Department of Electronic and Telecommunication Engineering

University of Moratuwa, Sri Lanka

EN3030 - Circuits and Systems Design

FPGA Based Processor Design

Project Report

Submitted by

Kumarasinghe H.A.N.H	180337M
Mendis N.P.A.	180398A
Nagasinghe K.R.Y.	180411K
Thalagala B.P.	180631J

Submitted on

July 13, 2022

Abstract

Application-specific custom processors are popular at present due to factors such as low power consumption, economic feasibility, efficiency, reliability, ability to be enclosed in very small spaces, etc. Here we approach the problem of downsampling an image which can be easily achieved in a high-level programming paradigm, using a reduced instruction set computer, or RISC, and present a low-level design-based solution.

This project is intended to perform a fully automated custom processor in order to image downsampling operations using the software tool, Vivado. The purpose of the project is to downsample an image by factor 2. This report contains a detailed discussion of the Algorithm, Instruction Set Architecture(ISA) and other resources that were utilised in the project.

Table of Contents

1. Introduction

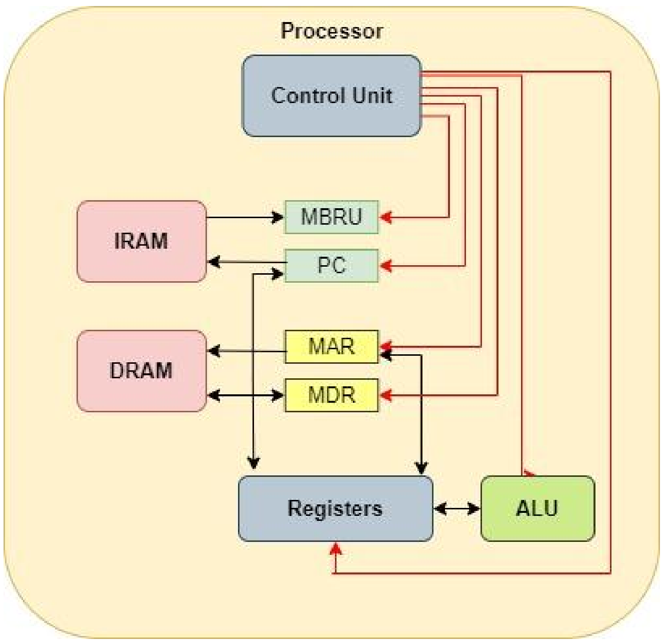
1.1. Problem Statement

Computational speed has long been a major research focus and a factor driving the development of modern computing. Computing speed is highly dependent on hardware capacity, code complexity, resource availability and scheduling, computing algorithms, and other factors. Our main objective of the given project was to design a customised processor to downsample a given grayscale image by a factor of two.

1.2. Overview of Processor Design

The primary objective of this project is to design a Microprocessor and a CPU (Central Processing Unit) which can downsample the given image and generate correct results. The coding was done using Verilog Hardware Description Language (HDL). Xilinx Vivado was used as an analysis and synthesis tool for the design of the processor and for the simulation of Verilog HDL designs. The report includes the Microprocessor and CPU design, the test codes that are used, and the simulations were done using all the components.

The basic arithmetic, logical, input/output (I/O) operations specified by the instructions need to be executed to carry out the instructions of a computer program. The electronic circuitry within a computer which performs the above-mentioned task is called the Central Processing Unit (CPU). The main two building blocks of a CPU are the Processor and the Control Unit (CU).



1.3. Methodology

The following steps were followed when designing the processor.

1. Develop algorithm
2. Simulate the algorithm using *Python* High-Level language with the help of *OpenCV* and *NumPy* libraries.
3. Define the specification of the processor such as the number of registers, memory requirements and bus size according to the algorithm.
4. Design the ISA including instruction building blocks, microinstructions, datapath and control lookup table.
5. Convert the algorithm to assembly code using the defined instructions
6. Implement the modules in Verilog.
7. Testing and verifying the results.

2. The Algorithm

This section explains the working principles of the algorithm which was implemented inside the designed custom processor, to downsample the given 256x256 grayscale (single-channel) 8-bit image. Since the image is an 8-bit single channel, the maximum and minimum values a pixel can take become 255 and 0 respectively.

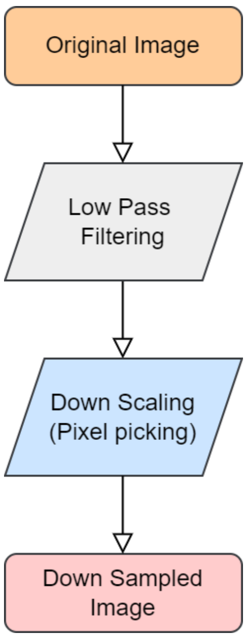
"*Downsampling an image*" refers to the process of representing a larger image, using a much lower number of pixels which were carefully picked from the original image. The rule of picking those pixels can be defined according to the application.

A few of the advantages of image downsampling can be presented as follows.

- Downsampling converts the images to more manageable sizes.
- Downsampled images can be processed faster in resource-constrained environments due to their size reduction.
- Downsampled images consume less storage when compared to the original images.

2.1. Addressing the Problem

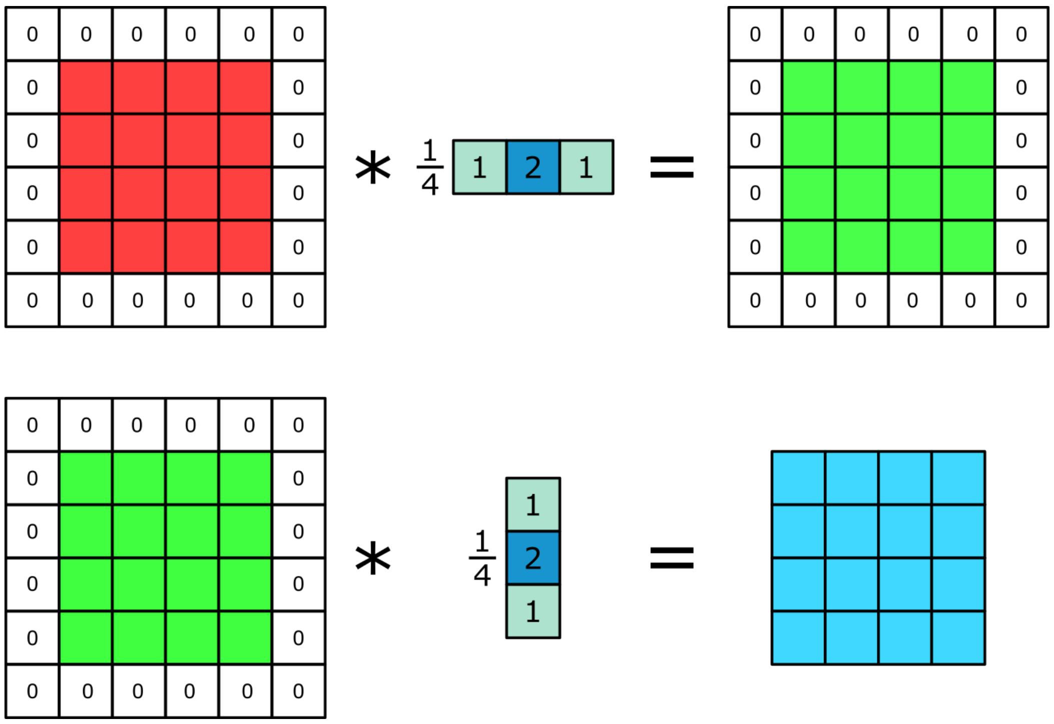
Downsampling the given 256x256 image consists of two major steps. As the initial step, the image undergoes low pass filtering to remove unwanted high-frequency signal components of the image (an image can be considered as a two-dimensional signal). This reduces the possible significant differences between the consecutive pixel intensities in the image and eventually increases the quality of the downsampled image. The next step of the downsampling is to pick the required pixels (every other pixel in the case of this project) from the filtered image to generate the resized image. In this project, the downscaling factor was two. Hence the resultant image size was 128x128. Upcoming sub-sections explain the mentioned processes in detail. The figure given below represents the downsampling process in summary.



2.2. Low Pass Filtering

To remove the high-frequency components in the image to be downsampled, low pass filtering is done before downscaling. In this project, low pass filtering is achieved through convolution of the image with a one-dimensional discrete averaging filter. Convolution is carried out horizontally and vertically separately to reduce the data memory (RAM) access frequency. The convolution of a 4x4 image with the averaging filter is illustrated below. To keep the dimensions of the original image unchanged, zeros are padded around the intermediate image and the original image.

At first horizontal convolution is carried out on the original image. Next vertical convolution is carried out on the image obtained from the horizontal convolution. This gives the required low pass filtered image. The process is illustrated in the below figure.



The values (1, 2, 4) in the filter depicted in the above figure were chosen in such a way that they make the arithmetic operations easy to implement in terms of hardware. Since the final goal of the project is to optimise the processor for image processing, special attention must be given to the speed of the frequently used arithmetic operations. Since convolution consists of a massive amount of multiplications and divisions, those operations were implemented as described below.

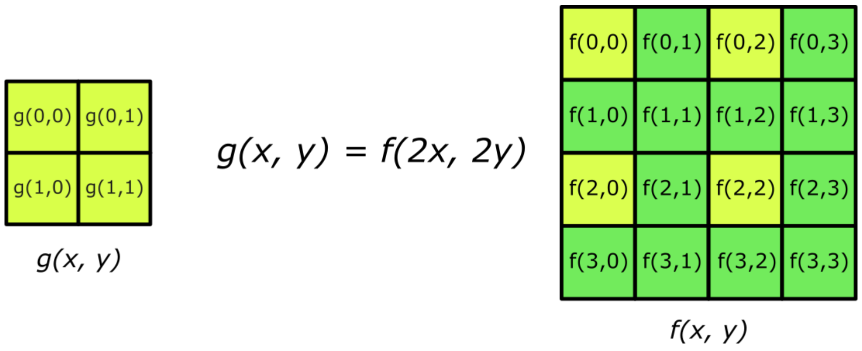
Multiplying a pixel value by powers of two means left shifting the binary value of that pixel by the power. As an example multiplying 5 by 4 is the same as left shifting *'101'* by 2 bits. That is, the answer becomes *'10100'* which is equal to 20. In the same way division from the powers of two becomes simple right shifting. This fact was kept in mind when implementing the processor and that led to faster arithmetic operations which eventually increased the processing speed.

2.3. Down Scaling

In this step pixels from the low pass filtered image are picked depending on a predefined rule. This rule maps the pixels from the original image to the pixels on the downscaled image. Since the downscaling factor was two in this project, the rule is given below was used for the mentioned pixel mapping. That $f(x, y)$ indicates the original image and $g(x, y)$ indicates the downsampled image. Note that coordinate origin is the top left pixel of an image and the x and y axes are vertical and horizontal respectively.

$$g(x, y) = f(2x, 2y)$$

The below figure illustrates the mentioned mapping process.



2.4. Implementation of the Algorithm using Python High-Level Language

A Python script was developed in order to simulate the behaviour of the proposed downsampling algorithm. In addition, two supporting scripts were developed using the same language, for converting a given image to a format compatible to include inside Verilog HDL files and for converting such a format back to a normal image for performance analysis of the processor. The table given below summarises the purpose of each script.

Script	**Purpose**
Script 1	Simulate the behaviour of the proposed downsampling algorithm. The script includes methods to open an image, convert it to a 1D array, carry out horizontal and vertical convolution for low pass filtering and finally downscaling (picking necessary pixels).

Script 2	Convert any image format to a format that is compatible with Verilog HDL. The script includes methods to open any image in grayscale with the help of a third-party library and write a 1D representation of the image to a text file in a format compatible with Verilog HDL.
Script 3	Converts a series of 8-bit binary values stored in a text file back to a normal image.

All the scripts were implemented and tested on the Google Colab platform. Therefore, all the required third-party libraries are pre-installed and no initial configuration is required. The live scripts can be accessed from here (Google Colab Notebook).

Once the high-level representation of the algorithm is tested and verified. The algorithm is then converted into an **Assembly Code** using the developed Instruction Set Architecture (ISA). Since the ISA is custom designed, the resultant assembly code is unique for this project. This assembly code is then stored inside the Instruction Random Access Memory (IRAM) of the implemented processor. Once the processor is enabled, those assembly instructions are fetched, decoded and executed to produce the downsampled image.

3. Instruction Set Architecture (ISA)

The Instruction Set Architecture defines the structure used withinside the layout in depth. This segment covers the structure of the Central Processing Unit (CPU), datapath and the connectivity among numerous modules. The execution of the determined practice set is also mentioned here.

3.1. General Architecture

The design in consideration is a special-purpose processor of which the objective is to down-sample 256 by 256 image into a 128 by 128 image. This architecture provides sufficient storage for data and instructions and handles various ALU and datapath constraints successfully to achieve that objective.
Given below are some of the design specifications of the implemented processor.

1. **Instruction Memory** (**IRAM**) - This RAM block is essentially 26 by 256 (word size = 26 and depth = 256) in size and houses the assembly code of algorithms for filtering (Horizontal and Vertical convolution) and downsampling the image. [25:18] bits contain the opcode and [17:0] bits contain the immediate data.

2. **Data Memory** (**DRAM**) - This is a Data RAM which is designed to accommodate 256 by 256 image which has pixel values varying from 0-255 thus the width of the memory unit is 8 bits (1 byte). We decided the depth of the DRAM to be 2^18 as our original image needed 65536 (2^16) memory locations and we stored the down-sampled image in a new location. We gave 1 bit extra to avoid any problems that may occur due to memory constraints. Therefore our data address is 18 bits.

3. **Memory Address Register** (**MAR**) - The MAR is a 28-bit register that can hold the addresses pointing to the DRAM so that reading from the DRAM or writing to the DRAM can be carried out. The [17:0] bits of the MAR contain the data address needed.

4. **Memory Data Register** (**MDR**) - This 28- bit register has bi-directional connectivity to the Data Memory since it is required to perform both read and write functions. When writing to memory [7:0] bits of the MDR are used. We give the data from the memory to MDR as "20'b0 + data_in".

5. **Memory Buffer Register Unit** (**MBRU**) - The 28-bit register stores the locations of instructions that are read from the instruction memory. These instructions are used to generate the relevant control signal to the units through the control store. [25:18] bits of the instruction input to MBRU are given as instruction output to the control store.

6. **Arithmetic and Logical Unit** (**ALU**) - This performs all the arithmetic and logical operations that are required to filter and down-sample the image. A and B buses provide the inputs and the C bus is reserved for the output of the ALU. The Accumulator has a direct connection (A bus) to the ALU. ALU is fed by 4 control bits from the control unit, which would command the ALU to perform the required task.

7. **Programme Counter** (**PC**) - This 28-bit register keeps the address of the next instruction in the instruction memory. The corresponding instruction will be loaded to the Memory Buffer Register Unit. It has an inbuilt increment operation so that it does not need to access the ALU for incrementing.

8. **Accumulator** (**AC**) - The accumulator is a 28-bit register with a direct connection to the ALU. This is the most commonly used register in the ISA.

9. **Control Unit** (**CU**) - This provides all the control signals to the processor according to the input provided by the MBRU. Outputs a 36-bit control signal which consists of the following. [35:28] bits = next microinstruction address. [27:0] bits = control signals for ALU and registers and B-bus multiplexer.

10. **General-Purpose registers** (**R1, R2, R3, R4, R5, R6, R7, R8, R9**) - These 28-bit general-purpose registers are utilised in storing the intermediate values in the process. Has a control signal as write enable to write to the register.

11. **B bus Multiplexer** - This is used to select the correct register to write to the B-bus. Receives 4 bits as control signals to select the correct register output path to write to B-bus.

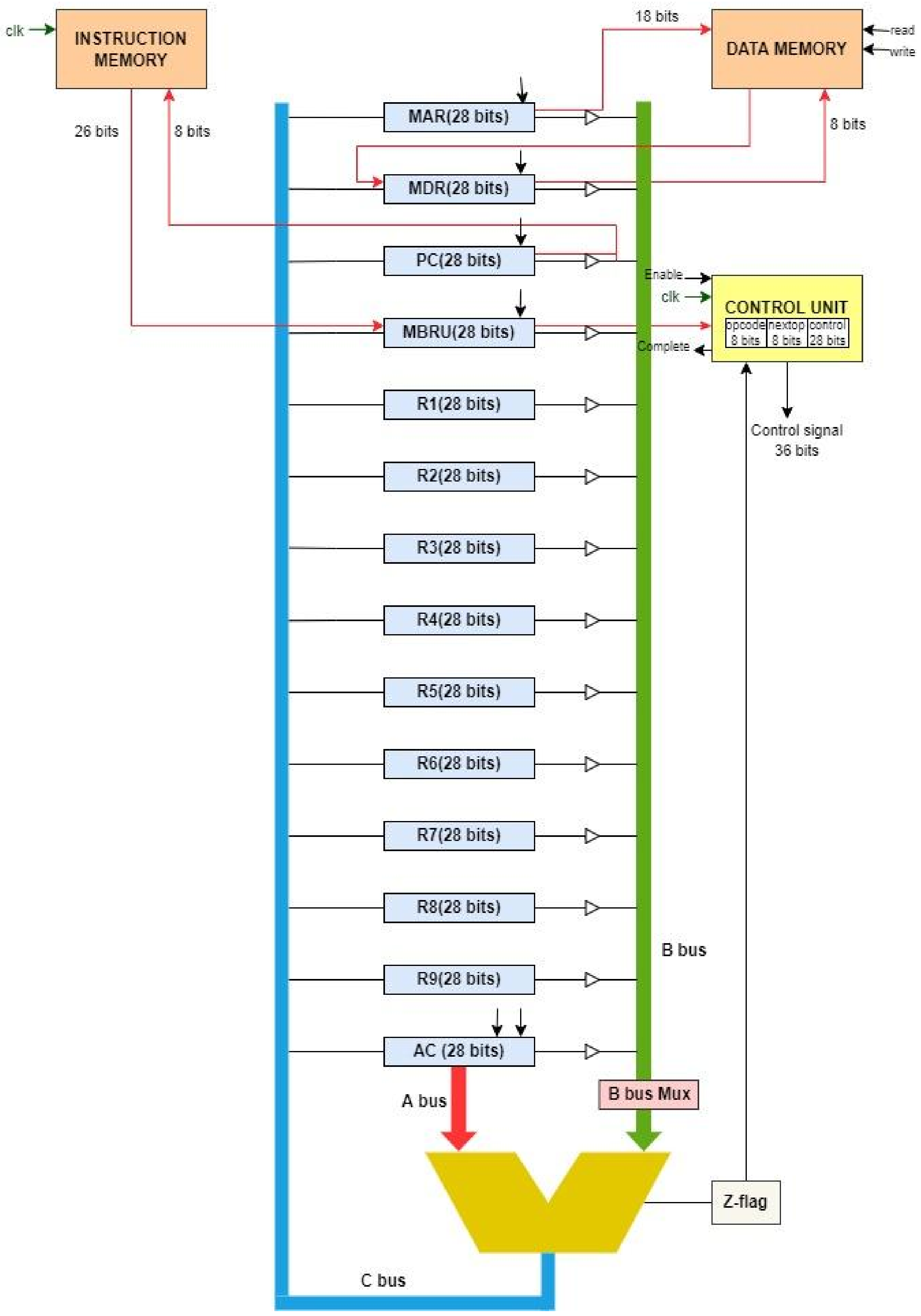
12. **A bus** - This is a 28-bit wired connection to transmit data between the ALU and the accumulator.

13. **B bus** - This is a 28-bit wired connection to transmit data between the B-bus Multiplexer and the ALU.

14. **C bus** - This is a 28-bit wired connection to transmit data between the ALU and the set of registers. Used to write data to the registers depending on the control signals given to the registers.

3.2. Data Path

The data path that facilitates the data transmission between the Instruction Memory, Data Memory, Control Store, ALU and the set of registers can be illustrated as follows.



3.3. Instruction Set

Instruction set contains the assembly code that we write for the processor. The following table represents all the instructions that we used to build the assembly code that we intend to use to perform the down-sampling operation.

Memory address	Instruction	Microinstruction	Break down of microinstruction
0	FETCH	FETCH1	MBRU ← IRAM[PC]; FETCH
1		FETCH2	IDLE
2		FETCH3	PC←PC+1
3	NOOP	NOOP	IDLE
4	CLAC	CLAC	AC←0,Z=1
5	LDAC	LDAC 1	AR ← AC; READ
6		LDAC 2	IDLE
7		LDAC 3	DR ← M(AC)
8		LDAC 4	AC ← DR
9	LDX1R1	LDX1R1 1	AR ← MBRU; READ
10		LDX1R1 2	IDLE
11		LDX1R1 3	DR ← M(X1)
12		LDX1R1 4	AC ← DR
13		LDX1R1 5	R1 ← AC
14	LDX2R2	LDX2R2 1	AR ← MBRU; READ
15		LDX2R2 2	IDLE
16		LDX2R2 3	DR ← M(X2)
17		LDX2R2 4	AC ← DR
18		LDX2R2 5	R2 ← AC
19	LDX3R3	LDX3R3 1	AR ← MBRU; READ

20		LDX3R3 2	IDLE
21		LDX3R3 3	DR ← M(X2)
22		LDX3R3 4	AC ← DR
23		LDX3R3 5	R3 ← AC
24		LDX12R9 1	AR ← MBRU; READ
25		LDX12R9 2	IDLE
26	LDXXR9	LDX12R9 3	DR ← M(X12)
27		LDX12R9 4	AC ← DR
28		LDX12R9 5	R9 ← AC
29		LDX4R6 1	AR ← MBRU;READ
30		LDX4R6 2	IDLE
31	LDX4R6	LDX4R6 3	DR ← M(X4)
32		LDX4R6 4	AC ← DR
33		LDX4R6 5	R6 ← AC
34	SWX5R3	SWX5R3 1	AC←R3
35	SWX12R2	SWX12R2 1	AC←R2
36	SWXXAC	SWX5AC 1	AR← MBRU
37		SWX5AC 2	DR ← AC ;WRITE
38	STAC	STAC	DR ← AC ; WRITE
39	MOVACR1	MOVACR1	R1←AC
40	MOVACR2	MOVACR2	R2←AC
41	MOVACR3	MOVACR3	R3←AC
42	MOVACR4	MOVACR4	R4←AC
43	MOVACR5	MOVACR5	R5←AC
44	MOVACR6	MOVACR6	R6←AC
45	MOVACR7	MOVACR7	R7←AC
46	MOVACR8	MOVACR8	R8←AC
47	MOVACR9	MOVACR9	R9←AC

48	MOVR1AC	MOVR1AC	AC←R1
49	MOVR2AC	MOVR2AC	AC←R2
50	MOVR3AC	MOVR3AC	AC←R3
51	MOVR4AC	MOVR4AC	AC←R4
52	MOVR5AC	MOVR5AC	AC←R5
53	MOVR6AC	MOVR6AC	AC←R6
54	MOVR7AC	MOVR7AC	AC←R7
55	MOVR8AC	MOVR8AC	AC←R8
56	MOVR9AC	MOVR9AC	AC←R9
57	MOVACMAR	MOVACMAR	MAR←AC
58	MOVACMDR	MOVACMDR	MDR ←AC ; WRITE
59	ADDR1	ADDR1	AC←AC+R1
60	ADDR2	ADDR2	AC←AC+R2
61	ADDR4	ADDR4	AC←AC+R4
62	ADDR5	ADDR5	AC←AC+R5
63	ADDR6	ADDR6	AC←AC+R6
64	ADDR7	ADDR7	AC←AC+R7
65	LSHIFT1	LSHIFT1	AC ← AC<<1
66	RSHIFT1	RSHIFT1	AC ← AC>>1
67	LSHIFT8	LSHIFT8	AC ← AC<<8
68	INCREMENTPC	INCREMENTPC	PC←PC+1
69	INCREMENTAC	INCREMENTAC	AC←AC+1
70	DECREMENTAC	DECREMENTAC	AC←AC-1
71	JUMP	JUMP1	AC← MBRU
72		JUMP2	PC←AC
73	JMPZ	JUMPZ1	
74		JMPZN1 (Z = 0)	IDLE
75		JMPZY1 (Z = 1)	AC ← MBRU

76		JMPZY2 (Z = 1)	PC← AC
77	JMPNZ	JMPNZ1	
78		JMPNZY1 (Z = 1)	IDLE
79		JMPNZN1 (Z = 0)	AC ← MBRU
80		JMPNZN2 (Z = 0)	PC← AC
88		JMPNZN2 (Z = 0)	CLAC
81	LDI R2	LDIR2 1	AC← MBRU
82		LDIR2 2	R2 ← AC
83	LDI R3	LDIR3 1	AC ← MBRU
84		LDIR3 2	R3 ← AC
85	LDI R1	LDIR1 1	R1← MBRU
86	LDI R6	LDIR6 1	R6 ← MBRU
87	DONE	DONE	

3.4. Control Store

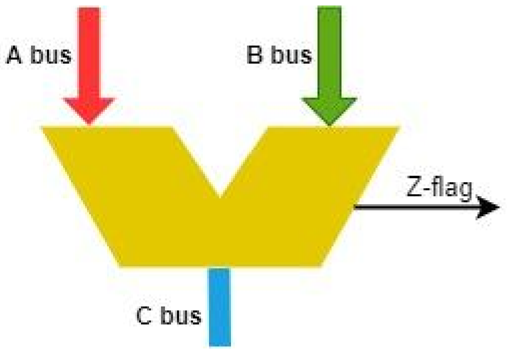
The control store contains all the control signals for each of the micro-instruction of each instruction in the ISA. These control signals are stored in a Look-Up table in the form of a bit pattern. (In our design each of these control signals was 36 bits). The following table gives a great description of the LookUp Table that is implemented in the processor.

Control Lookup table

							ALU (4 bits)				C Bus WRITE Enable														Memory Signals			Incremetation		B bus mux			
Memory address	Instruction	Microinstruction	Break down of microinstruction	Next address	Next address in binary	Jump	3	2	1	0	MAR	MDR	PC	MBRU	R1	R2	R3	R4	R5	R6	R7	R8	R9	Accumulator	FETCH	READ	WRITE	PC	Accumulator	3	2	1	0
0	FETCH	FETCH1	MBRU ← IRAM[PC]; FETCH	1	b00000001	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1		FETCH2	IDLE	2	b00000010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2		FETCH3	PC←PC+1	xxxxxxxxxxxx	xxxxxxxxxx	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
3	NOOP	NOOP	IDLE	0	b00000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	CLAC	CLAC	AC←0,Z=1	0	b00000000	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
5	LDAC	LDAC 1	AR ← AC; READ	6	b00000110	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
6		LDAC 2	IDLE	7	b00000111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7		LDAC 3	DR ← M(AC)	8	b00001000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8		LDAC 4	AC ← DR	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
9	LDX1R1	LDX1R1 1	AR ← MBRU; READ	10	b00001010	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1
10		LDX1R1 2	IDLE	11	b00001011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11		LDX1R1 3	DR ← M(X1)	12	b00001100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12		LDX1R1 4	AC ← DR	13	b00001101	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
13		LDX1R1 5	R1 ← AC	0	b00000000	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	LDX2R2	LDX2R2 1	AR ← MBRU; READ	15	b00001111	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1
15		LDX2R2 2	IDLE	16	b00010000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16		LDX2R2 3	DR ← M(X2)	17	b00010001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17		LDX2R2 4	AC ← DR	18	b00010010	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
18		LDX2R2 5	R2 ← AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	LDX3R3	LDX3R3 1	AR ← MBRU; READ	20	b00010100	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1
20		LDX3R3 2	IDLE	21	b00010101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21		LDX3R3 3	DR ← M(X2)	22	b00010110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22		LDX3R3 4	AC ← DR	23	b00010111	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
23		LDX3R3 5	R3 ← AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	LDXXR9	LDX12R9 1	AR ← MBRU; READ	25	b00011001	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1
25		LDX12R9 2	IDLE	26	b00011010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26		LDX12R9 3	DR ← M(X12)	27	b00011011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27		LDX12R9 4	AC ← DR	28	b00011100	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
28		LDX12R9 5	R9 ← AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
29	LDX4R6	LDX4R6 1	AR ← MBRU;READ	30	b00011110	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1
30		LDX4R6 2	IDLE	31	b00011111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31		LDX4R6 3	DR ← M(X4)	32	b00100000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32		LDX4R6 4	AC ← DR	33	b00100001	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
33		LDX4R6 5	R6 ← AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
34	SWX5R3	SWX5R3 1	AC←R3	36	b00100100	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
35	SWX12R2	SWX12R2 1	AC←R2	36	b00100100	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
36	SWXXAC	SWX5AC 1	AR← MBRU	37	b00100101	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
37		SWX5AC 2	DR ← AC ;WRITE	0	b00000000	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
38	STAC	STAC	DR ← AC ; WRITE	0	b00000000	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
39	MOVACR1	MOVACR1	R1←AC	0	b00000000	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	MOVACR2	MOVACR2	R2←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
41	MOVACR3	MOVACR3	R3←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
42	MOVACR4	MOVACR4	R4←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
43	MOVACR5	MOVACR5	R5←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
44	MOVACR6	MOVACR6	R6←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
45	MOVACR7	MOVACR7	R7←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
46	MOVACR8	MOVACR8	R8←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
47	MOVACR9	MOVACR9	R9←AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
48	MOVR1AC	MOVR1AC	AC←R1	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
49	MOVR2AC	MOVR2AC	AC←R2	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
50	MOVR3AC	MOVR3AC	AC←R3	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
51	MOVR4AC	MOVR4AC	AC←R4	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1
52	MOVR5AC	MOVR5AC	AC←R5	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
53	MOVR6AC	MOVR6AC	AC←R6	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1
54	MOVR7AC	MOVR7AC	AC←R7	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0
55	MOVR8AC	MOVR8AC	AC←R8	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1
56	MOVR9AC	MOVR9AC	AC←R9	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0
57	MOVACMAR	MOVACMAR	MAR←AC	0	b00000000	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
58	MOVACMDR	MOVACMDR	MDR ←AC ; WRITE	0	b00000000	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
59	ADDR1	ADDR1	AC←AC+R1	0	b00000000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
60	ADDR2	ADDR2	AC←AC+R2	0	b00000000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
61	ADDR4	ADDR4	AC←AC+R4	0	b00000000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1
62	ADDR5	ADDR5	AC←AC+R5	0	b00000000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
63	ADDR6	ADDR6	AC←AC+R6	0	b00000000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1
64	ADDR7	ADDR7	AC←AC+R7	0	b00000000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0
65	LSHIFT1	LSHIFT1	AC ← AC<<1	0	b00000000	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
66	RSHIFT1	RSHIFT1	AC ← AC>>1	0	b00000000	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
67	LSHIFT8	LSHIFT8	AC ← AC<<8	0	b00000000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
68	INCREMENTPC	INCREMENTPC	PC←PC+1	0	b00000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
69	INCREMENTAC	INCREMENTAC	AC←AC+1	0	b00000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
70	DECREMENTAC	DECREMENTAC	AC←AC-1	0	b00000000	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
71	JUMP	JUMP1	AC← MBRU	72	b01001000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
72		JUMP2	PC←AC	0	b00000000	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
73	JMPZ	JUMPZ1		XXXXXXXX	xxxxxxxxxx	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
74		JMPZN1 (Z = 0)	IDLE	0	b00000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
75		JMPZY1 (Z = 1)	AC ← MBRU	76	b01001100	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
76		JMPZY2 (Z = 1)	PC← AC	89	b001011001	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
89		JMPZY3 (Z = 0)	AC ← 0	0	b00000000	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
77	JMPNZ	JMPNZ1		XXXXXXXX	xxxxxxxxxx	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
78		JMPNZY1 (Z = 1)	IDLE	0	b00000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
79		JMPNZN1 (Z = 0)	AC ← MBRU	80	b01010000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
80		JMPNZN2 (Z = 0)	PC← AC	88	b01011000	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
88		JMPNZN3 (Z = 0)	AC ← 0	0	b00000000	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
81	LDI R2	LDIR2 1	AC← MBRU	82	b01010010	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
82		LDIR2 2	R2 ← AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
83	LDI R3	LDIR3 1	AC ← MBRU	84	b01010100	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
84		LDIR3 2	R3 ← AC	0	b00000000	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
85	LDI R1	LDIR1 1	R1← MBRU	0	b00000000	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
86	LDI R6	LDIR6 1	R6 ← MBRU	0	b00000000	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1
87	DONE	DONE		z	bzzzzzzzz	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z

3.5. The Arithmetic and Logic Unit (ALU)

The ALU carries out the arithmetic tasks of the processor. ALU has two inputs A bus and B bus and two outputs namely, the C bus and the Z flag. The ALU is also fed by a 4-bit control signal from the control unit which specifies the task the ALU should perform when the



processor is at work. The following table represents the control signals that are fed into the ALU to perform a specific task.

ALU COMMAND	TASK	BIT PATTERN
NONE	None	0000
ADD	C = A+B	0001
LSHIFT1	C=A<<1	0010
RSHIFT1	C=A>>1	0011
LSHIFT8	C=A<<8	0100
DEAC	C=A-1	0101
RESET	C=0	0110
PASSBTOC	C=B	0111
PASSATOC	C=A	1000

3.6. B bus from multiplexer

From the Data path we observe that the B bus multiplexer has multiple inputs from the registers MAR, MDR, PC, MBRU, R1, R2, R3, R4, R5, R6, R7, R8 and R9. But B bus can accept only one input from these registers at a given time. Hence, there should be a multiplexing scheme that ensures the B bus is accessed by strictly one of these registers at a given moment. The following graph represents a breakdown of the selection bits of the multiplexer used to perform this task.

REGISTER	SELECTION BITS
NONE	0000
MAR	0001
MDR	0010
MBRU	0011
R1	0100
R2	0101
R3	0110
R4	0111
R5	1000
R6	1001
R7	1010
R8	1011
R9	1100

3.7. Assembly Code of the Integrated Down-sample Algorithm

In the IRAM of the processor the assembly version of the integrated down-sampling algorithm should be stored. This assembly code consists of the instructions that were defined in the ISA. When the processor is enabled, the Programme Counterpoints to each of these instructions so that the intended task is executed by each instruction. The assembly instructions and the immediate data of those assembly instructions that is stored in the instruction RAM is as stated below.

	HIGH LEVEL CODE	ASSEMBLY CODE	IMMEDIATE DATA	MICRO INSTRUCTIONS
HORIZONTAL CONVOLUTION				

0		FETCH	18’d0	MBRU←IRAM[PC];FETCH
				IDLE
				PC←PC+1

1		CLAC	18’d0	AC ← 0
2	int image[]; // original image	LDIR1	18’d20	R1 ← MBRU ;READ

3	int image_width = 256;	LDIR2	18’d256	AC← MBRU
				R2 ← AC

4	int image_height = 256;	LDIR3	18’d256	AC ← MBRU
				R3 ← AC

5	int height_count = image_height;	SWX5R3	18’d3	AC ←R3
				AR ← MBRU
				DR ← AC ; WRITE

6		CLAC	18’d0	AC ← 0
7	int X =0	MOVACR4	18’d0	R4 ←AC
8	Int y = 0	MOVACR5	18’d0	R5 ←AC
9	Int a = 0	MOVACR6	18’d0	R6 ←AC
10	int b = 2 * image[x*image_width + y];// value b is stored in “R7”	MOVR4AC	18’d0	AC ←R4
11		LSHIFT8	18’d0	AC ← AC<<7
12		ADDR5	18’d0	AC ←AC +R5
13		ADDR1	18’d0	AC ← AC+R1

14		LDAC	18’d0	AR ← AC; READ
				IDLE
				DR ← M(AC)
				AC ← DR

15		LSHIFT1	18’d0	AC ← AC << 1

16		MOVACR7	18’d0	R7←AC
17	int width_count = image_width - 1;	MOVR2AC	18’d0	AC←R2
18		DECREMENTAC	18’d0	AC←AC -1

19		SWX12AC	18’d7	AR← MBRU
				DR ←AC; WRITE

20	int c = image[x*image_width + y + 1]; // c is stored in R8	MOVR4AC	18’d0	AC ←R4
21		LSHIFT8	18’d0	AC ← AC<<7
22		ADDR5	18’d0	AC ←AC +R5
23		INCREMENTAC	18’d0	AC←AC+1
24		ADDR1	18’d0	AC ← AC+R1

25		LDAC	18’d0	AR ← AC; READ
				IDLE
				DR ← M(AC)
				AC ← DR

26		MOVACR8	18’d0	R8←AC
27	int new_pixel = (a + b + c)/4; // new pixel value will be stored in R9	ADDR7	18’d0	AC ← AC+R7
28		ADDR6	18’d0	AC ← AC+R6
29		RSHIFT1	18’d0	AC ← AC >> 1
30		RSHIFT1	18’d0	AC ← AC >> 1
31		MOVACR9	18’d0	R9 ←AC
32	image[x*image_width + y] = new_pixel;	MOVR4AC	18’d0	AC ←R4
33		LSHIFT8	18’d0	AC ← AC<<7
34		ADDR5	18’d0	AC ← AC+R5
35		ADDR1	18’d0	AC ← AC+R1
36		MOVACMAR	18’d0	MAR←AC
37		MOVR9AC	18’d0	AC ← R9

38		MOVACMDR	18’d0	MDR ←AC; WRITE
39	a = b/2;	MOVR7AC	18’d0	AC←R7
40		RSHIFT1	18’d0	AC ← AC >> 1
41		MOVACR6	18’d0	R6←AC
42	b = c*2;	MOVR8AC	18’d0	AC←R8
43		LSHIFT1	18’d0	AC ← AC << 1
44		MOVACR7	18’d0	R7←AC
45	y += 1;	MOVR5AC	18’d0	AC←R5
46		INCREMENTAC	18’d0	AC←AC+1
47		MOVACR5	18’d0	R5←AC

48	width_count -= 1;	LDX12R9	18’d7	MAR ← MBRU ; READ
				IDLE
				DR ← M(X12)
				AC ← DR
				R9 ← AC

49		MOVR9AC	18’d0	AC←R9
50		DECREMENTAC	18’d0	AC←AC-1
51		STAC	18’d0	DR ← AC; WRITE

52	while (width_count > 0): repeat from 20 if the flag is not zero.	JMPNZ	18’d20	AC ←MBRU
				PC ← AC

53	int c = 0; //zero padding(right)
	int new_pixel = (a + b + c)/4;
		CLAC	18’d0	AC←0

54		ADD R7	18’d0	AC ← AC+R7
55		ADD R6	18’d0	AC ← AC+R6
56		RSHIFT1	18’d0	AC ← AC >> 1
57		RSHIFT1	18’d0	AC ← AC >> 1
58		MOVACR9	18’d0	R9 ←AC

59	image[x*image_width + y] = new_pixel;	MOVR4AC	18'd0	AC ←R4
60		LSHIFT8	18'd0	AC ← AC<<7
61		ADDR5	18'd0	AC ← AC+R5
62		ADDR1	18'd0	AC ← AC+R1
63		MOVACMAR	18'd0	MAR←AC
64		MOVR9AC	18'd0	AC ← R9
65		MOVACMDR	18'd0	MDR ←AC WRITE
66	x += 1; // moving to next row	MOVR4AC	18'd0	AC←R4
67		INCREMENTAC	18'd0	AC←AC+1
68		MOV ACR4	18'd0	R4←AC

69	height_count -= 1;	LDX5R9	18'd3	MAR ← MBRU; READ
				IDLE
				DR ← M(X5)
				AC ← DR
				R9 ← AC

70		MOV R9AC	18'd0	AC←R9
71		DECREMENTAC	18'd0	AC←AC-1
72		STAC	18'd0	DR ← AC; WRITE

73	while (height_count > 0): repeat from step 8	JMPNZ	18'd8	AC ← MBRU
				PC ← AC

VERTICAL CONVOLUTION				

74	int width_count = image_width;	SWX12R2	18'd7	AC←R2
				AR← MBRU
				DR ← AC
				WRITE

75	int y = 0;	CLAC	18'd0	AC←0
76		MOVACR5	18'd0	R5←AC

77	int x = 0;	MOVACR4	18’d0	R4←AC
78	int a = 0; //zero padding(top)	MOVACR6	18’d0	R6 ←AC
79	int b = 2*image[x*image_width + y];	MOVR4AC	18’d0	AC ←R4
80		LSHIFT8	18’d0	AC ← AC<<7
81		ADDR5	18’d0	AC ←AC +R5
82		ADDR1	18’d0	AC ← AC+R1

83		LDAC	18’d0	AR ← AC;READ
				IDLE
				DR ← M(AC)
				AC ← DR

84		LSHIFT1	18’d0	AC ← AC << 1
85		MOVACR7	18’d0	R7←AC
86	int height_count = image_height - 1;	MOVR3AC	18’d0	AC←R3
87		DECREMENTAC	18’d0	AC←AC -1

88		SWX5AC	18’d3	AR← MBRU
				DR ← AC
				WRITE

89	int c = image[x*image_width + image_width + y];	MOVR4AC	18’d0	AC ←R4
90		LSHIFT8	18’d0	AC ← AC<<7
91		ADDR5	18’d0	AC ←AC +R5
92		ADDR2	18’d0	AC ←AC+R2
93		ADDR1	18’d0	AC ← AC+R1

94		LDAC	18’d0	AR ← AC; READ
				IDLE
				DR ← M(AC)
				AC ← DR

95		MOVACR8	18’d0	R8←AC
96	int new_pixel = (a + b + c)/4;	ADDR7	18’d0	AC ← AC+R7

97	// new pixel value will be stored in R9	ADDR6	18'd0	AC ← AC+R6
98		RSHIFT1	18'd0	AC ← AC >> 1
99		RSHIFT1	18'd0	AC ← AC >> 1
100		MOVACR9	18'd0	R9 ←AC
101	image[x*image_width + y] = new_pixel;	MOVR4AC	18'd0	AC ←R4
102		LSHIFT8	18'd0	AC ← AC<<8
103		ADDR5	18'd0	AC ← AC+R5
104		ADDR1	18'd0	AC ← AC+R1
105		MOVACMAR	18'd0	MAR←AC
106		MOVR9AC	18'd0	AC ← R9
107		MOVACMDR	18'd0	MDR ←AC WRITE
108	a = b/2;	MOVR7AC	18'd0	AC←R7
109		RSHIFT1	18'd0	AC ← AC >> 1
110		MOVACR6	18'd0	R6←AC
111	b = c*2;	MOVR8AC	18'd0	AC←R8
112		LSHIFT1	18'd0	AC ← AC << 1
113		MOVACR7	18'd0	R7←AC
114	X+=1	MOVR4AC	18'd0	AC←R4
115		INCREMENTAC	18'd0	AC←AC+1
116		MOVACR4	18'd0	R4←AC

117	height_count -= 1;	LDX5R9	18'd0	MAR ← MBRU; READ
				IDLE
				DR ← M(X5)
				AC ← DR
				R9 ← AC

118		MOVR9AC	18’d0	AC←R9
119		DECREMENTAC	18’d0	AC←AC-1
120		STAC	18’d0	DR ← AC WRITE

121	while (height_count > 0):
	repeat from step 89
		JMPNZ	18’d89	AC ← MBRU
				PC ← AC

122	int c = 0; //zero padding(right)
	int new_pixel = (a + b + c)/4;
		CLAC	18’d0	AC←0

123		ADDR7	18’d0	AC ← AC+R7
124		ADDR6	18’d0	AC ← AC+R6
125		RSHIFT1	18’d0	AC ← AC >> 1
126		RSHIFT1	18’d0	AC ← AC >> 1
127		MOVACR9	18’d0	R9 ←AC
128	image[x*image_width + y] = new_pixel;	MOVR4AC	18’d0	AC ←R4
129		LSHIFT8	18’d0	AC ← AC<<7
130		ADDR5	18’d0	AC ← AC+R5
131		ADDR1	18’d0	AC ← AC+R1
132		MOVACMAR	18’d0	MAR←AC
133		MOVR9AC	18’d0	AC ← R9
134		MOVACMDR	18’d0	MDR ←AC WRITE
135	Y += 1; // moving to next row	MOVR5AC	18’d0	AC←R5
136		INCREMENTAC	18’d0	AC←AC+1
137		MOV ACR5	18’d0	R5←AC

138	width_count -= 1;	LD X12R9	18’d7	MAR ← MBRU ;READ
				IDLE
				DR ← M(X12)
				AC ← DR

				R9 ← AC
139		MOVR9AC	18’d0	AC←R9
140		DECREMENTAC	18’d0	AC←AC-1
141		STAC	18’d0	DR ← AC WRITE

142	while (width_count > 0):
	repeat from 77 if the flag is not zero.
		JMPNZ	18’d77	AC ← MBRU
				PC ← AC

DOWNSAMPLING				

143	int downsampledimage[4];
	// base address to store the downsampled image
		LDIR6	18’d70000	AC ← MBRU ;READ
				R6 ← AC

144	height_count = image_height/2;
	// downsampled image_height;
		MOVR3AC	18’d0	AC←R3

145		RSHIFT1	18’d0	AC← AC>>1

146		SWX5AC	18’d3	AR← MBRU
				DR ← AC; WRITE

147	x = 0;	CLAC	18’d0	AC←0
148		MOVACR4	18’d0	R4←AC
149	int y = 0;	MOVACR5	18’d0	R5←AC

150	int width_count = image_width/2;
	// dsimage_width;
		MOVR2AC	18’d0	AC←R2

151		RSHIFT1	18’d0	AC← AC>>1

152		SWX12AC	18’d7	AR← MBRU
				DR ← AC; WRITE

153	int pixel_value = image[2*y*image_width + 2*x];	MOVR5AC	18’d0	AC←R5
154		LSHIFT8	18’d0	AC <~ AC<<8
155		ADDR4	18’d0	AC ← AC +R4
156		LSHIFT1	18’d0	AC ← AC << 1
157		ADDR1	18’d0	AC ← AC +R1

158		LDAC	18’d0	AR ← AC; READ
				IDLE
				DR ← M(AC)
				AC ← DR

159		MOVACR9	18’d0	R9←AC
160	downsampledimage[x*(image_width/2) + y] = pixel_value;	MOVR6AC	18’d0	AC←R6
161		MOVACMAR	18’d0	MAR←AC
162		INCREMENTAC	18’d0	AC ← AC+1
163		MOVACR6	18’d0	R6 ← AC
164		MOVR9AC	18’d0	AC ← R9
165		MOVACMDR	18’d0	MDR ←AC WRITE
166	y += 1; // moving to next pixel	MOVR5AC	18’d0	AC←R5
167		INCREMENTAC	18’d0	AC←AC+1
168		MOVACR5	18’d0	R5←AC

169	width_count -= 1;	LDX12R9	18’d7	MAR ← MBRU;READ
				IDLE
				DR ← M(X12)
				AC ← DR
				R9 ← AC

170		MOVR9AC	18’d0	AC←R9
171		DECREMENTAC	18’d0	AC←AC-1
172		STAC	18’d0	DR ← AC WRITE

173	while (width_count > 0): repeat from step 153	JMPNZ	18’d153	AC ← IM(t)
				PC ← AC
				PC ←PC+1

174	x += 1; // moving to next row	MOVR4AC	18’d0	AC←R4
175		INCREMENTAC	18’d0	AC←AC+1

176		MOVACR4	18’d0	R4←AC

177	height_count -= 1;	LDX5R9	18’d3	MAR ← MBRU; READ
				IDLE
				DR ← M(X5)
				AC ← DR
				R9 ← AC

178		MOVR9AC	18’d0	AC←R9
179		DECREMENTAC	18’d0	AC←AC-1
180		STAC	18’d0	DR ← AC; WRITE

181	while (width_count > 0):
	repeat from step 149
		JMPNZ	18’d149	AC ← MBRU
				PC ← AC

182		NOOP	18’d0	IDLE
183	Complete =1	DONE	18’d0	DONE

3.8. Instruction Cycle

The instruction cycle is the basic operating mechanism of a computer. The cycle consists of three stages.

1. **FETCH** - Retrieving the relevant instruction from a specific memory location.
2. **DECODE** - Understanding the mechanism the fetched instruction invokes.
3. **EXECUTE** - Carrying out the required steps.

This cycle continues repeatedly inside the Central Processing Unit until all the instructions are successfully executed. Usually, in a simple processor, these steps occur sequentially, finishing one Execution before the next Fetch occurs. Each of the instructions is described below.

- **FETCH** - The current address in the program counter (PC) points to the address of the next instruction to be fetched. This instruction is fetched from Instruction RAM and stored in the Memory Buffer Register Unit (MBRU). At the end of each FETCH cycle, the PC is incremented by one representing the next instruction to be fetched.

FETCH instruction executes in three steps.

FETCH 1: MBRU ← IRAM[PC]; FETCH
FETCH 2: IDLE
FETCH 3: PC ← PC + 1

- **DECODE** - The processor has to differentiate between the instructions that are being fetched to correctly call the correct execution cycle. The MBRU gives the fetched instruction to the Control Store and the control store outputs the corresponding control signal.

- **EXECUTE** - Brief explanations of the instructions used in the processor is given below.

1. **NOOP**

   No operation performed. This operation is utilised when a waiting cycle is required in order to have some processed data available at an endpoint.

2. **CLAC**

   Zero is assigned to the AC. (AC is cleared.) The zero flag is indicated. The next FETCH cycle begins afterwards.

   CLAC: AC ← 0; Z ← 1

3. **JUMP**

   The address we need to jump to is given as immediate data with the instruction. Therefore it is stored in the MBRU. That address is then loaded to AC and then copied to PC. Therefore in the next fetch cycle, the fetched instruction will be at the specified location on the PC.

   JUMP1: AC ← MBRU
   JUMP2: PC ← AC

4. **JUMPZ**

   In this case the zero flag is checked and if Z=0 (i.e. AC is not zero), then no jumping occurs. So we give an IDLE statement as our PC is already pointing to the next instruction from the FETCH 3. In the case of Z=1, the two states similar to that of JUMP instruction proceeds. Then we clear the Accumulator because in our processor after the jumping happens we need AC =0 to continue with the algorithm. At the end the next FETCH cycle will occur at the branched location.

JMPZN1 (Z=0) : IDLE

JMPZY1 (Z=1) : AC ← MBRU
JMPZY2 (Z=1) : PC ← AC
JMPZY3 (Z=1) : AC ← 0

5. <u>JUMPNZ</u>

In this case the zero flag is checked and if Z=1 (i.e. AC is zero), then no jumping occurs. So we give an IDLE statement as our PC is already pointing to the next instruction from the FETCH 3. In the case of Z=0, the two states similar to that of JUMP instruction proceeds. Then we clear the Accumulator because in our processor after the jumping happens we need AC =0 to continue with the algorithm. In the end the next FETCH cycle will occur at the branched location.

JMPNZY1 (Z=1) : IDLE

JMPNZN1 (Z=0) : AC ← MBRU
JMPNZN2 (Z=0) : PC ← AC
JMPNZN3 (Z=0) : AC ← 0

6. <u>LDAC</u>

The memory location in the AC is loaded to the MAR and the read signal is given to DRAM. Then data in the DRAM with the location pointed by the current address in the MAR is read and loaded to MDR. Then that data is loaded to AC. The next FETCH cycle will commence afterwards.

LDAC 1: AR ← AC; READ
LDAC2: IDLE
LDAC3: MDR ← DRAM[MAR]
LDAC4: AC ← MDR

7. **<u>LDIR1, LDIR2, LDIR3, LDIR6</u>**

Immediate data given with the instruction is loaded to the relevant register from the MBRU.

LDIR1 1: R1← MBRU

LDIR2 1: AC← MBRU
LDIR2 2: R2 ← AC

LDIR3 1: AC← MBRU
LDIR3 2: R2 ← AC

LDIR6 1: R6← MBRU

8. **<u>LDXXR9, LDX4R6</u>**

We give the required address to load data from DRAM as immediate data with the instruction. Therefore first we need to load the address in the MBRU to MAR. Then the read signal to DRAM is given. Then data in the DRAM with the location pointed by the current address in the MAR is read and loaded to MDR. Then that data is loaded to AC. Then it is loaded to the required register.

LDX4R6 1: MAR ← MBRU;READ
LDX4R6 2: IDLE
LDX4R6 3: MDR ← DRAM(X4)
LDX4R6 4: AC ← MDR
LDX4R6 5: R6 ← AC

LDXXR9 1: MAR ← MBRU;READ
LDXXR9 2: IDLE
LDXXR9 3: MDR ← DRAM(XX)
LDXXR9 4: AC ← MDR
LDXXR9 5: R6 ← AC

9. <u>STAC</u>

The data in AC is loaded to MDR. Then data is written to the location in DRAM provided by the address in MAR. (In all the operations in our processor, the relevant address has been already loaded to MAR before every STAC operation. Therefore don't need to put the address into MAR in STAC operation as well) Then the next FETCH cycle begins.

STAC: MDR ← AC; WRITE

10. <u>SWXXAC</u>

The data in AC is loaded to MDR. Then data is written to the location in DRAM provided by the address in MAR. The address needs to be given as immediate data with the instruction and then it is loaded to MBRU and then to MAR.

SWXXAC 1: AR ← MBRU ;
SWXXAC 2: DR ← AC; WRITE

11. <u>SWX5R3, SWX12R2</u>

The data in R2, and R3 are loaded to MDR. Then data is written to the location in DRAM provided by the address in MAR. The address needs to be given as immediate data with the instruction and then it is loaded to MBRU and then to MAR.
Here instead of using separate micro instructions for step 2 and 3 of SWX5R3 and SWX12R2 instructions, we reuse the same micro instructions of SWXXAC to reduce the size of the control store.

SWX5R3 1: AC ← R3 ;
SWX5R3 2 (SWXXAC 1): AR ← MBRU ;
SWX5R3 3 (SWXXAC 2): DR ← AC ;WRITE

SWX12R2 1: AC ← R3 ;
SWX12R2 2 (SWXXAC 1): AR ← MBRU ;
SWX12R2 3 (SWXXAC 2): DR ← AC ;WRITE

12. **<u>MOVACR1, MOVACR2, MOVACR3, MOVACR4, MOVACR5, MOVACR6, MOVACR7, MOVACR8, MOVACR9, MOVACMDR, MOVACMAR</u>**

Moving the value in the AC to the specified register is the basic operation of all of the above Instructions. The next FETCH cycle begins after each of these single-state instructions.

MOVACMAR: MAR ← AC

MOVACMDR: MDR ← AC

MOVACR1 : R1 ← AC

MOVACR2 : R2 ← AC

MOVACR3 : R3 ← AC

…………………………

MOVACR9 : R9 ← AC

13. **<u>MOVR1AC, MOVR2AC, MOVR3AC, MOVR4AC, MOVR5AC, MOVR6AC, MOVR7AC, MOVR8AC, MOVR9AC, MOVMDRAC</u>**

Moving the value stored in the different registers to AC (Accumulator) is the basic function in these instructions. The next FETCH cycle begins after each of these single-state instructions.

MOVR1AC : AC ← R1

MOVR2AC : AC ← R2

…………………………

MOVR9AC : AC ← R9

MOVMDRAC: AC ← MDR

14. **<u>ADDR1, ADDR2, ADDR4, ADDR5, ADDR6, ADDR7</u>**

In all these instructions, the value in the specified register is added to the value in the AC currently and loaded to AC itself. The next FETCH cycle begins after this single state operation.

ADDR1: AC ← AC + R1

ADDR2: AC ← AC + R2

…………………………

ADDR7: AC ← AC + R7

15. LSHIFT1, RSHIFT1, LSHIFT8

Here shifting the value in AC occurs. In LSHIFT1 we shift the value in the AC register to the left by one bit. This is equivalent to multiplying by 2. In RSHIFT1 we shift the value in the AC register to the right by one bit. This is equivalent to dividing by 2. In LSHIFT8 we shift the value in the AC register to the left by one bit. This is equivalent to multiplying by 256.

LSHIFT1: AC ← AC<<1
LSHIFT8: AC ← AC<<8
RSHIFT1: AC ← AC >>1

16. INCREMENTPC

Incrementing the value in PC by one occurs in INCREMENTPC instruction. This is done through the program counter, not the ALU.

INCREMENTPC: PC ← PC + 1

17. INCREMENTAC, DECREMENTAC

Incrementing the value in AC by one occurs in INCREMENTAC instruction. Decrementing the value in AC by one occurs in DECREMENTAC instruction. The next FETCH cycle begins after this single state instruction.

INCREMENTAC: AC ← AC + 1
DECREMENTAC: AC ← AC - 1 ; IF AC == 0 THEN Z = 1 ELSE Z = 0

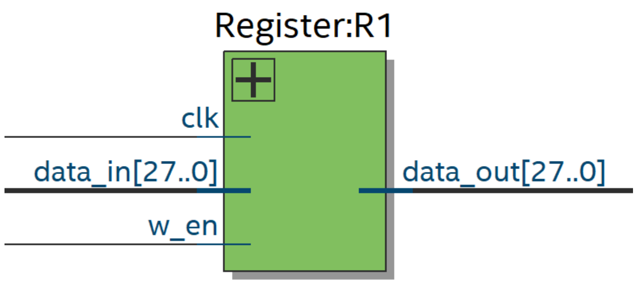
18. DONE

This instruction is given at the end of all the operations to stop the processor from running. Then the complete bit becomes 1.

4. RTL Modules

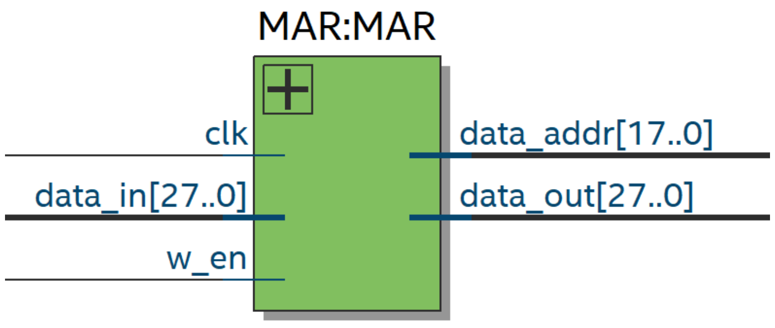
4.1. 28 -Bit General Purpose Registers (GPR)

The purpose of this module is to store intermediate values which are required for the functioning of our processor. It is 28-bit wide which is sufficient to hold values to feed and be stored from the main communication busses. It has a 28-bit wide data in port with a dedicated write enable signal. To output the data held by the register it has a 28bit data out port.



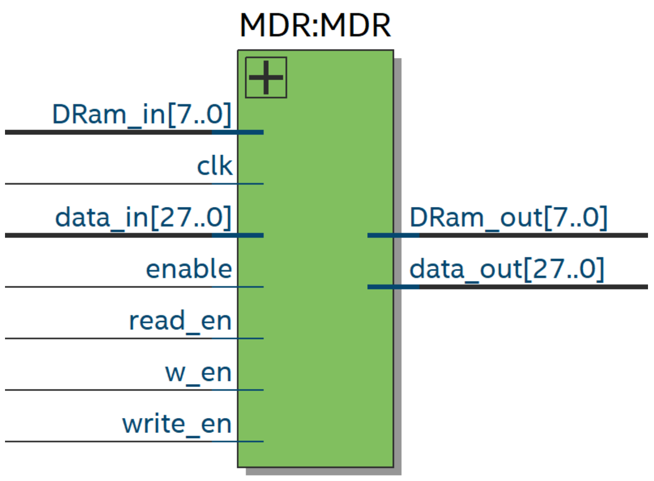
4.2. 18 - Bit Memory Address Register (MAR)

This module is used to store 18bit wide addresses to address the data memory. It has a 28-bit wide data in port with a write enable signal to store data. Also, it has a 28-bit data out port to feed the stored data. We sliced the 28-bit stored value so that this module outputs 18 bits to communicate with our Data Ram.



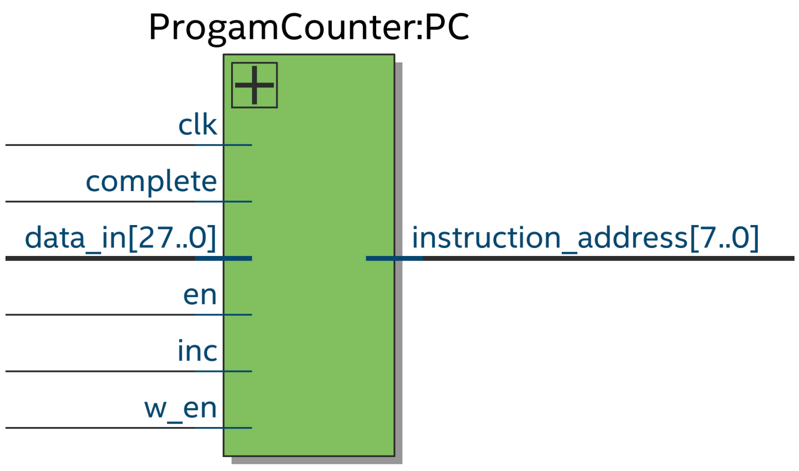
4.3. 28 - Bit Memory Data Register (MDR)

The main purpose of this module is to hold the 8-bit data which is read or written into the DRAM. This module has a master enable signal to enable its operations. It has dedicated 8-bit DRAM input and output ports to communicate with the DRAM along with reading enable and write enable control signals. Also, it has 28-bit data in port with write enable signals to write data from the input bus and a dedicated data out port to feed the data to the data bus.



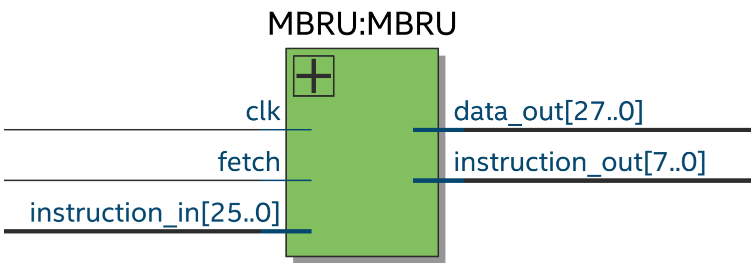
4.4. 8 - bit Program Counter (PC)

The main purpose of this module is to store the instruction address which will point to our instruction stored in the instruction ROM. It has master enable and complete signals to control its operation. Also, it has a 28-bit data in port with a write enable signal to store data from the input data bus. We have included dedicated increment hardware so that the value can easily be incremented within the Program Counter. It has an 8-bit instruction address port which will point to the next instruction within the Instruction ROM.



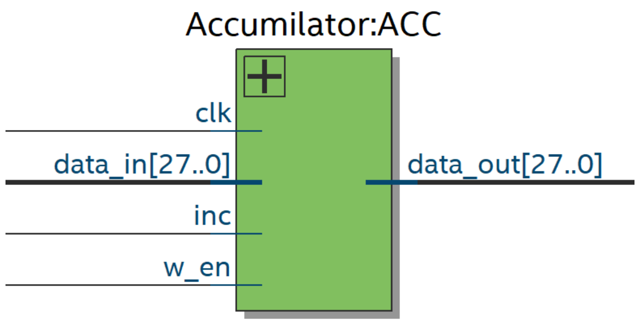
4.5. 28 - Bit Memory Buffer Register Unit (MBRU)

The main purpose of this module is to store the address of the main microinstruction which is used by the control unit. It has a 26-bit instruction input port along with the fetch control signal. This port will store the next instruction fed by the I ROM. We have 2 output ports in this module. The instruction out port is an 8-bit port which will be connected to the Control Unit holding the address of the microinstruction. The data out port is a 28bit wide port connected to the data bus which will output the immediate data within the instruction.



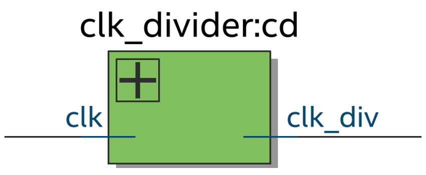
4.6. 28 - Bit Accumulator (AC)

The accumulator is a special data register which has the internal increment capability. This incrementation is controlled by the inc control signal. It has a 28bit wide data in port to store data from the data bus along with a write enable signal. Also, it has a 28-bit data out port to feed the data bus.



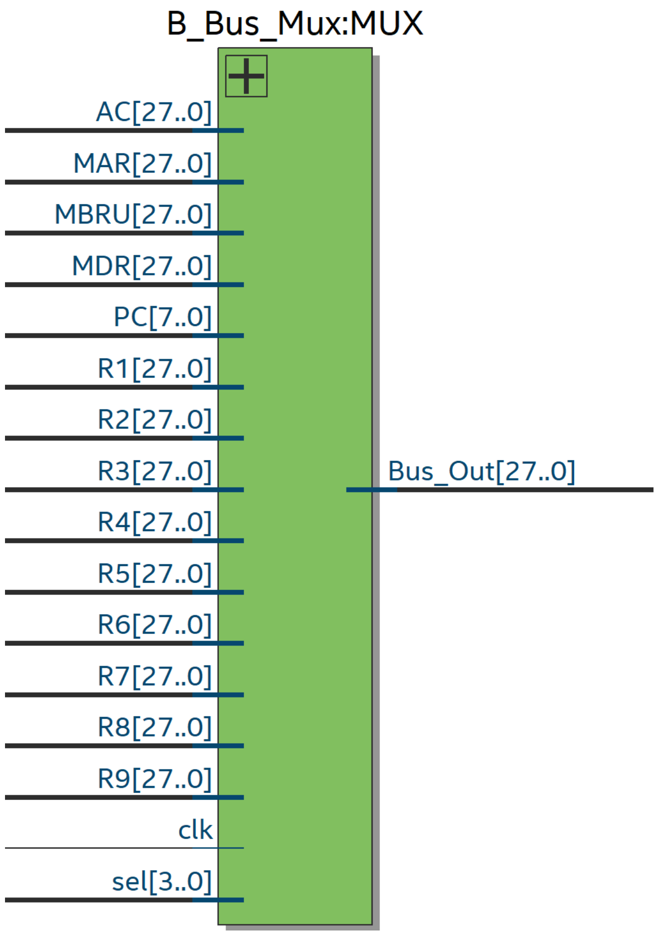
4.7. Clock Generator

The main purpose of the clock divider is to slow down the main clock by a factor of 8 so that the Control Unit signals are kept stable for the other modules to function properly. This has a clk in port which takes the fast clock and returns the slowed clock clk_div port.



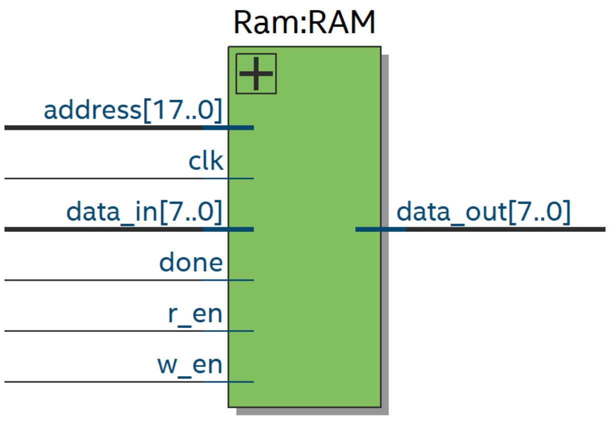
4.8. B bus Multiplexer

This multiplexer is used to select the required register output from 14 different registers and feed it to the B input of the ALU. It has 14 input ports to get the data from all 14 registers along with a 4-bit control signal to select the output. It has a 28-bit data output port to feed the B input bus of the ALU.



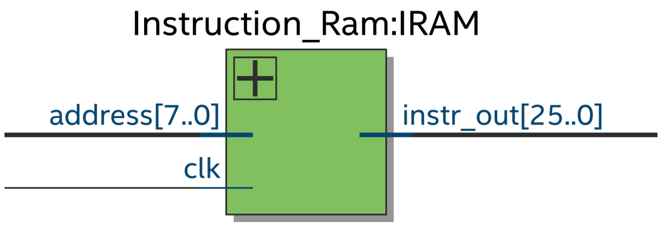
4.9. Data Random Access Memory

This module is used to store the image data required for processing. Since we are storing a 256 x 256 image we have used 18-bit addressing which gives us enough memory depth. In each memory address, we store an 8-bit value which will record the pixel value. This module takes an 18-bit address using the address port and depending on the control signals read enable and write enable will write the value in the 8-bit data in port to the specified memory address or output the data using the 8-bit data out port.



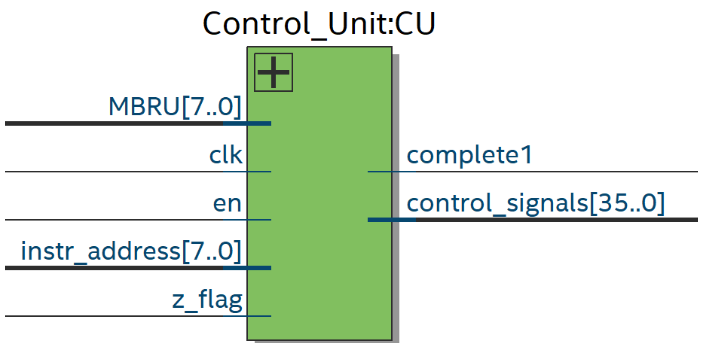
4.10. Instruction Read-Only Memory

This module is used to store the instructions required to run our algorithm. It has an 8-bit address span to store the required instructions. It will take an 8-bit address using the address port and in each clock cycle output the specific 26-bit instruction using the instruction out port.

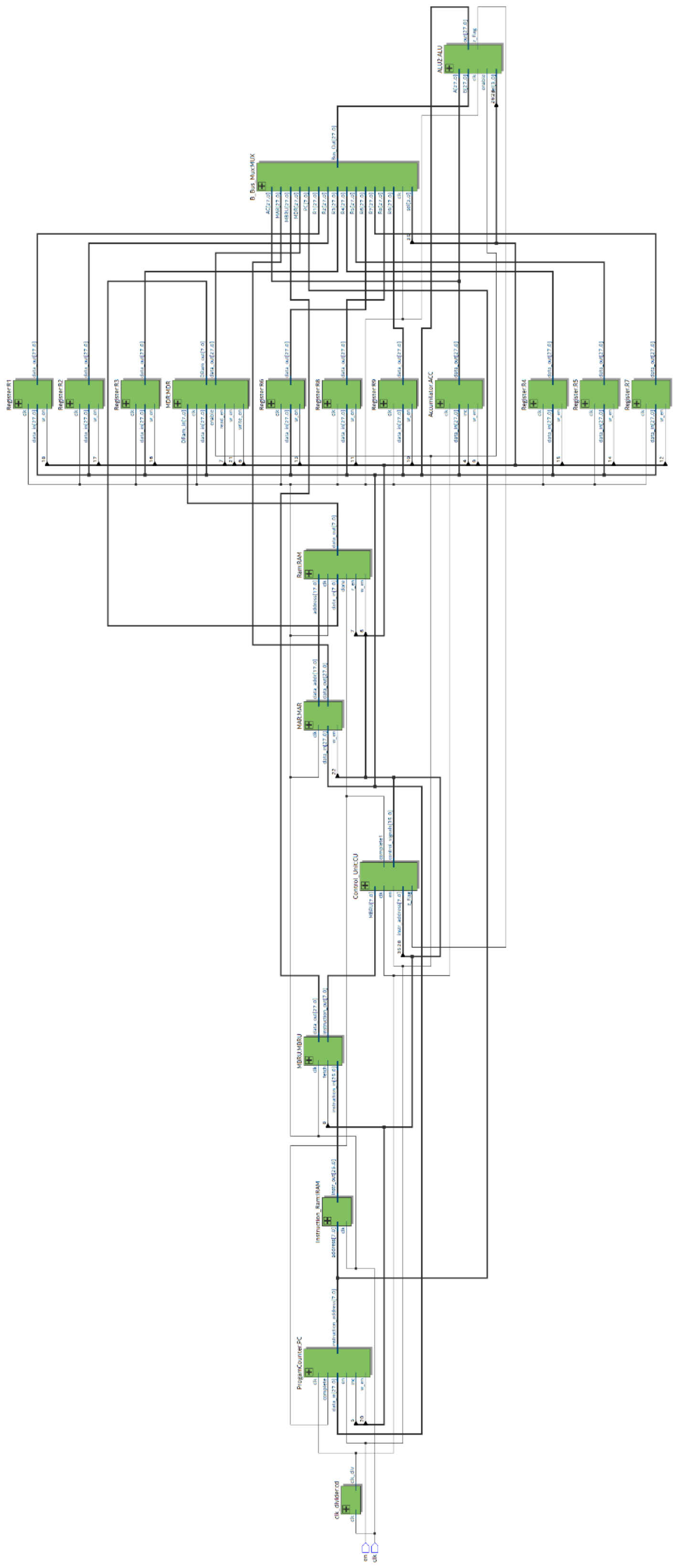


4.11. Control Unit

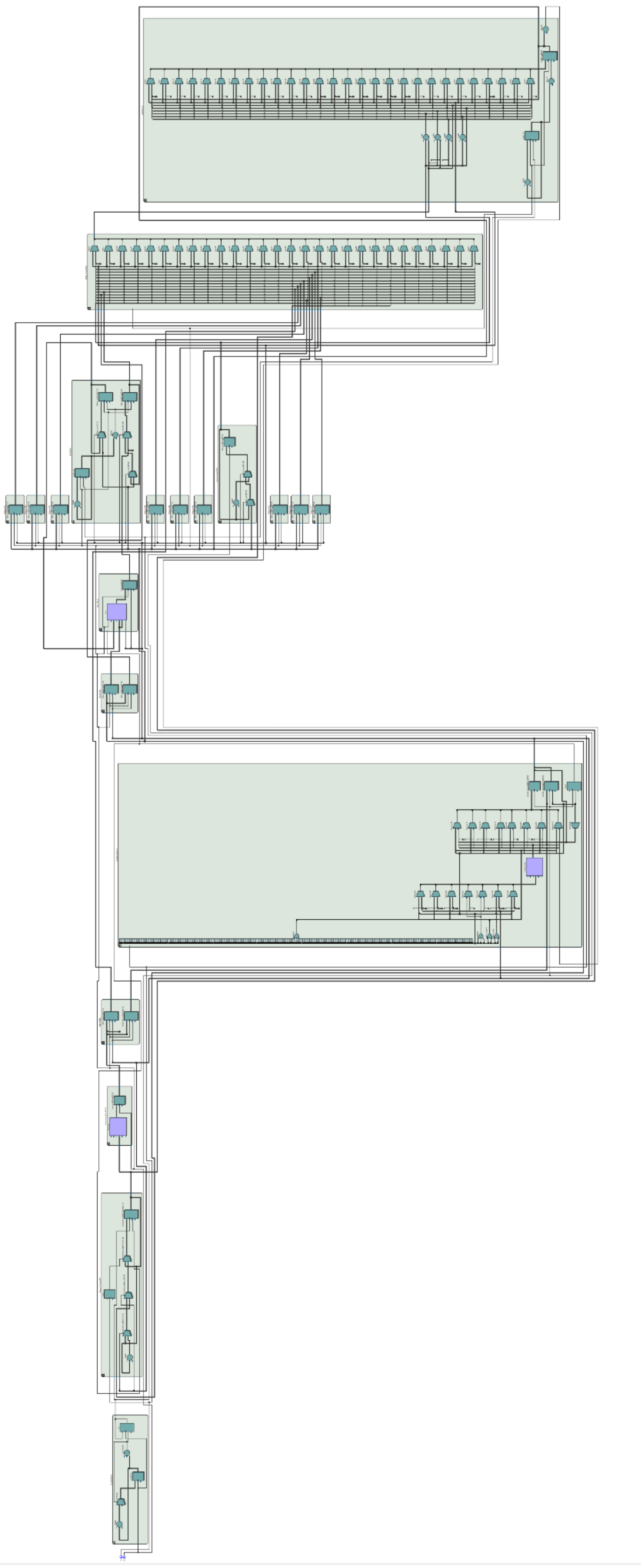
This module is used to release the specific control signals for each microinstruction. It has a master enable and compete port to indicate the start and complete status of the processor. It takes 2 8 bit inputs which are pointers to the next micro address. The MBRU port will point to main instructions addresses which come from the instruction ram and the instruction address port point to the internal microinstructions. In each cycle, this module will output a 36-bit control signal which will control the whole processor for each state.



4.12. Processor



4.13. Expanded RTL view



5. Testing, Simulation and Modifications

After the algorithm was designed, they were tested using a python script. The relevant codes can be found in Appendix A.

After the ISA and the assembly code were written, it was tested by the virtual processor we coded in Verilog. The relevant codes can be found in Appendix B.
After the processor was implemented in Verilog, we tested its function against an 8 by 8 image and checked whether its results were correct.

6. Error Analyzing and Verification

As a metric of performance analysis of the processor, the image obtained from the processor $f_1(x,y)$ and the image that was obtained from running the python script $f_2(x, y)$ was compared at the end. The difference between the results was evaluated using the Mean Squared Error (MSD). Here, the mean of the squares of pixel-wise difference was obtained for each comparison. The mathematical expression of the mentioned evaluation matric MSE is given below. L in that equation represents the length of a side of the resultant image (in this case 128).

$$MSE = \frac{1}{L^2} * \sum_{x=0}^{L-1} \sum_{y=0}^{L-1} (f_1(x,y) - f_2(x, y))^2$$

In the evaluations it was observed that this error always becomes zero due to the design decisions that were taken during the algorithm implementation stage. Basically, this is due to the bitwise shift operations that were used instead of arithmetic multiplication and division operations.

7. Discussion

In this project, we have successfully implemented a custom FPGA processor to perform image downsampling by factor two. We were able to get the correct output successfully without errors as we only used left shift and right shift operations for multiplication and division. We are planning to implement this on the FPGA board in future. The compiling instructions and initializing process of the processor are done automatically by the compiler and the Verilog program. Finally, we can conclude that our processor can give accurate results in an optimised running time with more performance increases.

8. Appendices

8.1. Appendix A - Python script

8.1.1. Downsampling the original image using the algorithms

```
# downsampling using the python algorithm implemented by team

# reading the image
print("Reading the image...")
file = '/lenna.jpg'
image = cv.imread(file, cv.IMREAD_GRAYSCALE)

# converting to a 1D array
image = image.flatten()

# properties of the original image
image_width  = 256; # width of the image
image_height = 256; # height of the image

# properties of the downsampled image
dsimage_width  = 128; # width of the downsampled image
dsimage_height = 128; # height of the downsampled image

# downsampled image
downsampledimage = [0] * dsimage_width * dsimage_height;
# variable to store downsampled image

# Horizontal Convolution
print("Horizontal Convolution...")
height_count = image_height;
x = 0
while (height_count > 0):
        y = 0
        a = 0 # zero padding(left)
        b = 2 * image[x*image_width + y]
        width_count = image_width - 1

        while (width_count > 0):
                c = image[x*image_width + y + 1]
                new_pixel = (a + b + c)/4
                image[x*image_width + y] = (new_pixel);
                # sliding window
                a = b/2
                b = c*2
                y += 1; # moving to next pixel
                width_count -= 1

        c = 0; # zero padding(right)
        new_pixel = (a + b + c)/4
        image[x*image_width + y] = (new_pixel)
        x += 1; # moving to next row
        height_count -= 1

# Vertical Convolution
print("Vertical Convolution...")
width_count = image_width
y = 0
while (width_count > 0):
        x = 0
        a = 0 # zero padding(top)
        b = 2*image[x*image_width + y]
        height_count = image_height - 1

        while (height_count > 0):
                c = image[x*image_width + image_width + y]
                new_pixel = (a + b + c)/4
                image[x*image_width + y] = (new_pixel)
                # sliding window
                a = b/2
                b = c*2
                x += 1 # moving to next pixel
                height_count -= 1;

        c = 0 # zero padding(bottom)
        new_pixel = (a + b + c)/4
        image[x*image_width + y] = (new_pixel)
        y += 1 #  moving to next column
        width_count -= 1


#  downsampling
print("Downsampling...")
height_count = image_height/2 #  dsimage_height;
x = 0
while (height_count > 0):
        y = 0
        width_count = image_width/2; #  dsimage_width

        while (width_count > 0):
                pixel_value = image[2 * (y*image_width + x)]
                downsampledimage[int(x* (image_width/2) + y)] =
(pixel_value)
                y += 1; #  moving to next pixel
                width_count -= 1;

        x += 1; #  moving to next  row
        height_count -= 1

# convert the 1D array to 2D image for visualization
print("Visualizing...")
dsimage_algo = np.zeros((dsimage_width, dsimage_height))
pixel_location = 0
for row in range(dsimage_height):
    for col in range(dsimage_width):
        pixel_val =  (downsampledimage[pixel_location])
        # print(pixel_val)
        dsimage_algo[col, row] = pixel_val

        pixel_location += 1
```

```
# visualise the image
fig, ax = plt.subplots()
ax.imshow(dsimage_algo, cmap='gray', vmin = 0, vmax = 255)
plt.show()
```

8.1.2. Python Script to Generate the binary values of an image to feed into Vivado simulator

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

# read and display the image
file = '/lenna.jpg'
image = cv.imread(file, cv.IMREAD_GRAYSCALE)
fig, ax = plt.subplots()
ax.imshow(image, cmap='gray', vmin = 0, vmax = 255)
plt.show()

# write the binary representation of the image pixels to a txt file
to replace data in Ram.v
ram_index = 20 # we will be storing the image starting from the
address 20 in our Data RAM
with open('/'+file.split('.')[0] + '.txt', 'w') as imgfile:
    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            # print(image[row, col])
            imgfile.write("ram[{}] = 8'b{:08b};\n".format(ram_index,
image[row, col]))
            ram_index += 1
```

8.1.3. Visualize the output generated by the Vivado Simulation

```
# read the text file output from the vivado and reconstruct the image
import numpy as np
import matplotlib.pyplot as plt

dsimagesize = 128 # size of the downsampled image
dsimage_vivado = np.zeros((dsimagesize, dsimagesize))
with open('/content/clock_tower256-processor_output.txt', 'r') as
imgfile:
    lines = imgfile.readlines()
    line = 0
    for row in range(dsimagesize):
        for col in range(dsimagesize):
            pixel_val =  int(lines[line].split('\n')[0][-8:], 2)
            # print(pixel_val)
            dsimage_vivado[col, row] = pixel_val
            line += 1

# visualise the image
fig, ax = plt.subplots()
ax.imshow(dsimage_vivado, cmap='gray', vmin = 0, vmax = 255)
plt.show()
```

8.1.4. Error calculation

```
# Mean Squared Error between the generated images by two methods
mse = (np.square(dsimage_algo - dsimage_vivado)).mean(axis=None)
print("Mean Squared Error = {0:.5f}".format(mse))
```

8.2. Appendix B - Verilog Scripts

8.2.1. Processor

```
`timescale 1ns / 1ps

module Processor(
    input en,
    input clk
    );

    // used in Instruction_Ram
    wire [7:0] instruction_address;
    wire [25:0] instruction_out_memory;

    // used in Control Unit
    wire z_flag,complete;
    wire [7:0] instruction_out_MBRU;
    wire [35:0] control_signals;
    // USED IN MDR
    wire [7:0] DRam_in;
    wire [17:0] data_addr;
    wire [7:0]  DRam_out;
    // used in MBRU
    wire [27:0] mbru_out;

    // used in MAR
    wire [27:0] mar_out;
    wire [27:0] B_bus;
    wire [27:0] C_bus;

    wire [27:0] R1_bus;
    wire [27:0] R2_bus;
    wire [27:0] R3_bus;
    wire [27:0] R4_bus;
    wire [27:0] R5_bus;
    wire [27:0] R6_bus;
    wire [27:0] R7_bus;
    wire [27:0] R8_bus;
    wire [27:0] R9_bus;
    // used in Accumulator
    wire [27:0] A_bus;

    // used in mdr
    wire [27:0] mdr_out;

    // clock divider
    wire clk_div;
    Control_Unit CU(
        .clk(clk_div),
        .en(en),
        .z_flag(z_flag),
        .instr_address(control_signals[35:28]), // from bits 35 to
28(inclusive)
        .MBRU(instruction_out_MBRU),
        .control_signals(control_signals),
        .complete1(complete)
    );

    Instruction_Ram IRAM(
        .clk(clk),      // input clk
        .address(instruction_address),// to be initialized [7:0] address
        .instr_out(instruction_out_memory)//to be initialized reg [25:0]
instr_out
    );

    MBRU MBRU(
        .clk(clk),
        .fetch(control_signals[8]),
        .instruction_in(instruction_out_memory),
        .instruction_out(instruction_out_MBRU),
        .data_out(mbru_out)
    );

    MAR MAR(
        .clk(clk),
        .w_en(control_signals[22]),
        .data_out(mar_out),
        .data_in(C_bus),
        .data_addr(data_addr)
    );

    MDR MDR(
        .clk(clk),
        .enable(en),
        .w_en(control_signals[21]),
        .write_en(control_signals[6]),
        .read_en(control_signals[7]),
        .data_out(mdr_out),
        .data_in(C_bus),
        .DRam_in(DRam_in),
        .DRam_out(DRam_out)
    );

    Accumilator ACC(
        .clk(clk_div), //added clk divider otherwise 4 operations in
1 clock cycle
        .data_in(C_bus),
        .data_out(A_bus),
        .inc(control_signals[4]),
        .w_en(control_signals[9])
    );

    ALU2 ALU(
        .enable(en),
        .clk(clk),   //added clk divider otherwise 4 operations in 1
clock cycle
        .A(A_bus),
        .B(B_bus),
    .out(C_bus),
    .z_flag(z_flag),
    .sel(control_signals[26:23])
);

Register R1(
    .clk(clk),
    .w_en(control_signals[18]),
    .data_in(C_bus),
    .data_out(R1_bus)
);

Register R2(
    .clk(clk),
    .w_en(control_signals[17]),
    .data_in(C_bus),
    .data_out(R2_bus)
);

Register R3(
    .clk(clk),
    .w_en(control_signals[16]),
    .data_in(C_bus),
    .data_out(R3_bus)
);

Register R4(
    .clk(clk),
    .w_en(control_signals[15]),
    .data_in(C_bus),
    .data_out(R4_bus)
);

Register R5(
    .clk(clk),
    .w_en(control_signals[14]),
    .data_in(C_bus),
    .data_out(R5_bus)
);

Register R6(
    .clk(clk),
    .w_en(control_signals[13]),
    .data_in(C_bus),
    .data_out(R6_bus)
);

Register R7(
    .clk(clk),
    .w_en(control_signals[12]),
    .data_in(C_bus),
    .data_out(R7_bus)
);

Register R8(
    .clk(clk),
    .w_en(control_signals[11]),
    .data_in(C_bus),
    .data_out(R8_bus)
);
```

```
    Register R9(
        .clk(clk),
        .w_en(control_signals[10]),
        .data_in(C_bus),
        .data_out(R9_bus)
    );

    ProgamCounter PC(
        .clk(clk_div),
        .en(en),
        .w_en(control_signals[20]),
        .complete(complete),
        .inc(control_signals[5]),
        .data_in(C_bus),
        .instruction_address(instruction_address)
    );

    B_Bus_Mux MUX(
        .clk(clk),
        .sel(control_signals[3:0]),
        .AC(A_bus),
        .MAR(mar_out),
        .MDR(mdr_out),
        .PC(instruction_address),
        .MBRU(mbru_out),
        .R1(R1_bus),
        .R2(R2_bus),
        .R3(R3_bus),
        .R4(R4_bus),
        .R5(R5_bus),
        .R6(R6_bus),
        .R7(R7_bus),
        .R8(R8_bus),
        .R9(R9_bus),
        .Bus_Out(B_bus)
    );

    Ram RAM(
        .clk(clk),
        .w_en(control_signals[6]),
        .r_en(control_signals[7]),
        .done(complete),
        .address(data_addr),
        .data_in(DRam_out), //maximum value is 256 (8 bits)
        .data_out(DRam_in)
    );

    clk_divider cd(
        .clk(clk),
        .clk_div(clk_div)
        );
endmodule
```

8.2.2. Control Unit

```
`timescale 1ns / 1ps

module Control_Unit(
    input clk,
    input en,
    input z_flag,
    input [7:0] instr_address,
    input [7:0] MBRU,
    output reg [35:0] control_signals,
    output complete1
    );

    parameter  control_signal_length=35; //36-1=35

    reg [control_signal_length:0] control_store[88:0];

    reg finish = 1'b0;

    //instructions which jumps
    parameter  FETCH3=8'd2;

    parameter  JUMPZ1=8'd73;
    parameter  JMPZN1=8'd74;
    parameter  JMPZY1=8'd75;

    parameter  JMPNZ1=8'd77;
    parameter  JMPNZY1=8'd78;
    parameter  JMPNZN1=8'd79;


    parameter  NOOP=8'd3;
    parameter  DONE=8'd87;

    assign complete1 = finish;

    initial
      begin
        control_signals=
36'b00000000_0_0000_00000000000000_000_00_0000; // Start
      end


    always@(posedge clk)
        begin
            case(control_signals[35:28])
            FETCH3:
                begin

control_signals={MBRU,control_store[FETCH3][27:0]};
                end
            JUMPZ1:
                begin
                    if (z_flag==1'b0)

control_signals={JMPZN1,control_store[JUMPZ1][27:0]};
                    else

control_signals={JMPZY1,control_store[JUMPZ1][27:0]};
                  end
            JMPNZ1:
                begin
                    if (z_flag==1'b1)

control_signals={JMPNZY1,control_store[JMPNZ1][27:0]};
                    else
```

```
control_signals={JMPNZN1,control_store[JMPNZ1][27:0]};
                end

             DONE: finish = 1'b1;   //COMPLETE COMMAND

             default: control_signals=control_store[instr_address];
            endcase
         end

    initial
    begin

        control_store[0]  =
36'b00000001_0_0000_00010000000000_100_00_0000; //  FETCH1
        control_store[1]  =
36'b00000010_0_0000_00000000000000_000_00_0000; //  FETCH2
        control_store[2]  =
36'bxxxxxxxx_0_0000_00000000000000_000_10_0000; //  FETCH3
        control_store[3]  =
36'b00000000_0_0000_00000000000000_000_00_0000; //  NOOP
        control_store[4]  =
36'b00000000_0_0110_00000000000001_000_00_0000; //  CLAC
        control_store[5]  =
36'b00000110_0_1000_10000000000000_010_00_0000; //  LDAC 1
        control_store[6]  =
36'b00000111_0_0000_00000000000000_000_00_0000; //  LDAC 2
        control_store[7]  =
36'b00001000_0_0000_00000000000000_000_00_0000; //  LDAC 3
        control_store[8]  =
36'b00000000_0_0111_00000000000001_000_00_0010; //  LDAC 4
        control_store[9]  =
36'b00001010_0_0111_10000000000000_010_00_0011; //  LDX1R1 1
        control_store[10] =
36'b00001011_0_0000_00000000000000_000_00_0000; //  LDX1R1 2
        control_store[11] =
36'b00001100_0_0000_00000000000000_000_00_0000; //  LDX1R1 3
        control_store[12] =
36'b00001101_0_0111_00000000000001_000_00_0010; //  LDX1R1 4
        control_store[13] =
36'b00000000_0_1000_00001000000000_000_00_0000; //  LDX1R1 5
        control_store[14] =
36'b00001111_0_0111_10000000000000_010_00_0011; //  LDX2R2 1
        control_store[15] =
36'b00010000_0_0000_00000000000000_000_00_0000; //  LDX2R2 2
        control_store[16] =
36'b00010001_0_0000_00000000000000_000_00_0000; //  LDX2R2 3
        control_store[17] =
36'b00010010_0_0111_00000000000001_000_00_0010; //  LDX2R2 4
        control_store[18] =
36'b00000000_0_1000_00000100000000_000_00_0000; //  LDX2R2 5
        control_store[19] =
36'b00010100_0_0111_10000000000000_010_00_0011; //  LDX3R3 1
        control_store[20] =
36'b00010101_0_0000_00000000000000_000_00_0000; //  LDX3R3 2
        control_store[21] =
36'b00010110_0_0000_00000000000000_000_00_0000; //  LDX3R3 3
        control_store[22] =
36'b00010111_0_0111_00000000000001_000_00_0010; //  LDX3R3 4
        control_store[23] =
36'b00000000_0_1000_00000010000000_000_00_0000; //  LDX3R3 5
```

```
        control_store[24] =
36'b00011001_0_0111_10000000000000_010_00_0011; //  LDX12R9 1
        control_store[25] =
36'b00011010_0_0000_00000000000000_000_00_0000; //  LDX12R9 2
        control_store[26] =
36'b00011011_0_0000_00000000000000_000_00_0000; //  LDX12R9 3
        control_store[27] =
36'b00011100_0_0111_00000000000001_000_00_0010; //  LDX12R9 4
        control_store[28] =
36'b00000000_0_1000_00000000000010_000_00_0000; //  LDX12R9 5
        control_store[29] =
36'b00011110_0_0111_10000000000000_010_00_0011; //  LDX4R6 1
        control_store[30] =
36'b00011111_0_0000_00000000000000_000_00_0000; //  LDX4R6 2
        control_store[31] =
36'b00100000_0_0000_00000000000000_000_00_0000; //  LDX4R6 3
        control_store[32] =
36'b00100001_0_0111_00000000000001_000_00_0010; //  LDX4R6 4
        control_store[33] =
36'b00000000_0_1000_00000000010000_000_00_0000; //  LDX4R6 5
        control_store[34] =
36'b00100100_0_0111_00000000000001_000_00_0110; //  SWX5R3 1
        control_store[35] =
36'b00100100_0_0111_00000000000001_000_00_0101; //  SWX12R2 1
        control_store[36] =
36'b00100101_0_0111_10000000000000_000_00_0011; //  SWX5AC 1
        control_store[37] =
36'b00000000_0_1000_01000000000000_001_00_0000; //  SWX5AC 2
        control_store[38] =
36'b00000000_0_1000_01000000000000_001_00_0000; //  STAC
        control_store[39] =
36'b00000000_0_1000_00001000000000_000_00_0000; //  MOVACR1
        control_store[40] =
36'b00000000_0_1000_00000100000000_000_00_0000; //  MOVACR2
        control_store[41] =
36'b00000000_0_1000_00000010000000_000_00_0000; //  MOVACR3
        control_store[42] =
36'b00000000_0_1000_00000001000000_000_00_0000; //  MOVACR4
        control_store[43] =
36'b00000000_0_1000_00000000100000_000_00_0000; //  MOVACR5
        control_store[44] =
36'b00000000_0_1000_00000000010000_000_00_0000; //  MOVACR6
        control_store[45] =
36'b00000000_0_1000_00000000001000_000_00_0000; //  MOVACR7
        control_store[46] =
36'b00000000_0_1000_00000000000100_000_00_0000; //  MOVACR8
        control_store[47] =
36'b00000000_0_1000_00000000000010_000_00_0000; //  MOVACR9
        control_store[48] =
36'b00000000_0_0111_00000000000001_000_00_0100; //  MOVR1AC
        control_store[49] =
36'b00000000_0_0111_00000000000001_000_00_0101; //  MOVR2AC
        control_store[50] =
36'b00000000_0_0111_00000000000001_000_00_0110; //  MOVR3AC
        control_store[51] =
36'b00000000_0_0111_00000000000001_000_00_0111; //  MOVR4AC
        control_store[52] =
36'b00000000_0_0111_00000000000001_000_00_1000; //  MOVR5AC
        control_store[53] =
36'b00000000_0_0111_00000000000001_000_00_1001; //  MOVR6AC
```

```
        control_store[54] =
36'b00000000_0_0111_00000000000001_000_00_1010; //  MOVR7AC
        control_store[55] =
36'b00000000_0_0111_00000000000001_000_00_1011; //  MOVR8AC
        control_store[56] =
36'b00000000_0_0111_00000000000001_000_00_1100; //  MOVR9AC
        control_store[57] =
36'b00000000_0_1000_10000000000000_000_00_0000; //  MOVACMAR
        control_store[58] =
36'b00000000_0_1000_01000000000000_001_00_0000; //  MOVACMDR
        control_store[59] =
36'b00000000_0_0001_00000000000001_000_00_0100; //  ADDR1
        control_store[60] =
36'b00000000_0_0001_00000000000001_000_00_0101; //  ADDR2
        control_store[61] =
36'b00000000_0_0001_00000000000001_000_00_0111; //  ADDR4
        control_store[62] =
36'b00000000_0_0001_00000000000001_000_00_1000; //  ADDR5
        control_store[63] =
36'b00000000_0_0001_00000000000001_000_00_1001; //  ADDR6
        control_store[64] =
36'b00000000_0_0001_00000000000001_000_00_1010; //  ADDR7
        control_store[65] =
36'b00000000_0_0010_00000000000001_000_00_0000; //  LSHIFT1
        control_store[66] =
36'b00000000_0_0011_00000000000001_000_00_0000; //  RSHIFT1
        control_store[67] =
36'b00000000_0_0100_00000000000001_000_00_0000; //  LSHIFT8
        control_store[68] =
36'b00000000_0_0000_00000000000000_000_10_0000; //  INCREMENTPC
        control_store[69] =
36'b00000000_0_0000_00000000000000_000_01_0000; //  INCREMENTAC
        control_store[70] =
36'b00000000_0_0101_00000000000001_000_00_0000; //  DECREMENTAC
        control_store[71] =
36'b01001000_0_0111_00000000000001_000_00_0011; //  JUMP1
        control_store[72] =
36'b00000000_1_1000_00100000000000_000_00_0000; //  JUMP2
        control_store[73] =
36'bxxxxxxxx_1_0000_00000000000000_000_00_0000; //  JUMPZ1
        control_store[74] =
36'b00000000_0_0000_00000000000000_000_00_0000; //  JMPZN1  (Z = 0)
        control_store[75] =
36'b01001100_0_0111_00000000000001_000_00_0011; //  JMPZY1  (Z = 1)
        control_store[76] =
36'b00000000_1_1000_00100000000000_000_00_0000; //  JMPZY2  (Z = 1)
        control_store[77] =
36'bxxxxxxxx_1_0000_00000000000000_000_00_0000; //  JMPNZ1
        control_store[78] =
36'b00000000_0_0000_00000000000000_000_00_0000; //  JMPNZY1  (Z = 1)
        control_store[79] =
36'b01010000_0_0111_00000000000001_000_00_0011; //  JMPNZN1  (Z = 0)
        control_store[80] =
36'b01011000_1_1000_00100000000000_000_00_0000; //  JMPNZN2  (Z = 0)
        control_store[81] =
36'b01010010_0_0111_00000000000001_000_00_0011; //  LDIR2 1
        control_store[82] =
36'b00000000_0_1000_00000100000000_000_00_0000; //  LDIR2 2
        control_store[83] =
36'b01010100_0_0111_00000000000001_000_00_0011; //  LDIR3 1
```

```
        control_store[84] =
36'b00000000_0_1000_00000010000000_000_00_0000; //  LDIR3 2
        control_store[85] =
36'b00000000_0_0111_00001000000000_000_00_0011; //  LDIR1 1
        control_store[86] =
36'b00000000_0_0111_00000000010000_000_00_0011; //  LDIR6 1
        control_store[87] = {36{1'bz}}; // DONE
        control_store[88] =
36'b00000000_0_0110_00000000000001_000_00_0000; //JMPNZN3 (Z = 0)

    end

endmodule
```

8.2.3. RAM

```
`timescale 1ns / 1ps

module Ram#(
  parameter DATA_WIDTH = 8, // maximum 255
  parameter ADDR_WITDH = 18
)(
    input clk,
    input w_en,
    input r_en,
    input done,
    input  [(ADDR_WITDH-1):0] address,
    input  [(DATA_WIDTH-1):0] data_in,//maximum value is 256 (8 bits)
    output [(DATA_WIDTH-1):0] data_out
);

    // localparam ram_size=131072; //17 bits
    reg [(DATA_WIDTH-1):0] ram [2**ADDR_WITDH -1: 0];
    reg [(DATA_WIDTH-1):0] read_data;
    reg [17:0] index;
    reg [17:0] index_conv;
    integer f;
    integer g;

    always @(posedge clk)
        begin
            if (w_en==1)
                ram[address]<=data_in;
    end

    always @ (posedge clk)
        begin
            if(r_en == 1)
                read_data<= ram[address];
    end

    assign data_out = read_data;


    always@(posedge clk)
        begin
          if ( done  == 1 )
          begin
              index <= index+1;
              $fwrite(f,"%b\n",ram[index] ); //NOT DRAM < DMEM
            end
            if(index == 86383)
                begin
                  $fclose(f);
                  $finish;
                end
        end
    initial
    begin

        f = $fopen("E:\\Work\\ENTC\\5
CSD\\DownSampleMe-main_3\\Images\\clock_tower256-processor_output.txt
","w");
        index = 70000;              //start index
        ram[3]  = 8'bx; // height_count
        ram[4]  = 8'bx; // a: a local variable
        ram[5]  = 8'bx; // b: a local variable
        ram[6]  = 8'bx; // c: a local variable
        ram[7]  = 8'bx; // width_count

        // image data: 8 bit single channel image
        ram[20] = 8'b11010000;
        ram[21] = 8'b11010000;
        ram[22] = 8'b11010001;
        ram[23] = 8'b11010001;
        ram[24] = 8'b11010001;
        ………………
        ………………
        ………………
        ram[65555] = 8'b01100101;

    end

endmodule
```

8.2.4. MBRU

```
`timescale 1ns / 1ps

module MBRU(
    input clk,
    //input clk_devider,
    input fetch,
    input [25:0] instruction_in,
    output reg [7:0] instruction_out,
    output reg [27:0] data_out
    );

      always @(posedge clk)

        begin
            if (fetch == 1)
            begin
            instruction_out<=instruction_in[25:18];

             data_out<={10'b0000000000,instruction_in[17:0]};

            end
         end
endmodule
```

8.2.5. Instruction RAM

```
`timescale 1ns / 1ps
module Instruction_Ram(
    input clk,
    input [7:0] address,
    output reg [25:0] instr_out
    );


    parameter inst_count = 200;
    reg [25:0] inst_ram [inst_count:0];


    //define instructions
    parameter FETCH          =   8'd0;
    parameter NOOP           =   8'd3;
    parameter CLAC           =   8'd4;
    parameter LDAC           =   8'd5;
    parameter LDX1R1         =   8'd9;
    parameter LDX2R2         =   8'd14;
    parameter LDX3R3         =   8'd19;
    parameter LDXXR9         =   8'd24;
    parameter LDX4R6         =   8'd29;
    parameter SWX5R3         =   8'd34;
    parameter SWX12R2        =   8'd35;
    parameter SWXXAC         =   8'd36;
    parameter STAC           =   8'd38;
    parameter MOVACR1        =   8'd39;
    parameter MOVACR2        =   8'd40;
    parameter MOVACR3        =   8'd41;
    parameter MOVACR4        =   8'd42;
    parameter MOVACR5        =   8'd43;
    parameter MOVACR6        =   8'd44;
    parameter MOVACR7        =   8'd45;
    parameter MOVACR8        =   8'd46;
    parameter MOVACR9        =   8'd47;
    parameter MOVR1AC        =   8'd48;
    parameter MOVR2AC        =   8'd49;
    parameter MOVR3AC        =   8'd50;
    parameter MOVR4AC        =   8'd51;
    parameter MOVR5AC        =   8'd52;
    parameter MOVR6AC        =   8'd53;
    parameter MOVR7AC        =   8'd54;
    parameter MOVR8AC        =   8'd55;
    parameter MOVR9AC        =   8'd56;
    parameter MOVACMAR       =   8'd57;
    parameter MOVACMDR       =   8'd58;
    parameter ADDR1          =   8'd59;
    parameter ADDR2          =   8'd60;
    parameter ADDR4          =   8'd61;
    parameter ADDR5          =   8'd62;
    parameter ADDR6          =   8'd63;
    parameter ADDR7          =   8'd64;
    parameter LSHIFT1        =   8'd65;
    parameter RSHIFT1        =   8'd66;
    parameter LSHIFT8        =   8'd67;
    parameter INCREMENTPC    =   8'd68;
    parameter INCREMENTAC    =   8'd69;
    parameter DECREMENTAC    =   8'd70;
    parameter JUMP           =   8'd71;
    parameter JMPZ          =   8'd73;
    parameter JMPNZ         =   8'd77;
    parameter LDIR2         =   8'd81;
    parameter LDIR3         =   8'd83;
    parameter LDIR1         =   8'd85;
    parameter LDIR6         =   8'd86;
    parameter DONE          =   8'd87;
// completion of the downsampling

    initial
    begin
        inst_ram[0]     = {FETCH,   18'd0};
        inst_ram[1]     = {CLAC,    18'd0};
        inst_ram[2]     = {LDIR1,  18'd20}; // start of original
image
        inst_ram[3]     = {LDIR2,  18'd256}; // image width as an
immediate data - change
        inst_ram[4]     = {LDIR3,  18'd256}; // image height as an
immediate data - change
        inst_ram[5]     = {SWX5R3,  18'd3};
        inst_ram[6]     = {CLAC,    18'd0};
        inst_ram[7]     = {MOVACR4, 18'd0};
        inst_ram[8]     = {MOVACR5, 18'd0};
        inst_ram[9]     = {MOVACR6, 18'd0};
        inst_ram[10]    = {MOVR4AC, 18'd0};
        inst_ram[11]    = {LSHIFT8, 18'd0};
        inst_ram[12]    = {ADDR5,   18'd0};
        inst_ram[13]    = {ADDR1,   18'd0};
        inst_ram[14]    = {LDAC,    18'd0};
        inst_ram[15]    = {LSHIFT1, 18'd0};
        inst_ram[16]    = {MOVACR7, 18'd0};
        inst_ram[17]    = {MOVR2AC, 18'd0};
        inst_ram[18]    = {DECREMENTAC, 18'd0};
        inst_ram[19]    = {SWXXAC, 18'd7};
        inst_ram[20]    = {MOVR4AC, 18'd0};
        inst_ram[21]    = {LSHIFT8, 18'd0};
        inst_ram[22]    = {ADDR5, 18'd0};
        inst_ram[23]    = {INCREMENTAC, 18'd0};
        inst_ram[24]    = {ADDR1, 18'd0};
        inst_ram[25]    = {LDAC, 18'd0};
        inst_ram[26]    = {MOVACR8, 18'd0};
        inst_ram[27]    = {ADDR7, 18'd0};
        inst_ram[28]    = {ADDR6, 18'd0};
        inst_ram[29]    = {RSHIFT1, 18'd0};
        inst_ram[30]    = {RSHIFT1, 18'd0};
        inst_ram[31]    = {MOVACR9, 18'd0};
        inst_ram[32]    = {MOVR4AC, 18'd0};
        inst_ram[33]    = {LSHIFT8, 18'd0};
        inst_ram[34]    = {ADDR5, 18'd0};
        inst_ram[35]    = {ADDR1, 18'd0};
        inst_ram[36]    = {MOVACMAR, 18'd0};
        inst_ram[37]    = {MOVR9AC, 18'd0};
        inst_ram[38]    = {MOVACMDR, 18'd0};
        inst_ram[39]    = {MOVR7AC, 18'd0};
        inst_ram[40]    = {RSHIFT1, 18'd0};
        inst_ram[41]    = {MOVACR6, 18'd0};
        inst_ram[42]    = {MOVR8AC, 18'd0};
        inst_ram[43]    = {LSHIFT1, 18'd0};
        inst_ram[44]    = {MOVACR7, 18'd0};
        inst_ram[45]    = {MOVR5AC, 18'd0};
        inst_ram[46]    = {INCREMENTAC, 18'd0};
```

```
inst_ram[47]    = {MOVACR5, 18'd0};
inst_ram[48]    = {LDXXR9, 18'd7};
inst_ram[49]    = {MOVR9AC, 18'd0};
inst_ram[50]    = {DECREMENTAC, 18'd0};
inst_ram[51]    = {STAC, 18'd0};
inst_ram[52]    = {JMPNZ, 18'd20}; // double check this
inst_ram[53]    = {CLAC, 18'd0};
inst_ram[54]    = {ADDR7, 18'd0};
inst_ram[55]    = {ADDR6, 18'd0};
inst_ram[56]    = {RSHIFT1, 18'd0};
inst_ram[57]    = {RSHIFT1, 18'd0};
inst_ram[58]    = {MOVACR9, 18'd0};
inst_ram[59]    = {MOVR4AC, 18'd0};
inst_ram[60]    = {LSHIFT8, 18'd0};
inst_ram[61]    = {ADDR5,   18'd0};
inst_ram[62]    = {ADDR1, 18'd0};
inst_ram[63]    = {MOVACMAR, 18'd0};
inst_ram[64]    = {MOVR9AC, 18'd0};
inst_ram[65]    = {MOVACMDR, 18'd0};
inst_ram[66]    = {MOVR4AC, 18'd0};
inst_ram[67]    = {INCREMENTAC, 18'd0};
inst_ram[68]    = {MOVACR4, 18'd0};
inst_ram[69]    = {LDXXR9, 18'd3};
inst_ram[70]    = {MOVR9AC, 18'd0};
inst_ram[71]    = {DECREMENTAC, 18'd0};
inst_ram[72]    = {STAC, 18'd0};
inst_ram[73]    = {JMPNZ, 18'd8}; // double check this
inst_ram[74]    = {SWX12R2, 18'd7};
inst_ram[75]    = {CLAC, 18'd0};
inst_ram[76]    = {MOVACR5, 18'd0};
inst_ram[77]    = {MOVACR4, 18'd0};
inst_ram[78]    = {MOVACR6, 18'd0};
inst_ram[79]    = {MOVR4AC, 18'd0};
inst_ram[80]    = {LSHIFT8, 18'd0};
inst_ram[81]    = {ADDR5, 18'd0};
inst_ram[82]    = {ADDR1, 18'd0};
inst_ram[83]    = {LDAC, 18'd0};
inst_ram[84]    = {LSHIFT1, 18'd0};
inst_ram[85]    = {MOVACR7, 18'd0};
inst_ram[86]    = {MOVR3AC, 18'd0};
inst_ram[87]    = {DECREMENTAC, 18'd0};
inst_ram[88]    = {SWXXAC, 18'd3};
inst_ram[89]    = {MOVR4AC, 18'd0};
inst_ram[90]    = {LSHIFT8, 18'd0};
inst_ram[91]    = {ADDR5,   18'd0};
inst_ram[92]    = {ADDR2, 18'd0};
inst_ram[93]    = {ADDR1, 18'd0};
inst_ram[94]    = {LDAC, 18'd0};
inst_ram[95]    = {MOVACR8, 18'd0};
inst_ram[96]    = {ADDR7, 18'd0};
inst_ram[97]    = {ADDR6, 18'd0};
inst_ram[98]    = {RSHIFT1, 18'd0};
inst_ram[99]    = {RSHIFT1, 18'd0};
inst_ram[100]   = {MOVACR9, 18'd0};
inst_ram[101]   = {MOVR4AC, 18'd0};
inst_ram[102]   = {LSHIFT8, 18'd0};
inst_ram[103]   = {ADDR5, 18'd0};
inst_ram[104]   = {ADDR1, 18'd0};
inst_ram[105]   = {MOVACMAR, 18'd0};
inst_ram[106]   = {MOVR9AC, 18'd0};
inst_ram[107]   = {MOVACMDR, 18'd0};
```

```
        inst_ram[108]   = {MOVR7AC, 18'd0};
        inst_ram[109]   = {RSHIFT1, 18'd0};
        inst_ram[110]   = {MOVACR6, 18'd0};
        inst_ram[111]   = {MOVR8AC, 18'd0};
        inst_ram[112]   = {LSHIFT1, 18'd0};
        inst_ram[113]   = {MOVACR7, 18'd0};
        inst_ram[114]   = {MOVR4AC, 18'd0};
        inst_ram[115]   = {INCREMENTAC, 18'd0};
        inst_ram[116]   = {MOVACR4, 18'd0};
        inst_ram[117]   = {LDXXR9, 18'd3};
        inst_ram[118]   = {MOVR9AC, 18'd0};
        inst_ram[119]   = {DECREMENTAC, 18'd0};
        inst_ram[120]   = {STAC, 18'd0};
        inst_ram[121]   = {JMPNZ, 18'd89};
        inst_ram[122]   = {CLAC, 18'd0};
        inst_ram[123]   = {ADDR7, 18'd0};
        inst_ram[124]   = {ADDR6, 18'd0};
        inst_ram[125]   = {RSHIFT1, 18'd0};
        inst_ram[126]   = {RSHIFT1, 18'd0};
        inst_ram[127]   = {MOVACR9, 18'd0};
        inst_ram[128]   = {MOVR4AC, 18'd0};
        inst_ram[129]   = {LSHIFT8, 18'd0};
        inst_ram[130]   = {ADDR5,   18'd0};
        inst_ram[131]   = {ADDR1, 18'd0};
        inst_ram[132]   = {MOVACMAR, 18'd0};
        inst_ram[133]   = {MOVR9AC, 18'd0};
        inst_ram[134]   = {MOVACMDR, 18'd0};
        inst_ram[135]   = {MOVR5AC, 18'd0};
        inst_ram[136]   = {INCREMENTAC, 18'd0};
        inst_ram[137]   = {MOVACR5, 18'd0};
        inst_ram[138]   = {LDXXR9, 18'd7};
        inst_ram[139]   = {MOVR9AC, 18'd0};
        inst_ram[140]   = {DECREMENTAC, 18'd0};
        inst_ram[141]   = {STAC, 18'd0};
        inst_ram[142]   = {JMPNZ, 18'd77};
        inst_ram[143]   = {LDIR6, 18'd70000};
    // start of downsampled image
        inst_ram[144]   = {MOVR3AC, 18'd0};
        inst_ram[145]   = {RSHIFT1, 18'd0};
        inst_ram[146]   = {SWXXAC, 18'd3};
        inst_ram[147]   = {CLAC, 18'd0};
        inst_ram[148]   = {MOVACR4, 18'd0};
        inst_ram[149]   = {MOVACR5, 18'd0};
        inst_ram[150]   = {MOVR2AC, 18'd0};
        inst_ram[151]   = {RSHIFT1, 18'd0};
        inst_ram[152]   = {SWXXAC, 18'd7};
        inst_ram[153]   = {MOVR5AC, 18'd0};
        inst_ram[154]   = {LSHIFT8, 18'd0};
        inst_ram[155]   = {ADDR4, 18'd0};
        inst_ram[156]   = {LSHIFT1, 18'd0};
        inst_ram[157]   = {ADDR1, 18'd0};
        inst_ram[158]   = {LDAC, 18'd0};
        inst_ram[159]   = {MOVACR9, 18'd0};
        inst_ram[160]   = {MOVR6AC, 18'd0};
        inst_ram[161]   = {MOVACMAR, 18'd0};
        inst_ram[162]   = {INCREMENTAC, 18'd0};
        inst_ram[163]   = {MOVACR6,   18'd0};
        inst_ram[164]   = {MOVR9AC, 18'd0};
        inst_ram[165]   = {MOVACMDR, 18'd0};
        inst_ram[166]   = {MOVR5AC, 18'd0};
        inst_ram[167]   = {INCREMENTAC, 18'd0};
        inst_ram[168]    = {MOVACR5, 18'd0};
        inst_ram[169]    = {LDXXR9, 18'd7};
        inst_ram[170]    = {MOVR9AC, 18'd0};
        inst_ram[171]    = {DECREMENTAC, 18'd0};
        inst_ram[172]    = {STAC, 18'd0};
        inst_ram[173]    = {JMPNZ, 18'd153};
        inst_ram[174]    = {MOVR4AC, 18'd0};
        inst_ram[175]    = {INCREMENTAC, 18'd0};
        inst_ram[176]    = {MOVACR4, 18'd0};
        inst_ram[177]    = {LDXXR9, 18'd3};
        inst_ram[178]    = {MOVR9AC, 18'd0};
        inst_ram[179]    = {DECREMENTAC, 18'd0};
        inst_ram[180]    = {STAC, 18'd0};
        inst_ram[181]    = {JMPNZ, 18'd149};
        inst_ram[182]    = {NOOP, 18'd0};
        inst_ram[183]    = {DONE, 18'd0};

    end

    always @(posedge clk)
    begin
        instr_out<= inst_ram[address];

    end

endmodule
```

8.2.6. Program Counter

```
`timescale 1ns / 1ps


module ProgamCounter(
    input clk,
    input en,
    input w_en,
    input complete,
    input inc,
    input [27:0] data_in,
    output reg [7:0] instruction_address //input to instruction
register and B bus mux
    );

    reg start=1'b0;

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

     initial
        begin
            instruction_address=8'b0;
        end

    always @(posedge clk)
        begin
            if (complete==1)
                begin
                instruction_address <= instruction_address;
                end
            else if (start)
```

```
                begin
                    if(w_en==1)
                        begin
                            instruction_address <= data_in[7:0];
                        end
                     else if (inc==1)
                         begin
                            instruction_address <=
instruction_address+1;
                         end
                     else
                         begin
                            instruction_address<=instruction_address;
                         end
                 end
        end

endmodule
```

8.2.7. MAR

```
`timescale 1ns / 1ps

module MAR(
    input clk,
    input w_en,
    output reg [27:0] data_out,
    input [27:0] data_in,
    output reg [17:0] data_addr
    );
     always @ (posedge clk)
        begin
            if (w_en)
                begin                          //error - add begin
and end
                data_addr<=data_in[17:0];
                data_out<=data_in;
                end
        end

endmodule
```

8.2.8. MDR

```
`timescale 1ns / 1ps

module MDR(
    input clk,
    input enable,
    input w_en,
    input write_en,
    input read_en,
    output reg [27:0] data_out,
    input [27:0] data_in,
    input [7:0] DRam_in,
    output reg [7:0] DRam_out
    );


    reg [1:0] state = 2'b0;
    always @(posedge clk)
        if (enable==1'b1)
            begin
            state<=state+1;
            end


    always @(posedge clk)
    begin
        if (state==2'b10)
            begin
                if (w_en==1)
                    begin
                    data_out<=data_in;
                    end
                if (read_en)
                    begin
                    data_out<= {20'd0,DRam_in};
                    end
                if (write_en)
                    begin
                    DRam_out<=data_in[7:0];
                    end
                end
    end

endmodule
```

8.2.9. ALU

```
`timescale 1ns / 1ps

module ALU1(
    input enable,
    input clk,
    input [27:0] A,
    input [27:0] B,
    output reg [27:0] out,
    output reg z_flag,
    input [3:0] sel
    );

    parameter ADD = 4'b0001;
    parameter LSHIFT1 = 4'b0010;
    parameter RSHIFT1 = 4'b0011;
    parameter LSHIFT8 = 4'b0100;
    parameter DEAC = 4'b0101;
    parameter RESET = 4'b0110;
    parameter PASS_B = 4'b0111;
    parameter PASS_A = 4'b1000;
    parameter SUB = 4'b1001;
    parameter INCAC = 4'b1011;

    //events performed at positive edge

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

    initial
        begin
            out = 24'b0;
            z_flag = 1'b0;
        end

    always @(posedge clk)
        if (enable==1'b1)
            begin
            state<=state+1;
            end

    always @(posedge clk)
    begin
        if (state==2'b11)
            begin
                case(sel)
                ADD:
                    out<=A+B;
                SUB:
                    begin
                    out<=A-B;
                    if (out==28'b0)
                        z_flag = 1'b1;
                    else
                        z_flag = 1'b0;
                     end
                PASS_A:
                    out<=A;
                PASS_B:
                    out<=B;
                INCAC:
                    out<=A+1;
                DEAC:
                    out<=A-1;
                LSHIFT1:
                    out<=A<<1;
                LSHIFT8:
                    out<=A<<2;
                RSHIFT1:
                    out<=A>>1;
                RESET:
                    out<=24'b0;
                default:
                    out<=24'b0;


                endcase
             end
    end
endmodule
```

8.2.10. B bus multiplexer

```
`timescale 1ns / 1ps

module B_Bus_Mux(
    input clk,
    input [3:0] sel,
    input [27:0] AC,
    input [27:0] MAR,
    input [27:0] MDR,
    input [7:0] PC,
    input [27:0] MBRU,
    input [27:0] R1,
    input [27:0] R2,
    input [27:0] R3,
    input [27:0] R4,
    input [27:0] R5,
    input [27:0] R6,
    input [27:0] R7,
    input [27:0] R8,
    input [27:0] R9,
    output reg [27:0] Bus_Out
    );

    always @(sel or MAR or MDR or PC or MBRU or R1 or R2 or R3 or R4
or R5 or R6 or R7 or R8 or R9)

        begin
            case(sel)
            4'b0000 : Bus_Out <= AC;
            4'b0001 : Bus_Out <= MAR;
            4'b0010 : Bus_Out <= MDR;
            4'b0011 : Bus_Out <= MBRU;
            4'b0100 : Bus_Out <= R1;
            4'b0101 : Bus_Out <= R2;
            4'b0110 : Bus_Out <= R3;
            4'b0111 : Bus_Out <= R4;
            4'b1000 : Bus_Out <= R5;
            4'b1001 : Bus_Out <= R6;
            4'b1010 : Bus_Out <= R7;
            4'b1011 : Bus_Out <= R8;
            4'b1100 : Bus_Out <= R9;
            4'b1101 : Bus_Out <= {20'b0,PC};
            default : Bus_Out <= 18'b0;
            endcase
         end
endmodule
```

8.2.11. Accumulator

```
`timescale 1ns / 1ps

module Accumilator(
    input clk,
    input [27:0] data_in,
    output reg [27:0] data_out,
    input inc,
    input w_en
    );

    //events done at pose edge
    always @(posedge clk)
    begin
        if (inc==1)
            data_out<= data_out+1;
        else if (w_en==1)
            data_out<= data_in;

    end
endmodule
```

8.2.12. Registers

```
`timescale 1ns / 1ps

module Register(
    input clk,
    input w_en,
    input [27:0] data_in,
    output reg [27:0] data_out
    );


    //register set at posedge clk
    always @(posedge clk)
    begin
        if(w_en == 1)
            data_out<=data_in;
    end

endmodule
```

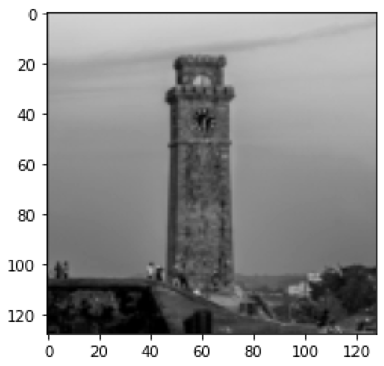
8.2.13. Clock divider

```
`timescale 1ns / 1ps

module clk_divider(
    input clk,
    output clk_div
    );
reg [3:0]count=4'd0;
parameter limit=4'd4;
reg out=0;

assign clk_div=out;

always @(posedge clk)
    begin

    if (count>=(limit-1))
        begin
        count<=4'd0;
        out<= ~out;
        end

    else
        count <= count+4'd1;

    end

endmodule
```

8.3. Appendix C - Images

- Original 256 by 256 image

- Downsampled Image using designed processor



- Image generated using python