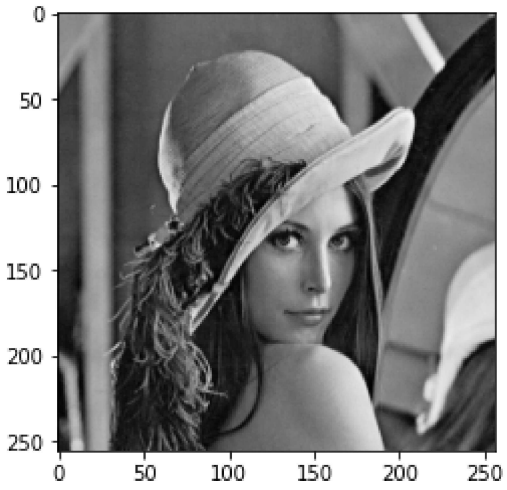
7.1.2 Output down sampled Image Formation and Comparison of Results

In [1]:
```
import cv2 as cv
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

In [2]:
```
def dec_to_binary(d):
    bnr=np.binary_repr(d,8)
    return bnr
```

In [3]:
```
image=cv.imread("lena.tif",cv.IMREAD_GRAYSCALE)
plt.imshow(image,cmap='gray')
print(image)
```

```
[[137 136 133 ... 145 148 114]
 [137 136 133 ... 145 148 114]
 [138 133 134 ... 133 125  87]
 ...
 [ 28  28  29 ...  53  62  59]
 [ 20  25  26 ...  64  69  65]
 [ 22  30  25 ...  71  68  72]]
```



In [4]:
```
f = open("lena.txt", "a")
for row in image:
    for i in row:
        f.write(dec_to_binary(i)+"\n")
f.close()
```

In [ ]:
```
import cv2 as cv
import matplotlib.pyplot as plt
import numpy as np
```

In [ ]:
```
pro_image=[]
f = open("C:\\Users\\hirun\\CSD 3\\Output.txt", "r")
for i in range(16385):
    k=f.readline()
    if(i==0):
        continue

    #print(k)
    pro_image.append(int(k,2))

pro_image=np.array(pro_image)
pro_image=pro_image.reshape(128,128)
print(pro_image)
```

In [ ]:
```
plt.imshow(pro_image,cmap='gray')
```

In [ ]:
```
def mse(imageA, imageB):
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return err
```

In [ ]:
```
img=cv.imread('lena.tif',cv.IMREAD_ANYCOLOR)
[width,hight]=img.shape
print(img)
plt.imshow(img,cmap='gray')
plt.show()
```

In [ ]:
```
mem = img.reshape(-1,1)
```

In [ ]:
```
AC = 0 # 16 bit
Z  = 1 # 1 bit
R  = 0 # 16 bit - store calculated data for filtering
R0 = 0 # 16 bit - read address for filtering
R1 = 0 # 16 bit - limit of loops for filtering - read address for sampling
R2 = 0 # 16 bit - write address for sampling
R3 = 0 # 16 bit - no of rows in down sampled image
R4 = 0 # 16 bit - filtering address - no of columns in down sampled image
AC,Z = 0,1
R1 = 65022 # (256*(256-2))-2
AC,Z = 0,1
R4 = 257 # first pixel to be filtered
while True:
    #taking values from middle row of the kernal
    R = 0
    R0 = R4
    AC = mem[R0][0]
    R = AC*16
    R0 += 1
    AC = mem[R0][0]
    R += (AC*3)
    R0 -= 2
    AC = mem[R0][0]
    R += (AC*3)
```

```
    #taking values from lower row of the kernal
    R0 = R4 + 256
    AC = mem[R0][0]
    R += (AC*3)
    R0 += 1
    AC = mem[R0][0]
    R += AC
    R0 -= 2
    AC = mem[R0][0]
    R += AC
    #taking values from upper row of the kernal
    R0 = R4 - 256
    AC = mem[R0][0]
    R += (AC*3)
    R0 += 1
    AC = mem[R0][0]
    R += AC
    R0 -= 2
    AC = mem[R0][0]
    R += AC
    R = R/32
    mem[R4][0] = R
    R4 += 1
    R1 -= 1
    if R1 == 0:
        Z = 0
    if Z == 0:
        break
filtered_image = mem.reshape(256,256)
plt.imshow(filtered_image,cmap='gray')
plt.show()
print(filtered_image)
```

In [ ]:

```
R3 = 128
R1 = 0
R2 = 0
while True:
    R4 = 128
    while True:
        AC,Z = 0,1
        AC = mem[R1][0]
        mem[R2][0] = AC
        R2 += 1
        R1 += 2
        R4 -= 1
        if R4 == 0:
            Z = 0
        if Z == 0:
            break
    AC,Z = 0,1
    R1 += 256
    R4 = 128
    R3 -= 1
    if R3 == 0:
        Z = 0
    if Z == 0:
        break
dwn_smpld_image = mem[0:16384].reshape(128,128) #16384 = 128*128
plt.imshow(dwn_smpld_image,cmap='gray')
plt.show()
print(dwn_smpld_image)
```

In [ ]:

```
mse(pro_image,dwn_smpld_image)
```

7.2 Verilog Code – Vivado 2018.2

7.2.1 Processor – Top Module

```
module PROCESSOR (
   input MAIN_CLOCK,
   output [15:0] CURRENTADDRESS, REG_AC, REG_1, REG_2,
   output [7:0] INSTRUCTIONADDRESS, CURRENTINSTRUCTION,OUTPUT_FROM_RAM,
   output [5:0] CMD,
   output PROCESS_DONE,
   input wire [15:0] ex_address,
   output wire [7:0] ex_dataout
);
   wire [7:0] DATA_FROM_RAM ;
   wire RECEIVED_8_BITS, TRANSMITTED_8_BITS, PROCESS_FINISHED_FLAG,CPU_CLOCK;
   wire CPU_WRITE_EN, RAM_CLOCK, WRITE_TO_RAM,START_PROCESSING, START_TRANSMISSION;
   wire [7:0] CPU_DATA, RAM_DATA, INSTRUCTION_ADDRESS,INSTRUCTION_DATA;
   wire [15:0] CPU_ADDRESS, RAM_ADDRESS;
   wire T,Z_Flag;
   assign CURRENTADDRESS = CPU_ADDRESS;
   assign INSTRUCTIONADDRESS = INSTRUCTION_ADDRESS;
   assign CURRENTINSTRUCTION = INSTRUCTION_DATA;
   assign OUTPUT_FROM_RAM = DATA_FROM_RAM;
   assign PROCESS_DONE = ~PROCESS_FINISHED_FLAG;
CPU Processor(
   .MAIN_CLOCK(MAIN_CLOCK),
   .PROCESS_FINISHED(PROCESS_FINISHED_FLAG),
   .DATA_FROM_RAM(DATA_FROM_RAM),
   .INSTRUCTION(INSTRUCTION_DATA),
   .CPU_CLOCK(CPU_CLOCK),
   .CPU_WRITE_EN(CPU_WRITE_EN),
   .CPU_ADDRESS(CPU_ADDRESS),
   .CPU_DATA(CPU_DATA),
   .REG_AC(REG_AC),
   .REG_1(REG_1),
```

```
   .REG_2(REG_2),
   .CMD(CMD),
   .Z_Flag(Z_Flag),
   .INSTRUCTION_ADDRESS(INSTRUCTION_ADDRESS)
);
INSTRUCTION_ROM IROM (
   .clock(CPU_CLOCK),
   .address(INSTRUCTION_ADDRESS),
   .q(INSTRUCTION_DATA)
);
IMAGE_RAM ImageRam (
   .address(CPU_ADDRESS),
   .clock(CPU_CLOCK),
   .data(CPU_DATA),
   .wren(CPU_WRITE_EN),
   .ex_address(ex_address),
   .ex_dataout(ex_dataout),
   .dataout(DATA_FROM_RAM)
);
endmodule
```

7.2.2 CPU

```
module CPU
(
   input MAIN_CLOCK,
   input [7:0] DATA_FROM_RAM, INSTRUCTION,
   output wire status,
   output PROCESS_FINISHED, CPU_CLOCK,
   output CPU_WRITE_EN, Z_Flag,
   output [15:0] CPU_ADDRESS, REG_AC, REG_1, REG_2,
   output [7:0] CPU_DATA, INSTRUCTION_ADDRESS,
   output [5:0] CMD
);
   wire INTERNAL_CLOCK, FLAG_Z, FETCH;
```

```
  wire [2:0] ALU_OP;
  wire [3:0] REG_IN_B_BUS;
  wire [7:0] IR;
  wire [15:0] B_BUS, PC, R1, R2, TR, R, AC, ALU_OUT;
  wire [12:0] SELECTORS;
  wire SIGNAL_TO_FINISH_PROCESS;

  assign CPU_CLOCK = INTERNAL_CLOCK;
  assign CPU_WRITE_EN = SELECTORS[0];
  assign CPU_DATA = B_BUS[7:0];
  assign INSTRUCTION_ADDRESS = PC[7:0];
  assign PROCESS_FINISHED = SIGNAL_TO_FINISH_PROCESS;
  assign REG_AC = AC;
  assign REG_1 = R1;
  assign REG_2 = R2;

  CPU_CLOCK_GENERATOR Clock (
     .MAIN_CLOCK(MAIN_CLOCK),
     .PROCESS_FINISHED(SIGNAL_TO_FINISH_PROCESS),
     .TICK(INTERNAL_CLOCK)
  );
  MUX_FOR_BUS_B Muxer (
     .SELECT(REG_IN_B_BUS),
     .PC(PC),
     .R1(R1),
     .R2(R2),
     .TR(TR),
     .R(R),
     .AC(AC),
     .AR(CPU_ADDRESS),
     .INSTRUCTIONS(INSTRUCTION),
     .DATA_FROM_RAM(DATA_FROM_RAM),
     .BUS(B_BUS)
  );
  CONTROL_UNIT CUnit (
```

```
    .CLOCK(INTERNAL_CLOCK),
    .FLAG_Z(FLAG_Z),
    .INSTRUCTION(IR),
    .FETCH(FETCH),
    .FINISH(SIGNAL_TO_FINISH_PROCESS),
    .REG_IN_B_BUS(REG_IN_B_BUS),
    .ALU_OP(ALU_OP),
    .CMD(CMD),
    .SELECTORS(SELECTORS),
    .status(status)
  );
  REGISTER #(8) INSTRUCTION_REGISTER (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(FETCH),
    .INCREMENT(1'b0),
    .DATA(B_BUS[7:0]),
    .OUT(IR)
  );
  REGISTER #(16) REGISTER_AR (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(SELECTORS[7]),
    .INCREMENT(SELECTORS[8]),
    .DATA(B_BUS),
    .OUT(CPU_ADDRESS)
  );
  REGISTER #(16) GENERAL_REGISTER (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(SELECTORS[2]),
    .INCREMENT(1'b0),
    .DATA(B_BUS),
    .OUT(R)
  );
  REGISTER #(16) REGISTER_01 (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(SELECTORS[5]),
```

```
    .INCREMENT(SELECTORS[11]),
    .DATA(B_BUS),
    .OUT(R1)
  );
  REGISTER #(16) REGISTER_02 (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(SELECTORS[4]),
    .INCREMENT(SELECTORS[10]),
    .DATA(B_BUS),
    .OUT(R2)
  );
  REGISTER #(16) REGISTER_TR (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(SELECTORS[3]),
    .INCREMENT(SELECTORS[9]),
    .DATA(B_BUS),
    .OUT(TR)
  );
  REGISTER #(16) REGISTER_AC (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(SELECTORS[1]),
    .INCREMENT(1'b0),
    .DATA(ALU_OUT),
    .OUT(AC)
  );
  REGISTER #(16) PROGRAM_COUNTER (
    .CLOCK(INTERNAL_CLOCK),
    .LOAD(SELECTORS[6]),
    .INCREMENT(SELECTORS[12]),
    .DATA(B_BUS),
    .OUT(PC)
  );
  ALU ALUnit (
    .A_bus(AC),
    .B_bus(B_BUS),
```

```
    .ALU_OP(ALU_OP),
    .C_bus(ALU_OUT),
    .FLAG_Z(FLAG_Z)
  );
endmodule
```

7.2.3 Control Unit (CU)

```
module CONTROL_UNIT
(
  input CLOCK, FLAG_Z,
  input [7:0] INSTRUCTION,
  output reg FETCH, FINISH = 0,
  output [5:0] CMD,
  output reg [3:0] REG_IN_B_BUS,
  output reg [2:0] ALU_OP,
  output reg [12:0] SELECTORS,
  output reg status
);
  localparam
  FETCH1 = 6'd0, FETCH2 = 6'd1, FETCH3 = 6'd2, FETCH4 = 6'd3, CLAC = 6'd6, LDAC = 6'd7, STAC = 6'd8,
  MVACR = 6'd10, MVACR1 = 6'd11, MVACR2 = 6'd12, MVACTR = 6'd13, MVACAR = 6'd14,
  MVR = 6'd15, MVR1 = 6'd16, MVR2 = 6'd17, MVTR = 6'd18, MVAR = 6'd33,
  INCAR = 6'd19, INCR1 = 6'd20, INCR2 = 6'd21, JPNZ = 6'd22, JPNZY = 6'd23, JPNZY1 = 6'd34,
  JPNZN = 6'd24, JPNZN1 = 6'd25, JPNZN2 = 6'd26, ADD = 6'd27,  SUB = 6'd28, MUL4 = 6'd29, DIV2 = 6'd30,
  ADDM = 6'd31, ADDM1 = 6'd35, NOP = 6'd5, END = 6'd32;
  localparam
  DATA_FROM_RAM = 4'd0, PC = 4'd1, R1 = 4'd2, R2 = 4'd3, TR = 4'd4, R = 4'd5, AC = 4'd6,
  INSTRUCTIONS = 4'd7, AR = 4'd8;
  localparam
  ADDAB = 3'd0, SUBAB = 3'd1, PASS = 3'd2, ZER = 3'd3, MUL4A = 3'd5, DIV2A = 3'd6;
  reg [5:0] CONTROL_COMMAND;
  reg [5:0] NEXT_COMMAND = FETCH1;
  always @ (negedge CLOCK) CONTROL_COMMAND <= NEXT_COMMAND;
```

```
always @ (CONTROL_COMMAND or FLAG_Z or INSTRUCTION)
  case(CONTROL_COMMAND)
    FETCH1:
      begin
        FETCH <= 0;
        FINISH <= 0;
        REG_IN_B_BUS <= INSTRUCTIONS;
        ALU_OP <= PASS;
        SELECTORS <= 13'b0_0000_0000_0000;
        NEXT_COMMAND <= FETCH2;
      end
    FETCH2:
      begin
        FETCH <= 1;
        FINISH <= 0;
        REG_IN_B_BUS <= INSTRUCTIONS;
        ALU_OP <= PASS;
        SELECTORS <= 13'b0_0000_0000_0000;
        NEXT_COMMAND <= FETCH3;
      end
    FETCH3:
      begin
        FETCH <= 0;
        FINISH <= 0;
        REG_IN_B_BUS <= DATA_FROM_RAM;
        ALU_OP <= PASS;
        SELECTORS <= 13'b1_0000_0000_0000;
        NEXT_COMMAND <= FETCH4;
      end
    FETCH4:
      begin
        FETCH <= 0;
        FINISH <= 0;
        REG_IN_B_BUS <= DATA_FROM_RAM;
        ALU_OP <= PASS;
```

```
            SELECTORS <= 13'b0_0000_0000_0000;
            NEXT_COMMAND <= INSTRUCTION[5:0];
        end
    CLAC:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= ZER;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
        end
    LDAC:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= PASS;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
        end
    STAC:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= AC;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_0000_0001;
            NEXT_COMMAND <= FETCH1;
        end
    MVACR:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <=3'd6;
```

```
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_0000_0100;
            NEXT_COMMAND <= FETCH1;
          end
        MVACR1:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= AC;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_0010_0000;
            NEXT_COMMAND <= FETCH1;
          end
        MVACR2:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= AC;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_0001_0000;
            NEXT_COMMAND <= FETCH1;
          end
        MVACTR:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= AC;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_0000_1000;
            NEXT_COMMAND <= FETCH1;
          end
        MVACAR:
          begin
            FETCH <= 0;
            FINISH <= 0;
```

```
            REG_IN_B_BUS <= AC;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_1000_0000;
            NEXT_COMMAND <= FETCH1;
        end
    MVR:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= R;
            ALU_OP <= PASS;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
        end
    MVR1:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= R1;
            ALU_OP <= PASS;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
        end
    MVR2:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= R2;
            ALU_OP <= PASS;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
        end
    MVTR:
        begin
            FETCH <= 0;
```

```
            FINISH <= 0;
            REG_IN_B_BUS <= TR;
            ALU_OP <= PASS;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
          end
        MVAR:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= AR;
            ALU_OP <= PASS;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
          end
        INCAR:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0001_0000_0000;
            NEXT_COMMAND <= FETCH1;
          end
        INCR1:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_1000_0000_0000;
            NEXT_COMMAND <= FETCH1;
          end
        INCR2:
          begin
```

```
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0100_0000_0000;
            NEXT_COMMAND <= FETCH1;
          end
        ADD:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= R;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
          end
        SUB:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= R;
            ALU_OP <= SUBAB;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
          end
        MUL4:
          begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= MUL4A;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
          end
        DIV2:
```

```
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= DIV2A;
            SELECTORS <= 13'b0_0000_0000_0010;
            NEXT_COMMAND <= FETCH1;
        end
    ADDM:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= INSTRUCTIONS;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b1_0000_0000_0000;
            NEXT_COMMAND <= ADDM1;
        end
    ADDM1:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= INSTRUCTIONS;
            ALU_OP <= ADDAB;
            SELECTORS <= 13'b0_0000_0000_0010.
            NEXT_COMMAND <= FETCH1;
        end
    JPNZ:
        begin
            FETCH <= 0;
            FINISH <= 0;
            REG_IN_B_BUS <= DATA_FROM_RAM;
            ALU_OP <= PASS;
            SELECTORS <= 13'b0_0000_0000_0000;
            NEXT_COMMAND <= (FLAG_Z) ? JPNZY : JPNZN;
        end
```

```
JPNZY:
  begin
    FETCH <= 0;
    FINISH <= 0;
    REG_IN_B_BUS <= DATA_FROM_RAM;
    ALU_OP <= PASS;
    SELECTORS <= 13'b1_0000_0000_0000;
    NEXT_COMMAND <= JPNZY1;
  end
JPNZY1:
  begin
    FETCH <= 0;
    FINISH <= 0;
    REG_IN_B_BUS <= DATA_FROM_RAM;
    ALU_OP <= PASS;
    SELECTORS <= 13'b0_0000_0000_0000;
    NEXT_COMMAND <= FETCH1;
  end
JPNZN:
  begin
    FETCH <= 0;
    FINISH <= 0;
    REG_IN_B_BUS <= INSTRUCTIONS;
    ALU_OP <= ADDAB;
    SELECTORS <= 13'b0_0000_0000_0000;
    NEXT_COMMAND <= JPNZN1;
  end
JPNZN1:
  begin
    FETCH <= 0;
    FINISH <= 0;
    REG_IN_B_BUS <= INSTRUCTIONS;
    ALU_OP <= PASS;
    SELECTORS <= 13'b0_0000_0100_0000;
    NEXT_COMMAND <= JPNZN2;
```

```
            end
        JPNZN2:
            begin
                FETCH <= 0;
                FINISH <= 0;
                REG_IN_B_BUS <= AC;
                ALU_OP <= ADDAB;
                SELECTORS <= 13'b0_0000_0000_0010;
                NEXT_COMMAND <= FETCH1;
            end
        NOP:
            begin
                FETCH <= 0;
                FINISH <= 0;
                REG_IN_B_BUS <= DATA_FROM_RAM;
                ALU_OP <= ADDAB;
                SELECTORS <= 13'b0_0000_0000_0000;
                NEXT_COMMAND <= FETCH1;
            end
        END:
            begin
                FETCH <= 0;
                FINISH <= 1;
                REG_IN_B_BUS <= DATA_FROM_RAM;
                ALU_OP <= ADDAB;
                SELECTORS <= 13'b0_0000_0000_0000;
                NEXT_COMMAND <= END;
                status <= 1'b1;
            end
    endcase
  assign CMD = CONTROL_COMMAND;
endmodule
```

7.2.4 Arithmetic and Logic Unit (ALU)

```
module ALU
(
   input [15:0] A_bus, B_bus,
   input [2:0] ALU_OP,
   output reg [15:0] C_bus,
   output reg FLAG_Z
);
parameter ADD = 3'd0, SUB = 3'd1, PASS = 3'd2, ZER = 3'd3, MUL4 = 3'd5, DIV2 =3'd6;
always @ (ALU_OP or A_bus or B_bus)
   begin
      case (ALU_OP)
         ADD:
           begin
            C_bus = A_bus + B_bus;
            FLAG_Z = 0;
           end
         SUB:
           begin
            C_bus = A_bus - B_bus;
            FLAG_Z = (C_bus == 16'd0) ? 1'b1 : 1'b0;
           end
         PASS:
            begin
             C_bus = B_bus;
             if (C_bus == 16'd0) FLAG_Z = 1;
            end
         ZER:
            begin
               C_bus = 0;
               FLAG_Z = 0;
            end
         MUL4:
            begin
               C_bus = A_bus << 2;
               FLAG_Z = 0;
```

```
            end
        DIV2:
            begin
                C_bus = A_bus >> 1;
                FLAG_Z = 0;
            end
    endcase
  end
endmodule
```

7.2.5 Multiplexer

```
module MUX_FOR_BUS_B
(
  input [3:0] SELECT,
  input [15:0] PC, R1, R2, TR, R, AC, AR,
  input [7:0] INSTRUCTIONS, DATA_FROM_RAM,
  output reg [15:0] BUS
);
  always @ (*) //SELECT or DATA_FROM_RAM or PC or R1 or R2 or TR or R or AC or INSTRUCTIONS or AR
  begin
    case(SELECT)
      4'd0: BUS = {8'b0000_0000, DATA_FROM_RAM};
      4'd1: BUS = PC;
      4'd2: BUS = R1;
      4'd3: BUS = R2;
      4'd4: BUS = TR;
      4'd5: BUS = R;
      4'd6: BUS = AC;
      4'd7: BUS = {8'b0000_0000, INSTRUCTIONS};
      4'd8: BUS = AR;
    endcase
  end
endmodule
```

7.2.6 CPU Clock Generator

```
module CPU_CLOCK_GENERATOR
(
   input MAIN_CLOCK, PROCESS_FINISHED,
   output reg TICK = 1'b1
);
   always @ (posedge MAIN_CLOCK)
      begin
         if (~PROCESS_FINISHED)
            TICK = ~TICK;
         else
            TICK = 1'b0;
      end
endmodule
```

7.2.7 Data Memory (DRAM)

```
module IMAGE_RAM (
   address,
            clock,
            data,
            wren,
            dataout,
            ex_address,
   ex_dataout
   );
            input    [15:0]  address;
            input      clock;
            input    [7:0]  data;
            input      wren=1;
            input [15:0] ex_address;
            output reg [7:0] ex_dataout;
```

```
        output reg [7:0]  dataout;
        reg[7:0] MEMORY [65535:0];
  initial begin
    $readmemb("C:\\Users\\hiruna\\CSD \\lena.txt",MEMORY);
  end
  always @ (ex_address)
  begin
    ex_dataout <= MEMORY[ex_address];
  end
  always @ (posedge clock)
  begin
    if(wren == 1)begin
      MEMORY[address] <= data[7:0];
    end
    else begin
      dataout <= MEMORY[address];
    end
  end
endmodule
```

7.2.8 Instruction Memory (IRAM)

```
module INSTRUCTION_ROM(
  input clock,
  input [7:0] address,
  output reg [7:0] q
);
  reg[7:0] memory[0:255];
always @(posedge clock)
        q<=memory[address];
  initial begin
    memory[0]=8'b0000_0110;
    memory[1]=8'b0000_1011;
    memory[2]=8'b0000_1100;
```

```
memory[3]=8'b0001_0001;
memory[4]=8'b0001_1101;
memory[5]=8'b0001_1101;
memory[6]=8'b0001_1101;
memory[7]=8'b0001_1101;
memory[8]=8'b0000_1010;
memory[9]=8'b0001_0000;
memory[10]=8'b0001_1011;
memory[11]=8'b0000_1110;
memory[12]=8'b0000_0111;
memory[13]=8'b0000_1010;
memory[14]=8'b0001_0011;
memory[15]=8'b0000_0111;
memory[16]=8'b0001_1101;
memory[17]=8'b0001_1110;
memory[18]=8'b0001_1011;
memory[19]=8'b0000_1010;
memory[20]=8'b0001_0011;
memory[21]=8'b0000_0111;
memory[22]=8'b0001_1011;
memory[23]=8'b0000_1010;
memory[24]=8'b0010_0001;
memory[25]=8'b0001_1111;
memory[26]=8'b1111_1110;
memory[27]=8'b0000_1110;
memory[28]=8'b0000_0111;
memory[29]=8'b0001_1101;
memory[30]=8'b0001_1110;
memory[31]=8'b0001_1011;
memory[32]=8'b0000_1010;
memory[33]=8'b0001_0011;
memory[34]=8'b0010_0001;
memory[35]=8'b0000_1101;
memory[36]=8'b0000_0111;
memory[37]=8'b0001_1101;
```

```
memory[38]=8'b0001_1011;
memory[39]=8'b0000_1010;
memory[40]=8'b0001_0011;
memory[41]=8'b0000_0111;
memory[42]=8'b0001_1101;
memory[43]=8'b0001_1110;
memory[44]=8'b0001_1011;
memory[45]=8'b0000_1010;
memory[46]=8'b0010_0001;
memory[47]=8'b0001_1111;
memory[48]=8'b1111_1110;
memory[49]=8'b0000_1110;
memory[50]=8'b0000_0111;
memory[51]=8'b0001_1011;
memory[52]=8'b0000_1010;
memory[53]=8'b0001_0011;
memory[54]=8'b0000_0111;
memory[55]=8'b0001_1101;
memory[56]=8'b0001_1110;
memory[57]=8'b0001_1011;
memory[58]=8'b0000_1010;
memory[59]=8'b0001_0011;
memory[60]=8'b0000_0111;
memory[61]=8'b0001_1011;
memory[62]=8'b0001_1110;
memory[63]=8'b0001_1110;
memory[64]=8'b0001_1110;
memory[65]=8'b0001_1110;
memory[66]=8'b0000_1010;
memory[67]=8'b0001_0010;
memory[68]=8'b0000_1110;
memory[69]=8'b0000_1111;
memory[70]=8'b0000_1000;
memory[71]=8'b0001_0100;
memory[72]=8'b0001_0000;
```

```
memory[73]=8'b0000_1010;
memory[74]=8'b0000_0110;
memory[75]=8'b0001_1111;
memory[76]=8'b1111_1110;
memory[77]=8'b0001_1100;
memory[78]=8'b0001_0110;
memory[79]=8'b0000_0011;
memory[80]=8'b0000_1011;
memory[81]=8'b0001_0100;
memory[82]=8'b0001_0001;
memory[83]=8'b0000_1010;
memory[84]=8'b0000_0110;
memory[85]=8'b0001_1111;
memory[86]=8'b1111_1110;
memory[87]=8'b0001_1100;
memory[88]=8'b0001_0110;
memory[89]=8'b0000_0011;
memory[90]=8'b0000_0110;
memory[91]=8'b0000_1011;
memory[92]=8'b0000_1100;
memory[93]=8'b0001_0001;
memory[94]=8'b0001_1101;
memory[95]=8'b0001_1101;
memory[96]=8'b0001_1101;
memory[97]=8'b0001_1101;
memory[98]=8'b0000_1010;
memory[99]=8'b0001_0000;
memory[100]=8'b0001_1011;
memory[101]=8'b0000_1110;
memory[102]=8'b0001_1110;
memory[103]=8'b0000_1101;
memory[104]=8'b0000_0111;
memory[105]=8'b0000_1010;
memory[106]=8'b0001_0011;
memory[107]=8'b0000_0111;
```

```
memory[108]=8'b0001_1011;
memory[109]=8'b0000_1010;
memory[110]=8'b0010_0001;
memory[111]=8'b0001_1111;
memory[112]=8'b1111_1111;
memory[113]=8'b0000_1110;
memory[114]=8'b0000_0111;
memory[115]=8'b0001_1011;
memory[116]=8'b0000_1010;
memory[117]=8'b0001_0011;
memory[118]=8'b0000_0111;
memory[119]=8'b0001_1011;
memory[120]=8'b0001_1110;
memory[121]=8'b0001_1110;
memory[122]=8'b0000_1010;
memory[123]=8'b0001_0010;
memory[124]=8'b0000_1110;
memory[125]=8'b0000_1111;
memory[126]=8'b0000_1000;
memory[127]=8'b0001_0100;
memory[128]=8'b0001_0100;
memory[129]=8'b0001_0000;
memory[130]=8'b0000_1010;
memory[131]=8'b0000_0110;
memory[132]=8'b0001_1111;
memory[133]=8'b1111_1111;
memory[134]=8'b0001_1111;
memory[135]=8'b0000_0001;
memory[136]=8'b0001_1100;
memory[137]=8'b0001_0110;
memory[138]=8'b0101_1101;
memory[139]=8'b0000_1011;
memory[140]=8'b0001_0101;
memory[141]=8'b0001_0101;
memory[142]=8'b0001_0001;
```

```
        memory[143]=8'b0000_1010;
        memory[144]=8'b0000_0110;
        memory[145]=8'b0001_1111;
        memory[146]=8'b1111_1111;
        memory[147]=8'b0001_1111;
        memory[148]=8'b0000_0001;
        memory[149]=8'b0001_1100;
        memory[150]=8'b0001_0110;
        memory[151]=8'b0101_1101;
        memory[152]=8'b0010_0000;
    end
endmodule
```

7.2.9 Processor Test bench

```
module processor_tb;
    reg MAIN_CLOCK;
    wire PROCESS_DONE;
    wire [15:0] CURRENTADDRESS, REG_AC, REG_1, REG_2;
    wire [5:0] CMD;
    wire [7:0] INSTRUCTIONADDRESS, CURRENTINSTRUCTION,OUTPUT_FROM_RAM;
    reg [15:0] ADD_FROM_OUT;
    wire [7:0] DATA_TO_OUT;
    integer file,i;
    PROCESSOR UUT (
        .MAIN_CLOCK(MAIN_CLOCK),
        .CURRENTADDRESS(CURRENTADDRESS),
        .REG_AC(REG_AC),
        .REG_1(REG_1),
        .REG_2(REG_2),
        .OUTPUT_FROM_RAM(OUTPUT_FROM_RAM),
        .INSTRUCTIONADDRESS(INSTRUCTIONADDRESS),
        .CURRENTINSTRUCTION(CURRENTINSTRUCTION),
        .CMD(CMD),
        .PROCESS_DONE(PROCESS_DONE),
```

```
      .ex_dataout(DATA_TO_OUT),
      .ex_address(ADD_FROM_OUT)
  );
  always
    #10 MAIN_CLOCK = ~MAIN_CLOCK;
  initial begin
    MAIN_CLOCK = 1'b0;
    #100000000
    file=$fopen("C:\\Users\\hiruna\\CSD\\output.txt","w");
    for (i=2; i< 65539; i=i+4)
      begin
        @(posedge MAIN_CLOCK)
        ADD_FROM_OUT = i;
        $fwrite(file,"%b",DATA_TO_OUT);
        $fwrite(file,"\n");
      end
   $fclose(file);
   $finish;
 end
endmodule
```