Department of Electronic and Telecommunication Engineering

University of Moratuwa, Sri Lanka

EN3030 - Circuits and System Design

FPGA Based Processor Design

Project Report - Group 28

Group Member	**Index**
H. W. C. Sauranga	180574K
H. D. M. Prematthilaka	180497C
H. K. C. A. Sandeepa	180564F
P. Samarasingha	180554B

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

Date of Submission: July 11, 2022

Abstract

Over the past few decades, field-programmable gate array (FPGA) has become a popular platform for digital design implementations due to its versatility. This report is based on designing a processor that would down-sample a 256 by 256 image by a factor of 2. This report includes the algorithms, proposed 'Instruction Set Architecture (ISA), the functionality of individual modules, supplementary resources used during the designing process, and a comparison of downsampled images using other techniques. `Vivado 2018.2, ModelSim 20.1` were utilized to compile and simulate the Verilog modules and `Python` was used as the software framework.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview on Processor Design

1.1.1 Central Processing Unit (CPU)

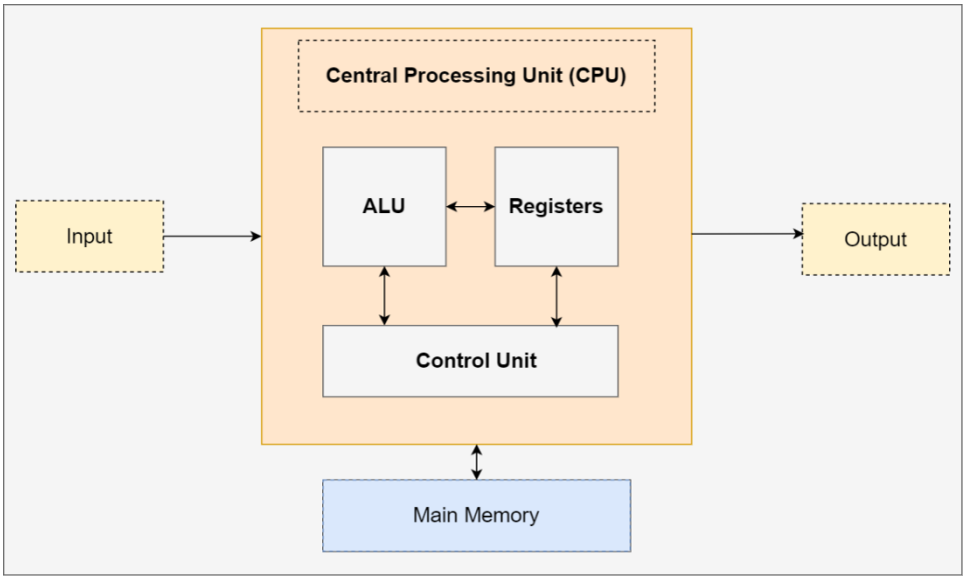

Figure 1: *Block Diagram of a CPU*

A simple overview of a CPU is represented in the above diagram. The central processing unit can be considered as the brain of the circuitry that coordinates and conducts arithmetic as well as logical operations to produce the desired outputs for a given input to the system. The CPU consists of an arithmetic and logic unit(ALU), control unit, registers, and interconnections.

1.1.2 Microprocessor

Microprocessors are Central Processing Units (CPUs) for computers that are constructed from a single Integrated Circuit (IC). Microcomputer refers to a digital computer with a single CPU-functioning microprocessor. It is a programmable, multipurpose, clock-driven, register-based electrical device that accepts binary data as input, processes the data in accordance with the instructions, and outputs the results. It reads binary instructions from a storage device called memory. Millions of small parts, including transistors, registers, and diodes, all work together in the microprocessor.

Features of Microprocessor

1. Low Cost - *Because of advances in integrated circuit technology, microprocessors are quite inexpensive. It will make computer systems less expensive*
2. High Speed - *The microprocessor's technology allows it to operate at a high rate of speed. It has a million instructions per second processing speed*
3. Small Size - *Due to very large scale and ultra large scale integration technology, a microprocessor can be manufactured with a very small footprint*
4. Versatile - *Microprocessors are adaptable because they can run multiple programs on the same chip*

5. Low Power Consumption - *Microchips are utilizing metal oxide semiconductor innovation, which consumes less power*

1.2 Problem Statement

The project requirement is to design and simulate/implement a processor that could down-sample a given **256x256** image by a factor of 2. That is, to obtain a **128x128** image from the input image. The input image needs to be sent to the processor where the image is down-sampled and once finished, the results should be sent back for display purposes.

1.2.1 Proposed Solution

The images are 2D/ 3D arrays with elements having values from 0-255. Yet, from the processor's perspective, the image can't be treated as a 2D/3D array. Thus, the image needs to be flattened into a 1D array initially. We are planning to do the conversion using Python/ MATLAB. Once flattened, the 1D array is then transmitted and saved into the RAM.

Now, before down-sampling the image, it is a must to pass it through an LPF to avoid aliasing. A common LPF is a Gaussian kernel. Thus, the array would be convolved with the Gaussian kernel, down-sampled, and will be stored in the memory where the resulting values are transmitted back to the computer to display the final image.

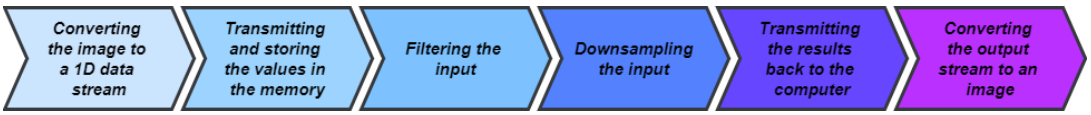

Figure 2: *Flow Diagram of Solution*

2 Algorithm

2.1 Kernel Algorithm

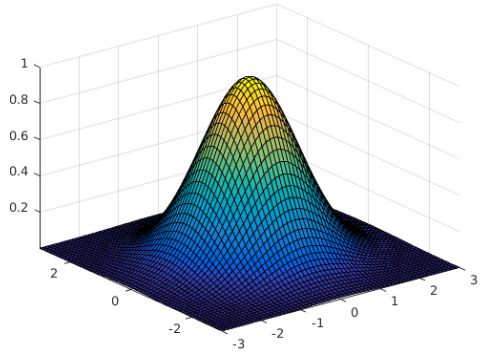

Figure 3: *A Visual Representation of a Gaussian Kernel*

In image processing, Gaussian kernels and mean kernels are used as low pass filters. But Gaussian kernels are generally preferred over mean kernels as they preserve locality by assigning weights to neighbor pixels based on the distance from the neighbor pixels to the

center pixel. The 2D Gaussian kernel equation is given as follows.

$$q(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x^2+y^2)}{2\sigma^2}\right) \tag{1}$$

As a result, the high-frequency components of the original image are suppressed, preparing the image for down-sampling. The following 3 by 3 discrete Gaussian Kernel approximation was utilized because the processor we created only supports integers and not a true Gaussian Kernel.

1	2	1
2	4	2
1	2	1

Table 1: *Gaussian Kernel*

To LPF the image, the above Gaussian kernel needs to be convolved with the image. The advantages of the above-modified filter are two folds.

1. *All the elements are powers of 2*
2. *The Sum of the kernel elements is 16 which is a power of 2*

Owing to these reasons, we don't need to store the kernel in the memory. We can simply multiply the relevant pixel values by 2 or 4 using left shift and get the average using right shift. 2D convolution (g(x,y)) of an image (I(x,y)) can be represented by the following equation.

$$(I * g)(x,y) = \sum_{-\infty}^{\infty}\sum_{-\infty}^{\infty} I(x-u, y-v)g(u,v) \tag{2}$$

2.2 Down-sampling Algorithm

Since 256x256 images are being used, it is not necessary to use interpolating algorithms to down-sample the image. To avoid unnecessary complexity, a simple down-sampling algorithm is utilized. The down-sampling algorithm is illustrated in the diagram below.

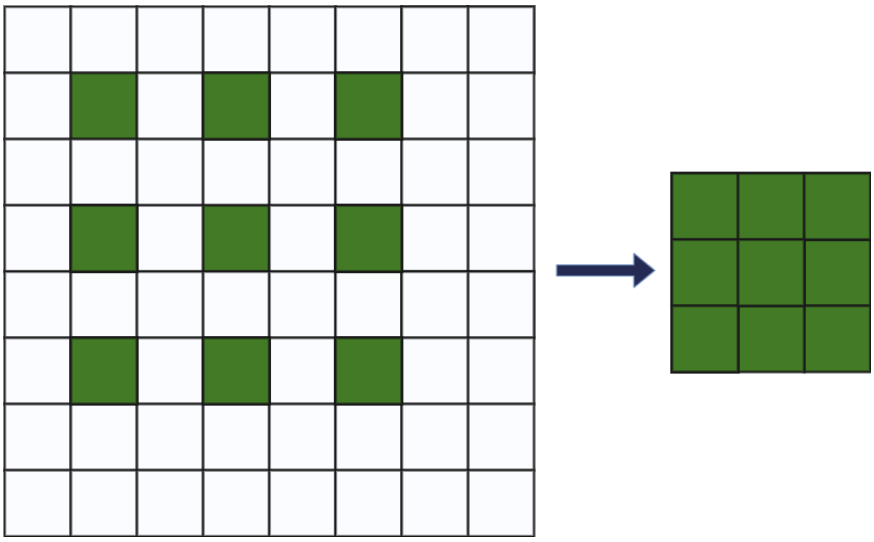

Figure 4: *A Visual Representation of Down-Sampling Algorithm*

2.3 Integrated filtering and down-sampling algorithm

As we are down-sampling the image, it is not essential to convolve all the image pixels. The time incurred for the task can be reduced if the convolution is carried out only for the pixels that would be there in the final image. This is achieved by convolving the image with the 3x3 kernel with a stride of 2. This concept is widely used in convolutional neural networks. The new task flow would be as follows.

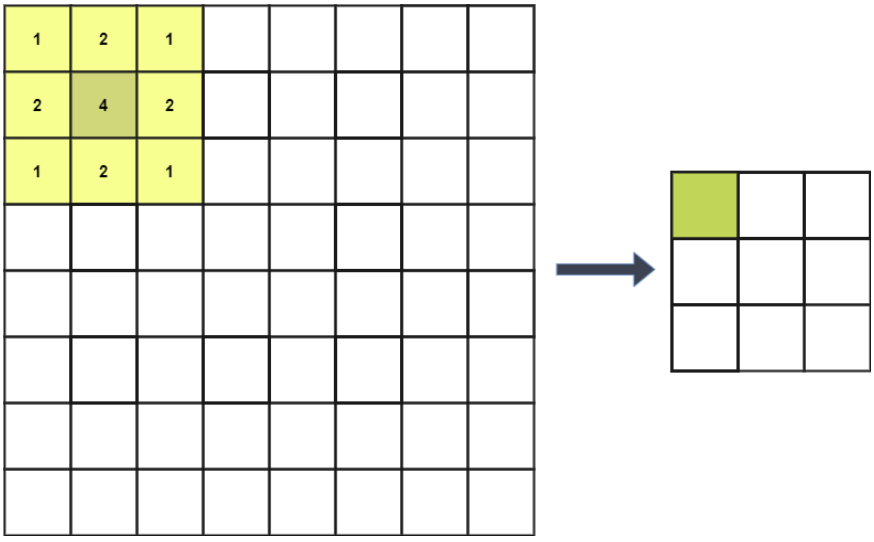

Figure 5: *A Visual Representation of Filtering + Down-Sampling Algorithm*

Let's say **L = 256 (image width)**. Let the **image reshaped to 1D array be I** and the **resulting imaging be J**. The algorithm to be followed for the combined procedure is as follows.

Algorithm 1 Filtering + Down-sampling Algorithm

procedure IMAGE DOWN-SAMPLING WITH FILTERING
- $DI \leftarrow$ empty array ▷ Empty image array to store down-sampled data
- $C1 \leftarrow L + 2$ ▷ Starting row pixel number
- $C2 \leftarrow 2$ ▷ Starting column number
- $C3 \leftarrow 1$
- $E \leftarrow L \times (L - 2)$ ▷ Last pixel of last convolving row
- **while** $C1 < E$ **do**
  - **if** $C2 == L$ **then** ▷ Checking end of the row
    - $C1 \leftarrow C1 + (L + 2)$
    - $C2 \leftarrow 2$
  - **end if**
  - ▷ Start convolution with stride = 2
  - $total \leftarrow 0$
  - $total \leftarrow total + I[C1 - L - 1] \times 1$
  - $total \leftarrow total + I[C1 - L] \times 2$
  - $total \leftarrow total + I[C1 - L + 1] \times 1$
  - $total \leftarrow total + I[C1 - 1] \times 2$
  - $total \leftarrow total + I[C1] \times 4$
  - $total \leftarrow total + I[C1 + 1] \times 2$
  - $total \leftarrow total + I[C1 + L - 1] \times 1$
  - $total \leftarrow total + I[C1 + L] \times 2$
  - $total \leftarrow total + I[C1 + L + 1] \times 1$
  - $DI[C3] \leftarrow \frac{total}{16}$
  - $C1 \leftarrow C1 + 2$
  - $C2 \leftarrow C2 + 2$
  - $C3 \leftarrow C3 + 1$
- **end while**
- $J \leftarrow$ reshaped DI ▷ Reshaped Down-sampled image array

end procedure

Since the terms **C1, C2, C3, E, L**, and **T(total)** are used frequently, we decided to allocate **General Purpose Registers** to each of them.

3 Instruction Set Architecture

The abstract model of a computer includes an Instruction Set Architecture (ISA) that specifies how the CPU is controlled by the software. The ISA serves as a conduit between the hardware and the software, defining both the tasks the processor can perform and how they are carried out. ISA generally includes,

- *Instruction set (Assembly language instructions)*
- *Registers and their uses*
- *Register sizes*
- *Information necessary to interact with memory*

The following are some of the register sizes, memory sizes, and address widths proposed to design the image down-sampling processor.

- **Data Memory** - *Data memory is used to hold the pixel values of the 256 by 256 image. As we need to store values such as* $65024(256 \times (256 - 2))$*, the width of a memory unit is 16bits. Since we need 65536 locations, the selected address width is 16bits.*
- **Instruction Memory** - *Instruction memory harbors the assembly code of the algorithm that is used to down-sample the image. This ROM block is essentially 8 by 256 (word size = 8 and depth = 256) in size.*
- **Program Counter (PC)** - *The program counter is a register that is dedicated to keeping track of the next instruction to be fetched. We have decided to use an 8-bit register for PC.*
- **Memory Buffer Register Unit (MBRU)** - *The 8-bit register loads and keeps the location of the instruction to be decoded.*
- **Memory Address Register (MAR)** - *MAR is a 16-bit register that contains the address of the memory where data is read from/written to.*
- **Memory Data Register (MDR)** - *MDR is an 8bit register that contains the data that is read from/written to the memory.*
- **L, C1, C2, C3, T, E Registers** - *These 16-bit general purpose registers are utilized in storing the intermediate values in the process.*
- **Arithmetic and Logical Unit (ALU)** - *This performs all the arithmetic and logical operations that are required to filter and down-sample the image. The inputs are provided by the A(Accumulator) and B buses, and the C bus is set aside for the ALU's output. Four control bits from the control unit are supplied into the ALU, instructing it to carry out the necessary task.*
- **Accumulator (AC)** - *The 16-bit register with a direct connection to the ALU.*
- **Control Store** - *According to the input given by the MBRU, this sends all the control signals to the CPU.*
- **A, B, C Buses** - *Data is transmitted between the Data Memory, Instruction Memory, ALU, and set of registers using these 16-bit parallel wired connections.*

3.1 Data Path

An overview of the data path that would facilitate the data/ instruction exchange between the memory, ALU, and the registers is illustrated in the following figure.

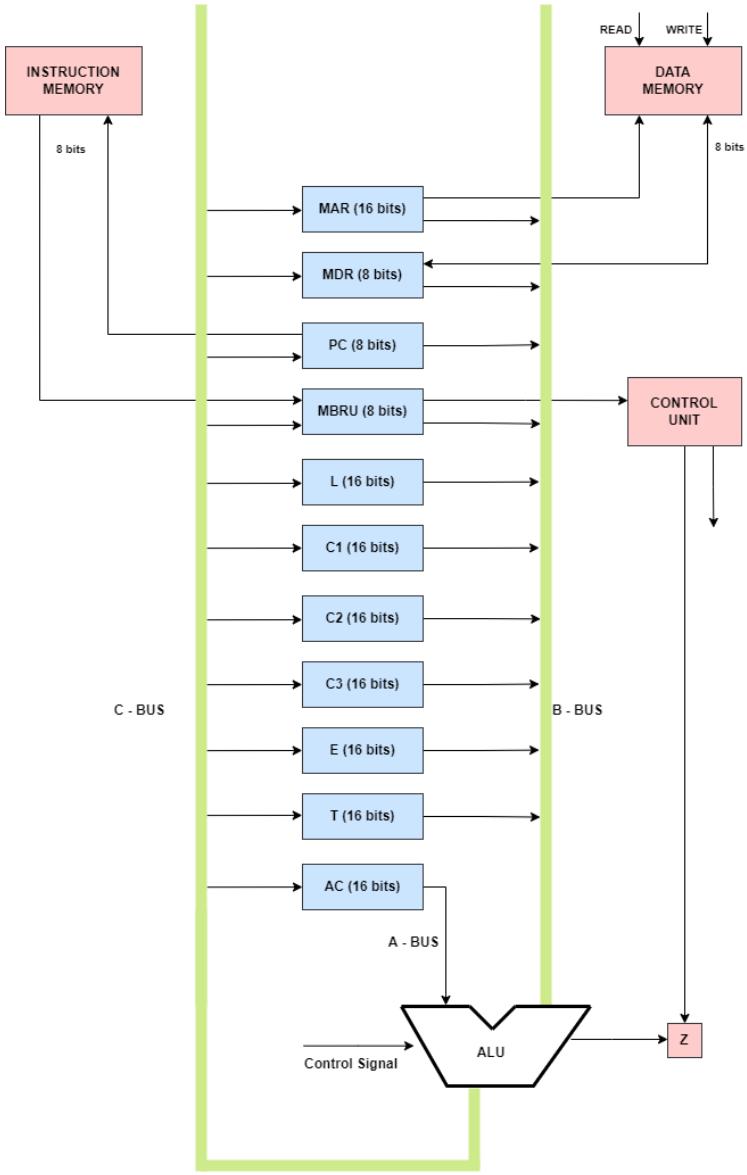

Figure 6: *Data Path of The Processor*

3.2 Instruction Set

The instructions that are included in the instruction set act as the building blocks of the assembly code. The following table (2) summarizes the set of instructions that was used when implementing the processor.

Note: The instructions and their corresponding microinstructions presented in the mid-review report were amended. This report includes the finalized instruction set

Table 2: *Instruction Set of The Downsampling Processor*

IRAM Address	Instruction	Microinstruction(s)	Description
0	FETCH	FETCH1	MBRU $\leftarrow$ IRAM[PC]; FETCH
1		FETCH2	PC $\leftarrow$ PC $+ 1$
2	NOP	NOP	No Operation
3	LDAC	LDAC1	MDR $\leftarrow$ DRAM[MAR]; READ
4		LDAC2	AC $\leftarrow$ MDR
5	STAC	STAC1	MDR $\leftarrow$ AC
6		STAC2	DRAM[MAR $\leftarrow$ MDR; WRITE
7	CLAC	CLAC	AC $\leftarrow 0$, Z $\leftarrow 1$
8	MVACMAR	MVACMAR	MAR $\leftarrow$ AC
9	MVACC1	MVACC1	C1 $\leftarrow$ AC
10	MVACC2	MVACC2	C2 $\leftarrow$ AC
11	MVACC3	MVACC3	C3 $\leftarrow$ AC
12	MVACL	MVACL	L $\leftarrow$ AC
13	MVACE	MVACE	E $\leftarrow$ AC
14	MVACT	MVACT	T $\leftarrow$ AC
15	MVC1	MVC1	AC $\leftarrow$ C1
16	MVC2	MVC2	AC $\leftarrow$ C2
17	MVC3	MVC3	AC $\leftarrow$ C3
18	MVCT	MVCT	AC $\leftarrow$ T
19	INAC	INAC	AC $\leftarrow$ AC $+ 1$
20	DEAC	DEAC	AC $\leftarrow$ AC $- 1$; IF AC=0, THEN Z=1
21	ADDT	ADDT	AC $\leftarrow$ AC $+ T$
22	ADDL	ADDL	AC $\leftarrow$ AC $+ L$
23	SUBE	SUBE	AC $\leftarrow$ AC $- E$; IF AC=0, THEN Z=1
24	SUBL	SUBL	AC $\leftarrow$ AC $- L$; IF AC=0, THEN Z=1
25	DIV	DIV	AC $\leftarrow$ AC $\gg 4$
26	MUL2	MUL2	AC $\leftarrow$ AC $\ll 1$
27	MUL4	MUL4	AC $\leftarrow$ AC $\ll 2$
28	MUL256	MUL256	AC $\leftarrow$ AC $\ll 8$
29	JUMP	JUMP1	MBRU $\leftarrow$ IRAM[PC]; FETCH
30		JUMP2	C $\leftarrow$ MBRU
31		JUMP3	PC $\leftarrow$ C
32	JMPZ	JMPZ1	
33		JMPZN1	PC $\leftarrow$ PC $+ 1$
34		JMPZY1	MBRU $\leftarrow$ IRAM[PC]; FETCH
35		JMPZY2	C $\leftarrow$ MBRU
36		JMPZY3	PC $\leftarrow$ C
37	JMNZ	JMNZ1	
38		JMNZY1	PC $\leftarrow$ PC $+ 1$
39		JMNZN1	MBRU $\leftarrow$ IRAM[PC]; FETCH
40		JMNZN2	C $\leftarrow$ MBRU
41		JMNZN3	PC $\leftarrow$ C
42	MVL	MVL	AC $\leftarrow$ L

3.3 Microinstruction Sequence

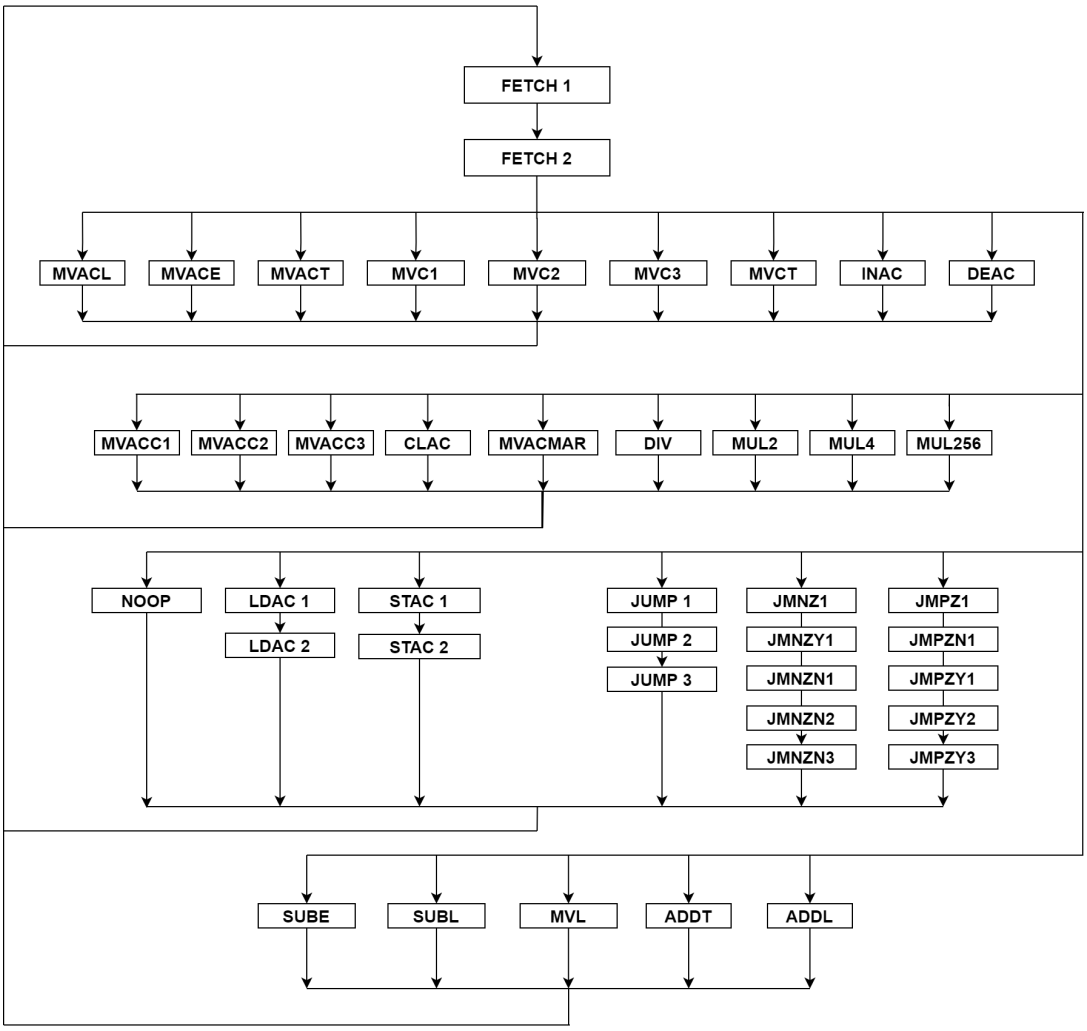

Figure 7: *Microinstruction Sequence*

3.4 Control Unit

The control unit comprises all possible combinations of control signals that govern the behaviour of other modules in the processor such as the ALU, MDR, AC, general purpose regiters, etc. Each control signal is 30-bit long and when an instruction is transmitted to the CU, the corresponding microinstruction is looked up in the control unit. The following table represents the control unit look-up-table.

Table 3: *Control Unit LUT*

Next Address	ALU	Write Enable Signal : C Bus	Memory Signal	PC	Read Enable Signal : B Bus
00000001	0000	0000000000	100	0	0000
xxxxxxxx	0000	0000000000	000	1	0000
00000010	0000	0000000000	000	0	0000
00000100	0000	0000000000	010	0	0000
00000000	0100	0000000001	000	0	0001
00000110	0011	0100000000	000	0	0000
00000000	0000	0000000000	001	0	0000
00000000	1011	0000000001	000	0	0000
00000000	0011	1000000000	000	0	0000
00000000	0011	0000100000	000	0	0000
00000000	0011	0000010000	000	0	0000
00000000	0011	0000001000	000	0	0000
00000000	0011	0001000000	000	0	0000
00000000	0011	0000000010	000	0	0000
00000000	0011	0000000100	000	0	0000
00000000	0100	0000000001	000	0	0101
00000000	0100	0000000001	000	0	0110
00000000	0100	0000000001	000	0	0111
00000000	0100	0000000001	000	0	1000
00000000	0101	0000000001	000	0	0000
00000000	0110	0000000001	000	0	0000
00000000	0001	0000000001	000	0	1000
00000000	0001	0000000001	000	0	0100
00000000	0010	0000000001	000	0	1001
00000000	0010	0000000001	000	0	0100
00000000	1010	0000000001	000	0	0000
00000000	0111	0000000001	000	0	0000
00000000	1000	0000000001	000	0	0000
00000000	1001	0000000001	000	0	0000
00011110	0000	0000000000	100	0	0000
00011111	0100	0000000000	000	0	0011
00000000	0000	0010000000	000	0	0000
xxxxxxxx	0000	0000000000	000	0	0000
00000000	0000	0000000000	000	1	0000
00100011	0000	0000000000	100	0	0000
00100100	0100	0000000000	000	0	0011
00000000	0000	0010000000	000	0	0000
xxxxxxxx	0000	0000000000	000	0	0000
00000000	0000	0000000000	000	1	0000
00101000	0000	0000000000	100	0	0000
00101001	0100	0000000000	000	0	0011
00000000	0000	0010000000	000	0	0000
00000000	0100	0000000001	000	0	0100

3.4.1 Arithmetic & Logic Unit

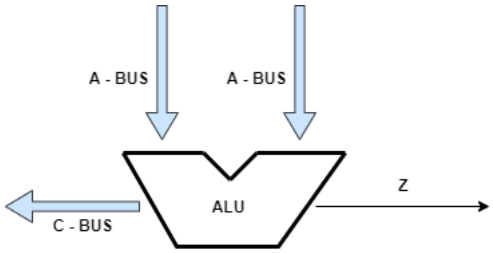

Figure 8: *ALU*

The ALU is responsible for carrying out the arithmetic operations. The operands for the arithmetic operations are fed to the ALU via the A and B buses. The result is transferred to the C bus and a Z flag is used to indicate whether the output is zero or not. The mathematical operation to be executed is decided by the 4-bit control signal from the control unit.

Table 4: *ALU Control Signals : Microinstruction[21:18]*

ALU Operation	**Description**	**Control Signal**
NONE	No ALU operation	4b'0000
ADD	$C = A + B$	4b'0001
SUB	$C = A - B$; IF $C = 0$, THEN $Z = 1$	4b'0010
PASSATOC	$C = A$	4b'0011
PASSBTOC	$C = B$	4b'0100
INCAC	$C = A + 1$	4b'0101
DECAC	$C = A - 1$; IF $C = 0$, THEN $Z = 1$	4b'0110
LSHIFT1	$C = A \ll 1$	4b'0111
LSHIFT2	$C = A \ll 2$	4b'1000
LSHIFT8	$C = A \ll 8$	4b'1001
RSHIFT4	$C = A \gg 4$	4b'1010
RESET	$C = 0$	4b'1011

3.4.2 Write Enable Signals : C Bus

The bits numbered from 8 to 17 (10 bits) of the microinstruction are allocated to indicate the registers to which the results from the ALU operation are written. The order of the relevant registers are indicated in the diagram below.

Table 5: *Write Enable Signal: Microinstruction[17:8]*

MAR	MDR	PC	L	C1	C2	C3	T	E	AC

3.4.3 Memory Signals

The memory signal component of the microinstruction corresponds to fetching, reading from the data memory and writing to the data memory.

Table 6: *Memory Signal: Microinstruction[7:5]*

FETCH	READ	WRITE

3.4.4 Read Enable Signals : B Bus

Multiple registers (i.e. MDR, PC, MBRU, and the general purpose registers) are connected to the B bus. Accordingly, it is important to ensure that only one register could release a value that is being held, into the B bus, at a given instance. Accordingly, a few bits of the microinstruction are allocated to handle the register selection.

Table 7: *Read Enable Signal : Microinstruction[3:0]*

Register	**Control Signal**
NONE	4b'0000
MDR	4b'0001
PC	4b'0010
MBRU	4b'0011
L	4b'0100
C1	4b'0101
C2	4b'0110
C3	4b'0111
T	4b'1000
E	4b'1001

3.5 Assembly Code of The Algorithm

When implementing the processor, the instructions involved in carrying out the task (instructions related to the algorithm and their order) should be stored in the instruction memory. The assembly code is generated using the instructions defined in the ISA and it's a mere representation of the algorithm steps, in assembly language. When the processor functions, the program counter will indicate or point to the next instruction to be fetched. The following list contains the assembly code.

..... *Setting the initial values of C1, C2, C3, L*

ROM[0] = CLAC $\triangleright\ AC \leftarrow 0$

ROM[1] = MVACMAR $\triangleright\ MAR \leftarrow AC$

ROM[2] = LDAC $\triangleright\ AC \leftarrow DRAM[MAR = 0]$

ROM[3] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[4] = MVACL $\triangleright\ L \leftarrow AC(L = 256)$

ROM[5] = MVL $\triangleright\ AC \leftarrow L$

ROM[6] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[7] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[8] = MVACC1 $\triangleright\ C1 \leftarrow AC(C1 = 258)$

ROM[9] = CLAC $\triangleright AC \leftarrow 0$

ROM[10] = INAC $\triangleright AC \leftarrow AC + 1$

ROM[11] = MVACC3 $\triangleright C3 \leftarrow AC(C3 = 1)$

ROM[12] = CLAC $\triangleright AC \leftarrow 0$

ROM[13] = INAC $\triangleright AC \leftarrow AC + 1$

ROM[14] = INAC $\triangleright AC \leftarrow AC + 1$

ROM[15] = MVACC2 $\triangleright C2 \leftarrow AC(C2 = 2)$

..... *Setting the value of reg* $E = L \times (L - 2)$

ROM[16] = MVL $\triangleright AC \leftarrow L$

ROM[17] = DEAC $\triangleright AC \leftarrow AC - 1$

ROM[18] = DEAC $\triangleright AC \leftarrow AC - 1$

ROM[19] = MUL256 $\triangleright AC \leftarrow AC \times 256$

ROM[20] = MVACE $\triangleright E \leftarrow AC(E = 65024)$

ROM[21] = CLAC $\triangleright AC \leftarrow 0$

ROM[22] = MVACT $\triangleright T \leftarrow AC(T = 0)$

..... *Total* $\leftarrow$ *Total + Top left pixel*

ROM[23] = MVC1 $\triangleright AC \leftarrow C1$

ROM[24] = SUBL $\triangleright AC \leftarrow AC - L$

ROM[25] = DEAC $\triangleright AC \leftarrow AC - 1$

ROM[26] = MVACMAR $\triangleright MAR \leftarrow AC$

ROM[27] = LDAC $\triangleright AC \leftarrow DRAM[MAR]$(top left pixel)

ROM[28] = ADDT $\triangleright AC \leftarrow AC + T$

ROM[29] = MVACT $\triangleright T \leftarrow AC$

..... *Total* $\leftarrow$ *Total* $+ (2 \times$ *Top mid pixel)*

ROM[30] = MVC1 $\triangleright AC \leftarrow C1$

ROM[31] = SUBL $\triangleright AC \leftarrow AC - L$

ROM[32] = MVACMAR $\triangleright MAR \leftarrow AC$

ROM[33] = LDAC $\triangleright AC \leftarrow DRAM[MAR]$(top mid pixel)

ROM[34] = MUL2 $\triangleright AC \leftarrow AC \times 2$

ROM[35] = ADDT $\triangleright AC \leftarrow AC + T$

ROM[36] = MVACT $\triangleright T \leftarrow AC$

..... *Total* $\leftarrow$ *Total + Top right pixel*

ROM[37] = MVC1 $\triangleright AC \leftarrow C1$

ROM[38] = SUBL $\triangleright AC \leftarrow AC - L$

ROM[39] = INAC $\triangleright AC \leftarrow AC + 1$

ROM[40] = MVACMAR $\triangleright MAR \leftarrow AC$

ROM[41] = LDAC $\triangleright AC \leftarrow DRAM[MAR]$(top right pixel)

ROM[42] = ADDT $\triangleright AC \leftarrow AC + T$

ROM[43] = MVACT $\triangleright T \leftarrow AC$

..... *Total* $\leftarrow$ *Total* $+ (2 \times$ *Mid left pixel)*

ROM[44] = MVC1 $\triangleright AC \leftarrow C1$

ROM[45] = DEAC $\triangleright AC \leftarrow AC - 1$

ROM[46] = MVACMAR $\triangleright MAR \leftarrow AC$

ROM[47] = LDAC $\triangleright AC \leftarrow DRAM[MAR]$(mid left pixel)

ROM[48] = MUL2 $\triangleright AC \leftarrow AC \times 2$

ROM[49] = ADDT $\triangleright AC \leftarrow AC + T$

ROM[50] = MVACT $\triangleright T \leftarrow AC$

..... *Total* $\leftarrow$ *Total* $+ (4 \times$ *Centre pixel)*

ROM[51] = MVC1 $\triangleright AC \leftarrow C1$

ROM[52] = MVACMAR $\triangleright MAR \leftarrow AC$

ROM[53] = LDAC $\triangleright AC \leftarrow DRAM[MAR]$(centre pixel)

ROM[54] = MUL4 $\triangleright AC \leftarrow AC \times 4$

ROM[55] = ADDT $\triangleright AC \leftarrow AC + T4$

ROM[56] = MVACT $\triangleright T \leftarrow AC$

..... *Total* $\leftarrow$ *Total* $+ (2 \times$ *Mid right pixel)*

ROM[57] = MVC1 $\triangleright AC \leftarrow C1$

ROM[58] = INAC $\triangleright AC \leftarrow AC + 1$

ROM[59] = MVACMAR $\triangleright MAR \leftarrow AC$

ROM[60] = LDAC $\triangleright AC \leftarrow DRAM[MAR]$(mid right pixel)

ROM[61] = MUL2 $\triangleright AC \leftarrow AC \times 2$

ROM[62] = ADDT $\triangleright\ AC \leftarrow AC + T$

ROM[63] = MVACT $\triangleright\ T \leftarrow AC$

..... *Total* $\leftarrow$ *Total + Bottom left pixel*

ROM[64] = MVC1 $\triangleright\ AC \leftarrow C1$

ROM[65] = ADDL $\triangleright\ AC \leftarrow AC + L$

ROM[66] = DEAC $\triangleright\ AC \leftarrow AC - 1$

ROM[67] = MVACMAR $\triangleright\ MAR \leftarrow AC$

ROM[68] = LDAC $\triangleright\ AC \leftarrow DRAM[MAR]$(bottom left pixel)

ROM[69] = ADDT $\triangleright\ AC \leftarrow AC + T$

ROM[70] = MVACT $\triangleright\ T \leftarrow AC$

..... *Total* $\leftarrow$ *Total* $+ (2 \times$ *Bottom mid pixel)*

ROM[71] = MVC1 $\triangleright\ AC \leftarrow C1$

ROM[72] = ADDL $\triangleright\ AC \leftarrow AC + L$

ROM[73] = MVACMAR $\triangleright\ MAR \leftarrow AC$

ROM[74] = LDAC] $\triangleright\ AC \leftarrow DRAM[MAR]$(bottom mid pixel)

ROM[75] = MUL2 $\triangleright\ AC \leftarrow AC \times 2$

ROM[76] = ADDT $\triangleright\ AC \leftarrow AC + T$

ROM[77] = MVACT $\triangleright\ T \leftarrow AC$

..... *Total* $\leftarrow$ *Total + Bottom right pixel*

ROM[78] = MVC1 $\triangleright\ AC \leftarrow C1$

ROM[79] = ADDL $\triangleright\ AC \leftarrow AC + L$

ROM[80] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[81] = MVACMAR $\triangleright\ MAR \leftarrow AC$

ROM[82] = LDAC $\triangleright\ AC \leftarrow DRAM[MAR]$(bottom right pixel)

ROM[83] = ADDT $\triangleright\ AC \leftarrow AC + T$

ROM[84] = MVACT $\triangleright\ T \leftarrow AC$

ROM[85] = MVT $\triangleright\ AC \leftarrow T$

ROM[86] = DIV $\triangleright\ AC \leftarrow AC/16$

ROM[87] = MVC3 $\triangleright\ AC \leftarrow C3$

ROM[88] = MVACMAR $\triangleright\ MAR \leftarrow AC$

ROM[89] = STAC $\triangleright\ DRAM[MAR] \leftarrow AC$

..... *Updating C1, C2, C3 values*

ROM[90] = MVC3 $\triangleright\ AC \leftarrow C3$

ROM[91] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[92] = MVACC3 $\triangleright\ C3 \leftarrow AC$

ROM[93] = MVC2 $\triangleright\ AC \leftarrow C2$

ROM[94] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[95] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[96] = MVACC2 $\triangleright\ C2 \leftarrow AC$

ROM[97] = MVC1 $\triangleright\ AC \leftarrow C1$

ROM[98] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[99] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[100] = MVACC1 $\triangleright\ C1 \leftarrow AC$

..... *Checking the conditions before starting the next cycle*

ROM[101] = MVC1 $\triangleright\ AC \leftarrow C1$

ROM[102] = SUBE $\triangleright\ AC \leftarrow AC - E(\text{If C1} \geq 65024, \text{stop calculations})$

ROM[103] = JMPZ $\triangleright$ If Z = 1, go to the endop or else continue

ROM[104] = L1

..... *Checking if we are at the rightmost column*

ROM[105] = MVC2 $\triangleright\ AC \leftarrow C2$

ROM[106] = SUBL $\triangleright\ AC \leftarrow AC - L(\text{If } C2 = 256, \text{re update the C1, C2 values})$

ROM[107] = JMNZ $\triangleright$ If Z = 0, go to L2 (clear AC & continue)

ROM[108] = L2

..... *Re-updating C1, C2 values*

ROM[109] = MVC1 $\triangleright\ AC \leftarrow C1$

ROM[110] = ADDL $\triangleright\ AC \leftarrow AC + L$

ROM[111] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[112] = INAC $\triangleright\ AC \leftarrow AC + 1$

ROM[113] = MVACC1 $\triangleright\ C1 \leftarrow AC$

ROM[114] = CLAC $\triangleright\ AC \leftarrow 0$

ROM[115] = INAC ▷ $AC \leftarrow AC + 1$

ROM[116] = INAC ▷ $AC \leftarrow AC + 1$

ROM[117] = MVACC2 ▷ $C2 \leftarrow AC$

..... *Continuing convolution*

ROM[118] = JUMP ▷ Go to L2

ROM[119] = L2 ▷ Clear AC and then convolve

..... *Redirecting to endop when C1 = 65024*

ROM[120] = NOP

3.6 Instruction Cycle

The instruction cycle consists of three stages; FETCH, DECODE, and EXECUTE. Once the processor is enabled, these three stages would cycle until the process is completed.

1. FETCH
   The FETCH stage could be decomposed to two sub-stages. In the first stage, the next instruction to be executed is fetched from the instruction memory using the current value stored in the program counter. Once, the instruction is fetched and stored in the memory buffer register unit, the second sub-stage commences and the program counter value is incremented by 1.
   (a) FETCH1 : MBRU $\leftarrow$ IRAM[PC]; FETCH
   (b) FETCH2 : PC $\leftarrow$ PC + 1
2. DECODE
   The fetched instruction is then passed on to the control unit and the relevant microinstruction is determined by referring to the look-up table (The control signals that govern the modules of the processor are embedded in microinstructions). Moreover, to speed up the process, the next FETCH cycle is commenced if the current instruction has got only one state.
3. EXECUTE
   - NOP
     NOP stands for no operation. This instruction is utilized when it is required to wait until the requested data is available.
   - LDAC
     The 'load AC' instruction is executed in two steps. Initially, the MDR is loaded with data that was stored in the requested location of the DRAM. Then, the data is redirected to the AC from the DRAM.
     (a) LDAC1 : MDR $\leftarrow$ DRAM[MAR]; READ
     (b) LDAC2 : AC $\leftarrow$ MDR
   - STAC
     The 'store AC' instruction is the complementary instruction to LDAC. The AC value

is transferred to the MDR and then, the data is written to the desired location in DRAM.

(a) STAC1 : MDR $\leftarrow$ AC

(b) STAC2 : DRAM[MAR] $\leftarrow$ MDR; WRITE

- CLAC
  The 'clear AC' instruction is used to flush and equate AC to zero.

  (a) CLAC : AC $\leftarrow$ 0; Z $\leftarrow$ 1

- MVACMAR, MVACC1, MVACC2, MVACC3, MVACL, MVACE, MVACT
  This set of instructions transfers the value of AC to MAR and the general purpose registers.

  (a) MVACMAR : MAR $\leftarrow$ AC

  (b) MVACC1 : C1 $\leftarrow$ AC

  (c) MVACC2 : C2 $\leftarrow$ AC

  (d) MVACC3 : C3 $\leftarrow$ AC

  (e) MVACL : L $\leftarrow$ AC

  (f) MVACE : E $\leftarrow$ AC

  (g) MVACT : T $\leftarrow$ AC

- MVC1, MVC2, MVC3, MVCT, MVL
  These instructions are complementary to the previous set of instrucions. That is, these instructions are used to transfer values to the AC from other registers in the processor.

  (a) MVC1 : AC $\leftarrow$ C1

  (b) MVC2 : AC $\leftarrow$ C2

  (c) MVC3 : AC $\leftarrow$ C3

  (d) MVCT : AC $\leftarrow$ T

  (e) MVL : AC $\leftarrow$ L

- INAC
  The 'increase AC' instruction is used to increment the value of AC by 1.

  (a) INAC : AC $\leftarrow$ AC + 1

- DEAC
  The 'decrease AC' instruction is used to decrement the value of AC by 1.

  (a) DEAC : AC $\leftarrow$ AC $-$ 1; IF AC $= 0$, THEN $Z = 1$, ELSE $Z = 0$

- ADDT, ADDL
  The values in AC and T/ L registers are summed together and assigned back to AC. ADDT was used in each convolution step and ADDL was used to update the element number (C1)

  (a) ADDT : AC $\leftarrow$ AC + T

(b) ADDL : AC $\leftarrow$ AC + L

- SUBL, SUBE
  The difference between AC and L/ E registers are calculated and the result is assigned to AC. If the difference is zero in any of the two cases, the zero flag would be set to 1. SUBL is used to check whether the centre pixel (with respect to the kernel) is at the rightmost column and SUBE is used to check whether the centre pixel has reached the last convolution step.

  (a) SUBL : AC $\leftarrow$ AC $- L$; IF AC $= 0$, THEN $Z = 1$, ELSE $Z = 0$

  (b) SUBE : AC $\leftarrow$ AC $- E$; IF AC $= 0$, THEN $Z = 1$, ELSE $Z = 0$

- DIV
  DIV represents the only division instruction utilized in the processor. The convolution kernel sum is 16. Accordingly, at the end of each convolution step, the total should be divided by 16 which is achieved by right shifting the total by 4 bits.

  (a) DIV : AC $\leftarrow$ AC $\gg 4$

- MUL2, MUL4, MUL256
  The multiplication instructions MUL2 and MUL4 are repeatedly used throughout the convolution process whereas the only time MUL256 is used, is when initializing the value of register E.

  (a) MUL2 : AC $\leftarrow$ AC $\ll 1$

  (b) MUL4 : AC $\leftarrow$ AC $\ll 2$

  (c) MUL256 : AC $\leftarrow$ AC $\ll 8$

- JUMP
  Just as any other instruction, the instruction pointed by the PC is retrieved and stored in the MBRU. Then, it is passed on to the PC (to update the program counter).

  (a) JUMP1 : MBRU $\leftarrow$ IRAM[PC]; FETCH

  (b) JUMP2 : C $\leftarrow$ MBRU

  (c) JUMP3 : PC $\leftarrow$ C

- JMPZ
  The flow of instructions is branched if and only the Z flag is set to 1. That is, if Z = 1, the PC is branched to a new value and otherwise, the PC is incremented by 1 as usual.

  (a) JMPZN1 $(Z = 0)$: PC $\leftarrow$ PC + 1

  (b) JMPZY1 $(Z = 1)$: MBRU $\leftarrow$ IRAM[PC]; FETCH

  (c) JMPZY2 $(Z = 1)$: C $\leftarrow$ MBRU

  (d) JMPZY3 $(Z = 1)$: PC $\leftarrow$ C

- JMNZ
  The flow of instructions is branched if and only the Z flag is set to 0. That is, if Z =

0, the PC is branched to a new value and otherwise, the PC is incremented by 1 as usual.

(a) JMNZY1 $(Z = 1)$: PC $\leftarrow$ PC $+ 1$

(b) JMNZN1 $(Z = 0)$: MBRU $\leftarrow$ IRAM[PC]; FETCH

(c) JMNZN2 $(Z = 0)$: C $\leftarrow$ MBRU

(d) JMNZN3 $(Z = 0)$: PC $\leftarrow$ C

4 Modules

4.1 16 -Bit General Purpose Registers (GPR)

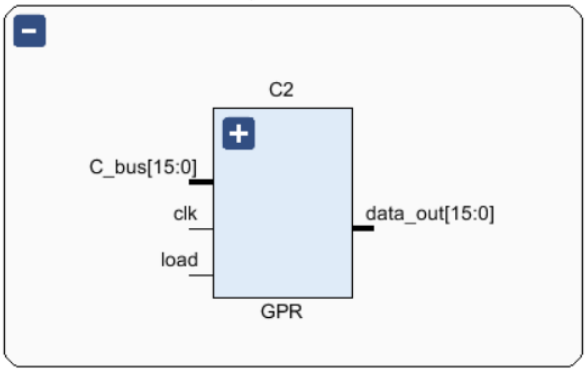

Figure 9: *Block Diagram : General Purpose Register*

General purpose registers are used to hold data from in-between stages of processing. The GPRs used in the implementation are C1, C2, C3, L, E, and T.
The purpose of these registers are,

- **C1** - Keeps track of the element number
- **C2** - Keeps track of the column number
- **C3** - Keeps track of the memory location to which the result is written at the end a convolution step
- **L** - Holds the height/ width of the original image
- **E** - Holds the last value that C1 would take at the end of downsampling
- **T** - Holds the running total during a convolution step

Inputs of GPR

- C_bus - Contains the result from the latest ALU operation
- clk - Generated clock input
- load - Indicates the write enable signal. When load is 1, then C bus value is transferred to the GPR

Output of GPR

- data_out - When the read enable signal is 1, data stored in the GPR would be passed into the B bus.

4.2 Arithmetic & Logic Unit

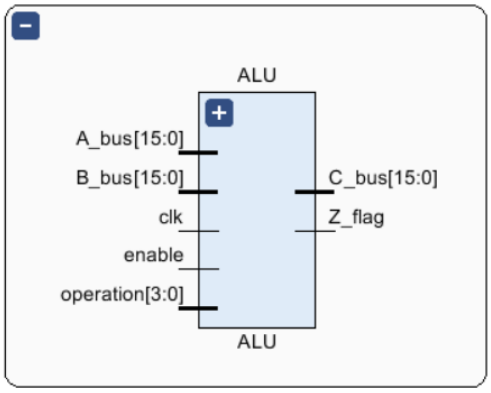

Figure 10: *Block Diagram : ALU*

The ALU is responsible for carrying out the arithmetic operations. The operands for the arithmetic operations are fed to the ALU via the A and B buses. The result is transferred to the C bus and a Z flag is used to indicate whether the output is zero or not. The mathematical operation to be executed is decided by the 4-bit control signal from the control unit.

Inputs of ALU

- A_bus - Operand from the A BUS
- B_bus - Operand from the B BUS
- clk - Generated clock input
- enable - Enabling the ALU
- operation - 4-bit control signal from the control unit, deciding which operation to be carried out

Output of ALU

- C_bus - Result from the ALU operation
- Z_flag - Flag to indicate whether the output is zero or not

4.3 Control Unit

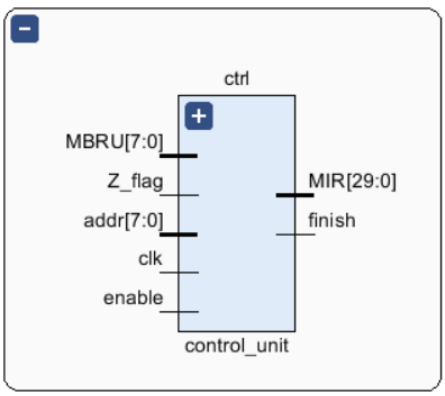

Figure 11: *Block Diagram : Control Unit*

The control unit consists of all possible combinations of the control signals that direct the actions of other processor modules, including the ALU, MDR, AC, general-purpose registers, etc.

Inputs of CU

- MBRU - Instruction to be decoded
- Z_flag - Flag to check whether the output of ALU is zero or not
- addr - Location of the instruction
- clk - Generated clock input
- enable - Enabling the CU

Output of CU

- MIR - 30 bit microinstruction
- finish - Flag to indicate whether the process is completed or not

4.4 Decoder

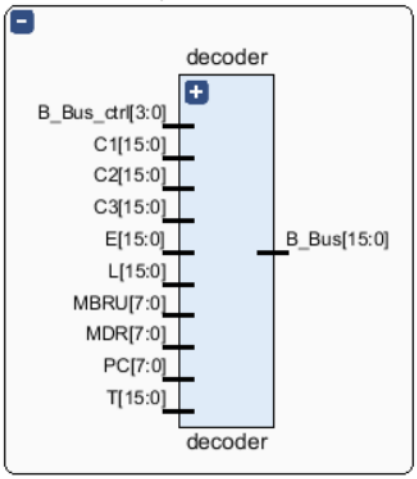

Figure 12: *Block Diagram : Decoder*

As several registers are connected to the B bus, it is of paramount importance to ensure that only one register could transfer values to the B bus at a give time. This is achieved using decoder that is controlled by the control unit.

Inputs of Decoder

- GPRs - C1, C2, C3, L, E, and T
- PC - Program counter
- MDR - Memory data register
- MBRU - Memory buffer register unit
- B_bus_ctrl - Control signal from the control unit

Output of Decoder

- B_bus - Acts as an ALU operand

4.5 IRAM

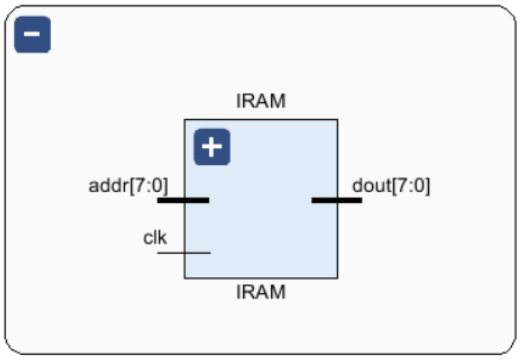

Figure 13: *Block Diagram : IRAM*

The IRAM holds the set of instructions, in the order they are executed by the processor.

Inputs of IRAM

- addr - An 8 bit address that indicates the location of the instruction. The address is given to the IRAM by the PC
- clk - Generated clock input

Output of IRAM

- dout - An 8 bit instruction which is passed to the MBRU

4.6 State Control

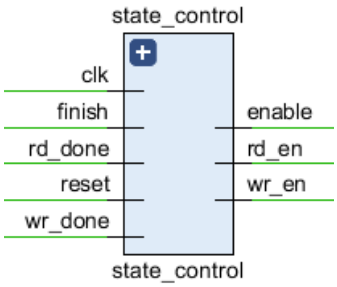

Figure 14: *Block Diagram : State Control*

To facilitate the sequential operation of the DRAM module and the processor module, the functionality is separated into 03 main states.

- DRAM_rd - Data from the text file containing pixel values is read and stored into the DRAM.
- proc_run - Processor executes its functions while reading from and writing to the DRAM as warranted by the respective instructions.

- DRAM_wr - Final pixel values stored in the DRAM is written into the text file as output.

These states are coordinated using the state control module.

Inputs of State Control

- clk - Generated clock input
- reset - Bring the processor to the initial state and proceed onto the DRAM_rd state subsequently.
- finish - Signal provided by the processor indicating it has completed its function.
- rd_done - Signal provided by DRAM indicating that it has completed reading the text file.
- wr_done - Signal provided by DRAM indicating that it has completed writing to the text file.

Outputs of State Control

- enable - Signal given to processor to commence its functions
- wr_en - Signal given to DRAM to start reading from the text file
- rd_en - Signal given to DRAM to start writing to the text file

4.7 DRAM

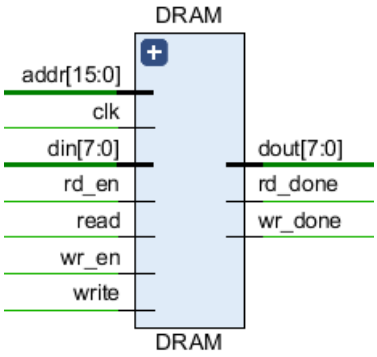

Figure 15: *Block Diagram : DRAM*

DRAM consists of 65536 registers of 8 bit width each. The address width of the DRAM is hence 16 bits. Since the processor should not access the DRAM during the read from text and write to text instances, this was implemented as a state machine with a sequential flow between the states.

Inputs of DRAM

- clk - Generated clock input
- addr - 8 bit address for the register location in DRAM
- write - Write enable signal from the processor
- read - Read enable signal from the processor
- rd_en - Read enable signal from the state control

- wr_en - Write enable signal from the state control
- d_in - data input from the processor

Output of DRAM

- dout - Data to be fed into the processor
- rd_done - Signal to indicate completion of reading from text file
- wr_done - Signal to indicate completion of writing to text file

4.8 MAR

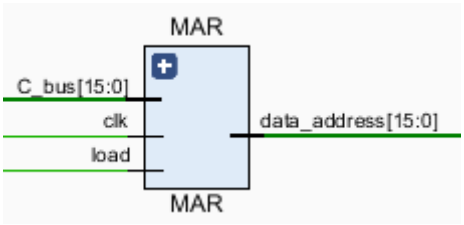

Figure 16: *Block Diagram : MAR*

Memory address register holds the address of the memory location from/ to which data is to be fetched/ stored.

Inputs of MAR

- C_bus - Contains the result from the latest ALU operation
- clk - Generated clock input
- load - Indicates the write enable signal. When load is 1, then C bus value is transferred to the MAR

Output of MAR

- data_address - A 16 bit address that indicates the location of the DRAM from/ to which data is to be read/ written.

4.9 MDR

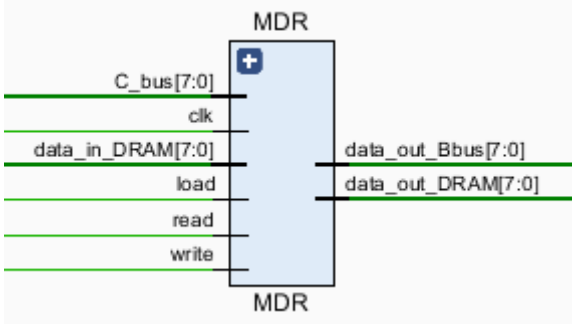

Figure 17: *Block Diagram : MDR*

Memory data register holds data that is loaded from the memory or to be loaded to the memory.

Inputs of MDR

- C_bus - Contains the result from the latest ALU operation
- clk - Generated clock input
- load - Indicates the write enable signal. When load is 1, then C bus value is transferred to the MDR
- write - When this bit is 1, data in the MDR will be written to the DRAM
- read - When this bit is 1, data will be read from the DRAM into the MDR
- data_in_dram - Contains the data that is transferred to the MDR when the 'read' bit is 1

Output of MDR

- data_out_Bbus - When the load enable signal is 1, data stored in the MDR would be passed into the B bus.
- data_out_DRAM - When the write enable signal is 1, data stored in the MDR would be passed into the DRAM

4.10 MBRU

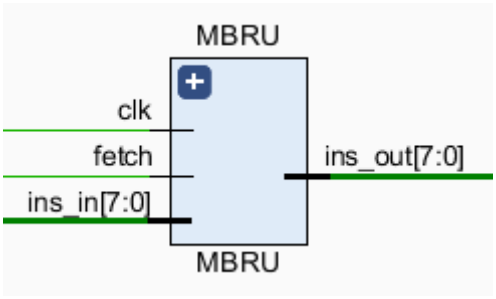

Figure 18: *Block Diagram : MBRU*

The memory buffer register unit stores the instruction to be decoded, which was received as a response by the IRAM to the PC's request.

Inputs of MBRU

- fetch - Indicates whether the stored instruction should be passed to the CU or not
- clk - Generated clock input
- ins_in - The 8 bit instruction received from the IRAM

Output of MBRU

- ins_out - If the 'fetch' bit is 1, the stored instruction is passed to the CU

4.11 PC

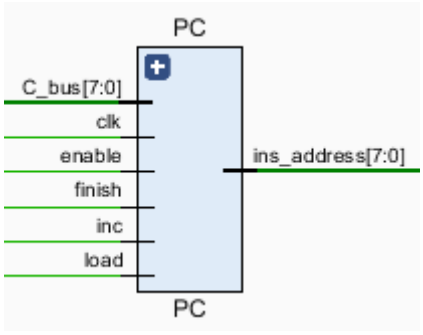

Figure 19: *Block Diagram : PC*

Program counter acts as a pointer to the next instruction to be fetched. Since incrementing the PC by one is a common occurrence, it also supports the increment function in-house without having to route through the ALU.

Inputs of PC

- clk - Generated clock input
- enable - Signal indicating the commencing of processor operations
- finish - Signal indicating the completion of processor operations
- load - Enabling the loading of the next instruction from the bus input
- inc - Enabling the incrementing of the PC
- C_bus - Feeding the instruction address through the bus input

Outputs of PC

- ins_address - Location of the next instruction to be fetched

4.12 Overview of The Processor

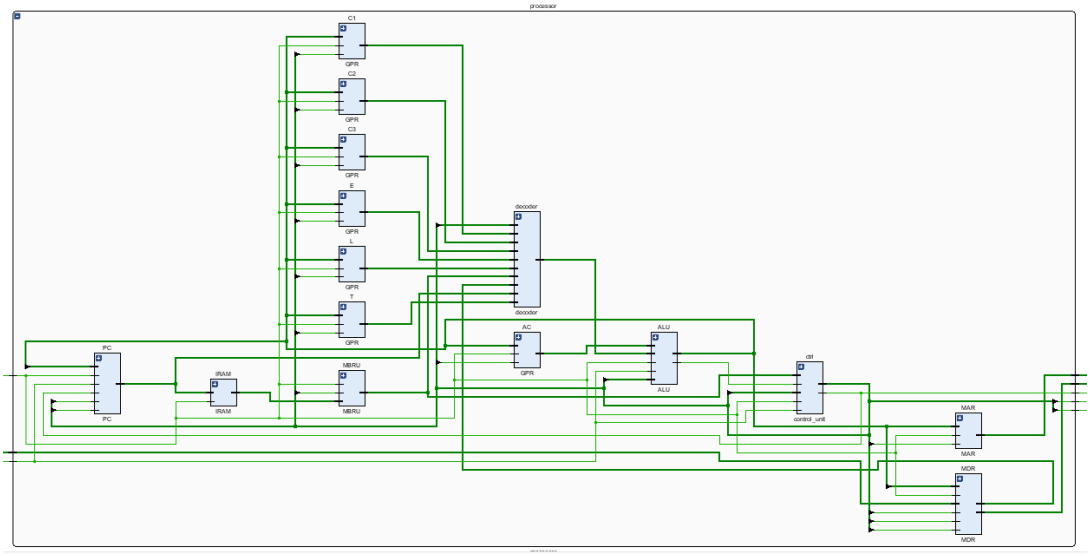

Figure 20: *Block Diagram : Processor*

4.13 Overview of The Designed CPU

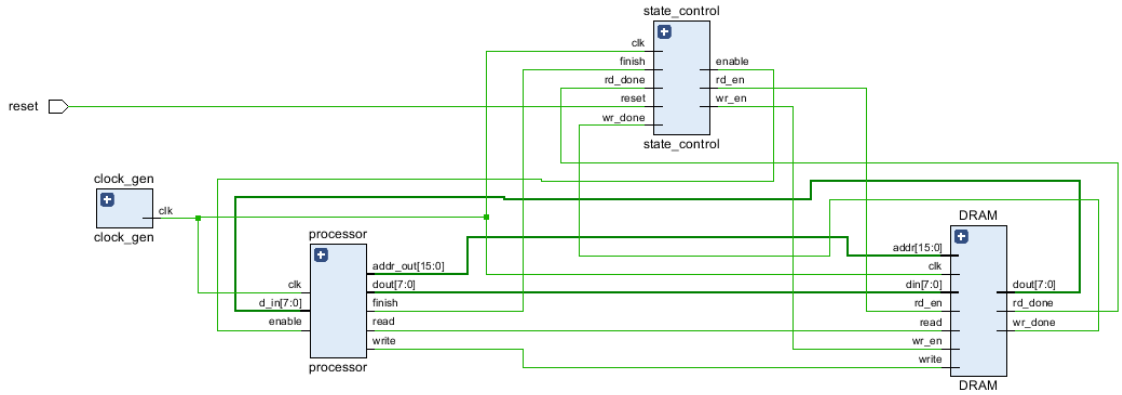

Figure 21: *Block Diagram : Designed CPU*

MASTER module acts as the high level wrapper for the entire system. It interfaces the clock module, processor and the DRAM with the state control module.

5 Testing and Simulations

Each module was simulated on ModelSim to verify that desired outputs are obtained for all possible input combinations.

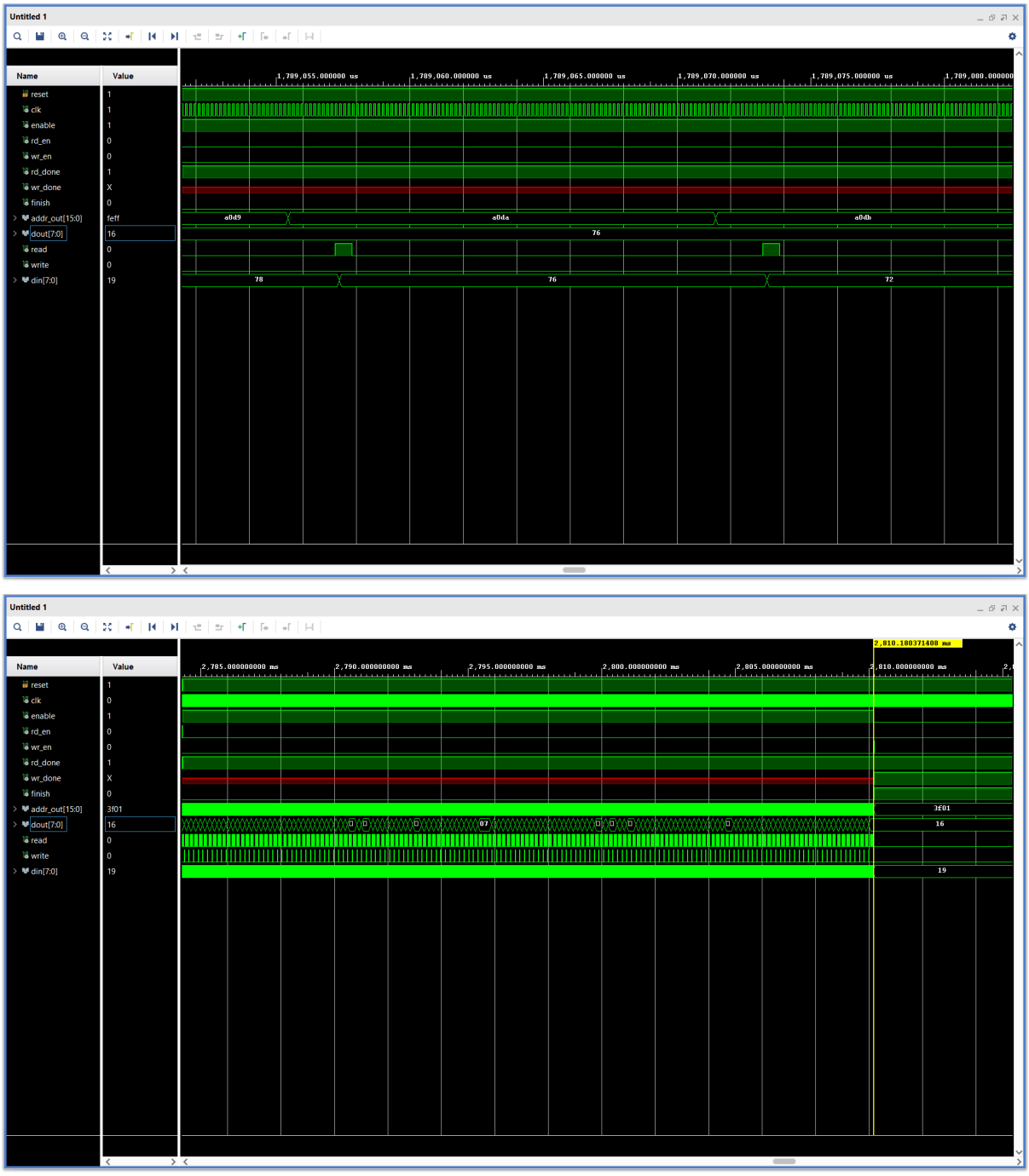

Figure 22: *Processor Simulations*

6 Results and Error Analysis

The processor was designed to downsample a 256 by 256 grayscale image by a factor of 2. The downsampled image from the processor was compared with an image that was downsampled using OpenCV's resize command.

Notation:

- $I^{(x)}$ - Downsampled image from the processor
- $I^{(y)}$ - Downsampled image using OpenCV
- μ_x - Mean of $I^{(x)}$
- μ_y - Mean of $I^{(y)}$
- σ_x - Standard deviation of $I^{(x)}$
- σ_y - Standard deviation of $I^{(y)}$
- σ_{xy} - Covariance of $I^{(x)}$ & $I^{(y)}$
- L - Height/ width of the downsampled image

The following metrics were used for the error analysis.

- **Root Mean Squared Error (RMSE)**

$$RMSE = \sqrt{\frac{\sum_{i=1}^{L}\sum_{j=1}^{L}\left(I_{i,j}^{(x)} - I_{i,j}^{(y)}\right)^2}{L^2}} = \frac{\sqrt{\sum_{i=1}^{L}\sum_{j=1}^{L}\left(I_{i,j}^{(x)} - I_{i,j}^{(y)}\right)^2}}{L}$$

- **Structure Similarity Index (SSIM)**
  Instead of directly calculating the SSIM, we decided to consider all three aspects of SSIM. Those are, luminance similarity, contrast similarity and structure similarity. The following parameters were calculated beforehand.

$$C_1 = (0.01 \times L)^2, C_2 = (0.03 \times L)^2, C_3 = C_2/2$$

$$\text{Luminance Similarity} = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

$$\text{Contrast Similarity} = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

$$\text{Structure Similarity} = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

The following 256 by 256 grayscale image was used as the test image.

Figure 23: *Original Image*

The diagrams below represent the downsampled images from the processor and OpenCV library.

Figure 24: *Downsampled Image from The Processor*

Figure 25: *Downsampled Image Using OpenCV*

For the above test image, following metrics were received.

```
Error analysis: Started
...
Computing the root mean squared error...
The root mean squared error: 5.36559722484526


Computing the luminance similarity...
The luminance similarity: 0.9999896148048144


Computing the contrast similarity...
The contrast similarity: 1.0039373498088133


Computing the structure similarity...
The structure similarity: 0.9923780774992024
...
Error analysis : Completed!
```

The above results show that our processor is capable of producing desired outputs once an image is fed to it. The root mean squared error per pixel is close to 5 which might be due to the approximations we made while assigning values to the Gaussian kernel (For the ease of computations, we chose powers of 2 as the kernel elements). Nevertheless, we can see that the similarity index is close to 1, indicating that our downsampled image and OpenCV's downsampled image are similar regarding luminance, contrast and structure.

7 Summary

In this section, we will outline the general procedure to be followed when using the designed processor to downsample an image. A detailed explanation is given below.

Using a Python script, an image is loaded and converted to an array of unsigned integers. Even though there are 65536 memory locations, all those locations would not be utilized as only 65280-pixel values are transmitted (The last row of the image is not used during convolution). In addition to the pixel values corresponding to the first 255 rows, the height/ width of the image would also be transmitted.

These values are written to a binary text file before being fed to the processor. Once, the processor is enabled, the text file would be read and the pixel values would be stored in the DRAM serially. When the text file is completely read to the DRAM, the processor commences implementing the downsampling algorithm. which is repeated until the tag of the centre pixel of the convolving kernel becomes 65024.

The final values stored in the DRAM are written back to a binary text file. The generated binary text file is read from a Python script where it is converted to an array. The downsampled image is generated using this array and it is saved to the desired location.

8 Discussion

HDL code has to be further amended by implementing the MASTER module to function as a finite state machine to facilitate multiple runs sequentially. Currently the processor completes a single run and idles with only the clock running. With a finite state machine, we can revert it back to the initial state such that another run can be started at the press of the 'reset' button.

Clock period we have specified is 160 ns which implies a clock frequency of 6.25 MHz. However, the clock frequency on many existing FPGA devices may be different to this. We would require to amend the clock module to combine several clock pulses to generate the required clock pulse duration for our duration.

In this implementation we have employed verilog functions 'readmemb' and 'writememb' to read and write data from the file. In the hardware implementation, this data transmission has to be formally carried out through the use of UART protocol. The final design including such data transmission capabilities may require additional functional units such as multiplexers to be used.

To use this processor for tasks other than image down sampling would require additional instructions to be defined. Hence it is important to have a compiler that can expedite the process of adding more instructions to the system. Currently, we have manually added the binary translations of the instructions used.

Appendices

A Python Scripts : Image to Text Conversion

A.0.1 Imports

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

A.0.2 Loading the original image

[ ]:
```
parent_dir = './standard_test_images'
image_name = 'cameraman.tif'

#Loading a grayscale 256 by 256 image
img = cv.resize(cv.imread(os.path.join(parent_dir, image_name), 0), (256,256))

print(f'The shape of the original image: {img.shape}')

#Visualizing the original image
plt.imshow(img, cmap = 'gray')
```

A.0.3 Preprocessing the image

[ ]:
```
print('Preprocessing: Started')
print('.....................................')
# Converting the image into a numpy array
img_array =np.array(img)

# The last row is trimmed from the image before converting to a text file
img_array_cropped = img[0:-1,:]

print(f'The shape of the trimmed image : {img_array_cropped.shape}')

# Flattening the image array before converting to a text file
cropped_img_flatten = img_array_cropped.flatten()

print(f'The shape of the flattened image : {cropped_img_flatten.shape}')

# Creating an empty array
final_array = np.zeros((cropped_img_flatten.shape[0]+1,)).astype(np.uint8)

# Attaching 255 (width - 1) as the first element
final_array[0] = 255

# Attaching the flattened image
final_array[1:] = cropped_img_flatten
```

```
print(f'The shape of the final vector: {final_array.shape}')
print('.....................................')
print('Preprocessing: Finished Successfully!')
```

A.0.4 Image to text conversion

[ ]:
```
print('Image to text conversion: Started')
print('.....................................')
print('Image to text conversion: Pending')

# Writing the image pixels to an 8bit binary text file
with open('image_cameraman.txt', 'w') as f:
  for i in range(final_array.shape[0]-1):
    f.write('{0:08b}\n'.format(final_array[i]))
  f.write('{0:08b}'.format(final_array[final_array.shape[0]-1]))

print('.....................................')
print('Image to text conversion: Finished Successfully!')
print('You may check the destination folder now')
```

B Python Scripts : Text to Image Conversion

B.0.1 Imports

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

B.0.2 Text to array conversion

[ ]:
```
print('Text to array conversion: Started')
print('.....................................')
print('Text to array conversion: Pending')

# Open and read the text file
with open('downsampled_image_cameraman.txt', 'r') as f:
    pixels = [line.strip() for line in f]

print('.....................................')
print('Text to array conversion: Finished Successfully!')
```

B.0.3 Preprocessing

[ ]:
```
print('Preprocessing: Started')
print('.....................................')
# Extract the relevant elements from the generated list
pixels_req = pixels[1:16130]
```

```
# Convert the binary value to unsigned integers
pixels_int = [int(i,2) for i in pixels_req]

# Convert the list to a numpy array
pixels_arr = np.array(pixels_int)

# Reshape the array to the size of the downsampled image
pixels_arr_final = np.reshape(pixels_arr, (127,127))

print(f'The shape of the downsampled image: {pixels_arr_final.shape}')
print('.....................................')
print('Preprocessing: Finished Successfully!')
```

[ ]:
```
# Visualizing the downsampled image
plt.imshow(pixels_arr_final, cmap = 'gray')
```

B.0.4 Saving the image

[ ]:
```
parent_dir = './standard_test_images_downsampled'
image_name = 'cameraman_downsampled.jpg'
cv.imwrite(os.path.join(parent_dir, image_name), pixels_arr_final)
print('Image saved to the destination folder successfully!')
```

C Python Scripts : Error Analysis

C.0.1 Imports

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

C.0.2 Loading the original image

[ ]:
```
parent_dir_orig = './standard_test_images'
image_name_orig = 'cameraman.jpg'

#Loading a grayscale 256 by 256 image
img = cv.resize(cv.imread(os.path.join(parent_dir_orig, image_name_orig), 0),␣
↪(256,256))

print(f'The shape of the original image: {img.shape}')

#Visualizing the original image
plt.imshow(img, cmap = 'gray')
```

C.0.3 Downsampling the original image

[ ]:
```
img_ds_orig = cv.resize(img, (127,127))

print(f'The shape of the downsampled image (Using OpenCV) : {img_ds_orig.
 ↪shape}')

#Visualizing the downsampled image
plt.imshow(img_ds_orig, cmap = 'gray')
```

C.0.4 Loading the downsampled image (From the processor)

[ ]:
```
parent_dir_processed = './standard_test_images_downsampled'
image_name_processed = 'cameraman_downsampled.jpg'

#Loading a grayscale 256 by 256 image
img_ds = cv.imread(os.path.join(parent_dir_processed, image_name_processed),␣
 ↪0)

print(f'The shape of the downsampled image (Using our processor): {img_ds.
 ↪shape}')

#Visualizing the original image
plt.imshow(img_ds, cmap = 'gray')
```

C.0.5 Error analysis

[ ]:
```
print('Error analysis: Started')
print('.............................................')

img_ds_flat = np.reshape(img_ds, 16129)
img_ds_orig_flat = np.reshape(img_ds_orig, 16129)

# calculating the parameters for the analysis
c_1 = (0.01*255)**2
c_2 = (0.03*255)**2
c_3 = c_2/2
mu_p = np.mean(img_ds_flat)
mu_o = np.mean(img_ds_orig_flat)
sigma_p = np.std(img_ds_flat)
sigma_o = np.std(img_ds_orig_flat)
sigma_op = np.mean(np.cov(img_ds_flat, img_ds_orig_flat))

print('Computing the root mean squared error...........')

# Computing the root mean squared error
mse = np.power(np.sum(np.power(img_ds_flat - img_ds_orig_flat,2))/16129, 0.5)
print(f'The root mean squared error: {mse}')
print('\n')
```

```
print('Computing the luminance similarity..........')

# Computing the luminance similarity
lum = (2*mu_o*mu_p + c_1)/(mu_o**2 + mu_p**2 + c_1)
print(f'The luminance similarity: {lum}')
print('\n')

print('Computing the contrast similarity..........')

# Computing the contrast similarity
con = (2*sigma_o*sigma_p + c_2)/(sigma_o**2 + sigma_p**2 + c_3)
print(f'The contrast similarity: {con}')
print('\n')


print('Computing the structure similarity..........')

# Computing the structure similarity
struc = (sigma_op + c_3)/(sigma_o*sigma_p + c_3)
print(f'The structure similarity: {struc}')

print('.............................................')
print('Error analysis : Completed!')
```

D Verilog Scripts : Processor and Its Modules

Listing 1: ALU

```
/*************************************************************
EN3030 - Circuits and System Design
180497C - 180554B - 180564F - 180574K
Designing a custom processor for Image Downsampling

Module : ALU
*************************************************************/

module ALU(
    input [15:0] A_bus,
    input [15:0] B_bus,
    input [3:0] operation,
    input enable,
    input clk,
    output reg[15:0] C_bus,
    output reg Z_flag

);
//Define the ALU operations
parameter ADD = 4'b0001 ;
parameter SUB = 4'b0010 ;
parameter PASSATOC = 4'b0011 ;
parameter PASSBTOC = 4'b0100 ;
parameter INCAC = 4'b0101 ;
parameter DECAC = 4'b0110 ;
parameter LSHIFT1 = 4'b0111 ;
parameter LSHIFT2 = 4'b1000 ;
parameter LSHIFT8 = 4'b1001 ;
parameter RSHIFT4 = 4'b1010 ;
```

```
parameter RESET = 4'b1011;

initial begin
    C_bus = 16'b0;
    Z_flag = 1'b0;
end

reg [1:0] state = 2'b0;
reg start = 1'b0;

always@ ( posedge enable )
    start <= 1'b1;

always@ ( negedge clk )
    begin
            if ( start ) begin
                case ( state )
                    2'b00 : state = 2'b01 ;
                    2'b01 : state = 2'b10 ;
                    2'b10 : state = 2'b11 ;
                    2'b11 : state = 2'b00 ;
                    default : state = state ;
                endcase
            end
    end
always@ ( posedge clk )
    begin
    if ( start ) begin
    if ( state == 2'b11) begin
    case ( operation )
        ADD : C_bus <= A_bus + B_bus ;

        SUB : begin
        C_bus = A_bus - B_bus ;
        Z_flag = ( C_bus == 16'b0) ? 1'b1 : 1'b0;
        end

        PASSATOC : C_bus <= A_bus ;

        PASSBTOC : C_bus <= B_bus ;

        INCAC : C_bus <= A_bus + 1;

        DECAC : C_bus <= A_bus - 1; // can put a Z_flag value too

        LSHIFT1 : C_bus <= A_bus << 1;

        LSHIFT2 : C_bus <= A_bus << 2;

        LSHIFT8 : C_bus <= A_bus << 8;

        RSHIFT4 : C_bus <= A_bus >> 4;

        RESET : C_bus <= 16'b0;

        // default : C_bus = C_bus ;
    endcase
    end
    end
    end
endmodule
```

Listing 2: GPR

```
// Author : Dasun Prematilaka
// Last Updated : 29/06/2022

// GPR - General Purpose Register
// These registers will be storing intermediate values during convolution

module GPR(
    input clk,
    input load,
    input [15:0] C_bus, //connects to the MIR load signal
    output reg [15:0] data_out //connects to the b bus via a mux

);
always@(posedge clk)
    begin
        if (load) data_out <= C_bus;
    end

endmodule
```

Listing 3: IRAM

```
// Author : Dasun Prematilaka
// Last Updated : 29/06/2022

//IRAM - Instruction RAM
//Stores the instructions to be executed sequentially

module IRAM (
    input clk,
    input [7:0] addr, //connects to the PC
    output reg [7:0] dout //connects to the MBRU
);
    reg [7:0] ROM [120:0];

    // Defining the assembly code

    parameter FETCH = 8'd0;
    parameter NOP = 8'd2;
    parameter LDAC = 8'd3;
    parameter STAC = 8'd5;
    parameter CLAC = 8'd7;
    parameter MVACMAR = 8'd8;
    parameter MVACC1 = 8'd9;
    parameter MVACC2 = 8'd10;
    parameter MVACC3 = 8'd11;
    parameter MVACL = 8'd12;
    parameter MVACE = 8'd13;
    parameter MVACT = 8'd14;
    parameter MVC1 = 8'd15;
    parameter MVC2 = 8'd16;
    parameter MVC3 = 8'd17;
    parameter MVT = 8'd18;
    parameter INAC = 8'd19;
    parameter DEAC = 8'd20;
    parameter ADDT = 8'd21;
    parameter ADDL = 8'd22;
    parameter SUBE = 8'd23;
    parameter SUBL = 8'd24;
    parameter DIV = 8'd25;
    parameter MUL2 = 8'd26;
    parameter MUL4 = 8'd27;
    parameter MUL256 = 8'd28;
    parameter JUMP = 8'd29;
```

```
    parameter JMPZ = 8'd32;
    parameter JMNZ = 8'd37;
    parameter MVL = 8'd42;

    parameter L1 = 8'd120;
    parameter L2 = 8'd21;


    always@(posedge clk)
        begin
            dout <= ROM[addr];
        end

    initial begin
        // setting the initial values of C1, C2, C3, L
        ROM[0] = CLAC; //AC <- 0
        ROM[1] = MVACMAR; //MAR <- AC
        ROM[2] = LDAC; //AC <- DRAM[MAR=0]
        ROM[3] = INAC; //AC <- AC + 1
        ROM[4] = MVACL; //L <- AC (L = 256)
        ROM[5] = MVL; //AC <- L
        ROM[6] = INAC; //AC <- AC + 1
        ROM[7] = INAC; //AC <- AC + 1
        ROM[8] = MVACC1; //C1 <- AC (C1 = 258)
        ROM[9] = CLAC; //AC <- 0
        ROM[10] = INAC; //AC <- AC + 1
        ROM[11] = MVACC3; //C3 <- AC (C3 = 1)
        ROM[12] = CLAC; //AC <- 0
        ROM[13] = INAC; //AC <- AC + 1
        ROM[14] = INAC; //AC <- AC + 1
        ROM[15] = MVACC2; // C2 <- AC (C2 = 2)

        //setting the value of reg E = L * (L-2)
        ROM[16] = MVL; //AC <- L
        ROM[17] = DEAC; //AC <- AC - 1
        ROM[18] = DEAC; //AC <- AC - 1
        ROM[19] = MUL256; //AC <- AC*256
        ROM[20] = MVACE; //E <- AC (E = 65024)

        ROM[21] = CLAC; //AC <- 0
        ROM[22] = MVACT; //T <- AC (T = 0)

        //total <- total + top left pixel
        ROM[23] = MVC1; //AC <- C1
        ROM[24] = SUBL; //AC <- AC - L
        ROM[25] = DEAC; //AC <- AC - 1
        ROM[26] = MVACMAR; //MAR <- AC
        ROM[27] = LDAC; //AC <- DRAM[MAR] (top left pixel)
        ROM[28] = ADDT; //AC <- AC + T
        ROM[29] = MVACT; //T <- AC

        //total <- total + 2*top mid pixel
        ROM[30] = MVC1; //AC <- C1
        ROM[31] = SUBL; //AC <- AC - L
        ROM[32] = MVACMAR; //MAR <- AC
        ROM[33] = LDAC; //AC <- DRAM[MAR] (top mid pixel)
        ROM[34] = MUL2; //AC <- AC * 2
        ROM[35] = ADDT; //AC <- AC + T
        ROM[36] = MVACT; //T <- AC

        //total <- total + top right pixel
        ROM[37] = MVC1; //AC <- C1
        ROM[38] = SUBL; //AC <- AC - L
```

```
        ROM[39] = INAC; //AC <- AC + 1
        ROM[40] = MVACMAR; //MAR <- AC
        ROM[41] = LDAC; //AC <- DRAM[MAR] (top right pixel)
        ROM[42] = ADDT; //AC <- AC + T
        ROM[43] = MVACT; //T <- AC

        //total <- total + 2 * mid left pixel
        ROM[44] = MVC1; //AC <- C1
        ROM[45] = DEAC; //AC <- AC - 1
        ROM[46] = MVACMAR; //MAR <- AC
        ROM[47] = LDAC; //AC <- DRAM[MAR] (mid left pixel)
        ROM[48] = MUL2; //AC <- AC * 2
        ROM[49] = ADDT; //AC <- AC + T
        ROM[50] = MVACT; //T <- AC

        //total <- total + 4 * centre pixel
        ROM[51] = MVC1; //AC <- C1
        ROM[52] = MVACMAR; //MAR <- AC
        ROM[53] = LDAC; //AC <- DRAM[MAR] (centre pixel)
        ROM[54] = MUL4; //AC <- AC * 4
        ROM[55] = ADDT; //AC <- AC + T
        ROM[56] = MVACT; //T <- AC

        //total <- total + 2 * mid right pixel
        ROM[57] = MVC1; //AC <- C1
        ROM[58] = INAC; //AC <- AC + 1
        ROM[59] = MVACMAR; //MAR <- AC
        ROM[60] = LDAC; //AC <- DRAM[MAR] (mid right pixel)
        ROM[61] = MUL2; //AC <- AC * 2
        ROM[62] = ADDT; //AC <- AC + T
        ROM[63] = MVACT; //T <- AC

        //total <- total + bottom left pixel
        ROM[64] = MVC1; //AC <- C1
        ROM[65] = ADDL; //AC <- AC + L
        ROM[66] = DEAC; //AC <- AC - 1
        ROM[67] = MVACMAR; //MAR <- AC
        ROM[68] = LDAC; //AC <- DRAM[MAR] (bottom left pixel)
        ROM[69] = ADDT; //AC <- AC + T
        ROM[70] = MVACT; //T <- AC

        //total <- total + 2 * bottom mid pixel
        ROM[71] = MVC1; //AC <- C1
        ROM[72] = ADDL; //AC <- AC + L
        ROM[73] = MVACMAR; //MAR <- AC
        ROM[74] = LDAC; //AC <- DRAM[MAR] (bottom mid pixel)
        ROM[75] = MUL2; //AC <- AC * 2
        ROM[76] = ADDT; //AC <- AC + T
        ROM[77] = MVACT; //T <- AC

        //total <- total + bottom right pixel
        ROM[78] = MVC1; //AC <- C1
        ROM[79] = ADDL; //AC <- AC + L
        ROM[80] = INAC; //AC <- AC + 1
        ROM[81] = MVACMAR; //MAR <- AC
        ROM[82] = LDAC; //AC <- DRAM[MAR] (bottom right pixel)
        ROM[83] = ADDT; //AC <- AC + T
        ROM[84] = MVACT; //T <- AC
        ROM[85] = MVT; //AC <- T
        ROM[86] = DIV; //AC <- AC / 16
        ROM[87] = MVC3; //AC <- C3
        ROM[88] = MVACMAR; //MAR <- AC
        ROM[89] = STAC; //DRAM[MAR] <- AC
```

```
169
170         //updating C1, C2, C3 values
171         ROM[90] = MVC3; //AC <    C3
172         ROM[91] = INAC; //AC <    AC + 1
173         ROM[92] = MVACC3; //C3 <    AC
174         ROM[93] = MVC2; //AC <    C2
175         ROM[94] = INAC; //AC <    AC + 1
176         ROM[95] = INAC; //AC <    AC + 1
177         ROM[96] = MVACC2; //C2 <    AC
178         ROM[97] = MVC1; //AC <    C1
179         ROM[98] = INAC; //AC <    AC + 1
180         ROM[99] = INAC; //AC <    AC + 1
181         ROM[100] = MVACC1; //C1 <    AC
182
183         //checking the conditions before starting the next cycle
184         ROM[101] = MVC1; //AC <- C1
185         ROM[102] = SUBE; //AC <- AC - E (If C1 > 65024, stop calculations)
186         ROM[103] = JMPZ; //If Z == 1, go to the endop or else continue
187         ROM[104] = L1;
188
189         //checking if we are at the rightmost column
190         ROM[105] = MVC2; //AC <- C2
191         ROM[106] = SUBL; //AC <- AC - L
192         // (If C2 = 256, go to next row, re update the C1, C2 values)
193         ROM[107] = JMNZ; // If Z == 0,
194         // go to L2 (clear AC - continue without reupdating C1, C2)
195         ROM[108] = L2;
196
197         //reupdating C1, C2 values
198         ROM[109] = MVC1; //AC <- C1
199         ROM[110] = ADDL; //AC <- AC + L
200         ROM[111] = INAC; //AC <- AC + 1
201         ROM[112] = INAC; //AC <- AC + 1
202         ROM[113] = MVACC1; //C1 <- AC
203
204         ROM[114] = CLAC; //AC <- 0
205         ROM[115] = INAC; //AC <- AC + 1
206         ROM[116] = INAC; //AC <- AC + 1
207         ROM[117] = MVACC2; //C2 <- AC
208
209         //continuing convolution
210         ROM[118] = JUMP; //go to L2
211         ROM[119] = L2; //clear AC and then convolve
212
213         //redirecting to endop when C1 = 65024
214         ROM[120] = NOP;
215
216     end
217
218 endmodule
```

Listing 4: Control Unit

```
1  // /*************************************************************
2  // EN3030 - Circuits and System Design
3  // 180497C - 180554B - 180564F - 180574K
4  // Designing a custom processor for Image Downsampling
5
6  // Module : Control Unit
7  // *************************************************************/
8
9
10 module control_unit(
11
```

```
input enable,
input clk,
input Z_flag,
input [7:0] addr,
input [7:0] MBRU,
output reg [29:0] MIR,
output finish

);

reg [1:0] state = 2'b00;
reg start = 1'b0;
reg [29:0] ROM[0:42];
reg check = 1'b0;

assign finish = check;

//parameter define

parameter JMPZ1 = 8'd32;
parameter JMPZN1 = 8'd33;
parameter JMPZY1 = 8'd34;
parameter JMNZ1 = 8'd37;
parameter JMNZY1 = 8'd38;
parameter JMNZN1 = 8'd39;
parameter FETCH2 = 8'd1;
parameter NOP = 8'd2;

initial
  begin
    MIR = 30'b0;
  end

always@(posedge enable)
  start = 1'b1;

always@(negedge clk)
  begin
    if (start) begin
      case(state)
        2'b00 : state = 2'b01;
        2'b01 : state = 2'b10;
        2'b10 : state = 2'b11;
        2'b11 : state = 2'b00;
        default : state = state;
      endcase
    end
  end

always @(posedge clk)
  begin
    if (state == 2'b11) begin
      case(addr)
        FETCH2 : MIR = {MBRU,ROM[FETCH2][21:0]};
        JMPZ1  : if (Z_flag == 1'b0) MIR = {8'd33,ROM[JMPZ1][21:0]};
                 else MIR = {8'd34,ROM[JMPZ1][21:0]};
         JMNZ1  : if (Z_flag == 1'b1) MIR = {8'd38,ROM[JMNZ1][21:0]};
                 else MIR = {8'd39,ROM[JMNZ1][21:0]};
        NOP    : check = 1'b1;
        default : MIR = ROM[addr];
      endcase
    end
  end
```

```
75
76 //define the microinstruction in the control store
77
78 initial
79 begin
80
81 ROM[0]  = 30'b00000001_0000_0000000000_100_0_0000; //FETCH1
82 ROM[1]  = 30'bXXXXXXXX_0000_0000000000_000_1_0000; //FETCH2
83 ROM[2]  = 30'b00000010_0000_0000000000_000_0_0000; //NOP (check this)
84 ROM[3]  = 30'b00000100_0000_0000000000_010_0_0000; //LDAC1
85 ROM[4]  = 30'b00000000_0100_0000000001_000_0_0001; //LDAC2
86 ROM[5]  = 30'b00000110_0011_0100000000_000_0_0000; //STAC1
87 ROM[6]  = 30'b00000000_0000_0000000000_001_0_0000; //STAC2
88 ROM[7]  = 30'b00000000_1011_0000000001_000_0_0000; //CLAC
89 ROM[8]  = 30'b00000000_0011_1000000000_000_0_0000; //MVACMAR
90 ROM[9]  = 30'b00000000_0011_0000100000_000_0_0000; //MVACC1
91 ROM[10] = 30'b00000000_0011_0000010000_000_0_0000; //MVACC2
92 ROM[11] = 30'b00000000_0011_0000001000_000_0_0000; //MVACC3
93 ROM[12] = 30'b00000000_0011_0001000000_000_0_0000; //MVACL
94 ROM[13] = 30'b00000000_0011_0000000010_000_0_0000; //MVACE
95 ROM[14] = 30'b00000000_0011_0000000100_000_0_0000; //MVACT
96 ROM[15] = 30'b00000000_0100_0000000001_000_0_0101; //MVC1
97 ROM[16] = 30'b00000000_0100_0000000001_000_0_0110; //MVC2
98 ROM[17] = 30'b00000000_0100_0000000001_000_0_0111; //MVC3
99 ROM[18] = 30'b00000000_0100_0000000001_000_0_1000; //MVT
100 ROM[19] = 30'b00000000_0101_0000000001_000_0_0000; //INAC
101 ROM[20] = 30'b00000000_0110_0000000001_000_0_0000; //DEAC
102 ROM[21] = 30'b00000000_0001_0000000001_000_0_1000; //ADDT
103 ROM[22] = 30'b00000000_0001_0000000001_000_0_0100; //ADDL
104 ROM[23] = 30'b00000000_0010_0000000001_000_0_1001; //SUBE
105 ROM[24] = 30'b00000000_0010_0000000001_000_0_0100; //SUBL
106 ROM[25] = 30'b00000000_1010_0000000001_000_0_0000; //DIV
107 ROM[26] = 30'b00000000_0111_0000000001_000_0_0000; //MUL2
108 ROM[27] = 30'b00000000_1000_0000000001_000_0_0000; //MUL4
109 ROM[28] = 30'b00000000_1001_0000000001_000_0_0000; //MUL512
110
111 ROM[29] = 30'b00011110_0000_0000000000_100_0_0000; //JUMP1
112 ROM[30] = 30'b00011111_0100_0000000000_000_0_0011; //JUMP2
113 ROM[31] = 30'b00000000_0000_0010000000_000_0_0000;
114
115 ROM[32] = 30'bxxxxxxxx_0000_0000000000_000_0_0000; //JMPZ1
116 ROM[33] = 30'b00000000_0000_0000000000_000_1_0000; //JMPZN1
117 ROM[34] = 30'b00100011_0000_0000000000_100_0_0000; //JMPZY1
118 ROM[35] = 30'b00100100_0100_0000000000_000_0_0011; //JMPZY2
119 ROM[36] = 30'b00000000_0000_0010000000_000_0_0000; //JMPZY3
120
121 ROM[37] = 30'bxxxxxxxx_0000_0000000000_000_0_0000; //JMNZ1
122 ROM[38] = 30'b00000000_0000_0000000000_000_1_0000; //JMNZY1
123 ROM[39] = 30'b00101000_0000_0000000000_100_0_0000; //JMNZN1
124 ROM[40] = 30'b00101001_0100_0000000000_000_0_0011; //JMNZN2
125 ROM[41] = 30'b00000000_0000_0010000000_000_0_0000; //JMNZN3
126
127 ROM[42] = 30'b00000000_0100_0000000001_000_0_0100; //MVL
128
129 end
130
131 endmodule
132
133
134
135
136
137
```

```
138
139
140
141
```

Listing 5: MAR

```
// Author : Dasun Prematilaka
// Last Updated : 29/06/2022

// MAR - Memory Address Register
// Stores the address of the memory location from/
// to which data is to be fetched/ stored

module MAR (
    input clk,
    input load,
    input [15:0] C_bus,
    output reg [15:0] data_address
);
    always@(posedge clk)
        begin
            if (load) data_address <= C_bus;
        end

endmodule
```

Listing 6: MBRU

```
// Author : Dasun Prematilaka
// Last Updated : 28/06/2022

// MBRU - Memory Buffer Register Unit
// Loads and keeps the location of the instruction to be decoded

module MBRU (
    input clk,
    input fetch, //connects to the MIR fetch signal
    input [7:0] ins_in, //connects to the IRAM out
    output reg [7:0] ins_out //connects to the control store
);
    always@(posedge clk)
        begin
            if (fetch) ins_out <= ins_in;
        end

endmodule
```

Listing 7: PC

```
/*************************************************************
EN3030 - Circuits and System Design
180497C - 180554B - 180564F - 180574K
Designing a custom processor for Image Downsampling

Module : Program Counter
*************************************************************/
module PC(
    input clk,
    input enable,
    input finish,
    input load,
    input inc,
    input [7:0] C_bus,
```

```
15
16      output reg [7:0] ins_address
17  );
18
19  //initializing start flag and instruction address
20  reg start = 1'b0;
21
22  initial begin
23      ins_address = 8'b0;
24  end
25
26  //State block (to ensure CPI=4)
27  reg [1:0] state = 2'b0;
28
29  always@(posedge enable)
30      start <= 1'b1;
31
32  always@(negedge clk)
33      begin
34          if (start) begin
35              case (state)
36                  2'b00 : state = 2'b01;
37                  2'b01 : state = 2'b10;
38                  2'b10 : state = 2'b11;
39                  2'b11 : state = 2'b00;
40                  default : state = state;
41              endcase
42          end
43      end
44
45  //Program counter functions
46  always@(posedge clk)
47      begin
48          if (finish) ins_address <= ins_address;
49
50          else if (start) begin
51              if (state == 2'b11 && load) begin
52                  ins_address <= C_bus;
53              end
54
55              else if (state == 2'b11 && inc) begin
56                  ins_address <= ins_address + 8'b00000001;
57              end
58
59              else begin
60                  ins_address <= ins_address;
61              end
62          end
63      end
64
65  endmodule
```

Listing 8: B Bus Decoder

```
1  /*************************************************************
2  EN3030 - Circuits and System Design
3  180497C - 180554B - 180564F - 180574K
4  Designing a custom processor for Image Downsampling
5
6  Author : Avishka Sandeepa
7  Last Updated : 28/06/2022
8  Module : B Bus Multiplexer
9  *************************************************************/
10
```

```
module decoder (
    // initializing registers
    input [15:0] L, C1 , C2 , C3 , T , E ,
    input [7:0] PC ,
    input [7:0] MDR ,
    input [7:0] MBRU ,
    input [3:0] B_Bus_ctrl ,
    output reg [15:0] B_Bus
);

always @(L or C1 or C2 or C3 or T or E or PC or MDR or MBRU or B_Bus_ctrl or B_Bus)
begin
    case (B_Bus_ctrl)
        4'b0001 : B_Bus <= {8'b0 , MDR};
        4'b0010 : B_Bus <= {8'b0 , PC};
        4'b0011 : B_Bus <= {8'b0 , MBRU};
        4'b0100 : B_Bus <= L;
        4'b0101 : B_Bus <= C1;
        4'b0110 : B_Bus <= C2;
        4'b0111 : B_Bus <= C3;
        4'b1000 : B_Bus <= T;
        4'b1001 : B_Bus <= E;
        default : B_Bus <= 16'b0;
    endcase

end

endmodule
```

Listing 9: Processor

```
module processor(
    input enable,
    input clk,
    input [7:0] d_in,

    output [15:0] addr_out,
    output [7:0] dout,
    output finish,
    output read,
    output write
);

wire [7:0] ins_address;
wire [7:0] ins_in;
wire [7:0] ins_out;
wire [29:0] MIR;
wire [15:0] A_bus;
wire [15:0] B_bus;
wire [15:0] C_bus;
wire Z_flag;
wire [15:0] L_bus;
wire [15:0] C1_bus;
wire [15:0] C2_bus;
wire [15:0] C3_bus;
wire [15:0] T_bus;
wire [15:0] E_bus;
wire [7:0] MDR_bus;
wire finish_flag;

assign read = MIR[6];
assign write = MIR[5];

assign finish = finish_flag;
```

```
34
35 //control_unit
36
37 control_unit ctrl(
38     .enable(enable),
39     .clk(clk),
40     .Z_flag(Z_flag),
41     .addr(MIR[29:22]),
42     .MBRU(ins_out),
43     .MIR(MIR),
44     .finish(finish_flag)
45 );
46
47 //program_counter
48
49 PC PC(
50     .clk(clk),
51     .enable(enable),
52     .load(MIR[15]),
53     .inc(MIR[4]),
54     .C_bus(C_bus[7:0]),
55     .ins_address(ins_address),
56     .finish(finish)
57 );
58
59 //instruction memory
60
61 IRAM IRAM(
62     .clk(clk),
63     .addr(ins_address),
64     .dout(ins_in)
65 );
66
67 //MBRU
68
69 MBRU MBRU (
70     .clk(clk),
71     .fetch(MIR[7]),
72     .ins_in(ins_in),
73     .ins_out(ins_out)
74 );
75
76 //General Purpose Register - L
77 GPR L (
78     .clk(clk),
79     .load(MIR[14]),
80     .C_bus(C_bus),
81     .data_out(L_bus)
82 );
83
84 //General Purpose Register - C1
85 GPR C1 (
86     .clk(clk),
87     .load(MIR[13]),
88     .C_bus(C_bus),
89     .data_out(C1_bus)
90 );
91
92 //General Purpose Register - C2
93 GPR C2 (
94     .clk(clk),
95     .load(MIR[12]),
96     .C_bus(C_bus),
```

```
    .data_out(C2_bus)
);

//General Purpose Register - C3
GPR C3 (
    .clk(clk),
    .load(MIR[11]),
    .C_bus(C_bus),
    .data_out(C3_bus)
);

//General Purpose Register - T
GPR T (
    .clk(clk),
    .load(MIR[10]),
    .C_bus(C_bus),
    .data_out(T_bus)
);

//General Purpose Register - E
GPR E (
    .clk(clk),
    .load(MIR[9]),
    .C_bus(C_bus),
    .data_out(E_bus)
);

//Accumulator - AC
GPR AC (
    .clk(clk),
    .load(MIR[8]),
    .C_bus(C_bus),
    .data_out(A_bus)
);

//MDR
MDR MDR(
    .clk(clk),
    .load(MIR[16]),
    .read(MIR[6]),
    .write(MIR[5]),
    .C_bus(C_bus[7:0]),
    .data_in_DRAM(d_in),
    .data_out_Bbus(MDR_bus),
    .data_out_DRAM(dout)
);

//MAR
MAR MAR(
    .clk(clk),
    .load(MIR[17]),
    .C_bus(C_bus),
    .data_address(addr_out)
);

//ALU
ALU ALU (
    .A_bus(A_bus),
    .B_bus(B_bus),
    .operation(MIR[21:18]),
    .C_bus(C_bus),
    .enable(enable),
    .clk(clk),
```

```
    .Z_flag(Z_flag)
);

//decoder
decoder decoder(
    .L(L_bus),
    .C1(C1_bus),
    .C2(C2_bus),
    .C3(C3_bus),
    .T(T_bus),
    .E(E_bus),
    .PC(ins_address),
    .MDR(MDR_bus),
    .MBRU(ins_out),
    .B_Bus_ctrl(MIR[3:0]),
    .B_Bus(B_bus)
);

endmodule
```

E Verilog Scripts : DRAM

Listing 10: DRAM

```
/*************************************************************
EN3030 - Circuits and System Design
180497C - 180554B - 180564F - 180574K
Designing a custom processor for Image Downsampling

Author : Avishka Sandeepa
Last Updated : 03/07/2022
Module : Data Random Access Memory
*************************************************************/

module DRAM (
    input [15:0] addr,
    input clk,
    input write,
    input read,
    input rd_en,
    input wr_en,
    input [7:0] din,
    output reg rd_done,
    output reg wr_done,
    output reg [7:0] dout
);

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

reg [1:0] state = 2'b00;

initial begin
assign state = {wr_en,rd_en};
end

// parameter IDLE = 2'b00;
parameter DRAM_rd = 2'b01;
parameter proc_run = 2'b00;
parameter DRAM_wr = 2'b10;

// reg [2:0] state = IDLE;


```

```
always @(posedge clk) begin
    case (state)

        DRAM_rd :
            begin
                $readmemb("E:/Users/dasun/Documents
                /Proc_Downsampling/Python Scripts/image_cameraman.txt",RAM);
                rd_done <= 1'b1;
            end

        proc_run :
            begin
                if (wr_en == 1'b0)
                    if (write) begin
                        RAM[addr] <= din;
                    end

                    if (read) begin
                        dout <= RAM[addr];
                    end
                else if (wr_en == 1'b1) state <=DRAM_wr;
            end

        DRAM_wr :
            begin
                $writememb("E:/Users/dasun/Documents
                /Proc_Downsampling/Python Scripts/
                downsampled_image_cameraman.txt",RAM);
                wr_done=1;
            end


    endcase
end



endmodule
```

F Verilog Scripts : State Control

Listing 11: State Control

```
module state_control (
    input clk,
    input reset,
    input finish,
    input rd_done,
    input wr_done,

    output reg enable,
    output reg wr_en,
    output reg rd_en );


parameter IDLE = 3'b000;
parameter DRAM_rd = 3'b001;
parameter proc_run = 3'b010;
parameter DRAM_wr = 3'b011;
parameter complete = 3'b100;

reg [2:0] state = IDLE;

```

```
initial begin
    enable = 0;
    rd_en = 0;
    wr_en = 0;
end

always @(posedge clk) begin
    case (state)
        IDLE :
            begin
                if (reset) begin
                    rd_en <= 1'b1;
                    wr_en <= 1'b0;
                    enable <= 1'b0;
                    state <= DRAM_rd;
                end
            end

        DRAM_rd :
            begin
                if (rd_done) begin
                    rd_en <= 1'b0;
                    wr_en <= 1'b0;
                    enable <= 1'b1;
                    state <= proc_run;
                end
            end

        proc_run :
            begin
                if (finish) begin
                    rd_en <= 1'b0;
                    wr_en <= 1'b1;
                    enable <= 1'b0;
                    state <= DRAM_wr;
                end
            end

        DRAM_wr :
            begin
                if (wr_done) begin
                    rd_en <= 1'b0;
                    wr_en <= 1'b0;
                    enable <= 1'b0;
                    state <= complete;
                end
            end

        complete :
            begin
                state <= complete;
            end
    endcase
end

endmodule
```

G Verilog Scripts : Clock

Listing 12: Clock

```
`timescale 1ns/1ps
module clock_gen (
```

```
    output reg clk
);

parameter clk_pd = 160; //Clock period in ns

initial begin
    clk = 0;
    forever
         #(clk_pd/2) clk = ~clk;
end


endmodule
```

H Verilog Scripts : Master

Listing 13: Master

```
module MASTER (
    input reset
);

wire clk;
wire enable;
wire rd_en;
wire wr_en;
wire rd_done;
wire wr_done;
wire finish;
wire [15:0] addr_out;
wire [7:0] dout;
wire read;
wire write;
wire [7:0] din;

//State Control
state_control state_control(
    .reset(reset),
    .finish(finish),
    .clk(clk),
    .rd_done(rd_done),
    .wr_done(wr_done),
    .enable(enable),
    .rd_en(rd_en),
    .wr_en(wr_en)
);

//Clock Generator
clock_gen clock_gen(
    .clk(clk)
);

//DRAM
DRAM DRAM(
    .clk(clk),
    .rd_en(rd_en),
    .wr_en(wr_en),
    .rd_done(rd_done),
    .wr_done(wr_done),
    .din(dout),
    .addr(addr_out),
    .dout(din),
    .read(read),
```

```
    .write(write)
);

//Processor
processor processor(
    .clk(clk),
    .finish(finish),
    .enable(enable),
    .d_in(din),
    .addr_out(addr_out),
    .dout(dout),
    .read(read),
    .write(write)
);

endmodule
```