FPGA Based Pipelined Processor Design

A.K.H.S Samarasinghe	160546J
K.A.D.G Sameera	160549V
S.J Wanigasekara	160662K
H.O Weerasinghe	160674A

EN3030 – Circuit and System Design
Department of Electronic and Telecommunication Engineering
University of Moratuwa

Contents

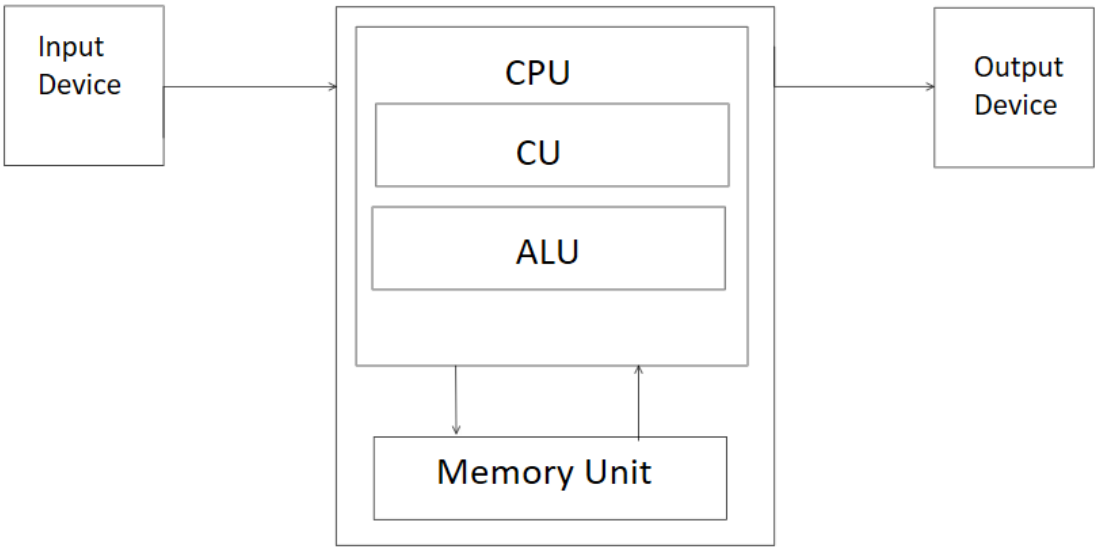
Introduction **3**
Microprocessor and Central Processing Unit 3
Problem Statement 4
Overview of the Solution 5
Instruction Set Architecture **7**
Note on Registers 7
Overview of Instructions 7
Compiler 11
Module Design **15**
Register Design 15
Program Counter 17
Arithmetic and Logic Unit (ALU) 17
The Multiplexer 19
Clock Divider 21
Serial communication from PC to FPGA 21
Algorithm **24**
Filtering Algorithm 24
Down Sampling Algorithm 25
Result Analysis **26**
Appendix **31**
ALU 31
DEMUX 32
Clock Divider 34
Instruction Memory 34
MUX 36
Program Counter 38
Register 39
SRAM 41
Processor 42
State Machine 47
Receiver 67
Transmitter 68
UART Clock 70
UART Controller 70
Compiler 73

Introduction

This projects main aim is to design a custom processor to complete a certain task, with the Verilog hardware description language on a Field Programmable Gate Array (FPGA) board. This document explains and describes the microprocessor and the design of the CPU , the coding of the microprocessor and testing process and the physical implementation of the project using FPGA.

Microprocessor and Central Processing Unit

CPU is the electronic circuit which accomplishes the instructions of certain computer programs by conducting arithmetic and logic operations , control and I/O operations which were specified by the instruction set. The CPU mainly consists of the processor and the control unit.



A microprocessor is multipurpose , clock driven and register based programmable device which implements the functions of a CPU on a single integrated circuit. It uses binary data as the input and processes them and provides processed results as the output. They normally contain both combinational and sequential logic.

In the early times only computers contained microprocessors and they very big in size. But as the technology developed they started to appear on other devices too. As for this time there is some kind of a microprocessor in every electronic device. From little IOT devices to large automobiles there are various variants of microprocessors.

So we can see that microprocessors has been a very essential element in today's electronic industry.

Problem Statement

Main objective of this processor is to filter and down sample a given image. The CPU will get the data about the pixels of the image from a computer through UART interface and save them in the main memory and process them. Then it will return the processed data to the computer to display the processed image.

There are several steps of this image down-sampling process

1) Get the needed data from the computer (MATLAB) and store it in the main memory

   A UART module is implemented inside the processor to receive the data of the image. Before that the image is converted to binary with MATLAB. Then it is sent to the FPGA's UART through the serial interface.

2) Filtering the Image

   Filtering is done before the down sampling of the image to reduce the high frequency components of the image because they can affect the final result of the image after down sampling. A gaussian filter is implemented on the FPGA to achieve this task

3) Down sampling the image.

   After the filtering process image is run through the down sampling process. As the requirement specifies to down sample it to half the size the assembly code is written for that purpose. After processing , the data is stored in the main memory.

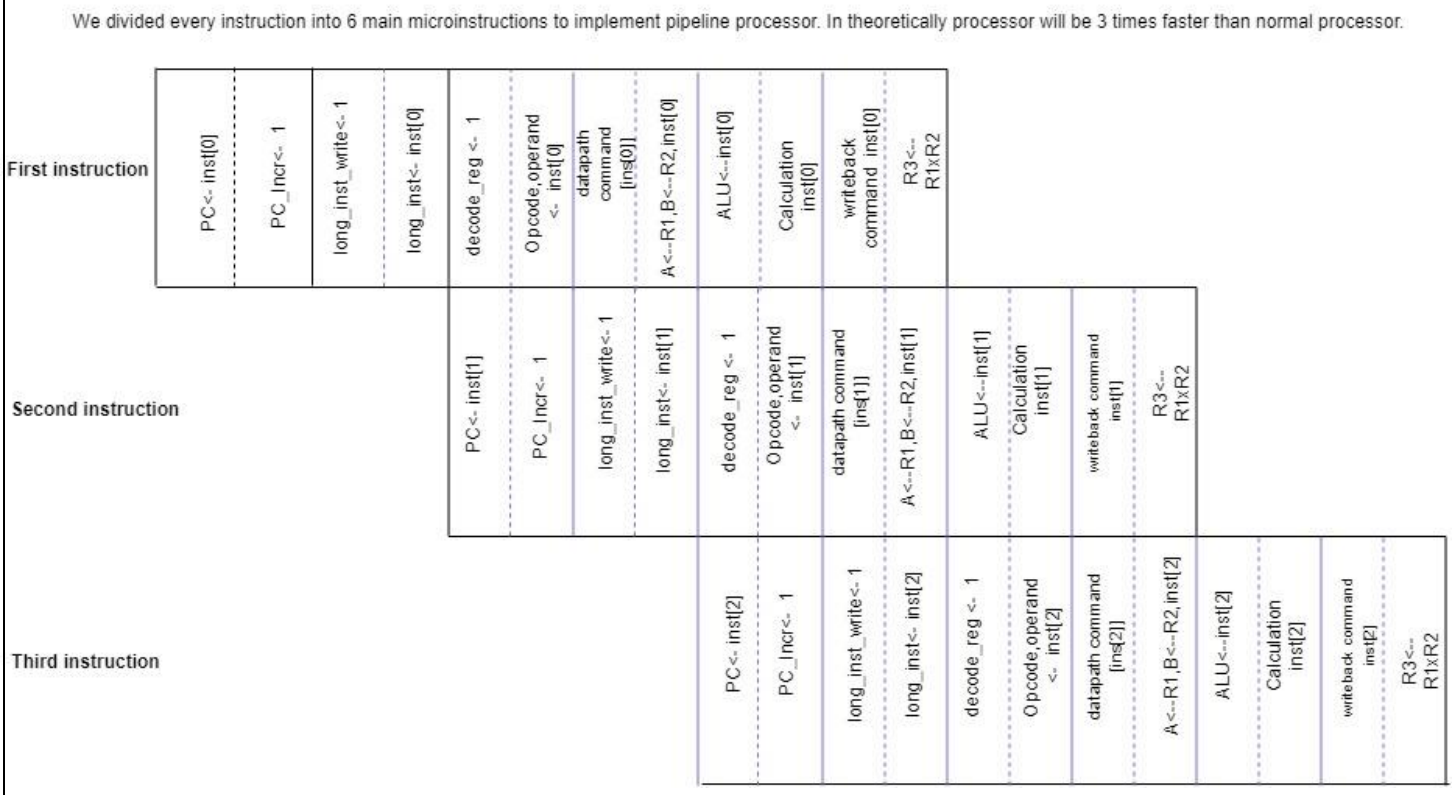
4) Output the Image to Computer

   After finishing the down sampling , image is transferred back to the computer. For this purpose UART Transmitter module is implemented inside the UART parent module. After transmitting the data back to MATLAB , a code is written in MATLAB to grab the data and display the final result.

Overview of the Solution

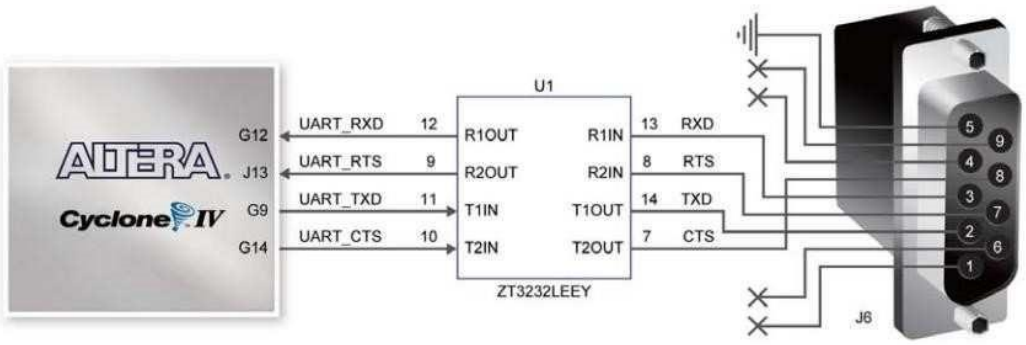
In implementing the processor, we have used pipelining. Each instruction is divided into 6 stages and each stage complete part of the instruction in parallel.

Instruction Execution with time



Alterra DE-2 FPGA board with Alterra Cyclone IV was used to implement the processor on the hardware. The UART module was implemented on the FPGA and it used the inbuilt serial port/RS232 to communicate with the computer.

SRAM of 2Mb was implemented on the FPGA to store in input/processed data. One RAM



module was implemented to store the image data and one ROM module was implemented to store the instructions (Instruction Memory).

All other modules were implemented by using Verilog HDL and included in the processor Top Module.

A USB to Serial cable was used to connect the computer to the FPGA device.

Instruction Set Architecture

Note on Registers

All instructions are stored on IRAM (18 bit data and address bus).We have used 16 bit Instruction Register(IR). IR has 1024 addresses, therefore PC is 10 bit (ISA needs to be changed to store instructions on the DRAM and access from there).

Registers available

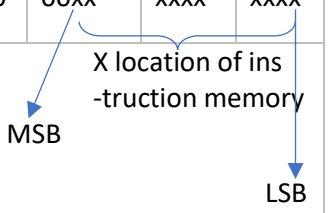
1) MAR:	18 bit
2) MDR:	16 bit
3) R1,R2,R3,R4,R5,R6,R7,R8(general purpose registers):	16 bit
4) PC	10 bit
5)R_I 1,R_I 2,R_I 3(general purpose registers, can increment by 1 directly)	18 bit
6)AC	18bit
7)Intermediate Registers for pipelining structure	varying

Overview of Instructions

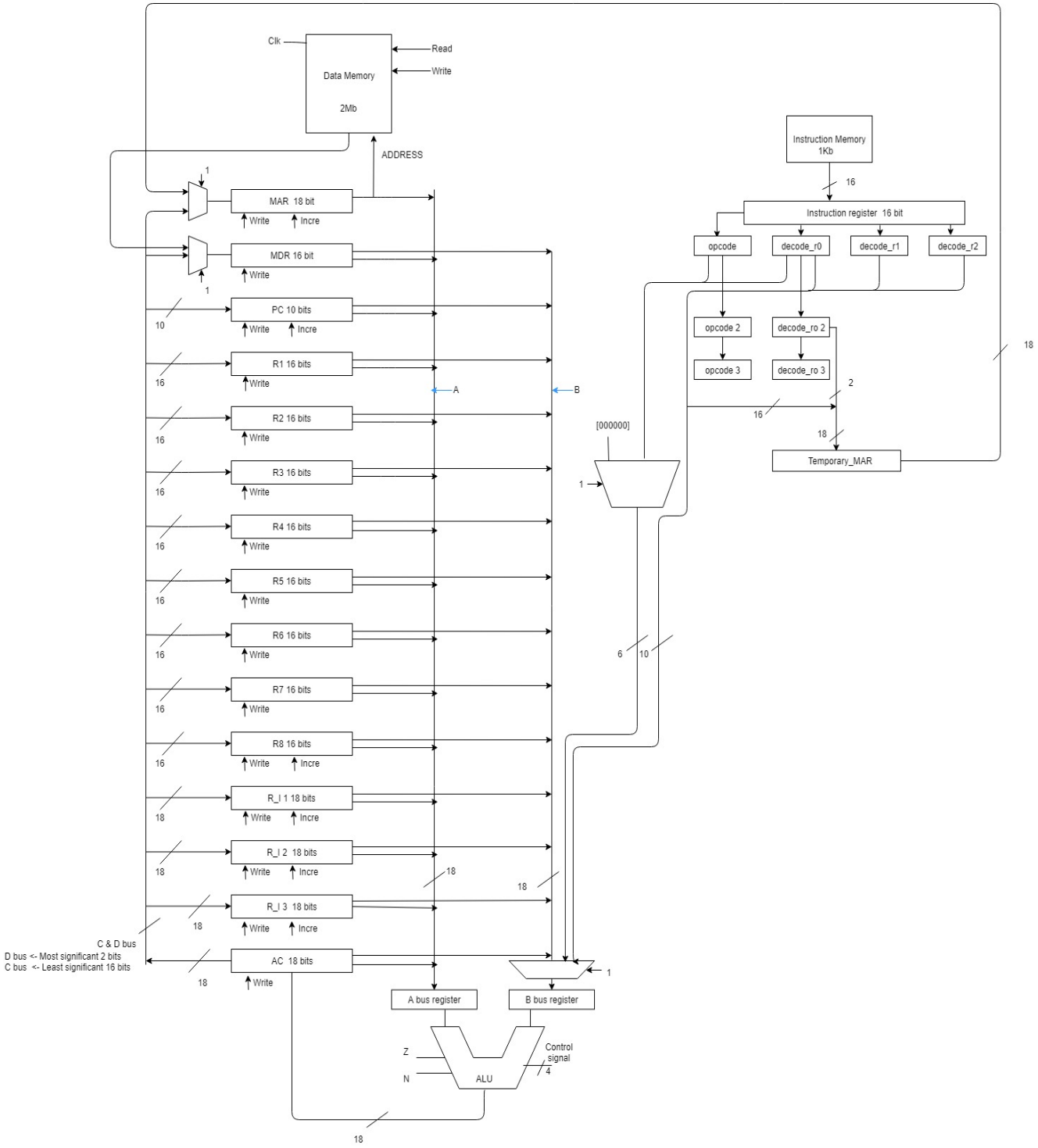
	OPCODE			
NOP No Operation	0000	0000	0000	0000
1)ADD a ← b + c	0001	Reg xxxx Destin-ation	Reg xxxx 1st No	Reg xxxx 2nd No
2)SUB a ← b - c	0010	Reg xxxx Destin-ation	Reg xxxx 1st No	Reg xxxx 2nd No
3)XOR a ← b ^ c	0011	Reg xxxx Destin-ation	Reg xxxx 1st No	Reg xxxx 2nd No
4)MUL a ← b * c	0100	Reg xxxx Destin-ation	Reg xxxx 1st No	Reg xxxx 2nd No

5)DIV a ← b / c	0101	Reg xxxx Destin-ation	Reg xxxx 1st No	Reg xxxx 2nd No
8)SUBI n a ← b – n	1000	Reg xxxx Destin-ation	Reg xxxx 1st No	0000
	xxxx xxxx xxxx xxxx number 'n' (16 bit number)			
9)ADDI n a ← b + n	1001	Reg xxxx Destin-ation	Reg xxxx 1st No	0000
	xxxx xxxx xxxx xxxx number 'n' (16 bit number)			
10)WRITE MEMORY[MAR] ← MDR Memory Address Register(18 bit) Data Register	1010	0000	0000	0000
11)READ MDR ← MEMORY[MAR]	1011	0000	0000	0000
13)MEMORY ADDRESS WRITE MAR ← α[18 bit]	1101	xx00	0000	0000
		α[17:16]		
15)CLAC Clear all register data	1111	0000	0000	0000
STOP Stop the processor	1111	1111	0000	0000
INCR	1100	00xx	0000	0000

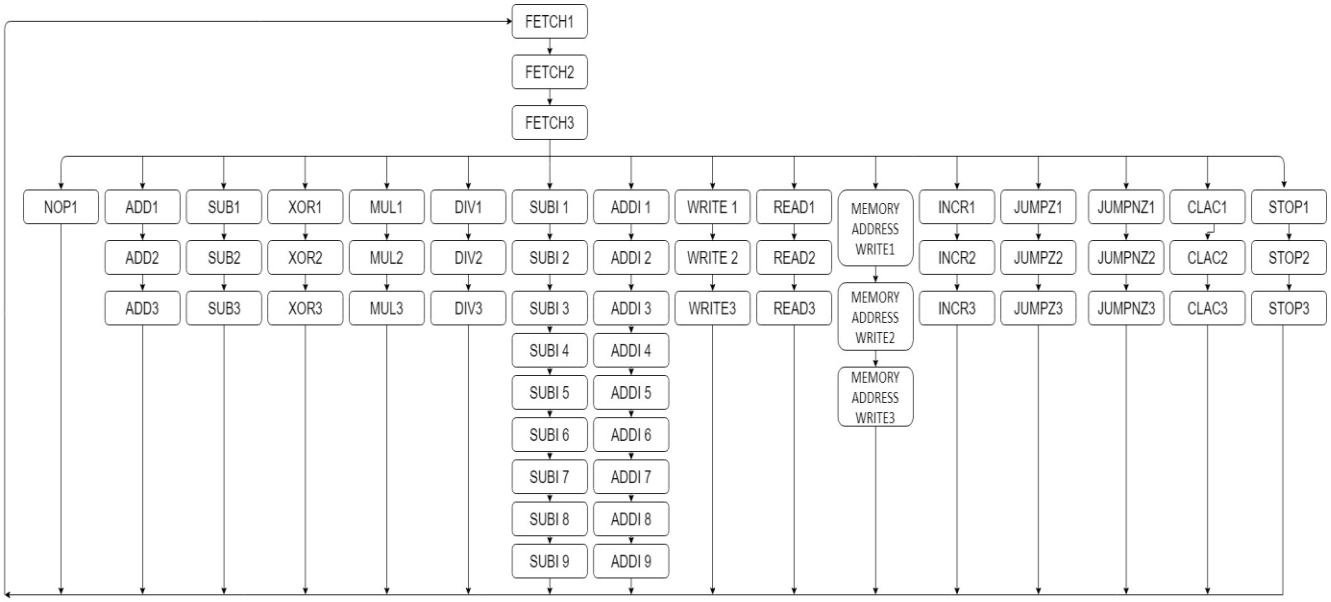
R_I 1 ◄— R_I 1+1 R_I 2 ◄— R_I 2+1 R_I 3 ◄— R_I 3+1 MAR ◄— MAR+1	1100 1100 1100 1100	0001 0010 0011 0000	0000 0000 0000 0000	0000 0000 0000 0000
JUMPZ Go to x Goto ‘x’ if AC=’0000’	0110	00xx	xxxx	xxxx
JUMPNZ Goto ‘x’ if AC !=’0000’	0111	00xx	xxxx	xxxx



Data Path


Clk
Data Memory
2Mb
Read
Write
ADDRESS
Instruction Memory
1Kb
16
Instruction register 16 bit
opcode
decode_r0
decode_r1
decode_r2
opcode 2
decode_ro 2
opcode 3
decode_ro 3
18
2
16
18
Temporary_MAR
[000000]
1
MAR 18 bit
Write
Incre
MDR 16 bit
PC 10 bits
R1 16 bits
R2 16 bits
R3 16 bits
R4 16 bits
R5 16 bits
R6 16 bits
R7 16 bits
R8 16 bits
R_I 1 18 bits
R_I 2 18 bits
R_I 3 18 bits
AC 18 bits
A
B
6
10
C & D bus
D bus <- Most significant 2 bits
C bus <- Least significant 16 bits
A bus register
B bus register
Z
N
ALU
Control
signal
4

State machine



Compiler

In order to remove pipeline hazards, we need to add NOP (no operation) instructions between crashing instructions. Therefore we have implemented a compiler using python in order to eliminate hazards in the assembly code given by the user and convert it to machine code.

Refer to the appendix for python code for the compiler.

Following is the machine code generated by the compiler.

mem[0]= 16'b 0000000000000000 ;

mem[1]= 16'b 1111000000000000 ;

mem[2]= 16'b 0000000000000000 ;

mem[3]= 16'b 1101000000000000 ;

mem[4]= 16'b 0000000000000000 ;

mem[5]= 16'b 1001100110010000 ;

mem[6]= 16'b 0000000001111110 ;

```
mem[ 7 ]= 16'b 1001101110110000 ;
mem[ 8 ]= 16'b 1111111000000000 ;
mem[ 9 ]= 16'b 1001010101010000 ;
mem[ 10 ]= 16'b 0000000011111101 ;
mem[ 11 ]= 16'b 1001011101110000 ;
mem[ 12 ]= 16'b 0000000000000010 ;
mem[ 13 ]= 16'b 1001100010000000 ;
mem[ 14 ]= 16'b 0000000000010000 ;
mem[ 15 ]= 16'b 1001110011000000 ;
mem[ 16 ]= 16'b 1111111000000001 ;
mem[ 17 ]= 16'b 1011000000000000 ;
mem[ 18 ]= 16'b 0000000000000000 ;
mem[ 19 ]= 16'b 1110001100010000 ;
mem[ 20 ]= 16'b 1100000000000000 ;
mem[ 21 ]= 16'b 1011000000000000 ;
mem[ 22 ]= 16'b 0000000000000000 ;
mem[ 23 ]= 16'b 1110010000010000 ;
mem[ 24 ]= 16'b 1100000000000000 ;
mem[ 25 ]= 16'b 1011000000000000 ;
mem[ 26 ]= 16'b 0000000000000000 ;
mem[ 27 ]= 16'b 0001001100110001 ;
mem[ 28 ]= 16'b 0001000000000101 ;
mem[ 29 ]= 16'b 1011000000000000 ;
mem[ 30 ]= 16'b 0000000000000000 ;
mem[ 31 ]= 16'b 0001010001000001 ;
mem[ 32 ]= 16'b 1100000000000000 ;
mem[ 33 ]= 16'b 1011000000000000 ;
mem[ 34 ]= 16'b 0000000000000000 ;
mem[ 35 ]= 16'b 1110011000010000 ;
mem[ 36 ]= 16'b 1100000000000000 ;
mem[ 37 ]= 16'b 1011000000000000 ;
```

```
mem[ 38 ]= 16'b 0000000000000000 ;
mem[ 39 ]= 16'b 0001010001000001 ;
mem[ 40 ]= 16'b 0001000000000101 ;
mem[ 41 ]= 16'b 0100011001100111 ;
mem[ 42 ]= 16'b 1011000000000000 ;
mem[ 43 ]= 16'b 0100011001100111 ;
mem[ 44 ]= 16'b 0001001100110001 ;
mem[ 45 ]= 16'b 1100000000000000 ;
mem[ 46 ]= 16'b 1011000000000000 ;
mem[ 47 ]= 16'b 0000000000000000 ;
mem[ 48 ]= 16'b 0001010001000001 ;
mem[ 49 ]= 16'b 1100000000000000 ;
mem[ 50 ]= 16'b 1011000000000000 ;
mem[ 51 ]= 16'b 0000000000000000 ;
mem[ 52 ]= 16'b 0001001100110001 ;
mem[ 53 ]= 16'b 0100010001000111 ;
mem[ 54 ]= 16'b 0001001100110110 ;
mem[ 55 ]= 16'b 0000000000000000 ;
mem[ 56 ]= 16'b 0001001100110100 ;
mem[ 57 ]= 16'b 1110000011000000 ;
mem[ 58 ]= 16'b 0101001100111000 ;
mem[ 59 ]= 16'b 0000000000000000 ;
mem[ 60 ]= 16'b 1110000100110000 ;
mem[ 61 ]= 16'b 1010000000000000 ;
mem[ 62 ]= 16'b 1100001000000000 ;
mem[ 63 ]= 16'b 0011111010101001 ;
mem[ 64 ]= 16'b 0110000001001000 ;
mem[ 65 ]= 16'b 0000000000000000 ;
mem[ 66 ]= 16'b 1001101010100000 ;
mem[ 67 ]= 16'b 0000000000000001 ;
mem[ 68 ]= 16'b 0001110111010111 ;
```

```
mem[ 69 ]= 16'b 0000000000000000 ;
mem[ 70 ]= 16'b 1110000011010000 ;
mem[ 71 ]= 16'b 0010111011101110 ;
mem[ 72 ]= 16'b 0110000000010001 ;
mem[ 73 ]= 16'b 0000000000000000 ;
mem[ 74 ]= 16'b 1001110111010000 ;
mem[ 75 ]= 16'b 0000000100000010 ;
mem[ 76 ]= 16'b 0010101010101010 ;
mem[ 77 ]= 16'b 1110000011010000 ;
mem[ 78 ]= 16'b 0011111010111101 ;
mem[ 79 ]= 16'b 0111000000010001 ;
mem[ 80 ]= 16'b 0000000000000000 ;
mem[ 81 ]= 16'b 0000000000000000 ;
mem[ 82 ]= 16'b 0000000000000000 ;
mem[ 83 ]= 16'b 1111111100000000 ;
>>>
```

Module Design

Register Design

Custom processor for down sampling an image consists of two main categories of registers. In functionality wise, they are incremented and non-incremented registers. The non-incremented registers should be manipulated by the programmer to be incremented by using the proper instructions by sending through the ALU. The incremented registers listen for the increment bit to be high and in the positive edge of the next clock cycle it will increment the contents of the register. Interconnection of the registers can be identified by using the data path.

Most significant aspect is that the C-in and C_out are both 16 bits long (Excluding Instruction Register (IR) as it is only 8 bits long) since the data busses arriving and departing from the registers are all 16bits. This simplifies the registers as the low order and high order selection is not required at the moment. Alternate design is to have low order and high order registers in two clusters with additional input bits corresponding to set_High and set_low. This in some instances is an added bonus as it doubles the utilizable registers, however, at the moment, is not required for the purpose of down-sampling. Instruction Register is the only exclusion as it is only 8 bits.

Register resetting is also not utilized, and the same results can be obtained by the programmer with the aid of the ALU. The load bit determines whether to write the content of C_in into the register or not and the C_out is always enabled as it is gated in next stages to avoid complexity.

Refer the Appendix for the verilog codes for the registers.

Block Diagram for the non incremented register.

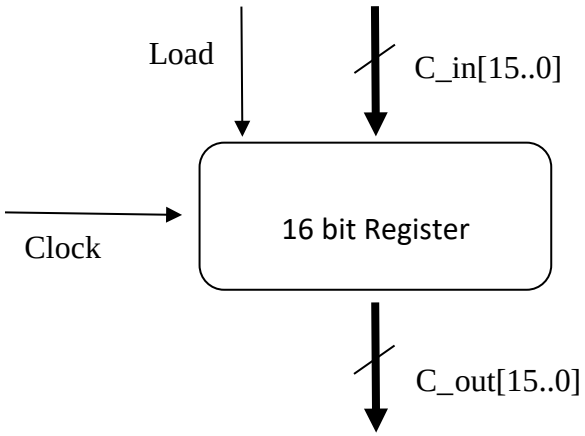


Figure 3.1

Block Diagram for the non incremented registers

Examples for the non incremented registers of the processor are Register 01 (R1), R2,…, R8, and the accumulator. (AC).

Block diagram for the incremented register

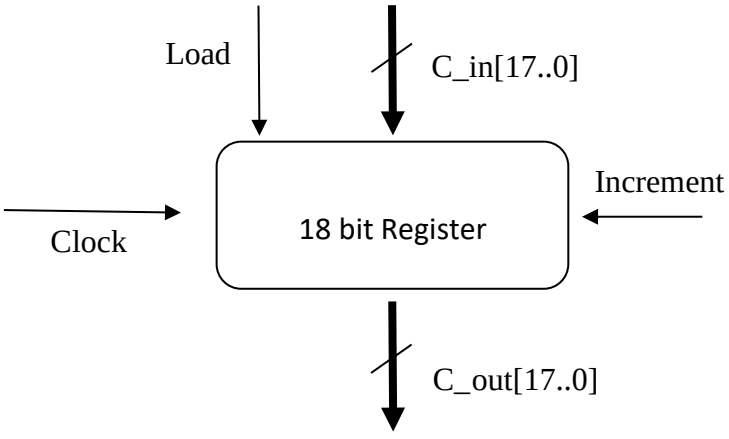


Figure 3.2

Block Diagram for the incremented registers

Examples for the non incremented registers are, Memory Address Register (AR), Register (R) and the Program counter (PC). The instruction register (IR) requires only 8 bits but it also is an incremented register.

Program Counter

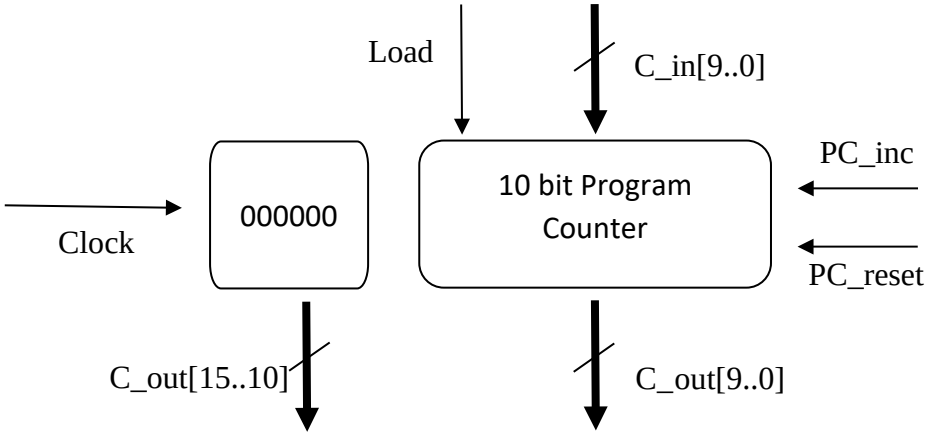


Figure 3.3

Block Diagram for the program counter

The program counter is used to hold the address of the next instruction . The data coming in to the program counter comes similarly from C_in which is the B-bus and the outputs are sent to the iram as well as to the multiplexer for usage in next clock cycles. The functionality and the testing of the program counter is same as the incremented register as mentioned above.

Arithmetic and Logic Unit (ALU)

A basic architecture of the Arithmetic and Logical Unit is used to be compatible with the custom processor requirement and it allows four basic operations. Input to the ALU are given by two 18 bit wide busses namely A and B buses. The register AC (accumulator) is directly connected to the output of the ALU which is also a 18 bit wide bus (C_Bus) and the output of the accumulator is connected to the A_Bus mentioned above. The R register acts as an operand provider for the unit while the other operand is naturally acquired from the AC register.

The inputs are always evaluated and the output bus is always activated, however the validity of the output is guaranteed on the next clock cycle. AC loading is required as mentioned in the non incrementing registers to properly maintain the accuracy. However the AC output has no need to be gated as the new value is loaded in the next clock cycle only.

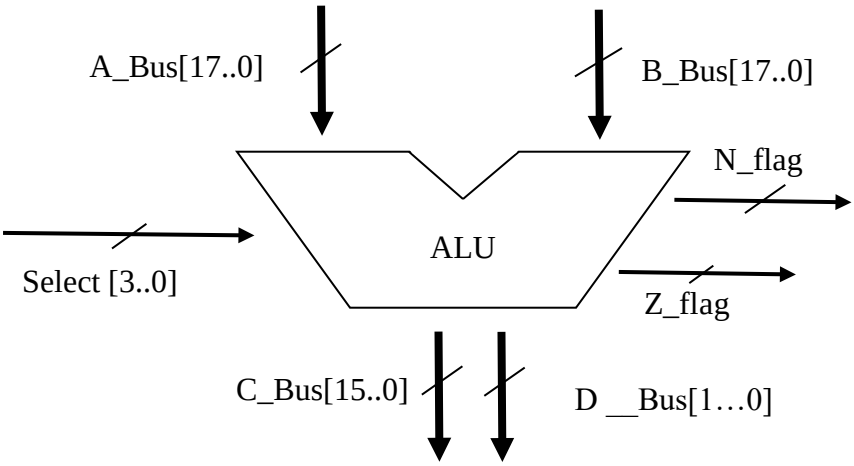


Figure 3.4

Block Diagram for the ALU

The operations that are executed through the ALU can be identified as follows,

ALU operation	Explanation	Operator [3..0]
ADD	Addition of the values in 2 registers that comes through the A_bus and B_Bus. The result is then stored in AC register.	0001
SUB	Subtraction of the values in 2 registers that comes through the A_bus and B_Bus. The result is then stored in AC register.	0010
XOR	Bitwise XOR of the values in 2 registers that comes through the A_bus and B_Bus. The result is then stored in AC register.	0011
MUL	Multiplication of the values in 2 registers that comes through the A_bus and B_Bus. The result is then stored in AC register. (In here we implemented only bit shifting operation. Therefore I can only multiply by power of 2)	0100

DIV	Division of the values in 2 registers that comes through the A_bus and B_Bus. The result is then stored in AC register. (In here we implemented only bit shifting operation. Therefore I can only multiply by power of 2)	0101
ADDI	Addition of the value in a registers that comes through the A_bus and user given number which is coming from B_Bus. The result is then stored in AC register.	1001
SUBI	Subtraction of the value in a registers that comes through the A_bus and user given number which is coming from B_Bus. The result is then stored in AC register.	1000

Table 3.1

ALU operations and their explanation

Whenever the result of the ALU is zero, the output "z" bit is set high and otherwise is kept low.

The value of the output "z" bit or also known as Z flag, is important in order to execute some ISA instructions like the jumping of the instructions when the ALU output is zero. Z flag is important in the next clock cycle only. Therefore, tracking the value of A_bus is the usual practice as it is kept unchanged for the next clock cycle while carrying the output of the current clock cycle.

Testing of the ALU

Testing of the ALU is obvious as our custom processor executes only four mathematical functions. For the tests to be valid, the output should not be greater than any 16 bit value but as the multiplication of A_Bus and B_Bus is not performed, this condition is satisfied most of the time. No additional flags for validation are used. Allowing two 18 bit values through A_Bus and B_Bus and selecting the proper operation as mentioned in the table immediately calculates the output and through C_Bus and therefore, monitoring the C_Bus will validate the functionality of the ALU.

The Multiplexer

Register loading is performed at the module level for each and every register but the output is always enabled. In order to properly latch the data to the B_Bus, the registers connect to the multiplexer and the multiplexer will deliver the proper register's data to the B_Bus

according to the MUX selection bits. There are eight input paths and those will be padded with leading zeros for the two paths that deliver effective 8 bit data into the multiplexer.

In an actual processor, additional output selectors at the end of each register module is required to avoid unnecessary output paths which are 16 bit wide, however for the application, optimality is not required.

Following table describes the MUX_selection bits and the path activated from the particular input.

MUX_sel	A_Bus
0000	MAR16
0001	MDR_out
0010	PC_extend
0011	R1_out
0100	R2_out
0101	R3_out
0110	R4_out
0111	R5_out
1000	R6_out
1001	R7_out
1010	R8_out
1011	R_I_1_out
1100	R_I_2_out
1101	R_I_3_out
1110	R_I_4_out
1111	AC16_out

Table 3.2

Multiplexer selection bits

Clock Divider

The internal clock of 50Mhz is divided to widen the clock width in time duration and the fan in to the clock divider is just one input from the board's internal clock and the fan out is the combination of all the modules that require a clock signal as the input. Loading effect can degrade the module's performance but such behavior is not observed within the implementation. Testing of the clock divide is straightforward as x edges gap is observed within each positive edge of the widened clock signal. X is the ratio of previous clock rate to the new clock rate.

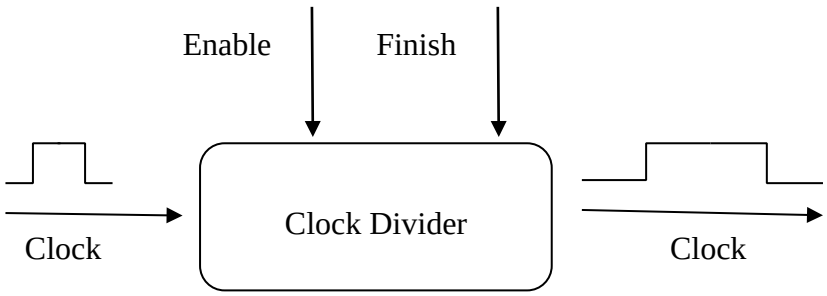


Figure 3.7

Block Diagram for the clock divider

Serial communication from PC to FPGA

According to the model processor architecture, a personal computer is constantly required for the customized task of image downsampling. The processor is all inside the FPGA while the DRAM acts as a traditional RAM of a pc and is also inside FPGA. Due to the inheritance transfer speeds, a cache is not required to speed up. The PC stands like a secondary non volatile storage option so that the image file is initially and finally stored in the PC. In order to achieve the mentioned task, serial communication is required from the PC to FPGA and vice versa. PC is equipped with USB interface and in application level, MATLAB is used to transmit the image data to the FPGA in 8 bit wide data packets.

The communication protocol is UART (universal asynchronous receiver and transmitter) and it transmits and receives data in the absence of a clock signal so that is called asynchronous. The transmitter and receiver are both agreed on the baud rate of 9600bits per

second transfer rate. The sampling of the input line is continuous and once the line transition from high to low, it identifies it as a word income. A word is 8 bit data set. It is important to notice that the sampling of the middle of each data bit out of the 8 bit word is important to avoid any erroneous receiving. Determined from the baud rate, FPGA waits for the middle of the each bit and samples the value of it.

The following two modules are used for UART communication.

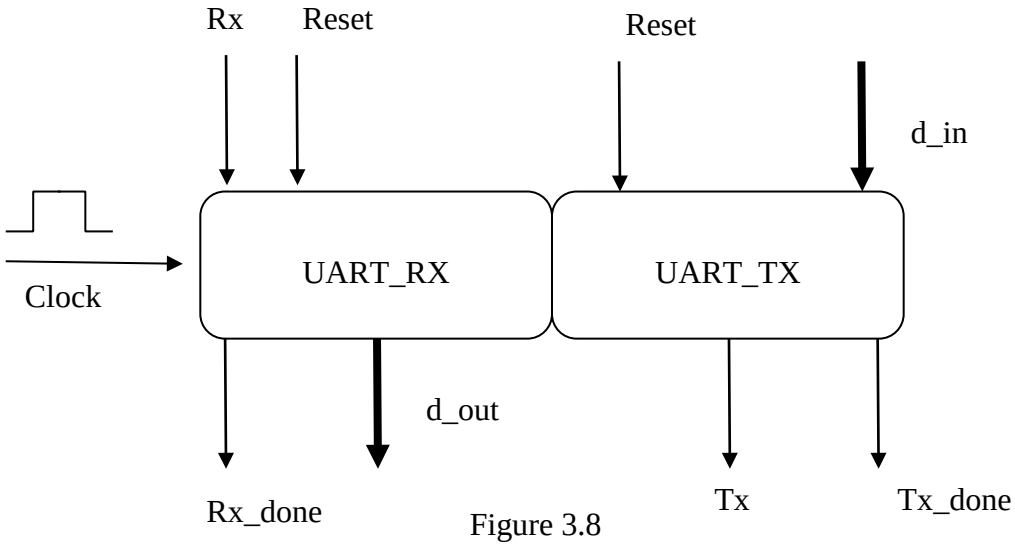


Figure 3.8

Block Diagram for the UART module

Testing of the UART

- Read an Black and Whiteimage from Matlab
- Use Matlab serial communication protocols to transmit the data at 9600 baud rate
- Use serial monitor softwares or virtual comm. Software to validate the Uint8 data transmission from Matlab. Most of the time the correct data format is disguised as PCs usually serial communicate using ASCII method.
- Have the UART_RX and UART_TX modules in the FPGA with an additional testbench register.

- Receive one byte of data from Matlab to the created register and transmit it back to Matlab at the next clock cycle.
- Meanwhile, Matlab should listen to the incoming data and store the receiving image data in a new image file
- Calculate the SSD value among the transmitted and received image.

Algorithm

The initial task that we were given was to down sample an event of the size 255 pixel X 255 pixel. First what we did was to store all those data in an array in a text file. Next the text file was sent over serially to an UART module with a speed of 9600 bits per second. Then the data is stored in the DRAM.

Then after all the calculations the data is transferred back from the FPGA to the UART module then to the we get the final object in an array and we makes it into a matrix and then get the image.

There were mainly two algorithms that were used, Filtering or low-pass algorithm and the down-sampling algorithm which are discussed below in detail.

Filtering Algorithm

A 3x3 Gaussian Kernel is used for filtering part and its weight distribution is shown in the

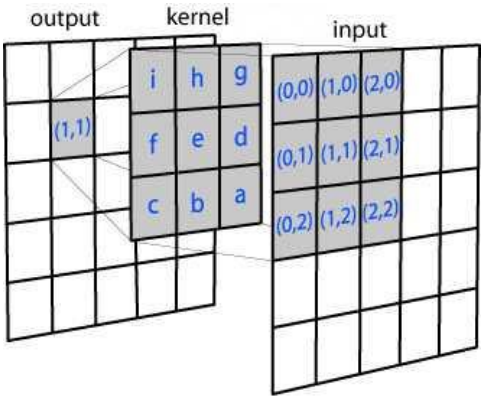
below figure. The weights are given with respect to the pixels distance to the given pixel. So after adding all the weights in the kernel and summing it up it will be assigned to the pixel in the middle. It is then divided by 16 to normalize it.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The below diagram shows how the low-pass algorithm is worked, Then the kernel moves towards the right until the end. After completing the row,the kernel will move to the next row moves along that row.

Likewise we can get the filtered (image with high frequency components removed) image

stored In the RAM in sequential behavior.



Down Sampling Algorithm

The objective is to down sample the image in to half the size of the original image. As an example what we do here is, we down sample the 256x256 image in to 128x128 image as graphically shown above.

In order to achieve this, one value per four pixels from the filtered image is stored in the memory to get the down sampled image.

Assembly Code For the Algorithm

```
1. NOP
2. CLAC
3. NOP
4. MEMADD 0
5. ADDI R7,R7,127 // width of down sampled image -1 // R_I_3 track
   the first cell of the kernel
6. ADDI R_I_1,R_I_1,65280//END address of first image 5X5 IMAGE
7. ADDI R3,R3,254//no of pixel skip to lower raw = column size( 5 )
   - 2
8. ADDI R5,R5,2//Gasssian kernal value 1,2,4
9. ADDI R6,R6,16//for normalize gaussian window
10.  ADDI R_I_2,R_I_2,65536 //location to save the filtered image
11.  READ
12.  MOV R1,MDR
13.  INCRE MAR
14.  READ
15.  MOV R2,MDR
16.  INCRE MAR
17.  READ
18.  ADD R1,R1,MDR
19.  ADD MAR,MAR,R3
20.  READ
```

```
21. ADD R2,R2,MDR
22. INCRE MAR
23. READ
24. MOV R4,MDR
25. INCRE MAR
26. READ
27. ADD R2,R2,MDR
28. ADD MAR,MAR,R3
29. MUL R4,R4,R5
30. READ
31. MUL R4,R4,R5
32. ADD R1,R1,MDR
33. INCRE MAR
34. READ
35. ADD R2,R2,MDR
36. INCRE MAR
37. READ
38. ADD R1,R1,MDR
39. MUL R2,R2,R5
40. ADD R1,R1,R4
41. ADD R1,R1,R2
42. MOV MAR,R_I_2
43. DIV R1,R1,R6
44. MOV MDR,R1
45. WRITE // STORE convoluted value
46. INCRE R_I_2//set the location to store next pixel
47. XOR AC,R8,R7 // CHECK WHETHER IT IS A END OF A RAW
48. JUMPZ 52
49. ADDI R8,R8,1
50. ADD R_I_3,R_I_3,R5 //INCREMENT THE START PIXEL OF THE KERNEL BY
    2
51. MOV MAR,R_I_3
52. SUB AC,AC,AC
53. JUMPZ 10
54. ADDI R_I_3,R_I_3,259 // CHANGE THE RAW
55. SUB R8,R8,R8
56. MOV MAR,R_I_3
57. XOR AC,R_I_1,R_I_3
58. JUMPNZ 10
59. STOP
```

Result Analysis

Analyzing results and comparing them with a sample image is very important for determining the error and get a clear idea about the correctness of our code.

Here the reference image is built using inbuilt MATLAB functions and then it is compared to the output of the FPGA. When we compare the two images we can get a clear idea about the accuracy of our processor architecture and design.

As same as in the processor ,the referencing image is made using the same image filtering process.

Same exact logic in the algorithms are implemented using MATLAB and then output of the two images were observed and discussed.

The error was generated in such a way that if there is a pixel value difference between any two alternative pixels has a difference the error count will be incremented by one. And after looping through the whole image the count will be divided by the total number and pixels and will be taken as an percentage. Fortunately for us the error percentage was exaxtly equal to zero which means that the logic we have implemented in FPGA has worked exactly correct.

Below is the matlab code of how it was generated.

```
instrfind
delete(instrfind)
img = imread('lenna.jpg');
gray_=rgb2gray(img);

[rows,columns]=size(gray_);
disp(size(gray_));
imshow(gray_);
COM_=serial('COM11','BaudRate',9600);
COM_.InputBufferSize=1;
fopen(COM_)
sampled_img=zeros(255,255,'uint8');

for i=1:rows
    for j=1:columns
        fwrite(COM_,gray_(i,j));

    end
   disp(i);
end

while(1)
    for i=1:127
        for j=1:127
            %Input1=char(Input1);
            %Input1=sscanf(Input1, '%d')
            Input1=fread(COM_);
            sampled_img(i,j)=Input1;
        end
        disp(i);
    end
    if(i==127 & j==127)
        break;
    end
end
subplot(1,2,1);
imshow(gray_);
subplot(1,2,2);
imshow(sampled_img);
```

```
disp(sampled_img);
%ERROR CALCULATION

    %MATLAB DOWNSAMPLED IMAGE
start_addr=[2 2];
addr=start_addr;
cal_pix=0;
Matlab_sampled_img=zeros(255,255,'double');

while(1)
    for i=1:127
        for j=1:127
            addr=[start_addr(1)+(j-1)*2 start_addr(2)+(i-1)*2];
            a11=[addr(1)-1 addr(2)-1];
            gray_(a11(2), a11(1));
            a12=[addr(1) addr(2)-1];
            gray_(a12(2), a12(1));
            a13=[addr(1)+1 addr(2)-1];
            gray_(a13(2), a13(1));
            a21=[addr(1)-1 addr(2)];
            gray_(a21(2), a21(1));
            a23=[addr(1)+1 addr(2)];
            gray_(a23(2), a23(1));
            a31=[addr(1)-1 addr(2)+1];
            gray_(a31(2), a31(1));
            a32=[addr(1) addr(2)+1];
            gray_(a32(2), a32(1));
            a33=[addr(1)+1 addr(2)+1];
            gray_(a33(2), a33(1));
            gray_(addr(2),addr(1));
            Matlab_sampled_img(i,j)=floor(((double(gray_(a11(2),
a11(1)))+double(gray_(a13(2), a13(1)))+double(gray_(a31(2),
a31(1)))+double(gray_(a33(2), a33(1))))+2*(double(gray_(a12(2),
a12(1)))+double(gray_(a21(2), a21(1)))+double(gray_(a23(2),
a23(1)))+double(gray_(a32(2),
a32(1))))+4*double(gray_(addr(2),addr(1))))/16);

        end
        disp(i);
    end
    if(i==127 & j==127)
        break;
    end
end
error=0;

while(1)
    for i=1:127
        for j=1:127

            if((uint8(Matlab_sampled_img(i,j))-sampled_img(i,j))~=0)
                error=error+1
            end
        end
    end
   if(i==127 & j==127)
        break;
    end
```

```
end
error=error*100/(127*127)
fclose(COM_);
delete(COM_);
```

Original Image

Matlab Image

FPGA Image

Following is the error matrix we got as a result.

	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0

After looking at the error matrix we can see that all the values are zero , so we can safely say that the image processing done on the FPGA is very accurate when comparing to the result from the MATLAB code. So we can say that the processor has accomplished the needed requirements which were given in the assignment.

Appendix

ALU

```
module ALU(A_bus,B_bus,AC16,AC2,sel,z_flag,N_flag,clk);
	output [15:0] AC16;
	output [1:0] AC2;
	output z_flag,N_flag;
	input [17:0] A_bus,B_bus;
	input [3:0] sel;
	input clk;
	reg [15:0] AC16;
	reg [1:0] AC2;
	wire z_flag,N_flag;
	parameter
NOP=4'd0,ADD=4'd1,SUB=4'd2,XOR=4'd3,MUL=4'd4,DIV=4'd5,JUMPZ=4'd6,JUMPNZ=4'd7,SUBI=4'd8
,ADDI=4'd9,WRITE=4'd10,READ=4'd11,INCR_R=4'd12,MARMEM=4'd13,MOV=4'd14,CLAC=4'd15,STO
P=8'd255;
	always @(posedge clk)
		begin
			case (sel)
				ADD	:	{AC2,AC16}=A_bus+B_bus;
				SUB	: {AC2,AC16}=A_bus-B_bus;
				XOR	: {AC2,AC16}=A_bus ^ B_bus;
				MUL	:begin
						if (B_bus== 16'd2)
							{AC2,AC16}=A_bus<<1;
						else if(B_bus== 16'd4)
							{AC2,AC16}=A_bus<<2;
						else if(B_bus== 16'd8)
							{AC2,AC16}=A_bus<<3;
						else if(B_bus== 16'd16)
							{AC2,AC16}=A_bus<<4;
					end
				DIV	: begin
						if (B_bus== 16'd2)
							{AC2,AC16}=A_bus>>1;
						else if(B_bus== 16'd4)
							{AC2,AC16}=A_bus>>2;
						else if(B_bus== 16'd8)
							{AC2,AC16}=A_bus>>3;
						else if(B_bus== 16'd16)
							{AC2,AC16}=A_bus>>4;
					end
				JUMPZ : {AC2,AC16}=B_bus;
				JUMPNZ: {AC2,AC16}=B_bus;
				SUBI	: {AC2,AC16}=A_bus-B_bus;
```

```
                                        ADDI    : {AC2,AC16}=A_bus+B_bus;
                                        MOV   : {AC2,AC16}=A_bus;
                                        default: {AC2,AC16}=AC16;
                        endcase
                end
        assign z_flag=({AC2,AC16}==18'd0)? 1'b1:1'b0;
        assign N_flag=(A_bus > B_bus)? 1'b1:1'b0;
endmodule
```

DEMUX

```
module
demux16_1_16(out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11,out12,out13,out1
4,out15,sel3);
        output
out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11,out12,out13,out14,out15;
        //input  [15:0] in;
        input [3:0] sel3;
        wire
m1,m2,m3,m4,out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11,out12,out13,out14,
out15;
        reg in=1'b1;
        demux16_1_4 mux4_1(in,m1,m2,m3,m4,sel3[3:2]);
        demux16_1_4 mux4_2(m1,out0,out1,out2,out3,sel3[1:0]);
        demux16_1_4 mux4_3(m2,out4,out5,out6,out7,sel3[1:0]);
        demux16_1_4 mux4_4(m3,out8,out9,out10,out11,sel3[1:0]);
        demux16_1_4 mux4_5(m4,out12,out13,out14,out15,sel3[1:0]);
endmodule

module demux16_1_2(in,out0,out1,sel);
        output out0,out1;
        input  in;
        input sel;
        wire out0,out1,in;
        assign out0=(sel==1'b0)? in:1'b0;
        assign out1=(sel==1'b1)? in:1'b0;
endmodule

module demux16_1_4(in,out0,out1,out2,out3,sel2);
```

```
        output out0,out1,out2,out3;
        input  in;
        input [1:0] sel2;
        wire  out0,out1,out2,out3,in,m1,m2;
        demux16_1_2 mux2_1(in,m1,m2,sel2[1]);
        demux16_1_2 mux2_2(m1,out0,out1,sel2[0]);
        demux16_1_2 mux2_3(m2,out2,out3,sel2[0]);

endmodule

/*module test_demux;
        wire
out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11,out12,out13,out14,out15;

        reg [3:0] sel3;
        demux16_1_16
D(out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11,out12,out13,out14,out15,sel3);
        initial
                begin
                        #5 sel3=4'd1;
                        #5 sel3=4'd2;
                        #5 sel3=4'd3;
                        #5 sel3=4'd4;
                        #5 sel3=4'd5;
                        #5 sel3=4'd6;
                        #5 sel3=4'd7;
                        #5 sel3=4'd8;
                        #5 sel3=4'd9;
                        #5 sel3=4'd10;
                        #5 sel3=4'd11;
                        #5 sel3=4'd13;
                #100 $finish;
                end
endmodule*/
```

Clock Divider

```
module CLOCK_DIVIDER(clk_IN,clk);

        input clk_IN;
        reg [3:0] counter=4'd0;
        output reg clk=1'd0;

        always@(posedge clk_IN)
                begin
                        if (counter==4'd9) // generate 1mhz frequency from 50mhz internal
oscillator
                                begin
                                        counter=4'd0;
                                        clk=1'b1;
                                end
                        else
                                begin
                                        counter=counter +4'd1;
                                        clk=1'b0;
                                end
                end
endmodule
```

Instruction Memory

```
module instuction_mem(data_in,data_out,addr,clk,write_en);
        output [15:0] data_out;
        input [15:0] data_in;
        input [9:0] addr;
        input clk,write_en;
        wire [15:0] data_in,data_out;
        wire clk,write_en;
        wire [9:0] addr;
        reg  [15:0] mem [1023:0];

        initial
                begin
mem[ 0 ]= 16'b 0000000000000000 ;
mem[ 1 ]= 16'b 1111000000000000 ;
mem[ 2 ]= 16'b 0000000000000000 ;
mem[ 3 ]= 16'b 1101000000000000 ;
mem[ 4 ]= 16'b 0000000000000000 ;
mem[ 5 ]= 16'b 1001100110010000 ;
mem[ 6 ]= 16'b 0000000000000001 ;
mem[ 7 ]= 16'b 1001101110110000 ;
mem[ 8 ]= 16'b 0000000000010100 ;
mem[ 9 ]= 16'b 1001010101010000 ;
mem[ 10 ]= 16'b 0000000000000011 ;
mem[ 11 ]= 16'b 1001011101110000 ;
```

```
mem[ 12 ]= 16'b 0000000000000010 ;
mem[ 13 ]= 16'b 1001100010000000 ;
mem[ 14 ]= 16'b 0000000000010000 ;
mem[ 15 ]= 16'b 1001110011000000 ;
mem[ 16 ]= 16'b 0000000000011001 ;
mem[ 17 ]= 16'b 1011000000000000 ;
mem[ 18 ]= 16'b 0000000000000000 ;
mem[ 19 ]= 16'b 1110001100010000 ;
mem[ 20 ]= 16'b 1100000000000000 ;
mem[ 21 ]= 16'b 1011000000000000 ;
mem[ 22 ]= 16'b 0000000000000000 ;
mem[ 23 ]= 16'b 1110010000010000 ;
mem[ 24 ]= 16'b 1100000000000000 ;
mem[ 25 ]= 16'b 1011000000000000 ;
mem[ 26 ]= 16'b 0000000000000000 ;
mem[ 27 ]= 16'b 0001001100110001 ;
mem[ 28 ]= 16'b 0001000000000101 ;
mem[ 29 ]= 16'b 1011000000000000 ;
mem[ 30 ]= 16'b 0000000000000000 ;
mem[ 31 ]= 16'b 0001010001000001 ;
mem[ 32 ]= 16'b 1100000000000000 ;
mem[ 33 ]= 16'b 1011000000000000 ;
mem[ 34 ]= 16'b 0000000000000000 ;
mem[ 35 ]= 16'b 1110011000010000 ;
mem[ 36 ]= 16'b 1100000000000000 ;
mem[ 37 ]= 16'b 1011000000000000 ;
mem[ 38 ]= 16'b 0000000000000000 ;
mem[ 39 ]= 16'b 0001010001000001 ;
mem[ 40 ]= 16'b 0001000000000101 ;
mem[ 41 ]= 16'b 0100011001100111 ;
mem[ 42 ]= 16'b 1011000000000000 ;
mem[ 43 ]= 16'b 0100011001100111 ;
mem[ 44 ]= 16'b 0001001100110001 ;
mem[ 45 ]= 16'b 1100000000000000 ;
mem[ 46 ]= 16'b 1011000000000000 ;
mem[ 47 ]= 16'b 0000000000000000 ;
mem[ 48 ]= 16'b 0001010001000001 ;
mem[ 49 ]= 16'b 1100000000000000 ;
mem[ 50 ]= 16'b 1011000000000000 ;
mem[ 51 ]= 16'b 0000000000000000 ;
mem[ 52 ]= 16'b 0001001100110001 ;
mem[ 53 ]= 16'b 0100010001000111 ;
mem[ 54 ]= 16'b 0001001100110110 ;
mem[ 55 ]= 16'b 0000000000000000 ;
mem[ 56 ]= 16'b 0001001100110100 ;
mem[ 57 ]= 16'b 1110000011000000 ;
mem[ 58 ]= 16'b 0101001100111000 ;
mem[ 59 ]= 16'b 0000000000000000 ;
```

```
mem[ 60 ]= 16'b 1110000100110000 ;
mem[ 61 ]= 16'b 1010000000000000 ;
mem[ 62 ]= 16'b 1100001000000000 ;
mem[ 63 ]= 16'b 0011111010101001 ;
mem[ 64 ]= 16'b 0110000001001000 ;
mem[ 65 ]= 16'b 0000000000000000 ;
mem[ 66 ]= 16'b 1001101010100000 ;
mem[ 67 ]= 16'b 0000000000000001 ;
mem[ 68 ]= 16'b 0001110111010111 ;
mem[ 69 ]= 16'b 0000000000000000 ;
mem[ 70 ]= 16'b 1110000011010000 ;
mem[ 71 ]= 16'b 0010111011101110 ;
mem[ 72 ]= 16'b 0110000000010001 ;
mem[ 73 ]= 16'b 0000000000000000 ;
mem[ 74 ]= 16'b 1001110111010000 ;
mem[ 75 ]= 16'b 0000000000001000 ;
mem[ 76 ]= 16'b 0010101010101010 ;
mem[ 77 ]= 16'b 1110000011010000 ;
mem[ 78 ]= 16'b 0011111010111101 ;
mem[ 79 ]= 16'b 0111000000010001 ;
mem[ 80 ]= 16'b 0000000000000000 ;
mem[ 81 ]= 16'b 0000000000000000 ;
mem[ 82 ]= 16'b 0000000000000000 ;
mem[ 83 ]= 16'b 1111111100000000 ;
                end


        always @(posedge clk)
                begin
                        if(write_en==1'b1)
                                mem[addr]=data_in;
                end
        assign  data_out=mem[addr];
endmodule
```

MUX

```
module mux16_2to1(out,in0,in1,sel);
        output [15:0] out;
        input [15:0] in0,in1;
        input sel;
        wire [15:0] out;
        assign out=(sel==0)? in0:in1;
endmodule
```

```
module mux6_2to1(out,in0,in1,sel);
	output [5:0] out;
	input [5:0] in0,in1;
	input sel;
	wire [5:0] out;
	assign out=(sel==0)? in0:in1;
endmodule

module mux16_4to1(out,in0,in1,in2,in3,sel);
	output [15:0] out;
	input [15:0] in0,in1,in2,in3;
	input [1:0] sel;
	wire [15:0] t1,t2,out;
	mux16_2to1 mux2_1(t1,in0,in1,sel[0]);
	mux16_2to1 mux2_2(t2,in2,in3,sel[0]);
	mux16_2to1 mux2_3(out,t1,t2,sel[1]);
endmodule

module mux16_16to1(out,in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13,in14,in15,sel);
	output [15:0] out;
	input [15:0] in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13,in14,in15;
	input [3:0] sel;
	wire [15:0] t1,t2,t3,t4,out;
	mux16_4to1 mux4_1(t1,in0,in1,in2,in3,sel[1:0]);
	mux16_4to1 mux4_2(t2,in4,in5,in6,in7,sel[1:0]);
	mux16_4to1 mux4_3(t3,in8,in9,in10,in11,sel[1:0]);
	mux16_4to1 mux4_4(t4,in12,in13,in14,in15,sel[1:0]);
	mux16_4to1 mux4_5(out,t1,t2,t3,t4,sel[3:2]);
endmodule

module mux18_2to1(out,in0,in1,sel);
	output [17:0] out;
	input [17:0] in0,in1;
	input sel;
	wire [17:0] out;
	assign out=(sel==0)? in0:in1;
endmodule

module mux18_4to1(out,in0,in1,in2,in3,sel);
	output [17:0] out;
	input [17:0] in0,in1,in2,in3;
	input [1:0] sel;
	wire [17:0] t1,t2,out;
	mux18_2to1 mux2_1(t1,in0,in1,sel[0]);
	mux18_2to1 mux2_2(t2,in2,in3,sel[0]);
	mux18_2to1 mux2_3(out,t1,t2,sel[1]);
endmodule
module mux18_16to1(out,in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13,in14,in15,sel);
```

```
        output [17:0] out;
        input [17:0] in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13,in14,in15;
        input [3:0] sel;
        wire [17:0] t1,t2,t3,t4,out;
        mux18_4to1 mux4_1(t1,in0,in1,in2,in3,sel[1:0]);
        mux18_4to1 mux4_2(t2,in4,in5,in6,in7,sel[1:0]);
        mux18_4to1 mux4_3(t3,in8,in9,in10,in11,sel[1:0]);
        mux18_4to1 mux4_4(t4,in12,in13,in14,in15,sel[1:0]);
        mux18_4to1 mux4_5(out,t1,t2,t3,t4,sel[3:2]);
endmodule

/*module test_muX;
                reg [15:0]
in0=16'd0,in1=16'd1,in2=16'd2,in3=16'd3,in4=16'd4,in5=16'd5,in6=16'd6,in7=16'd7,in8=16'd8,in9=1
6'd9,in10=16'd10,in11=16'd11,in12=16'd12,in13=16'd13,in14=16'd14,in15=16'd15;
                reg [3:0] sel;
                wire [15:0] out;

                mux16_16to1
M(out,in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13,in14,in15,sel);
                initial
                        begin
                                #5 sel= 4'd0;
                                #5 sel= 4'd1;
                                #5 sel= 4'd2;
                                #5 sel= 4'd3;
                                #5 sel= 4'd4;
                                #5 sel= 4'd5;
                                #5 sel= 4'd6;
                                #5 sel= 4'd7;
                                #5 sel= 4'd8;
                                #5 sel= 4'd9;
                                #5 sel= 4'd10;
                                #5 sel= 4'd11;
                                #5 sel= 4'd12;
                                #5 sel= 4'd13;
                                #5 sel= 4'd14;
                                #5 sel= 4'd15;
                        #100 $finish;
                end
        endmodule*/
```

Program Counter

```
module PC(out,in,clk,incr,rst,write_en);
        output [9:0] out;
        input [9:0] in;
        input clk,incr,rst,write_en;
        reg [9:0] out=10'b0;
```

```
	always @(posedge clk)
		begin
			if(rst==1'b1)
				out=10'b0;
			else
				begin
					if(write_en==1'b1)
						out=in;
					if(incr==1'b1)
						out=out+1'b1;
				end
		end

endmodule
```

Register

```
module MAR(Out18,in18, write_en,incre,clk);
	output [17:0] Out18;
	input [17:0] in18;

	//wire [1:0] in2;
	//wire [15:0] in16;
	input write_en,clk,incre;
	reg [17:0] Out18=18'b0;
	always @(posedge clk)
		begin
			if(write_en==1'b1)
				begin
					Out18=in18;
				end
			else if(incre==1'b1)
				begin
					Out18=Out18+1'b1;
				end
		end
endmodule

module TR_MAR(Out18,in16,in2, write_1,write_2,clk);
	output [17:0] Out18;
	input [15:0] in16;
	input [1:0] in2;
	//wire [1:0] in2;
	//wire [15:0] in16;
	input write_1,write_2,clk;
	reg [17:0] Out18;
	always @(posedge clk)
		begin
```

```
                        if(write_2==1'b1)
                                begin
                                        Out18[15:0]=in16;

                                end

                        if(write_1==1'b1)
                                begin
                                        Out18[17:16]=in2;
                                end
                end
endmodule

module dec_reg(out,in, write_en,clk);
        output [3:0] out;
        input [3:0] in;
        input write_en,clk;
        reg [3:0] out =4'b0;
        always @(posedge clk)
                begin
                if(write_en==1'b1)
                        out=in;
                end
endmodule

module A_reg(out,in1, write_en,clk,clear);
        output [17:0] out;
        input [17:0] in1;
        input write_en,clk,clear;
        reg [17:0] out=18'b0;
        always @(posedge clk)
                begin
                        if(write_en==1'b1)
                                begin
                                        out=in1;
                                end
                        if(clear==1'b1)
                                out=18'b0;
                end
endmodule


module register_ (out,in,clk, write_en,clear);
        input [15:0] in;
        input clk,write_en,clear;
        output [15:0] out;
        wire clk,write_en,clear;
        wire [15:0] in;
```

```
	reg [15:0] out=16'b0;
	always @(posedge clk)
		begin
			if(write_en == 1'b1)
				out=in;
			else if(clear==1'b1)
				out=16'b0;
		end
endmodule

module R_I(out,in,clk,incr,rst,write_en);
	output [17:0] out;
	input [17:0] in;
	input clk,incr,rst,write_en;
	reg [17:0] out=18'b0;
	always @(posedge clk)
		begin
			if(rst==1'b1)
				out=18'b0;
			else
				begin
					if(write_en==1'b1)
						out=in;
					else if(incr==1'b1)
						out=out+1'b1;
				end

		end


endmodule
```

SRAM

```
module SRAM(CE,WE,OE,LB,UB,Data_in_mem,Data_out_mem,data,write_1,read_1,clk);
	input [15:0] Data_in_mem;
	input write_1,read_1,clk;
	output [15:0] Data_out_mem;

	inout [15:0] data;
	output CE,WE,OE,LB,UB;
	reg CE=1'b0,WE=1'b1,OE=1'b1,LB=1'b0,UB=1'b0;
	assign data=(WE==1'b1)? Data_in_mem:Data_out_mem;

	always @(posedge clk)
		begin
			if(write_1 ==1'b1)
				begin
					WE=1'b0;
```

```
                    OE=1'b1;
                end
            else if(read_1==1'b1)
                begin
                    WE=1'b1;
                    OE=1'b0;
                end
        end
    endmodule
```

Processor

```
//module processor(MAR_in_sel,M_READ,M_WRITE,Out18,TR_in16,TR_in2,
write_TR_1,write_TR_2,Abus,Bbus,B_mux_out,c_bus,dem_out,A_bus,B_bus,halt,halt_PC,halt_DECO
DE,halt_IR_read,halt_ALU,halt_WRITEBACK,instr_reg_out,dec_r0_out,dec_r1_out,dec_r2_out,dec_r
0_out3,op_out,op_out2,op_out3,z_flag,clear,R_I_1_incre,R_I_2_incre,R_I_3_incre,sel_bits_A,sel_bit
s_B,sel_alu,A_reg_write,B_reg_write,long_inst_write,write_dec_reg,write_dec_reg2,write_dec_reg
3,sel_mux6_2,sel_B_bus_mux,sel_demux,imem_out,pc_incr,neg_edge,clk_main,data_out,addr,MAR
18,MDR_out,pc_out,r1_out,r2_out,r3_out,r4_out,r5_out,r6_out,r7_out,r8_out,r_i1_out,r_i2_out,r_i
3_out,AC16_out);//(clk_main,MAR18,MDR_out,Data_out_mem);

        //(MAR_in_sel,M_READ,M_WRITE,Out18,TR_in16,TR_in2,
write_TR_1,write_TR_2,Abus,Bbus,B_mux_out,c_bus,dem_out,A_bus,B_bus,halt,halt_PC,halt_DECO
DE,halt_IR_read,halt_ALU,halt_WRITEBACK,instr_reg_out,dec_r0_out,dec_r1_out,dec_r2_out,dec_r
0_out3,op_out,op_out2,op_out3,z_flag,clear,R_I_1_incre,R_I_2_incre,R_I_3_incre,sel_bits_A,sel_bit
s_B,sel_alu,A_reg_write,B_reg_write,long_inst_write,write_dec_reg,write_dec_reg2,write_dec_reg
3,sel_mux6_2,sel_B_bus_mux,sel_demux,imem_out,pc_incr,neg_edge,clk_main1,data_out1,addr1,
MAR181,MDR_out1,pc_out1,r1_out1,r2_out1,r3_out1,r4_out1,r5_out1,r6_out1,r7_out1,r8_out1,r_
i1_out1,r_i2_out1,r_i3_out1,in01,in151,AC16_out1);//(clk_main,MAR18,MDR_out,Data_out_mem);

module processor(CE,WE,OE,LB,UB,data,address_mem,start_pr,end_pr,led_out,clk_real,out,in);

    //output [17:0] address;

    //output [15:0] Data_in_mem;

    //input [15:0] Data_out_mem;

    input in;

    inout [15:0] data;

    output start_pr,end_pr,CE,WE,OE,LB,UB,out;

    output [7:0] led_out;

    wire [7:0] led_out;

    output [19:0] address_mem;



    input clk_real;
```

```
        wire clk_real,clk;
        //assign Data_in_mem=MDR_out;
        //assign address=MAR18;
        wire uart_en;
        wire start_pr,end_pr;
        wire [15:0] TR_in16;
        wire [1:0] TR_in2,d_bus;
        wire [17:0] Out18,MAR18,out_TR_MAR,r_i1_out,r_i2_out,r_i3_out;
        wire [15:0] Data_out_mem,data_out_from_reg;
        wire [17:0] Abus,A_bus,Bbus,B_bus,B_mux_out;
        wire [15:0]
MDR_in,B_mux_in2,instr_reg_out,data_in,imem_out,AC16_out,c_bus,MAR16,MDR_out,PC_extend,
r1_out,r2_out,r3_out,r4_out,r5_out,r6_out,r7_out,r8_out;
        wire [9:0] bit10line,pc_out;
        wire [5:0] bit6line,mux_in2;
        wire [3:0]
sel_demux,dec_r0_out,dec_r0_out2,dec_r0_out3,dec_r1_out,dec_r2_out,op_out2,op_out3,op_out,
sel_bits_B,sel_bits_A,sel_alu;
        wire
incre_MAR,MAR_in_sel,sel_mux6_2,write_TR_1,write_TR_2,MDR_in_sel,M_READ,M_WRITE,sel_B_
bus_mux,write_dec_op_reg,write_dec_reg,write_dec_reg2,write_dec_reg3,long_inst_write,write_e
n_imem,extra,A_reg_write,B_reg_write,z_flag,N_flag,PC_rst,clk_main,pc_incr,R_I_1_incre,R_I_2_inc
re,R_I_3_incre,clear,out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11,out12,out13,o
ut14,out15;
        reg [5:0] instr_extended=6'b0,PC_6bit=6'b0;
        reg [15:0] zero_reg=16'b0;
        reg [1:0] zero_reg2=2'b0;

        assign c_bus=AC16_out;

        wire  [15:0] to_mem,from_mem,Data_in_mem;
        wire [17:0] ADDR;
        wire write_back,write_2,read_2,write_1,read_1,bttn;
```

```
        wire [19:0] ADDR_add,MAR_add;
        assign ADDR_add= {2'b00,ADDR};
        assign MAR_add={2'b00,MAR18};
        assign address_mem=(uart_en==1'b1)? ADDR_add:MAR_add;
        assign write_1=(uart_en==1'b1)? write_2:M_WRITE;
        assign read_1=(uart_en==1'b1)? read_2:M_READ;
        CLOCK_DIVIDER clk_divider(clk_real,clk);
        UART_controller
UART_control(uart_en,led_out,out,in,start_pr,end_pr,ADDR,to_mem,from_mem,write_back,write_
2,read_2,clk,clk_real,bttn);
        SRAM mem_RAM(CE,WE,OE,LB,UB,Data_in_mem,Data_out_mem,data,write_1,read_1,clk);
        assign Data_in_mem=(uart_en==1'b1)? to_mem:MDR_out;
        assign Data_out_mem=(uart_en==1'b1)? from_mem:data_out_from_reg;


        mux18_2to1
MAR_input_mux(Out18,{d_bus,c_bus},out_TR_MAR,MAR_in_sel);//mux18_2to1(out,in0,in1,sel);
        mux16_2to1
MDR_input_mux(MDR_in,c_bus,data_out_from_reg,MDR_in_sel);//mux16_2to1(out,in0,in1,sel);
out=(sel==0)? in0:in1;
        assign TR_in2=dec_r2_out[3:2];
        assign TR_in16={op_out,dec_r0_out,dec_r1_out,dec_r2_out};
        assign PC_extend={PC_6bit,pc_out};
        TR_MAR    MAR_TR1(out_TR_MAR,TR_in16,TR_in2, write_TR_1,write_TR_2,clk);
        MAR      MAR1(MAR18,Out18, out0,incre_MAR,clk);// MAR(Out18,in18, write_en,incre,clk);
        register_ MDR(MDR_out,MDR_in,clk, out1,clear);//register_ (out,in,clk, write_en,clear);
        PC
pc1(pc_out,c_bus[9:0],clk,pc_incr,PC_rst,out2);//PC(out,in,clk,incr,rst,write_en);
        register_ R1(r1_out,c_bus,clk, out3,clear);//register_ (out,in,clk, write_en,clear);
        register_ R2(r2_out,c_bus,clk, out4,clear);//register_ (out,in,clk, write_en,clear);
        register_ R3(r3_out,c_bus,clk, out5,clear);//register_ (out,in,clk, write_en,clear);
        register_ R4(r4_out,c_bus,clk, out6,clear);//register_ (out,in,clk, write_en,clear);
        register_ R5(r5_out,c_bus,clk, out7,clear);//register_ (out,in,clk, write_en,clear);
        register_ R6(r6_out,c_bus,clk, out8,clear);//register_ (out,in,clk, write_en,clear);
```

```
        register_ R7(r7_out,c_bus,clk, out9,clear);//register_ (out,in,clk, write_en,clear);

        register_ R8(r8_out,c_bus,clk, out10,clear);//register_ (out,in,clk, write_en,clear);

        R_I
R_I_1(r_i1_out,{d_bus,c_bus},clk,R_I_1_incre,clear,out11);//R_I(out,in,clk,incr,rst,write_en);

        R_I
R_I_2(r_i2_out,{d_bus,c_bus},clk,R_I_2_incre,clear,out12);//R_I(out,in,clk,incr,rst,write_en);

        R_I
R_I_3(r_i3_out,{d_bus,c_bus},clk,R_I_3_incre,clear,out13);//R_I(out,in,clk,incr,rst,write_en);


        ALU
ALU1(A_bus,B_bus,AC16_out,d_bus,sel_alu,z_flag,N_flag,clk);//ALU(A_bus,B_bus,AC16,AC2,sel,z_fla
g,N_flag,clk);


        //mux16_16to1(out,in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,in11,in12,in13,in14,in15,sel);

        //mux16_16to1
mux_bus_A(Abus,in0,MDR_out,PC_extend,r1_out,r2_out,r3_out,r4_out,r5_out,r6_out,r7_out,r8_ou
t,r_i1_out,r_i2_out,r_i3_out,AC16_out,in15,sel_bits_A);

        mux18_16to1
mux_bus_A(Abus,MAR18,{zero_reg2,MDR_out},{zero_reg2,PC_extend},{zero_reg2,r1_out},{zero_re
g2,r2_out},{zero_reg2,r3_out},{zero_reg2,r4_out},{zero_reg2,r5_out},{zero_reg2,r6_out},{zero_reg2,
r7_out},{zero_reg2,r8_out},r_i1_out,r_i2_out,r_i3_out,{zero_reg2,AC16_out},{zero_reg2,zero_reg},s
el_bits_A);

        mux18_16to1
mux_bus_B(Bbus,{zero_reg2,zero_reg},{zero_reg2,MDR_out},{zero_reg2,PC_extend},{zero_reg2,r1_
out},{zero_reg2,r2_out},{zero_reg2,r3_out},{zero_reg2,r4_out},{zero_reg2,r5_out},{zero_reg2,r6_ou
t},{zero_reg2,r7_out},{zero_reg2,r8_out},r_i1_out,r_i2_out,r_i3_out,{zero_reg2,AC16_out},{zero_re
g2,zero_reg},sel_bits_B);


        //mux16_16to1
mux_bus_B(Bbus,zero_reg,MDR_out,PC_extend,r1_out,r2_out,r3_out,r4_out,r5_out,r6_out,r7_out,
r8_out,r_i1_out[15:0],r_i2_out[15:0],r_i3_out[15:0],AC16_out,zero_reg,sel_bits_B);


        //A_reg(out,in1,in2, write_en,extra,clk,clear);

        //A_reg A_bus_reg(A_bus,Abus,MAR2, A_reg_write,extra,clk,clear);

        //register_ A_bus_reg(A_bus,Abus,clk, A_reg_write,clear);//register_ (out,in,clk,
write_en,clear);

        A_reg A_bus_reg(A_bus,Abus, A_reg_write,clk,clear);//A_reg(out,in1, write_en,clk,clear);
```

```
        A_reg B_bus_reg(B_bus,B_mux_out, B_reg_write,clk,clear);

        //register_ B_bus_reg(B_bus,B_mux_out,clk, B_reg_write,clear);//register_ (out,in,clk,
write_en,clear);


        instuction_mem
inst_mem(data_in,imem_out,pc_out,clk,write_en_imem);//(data_in,data_out,addr,clk,write_en);


        register_ instr_reg(instr_reg_out,imem_out,clk, long_inst_write,clear);//register_ (out,in,clk,
write_en,clear);

        dec_reg op_code(op_out,instr_reg_out[15:12], write_dec_reg,clk);//dec_reg(out,in,
write_en,clk);

        dec_reg op_code2(op_out2,op_out, write_dec_reg2,clk);//dec_reg(out,in, write_en,clk);

        dec_reg op_code3(op_out3,op_out2, write_dec_reg3,clk);//dec_reg(out,in, write_en,clk);

        dec_reg dec_r0(dec_r0_out,instr_reg_out[11:8], write_dec_reg,clk);//dec_reg(out,in,
write_en,clk);

        dec_reg dec_r1(dec_r1_out,instr_reg_out[7:4], write_dec_reg,clk);//dec_reg(out,in,
write_en,clk);

        dec_reg dec_r2(dec_r2_out,instr_reg_out[3:0], write_dec_reg,clk);//dec_reg(out,in,
write_en,clk);


        dec_reg dec_r0_2(dec_r0_out2,dec_r0_out, write_dec_reg2,clk);//dec_reg(out,in,
write_en,clk);

        dec_reg dec_r0_3(dec_r0_out3,dec_r0_out2, write_dec_reg3,clk);//dec_reg(out,in,
write_en,clk);


        mux6_2to1
pc_inst_mux(bit6line,instr_extended,mux_in2,sel_mux6_2);//mux6_2to1(out,in0,in1,sel);

        assign mux_in2={op_out,dec_r0_out[3:2]};


        assign bit10line={dec_r0_out[1:0],dec_r1_out,dec_r2_out};

        assign B_mux_in2={bit6line,bit10line};

        //mux16_2to1
B_bus_mux(B_mux_out,Bbus,B_mux_in2,sel_B_bus_mux);//mux16_2to1(out,in0,in1,sel);
out=(sel==0)? in0:in1;
```

```
        mux18_2to1
B_bus_mux(B_mux_out,Bbus,{zero_reg2,B_mux_in2},sel_B_bus_mux);//mux18_2to1(out,in0,in1,sel
);

        demux16_1_16
demux1(out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11,out12,out13,out14,out15,
sel_demux);



 //state_machine
control_unit(MDR_in_sel,write_TR_1,write_TR_2,M_READ,M_WRITE,dec_r0_out3,op_out,op_out2,
op_out3,clk_in,N_flag,z_flag,clear,pc_incr,PC_rst,R_I_1_incre,R_I_2_incre,R_I_3_incre,sel_bits_A,sel
_bits_B,sel_alu,A_reg_write,B_reg_write,long_inst_write,write_dec_reg,write_dec_reg2,write_dec_
reg3,sel_mux6_2,sel_B_bus_mux,sel_demux);

        state_machine
control_unit(start_pr,end_pr,dec_r0_out,dec_r1_out,dec_r2_out,MDR_in_sel,write_TR_1,write_TR_
2,M_READ,M_WRITE,dec_r0_out3,op_out,op_out2,op_out3,clk,N_flag,z_flag,clear,pc_incr,PC_rst,R
_I_1_incre,R_I_2_incre,R_I_3_incre,sel_bits_A,sel_bits_B,sel_alu,A_reg_write,B_reg_write,long_inst
_write,write_dec_reg,write_dec_reg2,write_dec_reg3,sel_mux6_2,sel_B_bus_mux,sel_demux,MAR
_in_sel,incre_MAR);


        //(halt,halt_PC,halt_DECODE,halt_IR_read,halt_ALU,halt_WRITEBACK,neg_edge,dec_r0_out,
dec_r1_out,dec_r2_out,MDR_in_sel,write_TR_1,write_TR_2,M_READ,M_WRITE,dec_r0_out3,op_ou
t,op_out2,op_out3,clk_in,N_flag,z_flag,clear,pc_incr,PC_rst,R_I_1_incre,R_I_2_incre,R_I_3_incre,sel
_bits_A,sel_bits_B,sel_alu,A_reg_write,B_reg_write,long_inst_write,write_dec_reg,write_dec_reg2,
write_dec_reg3,sel_mux6_2,sel_B_bus_mux,sel_demux,MAR_in_sel,incre_MAR)


        //halt,halt_PC,halt_DECODE,halt_IR_read,halt_ALU,halt_WRITEBACK,neg_edge,

                //____ TEMPORARY MEMORY


        //^____ TEMPORARY MEMORY


endmodule
```

State Machine

```
module
state_machine(start_pr,end_pr,dec_r0_out,dec_r1_out,dec_r2_out,MDR_in_sel,write_TR_1,write_T
R_2,M_READ,M_WRITE,dec_r0_out3,op_out,op_out2,op_out3,clk_in,N_flag,z_flag,clear,pc_incr,PC_
rst,R_I_1_incre,R_I_2_incre,R_I_3_incre,sel_bits_A,sel_bits_B,sel_alu,A_reg_write,B_reg_write,long
```

```
_inst_write,write_dec_reg,write_dec_reg2,write_dec_reg3,sel_mux6_2,sel_B_bus_mux,sel_demux,
MAR_in_sel,incre_MAR);
        //halt,halt_PC,halt_DECODE,halt_IR_read,halt_ALU,halt_WRITEBACK,neg_edge,
        //output neg_edge,halt,halt_PC,halt_DECODE,halt_IR_read,halt_ALU,halt_WRITEBACK;
        input clk_in,N_flag,z_flag,start_pr;
        input [3:0] dec_r0_out,dec_r1_out,dec_r2_out,op_out,op_out2,op_out3,dec_r0_out3;
        output
end_pr,incre_MAR,MAR_in_sel,MDR_in_sel,write_TR_1,write_TR_2,M_WRITE,M_READ,clear,pc_inc
r,PC_rst,R_I_1_incre,R_I_2_incre,R_I_3_incre,A_reg_write,B_reg_write,long_inst_write,write_dec_r
eg,write_dec_reg2,write_dec_reg3,sel_mux6_2,sel_B_bus_mux;
        output [3:0] sel_bits_A,sel_bits_B,sel_demux,sel_alu;
        reg
long_inst=1'b0,long_inst_2=1'b0,mem_access_inst=1'b0,mem_access_inst2=1'b0,stop_com=1'b0;

        reg
incre_MAR=1'b0,MAR_in_sel=1'b0,halt=1'b0,halt_PC=1'b0,halt_DECODE=1'b0,halt_IR_read=1'b0,hal
t_ALU=1'b0,halt_WRITEBACK=1'b0;
        reg
MDR_in_sel=1'b0,write_back=1'b0,write_TR_1=1'b0,write_TR_2=1'b0,M_READ=1'b0,M_WRITE=1'b
0,clear=1'b0,pc_incr=1'b0,PC_rst=1'b0,R_I_1_incre=1'b0,R_I_2_incre=1'b0,R_I_3_incre=1'b0,A_reg_
write=1'b0,B_reg_write=1'b0,long_inst_write=1'b0,write_dec_reg=1'b0,write_dec_reg2=1'b0,write_
dec_reg3=1'b0,sel_mux6_2=1'b0,sel_B_bus_mux=1'b0,stop_clk=1'b0;
        reg [3:0] sel_bits_A,sel_bits_B,sel_demux,sel_alu;
        parameter
NOP=4'd0,ADD=4'd1,SUB=4'd2,XOR=4'd3,MUL=4'd4,DIV=4'd5,JUMPZ=4'd6,JUMPNZ=4'd7,SUBI=4'd8
,ADDI=4'd9,WRITE=4'd10,READ=4'd11,INCR_R=4'd12,MARMEM=4'd13,MOV=4'd14,CLAC=4'd15,STO
P=8'd255;
        wire clk;

        assign end_pr=(stop_com==1'b0)? 1'b0:1'b1;
        assign clk = (stop_com==1'b0 && start_pr==1'b1)? clk_in:stop_clk;
        //assign clk = clk_in;
        // PC
        reg neg_edge=1'b0;
        always @(posedge clk)
                begin
                        neg_edge<=~neg_edge;
                end
        always @(negedge clk )
                begin
                if(neg_edge == 1'b1)
                        begin//NEGATIVE
                                //PC->
                                        long_inst_write=1'b0;
                                        if(halt_PC==1'b0 )
                                                begin
                                                        pc_incr<=1'b1;
                                                end
```

```
                else
                        begin
                                halt_PC<=1'b0;
                                pc_incr<=1'b0;
                        end
                //<-PC
                //<-DECODE
                if(halt_DECODE==1'b0 && halt_IR_read==1'b1)
                        begin
                                halt_IR_read<=1'b0;
                        end
                if(halt_DECODE==1'b0 )
                        begin
                                write_dec_reg<=1'b1;
                        end
                else
                        begin
                                write_dec_reg<=1'b0;
                        end
        //<-DECODE

        //<-IR READ
                write_TR_1<=1'b0;
                write_TR_2<=1'b0;
                write_dec_reg2<=1'b0;
                //write_dec_r0_2<=1'b0;
                A_reg_write<=1'b0;
                B_reg_write<=1'b0;
        //<-IR READ

        //<-ALU
                if(halt_ALU==1'b0 && halt_WRITEBACK==1'b1)
                        begin
                                halt_WRITEBACK<=1'b0;
                        end
                if(halt_ALU==1'b0)// && op_out2 !=NOP)

                        begin
                                if (op_out2 ==JUMPZ )//z==1 goto x
                                        begin
                                                if(z_flag==1'b1)
                                                        begin

write_dec_reg3<=1'b1;

sel_alu<=op_out2;
                                                                        halt<=1'b1;
```

```
//jump_en<=1'b1;

write_back<=1'b1;
                                                        end
                                                else
                                                        begin

write_back<=1'b0;
                                                                halt<=1'b0;

//jump_en<=1'b0;
                                                        end
                                                end
                                        else if (op_out2==JUMPNZ )//z==1 goto x
                                                begin
                                                        if(z_flag==1'b0)
                                                        begin

write_dec_reg3<=1'b1;

sel_alu<=op_out2;
                                                                halt<=1'b1;

//jump_en<=1'b1;

write_back<=1'b1;
                                                        end
                                                else
                                                        begin

write_back<=1'b0;
                                                                halt<=1'b0;

//jump_en<=1'b0;
                                                        end
                                                end
                                        else if(long_inst==1'b1)
                                                begin
                                                        long_inst_2<=1'b1;
                                                        write_back<=1'b0;
                                                        write_dec_reg3<=1'b1;
                                                        sel_alu<=NOP;
                                                end
                                        else if(long_inst_2==1'b1)
                                                begin
                                                        long_inst_2<=1'b0;
                                                        write_back<=1'b1;
```

```
                                        write_dec_reg3<=1'b0;
                                        sel_alu<=op_out2;
                                end
                        else if(mem_access_inst==1'b1)
                                begin
                                        write_back<=1'b0;
                                        write_dec_reg3<=1'b1;
                                        mem_access_inst2<=1'b1;
                                        sel_alu<=NOP;
                                end
                        else if(mem_access_inst2==1'b1)
                                begin
                                        //write_back<=1'b0;
                                        write_dec_reg3<=1'b0;
                                        mem_access_inst2<=1'b0;
                                        sel_alu<=NOP;
                                end
                        else if(op_out2 ==NOP)
                                begin
                                        write_dec_reg3<=1'b1;
                                        sel_alu<=NOP;
                                end
                        else
                                begin
                                        write_back<=1'b1;
                                        write_dec_reg3<=1'b1;
                                        sel_alu<=op_out2;
                                end
                end
        else
                begin
                        write_dec_reg3<=1'b0;
                        sel_alu<=NOP;
                end
//<-ALU

//<-WRITE BACK
        clear<=1'b0;
        PC_rst<=1'b0;
        M_READ<=1'b0;
        M_WRITE<=1'b0;
        sel_demux <= 4'b1111;
        R_I_1_incre<=1'b0;
        R_I_2_incre<=1'b0;
        R_I_3_incre<=1'b0;
        incre_MAR<=1'b0;
//<-WRITE BACK
end
```

```
else//POSITVE
        begin
        //<- PC
                pc_incr<=1'b0;
                if(halt_PC==1'b0 && halt_DECODE==1'b1)
                        begin
                                halt_DECODE<=1'b0;
                        end
                if(halt_PC==1'b0 )
                        begin
                                long_inst_write<=1'b1;
                        end
                else
                        begin
                                long_inst_write<=1'b0;
                        end
        //<-PC

        //<-DECODE
                write_dec_reg<=1'b0;

        //<-DECODE

        //<-IR READ
                if(halt_IR_read==1'b0 && halt_ALU==1'b1)
                        begin
                                halt_ALU<=1'b0;
                        end
                if(halt_IR_read==1'b0 )

                        begin
                                if (mem_access_inst==1'b1)
                                        begin
                                                write_TR_1<=1'b0;
                                                write_TR_2<=1'b1;

                                                mem_access_inst<=1'b0;
                                                write_dec_reg2<=1'b0;
                                        end

                                else if (long_inst==1'b1)
                                        begin
                                                A_reg_write<=1'b0;
                                                B_reg_write<=1'b1;
                                                sel_B_bus_mux<=1'b1;
                                                sel_mux6_2<=1'b1;
                                                long_inst<=1'b0;
                                                write_dec_reg2<=1'b0;
```

```
        end
else
        begin
//write_dec_r0_2<=1'b1;
                if({op_out,dec_r0_out}==STOP)
                        stop_com<='b1;

                write_dec_reg2<=1'b1;
                case (op_out)

        ADD:begin
                        long_inst<=1'b0;
                        sel_bits_A<=dec_r1_out;
                        sel_bits_B<=dec_r2_out;
                        A_reg_write<=1'b1;
                        B_reg_write<=1'b1;
                        sel_B_bus_mux<=1'b0;
                end
        SUB:begin
                        long_inst<=1'b0;
                        sel_bits_A<=dec_r1_out;
                        sel_bits_B<=dec_r2_out;
                        A_reg_write<=1'b1;
                        B_reg_write<=1'b1;
                        sel_B_bus_mux<=1'b0;
                end
        XOR:begin
                        long_inst<=1'b0;
                        sel_bits_A<=dec_r1_out;
                        sel_bits_B<=dec_r2_out;
                        A_reg_write<=1'b1;
                        B_reg_write<=1'b1;
                        sel_B_bus_mux<=1'b0;
                end
        MUL:begin
                        long_inst<=1'b0;
                        sel_bits_A<=dec_r1_out;
                        sel_bits_B<=dec_r2_out;
                        A_reg_write<=1'b1;
                        B_reg_write<=1'b1;
                        sel_B_bus_mux<=1'b0;
                end
        DIV:begin
                        long_inst<=1'b0;
                        sel_bits_A<=dec_r1_out;
                        sel_bits_B<=dec_r2_out;
                        A_reg_write<=1'b1;
                        B_reg_write<=1'b1;
```

```
                                        sel_B_bus_mux<=1'b0;
                                end
                        JUMPZ:begin
                                        long_inst<=1'b0;
                                        A_reg_write<=1'b0;
                                        B_reg_write<=1'b1;

sel_B_bus_mux<=1'b1;
                                        sel_mux6_2<=1'b0;
                                end
                        JUMPNZ:begin
                                        long_inst<=1'b0;
                                        A_reg_write<=1'b0;
                                        B_reg_write<=1'b1;

sel_B_bus_mux<=1'b1;
                                        sel_mux6_2<=1'b0;
                                end
                        SUBI:begin
                                        //if (long_inst==1'b0)
                                                //begin

sel_bits_A<=dec_r1_out;

A_reg_write<=1'b1;

long_inst<=1'b1;
                                                //end
                                /*      else
                                                begin


A_reg_write<=1'b0;

B_reg_write<=1'b1;

sel_B_bus_mux<=1'b1;

sel_mux6_2<=1'b1;

long_inst<=1'b0;
                                                end*/
                                end
                        ADDI:begin

                                        //if (long_inst==1'b0)
                                        //      begin
```

```
        sel_bits_A<=dec_r1_out;
        A_reg_write<=1'b1;
        long_inst<=1'b1;
                                                        //end
                                                /*      else
                                                        begin

        A_reg_write<=1'b0;
        B_reg_write<=1'b1;
        sel_B_bus_mux<=1'b1;
        sel_mux6_2<=1'b1;
        long_inst<=1'b0;
                                                        end*/
                                                end

                                        MOV:begin
                                                        long_inst<=1'b0;
        sel_bits_A<=dec_r1_out;
                                                        A_reg_write<=1'b1;
                                                end
                                        WRITE:begin
                                                end
                                        READ: begin
                                                end

                                        MARMEM:begin
                                                //if
(mem_access_inst==1'b0)
                                                        //begin
        write_TR_1<=1'b1;
        write_TR_2<=1'b0;
```

```
mem_access_inst<=1'b1;
                                                        //end
                                                    //else
                                                        //begin
                                                        //
write_TR_1<=1'b0;
                                                        //
write_TR_2<=1'b1;

                                                        //
mem_access_inst<=1'b0;
                                                        //end
                                                    end
                                        CLAC: begin


                                                    end



                                        default:begin

                                                        A_reg_write<=1'b0;
                                                        B_reg_write<=1'b0;
                                                    end
                            endcase

                                                end
                            end
                        else
                            begin
                                write_dec_reg2<=1'b0;
                                //write_dec_r0_2<=1'b0;
                                A_reg_write<=1'b0;
                                B_reg_write<=1'b0;
                            end
                    //IR READ

                    //<-ALU
                        write_dec_reg3<=1'b0;
                        sel_alu<=NOP;
                    //<-ALU

                    //<-WRITE BACK
                        if (halt ==1'b1)
                            begin
                                write_back<=1'b1;
```

```
                halt<=1'b0;
                halt_PC<=1'b1;
                halt_ALU<=1'b1;
                halt_DECODE<=1'b1;
                halt_IR_read<=1'b1;
                halt_WRITEBACK<=1'b1;
                sel_demux <= 4'd2;
                write_back<=1'b0;
        end
else if (halt_WRITEBACK ==1'b0 && op_out3 !=NOP)
        begin
                if(op_out3==MARMEM)
                        begin
                                sel_demux <= dec_r0_out3;
                                MAR_in_sel<=1'b1;
                        end
                else if(op_out3==INCR_R)
                        begin
                        write_back<=1'b0;
                        if (dec_r0_out3==4'd0)
                                begin
                                        //R_I_1_incre<=1'b0;
                                        //R_I_2_incre<=1'b0;
                                        //R_I_3_incre<=1'b0;
                                        incre_MAR<=1'b1;

                                end
                        else if (dec_r0_out3==4'd1)
                                begin
                                        R_I_1_incre<=1'b1;
                                        //R_I_2_incre<=1'b0;
                                        //R_I_3_incre<=1'b0;
                                        //incre_MAR<=1'b0;

                                end
                        else if (dec_r0_out3==4'd2)
                                begin
                                        //R_I_1_incre<=1'b0;
                                        R_I_2_incre<=1'b1;
                                        //R_I_3_incre<=1'b0;
                                        //incre_MAR<=1'b0;

                                end
                        else
                                begin
                                        //R_I_1_incre<=1'b0;
                                        //R_I_2_incre<=1'b0;
                                        R_I_3_incre<=1'b1;
```

```
                                                        //increre_MAR<=1'b0;

                                                                end
                                                        end
                                                else if(op_out3==WRITE)
                                                        begin
                                                                write_back<=1'b0;
                                                                M_WRITE<=1'b1;
                                                        end
                                                else if(op_out3==READ)
                                                        begin
                                                                sel_demux <= 4'd1;
                                                                write_back<=1'b0;
                                                                M_READ<=1'b1;
                                                                MDR_in_sel<=1'b1;
                                                        end
                                                else if(op_out3==CLAC)
                                                        begin
                                                                write_back<=1'b0;
                                                                clear<=1'b1;
                                                                //PC_rst<=1'b1;
                                                        end
                                                else if(write_back == 1'b1)
                                                        begin
                                                                sel_demux <= dec_r0_out3;
                                                                MDR_in_sel<=1'b0;
                                                                MAR_in_sel<=1'b0;
                                                                write_back<=1'b0;

                                                        end
                                                else
                                                        begin
                                                                sel_demux <= 4'b1111;
                                                        end
                                        end
                        //<-WRITE BACK
                        end

        end
/*always @(posedge clk)
        begin
                pc_incr<=1'b0;
                if(halt_DECODE==1'b1)
                        begin
                                halt_DECODE<=1'b0;
                        end
                if(halt_PC==1'b0 )
                        long_inst_write<=1'b1;
```

```
                else
                        long_inst_write<=1'b0;
        end */

// DECODE

/*always @(negedge clk)
        begin
                if(halt_IR_read==1'b1)
                        begin
                                halt_IR_read<=1'b0;
                        end
                if(halt_DECODE==1'b0 )
                        write_dec_reg<=1'b1;
                else
                        write_dec_reg<=1'b0;
        end
always @(posedge clk2)
        begin
                write_dec_reg<=1'b0;
        end

//IR READ

always @(posedge clk )
        begin
                if(halt_ALU==1'b1)
                        begin
                                halt_ALU<=1'b0;
                        end
                if(halt_IR_read==1'b0 )

                        begin
                                write_dec_reg2<=1'b1;
                                //write_dec_r0_2<=1'b1;
                                if({op_out,dec_r0_out}==STOP)
                                        stop_com<='b1;


                                case (op_out)

                                        ADD:begin
                                                        long_inst<=1'b0;
                                                        sel_bits_A<=dec_r1_out;
                                                        sel_bits_B<=dec_r2_out;
                                                        A_reg_write<=1'b1;
                                                        B_reg_write<=1'b1;
                                                        sel_B_bus_mux<=1'b0;
```

```
        end
SUB:begin
                long_inst<=1'b0;
                sel_bits_A<=dec_r1_out;
                sel_bits_B<=dec_r2_out;
                A_reg_write<=1'b1;
                B_reg_write<=1'b1;
                sel_B_bus_mux<=1'b0;
        end
XOR:begin
                long_inst<=1'b0;
                sel_bits_A<=dec_r1_out;
                sel_bits_B<=dec_r2_out;
                A_reg_write<=1'b1;
                B_reg_write<=1'b1;
                sel_B_bus_mux<=1'b0;
        end
MUL:begin
                long_inst<=1'b0;
                sel_bits_A<=dec_r1_out;
                sel_bits_B<=dec_r2_out;
                A_reg_write<=1'b1;
                B_reg_write<=1'b1;
                sel_B_bus_mux<=1'b0;
        end
DIV:begin
                long_inst<=1'b0;
                sel_bits_A<=dec_r1_out;
                sel_bits_B<=dec_r2_out;
                A_reg_write<=1'b1;
                B_reg_write<=1'b1;
                sel_B_bus_mux<=1'b0;
        end
JUMPZ:begin
                        long_inst<=1'b0;
                        A_reg_write<=1'b0;
                        B_reg_write<=1'b1;
                        sel_B_bus_mux<=1'b1;
                        sel_mux6_2<=1'b0;
                end
JUMPNZ:begin
                        long_inst<=1'b0;
                        A_reg_write<=1'b0;
                        B_reg_write<=1'b1;
                        sel_B_bus_mux<=1'b1;
                        sel_mux6_2<=1'b0;
                end
SUBI:begin
```

```
                                        if (long_inst==1'b0)
                                                begin

sel_bits_A<=dec_r1_out;
                                                        A_reg_write<=1'b1;
                                                        long_inst<=1'b1;
                                                end
                                        else
                                                begin

                                                        A_reg_write<=1'b0;
                                                        B_reg_write<=1'b1;

sel_B_bus_mux<=1'b1;
                                                        sel_mux6_2<=1'b1;
                                                        long_inst<=1'b0;
                                                end
                                end
                        ADDI:begin

                                        if (long_inst==1'b0)
                                                begin

sel_bits_A<=dec_r1_out;
                                                        A_reg_write<=1'b1;
                                                        long_inst<=1'b1;
                                                end
                                        else
                                                begin

                                                        A_reg_write<=1'b0;
                                                        B_reg_write<=1'b1;

sel_B_bus_mux<=1'b1;
                                                        sel_mux6_2<=1'b1;
                                                        long_inst<=1'b0;
                                                end
                                end


                        INCR_R:begin

                                        end
                        WRITE:begin

                                        end
```

```
                                        READ: begin

                                                end


                                        MARMEM:begin
                                                if (mem_access_inst==1'b0)
                                                        begin
                                                                write_TR_1<=1'b1;
                                                                write_TR_2<=1'b0;

mem_access_inst<=1'b1;
                                                        end
                                                else
                                                        begin
                                                                write_TR_1<=1'b0;
                                                                write_TR_2<=1'b1;


mem_access_inst<=1'b0;
                                                        end
                                                end
                                        CLAC: begin


                                                end



                                        default:begin

                                                        A_reg_write<=1'b0;
                                                        B_reg_write<=1'b0;
                                                end
                                endcase

                                end
                        else
                                begin
                                        write_dec_reg2<=1'b0;
                                        //write_dec_r0_2<=1'b0;
                                        A_reg_write<=1'b0;
                                        B_reg_write<=1'b0;
                                end
        end

always @(negedge clk2)
        begin
```

```
                write_TR_1<=1'b0;
                write_TR_2<=1'b0;
                write_dec_reg2<=1'b0;
                //write_dec_r0_2<=1'b0;
                A_reg_write<=1'b0;
                B_reg_write<=1'b0;

        end

//ALU

always @(negedge clk )
        begin
                if(halt_WRITEBACK==1'b1)
                        begin
                                halt_WRITEBACK<=1'b0;
                        end
                if(halt_ALU==1'b0 && op_out2 !=NOP)

                        begin
                                if (op_out2 ==JUMPZ )//z==1 goto x
                                        begin
                                                if(z_flag==1'b1)
                                                        begin
                                                                write_dec_reg3<=1'b1;
                                                                sel_alu<=op_out2;
                                                                halt<=1'b1;
                                                                jump_en<=1'b1;
                                                                write_back<=1'b1;
                                                        end
                                                else
                                                        begin
                                                                write_back<=1'b0;
                                                                halt<=1'b0;
                                                                jump_en<=1'b0;
                                                        end
                                        end
                                else if (op_out2==JUMPNZ )//z==1 goto x
                                        begin
                                                if(z_flag==1'b0)
                                                        begin
                                                                write_dec_reg3<=1'b1;
                                                                sel_alu<=op_out2;
                                                                halt<=1'b1;
                                                                jump_en<=1'b1;
                                                                write_back<=1'b1;
                                                        end
                                                else
```

```
                                        begin
                                                write_back<=1'b0;
                                                halt<=1'b0;
                                                jump_en<=1'b0;
                                        end
                                end
                        else if(long_inst==1'b1)
                                begin
                                        long_inst_2<=1'b1;
                                        write_back<=1'b0;
                                        write_dec_reg3<=1'b1;
                                        sel_alu<=NOP;
                                end
                        else if(long_inst_2==1'b1)
                                begin
                                        long_inst_2<=1'b0;
                                        write_back<=1'b1;
                                        write_dec_reg3<=1'b0;
                                        sel_alu<=op_out2;
                                end
                        else if(mem_access_inst==1'b1)
                                begin
                                        write_back<=1'b0;
                                        write_dec_reg3<=1'b1;
                                        mem_access_inst2<=1'b1;
                                        sel_alu<=NOP;
                                end
                        else if(mem_access_inst2==1'b1)
                                begin
                                        write_back<=1'b1;
                                        write_dec_reg3<=1'b0;
                                        mem_access_inst2<=1'b0;
                                        sel_alu<=NOP;
                                end
                        else
                                begin
                                        write_back<=1'b1;
                                        write_dec_reg3<=1'b1;
                                        sel_alu<=op_out2;
                                end
                end
        else
                begin
                        write_dec_reg3<=1'b0;
                        sel_alu<=NOP;
                end

end
```

```
	always @(posedge clk2)
		begin
			write_dec_reg3<=1'b0;
			sel_alu<=NOP;
		end

//WRITE BACK

	always @(posedge clk)
		begin
			if (halt ==1'b1)
				begin
					write_back<=1'b1;
					halt<=1'b0;
					halt_PC<=1'b1;
					halt_ALU<=1'b1;
					halt_DECODE<=1'b1;
					halt_IR_read<=1'b1;
					halt_WRITEBACK<=1'b1;
				end
			if (halt_WRITEBACK ==1'b0 && op_out3 !=NOP)
				begin
					if(op_out3==INCR_R)
						begin
						write_back<=1'b0;
						if (dec_r0_out3==4'd1)
							begin
								R_I_1_incre<=1'b1;
								R_I_2_incre<=1'b0;
								R_I_3_incre<=1'b0;

							end
						else if (dec_r0_out3==4'd2)
							begin
								R_I_1_incre<=1'b0;
								R_I_2_incre<=1'b1;
								R_I_3_incre<=1'b0;

							end
						else
							begin
								R_I_1_incre<=1'b0;
								R_I_2_incre<=1'b0;
								R_I_3_incre<=1'b1;

							end
						end
					else if(op_out3==WRITE)
```

```
                                        begin
                                                write_back<=1'b0;
                                                M_WRITE<=1'b1;
                                        end
                                else if(op_out3==READ)
                                        begin
                                                write_back<=1'b0;
                                                M_READ<=1'b1;
                                                MDR_in_sel<=1'b1;
                                        end
                                else if(op_out3==CLAC)
                                        begin
                                                write_back<=1'b0;

                                        end
                                else if(write_back == 1'b1)
                                        begin
                                                sel_demux <= dec_r0_out3;
                                                MDR_in_sel<=1'b0;
                                                write_back<=1'b0;
                                        end
                                else
                                        begin
                                                sel_demux <= 4'b1111;
                                        end
                        end
                end
        always @(negedge clk2)
                begin
                        M_READ<=1'b0;
                        M_WRITE<=1'b0;
                        sel_demux <= 4'b1111;
                        R_I_1_incre<=1'b0;
                        R_I_2_incre<=1'b0;
                        R_I_3_incre<=1'b0;
                end
        */
endmodule

/*      always @(negedge clk)
                begin
                        long_inst_write=1'b0;
                        if(halt_PC==1'b0 )
                                pc_incr<=1'b1;
                        else
                                halt_PC<=1'b0;
                                pc_incr<=1'b0;
                end
```

```
always @(posedge clk)
	begin
		pc_incr<=1'b0;
		if(halt_DECODE==1'b1)
			begin
				halt_DECODE<=1'b0;
			end
		if(halt_PC==1'b0 )
			long_inst_write<=1'b1;
		else
			long_inst_write<=1'b0;
	end */
```

Receiver

```
module Rx(rx,s_tick,out_data,recieve_over,recieve_start,recieving);
	input s_tick,rx;
	output reg [7:0] out_data =8'd0;
	reg [3:0] tick_count=4'b0000;
	parameter IDLE=2'b00,start =2'b01, data_bits=2'b10,end_bit=2'b11;
	reg [1:0] state = IDLE;
	reg [2:0]data_index=3'b111;
	output reg recieve_over=1'b0,recieve_start=1'b0,recieving=1'b0;

	//output [7:0] rx_check;

	always@(posedge s_tick)
		begin
			case(state)
				start:
					begin
						recieving=1'b1;
						if (tick_count==4'b0111)
							begin
								state<= data_bits;
								tick_count<=4'b0000;
								data_index<=3'b000;
							end
						else
							begin
								tick_count= tick_count+4'd1;
							end
					end
				data_bits:
					begin
						recieving=1'b1;
						if (tick_count==4'b1111)
							begin
								out_data[data_index]=rx;
								tick_count=4'b0000;
```

```
                                                    if(data_index==3'b111)
                                                            begin
                                                                    state= end_bit;

            tick_count=4'b0000;
                                                            end
                                                    else data_index=data_index+1'b1;
                                            end
                                    else tick_count=tick_count+1'b1;
                            end
                    end_bit:
                            begin
                                    recieving=1'b0;
                                    if (tick_count==4'b1111)
                                            begin
                                                    state =IDLE;
                                                    recieve_over=1'b1;
                                            end
                                    else tick_count=tick_count+1'b1;
                            end
                    default:
                            begin
                                    state=IDLE;
                                    tick_count=4'b0000;
                                    data_index=3'b000;
                                    recieve_over=1'b1;
                                    recieve_start=1'b0;
                                    recieving=1'b0;
                                    if(rx==1'b0)
                                            begin
                                                    state=start;
                                                    recieve_over=1'b0;
                                                    recieve_start=1'b1;
                                            end
                            end
                    endcase
            end

            //assign rx_check = out_data[7:0];
endmodule
```

Transmitter

```
module Tx(tx_data,s_tick,transmit_over,tx,transmit_begin,transmit_active);
  input s_tick;
        input transmit_begin;
  input [7:0]tx_data;
  output reg transmit_over=1'b1;
  parameter IDLE = 2'b00, start_bit=2'b01, data_bits=2'b10,end_bit=2'b11;
```

```
reg [1:0] state1= 2'b00;
reg [2:0]data_index=3'b111;
reg [3:0] counter_tick;
        output reg tx=1'b1;
        output reg transmit_active=1'b0;

        //output [7:0] rx_check;

always@(posedge s_tick)
   begin
      case (state1)
         start_bit:
            begin
               tx=1'b0;
               if (counter_tick==4'b1111)
                  begin
                     state1= data_bits;
                     counter_tick= 4'b0000;
                  end
               else counter_tick=counter_tick+4'd1;
            end
         data_bits:
            begin
               tx= tx_data[data_index];
               if (counter_tick==4'b1111)
                  begin
                     counter_tick=  4'b0000;
                     if(data_index==3'b111) state1=end_bit;
                     else data_index=data_index+3'd1;
                  end
               else counter_tick=counter_tick+4'd1;
            end
         end_bit:
            begin
               tx=1'b1;
                                                          transmit_active=1'b0;
               if (counter_tick==4'b1111)
                                                          begin
                                                                state1=IDLE;
                                                                transmit_over=1'b1;
                                                          end
               else counter_tick=counter_tick+4'b1;
            end
         default:
            begin
               state1=IDLE;
               tx=1'b1;
               counter_tick=3'b000;
```

```
                data_index=3'b000;
                                                        transmit_over=1'b1;
                                                        transmit_active=1'b0;
                if(transmit_begin==1'b1)
                                                                begin// here We used a key
has a input therefore change when you use in the processor design
                                                                        state1=start_bit;

        transmit_active=1'b1;
                                                                        transmit_over=1'b0;
                                                                end
            end
        endcase
    end

                //assign rx_check= transmit_begin ;
endmodule
```

UART Clock

```
module UART_clk(clk, s_tick);
        input clk;
        output s_tick;
        parameter clk_devide_count=9'd320;
        reg [8:0]count =9'd0;

        always@(posedge clk)
                begin
                        if(clk_devide_count==count) count =9'd0;
                        else count=count + 1'b1;
                end

        assign s_tick =(count == clk_devide_count) ? 1'b1:1'b0;
endmodule
```

UART Controller

```
module
UART_controller(uart_en,led_out,out,in,start_pr,end_pr,ADDR,to_mem,from_mem,write_back,writ
e_2,read_2,clk,clk_real,bttn);
        output start_pr,write_2,read_2,out;
        output [15:0] to_mem;
        input [15:0] from_mem;
        wire  [15:0] to_mem,from_mem;
        output [17:0] ADDR;
        output [7:0]led_out;
        output uart_en;
```

```
	reg uart_en=1'b1;
	reg write_2=1'b0,read_2=1'b0;
	input clk,clk_real,bttn,end_pr,write_back,in;
	reg start_pr=1'b0;
	reg [17:0] ADDR;
	wire
s_tick,transmit_over,tx,transmit_active,recieve_over,recieve_start,recieving,clk,clk_real;
	UART_clk uartclk(clk_real, s_tick);
	Tx
TX(tx_data,s_tick,transmit_over,out,transmit_begin,transmit_active);//Tx(tx_data,s_tick,transmit_o
ver,tx,transmit_begin,transmit_active);
	Rx
RX(in,s_tick,led_out,recieve_over,recieve_start,recieving);//Rx(rx,s_tick,out_data,recieve_over,recie
ve_start,recieving);
	reg [7:0] tx_data;
	wire [7:0] led_out;
	parameter write_addr=18'd0;
	parameter read_addr=18'd25;
	parameter end_receive=18'd24;
	parameter end_transmit=18'd28;
	reg [17:0] rx_track=write_addr;
	reg [17:0] tx_track=read_addr;
	reg transmit_begin=1'b0;
	reg botton=1'b0;
	reg int_tx2=1'b0,int_tx=1'b0;
	reg last_receiving=1'b1;
	assign to_mem={8'b0,led_out};
	always @(negedge bttn)
		begin
			botton=1'b1;
		end


	always @(posedge clk)
		begin
			if(last_receiving!=recieving && recieving==1'b0)
				begin
					ADDR=rx_track;

					//tem_ADDR=1'b0;
				end
			last_receiving=recieving;
			if(end_pr==1'b0)
				if((1'b1+end_receive)==rx_track)
					begin
						start_pr=1'b1;
						uart_en=1'b0;
					end
```

```
                else
                        begin
                                if(ADDR==rx_track)
                                        begin

                                                write_2=1'b1;
                                                rx_track=rx_track+1'b1;
                                        end
                                else
                                        begin
                                                write_2=1'b0;
                                        end
                        end
        else
                begin
                        uart_en=1'b1;
                        start_pr=1'b1;
                        write_2=1'b0;
                        if(end_transmit!=(tx_track+1'b1))
                                begin
                                        if(transmit_active==1'b1)
                                                begin
                                                        transmit_begin=1'b0;
                                                end
                                        else if(transmit_over==1'b1 &&
transmit_begin==1'b0)
                                                begin
                                                        ADDR=tx_track;
                                                        int_tx=1'b1;


                                                end
                                        else if(int_tx==1'b1)
                                                begin
                                                        int_tx=1'b0;
                                                        tx_data=from_mem[7:0];
                                                        read_2=1'b1;
                                                        int_tx2=1'b1;
                                                end
                                        if(int_tx2==1'b1)
                                                begin
                                                        read_2=1'b0;
                                                        transmit_begin=1'b1;
                                                        int_tx2=1'b0;
                                                        tx_track=tx_track+18'b1;
                                                end
                                end
```

```
                                    end
                    end
endmodule
```

Compiler

```
def avoid_pipeline_hazard(codeline):
   if len(instruction_list)==(codeline+1):
      return "NOP"
   elif instruction_list[codeline][0]=="JUMPZ"or instruction_list[codeline][0]=="JUMPNZ" or
len(present_reg_no)==14  :
      return "NOP"
   if instruction_list[codeline][0]=="WRITE" or instruction_list[codeline][0]=="READ":
      if "MDR" in present_reg_no or "MAR" in present_reg_no:
         if "MDR" not in present_reg_no:
            present_reg_no.append("MDR")
         if "MAR" not in present_reg_no:
            present_reg_no.append("MAR")
      else:
         return codeline

   return avoid_cache_hazard(codeline+1)


assembly_code = open("assembly_code.txt", "r")
#Read assembly code
assembly_text = assembly_code.read()
assembly_list = assembly_text.split('\n')
assembly_code.close()
#print (assembly_list)
instruction_list=[]
for i in range(len(assembly_list)):
   k=assembly_list[i]
   if "//" in k:
      #print (k)
      for j in range(len(k)):
         #print (k[j])
         if k[j]=="/":
            k=k[0:j]
            break
   k=k.split(' ')
   instruction=[]
   for l in k:
      if(l!=''):
         instruction.append(l)
   if len(instruction)>2:
      print ("Compile error line no:",i+1)
      #return
   if len(instruction)==2:
```

```
        instruction=instruction[0:1]+instruction[1].split(',')
    instruction_list.append(instruction)
print (instruction_list)
Assembly_inst=
['NOP','ADD','SUB','XOR','MUL','DIV','JUMPZ','JUMPNZ','SUBI','ADDI','WRITE','READ','INCRE','MEMAD
D','MOV','CLAC','STOP']
Opcode_bin_code=['0000','0001','0010','0011','0100','0101','0110','0111','1000','1001','1010','1011',
'1100','1101','1110','1111','1111','11111111']
operands=['MAR','MDR','PC','R1','R2','R3','R4','R5','R6','R7','R8','AC','R_I_1','R_I_2','R_I_3']
operand_dic={'MAR': '0000','MDR': '0001','PC': '0010','R1': '0011','R2': '0100','R3': '0101','R4':
'0110','R5': '0111','R6': '1000','R7': '1001','R8': '1010','R_I_1': '1011','R_I_2': '1100','R_I_3':
'1101','AC': '1110'}



def macinecodegenerate(instruction_list):

    machine_no=[]
    instruction_list_without_comments=[]
    machinecode=[]
    zero_str="0000000000000000"
    machine_code_line=0
    for line_no in range(len(instruction_list)):
        machine_no.append(machine_code_line)
        position=instruction_list[line_no]
        #print ("position",position)
        if len(position)>0:

            instruction_list_without_comments.append(position)
            machinecode.append(0)
            if position[0]=='NOP':
                machinecode[machine_code_line]='0000000000000000'
            elif position[0]=='ADD':
                machinecode[machine_code_line]='0001'
            elif position[0]=='SUB':
                machinecode[machine_code_line]='0010'
            elif position[0]=='XOR':
                machinecode[machine_code_line]='0011'
            elif position[0]=='MUL':
                machinecode[machine_code_line]='0100'
            elif position[0]=='DIV':
                machinecode[machine_code_line]='0101'
            elif position[0]=='JUMPZ':
                machinecode[machine_code_line]='0110'
            elif position[0]=='JUMPNZ':
                machinecode[machine_code_line]='0111'
            elif position[0]=='SUBI':
                machinecode[machine_code_line]='1000'
```

```
        elif position[0]=='ADDI':
            machinecode[machine_code_line]='1001'
        elif position[0]=='WRITE':
            machinecode[machine_code_line]='1010000000000000'
        elif position[0]=='READ':
            machinecode[machine_code_line]='1011000000000000'
        elif position[0]=='INCRE':
            machinecode[machine_code_line]='1100'
        elif position[0]=='MEMADD':
            machinecode[machine_code_line]='1101'
        elif position[0]=='MOV':
            machinecode[machine_code_line]='1110'
        elif position[0]=='CLAC':
            machinecode[machine_code_line]='1111000000000000'
        elif position[0]=='STOP':
            machinecode[machine_code_line]='1111111100000000'
        else:
            print ("Compile error line no:",line_no+1)
            #return
        if position[0] in Assembly_inst[1:6]:
            if len(position)!=4:
                print ("Compile error line no:",line_no+1)
                #return
            else:
                i=position[1]
                if i in operands:
                    machinecode[machine_code_line]=machinecode[machine_code_line]+operand_dic[i]
                    i=position[2]
                    if i in operands:

machinecode[machine_code_line]=machinecode[machine_code_line]+operand_dic[i]
                        i=position[3]
                        #if i in operands[1:12]:

machinecode[machine_code_line]=machinecode[machine_code_line]+operand_dic[i]
                        #else:
                         #   print ("Compile error line no:",line_no+1," can't use ",i,"as third register. Use only
R1,R2,R3,R4,R5,R6,R7,R8,MDR,PC,AC")
                            #return
                    else:
                        print ("Compile error line no:",line_no+1," can't use",i)
                        #retern
                else:
                    print ("Compile error line no:",line_no+1," can't use",i)
                    #return
        elif position[0]=='JUMPZ' or position[0]=='JUMPNZ':
            if len(position)!=2 :#check whether a number
                print ("Compile error line no:",line_no+1)
```

```
            #return
        else:
            binaryno=str(bin(int(position[1])))[2:]

            if len(binaryno)<11:
                machinecode[machine_code_line]=machinecode[machine_code_line]+zero_str[:12-
len(binaryno)]+binaryno
            else:
                print ("Compile error line no:",line_no+1)
                #return
    elif position[0]=='ADDI' or position[0]=='SUBI':
        if len(position)!=4 :#check whether a number
            print ("Compile error line no:",line_no+1)
            #return
        else:
            i=position[1]
            if i in operands:
                machinecode[machine_code_line]=machinecode[machine_code_line]+operand_dic[i]
                i=position[2]
                if i in operands:

machinecode[machine_code_line]=machinecode[machine_code_line]+operand_dic[i]+"0000"
                else:
                    print ("Compile error line no:",line_no+1," can't use",i)
                    #retern
            else:
                print ("Compile error line no:",line_no+1," can't use",i)
                #return
            binaryno=str(bin(int(position[3])))[2:]
            if len(binaryno)<17:
                machinecode.append(0)
                machine_code_line=machine_code_line+1
                machinecode[machine_code_line]=zero_str[:16-len(binaryno)]+binaryno
            else:
                print ("Compile error line no:",line_no+1)
                #return
    elif position[0] == 'MOV':
        if len(position)!=3 :#check whether a number
            print ("Compile error line no:",line_no+1)
            #return
        else:
            i=position[1]
            if i in operands:
                machinecode[machine_code_line]=machinecode[machine_code_line]+operand_dic[i]
                i=position[2]
                if i in operands:

machinecode[machine_code_line]=machinecode[machine_code_line]+operand_dic[i]+"0000"
```

```
                else:
                    print ("Compile error line no:",line_no+1)
                    #return
            else:
                print ("Compile error line no:",line_no+1)
                #return
        elif position[0] == 'INCRE':
            if len(position)!=2 :#check whether a number
                print ("Compile error line no:",line_no+1)
                #return
            else:
                i=position[1]
                if i =="R_I_1":

machinecode[machine_code_line]=machinecode[machine_code_line]+"000100000000"
                elif  i =="R_I_2":

machinecode[machine_code_line]=machinecode[machine_code_line]+"001000000000"
                elif i =="R_I_3":

machinecode[machine_code_line]=machinecode[machine_code_line]+"001100000000"
                elif i =="MAR":

machinecode[machine_code_line]=machinecode[machine_code_line]+"000000000000"
                else:
                    print ("Compile error line no:",line_no+1)
                    #return
        elif position[0] == 'MEMADD':
            if len(position)!=2 :#check whether a number
                print ("Compile error line no:",line_no+1)
                #return
            else:
                binaryno=str(bin(int(position[1])))[2:]

                if len(binaryno)<19:
                    if len(binaryno)>16:
                        machinecode[machine_code_line]=machinecode[machine_code_line]+zero_str[:2-
len(binaryno[16:])]+binaryno[16:]+"0000000000"
                        machine_code_line=machine_code_line+1
                        machinecode.append(0)
                        machinecode[machine_code_line]=zero_str[:16-len(binaryno)]+binaryno
                    else:

machinecode[machine_code_line]=machinecode[machine_code_line]+"000000000000"
                        machine_code_line=machine_code_line+1
                        machinecode.append(0)
                        machinecode[machine_code_line]=zero_str[:16-len(binaryno)]+binaryno
                else:
```

```
                print ("Compile error line no:",line_no+1)
                #return

        machine_code_line=machine_code_line+1
    print (len(instruction_list))
    for i in range(len(machine_no)):
        print (i," ",instruction_list[i][0] ," ",machine_no[i])
    for i in range(len(instruction_list)):
        if len(instruction_list[i])>0:
            if instruction_list[i][0]=='JUMPZ' or instruction_list[i][0]=='JUMPNZ':
                #print ("_________________",machine_no[int(instruction_list[i][1])])
                print ("_________________",(instruction_list[i][1]))
                binaryno=str(bin(machine_no[int(instruction_list[i][1])]))[2:]
                print (machinecode[machine_no[i]],"wqwqw")
                machinecode[machine_no[i]]=machinecode[machine_no[i]][0:4]+zero_str[:12-
len(binaryno)]+binaryno
    return instruction_list_without_comments,machinecode
instruction_list_without_comments,machinecode=macinecodegenerate(instruction_list)
for i  in machinecode:
    print (i)
present_reg_no=[]
new_code=[]
instruction_list=instruction_list_without_comments

for checkline in range(len(instruction_list)):
    if checkline < len(instruction_list)-1:
        print ("AA",instruction_list[checkline][0])
        if instruction_list[checkline][0]=="JUMPZ" or instruction_list[checkline][0]=="JUMPNZ" :

            new_code.append(checkline)
            new_code.append("NOP")
        elif instruction_list[checkline][0] in Assembly_inst[1:6]:
            if instruction_list[checkline+1][0] in Assembly_inst[1:6]:
                if instruction_list[checkline][1]==instruction_list[checkline+1][3] or
instruction_list[checkline][1]==instruction_list[checkline+1][2]:
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            elif instruction_list[checkline+1][0]=="MOV" or instruction_list[checkline+1][0]=="SUBI" or
instruction_list[checkline+1][0]=="ADDI" :
                if instruction_list[checkline][1]==instruction_list[checkline+1][2] :
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
```

```
            else:
                new_code.append(checkline)
        elif instruction_list[checkline][0]=="MOV" or instruction_list[checkline][0]=="SUBI" or
instruction_list[checkline][0]=="ADDI" :
            if instruction_list[checkline+1][0] in Assembly_inst[1:6]:
                if instruction_list[checkline][1]==instruction_list[checkline+1][3] or
instruction_list[checkline][1]==instruction_list[checkline+1][2]:
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            elif instruction_list[checkline+1][0]=="MOV" or instruction_list[checkline+1][0]=="SUBI" or
instruction_list[checkline+1][0]=="ADDI" :
                if instruction_list[checkline][1]==instruction_list[checkline+1][2] :
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            else:
                new_code.append(checkline)
        elif instruction_list[checkline][0]=="READ" :
            if instruction_list[checkline+1][0] in Assembly_inst[1:6]:
                if "MDR"==instruction_list[checkline+1][3] or "MDR"==instruction_list[checkline+1][2]:
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            elif instruction_list[checkline+1][0]=="MOV" or instruction_list[checkline+1][0]=="SUBI" or
instruction_list[checkline+1][0]=="ADDI" :
                if "MDR"==instruction_list[checkline+1][2] :
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            else:
                new_code.append(checkline)
        elif instruction_list[checkline][0]=="INCRE" :
            if instruction_list[checkline+1][0] in Assembly_inst[1:6]:
                if instruction_list[checkline][1]==instruction_list[checkline+1][3] or
instruction_list[checkline][1]==instruction_list[checkline+1][2]:
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
```

```
                new_code.append(checkline)
            elif instruction_list[checkline+1][0]=="MOV" or instruction_list[checkline+1][0]=="SUBI" or
instruction_list[checkline+1][0]=="ADDI" :
                if instruction_list[checkline][1]==instruction_list[checkline+1][2] :
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            else:
                new_code.append(checkline)
        elif instruction_list[checkline][0]=="MEMADD" :
            if instruction_list[checkline+1][0] in Assembly_inst[1:6]:
                if "MAR"==instruction_list[checkline+1][3] or "MAR"==instruction_list[checkline+1][2]:
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            elif instruction_list[checkline+1][0]=="MOV" or instruction_list[checkline+1][0]=="SUBI" or
instruction_list[checkline+1][0]=="ADDI" :
                if "MAR"==instruction_list[checkline+1][2] :
                    #avoid_cache_hazard(checkline)
                    new_code.append(checkline)
                    new_code.append("NOP")
                else:
                    new_code.append(checkline)
            else:
                new_code.append(checkline)
        elif instruction_list[checkline][0]=="STOP" :
            new_code.append("NOP")
            new_code.append("NOP")
            new_code.append(checkline)
        else:
            new_code.append(checkline)
    elif instruction_list[checkline][0]=="STOP" :
        new_code.append("NOP")
        new_code.append("NOP")
        new_code.append(checkline)

    else:
        new_code.append(checkline)
print (new_code)
instruction_list_without_comments=[]
inst_order=[]
inst_order_no=0;
for j in new_code:
    if j=="NOP":
```

```
        inst_order_no=inst_order_no+1;
        instruction_list_without_comments.append(['NOP'])
        print ("NOP")
    else:
        inst_order.append(inst_order_no)
        inst_order_no=inst_order_no+1;
        i=instruction_list[j]
        instruction_list_without_comments.append(i)
        if len(i)==4:
            print(i[0],i[1],",",i[2],",",i[3])
        elif len(i)==2:
            print(i[0],i[1])
        else:
            print(i[0])
for i in range(len(inst_order)):
    print (i,instruction_list[i],inst_order[i])
for i in range(len(instruction_list_without_comments)):
    if instruction_list_without_comments[i][0]=='JUMPZ' or
instruction_list_without_comments[i][0]=='JUMPNZ':

instruction_list_without_comments[i][1]=inst_order[int(instruction_list_without_comments[i][1])]
print ("instruction_list_without_comments")
for i in range(len(instruction_list_without_comments)):

    print (i,"  ",instruction_list_without_comments[i])
print ("instruction_list_without_comments")
instruction_list_without_comments,machinecode=macinecodegenerate(instruction_list_without_co
mments)
for i  in range(len(machinecode)):
    print ("mem[",i,"]= 16'b",machinecode[i],";")
```