Department of Electronics and Telecommunication Engineering

University of Moratuwa

EN3030 - ELECTRONIC CIRCUITS AND SYSTEM DESIGN

<u>PROCESSOR DESIGN REPORT</u>

Group-24

S.Prarththanan 180489E
Nerththiga.N 180418M
T.Mayooran 180391V
S.Vithurabiman 180588G

Contents

Abstract

FPGAs belong to a class of devices known as programmable logic, or sometimes referred to as programmable hardware. In this project, we down-sample a 256 * 256 image by a factor of 2. The input of for the FPGA is given as a text file while the output is retrieved as a text file too. To compare the outcome we analyze the FPGA down-sampled image to a python down-sampled image. Verilog is used as the hardware language (HDL). The designed processor and its Instruction Set Architecture is clearly discussed.

keywords : Processor Design, FPGA, Verilog, Down-sampling, ISA

1 Introduction

Objective of this project is to design a custom processor using FPGA, which could accept an image of size 256 x 256 as the input and down sample it to the size of 128 x 128. This report consists of the Algorithm we developed to perform down sampling, Instruction Set Architecture of our processor and other resources.

A central processing unit is an integrated electronic circuit that performs arithmetical, logical, input/output, and other instructions which are essential for the functionality of a computer. Microprocessor is a single chip that includes the functionality of CPU and some few new circuits integrated within. Design of a microprocessor occupies an important part to perform the required task successfully.

2 Algorithm

As we are required to down sample an image, it simply means selecting a few pixels which could represent the original image. So before picking up those selective pixels it is very important to filter the image in order to avoid the aliasing effect. Here we chose 2D Gaussian kernel to smoothen out the input image. So, we could pick the pixels for the down sampled image after filtering.

1	2	1
2	4	2
1	2	1

The 2D convolution of image (I) and Gausssian kernel(G) can be computed by the below equation.

$$I(x,y) * G(x,y) = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} I[n_1, n_1].G[x-n_1, y-n_2]$$

Picking up the pixels for downsampled image can be done as shown below. We would be able to produce a down sampled image of size 3 x 3 from an input image of size 8 x 8.

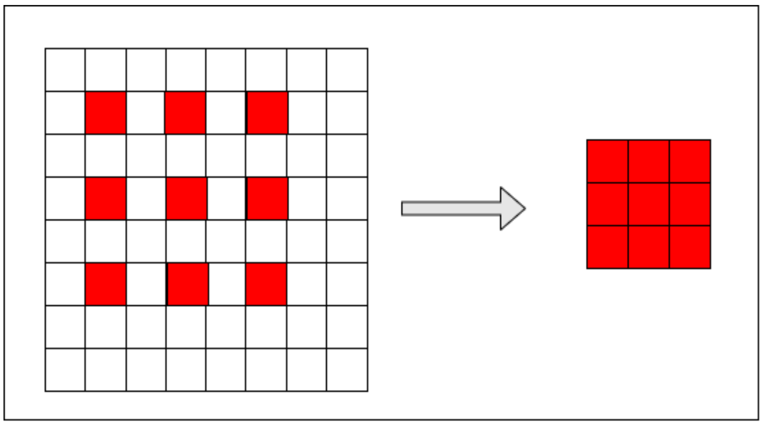


Figure 1: Downsampling Image

Further, image filtering and downsampling can be integrated together. This could be achieved by smoothening only the selective pixels that are to be used for downsampled image rather than smoothening each and every pixel of the input image. This would help to attain a less complex and comparatively fast processer through avoiding unwanted calculations.

As the first step, the input image should be converted into a text file containing its pixel values in a 1D array. This text file would be used to store the pixel values in data memory. Following that, the down sampling process would be performed by processor and finally the selective pixel values for down sampled image would be written in an output text.
Algorithm of the down sampling process.

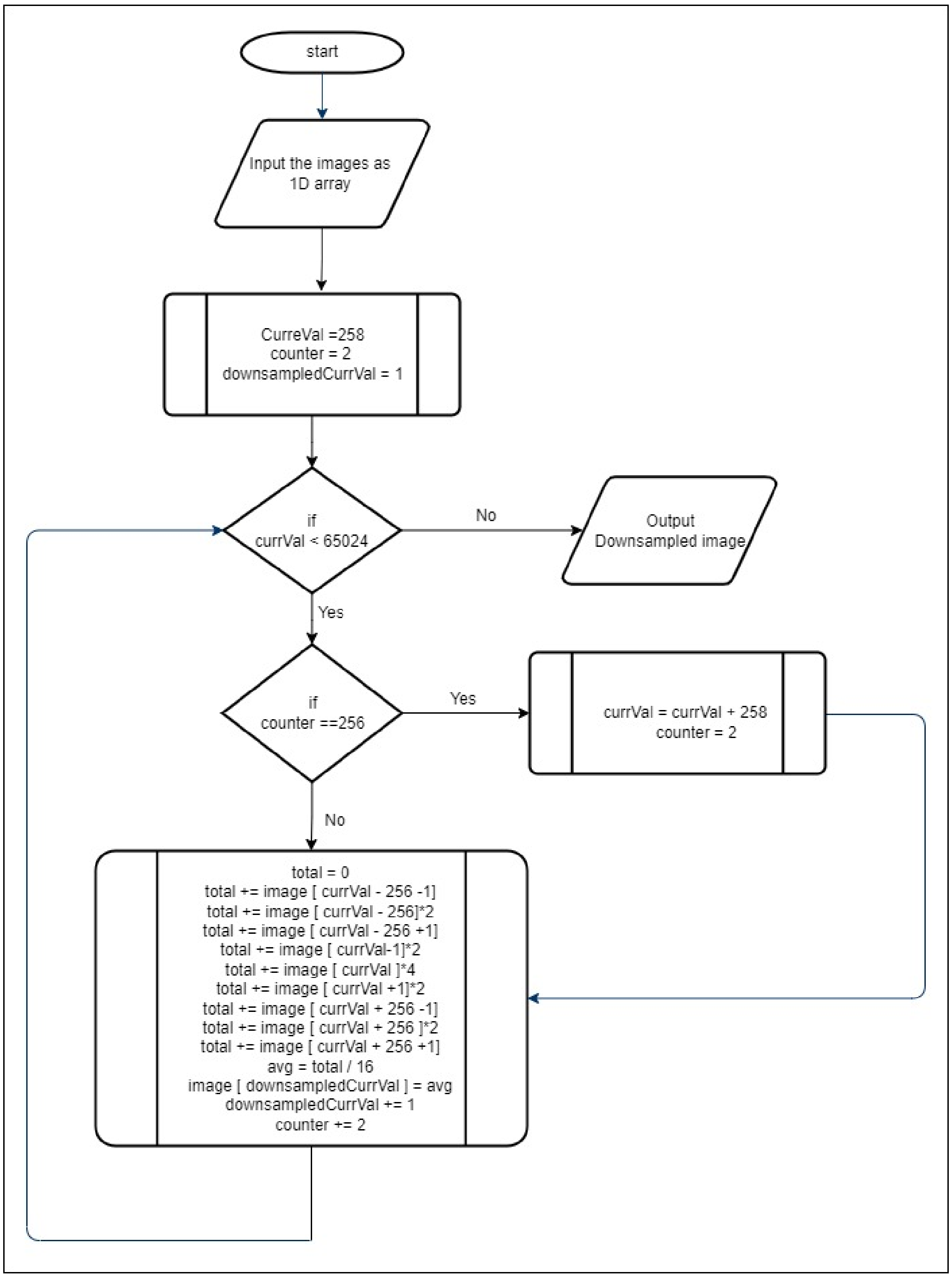

start
Input the images as 1D array
CurreVal =258
counter = 2
downsampledCurrVal = 1
if currVal < 65024
No
Output Downsampled image
Yes
if counter ==256
Yes
currVal = currVal + 258
counter = 2
No
total = 0
total += image [currVal - 256 -1]
total += image [currVal - 256]*2
total += image [currVal - 256 +1]
total += image [currVal-1]*2
total += image [currVal]*4
total += image [currVal +1]*2
total += image [currVal + 256 -1]
total += image [currVal + 256]*2
total += image [currVal + 256 +1]
avg = total / 16
image [downsampledCurrVal] = avg
downsampledCurrVal += 1
counter += 2

Figure 2: Flow chart of the algorithm.

3 Instruction Set Architecture

In simple terms, the ISA basically defines the communication between software and hardware of the computer.

- **Data Memory (DRAM):** DRAM is used to store the pixel values of input image. With 65536 memory locations and 8 bits width.
- **Instruction Memory:** ROM with 256 memory locations and 8 bits word size.
- **Memory Address Register (MAR):** It is a 16-bit address register, used to point the addresses in DRAM.
- **Program Counter (PC):** 8-bit register, used to point the address of next instruction in Instruction Memory.
- **Memory Buffer Register Unit (MBRU):** 8-bit register, used to store the location of instructions.
- **Arithmetic and Logical Unit (ALU):** A, B are input buses and C is the output bus.
- **Control Unit:** Used to generate control signals to processor according to MBRU input.
- **Accumulator:** 24-bit accumulator with direct access to ALU.
- **Registers (X, Y, Z, CV, DCV, C1):** 16-bit General Purpose Registers, used to store intermediate values.
- **Buses (A, B, C):** 16 bit wires, used for data transmissions between DRAM, IRAM, ALU and the Registers.

3.1 Datapath

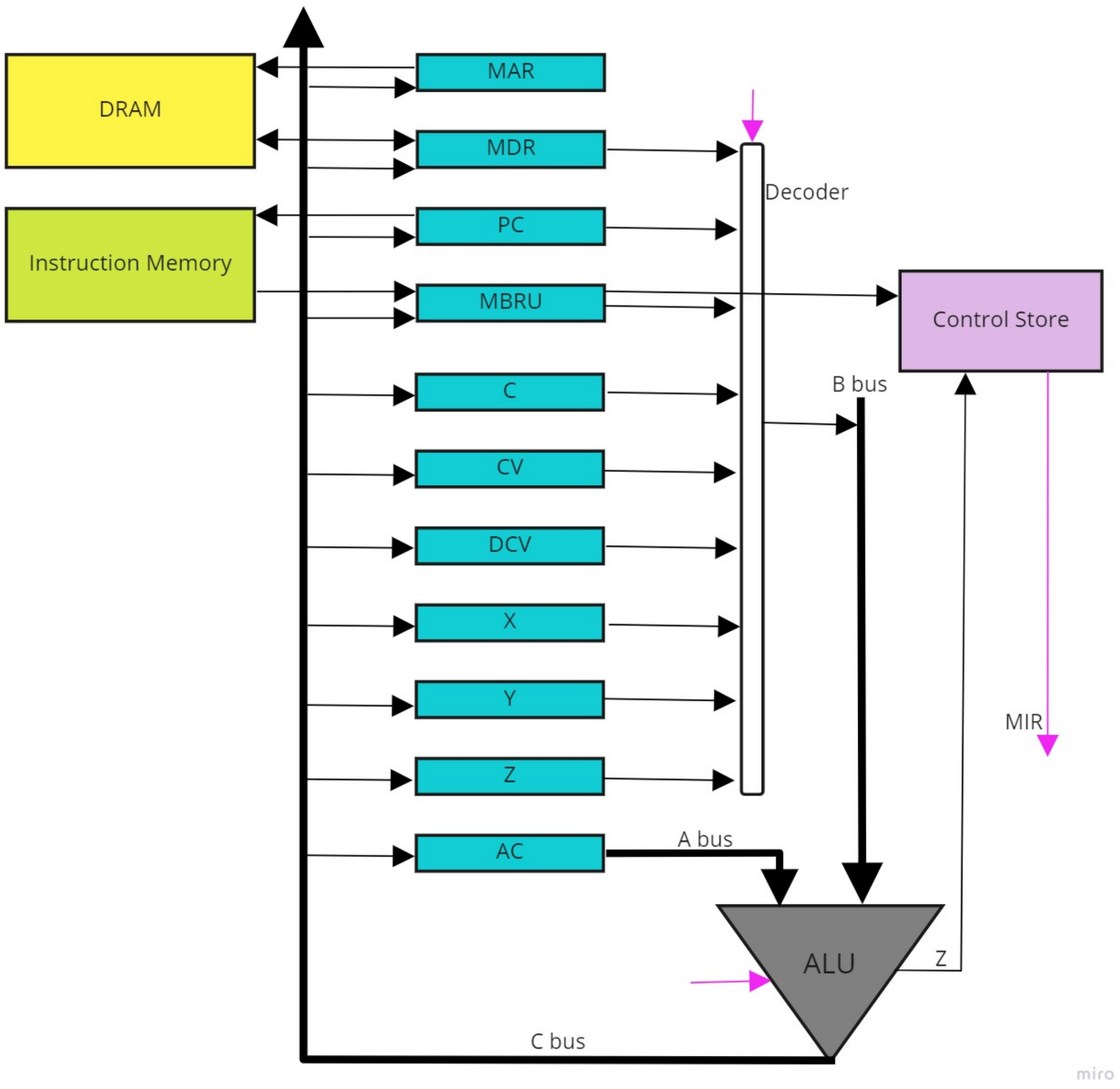
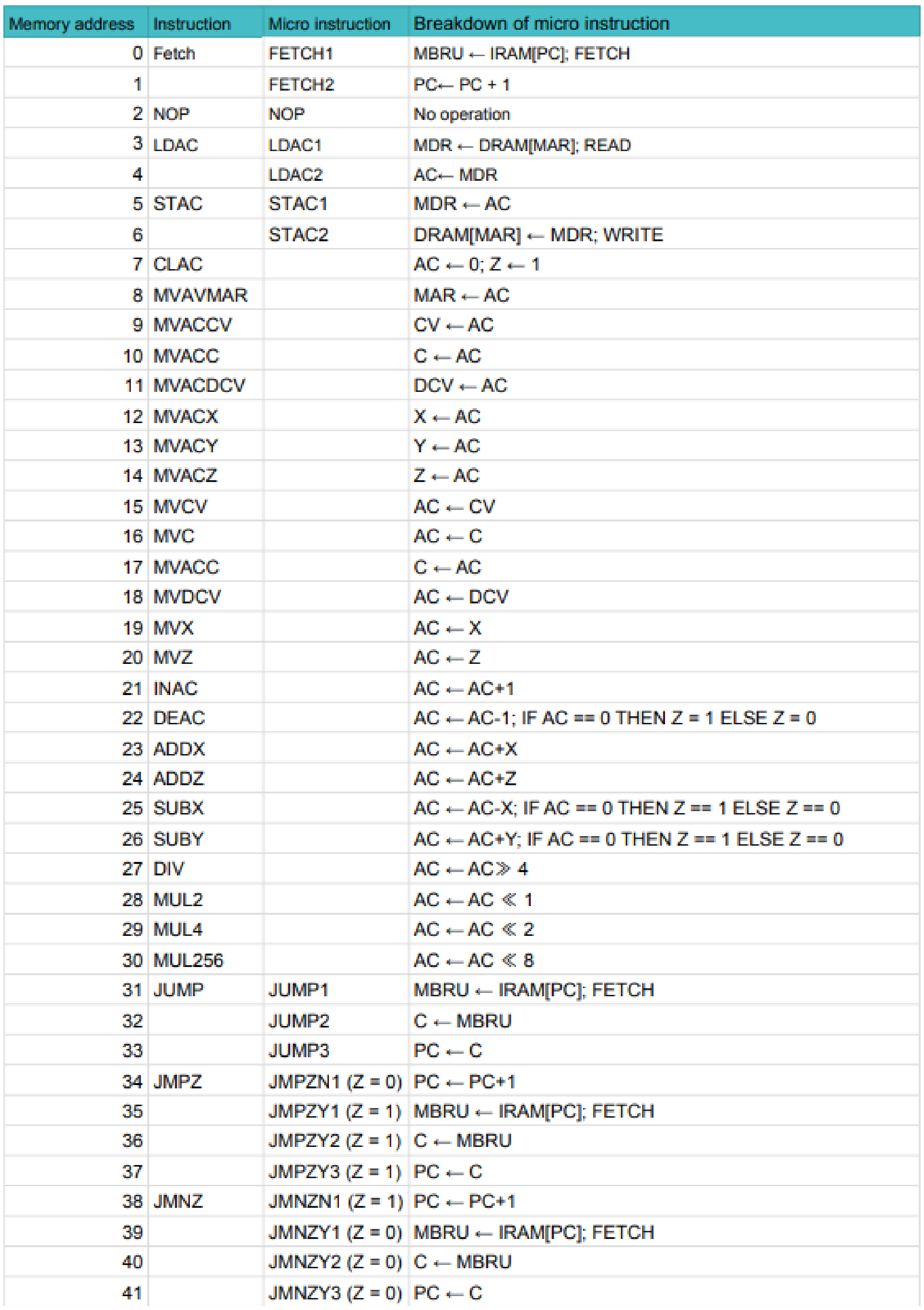


Figure 3: Data path of the processor

3.2 Instruction Set



Memory address	Instruction	Micro instruction	Breakdown of micro instruction
0	Fetch	FETCH1	MBRU ← IRAM[PC]; FETCH
1		FETCH2	PC← PC + 1
2	NOP	NOP	No operation
3	LDAC	LDAC1	MDR ← DRAM[MAR]; READ
4		LDAC2	AC← MDR
5	STAC	STAC1	MDR ← AC
6		STAC2	DRAM[MAR] ← MDR; WRITE
7	CLAC		AC ← 0; Z ← 1
8	MVAVMAR		MAR ← AC
9	MVACCV		CV ← AC
10	MVACC		C ← AC
11	MVACDCV		DCV ← AC
12	MVACX		X ← AC
13	MVACY		Y ← AC
14	MVACZ		Z ← AC
15	MVCV		AC ← CV
16	MVC		AC ← C
17	MVACC		C ← AC
18	MVDCV		AC ← DCV
19	MVX		AC ← X
20	MVZ		AC ← Z
21	INAC		AC ← AC+1
22	DEAC		AC ← AC-1; IF AC == 0 THEN Z = 1 ELSE Z = 0
23	ADDX		AC ← AC+X
24	ADDZ		AC ← AC+Z
25	SUBX		AC ← AC-X; IF AC == 0 THEN Z == 1 ELSE Z == 0
26	SUBY		AC ← AC+Y; IF AC == 0 THEN Z == 1 ELSE Z == 0
27	DIV		AC ← AC≫ 4
28	MUL2		AC ← AC ≪ 1
29	MUL4		AC ← AC ≪ 2
30	MUL256		AC ← AC ≪ 8
31	JUMP	JUMP1	MBRU ← IRAM[PC]; FETCH
32		JUMP2	C ← MBRU
33		JUMP3	PC ← C
34	JMPZ	JMPZN1 (Z = 0)	PC ← PC+1
35		JMPZY1 (Z = 1)	MBRU ← IRAM[PC]; FETCH
36		JMPZY2 (Z = 1)	C ← MBRU
37		JMPZY3 (Z = 1)	PC ← C
38	JMNZ	JMNZN1 (Z = 1)	PC ← PC+1
39		JMNZY1 (Z = 0)	MBRU ← IRAM[PC]; FETCH
40		JMNZY2 (Z = 0)	C ← MBRU
41		JMNZY3 (Z = 0)	PC ← C

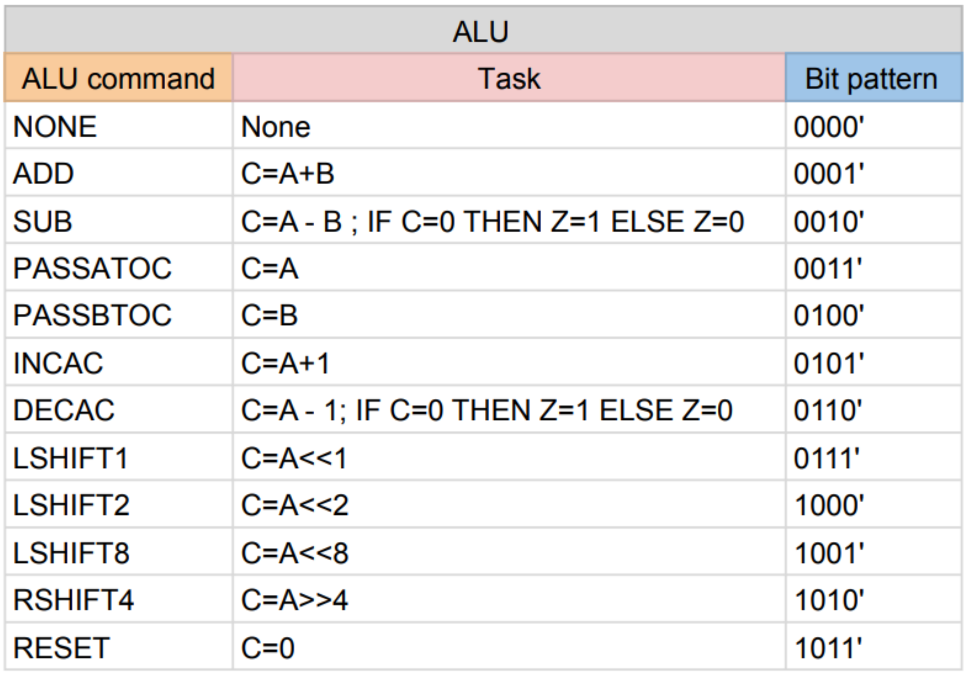
Figure 4: Instruction set

3.3 Control Unit

Control unit contains all control signals in Look Up table as bit patterns. In our purpose each control signal consists of 30 bits.

3.4 ALU

Here A bus and B bus are the two inputs of ALU. Here AC is directly connected through A bus. C bus and Z flag are the two outputs. Control signal for the ALU consists of 4 bits, denoting the tasks need to be performed.

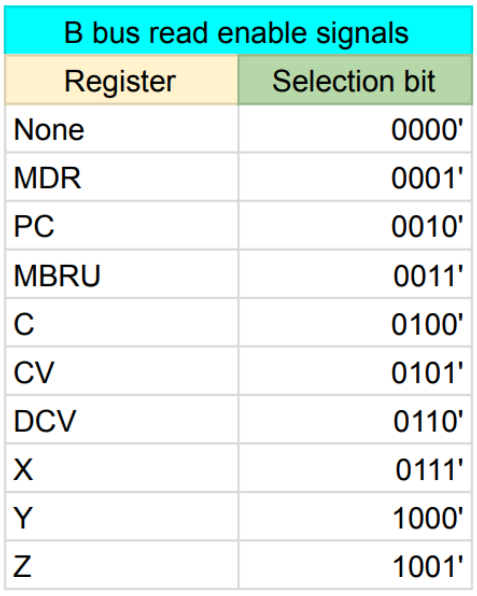


	ALU	
ALU command	Task	Bit pattern
NONE	None	0000'
ADD	C=A+B	0001'
SUB	C=A - B ; IF C=0 THEN Z=1 ELSE Z=0	0010'
PASSATOC	C=A	0011'
PASSBTOC	C=B	0100'
INCAC	C=A+1	0101'
DECAC	C=A - 1; IF C=0 THEN Z=1 ELSE Z=0	0110'
LSHIFT1	C=A<<1	0111'
LSHIFT2	C=A<<2	1000'
LSHIFT8	C=A<<8	1001'
RSHIFT4	C=A>>4	1010'
RESET	C=0	1011'

Figure 5: ALU control signal breakdown

3.5 Mux for B bus

B bus would get inputs from multiple registers such as PC, MDR, MBRU, X, Y, Z, C1, CV and DCV. So there should be a multiplexer to ensure that B bus accepts only one input at a time. Therefore, read enable signal for B bus should be decided from the selection bit of the multiplexer.



B bus read enable signals	
Register	Selection bit
None	0000'
MDR	0001'
PC	0010'
MBRU	0011'
C	0100'
CV	0101'
DCV	0110'
X	0111'
Y	1000'
Z	1001'

Figure 6: Read enable signals for B bus

3.6 Assembly code of the Algorithms

The assembly code of our algorithm should be stored in Instruction ROM of the processor. So that the PC could point those instructions that need to be executed.

```
CLAC
MVACMAR
LDAC

CLAC
MVACMAR
LDAC
INAC
MVACX
MVX
INAC
INAC
MVACCV
CLAC
INAC
MVACDCV
CLAC
INAC
INAC
MVACC

MVX
```

```
DEAC
DEAC
MUL256
MVACY

CLAC
MVACZ
MVCV
SUBL
DEAC
MVACMAR
LDAC
ADDT MVACZ
MVCV
SUBL
MVACMAR
LDAC
MUL2
ADDT
MVAC
MVCV
SUBL
INAC
MVACMAR
DAC
ADDT
MVACZ
MVCV
DEAC
MVACMAR
LDAC
MUL2
ADDT
MVACZ
MVCV
MVACMAR
LDAC
MUL4
ADDT
MVACZ
MVCV
INAC
MVACMAR
LDAC
MUL2
ADDT
MVACZ
MVCV
ADDL
DEAC
MVACMAR
LDAC
ADDT
```

```
MVACZ
MVCV
ADDL
MVACMAR
LDAC
MUL2
ADDT
MVACZ
MVCV
ADDL
INAC
MVACMAR
LDAC
ADDT
MVACZ

MVZ
DIV
MVDCV
MVACMAR
STAC

MVDCV
INAC
MVACDCV
MVC
INAC
INAC
MVACC
MVCV
INAC
INAC
MVACCV
MVCV
SUBE
JMPZ;
L1

MVC
SUBL
JMNZ
L2
MVCV ADDL
INAC
INAC
MVACCV; CLAC
INAC
INAC
MVACC
JUMP
L2
NOP
```

3.7 Instruction Cycle

Instruction cycle defines the basic operating mechanism of a computer, which consists of 3 stages in it. This cycle would continue until all instructions are finished.

- **FETCH:**
  Fetch retrieves required instructions from the specific location. It fetches the instruction denoted by PC from Instruction ROM and stores it in MBRU. PC would get incremented after each cycle of FETCH to point the next instruction need to be fetched.

FETCH1	MBRU ← IRAM[PC]; FETCH
FETCH2	PC← PC + 1

- **DECODE:**
  In the decoding phase, the fetched instruction in MBRU would be sent to Control Unit. Control unit would decode the instruction and generate corresponding control signals.

- **EXECUTE**

  1. **NOP**
     This means No Operation and is used when there is a need of waiting cycle in order to get the processed data at end point.
  2. **LDAC**
     This means loading AC with the data in DRAM. First the data in the location pointed by MAR would be read and loaded in MDR. Then it would be loaded to AC.

LDAC1	MDR ← DRAM[MAR]; READ
LDAC2	AC← MDR

  3. **STAC**
     Here the data in AC would be loaded to MDR first and then written to the address pointed by MAR in DRAM.

STAC1	MDR ← AC
STAC2	DRAM[MAR] ← MDR; WRITE

  4. **CLAC**
     AC would be cleared and set to zero. Zero flag would be issued.

CLAC	AC ← 0; Z ← 1

  5. **MVACMAR, MVACCV, MVACC, MVACDCV, MVACX, MVACY, MVACZ**
     These instructions mean to move the data in AC to the specific registers.

MVAVMAR	MAR ← AC
MVACCV	CV ← AC
MVACC	C ← AC
MVACDCV	DCV ← AC
MVACX	X ← AC
MVACY	Y ← AC
MVACZ	Z ← AC

6. **MVCV, MVC, MVDCV, MVX, MVZ**
   These instructions mean to move the data in specific registers to AC.

MVCV	AC $\leftarrow$ CV
MVC	AC $\leftarrow$ C
MVACC	C $\leftarrow$ AC
MVDCV	AC $\leftarrow$ DCV
MVX	AC $\leftarrow$ X
MVZ	AC $\leftarrow$ Z

7. **INAC**
   Here the value in AC would be incremented by 1.

   INAC AC $\leftarrow$ AC+1

8. **DEAC**
   The value in AC would be decremented by 1. If that value set to zero, then zero flag would be issued. (Z = 1)

   DEAC AC $\leftarrow$ AC-1; IF AC == 0 THEN Z = 1 ELSE Z = 0

9. **ADDX, ADDZ**
   Here the value in specific register would be added with the value in AC and the result would be loaded to AC.

ADDX	AC $\leftarrow$ AC+X
ADDZ	AC $\leftarrow$ AC+Z

10. **SUBX, SUBY**
    Here the value in specific register would be subtracted from the value in AC and result would be loaded to AC.

SUBX	AC $\leftarrow$ AC-X; IF AC == 0 THEN Z == 1 ELSE Z == 0.
SUBY	AC $\leftarrow$ AC+Y; IF AC == 0 THEN Z == 1 ELSE Z == 0.

11. **DIV**
    Here the value in AC would be divided by 16. This is equivalent to shift the binary value to right by 4 digits.

    DIV AC $\leftarrow$ AC$\gg$ 4

12. **MUL2, MUL4, MUL256**
    These instructions mean to multiply the value in AC by 2, 4, 256 respectively. So it is equivalent to left by 1, 2, 8 digits respectively.

MUL2	AC $\leftarrow AC \ll 1$
MUL4	AC $\leftarrow AC \ll 2$
MUL256	AC $\leftarrow AC \ll 8$

13. **JUMP**
    Here the address pointed by the current PC would be read from IRAM and loaded to MBRU. Then it would be loaded to the register C and then to the PC. So that

specific location could be fetched in the next fetch cycle.

JUMP1	MBRU $\leftarrow$ IRAM[PC]; FETCH
JUMP2	C $\leftarrow$ MBRU
JUMP3	PC $\leftarrow$ C

14. **JMPZ**
    This would check the Zero flag. If Z = 0, then the PC value would be incremented by 1. If Z = 1, then it would proceed steps similar to JUMP instruction.

JMPZN1 (Z = 0)	PC $\leftarrow$ PC+1
JMPZY1 (Z = 1)	MBRU $\leftarrow$ IRAM[PC]; FETCH
JMPZY2 (Z = 1)	C $\leftarrow$ MBRU
JMPZY3 (Z = 1)	PC $\leftarrow$ C

15. **JMNZ**
    Here this would check zero flag. If Z = 1, then the PC value would be incremented by 1. If Z = 0, then it would proceed steps similar to JUMP instruction..

JMNZN1 (Z = 1)	PC $\leftarrow$ PC+1
JMNZY1 (Z = 0)	MBRU $\leftarrow$ IRAM[PC]; FETCH
JMNZY2 (Z = 0)	C $\leftarrow$ MBRU
JMNZY3 (Z = 0)	PC $\leftarrow$ C

3.8 State Diagram

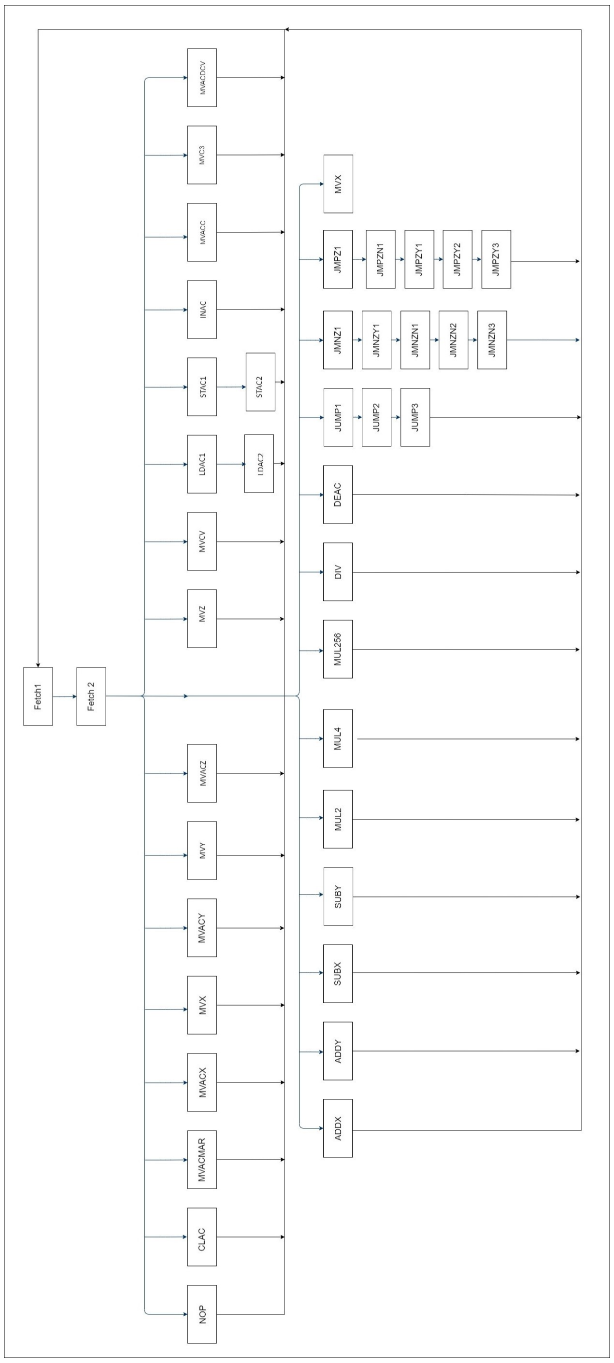


Figure 7: State Diagram

4 RTL Modules

4.1 Processor.

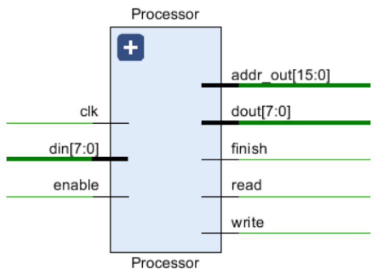


Figure 8: Processor

- **Inputs:**
  - **Clk** : Generated clock input.
  - **enable** : To activate the processor unit.
  - **din**: To input data to be processed.
- **Outputs:**
  - **addr out:** The location in the memory to which the output data should be written.
  - **dout:** Data to be written
  - **read:** READ enable signal.
  - **write:** WRITE enable signal.
  - **finish:** This indicates the user that the processor has completed its task.
- **Processor consists of following modules which are vital for the down sampling.**
  - ALU.
  - Control store.
  - PC.
  - IRAM.
  - MBRU.
  - Decoder.
  - 7 GPR.
  - MDR.
  - MAR.

All the registers used inside the processor can be overlooked in the following table.

Register	Size	Inputs	Outputs
GPR	24-bit	Enable, Data In	Data out
AC	24-bit	Enable, Data In	ALU
MAR	16-bit	Enable, Data In	Data memory
MDR	8-bit	Enable, Data In, Data memory	Data memory
PC	8-bit	Enable, Data In, Inc	Data out, Instruction memory
MBDU	8-bit	Enable, Data In, Instruction Memory	Data out, Control unit

Table 1: Registers

4.1.1 General Purpose Register

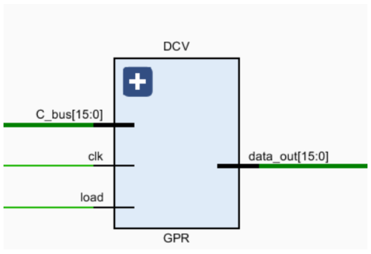


Figure 9: GPR

- **Purpose:** Store intermediate values of data during processing.
- **Input Strategy:** When the enabling bit (EN) is set to 1, the data available at the input to the register will be written to the register at the next proceeding positive edge of the clock.
- **Output Strategy:** A demultiplexer is placed to select which register can output data to the bus line.

4.1.2 Accumulator

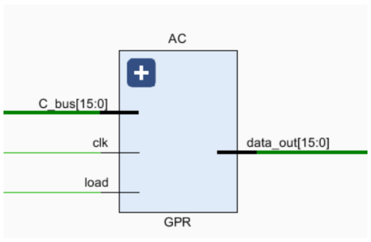


Figure 10: AC

- **Purpose:** Provides a direct input to the ALU.
- Accumulator has a CLAC command to clear the current register value. This command is essential to reduce the number of clock cycles and the complexity of architecture. The AC is used in almost all the ALU operation because of the direct Data in to the ALU.

.

4.1.3 Memory Address Register (MAR)

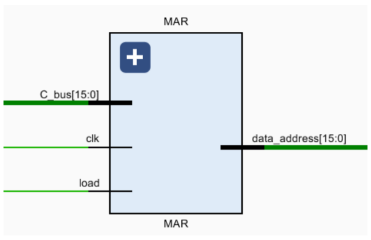


Figure 11: MAR

- **Purpose:** The locations in Data Memory to which data is required to be written will be provided by MAR while the data will be available at the MDR.
- **Output Strategy:** Data out of MAR is directly connected to the B bus without a Demultiplexer.

4.1.4 Program Counter (PC)

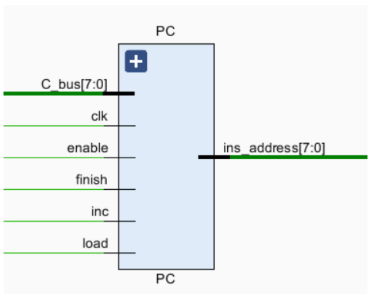


Figure 12: PC

- **Purpose:** This stores the address of the next instruction to be executed.
- **Input Strategy:** The address is incremented by one if the INC is kept HIGH at a positive clock edge.
- **Output Strategy:** 'INC' to increment the address by one so that it is not necessary for the address to be sent to the ALU for incrementing.

4.1.5 Memory Buffer Register Unit (MBRU)

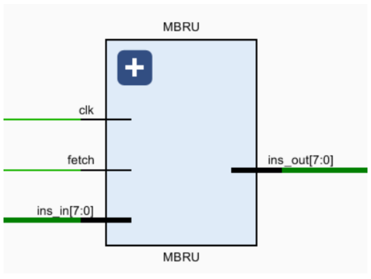


Figure 13: MBRU

- **Purpose:** This is to keep the location of an instruction to be decoded by the control Unit in order to generate relevant control signals.
- **Input Strategy:** If fetch is HIGH at the positive edge of the clock, then the location of an instruction is read and loaded from the Instruction Memory.

- **Output Strategy:** This sends an address of an instruction which is later decoded by the instruction store.

4.1.6 Instruction ROM (IRAM)

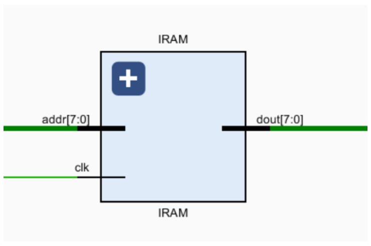


Figure 14: IRAM

- **Purpose:** Retrieve the location of the instruction which has to executed to the MBRU once PC requests it.

- **Input:**
  - **clk:** Generated clock input.
  - **address:** The counter keeps track of the next instruction which has to be executed next.
- **Output:**

– **d**$_{o}ut : Theaddressofthenextinstructionwhichhastobeexecutednext.$

4.1.7 Control Unit

- **Purpose:** This retrieves the hard wired control signals which is to be routed to each registers in order to perform a certain instruction.
- **Input:**
  - **MBRU :** Fetched address of the instruction which has to be executed next.
  - **clk:** Generated clock input.
  - **Z flag:** the result from the ALU whether an operation returns zero or not.
  - **address:** Address of the next micro instruction, if the previous instruction leads to another micro instruction.
  - **enable:** TO enable the module.
- **Output:**
  - **MIR:** 30 bit register which is hardwired to required modules in order to perform an instruction.
  - **finish:** A flag to indicate the end of the operation.

4.1.8 Decoder

- **Purpose:** This module is a multiplexer to pass the outputs of the registers to the B bus one at a time.
- **Input:**

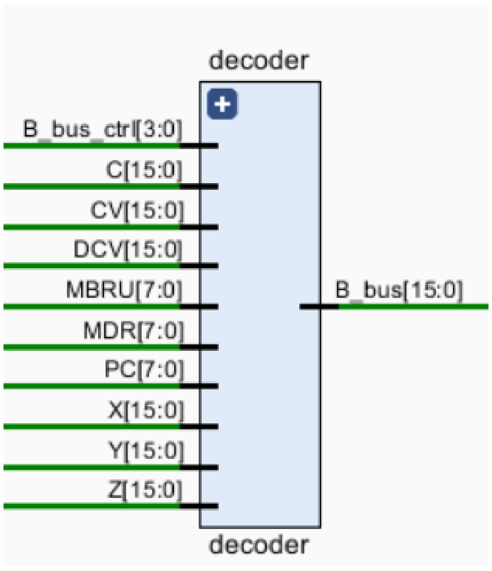


Figure 15: Decoder

  - All the outputs of the registers shown in the figure.
  - B bus control (selection bit)

- **Output:**
  - B bus

4.1.9 Overview of the processor

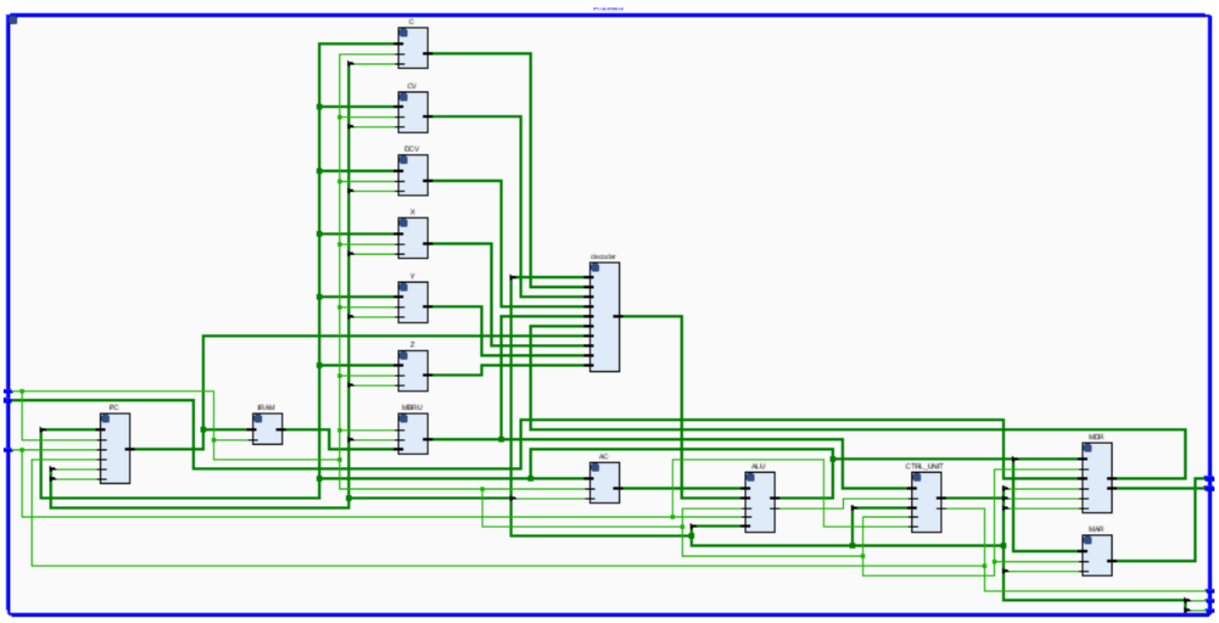


Figure 16: Processor

4.2 Data RAM (DRAM)

- **Purpose:** Keeps the actual data (in our example, the image) inside it, to be processed by the processor.
- **Input:**

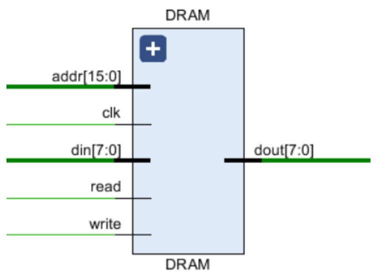


Figure 17: DRAM

- **addr:** Address of the memory location to which data should be written/read from provided by the user or processor.
- **clk:** Generated clock input.
- **din:** 8-bit long set of data to be written to the DRAM at the provided location.
- **read:** READ enable signal from the user or processor.
- **write:** WRITE enable signal from the user or processor.

- **Output:**
  - **dout:** A byte of data read from the DRAM (May be the input to the Processor or to the user selected accordingly from a demultiplexer).

4.3 Clock Generator

The architecture of the processor is inherently depended on a clock. The designed processor must read data from the memory, process them through the ALU and write the processed data back in relevant locations within the negative edge and the positive edge of the clock. Clock with 20 ns time period is used here because a clock with every possible time period is not available in industry. Depending on that high frequency clock, we are generating a clock with a time period of 160ns.

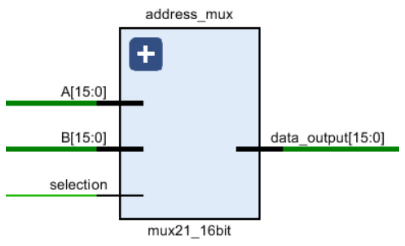
More importantly, we implement this clock only in test bench. The reason is in FPGAs there is a clock exists as a hardware.

4.4 Multiplexers and Demultiplexers

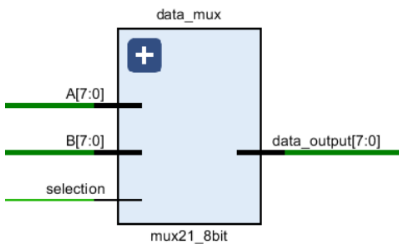
Multiplexer means many into one. A multiplexer is a circuit used to select and route any one of the several input signals to a single output.Few types of multiplexer are 2-to-1, 4-to-1, 8-to-1 and 16-to-1 multiplexer.

In our architecture, we use multiplexers in only one size : 2 to 1 MUX.

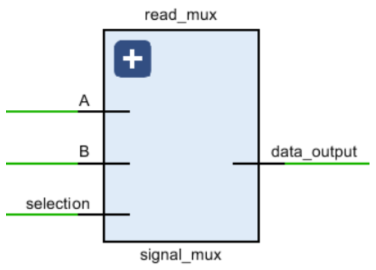
- **address mux:** This is to route the address from the user or from the processor to the DRAM.
- **data mux:** This is to route the data from the user or from the processor to the DRAM to write.
- **read mux:** This is to route the read enable signal given by the user or by the processor to the DRAM.
- **write mux:** This is to route the write enable signal given by the user or by the processor to the DRAM.



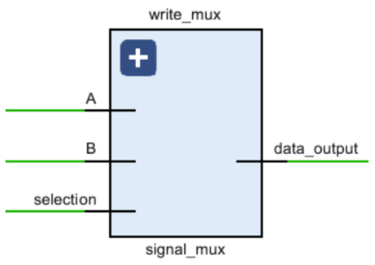
((a)) address mux



((b)) data mux



((c)) read mux



((d)) write mux

Figure 18: Multiplexers

Demultiplexer means one to many. A demultiplexer is a circuit with one input and many outputs. By applying control signal, we can steer any input to the output. Few types of demultiplexer are 1-to 2, 1-to-4, 1-to-8 and 1-to 16 demultiplexer.
Here we use only one demultiplexer.

- **data demux:** This is to route the data output from the DRAM to the user or to the processor.

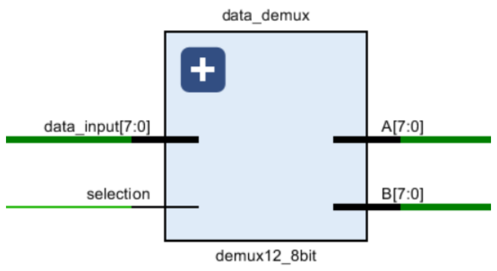


Figure 19: data demux

4.5 Over view of the CPU

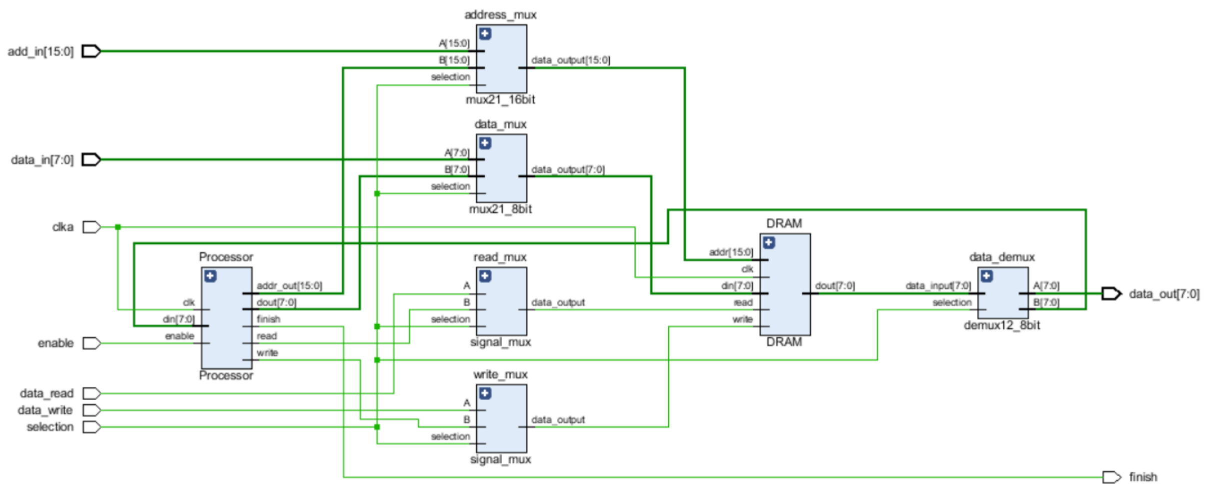


Figure 20: CPU

5 Results and Error analysis

A gray-scale 256 by 256 image was used for the down-sampling task. Then the downsampled image obtained from the processor was compared against a python simulated (assembly code algorithm implemented in python) image. The following metrics were used for the comparison.

- **Absolute Error**
  Here, the sum of absolute element-wise difference was obtained for the each comparison.

$$AbsoluteError = \sum_{i=1}^{L}\sum_{j=1}^{L} |R_{i,j} - C_{i,j}|$$

- **Sum of Square Differences**
  Here, the sum of the squares of element-wise difference was obtained for the each comparison.

$$SSD = \sum_{i=1}^{L}\sum_{j=1}^{L} |R_{i,j} - C_{i,j}|^2$$

- **Maximum Error**
  Here, the maximum difference between in the corresponding pixels of the two images was obtained.

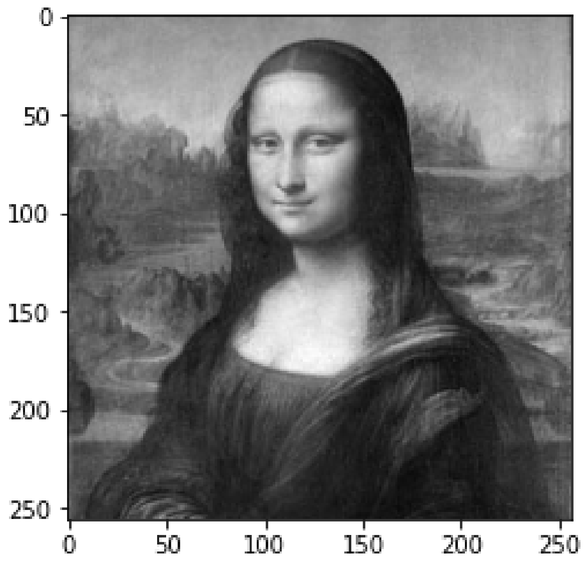
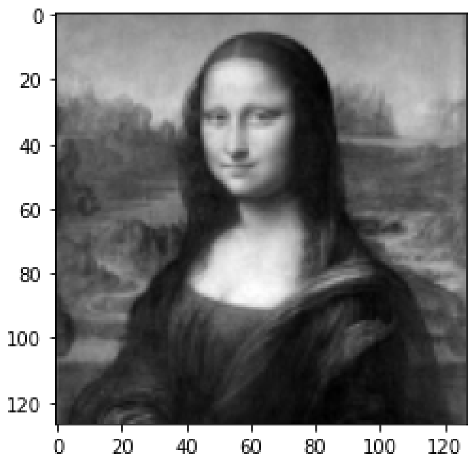
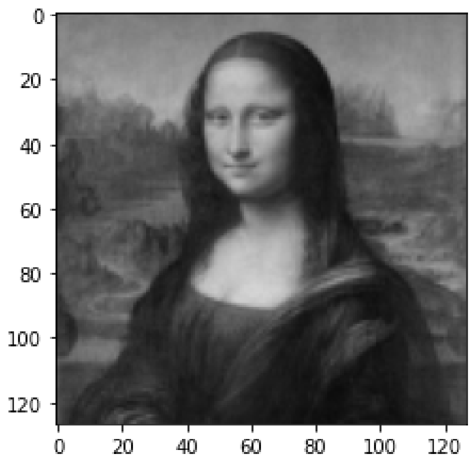


Figure 21: Original image



((a)) Python downsampled



((b)) FPGA downsampled

Figure 22: Downsampled images

- Absolute Error with simulated image : 0.0
- Sum of Square Difference with simulated image : 0.0
- Maximum Error with simulated image : 0.0

6 Appendix

6.1 Python Codes

6.1.1 Image Downsampling

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

# downsample the image using python script
im = cv2. imread ('dog1.jpg' ,cv2.IMREAD_GRAYSCALE)
im = cv2. resize (im ,(256 ,256) )

plt.subplot(1,2,1)
plt.imshow(im,cmap='gray')

im. astype (int)
im = np. reshape (im ,65536)
image = np. zeros (65536)
image[0]=255
image[1:65536]=im[0:65535]
currVal = 258
counter = 2
downsampledCurrVal = 1
while ( currVal < 65024) :
    if ( counter == 256) :
        currVal = currVal + 258
        counter = 2

    total = 0
    total += image [ currVal - 256 -1]
    total += image [ currVal - 256]*2
    total += image [ currVal - 256 +1]
    total += image [ currVal-1]*2
    total += image [ currVal ]*4
    total += image [ currVal +1]*2
    total += image [ currVal + 256 -1]
    total += image [ currVal + 256 ]*2
    total += image [ currVal + 256 +1]
    avg = total / 16
    image [ downsampledCurrVal ] = avg
    downsampledCurrVal += 1
    counter += 2
    currVal += 2

downsampled_im = image [1:127*127+1]
downsampled_im = np. reshape ( downsampled_im ,(127 ,127) )

downsampled_im = np. array ( downsampled_im , dtype = np. uint8 )


plt.subplot(1,2,2)
plt.imshow(downsampled_im,cmap='gray')
```

6.1.2 Generating Input Text

```
import cv2
import numpy as np

im = cv2. imread ('dog1.jpg' ,cv2.IMREAD_GRAYSCALE)
im = cv2. resize (im ,(256 ,256) )

plt.subplot(1,2,1)
plt.imshow(im,cmap='gray')

im = np. reshape (im ,65536)
print(im)
a= open("input2.txt","w")
a.write("255")
for i in im[:-1]:
    a.write("\n%s"%i)
a.close()
```

6.1.3 Generating Image From Text

```
txt_file = open("D:/DownSampling/Downsampling/Downsampling.sim/sim_1/behav/xsim/
l=127
file_content = txt_file.read()
txt_file.close()
content_list = file_content.split("\n")
for i in range (1,len(content_list)-1):
    content_list[i]=int(content_list[i])

content_list.pop(0)
content_list.pop()

print(len(content_list))

for i in range (len(content_list)):
    if (type(content_list[i])!=int):
        print("i"+str(i))
        print(content_list[i])

arr=np.asarray(content_list ,dtype=np.float32)
txt_file.close()
print(type(arr[0]))
arr = np. reshape ( arr ,(l,l) )

import matplotlib.pyplot as plt
plt.imshow(arr, cmap='gray', vmin=0, vmax=255)
```

6.1.4 Error Analysis

```
#im = cv2. imread ('einstein.jpg' ,cv2.IMREAD_GRAYSCALE)
#im = cv2. resize (im ,(127,127) )

arr = np. reshape ( arr ,127*127)
```

```
downsampled_im = np.reshape(downsampled_im , 127*127)

diff = arr - downsampled_im

abs_error = sum(np.absolute( diff ) )

sq_diff = np.power(diff ,2)
SSD = sum( sq_diff )

non_zero_count = np.count_nonzero( diff )
max_error = max(np.absolute( diff ) )

print ('Absolute Error with simulated image : ',abs_error )
print ('Sum of Square Difference with simulated image : ',SSD)
print ('Maximum Error with simulated image :' ,max_error )
```

6.2 Verilog Codes

6.2.1 ALU

```
module ALU(

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

);

//Define the ALU operations

parameter ADD = 4'b0001;
parameter SUB = 4'b0010;
parameter PASSATOC = 4'b0011;
parameter PASSBTOC = 4'b0100;
parameter INCAC = 4'b0101;
parameter DECAC = 4'b0110;
parameter LSHIFT1 = 4'b0111;
parameter LSHIFT2 = 4'b1000;
parameter LSHIFT8 = 4'b1001;
parameter RSHIFT4 = 4'b1010;
parameter RESET = 4'b1011;

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

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

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

always@(negedge clk)
```

```verilog
        begin
                if (start) begin
            state = state+ 2'b01;
                end
        end

always@(posedge clk)
        begin
         if (start) begin
        if (state == 2'b11) begin
        case(operation)
                ADD : C_bus <= A_bus + B_bus;

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

                PASSATOC : C_bus <= A_bus;
                PASSBTOC : C_bus <= B_bus;
                INCAC : C_bus <= A_bus + 1;
                DECAC : C_bus <= A_bus - 1;
                LSHIFT1 : C_bus <= A_bus << 1;
                LSHIFT2 : C_bus <= A_bus << 2;
                LSHIFT8 : C_bus <= A_bus << 8;
                RSHIFT4 : C_bus <= A_bus >> 4;
                RESET : C_bus <= 16'b0;

        endcase
        end
        end
        end
endmodule
```

6.2.2 Control Unit

```verilog

module control_unit(

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

);

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

assign finish = check;



parameter JMPZ1 = 8'd33;
parameter JMPZN1 = 8'd34;
parameter JMPZY1 = 8'd35;
parameter JMNZ1 = 8'd38;
parameter JMNZY1 = 8'd39;
parameter JMNZN1 = 8'd40;
parameter FETCH2 = 8'd1;
parameter NOP = 8'd2;

initial
        begin
                MIR = 30'b0;
        end

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

always@(negedge clk)
        begin
                if (start) begin
            state = state+ 2'b01;
                end
end

always @(posedge clk)
begin
   if(enable) begin
    if (state == 2'b11) begin
        case(addr)
                FETCH2 : MIR = {MBRU,ROM[FETCH2][21:0]};
                JMPZ1  : if (Z_flag == 1'b0) MIR = {8'd34,ROM[JMPZ1][21:0]};
                         else MIR = {8'd35,ROM[JMPZ1][21:0]};
           JMNZ1  : if (Z_flag == 1'b1) MIR = {8'd39,ROM[JMNZ1][21:0]};
                         else MIR = {8'd40,ROM[JMNZ1][21:0]};
                NOP    : check = 1'b1;
                default : MIR = ROM[addr];
        endcase
    end
    end
end

//microinstruction
initial
begin

ROM[0]  = 30'b00000001_0000_0000000000_100_0_0000; //FETCH1
ROM[1]  = 30'bXXXXXXXX_0000_0000000000_000_1_0000; //FETCH2
ROM[2]  = 30'b00000010_0000_0000000000_000_0_0000; //NOP
ROM[3]  = 30'b00000100_0000_0000000000_010_0_0000; //LDAC1
ROM[4]  = 30'b00000000_0100_0000000001_000_0_0001; //LDAC2
ROM[5]  = 30'b00000110_0011_0100000000_000_0_0000; //STAC1
ROM[6]  = 30'b00000000_0000_0000000000_001_0_0000; //STAC2
ROM[7]  = 30'b00000000_1011_0000000001_000_0_0000; //CLAC
ROM[8] = 30'b00000000_0101_0000000001_000_0_0000; //INAC
ROM[9] = 30'b00000000_0110_0000000001_000_0_0000; //DEAC
ROM[10] = 30'b00000000_0001_0000000001_000_0_1000; //ADDZ
ROM[11] = 30'b00000000_0001_0000000001_000_0_0100; //ADDX
ROM[12] = 30'b00000000_0010_0000000001_000_0_1001; //SUBY
ROM[13] = 30'b00000000_0010_0000000001_000_0_0100; //SUBX
ROM[14] = 30'b00000000_1010_0000000001_000_0_0000; //DIV
ROM[15] = 30'b00000000_0111_0000000001_000_0_0000; //MUL2
ROM[16] = 30'b00000000_1000_0000000001_000_0_0000; //MUL4
ROM[17] = 30'b00000000_1001_0000000001_000_0_0000; //MUL256
```

```
ROM[18]  = 30'b00000000_0011_1000000000_000_0_0000; //MVACMAR
ROM[19]  = 30'b00000000_0011_0000100000_000_0_0000; //MVACCV
ROM[20] = 30'b00000000_0011_0000010000_000_0_0000; //MVACC
ROM[21] = 30'b00000000_0011_0000001000_000_0_0000; //MVACDCV
ROM[22] = 30'b00000000_0011_0001000000_000_0_0000; //MVACX
ROM[23] = 30'b00000000_0011_0000000010_000_0_0000; //MVACY
ROM[24] = 30'b00000000_0011_0000000100_000_0_0000; //MVACZ
ROM[25] = 30'b00000000_0100_0000000001_000_0_0101; //MVCV
ROM[26] = 30'b00000000_0100_0000000001_000_0_0110; //MVC
ROM[27] = 30'b00000000_0100_0000000001_000_0_0111; //MVDCV
ROM[28] = 30'b00000000_0100_0000000001_000_0_1000; //MVZ
ROM[29] = 30'b00000000_0100_0000000001_000_0_0100; //MVX


ROM[30] = 30'b00011111_0000_0000000000_100_0_0000; //JUMP1
ROM[31] = 30'b00100000_0100_0000000000_000_0_0011; //JUMP2
ROM[32] = 30'b00000000_0000_0010000000_000_0_0000;

ROM[33] = 30'bxxxxxxxx_0000_0000000000_000_0_0000; //JMPZ1
ROM[34] = 30'b00000000_0000_0000000000_000_1_0000; //JMPZN1
ROM[35] = 30'b00100100_0000_0000000000_100_0_0000; //JMPZY1
ROM[36] = 30'b00100101_0100_0000000000_000_0_0011; //JMPZY2
ROM[37] = 30'b00000000_0000_0010000000_000_0_0000; //JMPZY3

ROM[38] = 30'bxxxxxxxx_0000_0000000000_000_0_0000; //JMNZ1
ROM[39] = 30'b00000000_0000_0000000000_000_1_0000; //JMNZY1
ROM[40] = 30'b00101001_0000_0000000000_100_0_0000; //JMNZN1
ROM[41] = 30'b00101010_0100_0000000000_000_0_0011; //JMNZN2
ROM[42] = 30'b00000000_0000_0010000000_000_0_0000; //JMNZN3




end

endmodule
```

6.2.3 IRAM

```






module IRAM(

input clk,
input [7:0] addr,
output reg [7:0] dout

);

reg [7:0] ROM [120:0];

// Assembly Instructions

parameter FETCH = 8'd0;
parameter NOP = 8'd2;
parameter LDAC = 8'd3;
```

```verilog
parameter STAC = 8'd5;
parameter CLAC = 8'd7;
parameter INAC = 8'd8;
parameter DEAC = 8'd9;
parameter ADDT = 8'd10;
parameter ADDL = 8'd11;
parameter SUBE = 8'd12;
parameter SUBL = 8'd13;
parameter DIV  = 8'd14;
parameter MUL2 = 8'd15;
parameter MUL4 = 8'd16;
parameter MUL256 = 8'd17;
parameter MVACMAR = 8'd18;
parameter MVACCV = 8'd19;
parameter MVACC = 8'd20;
parameter MVACDCV = 8'd21;
parameter MVACX = 8'd22;
parameter MVACY = 8'd23;
parameter MVACZ = 8'd24;
parameter MVCV = 8'd25;
parameter MVC = 8'd26;
parameter MVDCV = 8'd27;
parameter MVZ = 8'd28;
parameter MVX = 8'd29;
parameter JUMP = 8'd30;
parameter JMPZ = 8'd33;
parameter JMNZ = 8'd38;
parameter L1 = 8'd120;
parameter L2 = 8'd21;


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

initial
begin

ROM[0] = CLAC;
ROM[1] = MVACMAR;
ROM[2] = LDAC;
ROM[3] = INAC;
ROM[4] = MVACX;
ROM[5] = MVX;
ROM[6] = INAC;
ROM[7] = INAC;
ROM[8] = MVACCV;
ROM[9] = CLAC;
ROM[10] = INAC;
ROM[11] = MVACDCV;
ROM[12] = CLAC;
ROM[13] = INAC;
ROM[14] = INAC;
ROM[15] = MVACC;

ROM[16] = MVX;
ROM[17] = DEAC;
ROM[18] = DEAC;
ROM[19] = MUL256;
ROM[20] = MVACY;
```

```

ROM[21] = CLAC;
ROM[22] = MVACZ;
ROM[23] = MVCV;
ROM[24] = SUBL;
ROM[25] = DEAC;
ROM[26] = MVACMAR;
ROM[27] = LDAC;
ROM[28] = ADDT;
ROM[29] = MVACZ;
ROM[30] = MVCV;
ROM[31] = SUBL;
ROM[32] = MVACMAR;
ROM[33] = LDAC;
ROM[34] = MUL2;
ROM[35] = ADDT;
ROM[36] = MVACZ;
ROM[37] = MVCV;
ROM[38] = SUBL;
ROM[39] = INAC;
ROM[40] = MVACMAR;
ROM[41] = LDAC;
ROM[42] = ADDT;
ROM[43] = MVACZ;
ROM[44] = MVCV;
ROM[45] = DEAC;
ROM[46] = MVACMAR;
ROM[47] = LDAC;
ROM[48] = MUL2;
ROM[49] = ADDT;
ROM[50] = MVACZ;
ROM[51] = MVCV;
ROM[52] = MVACMAR;
ROM[53] = LDAC;
ROM[54] = MUL4;
ROM[55] = ADDT;
ROM[56] = MVACZ;
ROM[57] = MVCV;
ROM[58] = INAC;
ROM[59] = MVACMAR;
ROM[60] = LDAC;
ROM[61] = MUL2;
ROM[62] = ADDT;
ROM[63] = MVACZ;
ROM[64] = MVCV;
ROM[65] = ADDL;
ROM[66] = DEAC;
ROM[67] = MVACMAR;
ROM[68] = LDAC;
ROM[69] = ADDT;
ROM[70] = MVACZ;
ROM[71] = MVCV;
ROM[72] = ADDL;
ROM[73] = MVACMAR;
ROM[74] = LDAC;
ROM[75] = MUL2;
ROM[76] = ADDT;
ROM[77] = MVACZ;
ROM[78] = MVCV;
ROM[79] = ADDL;
ROM[80] = INAC;
```

```
ROM[81] = MVACMAR;
ROM[82] = LDAC;
ROM[83] = ADDT;
ROM[84] = MVACZ;

ROM[85] = MVZ;
ROM[86] = DIV;
ROM[87] = MVDCV;
ROM[88] = MVACMAR;
ROM[89] = STAC;

ROM[90] = MVDCV;
ROM[91] = INAC;
ROM[92] = MVACDCV;
ROM[93] = MVC;
ROM[94] = INAC;
ROM[95] = INAC;
ROM[96] = MVACC;
ROM[97] = MVCV;
ROM[98] = INAC;
ROM[99] = INAC;
ROM[100] = MVACCV;

ROM[101] = MVCV;
ROM[102] = SUBE;
ROM[103] = JMPZ;
ROM[104] = L1;

ROM[105] = MVC;
ROM[106] = SUBL;
ROM[107] = JMNZ;
ROM[108] = L2;
ROM[109] = MVCV;
ROM[110] = ADDL;
ROM[111] = INAC;
ROM[112] = INAC;
ROM[113] = MVACCV;
ROM[114] = CLAC;
ROM[115] = INAC;
ROM[116] = INAC;
ROM[117] = MVACC;
ROM[118] = JUMP;
ROM[119] = L2;

ROM[120] = NOP;


end
endmodule
```

6.2.4 Clock Generator

```

module clock_gen(

input clk_in,
output clk_out

);

```

```
parameter factor = 4;

reg [7:0] counter = 0;
reg out = 0;

assign clk_out = out;

always @(posedge clk_in)
        begin
                if (counter < factor - 1) begin
                        counter <= counter + 1;
                end
                else begin
                        out <= ~out;
                        counter <= 0;
                end
        end

endmodule
```

6.2.5 Decoder

```
module decoder(

input [15:0] X,CV,C,DCV,Z,Y,
input [7:0] PC,
input [7:0] MDR,
input [7:0] MBRU,
input [3:0] B_bus_ctrl,
output reg [15:0] B_bus

);

always@( B_bus_ctrl or X or CV or C or DCV or Z or Y or PC or MBRU or MDR)
        begin
                case(B_bus_ctrl)

                        4'b0001 : B_bus <= {16'b0,MDR};
                        4'b0010 : B_bus <= {16'b0,PC};
                        4'b0011 : B_bus <= {16'b0, MBRU};
                        4'b0100 : B_bus <= X;
                        4'b0101 : B_bus <= CV;
                        4'b0110 : B_bus <= C;
                        4'b0111 : B_bus <= DCV;
                        4'b1000 : B_bus <= Z;
                        4'b1001 : B_bus <= Y;
                        default : B_bus <= 16'b0;

                endcase
        end

endmodule
```

6.2.6 Demultiplexer

```

module demux12_8bit(

input [7:0] data_input,
input selection,
output reg [7:0] A,
output reg [7:0] B

);

always @(data_input or selection)

        begin
                case(selection)
                        1'b0 : A = data_input;
                        1'b1 : B = data_input;
                endcase
        end

endmodule
```

6.2.7 GPR

```

module GPR(

input clk,
input load,
input [15:0] C_bus, //connects to the MIR load signal
output reg [15:0] data_out //connects to the B_bus (MUX)

);

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

endmodule
```

6.2.8 MAR

```
module MAR(

input clk,
input load,
input [15:0] C_bus,
output reg [15:0] data_address

);

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

endmodule
```

6.2.9 MBRU

```


```

```

module MBRU(

input clk,
input fetch,
input [7:0] ins_in,
output reg [7:0] ins_out

);

always@(posedge clk)
        begin
        if (fetch) ins_out <= ins_in;
        end

endmodule
```

6.2.10 MDR

```



module MDR(

input clk,
input load,
input read,
input write,
input [7:0] C_bus,
input [7:0] data_in_DRAM,
output reg [7:0] data_out_Bbus,
output reg [7:0] data_out_DRAM

);

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

                if (read) data_out_Bbus <= data_in_DRAM;

                if (write) data_out_DRAM <= C_bus;

        end

endmodule
```

6.2.11 Multiplexer (1-bit)

```

module signal_mux(

input A,
input B,
input selection,
output reg data_output

);

```

```
always @(A or B or selection)

        begin
                case(selection)
                        1'b0 : data_output = A;
                        1'b1 : data_output = B;
                endcase
        end

endmodule
```

6.2.12 Multiplexer (8-bit)

```

module mux21_8bit(

input [7:0] A,
input [7:0] B,
input selection,
output reg [7:0] data_output

);

always @(A or B or selection)
        begin
                case(selection)
                        1'b0 : data_output = A;
                        1'b1 : data_output = B;
                endcase
        end

endmodule
```

6.2.13 Multiplexer (16-bit)

```
module mux21_16bit(

input [15:0] A,
input [15:0] B,
input selection,
output reg [15:0] data_output

);

always @(A or B or selection)
        begin
                case(selection)
                        1'b0 : data_output = A;
                        1'b1 : data_output = B;
                endcase
        end

endmodule
```

6.2.14 PC

```
module PC(

input enable,
input clk,
input finish,
input load, //connects to PC Write signal (for tb 0)
input inc, //inc signal
input [7:0] C_bus,
output reg [7:0] ins_address //MBRU ins_in

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

initial begin
        ins_address <= 8'b0;

end

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

always@(negedge clk)
        begin
            if (start) begin
                state = state+ 2'b01;
            end
       end

always@(posedge clk)
        begin
                if (finish) begin ins_address <= ins_address; end
                else if (enable) begin
                 if (load && state == 2'b11) begin
                                ins_address <= C_bus;
                                end
                        else if (inc && state == 2'b11) begin
                                ins_address <= ins_address + 8'b00000001;
                        end
                        else begin
                                ins_address <= ins_address;
                end
                end
        end

endmodule
```

6.2.15 Processor

```

module Processor(

input enable,
input clk,
input [7:0] din,
output [15:0] addr_out,
output [7:0] dout,
output finish,
output read,
output write

);
```

```verilog

wire [7:0] ins_address;
wire [7:0] ins_in;
wire [7:0] ins_out;
wire [29:0] MIR;
wire [15:0] A_bus;
wire [15:0] B_bus;
wire [15:0] C_bus;
wire Z_flag;
wire [15:0] X_bus;
wire [15:0] Y_bus;
wire [15:0] Z_bus;
wire [15:0] CV_bus;
wire [15:0] Counter_bus;
wire [15:0] DCV_bus;
wire [7:0] MDR_bus;
wire finish_flag;

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

assign finish = finish_flag;

//Define the Control Store

control_unit CTRL_UNIT(
.enable(enable),
.clk(clk),
.Z_flag(Z_flag),
.addr(MIR[29:22]),
.MBRU(ins_out),
.MIR(MIR),
.finish(finish_flag)

);


PC PC(
.enable(enable),
.clk(clk),
.inc(MIR[4]),
.C_bus(C_bus),
.finish(finish_flag),
.ins_address(ins_address),
.load(MIR[15])
);


IRAM IRAM(
.clk(clk),
.addr(ins_address),
.dout(ins_in)
);


MBRU MBRU(
.clk(clk),
.fetch(MIR[7]),
.ins_in(ins_in),
.ins_out(ins_out)
);
```

```


GPR X(
.clk(clk),
.load(MIR[14]),
.C_bus(C_bus),
.data_out(X_bus)
);



GPR CV( //CURRENT VALUE
.clk(clk),
.load(MIR[13]),
.C_bus(C_bus),
.data_out(CV_bus)
);



GPR C( //COUNTER
.clk(clk),
.load(MIR[12]),
.C_bus(C_bus),
.data_out(Counter_bus)
);



GPR DCV(//DOWNSAMPLED CURRENT VALUE
.clk(clk),
.load(MIR[11]),
.C_bus(C_bus),
.data_out(DCV_bus)
);



GPR Z(
.clk(clk),
.load(MIR[10]),
.C_bus(C_bus),
.data_out(Z_bus)
);



GPR Y(
.clk(clk),
.load(MIR[9]),
.C_bus(C_bus),
.data_out(Y_bus)
);



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



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


MAR MAR(
.clk(clk),
.load(MIR[17]),
.C_bus(C_bus[15:0]),
.data_address(addr_out)
);


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


decoder decoder(
.X(X_bus),
.CV(CV_bus),
.C(Counter_bus),
.DCV(DCV_bus),
.Z(Z_bus),
.Y(Y_bus),
.PC(ins_address),
.MDR(MDR_bus),
.MBRU(ins_out),
.B_bus_ctrl(MIR[3:0]),
.B_bus(B_bus)
);

endmodule
```

6.2.16 Downsampling CPU

```

`timescale 1ns / 1ps

module downsampling_CPU(

input clka,
input enable,
input [7:0] data_in,
input [15:0] add_in,
input data_write,
input data_read,
input selection,
output finish,
output [7:0] data_out
);

//define busses

wire [15:0] address_bus;
wire [7:0] data_in_bus;
wire [7:0] data_out_bus;
wire [7:0] dram_in_bus;
wire [15:0] dram_addr;
wire [7:0] dram_out;
wire write;
wire read;
wire dram_write;
wire dram_read;


Processor Processor(

.din(data_out_bus),
.enable(enable),
.clk(clka),
.dout(data_in_bus),
.addr_out(address_bus),
.finish(finish),
.write(write),
.read(read)

);

DRAM DRAM(

.addr(dram_addr),
.din(dram_in_bus),
.read(dram_read),
.write(dram_write),
.clk(clka),
.dout(dram_out)

);






mux21_8bit data_mux(
.B(data_in_bus),
.A(data_in),
.selection(selection),
.data_output(dram_in_bus)

);

signal_mux write_mux(
.A(data_write),
.B(write),
.selection(selection),
.data_output(dram_write)
);

signal_mux read_mux(
.A(data_read),
.B(read),
.selection(selection),
.data_output(dram_read)
);

mux21_16bit address_mux(
.B(address_bus),
.A(add_in),
.selection(selection),
.data_output(dram_addr)

);

 demux12_8bit data_demux (
 .data_input(dram_out),
 .selection(selection),
 .A(data_out),
 .B(data_out_bus)

 );


endmodule
```