Report on Processor Design and Implementation for the Down-Sampling of an Image

Department of Electronic and Telecommunication Engineering
University of Moratuwa

140072H	B.H.A.R.A. Bangamuarachchi
140266G	A.M.W. Jayawardane
140576G	M.J. Seneviratne
140635M	K.G. Vidanapathirana

This is submitted as a partial fulfillment for the module
EN3030: Circuits & Systems Design

Contents

1. Introduction

This document describes in detail the approach taken to solve the following problem statement. It highlights the unique features present in the processor, justifies the design decisions taken in order to implement those features and finally tests the algorithm and the processor performance against expected output.

1.1 Problem Statement

Design a processor capable of down-sampling an input image in the ratio 2:1 and implement the processor using Verilog HDL on an FPGA.

1.2 Approach

- Image is input in to the RAM as a bit stream using UART protocol.
- Instructions are loaded on to the processor (programme memory)
- The algorithm filters the image (using a 3x3 Gaussian kernel) and down-samples at the same time (in order to avoid unnecessary calculations involved in smoothing out pixels that won't be in the output image.). Zero padding is used when necessary to avoid black lines on edges the kernel can't reach.

  This algorithm is capable of down-sampling images of variable size with a maximum image size of 512x512. The image can also be rectangular and have odd height or width.

- Final result is stored in the output image RAM and will be obtained as a bit stream through UART and later resized to form a 2D image

1.3 Additional Features

- Zero 'Sum of Squared Difference' (SSD) between the Matlab execution and Processor(FPGA) execution of the algorithm.
- Cache Implementation.
- Maximum Image Size of 512*512 pixels.
- Variable Image Dimensions. Image can be square or rectangular. Dimensions for both height and width can be either odd or even.

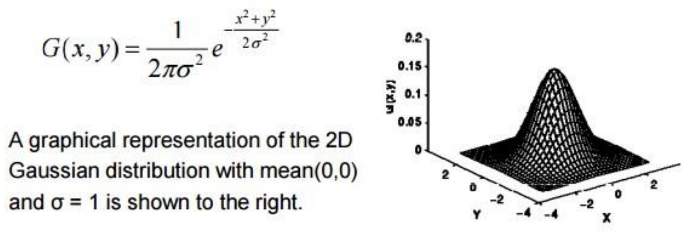
2. Algorithm

2.1 Filtering and down sampling algorithm

As specified in the problem statement the objective of the algorithm is to down sample a given image to half of its original size. When down sampling a given image the effect of high frequency components increases. Hence to reduce the effect fist the image should be low pass filtered to remove high frequency content. Then the down sampling can be carried out.

The Filter

When deciding the filter there we few options. Median filter, mean filter and Gaussian filter. Mean filter doesn't have a have a good cut off performance but implementation is the easiest. Median filter require consumes lot of hardware resources because there is a sorting process. Gaussian filtering only require addition multiplication and division.

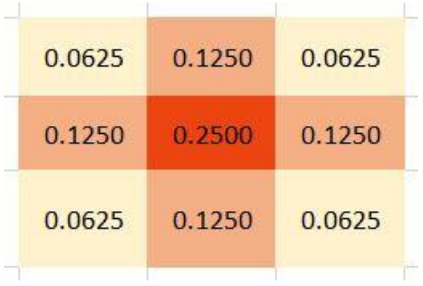


$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

A graphical representation of the 2D Gaussian distribution with mean(0,0) and σ = 1 is shown to the right.

Figure 2.1: 2D Gaussian plot [1]

Since the processor has only basic arithmetic operations the filter was modified to suite the processor specifications



0.0625	0.1250	0.0625
0.1250	0.2500	0.1250
0.0625	0.1250	0.0625

Figure 2.2: Simplified Gaussian Kernel

To simplify the process further and to remove flatting point multiplication the filter was modified as follows.

1	2	1
2	4	2
1	2	1

Figure 2.3: Gaussian Kernel used in processor

Filtering and the Down sampling Process

For each pixel the kernel will fit its center with the pixel and calculate average value of that image section. Then the kernel will traverse the image as shown in the figure 2.4. After finishing the downward pixels, kernel will move to the next column and store the average value as shown in the figure 2.4. Likewise the filtered image can be stored in the RAM in sequential manner.

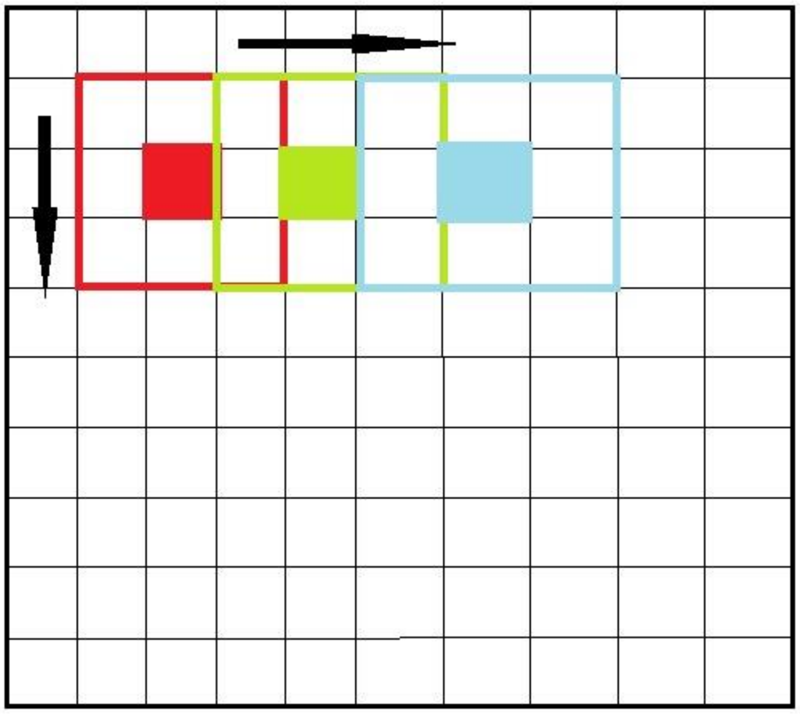


Figure 2.4: Filtering Process

As the kernel values only include powers of two in the processor the multiplication is done using left shifting and division is also done using right shifting hence the equivalent arithmetic operation is floor division.

As specified in the problem statement the image should be down sampled in to half of its original size. To optimize the processes rather than first filtering then down sampling, both were done at the same time. As shown in the figure 2.5 only the relevant down sampled pixel values were calculated.

In order to down sample the image into half the size (height and width) of the original image, this algorithm will take one value per four pixels from the filtered image and store it in the memory to get the output. Values taken from 9x9 filtered image is shown in the Figure 2.5.

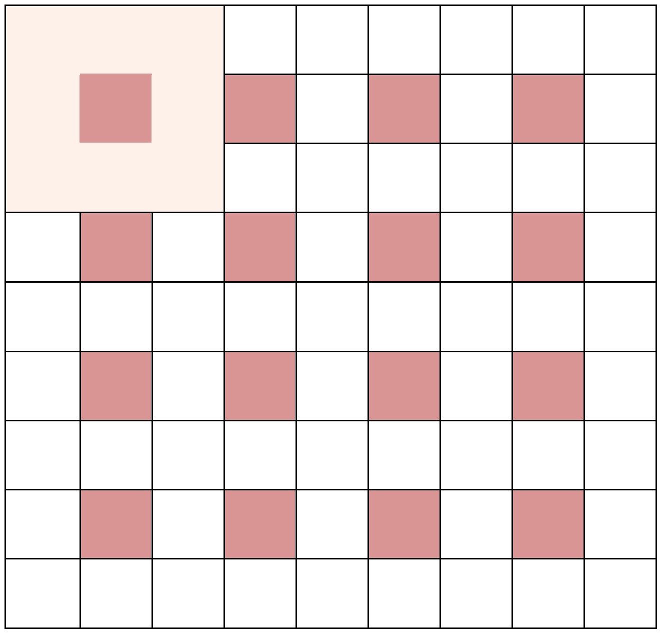


Figure 2.4: Selected pixels

Basic Pseudo code

```
for i = 1 : Height
 for j = 1 : Width
        sum = 0
        if ( (I, j) mod (2) == (0,0))
         sum = sum + image(I,j)*4
         sum =sum + image (I,j+1)*2
         sum = sum + image(I,j-1)*2
         sum =sum + image (I+1,j)*2
         sum =sum + image (I-1,j)*2
         sum =sum + image (I-1,j-1)
         sum =sum + image (I-1,j+1)
         sum =sum + image (I+1,j-1)
         sum =sum + image (I+1,j+1)
         filterd_matrix (i/2, j/2) = sum/16
        end
 end
end
```

Boundary cases

The implemented algorithm was developed for variable image sizes where the height and the width of the image is passed to the processor. When implementing the variable image size algorithm the main problem faced was the boundary cases. As shown in figure **** For odd sized image there were no effect of boundary cases but for even sized images the lack of data at the edges comes in to play.

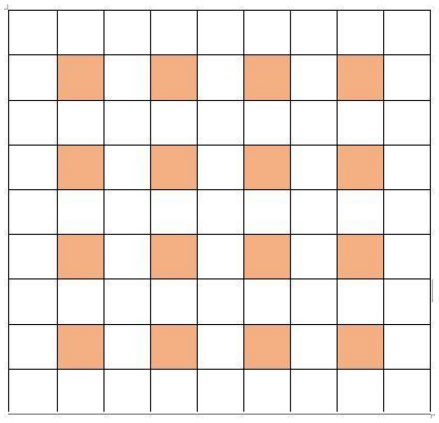


Figure 2.5: Selected Pixels in an image of size 9x9

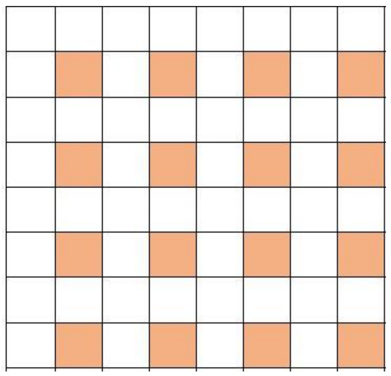


Figure 2.6: Selected Pixels in an image of size 8x8

To cater to the problem Zero padding was introduced.

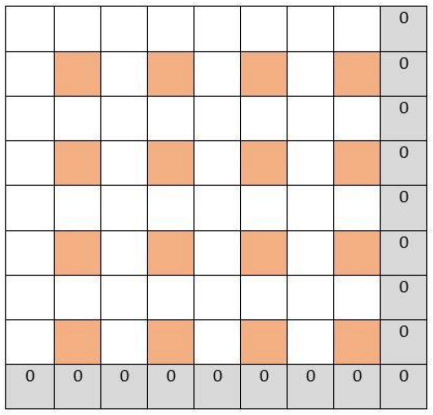


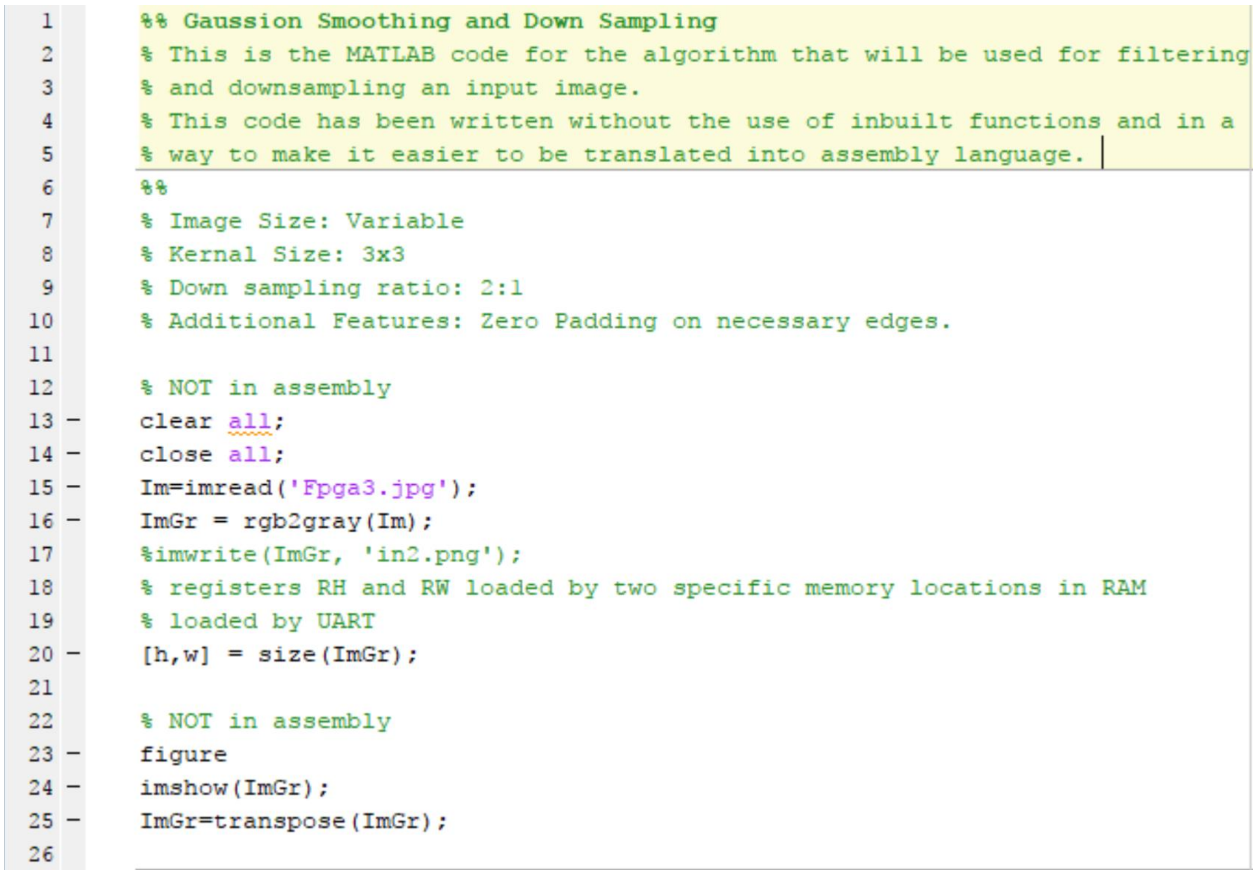
Figure 2.7: After Zero padding

2.2 Matlab Code

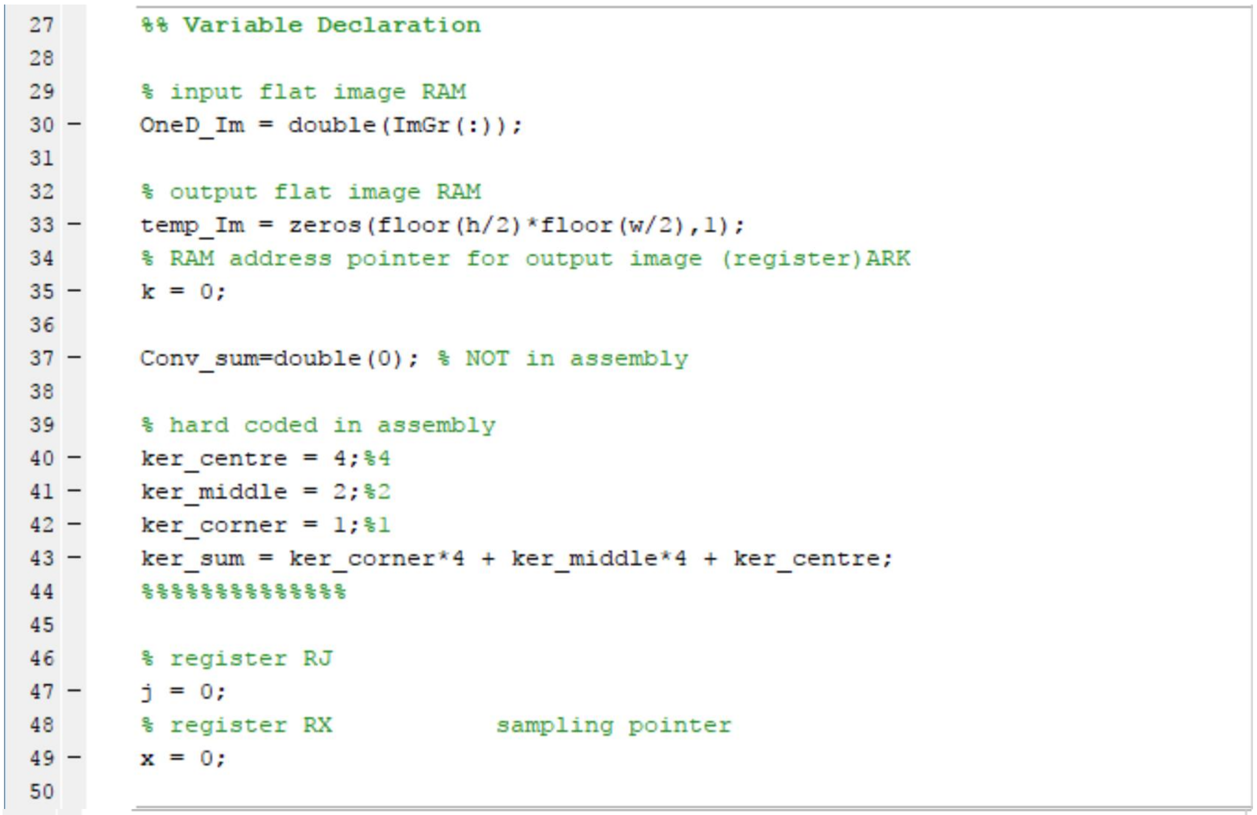
The algorithm was designed with the following in mind:

- should be as fast and as efficient as possible. Unnecessary calculations are avoided. Due to this, the Gaussian Smoothing and Down-Sampling of the image are carried out simultaneously. This increases efficiency by:
  - Pixels that will not be in the final (down-sampled) image will not smoothed out. Hence reducing the amount of necessary calculations by 4 times.
  - There is no need to iterate through the 2D image array twice due to simultaneous Gaussian Smoothing and Down-Sampling. This reduces execution time by about half.
  - The algorithm should be capable of supporting all the additional features mentioned in the introduction above.

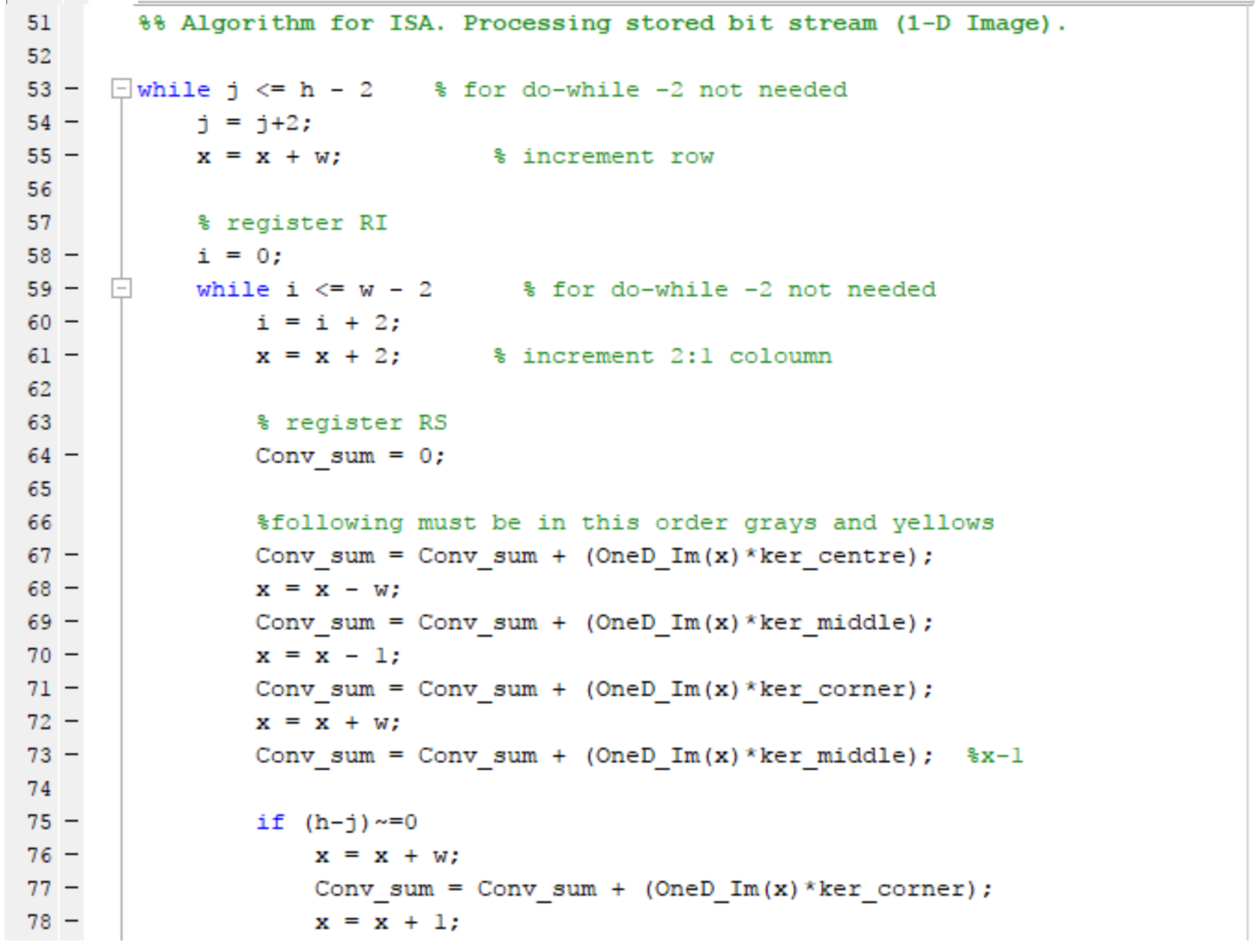
The Matlab implementation of the algorithm is given below:



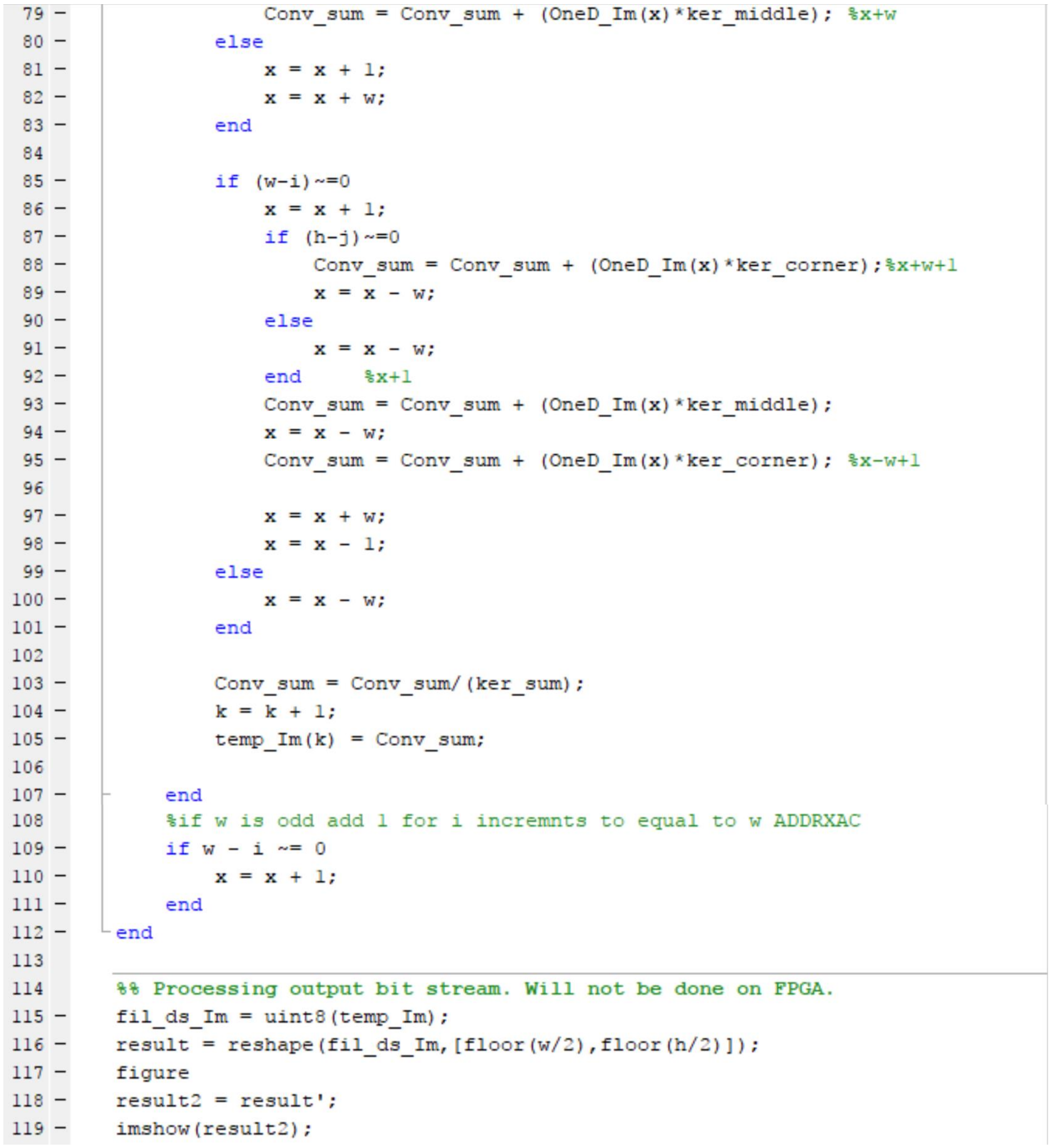
```
%% Gaussion Smoothing and Down Sampling
% This is the MATLAB code for the algorithm that will be used for filtering
% and downsampling an input image.
% This code has been written without the use of inbuilt functions and in a
% way to make it easier to be translated into assembly language.
%%
% Image Size: Variable
% Kernal Size: 3x3
% Down sampling ratio: 2:1
% Additional Features: Zero Padding on necessary edges.

% NOT in assembly
clear all;
close all;
Im=imread('Fpga3.jpg');
ImGr = rgb2gray(Im);
%imwrite(ImGr, 'in2.png');
% registers RH and RW loaded by two specific memory locations in RAM
% loaded by UART
[h,w] = size(ImGr);

% NOT in assembly
figure
imshow(ImGr);
ImGr=transpose(ImGr);

```



```
%% Variable Declaration

% input flat image RAM
OneD_Im = double(ImGr(:));

% output flat image RAM
temp_Im = zeros(floor(h/2)*floor(w/2),1);
% RAM address pointer for output image (register)ARK
k = 0;

Conv_sum=double(0); % NOT in assembly

% hard coded in assembly
ker_centre = 4;%4
ker_middle = 2;%2
ker_corner = 1;%1
ker_sum = ker_corner*4 + ker_middle*4 + ker_centre;
%%%%%%%%%%%%%%

% register RJ
j = 0;
% register RX           sampling pointer
x = 0;

```



```
%% Algorithm for ISA. Processing stored bit stream (1-D Image).

while j <= h - 2    % for do-while -2 not needed
    j = j+2;
    x = x + w;          % increment row

    % register RI
    i = 0;
    while i <= w - 2      % for do-while -2 not needed
        i = i + 2;
        x = x + 2;      % increment 2:1 coloumn

        % register RS
        Conv_sum = 0;

        %following must be in this order grays and yellows
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_centre);
        x = x - w;
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle);
        x = x - 1;
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner);
        x = x + w;
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle);  %x-1

        if (h-j)~=0
            x = x + w;
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner);
            x = x + 1;
```



```
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle); %x+w
        else
            x = x + 1;
            x = x + w;
        end

        if (w-i)~=0
            x = x + 1;
            if (h-j)~=0
                Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner);%x+w+1
                x = x - w;
            else
                x = x - w;
            end     %x+1
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle);
            x = x - w;
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner); %x-w+1

            x = x + w;
            x = x - 1;
        else
            x = x - w;
        end

        Conv_sum = Conv_sum/(ker_sum);
        k = k + 1;
        temp_Im(k) = Conv_sum;

    end
    %if w is odd add 1 for i incremnts to equal to w ADDRXAC
    if w - i ~= 0
        x = x + 1;
    end
end

%% Processing output bit stream. Will not be done on FPGA.
fil_ds_Im = uint8(temp_Im);
result = reshape(fil_ds_Im,[floor(w/2),floor(h/2)]);
figure
result2 = result';
imshow(result2);
```

2.2 Assembly Code

```
1.  CLAC        AC   ←    0
2.  CLRX        RX   ←    0        (sampling pointer)
3.  LDACRX      AC   ←    M_I[RX]
4.  MVACRH      RH   ←    AC
5.  INCRX_1     RX   ←    RX + 1
6.  LDACRX      AC   ←    M_I[RX]
7.  MVACRW      RW   ←    AC
8.  CLRK        RK   ←    0        (output img pointer)
9.  CLRJ        RJ   ←    0        (row pointer)
10. INCRJ       RJ   ←    RJ + 2   (start of row while loop)
11. INCRX_W     RX   ←    RX + RW
12. CLRI        RI   ←    0        (col pointer)
13. INCRI       RI   ←    RI + 2   (start of col while loop)
14. INCRX       RX   ←    RX + 1
15. INCRX
16. CLRS        RS   ←    0
17. LDACRX      AC   ←    M_I[RX]
18. SHFT2       AC   ←    AC*4
19. ADDSUM      RS   ←    RS + AC            (Matlab line 67)
20. DECRX_W     RX   ←    RX – RW
21. LDACRX      AC   ←    M_I[RX]
22. SHFT1       AC   ←    AC*2
23. ADDSUM      RS   ←    RS + AC
24. DECRX       RX   ←    RX – 1
25. LDACRX      AC   ←    M_I[RX]
26. ADDSUM      RS   ←    RS + AC
27. INCRX_W     RX   ←    RX + RW
28. LDACRX      AC   ←    M_I[RX]
29. SHFT1       AC   ←    AC*2
30. ADDSUM      RS   ←    RS + AC            (Matlab line 73)
31. SUBRHRJ     AC   ←    RH – RJ ( Z ← 1 (if AC = 0)or Z ← 0)
32. JMPZ [41]   (if Z = 1 : jump   else : continue)
33. INCRX_W     RX   ←    RX + RW
34. LDACRX      AC   ←    M_I[RX]
35. ADDSUM      RS   ←    RS + AC
36. INCRX       RX   ←    RX + 1
37. LDACRX      AC   ←    M_I[RX]
38. SHFT1       AC   ←    AC*2
39. ADDSUM      RS   ←    RS + AC
40. JUMP [43]   (Jump to given address)
41. INCRX       RX   ←    RX + 1
```

42. INCRX_W	RX	←	RX + RW	(Matlab line 82)
43. SUBRWRI	AC	←	RW – RI (Z ← 1 (if AC = 0) or Z ←0)	
44. JMPZ [62]		(if Z = 1 : jump else : continue)		
45. INCRX	RX	←	RX + 1	(Matlab line 86)
46. SUBRHRJ	AC	←	RH – RJ (Z ← 1 or Z ← 0)	
47. JMPZ [52]		(if Z = 1 : jump else : continue)		
48. LDACRX	AC	←	M_I[RX]	
49. ADDSUM	RS	←	RS + AC	
50. DECRX_W	RX	←	RX – RW	
51. JUMP [53]	(Jump to given address)			
52. DECRX_W	RX	←	RX – RW	
53. LDACRX	AC	←	M_I[RX]	
54. SHFT1	AC	←	AC*2	
55. ADDSUM	RS	←	RS + AC	(Matlab line 93)
56. DECRX_W	RX	←	RX – RW	
57. LDACRX	AC	←	M_I[RX]	
58. ADDSUM	RS	←	RS + AC	
59. INCRX_W	RX	←	RX + RW	
60. DECRX	RX	←	RX - 1	
61. JUMP [63]	(Jump to given address)			
62. DECRX_W	RX	←	RX - RW	
63. DIVRS	RS	←	RS/16	
64. INCRK	RK	←	RK + 1	
65. STRSRK	M_O[RK] ←		RS	
66. SUBRWRI	AC	←	RW – RI (Z ← 1 or Z ←0)	
67. JMPSP [12]	(if Z = 1 or AC = 1: continue else : jump) (end of col while loop)			
68. ADDRXAC	RX	←	RX + AC (adding 1 for odd, 0 for even)	
69. SUBRHRJ	AC	←	RH – RJ (Z ← 1 or Z ← 0)	
70. JMPSP [9]	(if Z = 1 or AC = 1: continue else : jump) (end of row while loop)			
71. END (Notify UART)				

3. Instruction Set Architecture

3.1 Instruction Set

Instruction	Operation
DATA TRANSFER	
CLAC	AC←0, Z←1
LDACRX	AC ← M_I[RX]
MVACRH	RH←AC
MVACRW	RW←AC
STRSRK	M_O[RK] ←RS
CLRX	RX←0
CLRS	RS←0
CLRI	RI←0
CLRJ	RJ←0
CLRK	RK←0
CONTROL	
JUMP [T]	GOTO Ƭ
JMPZ [T]	IF (Z=1) GOTO Ƭ, else continue
JMPSP [T]	IF (Z = 0 or AC = 1): continue, else: GOTO Ƭ
END	Notify UART
ALU	
ADDRXAC	RX←RX+AC
ADDSUM	RS←RS+AC
DIVRS	RS←RS/16
SUBRHRJ	AC←RH-RJ; (Z ← 1 (if AC = 0),else Z ← 0)
SUBRWRI	AC←RW-RI; (Z ← 1 (if AC = 0),else Z ← 0)
INRX	RX←RX+1
INRX_W	RX←RX+RW
DECRX	RX←RX-1
DECRX_W	RX←RX-RW
INRI	RI←RI+2
INRJ	RJ←RJ+2
INRK	RK←RK+1
SHFT1	AC←AC*2
SHFT2	AC←AC*4

3.2 Instruction Set Design Decisions

.

Data Transfer Instructions

- **LDACRX** (AC ← M_I[RX])

This is the only method of reading the input memory and getting image pixel values into the processor in order to be smoothed.

- **MVACRH** (RH←AC) & **MVACRW** (RW←AC)

These two instructions are used to transfer the image height and width values to the special purpose registers RH and RW.

- **CLAC** (AC←0), **CLRX** (RX←0), **CLRS** (RS←0), **CLRI** (RI←0), **CLRJ** (RJ←0), **CLRK** (RK←0)

These instructions are used to clear (make zero) the specific registers.

- **STRSRK** (M_O[RK] ←RS)

This is the only method for storing the processed data on output memory. It stores the value in RS register in the output memory location pointed to by the RK register.

Control Instructions

- **JUMP** [T] [GOTO T]

Used mainly to skip the next few lines of assembly code which are carried out under a different condition.

- **JMPZ** [T] [IF (Z=1) GOTO T, else continue]

Used in the loop of the Gaussian Kernel as it allows to check for a given condition.

- **JMPSP** [T] [IF (Z = 1 or AC = 1): continue, else: GOTO T]

Used in the two while loops two conditions are checked. This was necessary in order to support variable sized images of odd or even dimensions.

- **END** [Notify UART]

Notifies the UART state machine that the processing is complete.

Arithmetic Instructions

- **ADDRXAC** (RX←RX+AC)

Used to update the RX register which is the pointer to the input memory.

- **ADDSUM** (RS←RS+AC)

Used to update the RS register which stores the current sum of the gaussian kernel operations.

- **DIVRS** (RS←RS/16)

Used to calculate the final smoothed pixel value by dividing the sum of the kernel by 16.

- **SUBRHRJ** (AC←RH-RJ)

Used to decide whether all rows of the image have been processed.

- **SUBRWRI** (AC←RW-RI)

Used to decide whether all columns of the image have been processed.

- **INRI** (RI←RI+2), **INRJ** (RJ←RJ+2), **INRK** (RK←RK+1), **INRX** (RX←RX+1), **DECRX** (RX←RX-1)

Used to increment or decrement the value in the specified register by 1.

- **INRX_W** (RX←RX+RW), **DECRX_W** (RX←RX-RW)

Used to increment or decrement the value in the RX register by the width of the image. This is used when traversing through the pixels inside the kernel.

- **SHFT1** (AC←AC*2), **SHFT2** (AC←AC*4)

Used to multiply the value in the AC register by 2 or 4. Used mainly in the kernel calculations.

3.3 Data Path

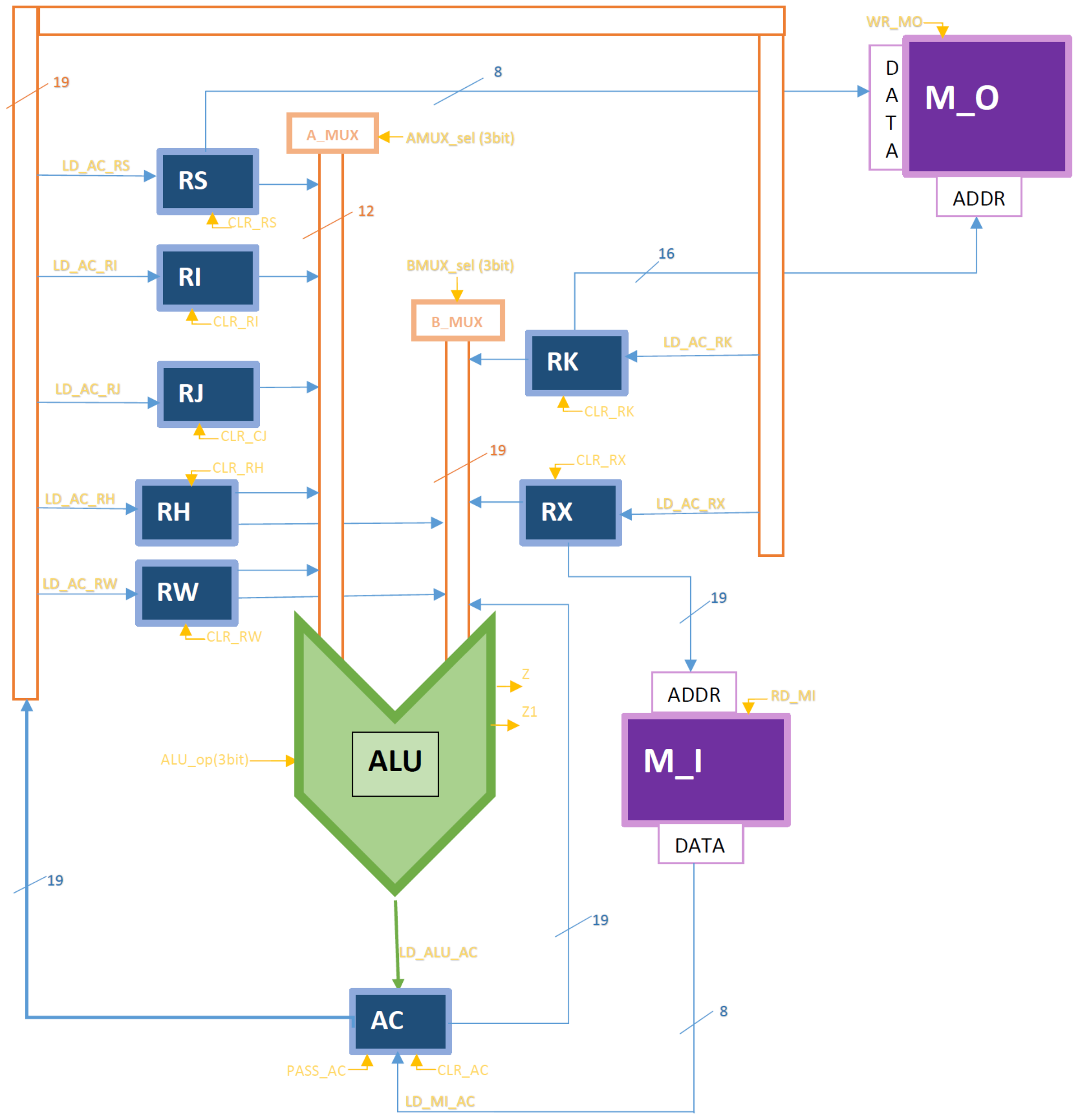


Figure 3.1: Datapath

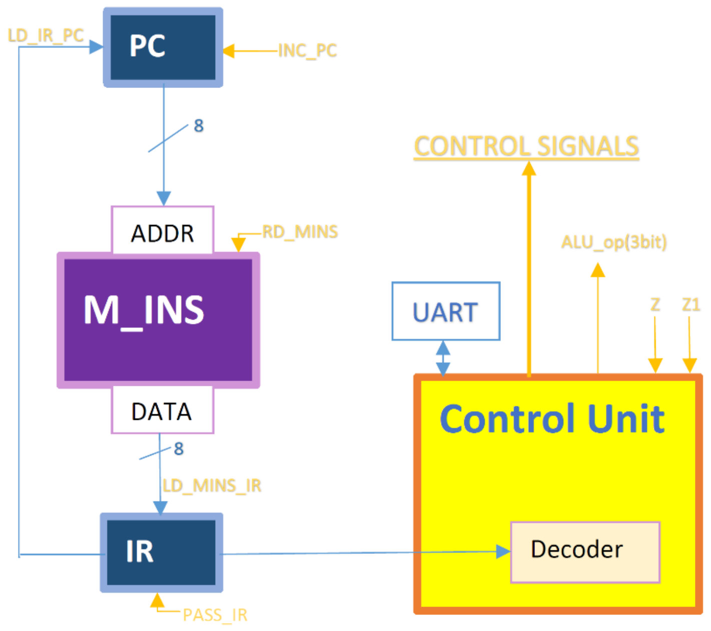


Figure 3.2: Control Unit

Registers and Memory

Register	**Code**	**Size** (bit)	**Purpose**
Program Counter	PC	8	Tracks Instructions.
Instruction Register	IR	8	Current Instruction.
Accumulator	AC	19	Output register of ALU.
Register X	RX	19	Input image current pixel.
Register S	RS	12	Store sum of convolution.
Register I	RI	12	Image column position.
Register J	RJ	12	Image row position.
Register K	RK	19	Output image current pixel.
Register H	RH	12	Store input image height.
Register W	RW	12	Store input image width.
MEMORY PARTS			
Instruction RAM	M_INS	2^8	Store program instructions
Input Image RAM	M_I	2^19	Store input image
Output Image RAM	M_O	2^17	Store output image

3.4 Data Path Design Decisions

Registers and Busses

- PC and IR are 8-bit and allows a program of 0-255 instructions to be stored. The Assembly code is 71 instructions and therefore it's within the range.

The processor supports a maximum image size of 512*512 pixels. Therefor the size of registers in the data path are determined by the maximum range of numerical data they should be able to store:

- AC and RX which store the pixel number (address) are 19-bit in order to addresses a 512*512 image. [512*512 => 9bit+9bit = 18bit => 19bit]
- Because of AC and RX, B-Bus (B-Mux) is 19-bit.
- RS is used to store the sum of the Gaussian Smoothing kernel. Therefore. It should be able to store values ranging from 0 to a maximum of 9(pixels) * 255(max 8-bit pixel value). 9 => 4-bit & 255 => 8-bit. So, RS is 4+8 = 12-bit.
- Because of RS, A-Bus (A-Mux) was made 12-bit. All other registers which are connected to A-Bus were also made 12-bit in order to avoid unnecessary complexity as 12-bit is more than enough for them.
- RW is connected to both A-Bus and B-Bus in order to support both SUBRWRI and INRX_W instructions. (RH only needs to be connected to B-Bus for the purpose of this algorithm but is connected to both busses to maintain symmetricity.)

Control Signals

Each register has two control signals to Clear and Load-From-AC.

AC register has 4 control signals as it has two input lines and a pass function.

ALU control signals are 3-bit as there are exactly 8 ALU operations.

3.5 Micro Instructions

Instruction	**Micro Instructions (Operations)**	**Control Signals**
FETCH	FETCH1	RD_MINS
	FETCH2 (IR← M_INS)	RD_MINS, LD_MINS_IR
	(PC←PC+1)	INC_PC
LDACRX (AC ← M_I[RX])	LDACRX1	RD_MI
	LDACRX2 (AC←M_I)	RD_MI, LD_MI_AC
MVACRH (RH←AC)	MVACRH (RH←AC)	LD_AC_RH
MVACRW (RW←AC)	MVACRW (RW←AC)	LD_AC_RW
STRSRK (M_O[RK] ←RS)	STRSRK1	WR_MO
	STRSRK2 (M_O←RS)	WR_MO
CLAC (AC←0)	CLAC (AC←0, Z = 0)	CLR_AC
CLRX (RX←0)	CLRX (RX←0)	CLR_RX
CLRS (RS←0)	CLRS (RS←0)	CLR_RS
CLRI (RI←0)	CLRI (RI←0)	CLR_RI
CLRJ (RJ←0)	CLRJ (RJ←0)	CLR_RJ
CLRK (RK←0)	CLRK (RK←0)	CLR_RK
ADDRXAC (RX←RX+AC)	ADDRXAC1 (RS←AC)	LD_AC_RS
	ADDRXAC2 (AC←RS+RX)	AMUX_RS
		BMUX_RX
	(Cbus←Abus + Bbus)	ALU_1
		LD_ALU_AC
	ADDRXAC3 (RX←AC)	LD_AC_RX
ADDSUM (RS←RS+AC)	ADDSUM1 (AC←RS+AC)	AMUX_RS
		BMUX_AC
	(Cbus←Abus + Bbus)	ALU_1
		LD_ALU_AC
	ADDSUM2 (RS←AC)	LD_AC_RS
DIVRS (RS←RS/16)	DIVRS1 (RS←RS>>>4)	AMUX_RS
	(Cbus←Abus>>>4)	ALU_2
		PASS_AC

		LD_AC_RS
SUBRHRJ (AC←RH-RJ)	SUBRHRJ (AC←RH-RJ)	AMUX_RJ
		BMUX_RH
	(Cbus← Bbus – Abus)	ALU_3
		LD_ALU_AC
SUBRWRI (AC←RW-RI)	SUBRWRI (AC←RW-RI)	AMUX_RI
		BMUX_RW
	(Cbus← Bbus – Abus)	ALU_3
		LD_ALU_AC
INRI (RI←RI+2)	INRI1	AMUX_RI
	(Cbus←Abus + 2)	ALU_4
		PASS_AC
	(RI←AC)	LD_AC_RI
INRJ (RJ←RJ+2)	INRJ1	AMUX_RJ
	(Cbus←Abus + 2)	ALU_4
		PASS_AC
	(RJ←AC)	LD_AC_RJ
INRK (RK←RK+1)	INRK	BMUX_RK
	(Cbus←Bbus + 1)	ALU_5
		PASS_AC
	(RK←AC)	LD_AC_RK
INRX (RX←RX+1)	INRX	BMUX_RX
	(Cbus←Bbus + 1)	ALU_5
		PASS_AC
		LD_AC_RX
DECRX (RX←RX-1)	DECRX	BMUX_RX
	(Cbus←Bbus - 1)	ALU_6
		PASS_AC
		LD_AC_RX
INRX_W (RX←RX+RW)	INRX_W	AMUX_RW
		BMUX_RX
	(Cbus←Abus + Bbus)	ALU_1
		PASS_AC
		LD_AC_RX

DECRX_W (RX←RX-RW)	DECRX_W	AMUX_RW
		BMUX_RX
	(Cbus← Bbus – Abus)	ALU_3
		PASS_AC
		LD_AC_RX
SHFT1 (AC←AC*2)	SHFT1	BMUX_AC
	(Cbus←Bbus << 1)	ALU_7
		LD_ALU_AC
SHFT2 (AC←AC*4)	SHFT2	BMUX_AC
	(Cbus←Bbus << 2)	ALU_8
		LD_ALU_AC
JUMP [Ƭ]	[GOTO Ƭ]	
	JUMP1	RD_MINS
	JUMP2 (IR← M_INS)	RD_MINS
	(PC←IR)	LD_IR_PC
JMPZ [Ƭ]	[IF (Z=1) GOTO Ƭ, else continue]	
(Z!=0)	JMPZ1	RD_MINS
	JMPZ2 (IR← M_INS)	RD_MINS
	(PC←IR)	LD_IR_PC
(Z=0)	JMPZN1 (PC← PC+1)	INC_PC
JMPSP [Ƭ]	[IF (Z = 1 or AC = 1): continue, else: GOTO Ƭ]	
(Z!=1 and AC!=1)	JMPSP1	RD_MINS
	JMPSP2 (IR← M_INS)	RD_MINS
	(PC←IR)	LD_IR_PC
(Z=1 or AC=1)	JMPSPN1 (PC← PC+1)	INC_PC
END	[Notify UART]	END

3.6 Micro-Sequencer

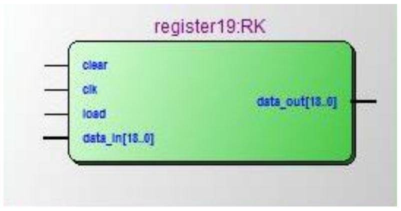
	(STATE) OPCODE	Microinstruction	Next	START	Z	Z1	RD_MINS	RD_MI	WR_MO	LD_MINS_IR	LD_MI_AC(0)	LD_ALU_AC(1)	LD_AC_RH(2)	LD_AC_RW(3)	LD_AC_RS(4)	LD_AC_RX(5)	LD_AC_RI(6)	LD_AC_RJ(7)	LD_AC_RK(8)	LD_IR_PC	INC_PC	CLR_AC(0)	CLR_RS(1)	CLR_RX(2)	CLR_RI(3)	CLR_RJ(4)	CLR_RK(5)	AMUX_RS(0)	AMUX_RJ(2)	AMUX_RI(1)	AMUX_RW(4)	BMUX_AC(4)	BMUX_RH(0)	BMUX_RX(3)	BMUX_RW(1)	BMUX_RK(2)	ALU_1(0)	ALU_2(1)	ALU_3(2)	ALU_4(3)	ALU_5(4)	ALU_6(5)	ALU_7(6)	ALU_8(7)	PASS_AC	PASS_IR	END	
	0	START	FETCH1	1																																												
	1	FETCH1	FETCH2				1																																									
	2	FETCH2	OPCODE				1			1												1																												
1	3	LDACRX1	LDACRX2					1																																								
	4	LDACRX2	FETCH1					1				1																																					
2	5	MVACRH1	FETCH1											1																																		
3	6	MVACRW1	FETCH1												1																																	
4	7	STRSRK1	STRSRK2						1																																								
	8	STRSRK2	FETCH1						1																																								
5	9	CLAC	FETCH1																			1																									
6	10	CLRX	FETCH1																					1																							
7	11	CLRS	FETCH1																				1																								
8	12	CLRI	FETCH1																						1																						
9	13	CLRJ	FETCH1																							1																					
10	14	CLRK	FETCH1																								1																				
11	15	ADDRXAC1	ADDRXAC2													1																																
	16	ADDRXAC2	FETCH1									1				1												1						1			1								1			
12	17	ADDSUM1	ADDSUM2									1																1				1					1											
	18	ADDSUM2	FETCH1													1																																
13	19	DIVRS1	FETCH1									1			1												1										1							1			
14	20	SUBRHRJ	FETCH1									1																	1				1						1									
15	21	SUBRWRI	FETCH1									1																		1				1				1									
16	22	INRI1	FETCH1									1					1													1										1					1			
17	23	INRJ1	FETCH1									1						1											1													1					1			
18	24	INRK	FETCH1									1							1																			1					1				1			
19	25	INRX	FETCH1									1				1																		1								1				1			
20	26	DECRX	FETCH1									1				1																		1									1			1			
21	27	INRX_W	FETCH1									1				1															1			1			1								1			
22	28	DECRX_W	FETCH1									1				1															1			1					1						1			
23	29	SHFT1	FETCH1									1																				1											1					
24	30	SHFT2	FETCH1									1																				1												1				
25	31	JUMP1	JUMP2				1																																									
	32	JUMP2	FETCH1				1			1										1																										1		
26	33	JMPZ1	JMPZ2		1		1																																									
			FETCH1		0																1																										
	34	JMPZ2	FETCH1		1		1			1										1																										1		
27	35	JMPSP1	JMPSP2		0	0	1																																									
			FETCH1			1																1																										
	36	JMPSP2	FETCH1		0	0	1			1										1																										1		
28	37	END																																													1	

4. Modules

4.1 Registers

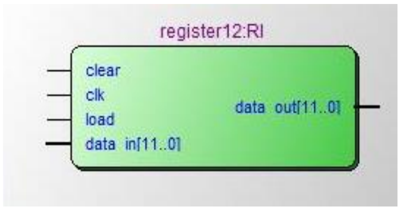
19 and 12 bit Register

Registers are modules that are used to store data temporarily during the processing cycle. This type of registers can store 19 bit data. Data stored in the register is always available at the 'data_out' and it is connected to one or both of the two de-multiplexers, so that each multiplexer can select which data out of many registers should be read in to bus A or bus B. These registers don't have any increment flags. If the stored values of these registers need to be incremented it has to go through an ALU increment operation and written back to the register.



As in the figure this register has a 19 bit input port and a 19 bit output port. If the 'load' flag is '1', it can write the data available in 'data_in' to the register at the negative edge of the clock. If the 'clear' flag is '1', it can clear data in the register to 19'b0.

The architecture that has been designed uses a similar type of 12 bit register with a 12bit input and output port.



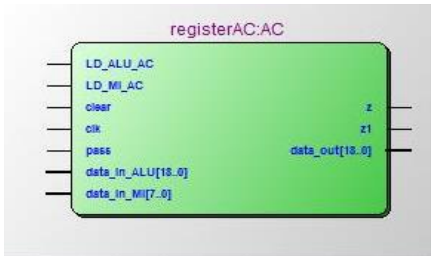
AC Register

Structure of this module is the same as 19 and 12 bit register with few additional functions. This special type of register is a 19 bit register and is used to store data available from the memory as well as the ALU during a processing cycle.

Data available from 'data_in_ALU' can be written into the register at the negative edge of the clock when 'LD_ALU_AC' is '1'. Similarly data available from 'data_in_MI' (image memory) can be written into the register at the negative edge of the clock when 'LD_MI_AC' is '1'. This register makes use of the 'clear' flag to clear data in AC register to 19'b0.

An added capability of this register is to pass data from the ALU at 'pass' flag, and to make data readily available at 'data_out' without waiting to be written to the register at a negative edge of the clock. This design decision saves us the cost of time in many arithmetic instructions by a clock cycle.

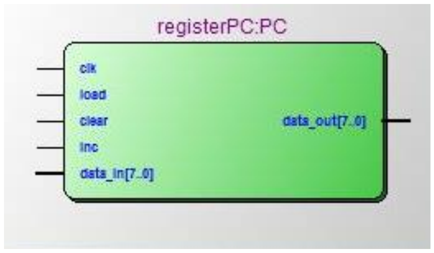
Two special flags are given out of the AC register to the control unit. ‘z’ flag is driven ‘1’ if AC register value is zero and ‘z1’ flag is driven ‘0’ if AC register value is either one or zero. ‘z’ flag was made use in implementing the ‘JMPZ’ instruction while ‘z1’ was used in ‘JMPSP’ by the control unit.



PC Register

PC register is an 8 bit register which holds the value of the memory location of the next instruction. If the ‘load’ flag is ‘1’, it can write the data available in ‘data_in’ to the register at the negative edge of the clock.

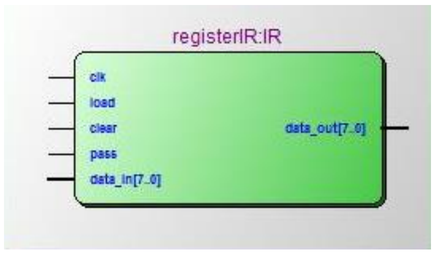
If the ‘clear’ flag is ‘1’, it can clear data in the register to 8’b0. Additionally if the ‘inc’ flag is ‘1’, it can increment the value stored in the register by 8’b1 at the negative edge of the clock, pointing at the next instruction memory location.



IR Register

IR is an 8 bit register which holds the instruction to be executed by the processor. It loads data from the instruction memory at ‘data_in’ when ‘load’ flag is ‘1’ at the negative edge of the clock. If the ‘clear’ flag is ‘1’, it can clear data in the register to 8’b0.

If the ‘pass’ flag is ‘1’, this register can pass data from its input port and to make data readily available at ‘data_out’ without waiting to be written to the register at a negative edge of the clock. It is made use to load jump addresses directly to the PC register.



Register	**Code**	**Size** (bit)	**Purpose**
Program Counter	PC	8	Tracks Instructions.
Instruction Register	IR	8	Current Instruction.
Accumulator	AC	19	Output register of ALU.
Register X	RX	19	Input image current pixel.
Register S	RS	12	Store sum of convolution.
Register I	RI	12	Image column position.
Register J	RJ	12	Image row position.
Register K	RK	19	Output image current pixel.
Register H	RH	12	Store input image height.
Register W	RW	12	Store input image width.

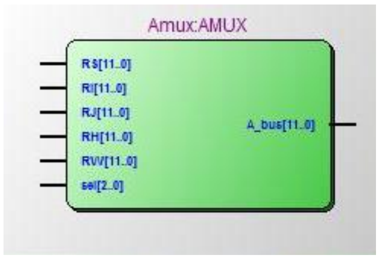
4.2 Multiplexer

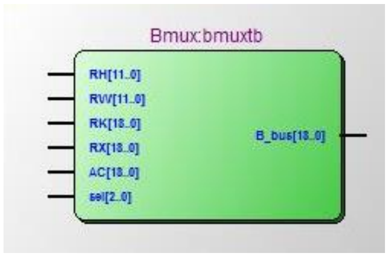
In the designed architecture there are three data buses. ‘Data bus A’ and ‘Data bus B’ acts as multiplexers reading data from one register at a time to the bus. ‘Data bus C’ acts just as a wire making AC data available to all other registers in the data path.

In the architecture 12 bit registers RS, RI, RJ, RH and RW are connected to the 12 bit data bus, ‘MuxA’. 19 bit registers RX, RK and AC are connected to the 19 bit data bus ‘MuxB’. Additionally RH and RW are also connected to ‘MuxB’ with seven ‘0’ bits added to the front when read into the bus. In both multiplexers a 3 bit selector is used to select which register is to be read into the bus.

Selector is set as below to read data from registers to the buses,

Bus A		Bus B	
RS	3'b000	RH	3'b000
RI	3'b001	RW	3'b001
RJ	3'b010	RK	3'b010
RH	3'b011	RX	3'b011
RW	3'b100	AC	3'b100





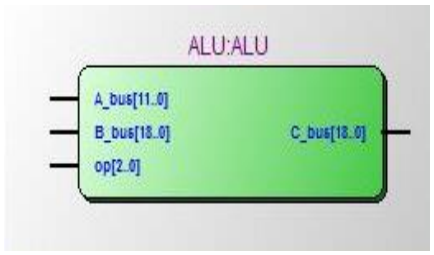
4.3 ALU

All the arithmetic and logical operations in the processor are done through this module. It has 2 inputs which are ‘A bus’ and ‘B bus’. ‘A bus’ is 12 bit and ‘B bus’ is 19 bit. There is only one output from the ALU which is the ‘C bus’. It is a 19 bit bus which is directly connected to the AC register as ‘data_in_ALU’.

For an ALU operation to be performed, ‘A bus’ needs to be concatenated with seven ‘0’ bits upfront since it is only 12 bits compared to 19 bits of ‘B bus’. The result of the ALU operation is then read onto ‘C bus’ and is readily available for the AC register. ALU operation to be performed on the operands will be executed according to the 3 bit flag ‘op’ which is sent out by the control unit as per the instruction.

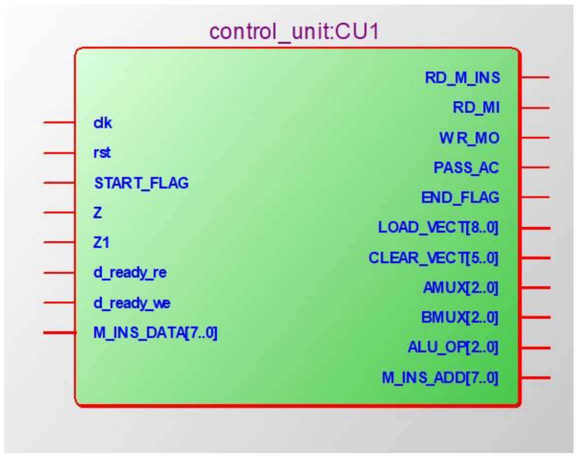
The operations of the designed ALU are,

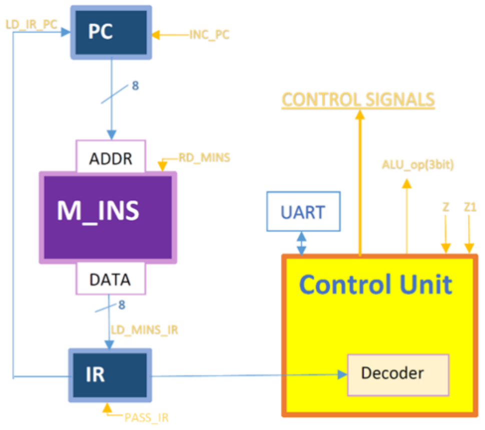
Operation	Op Flag	
ADD	3'd0	C ← {7'b0, A} + B
DIV16	3'd1	C ←{7'b0, A} >> 4
SUB	3'd2	C ← B - {7'b0, A}
INC2	3'd3	C ←{7'b0, A} + 19'd2
INC1	3'd4	C ← B + 19'd1
DEC1	3'd5	C ←B - 19'd1
MUL2	3'd6	C ←B << 1
MUL4	3'd7	C ← B << 2



4.4 Control Unit

The processor is mainly composed of the data-path and the control unit. While the data-path contains the registers with their interconnections, required for the execution of each instruction, the control unit decodes the current instruction and outputs the required control signals to the datapath.

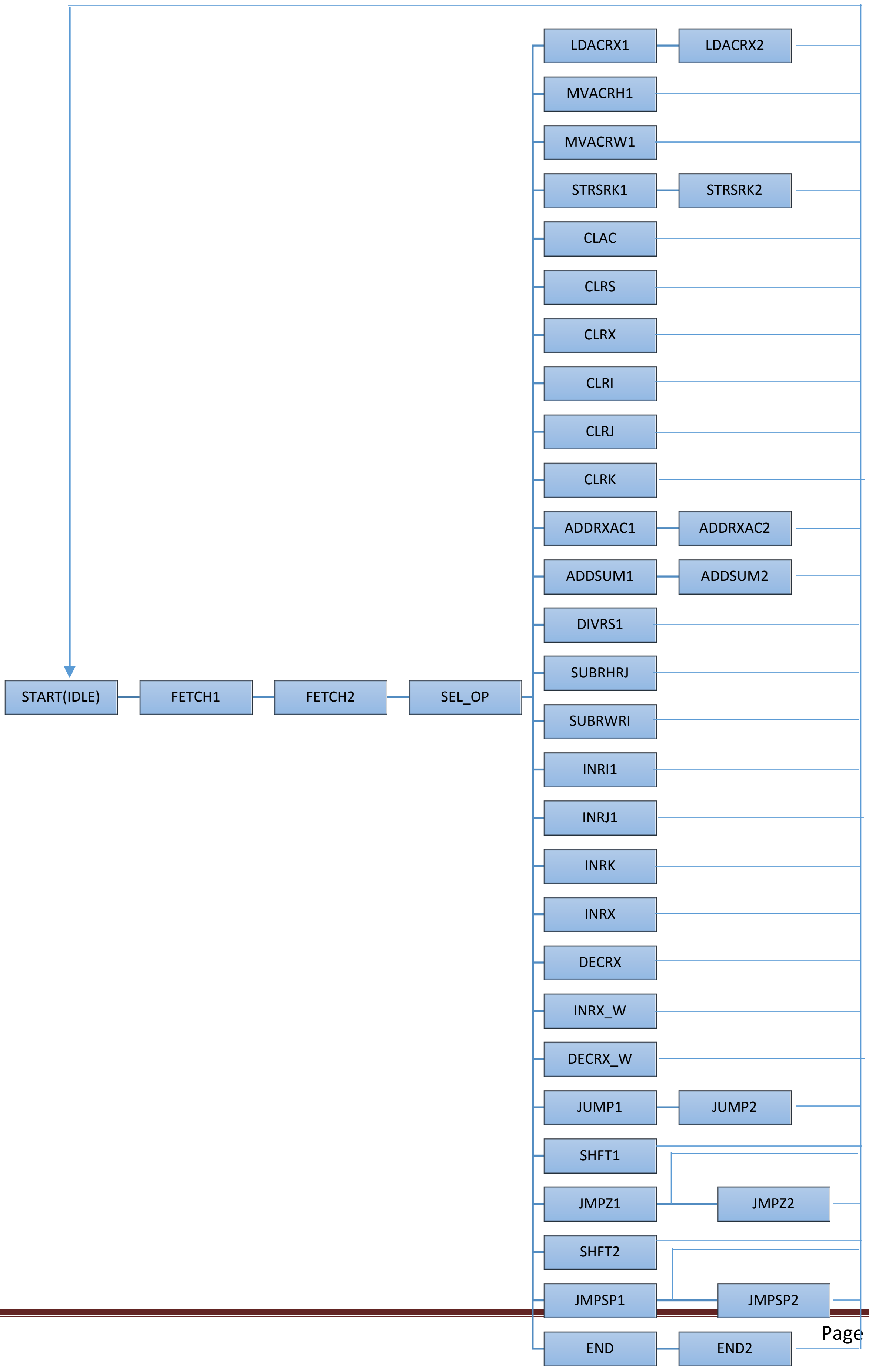




The control unit contains a finite state machine including the microinstructions of each instruction as its state. By switching through these states instructions are executed as necessary. To simplify the decoding process, the parameter number of the first state of each instruction was selected as the op-code for that instruction. So that the op-code loaded to the Instruction Register (IR) by the FETCH instruction, can be directly used to select the first state of the corresponding instruction.

The program counter register (PC) and instruction register (IR) was located separately from the data-path as a separated fetch unit because a separate memory was used to store instructions and its operation was independent from the main data-path.

Control Signal	Source	Destination	Description
START_FLAG	UART FSM	-	Flag to start executing instructions
Z	AC register	-	Zero flag from AC register
Z1	AC register	-	Zero or One flag from AC register
d_ready_re	CacheTop	-	Data valid signal for memory reading
d_ready_we	CacheTop	-	Write done signal for memory writing
M_INS_DATA	Memory_INS	-	Data from instruction memory
RD_M_INS	-	Memory_INS	Read ready signal to instruction memory
M_INS_ADD	-	Memory_INS	Address for instruction memory
RD_MI	-	Memory_IMG	Read ready signal to image memory
WR_MO	-	Memory_IMG	Write enable signal to image memory
PASS_AC	-	AC register	By pass AC register in data-path signal
LOAD_VECT	-	Datapath registers	Register selection signal in data-path to load ALU output
CLEAR_VECT	-	Datapath registers	Register selection signal in data-path to clear its value to zero
AMUX	-	Path A Multiplexer	ALU path A register selection
BMUX	-	Path B Multiplexer	ALU path B register selection
ALU_OP	-	ALU	ALU operation selection
END_FLAG	-	UART FSM	End of program execution indicator


START(IDLE)
FETCH1
FETCH2
SEL_OP
LDACRX1
LDACRX2
MVACRH1
MVACRW1
STRSRK1
STRSRK2
CLAC
CLRS
CLRX
CLRI
CLRJ
CLRK
ADDRXAC1
ADDRXAC2
ADDSUM1
ADDSUM2
DIVRS1
SUBRHRJ
SUBRWRI
INRI1
INRJ1
INRK
INRX
DECRX
INRX_W
DECRX_W
JUMP1
JUMP2
SHFT1
JMPZ1
JMPZ2
SHFT2
JMPSP1
JMPSP2
END
END2

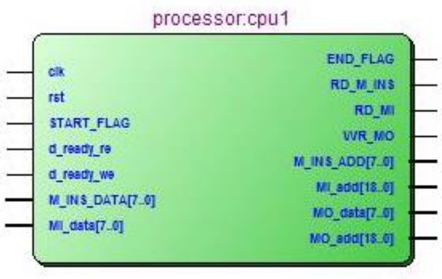
4.5 Processor

This module contains all the instances of modules related to processing and controlling. Memory modules and uart communication modules are not incorporated into this module. This is the CPU of our design with the following inputs.

- Clk - gives clock pulse for the synchronization
- Rst - resets the fpga to its original state, implemented by the push button
- START_FLAG - to receive signal from uart to processor to start processing
- M_INS_DATA - takes in instructions from instruction memory to the Instruction Register Of CPU
- MI_data - takes in input image data from DDR ram(image memory) to AC register of CPU(8 bit)
- d_ready_re - receives an indication from DDR ram to CPU that it is capable of giving out data from the memory location requested by the CPU
- d_ready_we - receives an indication from DDR to CPU that it is ready for data to be written by the CPU, at the requested memory location

The outputs from the processor module are,

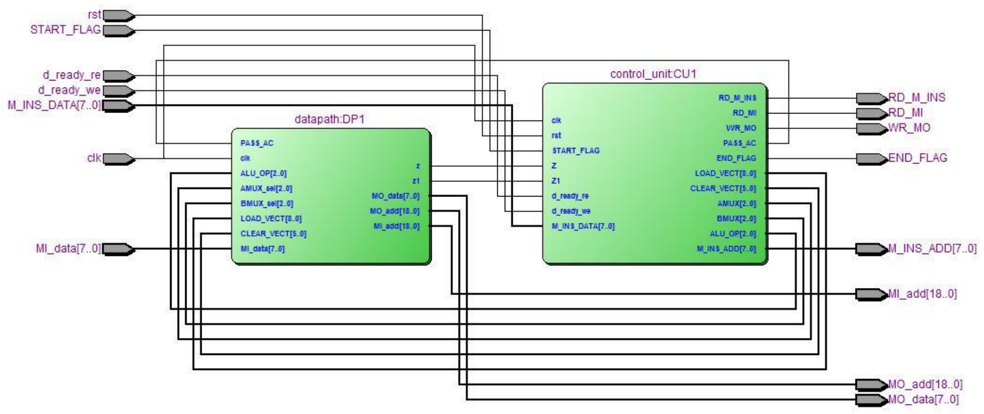
- END_FLAG - to send signal from processor to uart to indicate end of processing
- M_INS_ADD - gives memory location of the Instruction RAM (8 bit)
- RD_M_INS - gives signal from CPU to instruction memory to give out instruction data to be read(read enable)
- MI_add - gives memory location of DDR ram for input image data(19 bit)
- RD_MI - gives signal from CPU to DDR ram to give out image data to be read(read enable)
- MO_data - gives the data value which needs to be written in to the DDR ram from RS register of CPU(8 bit) .
- MO_add - gives memory location of the DDR ram for output image data(19 bit)
- WR_MO - gives signal from CPU to DDR ram to write data on defined memory location(write enable)



The modules integrated into the processor module are,

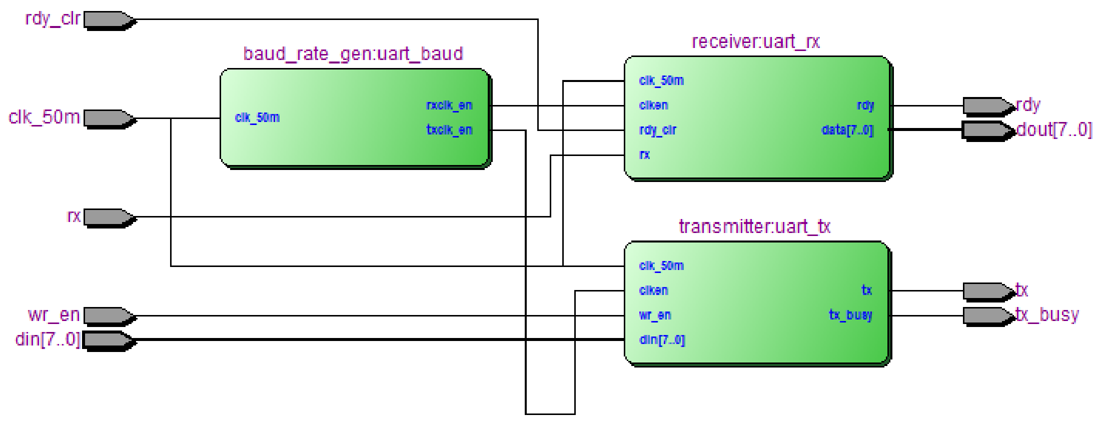
- datapath
  - register12(12 bit)
  - register19(19 bit)
  - register AC(19 bit)
  - ALU
  - Amux
  - Bmux

- control_unit
  - registerIR(8 bit)
  - registerPC(8 bit)



4.6 Communication related modules

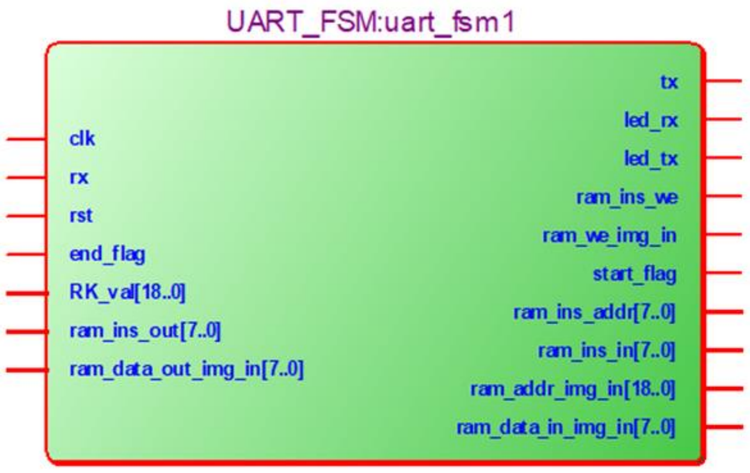
UART module



Universal asynchronous receiver transmitter (UART) module is used to communicate with the computer. This modules handles the transmission of a data byte between the FPGA and the computer according to the predefined UART standards. A baud rate of 115200 b/s was selected as a suitable rate for the data transmission considering the hardware capabilities of the FPGA and the required transmission durations.

Transmitter module works independently to transmit a data byte placed at **din** when **wr_en** is given. Similarly the Receiver module will receive a data byte serially from the computer and it will place the data byte at **dout** when it is fully received. **rdy** flag will be used to indicate to higher modules that a byte has been received. Since the communication is done asynchronously, 16x oversampling is done at the receiving input (**rx**) to detect the 'start bit' and to synchronize with the succeeding data bits. The baud rate generator module is used to generate the required baud rate clock (**txclk_en**) for the transmitter and the 16x baud rate clock (**rxclk_en**) for the receiver from the 50MHz system clock (**clk_50m**).

UART FSM Module



This module acts as a controller for the UART module and handles communication with the computer at a higher level. The individual bytes received from the UART module are decoded and the relevant operations are carried out. The services given to the computer by this module are,

1. Write data to image/instruction memory from the computer
2. Read data from image/instruction memory to the computer
3. Read output data from execution of a program in the processor to the computer
4. Start execution
5. Indicate the end of execution

The finite state machine of this module starts from the **IDLE** state and goes to a corresponding state if either a data byte is received or the end flag is received. If the end flag is received, the module will transmit command byte (**END**) with a specific predefined value, to the computer so that computer will identify this byte as the end flag transmission. If a data byte is received from the computer, initially this will be considered as command byte. From the command byte, the module can identify the service required by the computer. After all operation related to the command are carried out, the state machine will return to the **IDLE** state. The remaining command bytes and their operations are given below.

1. **START** – the start flag will be set, indicating the start of execution to the processor
2. **RX_INS** – write to instruction memory

   Following this byte, the number of bytes to be written will be sent using 3 bytes, where the most significant byte will be sent first. Then the bytes to be transmitted will be received by the module and written to the instruction memory. After all the bytes are received the state machine will return to the **IDLE** state
3. **RX_IMG_I** – write to image memory

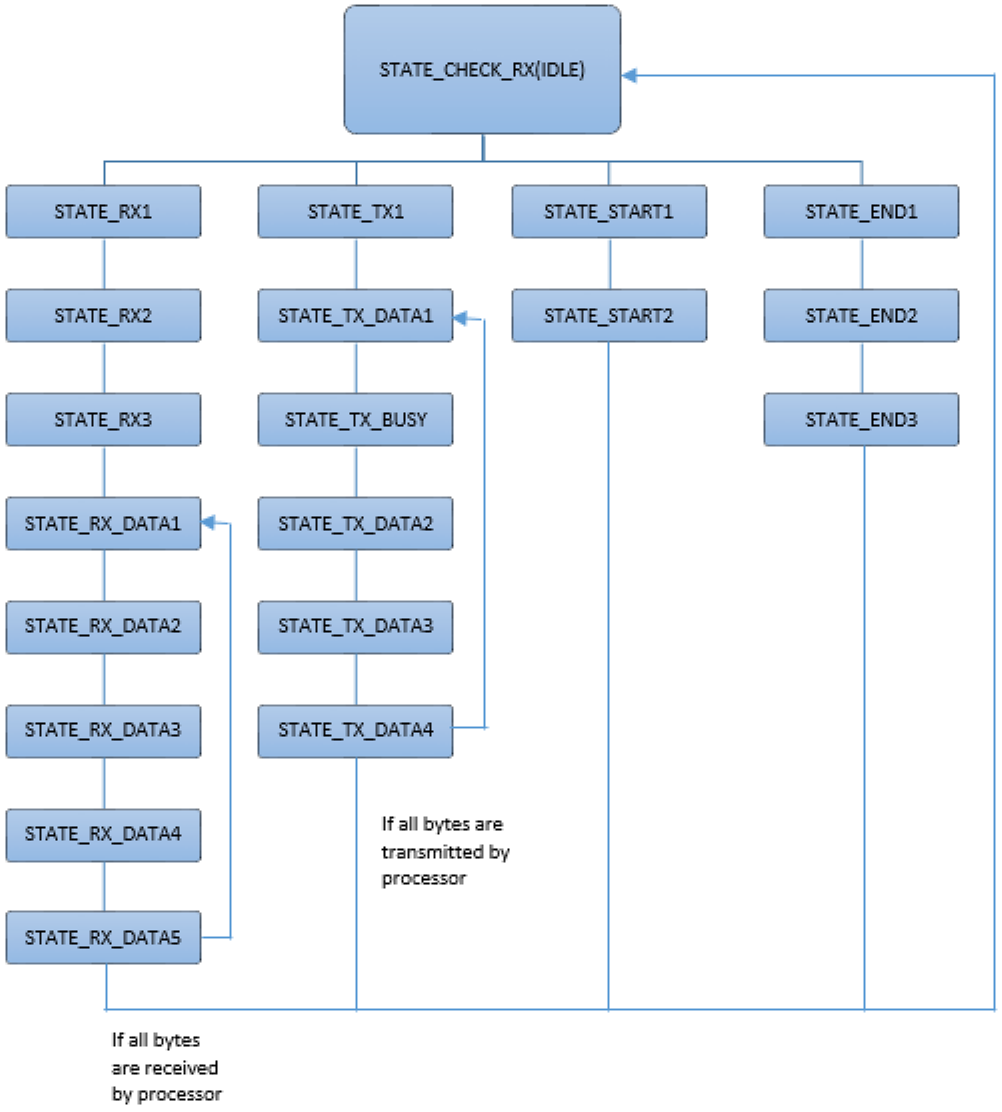
   Similar to **RX_INS**
4. **TX_INS** – read instruction memory

After this byte is received, number of bytes of data that was written by **RX_INS** will be transmitted back to the computer. This command can be used to verify the written data by **RX_INS**. (**RX_INS** must be called first, to use this service)

5. **TX_IMG_I** – read image memory
   Similar to **TX_INS**

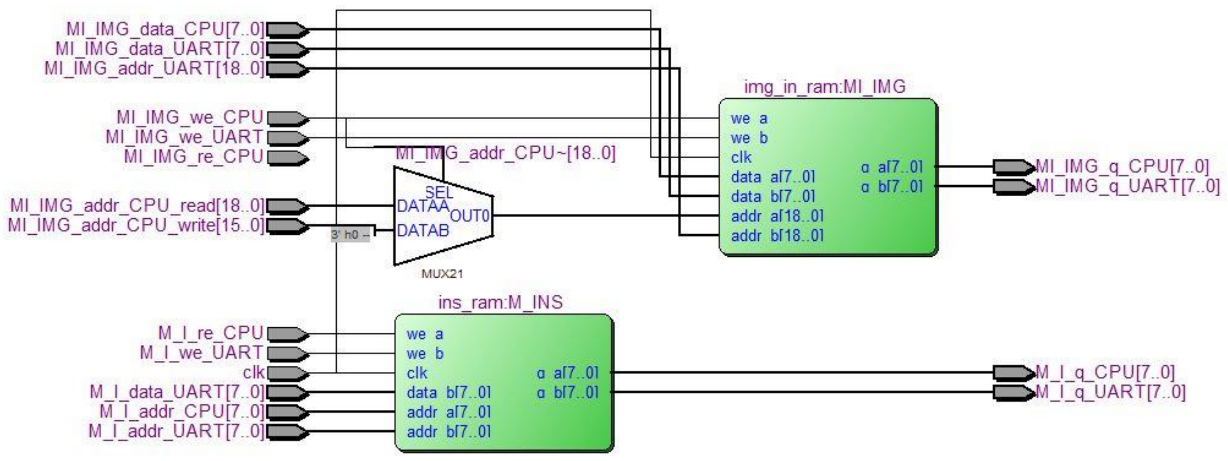
6. **READ_IMG_O** – read executed output
   The data bytes written by the processor to image memory will be transmitted to the computer.

Command	Value
START	240
END	239
RX_INS	255
TX_INS	254
RX_IMG_I	253
TX_IMG_I	252
READ_IMG_O	249



4.7 Memory modules

The down sampling processor consist of 3 main modules which are the UART controller module, Processor and the Memory Module. The processor consist of 2 main memories which are Image memory and the instructions memory. Each of the memory can be accessed by the processor by addressing though different registers separately.



Instructions Ram

The main feature about this ram is to store instructions. All the instructions which are coded by assembly language is stored in this memory. When the processor needs instructions it fetches instructions from this ram. The instructions are saved in the memory according to the order in which the instructions are executed. Processor can access the instructions by placing the address in the address bus and reading the data.

The instructions memory has 256 Byte memory size and is implemented in FPGA as a block ram. This memory is a small memory compared to the image memory. The address width of the bus is 8 bit and data width is also 8 bit.

The simple dual port architecture is used to remove the collision between top UART module and the processor address busses. Hence at the start the UART will write to the Instruction ram using port B of the simple duel port memory and then once the processor starts the instructions will be read by the port a of the memory. There is no need of reading the memory after the process finish.

Image Ram

This Memory is used to store the input image and the output image. This Ram is also a Simple Dual port memory implemented using Block rams in the FPGA. This memory has 263168 byte size which can hold 512 x512 image where bit depth of each pixel value is one byte. The address width of the memory is 19 bits to access the 8 bit data width memory locations.

The processor is programed to process variable sized images. The maximum image size the processor can process is 512 x512 and to store the variables height and the width of the image.

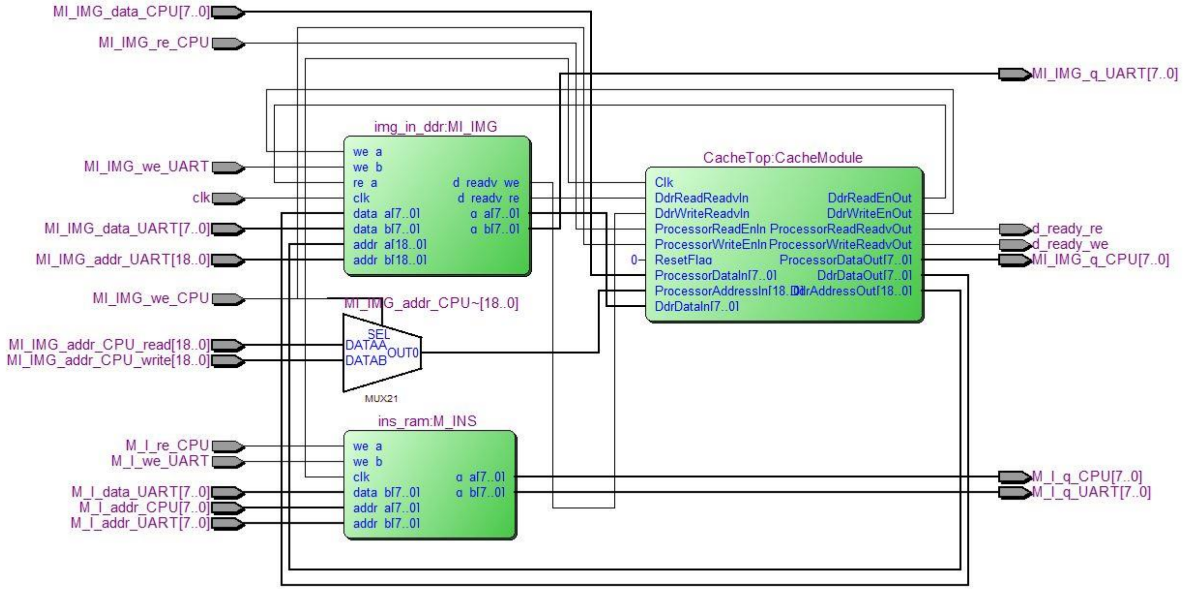
The top UART module will access the port b and write the input image and the variables to the memory. Processor can access the memory using the port b. The output down sampled image will be overwritten to the image ram and the after the processor finish the down sampling process the UART will read only the re written down sampled image and output.

RAM Simulation in Memory Unit

In FPGA block rams the latency for one data access is only has one clock cycle delay. But in hardware level rams after placing address in the address bus there is a considerable amount of delay to read the data. Hence in most of the practical applications high speed cache memories are used.

When implementing a cache memory to stimulate the effect of a cache in FPGA there will not be a efficiency in time consumption by the processor but a drawback. This is because the when there is a cache miss the processor has to wait several clock cycles to get the data because cache takes time to download a data line serially from the main memory.

Hence to study the effect of cache on the processor the Block ram memory implantation is not useful. In order to compensate to that the ram module was developed. When an address is placed in the address bus to output the first data the ram will wait for a certain amount of clock cycle delay. For all the consecutive data the Image ram ram will maintain the throughput of one.



The communication interface will act as an AXI synchronous boundary which use the write enable, read enable and the ready signal. This upgrade was only done for image ram because the image ram has the largest memory

For the UART module port b there will not be a change in delay values.

The input ports of the DDR ram include,

- data_a - data value to be written onto DDR ram from CPU(8 bit)
- data_b - data value to be written onto DDR ram from UART(8 bit)
- addr_a - memory location of DDR ram to be written from CPU(19 bit)
- addr_b - memory location of DDR ram to be written from UART(19 bit)

- we_a - write signal from CPU
- we_b - write signal from UART
- re_a - read signal from CPU
- clk - gives clock pulse for the synchronization

The output ports of the DDR ram include,

- q_a - data value available from DDR ram to CPU(8 bit)
- q_b - data value available from DDR ram to UART(8 bit)
- d_ready_we - write ready flag given from DDR ram to CPU for data to be written
- d_ready_re - read ready flag given from DDR ram to CPU for data to be read

For a change in address from CPU ('addr_a'), DDR ram waits for a read enable or write enable. For instance to write 'data_a' at 'addr _a' it waits until 'we_a' (write enable) is received and compares the previous address from the current address. If they are adjacent addresses, a 'd_ready_we' (write ready) signal is given out at the positive edge of the next clock cycle. If the addresses are non-adjacent it gives out the 'd_ready_we' signal after a predefined delay which is defined by the register 'DELAY'. The DDR delay used is 250 clock cycles.

Even if there is no address change, the 'd_ready_we' signal is given out in the next clock cycle after a 'we_a' is detected. The 'd_ready_we' stays high for a clock cycle and then turns off. The same implementation is used when a 're_a' (read enable) from CPU is received to validate data at 'q_a'.

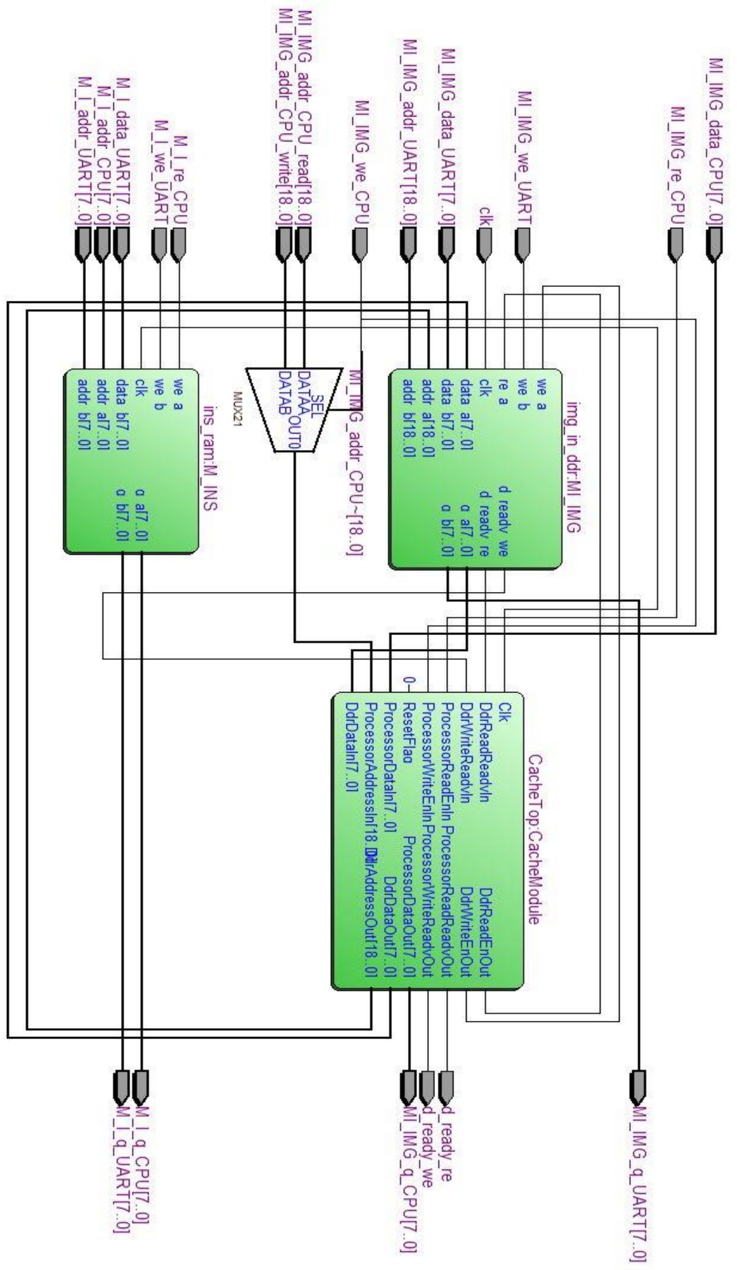
Data from UART is written onto DDR ram at the positive edge of clock when a 'we_b' (write enable) is detected. Data from DDR ram at 'addr_b' is readily available to UART at output port 'q_b'.

4.8 Cache

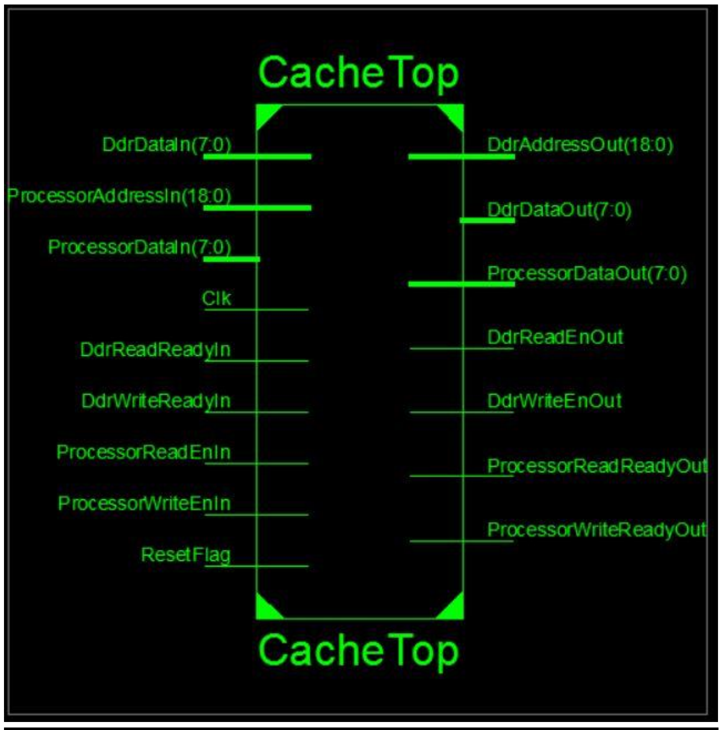
Cache Module

Cache memory, also called CPU memory, is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM. This memory is typically integrated directly with the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU.

In this project the out of different cache implementation we have implemented the cache using direct mapping method. The cache will act as an interface between the image memory and the processor as shown in Figure 1.


M_I_addr_UART[7..0]
M_I_addr_CPU[7..0]
M_I_data_UART[7..0]
M_I_we_UART
M_I_re_CPU
MI_IMG_addr_CPU_write[18..0]
MI_IMG_addr_CPU_read[18..0]
MI_IMG_we_CPU
MI_IMG_addr_UART[18..0]
MI_IMG_data_UART[7..0]
clk
MI_IMG_we_UART
MI_IMG_re_CPU
MI_IMG_data_CPU[7..0]
ins_ram:M_INS
img_in_ddr:MI_IMG
MUX21
MI_IMG_addr_CPU~[18..0]
CacheTop:CacheModule
M_I_q_CPU[7..0]
M_I_q_UART[7..0]
MI_IMG_q_CPU[7..0]
d_ready_we
d_ready_re
MI_IMG_q_UART[7..0]

Cache top Module



This module will act as a bridge between the processor and the image memory. The cache memory size is 1024 bytes where a cache line width is 4 bytes. When the processor need to read data from the main memory the cache first will check whether the data is available in the cache memory and if so in the next clock cycle the data will be available for the processor.

Cache memory is a high speed memory and although by using a cache memory the throughput of the processor increase in hardware, in FPGA as the main memory is a block ram which has a maximum latency of one clock cycle the effect of adding a cache memory can be considered negligible or it can reduce the performance.

Cache address classification

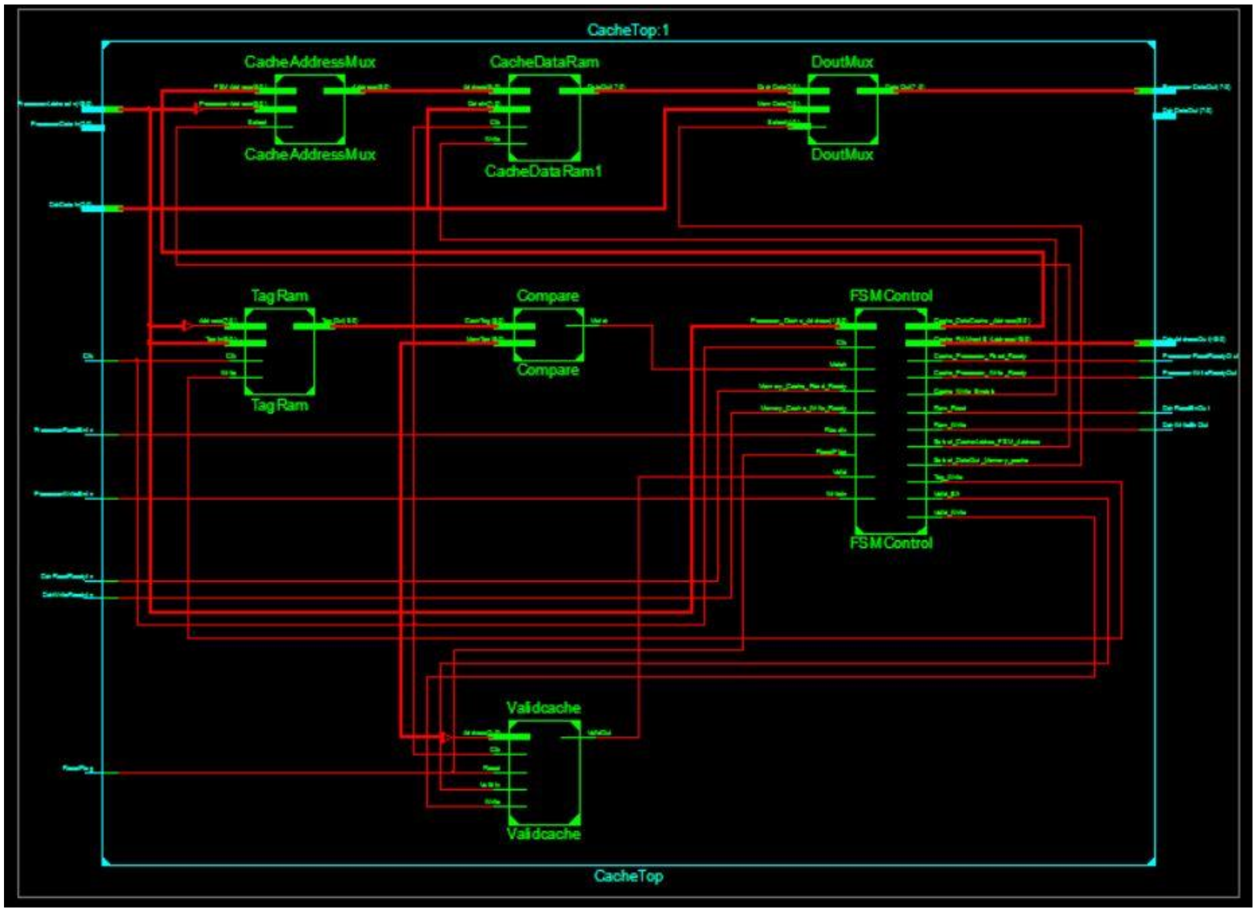
Tag address (9 bits)	Line address (8 bits)	Line Location (2 bits)

Cache Memory Architecture

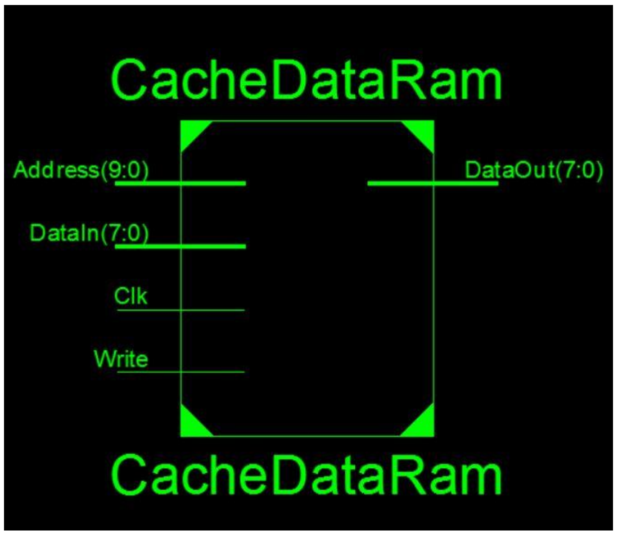
Cache memory architecture consist of

- Cache Data Memory
- Tag ram
- Valid Ram

- Address Compare module
- Data output Multiplexer
- Cache Controller –FSM
- Cache Memory Address Multiplexer

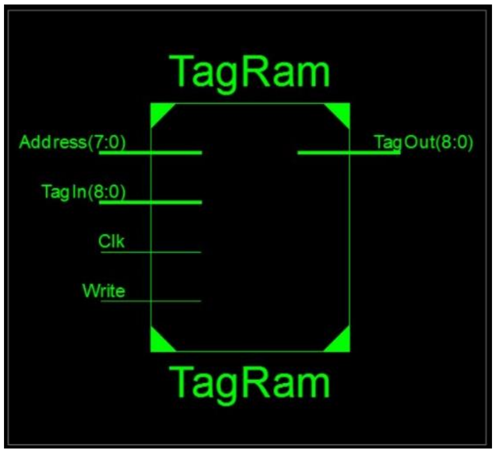


Cache Data Memory



The data ram size is 1 kilo byte or 1024 bytes. The cache line length is 4 bytes where the cache has 256 lines. Compared to the main memory the cache memory size is negligible. The data with of the memory address is 19 bits and cache memory address size is 10 bits which allow to access each 1024 bytes. Cache memory data will be serially updated by the 8 bit width data in bus. The memory is synchronized with the negative edge of the clock where according to the address the data will output and cache update will happen.

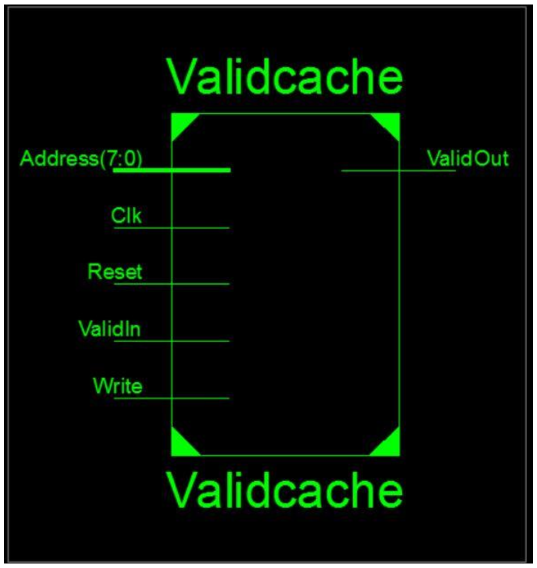
Tag Ram



This is a memory which has 2034 bits of size. This memory is used for acidification of the data in the cache memory. When the processor needs data, according to the address requested the tag ram will output the tag address for the data which is currently in cache ram. Because the number of lines in the cache memory is 256 the address with required is 8 bits.

Tag ram is also synchronized with the negative edge of the clock where data input and output will be done according to the request.

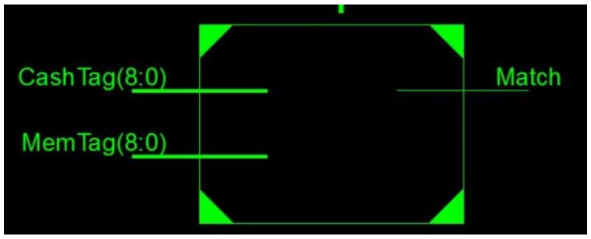
Valid Ram



This memory has 256 bits. The purpose of the memory is to tell the control system of the cache whether the data available for the requested address is valid or not. As above modules this module is also synchronize with the negative edge of the clock.

When there is a cache miss and the data is being loaded to the valid memory to update the record of that memory location. When the data become invalid in the cache the FSM will inform the valid ram to delete the record.

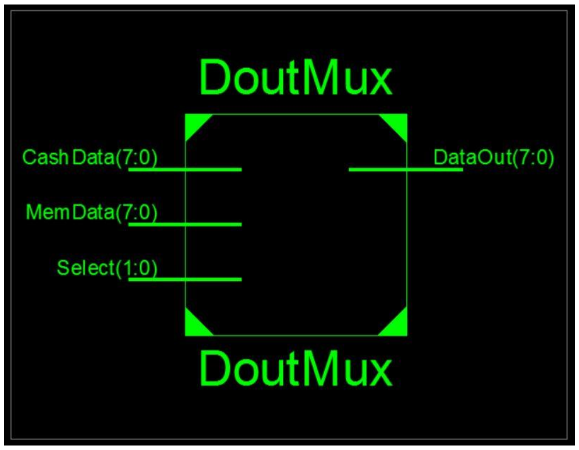
Address compare module



When the processor request data, according to the address the cache must decide whether it has the valid data. Here the mem Tag bus is the first 9 bits or the Tag address of the requested address (Figure 3) .Cache tag comes from the tag ram module as described earlier. The module will output whether there is a cache hit or cache miss.

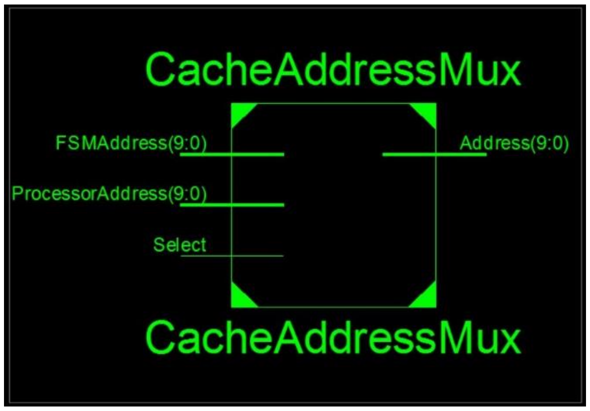
This module only has combinational logics hence there is no need of a clock synchronization.

Data out Multiplexer



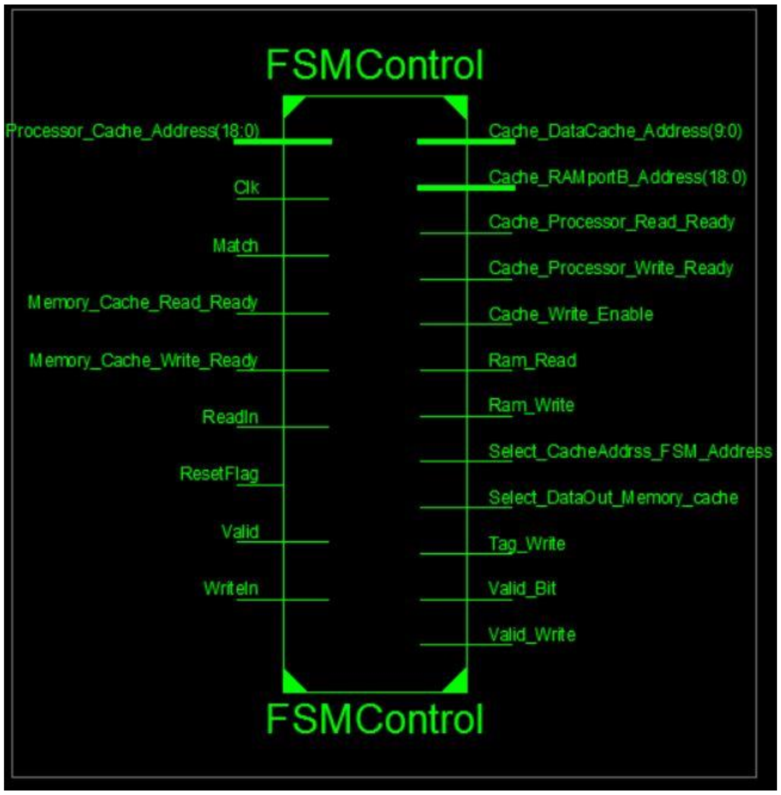
This module controls the data output to the processor. By using the select pin the cache controller can switch the output between memory data output or to cache data output. This module has only combinational logics.

Cache Address selection multiplexer



This multiplexer is used to address the cache data ram. By using selection pin the cache memory can be accessed by the processor address or by the FSM address. This also consist of combinational logic only.

Cache Controller – FSM


FSMControl
Processor_Cache_Address(18:0)
Clk
Match
Memory_Cache_Read_Ready
Memory_Cache_Write_Ready
ReadIn
ResetFlag
Valid
WriteIn
Cache_DataCache_Address(9:0)
Cache_RAMportB_Address(18:0)
Cache_Processor_Read_Ready
Cache_Processor_Write_Ready
Cache_Write_Enable
Ram_Read
Ram_Write
Select_CacheAddrss_FSM_Address
Select_DataOut_Memory_cache
Tag_Write
Valid_Bit
Valid_Write
FSMControl

This is the main controller of the cache memory. This has a Finite state machine which controls the modules and send control signals to the sub modules in the cache.

The FSM will wait in the idle state till the processor data read or data write command come. If the processor is requesting data according to whether data is the cache (this uses the match signal from the address compare module and the flag from the valid ram module) . If there is a match in the imidiate clock cycle the output data will be valid to the processor.

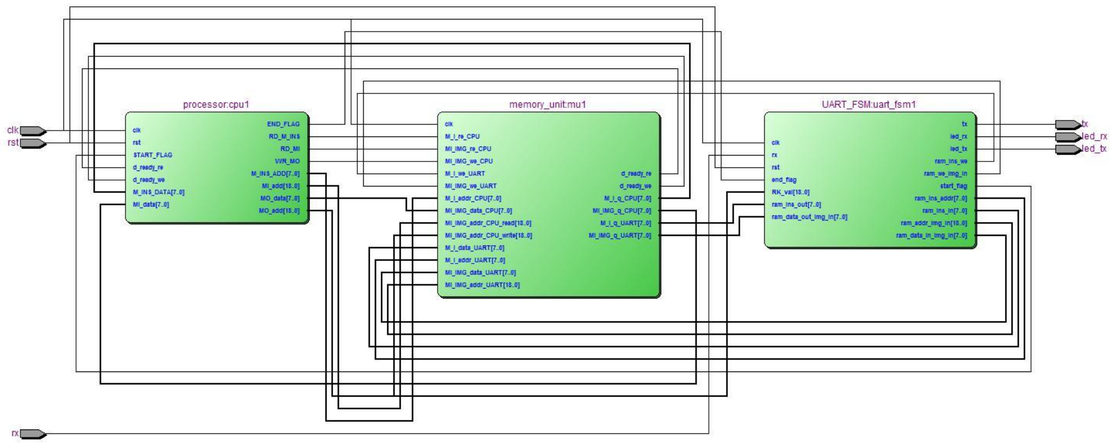
If there is a cache miss the FSM controller will accress the main memory for the fist data and write into the cache memory and data will be availabe for the processor to execute. Other data will be requested from the main memory serially to fill the cache line and the records in Tag memory and Valid memory will be updated.

If the processor is writing to a existing location in cache memory the validity of the data will be removed.

This module is a sequential module which uses positive edge for syncronization.

4.9 top module

This is the module which inter connects communication, processing and memory modules. All module instances have been created inside this module. It has a clock signal and a reset pin as its inputs together with 'rx' pin for receiving the data coming in to the FPGA serially. It has only three output pins. 'tx' is to transmit data serially to the computer while 'led_rx' and 'led_tx' are indication LED pins when receiving and transmitting data. It contains instances of Processor, memory_unit and UART_FSM.



5. Testing, simulations & modifications

The modules that the processor is comprised of were first created unit by unit. For each module separate test codes were run to check the functionality of the module (Unit Testing). To simulate the test results, the software 'ModelSim' was used. To create test benches, Verilog syntaxes which are only simulation supported and not FPGA synthesizable were used. Test code created to make a clock pulse is given below.

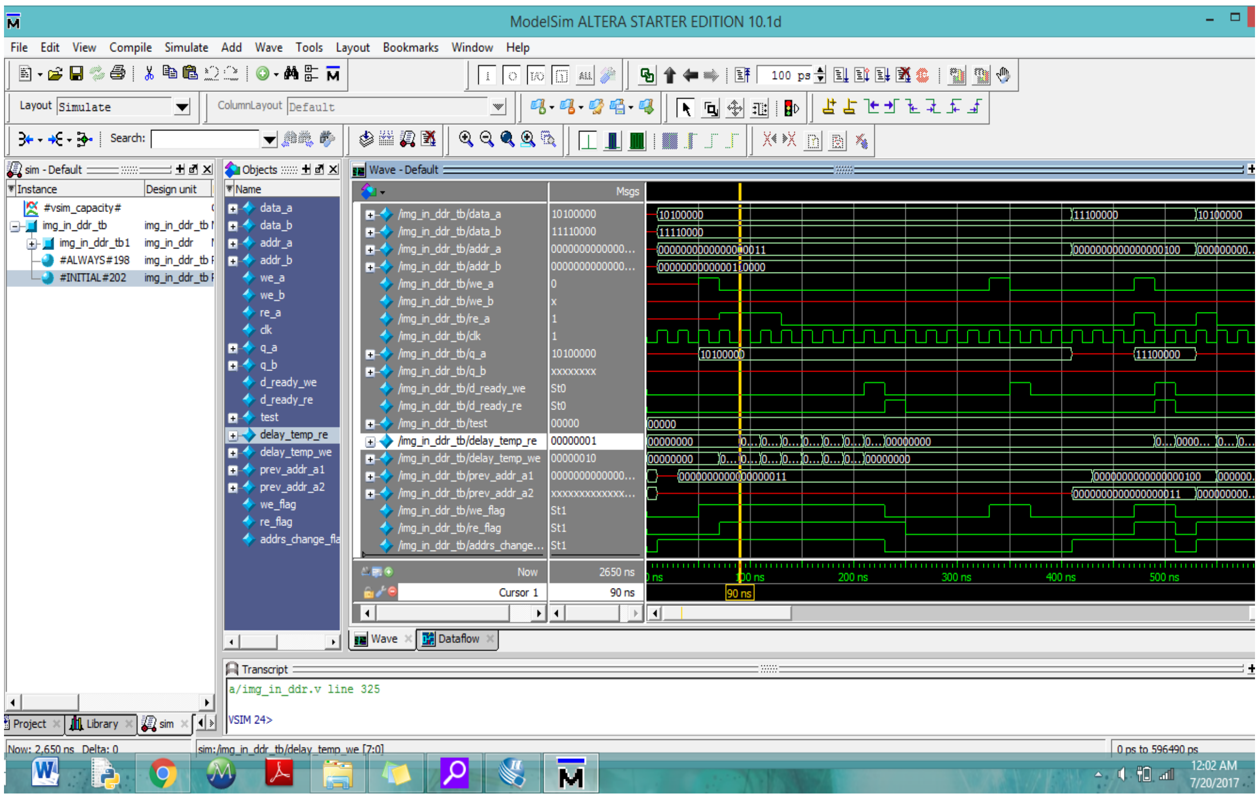
```
always begin
        #10 clk <= ~clk;
end
```

A simple test case,

```
initial begin
        #10
        data_a <= 8'b1010_0000;
        data_b <= 8'b1111_0000;
        addr_a <= 19'b000_0000_0000_0000_0011;
        addr_b <= 19'b000_0000_0000_0011_0000;
        #40
        we_a <= 1;
        we_b <= 1;
        #20
        we_a <= 0;
        we_b <= 0;
        $finish;
end
```

To verify that our processor and related modules are working as expected, temporary outputs were taken out for inspection. For an example to test the functionality of 'ddr_sim' which is our DDR ram simulator,

- 'test' - to test on which if-else condition the code runs at each test scenario
- 'delay_temp_re' - to observe the count up to predefined delay, when a non-adjacent address data value is requested by the CPU
- 'delay_temp_we' - to observe the count up to predefined delay, when the CPU intends to write data at a non-adjacent address
- 'we_flag' - register to be driven high when a write enable signal arrives until a write ready is given out by the DDR ram
- 're_flag' - register to be driven high when a read enable signal arrives until a read ready is given out by the DDR ram
- 'addrs_change_flag' - register to be driven high when an address change occurs until a read ready or a write ready is given out by the DDR ram.



Instance	Design unit
#vsim_capacity#	
img_in_ddr_tb	img_in_ddr_tb
img_in_ddr_tb1	img_in_ddr
#ALWAYS#198	img_in_ddr_tb
#INITIAL#202	img_in_ddr_tb

	Msgs
/img_in_ddr_tb/data_a	10100000
/img_in_ddr_tb/data_b	11110000
/img_in_ddr_tb/addr_a	0000000000000...
/img_in_ddr_tb/addr_b	0000000000000...
/img_in_ddr_tb/we_a	0
/img_in_ddr_tb/we_b	x
/img_in_ddr_tb/re_a	1
/img_in_ddr_tb/clk	1
/img_in_ddr_tb/q_a	10100000
/img_in_ddr_tb/q_b	xxxxxxxx
/img_in_ddr_tb/d_ready_we	St0
/img_in_ddr_tb/d_ready_re	St0
/img_in_ddr_tb/test	00000
/img_in_ddr_tb/delay_temp_re	00000001
/img_in_ddr_tb/delay_temp_we	00000010
/img_in_ddr_tb/prev_addr_a1	0000000000000...
/img_in_ddr_tb/prev_addr_a2	xxxxxxxxxxxxx...
/img_in_ddr_tb/we_flag	St1
/img_in_ddr_tb/re_flag	St1
/img_in_ddr_tb/addrs_change...	St1
Now	2650 ns
Cursor 1	90 ns

```
a/img_in_ddr.v line 325

VSIM 24>
```

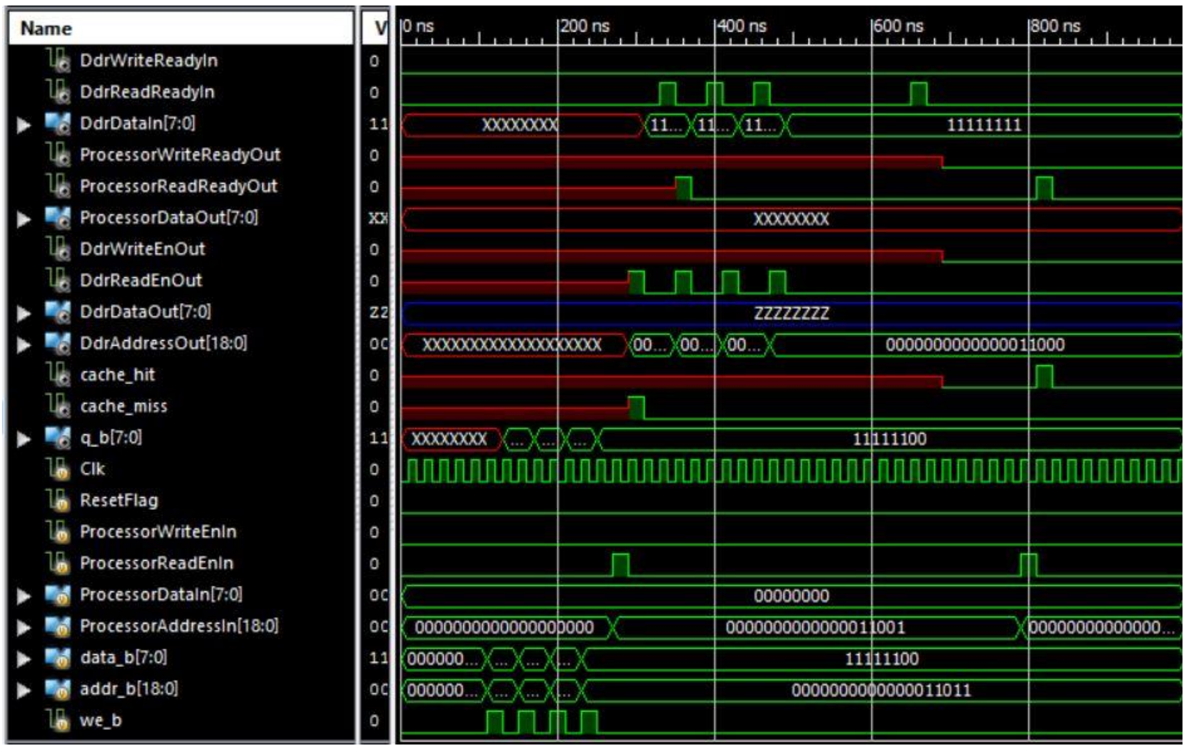
Now: 2,650 ns Delta: 0   sim:/img_in_ddr_tb/delay_temp_we [7:0]   0 ps to 596490 ps

Simulated Results

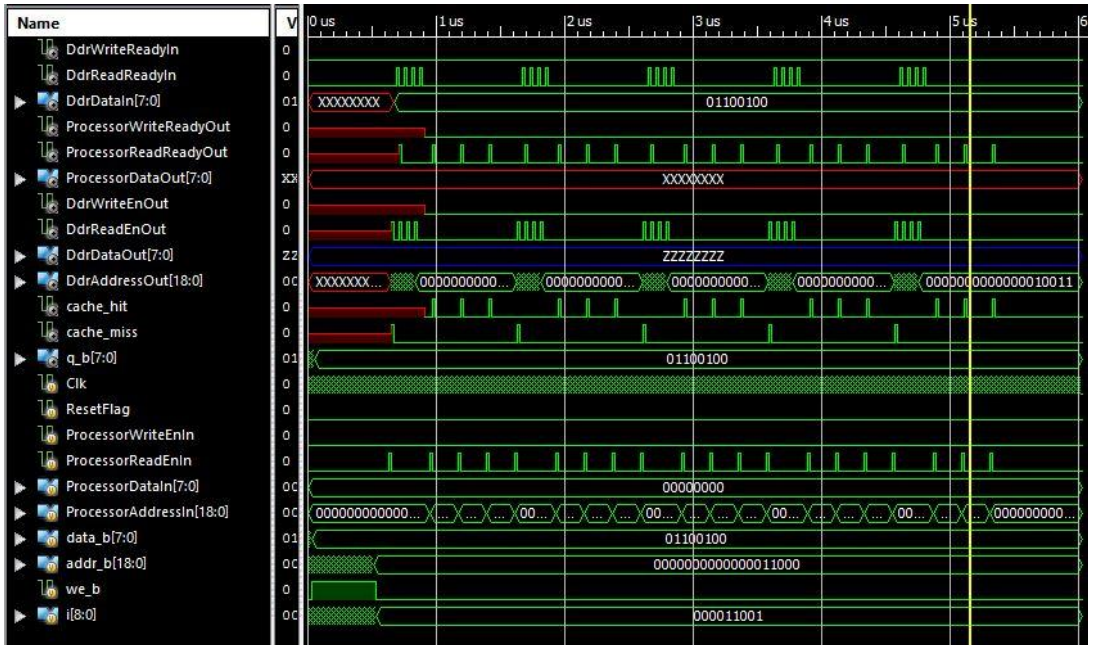
5.1 Cache Test Results

As described earlier the memory module the delay parameter can be changed.

Timing diagram for cache read and write



Timing diagram for data read from cache for addresses from 0 to 19



The periodic nature of the cache hit and miss can be observed.

6. Result analyzing and verification

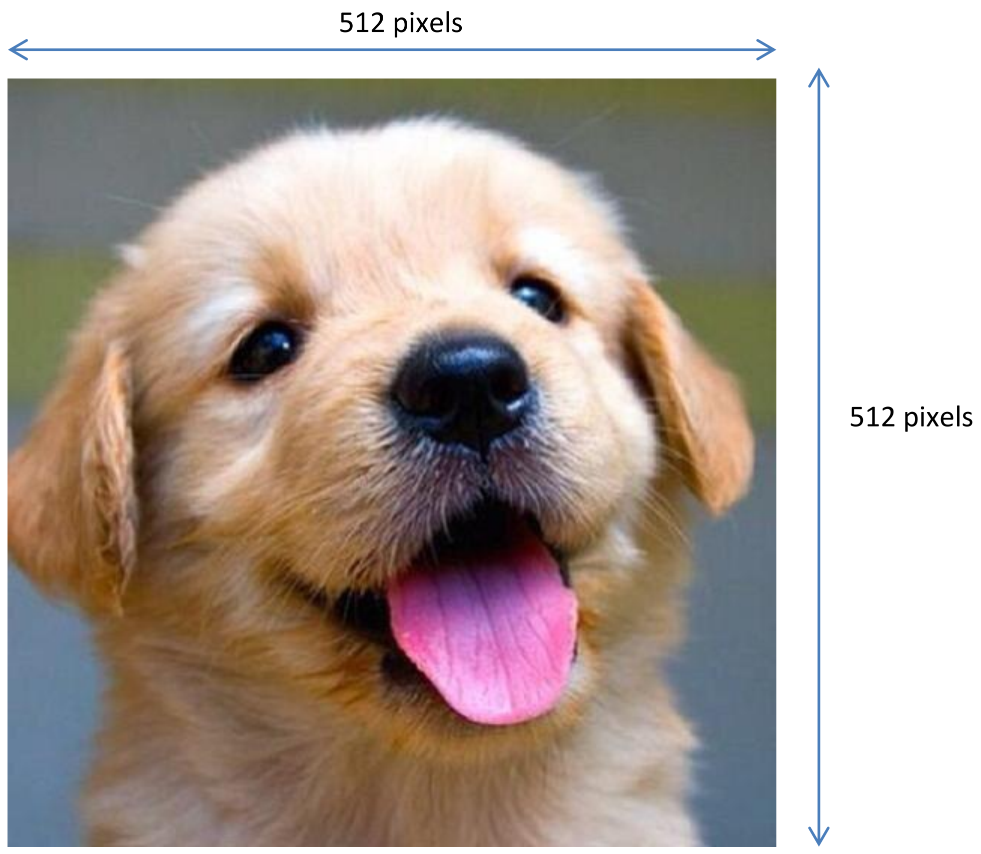
After getting the processed data from FPGA to computer, analyzing the result is very important to get an idea on the accuracy of the designed ISA and algorithms. In order to compare and calculate the error rate and accuracy of the microprocessor output image, a Matlab reference image is first created.

6.1 Generate reference output image

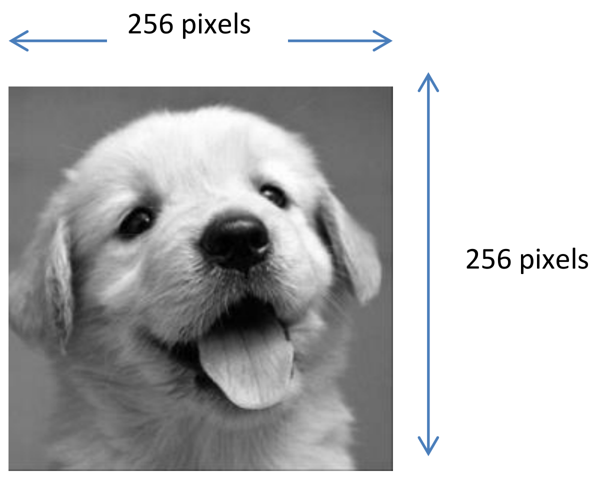
The algorithm used by Matlab to generate a 2:1 down sampled image is,

- A One Dimensional pixel array is created from input image to be processed
- Filter the image (using a 3x3 Gaussian kernel) and down-sample at the same time (in order to avoid unnecessary calculations involved in smoothing out pixels that won't be in the output image.)

- Zero padding is used when necessary to avoid black lines on edges the kernel can't reach
- The One Dimensional output image pixel array (after being processed) is reshaped to half the width and height of original image as suited for display and verification purposes



Original Image



Down Sampled Image Using Matlab

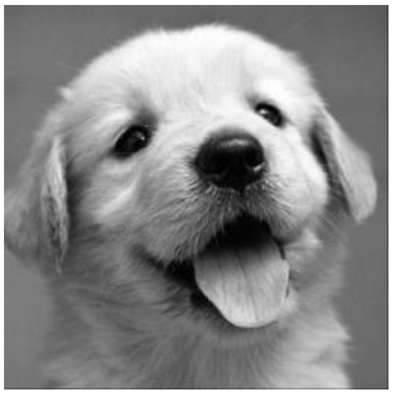
6.2 Results verification and analysis

The FPGA implemented microprocessor and CPU is used to process the given image and output byte array is given to the computer. Matlab software is used to reshape this one dimensional byte array to half the width and height of original input image as suited, display as an image and to be represented by a two dimensional unsigned integer array of pixel values for analysis.

To compare the accuracy of the FPGA processed image, the Matlab reference image and FPGA processed image can be compared using human eye. But it is not very suitable to get a measure of accuracy or error. Therefore data array of the processed data is compared with the reference image data array using sum of squared differences calculation.

FPGA and Matlab input image (to be processed)

Reference Down-Sampled image



FPGA Processed Down-Sampled image

6.3 Error analysis

For error analysis, data array of the FPGA processed data is compared with the reference image data array using sum of squared differences calculation.

The Matlab code excerpt for SSD calculation is given below

```
% img_array  -  CPU downsampled one dimensional image pixel value array
% fil_ds_Im  -  Matlab downsampled one dimensional image pixel value array

ssd = fil_ds_Im - img_array;
ssd_img = reshape(ssd,[floor(w/2),floor(h/2)]);
ssd = ssd .* ssd;
ssd_tot = sum(ssd);

display(ssd_tot);
```

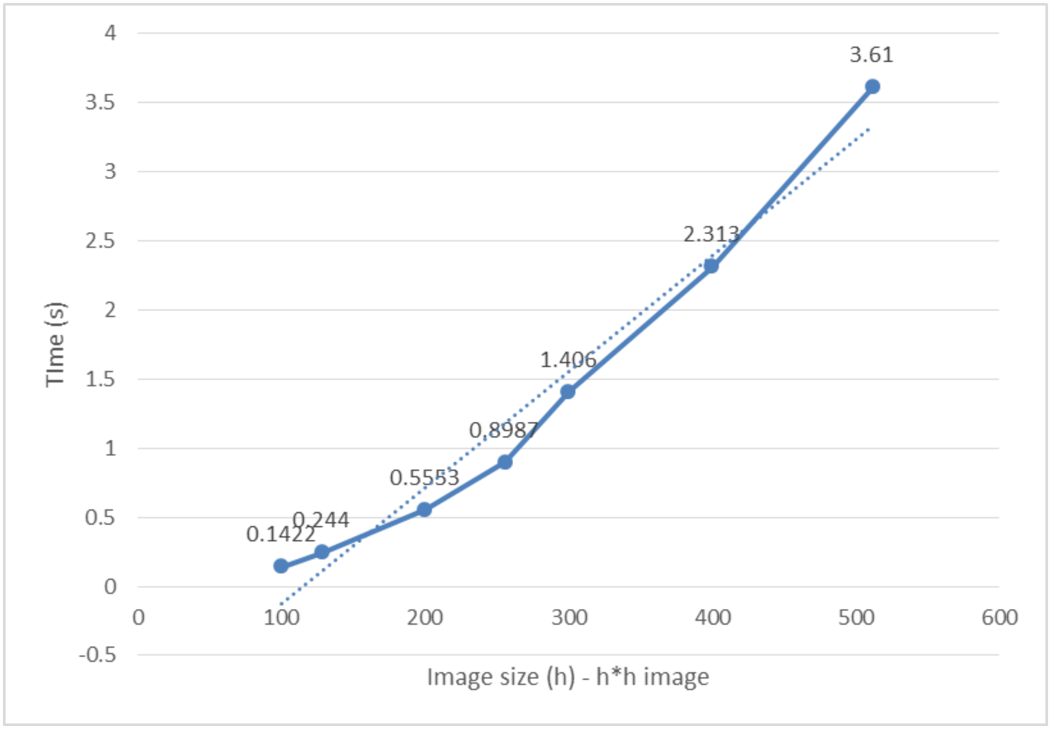
We got a ssd_total (sum of squared differences value) of **zero** for our comparison of FPGA processed and Matlab reference down sampled images. Therefore we can come to the conclusion that our processing method for image down sampling and implementation on FPGA is working accurately as expected.

The reasons for **zero error** can be justified as follows,

- The algorithm in Matlab is exactly replicated by the assembly code and each operation is exactly done in same order.
- But in FPGA implementation, we shift instead of dividing (by dividers) to avoid the greater latency of execution of division instruction.
- In our algorithm the only division requirement is to divide by 16 and if any error is to occur, it should be the error introduced by shifting by 4 (since base 2 is used) instead of dividing in FPGA. Therefore a SSD error can only occur if there is a decimal part after the radix point of Matlab processed data array.
- To confirm the above hypothesis we obtained the floor value of Matlab divided value (to remove the decimal part) and calculated SSD with FPGA processed image and obtained error value as **zero.**
- Thereby we were able to authenticate the correctness of the FPGA implementation of the down sampling algorithm.

6.4 Timing Analysis

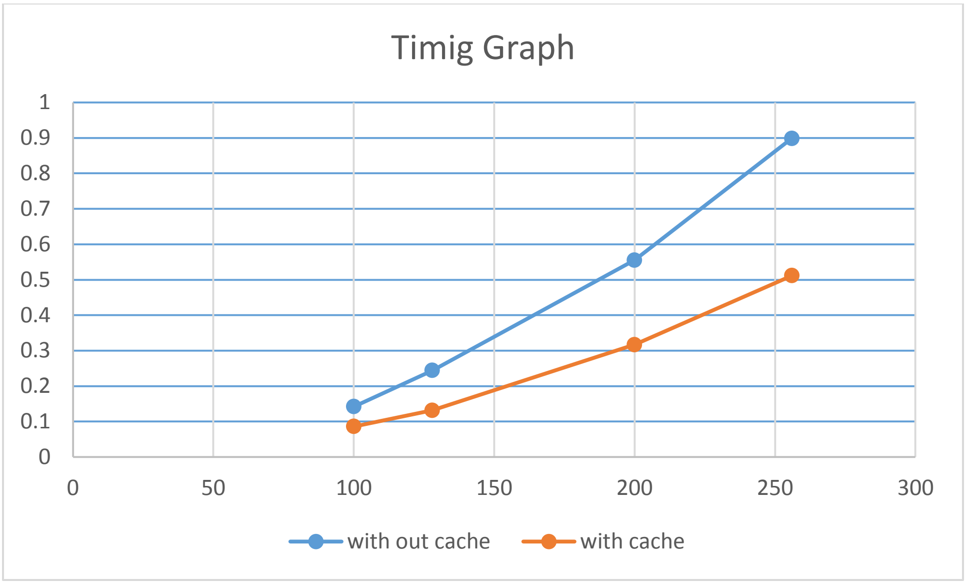
Variation of execution time with image size (without cache)



Square shaped images of varying size was used to analyze the time complexity of down sampling algorithm, performing on the developed processor. These times were observed when the processor was directly connected to the DDR RAM with delay parameter set to 250. A linear trend can be observed from the graph, but execution time has slightly increased from the linearly expected value as well.

Timing analysis comparison (With cache)

Image Size	Delay	Without cache(s)	With cache(s)	Timing efficiency (%)
100 x 100	250	0.1422	0.0859	39.6
128 x 128	250	0.2440	0.1319	46
200 x 200	250	0.5553	0.3171	43
256 x 256	250	0.8987	0.5115	43



7. Discussion

Processor Implementation

A stable UART communication channel was crucial for this project. Therefore several physical connectors were tested. When a USB to RS232 module connected with jumper wires to the FPGA board was used, it was observed that a significant amount of error is introduced to the transmitted data. Then a USB to RS232 converter cable was used and it was connected to the FPGA board using a DB-9 connector port. This was observed to be a more stable channel which is due to the stability of the connection port. To test the stability of the channel, data bytes were transmitted to the FPGA board and they were retransmitted back to the computer. By comparing the differences, stability was determined.

Initially, the processor was designed to contain two different memories for input image and output image. So that the down sampling code could be re executed without re transmitting the input image which takes a significantly long time duration. But due to limitation of dedicated memory blocks of the selected FPGA, according to our memory requirements, a single memory was used to store both the input and the output images. The disadvantage of this implementation is that the input image will be overwritten by the output image once the code is executed. Therefore the input image has to be transmitted for every execution.

The instructions that requires arithmetic operations were designed to be executed through the ALU to minimize resource usage. This process takes two clock cycles to be fully executed. During the first cycle, the required input registers are directed to ALU input by multiplexers and the ALU output will be loaded to the AC register. Then in the second clock cycle the value will be loaded into the required final register from the AC register. This was done during the designing phase, considering the time constraints and the propagation delays related to the given data paths. But after the actual implementation, it was found that the propagation delay related to the complete data flow, starting from input registers, through ALU, to the output register, was within the time constraints. Therefore PASS AC control signal was introduced to bypass the AC register and carry out the instruction within just one clock cycle. This modification was valid for almost half of the instructions of the ISA. Therefore the overall program execution times were greatly reduced.

Cache Implementation

The processor is implemented in a FPGA. Hence the main memory of the processor is implemented using Block rams which has a latency of one clock cycle. Because of that use of a cache memory for the project to increase the throughput is not needed. Hence adding a cache memory will reduce the throughput of the processor. To stimulate the effect of the cache memory the memory architecture has to be changed.

This was done by introducing a wrapper to the existing Block ram where by introducing a delay when addressing a data. By doing so the processor execution time was dependent on the main memory. Then the effect of the cache will be visible.

The cache module was design separately as it is an independent module. To synchronize the data transactions between cache-processor interface and cache-main memory interface AXI handshake method was used.

Increasing the cache line length will increase the time taken for data access when there is a cache miss as all line data in the main memory should be serially transferred to the memory first and then to the processor. But this will increase the hit rate of the cache memory which will increase the processor throughput. Increasing the number of lines will remove number of cache over writes. Hence to balance of the effect the line width of the cache was taken as 4 and the number of lines were taken as 256. Hence the total cache memory was 1024 Bytes.

The cache memory was divided in to sub modules and developed. Using test benching the sub modules were tested and verified and the final design was tested. The cache module was added to the memory module in the top design.

As shown in the results with a memory delay of 250 there is a 40% execution time reduction when cache is implemented. With the size of image increase the execution time increases.

8. Design summary

Total logic elements	15,258 / 114,480 (13 %)
Total combinational functions	10,536 / 114,480 (9 %)
Dedicated logic registers	11,339 / 114,480 (10 %)
Total registers	11339
Total pins	6 / 529 (1 %)
Total virtual pins	0
Total memory bits	2,107,400 / 3,981,312 (53 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Module	**Total logic elements**	**Total combinational functions**	**Dedicated logic registers**	**Total registers**	**Total memory bits**
Processor	551	538	176	176	0
Control Unit	127	126	59	59	0
Data-path	432	420	117	117	0
Memory unit	20187	9615	10928	10928	2,107,400
Cache	869	816	343	343	10,496
DDR_IMAGE_ram	607	557	72	72	2,105,352
INSTRUCTION_ram	1	1	0	0	2,048
UART_FSM	394	278	236	236	0
UART module	123	106	58	58	0

9. Matlab functions used to get FPGA services

The following Matlab functions communicate with the UART FSM to perform the respective services to carry out the down sampling implementation by FPGA.

- program - to load instruction byte array to instruction memory of FPGA
- write_mem - to write instruction/ image data to respective memory locations of FPGA. For this a byte indicating which memory location to write onto, three bytes indicating the size of the following byte array followed by the data byte array is sent to FPGA through serial communication.
- read_mem - to read data from instruction/image data stored on FPGA. For this a byte indicating which memory location is to be read from is sent to FPGA using Matlab serial communication and the resulting serial data byte array is loaded onto Matlab.
- mem_write_verify - to verify the data written onto instruction/ data memory of FPGA. For this write_mem is first implemented followed by read_mem and compared to check if it is the same byte array.
- ImageIn - to create ‘ input_image_bytefile’ (input image byte array) out of input image to be loaded onto FPGA on image memory location.
- load_image - to load image byte array onto image memory of FPGA. Here ‘mem_write_verify ‘function is also made use together with ‘input_image_bytefile’.
- execute - to signal the start of execution of down sampling process. Here a timer is also included to measure the down sampling execution time.
- build_image - to build the processor down sampled image using the received one dimensional data array.
- Gauss_dwnSmpl3_while - to generate the Matlab down sampled reference image, and to calculate the SSD error together with the processor down sampled image.

10.References

[1] Quartus II. Altera Corporation, 2013

[2] MATLAB R2014b. UNITED STATES: The MathWorks, Inc., 2014.

[3]Srivastava, "Comparative study of Salt & Pepper filters and Gaussian filters", Slideshare.net, 2017. [Online]. Available: https://www.slideshare.net/sriAnkush/comparative-study-of-salt-pepper-filters-and-gaussian-filters. [Accessed: 28- Aug- 2017].

[4]"CS 552 Spring 2012", Pages.cs.wisc.edu, 2017. [Online]. Available: http://pages.cs.wisc.edu/~david/courses/cs552/S12/includes/cache-mod.html. [Accessed: 28- Aug- 2017].

[5]"What is cache memory? - Definition from WhatIs.com", SearchStorage, 2017. [Online]. Available: http://searchstorage.techtarget.com/definition/cache-memory. [Accessed: 28- Aug- 2017].

11. Appendix

11.1FPGA Codes

1. Top Module

    1.1. Processor
        1.1.1.Data-path
            1.1.1.1. ALU
            1.1.1.2. Path A Multiplexer
            1.1.1.3. Path B Multiplexer
            1.1.1.4. Register 12
            1.1.1.5. Register 19
        1.1.2.Control Unit
            1.1.2.1. Register 8
    1.2. Memory Unit
        1.2.1.Instruction RAM
        1.2.2.Image DDR RAM
    1.3. UART Control FSM
        1.3.1.UART Module
            1.3.1.1. Baud rate generator
            1.3.1.2. Receiver
            1.3.1.3. Transmitter
    1.4. Cache Memory –Top module
        1.4.1.Cache Controller FSM
        1.4.2.Cache Memory
        1.4.3.Tag Memory
        1.4.4.Valid Memory
        1.4.5.Data Output Multiplexer
        1.4.6.Cache Address selection Multiplexer
        1.4.7.Compare module

Top Module

```
module top_mod

#(parameter INS_ADDR_WIDTH=8, parameter IMG_IN_ADDR_WIDTH=15)

(        input wire clk,

         input wire rx,

         output wire tx,

         input wire rst,

         output wire led_rx,

         output wire led_tx

);


// processor side port a

wire START_FLAG,END_FLAG;


wire [7:0] M_I_addr_CPU;

wire M_I_re_CPU;

wire [7:0] M_I_q_CPU;


wire [18:0] MI_IMG_addr_CPU;

wire MI_IMG_re_CPU;

wire [7:0] MI_IMG_q_CPU;


wire [7:0] MO_IMG_data_CPU;

wire [18:0] MO_IMG_addr_CPU;

wire MO_IMG_we_CPU;


// UART side port b

wire [7:0] M_I_data_UART;

wire [INS_ADDR_WIDTH-1:0] M_I_addr_UART;

wire M_I_we_UART;

wire [7:0] M_I_q_UART;
wire [7:0] MI_IMG_data_UART;

wire [18:0] MI_IMG_addr_UART;

wire MI_IMG_we_UART;

wire [7:0] MI_IMG_q_UART;

wire d_ready_re;

wire d_ready_we;


processor cpu1( clk,

                        rst,

                        START_FLAG,

                        END_FLAG,

                        // instruction memory

                        M_I_q_CPU,

                        M_I_addr_CPU,

                        M_I_re_CPU,

                        // input image memory

                        MI_IMG_q_CPU,

                        MI_IMG_addr_CPU,

                        MI_IMG_re_CPU,


                        d_ready_re,

                        // output image memory

                        MO_IMG_data_CPU,

                        MO_IMG_addr_CPU,

                        MO_IMG_we_CPU,

                        d_ready_we);


memory_unit mu1(    clk,

                        // CPU side port a

                        M_I_addr_CPU[INS_ADDR_WIDTH-1:0],

                        M_I_re_CPU,

                        M_I_q_CPU,
                        MO_IMG_data_CPU,
                        MI_IMG_addr_CPU,
                        MO_IMG_addr_CPU,
                        MI_IMG_re_CPU,
                        MO_IMG_we_CPU,
                        MI_IMG_q_CPU,

                        d_ready_re,
                        d_ready_we,

                        // UART side port b
                        M_I_data_UART,
                        M_I_addr_UART,
                        M_I_we_UART,
                        M_I_q_UART,

                        MI_IMG_data_UART,
                        MI_IMG_addr_UART,
                        MI_IMG_we_UART,
                        MI_IMG_q_UART);

UART_FSM uart_fsm1(clk,
                        rx,
                        tx,
                        rst,
                        led_rx,
                        led_tx,
                        MO_IMG_addr_CPU, // RK value

                        // instruction memory
                        M_I_q_UART,
                        M_I_we_UART,
                        M_I_addr_UART,
                        M_I_data_UART,

                        // input image memory
                        MI_IMG_q_UART,
                        MI_IMG_we_UART,
                        MI_IMG_addr_UART,
                        MI_IMG_data_UART,

                        START_FLAG,
                        END_FLAG);
endmodule
```

Processor

```
module processor(   input wire clk,
                        input wire rst,
                        input wire START_FLAG,
                        output wire END_FLAG,
                        // instruction memory
                        input wire [7:0] M_INS_DATA,
                        output wire [7:0] M_INS_ADD,
                        output wire RD_M_INS,
                        // input image memory
                        input wire [7:0] MI_data,
                        output wire [18:0] MI_add,
                        output wire RD_MI,

                        input wire d_ready_re,

                        // output image memory
                        output wire [7:0] MO_data,
                        output wire [18:0] MO_add,
                        output wire WR_MO,
                        input wire d_ready_we);




wire [2:0] ALU_OP;
wire [2:0] AMUX_sel;
wire [2:0] BMUX_sel;
wire [8:0] LOAD_VECT;
wire [5:0] CLEAR_VECT;
wire PASS_AC;
wire z;
wire z1;
```

```
datapath DP1(   ALU_OP,
                AMUX_sel,
                BMUX_sel,
                LOAD_VECT,
                CLEAR_VECT,
                PASS_AC,
                MO_data,
                MO_add,
                MI_data,
                MI_add,
                clk,
                z,
                z1
                );

control_unit CU1(   clk,
                    rst,
                    START_FLAG,
                    z,
                    z1,
                    RD_M_INS,
                    RD_MI,
                    WR_MO,
                    LOAD_VECT,
                    CLEAR_VECT,
                    AMUX_sel,
                    BMUX_sel,
                    ALU_OP,
                    PASS_AC,
                    END_FLAG,
                    M_INS_DATA,
                    M_INS_ADD,


                    d_ready_re,
                    d_ready_we
                    );
endmodule
```

Data-path

```
module datapath(input [2:0] ALU_OP,
                    input [2:0] AMUX_sel,
                    input [2:0] BMUX_sel,
                    input [8:0] LOAD_VECT,
                    input [5:0] CLEAR_VECT,
                    input PASS_AC,
                    output reg [7:0] MO_data = 0,
                    output reg [18:0] MO_add = 0,
                    input [7:0] MI_data,
                    output reg [18:0] MI_add = 0,
                    input clk,
                    output z,
                    output z1
                    );


reg CLR_RH = 0;
reg CLR_RW = 0;
wire [18:0] AC_out;
wire [11:0] RS_out;
wire [11:0] RI_out;
wire [11:0] RJ_out;
wire [11:0] RH_out;
wire [11:0] RW_out;
wire [18:0] RK_out;
wire [18:0] RX_out;
wire [18:0] C_bus;
wire [18:0] B_bus;
wire [11:0] A_bus;
```

```
register12 RS(.clk(clk), .load(LOAD_VECT[4]), .clear(CLEAR_VECT[1]),
.data_in(AC_out[11:0]), .data_out(RS_out));

register12 RI(.clk(clk), .load(LOAD_VECT[6]), .clear(CLEAR_VECT[3]),
.data_in(AC_out[11:0]), .data_out(RI_out));

register12 RJ(.clk(clk), .load(LOAD_VECT[7]), .clear(CLEAR_VECT[4]),
.data_in(AC_out[11:0]), .data_out(RJ_out));

register12 RH(.clk(clk), .load(LOAD_VECT[2]), .clear(CLR_RH),
.data_in(AC_out[11:0]), .data_out(RH_out));

register12 RW(.clk(clk), .load(LOAD_VECT[3]), .clear(CLR_RW),
.data_in(AC_out[11:0]), .data_out(RW_out));


register19 RK(.clk(clk), .load(LOAD_VECT[8]), .clear(CLEAR_VECT[5]),
.data_in(AC_out), .data_out(RK_out));

register19 RX(.clk(clk), .load(LOAD_VECT[5]), .clear(CLEAR_VECT[2]),
.data_in(AC_out), .data_out(RX_out));


registerAC AC(.clk(clk), .LD_ALU_AC(LOAD_VECT[1]), .LD_MI_AC(LOAD_VECT[0]),
.clear(CLEAR_VECT[0]), .pass(PASS_AC), .data_in_ALU(C_bus),
.data_in_MI(MI_data), .data_out(AC_out), .z(z), .z1(z1));


ALU ALU(.A_bus(A_bus), .B_bus(B_bus), .op(ALU_OP), .C_bus(C_bus));


Amux AMUX(.RS(RS_out), .RI(RI_out), .RJ(RJ_out), .RH(RH_out), .RW(RW_out),
.A_bus(A_bus), .sel(AMUX_sel));

Bmux bmuxtb(.RH(RH_out), .RW(RW_out), .RK(RK_out), .RX(RX_out), .AC(AC_out),
.B_bus(B_bus), .sel(BMUX_sel));


always @(*) begin

    MO_data <= RS_out[7:0];

    MO_add <= RK_out;

    MI_add <= RX_out;


end


endmodule
```

ALU

```
module ALU(A_bus, B_bus, op, C_bus);
    input [11:0] A_bus;
    input [18:0] B_bus;
    input [2:0] op;
    output reg [18:0] C_bus;

    parameter ADD = 3'd0,
              DIV16 = 3'd1,
              SUB = 3'd2,
              INC2 = 3'd3,
              INC1 = 3'd4,
              DEC1 = 3'd5,
              MUL2 = 3'd6,
              MUL4 = 3'd7;
    always@(op or A_bus or B_bus)
        begin
            case(op)
                ADD:
                   begin
                       C_bus = {7'b0, A_bus} + B_bus;
                   end

                DIV16:
                   begin
                       C_bus = {7'b0, A_bus} >> 4;
                   end

                SUB:
                   begin
                       C_bus = B_bus - {7'b0, A_bus};
                   end
                INC2:
                   begin
                       C_bus = {7'b0, A_bus} + 19'd2;
                   end

                INC1:
                   begin
                       C_bus = B_bus + 19'd1;
                   end

                DEC1:
                   begin
                       C_bus = B_bus - 19'd1;
                   end

                MUL2:
                   begin
                       C_bus = B_bus << 1;
                   end

                MUL4:
                   begin
                       C_bus = B_bus << 2;
                   end
           endcase
       end
endmodule
```

A MUX

```
module Amux(RS, RI, RJ, RH, RW, A_bus, sel);
    input [11:0] RS, RI, RJ, RH, RW;
    input [2:0] sel;
    output reg [11:0] A_bus = 0;

    always @(*)
    begin
        case(sel)
            3'b000: A_bus = RS;
            3'b001: A_bus = RI;
            3'b010: A_bus = RJ;
            3'b011: A_bus = RH;
            3'b100: A_bus = RW;
            default: A_bus = 0;
        endcase
    end
endmodule
```

B MUX

```
module Bmux(RH, RW, RK, RX, AC, B_bus, sel); // H,W,K,X,AC
    input [18:0] RK, RX, AC;
    input [11:0]    RH, RW;
    input [2:0] sel;
    output reg [18:0] B_bus = 0;

    always @(*)
    begin
        case(sel)
            3'b000: B_bus = {7'b0, RH};
            3'b001: B_bus = {7'b0, RW};
            3'b010: B_bus = RK;
            3'b011: B_bus = RX;
            3'b100: B_bus = AC;
            default: B_bus = 0;
        endcase
    end
endmodule
```

Register 12

```
module register12(clk, load, clear, data_in, data_out); // RS,RI,RJ,RH,RW
    input clk;
    input load;
    input clear;
    input [11:0] data_in;
    output reg [11:0] data_out = 0;

    always @(negedge clk)
        begin
            if (load) data_out <= data_in;
            else if (clear) data_out <= 12'd0;
        end
endmodule
```

Register 19

```
module register19(clk, load, clear, data_in, data_out); //RX, RK

    input clk;

    input load;

    input clear;

    input [18:0] data_in;

    output reg [18:0] data_out = 0;


    always @(negedge clk)

        begin

            if (load) data_out = data_in;

            else if (clear)data_out = 19'd0;

       end

endmodule




module registerAC(clk, LD_ALU_AC, LD_MI_AC, clear, pass, data_in_ALU,
data_in_MI, data_out, z, z1);

    input clk;

    input LD_ALU_AC;

    input LD_MI_AC;

    input clear;

    input pass;

    input [18:0] data_in_ALU;

    input [7:0] data_in_MI;

    output reg [18:0] data_out = 0;


    reg [18:0] data_out2 = 0;

    output reg z = 1'b0;

    output reg z1 = 1'b1;


    always@(data_out) begin
```

```
        z = (data_out == 19'b0) ? 1'b1 : 1'b0;

        z1 = ((data_out == 19'b0) || (data_out == 19'b1)) ? 1'b0 : 1'b1;

    end


    always @(negedge clk)

        begin

            if (LD_ALU_AC) data_out2 = data_in_ALU;

            else if (LD_MI_AC) data_out2 = {11'b0, data_in_MI};

            else if (clear)data_out2 = 19'd0;

       end


    always @(*)

        begin

            if (pass) data_out <= data_in_ALU;

            else data_out <= data_out2;

        end

endmodule
```

Control Unit

```
module control_unit( input wire clk,
                            input wire rst,
                            input wire START_FLAG,
                            input wire Z,
                            input wire Z1,
                            output reg RD_M_INS = 1'b0,
                            output reg RD_MI = 1'b0,
                            output reg WR_MO = 1'b0,
                            output reg [8:0] LOAD_VECT = 9'b0,
                            output reg [5:0] CLEAR_VECT = 6'b0,
                            output reg [2:0] AMUX = 3'b0,
                            output reg [2:0] BMUX = 3'b0,
                            output reg [2:0] ALU_OP = 3'b0,
                            output reg PASS_AC = 1'b0,
                            output reg END_FLAG = 1'b0,
                            input wire [7:0] M_INS_DATA,
                            output wire [7:0] M_INS_ADD,
                            input wire d_ready_re,
                            input wire d_ready_we
                            );


    parameter START         = 8'd0, CLAC            = 8'd9, DIVRS1      =
8'd19,    DECRX_W        = 8'd28;

    parameter FETCH1        = 8'd1, CLRX            = 8'd10,    SUBRHRJ
= 8'd20,    SHFT1        = 8'd29;

    parameter FETCH2        = 8'd2, CLRS            = 8'd11,    SUBRWRI
= 8'd21,    SHFT2        = 8'd30;

    parameter LDACRX1     = 8'd3,    CLRI            = 8'd12,    INRI1        =
8'd22,    JUMP1        = 8'd31;

    parameter LDACRX2     = 8'd4,    CLRJ            = 8'd13,    INRJ1        =
8'd23,    JUMP2        = 8'd32;

    parameter MVACRH1       = 8'd5, CLRK            = 8'd14,    INRK
= 8'd24,    JMPZ1        = 8'd33;
```

```
    parameter MVACRW1       = 8'd6, ADDRXAC1    = 8'd15,    INRX             =
8'd25,    JMPZ2       = 8'd34;

    parameter STRSRK1       = 8'd7, ADDRXAC2    = 8'd16,    DECRX        =
8'd26,    JMPSP1      = 8'd35;

    parameter STRSRK2       = 8'd8, ADDSUM1         = 8'd17,    INRX_W       =
8'd27,    JMPSP2      = 8'd36;


    parameter ADDSUM2   = 8'd18;

    parameter END           = 8'd37,    END2            = 8'd38;


    parameter SEL_OP = 8'd39;


    reg [7:0] state = START;


    // standard control signals

    reg PASS_IR = 1'b0;

    reg LD_MINS_IR = 1'b0;

    reg LD_IR_PC = 1'b0;

    reg INC_PC = 1'b0;


    // init control signals

    reg CLR_PC = 1'b0;

    reg CLR_IR = 1'b0;


    wire [7:0] IR_PC_DATA ;


    reg d_ready_reg_write = 1'b0;

    reg d_ready_reg_read = 1'b0;

    always @(negedge clk) begin

        if (d_ready_re) begin

            d_ready_reg_read <= 1'b1;

        end

        else if (state == FETCH1) begin
```

```
            d_ready_reg_read <= 1'b0;
        end

        if (d_ready_we) begin
            d_ready_reg_write <= 1'b1;
        end
        else if (state == FETCH1) begin
            d_ready_reg_write <= 1'b0;
        end
    end

    always @(posedge clk) begin
        if (rst == 0) begin
            state <= START;
        end
        case(state)
            START: begin
                if (START_FLAG === 1'b1) begin
                    CLR_PC <= 1'b0;
                    CLR_IR <= 1'b0;

                    state <= FETCH1;
                end
                else begin
                    RD_M_INS    <= 1'b0;    // initialise
                    RD_MI       <= 1'b0;
                    WR_MO       <= 1'b0;
                    LD_MINS_IR  <= 1'b0;
                    LD_IR_PC    <= 1'b0;
                    LOAD_VECT   <= 9'b0;
                    INC_PC      <= 1'b0;
                    CLEAR_VECT  <= 6'b0;
                    AMUX                <= 3'b0;
                    BMUX                <= 3'b0;
                    ALU_OP          <= 3'b0;
                    PASS_AC             <= 1'b0;
                    PASS_IR             <= 1'b0;
                    END_FLAG        <= 1'b0;

                    // clear ir and pc
                    CLR_PC <= 1'b1;
                    CLR_IR <= 1'b1;

                    state <= START;
                end
            end

            FETCH1: begin
                RD_M_INS      <= 1'b1;
                RD_MI         <= 1'b0;
                WR_MO         <= 1'b0;
                LD_MINS_IR    <= 1'b0;
                LD_IR_PC      <= 1'b0;
                LOAD_VECT     <= 9'b0;
                INC_PC        <= 1'b0;
                CLEAR_VECT    <= 6'b0;
                AMUX                <= 3'b0;
                BMUX                <= 3'b0;
                ALU_OP        <= 3'b0;
                PASS_AC             <= 1'b0;
                PASS_IR             <= 1'b0;
                END_FLAG      <= 1'b0;

                state <= FETCH2;
```

```
end

FETCH2: begin
    RD_M_INS    <= 1'b1;
    RD_MI       <= 1'b0;
    WR_MO       <= 1'b0;
    LD_MINS_IR  <= 1'b1;
    LD_IR_PC    <= 1'b0;
    LOAD_VECT   <= 9'b0;
    INC_PC      <= 1'b1;
    CLEAR_VECT  <= 6'b0;
    AMUX            <= 3'b0;
    BMUX            <= 3'b0;
    ALU_OP      <= 3'b0;
    PASS_AC         <= 1'b0;
    PASS_IR         <= 1'b0;
    END_FLAG    <= 1'b0;

    state <= SEL_OP;
end

SEL_OP: begin
    RD_M_INS    <= 1'b0;
    RD_MI       <= 1'b0;
    WR_MO       <= 1'b0;
    LD_MINS_IR  <= 1'b0;
    LD_IR_PC    <= 1'b0;
    LOAD_VECT   <= 9'b0;
    INC_PC      <= 1'b0;
    CLEAR_VECT  <= 6'b0;
    AMUX            <= 3'b0;
    BMUX            <= 3'b0;
```

```
        ALU_OP      <= 3'b0;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= IR_PC_DATA;
    end

    LDACRX1: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b1;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0001;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b0;
        AMUX            <= 3'b0;
        BMUX            <= 3'b0;
        ALU_OP      <= 3'b0;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= LDACRX2;
    end

    LDACRX2: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b1;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        if (d_ready_reg_read) begin
            state <= FETCH1;
        end
        else begin
            state <= LDACRX1;
        end
    end

    MVACRH1: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0100;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    MVACRW1: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_1000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end


    STRSRK1: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b1;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
```

```
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        if (d_ready_reg_write) begin
            state <= FETCH1;
        end
        else begin
            state <= STRSRK1;
        end
    end

    STRSRK2: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b1;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    CLAC: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0001;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    CLRS: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
```

```
        CLEAR_VECT  <= 6'b00_0010;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    CLRX: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0100;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    CLRI: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_1000;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    CLRJ: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b01_0000;
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;
        state <= FETCH1;
    end

    CLRK: begin
        RD_M_INS     <= 1'b0;
        RD_MI        <= 1'b0;
        WR_MO        <= 1'b0;
        LD_MINS_IR   <= 1'b0;
        LD_IR_PC     <= 1'b0;
        LOAD_VECT    <= 9'b0_0000_0000;
        INC_PC       <= 1'b0;
        CLEAR_VECT   <= 6'b10_0000;
        AMUX             <= 3'b000;
        BMUX             <= 3'b000;
        ALU_OP       <= 3'b000;
        PASS_AC          <= 1'b0;
        PASS_IR          <= 1'b0;
        END_FLAG     <= 1'b0;

        state <= FETCH1;
    end

    ADDRXAC1: begin
        RD_M_INS     <= 1'b0;
        RD_MI        <= 1'b0;
        WR_MO        <= 1'b0;
        LD_MINS_IR   <= 1'b0;
        LD_IR_PC     <= 1'b0;
        LOAD_VECT    <= 9'b0_0001_0000;
        INC_PC       <= 1'b0;
        CLEAR_VECT   <= 6'b00_0000;
```

```
        AMUX            <= 3'b000;
        BMUX            <= 3'b000;
        ALU_OP      <= 3'b000;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= ADDRXAC2;
    end

    ADDRXAC2: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0010_0010;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd0;
        BMUX            <= 3'd3;
        ALU_OP      <= 3'd0;
        PASS_AC         <= 1'b1;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    ADDSUM1: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0010;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd0;
        BMUX            <= 3'd4;
        ALU_OP      <= 3'd0;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= ADDSUM2;
    end

    ADDSUM2: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0001_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd0;
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd0;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;
        state <= FETCH1;
    end

    DIVRS1: begin
        RD_M_INS     <= 1'b0;
        RD_MI        <= 1'b0;
        WR_MO        <= 1'b0;
        LD_MINS_IR   <= 1'b0;
        LD_IR_PC     <= 1'b0;
        LOAD_VECT    <= 9'b0_0001_0010;
        INC_PC       <= 1'b0;
        CLEAR_VECT   <= 6'b00_0000;
        AMUX             <= 3'd0;
        BMUX             <= 3'd0;
        ALU_OP       <= 3'd1;
        PASS_AC          <= 1'b1;
        PASS_IR          <= 1'b0;
        END_FLAG     <= 1'b0;

        state <= FETCH1;
    end

    SUBRHRJ: begin
        RD_M_INS     <= 1'b0;
        RD_MI        <= 1'b0;
        WR_MO        <= 1'b0;
        LD_MINS_IR   <= 1'b0;
        LD_IR_PC     <= 1'b0;
        LOAD_VECT    <= 9'b0_0000_0010;
        INC_PC       <= 1'b0;
        CLEAR_VECT   <= 6'b00_0000;
        AMUX             <= 3'd2;
```

```
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd2;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    SUBRWRI: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0010;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd1;
        BMUX            <= 3'd1;
        ALU_OP      <= 3'd2;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    INRI1: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0100_0010;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd1;
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd3;
        PASS_AC         <= 1'b1;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    INRJ1: begin
        RD_M_INS    <= 1'b0;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_1000_0010;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd2;
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd3;
        PASS_AC         <= 1'b1;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
```

```
end

INRK: begin
    RD_M_INS     <= 1'b0;
    RD_MI        <= 1'b0;
    WR_MO        <= 1'b0;
    LD_MINS_IR   <= 1'b0;
    LD_IR_PC     <= 1'b0;
    LOAD_VECT    <= 9'b1_0000_0010;
    INC_PC       <= 1'b0;
    CLEAR_VECT   <= 6'b00_0000;
    AMUX             <= 3'd0;
    BMUX             <= 3'd2;
    ALU_OP       <= 3'd4;
    PASS_AC          <= 1'b1;
    PASS_IR          <= 1'b0;
    END_FLAG     <= 1'b0;

    state <= FETCH1;
end

INRX: begin
    RD_M_INS     <= 1'b0;
    RD_MI        <= 1'b0;
    WR_MO        <= 1'b0;
    LD_MINS_IR   <= 1'b0;
    LD_IR_PC     <= 1'b0;
    LOAD_VECT    <= 9'b0_0010_0010;
    INC_PC       <= 1'b0;
    CLEAR_VECT   <= 6'b00_0000;
    AMUX             <= 3'd0;
    BMUX             <= 3'd3;
    ALU_OP      <= 3'd4;
    PASS_AC         <= 1'b1;
    PASS_IR         <= 1'b0;
    END_FLAG    <= 1'b0;

    state <= FETCH1;
end

DECRX: begin
    RD_M_INS    <= 1'b0;
    RD_MI       <= 1'b0;
    WR_MO       <= 1'b0;
    LD_MINS_IR  <= 1'b0;
    LD_IR_PC    <= 1'b0;
    LOAD_VECT   <= 9'b0_0010_0010;
    INC_PC      <= 1'b0;
    CLEAR_VECT  <= 6'b00_0000;
    AMUX            <= 3'd0;
    BMUX            <= 3'd3;
    ALU_OP      <= 3'd5;
    PASS_AC         <= 1'b1;
    PASS_IR         <= 1'b0;
    END_FLAG    <= 1'b0;

    state <= FETCH1;
end

INRX_W: begin
    RD_M_INS    <= 1'b0;
    RD_MI       <= 1'b0;
    WR_MO       <= 1'b0;
    LD_MINS_IR  <= 1'b0;
    LD_IR_PC     <= 1'b0;
    LOAD_VECT    <= 9'b0_0010_0010;
    INC_PC       <= 1'b0;
    CLEAR_VECT   <= 6'b00_0000;
    AMUX             <= 3'd4;
    BMUX             <= 3'd3;
    ALU_OP       <= 3'd0;
    PASS_AC          <= 1'b1;
    PASS_IR          <= 1'b0;
    END_FLAG     <= 1'b0;

    state <= FETCH1;
end

DECRX_W: begin
    RD_M_INS     <= 1'b0;
    RD_MI        <= 1'b0;
    WR_MO        <= 1'b0;
    LD_MINS_IR   <= 1'b0;
    LD_IR_PC     <= 1'b0;
    LOAD_VECT    <= 9'b0_0010_0010;
    INC_PC       <= 1'b0;
    CLEAR_VECT   <= 6'b00_0000;
    AMUX             <= 3'd4;
    BMUX             <= 3'd3;
    ALU_OP       <= 3'd2;
    PASS_AC          <= 1'b1;
    PASS_IR          <= 1'b0;
    END_FLAG     <= 1'b0;

    state <= FETCH1;
end
```

```
SHFT1: begin
    RD_M_INS     <= 1'b0;
    RD_MI        <= 1'b0;
    WR_MO        <= 1'b0;
    LD_MINS_IR   <= 1'b0;
    LD_IR_PC     <= 1'b0;
    LOAD_VECT    <= 9'b0_0000_0010;
    INC_PC       <= 1'b0;
    CLEAR_VECT   <= 6'b00_0000;
    AMUX             <= 3'd0;
    BMUX             <= 3'd4;
    ALU_OP       <= 3'd6;
    PASS_AC          <= 1'b0;
    PASS_IR          <= 1'b0;
    END_FLAG     <= 1'b0;

    state <= FETCH1;
end

SHFT2: begin
    RD_M_INS     <= 1'b0;
    RD_MI        <= 1'b0;
    WR_MO        <= 1'b0;
    LD_MINS_IR   <= 1'b0;
    LD_IR_PC     <= 1'b0;
    LOAD_VECT    <= 9'b0_0000_0010;
    INC_PC       <= 1'b0;
    CLEAR_VECT   <= 6'b00_0000;
    AMUX             <= 3'd0;
    BMUX             <= 3'd4;
    ALU_OP       <= 3'd7;
```

```
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    JUMP1: begin
        RD_M_INS    <= 1'b1;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b0;
        LD_IR_PC    <= 1'b0;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd0;
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd0;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b0;
        END_FLAG    <= 1'b0;

        state <= JUMP2;
    end

    JUMP2: begin
        RD_M_INS    <= 1'b1;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b1;
        LD_IR_PC    <= 1'b1;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd0;
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd0;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b1;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end

    JMPZ1: begin
        if (Z == 1'b1) begin
            RD_M_INS    <= 1'b1;
            RD_MI       <= 1'b0;
            WR_MO       <= 1'b0;
            LD_MINS_IR  <= 1'b0;
            LD_IR_PC    <= 1'b0;
            LOAD_VECT   <= 9'b0_0000_0000;
            INC_PC      <= 1'b0;
            CLEAR_VECT  <= 6'b00_0000;
            AMUX            <= 3'd0;
            BMUX            <= 3'd0;
            ALU_OP      <= 3'd0;
            PASS_AC         <= 1'b0;
            PASS_IR         <= 1'b0;
            END_FLAG    <= 1'b0;

            state <= JMPZ2;
        end
        else begin
            RD_M_INS    <= 1'b0;
            RD_MI       <= 1'b0;
            WR_MO       <= 1'b0;
            LD_MINS_IR  <= 1'b0;
            LD_IR_PC    <= 1'b0;
            LOAD_VECT   <= 9'b0_0000_0000;
            INC_PC      <= 1'b1;
            CLEAR_VECT  <= 6'b00_0000;
            AMUX            <= 3'd0;
            BMUX            <= 3'd0;
            ALU_OP      <= 3'd0;
            PASS_AC         <= 1'b0;
            PASS_IR         <= 1'b0;
            END_FLAG    <= 1'b0;

            state <= FETCH1;
        end
    end

    JMPZ2: begin
        RD_M_INS    <= 1'b1;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b1;
        LD_IR_PC    <= 1'b1;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd0;
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd0;
```

```
        PASS_AC         <= 1'b0;

        PASS_IR         <= 1'b1;

        END_FLAG    <= 1'b0;


        state <= FETCH1;

    end


    JMPSP1: begin

        if (Z1 == 1'b1) begin

            RD_M_INS    <= 1'b1;

            RD_MI       <= 1'b0;

            WR_MO       <= 1'b0;

            LD_MINS_IR  <= 1'b0;

            LD_IR_PC    <= 1'b0;

            LOAD_VECT   <= 9'b0_0000_0000;

            INC_PC      <= 1'b0;

            CLEAR_VECT  <= 6'b00_0000;

            AMUX            <= 3'd0;

            BMUX            <= 3'd0;

            ALU_OP      <= 3'd0;

            PASS_AC         <= 1'b0;

            PASS_IR         <= 1'b0;

            END_FLAG    <= 1'b0;


            state <= JMPSP2;

        end

        else begin

            RD_M_INS    <= 1'b0;

            RD_MI       <= 1'b0;

            WR_MO       <= 1'b0;

            LD_MINS_IR  <= 1'b0;

            LD_IR_PC    <= 1'b0;
            LOAD_VECT   <= 9'b0_0000_0000;
            INC_PC      <= 1'b1;
            CLEAR_VECT  <= 6'b00_0000;
            AMUX            <= 3'd0;
            BMUX            <= 3'd0;
            ALU_OP      <= 3'd0;
            PASS_AC         <= 1'b0;
            PASS_IR         <= 1'b0;
            END_FLAG    <= 1'b0;

            state <= FETCH1;
        end
    end

    JMPSP2: begin
        RD_M_INS    <= 1'b1;
        RD_MI       <= 1'b0;
        WR_MO       <= 1'b0;
        LD_MINS_IR  <= 1'b1;
        LD_IR_PC    <= 1'b1;
        LOAD_VECT   <= 9'b0_0000_0000;
        INC_PC      <= 1'b0;
        CLEAR_VECT  <= 6'b00_0000;
        AMUX            <= 3'd0;
        BMUX            <= 3'd0;
        ALU_OP      <= 3'd0;
        PASS_AC         <= 1'b0;
        PASS_IR         <= 1'b1;
        END_FLAG    <= 1'b0;

        state <= FETCH1;
    end
```

```
        END: begin
            RD_M_INS    <= 1'b0;
            RD_MI       <= 1'b0;
            WR_MO       <= 1'b0;
            LD_MINS_IR  <= 1'b0;
            LD_IR_PC    <= 1'b0;
            LOAD_VECT   <= 9'b0_0000_0000;
            INC_PC      <= 1'b0;
            CLEAR_VECT  <= 6'b00_0000;
            AMUX            <= 3'd0;
            BMUX            <= 3'd0;
            ALU_OP      <= 3'd0;
            PASS_AC         <= 1'b0;
            PASS_IR         <= 1'b0;
            END_FLAG    <= 1'b1;

            state <= END2;
        end

        END2: begin
            state <= START;
        end

        default: begin
            state <= START;
        end
    endcase
end

registerIR IR(clk, LD_MINS_IR, CLR_IR, PASS_IR, M_INS_DATA, IR_PC_DATA);
registerPC PC(clk, LD_IR_PC, CLR_PC, INC_PC, IR_PC_DATA, M_INS_ADD);
```

```
endmodule
```

Register 8

```
module registerIR(clk, load, clear,pass, data_in, data_out); // IR and PC
    input clk;
    input load;
    input clear;
    input pass;
    input [7:0] data_in;
    output [7:0] data_out = 8'b0;

    reg [7:0] data_out = 8'b0;
    reg [7:0] data_out2 = 8'b0;


    always @(negedge clk)
        begin
            if (load) data_out2 <= data_in;
            else if (clear) data_out2 <= 8'd0;
            else data_out2 <= data_out2;
       end

    always @(*)
        begin
            if (pass) data_out <= data_in;
            else data_out <= data_out2;
        end
endmodule

module registerPC(clk, load, clear, inc, data_in, data_out); // IR and PC
    input clk;
    input load;
    input clear;
    input inc;
    input [7:0] data_in;
    output [7:0] data_out = 8'b0;

    reg [7:0] data_out = 8'b0;

    always @(negedge clk)
        begin
            if (load) data_out <= data_in;
            else if (clear) data_out <= 8'd0;
            else if (inc)  data_out <= data_out + 8'd1;
            else data_out <= data_out;
       end

endmodule
```

Memory Unit

```
module memory_unit( input wire clk,
                    // CPU side port a
                    input wire [7:0] M_I_addr_CPU,
                    input wire M_I_re_CPU,
                    output wire [7:0] M_I_q_CPU,

                    input wire [7:0] MI_IMG_data_CPU,
                    input wire [18:0] MI_IMG_addr_CPU_read,
                    input wire [18:0] MI_IMG_addr_CPU_write,
                    input wire MI_IMG_re_CPU,
                    input wire MI_IMG_we_CPU,
                    output wire [7:0] MI_IMG_q_CPU,

                    output wire d_ready_re,
                    output wire d_ready_we,

                    // UART side port b
                    input wire [7:0] M_I_data_UART,
                    input wire [7:0] M_I_addr_UART,
                    input wire M_I_we_UART,
                    output wire [7:0] M_I_q_UART,

                    input wire [7:0] MI_IMG_data_UART,
                    input wire [18:0] MI_IMG_addr_UART,
                    input wire MI_IMG_we_UART,
                    output wire [7:0] MI_IMG_q_UART);

wire [18:0] MI_IMG_addr_CPU;
wire [7:0]  CH_MI_IMG_DATA_CPU;
wire [18:0] CH_MI_IMG_ADDR_CPU;
wire        CH_MI_IMG_WEA_CPU;
wire        CH_MI_IMG_REA_CPU;

wire  [7:0] MI_CH_IMG_DATA_q_CPU;

wire        MI_CH_IMG_READ_READY_CPU;

wire        MI_CH_IMG_WRITE_REEADY_CPU;


assign MI_IMG_addr_CPU = (MI_IMG_we_CPU == 1'b1) ? MI_IMG_addr_CPU_write :
MI_IMG_addr_CPU_read;


ins_ram M_INS(M_I_data_UART,

                    M_I_addr_CPU,M_I_addr_UART,

                    M_I_re_CPU, M_I_we_UART,

                    clk,

                    M_I_q_CPU,M_I_q_UART);


img_in_ddr MI_IMG(CH_MI_IMG_DATA_CPU,MI_IMG_data_UART,

                        CH_MI_IMG_ADDR_CPU,MI_IMG_addr_UART,

                        CH_MI_IMG_WEA_CPU, MI_IMG_we_UART, CH_MI_IMG_REA_CPU,

                        clk,

                        MI_CH_IMG_DATA_q_CPU,MI_IMG_q_UART,

                        MI_CH_IMG_WRITE_REEADY_CPU,MI_CH_IMG_READ_READY_CPU);


CacheTop CacheModule(

    .Clk(clk),

    .ResetFlag(0),


    .ProcessorWriteEnIn(MI_IMG_we_CPU),

    .ProcessorReadEnIn(MI_IMG_re_CPU),

    .ProcessorDataIn(MI_IMG_data_CPU),

    .ProcessorAddressIn(MI_IMG_addr_CPU),


    .ProcessorWriteReadyOut(d_ready_we),

    .ProcessorReadReadyOut(d_ready_re),
    .ProcessorDataOut(MI_IMG_q_CPU),



    .DdrWriteReadyIn(MI_CH_IMG_WRITE_REEADY_CPU),
    .DdrReadReadyIn(MI_CH_IMG_READ_READY_CPU),
    .DdrDataIn(MI_CH_IMG_DATA_q_CPU),

    .DdrWriteEnOut(CH_MI_IMG_WEA_CPU),
    .DdrReadEnOut(CH_MI_IMG_REA_CPU),
    .DdrDataOut(CH_MI_IMG_DATA_CPU),
    .DdrAddressOut(CH_MI_IMG_ADDR_CPU)
    );

endmodule
```

Instruction RAM

```
module ins_ram
(
    input [7:0] data_b,
    input [7:0] addr_a, addr_b,
    input we_a, we_b, clk,
    output reg [7:0] q_a, q_b
);

    // Declare the RAM variable
    reg [7:0] ram[255:0];

    // Port A
    always @ (posedge clk)
    begin
        if (we_a)
        begin
            q_a <= ram[addr_a];
        end
        else
        begin
            q_a <= ram[addr_a];
        end
    end

    // Port B
    always @ (posedge clk)
    begin
        if (we_b)
        begin
            ram[addr_b] <= data_b;
            q_b <= data_b;
        end
        else
        begin
            q_b <= ram[addr_b];
        end
    end

endmodule
```

Image DDR RAM

```
module img_in_ddr
(
    input [7:0] data_a, data_b,
    input [18:0] addr_a, addr_b,
    input we_a, we_b, re_a, clk,
    output reg [7:0] q_a, q_b,
    output reg d_ready_we = 1'b0,
    output reg d_ready_re = 1'b0
);

    // Declare the RAM variable
    reg [7:0] ram[263168:0];
    parameter DELAY = 8'd250;
    reg [18:0] prev_addr;
    reg [18:0] prev_addr_a;
    reg [18:0] prev_addr_a1 = 19'b0;
    reg [18:0] prev_addr_a2 = 19'b0;
    reg addrs_change_flag_re = 1'b0;
    reg addrs_change_flag_we = 1'b0;
    reg [7:0] delay_temp_re = 8'b0;
    reg [7:0] delay_temp_we = 8'b0;
    reg we_flag = 1'b0;
    reg re_flag = 1'b0;

    always @ (posedge clk)
    begin

    if (addrs_change_flag_re == 1'b1 && re_flag == 1 ) begin
        if (addr_a == prev_addr_a2+ 19'd1)begin
                d_ready_re <= 1'b1;
        end
        else
        begin
            delay_temp_re <= delay_temp_re + 8'b1;
            if (DELAY == delay_temp_re) begin
                d_ready_re <= 1'b1;
                delay_temp_re <= 8'b0;
            end
        end
    end

    if (addrs_change_flag_we == 1'b1 && we_flag == 1) begin
        if (addr_a == prev_addr_a2 + 19'd1)begin
                d_ready_we <= 1'b1;
        end

        else
        begin
            delay_temp_we <= delay_temp_we + 8'b1;
            if (DELAY == delay_temp_we) begin
                d_ready_we <= 1'b1;
                delay_temp_we <= 8'b0;
            end
        end
    end

    if ((we_flag == 1'b1 ) && addrs_change_flag_we== 1'b0) begin
            d_ready_we <= 1'b1;
            d_ready_re <= 1'b0;
    end

    if ((re_flag == 1'b1 ) && addrs_change_flag_re== 1'b0) begin
            d_ready_we <= 1'b0;
            d_ready_re <= 1'b1;
    end

    if (we_a == 1'b1) begin
            we_flag <= 1'b1;
    end

    if (re_a == 1'b1) begin
            re_flag <= 1'b1;
    end

    prev_addr_a <= addr_a;
    prev_addr_a1 <= prev_addr_a;

    if (addr_a != prev_addr_a1)begin
        addrs_change_flag_re <= 1'b1;
        addrs_change_flag_we <= 1'b1;
    end

    if (d_ready_we == 1'b1)
        begin
            d_ready_we <= 1'b0;
            we_flag <= 1'b0;
            addrs_change_flag_we <= 1'b0;
            delay_temp_we <= 8'b0;
        end

    if (d_ready_re == 1'b1)
        begin
            addrs_change_flag_re <= 1'b0;
            delay_temp_re <= 8'b0;
            d_ready_re <= 1'b0;
```

```
            re_flag <= 1'b0;
        end
        end
        // PORT A
//*********************

    always @ (posedge clk)
    begin
        if (we_a) begin
            ram[addr_a] <= data_a;
            q_a <= data_a;

        end
        else
        begin
            q_a <= ram[addr_a];
        end
        end

        // Port B
//*********************
always @ (posedge clk)
    begin
        if (we_b)
        begin
            ram[addr_b] <= data_b;
            q_b <= data_b;
        end
        else
        begin
            q_b <= ram[addr_b];
        end
    end



always @(addr_a)begin
     prev_addr <= addr_a;
     prev_addr_a2 <= prev_addr;
end
endmodule
```

UART FSM

```
module UART_FSM
#(parameter INS_ADDR_WIDTH=8, parameter IMG_IN_ADDR_WIDTH=19)
(       input wire clk,
        input wire rx,
        output wire tx,
        input wire rst,
        output reg led_rx = 1'b1,
        output reg led_tx = 1'b1,
        input wire [18:0] RK_val,
        // instruction memory
        input wire [7:0] ram_ins_out,
        output reg ram_ins_we = 0,
        output reg [INS_ADDR_WIDTH-1:0] ram_ins_addr= 0,
        output reg [7:0] ram_ins_in = 0,
        // input image memory
        input wire [7:0] ram_data_out_img_in,
        output reg ram_we_img_in = 0,
        output reg [IMG_IN_ADDR_WIDTH-1:0] ram_addr_img_in = 0,
        output reg [7:0] ram_data_in_img_in = 0,

        output reg start_flag = 1'b0,
        input wire end_flag);

// main states
parameter STATE_CHECK_RX= 5'd0;

parameter STATE_RX1 = 5'd1;
parameter STATE_RX2 = 5'd2;
parameter STATE_RX3 = 5'd3;

parameter STATE_RX_DATA1= 5'd4;
parameter STATE_RX_DATA2= 5'd5;
parameter STATE_RX_DATA3= 5'd6;
parameter STATE_RX_DATA4= 5'd7;
parameter STATE_RX_DATA5= 5'd8;

parameter STATE_TX1 = 5'd9;

parameter STATE_TX_DATA1 = 5'd10;
parameter STATE_TX_DATA2 = 5'd11;
parameter STATE_TX_DATA3 = 5'd12;
parameter STATE_TX_DATA4 = 5'd13;

parameter STATE_TX_BUSY = 5'd14;

parameter STATE_START1 = 5'd15;
parameter STATE_START2 = 5'd16;

parameter STATE_END1 = 5'd17;
parameter STATE_END2 = 5'd18;
parameter STATE_END3 = 5'd19;

// next_state register with initial next_state
reg [4:0] curr_state = STATE_CHECK_RX;
reg [4:0] next_state = STATE_CHECK_RX;

// connectors for serial interface
wire tx_busy_flag;
wire rx_rdy_flag;
wire [7:0] ser_data_out;

reg [7:0] ser_data_in;
reg rx_rdy_clr = 0;
reg ser_wr_en = 0;

// ram selection values
parameter IMG_IN_RAM = 2'b0;
parameter INS_RAM = 2'b10;

reg [1:0] ram_sel = IMG_IN_RAM;

//other
reg [23:0] len_cnt = 0;

reg [23:0] ins_len = 0;

reg [23:0] input_img_len = 0;

reg [23:0] output_read_img_len = 0; // from processor
reg imgo_cnt_sel = 0; // zero for normal , 1 - for processed out

parameter CMD_DEC = 2'd0;
parameter CMD_RX = 2'd1;
parameter CMD_TX = 2'd2;

reg [1:0] mode = CMD_DEC;

parameter BYTE_RX_INS = 8'd255;
parameter BYTE_TX_INS = 8'd254;

parameter BYTE_RX_IMG_I = 8'd253;
parameter BYTE_TX_IMG_I = 8'd252;

parameter BYTE_READ_IMG_O = 8'd249;
parameter BYTE_START = 8'd240;
parameter BYTE_END = 8'd239;

always @(posedge clk) begin
    if (rst == 0) begin
        curr_state <= STATE_CHECK_RX;
        next_state <= STATE_CHECK_RX;
        mode <= CMD_DEC;
        rx_rdy_clr <= 1'b0;
        ser_wr_en <= 1'b0;

        led_rx <= 1'b1;
        led_tx <= 1'b1;
    end

    case(curr_state)
        STATE_CHECK_RX: begin
            if(rx_rdy_flag && !rx_rdy_clr) begin
                led_rx <= 1'b0;
                case(mode)      // select mode
                    CMD_DEC: begin
                        case(ser_data_out)
                            BYTE_RX_IMG_I: begin        // receive image to
processor
                                next_state <= STATE_RX1;
                                curr_state <= STATE_CHECK_RX;
                                mode <= CMD_RX;

                                ram_sel <= IMG_IN_RAM;
                            end
                            BYTE_TX_IMG_I: begin        // transmit image
from processor
                                curr_state <= STATE_TX1;
                                mode <= CMD_TX;

                                ram_sel <= IMG_IN_RAM;
                                imgo_cnt_sel <= 1'b0;
                            end

                            BYTE_READ_IMG_O: begin      // transmit image
from processor
                                curr_state <= STATE_TX1;
                                mode <= CMD_TX;

                                ram_sel <= IMG_IN_RAM;
                                imgo_cnt_sel <= 1'b1;
                            end
                            BYTE_RX_INS: begin
                                next_state <= STATE_RX1;
                                curr_state <= STATE_CHECK_RX;
                                mode <= CMD_RX;

                                ram_sel <= INS_RAM;
                            end
                            BYTE_TX_INS: begin      // transmit image from
processor
                                curr_state <= STATE_TX1;
                                mode <= CMD_TX;

                                ram_sel <= INS_RAM;
                            end

                            BYTE_START: begin       // start processing
                                curr_state <= STATE_START1;
                                start_flag <= 1'b1;
                            end
```

```
                        default: begin
                            curr_state <= STATE_CHECK_RX;
                        end
                    endcase
                end
                CMD_RX: begin
                    curr_state <= next_state;
                end
                CMD_TX: begin
                    curr_state <= next_state;
                end
                default: begin
                    curr_state <= STATE_CHECK_RX;
                end
            endcase

            rx_rdy_clr <= 1'b1;
        end
        else if(end_flag) begin
            curr_state <= STATE_END1;

            output_read_img_len <= {5'b0, RK_val} + 24'b1;  // assuming
RK goes from zero to last address
        end
        else begin
            rx_rdy_clr <= 1'b0;
            curr_state <= STATE_CHECK_RX;
            led_rx <= 1'b1;
        end
    end
```

```
STATE_RX1: begin        // MSB1

    rx_rdy_clr <= 1'b0;

    next_state <= STATE_RX2;

    curr_state <= STATE_CHECK_RX;


    if (ram_sel == INS_RAM) begin

        ins_len[23:16] <= ser_data_out;

    end

    else begin

        input_img_len[23:16] <= ser_data_out;

    end

end


STATE_RX2: begin        // MSB2

    next_state <= STATE_RX3;

    curr_state <= STATE_CHECK_RX;


    if (ram_sel == INS_RAM) begin

        ins_len[15:8] <= ser_data_out;

    end

    else begin

        input_img_len[15:8] <= ser_data_out;

    end

end


STATE_RX3: begin        // MSB3

    next_state <= STATE_RX_DATA1;

    curr_state <= STATE_CHECK_RX;


    if (ram_sel == INS_RAM) begin

        ins_len[7:0] <= ser_data_out;

    end
    else begin
        input_img_len[7:0] <= ser_data_out;
    end
    len_cnt <= 24'b0;
end

STATE_RX_DATA1: begin       // write data to ram
    curr_state <= STATE_RX_DATA2;

    if (ram_sel == INS_RAM) begin
        ram_ins_addr <= len_cnt[INS_ADDR_WIDTH-1:0];
        ram_ins_in <= ser_data_out;
        ram_ins_we <= 1'b1;
    end
    else begin
        ram_addr_img_in <= len_cnt[IMG_IN_ADDR_WIDTH-1:0];
        ram_data_in_img_in <= ser_data_out;
        ram_we_img_in <= 1'b1;
    end
end

STATE_RX_DATA2: begin       // idle cycle
    curr_state <= STATE_RX_DATA3;
end

STATE_RX_DATA3: begin       // write data to ram
    curr_state <= STATE_RX_DATA4;

    if (ram_sel == IMG_IN_RAM) begin
        ram_we_img_in <= 1'b0;
    end
    else begin
```

```
                ram_ins_we <= 1'b0;

            end

        end


        STATE_RX_DATA4: begin

            curr_state <= STATE_RX_DATA5;

            len_cnt <= len_cnt + 24'b1;

        end


        STATE_RX_DATA5: begin


            if(((len_cnt == input_img_len)&&(ram_sel == IMG_IN_RAM)) ||

                ((len_cnt == ins_len)&&(ram_sel == INS_RAM)) ) begin


                curr_state <= STATE_CHECK_RX;

                mode <= CMD_DEC;

            end

            else begin

                next_state <= STATE_RX_DATA1;

                curr_state <= STATE_CHECK_RX;

            end


        end

//*************************************************************************


        STATE_TX1: begin        // Initialise

            rx_rdy_clr <= 1'b0;

            curr_state <= STATE_TX_DATA1;


            len_cnt <= 24'b0;

        end
```

```
STATE_TX_DATA1: begin
    led_tx <= 1'b0;
    curr_state <= STATE_TX_BUSY;

    if (ram_sel == IMG_IN_RAM) begin
        ram_addr_img_in <= len_cnt[IMG_IN_ADDR_WIDTH-1:0];
    end
    else begin
        ram_ins_addr <= len_cnt[INS_ADDR_WIDTH-1:0];
    end
end

STATE_TX_BUSY: begin
    if(tx_busy_flag) begin
        curr_state <= STATE_TX_BUSY;
    end
    else begin
        curr_state <= STATE_TX_DATA2;
    end
end

STATE_TX_DATA2: begin
    curr_state <= STATE_TX_DATA3;

    if (ram_sel == IMG_IN_RAM) begin
        ser_data_in <= ram_data_out_img_in;
    end
    else begin
        ser_data_in <= ram_ins_out;
    end
    ser_wr_en <= 1'b1;
end
```

```
        STATE_TX_DATA3: begin
            curr_state <= STATE_TX_DATA4;

            ser_wr_en <= 1'b0;
            len_cnt <= len_cnt + 24'b1;
        end

        STATE_TX_DATA4: begin
            if (((len_cnt == input_img_len)&&(ram_sel ==
IMG_IN_RAM)&&(imgo_cnt_sel == 1'b0)) ||
                ((len_cnt == ins_len)&&(ram_sel == INS_RAM)) ||
                ((len_cnt == output_read_img_len)&&(ram_sel ==
IMG_IN_RAM)&&(imgo_cnt_sel == 1'b1))) begin

                curr_state <= STATE_CHECK_RX;
                mode <= CMD_DEC;
            end
            else begin
                curr_state <= STATE_TX_DATA1;
            end
            led_tx <= 1'b1;
        end

        STATE_START1: begin
            rx_rdy_clr <= 1'b0;
            curr_state <= STATE_START2;
        end

        STATE_START2: begin
            curr_state <= STATE_CHECK_RX;
            start_flag <= 1'b0;
        end
        STATE_END1: begin
            rx_rdy_clr <= 1'b0;

            if(tx_busy_flag) begin
                curr_state <= STATE_END1;
            end
            else begin
                curr_state <= STATE_END2;
                ser_data_in <= 8'd239;
                ser_wr_en <= 1'b1;
            end
        end

        STATE_END2: begin
            curr_state <= STATE_END3;
        end

        STATE_END3: begin       // additional delay to avoid sending two
flags
            curr_state <= STATE_CHECK_RX;
            ser_wr_en <= 1'b0;
        end

        default: begin
            curr_state <= STATE_CHECK_RX;
        end
    endcase
end

    uart uart1(ser_data_in,
        ser_wr_en,
        clk,

        tx,

        tx_busy_flag,

        rx,

        rx_rdy_flag,

        rx_rdy_clr,

        ser_data_out);


endmodule
```

UART Module

```
module uart(input wire [7:0] din,
        input wire wr_en,
        input wire clk_50m,
        output wire tx,
        output wire tx_busy,
        input wire rx,
        output wire rdy,
        input wire rdy_clr,
        output wire [7:0] dout);

wire rxclk_en, txclk_en;

baud_rate_gen uart_baud(.clk_50m(clk_50m),
            .rxclk_en(rxclk_en),
            .txclk_en(txclk_en));
transmitter uart_tx(.din(din),
            .wr_en(wr_en),
            .clk_50m(clk_50m),
            .clken(txclk_en),
            .tx(tx),
            .tx_busy(tx_busy));
receiver uart_rx(.rx(rx),
         .rdy(rdy),
         .rdy_clr(rdy_clr),
         .clk_50m(clk_50m),
         .clken(rxclk_en),
         .data(dout));

endmodule
```

Baud rate Generator

```
module baud_rate_gen(input wire clk_50m,
             output wire rxclk_en,
             output wire txclk_en);

parameter RX_ACC_MAX = 50000000 / (115200 * 16);
parameter TX_ACC_MAX = 50000000 / 115200;
parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);
parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);
reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;
reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;

assign rxclk_en = (rx_acc == 5'd0);
assign txclk_en = (tx_acc == 9'd0);

always @(posedge clk_50m) begin
    if (rx_acc == RX_ACC_MAX[RX_ACC_WIDTH - 1:0])
        rx_acc <= 0;
    else
        rx_acc <= rx_acc + 5'b1;
end
always @(posedge clk_50m) begin
    if (tx_acc == TX_ACC_MAX[TX_ACC_WIDTH - 1:0])
        tx_acc <= 0;
    else
        tx_acc <= tx_acc + 9'b1;
end

endmodule
```

Receiver

```
module receiver(input wire rx,
        output reg rdy,
        input wire rdy_clr,
        input wire clk_50m,
        input wire clken,
        output reg [7:0] data);

initial begin
    rdy = 0;
    data = 8'b0;
end

parameter RX_STATE_START    = 2'b00;
parameter RX_STATE_DATA     = 2'b01;
parameter RX_STATE_STOP     = 2'b10;

reg [1:0] state = RX_STATE_START;
reg [3:0] sample = 0;
reg [3:0] bitpos = 0;
reg [7:0] scratch = 8'b0;

always @(posedge clk_50m) begin
    if (rdy_clr)
        rdy <= 0;
        if (clken) begin
            case (state)
            RX_STATE_START: begin
                if (!rx || sample != 0)
                    sample <= sample + 4'b1;

                if (sample == 15) begin
```

```
                    state <= RX_STATE_DATA;
                    bitpos <= 0;
                    sample <= 0;
                    scratch <= 0;
                end
            end
            RX_STATE_DATA: begin
                sample <= sample + 4'b1;
                if (sample == 4'h8) begin
                    scratch[bitpos[2:0]] <= rx;
                    bitpos <= bitpos + 4'b1;
                end
                if (bitpos == 8 && sample == 15)
                    state <= RX_STATE_STOP;
            end
            RX_STATE_STOP: begin
                if (sample == 15 || (sample >= 8 && !rx)) begin
                    state <= RX_STATE_START;
                    data <= scratch;
                    rdy <= 1'b1;
                    sample <= 0;
                end else begin
                    sample <= sample + 4'b1;
                end
            end
            default: begin
                state <= RX_STATE_START;
            end
            endcase
        end
end
```

```
endmodule
```

Transmitter

```
module transmitter(input wire [7:0] din,
           input wire wr_en,
           input wire clk_50m,
           input wire clken,
           output reg tx,
           output wire tx_busy);

initial begin
     tx = 1'b1;
end

parameter STATE_IDLE    = 2'b00;
parameter STATE_START   = 2'b01;
parameter STATE_DATA    = 2'b10;
parameter STATE_STOP    = 2'b11;
reg [7:0] data = 8'h00;
reg [2:0] bitpos = 3'h0;
reg [1:0] state = STATE_IDLE;

always @(posedge clk_50m) begin
    case (state)
    STATE_IDLE: begin
        if (wr_en) begin
            state <= STATE_START;
            data <= din;
            bitpos <= 3'h0;
        end
    end
    STATE_START: begin
        if (clken) begin
            tx <= 1'b0;
            state <= STATE_DATA;
        end
    end
    STATE_DATA: begin
        if (clken) begin
            if (bitpos == 3'h7)
                state <= STATE_STOP;
            else
                bitpos <= bitpos + 3'h1;
            tx <= data[bitpos];
        end
    end
    STATE_STOP: begin
        if (clken) begin
            tx <= 1'b1;
            state <= STATE_IDLE;
        end
    end
    default: begin
        tx <= 1'b1;
        state <= STATE_IDLE;
    end
    endcase
end

assign tx_busy = (state != STATE_IDLE);

endmodule
```

Cache Top Module

```
module CacheTop#(
parameter      DATA_WIDTH                =8,
parameter     ADDRESS_LENGTH           =19,
parameter     TAG_LENGTH               =9,
parameter     CASHE_ADDRESS_LENGTH     =10,
parameter     CASHE_BIT_DEPTH          =2

)
(
input                                  Clk,
input                                  ResetFlag,

input                                  ProcessorWriteEnIn,
input                                  ProcessorReadEnIn,
input       [(DATA_WIDTH-1):0]         ProcessorDataIn,
input       [(ADDRESS_LENGTH-1):0]     ProcessorAddressIn,

output                                 ProcessorWriteReadyOut,
output                                 ProcessorReadReadyOut,
output      [(DATA_WIDTH-1):0]         ProcessorDataOut,


input                                  DdrWriteReadyIn,
input                                  DdrReadReadyIn,
input           [(DATA_WIDTH-1):0]     DdrDataIn,

output                                 DdrWriteEnOut,
output                                 DdrReadEnOut,
output      [(DATA_WIDTH-1):0]         DdrDataOut,
output     [(ADDRESS_LENGTH-1):0]      DdrAddressOut,
output                                 cache_hit,
output                                 cache_miss
);
```

```
wire  [(ADDRESS_LENGTH-CASHE_ADDRESS_LENGTH-1):0]Tag_Compare_AddressBUS;
wire                                            Compare_Fsm_WIRE;
wire                                            Valid_Fsm_WIRE;
wire                                            Fsm_CacheMem_WriteEn_Wire;
wire                                            Fsm_DataOutMux_Select_Wire;
wire
Fsm_CacheAddressMux_Select_Wire;
wire                                            Fsm_TagMem_WriteEn_Wire;
wire                                            Fsm_ValidMem_WriteEn_Wire;
wire                                            Fsm_ValidMem_DataBit_Wire;
wire  [(CASHE_ADDRESS_LENGTH-1):0]
Fsm_CacheAddressMux_AddressBUS;
wire  [(CASHE_ADDRESS_LENGTH-1):0]
CacheAddressMux_CacheMem_AddressBUS;
wire  [(DATA_WIDTH-1):0]
CacheMem_CacheAddressMux_DataBUS;
wire                                            InterConnect;


assign ProcessorDataIn=InterConnect;
assign InterConnect=DdrDataOut;


//control Element FSM
FSMControl FSMControl(
    .WriteIn                        (ProcessorWriteEnIn),
    .ReadIn                         (ProcessorReadEnIn),
    .Match                          (Compare_Fsm_WIRE),
    .Valid                          (Valid_Fsm_WIRE),
    .Clk                            (Clk),
    .ResetFlag                      (ResetFlag),
    .Memory_Cache_Read_Ready        (DdrReadReadyIn),
    .Memory_Cache_Write_Ready       (DdrWriteReadyIn),
    .Processor_Cache_Address        (ProcessorAddressIn),
    .Cache_Write_Enable               (Fsm_CacheMem_WriteEn_Wire),
// 1= write to cashs

    .Select_DataOut_Memory_cache(Fsm_DataOutMux_Select_Wire),        // 1 =
memory

    .Select_CacheAddrss_FSM_Address(Fsm_CacheAddressMux_Select_Wire),    // 0
= processor 1= fsm address

    .Ram_Write                        (DdrWriteEnOut),    // 0 = memory 1= fsm
addres

    .Ram_Read                         (DdrReadEnOut),

    .Tag_Write                        (Fsm_TagMem_WriteEn_Wire),

    .Valid_Write                      (Fsm_ValidMem_WriteEn_Wire),

    .Valid_Bit                        (Fsm_ValidMem_DataBit_Wire),

    .Cache_Processor_Read_Ready       (ProcessorReadReadyOut),

    .Cache_Processor_Write_Ready      (ProcessorWriteReadyOut),

    .Cache_RAMportB_Address           (DdrAddressOut),

    .Cache_DataCache_Address          (Fsm_CacheAddressMux_AddressBUS),
//10 bits

    .cache_hit(cache_hit),

    .cache_miss(cache_miss)


);





//RAMS VAlid CacheMen and Tag

TagRam TagRam (

    .Address                      (ProcessorAddressIn[9:2]),    //only 8 bits
from the total 10 TAG = 10-2*

    .TagIn                        (ProcessorAddressIn[(ADDRESS_LENGTH-
1):CASHE_ADDRESS_LENGTH]),

    .Write                        (Fsm_TagMem_WriteEn_Wire),

    .Clk                          (Clk),


    .TagOut                       (Tag_Compare_AddressBUS)

);
```

```
Validcache Validcache(
    .Address                    (ProcessorAddressIn[9:2]),
    .ValidIn                    (Fsm_ValidMem_DataBit_Wire),
    .Write                      (Fsm_ValidMem_WriteEn_Wire),
    .Reset                      (ResetFlag),
    .Clk                        (Clk),

    .ValidOut                   (Valid_Fsm_WIRE)
);

CacheDataRam CacheDataRam1(
    .Address                    (CacheAddressMux_CacheMem_AddressBUS),
    .DataIn                     (DdrDataIn),
    .Write                      (Fsm_CacheMem_WriteEn_Wire),
    .Clk                        (Clk),

    .DataOut                    (CacheMem_CacheAddressMux_DataBUS)
);

//Muxes  CacheAddressMux and DoutMux
CacheAddressMux CacheAddressMux(
    .Select                     (Fsm_CacheAddressMux_Select_Wire),
    .FSMAddress                 (Fsm_CacheAddressMux_AddressBUS),
    .ProcessorAddress           (ProcessorAddressIn[(CASHE_ADDRESS_LENGTH-
1):0]),

    .Address                    (CacheAddressMux_CacheMem_AddressBUS)
);

DoutMux DoutMux(
    .Select                     (Fsm_DataOutMux_Select_Wire),
    .CashData                    (CacheMem_CacheAddressMux_DataBUS),

    .MemData                     (DdrDataIn),


    .DataOut                     (ProcessorDataOut)

);


//other

Compare Compare(

    .CashTag                     (Tag_Compare_AddressBUS),

    .MemTag                      (ProcessorAddressIn[(ADDRESS_LENGTH-
1):CASHE_ADDRESS_LENGTH]),


    .Match                       (Compare_Fsm_WIRE)

);



Endmodule
```

Cache Controller FSM

```
module FSMControl#(
parameter       ADDRESS_LENGTH              =19
)
(
input                                        WriteIn,
input                                                 ReadIn,
input                                        Match,
input                                        Valid,
input                                        Clk,
input                                        ResetFlag,
input                                        Memory_Cache_Read_Ready,
input                                        Memory_Cache_Write_Ready,
input        [18:0]                          Processor_Cache_Address,



output reg                                   Cache_Write_Enable,              // 1= write to cashs
output reg                                   Select_DataOut_Memory_cache,        // 1 = memory
output reg                                   Select_CacheAddrss_FSM_Address,     // 0 = processor 1=
fsm address
output reg                                   Ram_Write,     // 0 = memory 1= fsm addres
output reg                                   Ram_Read,
output reg                                   Tag_Write,
output reg                                   Valid_Write,
output reg                                   Valid_Bit,
output reg                                   Cache_Processor_Read_Ready,
output reg                                   Cache_Processor_Write_Ready,
output reg [18:0]                            Cache_RAMportB_Address,
output reg [9:   0]                          Cache_DataCache_Address,            //10 bits
output reg                                   cache_hit,
output reg                                   cache_miss

);
```

```
     reg Write_Sample                          =0;

     reg MReady_Read_sample                    =0;

     reg MReady_Write_sample                   =0;

     reg Read_Sample                           =0;

     reg [18:0] Address_sample                 =19'd0;

     reg [16:0] Address_temp                   =19'd0;

     reg [1:0]  temp                           =2'd0;

     localparam [2:0] idle=3'd0, Write_Wait=3'd1, Read_Wait=3'd2,
Data_Load_1st=3'd3,Data_Load_2nd=3'd4,Data_Load_3rd=3'd5,Reset_State=3'd6;

     reg [2:0] state                           =3'd0;


     always @(negedge Clk)

     begin

        Write_Sample                           <= WriteIn;

        Read_Sample                            <= ReadIn;

        MReady_Read_sample                     <= Memory_Cache_Read_Ready;

        MReady_Write_sample                    <= Memory_Cache_Write_Ready;

        Address_sample                         <=Processor_Cache_Address;

        if(ResetFlag) begin

        end

     end


     always @(posedge Clk)

     begin

        case (state)

            idle: if(Read_Sample) begin

                        if (Match && Valid) begin

                            Cache_Processor_Read_Ready          <=1'd1;

                            state                               <=Reset_State;

                            cache_hit                           <= 1;

                        end

                        else begin

                            Select_CacheAddrss_FSM_Address      <=1'd1;      //1 = to cache
address from fsm
```

```
                    Cache_RAMportB_Address            <=Address_sample;    // first
address to memory
                    Ram_Read                          <=1'd1;
                    Cache_Write_Enable                <=1'd1;
                    Cache_DataCache_Address           <=Address_sample[9:0];
                    Address_temp                      <=Address_sample[18:2];
                    temp[1:0]                         <=Address_sample[1:0]+1;
                    Select_DataOut_Memory_cache       <=1'd1;       // first data
from memory
                    Valid_Bit                         <=1'd1;
                    Valid_Write                       <=1'd1;
                    Tag_Write                         <=1'd1;
                    state                             <=Read_Wait;
                    cache_miss                        <= 1;
                end
            end
            else if (Write_Sample) begin
                if (Match && Valid) begin
                    Valid_Bit                         <=1'd0;
                    Valid_Write                       <=1'd1;
                    Cache_RAMportB_Address            <=Address_sample;
                    Ram_Write                         <=1'd1;
                    state                             <= Write_Wait;
                end
                else begin
                    state                             <= Write_Wait;
                    Cache_RAMportB_Address            <=Address_sample;
                    Ram_Write                         <=1'd1;
                end
            end
            else begin
                state                                 <=idle;
            end
        Reset_State: begin
                        Cache_Processor_Read_Ready    <=1'd0;
                        Cache_Processor_Write_Ready   <=1'd0;
                        Valid_Write                   <=1'd0;
                        Valid_Bit                           <=1'd1;
                        Ram_Write                           <=1'd0;
                        Tag_Write                           <=1'd0;
                        Select_CacheAddrss_FSM_Address      <=1'd0;
                        Select_DataOut_Memory_cache         <=1'd0;
                        Cache_Write_Enable                  <=1'd0;
                        state                               <=idle;
                        cache_hit                           <= 0;
                    end

        Write_Wait: if(MReady_Write_sample) begin
                        Cache_Processor_Write_Ready         <=1'd1;
                        Ram_Write                           <=1'd0;
                        state                               <=Reset_State;
                    end
                    else begin
                        Valid_Write                         <=1'd0;
                        Ram_Write                           <=1'd0;     //only for 1
clock cycle
                        state                               <=Write_Wait;
                    end
        Read_Wait:  if(MReady_Read_sample) begin                          //1st data in
the cache
                        Ram_Read                            <=1'd1;
                        Cache_RAMportB_Address              <={Address_temp,(temp[1:0])};
//addresing 2nd data
                        Cache_DataCache_Address
<={Address_temp[7:0],temp[1:0]};  //geting ready for  2nd Data input
                        Cache_Processor_Read_Ready          <=1'd1;
                        temp[1:0]                           <=temp+1;
                        state                               <=Data_Load_1st;
                        cache_miss                          <= 0;

                    end
                    else begin
                        Ram_Read                            <=1'd0;
//fist clock after addressing
                        Tag_Write                           <=1'd0;
                        Valid_Write                         <=1'd0;
```

```
                              state                              <=Read_Wait;
                              cache_miss                         <= 0;
                          end
            Data_Load_1st:if(MReady_Read_sample) begin                            //2nd data in
the cache
                              Cache_Processor_Read_Ready  <=1'd0;
                              Ram_Read                           <=1'd1;
                              Cache_RAMportB_Address             <={Address_temp,temp[1:0]};
//addresing 3rd data
                              Cache_DataCache_Address
<={Address_temp[7:0],temp[1:0]};  //geting ready for  3rd Data input
                              temp[1:0]                          <=temp+1;
                              state                              <=Data_Load_2nd;

                          end
                          else begin
                              Cache_Processor_Read_Ready  <=1'd0;
                              Ram_Read                           <=1'd0;
                              state                              <=Data_Load_1st;
                          end

            Data_Load_2nd:if(MReady_Read_sample) begin                            //3rd data in
the cache
                              Ram_Read                           <=1'd1;
                              Cache_RAMportB_Address             <={Address_temp,temp[1:0]};
//addresing 4th data
                              Cache_DataCache_Address
<={Address_temp[7:0],temp[1:0]};  //geting ready for  4th Data input
                              temp[1:0]                          <=temp+1;
                              state                              <=Data_Load_3rd;

                          end
                          else begin
                              Ram_Read                           <=1'd0;
                              state                              <=Data_Load_2nd;
                          end

            Data_Load_3rd:if(MReady_Read_sample) begin                            //4th data in
the cache
                              temp                               <=2'd0;
```

```
                                            state                                   <=Reset_State;


                                        end
                                        else begin
                                            Ram_Read                                <=1'd0;
                                            state                                   <=Data_Load_3rd;
                                        end
                        default: state                                              <=idle;
                    endcase
                end
            endmodule
```

Cache Memory

```
module CacheDataRam#(
parameter      DATA_WIDTH                =8,
parameter      CACHE_ADDR_SIZE           =10

)
(
input          [(CACHE_ADDR_SIZE-1):0]   Address,
input          [(DATA_WIDTH-1):0]        DataIn,
input                                    Write,
input                                    Clk,

output reg     [(DATA_WIDTH-1):0]        DataOut
);

reg            [(DATA_WIDTH-1):0]        DataRam[0:(2**(CACHE_ADDR_SIZE)-1)];

always @ (negedge Clk) begin
   if (Write) begin              // Write
        DataRam[Address]                    =DataIn;
   end
end

always @ (negedge Clk) begin        // Read
   DataOut                                          =DataRam[Address];
end
endmodule
```

Tag Memory

```
module TagRam #(
parameter      TAG_ADDR_WIDTH            =8,
parameter     TAG_LENGTH                =9
)


(
input         [(TAG_ADDR_WIDTH-1):0]       Address,    //only 8 bits from the
total 10 TAG = 10-2*
input         [(TAG_LENGTH-1):0]           TagIn,
input                                      Write,
input                                      Clk,
output  reg   [(TAG_LENGTH-1):0]           TagOut
);

reg           [TAG_LENGTH-1:0]             TagRam[0:(2**TAG_ADDR_WIDTH-1)];

always @ (negedge Clk)          // Write
   if (Write) begin
    TagRam[Address]                     = TagIn;
   end

always @ (negedge Clk) begin    // Read
   TagOut                               = TagRam[Address];
end
endmodule
```

Valid Memory

```
module Validcache #(
parameter     TAG_ADDR_WIDTH            =8
)

(
input          [(TAG_ADDR_WIDTH-1):0]          Address,
input                                           ValidIn,
input                                           Write,
input                                           Reset,
input                                           Clk,
output          reg                             ValidOut
);

reg             [(2**TAG_ADDR_WIDTH-1):0]       ValidBits;

integer i;

always @ (negedge Clk)          // Write or Reset
   if (Write && !Reset) begin           // Write
        ValidBits[Address]                         = ValidIn;
   end
   else if (Reset)                      // Reset
      for (i=0; i<(2**TAG_ADDR_WIDTH-1); i=i+1)
         ValidBits[i]                          = 0;

always @ (negedge Clk) begin    // Read
   ValidOut                                    = ValidBits[Address];
end

endmodule
```

Data Output multiplexer

```
module DoutMux#(
parameter     DATA_WIDTH                 =8
)
(
input         [1:0]                      Select,
input         [(DATA_WIDTH-1):0]         CashData,
input           [(DATA_WIDTH-1):0]          MemData,
output reg      [(DATA_WIDTH-1):0]          DataOut
    );

     always @*
     begin
        case (Select)
            1'b1: begin
                        DataOut=MemData;
                    end
            1'b0: begin
                        DataOut=CashData;
                    end
        endcase
    end
endmodule
```

Cache Address Multiplexer

```
module CacheAddressMux#(
parameter     ADDRESS_WIDTH                =10
)
(
input                                    Select,
input          [(ADDRESS_WIDTH-1):0]        FSMAddress,
input          [(ADDRESS_WIDTH-1):0]        ProcessorAddress,
output reg     [(ADDRESS_WIDTH-1):0]        Address
    );
     always @*
     begin
        case (Select)
            1'b1: begin
                        Address=FSMAddress;
                    end
            1'b0: begin
                        Address=ProcessorAddress;
                    end
        endcase
    end
endmodule
```

Compare Module

```
module Compare#(
parameter     TAG_LENGTH                =9
)
(
input          [(TAG_LENGTH-1):0]       CashTag,
input          [(TAG_LENGTH-1):0]       MemTag,
output reg                              Match
);
     always @*
     begin
        if (CashTag==MemTag)begin
            Match                                    =1'b1;
        end else begin
            Match                                    =1'b0;
        end
     end
endmodule
```

11.2Matlab Codes for FPGA services

Program

```
function program(filename,COM,baud_rate)
%filename = 'addsum.byt'; % PROGRAM FILE

delimiterIn = '\n';
%headerlinesIn = 1;
ins_array = importdata(filename,delimiterIn);
ins_array = bin2dec(num2str(ins_array));

ins_array = ins_array'; % flatten to 1 row

display('Programming....');
tic
v = mem_write_verify('program',ins_array,COM,baud_rate);
toc

if v
    display('Program Success');
else
    display('Program FAILED');
end
```

write_mem

```
function write_mem(mem_sel,byte_array,COM,baud_rate)

    if strcmp(mem_sel,'program')
        mem_byte = 255;
    elseif strcmp(mem_sel,'input')
        mem_byte = 253;
    elseif strcmp(mem_sel,'output')
        mem_byte = 251;
    else
        mem_byte = 0;
    end

    if mem_byte ~= 0
```

```
        s = serial(COM,'BaudRate',baud_rate);   %com8 19200
        fopen(s);

        datalen = length(byte_array);

        data_bin = fliplr(de2bi(datalen));
        data_bin = [zeros(1, max(0, 24 - numel(data_bin))), data_bin];

        MSB1 = binaryVectorToDecimal(data_bin(1:8));
        MSB2 = binaryVectorToDecimal(data_bin(9:16));
        MSB3 = binaryVectorToDecimal(data_bin(17:24));

        seq = [mem_byte MSB1 MSB2 MSB3 byte_array];

        % writing
        for i = 1:1:length(seq)
            %display(seq(i));
            %while (~strcmp(s.TransferStatus, 'idle'))
            %end
            fwrite(s,seq(i),'uint8');
        end

        fclose(s);
        delete(s);
        clear s;
    end
end
```

mem_write_verify

```
function verified = mem_write_verify(mem_sel,byte_array,COM,baud_rate)
    write_mem(mem_sel,byte_array,COM,baud_rate);
    read_array = read_mem(mem_sel,COM,baud_rate);

    if (isequal(byte_array,read_array))
        verified = 1;
    else
        verified = 0;
    end
end
```

ImageIn

```
function ImageIn(image,image_bytefile)
Im = imread(image);
ImGr = rgb2gray(Im);
[h,w] = size(ImGr);
ImGr=transpose(ImGr);%check if needed.
OneD_Im = double(ImGr(:));
fileID = fopen(image_bytefile,'w');

%h1 =dec2bin(h,8);
%w1 =dec2bin(w,8);
h_bin = fliplr(de2bi(h));
h_bin = [zeros(1, max(0, 16 - numel(h_bin))), h_bin];
h_MSB1 = binaryVectorToDecimal(h_bin(1:8));
h_MSB2 = binaryVectorToDecimal(h_bin(9:16));

w_bin = fliplr(de2bi(w));
w_bin = [zeros(1, max(0, 16 - numel(w_bin))), w_bin];
w_MSB1 = binaryVectorToDecimal(w_bin(1:8));
w_MSB2 = binaryVectorToDecimal(w_bin(9:16));

fprintf(fileID,'%s \n',dec2bin(h_MSB1,8));
fprintf(fileID,'%s \n',dec2bin(h_MSB2,8));
fprintf(fileID,'%s \n',dec2bin(w_MSB1,8));
fprintf(fileID,'%s \n',dec2bin(w_MSB2,8));
for i = 1:h*w
    pixel = OneD_Im(i);
    fprintf(fileID,'%s \n',dec2bin(pixel,8));
end

fclose(fileID);
end
```

load_image

```
function load_image(filename,COM, baud_rate)

%filename = 'ByteIMG4.txt'; % PROGRAM FILE

delimiterIn = '\n';
%headerlinesIn = 1;
img_array = importdata(filename,delimiterIn);
img_array = bin2dec(num2str(img_array));

img_array = img_array'; % flatten to 1 row

display('Loading....');
tic
v = mem_write_verify('input',img_array,COM, baud_rate);
toc

if v
    display('Loading Success');
```

```
else
    display('Loading FAILED');
end
```

execute

```
function execute(COM,baud_rate)

s = serial(COM,'BaudRate',baud_rate);
fopen(s);

fwrite(s,240,'uint8');
tic
display('Executing');

while(~s.BytesAvailable)     % wait for end byte to come
end
toc

byte_end = fread(s,1,'uint8');

if( byte_end == 239)
    display('Executed');
else
    display('ERROR');
end

fclose(s);
delete(s);
clear s;
end
```

build_image

```
function build_image(byte_arr,image, output_image_bytefile,output_image)
    %save('output_img.txt','byte_arr');
    fileID = fopen(output_image_bytefile,'w');

    for i=1:1:length(byte_arr)
        fprintf(fileID,'%u \n',byte_arr(i));
    end
    fclose(fileID);

    Im=imread(image);
    ImGr = rgb2gray(Im);
    [input_h,input_w] = size(ImGr);

    h = floor(input_h/2);
    w = floor(input_w/2);

    img = reshape(byte_arr,w,h);
    %img = reshape(byte_arr,128,128);
    img = (img')/255;
    imwrite(img,output_image);
    figure
    imshow(img);
end
```

Gauss_dwnSmpl3_while

```
function Gauss_dwnSmpl3_while(image,output_image_bytefile)%
% NOT in assembly
Im=imread(image);
ImGr = rgb2gray(Im);

% registers RH and RW loaded by two specific memory locations in RAM
% loaded by UART
[h,w] = size(ImGr);

% NOT in assembly
figure
imshow(ImGr);
ImGr=transpose(ImGr);

% input flat image RAM
OneD_Im = double(ImGr(:));

% output flat image RAM
temp_Im = zeros(floor(h/2)*floor(w/2),1);
% RAM address pointer for output image (register)ARK
k = 0;


Conv_sum=double(0); % NOT in assembly

% hard coded in assembly
ker_centre = 4;%4
ker_middle = 2;%2
ker_corner = 1;%1
ker_sum = ker_corner*4 + ker_middle*4 + ker_centre;
%%%%%%%%%%%%%%

% register RJ
j = 0;
% register RX           sampling pointer
x = 0;
while j <= h - 2    % for do-while -2 not needed
    j = j+2;
    x = x + w;          % increment row

    % register RI
    i = 0;
    while i <= w - 2      % for do-while -2 not needed
        i = i + 2;
        x = x + 2;      % increment 2:1 coloumn
        % register RS
        Conv_sum = 0;

        %following must be in this order grays and yellows
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_centre);
        x = x - w;
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle);
        x = x - 1;
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner);
        x = x + w;
        Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle);  %x-1

        if (h-j)~=0
            x = x + w;
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner);
            x = x + 1;
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle); %x+w
        else
            x = x + 1;
            x = x + w;
        end

        if (w-i)~=0
            x = x + 1;
            if (h-j)~=0
                Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner);%x+w+1
                x = x - w;
            else
                x = x - w;
                %Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner);%x+1
            end     %x+1
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_middle);
            x = x - w;
            Conv_sum = Conv_sum + (OneD_Im(x)*ker_corner); %x-w+1

            x = x + w;
            x = x - 1;
        else
            x = x - w;
        end



        Conv_sum = floor(Conv_sum/(ker_sum));
        k = k + 1;
        temp_Im(k) = Conv_sum;



    end
    %if w is odd add 1 for i incremnts to equal to w ADDRXAC
    if w - i ~= 0
        x = x + 1;
    end
end
```

```
fil_ds_Im = uint8(temp_Im);

%filename = 'output_img4.txt'; % PROGRAM FILE
filename = output_image_bytefile;

delimiterIn = '\n';
%headerlinesIn = 1;
img_array = uint8(importdata(filename,delimiterIn));
cpu_img = reshape(img_array,[floor(w/2),floor(h/2)]);
figure
imshow(cpu_img');
title('down sampled- CPU');

% img_array  -  CPU downsampled one dimensional image byte array
% fil_ds_Im  -  Matlab downsampled one dimensional image byte array

ssd = fil_ds_Im - img_array;
ssd_img = reshape(ssd,[floor(w/2),floor(h/2)]);
ssd = ssd .* ssd;
ssd_tot = sum(ssd);

display(ssd_tot);

result = reshape(fil_ds_Im,[floor(w/2),floor(h/2)]);
figure
imshow(result');
title('down sampled- MATLAB');

figure
imshow(ssd_img);
title('ssd');
end
```

Read_mem

```
function mem_out = read_mem(mem_sel,COM,baud_rate)
    if strcmp(mem_sel,'program')
        mem_byte = 254;
    elseif strcmp(mem_sel,'input')
        mem_byte = 252;
    elseif strcmp(mem_sel,'output')
        mem_byte = 250;
    elseif strcmp(mem_sel,'read')
        mem_byte = 249;
    else
        mem_byte = 0;
    end

    if mem_byte ~= 0
        s = serial(COM,'BaudRate',baud_rate); %com8
        s.InputBufferSize = 270000;%8192; % increase buffer size if
neccessary
```

```
        fopen(s);

        fwrite(s,mem_byte,'uint8');
        dataout = [];

        while(~s.BytesAvailable)    % wait for first byte to come
        end
        %pause(0.01);

        i = 0;
        while(s.BytesAvailable)
            i = i + 1;
            %display(i);
            dataout = [dataout fread(s,1,'uint8')];
        end

        fclose(s);
        delete(s);
        clear s;

        mem_out = dataout;
    end
end
```