EN 3030

CIRCUITS AND SYSTEMS DESIGN

FPGA BASED PROCESSOR IMPLEMENTATION

Group Members :

K.G. Abeywardena	160005C
H.L.S.H. Jayasundara	160243D
G.K.S.R. Karunasena	160285G
S.K. Sumanthiran	160616B

Supervisor :

Dr. Jayathu Samarawickrama

A Submitted Project Report In Partial Fulfilment Of The Requirement For The Module EN 3030.

Abstract

The following project was implemented to down sample any given image by a factor of two using any programmable logic device by relevant Hardware Description Language, using any method which seems suitable and analyse the error probability. As per the requirement, the architecture we implemented can only down sample any colour or grayscale image by a factor of two.

Contents

1. Introduction

The goal of this project is to design a microprocessor for a specified task and the Central Processing Unit (CPU) structure, while simulating it using a Hardware Description Language (HDL) such as Verilog and to implement the given task using a programmable logic device, preferably a Field Programmable Gate Arrays (FPGA). This report contains documents regarding the structure of the microprocessor and CPU design, the Verilog codes used to configure it and finally the physical configuration of the hardware.

1.1 Central Processing Unit

Central Processing Unit (CPU) widely considered as the brain of any computer device, is the place where all the instructions are carried out by performing logical operations, arithmetic operations, control operations and input/output operations. Hence, as explained above, CPU contains of two main components, the processor and the control unit while having a memory component and I/O components connecting to the CPU separately.

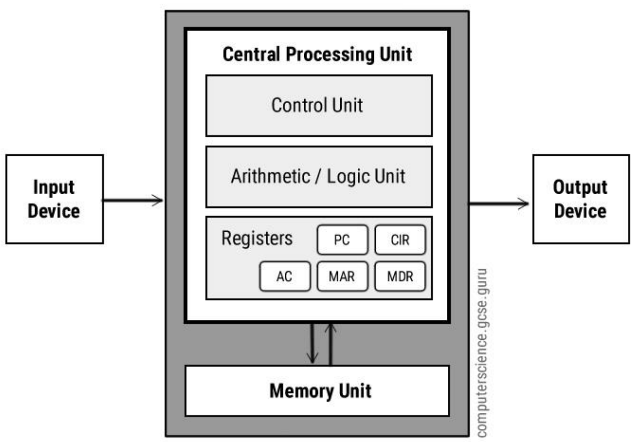


Figure 1.1.1 – Structure of the CPU

1.2 Microprocessor

Microprocessor can be considered as the most important part when it comes to a CPU. It is responsible for processing a unique set of instructions and processes. Its design is responsible to carry out logical and arithmetic operations. Microprocessors consists of many integrated circuits which holds thousands of components such as transistors etc.

Microprocessors are very important figures when it comes to modern computer designs and applications. With the high usage of microprocessors in the current market, various types of task-oriented microprocessors are available today. In this project we plan to design and implement a specific task-oriented microprocessor with maximum optimization.

2. Problem Statement

In this section we look at the problem we're trying to solve using the task-oriented microprocessor and the approach for that we'll be covered in the latter section.

As simple as it may see, our task was to down-sample any given image by a factor of two using a unique architecture of our own. Even though the task was only to down-sample an image, the process was much more than that. First, the serial input of the image had to be given to the processor and save it in the main memory for later calculations. Then, after the down-sampling process, image should be resaved into the memory and then transmitted back to the user.
Hence, in this section we'll discuss the problem statement under 3 main categories based on above details.

1. **Sending the serial encoded pixel values to the FPGA and storing process**

Since data should be given in a serial manner to the FPGA, it is a must to first convert the image pixels to serial values and then transmit these bits. These transmitted bits should be saved inside the memory for further calculations in the down-sampling process.

2. **Down-sampling process**

Before we begin the down-sampling process we should filter out the image to get rid of the high frequency components because it can cause aliasing problem. After filtering the source image, the filtered image should be resaved in the memory itself.

The filtered image should be then down sampled by a factor of two. Any method can be used for this process as explained above. After successful completion of the down sampling process, image should be saved back into the main memory.

3. **Transmitting the result back to the user**

After the image being down sampled, the resultant image should be transmitted back to the user in order to observe the results. Since, FPGA is only capable of transmitting data under serial communication method, before transmitting the image pixel values, we must convert them to serial data as we did in the first process.

3. Proposed Solution

The proposed solution for the problem statements will be discussed under the same 3 categories.

1. **Sending the serial encoded pixel values to the FPGA and storing process**

Image is transmitted to the FPGA using UART (Universal Asynchronous Receiver/Transmitter) with 50MHz clock frequency. Image is sent by byte by byte serially using 4 states. They are START, IDLE, DATA, STOP. We built a python kernel using several libraries such as pyserial to communicate with RS232 port of the FPGA via the serial cable. After the image is sent to the FPGA we are notified by a LED. Then the process is triggered by a flag which is sent from the data memory module we built

Since there were some errors when it comes to saving bits and storing them, we chose not to use the UART module and use the in-memory module for the data transfer process.

After the process is finished state of the processor is changed to the transmit state from process state. Then the UART transmitter is activated and it sends the processed down sampled image back to the computer for comparing and displaying purposes.

We can use any size of image in the architecture we're proposing, and pixel values will then be saved into the primary memory in the processor architecture. For the calculations and processing tasks in the latter sections, the values saved in these memory locations will be used.

2. **Down-sampling process**

With the values stored in the primary memory of the CPU, we begin our processing part by filtering the image using a gaussian kernel. The filtered image values will be saved in the remaining memory slots of the main memory of the CPU. Then we begin our down-sampling process and according to the instructions that have been pre-defined by us, the processor will carry them out and finally save the down-sampled image in the main memory, overlapping the original image at the first place. The down-sampling process will be given in order to the microprocessor by us according to an Instruction Set Architecture (ISA) that is being developed uniquely to our microprocessor. Each instruction in the ISA will contain few micro instructions and they will be fetched accordingly form the Instruction Register according to the algorithm we've developed. The ISA and each registers task will be discussed later in this report.

3. **Transmitting the result back to the user**

The down sampled image will be sent back to the user in a similar manner used in transmitting the image.

Finally, the down-sampled image from our microprocessor will be compared with down-sampled image with software-based simulations (Python- OpenCV) and the error will be calculated if there’s any.

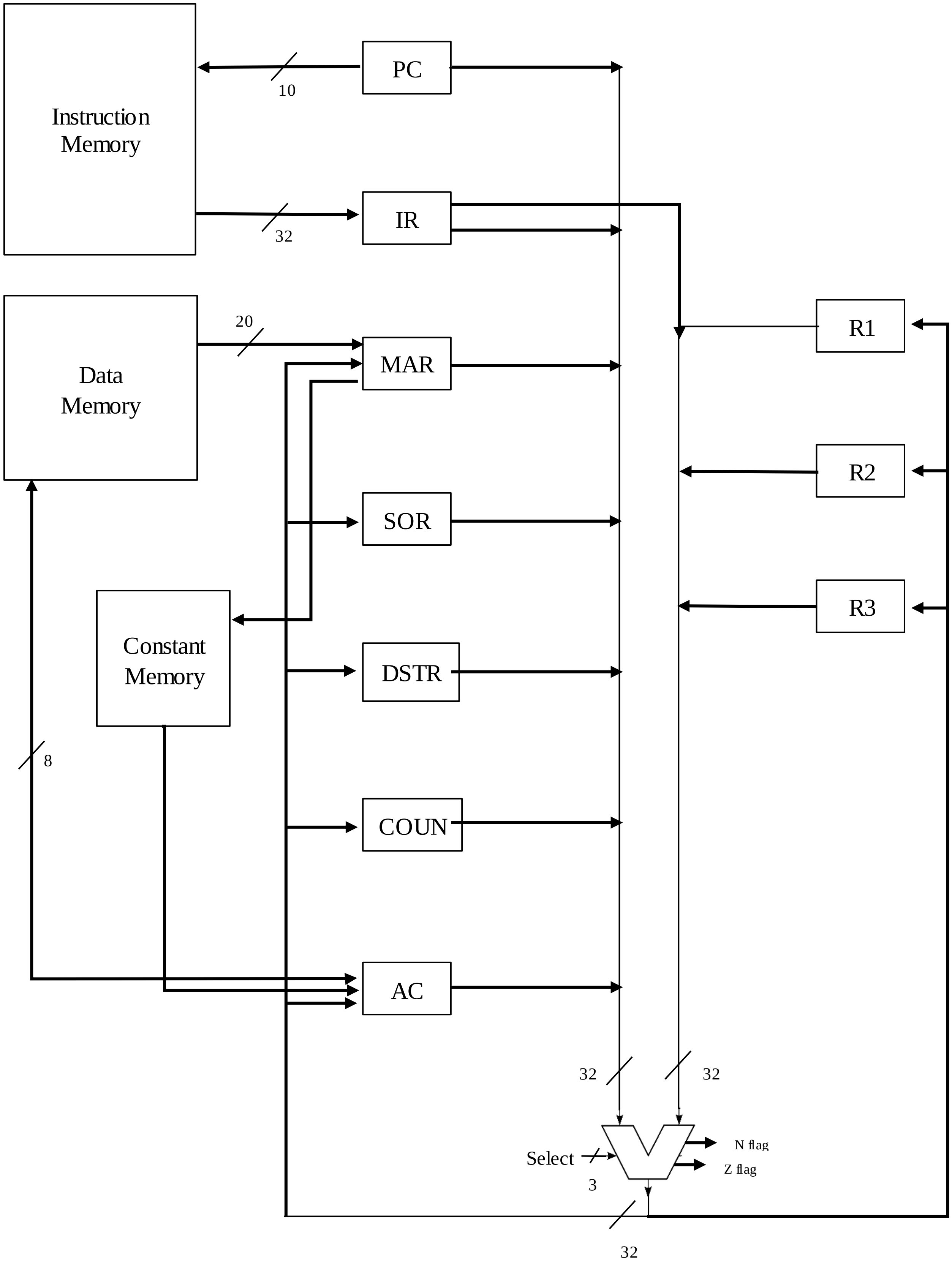
4. Instruction Set Architecture (ISA)

4.1 Overview

As explained above, the main task of this processor is to down-sample any given image by a factor of two. In this section, we'll look at the architecture we have implemented to acquire this task and the use of all the special case and general case registers.

- **PC** - A 10-bit Program Counter which keeps the main memory address of the next instruction to be fetched.
- **IR** - A 32-bit Instruction Register which holds the Instructions fetched.
- **MAR** - A 32-bit register
- **AC** - A 32-bit Accumulator which saves the intermediate values after a computation and the pixels values to be written/ read from data memory
- **SOR** - A 32-bit register which points the source location of the pixel
- **DSTR** - A 32-bit register which points the destination location of the pixel.
- **COUN** - A 32-bit register which is used for incrementing purposes.
- **R1** - A 32-bit register used for general purpose and to store constant data
- **R2** - A 32-bit register used for general purpose and to store constant data
- **R3** - A 32-bit register used for general purpose and to store constant data

4.2 Datapath


Instruction Memory
PC
10
IR
32
Data Memory
20
MAR
SOR
Constant Memory
DSTR
COUN
AC
8
R1
R2
R3
32
32
Select
3
N flag
Z flag
32

4.3 Instruction Set

The architecture of this processor consists of 4 bit instruction codes which are shown below.

Instruction	OP Code	Syntax	Parameters	Registers (R) Code	Registers (R) Name	Function	Instruction Code (32 bit) [31:28]	J [27]	R1[26:23]	R2[22:19]		[:0]
NOP	0001					No Operation	OPCODE	x	xxxx	xxxx	xxxx	xxxx
RSET	0010	Reg <- 0		4	COUN		OPCODE	x	R (5bits)		xxxx	xxxx
				3	DSTR							
				2	SOR							
				1	MAR							
				0	AC							
LOAD	0011	AC <- M[T]	J <- source			Loads data from the memory with address from the source	OPCODE	J	xxxx	xxxx	xxxx	A(20 bits)
			J = 0 -> T = MAR			Loads data to the AC Register						
			J = 1 -> T = Instruction Address (A)									

STOR	0100	M[T] <- AC	J <- source			Stores data to the memory with address from the source	OPCODE	J	xxxx	xxxx	xxxx	A(20 bits)
			J = 0 -> MAR			Stores data from the AC Register						
			J = 1 -> Instruction									
MVAR	0101	MAR <- Reg	J <- source	0	SOR	Moves data from Register to MAR	OPCODE	J	xxxx	xxxx	xxxx	xxxx
				1	DSTR							
MVAO	0110	Reg <- AC		12	MAR	Moves data from AC to a register	OPCODE	x	R	xxxx	xxxx	xxxx
				10	DSTR							
				9	SOR							
				6	R3							
				5	R2							
				4	R1							
MVAI	0111	AC<- A Bus <- Reg (J=0)	J <- Bus	6	COUN	Moves data from a register to AC	OPCODE	J	R	xxxx	xxxx	xxxx
			J = 0 <- Abus	4	MAR							
				2	DSTR							
				1	SOR							

		AC <- B Bus <- Reg (J=1)	J = 1 <- Bbus	6	R3							
				5	R2							
				4	R1							
INC	1000	Reg <- Reg + 1		4	COUNT	Increments the register(s) by 1	OPCODE	x	R (4 bits)	xxxx	xxxx	xxxx
				3	DSTR							
				2	SOR							
				1	MAR							
JUMP	1100	PC <- T	T -> Instruction to jump to	**Reg1**		Jumps to given address in Instruction Memory	OPCODE	x	Reg1	Reg2	N Z	T (10 Bits)
		Depending on Reg2 - Reg1	N -> Jump if negative flag is high	Any A Bus Register								
			Z -> Jump if zero flag is high	**Reg2**								
				Any B Bus Register								
ADD	1001	Reg1 <- Reg1 + K	J <- source	**Reg1**		Adds 2 values and returns the value to Reg1	OPCODE	J	Reg1	Reg2	xxxx	K (last 8 bit)
			J = 0 -> K = Constant	Any A Bus Register								
				Reg2								

			J = 1 -> K <= Reg2	Any B Bus Register							
SUB	1111	Reg1 <- K - Reg1	J <- source	**Reg1**	Subtracts 2 values and returns the value to Reg1	OPCODE	J	Reg1	Reg2	xxxx	K (last 8 bit)
			J = 0 -> K = Constant	Any A Bus Register							
				Reg2							
			J = 1 -> K <= Reg2	Any B Bus Register							
MUL	1101	Reg1 <- Reg1 *K	J <- source	**Reg1**	Multiplies 2 values and returns the value to Reg1	OPCODE	J	Reg1	Reg2	xxxx	K (last 8 bit)
			J = 0 -> K = Constant	Any A Bus Register							
				Reg2							
			J = 1 -> K <= Reg2	Any B Bus Register							
DIV	1110	Reg1 <- Reg1/ K	J <- source	**Reg1**	Divides 2 values and returns the value to Reg1	OPCODE	J	Reg1	Reg2	xxxx	K (8 bit)
			J = 0 -> K = Constant	Any A Bus Register							
				Reg2							
			J = 1 -> K <= Reg2	Any B Bus Register							
SFTR	1010	AC>> 1			Shifts AC Right by 1 (Floor division by 2)	OPCODE					

SFTL	1011	AC<< 1			Shifts AC Left by 1 (Multiplication by 2)	OPCODE					

4.4 Process Cycle

The processing of the written algorithm happens in 3 stages which are fetch, decode and execute. In this section we'll take a brief look at these 3 stages and the operation of all these stages.

4.4.1 Fetch Instructions

FETCH cycle consists of 2 stages with FETCH1 being set as the next state at the beginning.

FETCH1: Instruction Read
FETCH2: IR <- M, PC <- PC+1

4.4.2 Decode Instructions

After the instruction has been fetched, processor needs to identify which instruction has been fetched to invoke the correct execute cycle. During the decode cycle, processor identifies the execution cycle by the opcode of the instruction and passes down to the execute cycle to carry out the rest of the operation.

4.4.3 Execute Instruction

Execute instructions are the main types of instructions that carry out the major parts of the processing. The execute instructions can be classified according to the tasks they perform.

NOP Operation

NOP: No operation

RSET Operation

RSET: Reg <- 0;

LOAD Operation

J=0

LOAD1: M Read;
LOAD2: AC <- M[MAR];

J=1

LOAD1: MAR <- IR; M Read;
LOAD2: AC <- M[MAR];

STORE Operation

<u>J=0</u>

STORE1 : M Write;
STORE2 : M[MAR] <- AC;

<u>J=1</u>

STORE1 : MAR <- IR ; M Write;
STORE2 : AC <- M[MAR];

MVAR Operation

<u>J=0</u>

MVAR1 : MAR <- SOR;

<u>J=1</u>

MVAR1: MAR <- DSTR;

MVAC Operation

<u>J=0</u>

MVAC1 : Reg <- AC;

<u>J=1</u>

MVAC1 : AC <- Reg;

INC Operation

INC: : Reg <- Reg+1;

ADD Operation

<u>J=0</u>

ADD1 : Reg1 <- Reg1 + k;

<u>J=1</u>

ADD1 : Reg1 <- Reg1 + Reg2;

SUB Operation

<u>J=0</u>

SUB1 : Reg1 <- Reg1 - k;

<u>J=1</u>

SUB1 : Reg1 <- Reg1 - Reg2;

MUL Operation

<u>J=0</u>

MUL1 : Reg1 <- Reg1 * k;

<u>J=1</u>

MUL1 : Reg1 <- Reg1 * Reg2;

DIV Operation

J=0

DIV1 : Reg1 <- Reg1 / k;

J=1

DIV1 : Reg1 <- Reg1 /Reg2;

JUMP Instruction

JUMP1 : PC <- IR;

JMPX Instruction

JMPX1 : Reg1 – Reg2;

JMPXX Instruction

JMPXX1 : If instruction z flag and ALU z flag high - go to JUMP1
If instruction n flag and ALU n flag high - go to JUMP1
Else go to FETCH1

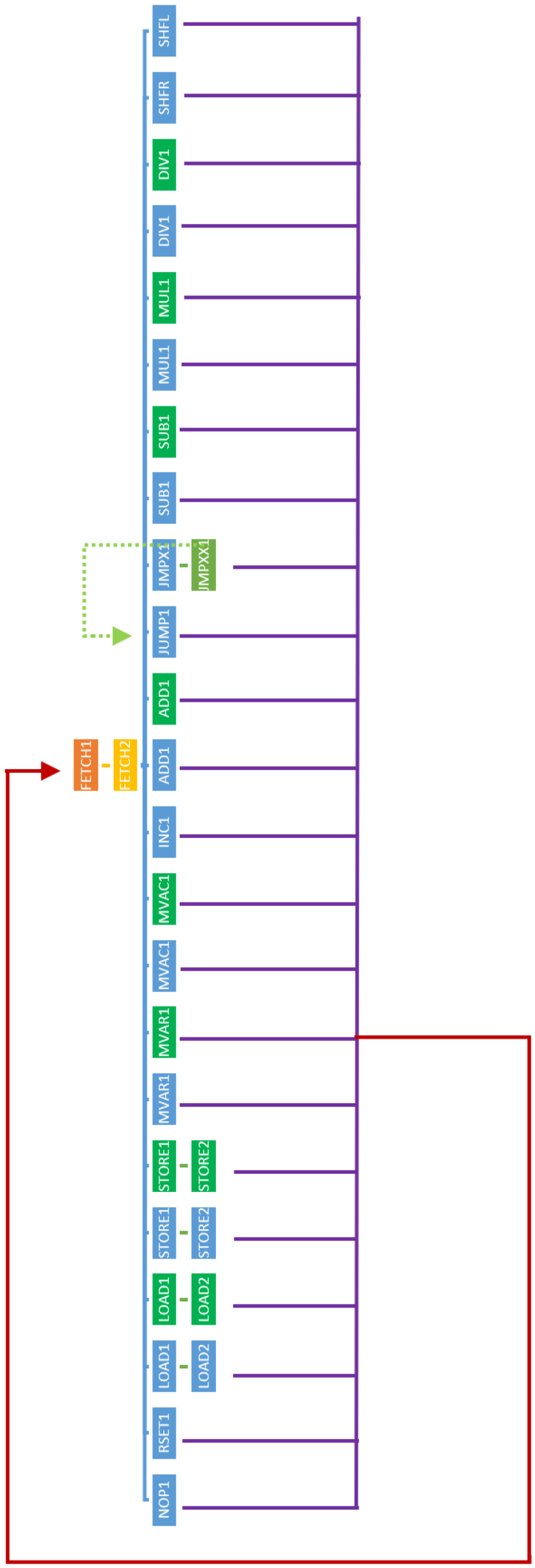
SHFR Operation

SHFR1: Shifts the input from the A Bus right by one bit

SHFL Operation

SHFL1: Shifts the input from the A Bus left by one bit

4.5 State Diagram



5. Algorithm

The algorithm for the image down-sampling process consists of two main sections which are the filtering of the image and the down-sampling itself. First, the filtering algorithm is applied to the whole image by selecting 3x1 and 1x3 pixel sets and then the down-sampling algorithm is applied. More details about each of these algorithms will be explained below.

5.1 Filtering Algorithm

Filtering is done by using a gaussian kernel. Instead of using a 3x3, 2-D gaussian kernel, we opt kernel separability where 2 kernels that are 1x3 and 3x1 run through the image as shown below.

$$\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix} = \begin{bmatrix}1\\2\\1\end{bmatrix} + \begin{bmatrix}1&2&1\end{bmatrix}$$

The main reason for the above selection was to make the algorithm more simple and easy to understand. At first vertical filtering was carried out and horizontal filtering was carried out on resultant image after the vertical filtering. Finally after the two filtrations, pixel values of the image were saved in the main memory. The weighted average was calculated by multiplying the pixel value and kernel value and adding the three likewise values together and dividing it by the weighted average of the kernel, which is 4 in both. The below illustration may give you a better understanding.

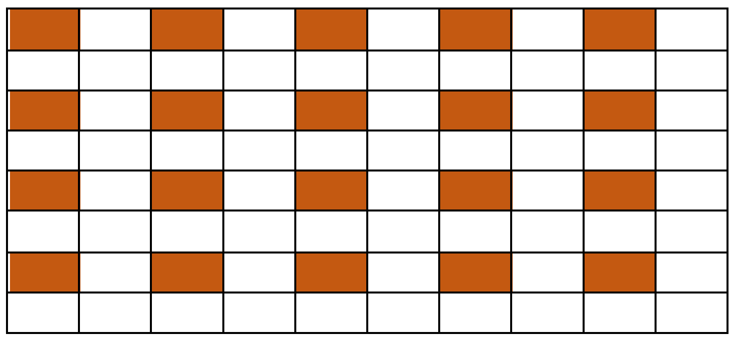
$$\begin{bmatrix}1\\2\\1\end{bmatrix}$$

x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x

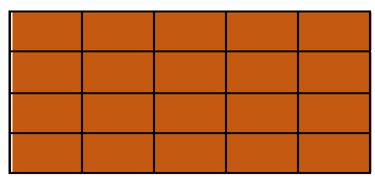
After applying the vertical kernel shown below to the three green pixels in the image, the centre pixel value (red one) will get updated.

5.2 Down Sampling Algorithm

After the filtering algorithm is applied to the image, we then apply the down sampling algorithm to the filtered image. The concept of the down sampling algorithm is to select each alternative pixel value. It is also done in two steps by selecting an alternative pixel values horizontally and skipping an entire row vertically. The below illustration will give you a better idea about this process.



Which will give us a resultant image as follows.



5.3 Pseudo Code

The pseudo code for the filtering and down sampling processes are shown below.

```
#Initialization of variables
reset all registers
SOR <- first pixel location of the source image
DSTR <- first pixel location of the destination image
last <- last pixel location of the source image
len <- width of the image
len <- len - 2

i = SOR
c = 0

#Begin Horizontal Gaussian Filtering
while (i+1 != last):

      filtered = mem[i] + 2*mem[i+1] + mem[i+2]
      filtered = filtered/4
```

```
    SOR <- SOR +1
    mem[DSTR] = filtered
    DSTR <- DSTR + 1
    c = c + 1
    if c == len:
        SOR <- SOR + 2
        C = 0
    i = SOR

last <- DSTR - 1
SOR <- first pixel location of the destination image
DSTR <- first pixel location of the source image

i = SOR

#Begin vertical filtering
while (i+1 != last):

    filtered = mem[i] + 2*mem[i+len] + mem[i+2*len]
    filtered = filtered/4
    SOR <- SOR +1
    mem[DSTR] = filtered
    DSTR <- DSTR + 1
    i = SOR

last <- DSTR - 1
SOR <- first pixel location of the source image
DSTR <- first pixel location of the destination image

#Begin Downsampling
while (SOR != last):
    mem[DSTR] = mem[SOR]
    DSTR <- DSTR + 1
    SOR <- SOR + 2

End
```

5.4 Assembly Code for the algorithm

For the assembly code, every instruction defined in the ISA above have been used.

No	Instruction	Binary Code	Comment
0	RSET	00100111110000000000000000000000	Resets all registers to 0
1	LOAD(J=1, A = M1)	00111000000000000000000000000011	
2	MVAO(SOR)	01100100100000000000000000000000	SOR always points to the current location in the source image
3	LOAD(J=1,A=M2)	00111000000000000000000000000100	
4	MVAO(DSTR)	01100101000000000000000000000000	DSTR always points to the current location in the destination image
5	LOAD(J=1,A=M3)	00111000000000000000000000000101	
6	MVAO(R2)	01100010100000000000000000000000	R2 always contains the last memory location of the current image
7	LOAD(J=1,A=M4)	00111000000000000000000000000110	
8	SUB (AC, J=0, K = 2)	11110101100000000000000000000010	
9	MVAO (R3)	01100011000000000000000000000000	R3 always contains the horizontal length of the image
Gaussian Smoothing occurs in 2 stages, vertically and horizontally			
Beginning of the Horizontal Smoothing			
10	MVAR (0)	01010000000000000000000000000000	Gaussian kernel is [1 2 1]
11	LOAD()	00110000000000000000000000000000	
12	MVAO (R1)	01100010000000000000000000000000	
13	INC (M=1)	10000000100000000000000000000000	
14	LOAD()	00110000000000000000000000000000	
15	SFTL ()	10110000000000000000000000000000	Multiplies the middle pixel by 2
16	ADD (AC,R1)	10011101101000000000000000000000	
17	MVAO (R1)	01100010000000000000000000000000	
18	INC (M=1)	10000000100000000000000000000000	
19	LOAD()	00110000000000000000000000000000	
20	ADD (AC,R1)	10011101101000000000000000000000	
21	SFTR()	10100000000000000000000000000000	Divides the filtered pixels by 4 to normalize it
22	SFTR()	10100000000000000000000000000000	

23	JUMP(Z=1, N=0, Reg1 =MAR, Reg2 = R2, T = 33)	11000110001010100000000000100001	Jumps if MAR is equal to R2 (i.e. we have reached the last location in the source image)
24	MVAR (1)	01011000000000000000000000000000	
25	STOR ()	01000000000000000000000000000000	
26	INC (S=1,D=1,C=1)	10000110000000000000000000000000	
27	JUMP(Z=1, N=0, Reg1 =COUN, Reg2 = R3, T = 29)	11000111001100100000000000011101	
28	JUMP (Z=0, N=0, T = 10)	11000000000000000000000000001010	
If reached end of line, should run			
29	INC (S=1)	10000001000000000000000000000000	
30	INC (S=1)	10000001000000000000000000000000	
31	RSET(C=1)	00100100000000000000000000000000	
32	JUMP(T = 10)	11000000000000000000000000001010	
If end of image is reached, should run			
33	MVAR (1)	01011000000000000000000000000000	
34	STOR()	01000000000000000000000000000000	
35	MVAI (DSTR)	01110101000000000000000000000000	
36	MVAO (R2)	01100010100000000000000000000000	
37	LOAD (J=1, A = M2)	00111000000000000000000000000100	
38	MVAO (SOR)	01100100100000000000000000000000	
39	LOAD (J=1, A = M1)	00111000000000000000000000000011	
40	MVAO (DSTR)	01100101000000000000000000000000	
41	JUMP (Z=0, N=0, T = 42)	11000000000000000000000000101010	
Beginning of the Vertical Smoothing			
42	MVAR (0)	01010000000000000000000000000000	Gaussian Kernel is [1 2 1]
43	LOAD()	00110000000000000000000000000000	
44	MVAO (R1)	01100010000000000000000000000000	
45	ADD (MAR,R3)	10011110001100000000000000000000	Updates the MAR address by increasing it by the value of one horizontal length.
46	LOAD()	00110000000000000000000000000000	Then MAR value points to the corresponding pixel in the next line
47	SFTL ()	10110000000000000000000000000000	Multiplies the middle pixel by 2
48	ADD (AC,R1)	10011101101000000000000000000000	
49	MVAO (R1)	01100010000000000000000000000000	

50	ADD (MAR, R3)	10011110001100000000000000000000	
51	LOAD()	00110000000000000000000000000000	
52	ADD (AC, R1)	10011101101000000000000000000000	
53	SFTR ()	10100000000000000000000000000000	Divides the filtered pixels by 4 to normalize it
54	SFTR ()	10100000000000000000000000000000	
55	JUMP (Z=1, N=0, Reg1 = MAR, Reg2 = R2, T = 60)	11000110001010100000000000111100	
56	MVAR (1)	01011000000000000000000000000000	
57	STOR ()	01000000000000000000000000000000	
58	INC (S=1, D=1)	10000011000000000000000000000000	
59	JUMP (Z=0, N=0, T = 42)	11000000000000000000000000101010	
Runs after the vertical smoothing finishes			
60	MVAR (1)	01011000000000000000000000000000	
61	STOR()	01000000000000000000000000000000	
62	MVAI (DSTR)	01110101000000000000000000000000	
63	MVAO (R2)	01100010100000000000000000000000	
64	LOAD (J=1, A = M1)	00111000000000000000000000000011	
65	MVAO (SOR)	01100100100000000000000000000000	
66	LOAD (J=1, A = M2)	00111000000000000000000000000100	
67	MVAO (DSTR)	01100101000000000000000000000000	
68	RSET(C=1)	00100100000000000000000000000000	
69	JUMP (Z=0, N=0, T = 70)	11000000000000000000000001000110	
Down sampling begins			
70	MVAR (0)	01010000000000000000000000000000	
71	LOAD ()	00110000000000000000000000000000	
72	MVAR (1)	01011000000000000000000000000000	
73	STOR ()	01000000000000000000000000000000	
74	INC (S = 1, D = 1, C = 1)	10000111000000000000000000000000	Destination image pointer is incremented by 1
75	JUMP (Z = 1, N = 0, Reg2 = R2, Reg1 = SOR, T = 86)	11000100101010100000000001010110	
76	INC (S= 1, C = 1)	10000101000000000000000000000000	Source image pointer is incremented by 2

77	JUMP (Z = 1, N = 0, Reg2 = R2, Reg1 = SOR, T = 86	11000100101010100000000001010110	
78	JUMP (Z = 1, N = 0, Reg2 = R3, Reg1 = COUN, T = 80)	11000111001100100000000001010000	
79	JUMP (Z=0, N=0, T = 69)	11000000000000000000000001000101	Jumps back to sample the next pixel
80	ADD (SOR, R3)	10011100101100000000000000000000	Runs if end of horizontal line is reached, skips the next line
81	RSET (C=1)	00100100000000000000000000000000	Resets the counter
82	SUB (J = 0, K = 1, Reg1 = SOR)	11110100100000000000000000000001	
83	JUMP (Z = 1, N = 0, Reg2 = R2, Reg1 = SOR, T = 86)	11000100101010100000000001010110	Checks if pointer has gone past the final image location, algorithm ends
84	INC(S = 1)	10000001000000000000000000000000	
85	JUMP (Z = 0, N = 0, T = 70)	11000000000000000000000001000110	Jumps to continue sampling
86	END	00011000000000000000000000000000	End of Algorithm

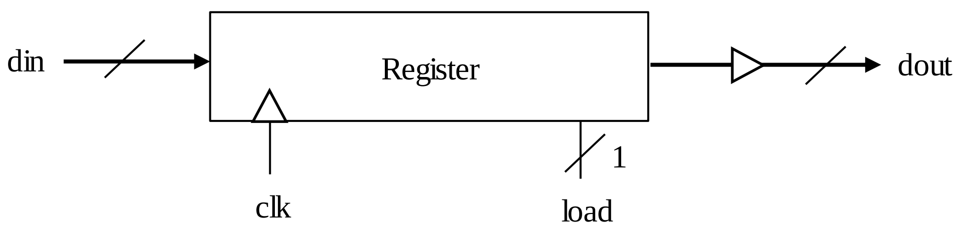
M1	Address of the first Memory location of the original image
M2	Address of the first Memory location of the destination image
M3	Address of the last Memory location of the original image
M4	Address of the memory location containing the length of the original image

6. Modules

In this section, we'll be looking at each type of modules used in our processor design. Hence, the following sections will contain thorough details about Registers, ALU, State Machine, Processor and Memory modules.

6.1 Registers

In this section we'll look at the architecture of the registers we have used in our design. Registers are used to store data temporally during the processing cycle. The size of the registers defer as explained above according to their relevant task. Data stored on these registers are directly connected to either A bus or B bus through a buffer, where buffers are activated according to the executing micro instruction at that instance. Since these registers doesn't have any increment flag, incrementation should be done through ALU if the stored values needs to be incremented.
A basic structure of a register used in our architecture is shown below. If the din flag is high, data is written into the register and if the output buffer is high, data is written into either A or B busses.



6.2 Arithmetic and Logic Unit (ALU)

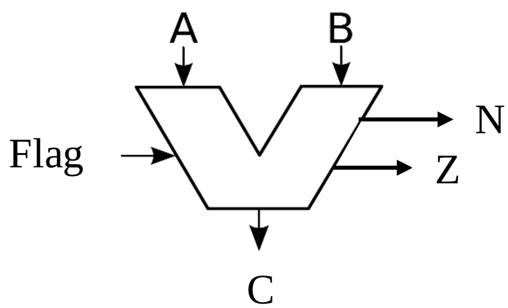
Each arithmetic and logical operation in the architecture we have designed are carried out by this module. ALU we have designed has 2 inputs which are A bus and B bus and one output which is the C bus. These busses carry 32 bits of data. PC, IR, MAR, SOR, DSTR, COUN and AC registers are connected to the A bus while R1, R2 and R3 registers are connected to the B bus. Output C bus is connected to each and every register except IR. Hence, as long as you have values the above mentioned registers, any ALU operation defined by the ISA can be performed.

The type of ALU operation is selected by the 3 bit OP flag which is predefined in the ALU module itself. Since, it is a 3 bit flag only 8 operations can be carried out. Those operations are listed below.

OP flag	Operation	Task
000	MOVA	abus_out
001	MOVB	bbus_out
010	ADD	abus_out + bbus_out
011	SUB	abus_out - bbus_out
100	MUL	abus_out * bbus_out
101	DIV	abus_out / bbus_out
110	SHIFTR	abus_out >> 1
111	SHIFTL	abus_out << 1

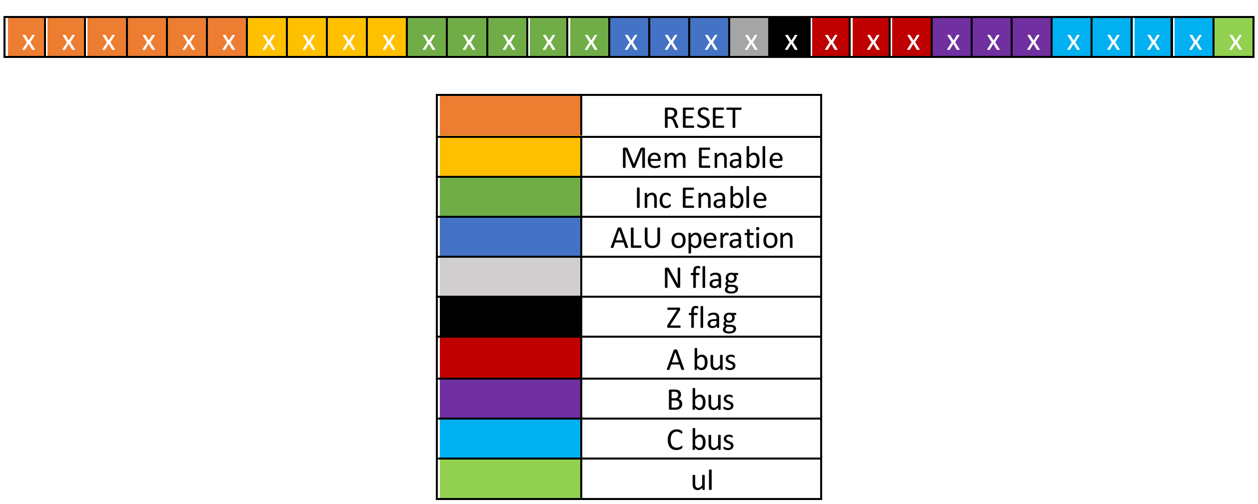
In our ALU, we have used two flags, Z and N respectively. Z and N are used for the jump instructions. if the jump instruction has 1 in z bit and if the flag Z returned from the ALU is 1, the jumping happens. Same with the N flag and n bit. The ALU and the flags used can be identified from the below figure. The bit value for each bus and selection is mentioned below.

- A bus – 32 bits
- B bus – 32 bits
- C bus – 32 bits
- Flag – 3 bits
- N flag – 1 bit
- Z flag – 1 bit



6.3 Controller (State Machine)

All instructions go to the controller, which in turn produces the control signals to be used in the rest of the processor. The controller has 5 inputs – the clock, the n and z flags, the current status of the processor and the current instruction. It produces 8 output signals – 3-bit signals for the ALU Op, A-Bus and B-Bus, 4 bit signals for the C-Bus and the memory enables, 5 bit increment control signal and 6 bit reset signal. It also has a 1-bit output to indicate the end of the process. The output control signals are as follows,



Control Signal	Operation
ALU OP	Determines the ALU operation
A bus/ B bus enables	Controls the buffers reading from the registers into the A and B buses
C bus enable	Controls the register reading from the C-Bus
INC	Controls the increment control signals going the registers which can be incremented.
MEM _EN	The read enables and write enable signals that go to the memory modules, and registers which read and write from memory.
RSET	Controls the reset control signal of reset-enabled registers
End	A flag raised when the process is ended

All instructions are mapped to an encoding value, which in turn points to the initial microinstruction for that instruction. All instructions and their encoding values are given below:

Instruction	Encoding Values
RSET	4
LOADR1	5
LOADK1	7
STORR1	10
STORK1	12
MVARS	15
MVARD	16
JUMP	17
MVACO	19
MVACA	20

MVACB	21
INC	22
ADDK	23
ADDR	24
SUBK	25
SUBR	26
MULK	27
MULR	28
DIVK	29
DIVR	30
SHFR	31
SFTL	32
END	33

At the beginning of each fetch cycle, the controller retrieves the instruction code from the IR. The first 5 bits of the instruction are unique for each instruction and map to the corresponding microinstruction. After the microinstructions for that instruction have been executed, then the cycle ends, and the next fetch cycle retrieves the next instruction.

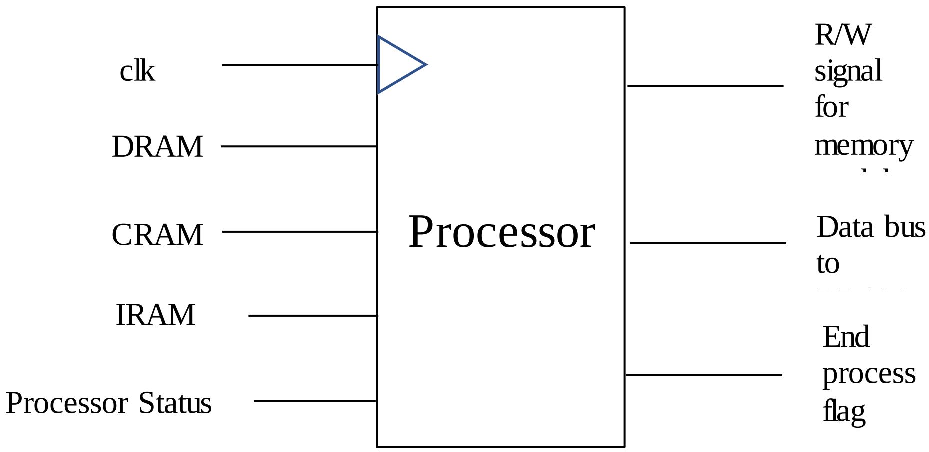
The only time this doesn't happen is when the END instruction is given, after which the controller sends the END flag high, and goes into an idle state.

6.4 Processor

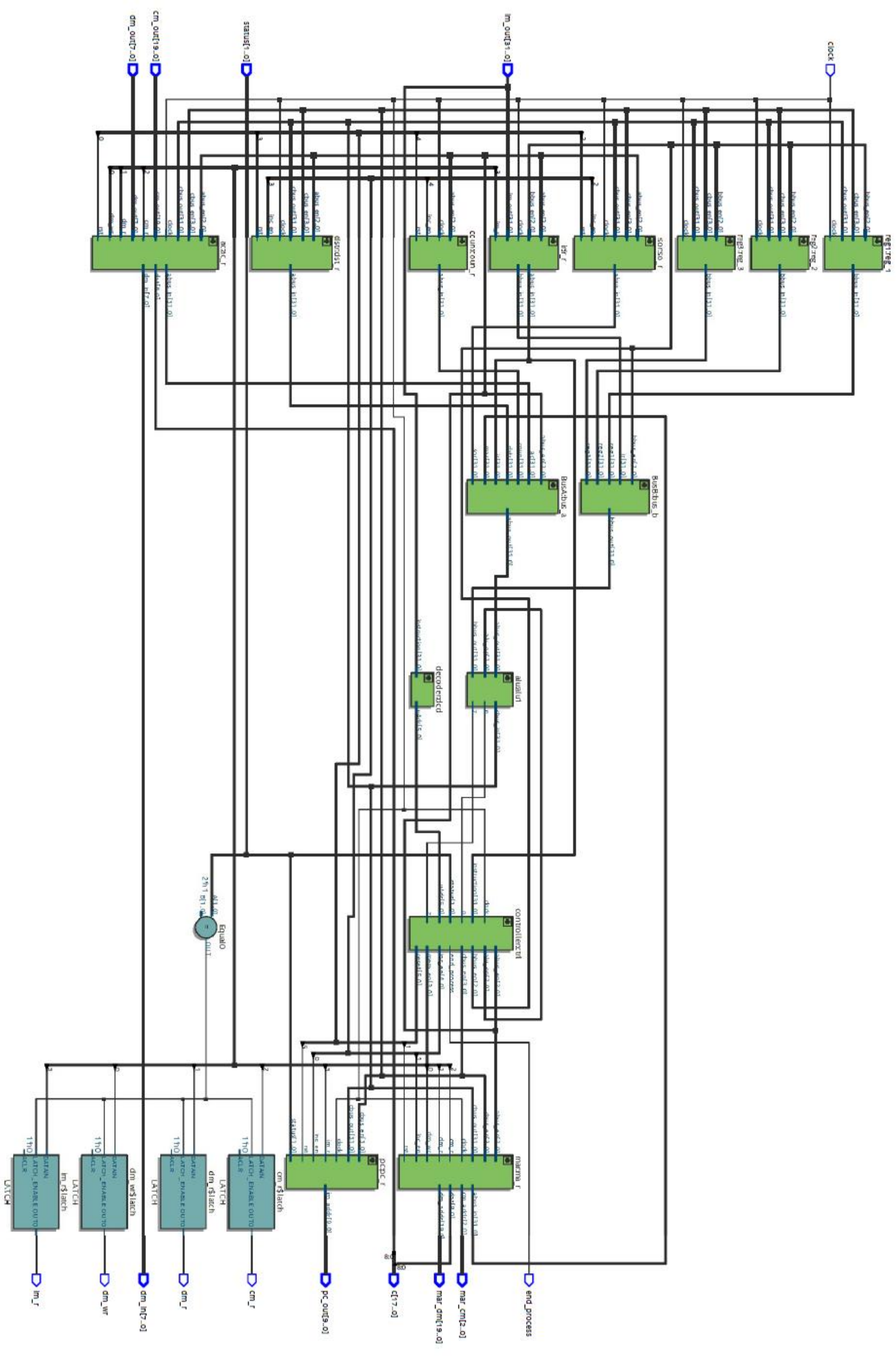
This module contains instances of all modules contained in the data path. The inputs for this module are the clock, the memory buses from the DRAM, CRAM and IRAM, and the processor status. The outputs are the read and write signals for each memory module, the data bus to the DRAM, and an end process flag.
The modules which have instantiations within the Processor are as follows:

- Reg1 Register
- Reg2 Register
- Reg3 Register
- DSTR Register
- SOR Register
- COUN Register
- AC Register
- PC Register
- IR Register
- Bus A
- Bus B
- ALU
- Decoder
- Controller

A complete figure of a processor with its inputs and outputs can be seen from the below diagram.



6.4 RTL View of the processor

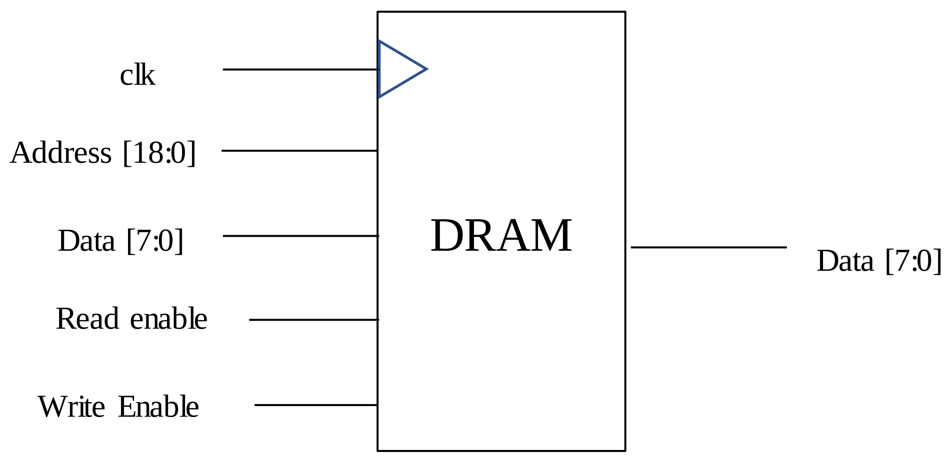


6.5 Memory Modules

In this section, we'll look at 3 memory modules we have used in this architecture which are IRAM, DRAM and CRAM.

6.4.1 DRAM

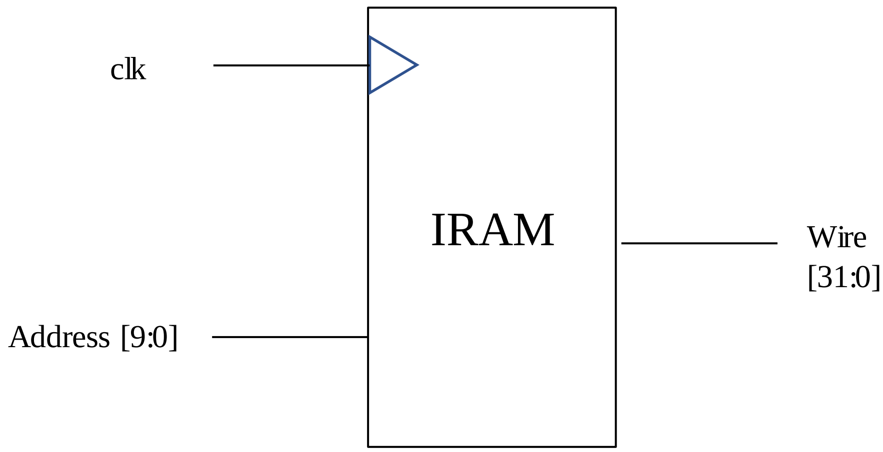
This memory module is mainly to save pixel values of image for further processing. The initial image values, pixel values after filtering, pixel values after down sampling are stored in this memory module. This module consists of 240,000 memory locations with each memory location being 8 bits in width. In these memory locations pixel values are saved which varies from 0 to 255. This memory module consists of 5 inputs and one output. The inputs are 19-bit address, clk, 8-bit data, read enable bit and write enable bit. The sole output is an 8-bit data wire carrying data out from the module. The figure of the DRAM module we have implemented in our architecture can be shown as below.



6.4.2 IRAM

Just like DRAM is specialized to store data values, IRAM is made to store all the instructions which are needed to run the down sampling algorithm correctly. All these instructions are coded in VHDL and are stored in memory locations of the IRAM. Hence, this memory module contains the assembly code for filtering and down sampling of the image. During the first fetch cycle, processor fetches instructions from the IRAM and then only the rest of the micro instructions will run accordingly. This memory module contains 256 memory locations with each being 32 bits in width. This module consists of 2 inputs which are clk and 10-bit memory address. The only output is the 32-bit wire carrying out the next instruction to be processed.

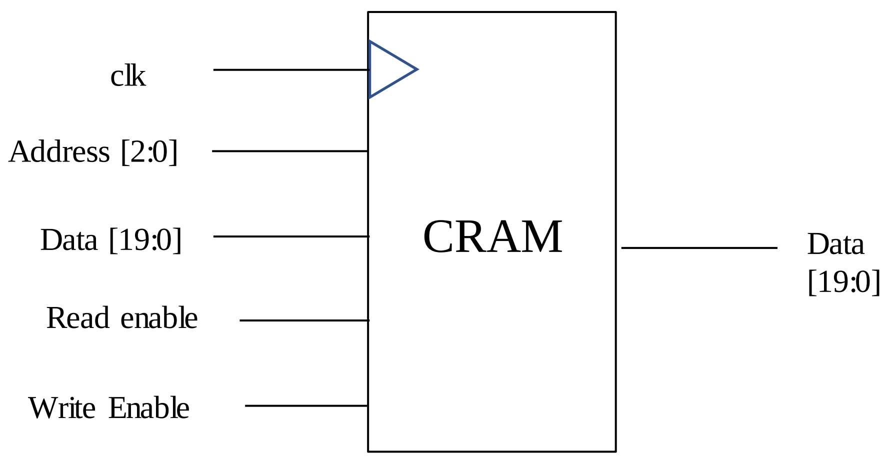
The figure of the IRAM module we have implemented in our architecture can be shown as below.



6.4.3 CRAM

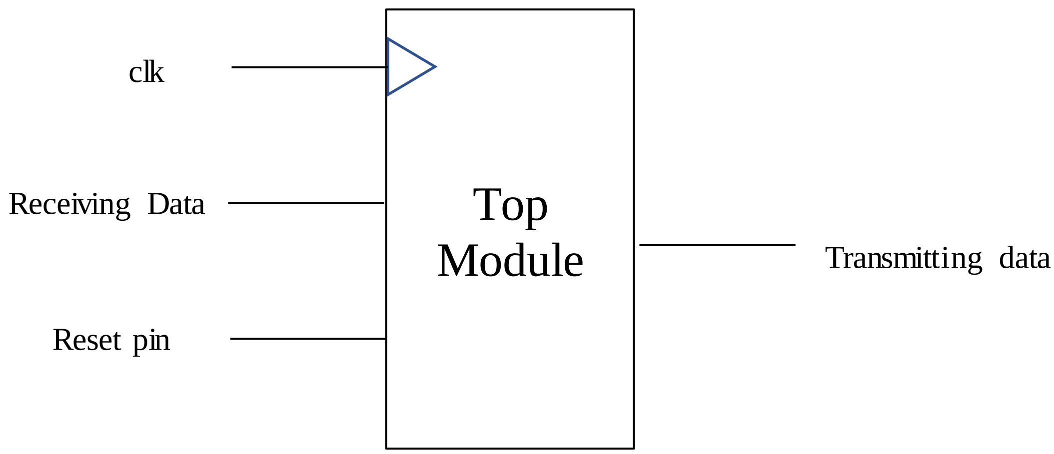
This memory module is used to store the constants of our architecture during the processing cycle. The length of the image, size of the image, first and last memory locations of the image are stored in CRAM. CRAM consists of 8 memory locations with each being 20 bits in width. This module has 5 inputs and one output. The five inputs are, 3-bit address, clk, 20-bit data, read enable and write enable. The single output is the 20-bit data output from the CRAM.

The figure of the CRAM module we have implemented in our architecture can be shown as below.

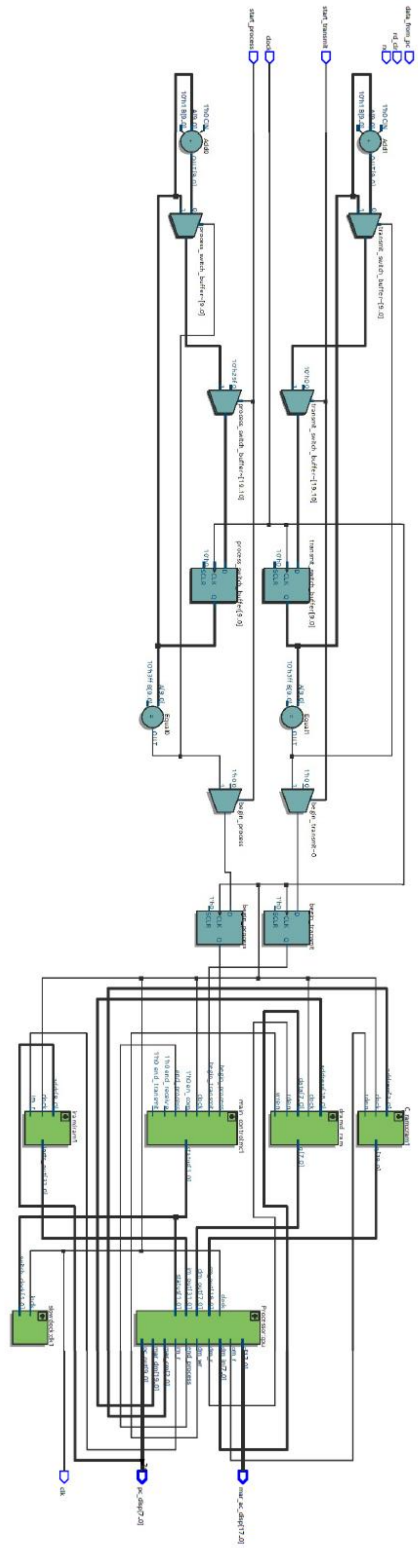


6.5 Top Module

This is the module which connects data storage and processing modules of our architecture. All the instances have been created inside this module. There are three inputs to this module which are clk, reset pin and receiving data. There is only one output which is the transmitting data. It contains the instances of the processor and memory modules.



6.6 RTL View of the Top Module



7. Testing and Simulations

After creating all the modules as explained above we needed to check whether we're obtaining the desired result from our processor. For the testing and simulation process, ModelSIm software was used. To create test modules Verilog syntaxes which are only supported simulation were used
Test code created to monitor the values inside the PC register.

```
module tb;
	reg clock = 0;
	reg data_from_pc, start_process, start_transmit, rx,rd_clr;
	wire rd_rx, tx, tx_busy, endImagereceived,endProcess,
	data_to_pc,g1, g2, g3,s0, s1, s2, s3;
	wire [7:0] LEDR;
	top_processor TP (clock, data_from_pc, start_process,
	start_transmit,rx,rd_clr,rd_rx,tx,
	tx_busy,LEDR,endImagereceived,endProcess,data_to_pc, g1, g2,
	g3,s00, s01, s02, s03,clk,s0);

	always begin
	#50 clock = ~clock;
	end


	initial
		begin
			$monitor("pc = %b ",LEDR);
			#100000 $finish;

		end
endmodule
```

8. Error Detection

In order to check how good our algorithm, we need to compare it with a reference result. Hence, for this task we used an opencv down sampled image from python. With the resultant image we get from python, we compared our resultant image from the processor and calculated the error.
Here, we have shown two images along with their opencv down sampled version and our processor down sampled version and the error between two images. Note that second image consists of many textures.

Original Image

Down sampled image using opencv

Down sampled image using FPGA

Error with the opencv down sampled image gives us 5% value. The main reason for this is the values saved on the memory of the FPGA was not ideal. Also the algorithm used by opencv is way more advanced than the one we used for out algorithm. Also numerical errors like floating points can affect this as well.

C:\Python27\python.exe

```
('Image shape', (127L, 127L, 3L))
('Error with OpenCV = ', 5.345418706425372)
('Error with own Algorithm = ', 0.0)
```

Original Image

Down sampled image using opencv

Down sampled image using FPGA

C:\Python27\python.exe

```
('Image shape', (127L, 127L, 3L))
('Error with OpenCV = ', 9.4459389749348315)
('Error with own Algorithm = ', 0.029461869187196389)
```

9. Appendixes

9.1 Appendix 1 : Python code for creating memory initializing files, encoding data, instructions and constant memory

```
import cv2 as cv
import numpy as np
import argparse


PC   =[0,0,0,1]
IR   =[1,1,1,1]
MAR  =[1,1,0,0]
AC   =[1,0,1,1]
R1   =[0,1,0,0]
R2   =[0,1,0,1]
R3   =[0,1,1,0]
SOR  =[1,0,0,1]
DSTR =[1,0,1,0]
COUN =[1,1,1,0]
NA      =[0,0,0,0]

def InstructionEncoder():

    count = 0

    M1 = 3 #First Memory Location of Orginal Image
    M2 = 4 #First Memory Location of Destination Image
    M3 = 5 #Last Memory Location of Original Image
    M4 = 6 #Memory Location containing width of original image


    #print ("Encoding Instructions")
    #print ("Algorithm in Binary")

    RSET(1,1,1,1,1)     #0
    LOAD(J=1, A = M1)   #1
    MVAO(SOR)           #2
    LOAD(J=1,A=M2)      #3
    MVAO(DSTR)          #4
    LOAD(J=1,A=M3)      #5
    MVAO(R2)            #6
    LOAD(J=1,A=M4)      #7
    SUB (AC, J=0, K = 2)#8 Changed to sort out line length issue
    MVAO (R3)           #9 Changed to sort out line length issue
```

```
    MVAR (0)            #10
    LOAD()              #11
    MVAO (R1)           #12
    INC (M=1)           #13
    LOAD()              #14
    SFTL ()             #15
    ADD (AC,R1)         #16
    MVAO (R1)           #17
    INC (M=1)           #18
    LOAD()              #19
    ADD (AC,R1)         #20
    SFTR()              #21
    SFTR()              #22
    JUMP(Z=1, N=0, Reg1 =MAR, Reg2 = R2, T = 33)#23
    MVAR (1)            #24
    STOR ()             #25
    INC (S=1,D=1,C=1)       #26 Changed to sort out line length issue
    JUMP(Z=1, N=0, Reg1 =COUN, Reg2 = R3, T = 29) #27  Changed to sort
out line length issue
    JUMP (Z=0, N=0, T = 10)#28 Changed to updated instruction number

    #If reached end of line, should run, at this point, C = L-2, and
SOR is pointing at the pixel before the last on that line
    INC (S=1)           #29
    INC (S=1)           #30
    RSET(C=1)           #31
    JUMP(T = 10)        #32

    #If end of image is reached, should run
    MVAR (1)            #33
    STOR()              #34
    MVAI (DSTR)         #35
    MVAO (R2)           #36
    LOAD (J=1, A = M2)  #37
    MVAO (SOR)          #38
    LOAD (J=1, A = M1)  #39
    MVAO (DSTR)         #40
    JUMP (Z=0, N=0, T = 42)#41

    #Begin Vertical Filtering
    MVAR (0)            #42
    LOAD()              #43
    MVAO (R1)           #44
    ADD (MAR,R3)        #45
    LOAD()              #46
    SFTL ()             #47
```

```
    ADD (AC,R1)         #48
    MVAO (R1)           #49
    ADD (MAR, R3)       #50
    LOAD()              #51
    ADD (AC,R1)         #52
    SFTR()              #53
    SFTR()              #54
    JUMP (Z=1, N=0, Reg1 =  MAR, Reg2 = R2, T = 60)#55
    MVAR (1)            #56
    STOR ()             #57
    INC (S=1, D=1)      #58
    JUMP (Z=0, N=0, T = 42)#59

    #Runs when vertical filtering finishes
    MVAR (1)            #60
    STOR()              #61
    MVAI (DSTR)         #62
    MVAO (R2)           #63
    LOAD (J=1, A = M1)  #64
    MVAO (SOR)          #65
    LOAD (J=1, A = M2)  #66
    MVAO (DSTR)         #67
    RSET(C=1)           #68
    JUMP (Z=0, N=0, T = 70)#69 Changed to updated instruction numbers

    #Downsampling begins
    MVAR(0)             #70
    LOAD()              #71
    MVAR(1)             #72
    STOR()              #73
    INC (S = 1, D = 1, C = 1)#74
    JUMP (Z = 1, N = 0, Reg2 = R2, Reg1 = SOR, T = 86)#75
    INC (S= 1, C = 1)#76
    #MVAI(COUN)
    JUMP (Z = 1, N = 0, Reg2 = R2, Reg1 = SOR, T = 86)#77
    #JUMP (Z = 1, N = 0, Reg2 = R3, Reg1 = AC, T= 76)
    JUMP (Z = 1, N = 0, Reg2 = R3, Reg1 = COUN, T = 80) #78Updated to
reflect increaded capabilites of instructions
    JUMP (Z=0, N=0, T = 69) #79

    ADD (SOR, R3)       #80
    RSET (C=1)          #81
    #------------------Test2-------------------------
    SUB(J = 0, K = 1, Reg1 = SOR)#82
    JUMP (Z = 1, N = 0, Reg2 = R2, Reg1 = SOR, T = 86 )#83
    INC(S = 1)#84
```

```
#-------------------Test2End-----------------------
#-----------------------------Test1-------------------------
'''
Fail
MVAI(SOR)   #83
MVAO(R1)    #84
MVAI(R3)    #85
JUMP (Z = 1, N = 1, Reg2 = R1, Reg1 = AC, T = 88)#86
'''
#----------------------------EndTest1-----------------------
JUMP (Z = 0, N = 0, T = 70)#85
END() #86

'''
#Testing Algorithm
RSET(1,1,1,1,1) #0
RSET(1,1,1,1,1) #1
INC(M=1)        #2
INC(M=1)        #3
INC(M=1)        #4
MVAI(MAR)       #5 AC=3
MVAO(MAR)       #6 MAR=3
INC(M=1)        #7 MAR=4
MVAI(MAR)       #8 AC=4
RSET(M=1)       #9 MAR=0
INC(M=1)        #10 MAR = 1
STOR()          #11Stores 4 in dram[1]
INC(M=1)        #12 MAR = 2
STOR()          #13 Stores 4 in dram [2]
MVAO(R3)        #14 R3 = 4
RSET(M=1)       #15 MAR=0
MVAI(MAR)       #16 AC=0
INC(M=1)        #17 MAR=1
LOAD()          #18 AC=4
JUMP(Z=1, Reg1 = AC, Reg2 = R3, T=0)#19
ADD(AC, R3)     #20
STOR()          #21
JUMP(T = 1)     #22
END()           #23
'''



#print ("Instructions Encoded")
return
```

```
def Print(code):
    global f
    #print (len(code))
    #global count
    #print (count, end=" ")

    #print ("    memory[",count,"] <=32'b",end='')
    #f.write("    memory[")
    #f.write(str(count))
    #f.write("]   = 32'b")
    for i in code:
        #print (i, end="")
        f.write(str(i))

    #print ()
    #f.write(';\n')
    f.write('\n')
    #count+=1
    return

def END():
    code = [0]*32
    code[0:5] = [0,0,0,1,1]
    Print (code)
    return
def NOP():
    #global PC,IR,MAR,AC,R1,R2,R3,SOR,DSTR,COUN
    code = [0]*32
    code[0:4] = [0,0,0,1]
    Print (code)
    return

def RSET(C=0,D=0,S=0,M=0,A=0):
    #global PC,IR,MAR,AC,R1,R2,R3,SOR,DSTR,COUN
    code = [0]*32
    code[0:4] = [0,0,1,0]
    code[5:10]=[C,D,S,M,A]
    Print (code)
    return

def LOAD(J=0, A=0):
    #global PC,IR,MAR,AC,R1,R2,R3,SOR,DSTR,COUN
    code = [0]*32
    code[0:4] = [0,0,1,1]
    if J ==0:
```

```
        Print (code)
        return
    else:
        code[4]=1
        A=[int(x) for x in str(bin(A)[2:])]
        for i in range (len(A)):
            code[-i-1] = A[-i-1]
        Print (code)
        return

def STOR(J=0, A=0):
    #global PC,IR,MAR,AC,R1,R2,R3,SOR,DSTR,COUN
    code = [0]*32
    code[0:4] = [0,1,0,0]
    if J ==0:
        Print (code)
        return
    else:
        code[4]=1
        A=[int(x) for x in str(bin(A)[2:])]
        for i in range (len(A)):
            code[-i-1] = A[-i-1]
        Print (code)
        return

def MVAR(J=0):
    #global PC,IR,MAR,AC,R1,R2,R3,SOR,DSTR,COUN
    code = [0]*32
    code[0:4] = [0,1,0,1]
    code[4]=J
    Print (code)
    return


def MVAO(Reg):
    code = [0]*32
    code[0:4] = [0,1,1,0]
    code[5:9] = Reg
    Print (code)
    return

def MVAI(Reg):
    code = [0]*32
    code[0:4] = [0,1,1,1]
    code[4]=int(not(Reg[0]))
    code[5:9] = Reg
```

```
    Print (code)
    return

def INC(C=0,D=0,S=0,M=0):
    code = [0]*32
    code[0:4] = [1,0,0,0]
    code[5:9]=[C,D,S,M]
    Print (code)
    return

def JUMP(N=0,Z=0,Reg1=NA,Reg2=NA,T=0):
    code = [0]*32
    code[0:4] = [1,1,0,0]
    code[5:9] = Reg1
    code[9:13] = Reg2
    code[13] = N
    code[14] = Z
    A=[int(x) for x in str(bin(T)[2:])]
    for i in range (len(A)):
        code[-i-1] = A[-i-1]
    Print (code)
    return

def ADD(Reg1, Reg2 = NA,J=1,  K=0):
    code = [0]*32
    code[0:4] = [1,0,0,1]
    code[4]=J
    code[5:9] = Reg1
    code[9:13]=Reg2
    A=[int(x) for x in str(bin(K)[2:])]
    for i in range (len(A)):
        code[-i-1] = A[-i-1]
    Print (code)
    return

def SUB(Reg1, Reg2 = NA, J=1, K=0):
    code = [0]*32
    code[0:4] = [1,1,1,1]
    code[4]=J
    code[5:9] = Reg1
    code[9:13]=Reg2
    A=[int(x) for x in str(bin(K)[2:])]
    for i in range (len(A)):
        code[-i-1] = A[-i-1]
    Print (code)
    return
```

```
def MUL(Reg1, Reg2 = NA, J=1, K=0):
    code = [0]*32
    code[0:4] = [1,1,0,1]
    code[4]=J
    code[5:9] = Reg1
    code[9:13]=Reg2
    A=[int(x) for x in str(bin(K)[2:])]
    for i in range (len(A)):
        code[-i-1] = A[-i-1]
    Print (code)
    return

def DIV(Reg1, Reg2 = NA, J=1, K=0):
    code = [0]*32
    code[0:4] = [1,1,1,0]
    code[4]=J
    code[5:9] = Reg1
    code[9:13]=Reg2
    A=[int(x) for x in str(bin(K)[2:])]
    for i in range (len(A)):
        code[-i-1] = A[-i-1]
    Print (code)
    return

def SFTR():
    code = [0]*32
    code[0:4] = [1,0,1,0]
    Print (code)
    return

def SFTL():
    code = [0]*32
    code[0:4] = [1,0,1,1]
    Print (code)
    return



def ToHex_Long(pixel):
    x = str(hex(pixel)[2:])
    x = "0"*(5 - len(x))+x
    return x

def DataEncoder(Image):
    line_width = len(Image[0])
```

```
    depth = len(Image)

    f = open("Const_Mem.mif", "w+")
    f.write("DEPTH = 8;\n") #CRAM Depth is 8 bits - needs only 3 bits
to get the reference
    f.write("WIDTH = 20;\n") #Width is 20 bits to hold the entire
address locations
    f.write("ADDRESS_RADIX=UNS;\nDATA_RADIX=HEX;\nCONTENT BEGIN\n")
    f.write("[0..7]: 00000;\n")

    f.write("3: 00000;\n") #First address of source image
    f.write("4: "+ToHex_Long(135000)+";\n") #First address of
destination Image
    f.write("5: "+ToHex_Long(depth*line_width-1)+";\n") #Last address
of source image
    f.write("6: "+ToHex_Long(line_width)+";\n") #Width of a Line
    f.write("END;\n")
    f.close()
    return

def ToHex(pixel):
    x = hex(pixel)[2:]
    if len(x) ==1:
        x = "0"+x
    return x

def ImageEncoder(Image):

    Image = np.array(Image)
    #print (Image)
    #print (len(Image))
    #print (len(Image[0]))
    Image = Image.flatten()

    depth = len(Image)
    if depth>135000:
        print ("Warning - Image too Big")
    f = open("Data_Mem.mif", "w+")
    f.write("DEPTH = 270000;\n")
    f.write("WIDTH = 8;\n")
    f.write("ADDRESS_RADIX=UNS;\nDATA_RADIX=HEX;\nCONTENT BEGIN\n")
    f.write("[0..269999]: 00;\n")

    for i in range (len(Image)):
        f.write(str(i)+": "+ToHex(Image[i])+";\n")
```

```
    f.write("END")
    f.close()
    return
'''
parser = argparse.ArgumentParser("Initialize Memory for Image
Downsampler")
parser.add_argument('Image', default = "Ironman.jpg",
                    help = "Select the image to be downsampled")
parser.add_argument("-size", nargs = 2, type = int, metavar =
('line_width', 'depth'), default = [256,256],
                    help = "Shape of Image to be downsampled. Will be
resized before downsampling. Default is the original Image size")
args = parser.parse_args()
'''
Image = cv.imread("Image.jpg")
print ("Loaded Image")
Image = cv.resize(Image, (256,256))
print ("Resized Image to 256, 256")
red,blue,green = cv.split(Image)

print ("Initializing Instruction Memory")
f = open("Inst_mem.mem", "w+")
InstructionEncoder()
f.close()
print ("Initializing Constant Memory")
DataEncoder(red)
print ("Initializing Data Memory")
ImageEncoder(red)
print ("All memories initialized. Compile and run downsampler")
```

9.2 Appendix 2 : Python code for getting the data, decoding the image, displaying the image and calculating the error

```
#%matplotlib notebook
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

f = open("Recieved_Memory_1.hex", "r")
depth = 254//2
line_width = 254//2

total_length = depth*line_width
lines = f.readlines()

image_decoded = np.zeros((15,15))
vector = []
i=135002

for j in range (total_length):
    a = lines[i+j][9:11]
    #print (a)
    a = '0x'+a
    pixel = int(a.encode(),16)
    vector.append(pixel)
f.close()
vector = np.array(vector)
vector = vector.reshape((depth,line_width))
red = vector.astype('uint8')
#plt.figure("Downsampled Image")
#plt.imshow(vector, cmap = 'gray')
#print (vector)

f = open("Recieved_Memory_2.hex", "r")

lines = f.readlines()

image_decoded = np.zeros((15,15))
vector = []
i=135002

for j in range (total_length):
```

```
    a = lines[i+j][9:11]
    #print (a)
    a = '0x'+a
    pixel = int(a.encode(),16)
    vector.append(pixel)
f.close()
vector = np.array(vector)
vector = vector.reshape((depth,line_width))
blue = vector.astype('uint8')
#plt.figure("Downsampled Image")
#plt.imshow(vector, cmap = 'gray')
#print (vector)

f = open("Recieved_Memory_3.hex", "r")

lines = f.readlines()

image_decoded = np.zeros((15,15))
vector = []
i=135002

for j in range (total_length):
    a = lines[i+j][9:11]
    #print (a)
    a = '0x'+a
    pixel = int(a.encode(),16)
    vector.append(pixel)
f.close()
vector = np.array(vector)
vector = vector.reshape((depth,line_width))
green = vector.astype('uint8')
#plt.figure("Downsampled Image")
#plt.imshow(vector, cmap = 'gray')
#print (vector)

Downsampled = cv.merge((red, blue, green))
plt.figure("Downsampled Image")
plt.imshow(Downsampled)

Image = cv.imread('Image.jpg')
Image2 = cv.resize(Image,(line_width*2,depth*2))
```

```
Image4 = cv.resize(Image, (line_width, depth))
#Image3 = np.zeros([width, length], dtype = 'uint8')
Image3 = cv.pyrDown(Image2, dstsize = (line_width,depth))
print (Image3)

mse = np.sqrt(((Downsampled - Image3)**2).mean())
print (mse)

#Image = cv.resize(Image,(14,14))

plt.figure("Original Image")
plt.imshow(Image3)

plt.show()

'''
Image = cv.imread("Ironman.jpg", 0)
#print (Image)
Image = np.array(Image)
#print (Image)
print (len(Image))
print (len(Image[0]))
Image = Image.flatten()
print (Image)
def ToHex(pixel):
    x = hex(pixel)[2:]
    if len(x) ==1:
        x = "0"+x
    return x
depth = len(Image)
f = open("Data_Mem.mif", "w+")
f.write("DEPTH = 500000;\n")
f.write("WIDTH = 8;\n")
f.write("ADDRESS_RADIX=UNS;\nDATA_RADIX=HEX;\nCONTENT BEGIN\n")
f.write("[0:499999]: 00;\n")
for i in range (len(Image)):
    f.write(str(i)+": "+ToHex(Image[i])+";\n")
f.close()
'''
```

9.3 Appendix 3 : Verilog code for PC register

```
module pc(      input [31:0] cbus_out,
                     input im_r, inc_en, clock,rst,
                     input [3:0] cbus_en,
                     input [1:0] status,
                     output wire [9:0] im_addr);

     reg [9:0] data = 10'b0;
     //reg [9:0] data = 10'b0;
     initial begin
          data = 0;
     end

/*
     always @(posedge clock)
          begin
               if (im_r == 1)
                    dataout <= datain;
          end
*/
     always @(negedge clock)
          begin
               if (inc_en)
                    data <= data + 1;
               if (cbus_en == 4'b1110)
                    data <= cbus_out[9:0];
               if (rst)
                    data <= 0;
          end

     assign im_addr = data;

endmodule
```

9.5 Appendix 4 : Verilog code for IR register

```
module ir (      input [31:0] im_out,
                         input [2:0] abus_en, bbus_en,
                         input im_r, clock,
                         output wire [31:0] bbus_in,
                         output wire [31:0] abus_in);

      reg [31:0] data = 32'b0;
      //reg [31:0] abus = 32'b0;
      //reg [31:0] bbus = 32'b0;
      initial begin
            data = 0;
      end
      /*
      always @(posedge clock)
            begin
                  if (bbus_en == 3'b111)
                        bbus[7:0] <= data[7:0];
                  if (abus_en == 3'b111)
                        abus <= data;
            end
      */
      always @(negedge clock)
            begin
                  if (im_r == 1)
                        data <= im_out;
            end

      assign abus_in = data;
      assign bbus_in = data[7:0];

endmodule
```

9.6 Appendix 5 : Verilog code for MAR register

```
module mar(      input [31:0] cbus_out,
                        input [2:0] abus_en,
                        input [3:0] cbus_en,
                        input inc_en, rst, clock, dm_wr, dm_r, cm_r,
                        output wire [19:0] dm_addr,
                        output wire [2:0] cm_addr,
                        output wire [31:0] abus_in,
                        output wire [8:0] dat);

/*      reg [31:0] abus = 0;
        reg [2:0] cm = 0;
        reg [19:0] dm = 0;*/
        reg [31:0] data = 0;
        initial begin
                data = 0;
        end
/*
        always @(posedge clock)
                begin
                        if (abus_en == 3'b100)
                                abus <= data;
                        if (dm_wr == 1)
                                dm <= data[19:0];
                        if (dm_r == 1)
                                dm <= data[19:0];
                        if (cm_r == 1)
                                cm <= data[2:0];
                end
*/
        always @(negedge clock)
                begin
                        if (rst == 1'b1)
                                data <= 32'b0;
                        if (inc_en == 1'b1)
                                data <= data + 1;
                        if (cbus_en == 4'b1100)
                                data <= cbus_out;
                end
        assign abus_in = data;
        assign dm_addr = data[19:0];
        assign cm_addr = data[2:0];
        assign dat = data[8:0];

endmodule
```

9.7 Appendix 6 : Verilog code for SOR register

```
module sor(     input [31:0] cbus_out,
                        input [2:0] abus_en,
                        input [3:0] cbus_en,
                        input rst, inc_en, clock,
                        output wire [31:0] abus_in);


        reg [31:0] data;
        initial begin
                data = 0;
        end
        /*
        reg [31:0] abus;

        always @(posedge clock)
                begin
                        if (abus_en == 3'b001)
                                abus <= data;
                end
                */
        always @(negedge clock)
                begin
                        if (rst)
                                data <= 32'b0;
                        if (inc_en)
                                data <= data + 1;
                        if (cbus_en == 4'b1001)
                                data <= cbus_out;
                end

        assign abus_in = data;

endmodule
```

9.8 Appendix 7 : Verilog code for DSTR register

```
module dstr(    input [31:0] cbus_out,
                           input [2:0] abus_en,
                           input [3:0] cbus_en,
                           input rst, inc_en, clock,
                           output wire [31:0] abus_in);

      //reg [31:0] abus;
      reg [31:0] data;

      initial begin
            data = 0;
      end
      /*
      always @(posedge clock)
            begin
                  if (abus_en == 3'b010)
                        abus <= data;
            end
      */
      always @(negedge clock)
            begin
                  if (rst)
                        data <= 32'd0;
                  if (inc_en)
                        data <= data + 1;
                  if (cbus_en == 4'b1010)
                        data <= cbus_out;
            end

      assign abus_in = data;

endmodule
```

9.9 Appendix 8 : Verilog code for COUN register

```
module coun(     input [2:0] abus_en,
                            input rst, inc_en, clock,
                            output wire [31:0] abus_in);

      reg [31:0] data = 0;
      //reg [31:0] abus = 0;
      initial begin
            data = 0;
      end
      /*
      always @(posedge clock)
            begin
                  if (abus_en == 3'b110)
                        abus <= data;
            end
      */
      always @(negedge clock)
            begin
                  if (rst)
                        data <= 32'b0;
                  if (inc_en)
                        data <= data + 1;
            end

      assign abus_in = data;

endmodule
```

9.10 Appendix 9 : Verilog code for AC register

```
module ac( input [31:0] cbus_out,
                      input [7:0] dm_out,
                      input [19:0] cm_out,
                      input dm_wr, rst, clock, dm_r, cm_r,
                      input [2:0] abus_en,
                      input [3:0] cbus_en,
                      output wire [31:0] abus_in,
                      output wire [8:0] dat,
                      output wire [7:0] dm_in);

     reg [31:0] data = 0;
     //reg [31:0] abus = 0;
     //reg [7:0] dm = 0;
     assign dat = data[8:0];

     initial begin
          data = 0;
     end
     /*
     always @(posedge clock)
          begin
               if (abus_en == 3'b011)
                    abus <= data;
               if (dm_wr)
                    dm <= data[7:0];
          end
     */
     always @(negedge clock)
          begin
               if (rst)
                    data <= 32'b0;
               if   (cbus_en == 4'b1011)
                    data <= cbus_out;
               if (dm_r)
                    data <= dm_out;
               if (cm_r)
                    data <= cm_out;
          end

     assign abus_in = data;
     assign dm_in = data[7:0];
endmodule
```

9.11 Appendix 10 : Verilog code for R1 register

```
module reg1(     input [31:0] cbus_out,
                            input clock,
                            input [3:0] cbus_en,
                            input [2:0] bbus_en,
                            output wire [31:0] bbus_in);


      reg [31:0] data = 0;
      //reg [31:0] b_out = 0;
      initial begin
            data = 0;
      end
      /*
      always @(posedge clock)
            begin
                  if (bbus_en == 3'b100)
                        b_out <= data;
            end
      */
      always @(negedge clock)
            begin
                  if (cbus_en == 4'b0100)
                        data <= cbus_out;
            end

      assign bbus_in = data;

endmodule
```

9.12 Appendix 11 : Verilog code for R2 register

```
module reg2(    input [31:0] cbus_out,
                           input clock,
                           input [3:0] cbus_en,
                           input [2:0] bbus_en,
                           output wire [31:0] bbus_in);
      reg [31:0] data = 0;
      initial begin
            data = 0;
      end
      /*
      reg [31:0] b_out = 0;

      always @(posedge clock)
            begin
                  if (bbus_en == 3'b101)
                        b_out <= data;
            end
      */
      always @(negedge clock)
            begin
                  if (cbus_en == 4'b0101)
                        data <= cbus_out;
            end

      assign bbus_in = data;

endmodule
```

9.13 Appendix 12 : Verilog code for R3 register

```
module reg3(     input [31:0] cbus_out,
                           input clock,
                           input [3:0] cbus_en,
                           input [2:0] bbus_en,
                           output wire [31:0] bbus_in);

     reg [31:0] data = 0;
     initial begin
          data = 0;
     end
     /*
     reg [31:0] b_out = 0;

     always @(posedge clock)
          begin
               if (bbus_en == 3'b110)
                    b_out <= data;
          end
     */
     always @(negedge clock)
          begin
               if (cbus_en == 4'b0110)
                    data <= cbus_out;
          end
     assign bbus_in = data;

endmodule
```

9.14 Appendix 13 : Verilog code for ALU

```
module alu(      input [31:0] abus_out, bbus_out,
                         input [2:0] alu_op,
                         output wire [31:0] cbus_in,
                         output wire n, z);


      parameter
      MOVA = 3'd0,
      MOVB = 3'd1,
      ADD = 3'd2,
      SUB = 3'd3,
      MUL = 3'd4,
      DIV = 3'd5,
      SHIFTR = 3'd6,
      SHIFTL = 3'd7;

      assign cbus_in =      (alu_op == MOVA)? abus_out:
                                 (alu_op == MOVB)? bbus_out:
                                 (alu_op == ADD)? abus_out + bbus_out:
                                 (alu_op == SUB)? abus_out - bbus_out:
                                 (alu_op == MUL)? abus_out * bbus_out:
                                 (alu_op == DIV)? abus_out / bbus_out:
                                 (alu_op == SHIFTR)? abus_out >> 1:
                                 (alu_op == SHIFTL)? abus_out << 1:32'd0;
      assign z = ~|cbus_in;
      assign n = cbus_in[31];

endmodule
```

```
/*
module alu(     input [31:0] abus_out, bbus_out,
                     input [2:0] alu_op,
                     input clock,
                     output wire [31:0] cbus_in,
                     output wire n, z);

     reg [31:0] result = 0;

     parameter
     MOVA = 3'd0,
     MOVB = 3'd1,
     ADD = 3'd2,
     SUB = 3'd3,
     MUL = 3'd4,
     DIV = 3'd5,
     SHIFTR = 3'd6,
     SHIFTL = 3'd7;

     always @(abus_out or bbus_out or alu_op)
          case (alu_op)
               MOVA: result <= abus_out;
               MOVB: result <= bbus_out;
               ADD: result <= abus_out + bbus_out;
               SUB: result <= abus_out - bbus_out;
               DIV: result <= abus_out / bbus_out;
               MUL: result <= abus_out * bbus_out;
```

```
                SHIFTR: result <= abus_out >> 1;
                SHIFTL:result <= abus_out << 1;

                //default: result <= 0;
            endcase

        assign z = ~|result;
        assign n = result[31];
        assign cbus_in = result;

endmodule
*/
```

9.15 Appendix 14 : Verilog code for Controller

```
// control unit

module controller(    input clock, n, z,
                                        input [5:0] uAdr,
                                        input [1:0] status,
                                        input [31:0] instruction,
                                        output reg [2:0] alu_op,
abus_en, bbus_en,
                                        output reg [3:0] cbus_en,
mem_en,
                                        output reg [4:0] inc_en,
                                        output reg [5:0] reset,
                                        output reg end_process);



      reg [5:0] present;
      reg [5:0] next;

      reg N, Z, uI; // these are not output from controller as these
are not used for any external logic from controller

      parameter
      FETCH1     =     6'd0,
      FETCHX     =     6'd1,
      FETCH2     =     6'd2,
      NOP        =     6'd3,
      RSET       =     6'd4,
      LOADR1     =     6'd5,
```

```
LOADR2    =    6'd6,
LOADK1    =    6'd7,
LOADK2    =    6'd8,
LOADK3    =    6'd9,
STORR1    =    6'd10,
STORR2    =    6'd11,
STORK1    =    6'd12,
STORK2    =    6'd13,
STORK3    =    6'd14,
MVARS     =    6'd15,
MVARD     =    6'd16,
JUMP      =    6'd17,
JMPX      =    6'd18,
MVACO     =    6'd19,
MVACA     =    6'd20,
MVACB     =    6'd21,
INC       =    6'd22,
ADDK      =    6'd23,
ADDR      =    6'd24,
SUBK      =    6'd25,
SUBR      =    6'd26,
MULK      =    6'd27,
MULR      =    6'd28,
DIVK      =    6'd29,
DIVR      =    6'd30,
SHFR      =    6'd31,
SFTL      =    6'd32,
END       =    6'd33,
```

```
IDLE       =    6'd34,
JMPXX      =  6'd35,
BEGIN      =    6'd36,
LOADR3     =    6'd37,
LOADK4     =    6'd38,
STORR3     =    6'd39,
STORK4     =    6'd40;

initial begin
     present <= IDLE;
     next <= IDLE;
end

always @ (negedge clock)
     present <= next;

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

always @ (posedge clock)
     case (present)

          IDLE: begin
```

```
        reset <= 6'b111111;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 0;
        if (status == 2'b01)
            next <= FETCH1;
        else
            next <= IDLE;
        end

    BEGIN: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 0;
        if (status == 2'b01)
```

```
            next <= IDLE;
        else
            next <= BEGIN;
        end

    FETCH1: begin
        reset <= 6'b000000;
        mem_en <= 4'b1000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;
        next <= FETCH2;
        end

    FETCH2: begin
        reset <= 6'b000000;
        mem_en <= 4'b1000;
        inc_en <= 5'b00001;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
```

```
        cbus_en <= 4'd0;
        uI <= 0;
        next <= uAdr;
        end

    NOP: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 0;
        next <= FETCH1;
        end

    RSET: begin
        reset [4:0]<= instruction[26:22];
        reset[5] <= 1'b0;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
```

```
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 0;
        next <= FETCH1;
        end

    LOADR1: begin
        reset <= 6'b000000;
        mem_en <= 4'b0010;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
            next <= LOADR2;
        end

    LOADR2: begin
        reset <= 6'b000000;
        mem_en <= 4'b0010;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
```

```
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
            next <= LOADR3;
        end

    LOADR3: begin
        reset <= 6'b000000;
        mem_en <= 4'b0010;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 0;

        //if (uI == 0)
            next <= FETCH1 ;
        end

    LOADK1: begin
        reset <= 6'b000000;
```

```
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd7;
        bbus_en <= 3'd0;
        cbus_en <= 4'd12;
        uI <= 1;

        //if (uI ==1)
             next <=    LOADK2;
        end

   LOADK2: begin
        reset <= 6'b000000;
        mem_en <= 4'b0100;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
             next <= LOADK3;
```

```
        end

    LOADK3: begin
        reset <= 6'b000000;
        mem_en <= 4'b0100;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
            next <= LOADK4;
        end

    LOADK4: begin
        reset <= 6'b000000;
        mem_en <= 4'b0100;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
```

```
        uI <= 0;

        //if (uI == 0)
        next <= FETCH1;
        end

    STORR1: begin
        reset <= 6'b000000;
        mem_en <= 4'b0001;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
            next <= STORR2;
        end

    STORR2: begin
        reset <= 6'b000000;
        mem_en <= 4'b0001;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
```

```
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
            next <= STORR3;
        end

    STORR3: begin
        reset <= 6'b000000;
        mem_en <= 4'b0001;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 0;

        //if (uI == 0)
            next <= FETCH1;
        end

    STORK1: begin
        reset <= 6'b000000;
```

```
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd7;
        bbus_en <= 3'd0;
        cbus_en <= 4'd12;
        uI <= 1;

        //if (uI == 1)
             next <= STORK2;
        end

    STORK2:    begin
        reset <= 6'b000000;
        mem_en <= 4'b0001;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
             next <= STORK3;
```

```
	end

STORK3:	begin
	reset <= 6'b000000;
	mem_en <= 4'b0001;
	inc_en <= 5'b00000;
	alu_op <= 3'd0;
	N <= 0;
	Z <= 0;
	abus_en <= 3'd0;
	bbus_en <= 3'd0;
	cbus_en <= 4'd0;
	uI <= 1;

	//if (uI == 1)
		next <= STORK4;
	end

STORK4: begin
	reset <= 6'b000000;
	mem_en <= 4'b0001;
	inc_en <= 5'b00000;
	alu_op <= 3'd0;
	N <= 0;
	Z <= 0;
	abus_en <= 3'd0;
	bbus_en <= 3'd0;
	cbus_en <= 4'd0;
```

```
        uI <= 0;

        //if (uI == 0)
             next <= FETCH1;
        end

    MVARS: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd1;
        bbus_en <= 3'd0;
        cbus_en <= 4'd12;
        uI <= 0;

        //if (uI == 0)
             next <= FETCH1;
        end

    MVARD: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
```

```
        Z <= 0;
        abus_en <= 3'd2;
        bbus_en <= 3'd0;
        cbus_en <= 4'd12;
        uI <= 0;

        //if (uI == 0)
            next <= FETCH1;
        end

    JUMP: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd7;
        bbus_en <= 3'd0;
        cbus_en <= 4'd14;
        uI <= 0;

        //if (uI == 0)
            next <= FETCH1;
        end

    JMPX: begin
        reset <= 6'b000000;
```

```
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd3;
        N <= instruction[18];
        Z <= instruction[17];
        abus_en <= instruction[25:23];
        bbus_en <= instruction[21:19];
        cbus_en <= 4'd0;
        uI <= 1;

        //if (uI == 1)
             next <= JMPXX;
        end

    JMPXX: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd3;
        N <= instruction[18];
        Z <= instruction[17];
        abus_en <= instruction[25:23];
        bbus_en <= instruction[21:19];
        cbus_en <= 4'd0;
        uI <= 1;

        if ((uI == 1) && (N == 1) && (n == 1))
             next <= JUMP;
```

```
	else if ((uI ==1) && (Z == 1) && (z == 1))
		next <= JUMP;

	else
		next <= FETCH1;
	end

MVACO: begin
	reset <= 6'b000000;
	mem_en <= 4'b0000;
	inc_en <= 5'b00000;
	alu_op <= 3'd0;
	N <= 0;
	Z <= 0;
	abus_en <= 3'd3;
	bbus_en <= 3'd0;
	cbus_en <= instruction[26:23];
	uI <= 0;

	//if (uI == 0)
		next <= FETCH1;
	end

MVACA: begin
	reset <= 6'b000000;
	mem_en <= 4'b0000;
	inc_en <= 5'b00000;
```

```
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= instruction[25:23];
        bbus_en <= 3'd0;
        cbus_en <= 4'd11;
        uI <= 0;

        //if (uI == 0)
            next <= FETCH1;
        end

    MVACB: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd1;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= instruction[25:23];
        cbus_en <= 4'd11;
        uI <= 0;

        //if (uI == 0)
            next <= FETCH1;
        end
```

```
INC: begin
	reset <= 6'b000000;
	mem_en <= 4'b0000;
	inc_en[4:1] <= instruction[26:23];
	inc_en[0]<=1'b0;
	alu_op <= 3'd0;
	N <= 0;
	Z <= 0;
	abus_en <= 3'd0;
	bbus_en <= 3'd0;
	cbus_en <= 4'd0;
	uI <= 0;

	//if (uI == 0)
		next <= FETCH1;
	end

ADDK: begin
	reset <= 6'b000000;
	mem_en <= 4'b0000;
	inc_en <= 5'b00000;
	alu_op <= 3'd2;
	N <= 0;
	Z <= 0;
	abus_en <= instruction[25:23];
	bbus_en <= 3'd7;
	cbus_en <= instruction[26:23];
	uI <= 0;
```

```
        //if (uI == 0)
            next <= FETCH1;
        end

    ADDR: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd2;
        N <= 0;
        Z <= 0;
        abus_en <= instruction[25:23];
        bbus_en <= instruction[21:19];
        cbus_en <= instruction[26:23];
        uI <= 0;

        //if (uI == 0)
            next <= FETCH1;
        end

    SUBK: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd3;
        N <= 0;
        Z <= 0;
```

```
        abus_en <= instruction[25:23];
        bbus_en <= 3'd7;
        cbus_en <= instruction[26:23];
        uI <= 0;

        if (uI == 0)
             next <= FETCH1;
        end

SUBR: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd3;
        N <= 0;
        Z <= 0;
        abus_en <= instruction[25:23];
        bbus_en <= instruction[21:19];
        cbus_en <= instruction[26:23];
        uI <= 0;

        if (uI == 0)
             next <= FETCH1;
        end

MULK: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
```

```
        inc_en <= 5'b00000;
        alu_op <= 3'd4;
        N <= 0;
        Z <= 0;
        abus_en <= instruction[25:23];
        bbus_en <= 3'd7;
        cbus_en <= instruction[26:23];
        uI <= 0;

        if (uI == 0)
            next <= FETCH1;
        end

    MULR: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd4;
        N <= 0;
        Z <= 0;
        abus_en <= instruction[25:23];
        bbus_en <= instruction[21:19];
        cbus_en <= instruction[26:23];
        uI <= 0;

        if (uI == 0)
            next <= FETCH1;
        end
```

```
DIVK: begin
      reset <= 6'b000000;
      mem_en <= 4'b0000;
      inc_en <= 5'b00000;
      alu_op <= 3'd5;
      N <= 0;
      Z <= 0;
      abus_en <= instruction[25:23];
      bbus_en <= 3'd7;
      cbus_en <= instruction[26:23];
      uI <= 0;

      if (uI == 0)
            next <= FETCH1;
      end

DIVR: begin
      reset <= 6'b000000;
      mem_en <= 4'b0000;
      inc_en <= 5'b00000;
      alu_op <= 3'd5;
      N <= 0;
      Z <= 0;
      abus_en <= instruction[25:23];
      bbus_en <= instruction[21:19];
      cbus_en <= instruction[26:23];
      uI <= 0;
```

```
        if (uI == 0)
            next <= FETCH1;
        end

    SHFR: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd6;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd3;
        bbus_en <= 3'd0;
        cbus_en <= 4'd11;
        uI <= 0;

        if (uI == 0)
            next <= FETCH1;
        end

    SFTL: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd7;
        N <= 0;
        Z <= 0;
```

```
        abus_en <= 3'd3;
        bbus_en <= 3'd0;
        cbus_en <= 4'd11;
        uI <= 0;

        if (uI == 0)
            next <= FETCH1;
        end

    END: begin
        reset <= 6'b111111;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
        N <= 0;
        Z <= 0;
        abus_en <= 3'd0;
        bbus_en <= 3'd0;
        cbus_en <= 4'd0;
        uI <= 0;
        next <= END;
        end

    default: begin
        reset <= 6'b000000;
        mem_en <= 4'b0000;
        inc_en <= 5'b00000;
        alu_op <= 3'd0;
```

```
                    N <= 0;
                    Z <= 0;
                    abus_en <= 3'd0;
                    bbus_en <= 3'd0;
                    cbus_en <= 4'd0;
                    uI <= 0;
                    next <= IDLE;
                    end
          endcase

endmodule
```

9.16 Appendix 15 : Verilog code for Top Processor

```
module top_processor( input wire clock,
                                                    input wire data_from_pc,

                                                    input wire start_process,
                                                    input wire start_transmit,
                                                    input wire rx,rd_clr,
                                                    //output wire rd_rx,
                                                    //output wire tx, tx_busy,
                                                    output [17:0] mar_ac_disp,

                                                    //output endImagereceived,
                                                    //output endProcess,
                                                    //output wire data_to_pc,

                                                    //output wire g1, g2, g3,
                                                    //output wire s00, s01,
s02, s03,
                                                    output clk,
                                                    output [7:0] pc_disp);




     // For processor
     wire [19:0] cm_out;
     wire [7:0] dm_processor;
     wire [31:0] im_out;
     wire [1:0] status;
     wire dm_r, dm_wr, cm_r, im_r;
```

```
	wire [7:0] ac_out;
	wire [9:0] pc_out;
	wire [17:0] c;
	wire [2:0] mar_cm;
	wire [19:0] mar_dm;
	wire end_process;
	wire slow_clk;
	wire e;
	assign clk = clock;
	assign mar_ac_disp = c;
	assign pc_disp= pc_out;

	assign endProcess = end_process;

	// Main Controller

	wire end_receive, end_transmit;		// signals to be given by
the communication module to change the state
	reg begin_process, begin_transmit;	// should be assigned

	assign endImagereceived = end_receive ;

	// Memory DRAM

	wire [7:0] data_in_com;				// data to the com
module to be transmitted
	wire [7:0] dm_in;						// data to
the dram
	wire [7:0] dm_out;
```

```
	// Muxes
	wire [19:0] tx_dm;				// address from transmitter module to write
	wire [19:0] dm_addr;				// data memory address ( from processor or from tx_addr)

	wire [19:0] rx_dm;

	// Tx and Rx
	wire [7:0] dm_transmitter;
	wire tx_en;
	wire tx_clk_en, rx_clk_en;
	//wire tx, tx_busy;
	wire [7:0] tx_data;
	wire [7:0] data_out_rx;				// data from receiver module
	wire en_com;						// to enable the communication

	//assign end_trasnmit = ~tx_busy;



	reg [9:0] process_switch_buffer = 10'd1001;	// waits 1023 clock cycles until the begin_process is being passed to main controller
	reg [9:0] transmit_switch_buffer = 10'd0;	// waits 1023 clock cycles until the transmit_begin is being passed to main controller

	always @(posedge clock)
```

```
    begin
        if (start_process )
        begin
            if (process_switch_buffer == 10'd1023 )
            begin
                process_switch_buffer <= process_switch_buffer ;
                begin_process <=1;
            end
            else
            begin
                process_switch_buffer <= process_switch_buffer +
10'd1;
                begin_process <=0;
            end
        end
        else
        begin
            process_switch_buffer <= 10'd1001;
            begin_process <= 0;
        end
    end

    always @(posedge clock)
    begin

        if (start_transmit )
        begin
            if (transmit_switch_buffer == 10'd1023 )
```

```
                begin
                        transmit_switch_buffer <= transmit_switch_buffer
;
                        begin_transmit <=1;
                end
                else
                begin
                        transmit_switch_buffer <= transmit_switch_buffer
+ 10'd1;
                        begin_transmit <=0;
                end
        end
        else
        begin
                transmit_switch_buffer <= 10'd0;
                begin_transmit <= 0;
        end
    end


/*
    cram            cram1(     .clock(slow_clk),
                                        .cm_r(cm_r),
                                        .cm_addr(mar_cm),
                                        .cm_out(cm_out));
*/
    C_ram           cram1(.address(mar_cm),
                                .clock(clock),
                                .rden(cm_r),
```

```
                                .q(cm_out));
iram            iram1(     .clock(clock),
                                     .im_r(im_r),
                                     .addr(pc_out),
                                     .instr_out(im_out));

Processor  cpu (      .clock(clock),
                                     .cm_out(cm_out),
                                     .dm_out(dm_processor),
                                     .im_out(im_out),
                                     .status(status),
                                     .dm_r(dm_r),
                                     .im_r(im_r),
                                     .cm_r(cm_r),
                                     .dm_wr(dm_wr),
                                     .dm_in(ac_out),
                                     .pc_out(pc_out),
                                     .mar_dm(mar_dm),
                                     .mar_cm(mar_cm),
                                     .end_process(end_process),
                                     .c(c));

dram  d_ram(     .address(mar_dm),
                           .clock(clock),
                           .data(ac_out),
                           .wren(dm_wr),
                           .rden(dm_r),
```

```
                                .q(dm_processor));


      main_control mc1(          .clock(clock),
                                          .en_com(en_com),
                                          .end_receive(end_receive),
                                          .end_process(end_process),

      .end_transmit(end_transmit),

      .begin_process(begin_process),

      .begin_transmit(begin_transmit),
                                          .status(status),
                                          .g1(g1), .g2(g2), .g3(g3),
                                          .s0(s00), .s1(s01),
.s2(s02), .s3(s03));

      slowclock clk1 (.inclk(clock),
                                     .outclk(slow_clk),
                                     .switch_clock(status));



endmodule
```

9.17 Appendix 16 : Verilog code for DRAM

```
module dram (
	address,
	clock,
	data,
	rden,
	wren,
	q);

	input	[18:0]  address;
	input	  clock;
	input	[7:0]  data;
	input	  rden;
	input	  wren;
	output	[7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
	tri1	  clock;
	tri1	  rden;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

	wire [7:0] sub_wire0;
	wire [7:0] q = sub_wire0[7:0];

	altsyncram altsyncram_component (
				.address_a (address),
				.clock0 (clock),
				.data_a (data),
				.rden_a (rden),
				.wren_a (wren),
				.q_a (sub_wire0),
				.aclr0 (1'b0),
				.aclr1 (1'b0),
				.address_b (1'b1),
				.addressstall_a (1'b0),
				.addressstall_b (1'b0),
				.byteena_a (1'b1),
				.byteena_b (1'b1),
				.clock1 (1'b1),
				.clocken0 (1'b1),
				.clocken1 (1'b1),
				.clocken2 (1'b1),
```

```
                      .clocken3 (1'b1),
                      .data_b (1'b1),
                      .eccstatus (),
                      .q_b (),
                      .rden_b (1'b1),
                      .wren_b (1'b0));
     defparam
          altsyncram_component.clock_enable_input_a = "BYPASS",
          altsyncram_component.clock_enable_output_a = "BYPASS",
          altsyncram_component.init_file = "Data_Mem.mif",
          altsyncram_component.intended_device_family = "Cyclone IV
E",
          altsyncram_component.lpm_hint =
"ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=1",
          altsyncram_component.lpm_type = "altsyncram",
          altsyncram_component.numwords_a = 270000,
          altsyncram_component.operation_mode = "SINGLE_PORT",
          altsyncram_component.outdata_aclr_a = "NONE",
          altsyncram_component.outdata_reg_a = "UNREGISTERED",
          altsyncram_component.power_up_uninitialized = "FALSE",
          altsyncram_component.read_during_write_mode_port_a =
"NEW_DATA_NO_NBE_READ",
          altsyncram_component.widthad_a = 19,
          altsyncram_component.width_a = 8,
          altsyncram_component.width_byteena_a = 1;


endmodule
```

9.18 Appendix 17 : Verilog code for IRAM

```
module iram(     input clock, im_r,
                              input [9:0] addr,
                              output wire [31:0] instr_out);


      reg [31:0] memory [128:0];
      reg [31:0] current_instruction;


      initial begin
            $readmemb("Inst_mem.mem", memory);
            /*
    memory[0]    = 32'b00100111110000000000000000000000;
    memory[1]    = 32'b00111000000000000000000000000011;
    memory[2]    = 32'b01100100100000000000000000000000;
    memory[3]    = 32'b00111000000000000000000000000100;
    memory[4]    = 32'b01100101000000000000000000000000;
    memory[5]    = 32'b00111000000000000000000000000101;
    memory[6]    = 32'b01100010100000000000000000000000;
    memory[7]    = 32'b00111000000000000000000000000110;
    memory[8]    = 32'b11110101100000000000000000000010;
    memory[9]    = 32'b01100011000000000000000000000000;
    memory[10]    = 32'b01010000000000000000000000000000;
    memory[11]    = 32'b00110000000000000000000000000000;
    memory[12]    = 32'b01100010000000000000000000000000;
    memory[13]    = 32'b10000000100000000000000000000000;
    memory[14]    = 32'b00110000000000000000000000000000;
    memory[15]    = 32'b10110000000000000000000000000000;
    memory[16]    = 32'b10011101101000000000000000000000;
```

```
memory[17]   = 32'b01100010000000000000000000000000;
memory[18]   = 32'b10000000100000000000000000000000;
memory[19]   = 32'b00110000000000000000000000000000;
memory[20]   = 32'b10011101101000000000000000000000;
memory[21]   = 32'b10100000000000000000000000000000;
memory[22]   = 32'b10100000000000000000000000000000;
memory[23]   = 32'b11000110001010100000000000100001;
memory[24]   = 32'b01011000000000000000000000000000;
memory[25]   = 32'b01000000000000000000000000000000;
memory[26]   = 32'b10000111000000000000000000000000;
memory[27]   = 32'b11000111001100100000000000011101;
memory[28]   = 32'b11000000000000000000000000001010;
memory[29]   = 32'b10000000100000000000000000000000;
memory[30]   = 32'b10000000100000000000000000000000;
memory[31]   = 32'b00100100000000000000000000000000;
memory[32]   = 32'b11000000000000000000000000001010;
memory[33]   = 32'b01011000000000000000000000000000;
memory[34]   = 32'b01000000000000000000000000000000;
memory[35]   = 32'b01110101000000000000000000000000;
memory[36]   = 32'b01100010100000000000000000000000;
memory[37]   = 32'b00111000000000000000000000000100;
memory[38]   = 32'b01100100100000000000000000000000;
memory[39]   = 32'b00111000000000000000000000000011;
memory[40]   = 32'b01100101000000000000000000000000;
memory[41]   = 32'b11000000000000000000000000101010;
memory[42]   = 32'b01010000000000000000000000000000;
memory[43]   = 32'b00110000000000000000000000000000;
memory[44]   = 32'b01100010000000000000000000000000;
```

```
memory[45]   = 32'b10011110001100000000000000000000;
memory[46]   = 32'b00110000000000000000000000000000;
memory[47]   = 32'b10110000000000000000000000000000;
memory[48]   = 32'b10011101101000000000000000000000;
memory[49]   = 32'b01100010000000000000000000000000;
memory[50]   = 32'b10011110001100000000000000000000;
memory[51]   = 32'b00110000000000000000000000000000;
memory[52]   = 32'b10011101101000000000000000000000;
memory[53]   = 32'b10100000000000000000000000000000;
memory[54]   = 32'b10100000000000000000000000000000;
memory[55]   = 32'b11000110001010100000000000111100;
memory[56]   = 32'b01011000000000000000000000000000;
memory[57]   = 32'b01000000000000000000000000000000;
memory[58]   = 32'b10000011000000000000000000000000;
memory[59]   = 32'b11000000000000000000000000101010;
memory[60]   = 32'b01011000000000000000000000000000;
memory[61]   = 32'b01000000000000000000000000000000;
memory[62]   = 32'b01110101000000000000000000000000;
memory[63]   = 32'b01100010100000000000000000000000;
memory[64]   = 32'b00111000000000000000000000000011;
memory[65]   = 32'b01100100100000000000000000000000;
memory[66]   = 32'b00111000000000000000000000000100;
memory[67]   = 32'b01100101000000000000000000000000;
memory[68]   = 32'b00100100000000000000000000000000;
memory[69]   = 32'b11000000000000000000000001000110;
memory[70]   = 32'b01010000000000000000000000000000;
memory[71]   = 32'b00110000000000000000000000000000;
memory[72]   = 32'b01011000000000000000000000000000;
```

```
    memory[73]   = 32'b01000000000000000000000000000000;
    memory[74]   = 32'b10000111000000000000000000000000;
    memory[75]   = 32'b11000100101010100000000001010110;
    memory[76]   = 32'b10000101000000000000000000000000;
    memory[77]   = 32'b11000100101010100000000001010110;
    memory[78]   = 32'b11000111001100100000000001010000;
    memory[79]   = 32'b11000000000000000000000001000101;
    memory[80]   = 32'b10011100101100000000000000000000;
    memory[81]   = 32'b00100100000000000000000000000000;
    memory[82]   = 32'b11110100100000000000000000000001;
    memory[83]   = 32'b11000100101010100000000001010110;
    memory[84]   = 32'b10000001000000000000000000000000;
    memory[85]   = 32'b11000000000000000000000001000110;
    memory[86]   = 32'b00011000000000000000000000000000;
*/
     end

     always @(posedge clock) begin
     if (im_r==1)
          current_instruction <= memory[addr];
     else
          current_instruction <= current_instruction;
     end

     assign instr_out = current_instruction;
endmodule
```

9.19 Appendix 18 : Verilog code for CRAM

```
module cram( input clock, cm_r,
                        input [2:0] cm_addr,
                        output reg [19:0] cm_out);

      reg [19:0] memory [7:0];

      initial begin
            //$readmemb("Const_Mem.mem", memory)
            /*
            memory[3] = 20'd0; // first address of source image
            memory[4] = 20'd10000; // first address of destination image
            memory[5] = 20'd8259; // last address of source image
            memory[6] = 20'd118; // width of the source image
            */
      end

      always @(posedge clock)
            begin
                  if (cm_r == 1)
                        cm_out <= memory[cm_addr];
            end
endmodule
```

10. References

1. Lecture notes of Dr. Jayathu Samarawickrama
2. Lecture notes of Dr. Ajitha Pasquel