Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN3030 - Circuits and Systems Design

Processor Design Report

Name	**Index**
D.S.Weerapperuma	180685C
P.S.C.Jayapala	180265N
D.S.B.C.L.Jayaweera	180288L
J.T.H.Randika	180520T

Table Of Contents

	Page No
1. Introduction	**3**
2. Instructions Set	**4**
3. Registers	**7**
4. Datapath	**8**
5. System Architecture	**9**
6. Timing Analysis	**20**
7. Algorithm	**25**
8. Assembler	**30**
9. Debugging	**34**
10.Results	**34**
11.Error Analysis	**35**
12.Conclusion	**36**
13.References	**36**
14.Appendices	**37**

1. Introduction

Processors can be categorized into two types such as general-purpose processors and application-based processors. In application-based processors are implementing for specific purpose. General purpose processors are generic. But application-based processors have designed specific registers, instructions, data path and so on. In this project that we are designing application-based processor. Finally, it is developed for special purpose of low pass filtering and image down sampling. Our Instructions set and Data path which suit to achieve the tasks.

Problem Statement

The main objective of the given task was to design a custom processor which can down sample a given image by factor of 2. There were no constrains upon the design such as down sampling factor, whether the original image is color or greyscale and resolution of the image. Therefore, it became clear that the filtering and down sampling the given image is the main requirement of the processor. In addition to that output image of the processor should be compared with the original image and the standard error should be calculated in order to verify the implementation.

General Overview of the Solution

The main task can be divided into several sub tasks as follows:

1. Convert given image into binary data and store in DRAM
   We used python script to convert given image into a 8 bit data stream and then stored those values in the DRAM.

2. Convert Assembly instructions into binary data and store in IRAM
   We used a python script to convert our assembly instruction set into 16-bit binary data stream and then stored those in the IRAM.

3. Filter the stored image
   According to the available instructions based on a filtering algorithm, processor access to the DRAM and get the data when it is required. The filtering was mainly done to get rid of high frequency components which can cause aliasing effects on the down sampled output image. After filtering, the filtered data was again stored in the memory unit.

4. Down sample the stored image
   At this level the filtered image gets down sampled by the factor 2. After that down sampled data also stored in the same memory unit.

5. Save processed image data into the output text file
   Before converting into an image, the processed data stored in the memory should be saved in a text file which we have implemented as "Write_txt" module in the processor.

6. Convert output into a image

We used python script to convert processed binary data in the text file to a matrix of the desired size and then convert it into an image.

7. Verification of the results
We processed the same original image through same process of filtering and down sampling using python and then compared the two down sampled images from two procedures considering the sum of squared error.

2. Instructions Set

Operand 1/ Operand 2

R1:	00000
R2:	00001
R3:	00010
R4:	00011
R5:	00100
R6:	00101
R7:	00110
R8:	00111
R9:	01000
R10:	01001
R11:	01010
AC:	01011
MAR:	01100

1. **NOP**

5 bits	11 bits
00000	**xxx xxxx xxxx**

No operation. Only one clock cycle.

2. **LOAD[R]**

5 bits	5 bits	1 bit	5 bits
00001	**xxxxx**	**x**	**Operand 1**

Loads data from data memory (DRAM) into the register. In which register to be loaded is determined by operand 1 The memory location of the DRAM is obtained by MAR

3. **LOADINC[R]**

5 bits	5 bits	1 bit	5 bits
00010	**xxxxx**	**x**	**Operand 1**

Loads data from data memory (DRAM) into the register. In which register to be loaded is determined by operand 1. The memory location of the DRAM is obtained by MAR+1.

4. **STORE[R/AC]**

5 bits	5 bits	1 bit	5 bits
00011	**xxxxx**	**x**	**Operand 1**

Store data from Register / AC into memory (DRAM). In which register’s value will be stored is determined by the operand 1. The memory location of the DRAM is obtained by MAR.

5. **COPY[R1/AC] [R2/AC/MAR]**

5 bits	5 bits	1 bit	5 bits
00100	**Operand 2**	**x**	**Operand 1**

Copy data from R1 / AC into R2/AC/MAR. Operand 1 represents a source register and operand 2 represents a destination register.

6. **JUMP[R]**

5 bits	5 bits	1 bit	5 bits
00101	**xxxxx**	**x**	**Operand 1**

Jump to given address in IRAM were stored in general purpose register R.

7. **JUMPNZ[R]**

5 bits	5 bits	1 bit	5 bits
00110	**xxxxx**	**x**	**Operand 1**

Jump to given address in IRAM were stored in general purpose register R when Z_inv ==1 .

8. **INC[R/AC/MAR]**

5 bits	5 bits	1 bit	5 bits
00111	**xxxxx**	**x**	**Operand 1**

Increment the value by one in anyone of the register presented in operand 1 field in the instruction.

9. **ADDI[n]**

5 bits	5 bits	1 bit	5 bits
01000	**xxxxx**	**x**	**Value**

Add the constant value to AC and result is stored in the AC.

10. **SUBI[n]**

5 bits	5 bits	1 bit	5 bits
01001	**xxxxx**	**x**	**Value**

Subtract the constant value from AC and the result is stored in AC.

11. **LSHI[n]**

5 bits	5 bits	1 bit	5 bits
01010	**xxxxx**	**x**	**Value**

Left shift the constant value from AC and result is stored in the AC.

12. **RSHI[n]**

5 bits	5 bits	1 bit	5 bits
01011	**xxxxx**	**x**	**Value**

Right shift the constant value from AC and result is stored in the AC.

13. **ADD[R]**

5 bits	5 bits	1 bit	5 bits
01100	**xxxxx**	**x**	**Operand 1**

Add the value in the register in the operand 1 field to the AC and result is stored in the AC.

14. SUB[R]

5 bits	5 bits	1 bit	5 bits
01101	xxxxx	x	Operand 1

Subtract the value in the register in the operand 1 field from the AC and result is stored in the AC.

15. MUL[R]

5 bits	5 bits	1 bit	5 bits
01110	xxxxx	x	Operand 1

Multiply the value in the register in the operand 1 field by the AC and result is stored in the AC.

16. RESET[R/AC/MAR]

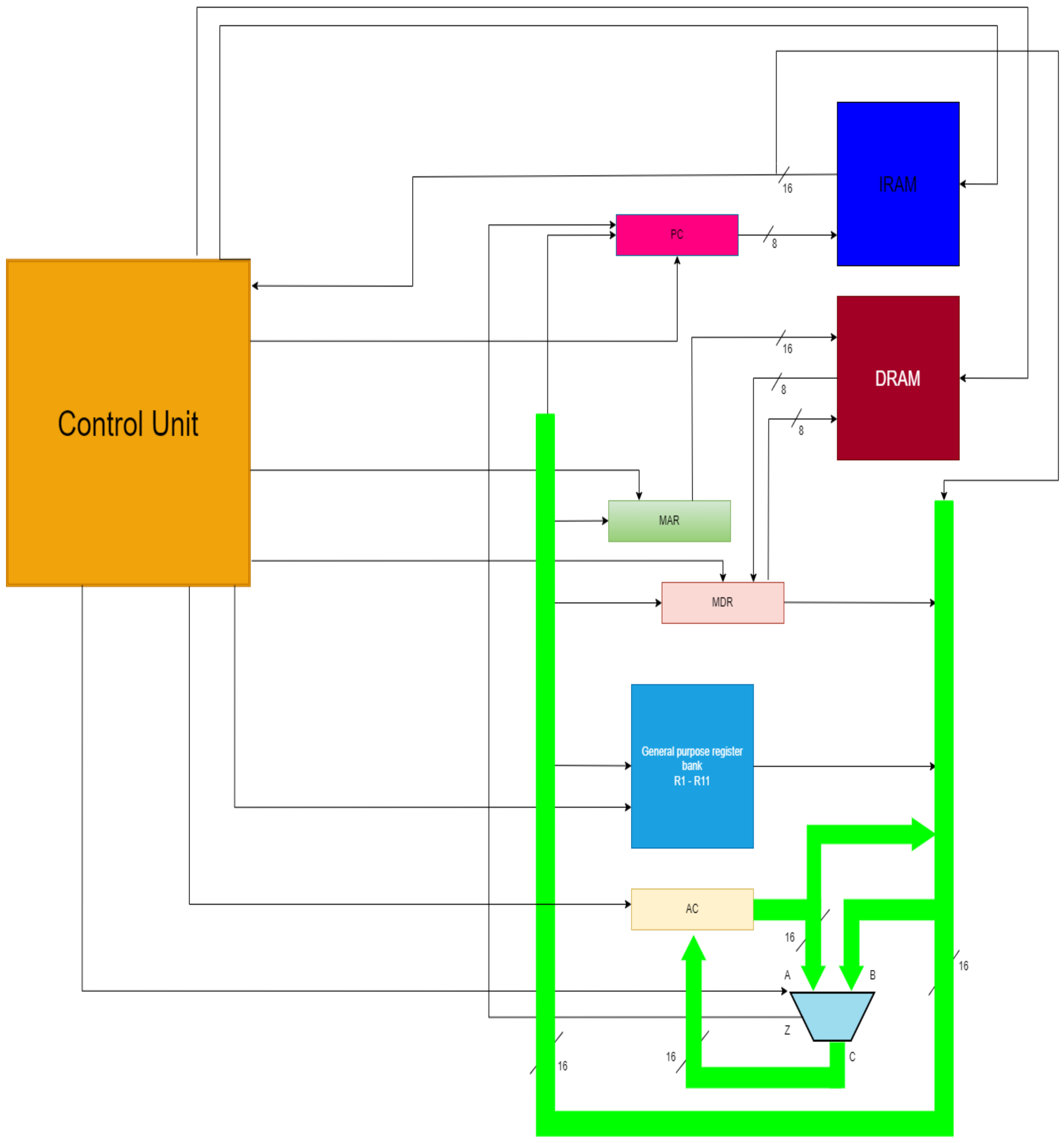
5 bits	5 bits	1 bit	5 bits
01111	xxxxx	x	Operand 1

Reset the value in the register in the operand 1 field.

3. Registers

Register	Name of the Register	Size in Bits	Purpose
MAR	Memory Address Register	16	The memory address which the processor is going to reach next stored in here.
MDR	Memory Data Register	8	Data read from or data will be written to the memory is stored here.
AC	Accumulator	16	Store the output of the ALU operations
PC	Program Counter	16	Stores address of the next Instruction in IRAM
(R1 - R11)	General Purpose Registers	16	These registers are using to general purposes. Additions, substructions, multiplication, temporary store and so on.

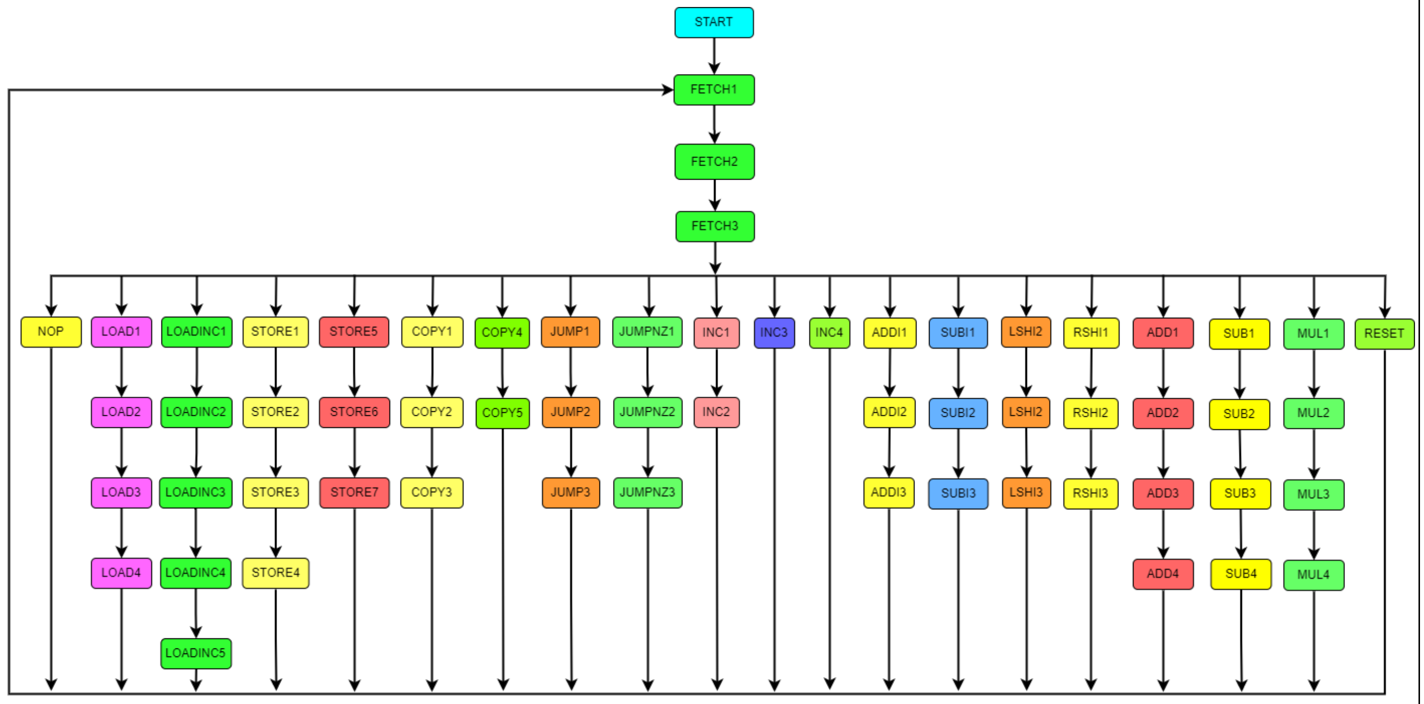
4. Datapath



Datapath of the processor

5. System Architecture

State Diagram



Micro Instructions

1. **START**
   State <=Fetch 1(If enable is high)
   State <= Start (If enable is low)

2. **FETCH 1**
   Read IRAM (load instruction)
   state<= Fetch 2

3. **FETCH 2**
   state<= Fetch 3

4. **FETCH 3**
   PC <= PC+1

5. **NOP**

6. **LOAD**
   a. READ
   b. MDR <= M
   c. BUS <= MDR
   d. R<= BUS

7. **LOAD_INC**
    a. MAR<=MAR+1
    b. READ
    c. MDR <= M
    d. BUS <= MDR
    e. R<= BUS

8. **STORE (If select R general purpose register)**
    a. R select
    b. BUS<= R
    c. MDR<= BUS
    d. write

    STORE (If select AC register)
    a. BUS<= AC
    b. MDR<= BUS
    c. write

9. **COPY (If select R general purpose register)**
    a. R select
    b. BUS<= R
    c. R/AC/MAR <= BUS

    COPY (If select AC register)
    a. BUS<= AC
    b. R/AC/MAR <= BUS

10. **JUMP**
    a. R select
    b. BUS<= R
    c. PC<=BUS

11. **JUMPNZ**
    a. R select
    b. BUS<= R
    c .PC<= BUS(if z_inv ==1)/ pc<= pc+1 (if z_inv == 0)

12. **INC (If select R general purpose register)**
    a.R<=R+1

    INC(If select AC register)
    a.ALU_out<=AC+1
    b.AC<=ALU_out

    INC(If select MAR register)
    a.MAR<=MAR+1(if MAR selcted)

13. ADDI
   a. BUS<= n
   b. ALU_out<= AC + BUS
   c. AC<=ALU_out

14. SUBI
   a. BUS<= n
   b. ALU_out<= AC - BUS
   c. AC<=ALU_out

15. LSHI
   a. BUS<= n
   b. ALU_out<= AC << BUS
   c. AC<=ALU_out

16. RSHI
   a. BUS<= n
   b. ALU_out<= AC >> BUS
   c. AC<=ALU_out

17. ADD
   a. R select
   b. BUS<= R
   c. ALU_out<= AC + BUS
   d. AC<=ALU_out

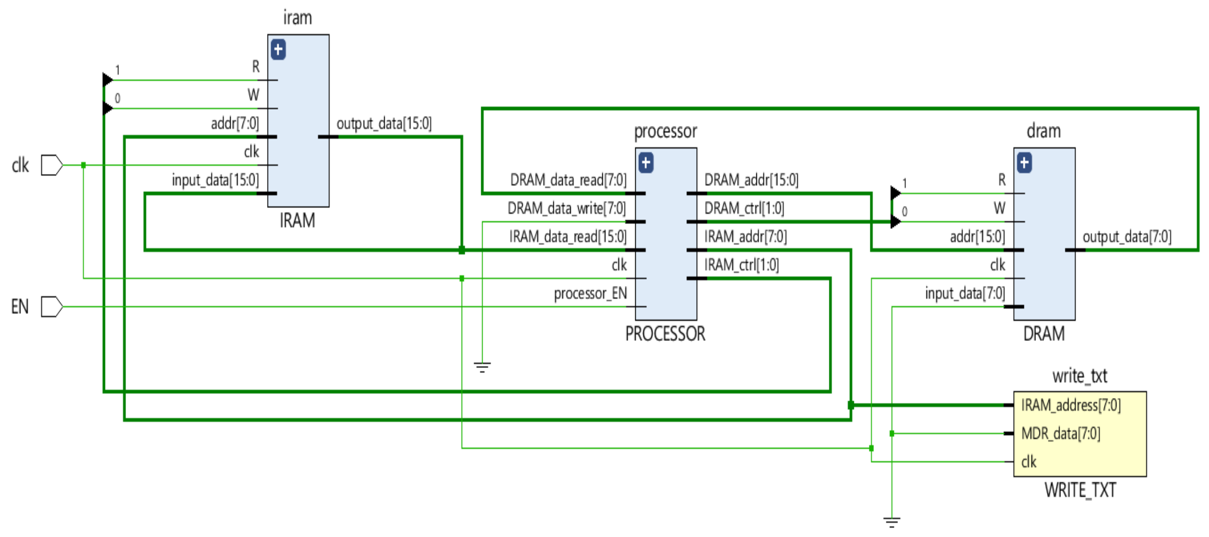
18. SUB
   a. R select
   b. BUS<= R
   c. ALU_out<= AC - BUS
   d. AC<=ALU_out

19. MUL
   a. R select
   b. BUS<= R
   c. ALU_out<= AC * BUS
   d. AC<=ALU_out

20. RESET
   a.AC/R/MAR<=0

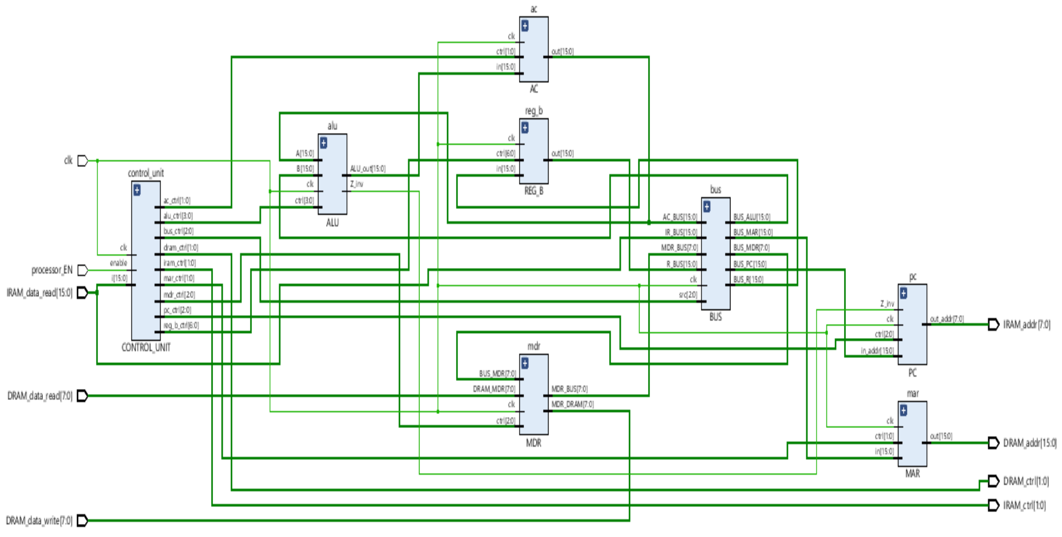
System Architecture

Top Module

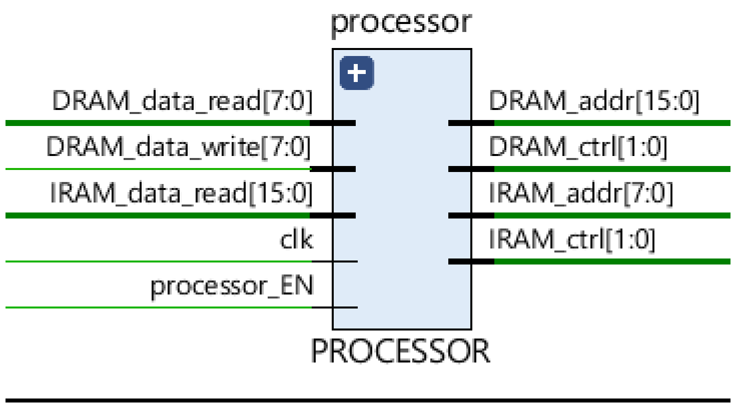


There are four main modules in Top Module which are IRAM, DRAM, Processor and Write_Txt. IRAM is used to store machine code of the binary code which has 256 memory locations and we used only 171 locations for store. DRAM has 65536 memory locations for store image pixel values. DRAM memory location size is 8 bits. Clock is connected to every module for synchronized the process. IRAM_address and DRAM connect to the write_txt module which is write the output of the processor to the text file. Processor is the most important module of the top module diagram. Because processor has a lot of modules. Those modules are very important to processor working. IRAM and DRAM process also depend on the processor internal operations because, DRAM and IRAM has control signals. Processor contains control unit, ALU, MAR, MDR, AC, PC, REG_B and BUS modules. Each one explains in the below sections.

Processor internal connections



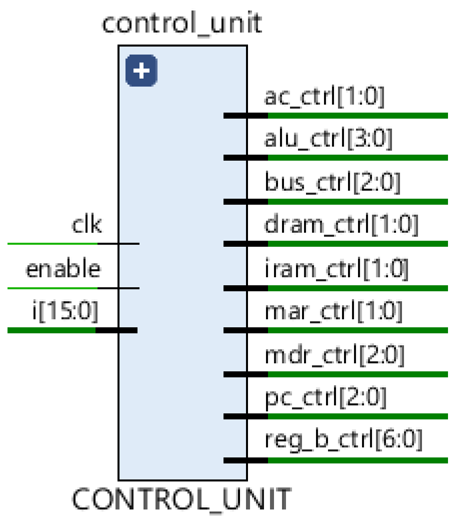
Processor



Processor is the main unit in verilog module implementation. It uses to image down sampling and other operations using the defined set of instructions. It is connected to the instruction memory using the IRAM_data_read and IRAM_addr ports that can use to fetch the instructions. Processor can read data and write data from data memory using DRAM_data_read and DRAM_data_write pins and DRAM address is given by DRAM_addr pin. There are 4 input such as processor enable, clock,16 bit IRAM_data_read and 8 bit DRAM_data_read. There are 5 outputs contain such as IRAM_addr,DRAM_addr, IRAM_ctrl,8 bit DRAM_data_write and DRAM_ctrl. DRAM input needs to read image pixels value. IRAM input needs to run machine code instructions of the processor. IRAM_ctrl and DRAM_ctrl helps to decide to read or write the data at the moment.
All the wire connections are specifically defined inside the Processor Verilog module. Inside the processor there are several main units which are Control unit, ALU, AC, REG_B, MDR, MAR, PC and BUS. Control unit inputs and control unit outputs wirings are specifically defined inside the code as follows.

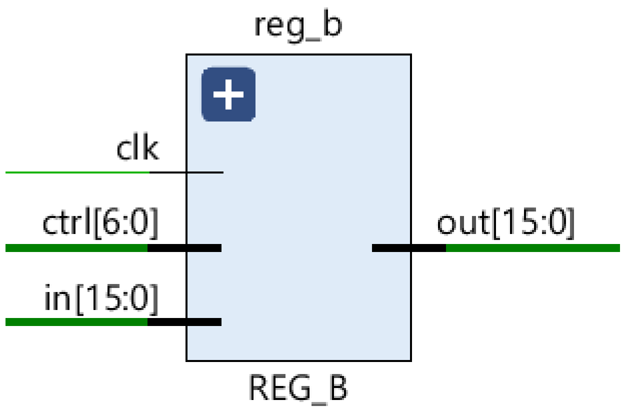
```
//control unit outputs
 wire [3:0] CTRL_ALU;
wire [1:0] CTRL_AC;
wire [6:0]  CTRL_R;
wire [2:0]  CTRL_MDR;
wire [1:0]  CTRL_MAR;
wire [2:0] CTRL_PC;
wire [2:0] CTRL_BUS;

//control unit inputs
 wire [15:0] ALU_AC;
 wire [15:0] BUS_ALU;
 wire [15:0] AC_ALU;
```

Control Unit



Control unit is the one necessary inside the Processor Module. Control unit ensures the correct control signals are given for every module unit inside the processor. Inside the control unit verilog module contains all the Instructions and micro instructions. The Control Unit switches the state of each positive edge of the clock cycle depend on the instruction type and fetch, decode, execute cycles depend on the state diagram of the processor. Each state gives necessary control signals for each unit in the processor. Control unit has 3 inputs such as clock, enable and instruction pins. At the beginning control unit is in START state and after jumps to the FETCH1 state when enabled the processor. The processor comes to high states in order to start the process. Control unit also synchronized with clock.
Control unit gives control signals for modules inside the processor. Alu_ctrl 4 bit control signals give for Arithmetic and Logic unit which used to select operation of arithmetic or logic depend on the current state. Pc_ctrl PC control signal has 3 bits which used to increment PC or change PC in jump instructions subjected to the jump conditions. We have 11 general purpose register bank that control by reg_b_ctrl signals. It has 7 bits. IRAM and DRAM are external modules to the processor. But Control unit gives 2 bit control signal for each. Load the instructions are used to read IRAM. DRAM contains data of the picture pixels values. Control unit signals decide to where is the write and where is from read the data as well. Control unit output 3bit control signal for MDR which used to control the functionalities in the MDR.

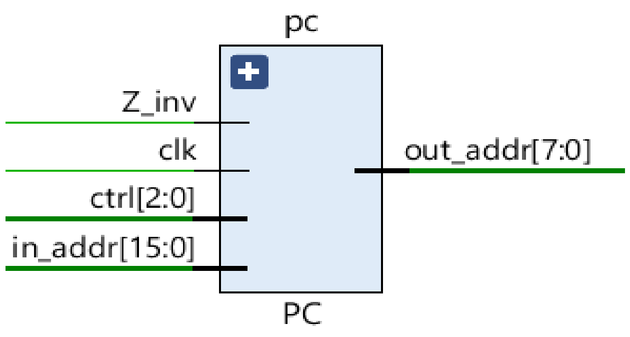
Register Bank



We design a register bank which contains 11 registers can be used as a general purpose registers. REG_B module has 3 inputs and only one output. Each register is 16-bit register. Output also 16 bits. 3 inputs such as clock, control, and input. 7-bit control signal comes to reg_b module which [3:0] range will select the register we want. In control signal [6:4] range related to 4 functionalities inside the REG_B module such as,

3'b001 : store input to the given register
3'b010 : clear the value of given register
3'b011 : increment the value of given register by 1.
3'b100 : output the value of given register

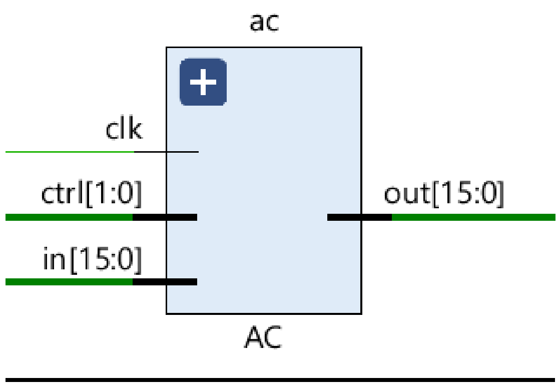
Program Counter



8 bit program counter points to next out_addr that should load into control unit in the next fetch cycle. Program counter operation is decided by ctrl signal port. There are 3 control signal bits. The clock is used to synchronized the PC. There are 4 inputs and only one output contain in the Program Counter. Z_inv, clock,ctrl and in_addr are inputs. Out_addr is an output. Z_inv is come from ALU. ctrl port 3 bits come from control unit. 16 bits in_addr is coming from BUS module. 8 bits out_addr is connected to the IRAM_addr in IRAM module. In normal conditions PC increment by 1 when ever in_addr is read and points to the instructions. In jump instructions, PC is modified to the jump address according to the jump condition. Jump condition is checked by value of Z_inv flag as an input. There are 4 control cases inside the PC module such as

3'b001 : increase the value of the PC by 1
3'b010 : update PC by jump_addr according to Z_inv ;
3'b011 : update PC by jump_addr;
3'b100 : clear PC;

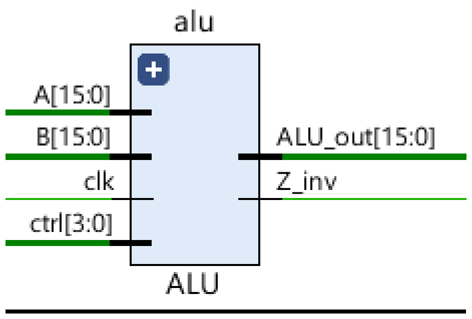
Accumulator



Accumulator acts as a one input of the ALU and also ALU output is connected to the directly Accumulator. Accumulator is connected to the Main BUS as well. Accumulator can store value temporarily. AC has 3 inputs which are 16 bit input, clock and control bits and one 16bit output. There are only 2 bits for control AC. Accumulator is synchronized with the clock. Initially AC stores zero. There only two control cases inside the Accumulator Verilog module which are,

2'b01 : Write ALU output to AC;
2'b10 : Clear AC;

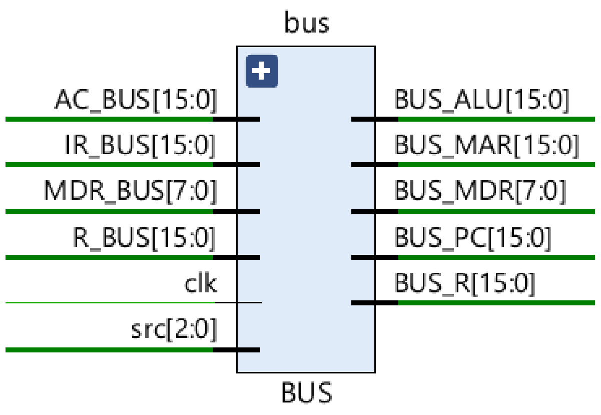
ALU



ALU is doing all the arithmetic and logic operations in the processor. In our instructions there are only basic arithmetic operations such as addition, subtraction, multiplication and shifting. Those instructions are enough to down sample the image. The 16 bit ALU_out register is inside same module and stores the result of ALU. ALU has 4 inputs and 2 outputs. Inputs such as A ,B 16 bit inputs buses, 4 bit control signal and clock. A and B use to operands for arithmetic operations. Outputs are the ALU_out and Z_inv. ALU_out connects to the Accumulator and Z_inv connects with the Program counter. There are 11 control signal cases have inside the ALU module such as,

4'b0001 : A + B	4'b0110 : A + 1
4'b0010 : A - B	4'b0111 : A - 1
4'b0011 : A*B;	4'b1000 : A << B[4:0]
4'b0100 : A + B[4:0]	4'b1001 : A >> B[4:0]
4'b0101 : A - B[4:0]	4'b1010 : B
4'b1011 : Clear ALU_out	

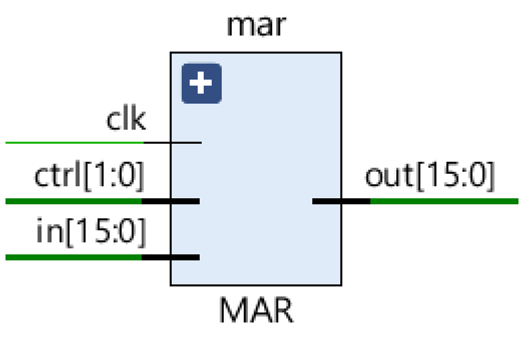
BUS



Bus module uses for data transferring between the registers in processor. It provides the second input to ALU through the BUS_ALU. Data bus can be loaded from many registers and also 8 bit registers are loaded those registers first 8 bits are set to zero. There are six inputs and five outputs contain in the BUS module.
Inputs are MDR_BUS, R_BUS, AC_BUS, IR_BUS, clock and src control signal.
MDR_BUS is contained 8 bit and src contains 3 bits. Other R,AC and IR buses are 16 bits.
Outputs are such as, BUS_MDR, BUS_R, BUS_ALU,BUS_MAR and BUS_PC. BUS_MDR is 8 bits and others are 16 bits.
Inside the BUS module it creates a temporary 16 bit register. That contains important assign statements.
Src input contains only 3 bits and which has 4 cases such as,

3'b001 : select MDR as input;
3'b010 : select R as input;
3'b011 : select AC as input;
3'b100 : select IRAM as input;

MAR

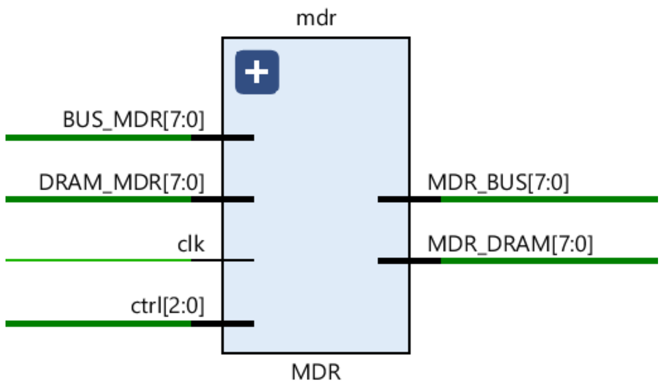


This 16 bit memory address register stores the memory address of the DRAM which should be loaded into the memory data register or which should be written from the contents in the MDR. the contents of MAR can be updated from BUS verilog module. There are 2 bit control signals as well as clock bit. BUS and MAR connect 16 bit wire. The module is synchronized with the clock and

MAR controls decide the source to modify the value stored in MAR. MAR out 16 bit pin directly connected to the DRAM addr pin.
There are 3 control signal cases inside the MAR such as,

2'b01 : Write input to MAR
2'b10 : Increment MAR by 1
2'b11 : Clear MAR

MDR

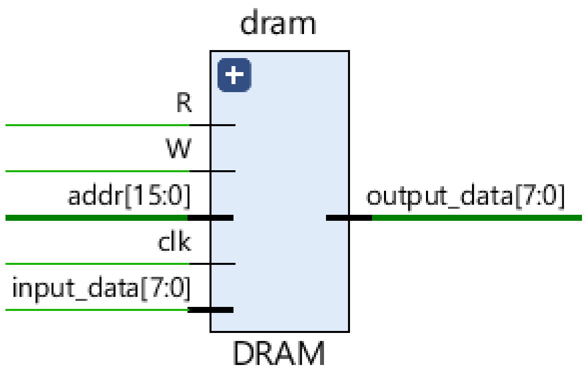


The 8 bit memory data register stores the data loaded by the DRAM or data which should be written to the DRAM. The contents of MDR can update two ways such as DRAM and BUS module . MDR module synchronized with the clock and MDR control decides the source to modify value store in MDR. There are 4 inputs and 2 outputs.
Inputs are DRAM_MDR, BUS_MDR, ctrl and clk. Outputs are MDR_DRAM and MDR_BUS. Initially MDR is assigned to zero inside the module code.

There three control cases using 3 control signals bit,

3'b001 :Write value from BUS
3'b010 : Write value from DRM
3'b011 : Clear MDR

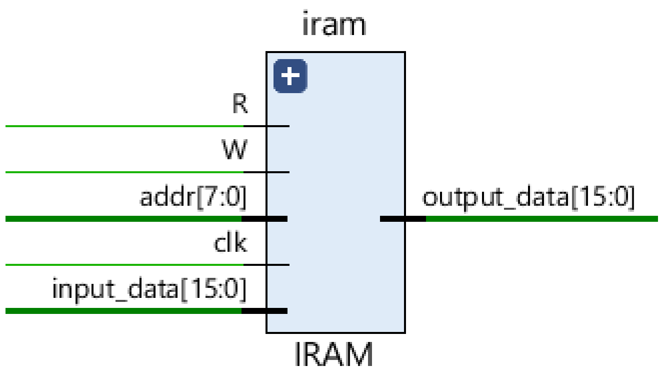
DRAM



Data memory we uses only 65536 memory locations in our processor implementation. But our Memory has 62520 memory locations. Each location can store an 8 bit binary value. Data memory

is used to store 250*250 resized image pixels values. When we down sample a new image first we need to put our new image pixels values to the DRAM. addr 15 bit pin connect to the processor DRAM_addr. Output_data pin connects with DRAM_data_read pin in processor. DRAM_ctrl is divided as R and W pins. Input_data pin connect to the write_txt module which does write the text file using the DRAM down sampled output.
There are five inputs and only one output. Inputs are input_data,addr,R,W and clock. Output is output_data.
There are some initial conditions which we used to further process. Those values are stored initially in the DRAM.

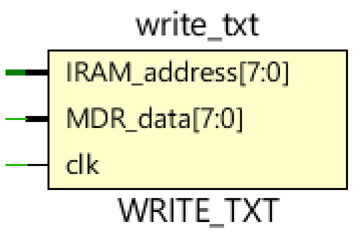
IRAM



Instruction memory have 256 memory locations. Each memory location can store 16 bit size values. IRAM has Read and Write pins as well. After convert from assembly code to binary machine code which we stored in IRAM. In our project has used 171 instructions. addr 8 bit port connected to the processor program counter. IRAM also has 5 inputs and only output.
Inputs are 8 bit address, 16 bit input_data, R, W and clock. Output is 16 bit output_data.
Initially output data equals to zero.

If R==1 we can read the data from memory where specified from addr port and if W==1 then write data into IRAM ,but we didn’t use that functionality at this project.

Write Txt

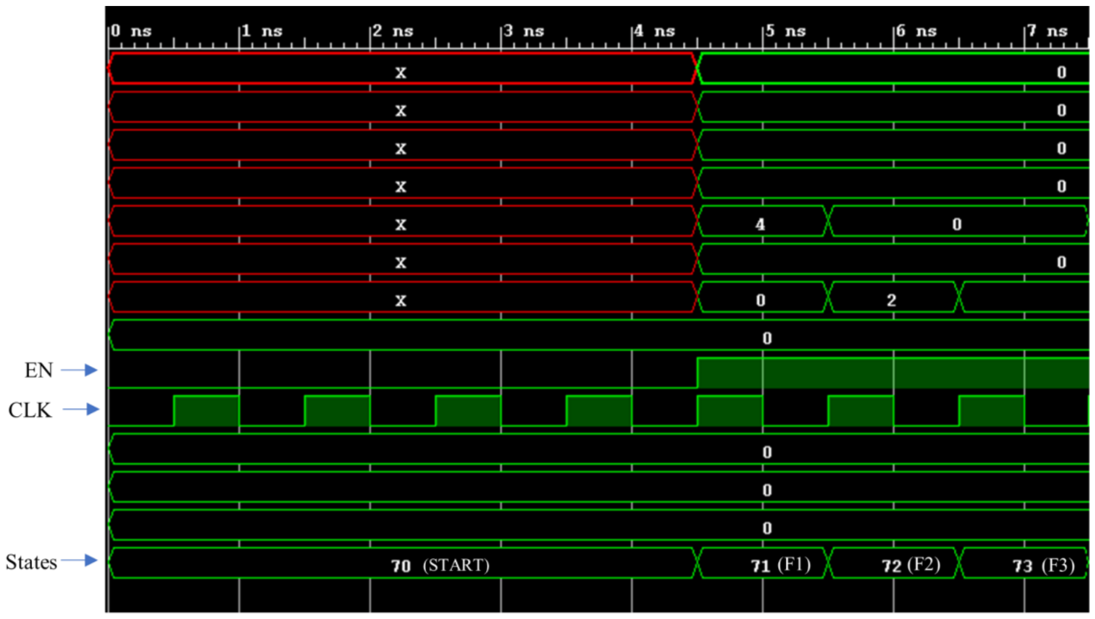


This module stores the final output data externally. It has 3 inputs only, which are MDR_data,clock and IRAM_address. We store down sampled pixels values inside the text file as 8-bit values. That can be used to get output of the processor for further processing.

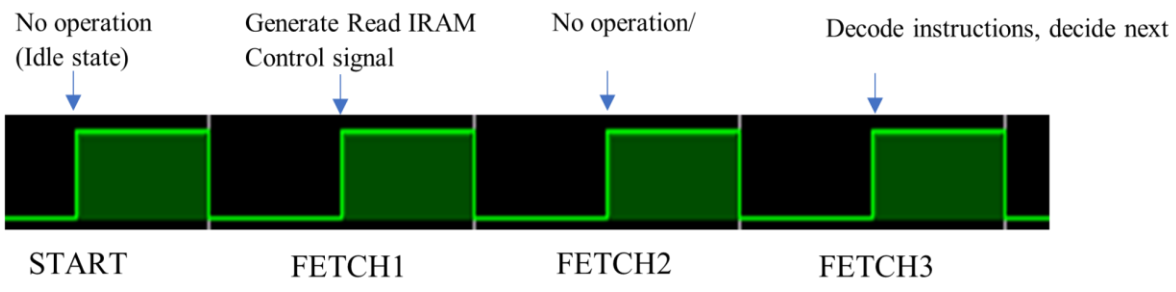
6.Timing Analysis

All the processor activities depend on the clock signals since all are the synchronized modules. Control signals are generated at rising clock edges and all modules start their activities at the rising clock edge. At the rising edge the microinstructions will start to execute and at the next rising edge the results are available. So that no conflicts occur. The number of clock cycles used to execute the instruction can be depend on the instruction and the selected operand register. The following figures are showing the time domain analysis of the instructions and fetching cycles.

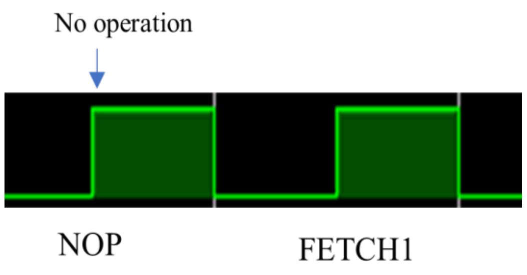
Start and fetching



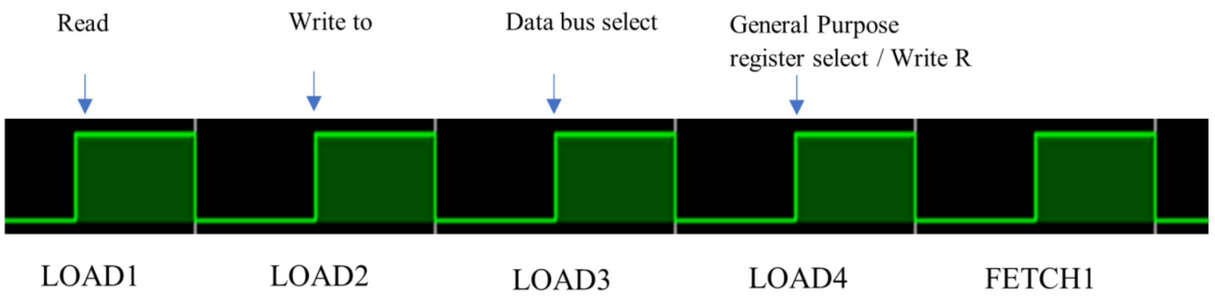
The processor doesn't start to change the state from the START (idle) to other states until the enable signal coming from the outside. After enable signal comes at the rising edge, processer state changes from START to FETCH1 then to FETCH2 and then to FETCH3. In the FETCH1 state processor request instruction from IRAM and in FETCH2 no operation done and in FETCH3 state, instructions will be decoded. According to the content of instruction the next state will be decided. This is described below timing diagram.



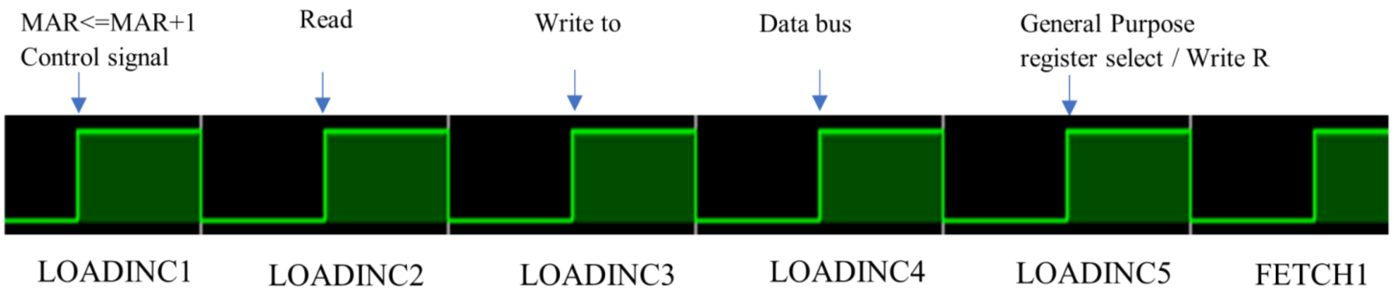
1.NOP



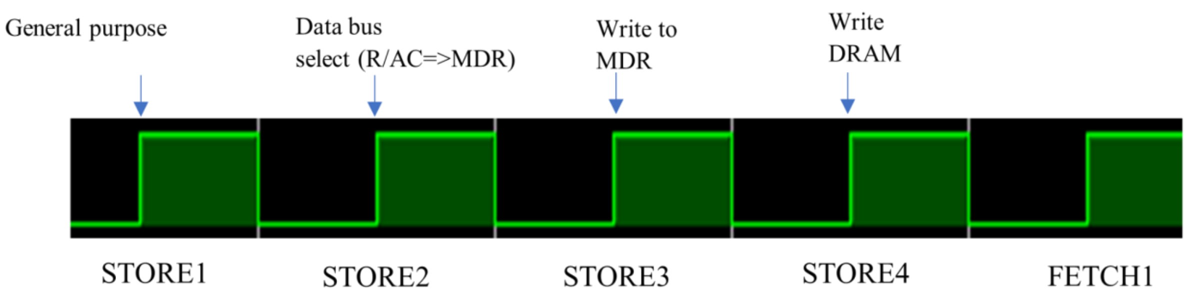
2.LOAD



3.LOADINC

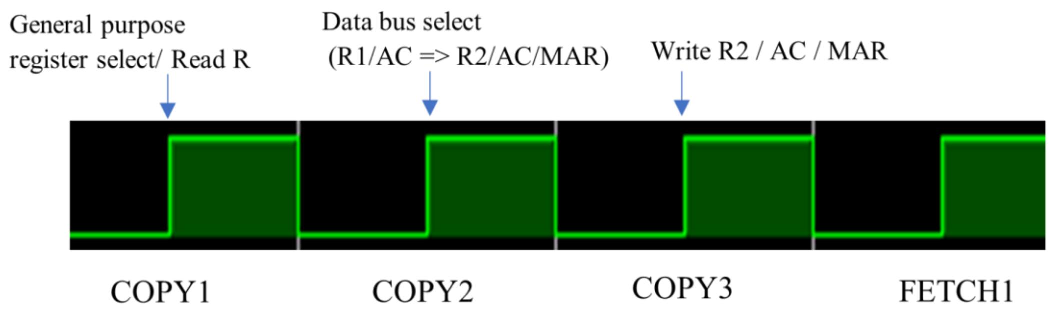

MAR<=MAR+1 Control signal
Read
Write to
Data bus
General Purpose register select / Write R
LOADINC1
LOADINC2
LOADINC3
LOADINC4
LOADINC5
FETCH1

4.STORE



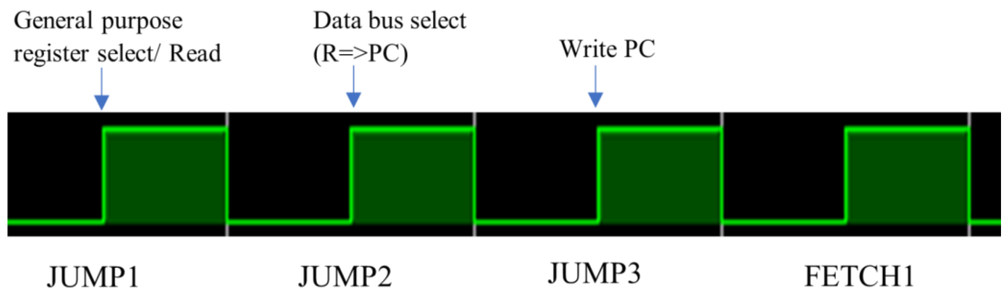
When data store from AC to DRAM, STORE1 will be ignored.

5.COPY

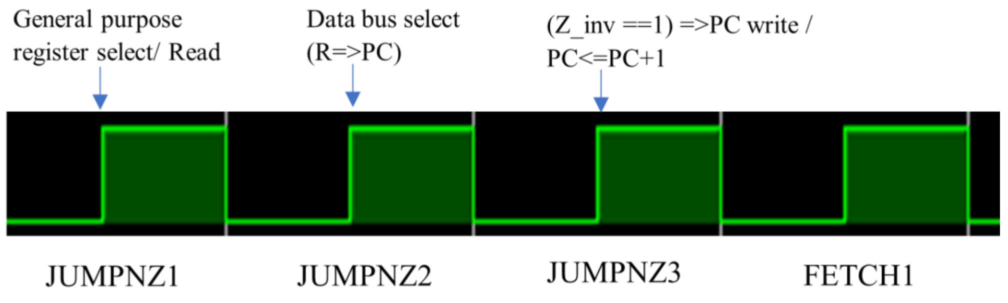

General purpose register select/ Read R
Data bus select (R1/AC => R2/AC/MAR)
Write R2 / AC / MAR
COPY1
COPY2
COPY3
FETCH1

When data store from AC COPY1 will be ignored.

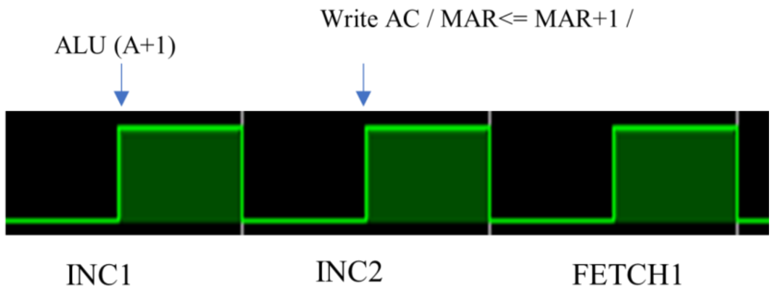
6.JUMP



7.JUMPNZ

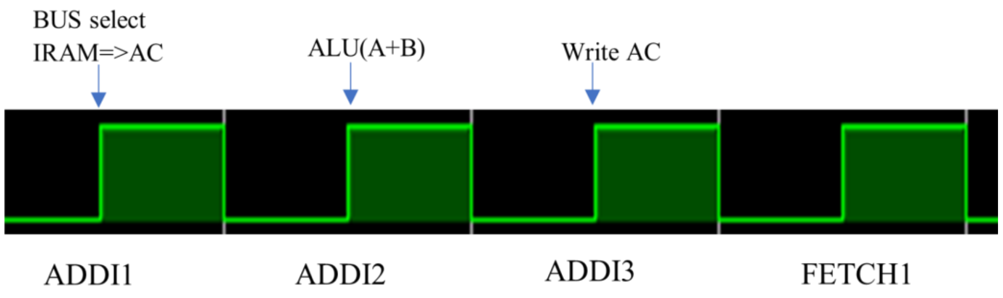

General purpose register select/ Read
Data bus select (R=>PC)
(Z_inv ==1) =>PC write / PC<=PC+1
JUMPNZ1
JUMPNZ2
JUMPNZ3
FETCH1

8.INC

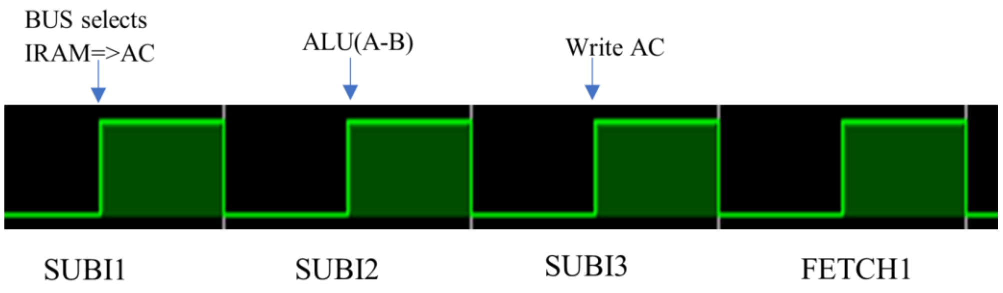


When increment MAR and general-purpose register, INC1 ignored.

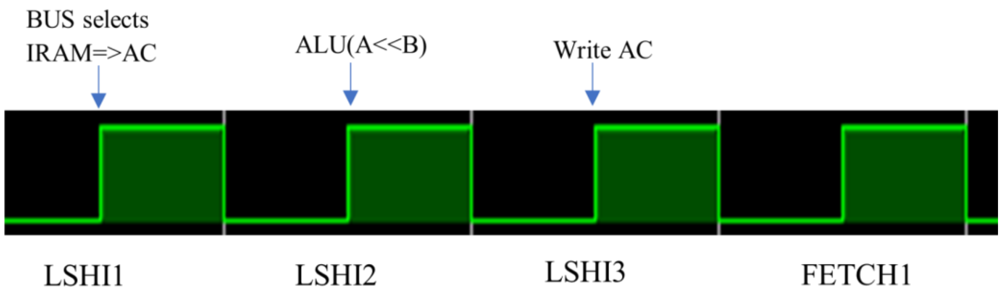
9.ADDI



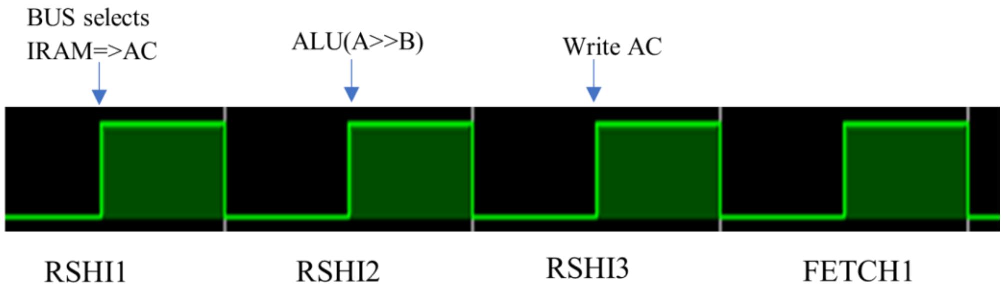
10. SUBI



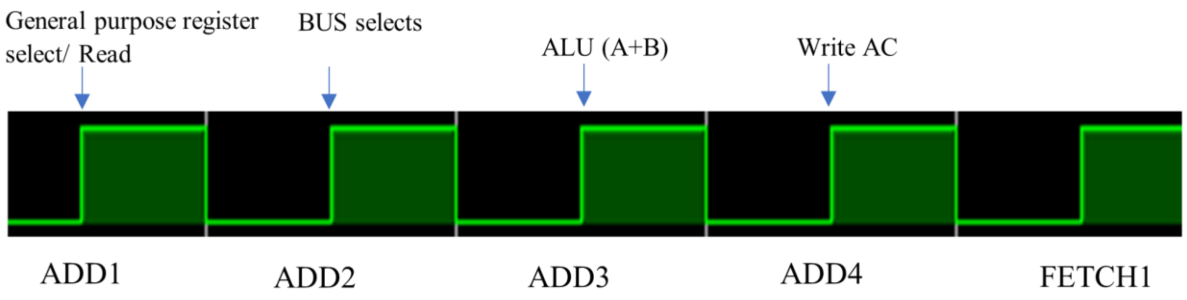
11.LFSHI



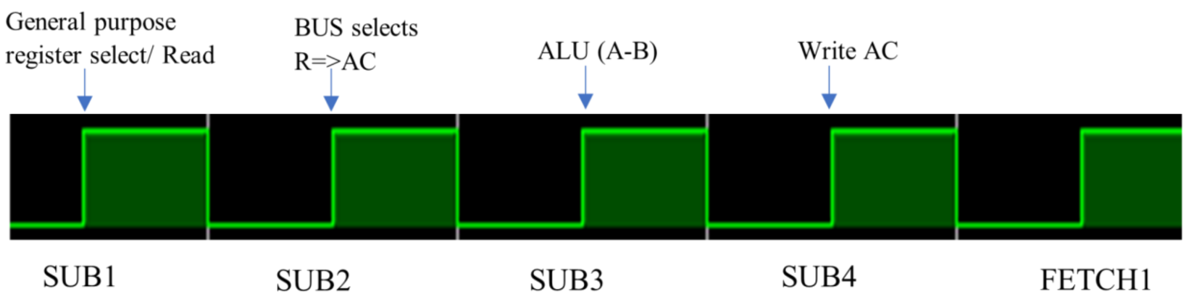
12.RSHI



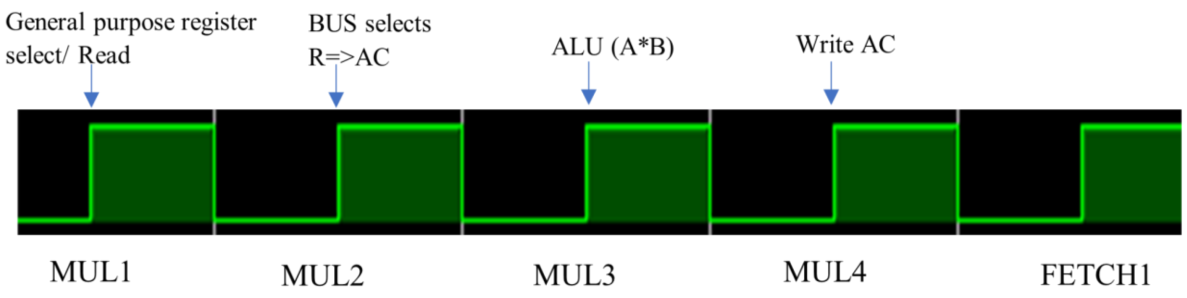
13.ADD



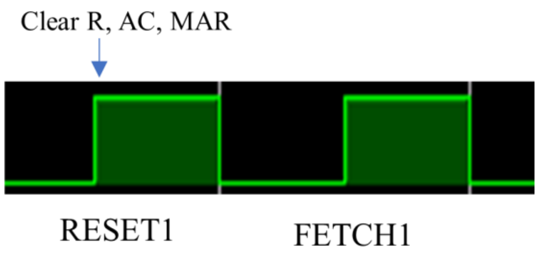
14.SUB



15. MUL



16.RESET



7.Algorithm

Down sampling using a Gaussian Filter

We can Gaussian smoothing for cut off high frequencies so that we can down samples the image without missing the image details. This is low pass filtering method in image processing. Downsam0ling without blurring can produce visually weird annoying aliasing artefacts. The 3x3 Gaussian kernel used in shown below.

0.0622	0.2489	0.0622
0.2489	1	0.2489
0.0622	0.2489	0.0622

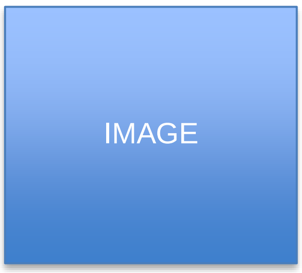
However, since our processor does not support floating point arithmetic an approximated Gaussian kernel of size 3x3 we can use, and result values is divided by total.

1	2	1
2	4	2
1	2	1

We can apply this image using these two linear separable kernels in vertical and horizontal directions as follows.

Vertical

1
2
1



Horizontal

1	2	1

After smoothing the image now, we can down sample the image by ignoring the pixels as follows.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Assembly Code for Down sampling using a Gaussian Filter

```
NOP
RESET AC
RESET MAR
RESET R1
RESET R2
LOAD R3
LOADINC R4
LOADINC R5
LOADINC R6
LOADINC R7
RESET R8
RESET R9
RESET R10
RESET R11
COPY R1 AC
MUL R3
ADD R5
COPY AC R8
COPY AC MAR
LOAD R9
LOADINC R10
LOADINC R11
COPY R9 AC
LSHI 1
ADD R10
RSHI 2
COPY R8 MAR
STORE AC
RESET R2
INC R2
COPY R10 AC
LSHI 1
ADD R9
ADD R11
```

```
RSHI 2
COPY AC R9
INC R8
COPY R8 MAR
STORE R9
COPY R10 R9
COPY R11 R10
COPY R1 AC
MUL R3
ADD R5
ADD R2
INC AC
INC AC
COPY AC MAR
LOAD R11
INC R2
COPY R3 AC
SUB R2
SUBI 1
JUMPNZ R7
COPY R10 AC
LSHI 1
ADD R9
RSHI 2
COPY AC R9
INC R8
COPY R8 MAR
STORE R9
INC R1
COPY R4 AC
SUB R1
JUMPNZ R6
RESET R1
RESET R2
RESET MAR
RESET AC
ADDI 5
COPY AC MAR
LOAD R6
INC MAR
LOAD R7
RESET R8
COPY R2 AC
ADD R5
COPY AC R8
COPY AC MAR
```

```
LOAD R9
ADD R3
COPY AC MAR
LOAD R10
ADD R3
COPY AC MAR
LOAD R11
COPY R9 AC
LSHI 1
ADD R10
RSHI 2
COPY R8 MAR
STORE AC
RESET R1
INC R1
COPY R10 AC
LSHI 1
ADD R9
ADD R11
RSHI 2
COPY AC R9
COPY R8 AC
ADD R3
COPY AC R8
COPY AC MAR
STORE R9
COPY R10 R9
COPY R11 R10
COPY R1 AC
INC AC
INC AC
MUL R3
ADD R5
ADD R2
COPY AC MAR
LOAD R11
INC R1
COPY R4 AC
SUB R1
SUBI 1
JUMPNZ R7
COPY R10 AC
LSHI 1
ADD R9
RSHI 2
COPY AC R9
COPY R8 AC
ADD R3
COPY AC R8
COPY AC MAR
STORE R9
INC R2
COPY R3 AC
SUB R2
JUMPNZ R6
RESET R1
RESET R2
RESET MAR
RESET AC
ADDI 7
COPY AC MAR
LOAD R6
INC MAR
LOAD R7
RESET AC
ADD R5
COPY AC R8
COPY R1 AC
MUL R3
ADD R2
ADD R5
COPY AC MAR
LOAD R9
COPY R8 MAR
INC R8
STORE R9
COPY R2 AC
ADDI 2
COPY AC R2
COPY R3 AC
SUB R2
JUMPNZ R7
RESET R2
COPY R1 AC
ADDI 2
COPY AC R1
COPY R4 AC
SUB R1
JUMPNZ R6
RESET R1
NOP
```

8. Assembler

A compiler uses to translate a program written in High Level Language into Binary code. In Assembler converts the assembly code that human can understandable into Binary Code that can understand to the Processor. Assembler code is written in Python. It takes a text file as the input in which the program is written in assembly code using the instructions from the pre-defined instruction set. Compiler converts each line in assembly code into binary code.

Binary Code

```
M[0] = 16'b0000000000000000;
M[1] = 16'b0111100000001011;
M[2] = 16'b0111100000001100;
M[3] = 16'b0111100000000000;
M[4] = 16'b0111100000000001;
M[5] = 16'b0000100000000010;
M[6] = 16'b0001000000000011;
M[7] = 16'b0001000000000100;
M[8] = 16'b0001000000000101;
M[9] = 16'b0001000000000110;
M[10] = 16'b0111100000000111;
M[11] = 16'b0111100000001000;
M[12] = 16'b0111100000001001;
M[13] = 16'b0111100000001010;
M[14] = 16'b0010000000001011;
M[15] = 16'b0111000000000010;
M[16] = 16'b0110000000000100;
M[17] = 16'b0010001011000111;
M[18] = 16'b0010001011001100;
M[19] = 16'b0000100000001000;
M[20] = 16'b0001000000001001;
M[21] = 16'b0001000000001010;
M[22] = 16'b0010001000001011;
M[23] = 16'b0101000000000001;
M[24] = 16'b0110000000001001;
M[25] = 16'b0101100000000010;
M[26] = 16'b0010000111001100;
M[27] = 16'b0001100000001011;
M[28] = 16'b0111100000000001;
M[29] = 16'b0011100000000001;
M[30] = 16'b0010001001001011;
M[31] = 16'b0101000000000001;
M[32] = 16'b0110000000001000;
M[33] = 16'b0110000000001010;
M[34] = 16'b0101100000000010;
```

```
M[35] = 16'b0010001011001000;
M[36] = 16'b0011100000000111;
M[37] = 16'b0010000111001100;
M[38] = 16'b0001100000001000;
M[39] = 16'b0010001001001000;
M[40] = 16'b0010001010001001;
M[41] = 16'b0010000000001011;
M[42] = 16'b0111000000000010;
M[43] = 16'b0110000000000100;
M[44] = 16'b0110000000000001;
M[45] = 16'b0011100000001011;
M[46] = 16'b0011100000001011;
M[47] = 16'b0010001011001100;
M[48] = 16'b0000100000001010;
M[49] = 16'b0011100000000001;
M[50] = 16'b0010000010001011;
M[51] = 16'b0110100000000001;
M[52] = 16'b0100100000000001;
M[53] = 16'b0011000000000110;
M[54] = 16'b0010001001001011;
M[55] = 16'b0101000000000001;
M[56] = 16'b0110000000001000;
M[57] = 16'b0101100000000010;
M[58] = 16'b0010001011001000;
M[59] = 16'b0011100000000111;
M[60] = 16'b0010000111001100;
M[61] = 16'b0001100000001000;
M[62] = 16'b0011100000000000;
M[63] = 16'b0010000011001011;
M[64] = 16'b0110100000000000;
M[65] = 16'b0011000000000101;
M[66] = 16'b0111100000000000;
M[67] = 16'b0111100000000001;
M[68] = 16'b0111100000001100;
M[69] = 16'b0111100000001011;
M[70] = 16'b0100000000000101;
M[71] = 16'b0010001011001100;
M[72] = 16'b0000100000000101;
M[73] = 16'b0011100000001100;
M[74] = 16'b0000100000000110;
M[75] = 16'b0111100000000111;
M[76] = 16'b0010000001001011;
M[77] = 16'b0110000000000100;
M[78] = 16'b0010001011000111;
M[79] = 16'b0010001011001100;
M[80] = 16'b0000100000001000;
```

```
M[81] = 16'b0110000000000010;
M[82] = 16'b0010001011001100;
M[83] = 16'b0000100000001001;
M[84] = 16'b0110000000000010;
M[85] = 16'b0010001011001100;
M[86] = 16'b0000100000001010;
M[87] = 16'b0010001000001011;
M[88] = 16'b0101000000000001;
M[89] = 16'b0110000000001001;
M[90] = 16'b0101100000000010;
M[91] = 16'b0010000111001100;
M[92] = 16'b0001100000001011;
M[93] = 16'b0111100000000000;
M[94] = 16'b0011100000000000;
M[95] = 16'b0010001001001011;
M[96] = 16'b0101000000000001;
M[97] = 16'b0110000000001000;
M[98] = 16'b0110000000001010;
M[99] = 16'b0101100000000010;
M[100] = 16'b0010001011001000;
M[101] = 16'b0010000111001011;
M[102] = 16'b0110000000000010;
M[103] = 16'b0010001011000111;
M[104] = 16'b0010001011001100;
M[105] = 16'b0001100000001000;
M[106] = 16'b0010001001001000;
M[107] = 16'b0010001010001001;
M[108] = 16'b0010000000001011;
M[109] = 16'b0011100000001011;
M[110] = 16'b0011100000001011;
M[111] = 16'b0111000000000010;
M[112] = 16'b0110000000000100;
M[113] = 16'b0110000000000001;
M[114] = 16'b0010001011001100;
M[115] = 16'b0000100000001010;
M[116] = 16'b0011100000000000;
M[117] = 16'b0010000011001011;
M[118] = 16'b0110100000000000;
M[119] = 16'b0100100000000001;
M[120] = 16'b0011000000000110;
M[121] = 16'b0010001001001011;
M[122] = 16'b0101000000000001;
M[123] = 16'b0110000000001000;
M[124] = 16'b0101100000000010;
M[125] = 16'b0010001011001000;
M[126] = 16'b0010000111001011;
```

```
M[127] = 16'b0110000000000010;
M[128] = 16'b0010001011000111;
M[129] = 16'b0010001011001100;
M[130] = 16'b0001100000001000;
M[131] = 16'b0011100000000001;
M[132] = 16'b0010000010001011;
M[133] = 16'b0110100000000001;
M[134] = 16'b0011000000000101;
M[135] = 16'b0111100000000000;
M[136] = 16'b0111100000000001;
M[137] = 16'b0111100000001100;
M[138] = 16'b0111100000001011;
M[139] = 16'b0100000000000111;
M[140] = 16'b0010001011001100;
M[141] = 16'b0000100000000101;
M[142] = 16'b0011100000001100;
M[143] = 16'b0000100000000110;
M[144] = 16'b0111100000001011;
M[145] = 16'b0110000000000100;
M[146] = 16'b0010001011000111;
M[147] = 16'b0010000000001011;
M[148] = 16'b0111000000000010;
M[149] = 16'b0110000000000001;
M[150] = 16'b0110000000000100;
M[151] = 16'b0010001011001100;
M[152] = 16'b0000100000001000;
M[153] = 16'b0010000111001100;
M[154] = 16'b0011100000000111;
M[155] = 16'b0001100000001000;
M[156] = 16'b0010000001001011;
M[157] = 16'b0100000000000010;
M[158] = 16'b0010001011000001;
M[159] = 16'b0010000010001011;
M[160] = 16'b0110100000000001;
M[161] = 16'b0011000000000110;
M[162] = 16'b0111100000000001;
M[163] = 16'b0010000000001011;
M[164] = 16'b0100000000000010;
M[165] = 16'b0010001011000000;
M[166] = 16'b0010000011001011;
M[167] = 16'b0110100000000000;
M[168] = 16'b0011000000000101;
M[169] = 16'b0111100000000000;
M[170] = 16'b0000000000000000;
```

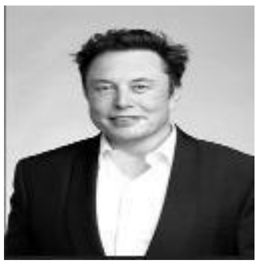
9.Debugging

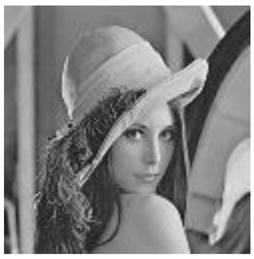
After writing each verilog module for implement processor components we wrote test benches to validate modules are working properly .We wrote test benches for ALU, control unit, DRAM ,IRAM, MDR,MAR, AC,general puropose Register bank and data BUS. After giving some stimulates through test benches and by looking waveforms at the waveform viewer of the Vivado, we did the changes in the corresponding verilog modules until they are proper working . After connecting all the module together via top module, we wrote final test bench for testing the complete processor. By using the waveform viewer,we did the all corrections in the modules and assembly code as well.

10.Results

Original image (250x250)

Down sampled image (125x 125)



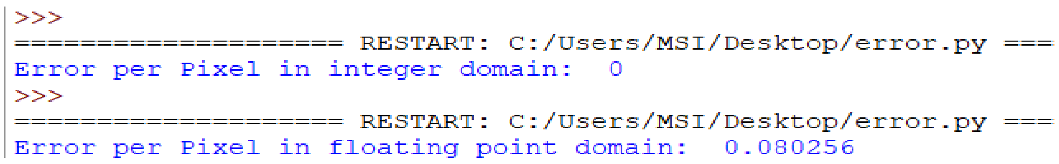


Disney
BIG
HERO
6

Disney
BIG
HERO
6

11.Error analysis

To verify the implementation of the algorithm and ALU arithmetic operations of the processor it is necessary to calculate the standard error and compare two down sampled images of processor and the external python program under same conditions. So we down sampled a image using the processor and then down sampled the same image using python where we used same algorithm for down sampling as in the assembly code. Then we fed both outputs to a python code to calculate the error where we get the difference between corresponding pixels of two outputs and then get SSD using them. After getting square root of the SSD we divided it from the number of pixels to get the error per pixel. The output of the python code is shown below.



```
>>>
==================== RESTART: C:/Users/MSI/Desktop/error.py ===
Error per Pixel in integer domain:  0
>>>
==================== RESTART: C:/Users/MSI/Desktop/error.py ===
Error per Pixel in floating point domain:  0.080256
```

According to the above results we can clearly see that the error is 0 since we are dealing in the integer domain. But if we are working with the floating points then there is a non-zero error because the processor only works in integer domain. Therefore, we can clearly see that there is a non-zero error when we use floating point arithmetic to down sample the image. Also, the error can be dependent on the image.

12. Conclusion

- In this processor design project, we should be able to design a processor for down sampling a given 250 x 250 image to its half size 125 x 125. To full fill our task we created an instruction set ourselves and implemented a processor which can specifically execute the given instructions in an optimum way. Since this is an application specific processor we ignored some register uses in general purpose processors which are not important for execution of assembly instructions.
- The algorithm used here is uses Gaussian filter. Also we would be able to get an error free result. All the multiplications and divisions involved in this filtering process are implemented by shifting bit values so that there will be no round off errors.
- Each module we implemented in verilog is synchronised with clock signal and that clock signal is given by the outside test bench module, where we are able to change clock frequency simply by changing test bench. For general test we used 1Ghz clock signal and it works accurately for higher speeds such as 4 Ghz and lower speeds like 10 Mhz as well.
- The data path we have implemented here includes one main data bus so that we can swap data between registers easily with less number of clock cycles. This will accelerate the process as the COPY is the most used assembly instruction in our assembly code.
- Our processor takes around 20ms to down sample an image. By further studying we decided that by introducing several new instructions that are not in the ISA we could reduce the processing time.
- To implement the general purpose register we used a register bank working with multiplexer which will facilitate a reduced number of control signals and it only uses one additional ALU to increase the value of any general-purpose register, so that resources will be minimized. However, as a drawback this will cost additional clock cycle for each register operation where the time consumption will be high. Since we were more weighted towards using minimal resources than increasing processing speed, we decided to use register bank instead of using single general purpose registers.
- Further, we can use pipelines to improve our processor so that overall time consumption will be reduced, and processor speed will be increased.
- We can use other image processing techniques such as edge detection blurring, normalizing and other things in this processor to improve output quality with or without using additional instructions depending on the image processing technique.
- We can also increase the accuracy of the down sampled image by using floating point arithmetic for the processor.

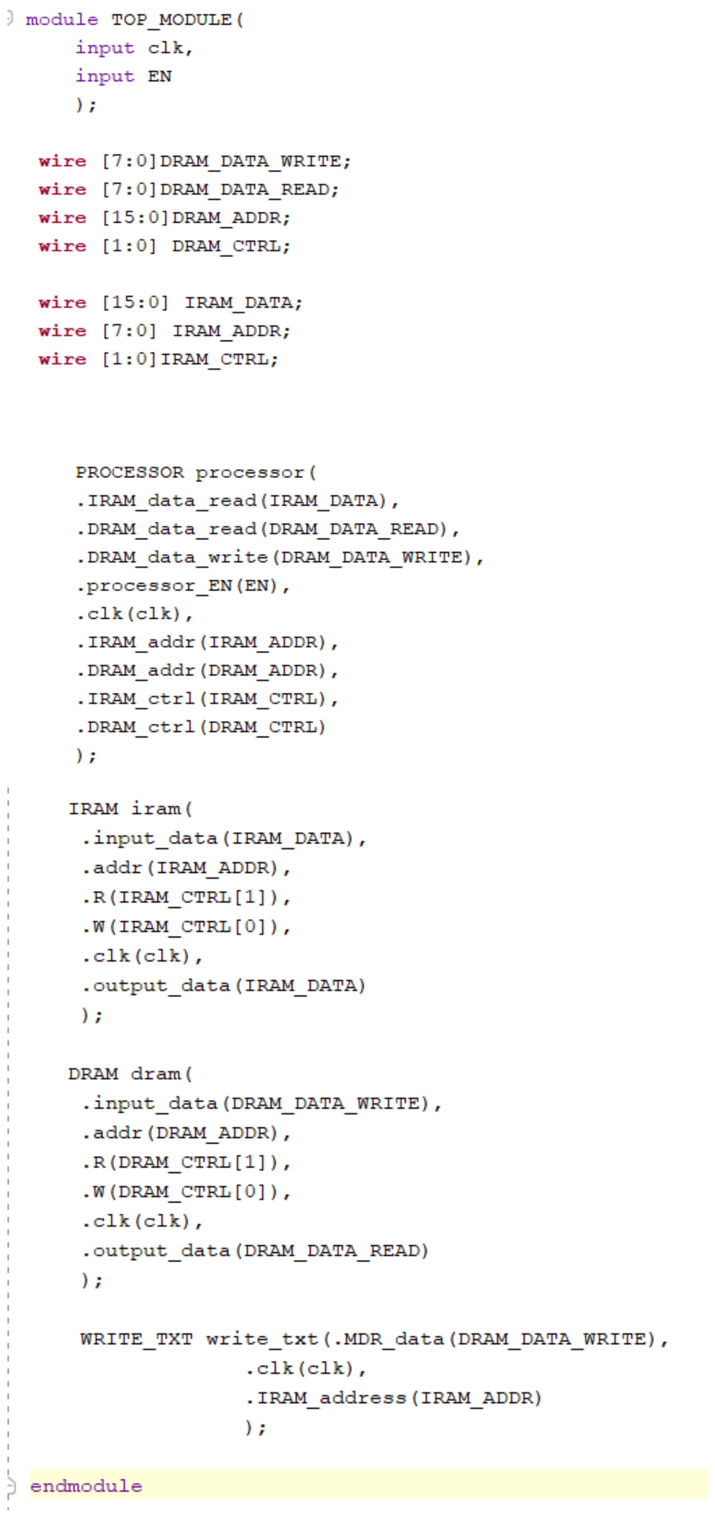
13.References

- https://www.youtube.com/watch?v=PJGvZSlsLKs
- https://www.youtube.com/watch?v=q1QwC3YlHG0
- https://youtube.com/playlist?list=PLUtfVcb-iqn-EkuBs3arreilxa2UKIChl
- https://en.wikipedia.org/wiki/Datapath#:~:text=A%20datapath%20is%20a%20collection,central%20processing%20unit%20(CPU).
- EN3030 Lecture notes

14.Appendices

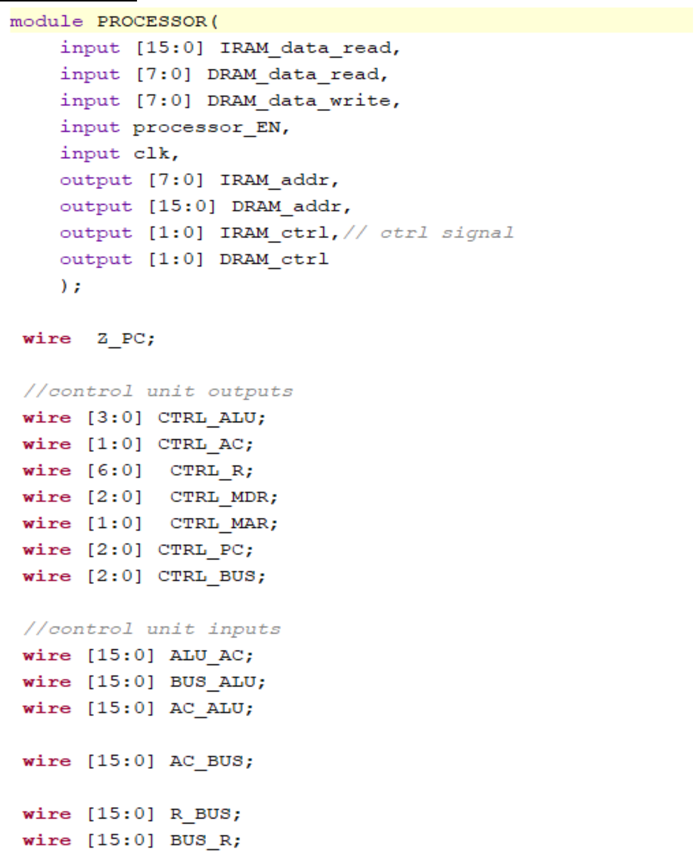
Verilog Codes

Top Module

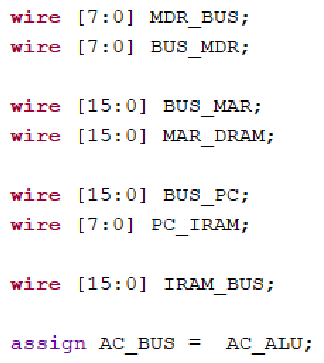


```verilog
module TOP_MODULE(
    input clk,
    input EN
    );

 wire [7:0]DRAM_DATA_WRITE;
 wire [7:0]DRAM_DATA_READ;
 wire [15:0]DRAM_ADDR;
 wire [1:0] DRAM_CTRL;

 wire [15:0] IRAM_DATA;
 wire [7:0] IRAM_ADDR;
 wire [1:0]IRAM_CTRL;



    PROCESSOR processor(
    .IRAM_data_read(IRAM_DATA),
    .DRAM_data_read(DRAM_DATA_READ),
    .DRAM_data_write(DRAM_DATA_WRITE),
    .processor_EN(EN),
    .clk(clk),
    .IRAM_addr(IRAM_ADDR),
    .DRAM_addr(DRAM_ADDR),
    .IRAM_ctrl(IRAM_CTRL),
    .DRAM_ctrl(DRAM_CTRL)
    );

    IRAM iram(
     .input_data(IRAM_DATA),
     .addr(IRAM_ADDR),
     .R(IRAM_CTRL[1]),
     .W(IRAM_CTRL[0]),
     .clk(clk),
     .output_data(IRAM_DATA)
     );

    DRAM dram(
     .input_data(DRAM_DATA_WRITE),
     .addr(DRAM_ADDR),
     .R(DRAM_CTRL[1]),
     .W(DRAM_CTRL[0]),
     .clk(clk),
     .output_data(DRAM_DATA_READ)
     );

     WRITE_TXT write_txt(.MDR_data(DRAM_DATA_WRITE),
                  .clk(clk),
                  .IRAM_address(IRAM_ADDR)
                  );

endmodule
```

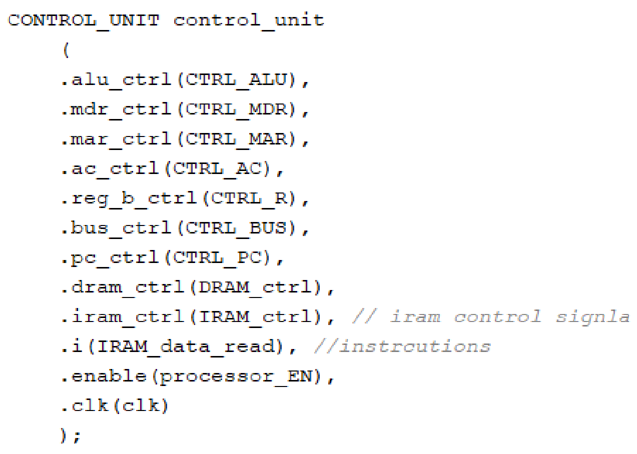
Processor



```
module PROCESSOR(
    input [15:0] IRAM_data_read,
    input [7:0] DRAM_data_read,
    input [7:0] DRAM_data_write,
    input processor_EN,
    input clk,
    output [7:0] IRAM_addr,
    output [15:0] DRAM_addr,
    output [1:0] IRAM_ctrl,// ctrl signal
    output [1:0] DRAM_ctrl
    );

 wire  Z_PC;

 //control unit outputs
 wire [3:0] CTRL_ALU;
 wire [1:0] CTRL_AC;
 wire [6:0]  CTRL_R;
 wire [2:0]  CTRL_MDR;
 wire [1:0]  CTRL_MAR;
 wire [2:0] CTRL_PC;
 wire [2:0] CTRL_BUS;

 //control unit inputs
 wire [15:0] ALU_AC;
 wire [15:0] BUS_ALU;
 wire [15:0] AC_ALU;

 wire [15:0] AC_BUS;

 wire [15:0] R_BUS;
 wire [15:0] BUS_R;
```



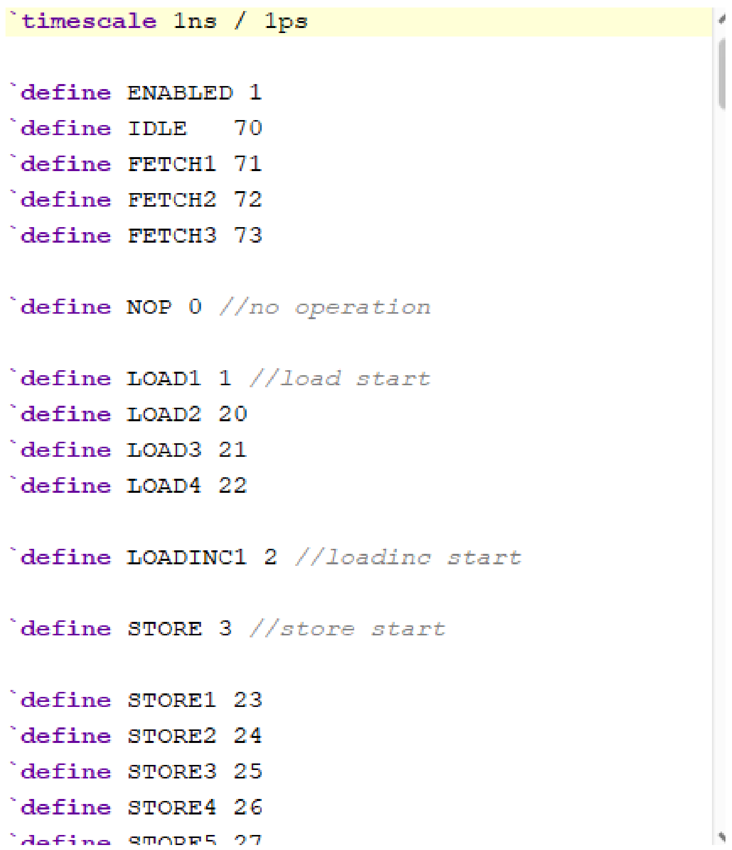
```
wire [7:0] MDR_BUS;
wire [7:0] BUS_MDR;

wire [15:0] BUS_MAR;
wire [15:0] MAR_DRAM;

wire [15:0] BUS_PC;
wire [7:0] PC_IRAM;

wire [15:0] IRAM_BUS;

assign AC_BUS =  AC_ALU;
```



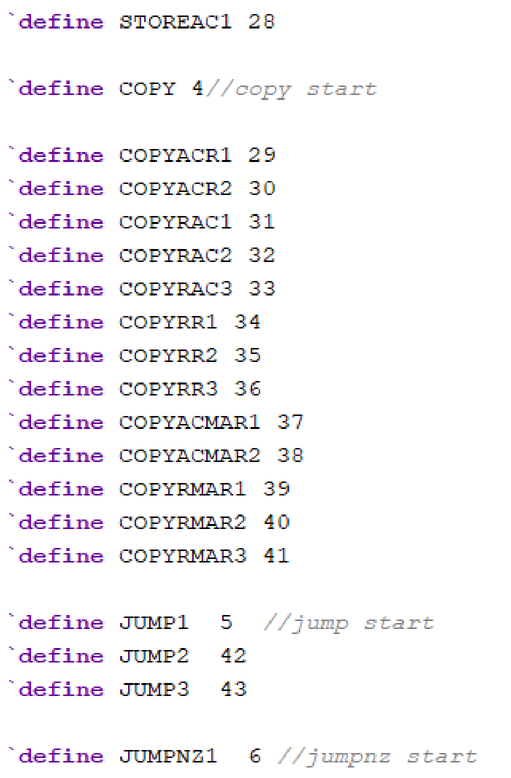
```
CONTROL_UNIT control_unit
    (
    .alu_ctrl(CTRL_ALU),
    .mdr_ctrl(CTRL_MDR),
    .mar_ctrl(CTRL_MAR),
    .ac_ctrl(CTRL_AC),
    .reg_b_ctrl(CTRL_R),
    .bus_ctrl(CTRL_BUS),
    .pc_ctrl(CTRL_PC),
    .dram_ctrl(DRAM_ctrl),
    .iram_ctrl(IRAM_ctrl), // iram control signla
    .i(IRAM_data_read), //instrcutions
    .enable(processor_EN),
    .clk(clk)
    );
```

```
ALU alu(
    .A(AC_ALU),
    .B(BUS_ALU),
    .ctrl(CTRL_ALU),
    .clk(clk),
    .ALU_out(ALU_AC),
    .Z_inv(Z_PC)
    );

AC ac(
    .in(ALU_AC),
    .ctrl(CTRL_AC),
    .clk(clk),
    .out(AC_ALU)
    );

REG_B reg_b(
    .in(BUS_R),
    .ctrl(CTRL_R),
    .clk(clk),
    .out(R_BUS)
    );

MDR mdr(
    .DRAM_MDR(DRAM_data_read),
    .BUS_MDR(BUS_MDR),
    .ctrl(CTRL_MDR),
    .clk(clk),
    .MDR_DRAM(DRAM_data_write),
    .MDR_BUS(MDR_BUS)
    );


MAR mar(
    .in(BUS_MAR),
    .ctrl(CTRL_MAR),
    .clk(clk),
    .out(DRAM_addr)
    );

PC pc(
    .Z_inv(Z_PC),
    .ctrl(CTRL_PC),
    .in_addr(BUS_PC),
    .clk(clk),
    .out_addr(IRAM_addr)
    );

BUS bus(
    .MDR_BUS(MDR_BUS),
    . R_BUS(R_BUS),
    .AC_BUS(AC_BUS),
    .IR_BUS(IRAM_data_read),
    .clk(clk),
    .src(CTRL_BUS),
    .BUS_MDR(BUS_MDR),
    .BUS_R(BUS_R),
    .BUS_ALU(BUS_ALU),
    .BUS_MAR(BUS_MAR),
    .BUS_PC(BUS_PC)
    );
endmodule
```

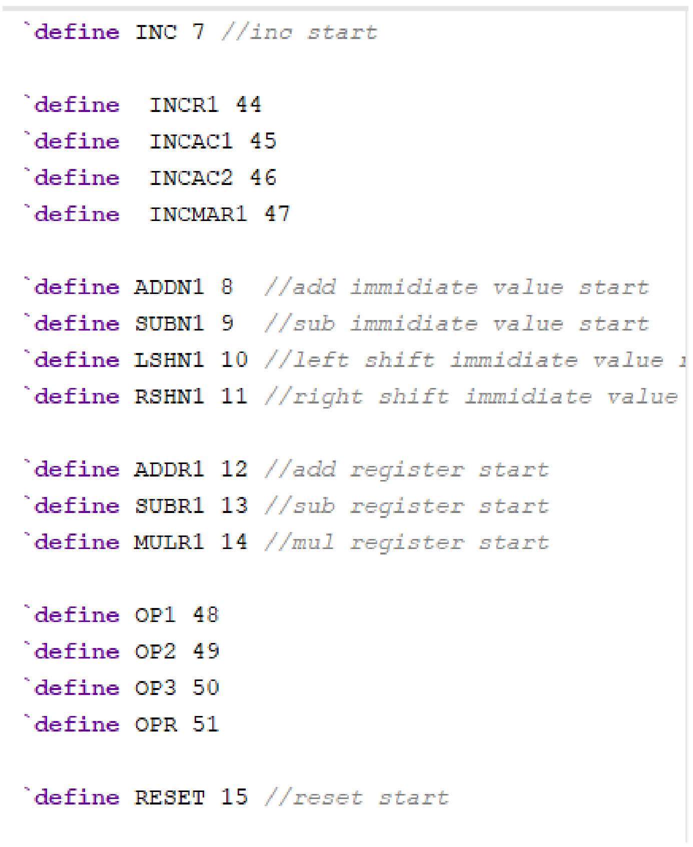
Control Unit



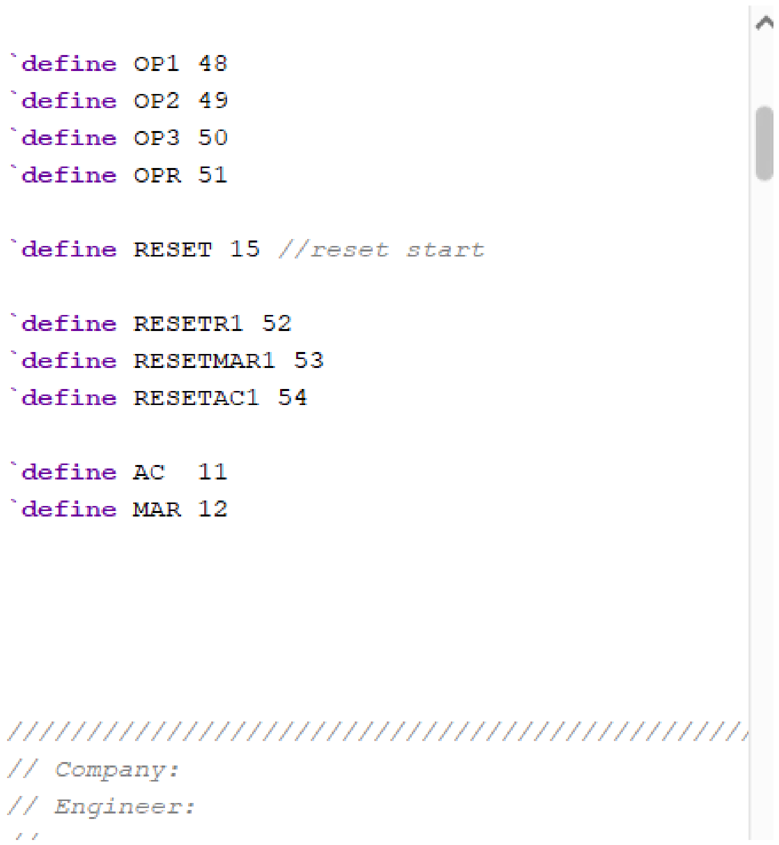
```
`timescale 1ns / 1ps

`define ENABLED 1
`define IDLE   70
`define FETCH1 71
`define FETCH2 72
`define FETCH3 73

`define NOP 0 //no operation

`define LOAD1 1 //load start
`define LOAD2 20
`define LOAD3 21
`define LOAD4 22

`define LOADINC1 2 //loadinc start

`define STORE 3 //store start

`define STORE1 23
`define STORE2 24
`define STORE3 25
`define STORE4 26
`define STORE5 27
```



```
`define STOREAC1 28

`define COPY 4//copy start

`define COPYACR1 29
`define COPYACR2 30
`define COPYRAC1 31
`define COPYRAC2 32
`define COPYRAC3 33
`define COPYRR1 34
`define COPYRR2 35
`define COPYRR3 36
`define COPYACMAR1 37
`define COPYACMAR2 38
`define COPYRMAR1 39
`define COPYRMAR2 40
`define COPYRMAR3 41

`define JUMP1  5  //jump start
`define JUMP2  42
`define JUMP3  43

`define JUMPNZ1  6 //jumpnz start
```



```
`define INC 7 //inc start

`define  INCR1 44
`define  INCAC1 45
`define  INCAC2 46
`define  INCMAR1 47

`define ADDN1 8  //add immidiate value start
`define SUBN1 9  //sub immidiate value start
`define LSHN1 10 //left shift immidiate value
`define RSHN1 11 //right shift immidiate value

`define ADDR1 12 //add register start
`define SUBR1 13 //sub register start
`define MULR1 14 //mul register start

`define OP1 48
`define OP2 49
`define OP3 50
`define OPR 51

`define RESET 15 //reset start
```



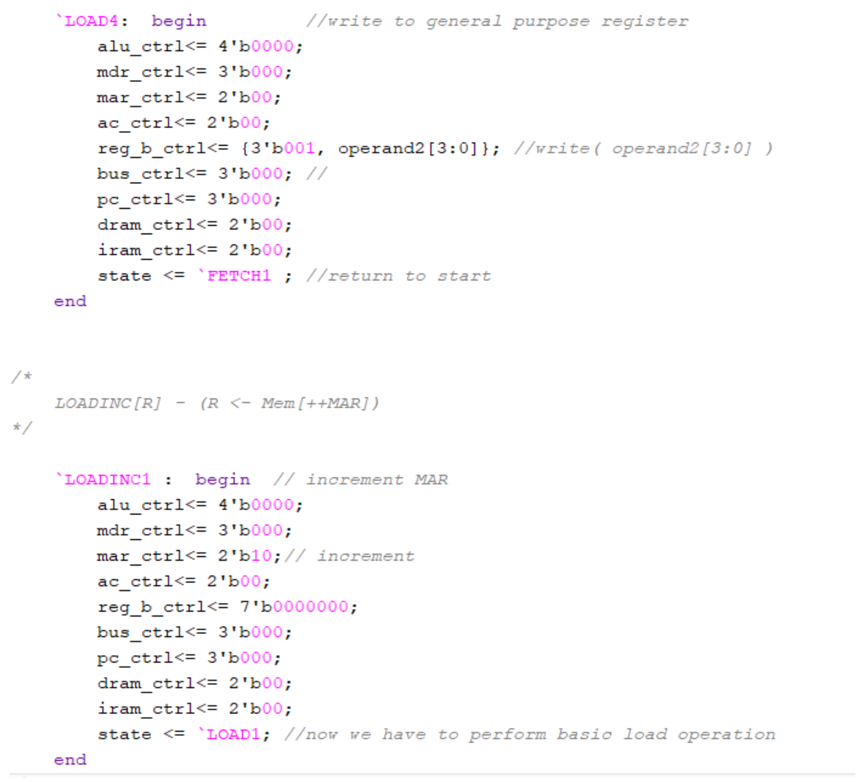
```
`define OP1 48
`define OP2 49
`define OP3 50
`define OPR 51

`define RESET 15 //reset start

`define RESETR1 52
`define RESETMAR1 53
`define RESETAC1 54

`define AC  11
`define MAR 12




///////////////////////////////////////////////
// Company:
// Engineer:
```

```
module CONTROL_UNIT(
    output reg [3:0] alu_ctrl,
    output reg [2:0] mdr_ctrl,
    output reg [1:0] mar_ctrl,
    output reg [1:0] ac_ctrl,
    output reg [6:0] reg_b_ctrl,
    output reg [2:0] bus_ctrl,
    output reg [2:0] pc_ctrl,
    output reg [1:0] dram_ctrl,
    output reg [1:0] iram_ctrl,
    input [15:0] i,
    input enable,
    input clk
    );


   reg [4:0] opcode;
   reg [4:0] operand1;
   reg [4:0] operand2;
   reg [8:0] state;

   initial state = `IDLE;

   always @(i) begin
    opcode = i[15:11];
    operand1 = i[10:6];
    operand2 = i[4:0];
    end

  always @(posedge clk)begin
  case (state)
  `IDLE : begin
      if(enable == `ENABLED) begin ; //in idle ,PC shold clear
           alu_ctrl<= 4'b0000;
           mdr_ctrl<= 3'b000;
           mar_ctrl<= 2'b00;
           ac_ctrl<= 2'b00;
           reg_b_ctrl<= 7'b0000000;
           bus_ctrl<= 3'b000;
           pc_ctrl<= 3'b100; // pc reset
           dram_ctrl<= 2'b00;
           iram_ctrl<= 2'b00;
           state <= `FETCH1;
           end
      end
  `FETCH1 :  begin //fetch1 ,enale write on iram
       alu_ctrl<= 4'b0000;
       mdr_ctrl<= 3'b000;
       mar_ctrl<= 2'b00;
       ac_ctrl<= 2'b00;
       reg_b_ctrl<= 7'b0000000;
       bus_ctrl<= 3'b000;
       pc_ctrl<= 3'b000;
       dram_ctrl<= 2'b00;
       iram_ctrl<= 2'b10; //write
       state <=`FETCH2;
   end
```

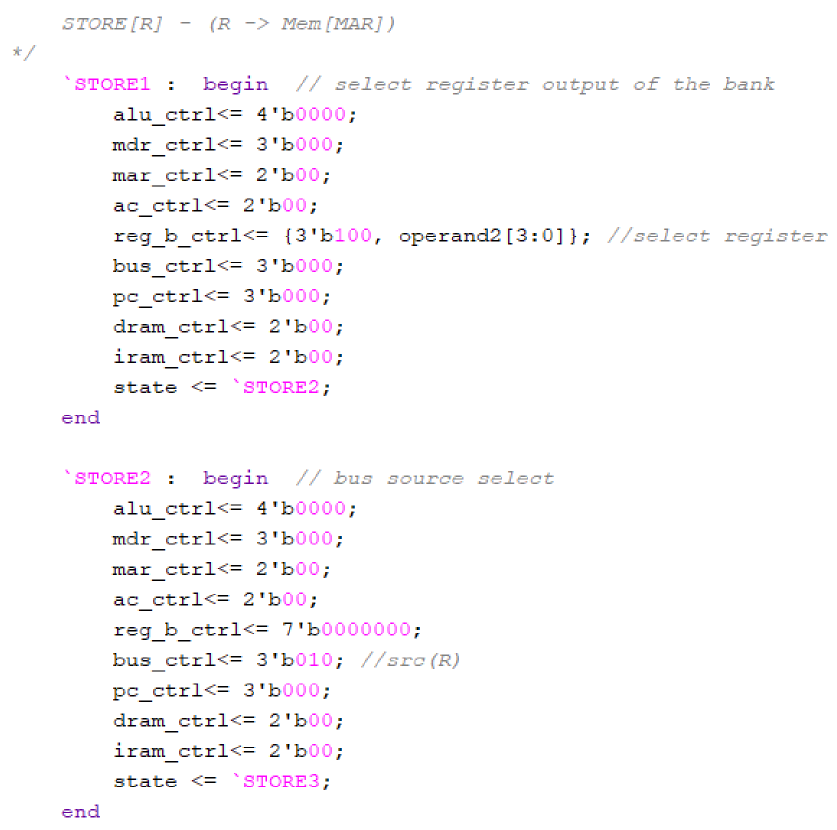
```
`FETCH2 :  begin //fetch2 ,nop
    alu_ctrl<= 4'b0000;
    mdr_ctrl<= 3'b000;
    mar_ctrl<= 2'b00;
    ac_ctrl<= 2'b00;
    reg_b_ctrl<= 7'b0000000;
    bus_ctrl<= 3'b000;
    pc_ctrl<= 3'b000;
    dram_ctrl<= 2'b00;
    iram_ctrl<= 2'b00;
    state <=`FETCH3;
end
`FETCH3 :  begin //fetch3 ,decode ,  increment pc
    alu_ctrl<= 4'b0000;
    mdr_ctrl<= 3'b000;
    mar_ctrl<= 2'b00;
    ac_ctrl<= 2'b00;
    reg_b_ctrl<= 7'b0000000;
    bus_ctrl<= 3'b000;
    pc_ctrl<= 3'b001; //increment
    dram_ctrl<= 2'b00;
    iram_ctrl<= 2'b00;
    case (opcode)
```

```
    `STORE : state <= (operand2 == `AC)? `STOREAC1 : `STORE1;
    `INC :
        if (operand2 == `AC)  state <= `INCAC1;
        else if (operand2 == `MAR)  state <= `INCMAR1;
        else state <= `INCR1;
    `COPY : begin
        if (operand1 == `AC) state <= (operand2 == `MAR)? `COPYACMAR1 : `COPYACR1;
        else begin
            if (operand2 == `AC) state <= `COPYRAC1;
            else if (operand2 == `MAR) state <= `COPYRMAR1;
            else  state <= `COPYRR1;
        end
    end
    `ADDN1 : state = `OP1;
    `SUBN1 : state = `OP1;
    `LSHN1 : state = `OP1;
    `RSHN1 : state = `OP1;
    `ADDR1 : state = `OPR;
    `SUBR1 : state = `OPR;
    `MULR1 : state = `OPR;
    `JUMPNZ1 : state = `JUMP1;
    `RESET : begin
            if (operand2 == `AC) state <= `RESETAC1;
            else if (operand2 == `MAR) state <= `RESETMAR1;
            else  state <= `RESETR1;
    end
    default state <= {8'b000 , opcode} ;
    endcase

end
```

```
    `NOP :  begin           //no control signal
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
    end
/*
  LOAD[R] - (R <- Mem[MAR])
*/

    `LOAD1 :  begin           //read from DRAM
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b10; //read
        iram_ctrl<= 2'b00;
        state <= `LOAD2 ;
    end
   ----
   `LOAD2:  begin            //write to MDR
       alu_ctrl<= 4'b0000;
       mdr_ctrl<= 3'b010; //write(DRAM_MDR)
       mar_ctrl<= 2'b00;
       ac_ctrl<= 2'b00;
       reg_b_ctrl<= 7'b0000000;
       bus_ctrl<= 3'b000;
       pc_ctrl<= 3'b000;
       dram_ctrl<= 2'b00;
       iram_ctrl<= 2'b00;
       state <= `LOAD3;
   end
   `LOAD3 :  begin        //BUS source select
       alu_ctrl<= 4'b0000;
       mdr_ctrl<= 3'b000;
       mar_ctrl<= 2'b00;
       ac_ctrl<= 2'b00;
       reg_b_ctrl<= 7'b0000000;
       bus_ctrl<= 3'b001; //src(MDR)
       pc_ctrl<= 3'b000;
       dram_ctrl<= 2'b00;
       iram_ctrl<= 2'b00;
       state <= `LOAD4 ;

   end
```

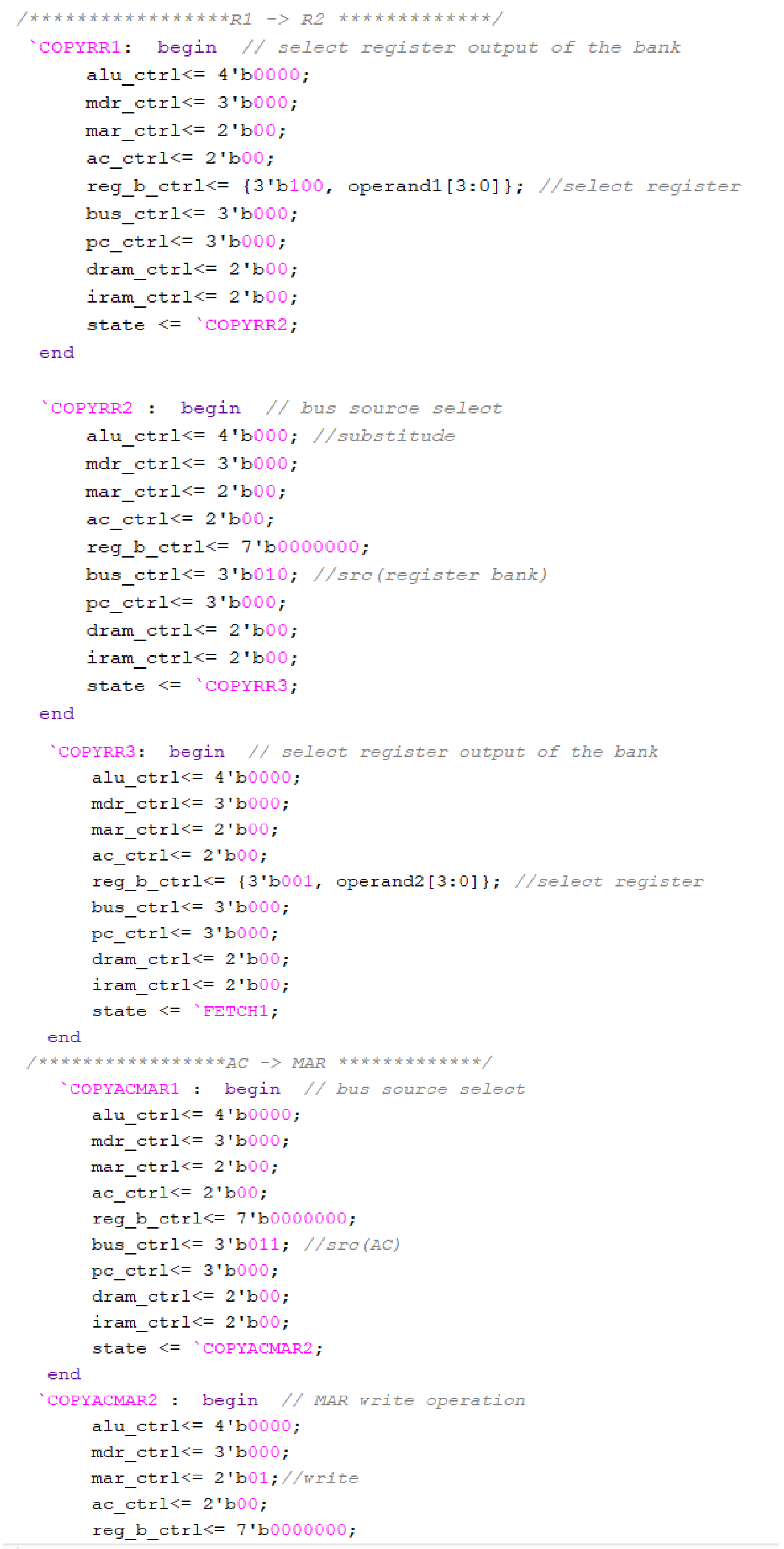


```
    `LOAD4:  begin          //write to general purpose register
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b001, operand2[3:0]}; //write( operand2[3:0] )
        bus_ctrl<= 3'b000; //
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ; //return to start
    end


/*
    LOADINC[R] - (R <- Mem[++MAR])
*/

    `LOADINC1 :  begin  // increment MAR
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b10;// increment
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `LOAD1; //now we have to perform basic load operation
    end
```



```
    STORE[R] - (R -> Mem[MAR])
 */
    `STORE1 :  begin  // select register output of the bank
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b100, operand2[3:0]}; //select register
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `STORE2;
    end

    `STORE2 :  begin  // bus source select
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b010; //src(R)
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `STORE3;
    end
    `STORE3:  begin      //write to MDR
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b001; //write(DRAM_MDR)
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `STORE4;
    end

    `STORE4 :  begin      //write to DRAM
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b01; //write
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
    end

    `STOREAC1 :  begin  // bus source select
         alu_ctrl<= 4'b0000;
         mdr_ctrl<= 3'b000;
         mar_ctrl<= 2'b00;
         ac_ctrl<= 2'b00;
         reg_b_ctrl<= 7'b0000000;
         bus_ctrl<= 3'b011; //src(AC)
         pc_ctrl<= 3'b000;
         dram_ctrl<= 2'b00;
         iram_ctrl<= 2'b00;
         state <= `STORE3;
     end


)  /*
   COPY[R1/AC][R2/AC/MAR]  -  (R1/AC - > R2/AC/MAR)
)  */

    /****************AC -> R ******/

    `COPYACR1 :  begin  // bus source select
         alu_ctrl<= 4'b0000;
         mdr_ctrl<= 3'b000;
         mar_ctrl<= 2'b00;
         ac_ctrl<= 2'b00;
         reg_b_ctrl<= 7'b0000000;
         bus_ctrl<= 3'b011; //src(AC)
         pc_ctrl<= 3'b000;
         dram_ctrl<= 2'b00;
         iram_ctrl<= 2'b00;
```

```
`COPYACR2 :  begin  // select register output of the bank
     alu_ctrl<= 4'b0000;
     mdr_ctrl<= 3'b000;
     mar_ctrl<= 2'b00;
     ac_ctrl<= 2'b00;
     reg_b_ctrl<= {3'b001, operand2[3:0]}; //select register
     bus_ctrl<= 3'b000;
     pc_ctrl<= 3'b000;
     dram_ctrl<= 2'b00;
     iram_ctrl<= 2'b00;
     state <= `FETCH1;
 end

 /*****************R -> AC *************/
`COPYRAC1:  begin  // select register output of the bank
     alu_ctrl<= 4'b0000;
     mdr_ctrl<= 3'b000;
     mar_ctrl<= 2'b00;
     ac_ctrl<= 2'b00;
     reg_b_ctrl<= {3'b100, operand1[3:0]}; //select register
     bus_ctrl<= 3'b000;
     pc_ctrl<= 3'b000;
     dram_ctrl<= 2'b00;
     iram_ctrl<= 2'b00;
     state <= `COPYRAC2;
 end
`COPYRAC2 :  begin  // bus source select , ALU operaation select
     alu_ctrl<= 4'b1010; //substitude
     mdr_ctrl<= 3'b000;
     mar_ctrl<= 2'b00;
     ac_ctrl<= 2'b00;
     reg_b_ctrl<= 7'b0000000;
     bus_ctrl<= 3'b010; //src(register bank)
     pc_ctrl<= 3'b000;
     dram_ctrl<= 2'b00;
     iram_ctrl<= 2'b00;
     state <= `COPYRAC3;
 end
`COPYRAC3 :  begin  // AC write operation
     alu_ctrl<= 4'b0000;
     mdr_ctrl<= 3'b000;
     mar_ctrl<= 2'b00;
     ac_ctrl<= 2'b01; //write
     reg_b_ctrl<= 7'b0000000;
     bus_ctrl<= 3'b000;
     pc_ctrl<= 3'b000;
     dram_ctrl<= 2'b00;
     iram_ctrl<= 2'b00;
     state <= `FETCH1;
 end
```



```
/*****************R1 -> R2 *************/
 `COPYRR1:  begin  // select register output of the bank
      alu_ctrl<= 4'b0000;
      mdr_ctrl<= 3'b000;
      mar_ctrl<= 2'b00;
      ac_ctrl<= 2'b00;
      reg_b_ctrl<= {3'b100, operand1[3:0]}; //select register
      bus_ctrl<= 3'b000;
      pc_ctrl<= 3'b000;
      dram_ctrl<= 2'b00;
      iram_ctrl<= 2'b00;
      state <= `COPYRR2;
  end

  `COPYRR2 :  begin  // bus source select
      alu_ctrl<= 4'b000; //substitude
      mdr_ctrl<= 3'b000;
      mar_ctrl<= 2'b00;
      ac_ctrl<= 2'b00;
      reg_b_ctrl<= 7'b0000000;
      bus_ctrl<= 3'b010; //src(register bank)
      pc_ctrl<= 3'b000;
      dram_ctrl<= 2'b00;
      iram_ctrl<= 2'b00;
      state <= `COPYRR3;
  end
   `COPYRR3:  begin  // select register output of the bank
      alu_ctrl<= 4'b0000;
      mdr_ctrl<= 3'b000;
      mar_ctrl<= 2'b00;
      ac_ctrl<= 2'b00;
      reg_b_ctrl<= {3'b001, operand2[3:0]}; //select register
      bus_ctrl<= 3'b000;
      pc_ctrl<= 3'b000;
      dram_ctrl<= 2'b00;
      iram_ctrl<= 2'b00;
      state <= `FETCH1;
   end
/*****************AC -> MAR *************/
    `COPYACMAR1 :  begin  // bus source select
      alu_ctrl<= 4'b0000;
      mdr_ctrl<= 3'b000;
      mar_ctrl<= 2'b00;
      ac_ctrl<= 2'b00;
      reg_b_ctrl<= 7'b0000000;
      bus_ctrl<= 3'b011; //src(AC)
      pc_ctrl<= 3'b000;
      dram_ctrl<= 2'b00;
      iram_ctrl<= 2'b00;
      state <= `COPYACMAR2;
   end
  `COPYACMAR2 :  begin  // MAR write operation
      alu_ctrl<= 4'b0000;
      mdr_ctrl<= 3'b000;
      mar_ctrl<= 2'b01;//write
      ac_ctrl<= 2'b00;
      reg_b_ctrl<= 7'b0000000;
```

```
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1;
    end

  /*****************R -> MAR *************/
  `COPYRMAR1:  begin  // select register output of the bank
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b100, operand1[3:0]}; //select register
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `COPYRMAR2;
    end

    `COPYRMAR2 :  begin  // bus source select
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b010; //src(register bank)
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `COPYACMAR2;
    end
     `COPYRR3:  begin  // select register output of the bank
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b001, operand2[3:0]}; //select register
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1;
    end
   /*****************AC -> MAR *************/
      `COPYACMAR1 :  begin  // bus source select
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b011; //src(AC)
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `COPYACMAR2;
    end
    `COPYACMAR2 :  begin  // MAR write operation
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b01;//write
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1;
    end

  /*****************R -> MAR *************/
  `COPYRMAR1:  begin  // select register output of the bank
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b100, operand1[3:0]}; //select register
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `COPYRMAR2;
    end

    `COPYRMAR2 :  begin  // bus source select
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b010; //src(register bank)
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `COPYACMAR2;
    end
  ADD[n] - (AC + n -> AC)
  SUB[n] - (AC - n -> AC)
  LSH[n] - (left shift , AC>>n)
  RSH[n] - (right shift , AC<<n)

  ADD[R]  - (AC + R -> AC )
  SUB[R] - (AC - R -> AC)
  MUL[R] - (AC * R -> AC)

    */
          `OP1  :  begin  //select source R or n
          alu_ctrl<= 4'b0000;
          mdr_ctrl<= 3'b000;
          mar_ctrl<= 3'b000;
          ac_ctrl<= 2'b00;
          reg_b_ctrl<= 7'b0000000;
          bus_ctrl<= (opcode== `ADDN1 ||
                      opcode== `SUBN1 ||
                      opcode== `LSHN1 ||
                      opcode== `RSHN1)?3'b100: 3'b010 ; //src(Instrcution - Register bank)
          pc_ctrl<= 3'b000;
          dram_ctrl<= 2'b00;
          iram_ctrl<= 2'b00;
          state <= `OP2 ;
          end
```

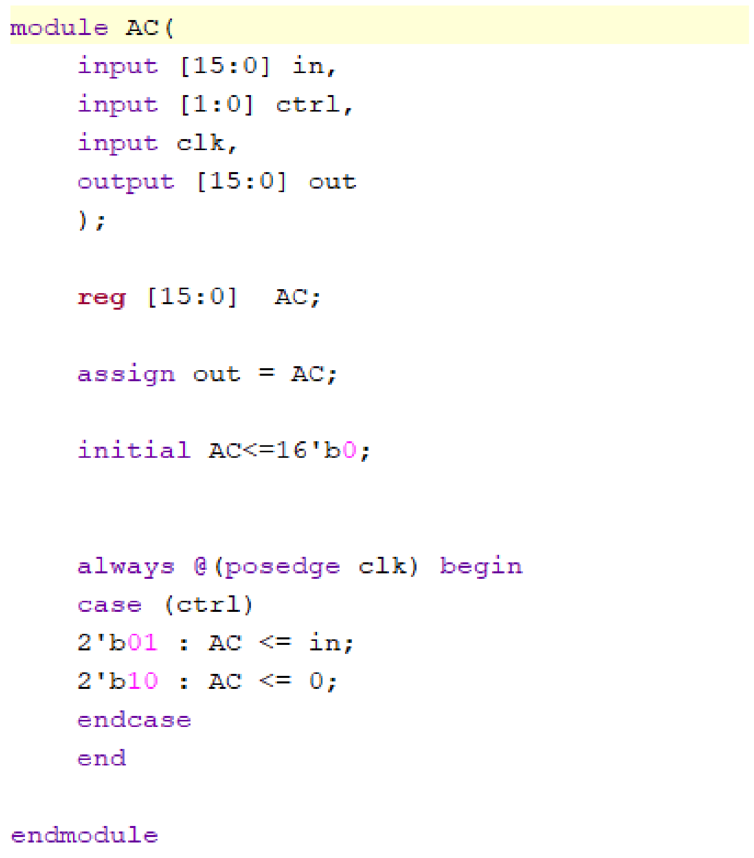
```
         `OP2 :  begin  //decide ALU operation
         case (opcode)
             `ADDN1 :  alu_ctrl<= 4'b0100;//add
             `ADDR1 :  alu_ctrl<= 4'b0001;//add
             `SUBN1 :  alu_ctrl<= 4'b0101;//sub
             `SUBR1 :  alu_ctrl<= 4'b0010;//sub
             `LSHN1 :  alu_ctrl<= 4'b1000;//left shift
             `RSHN1 :  alu_ctrl<= 4'b1001;//right shift
             `MULR1 :  alu_ctrl<= 4'b0011;//mul
         endcase
         mdr_ctrl<= 3'b000;
         mar_ctrl<= 3'b000;
         ac_ctrl<= 2'b00;
         reg_b_ctrl<= 7'b0000000;
         bus_ctrl<= 3'b000;
         pc_ctrl<= 3'b000;
         dram_ctrl<= 2'b00;
         iram_ctrl<= 2'b00;
         state <= `OP3 ;
         end

         `OP3 :  begin  //write in AC
         alu_ctrl<= 4'b0000;
         mdr_ctrl<= 3'b000;
         mar_ctrl<= 3'b000;
         ac_ctrl<= 2'b01; //write
         reg_b_ctrl<= 7'b0000000;
         bus_ctrl<= 3'b000;
         pc_ctrl<= 3'b000;
         dram_ctrl<= 2'b00;
        `OPR  :  begin  // select register output of the bank
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b100, operand2[3:0]}; //select register
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `OP1;
   end

/*
JUMP[I] - (jump I)
JUMPNZ[I] - (Zflag -0 , ->Jump I)

*/

        `JUMP1   :  begin  // select register from bnk
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b100, operand2[3:0]}; //select register
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `JUMP2 ;
        end
        `JUMP2 :  begin //select bus source as register bank
         alu_ctrl<= 4'b0000;
         mdr_ctrl<= 3'b000;
         mar_ctrl<= 3'b000;
         ac_ctrl<= 2'b00;
         reg_b_ctrl<= 7'b0000000;
         bus_ctrl<= 3'b010; //src(R)
         pc_ctrl<= 3'b000;
         dram_ctrl<= 2'b00;
         iram_ctrl<= 2'b00;
         state <= `JUMP3 ;
         end

        `JUMP3 :  begin //control p
         alu_ctrl<= 4'b0000;
         mdr_ctrl<= 3'b000;
         mar_ctrl<= 3'b000;
         ac_ctrl<= 2'b00;
         reg_b_ctrl<= 7'b0000000;
         bus_ctrl<= 3'b000;
         pc_ctrl<= (opcode == `JUMP1)? 3'b011 : 3'b010; //jump select
         dram_ctrl<= 2'b00;
         iram_ctrl<= 2'b00;
         state <= `FETCH1 ;
         end
        `INCMAR1 :   begin  //increment MAR by 1
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b10; //inc(1)
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
        end

/*
INC[R] - (R + 1 -> R)
  */

       `INCR1 :   begin
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 3'b000;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b011 , operand2[3:0]};
        bus_ctrl<= 6'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
        end
```

```
INCAC - (R + 1 -> R)
  */

        `INCAC1 :   begin
        alu_ctrl<= 4'b0110;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 3'b000;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `INCAC2 ;
        end
        `INCAC2 :   begin
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 3'b000;
        ac_ctrl<= 2'b01;
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 3'b000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
        end

/*
RESET[R/AC/MAR]  - (R/AC/MAR ->0)
*/

        `RESETR1 :   begin  //clear register bank
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b00;
        reg_b_ctrl<= {3'b010, operand2[3:0]};
        bus_ctrl<= 6'b000000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
        end
        `RESETAC1 :   begin  //clear AC
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b00;
        ac_ctrl<= 2'b10; //clear
        reg_b_ctrl<= 7'b0000000;
        bus_ctrl<= 6'b000000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
        end
        `RESETMAR1 :   begin //clear MAR
        alu_ctrl<= 4'b0000;
        mdr_ctrl<= 3'b000;
        mar_ctrl<= 2'b11; //clear
        ac_ctrl<= 2'b00;
        reg_b_ctrl<=  7'b0000000;
        bus_ctrl<= 6'b000000;
        pc_ctrl<= 3'b000;
        dram_ctrl<= 2'b00;
        iram_ctrl<= 2'b00;
        state <= `FETCH1 ;
        end
   endcase

   end

endmodule
```

ALU

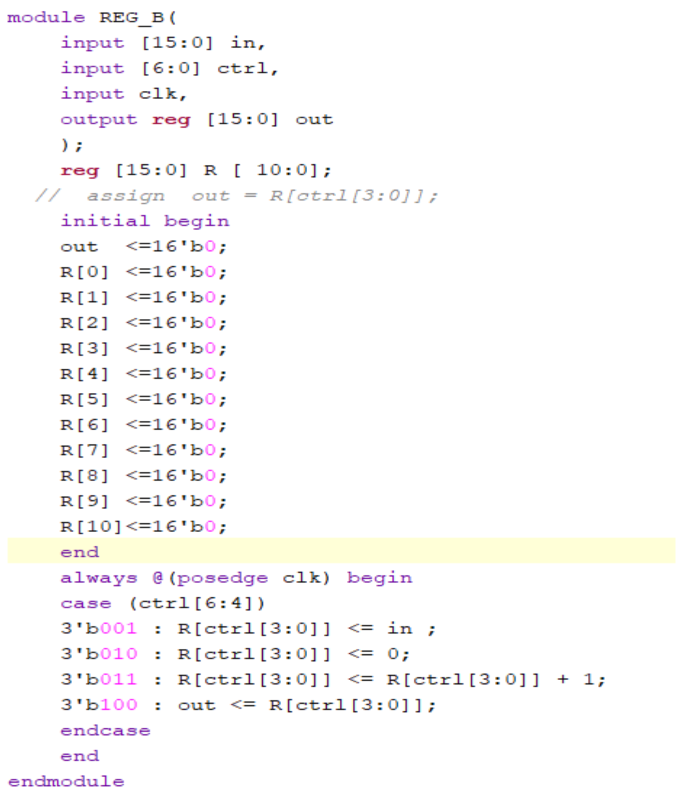
```
module ALU(
    input [15:0] A,
    input [15:0] B,
    input [3:0] ctrl,
    input clk,
    output reg [15:0] ALU_out,
    output reg Z_inv
    );
    reg [4:0] n;
    initial begin
    //assign n = B[4:0];
    ALU_out = 16'b0;
    assign Z_inv = (ALU_out>0);
   // Z_inv = 1;
    end
    always @(posedge clk) begin
    case (ctrl)
    4'b0001 :  ALU_out <= A + B; //addition
    4'b0010 :  ALU_out <= A - B; //subustraction
    4'b0011 :  ALU_out <= A*B; //multiplication
    4'b0100 :  ALU_out <= A + B[4:0]; //add n
    4'b0101 :  ALU_out <= A - B[4:0]; //sub n
    4'b0110 :  ALU_out <= A + 1; //inc
    4'b0111 :  ALU_out <= A - 1; //dec
    4'b1000 :  ALU_out <= A << B[4:0]; //left shift n
    4'b1001 :  ALU_out <= A >> B[4:0]; //left shift n
    4'b1010 :  ALU_out <= B; //substitude
    4'b1011 :  ALU_out <= 0;//reset
    endcase
    end
endmodule
```

ACCUMULATOR



```
module AC(
    input [15:0] in,
    input [1:0] ctrl,
    input clk,
    output [15:0] out
    );

    reg [15:0]  AC;

    assign out = AC;

    initial AC<=16'b0;


    always @(posedge clk) begin
    case (ctrl)
    2'b01 : AC <= in;
    2'b10 : AC <= 0;
    endcase
    end

endmodule
```

REG_B

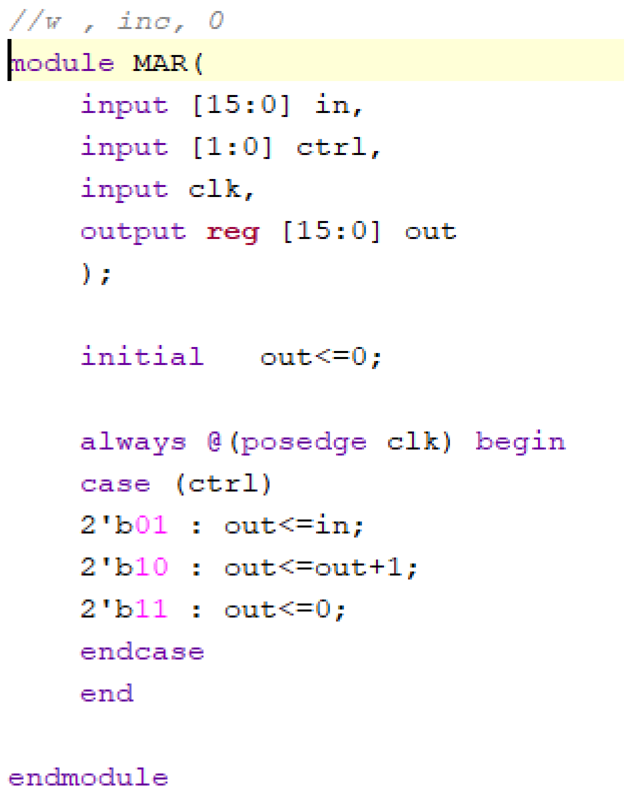


```
module REG_B(
    input [15:0] in,
    input [6:0] ctrl,
    input clk,
    output reg [15:0] out
    );
    reg [15:0] R [ 10:0];
  //  assign  out = R[ctrl[3:0]];
    initial begin
    out  <=16'b0;
    R[0] <=16'b0;
    R[1] <=16'b0;
    R[2] <=16'b0;
    R[3] <=16'b0;
    R[4] <=16'b0;
    R[5] <=16'b0;
    R[6] <=16'b0;
    R[7] <=16'b0;
    R[8] <=16'b0;
    R[9] <=16'b0;
    R[10]<=16'b0;
    end
    always @(posedge clk) begin
    case (ctrl[6:4])
    3'b001 : R[ctrl[3:0]] <= in ;
    3'b010 : R[ctrl[3:0]] <= 0;
    3'b011 : R[ctrl[3:0]] <= R[ctrl[3:0]] + 1;
    3'b100 : out <= R[ctrl[3:0]];
    endcase
    end
endmodule
```

MDR

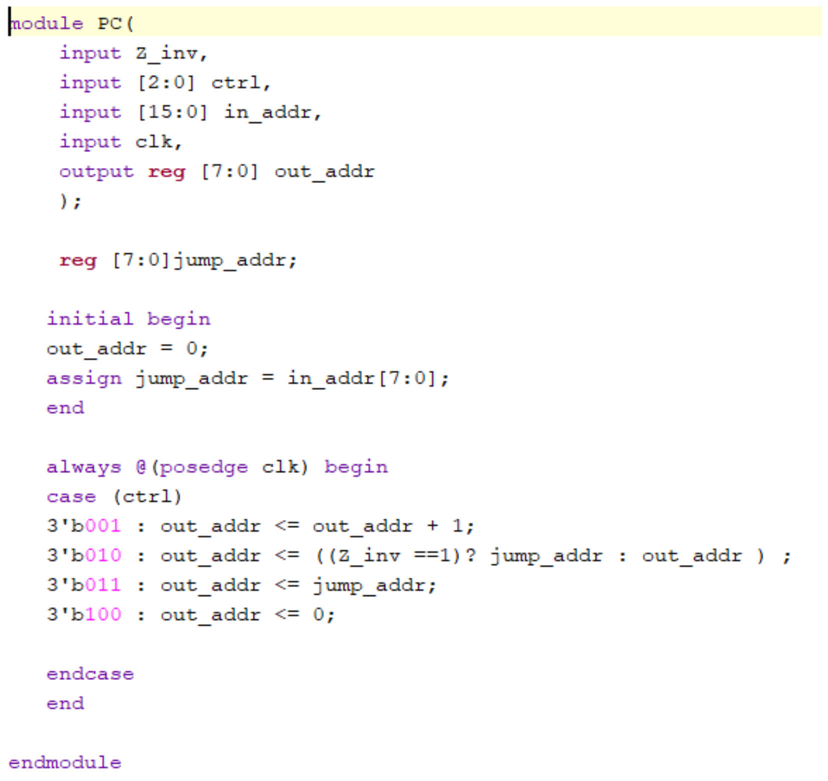
```
//1 - BUS_MDR, 2 - DRAM_MDR , 3 - reset
module MDR(
    input [7:0] DRAM_MDR,
    input [7:0] BUS_MDR,
    input [2:0] ctrl,
    input clk,
    output [7:0] MDR_DRAM,
    output [7:0] MDR_BUS
    );

    reg [7:0] MDR;
    assign  MDR_DRAM = MDR ;
    assign  MDR_BUS= MDR ;

    initial MDR <= 8'b0;

    always @(posedge clk) begin
    case (ctrl)
    3'b001 : MDR <=  BUS_MDR;
    3'b010 : MDR <=  DRAM_MDR;
    3'b011 : MDR <=  8'b0;
    endcase
    end

endmodule
```

MAR



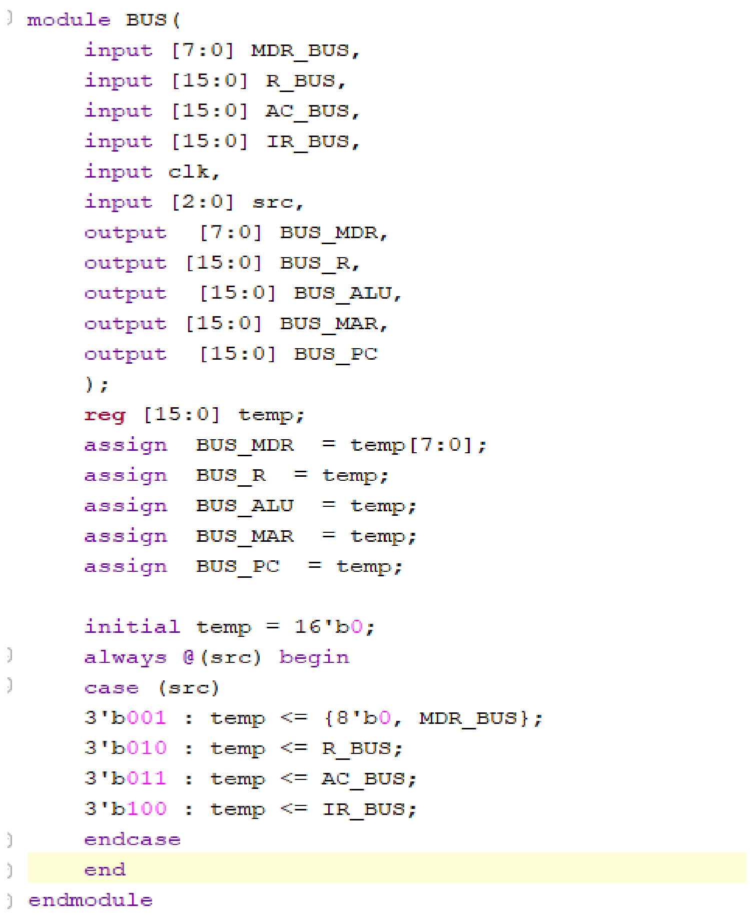
```
//w , inc, 0
module MAR(
    input [15:0] in,
    input [1:0] ctrl,
    input clk,
    output reg [15:0] out
    );

    initial   out<=0;

    always @(posedge clk) begin
    case (ctrl)
    2'b01 : out<=in;
    2'b10 : out<=out+1;
    2'b11 : out<=0;
    endcase
    end

endmodule
```

PC



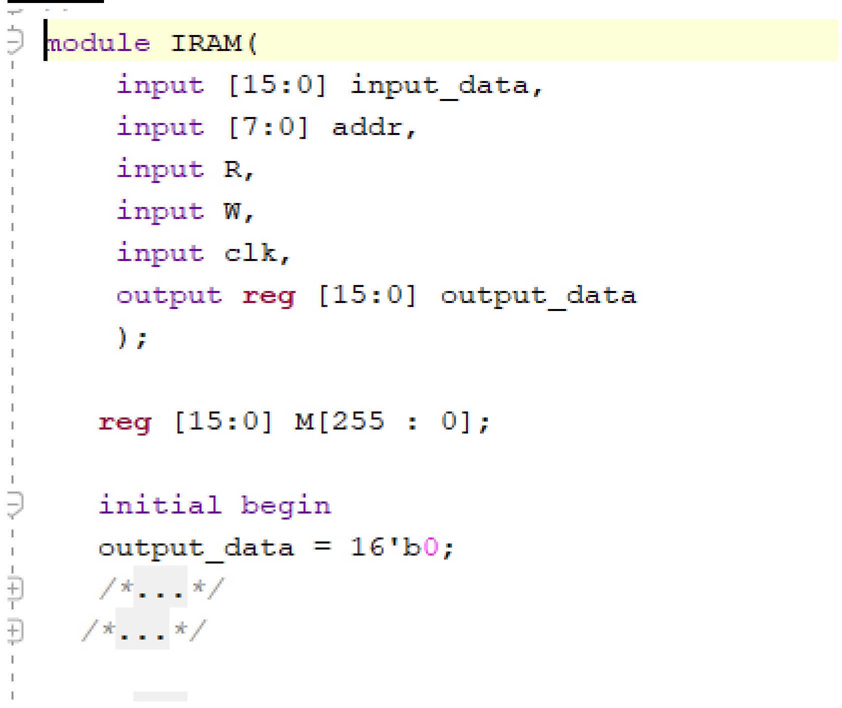
```
module PC(
    input Z_inv,
    input [2:0] ctrl,
    input [15:0] in_addr,
    input clk,
    output reg [7:0] out_addr
    );

    reg [7:0]jump_addr;

   initial begin
   out_addr = 0;
   assign jump_addr = in_addr[7:0];
   end

   always @(posedge clk) begin
   case (ctrl)
   3'b001 : out_addr <= out_addr + 1;
   3'b010 : out_addr <= ((Z_inv ==1)? jump_addr : out_addr ) ;
   3'b011 : out_addr <= jump_addr;
   3'b100 : out_addr <= 0;

   endcase
   end

endmodule
```

BUS



```
module BUS(
    input [7:0] MDR_BUS,
    input [15:0] R_BUS,
    input [15:0] AC_BUS,
    input [15:0] IR_BUS,
    input clk,
    input [2:0] src,
    output  [7:0] BUS_MDR,
    output [15:0] BUS_R,
    output  [15:0] BUS_ALU,
    output [15:0] BUS_MAR,
    output  [15:0] BUS_PC
    );
    reg [15:0] temp;
    assign  BUS_MDR  = temp[7:0];
    assign  BUS_R  = temp;
    assign  BUS_ALU  = temp;
    assign  BUS_MAR  = temp;
    assign  BUS_PC  = temp;

    initial temp = 16'b0;
    always @(src) begin
    case (src)
    3'b001 : temp <= {8'b0, MDR_BUS};
    3'b010 : temp <= R_BUS;
    3'b011 : temp <= AC_BUS;
    3'b100 : temp <= IR_BUS;
    endcase
    end
endmodule
```

IRAM



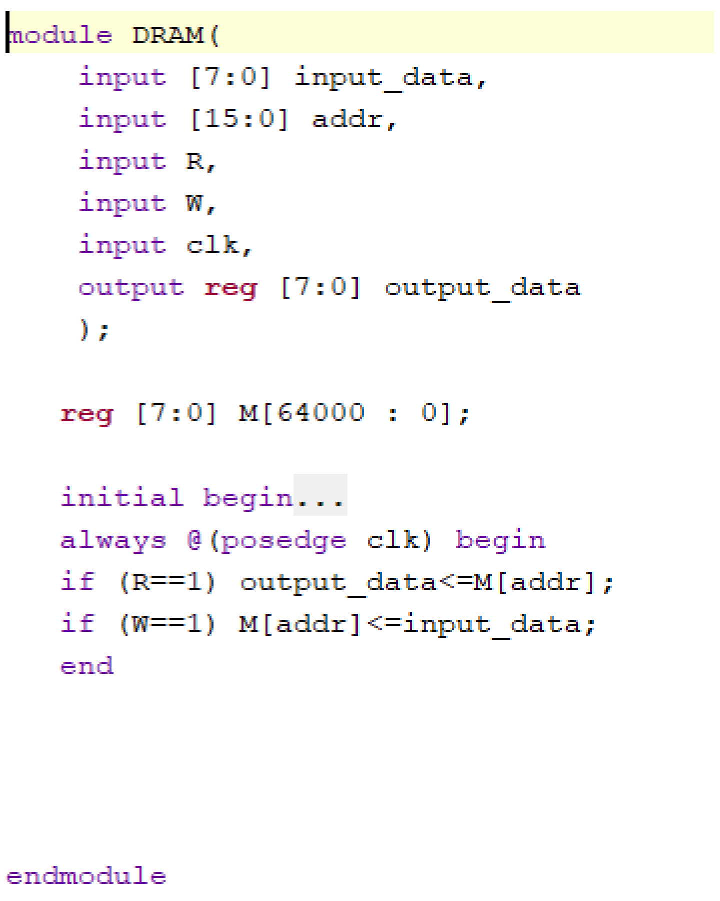
```
module IRAM(
    input [15:0] input_data,
    input [7:0] addr,
    input R,
    input W,
    input clk,
    output reg [15:0] output_data
    );

   reg [15:0] M[255 : 0];

   initial begin
   output_data = 16'b0;
   /*...*/
  /*...*/
```

```
M[168] = 16'b0011000000000101;
M[169] = 16'b0111100000000000;
M[170] = 16'b0000000000000000;
```

```
    end

    always @(posedge clk) begin
    if (R==1) output_data<=M[addr];
    if (W==1) M[addr]<=input_data;
    end



endmodule
```

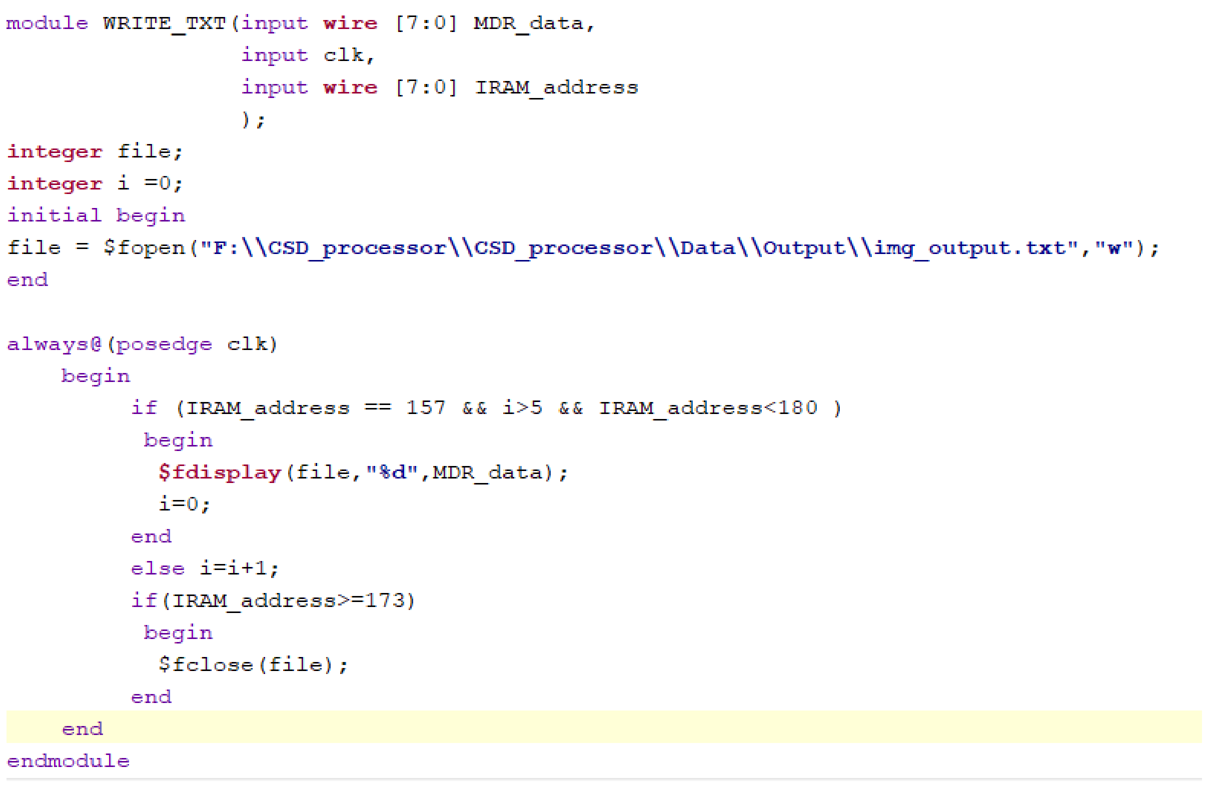
DRAM

```
module DRAM(
    input [7:0] input_data,
    input [15:0] addr,
    input R,
    input W,
    input clk,
    output reg [7:0] output_data
    );

   reg [7:0] M[64000 : 0];

   initial begin
   M[0] <= 250;//WIDTH
   M[1] <= 250; //HEIGHT
   M[2] <= 20; //offset;
   M[3] <= 14; //A
   M[4] <= 30; //B
   M[5] <= 76; //C
   M[6] <= 95; //D
   M[7] <= 147; //E
   M[8] <= 147; //F

M[20] = 162;
```



```
module DRAM(
     input [7:0] input_data,
     input [15:0] addr,
     input R,
     input W,
     input clk,
     output reg [7:0] output_data
     );

    reg [7:0] M[64000 : 0];

    initial begin...
    always @(posedge clk) begin
    if (R==1) output_data<=M[addr];
    if (W==1) M[addr]<=input_data;
    end




endmodule
```

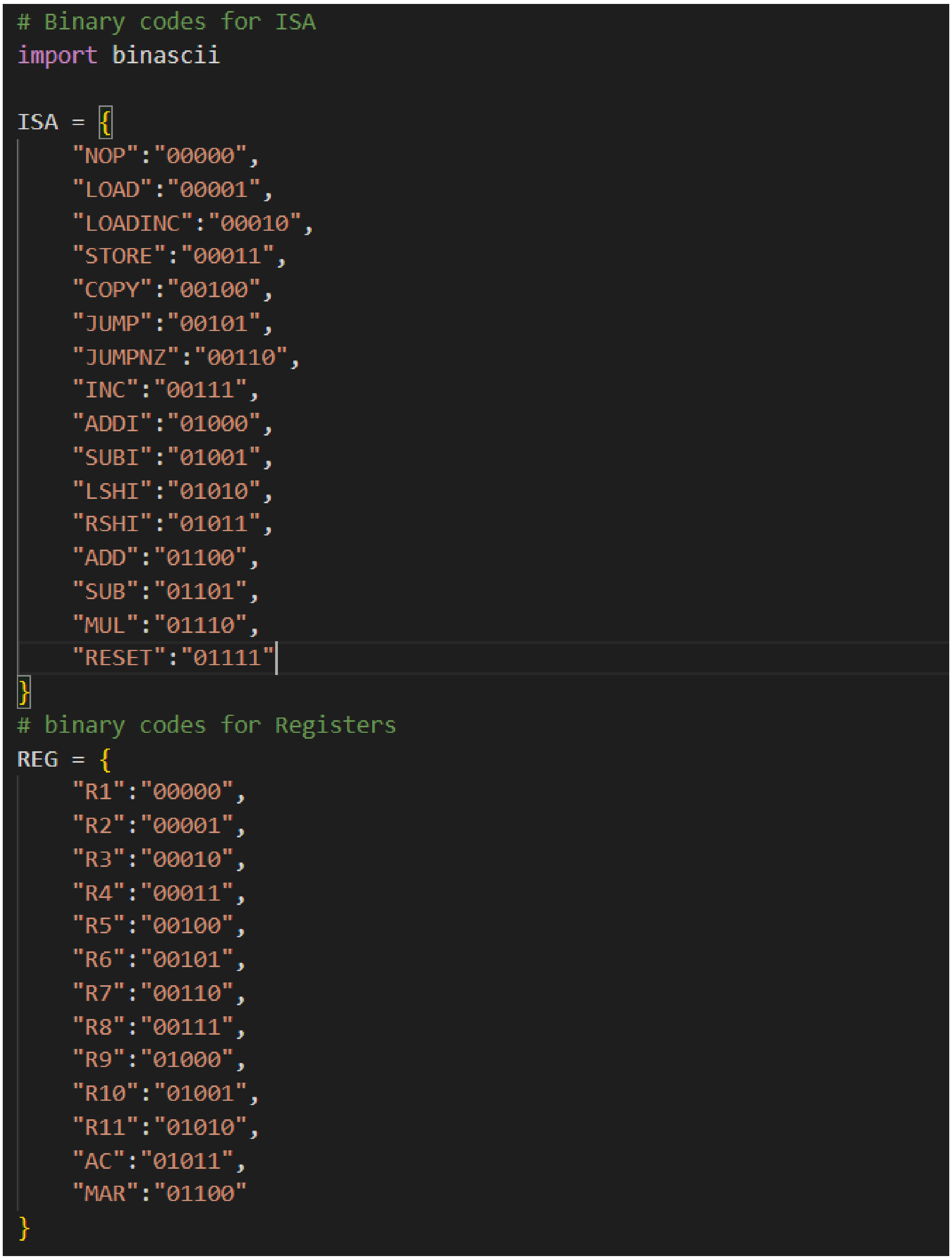
Write_txt



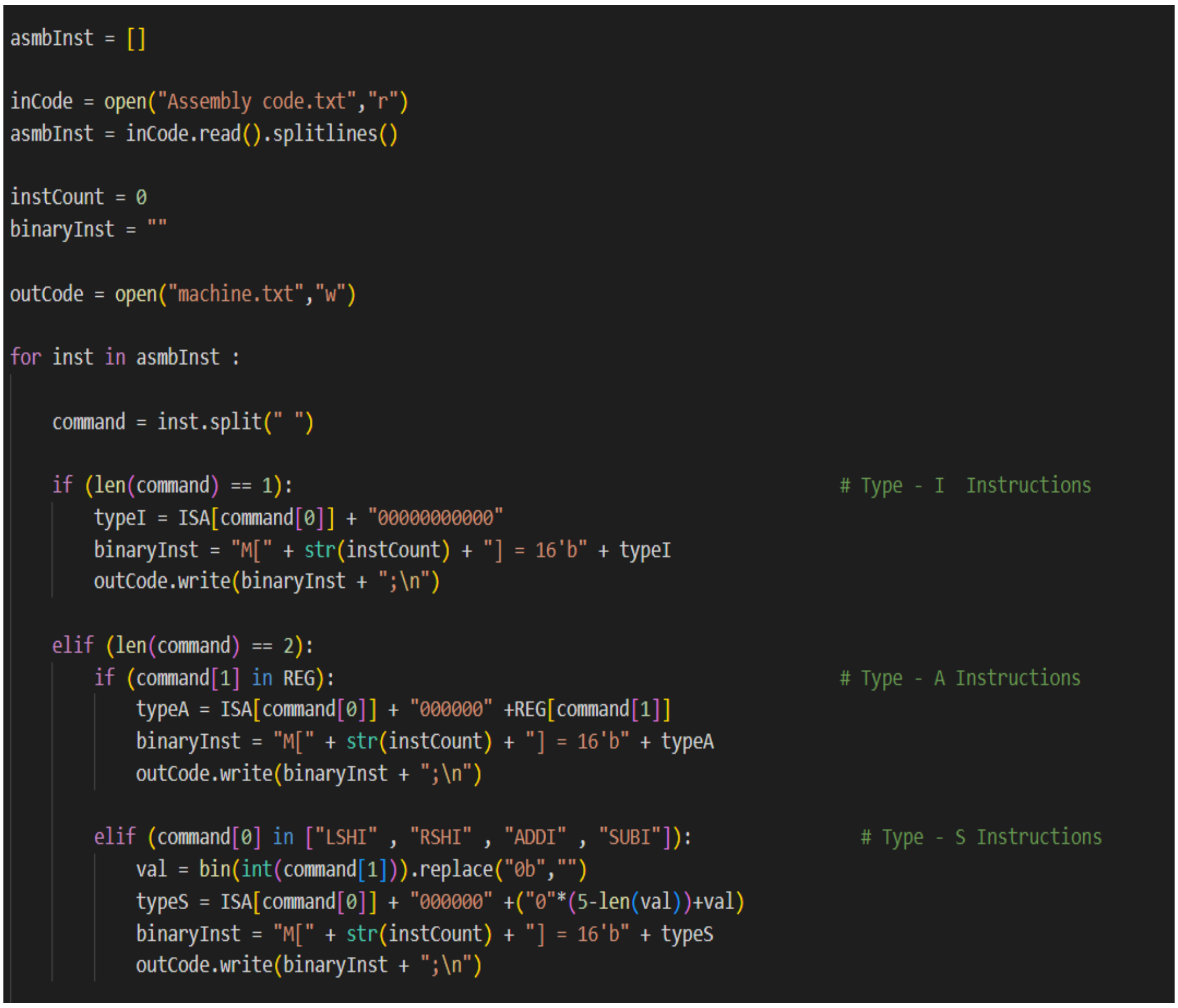
```
module WRITE_TXT(input wire [7:0] MDR_data,
                 input clk,
                 input wire [7:0] IRAM_address
                 );
integer file;
integer i =0;
initial begin
file = $fopen("F:\\CSD_processor\\CSD_processor\\Data\\Output\\img_output.txt","w");
end

always@(posedge clk)
    begin
        if (IRAM_address == 157 && i>5 && IRAM_address<180 )
         begin
          $fdisplay(file,"%d",MDR_data);
          i=0;
        end
        else i=i+1;
        if(IRAM_address>=173)
         begin
          $fclose(file);
        end
    end
endmodule
```

Python Codes

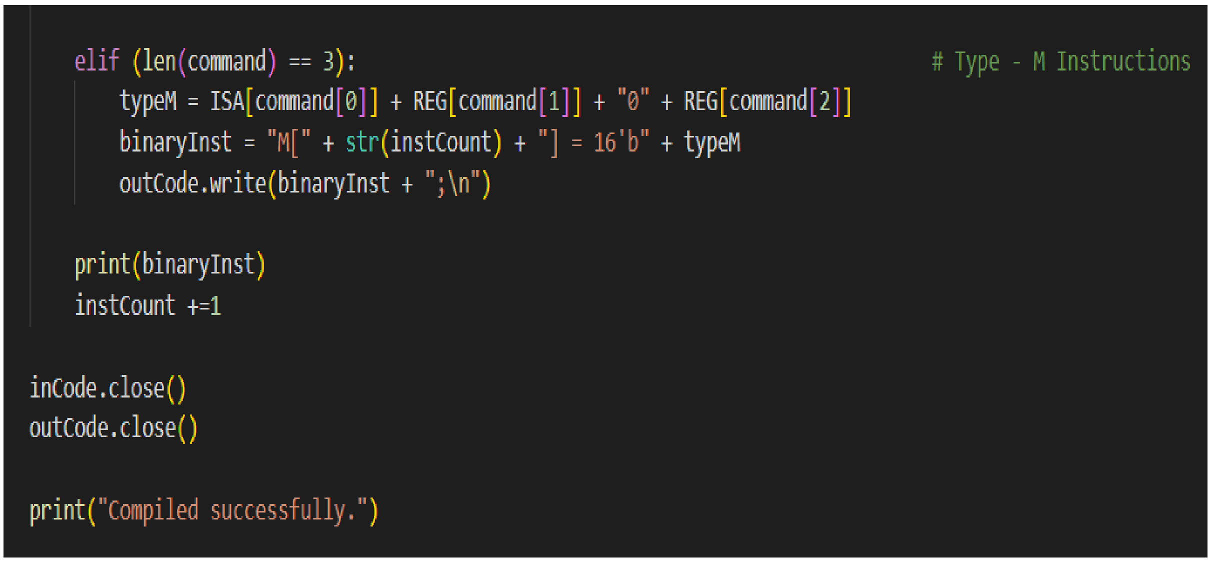
Assembly code convert to machine code



```python
# Binary codes for ISA
import binascii

ISA = {
        "NOP":"00000",
        "LOAD":"00001",
        "LOADINC":"00010",
        "STORE":"00011",
        "COPY":"00100",
        "JUMP":"00101",
        "JUMPNZ":"00110",
        "INC":"00111",
        "ADDI":"01000",
        "SUBI":"01001",
        "LSHI":"01010",
        "RSHI":"01011",
        "ADD":"01100",
        "SUB":"01101",
        "MUL":"01110",
        "RESET":"01111"
}
# binary codes for Registers
REG = {
        "R1":"00000",
        "R2":"00001",
        "R3":"00010",
        "R4":"00011",
        "R5":"00100",
        "R6":"00101",
        "R7":"00110",
        "R8":"00111",
        "R9":"01000",
        "R10":"01001",
        "R11":"01010",
        "AC":"01011",
        "MAR":"01100"
}
```

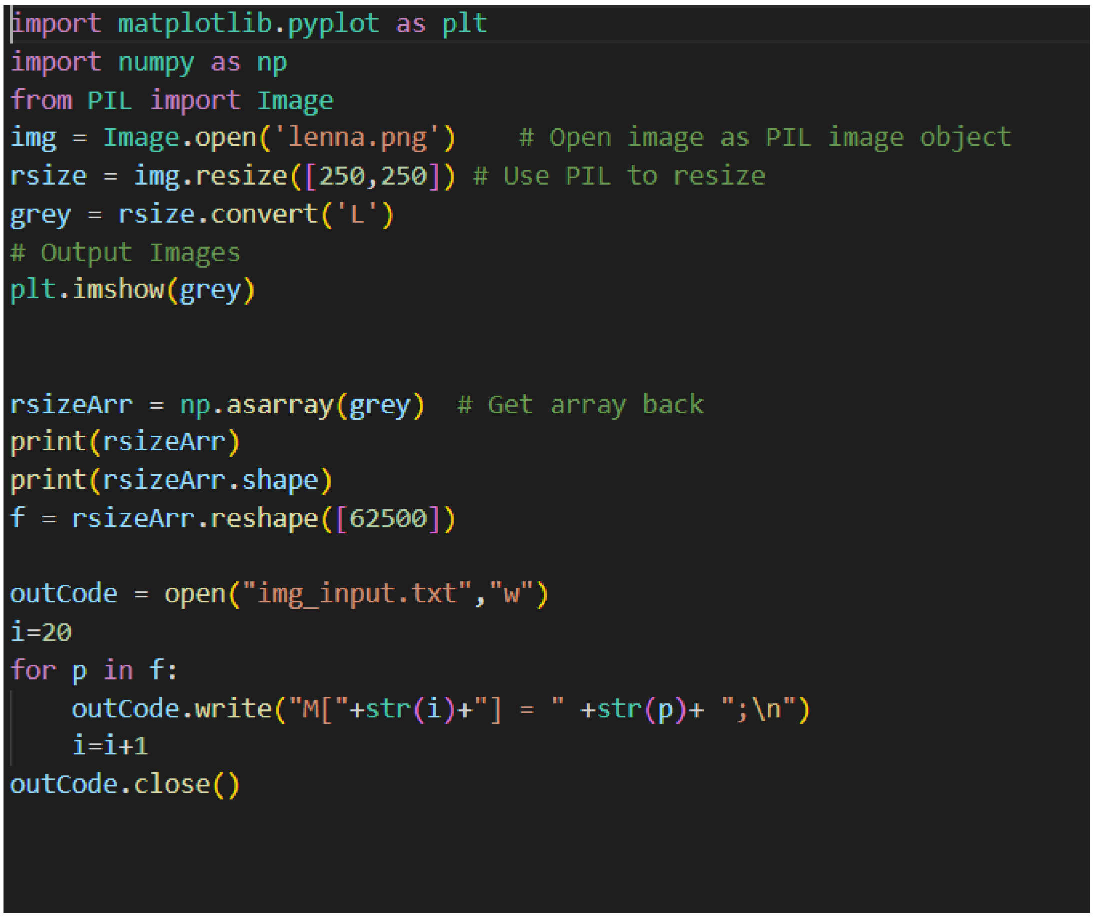


```
asmbInst = []

inCode = open("Assembly code.txt","r")
asmbInst = inCode.read().splitlines()

instCount = 0
binaryInst = ""

outCode = open("machine.txt","w")

for inst in asmbInst :

    command = inst.split(" ")

    if (len(command) == 1):                                   # Type - I  Instructions
        typeI = ISA[command[0]] + "00000000000"
        binaryInst = "M[" + str(instCount) + "] = 16'b" + typeI
        outCode.write(binaryInst + ";\n")

    elif (len(command) == 2):
        if (command[1] in REG):                               # Type - A Instructions
            typeA = ISA[command[0]] + "000000" +REG[command[1]]
            binaryInst = "M[" + str(instCount) + "] = 16'b" + typeA
            outCode.write(binaryInst + ";\n")

        elif (command[0] in ["LSHI" , "RSHI" , "ADDI" , "SUBI"]):   # Type - S Instructions
            val = bin(int(command[1])).replace("0b","")
            typeS = ISA[command[0]] + "000000" +("0"*(5-len(val))+val)
            binaryInst = "M[" + str(instCount) + "] = 16'b" + typeS
            outCode.write(binaryInst + ";\n")
```



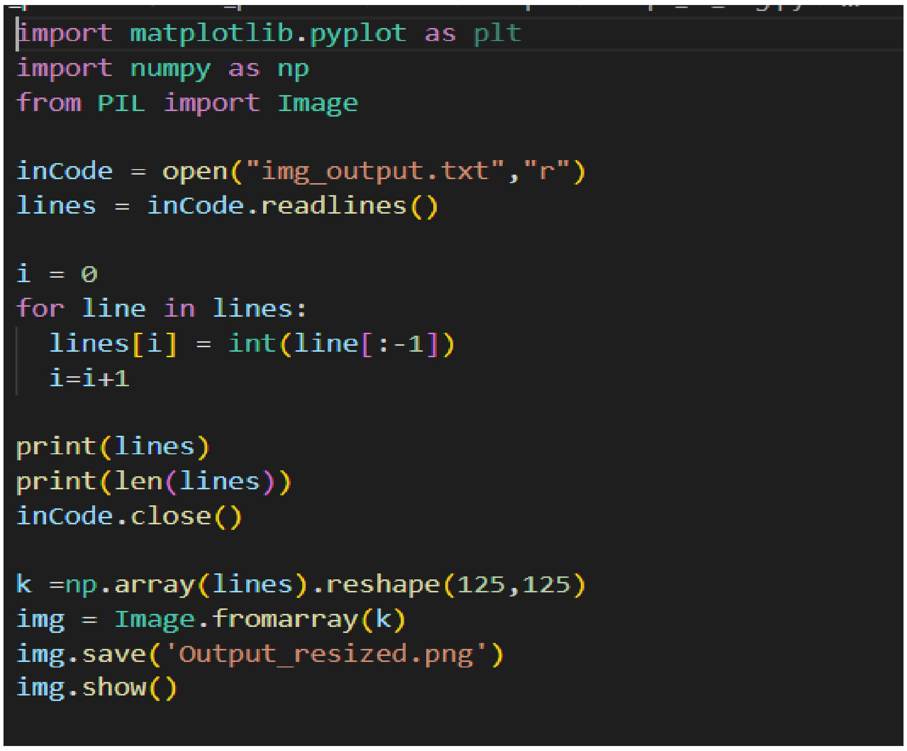
```
    elif (len(command) == 3):                                 # Type - M Instructions
        typeM = ISA[command[0]] + REG[command[1]] + "0" + REG[command[2]]
        binaryInst = "M[" + str(instCount) + "] = 16'b" + typeM
        outCode.write(binaryInst + ";\n")

    print(binaryInst)
    instCount +=1

inCode.close()
outCode.close()

print("Compiled successfully.")
```

Image to pixels



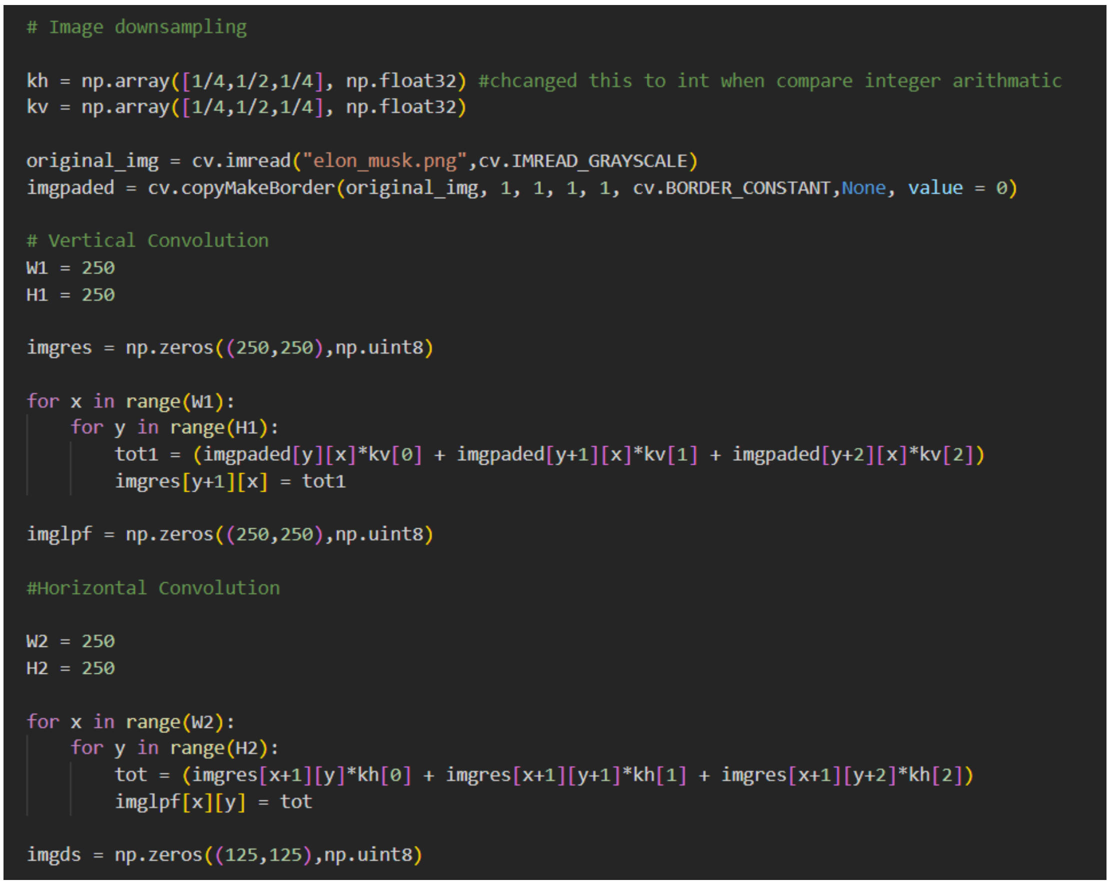
```python
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
img = Image.open('lenna.png')     # Open image as PIL image object
rsize = img.resize([250,250]) # Use PIL to resize
grey = rsize.convert('L')
# Output Images
plt.imshow(grey)


rsizeArr = np.asarray(grey)  # Get array back
print(rsizeArr)
print(rsizeArr.shape)
f = rsizeArr.reshape([62500])

outCode = open("img_input.txt","w")
i=20
for p in f:
    outCode.write("M["+str(i)+"] = " +str(p)+ ";\n")
    i=i+1
outCode.close()
```

Pixels to image

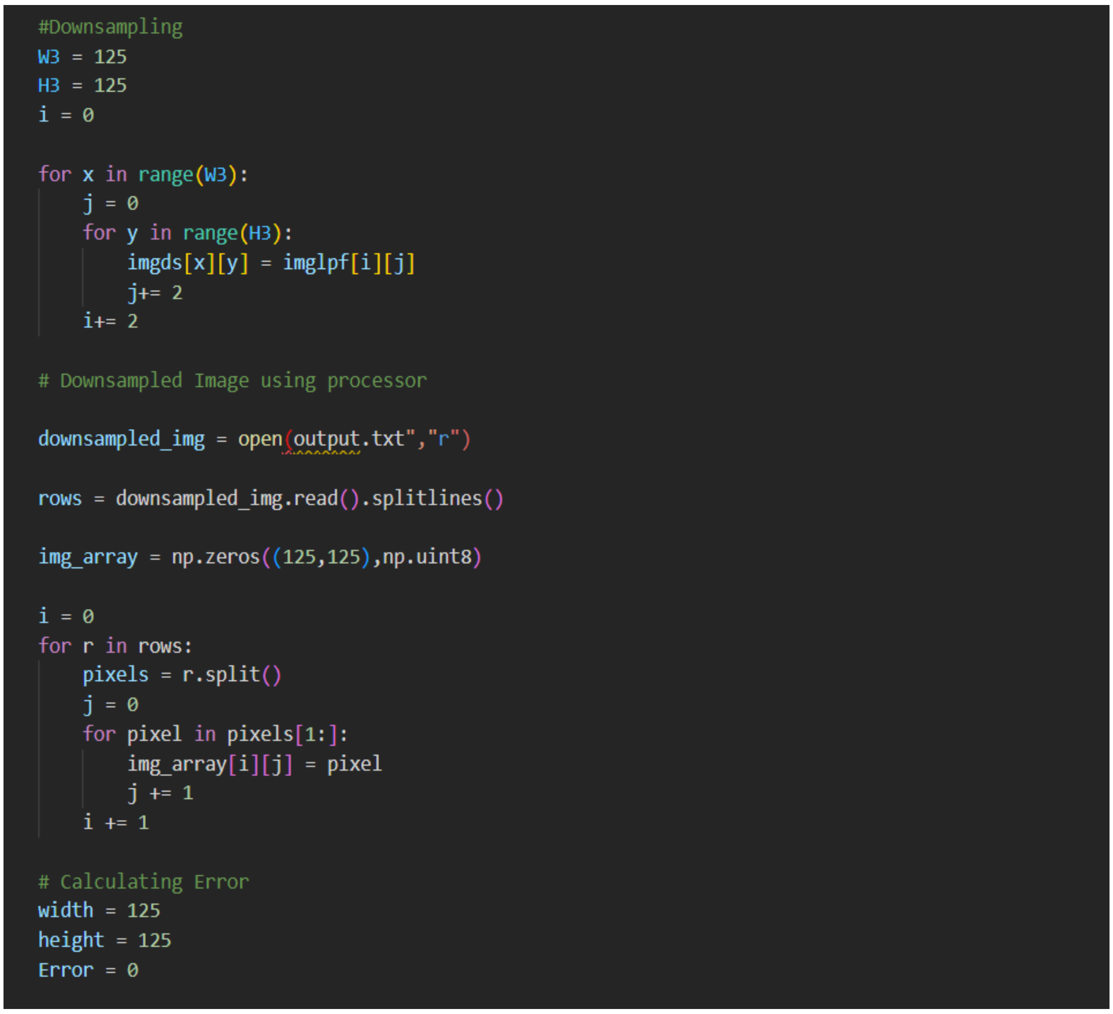


```python
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

inCode = open("img_output.txt","r")
lines = inCode.readlines()

i = 0
for line in lines:
  lines[i] = int(line[:-1])
  i=i+1

print(lines)
print(len(lines))
inCode.close()

k =np.array(lines).reshape(125,125)
img = Image.fromarray(k)
img.save('Output_resized.png')
img.show()
```

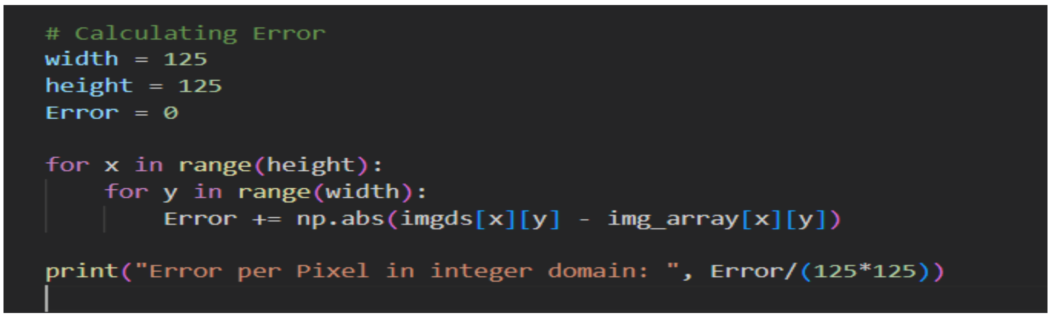
Error Analysis



```python
# Image downsampling

kh = np.array([1/4,1/2,1/4], np.float32) #chcanged this to int when compare integer arithmatic
kv = np.array([1/4,1/2,1/4], np.float32)

original_img = cv.imread("elon_musk.png",cv.IMREAD_GRAYSCALE)
imgpaded = cv.copyMakeBorder(original_img, 1, 1, 1, 1, cv.BORDER_CONSTANT,None, value = 0)

# Vertical Convolution
W1 = 250
H1 = 250

imgres = np.zeros((250,250),np.uint8)

for x in range(W1):
    for y in range(H1):
        tot1 = (imgpaded[y][x]*kv[0] + imgpaded[y+1][x]*kv[1] + imgpaded[y+2][x]*kv[2])
        imgres[y+1][x] = tot1

imglpf = np.zeros((250,250),np.uint8)

#Horizontal Convolution

W2 = 250
H2 = 250

for x in range(W2):
    for y in range(H2):
        tot = (imgres[x+1][y]*kh[0] + imgres[x+1][y+1]*kh[1] + imgres[x+1][y+2]*kh[2])
        imglpf[x][y] = tot

imgds = np.zeros((125,125),np.uint8)
```



```python
#Downsampling
W3 = 125
H3 = 125
i = 0

for x in range(W3):
    j = 0
    for y in range(H3):
        imgds[x][y] = imglpf[i][j]
        j+= 2
    i+= 2

# Downsampled Image using processor

downsampled_img = open(output.txt","r")

rows = downsampled_img.read().splitlines()

img_array = np.zeros((125,125),np.uint8)

i = 0
for r in rows:
    pixels = r.split()
    j = 0
    for pixel in pixels[1:]:
        img_array[i][j] = pixel
        j += 1
    i += 1

# Calculating Error
width = 125
height = 125
Error = 0
```



```
# Calculating Error
width = 125
height = 125
Error = 0

for x in range(height):
    for y in range(width):
        Error += np.abs(imgds[x][y] - img_array[x][y])

print("Error per Pixel in integer domain: ", Error/(125*125))
```