Department of Electronic and Telecommunication Engineering
University of Moratuwa

EN 3030

Circuits and Systems Design

FPGA Based Processor Design

Image Down-Sampling Processor

Group Members:

Jayasinghe H.A.D.N.M.	- 160236J
Kasthuriarachchi S.N.	- 160290R
Rajapaksha K.U.K.U.M.	- 160507R
Wijerathna I.G.C.J.	- 160703N

Supervisor:

Dr. Jayathu Samarawickrama

This is submitted as a partial fulfillment for the module EN 3030: Circuits and Systems Design

October 26, 2020

Abstract

Field Programmable Gate Array (FPGA) is an Integrated Circuit (IC) that can be programmed(customized) by the designer. This customization is usually done by a Hardware Description Language (HDL) like Verilog. In this project we have used an Altera Cyclone IV FPGA to implement a processor for downsampling a 256x256 image to a 128x128 image. This report contains the Instruction Set Architecture (ISA), Algorithms and other necessary resources that were used in the implementation of the above processor.

Contents

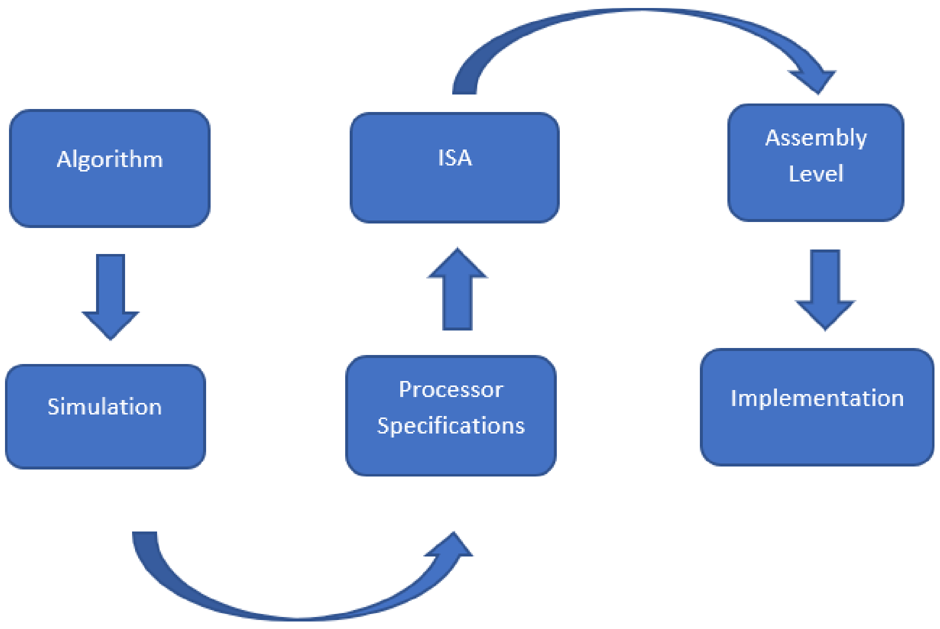
1 Introduction

1.1 Overview

Central Processing Unit (CPU) is the unit that executes the instruction of a program by performing Arithmetic and Logic (AL) operations and the basic control. The microprocessor which is however a multipurpose circuitry and different than CPU is driven by the stored data in its registries. Microprocessors consist of both sequential and combinational logic. In this project we have designed such a processor which can down-sample an image. High accuracy is the main concern in this processor design and implementation.

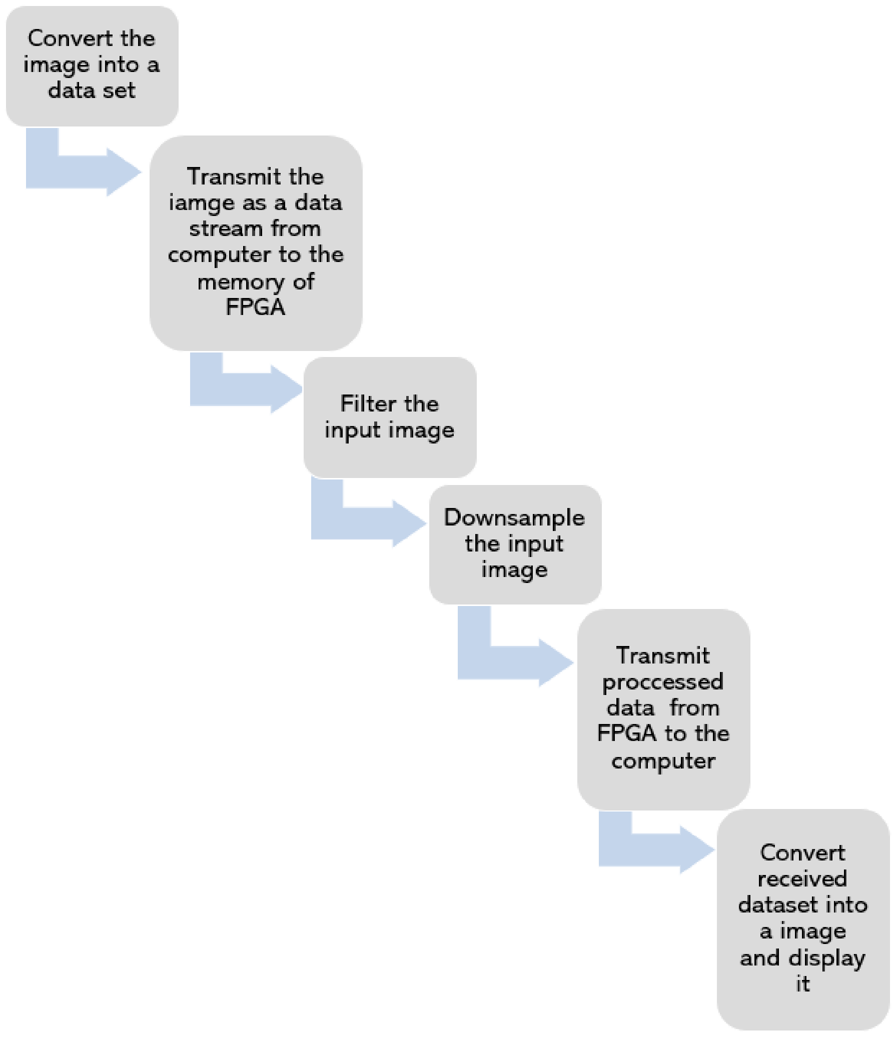
1.2 Problem Statement

The requirement is to implement a processor to down-sample a 256x256 image by a factor of 2. The main activities done during the process is sending data to the FPGA, down-sample the image and sending back the image to the computer to display the result.
In this project we have used Universal Asynchronous Receiver Transmitter (UART) protocol for sending image to the FPGA and getting back the down sampled image to the computer. We have used a MATLAB script to send and receive the images. Also, we have implemented a data RAM(DRAM) on the FPGA to store the image. Since the image may contain high frequency component, we used Gaussian filtering to overcome that issue.
Once the image has down sampled by a factor of 2, the RAM in the FPGA processor is replaced by those new pixel values of the image. For the comparison purposes we have also down sampled the same image using MATLAB.



2 Algorithm

We used main two algorithms based on two concepts in order to implement the task in the FPGA. First one is a filtering algorithms and second algorithm is a down sampling algorithm. Following diagram represent the entire task implementation with above mentioned two algorithms.


Convert the image into a data set
Transmit the iamge as a data stream from computer to the memory of FPGA
Filter the input image
Downsample the input image
Transmit proccessed data from FPGA to the computer
Convert received dataset into a image and display it

FILTERING ALGORITHM

High frequency components in the image cause aliasing effects. Filtering algorithm work as a low pass filter by reducing high frequency components of the picture. We use Gaussian low pass filter to implement filtering task. Used Gaussian function and the Gaussian kernel as follows.

$$g(r,c) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(r^2+c^2)}{2\sigma}}$$

0.0622	0.2489	0.0622
0.2489	1	0.2489
0.0622	0.2489	0.0622

We use a normalized kernel instead of above kernel since the complexity of working with decimal values when designing the processor. Normalization and some approximations was done so that the sum of the kernel be a power of two.

1	3	1
3	16	3
1	3	1

Then the original pixel values were replaced by new value which is obtained by overlapping the middle value of the kernel with a pixel value and getting normalized weighted summation value. Graphical illustration of applying Gaussian kernel on the image as follows.

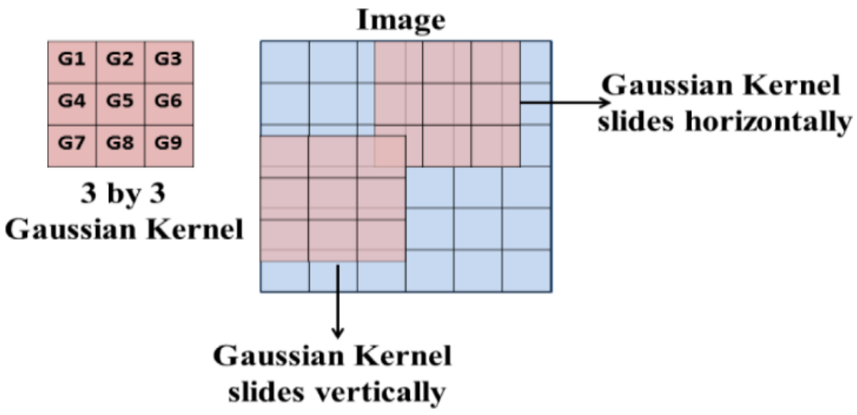


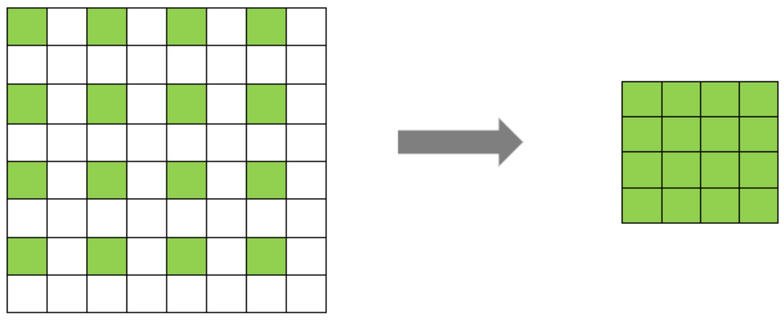
Image
G1 G2 G3
G4 G5 G6
G7 G8 G9
3 by 3 Gaussian Kernel
Gaussian Kernel slides horizontally
Gaussian Kernel slides vertically

The pseudocode for the filtering algorithm

```
for i=0; i<255 ; i++ do
    for (j=0; j<255 ; j++) do
        total = 0; total= total + img[ i + j ] x 1;
        total= total + img[ i + j + 1 ] x 3;
        total= total + img[ i + j + 2 ] x 1;
        total= total + img[ i + 256 + j ] x 1;
        total= total + img[ i + 256 + j + 1 ] x 16;
        total= total + img[ i + 256 + j + 2 ] x 1;
        total= total + img[ i + 512 + j ] x 1;
        total= total + img[ i + 512 + j + 1 ] x 3;
        total= total + img[ i + 512 + j + 2 ] x 1;
        img[ i + 256 + j + 1 ] = total / 31;
    end
end
```

DOWNSAMPLING ALGORITHM

A simple algorithm is used to downsample the given picture into its half of size. We design the processor to downsample 256×256 image into 128×128 image. Algorithm was selecting one pixel value from each four pixels. Following figure illustrate graphical representation of the downsampling algorithm.



The pseudocode for the down-sampling algorithm

```
for i=0; i<128 ; i++ do
    for (j=0; j<128 ; j++) do
        img[ x ] = img[ i x 512 + j x 2 ];
        x = x + 1;
    end
end
```

3 Architecture

Our general architecture is mainly focused on a specific task and developed mainly based on it. Our design stores the picture and successfully achieves the task above. Some special design specifications implemented in the design are denoted below.

1. Instruction Memory : This is a ROM type memory block of 16 by 180. (word size = 16 and depth = 180). Instruction Memory accommodates the assembly code of the whole algorithm.

2. Data Memory : This is a RAM type Memory of word size (width) 8 bits and it consists of 65536 words (depth). For convenience Data Memory is called as 'DRAM' after this point in this report. DRAM is designed to store 256 by 256 image as pixel value which are varying from 0 to 255.

3. Program Counter(PC) : The 16-bit register which keeps the address of the next instruction in the instruction memory. By this register corresponding instruction will be loaded to the CPU (Central Processing Unit).

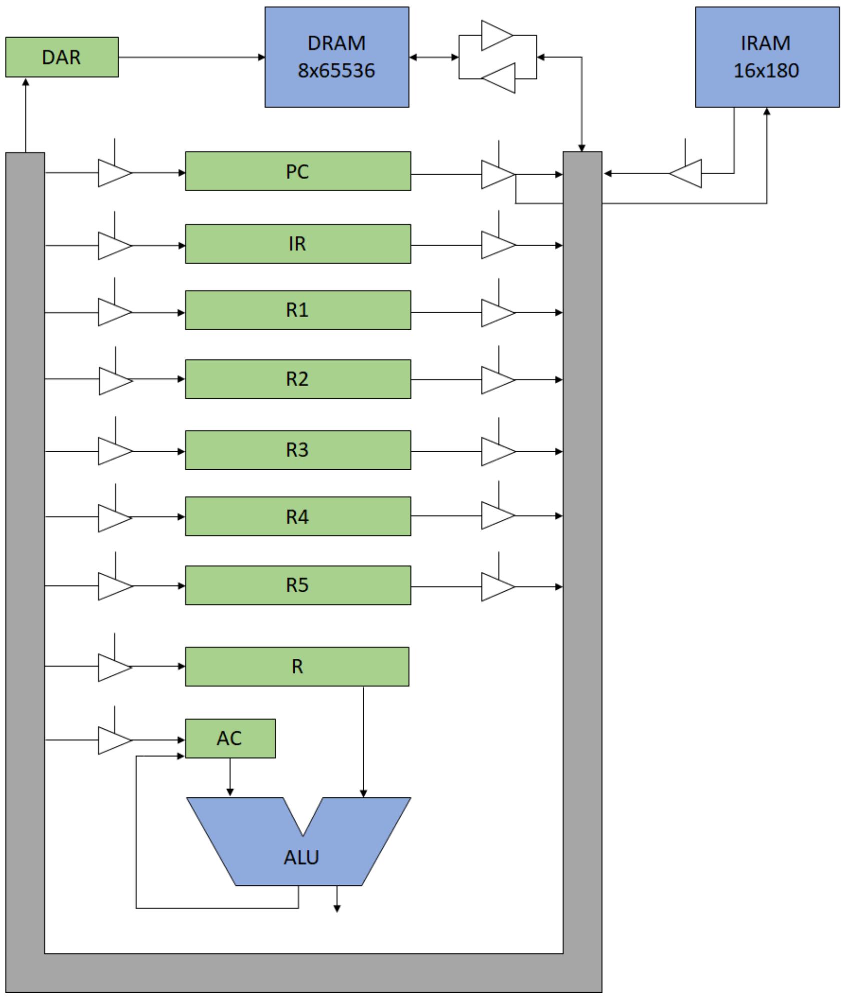
4. Accumulator(AC) : The 16-bit register which has a direct connection with the ALU. IN ALU operations this register is used most of times.

5. Data Address Register(DAR) : A special purpose register used to store an address (Memory location) of Data memory. This is used to store the memory address when reading the memory.

6. Control Unit : This module gives the control signals those are used to control the processing. This module is implemented as a state machine.

7. General purpose Registers(R, R1, R2, R3, R4, R5) : These registers are used to computational tasks and store data temporary in the CPU. Among those registers R, R4 and R5 are registers without increment and R1, R2 and R3 are registers with increment.

3.1 Datapath



Observing the above diagram of data-path, B bus have multiple inputs from registers; PC, IR, AC, DAR, R, R1, R2, R3, R4, R5. But it accepts only one input at once. This controlling is done by a multiplexer which controls the inputs of the data bus. Multiplexer accepts only one input data into data bus as the control signal given to it.

3.2 Instruction Set Architecture

The instruction set architecture is the interface between software and hardware. Furthermore, ISA defines the architecture used in the design deeply. This defines the architecture of the Central Processing Unit (CPU), Data path and the connections between different modules.

Instruction Cycle
A standard process with basic three stages of any processor is called Instruction cycle. In some places it is called as Fetch – Decode - Execute cycle. Instruction cycle is the basic mechanism of a computer. This consists of there stages. They are,

1. FETCH
2. DECODE
3. EXECUTE

These steps are executed repeatedly in the CPU. Usually those are executed in parallel modern processors. But in a simple processor like ours they run sequentially one after another.

FETCH

At the beginning both data and instructions are loaded from main memory to CPU's temporary memory elements (Registers). Then the CPU is ready to do some task according to the data which loaded from the memory. This operation is called as FETCH cycle of the Instruction cycle. For handle this operation CPU makes use of some special register called Instruction register(IR) and vital direct data path from the main memory.

EXECUTE
At execution stage activities are differ according to the instruction. Actually this is the stage which does the task of the instruction. Execution stages of different instructions are denoted in the next section.

COMPLETE INSTRUCTION SET

- LDAC
- ACTOR
- ACTOR1
- ACTOR2
- ACTOR3
- ACTOR4
- ACTOR5
- ACTODAR
- RTOAC
- R1TOAC
- R2TOAC
- R3TOAC

- R4TOAC
- R5TOAC
- DARTOAC
- STAC
- ADD
- SUB
- SHIFTLEFT
- SHIFTRIGHT
- INAC
- INDAR
- INR1
- INR2
- INR3
- LDIM
- JMPZ
- JMPNZ
- JMP
- NOP
- ENDOP

3.3 Micro-instructions

1. NOP: No operation performed. This instruction is used when previous instruction needs more time to place processed data at end points.

2. LDAC :This instruction is used obtain data from data memory (DRAM) to Accumulator register. Here address of the required Data is stored in A register. Then the data in that register is transferred to the data register from the memory. Afterwards next FETCH cycle is operated.
   LDAC1 : D[7:0] $\longleftarrow$ DRAM[DAR]; READ
   LDAC2 : AC $\longleftarrow$ [7:0]

3. STAC : This instruction is used to write data in the AC register to a given location in the data memory. At the beginning Data in AC is loaded to Data register and then it is stored in the memory location which is in the DAR.
   STAC1 : DAR⟵AC
   STAC2 : DRAM[DAR]⟵D[0:7] ; WRITE

4. ACTOR, ACTOR1, ACTOR2, ACTOR3, ACTOR4, ACTOR5: These are the instructions which moving data from AC to given special purpose register or general purpose register given in the instruction.
   ACTOR : R⟵AC
   ACTOR1 : R1⟵AC
   ACTOR2 : R2⟵AC
   ACTOR3 : R3⟵AC
   ACTOR4 : R4⟵AC
   ACTOR5 : R5⟵AC

5. RTOAC, R2TOAC, R3TOAC, R4TOAC, R5TOAC, DARTOAC: These instructions are used to move data stored in any given register to Accumulator register. At the beginning data is loaded to the data bus from the noted register in the instruction and then data is loaded to the accumulator.
   RTOAC : AC⟵R
   R1TOAC : AC⟵R1
   R2TOAC : AC⟵R2
   R3TOAC : AC⟵R3
   R4TOAC : AC⟵R4
   R5TOAC : AC⟵R5

6. ADD: In this instruction the values in the Accumulator (AC) and R register are loaded to the ALU and they are added together. After the sum is loaded to the AC register. The next FETCH cycle begins after this single state operation.
   ADD : AC⟵AC + R

7. SUB: As ADD in this instruction the value in the Accumulator (AC) R register are loaded to the ALU. Then the value in the R is subtracted from the value in AC. After the value in ALU is loaded to the AC register. The next FETCH cycle begins after this single state operation.
   SUB : AC⟵AC-R

8. SHIFTRIGHT: SHIFTRIGHT is used to divide the value in the ALU by a power of 2.After then the next FETCH cycle begins.
   SHIFTRIGHT : AC⟵AC »R

9. SHIFTLEFT: SHIFTLEFT is used to multiply the value in the ALU by a power of 2.After then the next FETCH cycle begins.
   SHIFTLEFT : AC⟵AC «R

10. INAC:This instruction is used to increase the value in AC by 1.
    INAC : AC ⟵ AC + 1

11. INR1, INR2, INR3: In these instructions are used to increase the value in R1, R2 or R3.
    INR1 : R1 ⟵ R1 + 1
    INR1 : R2 ⟵ R2 + 1
    INR1 : R3 ⟵ R3 + 1

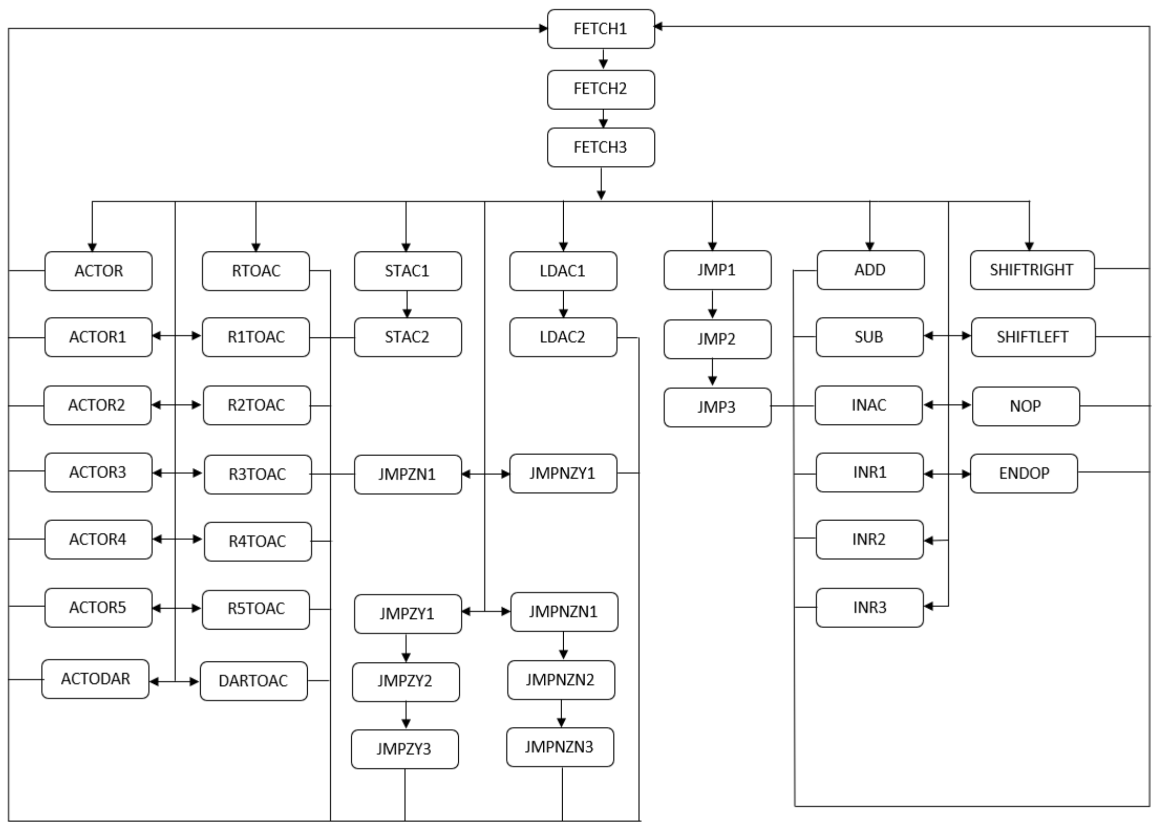
12. LDIM: This instruction is used to load a particular image pixel value to AC
    LDIM1 : PC⟵PC+1
    LDIMX : D[15:0]⟵IRAM[PC]
    LDIM2 : AC⟵D[15:0], PC⟵PC+1

13. JMPZ: In this instruction zero flag is checked and if Z=0, then no jumping is occurred and after the instruction PC is increased by 1. In the case of Z=1, states occur same as JUMP instruction. At the end the next FETCH cycle will occur at the branched location.
    JMPZN1 (Z=0) : (Z=0) PC⟵PC + 1
    JMPZY1 (Z=1) : D[15:0]⟵IRAM[PC] ; FETCH
    JMPZY2 (Z=1) : AC⟵D[15:0]
    JMPZY3 (Z=1) : PC⟵AC

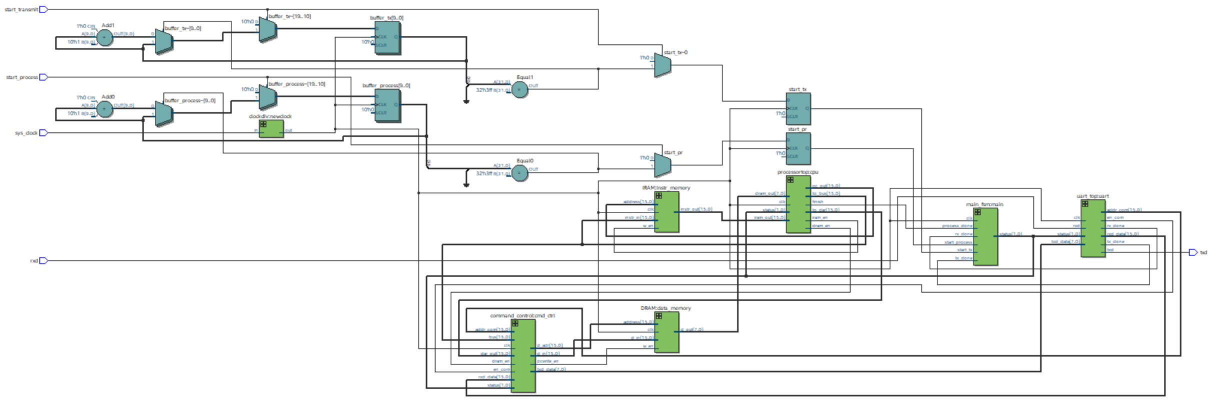
14. JMPNZ : In this instruction zero flag is checked and if Z=1, then no jumping is occurred and after the instruction PC is increased by 1. In the case of Z=0, states occur same as JUMP instruction. At the end the next FETCH cycle will occur at the branched location.
    JMPNZY1 (Z=1) : PC⟵PC + 1
    JMPNZN1 (Z=0) : D[15:0]⟵IRAM[PC] ; FETCH
    JMPNZN2 (Z=0) : AC⟵D[15:0]
    JMPNZN3 (Z=0) : PC⟵AC

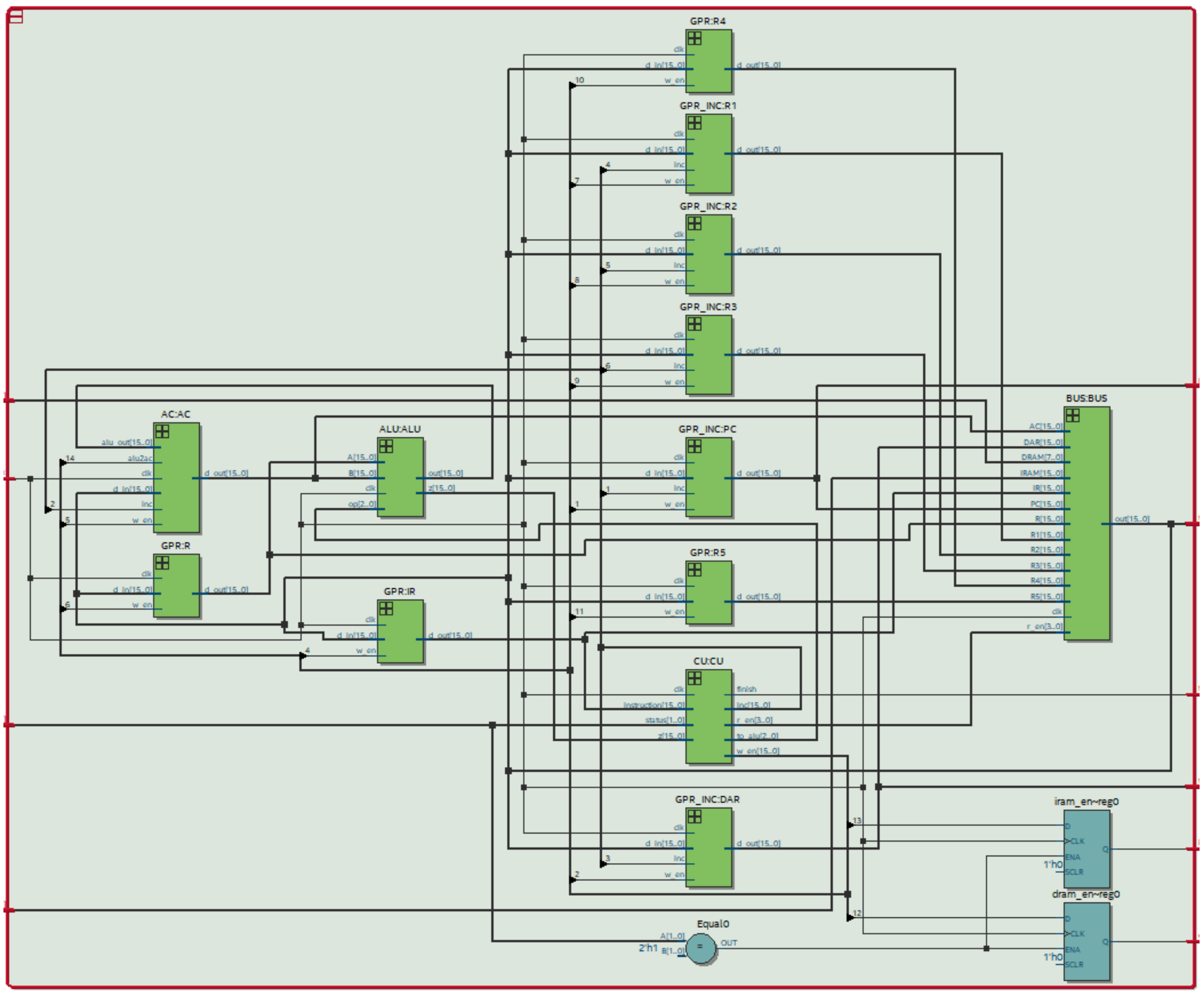
15. JMP: The address in the location in the IR pointed by the current PC value is read and loaded to DR. Then it is moved to PC via AC.
    JMP1 : D[15:0]⟵IRAM[PC] ;FETCH
    JMP2 : AC⟵D[15:0]
    JMP3 : PC⟵AC

3.4 State Machine



4 RTL View of the Design

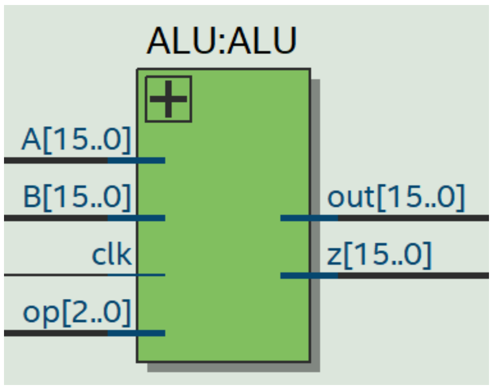

RTL View of the Processor



RTL View of the Datapath

5 RTL Modules

5.1 Arithmetic and Logic Unit

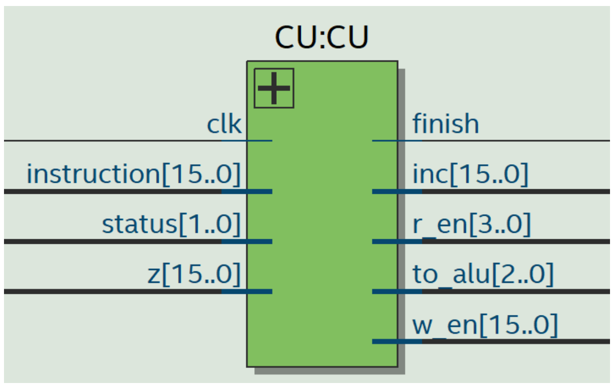


ALU handles the arithmetic and logic functions inside the processor. It takes two 16-bit inputs and according to the operation given, it gives a 16-bit output and updates the z-flag. Our ALU basically performs 4 functions;

ADD	$\longrightarrow$ A+B
SUBTRACT	$\longrightarrow$ A−B
SHIFT LEFT	$\longrightarrow$ A «B
SHIFT RIGHT	$\longrightarrow$ A »B

Z-flag is updated as follows;
Z=1 when out$=0$
Z=0 when out$\neq 0$

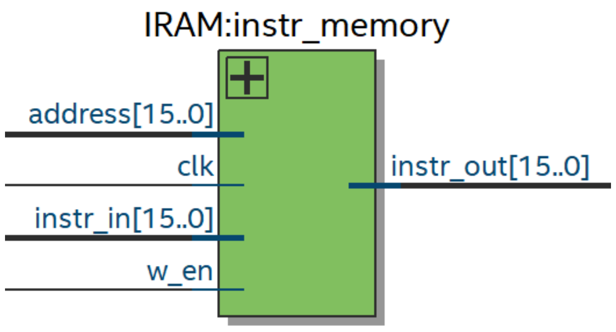
5.2 Control Unit



Control unit is responsible of decoding the instructions and generate control signals. Each instruction is decoded into 4 signals:

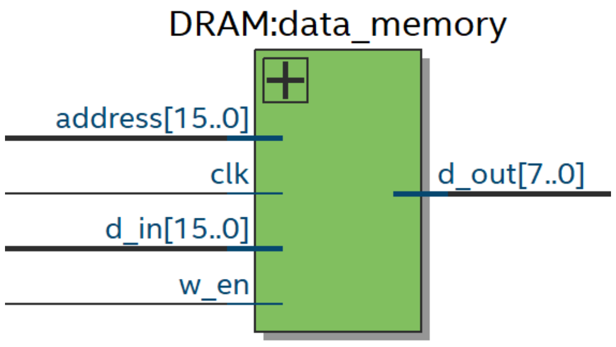
1. inc - register/registers to be incremented
2. r_en - which register is read from
3. w_en - which register is written to
4. to_alu - alu operation associated with the instruction

5.3 Instruction Memory



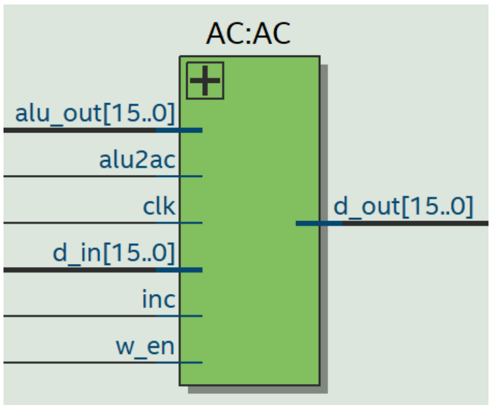
We used a **Read-only One-port ROM** for the Instruction memory which is used to store the **ASSEMBLY CODE** for the downsampling task. The assembly code consists of 177 lines and it is hardcoded to the IRAM verilog module.

5.4 Data Memory



We used a **One-port RAM** for the Data memory which is used to store the pixel values of the original image and the downsampled image. Before starting the downsampling process, pixel data is written to the DRAM by UART and when the processing is done, the final pixel values are also stored concurrently. Then when the process is done, the downsampled pixel data is sent back to the computer using UART to view using *Matlab*.

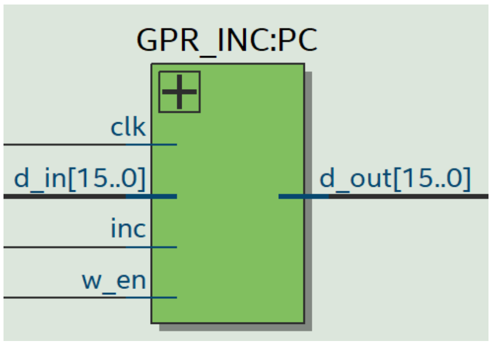
5.5 Accumulator



Accumulator (AC) accumulates the output of the ALU and serves as an input to the ALU as well. AC can load data from both ALU and Bus according to the active-high control bits *alu2ac* and *w_en*. Setting *inc* bit high increments the value in AC.

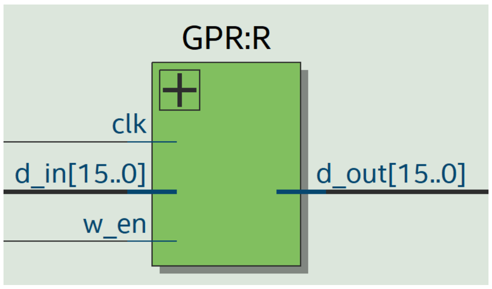
5.6 General Purpose Registers (GPR)

5.6.1 GPR with Increment



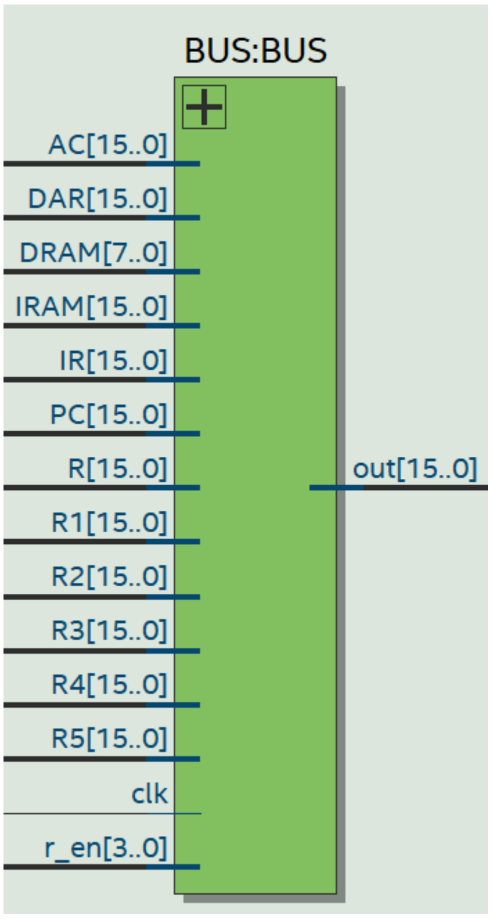
The General Purpose register is modeled such that when *w_en* is high, it loads whatever in the bus to the register. This has the ability to increment itself when *inc* is given. We have instantiated this module several times in our processor module to model; **PC, DAR, R1, R2** and **R3**.

5.6.2 GPR without Increment



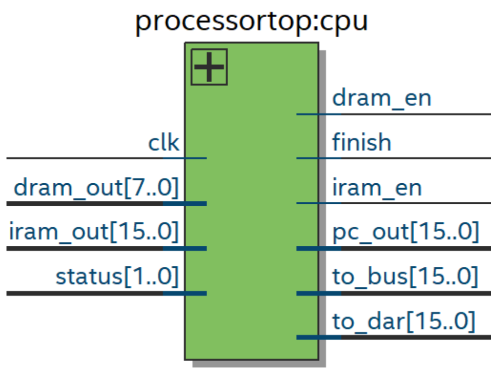
This functions similarly to the above module only it does not have the ability to increment itself. We have instantiated this module several times in our processor module to model; **IR, R, R4** and **R5**.

5.7 Bus



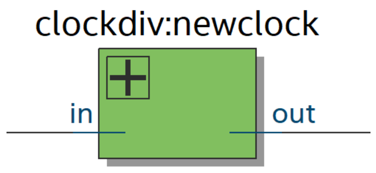
We used only a single 16-bit bus in our datapath to make it less complex. The outputs of all **R, R1, R2, R3, R4, R5, DAR, IR, PC, AC, DRAM** and **IRAM** are connected to the inputs of the BUS and it loads one input (when anyone of them changes) according to the input signal given by the **CU** ; *r_en*.

5.8 Processor



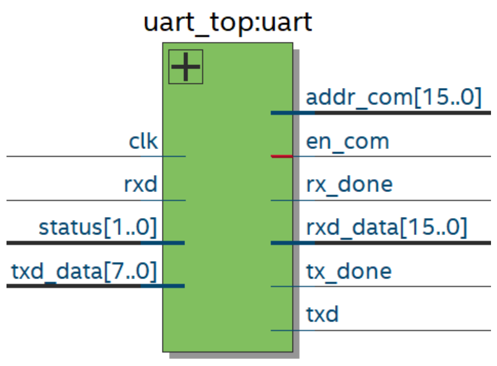
This is the module which connects all the datapath components. It has its inputs from the DRAM and IRAM and outputs PC value and enable signals to DRAM and IRAM. It takes *status* as an input and enables IRAM and DRAM when the status in the **main control** is in the "PROCESS" state.

5.9 Clock Divider



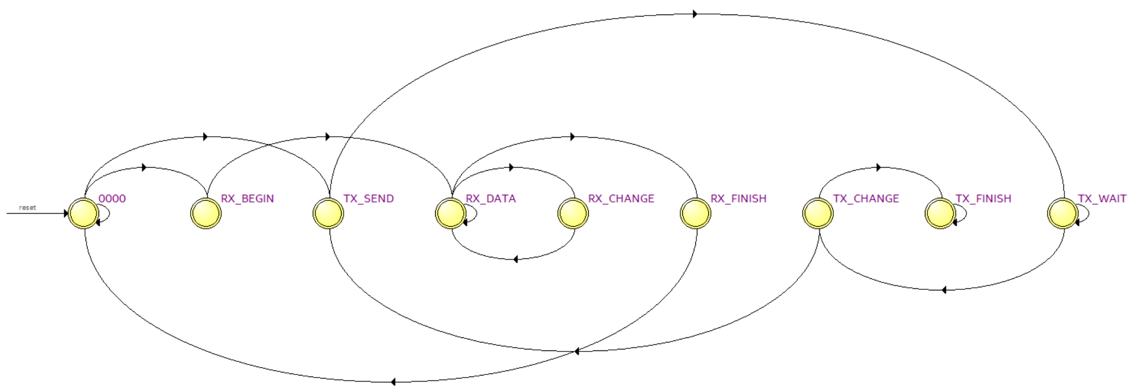
This module generates a clock with half the frequency as the input clock. The system clock of the DE-115 Altera Board is 50Hz. UART communication at this rate results in huge bit errors. Therefore, we use 25Hz clock generated by this module for the UART implentation as well as the processing part.

5.10 UART Modules

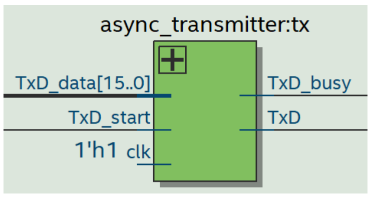


We used a UART IP Core from the `https://opencores.org/` and amended to suit our parameters and used it in our design. **uart_top** module is a finite state machine written to connect transmitter and reciever modules and to provide the status of transmission to the top module of the design.

Quartus State Machine Viewer gives the state machine as follows;

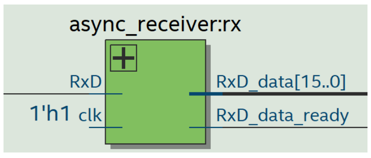


5.10.1 Transmitter



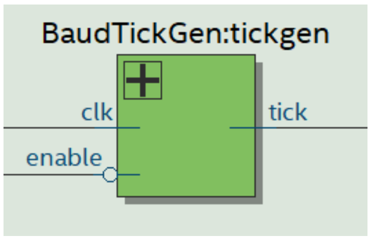
This module uses an internal state machine to send the processed serial data (Pixel values of the downsampled image) to the Computer from the FPGA Board.

5.10.2 Receiver



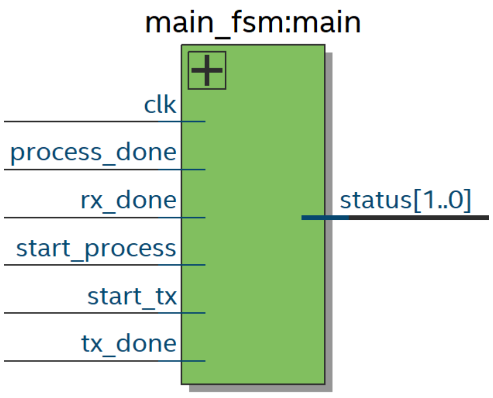
This module uses an internal state machine to receive the original serial data (Pixel values of the 256x256 image) to the FPGA Board.

5.10.3 Baud Rate Generator

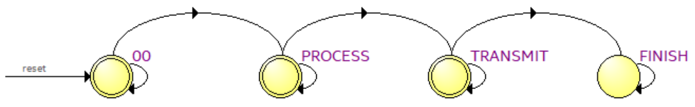


This module synchronizes the Baud rate (9600bps) in which the serial data is transmitted and received with the clock used in the design (25MHz).

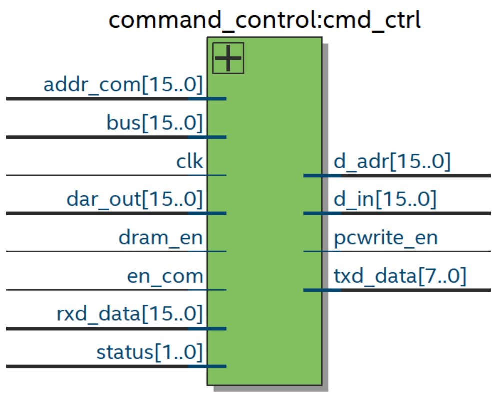
5.11 Main State Machine



This is the main finite state machine of the design and it controls the states where **RECEIVING, PROCESSING** and **TRANSMITTING** is done. It takes the inputs from the communicate and top module and shifts the states according to those inputs. Quartus State Machine viewer gives the fsm as follows;



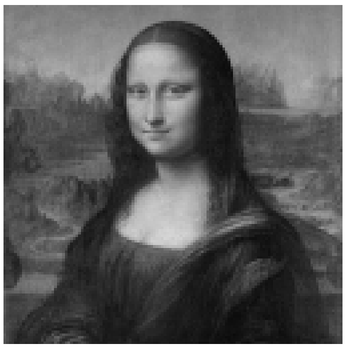
5.12 Command Control

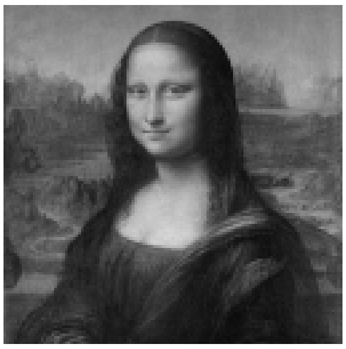


This is the module which gives commands to the Receiver, transmitter and the processor to start and end their tasks. The commands are given according to which state the Main State Machine runs on.

6 Results and Error-analysis

Original 256x256 Image


Matlab Downsampled Image

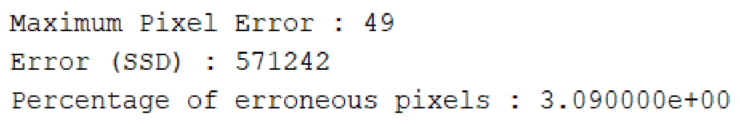

FPGA Downsampled Image

To verify the results obtained by the FPGA, we downsampled the same picture using the matlab and compared it with the fpga downsampled image. Following criteria were used to analyze the results.

1. Maximum pixel error : The maximum difference between any pixel of the FPGA downsampled image and the Matlab downsampled reference image.
2. SSD : Sum of the squared pixel differences of the two images.

3. Percentage Error : The percentage of number of erroneous pixels to all the pixels of the image.

The results are as follows;



```
Maximum Pixel Error : 49
Error (SSD) : 571242
Percentage of erroneous pixels : 3.090000e+00
```

7 Discussion

1. At the beginning of the project, our main focus was to decide which architecture to be used; RISC or CISC. Since many resources were there for CISC architecture on the internet and we were taught the basics in the Lecture, we selectec a CISC architecture for our design.

2. When we first designed the state machine, we observed that some tasks in the states could not be done in a single clock cycle and therefore our design failed in the timing analysing stage. Therefore we had to allocated more than one cycle to some tasks.

3. Our precentage error is less than 5% but we got a maximum pixel error of 49. We observed that only the slight difference between floating point arithmetic in matlab and integer arithmetic in our hardware implementation could not have resulted in such an error. The main reason for that amount of an error could be because of the faults in UART communication. We did not have much time to verify our IP Core before using. Had it been verified and used, this error could have been avoided.

8 References

1. Lecture Notes

2. Arduino FPGA interface using Verilog RS232 serial communication xilinx spartan 3 Waveshare : Juan Felipe P.V. `http://quitoart.blogspot.com/2018/01/arduino-fpga-interface-using-verilog.html`

3. Open Source IP Cores at `https://opencores.org/`

4. Computer Systems Organization Architecture: John D. Carpinelli

5. Hardware Modeling Using Verilog : Prof. Indranil Senguptha, NPTEL `https://nptel.ac.in/courses/106105165/`

9 Appendix

9.1 Matlab Scripts

9.1.1 Grayscale Image Down-sampling

```
instrreset;

size = 256;
orig_image = imread('C:\Users\DNM_PC\Desktop\Mona.JPG');
orig_image = rgb2gray(orig_image);
orig_image = orig_image(1:size,1:size);

inverted = orig_image';
row= inverted(:);
l_image=length(row);

%% Sending Image
disp('Press Space bar to start the Process');
pause;

UART= serial('COM11', 'BaudRate', 9600);
fopen(UART);

disp('Sending 65536 Pixels to the FPGA. This may take some time');

%Sending Pixels one by one
for (i=1:l_image)
   fwrite(UART,row(i),'uint8');
end
disp('Sending completed');
disp('Processing.......');

final_pixels = 16130;
recieve=zeros(final_pixels,1);

disp('Receiving downsampled image from the FPGA. This may take some time');

for (j=1:final_pixels)
  recieve(j,1)=fread(UART,1);
end

disp('Down-sampling Completed Successfully!!!!');

fclose(UART);
```

```
received = uint8(recieve(2:end));
down_image = reshape(received,127,127);

filter_kernel = [1 3 1; 3 16 3; 1 3 1];

filtered_img = floor(conv2(orig_image,filter_kernel)./32);
filtered_img = filtered_img(3:size,3:size);

down_image_matlab = zeros(size/2-1);

i=1;
j=1;
row_num = 1;
while i <= ((64516) - (255))

    down_image_matlab(j) =  filtered_img(i);
    j = j+1;
    if row_num == (size-2-1);
                i = i+size;
                row_num = 1;
    else
                row_num = row_num + 2;
                i = i+2;
    end
end

down_image_matlab = uint8(down_image_matlab);
down_image = down_image';

figure
imshow(orig_image)
title('Original Image')

figure
imshow(down_image_matlab)
title('Image Downsampled with Matlab')

figure
imshow(down_image)
img_error = down_image_matlab-down_image;
title('Image Downsampled with FPGA')


E = sprintf('Maximum Pixel Error : %d',max(max(img_error)));
```

```
disp(E);
sumError1 = sum(sum(abs(img_error)));
sumError2 = sum(sum(img_error.^2));
Y = sprintf('Absolute Error : %d',sumError1);
disp(Y);
X = sprintf('Error (SSD) : %d',sumError2);
disp(X);

num_errors =0;
for j=1:127
    for i=1:127
        if img_error(i,j)>10
            num_errors=num_errors+1;
            img_error(i,j)=1;
        else
            img_error(i,j)=0;
        end

    end
end


errorp=num_errors/16130;
P = sprintf('Percentage of erroneous pixels : %d',errorp);
disp(P);
```

9.1.2 Colour Image Down-sampling

```
instrreset;

size = 256;
orig_image = imread('Mona.jpg');

redChannel = orig_image(:,:,1); % Red channel
redChannel = redChannel(1:size,1:size);
greenChannel = orig_image(:,:,2); % Green channel
greenChannel = greenChannel(1:size,1:size);
blueChannel = orig_image(:,:,3); % Blue channel
blueChannel = blueChannel(1:size,1:size);

inverted = redChannel';
row= inverted(:);
l_image=length(row);

%% Sending Red channel Image
```

```
disp('Press Space bar to start the Process');
pause;

UART= serial('COM11', 'BaudRate', 9600);
fopen(UART);

disp('Sending 65536 Pixels to the FPGA. This may take some time');

for (i=1:l_image)
   fwrite(UART,row(i),'uint8');
end

disp('Sending completed');
disp('Processing.......');

final_pixels = 16130;
recieve=zeros(final_pixels,1);

disp('Receiving downsampled Red image from the FPGA. This may take some time');

for (j=1:final_pixels)
  recieve(j,1)=fread(UART,1);
end

fclose(UART);

received = uint8(recieve(2:end));
down_image_r = reshape(received,127,127);
%---------------------------------------------------------------
%Red Channel Completed
%---------------------------------------------------------------

inverted = greenChannel';
row= inverted(:);
l_image=length(row);

%% Sending Green Channel Image

UART= serial('COM11', 'BaudRate', 9600);
fopen(UART);

disp('Sending 65536 Pixels to the FPGA. This may take some time');

for (i=1:l_image)
   fwrite(UART,row(i),'uint8');
```

```
end

disp('Sending completed');
disp('Processing.......');

final_pixels = 16130;
recieve=zeros(final_pixels,1);

disp('Receiving downsampled Green image from the FPGA. This may take some time');

for (j=1:final_pixels)
  recieve(j,1)=fread(UART,1);
end

fclose(UART);

received = uint8(recieve(2:end));
down_image_g = reshape(received,127,127);
%-----------------------------------------------------------------
%Green Channel Completed
%-----------------------------------------------------------------

inverted = blueChannel';
row= inverted(:);
l_image=length(row);

%% Sending Blue Channel Image

UART= serial('COM11', 'BaudRate', 9600);
fopen(UART);

disp('Sending 65536 Pixels to the FPGA. This may take some time');

for (i=1:l_image)
   fwrite(UART,row(i),'uint8');
end

disp('Sending completed');
disp('Processing.......');

final_pixels = 16130;
recieve=zeros(final_pixels,1);

disp('Receiving downsampled Blue image from the FPGA. This may take some time');
```

```
for (j=1:final_pixels)
  recieve(j,1)=fread(UART,1);
end

fclose(UART);

received = uint8(recieve(2:end));
down_image_b = reshape(received,127,127);
%-----------------------------------------------------------------
%Blue Channel Completed
%-----------------------------------------------------------------

down_image = cat(3, down_image_r, down_image_g, down_image_b);

down_image = down_image';

figure
imshow(orig_image)
title('Original Image')

figure
imshow(down_image)
title('Image Downsampled by FPGA')
```

9.2 Verilog Scripts

9.2.1 ALU

```
module ALU(input clk,
                input [15:0] A,
                input [15:0] B,
                input [2:0] op,
                output reg [15:0] out,
                output reg [15:0] z );

always @(posedge clk)
                begin
                case(op)
                        3'b001: out <= A + B;
                        3'b010: out <= A - B;
                        3'b011: out <= A << B;
                        3'b100: out <= A >> B;
                endcase

                if (~out)
                        z <= 1;
                else
                        z <= 0;
                end

endmodule
```

9.2.2 CU

```
module CU(
                input clk,
                input [15:0] z,
                input [15:0] instruction,
                output reg[2:0] to_alu,
                output reg[15:0] w_en,
                output reg[15:0] inc,
                output reg[3:0] r_en,
                output reg finish,
                input [1:0] status
                );

reg [5:0] PS = 6'd0;
reg [5:0] NS = 6'd0;

localparam
```

```
IDLE                    = 6'd0,
FETCH1                     = 6'd1,
FETCH2                     = 6'd2,
FETCHX                     = 6'd44,
LDAC1                    = 6'd3,
LDAC2                    = 6'd4,
LDAC1X                     = 6'd50,
LDAC2X                     = 6'd51,
ACTOR                    = 6'd5,
ACTOR1                     = 6'd6,
ACTOR2                     = 6'd7,
ACTOR3                     = 6'd8,
ACTOR4                     = 6'd9,
ACTOR5                     = 6'd10,
ACTODAR                    = 6'd11,
RTOAC                    = 6'd12,
R1TOAC                     = 6'd13,
R2TOAC                     = 6'd14,
R3TOAC                     = 6'd15,
R4TOAC                     = 6'd16,
R5TOAC                     = 6'd17,
DARTOAC                    = 6'd18,
STAC                     = 6'd19,
STAC2                     = 6'd43,
STACX                     = 6'd49,
STACY                     = 6'd52,
ADD1                     = 6'd20,
ADD2                     = 6'd21,
SUB1                     = 6'd22,
SUB2                     = 6'd23,
SHIFTLEFT1            = 6'd24,
SHIFTLEFT2            = 6'd25,
SHIFTRIGHT1            = 6'd26,
SHIFTRIGHT2            = 6'd27,
INAC                     = 6'd28,
INDAR                     = 6'd29,
INR1                     = 6'd30,
INR2                     = 6'd31,
INR3                     = 6'd32,
LDIM1                     = 6'd33,
LDIM2                     = 6'd34,
LDIMX                     = 6'd45,
JMPZ1                     = 6'd35,
JMPZ2                     = 6'd36,
JMPZ3                     = 6'd37,
JMPZ4                  = 6'd38,
JMPZ2X                 = 6'd46,
JMPZ3X                 = 6'd47,
JMPZ4X                 = 6'd48,
JMPNZ1                 = 6'd39,
JMP1                 = 6'd40,
NOP                        = 6'd41,
ENDOP                 = 6'd42;


always @(posedge clk)
        PS <= NS;

always @(posedge clk)
        begin
                if (PS == ENDOP)
                        finish <= 1'd1;
                else
                        finish <= 1'd0;
        end

always @(PS or z or instruction or status)
        case(PS)
                IDLE: begin
                        if (status == 2'b01)
                                NS = FETCH1;
                        else
                                NS = IDLE;
                end

                FETCH1: begin
                        NS = FETCHX;
                end

                FETCHX: begin
                        NS = FETCH2;
                end

                FETCH2: begin
                        NS = instruction[5:0];
                end

                LDAC1: begin
                        NS = LDAC1X;
                end
```

```
LDAC1X: begin
        NS = LDAC2X;
end

LDAC2X: begin
        NS = LDAC2;
end

LDAC2: begin
        NS = FETCH1;
end

ACTOR: begin
        NS = FETCH1;
end

ACTOR1: begin
        NS = FETCH1;
end

ACTOR2: begin
        NS = FETCH1;
end

ACTOR3: begin
        NS = FETCH1;
end

ACTOR4: begin
        NS = FETCH1;
end

ACTOR5: begin
        NS = FETCH1;
end

ACTODAR: begin
        NS = FETCH1;
end

RTOAC: begin
        NS = FETCH1;
end
```

```
                R1TOAC: begin
                        NS = FETCH1;
                end

                R2TOAC: begin
                        NS = FETCH1;
                end

                R3TOAC: begin
                        NS = FETCH1;
                end

                R4TOAC: begin
                        NS = FETCH1;
                end

                R5TOAC: begin
                        NS = FETCH1;
                end

                DARTOAC: begin
                        NS = FETCH1;
                end

                STAC: begin
                        NS = STAC2;
                end

                STAC2: begin
                        NS = STACX;
                end

                STACX: begin
                        NS = FETCH1;
                end

                ADD1: begin
                        NS = ADD2;
                end

                ADD2: begin
                        NS = FETCH1;
                end

                SUB1: begin
                        NS = SUB2;
                end

                SUB2: begin
                        NS = FETCH1;
                end

                SHIFTLEFT1: begin
                        NS = SHIFTLEFT2;
                end

                SHIFTLEFT2: begin
                        NS = FETCH1;
                end

                SHIFTRIGHT1: begin
                        NS = SHIFTRIGHT2;
                end

                SHIFTRIGHT2: begin
                        NS = FETCH1;
                end

                INAC: begin
                        NS = FETCH1;
                end

                INDAR: begin
                        NS = FETCH1;
                end

                INR1: begin
                        NS = FETCH1;
                end

                INR2: begin
                        NS = FETCH1;
                end

                INR3: begin
                        NS = FETCH1;
                end

                LDIM1: begin
                        NS = LDIMX;
```

```
end

LDIMX: begin
        NS = LDIM2;
end

LDIM2: begin
        NS = FETCH1;
end

JMP1: begin
        NS = JMPZ2X;
end

JMPZ1: begin
        if (z == 1)
                NS = JMPZ2X;
        else
                NS = JMPZ4X;
end

JMPNZ1: begin
        if (z == 0 )
                NS = JMPZ2;
        else
                NS = JMPZ4;
end

JMPZ2X: begin
        NS = JMPZ2;
end

JMPZ2: begin
        NS = JMPZ3X;
end

JMPZ3X: begin
        NS = JMPZ3;
end

JMPZ3: begin
        NS = FETCH2;
end

JMPZ4X: begin
```

```
                NS = JMPZ4;
        end

        JMPZ4: begin
                NS = JMPZ3X;
        end

        NOP: begin
                NS = FETCH1;
        end

        ENDOP: begin
                NS = ENDOP;
        end

        default: begin
                NS = FETCH1;
        end
    endcase

always @(*)
begin
case(PS)
    IDLE, ADD1, ADD2, SUB1, SUB2, SHIFTLEFT1, SHIFTLEFT2, SHIFTRIGHT1,
    SHIFTRIGHT2, INAC, INDAR, INR1, INR2, INR3, LDIM1, JMP1,  JMPZ1,
    JMPNZ1, JMPZ4X, JMPZ4, NOP:
            r_en = 4'd0;
    FETCH1, FETCH2, FETCHX, LDIMX, LDIM2, JMPZ2X, JMPZ2, JMPZ3X, JMPZ3:
            r_en = 4'd13;
    LDAC1, LDAC1X, ACTOR, ACTOR1, ACTOR2, ACTOR3, ACTOR4, ACTOR5,
    ACTODAR, STAC, STAC2, STACX:
            r_en = 4'd5;
    LDAC2X, LDAC2, ENDOP:
            r_en = 4'd12;
    RTOAC:
            r_en = 4'd6;
    R1TOAC:
            r_en = 4'd7;
    R2TOAC:
            r_en = 4'd8;
    R3TOAC:
            r_en = 4'd9;
    R4TOAC:
            r_en = 4'd10;
    R5TOAC:
```

```
                r_en = 4'd11;
        DARTOAC:
                r_en = 4'd2;
endcase
end

always @(*)
begin
case(PS)
        IDLE, FETCH2, STAC, STACX, ADD1, SUB1, SHIFTLEFT1, SHIFTRIGHT1,
        INAC, INDAR, INR1, INR2, INR3, LDIM1, LDIMX, JMP1,  JMPZ1, JMPNZ1,
        JMPZ4X, JMPZ4, NOP, ENDOP:
                w_en = 16'b0000000000000000;
        FETCH1, FETCHX, JMPZ3X, JMPZ3:
                w_en = 16'b0000000000010000;
        LDAC1, LDAC1X, ACTODAR:
                w_en = 16'b0000000000000100;
        LDAC2X, LDAC2, RTOAC, R1TOAC, R2TOAC, R3TOAC, R4TOAC, R5TOAC,
        DARTOAC, LDIM2:
                w_en = 16'b0000000000100000;
        ACTOR:
                w_en = 16'b0000000001000000;
        ACTOR1:
                w_en = 16'b0000000010000000;
        ACTOR2:
                w_en = 16'b0000000100000000;
        ACTOR3:
                w_en = 16'b0000001000000000;
        ACTOR4:
                w_en = 16'b0000010000000000;
        ACTOR5:
                w_en = 16'b0000100000000000;
        STAC2:
                w_en = 16'b0001000000000000;
        ADD2, SUB2, SHIFTLEFT2, SHIFTRIGHT2:
                w_en = 16'b1100000000000000;
        JMPZ2X, JMPZ2:
                w_en = 16'b0000000000000010;

endcase
end

always @(*)
begin
case(PS)
```

```
        IDLE, FETCH1, FETCHX, FETCH2, LDAC1, LDAC1X, LDAC2X, STAC, STAC2,
        ADD1, SUB1, SHIFTLEFT1, SHIFTRIGHT1, LDIMX, JMPZ2X, JMPZ2, JMPZ3X,
        JMPZ3, JMPZ4X, ENDOP:
                inc = 16'b0000000000000000;
        LDAC2, RTOAC, R1TOAC, R2TOAC, R3TOAC, R4TOAC, R5TOAC, DARTOAC, ACTOR,
        ACTOR1, ACTOR2, ACTOR3, ACTOR4, ACTOR5, ACTODAR, STACX, ADD2, SUB2,
        SHIFTLEFT2, SHIFTRIGHT2, LDIM1, LDIM2, JMP1,  JMPZ1, JMPNZ1, JMPZ4, NOP:
                inc = 16'b0000000000000010;
        INAC:
                inc = 16'b0000000000000110;
        INDAR:
                inc = 16'b0000000000001010;
        INR1:
                inc = 16'b0000000000010010;
        INR2:
                inc = 16'b0000000000100010;
        INR3:
                inc = 16'b0000000001000010;
endcase
end

always @(*)
begin
case(PS)
        IDLE, FETCH1, FETCHX, FETCH2, LDAC1, LDAC1X, LDAC2X, LDAC2, RTOAC,
        R1TOAC, R2TOAC, R3TOAC, R4TOAC, R5TOAC, DARTOAC, ACTOR, ACTOR1, ACTOR2,
        ACTOR3, ACTOR4, ACTOR5, ACTODAR, STAC, STAC2, STACX, INAC, INDAR, INR1,
        INR2, INR3, LDIM1, LDIM2, LDIMX, JMP1,  JMPZ1, JMPNZ1, JMPZ2X, JMPZ2,
        JMPZ3X, JMPZ3, JMPZ4X, JMPZ4, NOP, ENDOP:
                to_alu = 3'd0;
        ADD1, ADD2:
                to_alu = 3'd1;
        SUB1, SUB2:
                to_alu = 3'd2;
        SHIFTLEFT1, SHIFTLEFT2:
                to_alu = 3'd3;
        SHIFTRIGHT1, SHIFTRIGHT2:
                to_alu = 3'd4;
endcase
end

endmodule
```

9.2.3 IRAM

```
module IRAM(
                input clk,
                input w_en,
                input [15:0] address,
                input [15:0] instr_in,
                output reg [15:0] instr_out
                );


reg [15:0] MEM [180:0];

parameter LDAC                        = 8'd3;
parameter ACTOR                        = 8'd5;
parameter ACTOR1                 = 8'd6;
parameter ACTOR2                 = 8'd7;
parameter ACTOR3                 = 8'd8;
parameter ACTOR4                 = 8'd9;
parameter ACTOR5                 = 8'd10;
parameter ACTODAR                = 8'd11;
parameter RTOAC                 = 8'd12;
parameter R1TOAC                 = 8'd13;
parameter R2TOAC                 = 8'd14;
parameter R3TOAC                 = 8'd15;
parameter R4TOAC                 = 8'd16;
parameter R5TOAC                 = 8'd17;
parameter DARTOAC                = 8'd18;
parameter STAC                        = 8'd19;
parameter ADD                         = 8'd20;
parameter SUB                         = 8'd22;
parameter SHIFTLEFT                = 8'd24;
parameter SHIFTRIGH                = 8'd26;
parameter INAC                         = 8'd28;
parameter INDAR                 = 8'd29;
parameter INR1                         = 8'd30;
parameter INR2                         = 8'd31;
parameter INR3                         = 8'd32;
parameter LDIM                         = 8'd33;
parameter JMPZ                         = 8'd35;
parameter JMPNZ                 = 8'd39;
parameter JMP                         = 8'd40;
parameter NOP                         = 8'd41;
parameter ENDOP                 = 8'd42;
```

```
initial begin
        MEM[0] = LDIM;
        MEM[1] = 16'd257;
        MEM[2] = ACTOR1;
        MEM[3] = R1TOAC;
        MEM[4] = LDAC;
        MEM[5] = ACTOR;
        MEM[6] = LDIM;
        MEM[7] = 8'd4;
        MEM[8] = SHIFTLEFT;
        MEM[9] = ACTOR4;
        MEM[10] = LDIM;
        MEM[11] = 8'd1;
        MEM[12] = ACTOR;
        MEM[13] = R1TOAC;
        MEM[14] = ADD;
        MEM[15] = LDAC;
        MEM[16] = ACTOR5;
        MEM[17] = R1TOAC;
        MEM[18] = SUB ;
        MEM[19] = LDAC;
        MEM[20] = ACTOR;
        MEM[21] = R5TOAC;
        MEM[22] = ADD;
        MEM[23] = ACTOR5;
        MEM[24] = LDIM;
        MEM[25] = 16'd256;
        MEM[26] = ACTOR;
        MEM[27] = R1TOAC;
        MEM[28] = ADD;
        MEM[29] = LDAC;
        MEM[30] = ACTOR;
        MEM[31] = R5TOAC;
        MEM[32] = ADD;
        MEM[33] = ACTOR5;
        MEM[34] = LDIM;
        MEM[35] = 16'd256;
        MEM[36] = ACTOR;
        MEM[37] = R1TOAC;
        MEM[38] = SUB;
        MEM[39] = LDAC;
        MEM[40] = ACTOR;
        MEM[41] = R5TOAC;
        MEM[42] = ADD;
        MEM[43] = ACTOR5;
        MEM[44] = ACTOR;
        MEM[45] = LDIM;
        MEM[46] = 8'd1;
        MEM[47] = SHIFTLEFT;
        MEM[48] = ACTOR;
        MEM[49] = R5TOAC;
        MEM[50] = ADD;
        MEM[51] = ACTOR;
        MEM[52] = R4TOAC;
        MEM[53] = ADD;
        MEM[54] = ACTOR4;
        MEM[55] = LDIM;
        MEM[56] = 16'd257;
        MEM[57] = ACTOR;
        MEM[58] = R1TOAC;
        MEM[59] = ADD;
        MEM[60] = LDAC;
        MEM[61] = ACTOR5;
        MEM[62] = R1TOAC;
        MEM[63] = SUB;
        MEM[64] = LDAC;
        MEM[65] = ACTOR;
        MEM[66] = R5TOAC;
        MEM[67] = ADD;
        MEM[68] = ACTOR5;
        MEM[69] = LDIM;
        MEM[70] = 8'd255;
        MEM[71] = ACTOR;
        MEM[72] = R1TOAC;
        MEM[73] = ADD;
        MEM[74] = LDAC;
        MEM[75] = ACTOR;
        MEM[76] = R5TOAC;
        MEM[77] = ADD;
        MEM[78] = ACTOR5;
        MEM[79] = LDIM;
        MEM[80] = 8'd255;
        MEM[81] = ACTOR;
        MEM[82] = R1TOAC;
        MEM[83] = SUB;
        MEM[84] = LDAC;
        MEM[85] = ACTOR;
        MEM[86] = R5TOAC;
        MEM[87] = ADD;
        MEM[88] = ACTOR5;
        MEM[89] = ACTOR;
        MEM[90] = R4TOAC;
        MEM[91] = ADD;
        MEM[92] = ACTOR;
        MEM[93] = LDIM;
        MEM[94] = 8'd5;
        MEM[95] = SHIFTRIGHT;
        MEM[96] = ACTOR4;
        MEM[97] = LDIM;
        MEM[98] = 16'd257;
        MEM[99] = ACTOR;
        MEM[100] = R1TOAC;
        MEM[101] = SUB;
        MEM[102] = ACTODAR;
        MEM[103] = R4TOAC;
        MEM[104] = STAC;
        MEM[105] = LDIM;
        MEM[106] = 16'd65278;
        MEM[107] = ACTOR;
        MEM[108] = R1TOAC;
        MEM[109] = SUB;
        MEM[110] = JMPZ;
        MEM[111] = 8'd138;
        MEM[112] = LDIM;
        MEM[113] = 8'd253;
        MEM[114] = ACTOR;
        MEM[115] = R2TOAC;
        MEM[116] = SUB ;
        MEM[117] = JMPZ;
        MEM[118] = 8'd129;
        MEM[119] = INR2;
        MEM[120] = INR1;
        MEM[121] = JMP;
        MEM[122] = 8'd9;
        MEM[123] = INR1;
        MEM[124] = INR1;
        MEM[125] = INR1;
        MEM[126] = LDIM;
        MEM[127] = 8'd0;
        MEM[128] = ACTOR2;
        MEM[129] = JMP;
        MEM[130] = 8'd9;
        MEM[131] = ENDOP;
        MEM[132] = LDIM;
        MEM[133] = 8'd0;
```

```
MEM[134] = ACTOR1;
MEM[135] = ACTOR2;
MEM[136] = ACTOR3;
MEM[137] = R1TOAC;
MEM[138] = LDAC;
MEM[139] = ACTOR4;
MEM[140] = R3TOAC;
MEM[141] = ACTODAR;
MEM[142] = R4TOAC;
MEM[143] = STAC;
MEM[144] = R1TOAC;
MEM[145] = ACTOR;
MEM[146] = LDIM;
MEM[147] = 16'd64764;
MEM[148] = SUB;
MEM[149] = JMPZ;
MEM[150] = 8'd186;
MEM[151] = R2TOAC;
MEM[152] = ACTOR;
MEM[153] = LDIM;
MEM[154] = 8'd252;
MEM[155] = SUB;
MEM[156] = JMPZ;
MEM[157] = 8'd171;
MEM[158] = INR2;
MEM[159] = INR2;
MEM[160] = INR1;
MEM[161] = INR1;
MEM[162] = INR3;
MEM[163] = JMP;
MEM[164] = 8'd143;
MEM[165] = LDIM;
MEM[166] = 8'd0;
MEM[167] = ACTOR2;
MEM[168] = LDIM;
MEM[169] = 16'd260;
MEM[170] = ACTOR;
MEM[171] = R1TOAC;
MEM[172] = ADD;
MEM[173] = ACTOR1;
MEM[174] = INR3;
MEM[174] = NOP;
MEM[175] = JMP;
MEM[176] = 8'd143;
MEM[177] = ENDOP;
```

```
end

always @(posedge clk) begin
        if (w_en == 1)
                MEM[address] <= instr_in[7:0];
        else
                instr_out <= MEM[address];
end
endmodule
```

9.2.4 DRAM

```
module DRAM(
        input clk,
        input w_en,
        input [15:0] address,
        input [15:0] d_in,
        output reg [7:0] d_out );

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

always @(posedge clk)
begin
        if (w_en == 1)
                MEM[address] <= d_in[7:0];
        else
                d_out <= MEM[address];
end

endmodule
```

9.2.5 AC

```
module AC(
        input clk,
        input w_en,
        input [15:0] d_in,
        output reg [15:0] d_out,
        input [15:0] alu_out,
        input alu2ac,
        input inc);

initial
        begin
                d_out<= 16'd0;
        end

always @(posedge clk)
        begin
        if (inc == 1)
                d_out <= d_out + 16'd1;
        if (w_en == 1)
                d_out <= d_in;
        if (alu2ac == 1)
                d_out <= alu_out;
        end

endmodule
```

9.2.6 GPR

```
module GPR(
        input clk,
        input w_en,
        input [15:0] d_in,
        output reg [15:0] d_out
        );

always @(posedge clk)
        begin
        if (w_en == 1)
                d_out <= d_in;
        end
endmodule
```

9.2.7 GPR-INC

```
module GPR_INC(
        input clk,
        input w_en,
        input [15:0] d_in,
        output reg [15:0] d_out,
        input inc
        );

initial begin
        d_out<= 16'd0;
        end

always @(posedge clk)
```

```
        begin
                if (w_en == 1)
                        d_out <= d_in;
                if (inc == 1)
                        d_out <= d_out + 16'd1;
        end

endmodule
```

9.2.8 BUS

```
module BUS(
        input clk,
        input [3:0] r_en,
        input [15:0] R1,
        input [15:0] R2,
        input [15:0] R3,
        input [15:0] R4,
        input [15:0] R5,
        input [15:0] R,
        input [15:0] DAR,
        input [15:0] IR,
        input [15:0] PC,
        input [15:0] AC,
        input [7:0] DRAM,
        input [15:0] IRAM,
        output reg [15:0] out ) ;


always @(*)
        begin
                case(r_en)
                        4'd1: out         = PC;
                        4'd2: out         = DAR;
                        4'd4: out         = IR;
                        4'd5: out         = AC;
                        4'd6: out         = R;
                        4'd7: out         = R1;
                        4'd8: out         = R2;
                        4'd9: out         = R3;
                        4'd10: out         = R4;
                        4'd11: out         = R5;
                        4'd12: out         = DRAM + 16'd0;
                        4'd13: out         = IRAM ;
                        default: out= 16'd0;
                endcase
        end
endmodule
```

9.2.9 PROCESSOR-TOP

```
module processortop(
        input clk,
        input [7:0] dram_out,
        input [15:0] iram_out,
        input [1:0] status,
        output [15:0] pc_out,
        output [15:0] dar_out,
        output [15:0]bus_out,
        output reg dram_en,
        output reg iram_en,
        output finish
        );



wire [15:0] r_out;
wire [15:0] r1_out;
wire [15:0] r2_out;
wire [15:0] r3_out;
wire [15:0] r4_out;
wire [15:0] r5_out;
wire [15:0] ac_out;
wire [15:0] ir_out;
wire [15:0] w_en;
wire [3:0] r_en;
wire [15:0] inc;
wire [2:0] alu_op;
wire [15:0] alu_out;
wire [15:0] z ;


always @ (posedge clk)
        if (status == 2'b01) begin
                dram_en <= w_en[12];
                iram_en <= w_en[13];
        end
```

```
GPR R(.clk(clk), .w_en(w_en[6]),.d_in(bus_out),.d_out(r_out));

GPR_INC R1(.clk(clk), .w_en(w_en[7]),.d_in(bus_out),.d_out(r1_out),
.inc(inc[4]));

GPR_INC R2(.clk(clk), .w_en(w_en[8]),.d_in(bus_out),.d_out(r2_out),
.inc(inc[5]));

GPR_INC R3(.clk(clk), .w_en(w_en[9]),.d_in(bus_out),.d_out(r3_out),
.inc(inc[6]));

GPR R4(.clk(clk), .w_en(w_en[10]),.d_in(bus_out),.d_out(r4_out));

GPR R5(.clk(clk), .w_en(w_en[11]),.d_in(bus_out),.d_out(r5_out));

GPR_INC DAR(.clk(clk), .w_en(w_en[2]),.d_in(bus_out),.d_out(dar_out),
.inc(inc[3]));

GPR IR(.clk(clk), .w_en(w_en [4]),.d_in(bus_out),.d_out(ir_out));

BUS BUS(.R1(r1_out),.R2(r2_out),.R3(r3_out),.R4(r4_out),.R5(r5_out),
.R(r_out),.DAR(dar_out),.IR(ir_out),.PC(pc_out),.AC(ac_out),.DRAM(dram_out),
.IRAM(iram_out),.out(bus_out),.r_en(r_en),.clk(clk));

AC AC(.clk(clk), .w_en(w_en[5]),.d_in(bus_out),.d_out(ac_out),
.alu_out(alu_out),.alu2ac(w_en[14]),.inc(inc[2]));

GPR_INC PC(.clk(clk), .w_en(w_en[1]),.d_in(bus_out),.d_out(pc_out),
.inc(inc[1]));

ALU ALU(.clk(clk),.A(r_out),.B(ac_out),.op(alu_op),.out(alu_out),.z(z));

CU CU(.clk(clk),.z(z),.instruction(ir_out),.to_alu(alu_op),
.w_en(w_en),.r_en(r_en),.inc(inc),.finish(finish),.status(status));

endmodule
```

9.2.10 CLOCK-DIVIDER

```
module clockdiv(
        input in,
        output reg out = 1'b0
        );

reg counter = 1'b0;

always @ (posedge in)
        begin
                counter <= counter + 1'b1;
                if(counter == 1'b1)
                        out <= ~out;
        end
endmodule
```

9.2.11 TX

```
module async_transmitter(
        input clk,
        input TxD_start,
        input [15:0] TxD_data,
        output TxD,
        output TxD_busy
);

/* Assert TxD_start for (at least) one clock cycle to start transmission of
TxD_data. TxD_data is latched so that it doesn't have to stay valid while
it is being sent*/

parameter ClkFrequency = 25000000;        // 25MHz
parameter Baud = 115200;

generate
if(ClkFrequency<Baud*8 && (ClkFrequency % Baud!=0))
        ASSERTION_ERROR
        PARAMETER_OUT_OF_RANGE("Frequency incompatible with requested Baud rate");
endgenerate

////////////////////////////////
`ifdef SIMULATION
wire BitTick = 1'b1;  // output one bit per clock cycle
`else
wire BitTick;
BaudTickGen #(ClkFrequency, Baud) tickgen(.clk(clk), .enable(TxD_busy),
.tick(BitTick));
`endif

reg [3:0] TxD_state = 0;
```

```
wire TxD_ready = (TxD_state==0);
assign TxD_busy = ~TxD_ready;

reg [7:0] TxD_shift = 0;
always @(posedge clk)
begin
        if(TxD_ready & TxD_start)
                TxD_shift <= TxD_data;
        else
        if(TxD_state[3] & BitTick)
                TxD_shift <= (TxD_shift >> 1);

        case(TxD_state)
                4'b0000: if(TxD_start) TxD_state <= 4'b0100;
                4'b0100: if(BitTick) TxD_state <= 4'b1000;  // start bit
                4'b1000: if(BitTick) TxD_state <= 4'b1001;  // bit 0
                4'b1001: if(BitTick) TxD_state <= 4'b1010;  // bit 1
                4'b1010: if(BitTick) TxD_state <= 4'b1011;  // bit 2
                4'b1011: if(BitTick) TxD_state <= 4'b1100;  // bit 3
                4'b1100: if(BitTick) TxD_state <= 4'b1101;  // bit 4
                4'b1101: if(BitTick) TxD_state <= 4'b1110;  // bit 5
                4'b1110: if(BitTick) TxD_state <= 4'b1111;  // bit 6
                4'b1111: if(BitTick) TxD_state <= 4'b0010;  // bit 7
                4'b0010: if(BitTick) TxD_state <= 4'b0011;  // stop1
                4'b0011: if(BitTick) TxD_state <= 4'b0000;  // stop2
                default: if(BitTick) TxD_state <= 4'b0000;
        endcase
end

assign TxD = (TxD_state<4) | (TxD_state[3] & TxD_shift[0]);
// put together the start, data and stop bits
endmodule
```

9.2.12 RX

```
module async_receiver(
        input clk,
        input RxD,
        output reg RxD_data_ready = 0,
        output reg [15:0] RxD_data = 0,
        // data received,valid only(for 1 clock cycle)when RxD_data_ready is
        //asserted

        // We also detect if a gap occurs in the received stream of characters
        // That can be useful if multiple characters are sent in burst
        //  so that multiple characters can be treated as a "packet"
        output RxD_idle,  // asserted when no data has been received for a while
        output reg RxD_endofpacket = 0  // asserted for one clock cycle when a
                //packet has been detected (i.e. RxD_idle is going high)
);

parameter ClkFrequency = 25000000; // 25MHz
parameter Baud = 115200;

parameter Oversampling = 8;  // needs to be a power of 2
// we oversample the RxD line at a fixed rate to capture each RxD data bit at the
//"right" time
// 8 times oversampling by default, use 16 for higher quality reception

generate
if(ClkFrequency<Baud*Oversampling)
        ASSERTION_ERROR
        PARAMETER_OUT_OF_RANGE("Frequency too low for Baudrate and oversampling");
if(Oversampling<8 || ((Oversampling & (Oversampling-1))!=0))
        ASSERTION_ERROR
        PARAMETER_OUT_OF_RANGE("Invalid oversampling value");
endgenerate

////////////////////////////////
reg [3:0] RxD_state = 0;

`ifdef SIMULATION
wire RxD_bit = RxD;
wire sampleNow = 1'b1;  // receive one bit per clock cycle

`else
wire OversamplingTick;
BaudTickGen #(ClkFrequency, Baud, Oversampling) tickgen(.clk(clk), .enable(1'b1),
 .tick(OversamplingTick));

// synchronize RxD to our clk domain
reg [1:0] RxD_sync = 2'b11;
always @(posedge clk) if(OversamplingTick) RxD_sync <= {RxD_sync[0], RxD};

// and filter it
reg [1:0] Filter_cnt = 2'b11;
reg RxD_bit = 1'b1;

always @(posedge clk)
if(OversamplingTick)
```

```
begin
        if(RxD_sync[1]==1'b1 && Filter_cnt!=2'b11) Filter_cnt <= Filter_cnt + 1'd1;
        else
        if(RxD_sync[1]==1'b0 && Filter_cnt!=2'b00) Filter_cnt <= Filter_cnt - 1'd1;

        if(Filter_cnt==2'b11) RxD_bit <= 1'b1;
        else
        if(Filter_cnt==2'b00) RxD_bit <= 1'b0;
end

// and decide when is the good time to sample the RxD line
function integer log2(input integer v); begin log2=0; while(v>>log2)
log2=log2+1; end endfunction
localparam l2o = log2(Oversampling);
reg [l2o-2:0] OversamplingCnt = 0;
always @(posedge clk) if(OversamplingTick)
OversamplingCnt <= (RxD_state==0) ? 1'd0 : OversamplingCnt + 1'd1;
wire sampleNow = OversamplingTick && (OversamplingCnt==Oversampling/2-1);
`endif

// now we can accumulate the RxD bits in a shift-register
always @(posedge clk)
case(RxD_state)
        4'b0000: if(~RxD_bit) RxD_state <= `ifdef SIMULATION 4'b1000
                                `else 4'b0001 `endif;  // start bit found?
        4'b0001: if(sampleNow) RxD_state <= 4'b1000;  // sync start bit to sampleNow
        4'b1000: if(sampleNow) RxD_state <= 4'b1001;  // bit 0
        4'b1001: if(sampleNow) RxD_state <= 4'b1010;  // bit 1
        4'b1010: if(sampleNow) RxD_state <= 4'b1011;  // bit 2
        4'b1011: if(sampleNow) RxD_state <= 4'b1100;  // bit 3
        4'b1100: if(sampleNow) RxD_state <= 4'b1101;  // bit 4
        4'b1101: if(sampleNow) RxD_state <= 4'b1110;  // bit 5
        4'b1110: if(sampleNow) RxD_state <= 4'b1111;  // bit 6
        4'b1111: if(sampleNow) RxD_state <= 4'b0010;  // bit 7
        4'b0010: if(sampleNow) RxD_state <= 4'b0000;  // stop bit
        default: RxD_state <= 4'b0000;
endcase

always @(posedge clk)
if(sampleNow && RxD_state[3]) RxD_data <= {RxD_bit, RxD_data[15:1]};

//reg RxD_data_error = 0;
always @(posedge clk)
begin
        RxD_data_ready <= (sampleNow && RxD_state==4'b0010 && RxD_bit);
        // make sure a stop bit is received
        //RxD_data_error <= (sampleNow && RxD_state==4'b0010 && ~RxD_bit);
        // error if a stop bit is not received
end

`ifdef SIMULATION
assign RxD_idle = 0;
`else
reg [l2o+1:0] GapCnt = 0;
always @(posedge clk) if (RxD_state!=0) GapCnt<=0;
else if(OversamplingTick & ~GapCnt[log2(Oversampling)+1])
GapCnt <= GapCnt + 1'h1;
assign RxD_idle = GapCnt[l2o+1];
always @(posedge clk)
RxD_endofpacket <= OversamplingTick & ~GapCnt[l2o+1] & &GapCnt[l2o:0];
`endif

endmodule
```

9.2.13 BAUD-GENERATOR

```
module BaudTickGen(
        input clk, enable,
        output tick  // generate a tick at the specified baud rate * oversampling
);
parameter ClkFrequency = 12500000;
parameter Baud = 9600;
parameter Oversampling = 1;

function integer log2(input integer v); begin log2=0; while(v>>log2)
log2=log2+1; end endfunction
localparam AccWidth = log2(ClkFrequency/Baud)+8;
// +/- 2% max timing error over a byte
reg [AccWidth:0] Acc = 0;
localparam ShiftLimiter = log2(Baud*Oversampling >> (31-AccWidth));
// this makes sure Inc calculation doesn't overflow
localparam Inc = ((Baud*Oversampling << (AccWidth-ShiftLimiter))
+(ClkFrequency>>(ShiftLimiter+1)))/(ClkFrequency>>ShiftLimiter);
always @(posedge clk) if(enable) Acc <= Acc[AccWidth-1:0] + Inc[AccWidth:0];
 else Acc <= Inc[AccWidth:0];
assign tick = Acc[AccWidth];
endmodule
```

9.2.14 UART-TOP

```
module uart_top(
        input clk,
        input [1:0] status,
        input rxd,
        input [7:0] txd_data,
        output reg rx_done = 0,
        output reg tx_done = 0,
        output [15:0] rxd_data,
        output reg [15:0] addr_com,
        output txd,
        output reg en_com);

reg [3:0] PS = 4'b0000;
reg [3:0] NS = 4'b0000;

reg tx_en;
wire tx_busy;
wire rx_ready;

reg [15:0] tx_adr        =16'b0;
reg [15:0] tx_adrnext        =16'b0;
reg [15:0] rx_adr        =16'b0;
reg [15:0] rx_adrnext        =16'b0;

parameter IDLE        = 4'b0000;
parameter TX_SEND        = 4'b0001;
parameter TX_WAIT        = 4'b0010;
parameter TX_CHANGE        = 4'b0011;
parameter TX_FINISH        = 4'b0100;
parameter RX_BEGIN        = 4'b0101;
parameter RX_DATA        = 4'b0110;
parameter RX_CHANGE        = 4'b0111;
parameter RX_FINISH        = 4'b1000;

always @(posedge clock)
        begin
                tx_adr <= tx_adrnext;
                rx_adr <= rx_adrnext;
        end

always @(posedge clock) begin
        case(NS)
                IDLE: begin
                        tx_en = 0;
                        addr_com <= 16'b0;
                        en_com <=0;
                        tx_adrnext <= 16'b0;
                        rx_adrnext <= 16'b0;

                        if (status == 2'b10)
                                NS <= TX_SEND;
                        else if (status == 2'b00)
                                NS<= RX_BEGIN;
                        else
                                NS<= IDLE;
                end

                TX_SEND:begin
                        tx_en = 1;
                        rx_done = 0;
                        tx_done = 0;
                        addr_com <= tx_adr;
                        en_com <=0;
                        NS <= TX_WAIT;
                end

                TX_WAIT:begin
                        if (~tx_busy) begin
                                NS <= TX_CHANGE;
                        end
                end

                TX_CHANGE:begin
                        tx_en <=0;
                        if (tx_adr > 16128)begin
                                tx_done <= 1;
                                NS<= TX_FINISH;
                                tx_adrnext <= tx_adr;
                                rx_adrnext <= rx_adr;
                        end
                        else begin
                                tx_done <= 0;
                                NS<=TX_SEND;
                                tx_adrnext <= tx_adr + 16'b1;
                                rx_adrnext <= rx_adr;
                        end
                end
```

```
TX_FINISH: begin
        tx_en = 0;
        rx_done = 1;
        tx_done = 1;
        addr_com <= 16'b0;
        en_com <=0;
        tx_adrnext <= 16'b0;
        rx_adrnext <= 16'b0;
        NS<=TX_FINISH;
end

RX_BEGIN: begin
        tx_en = 0;
        rx_done = 0;
        tx_done = 0;
        addr_com <= 16'b0;
        en_com <=0;
        tx_adrnext <= tx_adr;
        rx_adrnext <= rx_adr;
        NS<=RX_DATA;
end


RX_DATA:begin
        tx_en = 0;
        rx_done = 0;
        tx_done = 0;
        addr_com <= rx_adr;
        if (rx_ready) begin
                en_com <=1;
                if (rx_adr == 65535) begin
                        rx_done <= 1;
                        NS<= RX_FINISH;
                end
                else begin
                        NS <= RX_CHANGE;
                end
        end
        else begin
                NS <= RX_DATA;
        end
end

RX_CHANGE: begin
```

```
                        rx_adrnext <= rx_adr + 16'b1;
                        tx_adrnext <= tx_adr;
                        rx_done <= 0;
                        NS<=RX_DATA;
                end

                RX_FINISH: begin
                        tx_en = 0;
                        rx_done = 1;
                        tx_done = 0;
                        addr_com <= 16'b0;
                        en_com <=0;
                        tx_adrnext <= 16'b0;
                        rx_adrnext <= 16'b0;
                        NS<=IDLE;
                end

        endcase

end

async_transmitter #(12500000,9600)
tx(.clk(clock),.TxD_start(tx_en),.TxD_data(txd_data),.TxD(txd),.TxD_busy(tx_busy));

async_receiver #(12500000,9600)
rx(.clk(clock),.RxD(rxd),.RxD_data_ready(rx_ready),.RxD_data(rxd_data));

endmodule
```

9.2.15 MAIN STATE-MACHINE

```
module main_fsm(
        input clk,
        input rx_done,
        input process_done,
        input tx_done,
        input start_process,
        input start_tx,
        output reg [1:0] status
        );

reg [1:0] PS =          2'b00;
reg [1:0] NS =          2'b00;
```

```
parameter RECEIVE = 2'b00;
parameter PROCESS = 2'b01;
parameter TRANSMIT = 2'b10;
parameter FINISH = 2'b11;

always @(posedge clk)
                PS <= NS;

always @(*)
        case(PS)
                RECEIVE: begin
                        status = 2'b00;
                        if (rx_done && !process_done && !tx_done)
                                NS=PROCESS;
                        else
                                NS=RECEIVE;
                end

                PROCESS: begin
                        status = 2'b01;
                        if (rx_done && process_done && !tx_done && start_tx)
                                NS=TRANSMIT;
                        else
                                NS=PROCESS;
                end

                TRANSMIT: begin
                        status = 2'b10;
                        if (rx_done && process_done && tx_done)
                                NS=FINISH;
                        else
                                NS=TRANSMIT;
                end

                FINISH: begin
                        status = 2'b11;
                        NS=FINISH;
                end

        endcase
endmodule
```

9.2.16 COMMAND CONTROL

```
module command_control(
        input clk,
        input [1:0] status,
        input [15:0] bus,
        input dram_en,
        input [15:0] dar_out,
        input [15:0] rxd_data,
        input en_com,
        input [15:0] addr_com,
        output reg [15:0] d_in,
        output reg pcwrite_en,
        output reg [15:0] d_adr,
        output reg [7:0] txd_data);

always @(posedge clk)
        case (status)
                2'b00:
                begin
                d_in <= rxd_data;
                pcwrite_en <= en_com;
                d_adr <= addr_com;
                end

                2'b01:
                begin
                d_in <= bus;
                pcwrite_en <=dram_en;
                d_adr <= dar_out;
                end

                2'b10:
                begin
                txd_data <= bus[7:0];
                pcwrite_en <= en_com;
                d_adr <= addr_com;
                end
        endcase
endmodule
```

9.2.17 TOP-MODULE

```
module top(input wire rxd,
        input wire sys_clock,
        input wire start_process,
        input wire start_transmit,
        output wire txd
        );


wire        clock, dram_en, iram_en, uart_en, dwrite_en, inswrite_en,
                finish_rx, finish_process, finish_tx;
wire [1:0]        status;
wire [7:0]        dram_out, din_uart;
wire [15:0]        iram_out, bus_out, dar_out, pc_out, dout_uart, uart_addr, din,
                        data_addr, ins_addr;

reg                 start_pr, start_tx;
reg [9:0]        buffer_process, buffer_tx         = 0;

clockdiv newclock(
        .in(sys_clock),
        .out(clock));

IRAM instr_memory(.clk(clock),
        .w_en(iram_en),
        .address(pc_out),
        .instr_out(iram_out),
        .instr_in(bus_out));

DRAM data_memory(.clk(clock),
        .w_en(dwrite_en),
        .address(data_addr),
        .d_in(din),
        .d_out(dram_out));

processortop cpu(.clk(clock),
        .dram_out(dram_out),
        .iram_out(iram_out),
        .dram_en(dram_en),
        .iram_en(iram_en),
        .pc_out(pc_out),
        .to_dar(dar_out),
        .status(status),
        .to_bus(bus_out),
        .finish(finish_process));

command_control cmd_ctrl(.clk(clock),
        .status(status),
        .bus(bus_out),
        .dram_en(dram_en),
        .dar_out(dar_out),
        .rxd_data(dout_uart),
        .en_com(uart_en),
        .addr_com(uart_addr),
        .d_in(din),
        .pcwrite_en(dwrite_en),
        .d_adr(data_addr),
        .txd_data(din_uart));

uart_top communicate1(.clk(clock),
        .status(status),
        .rx_done(finish_rx),
        .tx_done(finish_tx),
        .txd_data(din_uart),
        .rxd_data(dout_uart),
        .addr_com(uart_addr),
        .txd(txd),
        .rxd(rxd),
        .en_com(uart_en));

main_fsm main(.clk(clock),
        .rx_done(finish_rx),
        .process_done(finish_process),
        .tx_done(finish_tx),
        .start_process(start_pr),
        .start_tx(start_tx),
        .status(status)
        );

always @(posedge clock)
        begin
        if (start_process)
                begin
                if (buffer_process == 1023)
                        begin
                        buffer_process <= buffer_process;
                        start_pr <=1;
                        end
                else
                        begin
                        buffer_process <= buffer_process + 1;
                        start_pr <=0;
                        end
                end
        else
                begin
                buffer_process <= 0;
                start_pr <= 0;
                end
        end

always @(posedge clock)
        begin
        if (start_transmit)
                begin
                if (buffer_tx == 1023)
                        begin
                        buffer_tx <= buffer_tx;
                        start_tx <=1;
                        end
                else
                        begin
                        buffer_tx <= buffer_tx + 1;
                        start_tx <=0;
                        end
                end
        else
                begin
                buffer_tx <= 0;
                start_tx<= 0;
                end
        end


endmodule
```