Department of Electronic and Telecommunication Engineering

Faculty of Engineering

University of Moratuwa, Sri Lanka

EN3030 Circuits and Systems Design

REPORT ON PROCESSOR DESIGN FOR IMAGE DOWN-SAMPLING

P.M.N.S. Bandara - 180066F

S.S. Hettiarachchi - 180237G

W.M.H.M. Wasala - 180675V

This is submitted as a partial fulfillment for the module

EN3030: Circuits and System Design

July 10, 2022

List of Figures

Contents

Chapter 1

Introduction

Many of complex electronic devices contain a central processing unit (CPU) like computers have. All the functions specified in the instructions such as arithmetic, logical, control and input/output operations are performed in this CPU. CPU can be divided two parts as Processing unit and Control Unit. All of the functionalities of the program are implemented by the central processing unit according to the defined instructions. Central processing unit is consisting with several microprocessors and they are capable of implementing functions. During the function implementations, microprocessors stored intermediate data in registers. As mentioned previously, microprocessors are extremely important for modern world applications. Electronic devices such as mobile phones, televisions and other smart devices contain microprocessors and there is value creating opportunity due to processor performance. High performed and fast microprocessors are high in price. Therefore, for modern technology devices has given a high priority for processors.

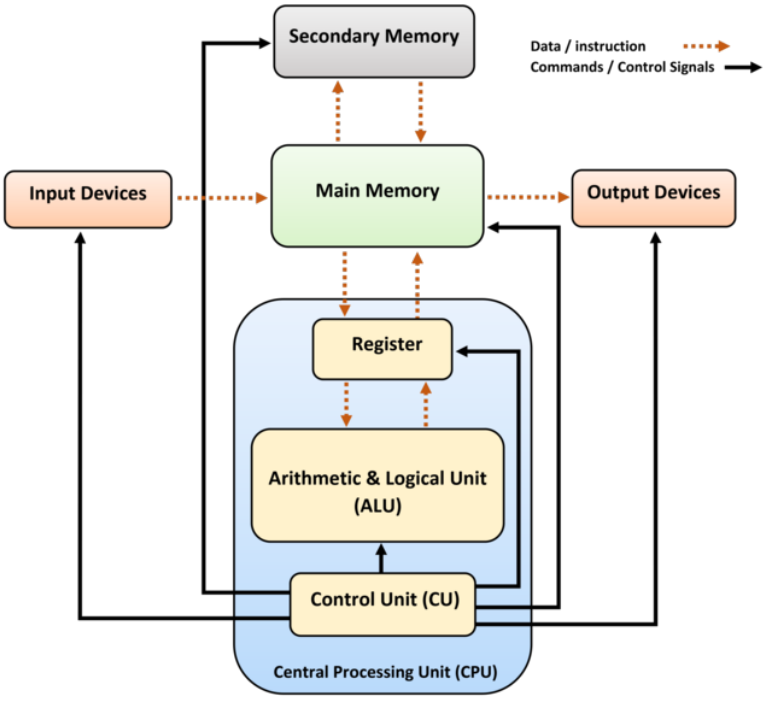

Figure 1.1: Generic Processor Diagram with crucial blocks and their inter-connections

1.1 Problem Statement

The main objective of this project is designing a custom microprocessor to down-sample an image of m x m pixels which is given as an input by a factor of n and displaying the output after down-sampling. (Due to resource limitations and COVID-19 situation proposed project has been limited to ISA designing and Vivado-based implementation.)

Therefore, it is concluded to specify the following parameters for the implementation.

- The input image size: (256×256)
- The input image type: Gray-scale
- The down-sampling factor: 2

1.2 Addressing the problem

For approaching the solution we can divide the whole process to several parts such as input data storing in the memory, filtering the image, downsampling the image and displaying the downsampled image.

1.2.1 Storing the input data in the memory

For the communication procedure, serial communication can be used to feed the input image to the FPGA board from the computer. For that UART receiver is implemented using Verilog hardware description language to receive one byte at a time. Since 8-bit images are used to down-sample, UART receiver can receive a one pixel value at a time. Since 256×256 pixel images are used, 65536 pixel values should be stored in the memory. For storing the m x m pixel values DRAM should be implemented along with the UART receiver.

1.2.2 Filtering and down sampling the image

Even though it could be simply possible to down-sample the input image by 2, through selecting or manipulating a pixel value from a square of 4 pixels from the input image, the output down-sampled image through this method does not preserve the satisfactory details of the original image, thus, introduce *aliasing*. Therefore, in order to reduce the effect from aliasing, it is needed to low-pass filter the image first to filter the high frequency components, and then pass the image for down-sampling.

This detailed procedure for the whole procedure will be discussed under the algorithm development chapter.

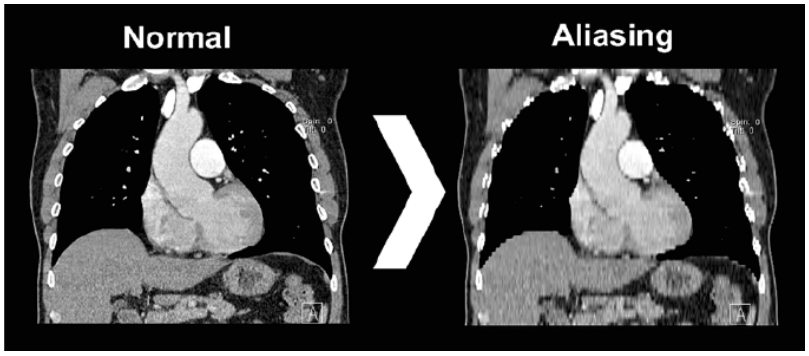

Figure 1.2: Distortions presented by aliasing effect

1.2.3 Storing the input data in the memory

In here output image should be send to the computer. Therefore, UART transmitter should be implemented in a FPGA board. (This part is also remained as we cannot use FPGA boards in current situation).

1.2.4 Implemented Approach

We approached to the required design as follows.

1. Designing the algorithm filtering before pixel
2. Simulation of the algorithm using MATLAB
3. Determining the Specifications of the Processor such as number of registers, memory requirements, number of buses etc.
4. Designing the ISA
5. Translating the algorithm into an assembly code using the defined instructions.
6. Testing and Verifying the algorithm
7. Implementation of the modules in Verilog
8. Simulation of the modules using Verilog
9. Final simulation and verification

Chapter 2

Algorithm Development

2.1 Introduction to algorithm development

2.1.1 Filtering algorithm

Even though the down-sampling procedure reduces the number of pixels in an image, it is required to ensure that all the essential information within the original image is preserved in the down-sampled output image as well. However, spurious high-frequency components in the original image introduce undesired aliasing effects in the down-sampled output image and thus, the general practice, in order to reduce the image's high frequency components, is to convolve the original image with a Gaussian function before resampling since the Gaussian function is a low-pass filter. Furthermore, no ringing effects would be present in the filtered output image since Gaussian blurs do not comprise sharp edges.

The one-dimensional (1D) Gaussian function is defined as follows:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{-\frac{x^2}{2\sigma^2}} \tag{2.1}$$

where x is the distance from the defined origin in the defined axis and σ is the standard deviation of the applied Gaussian distribution. This 1D function could be extended to two-dimensions (2D) as a two such Gaussian functions, one in each dimension, thus manipulates:

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{-\frac{(x+y)^2}{2\sigma^2}} \tag{2.2}$$

where typically x is the distance from the origin on the horizontal axis and y is the distance from the origin on the vertical axis.

The Gaussian function is implemented as a kernel of size 3×3 as presented below, where the location of the center pixel is considered as $(0,0)$ and the standard deviation of the Gaussian function is 1.

0.0585	0.0965	0.0585
0.0965	0.1592	0.0965
0.0585	0.0965	0.0585

As observed above, the weight of each element in the kernel is inversely proportional to the distance from the center element. However, since floating point arithmetic is not expected to deploy in the proposed processor, an approximation for the Gaussian kernel weights is determined through discretization as below:

1	2	1
2	4	2
1	2	1

Even though the introduced discretization is vital to reduce the computational cost of the filter operations, in contrast, it adds a considerate error in terms of the final output. Further, since the conversion of Gaussian continuous values to discrete values results in the sum of all kernel weights being different than 1, the final output image would be darkened or brightened. This effect could be eradicated by normalizing all kernel weights through dividing them by the sum of all weights, i.e. here by 16.

In the convolution operation, the original image $I(x,y)$ of size $(h \times w)$ is convolved with the Gaussian mask $G(x,y)$ through computing the sum of products along the image and the Gaussian matrix as presented in the following equation.

$$f(x,y) = \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} G(i,j)I(x-i,y-j) \tag{2.3}$$

2.1.2 Down-sampling algorithm

However, since the Gaussian kernel presented above is symmetric in both x and y dimensions, the equation for convolution in 2.3 could be simplified such that the four pixels in a square in the original image are mapped into one pixel in the down-sampled image. In contrast to the generic procedure where the Gaussian kernel is placed on the original image in a manner in which the centre value of the kernel is aligned with the pixel in the image that needs to be filtered. The sum of products with respect to kernel values and the image pixel values are then calculated and eventually, the sum is divided by 16 to normalize the pixel value in the down-sampled image.

Therefore, the symmetry of the Gaussian kernel leads to reduce the computational cost of the down-sampling operation through a lesser number of pixel selection for executing the convolution. The pixels are identified to be essential for generating same results after filtering which are to be put in the down-sampled image.

1. Along a row in the original image, one pixel is skipped after filtering before pixel
2. A complete row is skipped after filtering with the before row

In mathematical terms, this could be introduced as:

$$I_{DS} = I(x|x \ \ mod2 = 0; y|y \ \ mod2 = 0) \tag{2.4}$$

Thus, after the combination of low-pass filtering and down-sampling into one simplified computation, the following graphical representation in figure 2.1, for 8×8 image, could be introduced as the computational step for the designed processor.

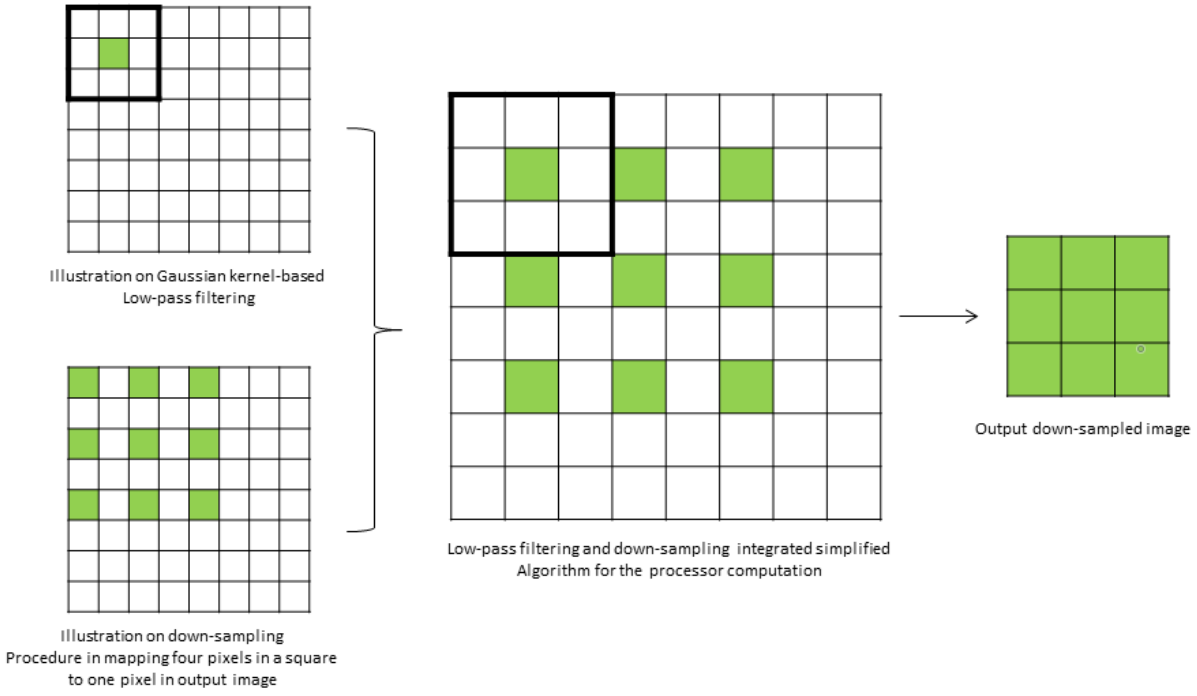

Figure 2.1: The developed algorithm for the custom processor for down-sampling an image

2.2 Integrated Algorithm in a Pseudo Code

Here, we expect to store the processed pixel values in the same memory in which the original image was stored (i.e. $256 \times 256 = 65536$ slots of bytes) through a replacement mechanism. Therefore, the overwriting the same pixel memory locations of the original image is executed with a re-ordering mechanism. Here, the re-ordering mechanism is introduced to facilitate the downs-sampled image storing and the replacement is specifically executed by placing the pixels of down-sampled image from 0^{th} index slot in order.

Therefore, the MATLAB-based pseudo code for the finalized algorithm for down-sampling a 256×256 image by a factor of 2 could be introduced as follows:

```
im;
Value=0;
m=1;
for j=2:2:254
    for i=2:2:254
        Value=0;
        K=256*(j-1)+(i-1)+1;
        Value=Value + double(im(K));
        Value=Value + double(im(K+1)*2);
        Value=Value + double(im(K+2));
        Value=Value + double(im(K+2+254)*2);
        Value=Value + double(im(K+2+254+1)*4);
        Value=Value + double(im(K+2+254+2)*2);
        Value=Value + double(im(K+2+254+2+254));
        Value=Value + double(im(K+2+254+2+254+1)*2);
        Value=Value + double(im(K+2+254+2+254+2));
        Value=Value/16;
        im(m) = Value;
        m=m+1;
    end
end
```

Chapter 3

Instruction Set Architecture

3.1 Requirements for the Instruction Set Architecture

In accordance with the pseudo code presented in section 2.2, the following variables are to remembered where the algorithm is being executed.

1. Pow number of the current pixel
2. Column number of the current pixel
3. Kernel position while convolving
4. Convolutional sum while convolving
5. Position of the Data Ram to store the processing pixel

Thus, five general purpose registers are needed to store the above values. In addition, the designed processor is expected to execute the following arithmetic operations, thus, arithmetic and logic unit must support there operations.

1. Addition
2. Substraction: for conditional statements
3. Passing the same value
4. Multiplication: by 2 & 4
5. Division: by 16
6. Reset
7. Increment
8. Decrement

3.2 General Architecture

The project requires the implementation of a unique processor to down-sample a 256 × 256 image by a factor of two. Following is the processor's architecture, which was created based on the demands of the task and algorithm.

- Memory
  - Data Memory - The image can be stored in 65536 memory places with an 8 bit width in data memory.
  - Instruction Memory - To hold the instructions, memory has 256 memory addresses that are each 8 bits wide.
- Registers
  - Accumulator (AC) - The AC register, which has direct access to the ALU, is used to read and write data.The AC register has 16 bits.
  - Program Counter (PC) - The address of the next instruction to be executed is stored in the 16-bit PC register.
  - Address Register (AR) - The next bit of data to be fetched or stored in data memory is kept at the address stored in the 16-bit AR register.
  - Instruction Register (IR) - The instruction read from the instruction memory is stored in the 8-bit IR register.
  - R, R1, R2, R3, R4, R5 - Six 16 bit General Purpose Registers (with increment flag)
- ALU - The ALU performs arithmetic and logical processes. In this architecture, the ALU executes 10 operations.
- Control Unit - The control unit decodes the instructions and generates control signals to execute them accordingly.

3.3 Set of Instructions

The table 4.3 presents 33 instructions, in which width of each instruction is 8-bit.

3.4 Instruction Cycle

Instruction cycle is consisted of the following three stages:

- Fetch
- Decode
- Execution

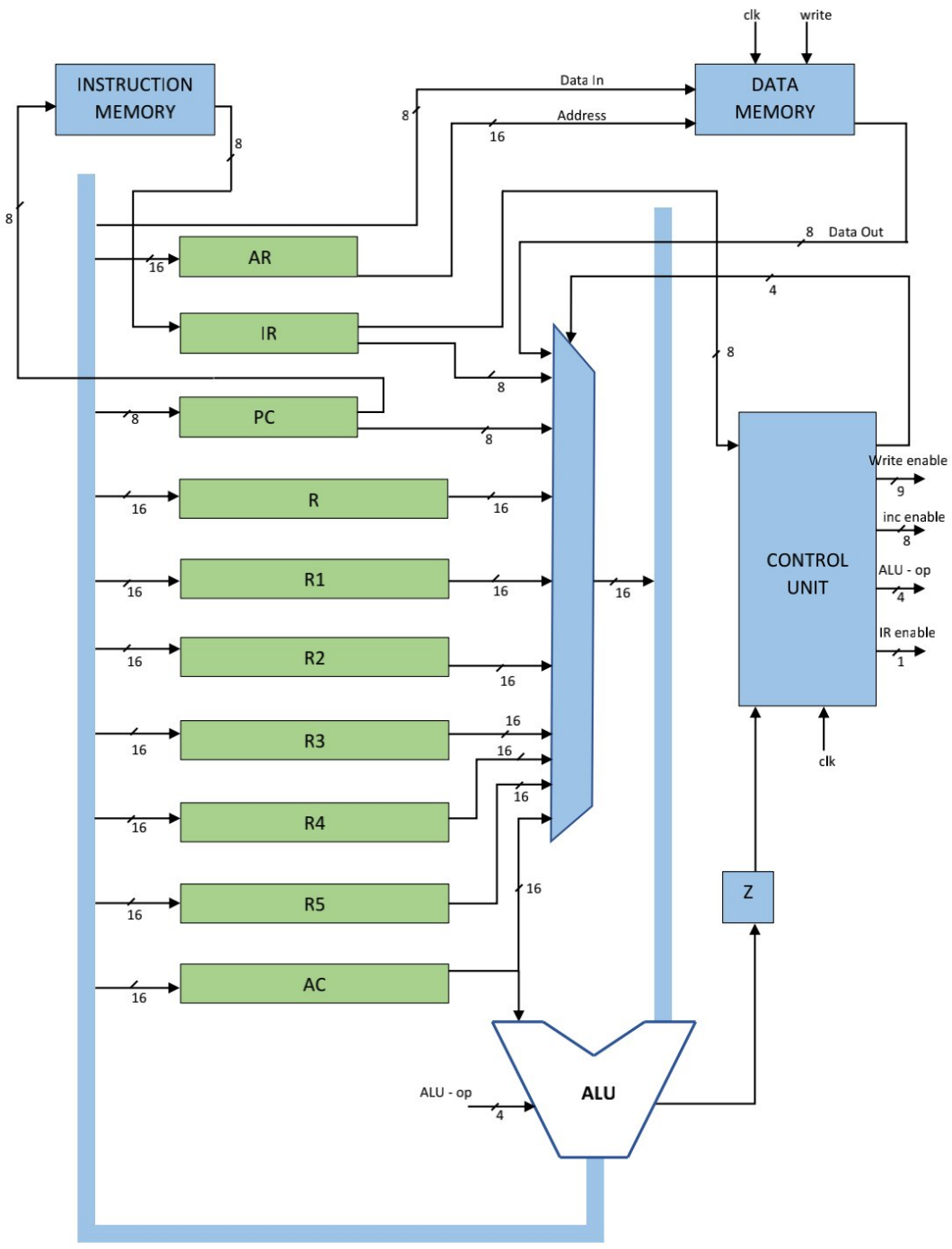

Figure 3.1. Data path

Table 3.1: Set of instructions developed for the task

Instruction Code	Operation
NOP	No operation
CLAC	$AC \leftarrow 0$
STAC	$DRAM[AR] \leftarrow AC$
LDAC	$AC \leftarrow DRAM[AR]$
INCAC	$AC \leftarrow AC + 1$
DECAC	$AC \leftarrow AC - 1$
MVACR	$R \leftarrow AC$
MVACR1	$R1 \leftarrow AC$
MVACR2	$R2 \leftarrow AC$
MVACR3	$R3 \leftarrow AC$
MVACR4	$R4 \leftarrow AC$
MVACR5	$R5 \leftarrow AC$
MVACAR	$AR \leftarrow AC$
MOVR	$AC \leftarrow R$
MOVR1	$AC \leftarrow R1$
MOVR2	$AC \leftarrow R2$
MOVR3	$AC \leftarrow R3$
MOVR4	$AC \leftarrow R4$
MOVR5	$AC \leftarrow R5$
INCR1	$R1 \leftarrow R1 + 1$
INCR2	$R2 \leftarrow R2 + 1$
INCR3	$R3 \leftarrow R3 + 1$
INCR4	$R4 \leftarrow R4 + 1$
INCR5	$R5 \leftarrow R5 + 1$
MUL2	$AC \leftarrow AC * 2$
MUL4	$AC \leftarrow AC * 4$
DIV16	$AC \leftarrow AC/16$
ADD	$AC \leftarrow AC + R$
SUB	$AC \leftarrow AC - R$
JPNZ "M"	If $Z = 0$, GO TO INSTRUCTION M
ADDM "M"	$AC \leftarrow AC + M$
END	$FINISH \leftarrow HIGH$

3.4.1 Fetch

The current address in the PC points to the address of the next instruction which is to be fetched. This instruction is fetched from the instruction ROM to IR such that the end of each FETCH cycle, the value in

PC is incremented by one, thus, representing the next instruction which is to be fetched. Hence, this stage is a two-step state machine.

$$FETCH1: \quad IR \leftarrow IROM[PC]$$

$$FETCH2: \quad PC \leftarrow PC + 1$$

3.4.2 Decode

Since the processor is expected to differentiate the instructions which are being fetched to invoke the correct execution cycle, the instruction register directs the fetched instruction to the control store such that the output of the control store is the corresponding control signal. Further, if the instruction is only consisted of one state, the next FETCH cycle is initiated.

3.4.3 Execute

NOP No operation is performed within this operation. This is typically utilized when a waiting cycle is needed in order to have a set of processed data available at an end point.

CLAC AC is cleared through assigning zero to it. Here, zero flag is indicated and afterwards, the FETCH cycle is triggered.

$$CLAC: \quad AC \leftarrow 0$$

STAC The data in AC is loaded to the corresponding memory address which is previously stored in the AR.

$$STAC: \quad M[AR] \leftarrow AC$$

LDAC In this three-clock cycle long operation, the data in the DRAM is read and loaded to AC with reference to the location pointed by the current address in AR. Afterwards, the next FETCH cycle is commenced.

$$LDAC1: \quad MEMORY\ READ$$

$$LDAC2: \quad MEMORY\ READ$$

$$LDAC3: \quad AC \leftarrow M[AR]$$

DECAC This operation is to decrement AC by one and if the value of AC after this operation is zero, then $Z = 0$ flag is issued, otherwise, $Z = 1$ displayed. Afterwards, the next FETCH cycle is commenced.

$$DECAC: \quad AC \leftarrow AC - 1;\ IF(AC == 0)\ THEN\ Z = 0\ ELSE\ Z = 1$$

MVACX where X $\in$ {R1,R2,R3,R4,R5,AR} This crucial operation is to move the value in AC to the specified register as per the naming of the instruction and the next FETCH cycle is commenced after the executed single state instruction which are stated below.

$MVACR: \quad R \leftarrow AC$

$MVACR1: \quad R1 \leftarrow AC$

$MVACR2: \quad R2 \leftarrow AC$

$MVACR3: \quad R3 \leftarrow AC$

$MVACR4: \quad R4 \leftarrow AC$

$MVACR5: \quad R5 \leftarrow AC$

$MVACAR: \quad AR \leftarrow AC$

MOVX where X $\in$ {R,R1,R2,R3,R4,R5} This operation is to move the value in the specified register to AC as per the identified name of the instruction and the next FETCH cycle is commenced after the executed single state instruction which are stated below.

$MOVR: \quad AC \leftarrow R$

$MOVR1: \quad AC \leftarrow R1$

$MOVR2: \quad AC \leftarrow R2$

$MOVR3: \quad AC \leftarrow R3$

$MOVR4: \quad ACR4 \leftarrow R4$

$MOVR5: \quad ACR5 \leftarrow R5$

INCX where X $\in$ {R1,R2,R3,R4,R5,AC} This operation is to increment the value in the specified register (i.e. as per the identified name of the instruction) by one and the next FETCH cycle is commenced after the execution of these single state instructions which are stated below.

$INCR1: \quad R1 \leftarrow R1 + 1$

$INCR2: \quad R2 \leftarrow R2 + 1$

$INCR3: \quad R3 \leftarrow R3 + 1$

$INCR4: \quad R4 \leftarrow R4 + 1$

$INCR5: \quad R5 \leftarrow R5 + 1$

$INCAC: \quad AC \leftarrow AC + 1$

MUL2 and MUL4 MUL2 is to multiply by 2, which is equivalent to shifting the value (in AC) to the left by 1 digit in binary format, where MUL4 is to multiply by 4 which is the binary equivalent of shifting the value (in AC) to the left by 2 digits. The next FETCH cycle is commenced after the execution of these single state instructions.

$$MUL2: \quad AC \leftarrow AC << 1$$

$$MUL4: \quad AC \leftarrow AC << 2$$

DIV16 This operation is to divide by 16 which is equivalent to shifting the value (in AC) to the right by 4 digits in binary format. The next FETCH cycle is commenced after the execution of this single state instruction.

$$DIV16: \quad AC \leftarrow AC >> 4$$

ADD and SUB These operations are to add the value in the specified register to the current value in AC and then, load to AC itself. The next FETCH cycle is commenced after the execution of these single state instructions.

$$ADD: \quad C = A + B$$

$$SUB: \quad C = A - B;\ IF(AC == 0)\ THEN\ Z = 0\ ELSE\ Z = 1$$

JUMPNZ This operation is to check the zero flag and if $Z = 0$ (i.e. AC is zero), then PC is incremented by one without any jump occurring. If $Z = 1$, then the three states which are similar to that of JUMP instruction proceeds. afterwards, the next FETCH cycle is commenced at the branched location.

$$JUMPNZ1: \quad NEXT\_INS = JUMPNZY1\ IF(Z == 1)\ ELSE\ JUMPNZN1$$

$$JUMPNZY1: \quad IR \leftarrow IROM[PC], FETCH$$

$$JUMPNZY2: \quad IR \leftarrow IROM[PC], FETCH$$

$$JUMPNZY3: \quad PC \leftarrow IROM[PC]$$

$$JUMPNZN1: \quad PC \leftarrow PC + 1$$

ADDM This operation is to add the value in the specified register to the current value in AC and then load to AC itself. The next FETCH cycle is commenced after the execution of this instruction.

$$ADDM1: \quad IR \leftarrow IROM[PC], FETCH$$

$$ADDM2: \quad R \leftarrow IROM[PC]$$

$$ADDM3: \quad AC \leftarrow AC + R$$

END This operation is to indicate finishing the computation and set the finish flag to HIGH

$$END: \quad FINISH \leftarrow HIGH$$

3.5 Assembly code

1. CLAC
2. MVACR
3. MVACR1
4. MVACR2
5. MVACR3
6. MVACR4
7. MVACR5
8. INCR1
9. INCR2
10. CLAC
11. MVACR4
12. MOVR1
13. DECAC
14. MUL4
15. MUL4
16. MUL4
17. MUL4
18. MVACR
19. MOVR2
20. DECAC
21. ADD
22. MVACR3
23. MVACAR
24. LDAC
25. MVACR

Micro-instruction	ALU Operation	C Bus Enable								Increment Enable								Memory Signals				Read Enable
		PC	AR	R5	R4	R3	R2	R1	R	AC	PC	R5	R4	R3	R2	R1	R	AC	R/W	FETCH	FINISH	
NOP	PASSA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
CLAC	RESET	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0000
LDAC1	PASSB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000
LDAC2	PASSB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000
LDAC3	PASSB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0000
DECAC	DEC	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0001
MVACR	PASSB	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0001
MVACR1	PASSB	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
MVACR2	PASSB	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
MVACR3	PASSB	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
MVACR4	PASSB	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
MVACR5	PASSB	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
MVACAR	PASSB	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
MOVR	PASSB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0011
MOVR1	PASSB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0100
MOVR2	PASSB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0101
MOVR3	PASSB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0110
MOVR4	PASSB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0111
MOVR5	PASSB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1000
INCR1	PASSA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0001
INCR2	PASSA	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0001
INCR3	PASSA	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0001
INCAC	PASSA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0001
INCR5	PASSA	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0001
STAC	PASSA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0001
MUL2	LSHIFT1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0001
MUL4	LSHIFT2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0001
DIV16	RSHIFT4	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0001
ADD	ADD	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0001
SUB	SUB	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0001
JUMPNZ1	DEC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000
JUMPNZY1	PASSA	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0000
JUMPNZY2	PASSA	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0000
JUMPNZY3	PASSA	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1001
JUMPNZN1	PASSA	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0000
ADDM1	PASSB	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0000
ADDM2	PASSB	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1001
ADDM3	ADD	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0011
END	PASSA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0001

Figure 3.2: Micro-instructions and the control signals

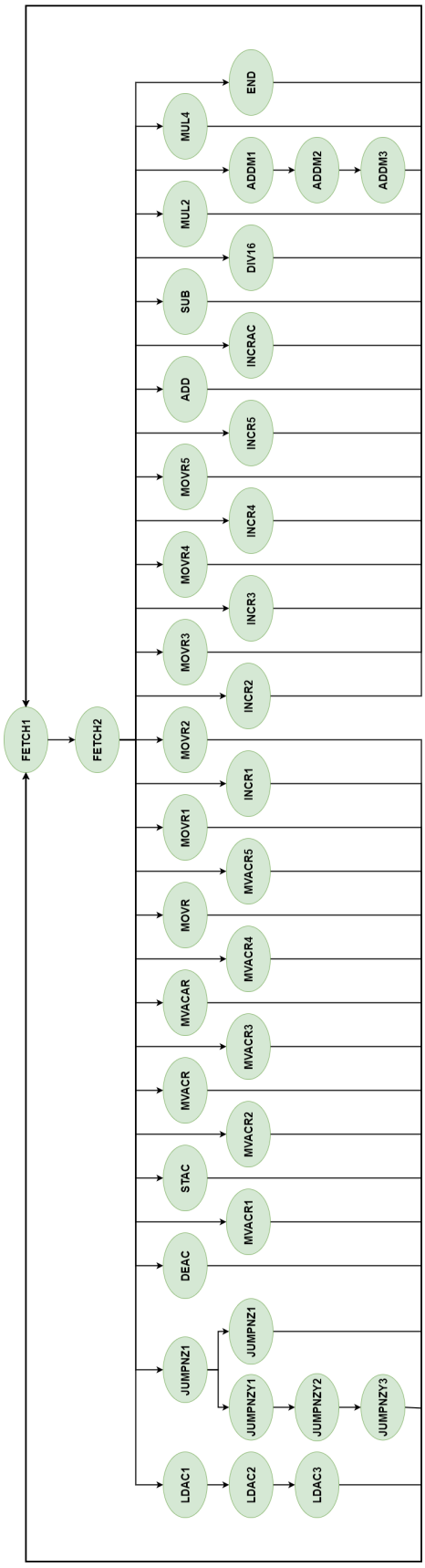

Figure 3.3: State machine

26. MOVR4
27. ADD
28. MVACR4
29. INCR3
30. MOVR3
31. MVACAR
32. LDAC
33. MUL2
34. MVACR
35. MOVR4
36. ADD
37. MVACR4
38. INCR3
39. MOVR3
40. MVACAR
41. LDAC
42. MVACR
43. MOVR4
44. ADD
45. MVACR4
46. MOVR3
47. ADDM
48. “254”
49. MVACAR
50. LDAC
51. MUL2
52. MVACR
53. MOVR4
54. ADD

55. MVACR4
56. INCR3
57. MOVR3
58. MVACAR
59. LDAC23
60. MUL4
61. MVACR
62. MOVR4
63. ADD
64. MVACR4
65. INCR3
66. MOVR3
67. MVACAR
68. LDAC
69. MUL2
70. MVACR
71. MOVR4
72. ADD
73. MVACR4
74. MOVR3
75. ADDM
76. “254”
77. MVACAR
78. LDAC
79. MVACR
80. MOVR4
81. ADD
82. MVACR4
83. INCR3

84. MOVR3
85. MVACAR
86. LDAC
87. MUL2
88. MVACR
89. MOVR4
90. ADD
91. MVACR4
92. INCR3
93. MOVR3
94. MVACAR
95. LDAC
96. MVACR
97. MOVR4
98. ADD
99. DIV16
100. MVACR4
101. MOVR5
102. MVACAR
103. MOVR4
104. STAC24
105. INCR2
106. INCR2
107. INCR5
108. MOVR2
109. MVACR
110. CLAC
111. ADDM
112. “253”

113. SUB
114. JPNZ
115. “10”
116. CLAC
117. MVACR2
118. INCR2
119. INCR1
120. INCR1
121. MOVR1
122. MVACR
123. CLAC
124. ADDM
125. “253”
126. SUB
127. JPNZ
128. “10”
129. END
130. NOP

Chapter 4

RTL Modules

4.1 16-bit register with increment

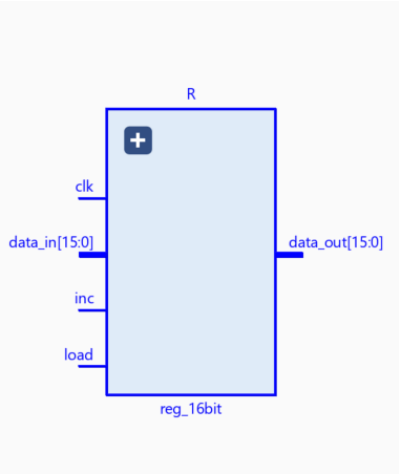

Figure 4.1: 16-bit register

This module is utilized to instantiate registers, and then those registers are intended to store intermediate 16-bit values while processing is being executed. The module consists of an increment flag that could increase the stored value in the register by 1.

The following represents the functions of the input and output ports of the module.

- load: Data is written to the register only if the load is given as 1
- inc: If the value of the register is needed to increment, this input should be 1
- clk: Data loading and increment are performed on the positive edge of the clock. This gives the generated clock as input
- data_in: Data, which is needed to be written in the register, should be given to this port
- data_out: The current value in the register is always available through this port

This module is designed to have seven instances in the processor, which are:

- AC: Accumulator
- R: General purpose register

- R1: General purpose register
- R2: General purpose register
- R3: General purpose register
- R4: General purpose register
- R5: General purpose register

4.2 Address Register

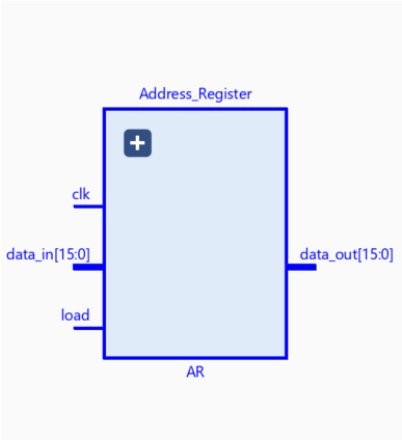

Figure 4.2: Address Register

Purpose of the register is containing the address of the data need to be loaded from the Data Ram or the address that need to be stored to the Data Ram.

The following represents the functions of the input and output ports of the module.

- load: When input is equal to one, address is written to the register.
- clk: This inputs the generated clock since address is loaded on clock.
- data_in: These inputs the 16-bit address to be written in the register.
- data_out: The 16 -bit address in the register is available through this output.

4.3 Program Counter

Program counter contains the next address of the instruction to be fetched in the instruction ROM. As this module contain a increment flag this increment can be done without an ALU operations.

The following represents the functions of the input and output ports of the module.

- load: When input is equal to one, address is written to the register.
- inc: When increment flag is one, PC is incremented by 1.
- clk: This inputs the generated clock since address is loaded and incremented on clock.
- data_in: These inputs the 8-bit address to be written in the register.
- data_out: The address in the 8-bit register is available through this output.

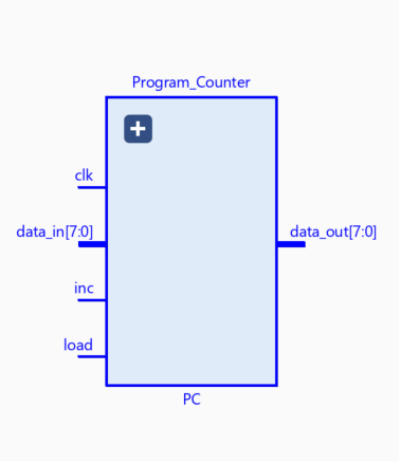

Figure 4.3: Program Counter

4.4 Instruction Register

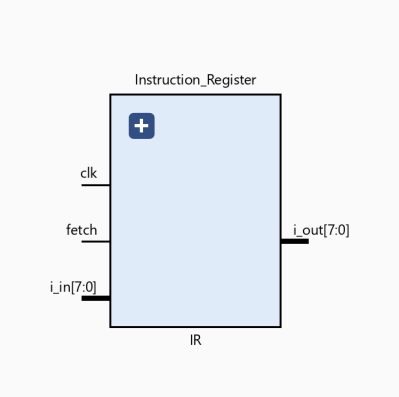

Figure 4.4: Instruction Register

Purpose of the register is storing the instructions fetched by the instruction ROM. Output of the instruction register connects with the control unit and given instructions are decoded in the control unit.

The following represents the functions of the input and output ports of the module.

- fetch: When input is equal to one, instruction is fetched.
- inc: When increment flag is one, PC is incremented by 1.
- clk: This inputs the generated clock to this clock driven register.
- i_in: 8-bit data is given as input from the instruction ROM.
- i_out: Output is connected to control unit and output instruction is decoded in the control unit.

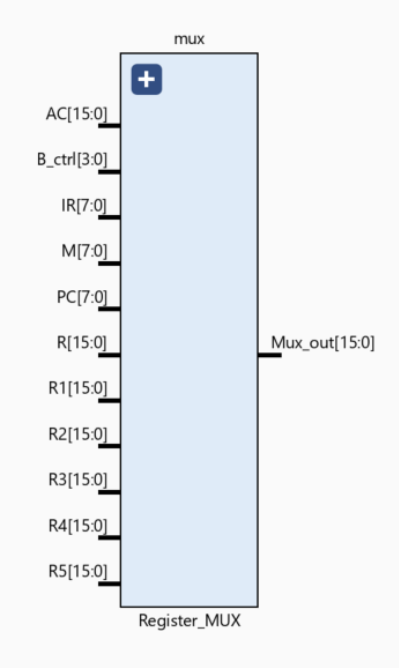

Figure 4.5: Register Multiplexer

4.5 Register Multiplexer

ALU required A bus value and B bus value; A bus value is the Accumulator (AC) value. For B bus there are several values coming from different registers. Therefore, the main purpose of the register multiplexer is selecting one input from several input signals according to the control signal. Ten register outputs are connected to B bus, therefore 4-bit control signal is sufficient for controlling the input. 16-bit value comes out as the output.

4.6 Clock Divider

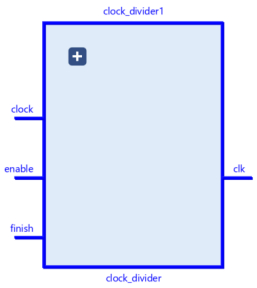

Figure 4.6: Clock Divider

Clock divider can be used to obtain the reduced clock rate of original clock rate. Original clock rate of the board is 50 MHz When it is divided to a reduced clock some operations can be done with a sufficient time. Following module clock input is 50MHz and Clock output is 1MHz.

4.7 Arithmetic and Logic unit

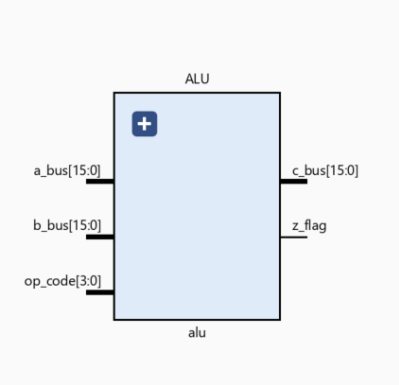

Figure 4.7: Arithmetic and Logic unit

Arithmetic and Logic operations are implemented with ALU unit. ALU unit consists with two input signals (A bus and B bus), one control signal and two output signals (C bus and Z flag). A bus is a 16-bit bus and it is the output of the accumulator (AC). B bus is a 16-bit bus which can take one of register value (AC, PC, data memory, IR, R, R1, R2, R3, R4 and, R5) using register multiplexer. Op code is a bit control signal which is capable of controlling the ALU operations. C bus is 16-bit bus and it is connected to input of the AC, AR, R, R1, R2, R3, R4 and R5 registers. Z flag is used for conditional statements. When the arithmetic operation is subtraction, if the output is zero, z flag is 1 and if the output is non-zero, z flag is 0.

Table 4.1: ALU Operations

ALU Operation	Operation	Opcode
ADD	C = A + B	0000
SUB	C = A - B	0001
RESET	C = 0	0010
PASSA	C = A	0011
PASSB	C = B	0100
INC	C = A + 1	0101
DEC	C = A - 1	0110
LSHIFT1	C = A * 2	0111
LSHIFT2	C = A * 4	1000
RSHIFT4	C = A / 16	1001

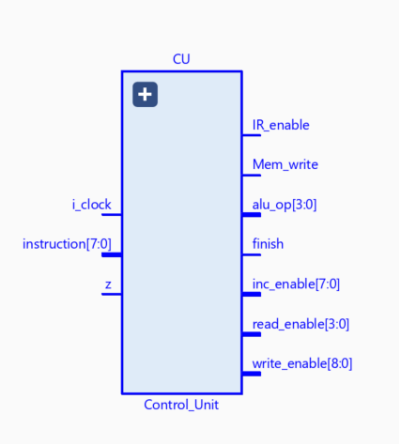

Figure 4.8: Control Unit

4.8 Control Unit

The main objective of the control unit is decoding the instructions and sending control signals to relevant places to execute the instructions. Control unit consists of all the control signals of all the micro instructions. Instructions are received from instruction register as an input. Other inputs are clock and the Z flag. As mentioned previously z flags are used for conditional statements. After decoding the instructions following outputs are sent as control signals.

- ir_enable: For fetching instructions instruction register should be received a control signal (HIGH state) as a input. This control signal is sent through the IR_enable output. If the IR_enable signal is HIGH IR can fetch the instructions.
- read_enable: 4-bit control signal send as output from this port. This control decides the which register or the memory is accessible for the B bus (AC, PC, data memory, IR, R, R1, R2, R3, R4 and, R5). Following table shows the accessibility of the register for B bus according to the 4-bit control signal.

Table 4.2: ALU Operations

alu_op	Operation	
ADD	0000	
SUB	0001	
RESET	0010	
PASSA	0011	
PASSB	0100	
INC	C = A + 1	0101
DEC	0110	
LSHIFT1	0111	
LSHIFT2	1000	
RSHIFT4	1001	

- mem_write: Output is connected to the data RAM. When the data RAM required to stored the data,

control signal coming through mem_write should be HIGH.

- i_in: 8-bit data is given as input from the instruction ROM.
- write_enable: After ALU operation one output is connected to C bus. Through the C bus values can be stored in relevant registers. According to relevant 9-bit control signal respective registers can be written. Each bit goes to load input of their respective register.

Figure 4.9: Bits of write enable signal

- alu_op: This output is sent to the Arithmetic Logic Unit for implementing ALU operation. In processor design we required 10 operations. Therfore, 4-bit control signal required for handling the ALU.
- inc_enable:8-bit control signal is received as output from the control unit which decides the registers that should be incremented. Those eight bits are assigned for the above registers separately.

Figure 4.10: Bits of increment enable signal

- finish: When the all the processing parts of the processor is finished, data is ready to send to the computer with UART communication. Communication part is enabled when the finish signal becomes HIGH.

Table 4.3: Bit patterns of the select signal of Register multiplexer

read_enable	B bus
0000	Memory
0001	AC
0010	PC
0011	R
0100	R1
0101	R2
0110	R3
0111	R4
1000	R5
1001	IR

4.9 Processor

All the modules that have been mentioned previously included in the module (Except memory modules and the communication modules). This is consisted with four inputs and six outputs.

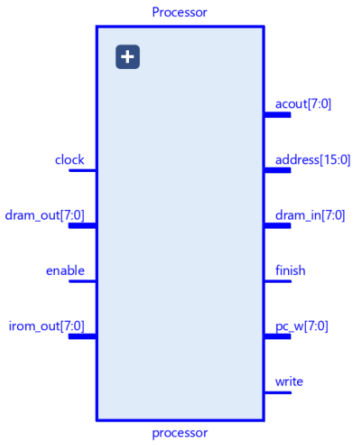

Figure 4.11: Processor

- clock: to input the generated clock
- enable: when the input value equals to one, it indicates the data communication with the computer is finished and processor is enabled for processing part.
- dram_out: Output of the data RAM used as the input.
- irom_out: The instructions come from the Instruction ROM gives as the input.
- address: The address of the data to be stored in data RAM or data to be read from the data RAM is given by this output.
- dram_in: This outputs the data that needs to be written in the Data Ram after processing.
- finish: When the all the processing parts of the processor is finished, finish flag goes to 1.
- write: If the processed data needs to be written in the data RAM, write signal should be HIGH.
- acout: Current value of the AC (accumulator) is given by this. This gives the current value of the accumulator.
- inc: When increment flag is one, PC is incremented by 1.
- pc_w: This gives the address of the next instruction to be fetched to the Instruction memory. register. .

4.10 DATA RAM

Data RAM module is used to store the original image and processed image. Original image is 256 X 256 pixels. Therfore required space for storing the data is 65536 memory locations. Pixel values are varying from 0-255; 8-bit is enough for represent a pixel value. Additional memory locations are not required for the processed image as processed image pixel values are saved in the relevant space of the original image. 16-bit address is required for represent 65536 memory locations.

- clock: Generated clock is given as input.
- write: When data need to be written write value should be 1 to enable the writing signal.
- data_in: Gives the required data to be written in the Data memory as an input.
- address: Gives the address of the data memory used to store or read data.

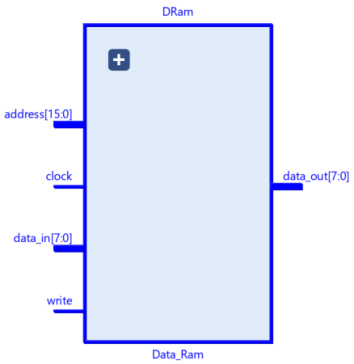

Figure 4.12: DATA RAM

- data_out: This outputs the value of the memory location given as address.

4.11 Instruction ROM

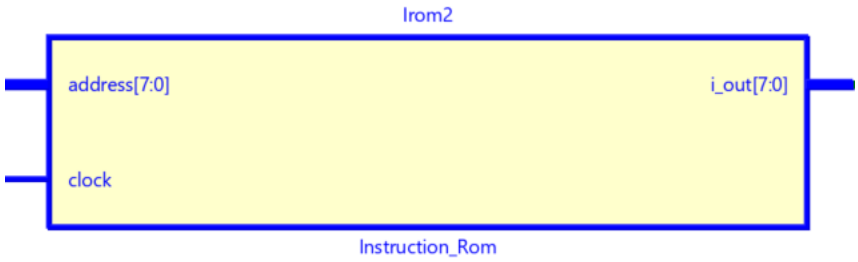

Figure 4.13: Instruction ROM

Instruction ROM module is used to store the instructions. Our designed assembly code contained 130 instructions. Therefore 256 memory locations are required and each memory locations can be represented with 8-bit memory address. Clock and address of the instruction to be fetched are given as the inputs. 8-bit instruction is available as output for given input address.

4.12 UART Transmitter

UART transmitter module is used to send 8-bit pixel values as serial data to the computer. Following inputs and outputs are included in this module.

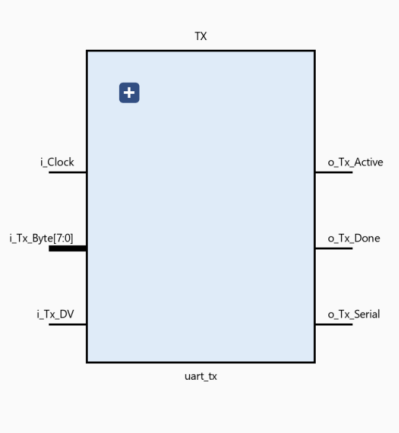

Figure 4.14: UART Transmitter

- i_Clock: to input generated clock.
- i_Tx_DV: When this flag is equal to one (HIGH), transmission is occurred.
- i_Tx_Byte: The 8-bit pixel value that requires to be sent.
- o_Tx_Active: This indicates that transmitter is busy while transmitting a pixel.
- o_Tx_Serial: Serial data goes through this output.
- o_Tx_Done: This indicates that transmission is done.

4.13 UART Receiver

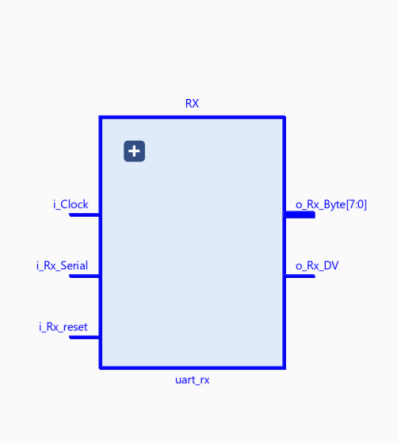

Figure 4.15: UART Receiver

UART receiver module is used to obtain the serial data values from the computer and send them to the Data RAM. In here incoming serial data are grouped to one byte. Following inputs and outputs are included in this module.

- i_clock: inputs generated clock.
- i_Rx_Serial: Inputs Serial data.
- i_Rx_reset: Reset the output value.

- o_Rx_DV: Indicates that receiving is done.
- o_Rx_Byte: Constructed byte using serial data.

4.14 UART

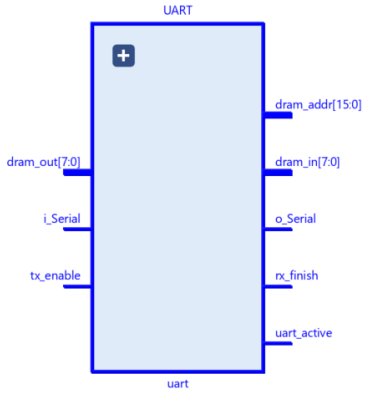

Figure 4.16: UART

With the combination of UART receiver and UART transmitter modules this module has been built. Sending and receiving the pixel values are the main process of the UART module. Following inputs and outputs are included in this module.

The following represents the functions of the input and output ports of the module.

- fetch: When input is equal to one, instruction is fetched.
- inc: When increment flag is one, PC is incremented by 1.
- clk: This inputs the generated clock to this clock driven register.
- i_in: 8-bit data is given as input from the instruction ROM.
- i_out: Output is connected to control unit and output instruction is decoded in the control unit.
- clock: to input generated clock.
- i_Serial: inputs serial data comes to the board from the computer.
- tx_enable: Data transmission is occurred when the tx_enable value is HIGH.
- dram_out: Inputs required pixel values to be transmitted from the data RAM.
- rx_finish: rx_finish value is assigned as HIGH, when all the pixels are received from the computer. This signal can be used to start the processing part of the processor
- uart_active: UART Transmitter or UART Receiver modules are in active mode when the uart_active value is HIGH.
- o_Serial: Processed image pixel values are sent as serial data in this output.
- dram_in: constructed byte by UART Receiver is sent to the Data memory using this output.
- dram_addr: Address of the data memory.

4.15 DATA RAM Input select

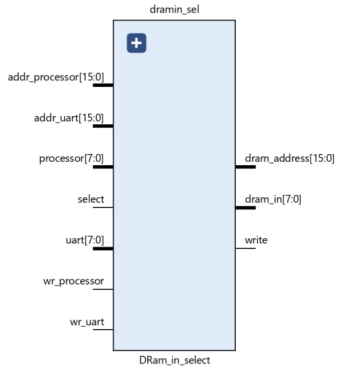

Figure 4.17: DATA RAM Input select

Data is received from two sources from UART communication (original image data comes from the computer) and from the processor (Processed image data after the down sampling procedure). data_in, address and write inputs are required for data writing procedure of the data memory. select input is decides the source where the followed 3 inputs are coming from. If select=1 inputs are coming from the processor, if select= 0 inputs are coming from the UART.

4.16 DATA RAM Output select

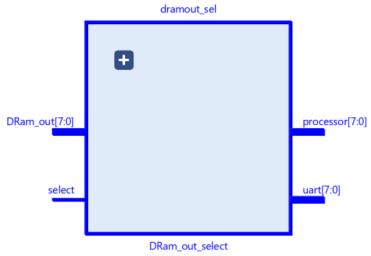

Figure 4.18: DATA RAM Output select

Data in the Data memory should have access for UART communication (To send processed image data to the computer) and for the processor (Original image data for the down sampling procedure). select input is decides the which one out of UART and processor get the output of the Data. If select=1 outputs are available for the UART communication, if select= 0 outputs are available for the processor.

4.17 TOP module

Combination of all main modules has been created the TOP module. Top module contains processor, clock divider, UART modules, memory modules and memory control modules. Following inputs and outputs are available in the TOP module.

- clock: Inputs original 50MHz clock of the board.
- serial_in: Inputs serial data.
- transmit: transmit value becomes HIGH, when the transmission of
- down-sampled image is completed.
- serial_out: outputs serial data.
- finish_out: Processing part of the processor is finished, this becomes HIGH.
- ac_w: Outputs the current value of the accumulator.

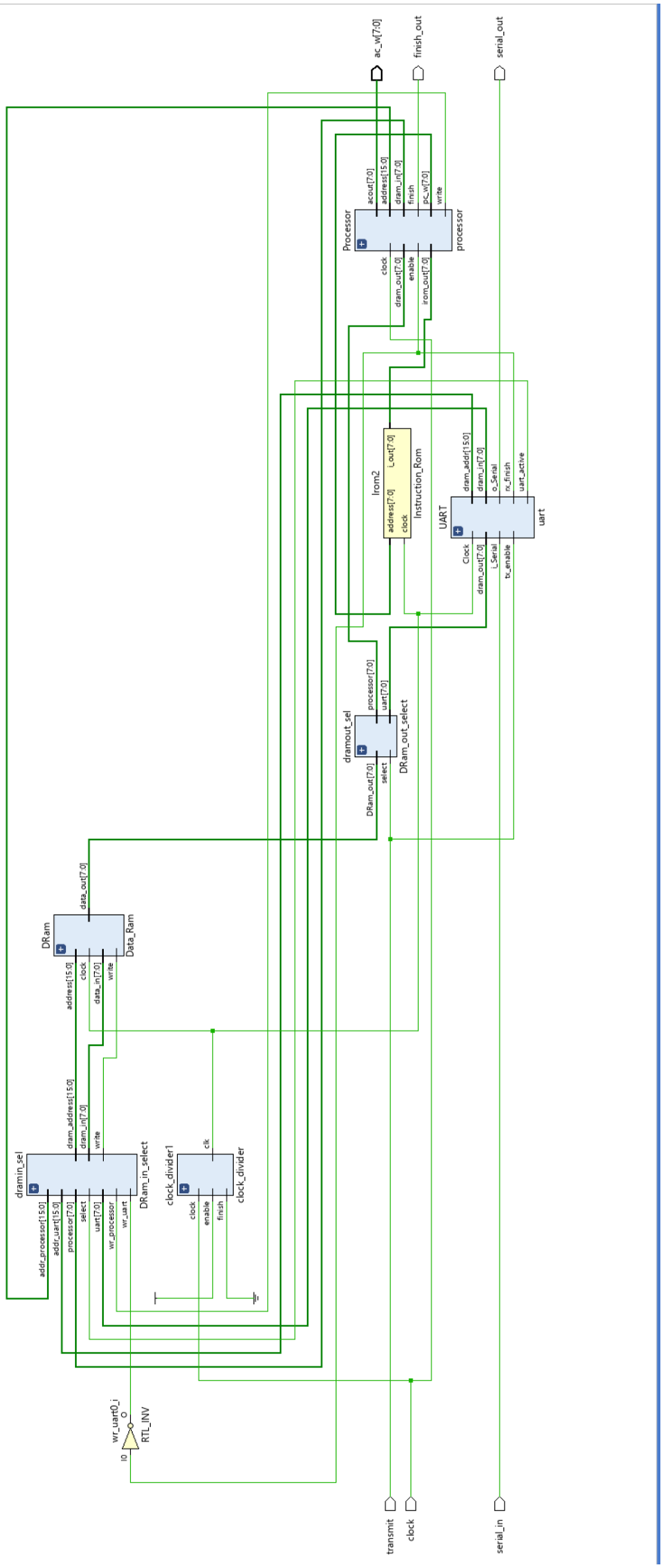

Figure 4.19: RTL view of TOP module

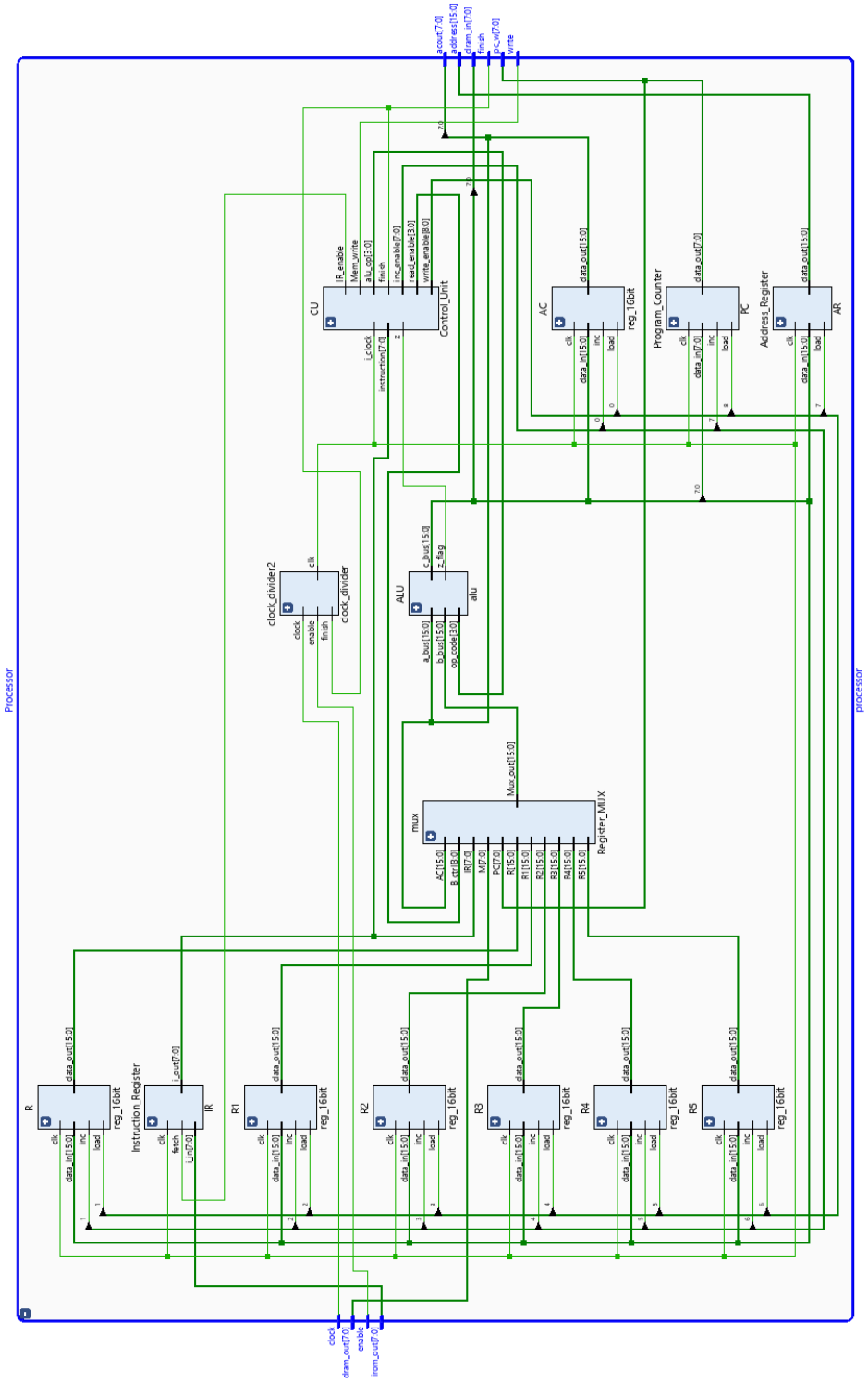

Figure 4.20: RTL view of Processor

Chapter 5

Principles of Operation and Simulation-based Testings

5.1 Simulation of Instruction Set Architecture

After the development of the Instruction Set Architecture and assembly code, initially, using MATLAB, the results were simulated and their satisfactory accuracy was confirmed. Figure 6.1 shows the input original image along with the output down-sampled image. The implementation is attached in the Appendix 8.1.1.

(a) Original input image with size (256×256)

(b) Down-sampled image with size (128×128)

Figure 5.1: ISA Simulation based on MATLAB implementation

5.2 Simulation of Processor

After each RTL module was implemented, we independently tested it using verilog testbenches. The Vivado simulator carried out the simulation. Vivado simulation results can be referred in case of recognizing whether instructions are synchronizing with the internal clock signal and no such faults. The information that follows explains how we executed the ALU simulation.

5.2.1 Simulation of the ALU

```
module ALU_tb();
    reg [15:0] a_bus;
    reg [15:0] b_bus;
    reg [3:0] op_code;
     wire [15:0] c_bus;
     wire z_flag;

    ALU dut(.a_bus(a_bus), .b_bus(b_bus), .c_bus(c_bus), .op_code(op_code), .z_flag(z_flag));

    initial begin
        a_bus = 16'd6; b_bus = 16'd6; op_code = 4'b0001;
        #100;
        a_bus = 16'd6; b_bus = 16'd5; op_code = 4'b0111;
        #50;
        //a_bus = 16'd6; b_bus = 16'd5; op_code = 4'b1000;
        //#50;
        //a_bus = 16'd6; b_bus = 16'd5; op_code = 4'b1001;
        //#50;
    end

endmodule //alu_TB
```

According to the ALU module's test bench, which is shown above, we first send an opcode to the ALU to perform the SUB instruction.

```
SUB: begin
        c_bus = a_bus - b_bus;
        z_flag=(c_bus==16'd0)?1'b1:1'b0;
    end
```

The value of the A bus is 6, and the value of the B bus is also 6. The result of subtracting 6 from 6 is 0. Therefore, once the operation is finished, the C bus value become zero. The z flag also changes to 1 at the same moment.

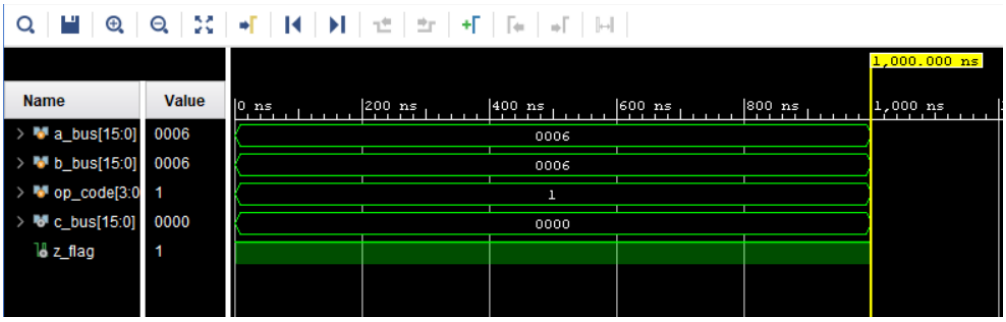

Figure 5.2: Simulation results for SUB

The LSHIFT1 instruction is sent to the ALU after the SUB operation has been completed in 100 time units.

```
LSHIFT1: c_bus = a_bus<<1;
```

The value of the A bus is set to 6, and the value of the B bus is set to 5. Shifting left 6 by 1 bit yields 12 as the outcome. As a result, once the procedure is finished, the C bus value is changed to 12. The previous operation has resulted in the z flag still being set to 1.

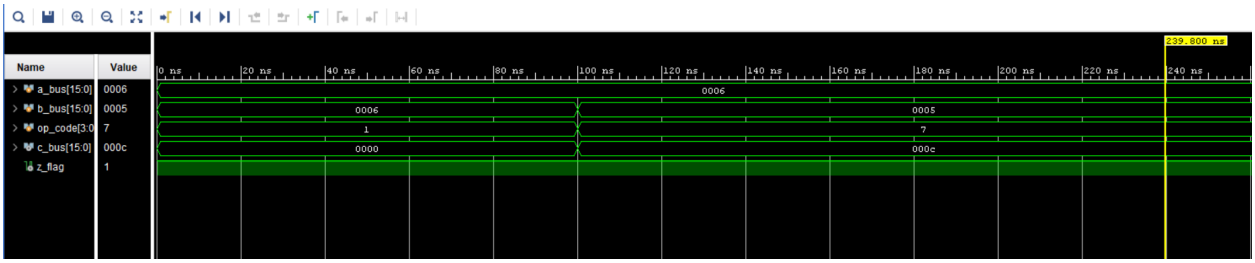

Figure 5.3: Simulation results for SUB and LSHIFT1

5.2.2 Processor simulation using ModelSim

The same module-wise simulations were carried upon Altera ModelSim simulator where RTL designs were synthesized using Quartus Prime 20.1 software. Apart from that, the complete pipeline of the processor in RTL design is also checked using ModelSim using the following procedure.

- To test the processor on addition operation, the following instruction ROM is developed.
  1. Rom[0] 8'd8
  2. Rom[1]: 8'd28
  3. Rom[2]: 8'd28
  4. Rom[3]: 8'd28
  5. Rom[4]: 8'd28
  6. Rom[5]: 8'd15
  7. Rom[6]: 8'd8
  8. Rom[7]: 8'd28
  9. Rom[8]: 8'd28
  10. Rom[9]: 8'd21
  11. Rom[10]: 8'd38
- ModelSim simulation results can be referred in case of recognizing whether instructions are synchronized and seemingly error-free.
- The following figure depicts the simulation results from ModelSim for the addition operation.
- Further, the development is extended to test another operation: AND with a replacement on ADD operation and the expected result is obtained through the implementation
- Therefore, it is concluded that the processor is behaving as its expectation and thus, has the potential to extend to the intended task with its own IROM.

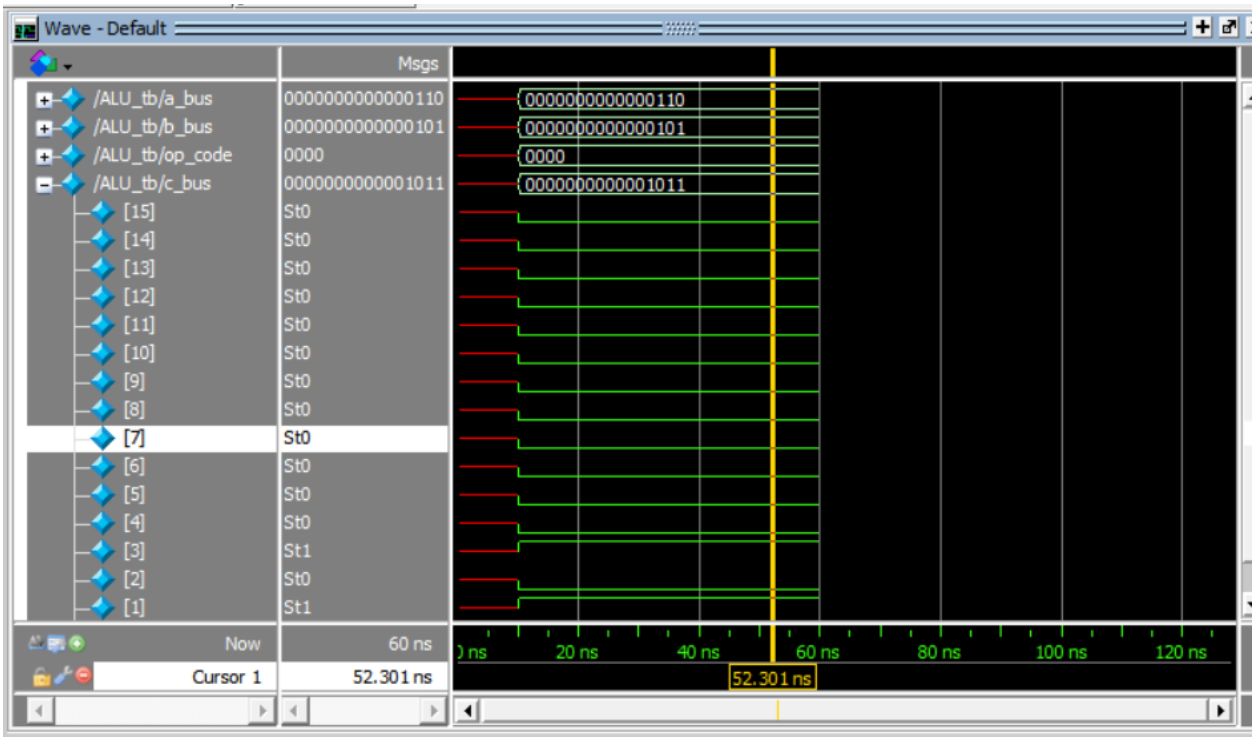

Figure 5.4: ModelSim-based processor simulation results for addition: Here, 6 and 5 are set as input and the correct output 11 is obtained through the processor (i.e. ALU)

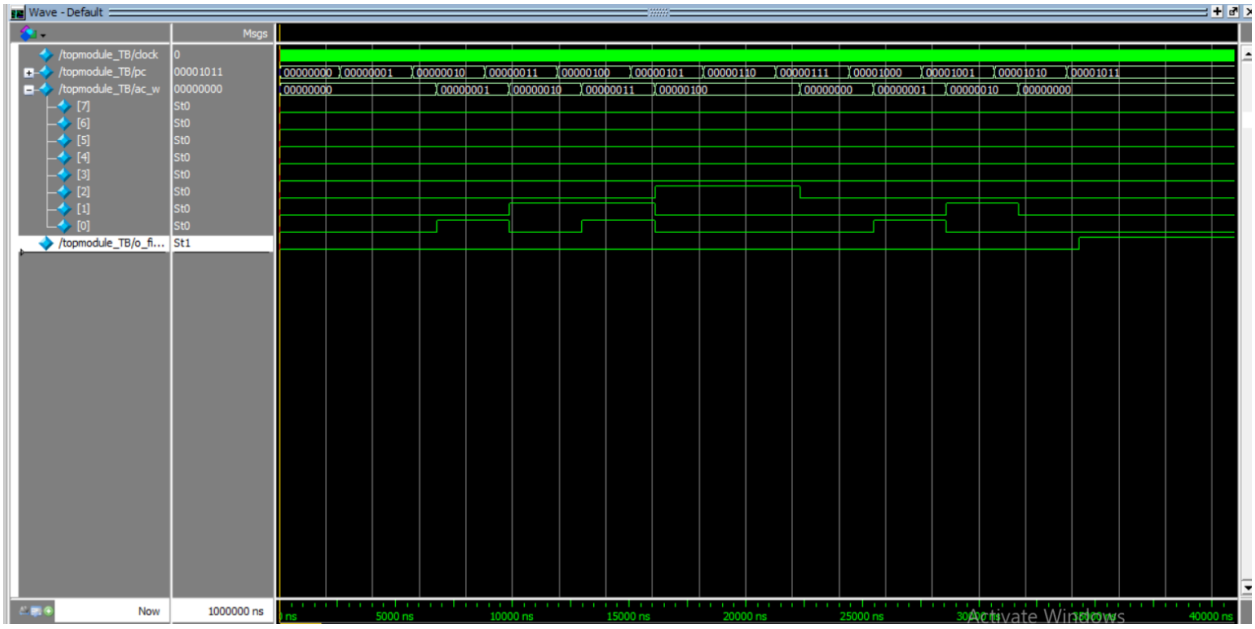

Figure 5.5: ModelSim-based processor simulation results for addition: Here, 2 and 4 are set as input and the correct output 0 is obtained through the processor

Chapter 6

Results Verification Procedure

6.1 Results Verification

The following figures visually represent the results through the processor implementation on the simulator platform for different image inputs. All input images are (256×256) images while the output images from the processor are of size (128×128).

(a) Input image 01 with size (256×256)

(b) Input image 02 with size (256×256)

(c) Down-sampled image 01 with size (128×128)

(d) Down-sampled image 02 with size (128×128)

Figure 6.1: Visual results from the developed custom processor

6.2 Error Analysis

Here, the output down-sampled images through the simulated processor, the base algorithm on MATLAB and the inbuilt *imresize* method are analyzed and compared using the following error analysis matrices. The error analysis and evaluation is implemented on MATLAB.

- Sum of square differences (SSD): Here, the pixel-wise error is considered through the fashion in-

troduced in equation 6.1 where the intensity difference in each corresponding pixels are calculated, squared and then, summed.

$$SSD = \sum_{i=1}^{127} \sum_{j=1}^{127} (F[i,j] - M[i,j])^2 \tag{6.1}$$

where F is the output down-sampled image from the simulated processor while M is the output down-sampled image from the MATLAB-based algorithm.

- Error pixel percentage (EPP): Here, the number of error pixels, the number of non-zero elements in error matrix $(F - M)$, is divided by the total pixels as a percentage.

$$EPP = \frac{NEP}{TP} \times 100\% \tag{6.2}$$

where NEP is the number of error pixels and TP is the total pixels.

- Maximum pixel error: The maximum difference between two corresponding pixels. This can be obtained by getting the maximum element of the error matrix.

The following represents the results from error evaluation through above MATLAB implementations.

```
Sum of Square Differences : 4052710
Maximum Pixel Error : 255
```

Figure 6.2: Error analysis between the output image from custom simulated processor vs the MATLAB *imresize* function results - MATLAB implementation. The SSD error between the simulated processor and the algorithm is observed to be zero.

Chapter 7

Discussion, Acknowledgements and Bibliography

Discussion

- In the ISA implementation, we decided to use five general-purpose registers (except R) to save time for computations. Therefore Gaussian filtering and the down sampling procedures occur at the same time.
- Rather than using multiplexers to route the access to the Data RAM, a two-port RAM could be utilized for more convenient routing access.
- Rearranging the processed data is not required in the proposed implementation. Data can be sent from the moment the last needed Gaussian filtering is done.
- Due to the FPGA board limitation in terms of access, the final output had to be obtained using a Vivado/ModelSim simulations.
- The processor is capable only of down-sampling one image at a time. Also, the program should be reprogrammed if the user uses another image (256 X 256) or uses a different size image (recommended image size 256 X 256). This could be achieved if we could reset the CPU back to the initial states after the processing is complete.
- In the future, the processor will be enhanced to perform various other image processing tasks such as inversion, gradient finding if assembly codes are written properly for the given task.

Acknowledgements We would like to extend our sincere gratitude to **Dr. Jayathu Samarawickrama** who kindly guided us with his immense expertise in the field of processor design and implementation through the initial introduction to Verilog/SystemVerilog up-to complex processor design architectures.

Bibliography

Lecture materials by Dr. Jayathu Samarawickrama

https://nandland.com/. (2022). Uart, serial port, rs-232 interface. https://nandland.com/uart-serial-port-module/

Chapter 8

Appendix

8.1 MATLAB Codes

8.1.1 Intruction Set Architecture Simulator

```
clear;
close all;

%Define all the used resgisters
global L;
global T;
global C1;
global C2;
global C3;
global E;
global DRAM;
global AC;
global MAR;
global Z;
global IRAM;
global i;

L = 0;T = 0;C1 = 0;
C2 = 0;C3 = 0;E = 0;
AC = 0;MAR = 0;Z = 0;

%Define the Data Memory
%=======================================

%Load the image
im = imread('ex2.jpg');

im = rgb2gray(im);
img = im;

%Display the original image
figure;
title('Original Image');
imshow(im);

im = imresize(im, 2);
```

```
im = double(im);

%Cast the image to a 1D array
DRAM = reshape(im',[1,512*512]);

%Define the Instruction Memory
%=======================================

IRAM = {'CLAC'
'MVACMAR'
'MVACC2'
'LDL'
'MVACL'
'INAC'
'INAC'
'MVACC1'
'CLAC'
'INAC'
'MVACC3'
'MVL'
'DEAC'
'MUL512'
'DEAC'
'DEAC'
'MVACE'
'CLAC'
'MVACT'
'MVC1'
'SUBL'
'DEAC'
'MVACMAR'
'LDAC'
'ADDT'
'MVACT'
'MVC1'
'SUBL'
'MVACMAR'
'LDAC'
'MUL2'
'ADDT'
'MVACT'
'MVC1'
'SUBL'
'INAC'
'MVACMAR'
'LDAC'
'ADDT'
'MVACT'
'MVC1'
'DEAC'
'MVACMAR'
'LDAC'
'MUL2'
'ADDT'
'MVACT'
'MVC1'
'MVACMAR'
'LDAC'
'MUL4'
```

```
'ADDT'
'MVACT'
'MVC1'
'INAC'
'MVACMAR'
'LDAC'
'MUL2'
'ADDT'
'MVACT'
'MVC1'
'ADDL'
'DEAC'
'MVACMAR'
'LDAC'
'ADDT'
'MVACT'
'MVC1'
'ADDL'
'MVACMAR'
'LDAC'
'MUL2'
'ADDT'
'MVACT'
'MVC1'
'ADDL'
'INAC'
'MVACMAR'
'LDAC'
'ADDT'
'MVACT'
'MVC3'
'MVACMAR'
'MVT'
'DIV'
'STAC'
'MVC3'
'INAC'
'MVACC3'
'MVC2'
'INAC'
'INAC'
'MVACC2'
'MVC1'
'INAC'
'INAC'
'MVACC1'
'MVC1'
'SUBE'
'JMPZ'
'MVC2'
'SUBL'
'JMNZ'
'MVC1'
'ADDL'
'INAC'
'INAC'
'MVACC1'
'CLAC'
'MVACC2'
```

```
'JUMP'
'NOP'};


%Execute the instructions
%=======================================

i = 1;
while true
    if strcmp(IRAM(i),'NOP') == 1
        break;
    end

    assembler(IRAM(i));
    i = i + 1;
    %disp(C1)
end

%Display the downsampled image
%=========================================

%Extract the image from M-array
ds_im = DRAM(2:255*255+1);
ds_im = reshape(ds_im,[255,255]);
ds_im = ds_im';

ds_im = uint8(ds_im);

figure;
imshow(ds_im);

%Operation using the MATLAB built-in function
imNew = imresize(img,0.5);
imwrite(imNew,'im04_downsampled.jpg');
figure;
imshow(imNew);

function assembler(ins)
       global L;
       global T;
       global C1;
       global C2;
       global C3;
       global E;
       global DRAM;
       global AC;
       global MAR;
       global Z;
       global i;
       if strcmp(ins,'CLAC') == 1
           AC = 0;
       elseif strcmp(ins,'LDL') == 1
           AC = 512;
       elseif strcmp(ins,'MVACMAR') == 1
           MAR = AC;
       elseif strcmp(ins,'MVACC1') == 1
           C1 = AC;
       elseif strcmp(ins,'MVACC2') == 1
           C2 = AC;
```

```
        elseif strcmp(ins,'MVACC3') == 1
            C3 = AC;
        elseif strcmp(ins,'MVACL') == 1
            L = AC;
        elseif strcmp(ins,'MVACT') == 1
            T = AC;
        elseif strcmp(ins,'MVACE') == 1
            E = AC;
        elseif strcmp(ins,'LDAC') == 1
            AC = DRAM(MAR);
        elseif strcmp(ins,'INAC') == 1
            AC = AC + 1;
        elseif strcmp(ins,'DEAC') == 1
            AC = AC - 1;
        elseif strcmp(ins,'MVL') == 1
            AC = L;
        elseif strcmp(ins,'MVC1') == 1
            AC = C1;
        elseif strcmp(ins,'MVC2') == 1
            AC = C2;
        elseif strcmp(ins,'MVC3') == 1
            AC = C3;
        elseif strcmp(ins,'MVT') == 1
            AC = T;
        elseif strcmp(ins,'ADDT') == 1
            AC = AC + T;
        elseif strcmp(ins,'ADDL') == 1
            AC = AC + L;
        elseif strcmp(ins,'SUBL') == 1
            AC = AC - L;
            if (AC == 0)
                Z = 1;
            else
                Z = 0;
            end
        elseif strcmp(ins,'SUBE') == 1
            AC = AC - E;
            if (AC == 0)
                Z = 1;
            else
                Z = 0;
            end
        elseif strcmp(ins,'DIV') == 1
            AC = AC/16;
        elseif strcmp(ins,'MUL2') == 1
            AC = AC*2;
        elseif strcmp(ins,'MUL4') == 1
            AC = AC*4;
        elseif strcmp(ins,'MUL512') == 1
            AC = AC*512;
        elseif strcmp(ins,'STAC') == 1
            DRAM(MAR) = AC;
        elseif strcmp(ins,'JMPZ') == 1
            if Z == 1;
                i = 111;
            end
        elseif strcmp(ins,'JMNZ') == 1
            if Z == 0
                i = 17;
```

```
            end
        elseif strcmp(ins,'JUMP') == 1
            i = 17;
    end
end
```

8.1.2 Algorithm

```
clear all; close all;

data_arry=imread('asd.bmp');
data_arry=rgb2gray(data_arry);
flat_d_arry=data_arry(:);
im=double(flat_d_arry);

Value=0;
m=1;
for j=1:2:254
    for i=1:2:254
        Value=0;
        K=256*(j-1)+(i-1)+1;
        Value=Value + double(im(K));
        Value=Value + double(im(K+1)*2);
        Value=Value + double(im(K+2));
        Value=Value + double(im(K+2+254)*2);
        Value=Value + double(im(K+2+254+1)*4);
        Value=Value + double(im(K+2+254+2)*2);
        Value=Value + double(im(K+2+254+2+254));
        Value=Value + double(im(K+2+254+2+254+1)*2);
        Value=Value + double(im(K+2+254+2+254+2));
        Value=Value/16;
        im(m) = Value;
        m=m+1;
    end
end

down_image=uint8(im);
down_image=down_image(1:16129);
down_image_algo=reshape(down_image,127,127);

down_image_matlab = imresize(data_arry,[127,127]);

figure;
imshow(down_image_matlab);
imwrite(down_image_matlab,'downsampled-imresize.jpg');

figure;
imshow(down_image_algo);
imwrite(down_image_algo,'downsampled-matlab_algorithm.jpg');
```

8.1.3 Communication, Results & Error

```
clear all; close all;
% Initial serial configuration
port='COM14';
delete(instrfind);
s = serial(port);

instrfind;
s.InputBufferSize = 10000000;
s.OutputBufferSize = 10000000;
s.BaudRate = 38400;
s.Timeout = 30;

%flatten array for image
im=imread('asd.bmp');
im=rgb2gray(im);
im=double(im)';
im_flatten=im(:);
im_flatten=uint8(im_flatten);

%transmitting the image
fopen(s);
fwrite(s,im_flatten);
fclose(s);

% Initial serial configuration
s = serial(port);
s.InputBufferSize = 1000000;
s.OutputBufferSize = 1000000;
s.BaudRate = 38400;
s.Timeout = 25;

%receiving the image
fopen(s);
im_down=fread(s);
fclose(s);
im_down = uint8(im_down);
im_down=im_down([1:16384]);
down_sampled_fpga=reshape(im_down,128,128);
down_sampled_fpga=down_sampled_fpga';

figure;
imshow(down_sampled_fpga);
imwrite(down_sampled_fpga,'downsampled-processor.jpg');
fclose(s);

data_arry=imread('asd.bmp');
data_arry=rgb2gray(data_arry);
downsampled_imresize = imresize(data_arry,[127,127]);
figure;
imshow(downsampled_imresize);

data_arry=imread('asd.bmp');
data_arry=rgb2gray(data_arry);
flat_d_arry=data_arry(:);
im=double(flat_d_arry);

Value=0;
m=1;
for j=2:2:254
```

```
    for i=2:2:254
        Value=0;
        K=256*(j-1)+(i-1)+1;
        Value=Value + double(im(K));
        Value=Value + double(im(K+1)*2);
        Value=Value + double(im(K+2));
        Value=Value + double(im(K+2+254)*2);
        Value=Value + double(im(K+2+254+1)*4);
        Value=Value + double(im(K+2+254+2)*2);
        Value=Value + double(im(K+2+254+2+254));
        Value=Value + double(im(K+2+254+2+254+1)*2);
        Value=Value + double(im(K+2+254+2+254+2));
        Value=Value/16;
        im(m) = Value;
        m=m+1;
    end
end

downsampled_matlab_algo=uint8(im);
downsampled_matlab_algo=downsampled_matlab_algo(1:16129);
downsampled_algo=reshape(downsampled_matlab_algo,127,127);

figure;
imshow(downsampled_algo);
imwrite(downsampled_algo,'downsampled-processor.jpg');

error = abs(down_sampled_fpga-downsampled_algo);
error = error(:);
errorpixel =sum(abs(down_sampled_fpga-downsampled_algo)>0);
errorSquared = error.^2;
ssd = sum(errorSquared);
sum_error=sum(errorpixel);
error_per= sum_error*100/16129;

disp("Sum of Square Differences : "+ssd);
disp("Error Pixel Percentage : "+error_per+"%");
disp("Maximum Pixel Error : "+max(error));
```

8.2 Verilog Code

8.2.1 ALU

```
`timescale 1ns / 1ps
module ALU(
    input [15:0] a_bus,
    input [15:0] b_bus,
    input [3:0] op_code,
    output reg [15:0] c_bus,
    output reg z_flag
    );


parameter ADD = 4'b0000;
parameter SUB = 4'b0001;
parameter RESET = 4'b0010;
parameter PASSA = 4'b0011;
parameter PASSB = 4'b0100;
```

```
parameter INC = 4'b0101;
parameter DEC = 4'b0110;
parameter LSHIFT1 = 4'b0111;
parameter LSHIFT2 = 4'b1000;
parameter RSHIFT4 = 4'b1001;

always@ (op_code or a_bus or b_bus)
begin
    case(op_code)
    ADD: c_bus = a_bus + b_bus;
    SUB: begin
        c_bus = a_bus - b_bus;
        z_flag=(c_bus==16'd0)?1'b1:1'b0;
        end

    RESET : c_bus = 8'b0;
    PASSA: c_bus = a_bus;
    PASSB: c_bus = b_bus;
    INC: c_bus = a_bus + 8'b1;
    DEC: c_bus = a_bus - 8'b1;
    LSHIFT1: c_bus = a_bus<<1;
    LSHIFT2: c_bus = a_bus<<2;
    RSHIFT4: c_bus = a_bus>>4;
    endcase
end
endmodule
```

8.2.2 AR

```
`timescale 1ns / 1ps
module AR(

    input load,
    input clk,
    input [15:0] data_in,
    output reg [15:0] data_out
    );

always @(posedge clk)

begin
    if (load)
    begin
        data_out <= data_in;
    end
end

endmodule
```

8.2.3 Clock Divider

```
`timescale 1ns / 1ps
module clock_divider(
    input clock,
    input enable,
    input finish,
    output reg clk = 0
    );

    integer count=0;
    always@(posedge clock)
        begin
            if (enable & !finish)
            begin
                if(count==25)
                begin
                    clk = !clk;
```

```
                    count=0;
                end
                else
                begin
                    count=count+1;
                end
            end
            else
            begin
                clk = 0;
            end
        end
endmodule
```

8.2.4 Control Unit

```
`timescale 1ns / 1ps
module Control_Unit(
    input i_clock,
    input [7:0] instruction ,
    input z ,
    output reg Mem_write ,
    output reg IR_enable ,
    output reg [3:0] read_enable ,
    output reg [8:0] write_enable,
    output reg [3:0] alu_op ,
    output reg [7:0] inc_enable,
    output reg finish

    );

reg [7:0] present ;
reg [7:0] next;

parameter  IDLE = 8'd0 ,
           FETCH1 = 8'd1 ,
           FETCH2 = 8'd2 ,
           NOP = 8'd3 ,
           LDAC1 = 8'd4 ,
           LDAC2 = 8'd5 ,
           LDAC3 = 8'd45,
           STAC = 8'd6 ,
           MVACAR = 8'd7 ,
           CLAC = 8'd8 ,
           MOVR = 8'd9 ,
           MOVR1 = 8'd10,
           MOVR2 = 8'd11,
           MOVR3 = 8'd12,
           MOVR4 = 8'd13,
           MOVR5 = 8'd14,
           MVACR = 8'd15,
           MVACR1 = 8'd16,
           MVACR2 = 8'd17,
           MVACR3 = 8'd18,
           MVACR4 = 8'd19,
           MVACR5 = 8'd20 ,
           ADD = 8'd21 ,
           MUL2 = 8'd22,
           MUL4 = 8'd23,
           DIV16 = 8'd24,
           INCR1 = 8'd25,
           INCR2 = 8'd26,
           INCR3 = 8'd27,
           INCAC = 8'd28,
           INCR5 = 8'd29,
           DECAC = 8'd30,
           JUMPNZ1 = 8'd31,
           JUMPNZY1 = 8'd32,
           JUMPNZY2 = 8'd33,
           JUMPNZY3 = 8'd46,
           JUMPNZN1 = 8'd34,
           ADDM1 = 8'd35,
```

```
            ADDM2 = 8'd36,
            ADDM3 = 8'd37,
            END = 8'd38,
            SUB = 8'd39;

always @(negedge i_clock)
    present =next;

always @(present or z or instruction)
    case(present)
        default:
            begin
                read_enable<=4'b0;
                write_enable<=16'b0;
                alu_op<=3'b0;
                inc_enable<=16'b0;
                Mem_write<=1'b0;
                finish<=0; next<=FETCH1 ;
            end
        FETCH1:
            begin
                IR_enable<=1'b1 ;
                Mem_write<=1'b0;
                write_enable<=9'b000000000 ;
                inc_enable<=8'b00000000 ;
                alu_op<=4'b0011; next<=FETCH2 ;
            end
        FETCH2:
            begin
                IR_enable<=1'b0 ;
                write_enable<=9'b0 ;
                inc_enable<=8'b10000000 ;
                alu_op<=3'b0;
                next<=instruction[7:0];
            end
        LDAC1:
            begin
                read_enable<=4'd0;
                write_enable<=9'b0 ;
                inc_enable<=8'b0;
                alu_op<=4'd4;
                Mem_write<=1'b0;
                next<=LDAC2 ;
            end
        LDAC2:
            begin
                read_enable<=4'd0;
                write_enable<=9'b0 ;
                inc_enable<=8'b0;
                alu_op<=4'd4;
                Mem_write<=1'b0; next<=LDAC3 ;
            end
        LDAC3:
            begin
                read_enable<=4'd0;
                write_enable<=9'b00000001 ;
                inc_enable<=8'b0;
                alu_op<=4'd4;
                Mem_write<=1'b0;
                next<=FETCH1;
            end
        STAC:
            begin
                read_enable<=4'd1;
                write_enable<=9'b0 ;
                inc_enable<=8'b0;
                alu_op<=4'd3;
                Mem_write<=1'b1;
                next<=FETCH1;
            end
        CLAC:
            begin
                write_enable<=9'b000000001;
                inc_enable<=8'b00000000 ;
                alu_op<=4'b0010;
                next<=FETCH1;
            end
        MOVR :
```

```
            begin
                read_enable<=4'd3;
                write_enable<=9'b000000001;
                inc_enable<=8'd0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MOVR1 :
            begin
                read_enable<=4'd4;
                write_enable<=9'b000000001;
                inc_enable<=8'd0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MOVR2:
            begin
                read_enable<=4'd5;
                write_enable<=9'b000000001;
                inc_enable<=8'd0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MOVR3:
            begin
                read_enable<=4'd6;
                write_enable<=9'b000000001;
                inc_enable<=8'd0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MOVR4:
            begin
                read_enable<=4'd7;
                write_enable<=9'b000000001;
                inc_enable<=8'd0 ;
                alu_op<=4'd4; next<=FETCH1;
            end
        MOVR5 :
            begin
                read_enable<=4'd8;
                write_enable<=9'b000000001;
                inc_enable<=8'd0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MVACR:
            begin
                read_enable<=4'd1;
                write_enable<=9'b000000010 ;
                inc_enable<=8'b0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MVACR1:
            begin
                read_enable<=4'd1;
                write_enable<=9'b000000100 ;
                inc_enable<=8'b0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MVACR2 :
            begin
                read_enable<=4'd1;
                write_enable<=9'b000001000 ;
                inc_enable<=8'b0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
        MVACR3 :
            begin
                read_enable<=4'd1;
                write_enable<=9'b000010000 ;
                inc_enable<=8'b0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
```

```
213         MVACR4 :
214             begin
215                 read_enable<=4'd1;
216                 write_enable<=9'b000100000 ;
217                 inc_enable<=8'b0 ;
218                 alu_op<=4'd4;
219                 next<=FETCH1;
220             end
221         MVACR5 :
222             begin
223                 read_enable<=4'd1;
224                 write_enable<=9'b001000000 ;
225                 inc_enable<=8'b0 ;
226                 alu_op<=4'd4;
227                 next<=FETCH1;
228             end
229         ADD :
230             begin
231                 read_enable<=4'd3;
232                 write_enable<=9'b000000001;
233                 inc_enable<=8'b0 ;
234                 alu_op<=4'd0 ;
235                 next<=FETCH1 ;
236             end
237         MUL2:
238             begin
239                 read_enable<=4'd1;
240                 write_enable<=9'b000000001;
241                 inc_enable<=8'b0 ;
242                 alu_op<=4'd7 ;
243                 next<=FETCH1;
244             end
245         MUL4:
246             begin
247                 read_enable<=4'd1;
248                 write_enable<=9'b000000001;
249                 inc_enable<=8'b0 ;
250                 alu_op<=4'd8 ;
251                 next<=FETCH1;
252                 end
253         DIV16 :
254             begin
255                 read_enable<=4'd1 ;
256                 write_enable<=9'b000000001 ;
257                 inc_enable<=8'b0 ;
258                 alu_op<=4'd9 ;
259                 next<=FETCH1 ;
260             end
261         INCR1:
262             begin
263                 read_enable<=4'd1;
264                 write_enable<=9'b0;
265                 inc_enable<=8'b00000100 ;
266                 alu_op<=4'd3; next<=FETCH1 ;
267             end
268         INCR2:
269             begin
270                 read_enable<=4'd1;
271                 write_enable<=9'b0;
272                 inc_enable<=8'b00001000 ;
273                 alu_op<=4'd3;
274                 next<=FETCH1 ;
275             end
276         INCR3:
277             begin
278                 read_enable<=4'd1;
279                 write_enable<=9'b0;
280                 inc_enable<=8'b00010000;
281                 next<=FETCH1;
282             end
283         INCAC:
284             begin
285                 read_enable<=4'd1;
286                 write_enable<=9'b0;
287                 inc_enable<=8'b00000001 ;
288                 alu_op<=4'd3 ;
289                 next<=FETCH1 ;
290             end
```

```
291         INCR5:
292             begin
293                 read_enable<=4'd1;
294                 write_enable<=9'b0;
295                 inc_enable<=8'b01000000 ;
296                 alu_op<=4'd3;
297                 next<=FETCH1 ;
298             end
299         DECAC:
300             begin
301                 read_enable<=4'd1;
302                 write_enable<=9'b000000001;
303                 inc_enable<=8'b00000000 ;
304                 alu_op<=4'd6;
305                 next<=FETCH1 ;
306             end
307         JUMPNZ1:
308             begin
309                 read_enable<=4'b0 ;
310                 write_enable<=9'b0;
311                 inc_enable<=8'b0 ;
312                 alu_op<=4'd6;
313                 if(z==1)
314                     next<=JUMPNZY1 ;
315                 else next<=JUMPNZN1 ;
316             end
317         JUMPNZY1:
318             begin
319                 read_enable<=4'b0 ;
320                 IR_enable<=1'b1;
321                 write_enable<=9'b100000000 ;
322                 inc_enable<=8'b0 ;
323                 alu_op<=4'd3;
324                 next<=JUMPNZY2 ;
325             end
326         JUMPNZY2:
327             begin
328                 read_enable<=4'b0 ;
329                 IR_enable<=1'b1;
330                 write_enable<=9'b100000000 ;
331                 inc_enable<=8'b0 ;
332                 alu_op<=4'd3;
333                 next<=JUMPNZY3 ;
334             end
335         JUMPNZY3:
336             begin
337                 read_enable<=4'd9;
338                 IR_enable<=1'b0;
339                 write_enable<=9'b100000000;
340                 inc_enable<=8'b0;
341                 alu_op<=4'd3;
342                 next<=FETCH1;
343             end
344         JUMPNZN1:
345             begin
346                 read_enable<=4'd0;
347                 write_enable<=9'd0;
348                 inc_enable<=8'b10000000 ;
349                 alu_op<=4'd3 ;
350                 next<=FETCH1 ;
351             end
352         ADDM1:
353             begin
354                 read_enable<=4'd0;
355                 write_enable<=9'b0 ;
356                 IR_enable<=1'b1;
357                 inc_enable<=8'b0 ;
358                 alu_op<=4'd4;
359                 next<=ADDM2 ;
360             end
361         ADDM2:
362             begin
363                 read_enable<=4'd9 ;
364                 write_enable<=9'b000000010 ;
365                 IR_enable<=1'b0;
366                 inc_enable<=8'b10000000;
367                 alu_op<=4'd4;
368                 next<=ADDM3 ;
```

```
            end
        ADDM3:
            begin
                read_enable<=4'd3 ;
                write_enable<=9'b000000001 ;
                inc_enable <=8'b00000000 ;
                alu_op<=4'd0;
                next<=FETCH1 ;
            end
        END:
            begin
                read_enable<=4'b1;
                write_enable<=9'b0;
                inc_enable<=8'b0;
                alu_op<=4'd3;
                finish<=1'b1;
                next<=END;
            end
        NOP:
            begin
                read_enable<=4'b1;
                write_enable<=9'b0;
                inc_enable<=8'b0 ;
                alu_op<=4'd3 ;
                next<=FETCH1 ;
            end
        SUB:
            begin
                read_enable<=4'd3;
                write_enable<=9'b000000001;
                inc_enable<=8'b0 ;
                alu_op<=4'd1 ;
                next<=FETCH1 ;
            end
        MVACAR:
            begin
                read_enable<=4'd1;
                write_enable<=9'b010000000 ;
                inc_enable<=8'b0 ;
                alu_op<=4'd4;
                next<=FETCH1;
            end
    endcase

endmodule
```

8.2.5 Data RAM

```
`timescale 1ns / 1ps
module Data_Ram(
    input clock,
    input write,
    input [7:0] data_in,
    input [15:0] address,
    output reg [7:0] data_out
    );

    reg [7:0] ram [65535:0];
    always @(posedge clock)
    begin
        if(write) ram[address] <= data_in;
        else data_out <= ram[address];
    end
endmodule
```

8.2.6 Data RAM in select

```
`timescale 1ns / 1ps
module DRam_in_select(
    input wire select,
    input wire [15:0] addr_uart,
    input wire [15:0] addr_processor,
    input wire [7:0] uart,
    input wire [7:0] processor,
    input wire wr_uart,
    input wire wr_processor,
    output reg [15:0]dram_address,
    output reg [7:0] dram_in,
    output reg write
    );

    always@(select or uart or processor)
    begin
        if(select) dram_in = processor;
         else dram_in = uart;
    end

    always@(select or addr_uart or addr_processor)
    begin
        if(select) dram_address = addr_processor;
        else dram_address = addr_uart;
    end

    always@(select or wr_uart or wr_processor)
    begin
        if(select) write = wr_processor;
        else write = wr_uart;
    end
endmodule
```

8.2.7 Data RAM out select

```
`timescale 1ns / 1ps
module DRam_out_select(
    input wire [7:0] DRam_out,
    input wire select,
    output reg [7:0] processor,
    output reg [7:0] uart
    );

    always@ (select or DRam_out)
    begin
        if(select) uart = DRam_out;
        else processor = DRam_out;
    end
endmodule
```

8.2.8 Instruction ROM

```
`timescale 1ns / 1ps
module Instruction_Rom(
    input clock,
    input [7:0] address,
    output [7:0] i_out
    );

    reg [7:0] rom[0:255];
    assign i_out = rom[address];
    initial
    begin
        rom[0] = 8'd8; //CLAC
        rom[1] = 8'd15; //MVACR
        rom[2] = 8'd16; // MVACR1
```

```
        rom[3] = 8'd17; //MVACR2
        rom[4] = 8'd18; //MVACR3
        rom[5] = 8'd19; //MVACR4
        rom[6] = 8'd20; //MVACR5
        rom[7] = 8'd25; //INCR1
        rom[8] = 8'd26; //INCR2
        rom[9] = 8'd8; //CLAC
        rom[10] = 8'd19; //MVACR4
        rom[11] = 8'd10; //MOVR1
        rom[12] = 8'd30; //DECAC
        rom[13] = 8'd23; //MUL4
        rom[14] = 8'd23; //MUL4
        rom[15] = 8'd23; //MUL4
        rom[16] = 8'd23; //MUL4
        rom[17] = 8'd15; //MVACR
        rom[18] = 8'd11; //MOVR2
        rom[19] = 8'd30; //DECAC
        rom[20] = 8'd21; //ADD
        rom[21] = 8'd18; //MVACR3
        rom[22] = 8'd7; //MVACAR
        rom[23] = 8'd4; //LDAC
        rom[24] = 8'd15; //MVACR
        rom[25] = 8'd19; //MOVR4
        rom[26] = 8'd21; //ADD
        rom[27] = 8'd19; //MVACR4
        rom[28] = 8'd27; //INCR3
        rom[29] = 8'd12; //MOVR3
        rom[30] = 8'd7; //MVACAR
        rom[31] = 8'd4; //LDAC
        rom[32] = 8'd22; //MUL2
        rom[33] = 8'd15; //MVACR
        rom[34] = 8'd13; //MOVR4
        rom[35] = 8'd21; //ADD
        rom[36] = 8'd19; //MVACR4
        rom[37] = 8'd27; // INCR3
        rom[38] = 8'd12; //MOVR3
        rom[39] = 8'd7; // MVACAR
        rom[40] = 8'd4; //LDAC
        rom[41] = 8'd15; // MVACR
        rom[42] = 8'd13; //MOVR4
        rom[43] = 8'd21; // ADD
        rom[44] = 8'd19; //MVACR4
        rom[45] = 8'd12; //MOVR3
        rom[46] = 8'd35; //ADDM
        rom[47] = 8'd254; //"254"
        rom[48] = 8'd7; //MVACAR
        rom[49] = 8'd4; //LDAC
        rom[50] = 8'd22; //MUL2
        rom[51] = 8'd15; //MVACR
        rom[52] = 8'd13; //MOVR4
        rom[53] = 8'd21; //ADD
        rom[54] = 8'd19; //MVACR4
        rom[55] = 8'd27; //INCR3
        rom[56] = 8'd12; //MOVR3
        rom[57] = 8'd7; //MVACAR
        rom[58] = 8'd4; //LDAC
        rom[59] = 8'd22; //MUL4
        rom[60] = 8'd15; //MVACR
        rom[61] = 8'd13; //MOVR4
        rom[62] = 8'd21; //ADD
        rom[63] = 8'd19; //MVACR4
        rom[64] = 8'd27; //INCR3
        rom[65] = 8'd12; //MOVR3
        rom[66] = 8'd7; //MVACAR
        rom[67] = 8'd4; //LDAC
        rom[68] = 8'd22; //MUL2
        rom[69] = 8'd15; //MVACR
        rom[70] = 8'd13; //MOVR4
        rom[71] = 8'd21; //ADD
        rom[72] = 8'd19; //MVACR4
        rom[73] = 8'd12; //MOVR3
        rom[74] = 8'd35; //ADDM
        rom[75] = 8'd254; //"254"
        rom[76] = 8'd7; //MVACAR
```

```
        rom[77] = 8'd4; //LDAC
        rom[78] = 8'd15; //MVACR
        rom[79] = 8'd13; //MOVR4
        rom[80] = 8'd21; //ADD
        rom[81] = 8'd19; //MVACR4
        rom[82] = 8'd27; //INCR3
        rom[83] = 8'd12; //MOVR3
        rom[84] = 8'd7; //MVACAR
        rom[85] = 8'd4; //LDAC
        rom[86] = 8'd22; //MUL2
        rom[87] = 8'd15; //MVACR
        rom[88] = 8'd13; //MOVR4
        rom[89] = 8'd21; //ADD
        rom[90] = 8'd19; //MVACR4
        rom[91] = 8'd27; //INCR3
        rom[92] = 8'd12; //MOVR3
        rom[93] = 8'd7; //MVACAR
        rom[94] = 8'd4; //LDAC
        rom[95] = 8'd15; //MVACR
        rom[96] = 8'd13; //MOVR4
        rom[97] = 8'd21; //ADD
        rom[98] = 8'd24; //DIV16
        rom[99] = 8'd19; //MVACR4
        rom[100] = 8'd14; //MOVR5
        rom[101] = 8'd7; //MVACAR
        rom[102] = 8'd13; //MOVR4
        rom[103] = 8'd6; //STAC
        rom[104] = 8'd26; //INCR2
        rom[105] = 8'd26; //INCR2
        rom[106] = 8'd29; //INCR5
        rom[107] = 8'd11; //MOVR2
        rom[108] = 8'd15; //MVACR
        rom[109] = 8'd8; //CLAC
        rom[110] = 8'd35; //ADDM
        rom[111] = 8'd253; //"253"
        rom[112] = 8'd39; //SUB
        rom[113] = 8'd31; //JPNZ
        rom[114] = 8'd10; //"10"
        rom[115] = 8'd8; //CLAC
        rom[116] = 8'd17; //MVACR2
        rom[117] = 8'd26; //INCR2
        rom[118] = 8'd25; //INCR1
        rom[119] = 8'd25; //INCR1
        rom[120] = 8'd10; //MOVR1
        rom[121] = 8'd15; //MVACR
        rom[122] = 8'd8; //CLAC
        rom[123] = 8'd35; //ADDM
        rom[124] = 8'd253; //"253"
        rom[125] = 8'd39; //SUB
        rom[126] = 8'd31; //JPNZ
        rom[127] = 8'd10; //"10"
        rom[128] = 8'd38; //END
        rom[129] = 8'd3; //NOP
    end

endmodule
```

8.2.9 IR

```
`timescale 1ns / 1ps
module IR(

    input [7:0] i_in,
    input fetch,
    input clk,
    output reg [7:0] i_out

    );

```

```
always@ (posedge clk)
begin
    if (fetch)
    begin
        i_out <= i_in;
    end
end

endmodule
```

8.2.10 PC

```
`timescale 1ns / 1ps
module PC(
    input [7:0] data_in,
    input load,
    input inc,
    input clk,
    output reg [7:0] data_out =0
    );

always @(posedge clk)
begin
    if(load )
    begin
        data_out <= data_in[7:0];
    end

    else if(inc)
    begin
        data_out <= data_out + 8'd1;
    end
end

endmodule
```

8.2.11 Processor

```
`timescale 1ns / 1ps
module processor(
    input clock,
    input enable,
    input [7:0] dram_out,
    input [7:0] irom_out,
    output [15:0] address,
    output [7:0] dram_in,
    output finish,
    output write,
    output wire [7:0] acout,
    output wire [7:0] pc_w
    );

wire fetch,z,clk;
wire[3:0] op,b_sel;
wire[7:0] inc,ir_w;
wire[8:0] wr_select;
wire[15:0] b_bus,c_bus;
wire [15:0] r_w,r1_w,r2_w,r3_w,r4_w,r5_w,ac_w;

assign dram_in = c_bus[7:0];
assign acout = ac_w [7:0];

clock_divider clock_divider2(
.clock(clock),
.enable(enable),
.finish(finish),
```

```
.clk(clk)
);

reg_16bit AC(
.load(wr_select[0] ),
.inc(inc[0]),
.clk(clk),
.data_in(c_bus),
.data_out(ac_w)
);

reg_16bit R(
.load(wr_select[1]),
.inc(inc[1] ),
.clk(clk),
.data_in(c_bus),
.data_out(r_w)
);

reg_16bit R1(
.load(wr_select[2]),
.inc(inc[2] ),
.clk(clk),
.data_in(c_bus),
.data_out(r1_w)
);

reg_16bit R2(
.load(wr_select[3]),
.inc(inc[3]),
.clk(clk),
.data_in(c_bus),
.data_out(r2_w)
);

reg_16bit R3(
.load(wr_select[4]),
.inc(inc[4]),
.clk(clk),
.data_in(c_bus),
.data_out(r3_w)
);

reg_16bit R4(
.load(wr_select[5]),
.inc(inc[5]),
.clk(clk),
.data_in(c_bus),
.data_out(r4_w)
);

reg_16bit R5(
.load(wr_select[6]),
.inc(inc[6]),
.clk(clk),
.data_in(c_bus),
.data_out(r5_w)
);

AR Address_Register(
.load(wr_select[7]),
.clk(clk),
.data_in(c_bus),
.data_out(address)
);

PC Program_Counter(
.data_in(c_bus[7:0]),
.clk(clk),
.load(wr_select[8]),
.inc(inc[7]),
.data_out(pc_w)
);

```

```
IR Instruction_Register(
.i_in(irom_out),
.fetch(fetch),
.clk(clk),
.i_out(ir_w)
);

ALU ALU(
.a_bus(ac_w),
.b_bus(b_bus),
.op_code(op),
.c_bus(c_bus),
.z_flag(z)
);

Register_MUX mux(
. M(dram_out),
. PC(pc_w),
. IR(ir_w),
. R(r_w),
. R1(r1_w),
. R2(r2_w),
. R3(r3_w),
. R4(r4_w),
. R5(r5_w),
. AC(ac_w),
. B_ctrl(b_sel),
. Mux_out(b_bus)
);

Control_Unit CU(
.i_clock(clk),
.instruction(ir_w),
.z(z),
.Mem_write(write),
.IR_enable(fetch),
.read_enable(b_sel),
.write_enable(wr_select),
.alu_op(op),
.inc_enable(inc),
.finish(finish)
);


endmodule
```

8.2.12 16 bit Register

```
`timescale 1ns / 1ps
module reg_16bit(

    input load,
    input inc,
    input clk,
    input [15:0] data_in,
    output reg [15:0] data_out=0

    );

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

    if (inc)
    begin
        data_out <= data_out + 16'b1;
    end
```

```
end
endmodule
```

8.2.13 Register Mux

```
`timescale 1ns / 1ps
module Register_MUX(
    input [7:0] M,
    input [7:0] PC,
    input [7:0] IR,
    input [15:0] R,
    input [15:0] R1,
    input [15:0] R2,
    input [15:0] R3,
    input [15:0] R4,
    input [15:0] R5,
    input [15:0] AC,
    input [3:0] B_ctrl,
    output reg [15:0] Mux_out
    );

    always@ (B_ctrl or M or PC or R or R1 or R2 or R3 or R4 or R5 or AC or IR)

    begin
        case(B_ctrl)
            4'b0000: Mux_out <= {8'b0,M};
            4'b0001: Mux_out <= AC;
            4'b0010: Mux_out <= {8'b0,PC};
            4'b0011: Mux_out <= R;
            4'b0100: Mux_out <= R1;
            4'b0101: Mux_out <= R2;
            4'b0110: Mux_out <= R3;
            4'b0111: Mux_out <= R4;
            4'b1000: Mux_out <= R5;
            4'b1001: Mux_out <= {8'b0,IR};
        endcase
    end
endmodule
```

8.2.14 Top Module

```
`timescale 1ns / 1ps
module top_module(

    input clock,
    input serial_in,
    input transmit,
    output wire serial_out,
    output wire finish_out,
    output wire [7:0] ac_w

    );

    wire receive_finish,clk ;
    wire wr_uart, wr_processor, write, uart_status;
    wire [7:0] uart, processor, dram_in, pc, irom_out_p, dram_out_p, uartTX, dram_out, uart_tx;
    wire [15:0] addr_uart, addr_processor, dram_address;

    wire enable =1'b1;
    wire finish = 1'b0;

    clock_divider clock_divider1(
    .clock(clock),
    .enable(enable),
    .finish(finish),
    .clk(clk));
```

```
26
27      processor Processor(
28      .clock(clock),
29      .enable(receive_finish),
30      .irom_out(irom_out_p),
31      .dram_out(dram_out_p),
32      .finish(finish_out),
33      .dram_in(processor),
34      .address(addr_processor),
35      .write(wr_processor),
36      .pc_w(pc),
37      .acout(ac_w));
38
39      Data_Ram DRam(
40      .clock(clk),
41      .write(write),
42      .data_in(dram_in),
43      .address(dram_address),
44      .data_out(dram_out));
45
46      Instruction_Rom Irom2(
47      .clock(clk),
48      .address(pc),
49      .i_out(irom_out_p));
50
51      DRam_in_select dramin_sel(
52      .uart(uart),
53      .processor(processor),
54      .addr_uart(addr_uart),
55      .addr_processor(addr_processor),
56      .wr_uart(~receive_finish),
57      .wr_processor(wr_processor),
58      .select(uart_status),
59      .dram_in(dram_in),
60      .dram_address(dram_address),
61      .write(write));
62
63      DRam_out_select dramout_sel(
64      .DRam_out(dram_out),
65      .select(transmit),
66      .processor(dram_out_p),
67      .uart(uart_tx));
68
69      uart UART(
70      .Clock(clk),
71      .i_Serial(serial_in),
72      .tx_enable(transmit),
73      .dram_out(uart_tx),
74      .rx_finish(receive_finish),
75      .uart_active(uart_status),
76      .o_Serial(serial_out),
77      .dram_in(uart),
78      .dram_addr(addr_uart)
79  );
80  endmodule
```

8.2.15 UART Rx

Inspired by (https://nandland.com/, 2022),

```
1   `timescale 1ns / 1ps
2   module uart_rx
3       #(parameter CLKS_PER_BIT=26)
4       (
5       input i_Clock,
6       input i_Rx_Serial,
7       input i_Rx_reset,
8       output o_Rx_DV,
9       output [7:0] o_Rx_Byte
```

```
10
11      );
12
13      parameter s_IDLE = 3'b000;
14      parameter s_RX_START_BIT = 3'b001;
15      parameter s_RX_DATA_BITS = 3'b010;
16      parameter s_RX_STOP_BIT = 3'b011;
17      parameter s_CLEANUP = 3'b100;
18
19
20      reg r_Rx_Data_R = 1'b1;
21      reg r_Rx_Data = 1'b1;
22
23      reg [7:0] r_Clock_Count = 0;
24      reg [2:0] r_Bit_Index = 0;
25      reg [7:0] r_Rx_Byte = 0;
26      reg r_Rx_DV = 0;
27      reg [2:0] r_SM_Main = 0;
28      reg [7:0] a=0;
29
30      always @(posedge i_Clock)
31      begin
32          r_Rx_Data_R <= i_Rx_Serial;
33          r_Rx_Data <= r_Rx_Data_R;
34      end
35
36      always @(posedge i_Clock)
37      begin
38          case (r_SM_Main)
39              s_IDLE :
40              begin
41                  r_Rx_DV <= 1'b0;
42                  r_Clock_Count <= 0;
43                  r_Bit_Index <= 0;
44                      if (r_Rx_Data == 1'b0)
45                          r_SM_Main <= s_RX_START_BIT;
46                      else r_SM_Main <= s_IDLE;
47              end
48
49              s_RX_START_BIT :
50              begin
51                  if (r_Clock_Count == (CLKS_PER_BIT-1)/2)
52                  begin
53                      if (r_Rx_Data == 1'b0)
54                      begin
55                          r_Clock_Count <= 0;
56                          r_SM_Main <= s_RX_DATA_BITS;
57                      end
58
59                      else r_SM_Main <= s_IDLE;
60                  end
61                  else
62                  begin
63                      r_Clock_Count <= r_Clock_Count + 1;
64                      r_SM_Main <= s_RX_START_BIT;
65                  end
66              end
67
68
69              s_RX_DATA_BITS :
70              begin
71                  if (r_Clock_Count < CLKS_PER_BIT/2-1)
72                  begin
73                      r_Clock_Count <= r_Clock_Count + 1;
74                      r_SM_Main <= s_RX_DATA_BITS;
75                  end
76
77                  else if (r_Clock_Count == CLKS_PER_BIT/2)
78                  begin
79                      r_Clock_Count <= r_Clock_Count + 1;
80                      r_Rx_Byte[r_Bit_Index]<= r_Rx_Data;
81                  end
82
83                  else if (r_Clock_Count < CLKS_PER_BIT-1)
84                  begin
85                      r_Clock_Count <= r_Clock_Count + 1;
86                      r_SM_Main <= s_RX_DATA_BITS;
```

```
                end

                else
                    begin
                        r_Clock_Count <= 0;
                        r_Rx_Byte[r_Bit_Index] <= r_Rx_Data;

                        if (r_Bit_Index < 7)
                        begin
                            r_Bit_Index <= r_Bit_Index+ 1;
                            r_SM_Main <= s_RX_DATA_BITS;
                        end

                        else
                        begin
                            r_Bit_Index <= 0;
                            r_SM_Main <= s_RX_STOP_BIT;
                        end
                    end
            end

            s_RX_STOP_BIT :
            begin
                if (r_Clock_Count < CLKS_PER_BIT-1)
                begin
                    r_Clock_Count <= r_Clock_Count + 1;
                    r_SM_Main <= s_RX_STOP_BIT;
                end
                else
                begin
                    r_Rx_DV <= 1;
                    r_Clock_Count <= 0;
                    r_SM_Main <= s_CLEANUP;
                end
            end

            s_CLEANUP :
            begin
                r_SM_Main <= s_IDLE;
                r_Rx_DV <= 0;
            end

            default :
                r_SM_Main <= s_IDLE;
        endcase

        if (i_Rx_reset == 1'b1)
        begin
            r_Rx_Byte <=0;
        end
    end

    assign o_Rx_DV = r_Rx_DV;
    assign o_Rx_Byte = r_Rx_Byte;

endmodule
```

8.2.16 UART Tx

```
`timescale 1ns / 1ps
module uart_tx
    #(parameter CLKS_PER_BIT=26)
    (
    input i_Clock,
    input i_Tx_DV,
    input [7:0] i_Tx_Byte,
    output o_Tx_Active,
    output reg o_Tx_Serial,
    output o_Tx_Done
    );

    //assign o_Tx_Serial = 1;
```

```
14      parameter s_IDLE = 3'b000;
15      parameter s_TX_START_BIT = 3'b001;
16      parameter s_TX_DATA_BITS = 3'b010;
17      parameter s_TX_STOP_BIT = 3'b011;
18      parameter s_CLEANUP = 3'b100;
19
20      reg [2:0] r_SM_Main = 0;
21      reg [7:0] r_Clock_Count = 0;
22      reg [2:0] r_Bit_Index = 0;
23      reg [7:0] r_Tx_Data = 0;
24      reg r_Tx_Done = 1'b0;
25      reg r_Tx_Active = 0;
26
27      always @(posedge i_Clock)
28      begin
29          case (r_SM_Main)
30              s_IDLE :
31              begin
32                  o_Tx_Serial <= 1'b1;
33                  r_Tx_Done <= 1'b0;
34                  r_Clock_Count <= 0;
35                  r_Bit_Index <= 0;
36                  if (i_Tx_DV == 1'b1)
37                  begin
38                      r_Tx_Active <= 1'b1;
39                      r_Tx_Data <= i_Tx_Byte;
40                      r_SM_Main <= s_TX_START_BIT;
41                  end
42                  else
43                      r_SM_Main <= s_IDLE;
44              end
45
46              s_TX_START_BIT :
47              begin
48                  o_Tx_Serial <= 1'b0;
49                  if (r_Clock_Count < CLKS_PER_BIT-1)
50                  begin
51                      r_Clock_Count <= r_Clock_Count + 1;
52                      r_SM_Main <= s_TX_START_BIT;
53                  end
54
55                  else
56                  begin r_Clock_Count <= 0;
57                  r_SM_Main <= s_TX_DATA_BITS;
58                  end
59              end
60
61              s_TX_DATA_BITS :
62                  begin
63                      o_Tx_Serial <= r_Tx_Data[r_Bit_Index];
64                      if (r_Clock_Count < CLKS_PER_BIT-1)
65                      begin
66                          r_Clock_Count <= r_Clock_Count + 1;
67                          r_SM_Main <= s_TX_DATA_BITS;
68                      end
69
70                      else
71                      begin
72                          r_Clock_Count <= 0;
73                          if (r_Bit_Index < 7)
74                          begin
75                              r_Bit_Index <= r_Bit_Index +1;
76                              r_SM_Main <= s_TX_DATA_BITS;
77                          end
78
79                          else
80                          begin
81                              r_Bit_Index <= 0;
82                              r_SM_Main <= s_TX_STOP_BIT;
83                          end
84                      end
85                  end
86
87              s_TX_STOP_BIT :
88              begin
89                  o_Tx_Serial <= 1'b1;
90                  if (r_Clock_Count < CLKS_PER_BIT-1)
```

```
                begin
                    r_Clock_Count <= r_Clock_Count + 1;
                    r_SM_Main <= s_TX_STOP_BIT;
                end

                else
                begin
                    r_Tx_Done <= 1'b1;
                    r_Clock_Count <= 0;
                    r_SM_Main <= s_CLEANUP;
                    r_Tx_Active <= 1'b0;
                end
            end

            s_CLEANUP :
            begin
                r_Tx_Done <= 1'b1;
                r_SM_Main <= s_IDLE;
            end

            default :
            r_SM_Main <= s_IDLE;

        endcase

    end

    assign o_Tx_Active = r_Tx_Active;
    assign o_Tx_Done = r_Tx_Done;

endmodule
```

8.2.17 UART

```
`timescale 1ns / 1ps
module uart(
    input Clock,
    input i_Serial,
    input tx_enable,
    input [7:0] dram_out,
    output reg rx_finish,
    output reg uart_active,
    output o_Serial,
    output reg [7:0] dram_in,
    output [15:0] dram_addr
);

    parameter Rx_Active=3'b000;
    parameter Rx_Done=3'b001;
    parameter Tx_Active=3'b010;
    parameter Tx_Done=3'b011;

    wire data_valid;
    wire [7:0] data_in;
    wire [7:0] data_out;
    wire tx_done;

    reg [2:0]state=Rx_Active;
    reg [15:0] address=16'd0;
    reg tx_dv=1'b0;
    reg addr_inc=0;
    reg [7:0] add_inc=0;

    uart_rx #(.CLKS_PER_BIT(26))RX
    (.i_Clock(clock),
    .i_Rx_Serial(i_Serial),
    .i_Rx_reset(1'b0),
    .o_Rx_DV(data_valid),
    .o_Rx_Byte(data_in) );

    uart_tx #(.CLKS_PER_BIT(26))TX
    (.i_Clock(clock),
```

```
     .i_Tx_DV(tx_dv),
    .i_Tx_Byte(dram_out),
    .o_Tx_Serial(o_Serial),
    .o_Tx_Done(tx_done) );

    initial
    begin
        rx_finish=0;
        uart_active=0;
    end

    always @(posedge clock)
    begin
        case (state)
            Rx_Active:
            begin
                if (address<16'd65535 && data_valid==1)
                begin
                    dram_in<=data_in;
                    addr_inc<=1;
                end

                else if(address<16'd65535 && addr_inc==1)
                begin
                    address<=address+16'd1;
                    addr_inc<=0;
                end

                else if(address==16'd65535 && data_valid==1)
                begin
                    dram_in<=data_in;
                    state<=Rx_Done;
                    address<=16'd0;
                end
            end

            Rx_Done:
                begin
                    rx_finish<=1;
                    uart_active<=1;
                    if (tx_enable==1)
                    begin
                        state<=Tx_Active;
                        address<=16'd0;
                        uart_active<=0;
                    end
                end

            Tx_Active:
            begin
                if (address<=16'd65535 && tx_dv==1'b0)
                begin
                    tx_dv<=1'b1;
                end

                else if(address<16'd65535 && tx_done==1'b1)
                begin
                    tx_dv<=1'b0; address<=address+16'd1;
                end

                else if (address==16'd65535 && tx_done==1'b1)
                begin
                    state<=Tx_Done;
                end
            end

            Tx_Done:
                begin
                    tx_dv<=1'b0;
                    address<=16'd0;
                    rx_finish<=1'b0;
                end
        endcase
    end

    assign dram_addr=address;

```

```
endmodule
```

8.3 Test-bench codes

8.3.1 ALU_tb

```
module ALU_tb();
    reg [15:0] a_bus;
    reg [15:0] b_bus;
    reg [3:0] op_code;
     wire [15:0] c_bus;
     wire z_flag;

    ALU dut(.a_bus(a_bus), .b_bus(b_bus), .c_bus(c_bus), .op_code(op_code), .z_flag(z_flag));
    //Clock
    //always begin
    //  clk = 1; #10; clk = 0; #10;
    //end

    //Test Case 01
    initial begin
        //a_bus = 16'd6; b_bus = 16'd5; op_code = 4'b0000;
        //$display("Expected output: %d, actual output: %d",100, data_in);
        //#50;
        a_bus = 16'd6; b_bus = 16'd6; op_code = 4'b0001;
        #100;
        a_bus = 16'd6; b_bus = 16'd5; op_code = 4'b0111;
        #50;
        a_bus = 16'd6; b_bus = 16'd5; op_code = 4'b1000;
        #50;
        a_bus = 16'd6; b_bus = 16'd5; op_code = 4'b1001;
        #50;
        $finish;
    end


endmodule //alu_TB
```

8.3.2 AR_tb

```
module AR_tb();
    reg load;
    reg clk;
    reg [15:0] data_in;
    wire [15:0] data_out;

    AR dut(.clk(clk), .load(load), .data_in(data_in), .data_out(data_out));
    //Clock
    always begin
        clk = 1; #10; clk = 0; #10;
    end

    //Test Case 01
    initial begin
        load = 1; data_in = 8'd100;
        #50;
        load = 0; data_in = 8'd120;
        #50;
        load = 1;
        #50;
        $finish;
    end


```

```
endmodule //AR_TB
```

8.3.3 Clock_Divider_tb

```
`timescale 1ns / 1ps
module clock_divider_tb();
    reg enable;
    reg finish;
    reg clock;
     wire clk;

    clock_divider dut(.clock(clock), .enable(enable), .finish(finish), .clk(clk));
    //Clock
    always begin
        clock = 1; #10; clock = 0; #10;
    end

    //Test Case 01
    initial begin
        enable = 1; finish = 0;
        #1100;
        enable = 1; finish = 1;
        #1100;
        enable = 0; finish = 1;
        #1100;
        //$finish;
    end


endmodule //clock_divider_TB
```

8.3.4 Control_Unit_tb

```
module Control_Unit_tb();
    reg i_clock;
    reg [7:0] instruction;
    reg z;
     wire Mem_write;
     wire IR_enable;
    wire finish;
    wire [3:0] read_enable ;
    wire [8:0] write_enable;
    wire [3:0] alu_op;
    wire [7:0] inc_enable;

    Control_Unit dut(.i_clock(i_clock), .instruction(instruction), .z(z), .Mem_write(Mem_write),
        .IR_enable(IR_enable), .finish(finish), .read_enable(read_enable),
        .write_enable(write_enable), .alu_op(alu_op), .inc_enable(inc_enable));
    //Clock
    always begin
        i_clock = 1; #10; i_clock = 0; #10;
    end

    //Test Case 01
    initial begin
        instruction = 8'd21; z = 0;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #100;
        instruction = 8'd20; z = 0;
        #100;
        instruction = 8'd25; z = 1;
        #100;
        $finish;
    end


```

```
endmodule //Control_Unit_TB
```

8.3.5 DRAM_in_select_tb

```
module DRam_in_select_tb();

    reg [7:0] uart;
    reg [7:0] processor;
    reg [15:0] addr_uart;
    reg [15:0] addr_processor;
    reg wr_uart;
    reg wr_processor;
    reg select;
    wire [7:0] dram_in;
    wire [15:0] dram_address;
    wire write;

    DRam_in_select dut(.uart(uart), .processor(processor), .addr_uart(addr_uart),
        .addr_processor(addr_processor), .wr_uart(wr_uart), .wr_processor(wr_processor),
        .select(select), .dram_in(dram_in), .dram_address(dram_address), .write(write));
    //Clock
    //always begin
      //  clk = 1; #10; clk = 0; #10;
    //end

    //Test Case 01
    initial begin
        uart = 8'd120; select = 1; processor = 8'd100; addr_uart = 16'd110; addr_processor =
            16'd130; wr_uart= 0; wr_processor = 1;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        uart = 8'd120; select = 0; processor = 8'd100; addr_uart = 16'd110; addr_processor =
            16'd130; wr_uart= 0; wr_processor = 1;
        #50;
        $finish;
    end


endmodule //DRam_out_TB
```

8.3.6 DRAM_out_select_tb

```
module DRam_out_tb();
    reg [7:0] DRam_out;
    reg select;
    wire [7:0] processor;
    wire [7:0] uart;

    DRam_out_select dut(.DRam_out(DRam_out), .select(select), .processor(processor), .uart(uart));
    //Clock
    //always begin
      //  clk = 1; #10; clk = 0; #10;
    //end

    //Test Case 01
    initial begin
        DRam_out = 8'd120; select = 1;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        select = 0; DRam_out = 8'd100;
        #50;
        $finish;
    end


endmodule //DRam_out_TB
```

8.3.7 DRAM_tb

```
`timescale 1ns / 1ps
module Data_Ram_tb();
    reg clock;
    reg write;
    reg [15:0] address;
     reg [7:0] data_in;
     wire [7:0] data_out;

    Data_Ram dut(.clock(clock), .write(write), .data_in(data_in), .address(address),
        .data_out(data_out));
    //Clock
    always begin
        clock = 1; #10; clock = 0; #10;
    end

    //Test Case 01
    initial begin
        write = 1; address = 16'd120; data_in = 8'd100;
        #50;
        write = 0; address = 16'd120; data_in = 8'd110;
        #50;
        write = 1;
        #50;
        write = 0;
        #50;
        $finish;
    end


endmodule //Data_Ram_TB
```

8.3.8 Instruction_Rom_tb

```
module Instruction_Rom_tb();
    reg clock;
    reg [7:0] address;
    wire [7:0] i_out;

    Instruction_Rom dut(.clock(clock), .address(address), .i_out(i_out));
    //Clock
    always begin
        clock = 1; #10; clock = 0; #10;
    end

    //Test Case 01
    initial begin
        address = 8'd100;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        address = 8'd70;
        #50;
        address = 8'd110;
        #50;
        $finish;
    end


endmodule //Instruction_Rom_TB
```

8.3.9 IR_tb

```
module IR_tb();
    reg fetch;
```

```
    reg clk;
    reg [7:0] i_in;
    wire [7:0] i_out;

    IR dut(.clk(clk), .fetch(fetch), .i_in(i_in), .i_out(i_out));
    //Clock
    always begin
        clk = 1; #10; clk = 0; #10;
    end

    //Test Case 01
    initial begin
        fetch = 0; i_in = 8'd100;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        fetch = 1;
        #50;
        $finish;
    end


endmodule //IR_TB
```

8.3.10 PC_tb

```
module PC_tb();
    reg load;
    reg inc;
    reg clk;
    reg [7:0] data_in;
    wire [7:0] data_out;

    PC dut(.clk(clk), .load(load), .inc(inc), .data_in(data_in), .data_out(data_out));
    //Clock
    always begin
        clk = 1; #10; clk = 0; #10;
    end

    //Test Case 01
    initial begin
        load = 1; inc = 1; data_in = 8'd100;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        load = 1; inc = 0; data_in = 8'd120;
        #50;
        load = 0; inc = 0;
        #50;
        load = 0; inc = 1;
        #50;
        load = 1; inc = 1;
        #50;
        $finish;
    end


endmodule //PC_TB
```

8.3.11 Processor_tb

```
module processor_tb();
    reg clock;
    reg enable;
    reg [7:0] dram_out;
    reg [7:0] irom_out;
    wire [15:0] address;
    wire [7:0] dram_in;
```

```
    wire finish;
    wire write;
    wire [7:0] acout;
    wire [7:0] pc_w;

    processor dut(.clock(clock), .enable(enable), .dram_out(dram_out), .irom_out(irom_out),
         .address(address), .dram_in(dram_in), .finish(finish), .write(write), .acout(acout),
         .pc_w(pc_w));
    //Clock
    always begin
        clock = 1; #10; clock = 0; #10;
    end

    //Test Case 01
    initial begin
        dram_out = 8'd21; enable = 1; irom_out = 8'd25;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        dram_out = 8'd21; enable = 0; irom_out = 8'd25;
        #50;
        dram_out = 8'd21; enable = 1;
        #50;
        $finish;
    end


endmodule //processor_TB
```

8.3.12 16_bit_register_tb

```
`timescale 1ns / 1ps
module reg_16bit_tb();
    reg load;
    reg inc;
    reg clk;
    reg [15:0] data_in;
    wire [15:0] data_out;

    reg_16bit dut(.clk(clk), .load(load), .inc(inc), .data_in(data_in), .data_out(data_out));
    //Clock
    always begin
        clk = 1; #10; clk = 0; #10;
    end

    //Test Case 01
    initial begin
        load = 0; inc = 0; data_in = 8'd100;
//        $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        load = 1; inc = 0;
        #50;
        load = 1; inc = 1;
        #50;
        load = 0; inc = 1;
        #50;
        load = 1; inc = 0;
        #50;
        $finish;
    end


endmodule //reg_16_bit_TB
```

8.3.13 Register_Mux_tb

```
module Register_MUX_tb();
```

```
    reg [7:0] M;
    reg [7:0] PC;
    reg [7:0] IR;
    reg [15:0] R;
    reg [15:0] R1;
    reg [15:0] R2;
    reg [15:0] R3;
    reg [15:0] R4;
    reg [15:0] R5;
    reg [15:0] AC;
    reg [3:0] B_ctrl;
    wire [15:0] Mux_out;

    Register_MUX dut(.M(M), .PC(PC), .IR(IR), .R(R), .R1(R1), .R2(R2), .R3(R3), .R4(R4), .R5(R5),
        .AC(AC), .B_ctrl(B_ctrl), .Mux_out(Mux_out));
    //Clock
    //always begin
      //  clk = 1; #10; clk = 0; #10;
    //end

    //Test Case 01
    initial begin
        M = 0; PC = 0; IR = 0; R = 0; R1 = 0;R2 =0 ;R3 = 0;R4 = 0;R5 = 0;AC = 16'd100;B_ctrl =
            4'b0001;// $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        M = 0; PC = 0; IR = 8'd110; R = 0; R1 = 0;R2 =0 ;R3 = 0;R4 = 16'b0111;R5 = 0;AC = 0;B_ctrl
            = 4'b1001;// $display("Expected output: %d, actual output: %d",100, data_in);
        #50;
        $finish;
    end


endmodule //Register_MUX_TB
```

8.3.14 Top_Module_tb

```
module top_module_tb();
    reg clock;
    reg serial_in;
    reg transmit;
    wire serial_out;
    wire finish_out;
    wire [7:0] ac_w;


    top_module dut(.clock(clock), .serial_in(serial_in), .transmit(transmit),
        .serial_out(serial_out), .finish_out(finish_out), .ac_w(ac_w));
    //Clock
    always begin
        clock = 1; #10; clock = 0; #10;
    end

    //Test Cases - Experimental
    initial begin
        serial_in = 0; transmit = 1;
//       $display("Expected output: %d, actual output: %d",100, data_in);
        #500;
        serial_in = 1; transmit = 1;
        #500;
        serial_in = 0; transmit = 0;
        #50;
        $finish;
    end


endmodule //top_module_TB
```

8.3.15 UART_tb

```
`timescale 1ns/10ps

//`include "uart_tx.v"
//`include "uart_rx.v"

module uart_tb ();

  // Testbench uses a 10 MHz clock
  // Want to interface to 115200 baud UART
  // 10000000 / 115200 = 87 Clocks Per Bit.
  parameter c_CLOCK_PERIOD_NS = 100;
  parameter c_CLKS_PER_BIT = 87;
  parameter c_BIT_PERIOD   = 8600;

  reg r_Clock = 0;
  reg r_Tx_DV = 0;
  wire w_Tx_Done;
  reg [7:0] r_Tx_Byte = 0;
  reg r_Rx_Serial = 1;
  wire [7:0] w_Rx_Byte;


  // Takes in input byte and serializes it
  task UART_WRITE_BYTE;
    input [7:0] i_Data;
    integer    ii;
    begin

      // Send Start Bit
      r_Rx_Serial <= 1'b0;
      #(c_BIT_PERIOD);
      #1000;


      // Send Data Byte
      for (ii=0; ii<8; ii=ii+1)
        begin
          r_Rx_Serial <= i_Data[ii];
          #(c_BIT_PERIOD);
        end

      // Send Stop Bit
      r_Rx_Serial <= 1'b1;
      #(c_BIT_PERIOD);
     end
  endtask // UART_WRITE_BYTE


  uart_rx #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_RX_INST
    (.i_Clock(r_Clock),
     .i_Rx_Serial(r_Rx_Serial),
     .o_Rx_DV(),
     .o_Rx_Byte(w_Rx_Byte)
     );

  uart_tx #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_TX_INST
    (.i_Clock(r_Clock),
     .i_Tx_DV(r_Tx_DV),
     .i_Tx_Byte(r_Tx_Byte),
     .o_Tx_Active(),
     .o_Tx_Serial(),
     .o_Tx_Done(w_Tx_Done)
     );


  always
    #(c_CLOCK_PERIOD_NS/2) r_Clock <= !r_Clock;


  // Main Testing:
  initial
    begin

```

```
74        // Tell UART to send a command (exercise Tx)
75        @(posedge r_Clock);
76        @(posedge r_Clock);
77        r_Tx_DV <= 1'b1;
78        r_Tx_Byte <= 8'hAB;
79        @(posedge r_Clock);
80        r_Tx_DV <= 1'b0;
81        @(posedge w_Tx_Done);
82
83        // Send a command to the UART (exercise Rx)
84        @(posedge r_Clock);
85        UART_WRITE_BYTE(8'h3F);
86        @(posedge r_Clock);
87
88        // Check that the correct command was received
89        if (w_Rx_Byte == 8'h3F)
90          $display("Test Passed - Correct Byte Received");
91        else
92          $display("Test Failed - Incorrect Byte Received");
93
94      end
95
96  endmodule
```