FPGA Based Processor Design

Down Sample an Image

Dias, K. M. G. D. A. W.	140125B
Fernando, H. D.	140154L
Fernando, W. A. S. C.	140163M
Rasanka, D. V. Y.	140517E

This is submitted as a partial fulfillment for the module
EN3030: Electronic Circuits and System Design
Department of Electronic and Telecommunication Engineering
University of Moratuwa

28th of August, 2017

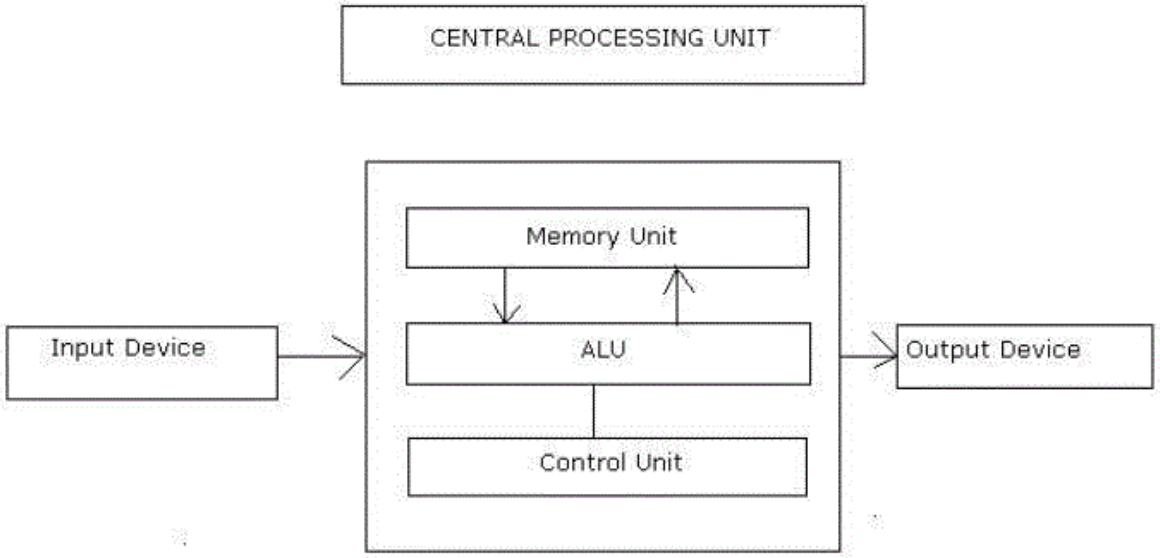
TABLE OF CONTENTS

1. INTRODUCTION

The objective of this project is to design a task specific microprocessor and CPU (Central Processing Unit) model to down-sample an image after eliminating its high frequency components and simulate the processor design using the Verilog hardware description language (HDL), and finally to implement it in hardware using programmable logic device such as Field Programmable Gate Array (FPGA). As an approach, design of an Instruction Set Architecture (ISA) and devising a suitable algorithm to down-sample the image are undertaken. The document describes the algorithm used to eliminate the high frequency components of the image, down-sample of the image and the ISA of the processor to be implemented on FPGA.

1.1. CPU AND PROCESSOR DESIGN

The architecture of the processor is such that the processing unit contains ALU, control unit and memory apart from the input and output of the system.



The following sections discuss the implementation of the processor with a collection of submodules mainly forming the units shown above. We designed 2 processors to down sample an image. One is based on the Reduced Instruction Set Architecture (RISC) which is faster in nature and tries to complete on operation in one clock cycle. The other one is based on Complex Instruction Set Architecture (CISC) which is more commonly used in most of the computers. We try to optimise both the design with the knowledge we have on processors,

their architecture and FPGA. This document provides the summary and details where needed about the process of completing the task, which is down-sampling an image using a processor implemented on FPGA.

1.2. PROBLEM STATEMENT

Problem state identifies the problem to be addressed in detail.

The main requirement is to design a CPU and a microprocessor for filtering and down sampling a given image converting the 512 x 512 pixel image to a 256 x 256 pixel image. The CPU should communicate with the computer in order to receive the image. Then it should save the received image in the main memory, as no such need for a secondary memory and process the image in order to down-sample. Afterwards, the CPU should send back the pixel values of the down-sampled image back to the computer Matlab programme to display the image. The task can be achieved by dividing the task into sub tasks.

Take input and store in the RAM

Matlab software generates a 1D array using the 2D image so that it can be transmitted to the RAM in FPGA board by Universal Asynchronous Receive and Transmit (UART) protocol byte by byte. UART protocol should be implemented inside the processor to receive and transmit data. RAM memory should be accessed and the pixel values should be stored one after the other in the memory. Transmission and receive of the pixel values is a major concern in the project. Apart from transmitting the image to the designed processor, there may be a need to transfer the instructions written in assembly language equivalent programme to down-sample the image.

Filter the image

Simplest way to down-sample the image is to take every other pixel from the received image and then transmit the pixel values. The issue with this method is that it cannot preserve the qualities of the original image. High frequency components in are the cause of the mentioned issue. This issue can be slightly overcome by averaging the near pixel value and then down-sampling the image but still it would not yield a good outcome. Therefore, in order to obtain

a reduced image with the features identical to the original image, the image should be filtered in order to eliminate the high frequency components from the image. This task can be done by using Gaussian filter implantation in the FPGA. After processing the raw data with Gaussian filter, processed data can be stored in the same primary memory.

Down sample the filtered image

After filtering the original image, in this section we consider down sampling the filtered image. As given in the requirements in this process image has to be down sampled into half of the size. Down-sampling process also should be implemented on the FPGA and after doing the down sampling process; data can be stored in the primary memory by overwriting the data written to the memory.

Send back the image and display

After doing the given tasks to the original image the processed data should be returned serially to back to the computer and the down-sampled image should be displayed. UART transmitter should be implemented on the FPGA in order to transmit the processed data from FPGA to the computer. Serial communication software (or Matlab) can be used to collect the data.

Down-sample image using Matlab

Apart from receiving the image, Matlab software should be used to down-sample image so that the results can be compared to access the quality of the image down-sampled inside the designed processor on FPGA board.

1.3. OVERVIEW OF THE SOLUTION

Field Programmable Gate Array can be used to design a processor capable of down-sampling an image after eliminating the high frequency components of the image. Atlys Spartan-6 FPGA board is used to implement the designed processor.

The first and foremost task is to design the UART receiving model so that the image can be transferred to the RAM inside the FPGA using serial communication. Then, a processor is specifically designed to handle the arithmetic and logical operations needed to perform the required operations. ISA is capable of addressing the operations required for the implementation of the algorithm.

Design can be done by dividing the complex solution into smaller sub modules.

The procedure can be identified as design of ISA and processor architecture, model the designed architecture using Verilog HDL, test the correctness of the implementation by simulation results, and implement the design in the FPGA board.

The solution includes receive the image through UART and writing the data in the memory, filter and down-sample the image using the designed processor, and send the down-sampled image back to the computer.

The effectiveness of the solution can be verified by carrying out an error analysis using MATLAB.

2. INSTRUCTION SET ARCHITECTURE (ISA)- CISC

Instruction architecture defines the architecture of the design in detail. This section includes details about the general architecture of a processing unit, data path and how the components are connected in register transfer level, instruction set itself and the procedure of the instruction execution.

2.1. GENERAL ARCHITECTURE

The processor is specifically designed to down sample an image of the size 512 x 512. The processor ISA designed so that it adheres to the requirement of the storage of all the pixel values and constraints of the FPGA board elements.

Data Memory - A data RAM which consists of 512 x 512, i.e. 218 memory locations with a width of 8 bits (1 Byte) to store the pixels of the image. The selection of the width of the memory was based on the pixel value range 0-255 of the image and depth of the memory was based on the number of pixel values in the image.

Instruction Memory - An instruction RAM which consists of instructions to be executed with 65536 memory locations and a width of 1 Byte same as in the DRAM. This basically contains the assembly code of the algorithm for filtering and down sampling the image.

Memory Address Register (MAR) – A 24 bit address register to select an address of the data RAM so that the data can be either read or write to the selected location of the data RAM.

Programme Counter (PC) – A 16 bit program counter which keeps the address of the next instruction in the instruction memory. PC selects the memory location of the instruction memory so that the instruction can be loaded to the Instruction Register.

Instruction Register (IR) – An 8 bit register to store the instructions that are read from the Instruction Memory. Instructions then are taken to the control store or state machine to generate the control signals to the units.

Accumulator (AC) – A 24 bit accumulator has direct access to the ALU via A bus. AC is the most widely used register in the ISA.

R, R1, R2, R3, R4 - Four 24 bit General Purpose Registers. These registers are used to store the intermediate variables used in the processing of the image.

Arithmetic and Logic Unit (ALU) –A 16 bit ALU performs different arithmetic and logical operations needed to filter and down-sample the image. A, B busses are used to input data in to the ALU and C bus gives the output. AC is directly connected to the ALU.

State Machine / Control Store –It generates all the control signals for the processor and makes the decision based on the given instruction. (From IR)

BUS – 24 bit wire which carries the data parallel from registers, Data memory, Instruction Memory, and ALU.

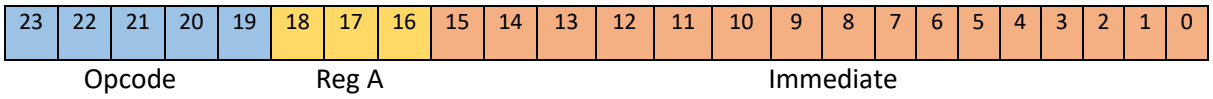
Reduced Instruction Set Architecture (RISC)

Since our group successfully completed the task using 2 different processors the below 5 pages corresponds to the processor design using RISC architecture.

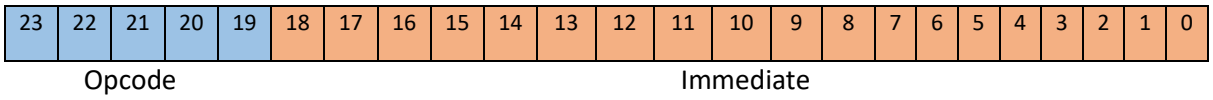
Instruction formats

1. RRR type

2. RI type



3. I type



Architecture

Data Memory (RAM) – 8M x 16 bits

Instruction Memory (ROM) – 4K x 24 bits

MAR – 24-bit Register

PC – 24-bit Register

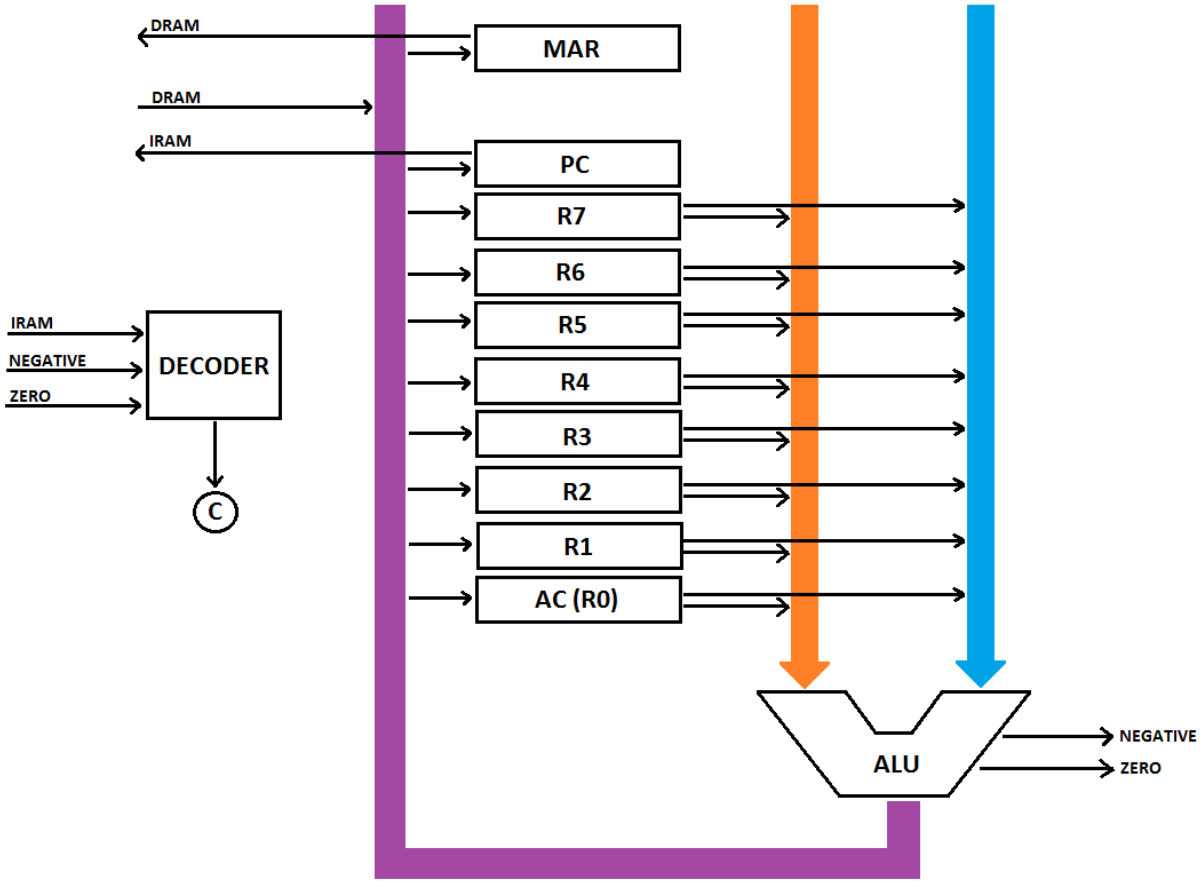
Accumulator (AC, R0) – 24-bit Register

R1 – R7 General Purpose Registers – 24-bit

Instruction Set - RISC

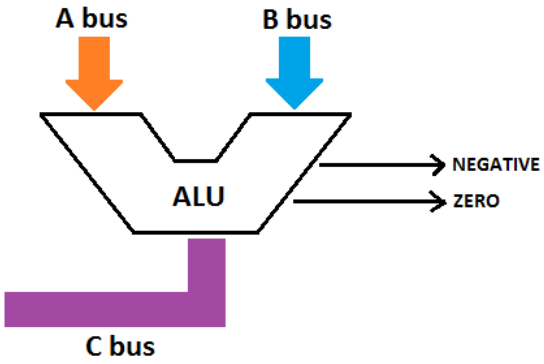
Instruction	Opcode	operation
NOP	00000	No operation
LOAD	00001	Reg A ← RAM[MAR]
STORE	00010	RAM[MAR] ← Reg B
MOVE	00011	Reg A ← Reg B
LDMAR	00100	MAR ← Reg A
LDMARI	00101	MAR ← signed immediate (19-bit)
LOADI	00110	Reg A ← signed immediate (16-bit)
LDACI	00111	AC ← signed immediate (19-bit)
ADD	01000	Reg A ← Reg B + Reg C
SUB	01001	Reg A ← Reg B - Reg C
MUL	01010	Reg A ← Reg B << Reg C
DIV	01011	Reg A ← Reg B >> Reg C
INC	01100	Reg A ← Reg A + 1
DEC	01101	Reg A ← Reg A - 1
NEG	01110	Reg A ← -Reg B
NOT	01111	Reg A ← Reg B (NOT) Reg C
AND	10000	Reg A ← Reg B (AND) Reg C
OR	10001	Reg A ← Reg B (OR) Reg C
XOR	10010	Reg A ← Reg B (XOR) Reg C
JGT	10011	If ALU out > 0 then PC ← IMM19 else PC ← PC + 1
JEQ	10100	If ALU out = 0 then PC ← IMM19 else PC ← PC + 1
JGE	10101	If ALU out >= 0 then PC ← IMM19 else PC ← PC + 1
JLT	10110	If ALU out < 0 then PC ← IMM19 else PC ← PC + 1
JNE	10111	If ALU out != 0 then PC ← IMM19 else PC ← PC + 1
JLE	11000	If ALU out <= 0 then PC ← IMM19 else PC ← PC + 1
JMP	11001	PC ← IMM19 (Unconditional Jump)

Data path



RISC Architecture Processor Design

Arithmetic and Logic Unit (ALU)



Opcode	ALU control bits	Operation
-	0000	No Operation
01000	0001	ADD (A + B)
01001	0010	SUB (A – B)
01010	0011	MUL (A << B)
01011	0100	DIV (A >> B)
01100	0101	INC (A + 1)
01101	0110	DEC (A – 1)
01110	0111	NEG (–A)
01111	1000	NOT (!A)
10000	1001	AND (A & B)
10001	1010	OR (A | B)
10010	1011	XOR (A ^ B)

If ALU output = 0, then Z = 1 else Z = 0

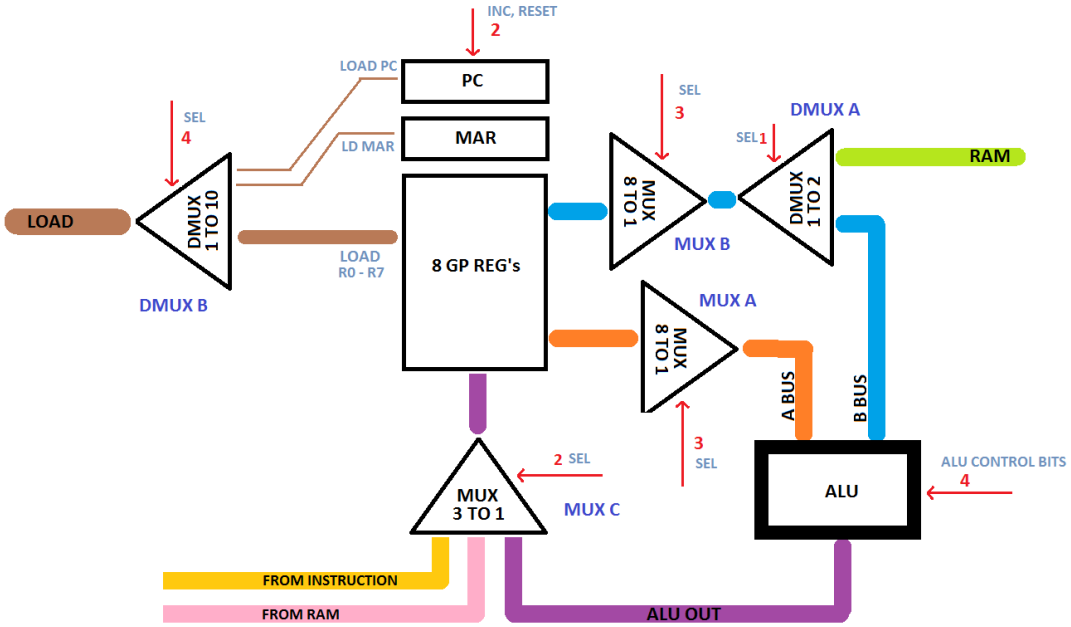
If ALU output < 0, the N = 1 else N = 0

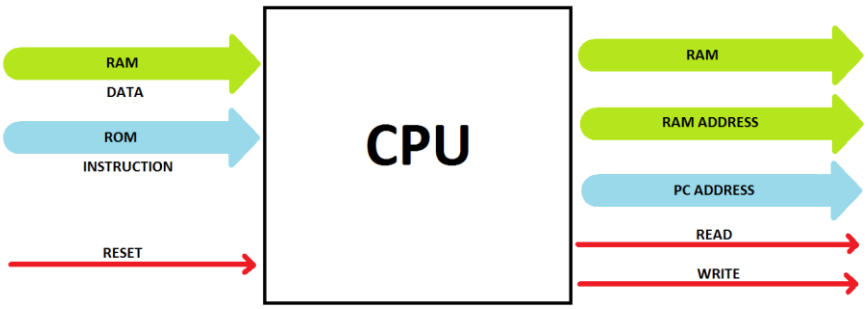
Jump Logic

Opcode	Operation	Z flag	N flag	Jump
JGT	If ALU out > 0	0	0	PC ← Reg A
JEQ	If ALU out = 0	1	x	PC ← Reg A
JGE	If ALU out >= 0	x	0	PC ← Reg A
JLT	If ALU out < 0	0	1	PC ← Reg A
JNE	If ALU out != 0	0	x	PC ← Reg A
JLE	If ALU out <= 0	1/0	1	PC ← Reg A
JMP	Unconditional Jump	x	x	PC ← Reg A

Control bits

Depending on the opcode and ALU flags the decoder will generate the control bits





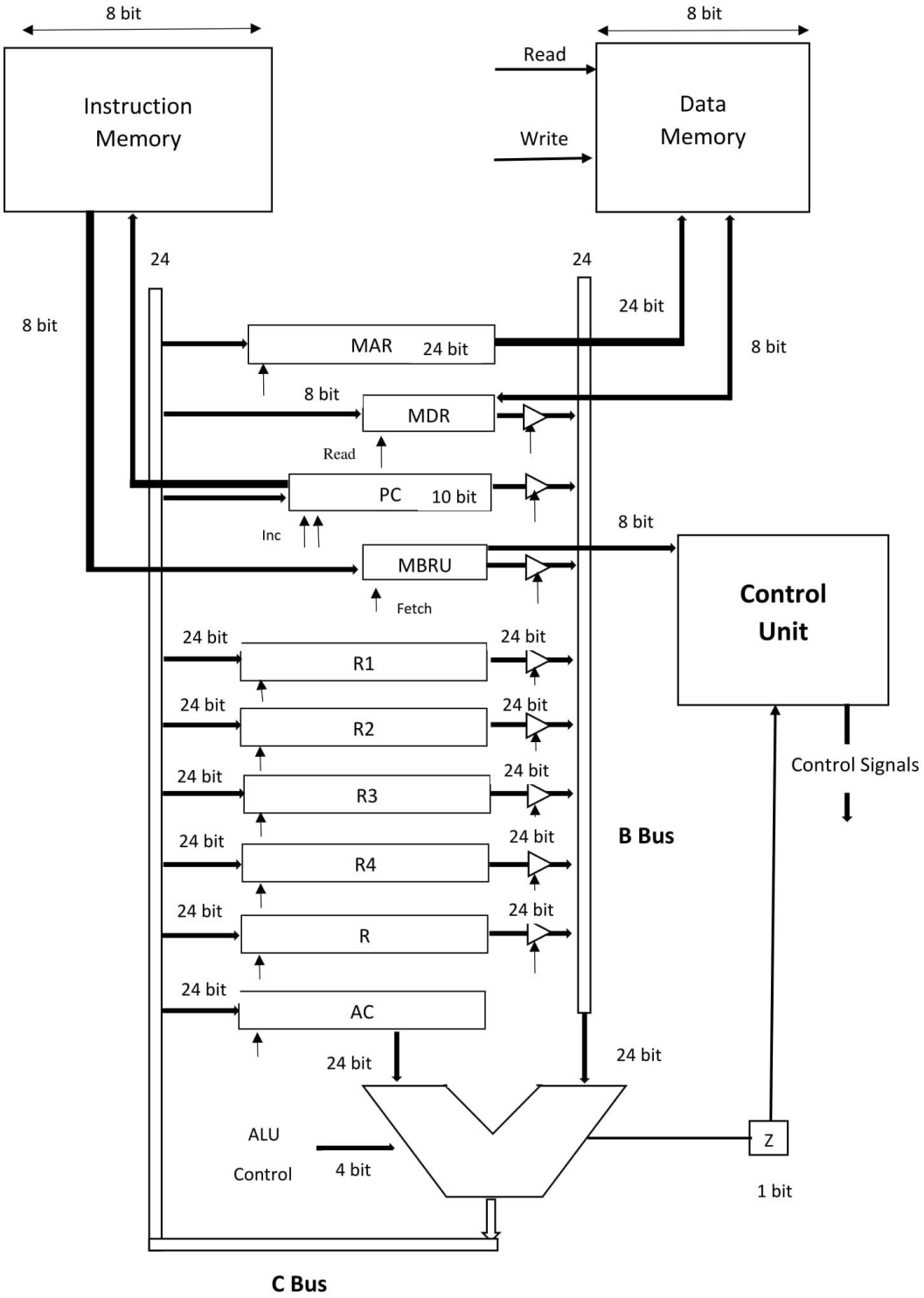
Design of the RISC processor

2.2. INSTRUCTION SET - CISC

Instruction	Instruction Code	Operation
NOP	8	No operation
LDAC	8 Ґ	AC ← DRAM [Ґ]; r is a 24 bit location in MAR
LDACV	8 V (16 bit)	AC ← V, V is loaded from IRAM
STAC	8 Ґ	DRAM [Ґ] ← AC; r is a 24 bit location in MAR
MVACMAR	8	MAR ← AC
MVACR	8	R ← AC
MVACR1	8	R1 ← AC
MVACR2	8	R2 ← AC
MVACR3	8	R3 ← AC
MVACR4	8	R4 ← AC
MOVR	8	AC ← R
MOVR1	8	AC ← R1
MOVR2	8	AC ← R2
MOVR3	8	AC ← R3
MOVR4	8	AC ← R4
JUMP	8 Ґ (16 bit)	GOTO IRAM [Ґ], Ґ is 16 bits
JMPZ	8 Ґ (16 bit)	IF (Z=1) THEN GOTO IRAM [Ґ], Ґ is 16 bits
JMNZ	8 Ґ (16 bit)	IF (Z=0) THEN GOTO IRAM [Ґ], Ґ is 16 bits
ADD	8	AC ← AC + R
ADDV	8 V(16 bit)	AC ← AC + V, V is loaded from IRAM
SUB	8	AC ← AC – R IF (AC – R = 0), THEN Z = 1, ELSE Z = 0
SUBV	8 V(16 bit)	AC ← V – AC, V is loaded from IRAM IF (AC – V = 0), THEN Z = 1, ELSE Z = 0
INAC	8	AC ← AC + 1
DEAC	8	AC ← AC – 1
MUL2	8	AC ← AC << 1
MUL4	8	AC ← AC << 2
MUL512	8	AC ← AC << 9
DIV16	8	AC ← AC >> 4
INR1	8	R1 ← R1 + 1
INR2	8	R2 ← R2 + 1
ADDR3	8	AC,R3 <- AC+R3
CLAC	8	AC ← 0; Z ← 1
AND	8	AC ← AC and R
OR	8	AC ← AC or R
XOR	8	AC ← AC xor R
NOT	8	AC ← AC’
ADDR3	8	AC, R3 ← AC + R3

2.3. DATAPATH OF THE DESIGN - CISC

The data path was designed taking the ISA and Mic-1 architecture into consideration. CISC processor.



2.4. INSTRUCTION CYCLE

An instruction cycle (sometimes called a fetch–decode–execute cycle) is the basic operational process of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction dictates, and carries out those actions. This cycle is repeated continuously by a computer's central processing unit (CPU). In simpler CPUs the instruction cycle is executed sequentially, each instruction being processed before the next one is started.

Fetch

The next instruction is fetched from the memory address that is currently stored in the program counter (PC), and stored in the instruction register (IR). At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle. Fetch cycle consists of only 2 states. Fetch cycle is run by the state machine with FETCH1 being set as the next state at the beginning.

FETCH1: MBRU ← Imem[pc]; fetch
FETCH2: PC ← PC+1

Decode

Decode of instructions is the next task of the CPU after fetching instructions from the instruction memory. The CPU has to differentiate between the instructions fetched from the instruction memory in order to invoke the correct execution cycle. This task is done by the state machine or the control unit of the processor. Memory Bus Register Unsigned (MBRU) inputs the fetched instruction to the control unit and the control unit decodes the instruction to output the control signals of the relevant state followed by the next states of the instruction or returns to fetch cycle if the instruction has only one state.

Execute

NOP Instruction
NOP instruction is used to do nothing. This instruction is useful to skip a clock cycle or two in order to wait until the data is ready at the end point.

LDAC Instruction
This instruction consists of two states.

LDAC1: MDR ← DRAM [MAR], read
LDAC2: AC ← MDR

1st state sends the read signal to the data memory and the MDR. This state involves loading the data from the address given in the previous state (AR) to AC. Then CPU moves to the fetch routine.

LDACV Instruction

This instruction consists of two states.

LDACV1: MBRU ← IRAM [PC], fetch
LDACV2: PC ← PC+1; AC ← MBRU
LDACV3: AC ← AC<<8; MBRU ← IRAM [PC], fetch
LDACV4: PC ← PC+1; AC ← AC+MBRU

1st state sends the fetch signal to the IR and it is loaded with the value stored in the instruction RAM. 2nd state increments PC to point to the next instruction and the moves the fetched value to the accumulator. Since the value is represented by 16 bits, the 3rd state involves fetching the second value from the instruction memory and in the final step the first and second values fetched are concatenated. CPU moves to the fetch routine after the completion of 4 states.

STAC Instruction

This instruction consists of 2 states.
STAC1: MDR ← AC
STAC2: DRAM [MAR] ← MDR; write

This involves copying the contents of the AC to the memory address in the data memory pointed by the MAR. Write data memory address should be available in the MAR. Move the data from the address stored register to the MAR register first before invoking this command.

MVACR1, MVACR2, MVACR3, MVACR4, MVACR and MVACMAR Instructions

These instructions consist of only one state. The CPU copies the contents of the AC to R1, R2, R3, R or MAR and moves to fetch routine.

MVACR11: R1←AC
MVACR21: R2←AC
MVACR31: R3←AC
MVACR41: R4←AC
MVACR1: R←AC
MVACMAR1: MAR←AC

MOVR, MOVR1, MOVR2, MOVR3 and MOVR4 Instructions

These instructions involve only one state. The CPU copies the contents of the R1, R2, R3 or R to AC and moves to fetch routine.

MOVR11: AC←R1
MOVR21: AC←R2
MOVR31: AC←R3
MOVR41: AC←R4
MOVR1: AC←R

JUMP Instruction

Four states are involved in the jump instruction. In the 2nd and 3rd states, jump address stored in the instruction is loaded into AC. Then value of AC is copied to PC. Then the CPU moves back to fetch routine with the new address loaded.

JUMP1: MBRU ← IRAM [PC], fetch
JUMP2: PC ← PC+1; AC ← MBRU
JUMP3: AC ← AC<<8; MBRU ← IRAM [PC], fetch
JUMP4: PC ← AC+MBRU

JMPZ Instruction

If Z flag equals to 0, PC is incremented by 2 and the CPU move to fetch routine and start to fetch the next instruction to be executed.

If Z=1;
JMPZN1: PC ← PC+1
JMPZN2: PC ← PC+1

If Z flag equals to zero, four states are involved. In the 2nd and 3rd states, jump address stored in the instruction is loaded into AC. Then value of AC is copied to PC. Then the CPU moves back to fetch routine with the new address loaded.

If Z=1;
JUMP1: MBRU ← IRAM [PC], fetch
JUMP2: PC ← PC+1; AC ← MBRU
JUMP3: AC ← AC<<8; MBRU ← IRAM [PC], fetch
JUMP4: PC ← AC+MBRU

JMNZ Instruction

If z flag equals to 1, PC is incremented by 2 and the CPU move to fetch routine and start to fetch the next instruction to be executed.

If Z=1;
JMNZY1: PC ← PC+1
JMNZY2: PC ← PC+1

If z flag equals to zero, four states are involved. In the 2nd and 3rd states, jump address stored in the instruction is loaded into AC. Then value of AC is copied to PC. Then the CPU moves back to fetch routine with the new address loaded.

If Z=0;
JUMP1: MBRU ← IRAM [PC], fetch
JUMP2: PC ← PC+1; AC ← MBRU
JUMP3: AC ← AC<<8; MBRU ← IRAM [PC], fetch
JUMP4: PC ← AC+MBRU

INAC and DEAC Instructions

INAC1: AC ← AC+1
INAC Instruction involves the CPU to add 1 to the contents of AC and write back to AC.

DEAC1: AC ← AC-1

This instruction is straightforward which involves subtracting 1 from the contents of AC and writing back to AC.
After the states described above, the CPU moves to the fetch routine.

SUB and ADD Instructions

SUB1: AC ← AC-R
SUB instruction relates to subtracting the contents of R from AC and writing back to AC.

ADD1: AC ← AC+R
ADD instruction adds the contents of R to AC and writes back to AC.

Then the CPU moves to fetch routine and starts fetching the next instruction from the instruction memory.

ADDV Instruction

This instruction consists of five states which add a value resides in the immediate memory location in IRAM to AC and writes back to AC, i.e. the states involves the value in the memory location which is next to “ADDV” is added to AC and written back to AC.

ADDV1: R ← AC; MBRU ← IRAM [PC], fetch
ADDV2: PC ← PC+1; AC ← MBRU
ADDV3: AC ← AC<<8; MBRU ← IRAM [PC], fetch
ADDV4: PC ← PC+1; AC ← MBRU
ADDV5: AC←AC+R

SUBV Instruction

This instruction consists of 2 states which add a value resides in the immediate memory location in IRAM to AC and writes back to AC.

SUBV1: R ← AC; MBRU ← IRAM [PC], fetch

SUBV2: PC ← PC+1; AC ← MBRU
SUBV3: AC ← AC<<8; MBRU ← IRAM [PC], fetch
SUBV4: PC ← PC+1; AC ← AC+MBRU
SUBV5: AC←AC-R

The value in the AC register is subtracted from the value in the memory location which is next to “SUBV” and written back to AC.

DIV and MUL2, MUL4, MUL512 Instructions

These 2 instructions consist of one state which divides the contents of AC by 2 and multiplies by 4 then writes back to AC.
DIV1: AC ← AC>>1

MUL21: AC ← AC<<1
MUL41: AC ← AC<<2

MUL5121: AC ← AC<<8
MUL5122: AC ← AC<<1

INR1 and INR2 Instructions

These 2 instructions consist of one state which divides the contents of AC by 2 and multiplies by 4 then writes back to AC.

INR11 AC ← R1
INR12: AC ← AC+1
INR13: R1 ← AC

INR21: AC ← R2
INR22: AC ← AC+1
INR23: R2 ← AC

ADDR3 Instruction

ADDR31: AC, R3 ← AC + R3
This instruction adds the value stored in R3 to the value stored in AC and puts the new value in both AC and R3 register. This is an operation specific instruction for down-sample an image.

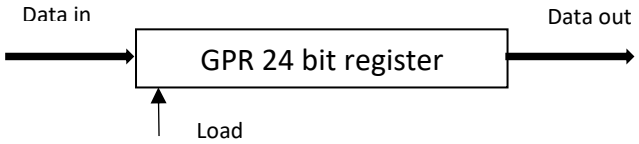
CLAC Instruction

CLAC1: AC ← 0; Z ← 1
The CLAC instruction can be executed by one state. It involves the CPU to clear the contents of the accumulator and start fetching the next instruction from the instruction memory.

3. MODULES

3.1. Registers

24 bit register



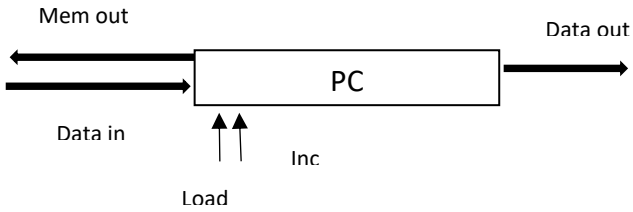
Block Diagram of 24 bit Register

The register modules are used to store data temporally during the process cycle. Each register can store 3 Bytes of data. Data stored in the register is always available at the data out and it is connected to de-multiplexer so that it can select which data should be read to the bus. These registers don't have any increment flag. If the stored values of these registers need to be incremented it has to go through an ALU increment operation and write back to the register. As in the figure this register has 16 bit input port and 16 bit output port. If the load flag is '1', it can write the data available in data in to the register at the positive edge of the clock.

Block Diagram of 24 bit Register with increment

Only difference is that this module contains an increment flag with in it. Therefore when the value of the register needed to be incremented by '1', it doesn't need to go through an ALU operation and write back. This can be done easily by enabling the increment flag of the register. Then at the positive edge of the clock if the increment flag is high value of that register will get incremented.

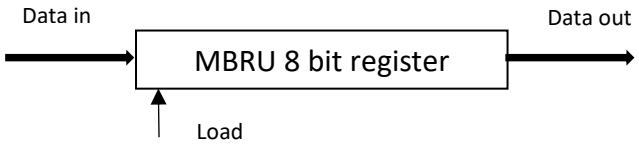
Programme Counter (PC)



The program counter keeps the address of the instruction to be executed. The size of the PC is 9 bits because the program memory has only 512 memory locations unlike the other registers having 8 bits

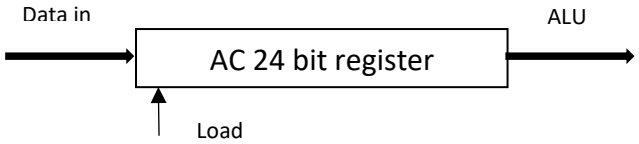
or 24 bits. This increment signal is used to increment the value of this register by “1” without needing to send its value through the ALU. If the PC inc is high at the positive edge of the clock, the register value is incremented. The PC is used in every fetch cycle, therefore the PC inc signal reduces the number of clock cycles.

Instruction Register (MBRU)



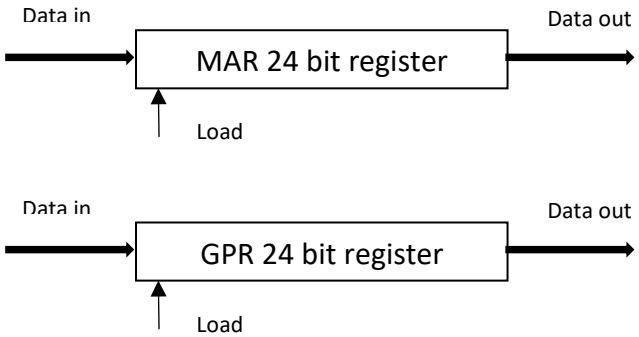
Instruction Register keeps instructions carried from the instruction memory pointed by the Programme Counter, at the positive edge of the clock if the read signal is high then it stores the instruction. Therefore IR is also slightly different from the other registers since it is the only register to have a read.

Accumulator (AC)



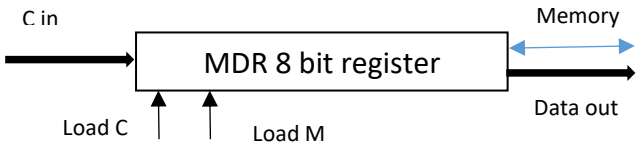
Accumulator is different from the other registers because its output is directly connected to the ALU as one of ALU’s inputs and AC has a clear command in order to reset its value. AC is used in almost all the ALU operations. The reset signal is used to reduce the number of clocks used and reduce the complexity of the software programme.

Memory Address Register (MAR) and General Purpose Registers.



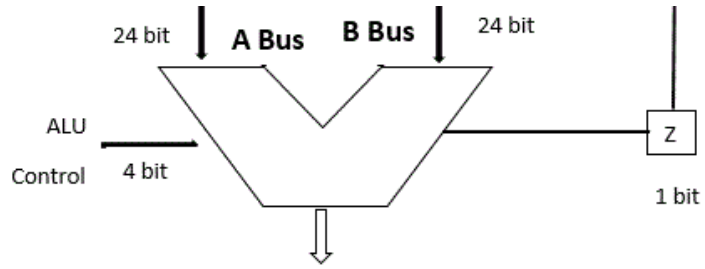
MAR and general purpose registers are 24 bit registers that has input from C bus whereas. All the registers output to the B bus module which is a multiplexer whereas MAR outputs are directly connected to the data memory as MAR points to the location of the data memory where we need to read/write and MDR contains the data that needs to be written to the RAM.

Memory Data Register (MDR)

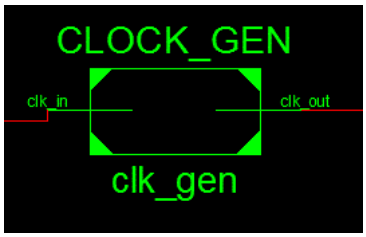


MDR is an 8 bit register. It has inputs from C bus and Data memory. Its output is connected to B bus and the Data memory. Its connections with the data memory is bi-directional.

3.3 ALU



3.4 CLOCK DIVIDER



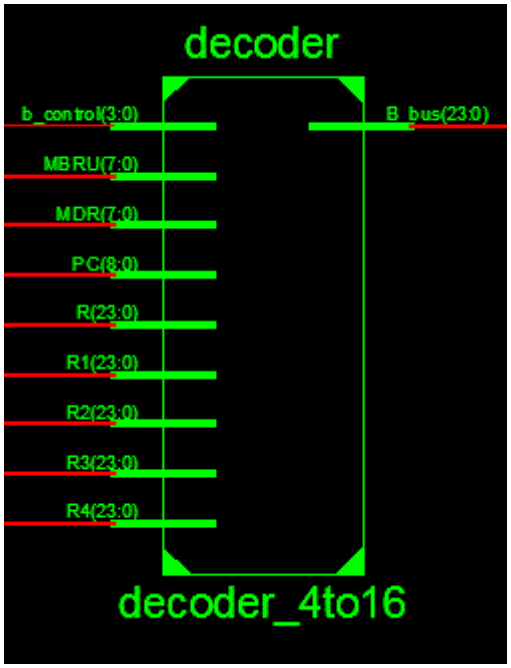
This module reduces the frequency of the original clock. There is an inbuilt 100MHz clock in the Spartan 6 FPGA development board. In one processing cycle we had to read the data from registers and perform arithmetic operations through ALU and output the values between the negative and positive edges of the each clock cycle. Therefore for a 100MHz clock this gap is 5ns. This is not sufficient to perform all the tasks required. Therefore it is necessary to increase the time gap between positive and negative edges of the clock. This module takes the original clock as the input and returns the divided clock (12.5MHz) as the output. All the components of the processor use this divided clock as their input.

3.2 CONTROL STORE

Control store decodes the instruction and releases the control signals in the design. This module is in the CISC design and the decoding of signals of the RISC design was discussed in the previous section.

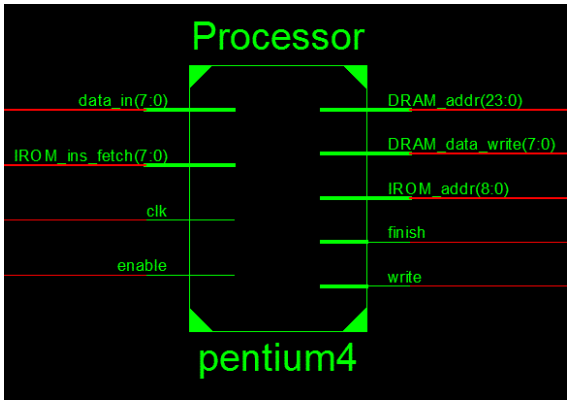
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1										C Bus write enable signal									Memory Signals			MBRU	MDR	PC	MDR,PC,MBRU,R1,R2,R3,R4,R	
2		Memory Address	Instruction	Microinstruction	Breakdown	Next_Addr (8)	JUMP	ADDR_SEL	ALU (4)	MAR	MDR	PC	R1	R2	R3	R4	R	AC	fetch	read	write	MBRULDIM	MDRLDDM	PCINC	B Bus read enable mux (4 to 16 decoder)	00000000_00_0000_000000000_000_0_0000
3	0	0	FETCH	FETCH1	MBRU <- Imem[pc]; fetch	1	0		0									0	1					0	10	00000001_00_0000_000000000_100_0_1010
4	1	1		FETCH2	PC <- PC+1	XXXXXXXX	0	1	0									0						1	10	XXXXXXXX_01_0000_000000000_000_1_1010
5	10	2	NOP	NOP1	No Operation	2	1		0									0						0	10	00000010_10_0000_000000000_000_0_0000
6	11	3	LDAC	LDAC1	MDR <- DRAM [MAR]; read	4	0		0									0		1				0	10	00000100_00_0000_000000000_010_0_0000
7	100	4		LDAC2	AC <- MDR	0	0		PASSBTOC									1						0	MDR	00000000_00_1000_000000001_000_0_0001
8	101	5	LDACV	LDACV1	MBRU ← IRAM [PC], fetch	6	0		0									0						0	10	00000110_00_0000_000000000_000_0_0000
9	110	6		LDACV2	PC <- PC+1; AC <- MBRU	7	0		PASSBTOC									1						1	MBRU	00000111_00_1000_000000001_000_1_0011
10	111	7		LDACV3	AC ← AC<<8; MBRU <- IRAM [PC], fetch	8	0		LSHFT8									1	1					0	10	00001000_00_0101_000000001_100_0_0000
11	1000	8		LDACV4	PC ← PC+1; AC <- AC+MBRU	0	0		ADD									1						1	MBRU	00000000_00_0001_000000001_000_1_0011
12	1001	9	STAC	STAC1	MDR <- AC	10	0		PASSATOC		1							0						0	10	00001010_00_0111_010000000_000_0_0000
13	1010	10		STAC2	DRAM[MAR] <- MDR; write	0	0		0									0			1			0	10	00000000_00_0000_000000000_001_0_0000
14	1011	11	MVACR1	MVACR11	R1 <- AC	0	0		PASSATOC				1					0						0	10	00000000_00_0111_000100000_000_0_0000
15	1100	12	MVACR2	MVACR21	R2 <- AC	0	0		PASSATOC					1				0						0	10	00000000_00_0111_000010000_000_0_0000
16	1101	13	MVACR3	MVACR31	R3 <- AC	0	0		PASSATOC						1			0						0	10	00000000_00_0111_000001000_000_0_0000
17	1110	14	MVACR4	MVACR41	R4 <- AC	0	0		PASSATOC							1		0						0	10	00000000_00_0111_000000100_000_0_0000
18	1111	15	MVACR	MVACR1	R <- AC	0	0		PASSATOC								1	0						0	10	00000000_00_0111_000000010_000_0_0000
19	10000	16	MVACMAR	MVACMAR1	MAR <- AC	0	0		PASSATOC	1								0						0	10	00000000_00_0111_100000000_000_0_0000
20	10001	17	MOVR1	MOVR11	AC <- R1	0	0		PASSBTOC									1						0	R1	00000000_00_1000_000000001_000_0_0100
21	10010	18	MOVR2	MOVR21	AC <- R2	0	0		PASSBTOC									1						0	R2	00000000_00_1000_000000001_000_0_0101
22	10011	19	MOVR3	MOVR31	AC <- R3	0	0		PASSBTOC									1						0	R3	00000000_00_1000_000000001_000_0_0110
23	10100	20	MOVR4	MOVR41	AC <- R4	0	0		PASSBTOC									1						0	R4	00000000_00_1000_000000001_000_0_0111
24	10101	21	MOVR	MOVR1	AC <- R	0	0		PASSBTOC									1						0	R	00000000_00_1000_000000001_000_0_1000
25	10110	22	*JMNZ*	*JMNZ1*	*JUMP*	XXXXXXXX	0		*0*									*0*						*0*	*10*	*XXXXXXXX_00_0000_000000000_000_0_0000*
26	10111	23		JMNZY1	PC <- PC+1	24	0		0									0						1	10	00011000_00_0000_000000000_000_0_0000
27	11000	24		JMNZY2	PC <- PC+1	0	0		0									0						0	10	00000000_00_0000_000000000_000_0_0000
28	11001	25		JMNZN1	MBRU ← IRAM [PC], fetch	26	0		0									0	1					0	10	00011010_00_0000_000000000_100_0_0000
29	11010	26		JMNZN2	PC ← PC+1; AC <- MBRU	27	0		PASSBTOC									1						1	MBRU	00011011_00_1000_000000001_000_1_0011
30	11011	27		JMNZN3	AC ← AC<<8; MBRU <- IRAM [PC], fetch	28	0		LSHFT8									1	1					0	10	00011100_00_0101_000000001_100_0_0000
31	11100	28		JMNZN4	PC ← AC+MBRU	0	0		ADD			1						0						0	MBRU	00000000_00_0001_001000000_000_0_0011
32	11101	29	INAC	INAC1	AC <- AC+1	0	0		INAC									1						0	10	00000000_00_1001_000000001_000_0_0000

3.5 DEMULTIPLEXER



In the processor architecture there is only one data bus. Therefore it can only read data from one register at a time to the bus. This implementation has been done by using a de-multiplexer. There are few de-multiplexers used inside the processor module as well as outside the processor module. Inside the processor module there are two multiplexers. One of that mux connected to all registers (MAR, PC, R1, R2, R3 etc.). Other side of the de-mux connected to another multiplexer. That mux is connected to RAM module and C bus. Therefore this configuration allows to read data from RAM or ALU output to any the registers and pass data through other way.

3.6 PROCESSOR



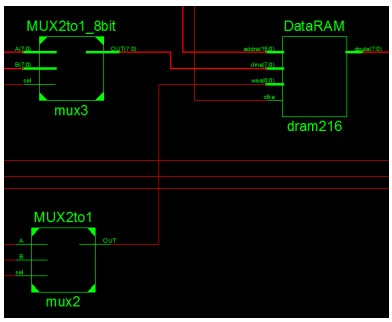
The processor module contains the all the instances of the modules used for the processing part. This does not contain instances of memory modules and communication modules. This has four inputs which is clock, processor enable, DRAM, and IROM.

- Clock - this gives clock pulse for the synchronization
- Enable – this is used to enables the module
- DRAM – this gives image data to the processor from the Data RAM (8 bit)
- IROM – this gives IROM data to the processor (8 bit)

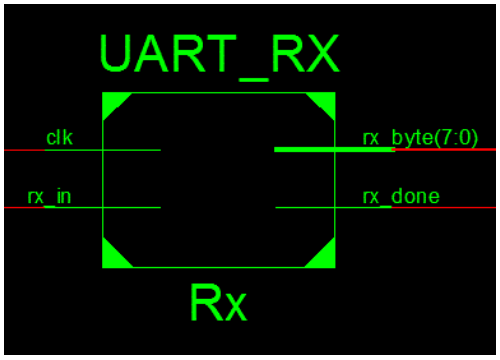
This module has five output data paths which are write, finish, DRAM_addr, IROM_addr and DRAM_data_Write

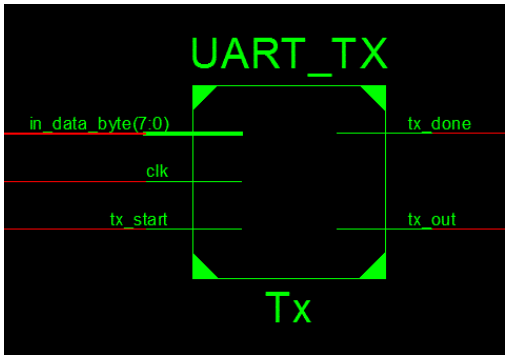
- write -Write enable signal
- finish -Indicate the end of the processing to the IO module
- DRAM_addr - gives memory location of the DRAM (24 bit)
- IROM_addr -gives memory location of the IRAM (9 bit)
- DRAM_data_write -gives the data which needs to be written in to the DRAM (8 bit)

3.7 COMMUNICATION RELATED MODULES



The above mux configuration changes the routing configuration between the processor and the UART modules so that when the trancieving with the computer takes place, the memory interacts with the UART modules and when the processing takes place, the processor gets to interact with the memory modules. This could have been done easily using a dual port RAM but unfortunately we have already done the design using single port RAM. Therefore, we had to use the mux configuration shown above to change the memory module interation with the processor and the UART modules accordingly.



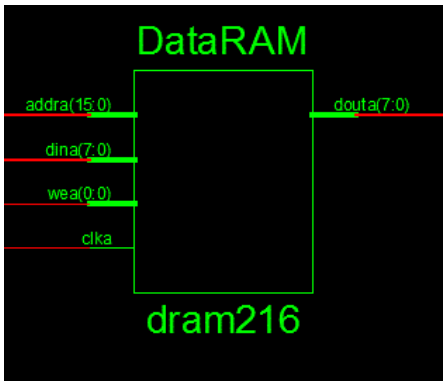


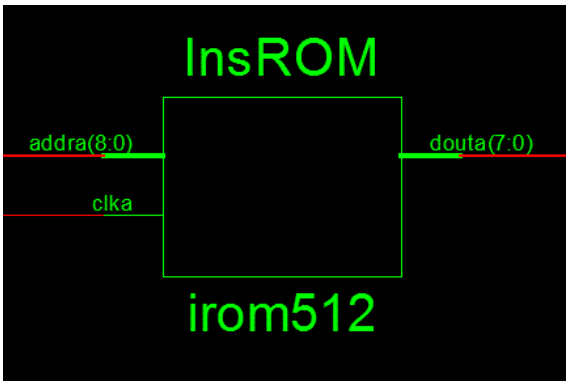
The receiver UART module interacts with the computer in the input side and the output side is connected to the memory module to store the result. Actually, the rx_done Boolean output is sent to IO_controller in order to save the received byte to the memory. Rx_byte is directly connected to the data memory module.

This module was used to receive the serial data and output one byte (8 bits) at a time. We used 'rx_in' pin to receive the data bit by bit and when it receives 8 bits, it generates 8 bit width word and returns it. Then the same time setting the output of the 'rx_done_tick' is high.

UART_TX modules works in the same way but input output interaction is changed. This module is used to transmit the pixel data from the FPGA to the computer via serial data communication. It has three inputs and two outputs. We have to set the 'tx_start' pin as high to stat the module. The "in_data_byte" input takes the 8 bit word at a time and transmits one bit by bit via 'tx_out' output pin. When it finishes transmitting all the 8 bits of a particular byte it sets the 'tx_done' output as high.

3.8 MEMORY MODULES





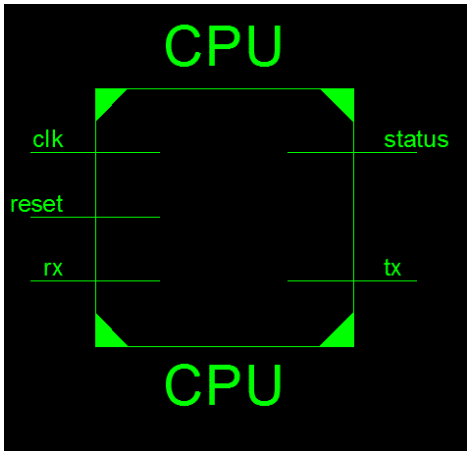
Instruction memory is a read only memory and the data memory is a random access memory. Both these memory modules are made with the use of memory avaiable in the FPGA using the

software available without writing code. The data and instruction memories are single port memories. The design would have been better if dual port memories have used.

InsROM module is used to store instruction in the memory. Instructions are coded by assembly language. Whenever the processor needs instructions it fetches instructions from this instruction memory (InsROM). This module consists of instructions to be executed with 512 memory locations with 9 bits width. This memory contains the assembly code of the algorithm for filtering and down sampling the image. This module has only two inputs and one data output. Inputs are used to feed the clock signal into the module and other input is used to input the memory location of the instruction. The output of the module has 8 bit width. We have instructions below 256. Therefore 8 bit width data path is sufficient to give the output signal what we need.

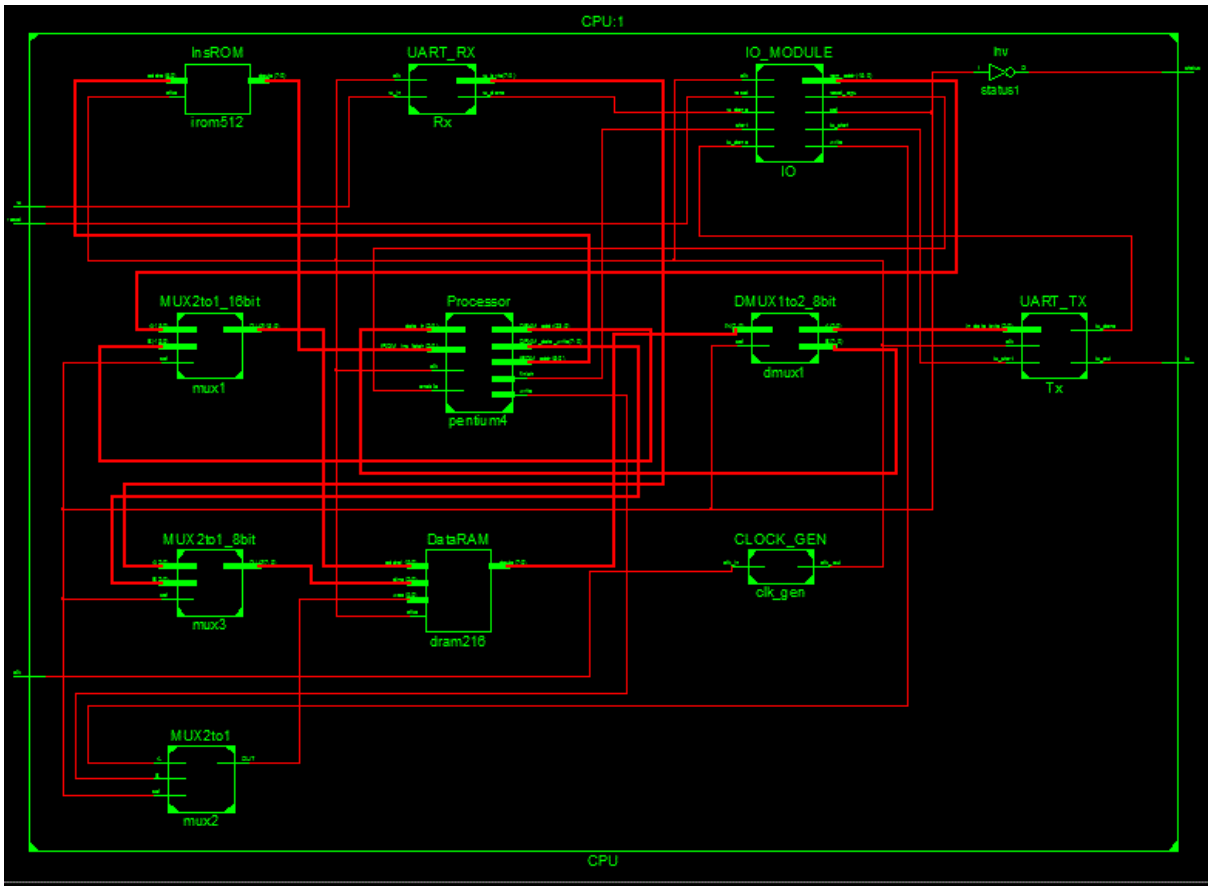
DataRAM module is the main data memory which is used for store data of the image and store the processed image data. First read the data from the UART-Rx module and all the data units are saved in the dram module for further operations. This module has 65536 memory locations of 8 bit width to store the pixels of the image. Every pixel has integer values which is in the range of 0 to 255. Therefore 8 bit width of a memory unit enough to store the pixels in dram module. This ram module has four input data paths and one output data path. “Din” input is used to give input data which has 8 bit data stream and “address” path gives the address of the memory location which the dram has to read or write data. This input contains a 16 bit data stream because the dram module has 65536 memory locations. “wea” is a one input which has one bit, is used write data to the memory from dina data path. There is a “clka” input which is used to input the clock signal which is given by the clock_gen module. “douta” is the output of this module. It is used to give output data to the data bus when request from the control unit.

3.9 TOP MODULE

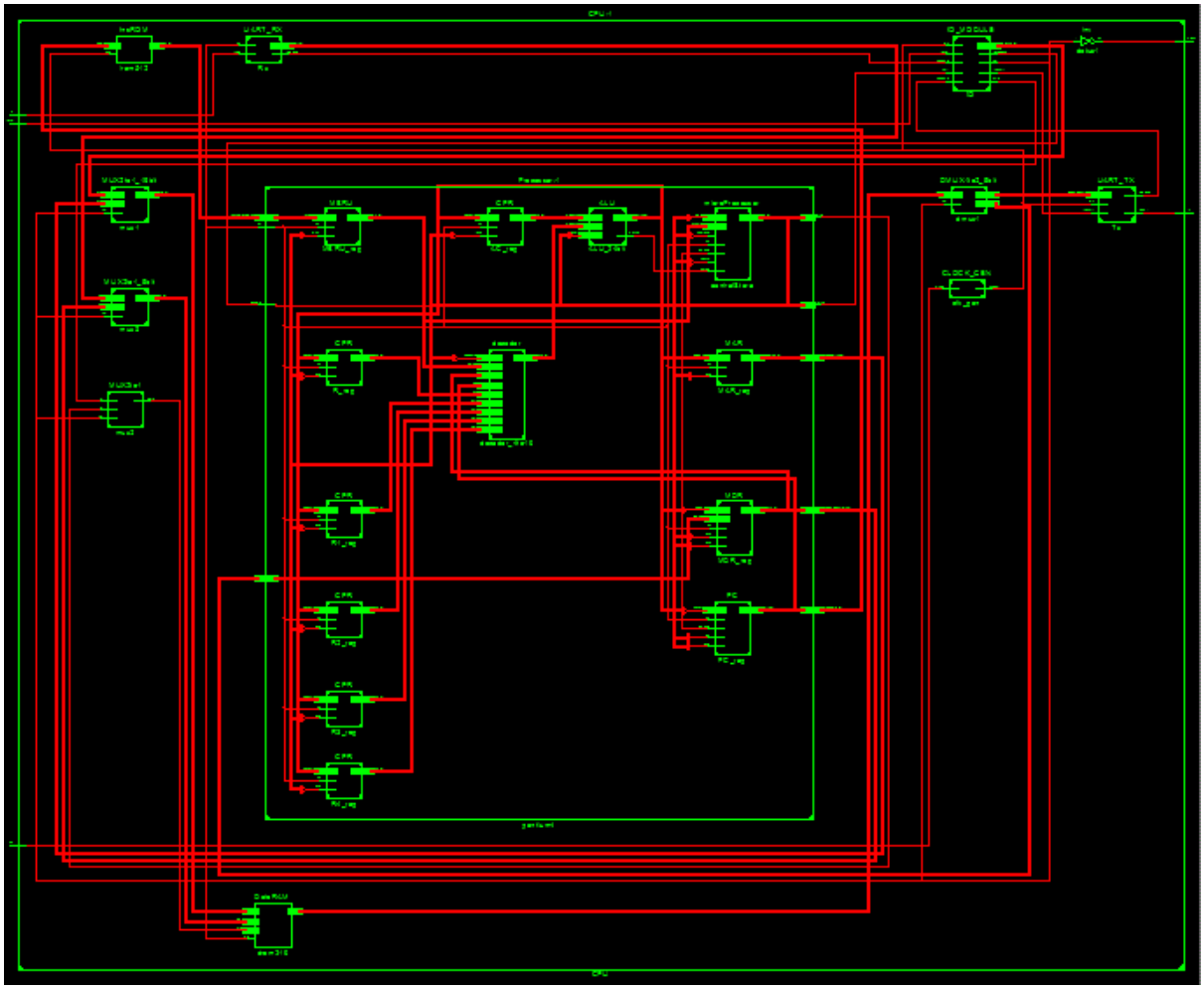


This module is used to connect processing and communicating modules. All the instances have been created inside this module. This module used a clock signal and a reset pin as its inputs. Also it uses the 'rx' pin for receiving the data coming serially. This has one output which is 'tx' to transmit the data serially. This contains instances of Processor, UART_RX, UART_TX and IO Module. The following figure shows the internal connected structure of the top module.

3.10 RTL VIEW OF TOP MODULE



3.11 RTL VIEW OF THE TOP PROCESSOR



The following diagram shows the structure of the processor shown in the data path diagram in the previous section.

4. ALGORITHM

ALGORITHM

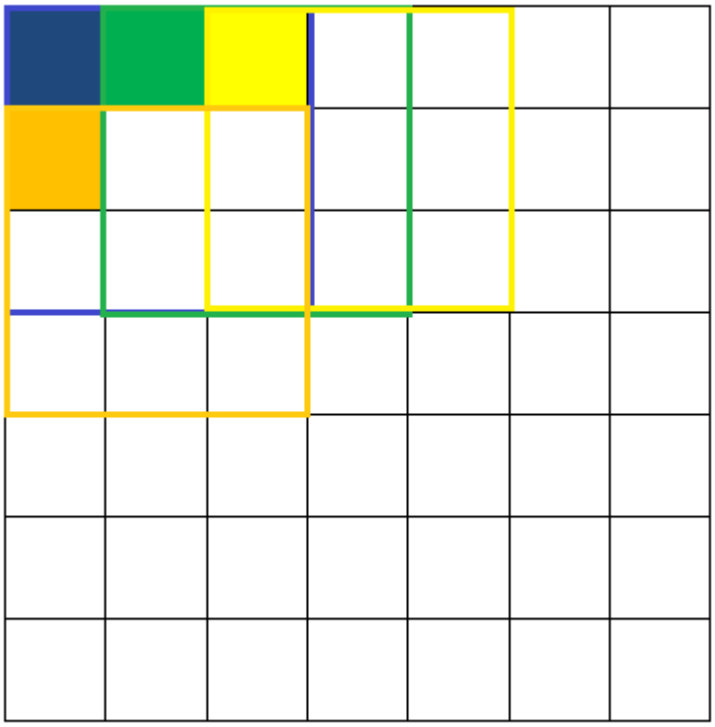
The algorithm for down sampling an image consists of two main parts. First phase of the algorithm is smoothing the image using a Gaussian Filter. Then the filtered image is down-sampled using a down sampling algorithm.

FILTERING ALGORITHM

For filtering the image, a 3x3 kernel is used. This kernel is weighted such that the kernel is symmetric in all directions (i.e. Gaussian). This kernel is shown in the figure below.

1	2	1
2	4	2
1	2	1

3x3 Gaussian Kernel

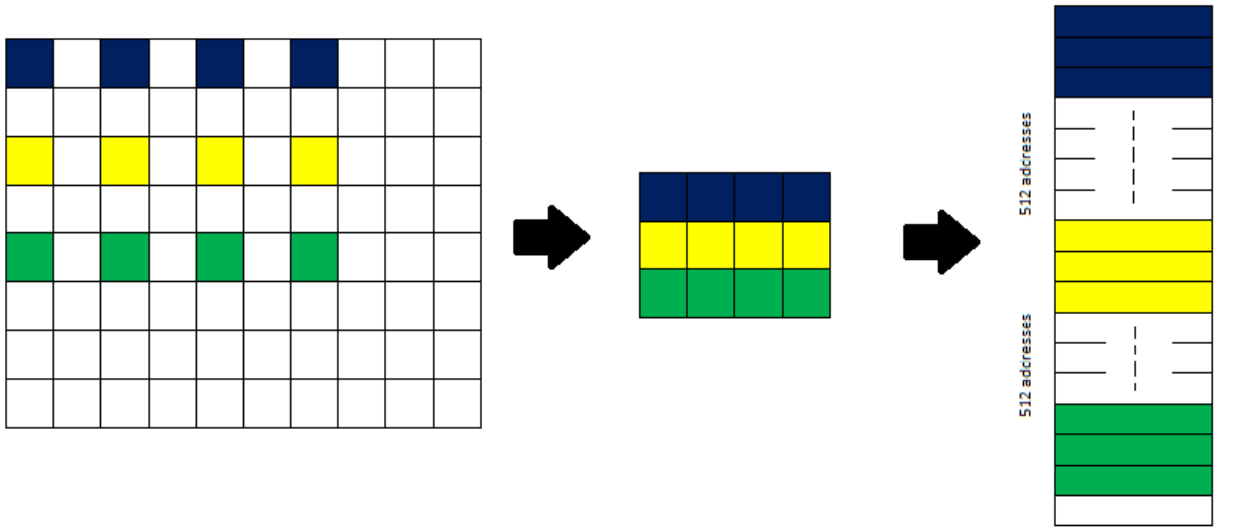


This kernel is initiated at the top left corner of the image and then traversed throughout the image horizontally. While the kernel is traversed, the value of the pixel values averaged using the overlapping weights of the kernel is stored at the top left corner pixel of the pixel block overlapping that kernel at the time. This pixel corresponds to the assigned RAM location. The following diagram describes the motion of the kernel and the location (hypothetical) of data storage.

The coloured pixel contains the weighted average sum of the pixel block squared by the similar colour.

DOWN SAMPLING ALGORITHM

After smoothing, the image is down sampled to the ratio 1:2. For this a pixel from each noon overlapping block of four pixels is taken and stored as the desired down sampled image. The following figure shows the (hypothetical) storage of the down sampled image.



Original Image Down Sampled Image Memory Allocation

As shown in the diagram, adjacent pixels vertically below are spaced 512 addresses. This is so because although the down sampled image is depicted in a 2D array, in memory it is stored as a 1D array.

```
% Read and map the 2D image to a 1D memory array
close all;
image = imread('C:\Users\adhit\Desktop\Processor Design\Matlab Code\Emacs_512.png'); %
Read the image and save 2D array
image = rgb2gray(image);
memory_array = uint16(image(:)); % Make 1D array, convert to 16 bit since registers are 16
imshow(image);                    % Display Image
imwrite(image, 'C:\Users\adhit\Desktop\Processor Design\Matlab Code\gray.png');     %
Write gray scale image
% Gaussian filter the image
total = 0;                % Initialize the totoal variable

for j = 1:1:510           % Loop through rows
   for i = 2:1:511        % Loop through columns
      x = j*512+i;
      % Convolve with the gaussian filter
      total = total + memory_array(x-1)*2;
      total = total + memory_array(x)*4;
      total = total + memory_array(x+1)*2;
      total = total + memory_array(x+513);
      total = total + memory_array(x+512)*2;
      total = total + memory_array(x+511);
      total = total + memory_array(x-513);
      total = total + memory_array(x-512)*2;
      total = total + memory_array(x-511);
      total = total/16;                  % Normalize the total value
      memory_array(x-513) = total;          % Store the filtered value in the memory
      total=0;
   end
end

figure, memory_array = uint8(memory_array);
filtered_image = memory_array; % Convert to uint8 format
c = reshape(filtered_image,512,512);  % Generate 2d array from vector
imshow(c);                        % Display the filtered image

% Downsample the filtered image
k = 1;                                % Set writing memory address
for j = 0:1:255                       % loop going through rows
   for i = 0:1:255                    % loop going through columns
      y = 2*512*j + 2*i + 1;              % Map every other pixel to RAM address in memory
      memory_array(k) = memory_array(y);      % Overwrite the pixel values with every other
      k = k+1;                        % Increment writing memory address
   end
end

figure, downsampled_image = memory_array(1:65536);      % Bytes relevant to 256*256
c = reshape(downsampled_image,256,256); %         Reshape the 1D array to an image
imshow(c); % Display the filted image
```

Original Image 512 x 512 size

Filtered Image 512 x 512 size

Down-sampled image 256 x 256 size

Published with MATLAB® R2014b

Assembly Code of the Algorithm

Shown below is the assembly code used to run in the CISC processor to down-sample the image. Since we used an interpreter to convert the assembly code to the machine code, writing the assembly code was much easier and the conversion to machine code every time there was a change in the assembly code saved a lot of time.

```
CLAC
MVACR1
MVACR2

L1

        CLAC
        MVACR3

        // Calculation of the first pixel value
        MOVR2
        MUL4
        MUL2
        MVACR
        MOVR1
        ADD

        // First pixel
        MVACR4
        MVACMAR
        LDAC
```

```
MVACR3

// Second pixel
MOVR4
INAC
MVACMAR
LDAC
MUL2
ADDR3

MOVR4
INAC
INAC
MVACMAR
LDAC
ADDR3

// Go to the second pixel line
MOVR4
ADDV 8
MVACR4
MVACMAR
LDAC
MUL2
ADDR3

// Multiply middle pixel by 4
MOVR4
INAC
MVACMAR
LDAC
MUL4
ADDR3

MOVR4
INAC
INAC
MVACMAR
LDAC
MUL2
ADDR3

// Go to the third pixel line
MOVR4
ADDV 8
MVACR4
MVACMAR
LDAC
ADDR3

MOVR4
```

```
        INAC
        MVACMAR
        LDAC
        MUL2
        ADDR3

        MOVR4
        INAC
        INAC
        MVACMAR
        LDAC
        ADDR3

        // Calculate the pixel store address
        MOVR2
        MUL4
        MUL2
        MVACR
        MOVR1
        ADD

        MVACMAR

        // Calculate the final convolutio value and store it
        MOVR3
        DIV
        STAC

        INR1
        SUBV 6

        JMNZ L1

        CLAC
        MVACR1

        INR2
        SUBV 6

        JMNZ L1


// Choose pixels

CLAC
MVACR1
MVACR2
MVACR3

L2
```

```
        MOVR2

        // Multiply by 512, Image_size x 2, 16
        MUL4
        MUL4
        MVACR

        MOVR1
        MUL2
        ADD

        MVACMAR
        LDAC
        MVACR

        MOVR3
        MVACMAR
        MOVR
        STAC

        MOVR3
        INAC
        MVACR3

        INR1

        // Substract Image_size/2
        SUBV 4

        JMNZ L2

        CLAC
        MVACR1
        INR2

        // Substract Image_size/2
        SUBV 4

        JMNZ L2

FINISH
NOP
```

Code of the interpreter – Conversion of the Assembly code to the machine code

This code was written in python to generate the binary machine code of the above shown assembly code. This setting makes life easy so that the software programmer does not have to worry about translating the programme line by line to the machine code.

```
import os

UINS = {
        'FETCH'    :'0',
        'FINISH'   :'2',
        'LDAC'     :'3',
        'LDACV'    :'5',
        'STAC'     :'9',
        'MVACR1'   :'11',
        'MVACR2'   :'12',
        'MVACR3'   :'13',
        'MVACR4'   :'14',
        'MVACR'    :'15',
        'MVACMAR'  :'16',
        'MOVR1'    :'17',
        'MOVR2'    :'18',
        'MOVR3'    :'19',
        'MOVR4'    :'20',
        'MOVR'     :'21',
        'JMNZ'     :'22',
        'JMNZY'    :'23',
        'JMNZN'    :'25',
        'INAC'     :'29',
        'DEAC'     :'30',
        'ADD'      :'31',
        'SUB'      :'32',
        'ADDV'     :'33',
        'SUBV'     :'38',
        'DIV'      :'43',
        'MUL2'     :'44',
        'MUL4'     :'45',
        'MUL512'   :'46',
        'INR1'     :'48',
        'INR2'     :'51',
        'CLAC'     :'54',
        'ADDR3'    :'55',
        'NOP'      :'57'
    }


def Translate(filename):

    a_code = []
    m_code = []
    jumps = {}
```

```
    n = 0
    with open(filename) as fp:
        for line in fp:
            x = line.strip()
            if len(x) > 0:
                if not (x[0] == '/' and x[1] == '/'):
                    if (x[0] == 'L' and x[1] != 'D'):
                        jumps.update({x: str(n)})
                    else:
                        a_code.append(x)
                        if(len(x) > 3 and x[0] == 'A' and x[1] == 'D' and x[2] == 'D' and x[3] == 'V'):       #ADDV
instruction
                            n = n+3
                            #print n
                        elif(len(x) > 3 and x[0] == 'S' and x[1] == 'U' and x[2] == 'B' and x[3] == 'V'):       #SUBV
instruction
                            n = n+3
                            #print n
                        elif(len(x) > 3 and x[0] == 'J' and x[1] == 'M' and x[2] == 'N' and x[3] == 'Z'):       #SUBV
instruction
                            n = n+3
                            #print n
                        else:
                            n = n+1

    for x in range(0, len(a_code)):
        if "//" in a_code[x]:
            t = a_code[x].split("//")
            a_code[x] = t[0].strip();

    print a_code
    print jumps

    for x in range(0, len(a_code)):

        if(a_code[x] in UINS.keys()):
            m_code.append(UINS[a_code[x]])

        elif(a_code[x][0] == 'J' and a_code[x][1] == 'M' and a_code[x][2] == 'N'):       #JMNZ instruction
            temp = a_code[x].split(' ')
            m_code.append(UINS[temp[0]])
            m_code.append( str(int(jumps[temp[1]])/256) )
            m_code.append( str(int(jumps[temp[1]])%256) )

        elif(a_code[x][0] == 'A' and a_code[x][1] == 'D' and a_code[x][2] == 'D'):       #ADDV instruction
            temp = a_code[x].split(' ')
            m_code.append(UINS[temp[0]])
            m_code.append(str(int(temp[1])/256))
            m_code.append(str(int(temp[1])%256))
```

```
        elif(a_code[x][0] == 'S' and a_code[x][1] == 'U' and a_code[x][2] == 'B'):       #ADDV instruction
            temp = a_code[x].split(' ')
            print temp
            m_code.append(UINS[temp[0]])
            m_code.append(str(int(temp[1])/256))
            m_code.append(str(int(temp[1])%256))

    print m_code

    # output machine code to .mcode file

    out = 'memory_initialization_radix=10;\nmemory_initialization_vector=\n'
    for x in m_code:
        out += x + ",\n"

    out = out[:-2] + ';';
    out = out.strip()

    f = filename.split('.')
    outname = f[0] + ".coe"

    file = open(outname, 'w')
    file.write(out)
    file.close();


path = os.path.dirname(os.path.realpath(__file__))

for file in os.listdir(path):
    if file.endswith(".navo"):
        Translate(os.path.join(path, file))
```

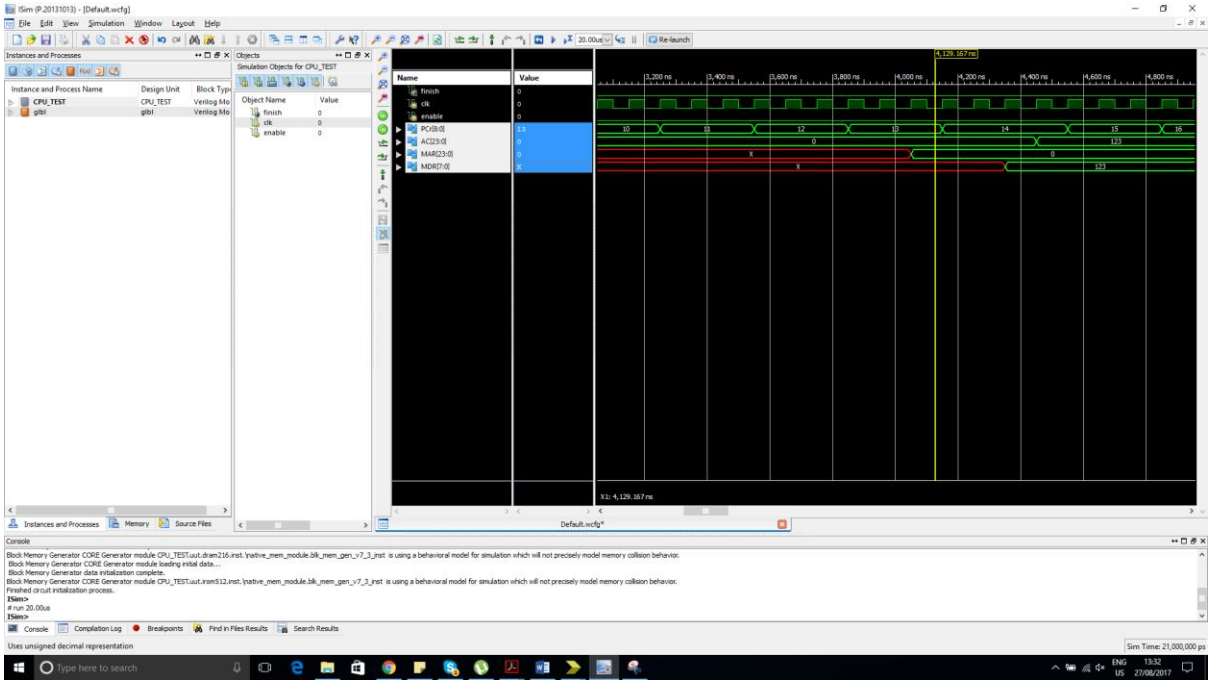
5. TESTING, SIMULATION AND MODIFICATION

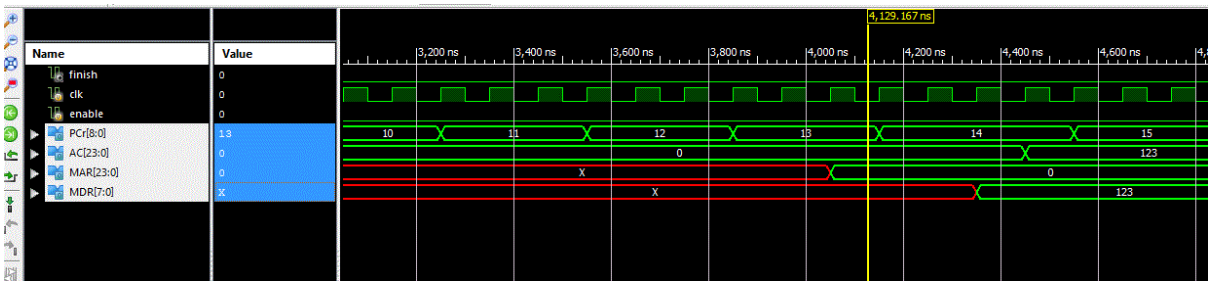
5.1 TESTING AND SIMULATION

To test and simulate the modules individually and as one unit, we write different Verilog modules for the simulation. We test each and every module we write. Below is the code we wrote to test the final CPU module. Test simulation results gives the intended output justifying the operation of the processor.

```
module CPU_TEST;

	// Inputs
	reg clk;
	reg enable;

	// Outputs
	wire finish;

	// Instantiate the Unit Under Test (UUT)
	CPU uut (
		.enable(enable),
		.clk(clk),
		.finish(finish)
	);

	initial begin
		// Initialize Inputs
		clk = 0;
		enable = 0;

		// Wait 100 ns for global reset to finish
		#200;

		// Add stimulus here
		enable = 1;
		#50
		enable = 0;

	end

	always begin
		#50
		clk = ~clk;

	end

endmodule
```

We run the simulation in iSim simulator and see the irregularities and changes that should be done in the algorithm. We use an 8x8 image to justify the operation of the 2 processors we made.

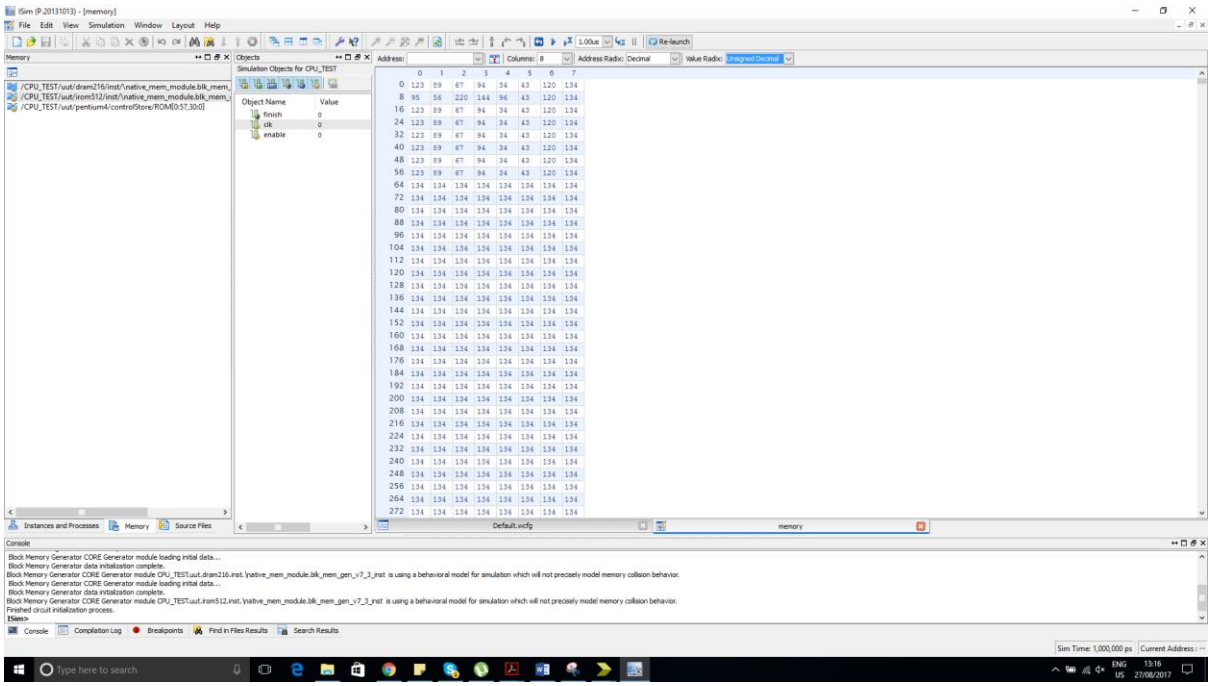




We use the values shown in PC, MAR, MDR and AC mainly for the justification and bugging and debugging. We see that the changes in the specified registers are as expected after debugging and correctly implementing the programme.

We used an 8x8 image to test and verify the down-sampling process of the algorithm. Below image shows the simulation results of the algorithm. Same simulation process is carried out in both the processors. Therefore, only one simulation corresponding to one processor is shown in the following steps.

Before processing the algorithm



	0	1	2	3	4	5	6	7
0	123	89	67	94	34	43	120	134
8	95	56	220	144	96	43	120	134
16	123	89	67	94	34	43	120	134
24	123	89	67	94	34	43	120	134
32	123	89	67	94	34	43	120	134
40	123	89	67	94	34	43	120	134
48	123	89	67	94	34	43	120	134
56	123	89	67	94	34	43	120	134
64	134	134	134	134	134	134	134	134
72	134	134	134	134	134	134	134	134

8x8 image before any processing takes place

	0	1	2	3	4	5	6	7
0	99	119	111	94	34	43	120	134
8	95	56	220	144	96	43	120	134
16	123	89	67	94	34	43	120	134
24	123	89	67	94	34	43	120	134
32	123	89	67	94	34	43	120	134
40	123	89	67	94	34	43	120	134
48	123	89	67	94	34	43	120	134
56	123	89	67	94	34	43	120	134
64	134	134	134	134	134	134	134	134

First 3 slots are changed with the application of the Gaussian filtering. It is seen that it gives the correct Gaussian filtered pixel result.

	0	1	2	3	4	5	6	7
0	99	119	111	73	67	104	120	134
8	95	99	91	62	63	104	120	134
16	92	79	72	51	60	104	120	134
24	92	79	72	51	60	104	120	134
32	92	79	72	51	60	104	120	134
40	92	79	72	51	60	104	120	134
48	123	89	67	94	34	43	120	134
56	123	89	67	94	34	43	120	134

Now the Gaussian filtering process has ended giving the correct Gaussian filtered image as the result. You can see the Gaussian filtered image in the square corresponding to 6x6 region. 0,0 value corresponds to the 1,1 pixel. We used this mechanism in order to keep the memory usage as low as possible and to keep the process as simple as possible.

	0	1	2	3	4	5	6	7
0	99	111	67	120	92	72	60	120
8	92	72	60	120	123	67	34	120
16	92	79	72	51	60	104	120	134
24	92	79	72	51	60	104	120	134
32	92	79	72	51	60	104	120	134
40	92	79	72	51	60	104	120	134
48	123	89	67	94	34	43	120	134
56	123	89	67	94	34	43	120	134

The above image shows the final 4x4 image produced by the complete process.

By running the simulation, we see that the number of clock cycles taken in the RISC processor is 4 times lower than the CISC processor but the resource usage of the CISC processor is much lower than the RISC processor.

5.2 MODIFICATIONS

1) We designed the processor such that it gave the correct results with the simulation but when we implemented the system using the memory generator in ISE the design started to behave in a bizarre manner. This was due to our implementation in the RAM module and the implementation in the memory generator RAM module. We wrote our RAM modules such that it gives the output when the input changes but in their memory implementation, they output the data at the positive edge of the clock cycle, so we had to add some clocks in our microprocessor design in order to make available the data at the output.
2) In the CISC design, we had to change the microinstruction a few times in order to get the desired output.
3) In the RISC design, since the number of changes we have to make in order to make the instruction RAM compatible with the memory block generated by the block memory generator, we used the memory block we wrote since that was convenient at the moment because it would not affect the performance of the processor.
4) In the CISC design, some of the microinstruction could be run in parallel. We changed the control store instructions such that those instruction could be run in parallel to reduce the number of clock cycles taken to complete the given task or rather any task run by the processor.
5) We modified the transmission code in python and the transmission speed, i.e, baud rate of the IO module in order to obtain a quick result. The processing time taken to down-sample the image was very low compared to the transmission time from the computer to FPGA and FPGA to the computer. Therefore, we carefully changed the speed of the IO module so that we can obtain the image quicker, this change we did to our own standard unlike the usual bit transmission speed used like 9600.
6) We tried to do the down-sampling to a 512x512 image but the resources available in Spartan 6 board was not enough to cater the memory needs of the system. One way to overcome this problem was to cut the original 512x512 image into 4 pieces such that each piece is 256x256 and transmit those 4 images separately to the FPGA and then receive the image iteratively and combine the final result in the computer. But we did not implement the design as such because then there will be an error at the edged when we stich the 4 received images from the FPGA and that would be an iteration of the same process. Therefore, we did it to an image with size 256x256. Both the 2 processors we designed are capable of down-sampling a 512x512 image because the register size is 24bit but this could not be done because of the memory constraint.

6. RESULT ANALYZING AND VERIFICATION

The error analysis plays an important role in the verification process since this is the final step verifying the operation of the processor implemented in FPGA. We do this in comparison with Gaussian filtered image and down-sampled image in OpenCV and the received image from the FPGA. We deduct one image from the other image and take the absolute value and then add all the absolute values in order to arrive at a final figure as the error value. We see that the received image from the RISC based processor yields zero absolute error while the CISC processor yields a much higher absolute error since it gives an image that is slanted.

The steps of the verification process and the resulting images are shown and discussed in the following sections.

6.1 GENERATE REFERENCE OUTPUT IMAGE

The following diagram shows the steps involved in the generation of the reference image.

Input image -> Gaussian Filter -> Down-sampled Image -> Comparison with the received image

6.2 RESULTS VERIFICATION AND ANALYSIS

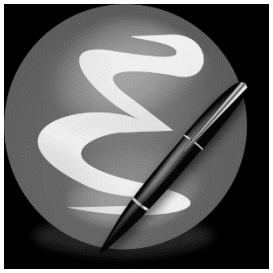
This section shows a comparison between the images taken from the FPGA based processor and the image obtained using MATLAB/ OpenCV in Python.

Results from the RISC based Processor

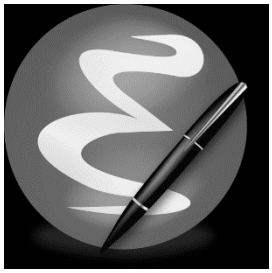
Original Image

Reference Gaussian filtered image from OpenCV

Filtered Image obtained from the FPGA



Down sampled image from OpenCV



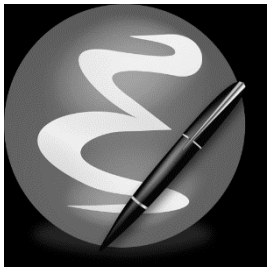
Down sampled image obtained from the FPGA

We used integer processing in the same algorithm we implemented using OpenCV. This processor gave no error at all. Total time taken for the completion of the task was around 7 seconds with a transmission bit rate of 200000. We did not use the standard transmission bit rates in order to obtain a maximum speed.

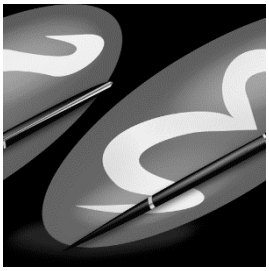
We checked the accuracy of the RISC based processor a few times and it gave no absolute error every time we checked.

Results from the CISC based Processor

Original Image



Down-sampled image using OpenCV



Down-sampled Image obtained using the FPGA

The CISC based processor produced an image shown as above every time we run the process. This may have happened due to a wrong implementation of the assembly code algorithm and hence producing an image with wrong indexes. We did not try our best to correct this at the last moment since the other processor gave 100% accurate results, 3 of the group members were to go abroad for their internship programme and because of that there was not enough time to debug and correct it.

6.3 SUMMARY REPORT

It is noticed that the resource usage in the RISC processor is twice as the resource usage in the CISC processor.

ROOT_SYSTEM Project Status			
Project File:	PROCESSOR.xise	**Parser Errors:**	No Errors
Module Name:	ROOT_SYSTEM	**Implementation State:**	Programming File Generated
Target Device:	xc6slx45-3csg324	• **Errors:**	
Product Version:	ISE 14.7	• **Warnings:**	
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	442	54,576	1%	
Number used as Flip Flops	370			
Number used as Latches	72			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	830	27,288	3%	
Number used as logic	820	27,288	3%	
Number using O6 output only	601			
Number using O5 output only	68			
Number using O5 and O6	151			
Number used as ROM	0			
Number used as Memory	0	6,408	0%	
Number used exclusively as route-thrus	10			
Number with same-slice register load	4			
Number with same-slice carry load	6			
Number with other load	0			
Number of occupied Slices	396	6,822	5%	
Number of MUXCYs used	220	13,644	1%	
Number of LUT Flip Flop pairs used	1,042			
Number with an unused Flip Flop	626	1,042	60%	
Number with an unused LUT	212	1,042	20%	
Number of fully used LUT-FF pairs	204	1,042	19%	
Number of unique control sets	38			
Number of slice register sites lost to control set restrictions	118	54,576	1%	
Number of bonded IOBs	5	218	2%	
Number of LOCed IOBs	5	5	100%	
Number of RAMB16BWERs	32	116	27%	
Number of RAMB8BWERs	0	232	0%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	4	16	25%	
Number used as BUFGs	4			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	8	0%	
Number of ILOGIC2/ISERDES2s	0	376	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	376	0%	
Number of OLOGIC2/OSERDES2s	0	376	0%	
Number of BSCANs	0	4	0%	
Number of BUFHs	0	256	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	0	58	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	0	4	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	4.22			

Performance Summary				[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report	
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report	
Timing Constraints:	All Constraints Met			

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Mon 8. May 20:59:25 2017	0	213 Warnings (2 new)	4 Infos (0 new)
Translation Report	Current	Mon 8. May 20:59:31 2017	0	0	0
Map Report	Current	Mon 8. May 21:00:02 2017			
Place and Route Report	Current	Mon 8. May 21:00:19 2017	0	0	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Mon 8. May 21:00:26 2017	0	0	4 Infos (0 new)
Bitgen Report	Current	Mon 8. May 21:00:39 2017	0	1 Warning (0 new)	0

Secondary Reports			[-]
Report Name	**Status**	**Generated**	
ISIM Simulator Log	Out of Date	Mon 8. May 03:22:30 2017	
Post-Synthesis Simulation Model Report	Current	Fri 26. May 16:47:09 2017	
WebTalk Report	Current	Mon 8. May 21:00:39 2017	
WebTalk Log File	Current	Mon 8. May 21:00:40 2017	

==========

HDL Synthesis Report

Macro Statistics
Adders/Subtractors : 2
 3-bit adder : 1
 4-bit adder : 1
Registers : 3
 3-bit register : 1
 4-bit register : 1
 8-bit register : 1
Multiplexers : 11
 1-bit 2-to-1 multiplexer : 2
 3-bit 2-to-1 multiplexer : 1
 4-bit 2-to-1 multiplexer : 7
 4-bit 4-to-1 multiplexer : 1
FSMs : 1

Advanced HDL Synthesis Report

Macro Statistics
RAMs : 1
 64x31-bit single-port distributed Read Only RAM : 1
Adders/Subtractors : 8
 17-bit adder : 2
 24-bit addsub : 1
 24-bit subtractor : 1
 3-bit adder : 2
 33-bit adder : 2
Counters : 2
 8-bit up counter : 1
 9-bit up counter : 1
Registers : 363
 Flip-Flops :363

```
# Comparators                                          : 8
 17-bit comparator greater                            : 1
 3-bit comparator greater                             : 2
 33-bit comparator greater                            : 4
 8-bit comparator greater                             : 1
# Multiplexers                                         : 58
 1-bit 2-to-1 multiplexer                             : 15
 1-bit 8-to-1 multiplexer                             : 1
 16-bit 2-to-1 multiplexer                            : 2
 17-bit 2-to-1 multiplexer                            : 4
 24-bit 2-to-1 multiplexer                            : 20
 3-bit 2-to-1 multiplexer                             : 4
 33-bit 2-to-1 multiplexer                            : 10
 8-bit 2-to-1 multiplexer                             : 2
# FSMs                                                 : 5
=========================================================================
Final Register Report

Macro Statistics
# Registers                                            : 352
 Flip-Flops                                           : 352
```

Timing Report

This analysis gives the timing report of the design. The clock of the design was set according to the timing analysis report results and actual performance of the professor. The clock of the processor was set to 12.5 MHz considering the input arrival time and the output required time. Detailed analysis of the timing constrains is shown below.

Timing Summary:

Speed Grade: -3

Minimum period: 12.860ns (Maximum Frequency: 77.760MHz)

Minimum input arrival time before clock: 3.937ns

Maximum output required time after clock: 5.387ns

Maximum combinational path delay: 6.925ns

```
=========================================================================
Timing Report

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 17    |
-----------------------------------+------------------------+-------+

Asynchronous Control Signals Information:
----------------------------------------
No asynchronous control signals found in this design

Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 2.811ns (Maximum Frequency: 355.771MHz)
   Minimum input arrival time before clock: 3.633ns
   Maximum output required time after clock: 6.104ns
   Maximum combinational path delay: 6.925ns

Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
```

Setup/Hold to clock clk

Source	Max Setup to clk (edge)	Process Corner	Max Hold to clk (edge)	Process Corner	Internal Clock(s)	Clock Phase
reset	0.655(R)	FAST	0.421(R)	SLOW	clk_BUFGP	0.000
rx	0.992(R)	FAST	0.100(R)	SLOW	clk_BUFGP	0.000
s_tick	1.797(R)	SLOW	-0.198(R)	SLOW	clk_BUFGP	0.000

Clock clk to Pad

Destination	Max (slowest) clk (edge) to PAD	Process Corner	Min (fastest) clk (edge) to PAD	Process Corner	Internal Clock(s)	Clock Phase
dout<0>	7.250(R)	SLOW	3.868(R)	FAST	clk_BUFGP	0.000
dout<1>	7.256(R)	SLOW	3.874(R)	FAST	clk_BUFGP	0.000
dout<2>	7.180(R)	SLOW	3.813(R)	FAST	clk_BUFGP	0.000
dout<3>	7.183(R)	SLOW	3.816(R)	FAST	clk_BUFGP	0.000
dout<4>	7.065(R)	SLOW	3.770(R)	FAST	clk_BUFGP	0.000
dout<5>	7.210(R)	SLOW	3.874(R)	FAST	clk_BUFGP	0.000
dout<6>	7.210(R)	SLOW	3.874(R)	FAST	clk_BUFGP	0.000
dout<7>	7.207(R)	SLOW	3.871(R)	FAST	clk_BUFGP	0.000
rx_done_tick	8.959(R)	SLOW	4.473(R)	FAST	clk_BUFGP	0.000

Clock to Setup on destination clock clk

Source Clock	Src:Rise Dest:Rise	Src:Fall Dest:Rise	Src:Rise Dest:Fall	Src:Fall Dest:Fall
clk	2.597			

Pad to Pad

Source Pad	Destination Pad	Delay
s_tick	rx_done_tick	7.985

Timing Details:

All values displayed in nanoseconds (ns)

==========

Timing constraint: Default period analysis for Clock 'clk'

Clock period: 2.811ns (frequency: 355.771MHz)

Total number of paths / destination ports: 132 / 20

Delay: 2.811ns (Levels of Logic = 2)

Source: s_reg_2 (FF)

Destination: s_reg_3 (FF)

Source Clock: clk rising

Destination Clock: clk rising

Data Path: s_reg_2 to s_reg_3

Cell:in->out	Gate fanout	Net Delay	Delay	Logical Name (Net Name)
FDC:C->Q	5	0.447	0.943	s_reg_2 (s_reg_2)
LUT4:I1->O	12	0.205	0.909	Mmux_s_next411 (Mmux_s_next41)
LUT5:I4->O	1	0.205	0.000	Mmux_s_next42 (s_next<3>)
FDC:D		0.102		s_reg_3

```
   ----------------------------------------
   Total                     2.811ns (0.959ns logic, 1.852ns route)
                               (34.1% logic, 65.9% route)

==========================================================================
```

Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'

```
  Total number of paths / destination ports: 41 / 34
--------------------------------------------------------------------------
Offset:          3.633ns (Levels of Logic = 3)
  Source:         s_tick (PAD)
  Destination:    s_reg_3 (FF)
  Destination Clock: clk rising

  Data Path: s_tick to s_reg_3
                            Gate     Net
    Cell:in->out     fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O           6   1.222   0.992  s_tick_IBUF (s_tick_IBUF)
     LUT4:I0->O         12   0.203   0.909  Mmux_s_next411 (Mmux_s_next41)
     LUT5:I4->O          1   0.205   0.000  Mmux_s_next42 (s_next<3>)
     FDC:D               0.102          s_reg_3
    ----------------------------------------
    Total                3.633ns (1.732ns logic, 1.901ns route)
                           (47.7% logic, 52.3% route)

==========================================================================
```

Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'

```
  Total number of paths / destination ports: 14 / 9
--------------------------------------------------------------------------
Offset:          6.104ns (Levels of Logic = 3)
  Source:         s_reg_2 (FF)
  Destination:    rx_done_tick (PAD)
  Source Clock:   clk rising

  Data Path: s_reg_2 to rx_done_tick
                            Gate     Net
    Cell:in->out     fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q            5   0.447   0.943  s_reg_2 (s_reg_2)
     LUT4:I1->O         12   0.205   1.156  Mmux_s_next411 (Mmux_s_next41)
     LUT4:I0->O          1   0.203   0.579  Mmux_rx_done_tick11 (rx_done_tick_OBUF)
     OBUF:I->O                2.571          rx_done_tick_OBUF (rx_done_tick)
    ----------------------------------------
    Total                6.104ns (3.426ns logic, 2.678ns route)
                           (56.1% logic, 43.9% route)

==========================================================================
```

Timing constraint: Default path analysis

```
  Total number of paths / destination ports: 1 / 1
--------------------------------------------------------------------------
Delay:           6.925ns (Levels of Logic = 4)
```

Source: s_tick (PAD)
Destination: rx_done_tick (PAD)

Data Path: s_tick to rx_done_tick

Cell:in->out	Gate fanout	Net Delay	Delay	Logical Name (Net Name)
IBUF:I->O	6	1.222	0.992	s_tick_IBUF (s_tick_IBUF)
LUT4:I0->O	12	0.203	1.156	Mmux_s_next411 (Mmux_s_next41)
LUT4:I0->O	1	0.203	0.579	Mmux_rx_done_tick11 (rx_done_tick_OBUF)
OBUF:I->O		2.571		rx_done_tick_OBUF (rx_done_tick)
Total		**6.925ns (4.199ns logic, 2.726ns route)**		
		(60.6% logic, 39.4% route)		

Summary

Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
Total **3.633ns (1.732ns logic, 1.901ns route)**
(47.7% logic, 52.3% route)

Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
Total **6.104ns (3.426ns logic, 2.678ns route)**
(56.1% logic, 43.9% route)

Timing constraint: Default path analysis
Total **6.925ns (4.199ns logic, 2.726ns route)**
(60.6% logic, 39.4% route)

Distribution amongst logic and routing is nearly 50%. And the maximum is around 7 nanoseconds. This corresponds to 77.42 MHz. Since we are trying to optimise the design by trying to release the signals at the negative clock edge and doing the operation in the positive clock edge, the maximum of clock frequency achievable is around 35 MHz. We set the frequency to 12.5 MHz considering all the other actual timing constraints that may be not included in timing analysis of place and route report.

Furthermore, the performance of the processor is mostly decided by the time taken in receiving and transferring the image from and to the computer. Compared to this time, the processing time is below 5% of the total operational time. Therefore, setting the clock frequency of the processor even below 12.5MHz still will not matter.

Minimum delay after place and route setting - 8.959(R)

After place and route clock has a slower clock at 9ns. Therefore, setting of 12.5MHz clock is further justified.

8. REFERENCES

[1] Andrew S. Tanenbaum, “Structured Computer organization”, 5th edition, Pearson Prentice Hall,Pearson Education, Inc., Upper Saddle River, NJ 07458, 2006, pp.51-331.

https://eleccompengineering.files.wordpress.com/2014/07/structured_computer_organization_5th_edition_801_pages_2007_.pdf

[2]Xilinx Inc., “Spartan-6 FPGA Data Sheet: DC and Switching Characterises”, DS162 (v3.1.1) January 30, 2015.
http://www.xilinx.com/support/documentation/data_sheets/ds162.pdf

[3] DigilentInc., “Atlys FPGA Board Reference Manual”, April 11, 2016.
https://reference.digilentinc.com/_media/atlys:atlys:atlys_rm.pdf

[4] Deepak Kumar Tala, “Verilog Tutorial”.
http://asic-world.com/

[5] http://www.aw-bc.com/info/carpinelli/sample.pdf

[6]Bryan H.Fletcher, “FPGA Embedded Processors”, Embedded Training program, Embeddd Systems Conference San Francisco, 2005, ETP-367.
http://www.xilinx.com/products/design_resources/proc_central/.../ETP-367paper.pdf

APPENDICES

APPENDIX I: SUPPLIMENTARY PYTHON CODES

APPENDIX I-A: Image Down Sampling

```
import cv2
import numpy as np
import time
from PIL import Image

image = cv2.imread('lena-256x256.jpg')
image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

kernel = [ [1,2,1], [2,4,2], [1,2,1]]

memory = []

for x in range(0, 256):
      for y in range(0, 256):
            memory.append(image[x][y])

#print memory

for j in range(1, 254+1):
      for i in range(1, 254+1):

            x = j*256 + i

            total = 0

            total += memory[x] * 4;
            total += memory[x-1] * 2;
            total += memory[x+1] * 2;
            total += memory[x+256] * 2;
            total += memory[x-256] * 2;
            total += memory[x+257];
            total += memory[x+255];
            total += memory[x-257];
            total += memory[x-255];

            total = total / 16

            #print x - 9
            memory[x-257] = total

count = 0
```

```
result = ''
for i in image:
      result += 'memory[' + str(count) + '] = ' + str(i) + ';\n'
      count += 1

#print result

print 'Memory after gaussian filtered'

#for m in memory:
      #print m

# sampling the gaussian filtered image

k = 0

for j in range(0, 127+1):
      for i in range(0, 127+1):

            y = 2*256*j + 2*i

            memory[k] = memory[y]

            k = k + 1

print 'Memory after sampled'

for m in memory:
      print m

p = 128
result = np.zeros(shape=(p, p))
k = 0

for x in range(0, p):
      for y in range(0, p):
            result[x][y] = memory[k]
            k = k + 1

print result[0][0]

result = np.array(result, dtype = np.uint8)

cv2.imshow('Original image', image)
cv2.imshow('Downsmpled image', result)
cv2.waitKey(0)
```

APPENDIX I-B: Compiler

```
import os

OPCODE = {  'NOP'       :'00000',
                  'LOAD'            :'00001',
                  'STORE'           :'00010',
                  'MOVE'            :'00011',
                  'LDMAR'           :'00100',
                  'LDMARI'    :'00101',
                  'LOADI'           :'00110',
                  'LDACI'           :'00111',
                  'ADD'       :'01000',
                  'SUB'       :'01001',
                  'MUL'       :'01010',
                  'DIV'       :'01011',
                  'INC'       :'01100',
                  'DEC'       :'01101',
                  'NEG'       :'01110',
                  'NOT'       :'01111',
                  'AND'       :'10000',
                  'OR'        :'10001',
                  'XOR'       :'10010',
                  'JGT'       :'10011',
                  'JEQ'       :'10100',
                  'JGE'       :'10101',
                  'JLT'       :'10110',
                  'JNE'       :'10111',
                  'JLE'       :'11000',
                  'JMP'       :'11001',
                  'FIN'       :'11010'}

PREDEF = {  'AC'        :'000',
                  'R1'        :'001',
                  'R2'        :'010',
                  'R3'        :'011',
                  'R4'        :'100',
                  'R5'        :'101',
                  'R6'        :'110',
                  'R7'        :'111'}

def Translate(filename):
      # This function will convert assembly code to machine code
      a_code = []
      m_code = []

      SYMBOL = {}
      SYMBOL_COUNT = 16;

      with open(filename) as fp:
            for line in fp:
```

```
            x = line.strip()
            if len(x) > 0:
                  if not (x[0] == '/' and x[1] == '/'):
                        a_code.append(x);

for x in range(0, len(a_code)):
      if "//" in a_code[x]:
            t = a_code[x].split("//")
            a_code[x] = t[0].strip();

print a_code

# replace (xxx)
for x in range(0, len(a_code)):
      if a_code[x][0] == '(':
            label = a_code[x][1:-1]
            # count the instruction number
            count = 0;
            for x in range(0, len(a_code)):
                  if a_code[x][1:-1] == label:
                        break
                  if not a_code[x][0] == '(':
                        count += 1
            SYMBOL.update({label:str(count)})
            a_code[x] = '('

a_code = filter(lambda x: x is not '(', a_code)

# split opcode and arguments
for x in range(0, len(a_code)):
      a_code[x] = a_code[x].replace(',', ' ')   ## remove ','
      a_code[x] = a_code[x].split(' ')
      a_code[x] =  [var for var in a_code[x] if var]


# replace symbols and variable names
for x in range(0, len(a_code)):
      for y in range(0, len(a_code[x])):
            if a_code[x][y][0] == '@':
                  if a_code[x][y][1:] in SYMBOL:
                        a_code[x][y] = SYMBOL[a_code[x][y][1:]]


print SYMBOL

for x in a_code:
      print x

# translate assembly code to machine code
for x in range(0, len(a_code)):
```

```
            instruction = ''
            opcode = a_code[x][0]

            # no operation instruction
            if opcode == 'NOP':
                  instruction = format(0, '024b')

            elif opcode == 'FIN':
                  instruction = OPCODE[opcode] + format(0, '019b')

            # I type instructions
            elif opcode == 'LDACI':
                  instruction = OPCODE[opcode] + format(int(a_code[x][1]),
'019b')

            elif opcode == 'LDMARI':
                  instruction = OPCODE[opcode] + format(int(a_code[x][1]),
'019b')

            # RI type instructions
            elif opcode == 'LOADI':
                  instruction = OPCODE[opcode] + PREDEF[a_code[x][1]] +
format(int(a_code[x][2]), '016b')

            # jump instructions
            elif opcode[0] == 'J': # all jump instruction opcode starts
with J
                  instruction = OPCODE[opcode] + format(int(a_code[x][1]),
'019b')

            # R type instructions
            # LOAD, STORE, LDMAR, INC, DEC instructions
            elif opcode == 'LOAD' or opcode == 'STORE' or opcode ==
'LDMAR' or opcode == 'INC' or opcode == 'DEC':
                  instruction = OPCODE[opcode] + PREDEF[a_code[x][1]] +
format(0, '016b')

            # RR instructions
            # MOVE, NEG instructions
            elif opcode == 'MOVE' or opcode == 'NEG':
                  instruction = OPCODE[opcode] + PREDEF[a_code[x][1]] +
PREDEF[a_code[x][2]] + format(0, '013b')

            # RRR type instructions
            else:
                  instruction = OPCODE[opcode] + PREDEF[a_code[x][1]] +
PREDEF[a_code[x][2]] + PREDEF[a_code[x][3]] + format(0, '010b')
```

```
            m_code.append(instruction)

      for ins in m_code:
            print ins

      # output machine code to .mcode file
      var_name = 'memory'
      index = 0
      out = ''
      for x in m_code:
            out += var_name + '[' + str(index) + '] = 24\'b' + x + ";\n"
            #out += x + ",\n"
            index += 1
      out = out.strip()

      f = filename.split('.')
      outname = f[0] + ".mcode"

      file = open(outname, 'w')
      file.write(out)
      file.close();


path = os.path.dirname(os.path.realpath(__file__))

for file in os.listdir(path):
    if file.endswith(".asm"):
        Translate(os.path.join(path, file))
```

APPENDIX I-C: Image Communication with the Processor

```
import cv2
import numpy as np
import serial
import time

s = serial.Serial('COM3', 250000);


image = cv2.imread('lena-256x256.jpg')
image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

#for x in range(0, 256):
#     for y in range(0, 256):
#           print image[x][y]

#cv2.imshow('image', img)
#cv2.waitKey(0)
```

```
print 'Sending data...'

L = 256
p = L/2

t1 = time.time()

for x in range(0, L):
      for y in range(0, L):
            s.write(bytearray([image[x][y]]))
            #time.sleep(0.0001)

t = time.time() - t1

print 'Done!'
print 'Elapsed time ' + str(t) + ' s'
print 'Receiving data...'

rx_count = 0
rx_img = []

t2 = time.time()

while(1):
      raw = s.read()
      rx_img.append(raw)
      #print ord(raw),

      rx_count = rx_count + 1
      if rx_count == p*p:
            break

t = time.time() - t2
total_t = time.time() - t1

print 'Done!.'
print 'Elapsed time ' + str(t) + ' s'
print 'Total time ' + str(total_t) + ' s'

result = np.zeros(shape=(p, p))
k = 0

for x in range(0, p):
      for y in range(0, p):
            result[x][y] = map(ord, rx_img[k])[0]
            k = k + 1
            #print result[x][y]
```

```
result = np.array(result, dtype = np.uint8)

#img2 = image.copy()
#downSamImage = cv2.pyrDown(img2)

# Gaussian filtering the image
kernel = [ [1,2,1], [2,4,2], [1,2,1]]

memory = []

for x in range(0, 256):
      for y in range(0, 256):
            memory.append(int(image[x][y]))

#print memory

for j in range(1, 254+1):
      for i in range(1, 254+1):

            x = j*256 + i

            total = 0

            total += memory[x] * 4;
            total += memory[x-1] * 2;
            total += memory[x+1] * 2;
            total += memory[x+256] * 2;
            total += memory[x-256] * 2;
            total += memory[x+257];
            total += memory[x+255];
            total += memory[x-257];
            total += memory[x-255];

            total = total / 16

            #print x - 257
            memory[x-257] = total

# sampling the gaussian filtered image

k = 0

for j in range(0, 127+1):
      for i in range(0, 127+1):

            y = 2*256*j + 2*i

            memory[k] = memory[y]

            k = k + 1
```

```
downSamImage = np.zeros(shape=(p, p))

k = 0

for x in range(0, p):
      for y in range(0, p):
            downSamImage[x][y] = memory[k]
            k = k + 1
            #print result[x][y]


downSamImage = np.array(downSamImage, dtype = np.uint8)

# calculating the error

nonZeroPxCount = 0
maxErrorPx = 0

for x in range(0, p):
      for y in range(0, p):

            absVal = abs(int(downSamImage[x][y]) - int(result[x][y]))
            #print str(downSamImage[x][y]) + ' ' + str(result[x][y]) + ' '
+ str(absVal)

            # count non zero pixels
            if absVal != 0:
                  nonZeroPxCount += 1

            if absVal > maxErrorPx:
                  maxErrorPx = absVal

print 'Non zero pixel count ' + str(nonZeroPxCount)
print 'Maximum error ' + str(maxErrorPx)

# displaying the images
print 'Displaying the images.'

cv2.imshow('Original image', image)
cv2.imshow('Python', downSamImage)
cv2.imshow('FPGA', result)
cv2.waitKey(0)
```

APPENDIX II: INPUT-OUTPUT MODULE

APPENDIX II-A: 2to1 Mux (16 bit)

```
module MUX2to1_16bit(
input [15:0] A,B,
input sel,
output reg [15:0] OUT
);

always @(A or B or sel) begin
case(sel)
1'b0: OUT = A;
1'b1: OUT = B;
endcase
end

endmodule
```

APPENDIX II-B: 2to1 Mux (8 bit)

```
module MUX2to1_8bit(
input [7:0] A,B,
input sel,
output reg [7:0] OUT
);

always @(A or B or sel) begin
case(sel)
1'b0: OUT = A;
1'b1: OUT = B;
endcase
end

endmodule
```

APPENDIX II-C: 2to1 Mux

```
module MUX2to1(
input A, B,
input sel,
output reg OUT
);

always @(A or B or sel) begin
case(sel)
1'b0: OUT = A;
1'b1: OUT = B;
endcase
end

endmodule
```

APPENDIX II-D: 1to2 Demux (8 bit)

```
module DMUX1to2_8bit(
input [7:0] IN,
input sel,
output reg [7:0] A,B
);

always @(IN or sel) begin

case(sel)
1'b0: A = IN;
1'b1: B = IN;
endcase
end
endmodule
```

APPENDIX II-E: UART Transmitter (TX)

```
module UART_TX(
input clk, tx_start,
input [7:0] in_data_byte,
output tx_out, tx_done
);

parameter CLOCKS_PER_BIT = 50; // 9600 baud rate (100 MHz / 9600) 10417
**** 115200 baud rate 868 *** 109 for 12.5 MHz

parameter IDLE    = 3'b000;
parameter START   = 3'b001;
parameter DATA_TX = 3'b010;
parameter STOP    = 3'b011;
parameter CLEANUP = 3'b100;

parameter DELAY = 2;    // 50 milli second delay = 5000000

reg [2:0] state               = 0;
reg [7:0] data_byte           = 0;
reg [32:0] clock_count  = 0;
reg [2:0] tx_bit_index  = 0;
```

```
reg tx_data                          = 1'b1;
reg tx_data_done              = 1'b0;

always @(posedge clk)
begin

case(state)
IDLE:
begin
tx_data <= 1'b1;
clock_count <= 0;
tx_bit_index <= 0;

if(tx_start)
begin
tx_data_done <= 0;
data_byte <= in_data_byte;
state <= START;
end
else
begin
state <= IDLE;
end
end
START:
begin
tx_data <= 1'b0;

if(clock_count < CLOCKS_PER_BIT - 1)
begin
clock_count <= clock_count + 1;
state <= START;
end
else
begin
clock_count <= 0;
state <= DATA_TX;
end
end
DATA_TX:
begin
tx_data <= data_byte[tx_bit_index];

if(clock_count < CLOCKS_PER_BIT - 1)
begin
clock_count <= clock_count + 1;
state <= DATA_TX;
end
else
begin
```

```
if(tx_bit_index < 7)
begin
clock_count <= 0;
tx_bit_index <= tx_bit_index + 1;
state <= DATA_TX;
end
else
begin
clock_count <= 0;
tx_bit_index <= 0;
state <= STOP;
end
end

end
STOP:
begin
tx_data <= 1'b1;

if(clock_count < CLOCKS_PER_BIT - 1)
begin
clock_count <= clock_count + 1;
state <= STOP;
end
else
begin
clock_count <= 0;
state <= CLEANUP;
end
end
CLEANUP:
begin
if(clock_count < DELAY - 1)
begin
clock_count <= clock_count + 1;
end
else
begin
state <= IDLE;
clock_count <= 0;
tx_data_done <= 1;
end
end

default:
state <= IDLE;

endcase
end
```

```
assign tx_out = tx_data;
assign tx_done = tx_data_done;

endmodule
```

APPENDIX II-F: UART Receiver (RX)

```
module UART_RX(
input clk, rx_in,
output [7:0] rx_byte,
output rx_done
);

parameter CLOCKS_PER_BIT = 50;  // 9600 baud rate (100 MHz / 9600) 10417
**** 115200 baud rate 868

parameter IDLE    = 3'b000;
parameter START   = 3'b001;
parameter DATA_RX = 3'b010;
parameter STOP    = 3'b011;
parameter CLEANUP = 3'b100;

parameter DELAY = 1;    // delay 8680 clock cycles

reg [2:0] state               = 0;
reg [7:0] data_byte           = 0;
reg [32:0] clock_count  = 0;
reg [2:0] rx_bit_index  = 0;
reg rxdone                          = 0;

assign rx_byte = data_byte;
assign rx_done = rxdone;

always @(posedge clk)
begin
case(state)

IDLE:
begin
if(rx_in == 1'b0)
begin
rxdone <= 1'b0;
data_byte <= 0;
state <= START;
end
else
begin
rx_bit_index <= 0;
clock_count <= 0;
```

```
state <= IDLE;
end
end

START:
begin
if(clock_count < ((CLOCKS_PER_BIT/2) - 1) )
begin
clock_count <= clock_count + 1;
state <= START;
end
else
begin
clock_count <= 0;
state <= DATA_RX;
end
end

DATA_RX:
begin
if(clock_count < CLOCKS_PER_BIT - 1)
begin
clock_count <= clock_count + 1;
state <= DATA_RX;
end
else
begin
data_byte[rx_bit_index] <= rx_in;

if(rx_bit_index < 7)
begin
rx_bit_index <= rx_bit_index + 1;
state <= DATA_RX;
clock_count <= 0;
end
else
begin
rx_bit_index <= 0;
clock_count <= 0;
state <= STOP;
end
end

end

STOP:
begin
if(clock_count < CLOCKS_PER_BIT - 1)
begin
clock_count <= clock_count + 1;
```

```
state <= STOP;
end
else
begin
state <= CLEANUP;
clock_count <= 0;
end
end

CLEANUP:
begin
if(clock_count < DELAY - 1)
begin
clock_count <= clock_count + 1;
end
else
begin
rxdone <= 1'b1;
state <= IDLE;
clock_count <= 0;
end
end

default:
state <= IDLE;

endcase
end

endmodule
```

APPENDIX II-G: IO Module

```
module IO_MODULE(
input clk,
input reset,
input start,
input rx_done,
output reg tx_start,
input tx_done,
output reg [15:0] ram_addr,
output reg write,
output reg reset_cpu,
output sel
);

parameter RAM_LEN             = 65536;

parameter IDLE_RX             = 0;
```

```
parameter RX_1                      = 1;
parameter RX_2                      = 2;
parameter RX_3                      = 3;
parameter WAIT_RX             = 4;

parameter IDLE_TX             = 5;
parameter TX_1                      = 6;
parameter TX_2                      = 7;
parameter TX_3                      = 8;
parameter TX_4                      = 9;
parameter WAIT_TX             = 10;


reg [3:0] STATE_RX            = IDLE_RX;
reg [3:0] STATE_TX            = IDLE_TX;

reg [16:0] write_addr   = 0;
reg [16:0] read_addr    = 0;

reg select                          = 0;

assign sel = select;

initial
begin
reset_cpu <= 1'b0;
end

always @(posedge clk) begin

case(STATE_RX)

IDLE_RX:
begin
if(rx_done) begin
ram_addr <= write_addr;
STATE_RX <= RX_1;
end
else begin
write <= 0;
STATE_RX <= IDLE_RX;
end

if(reset) begin
write_addr <= 0;
select <= 0;
end
else begin
write_addr <= write_addr;
//select <= select;
```

```
end
end
RX_1:
begin
write <= 1;
STATE_RX <= RX_2;
end
RX_2:
begin
write <= 0;
write_addr <= write_addr + 1;
STATE_RX <= RX_3;
end
RX_3:
begin
if(write_addr == RAM_LEN) begin
reset_cpu <= 1;
select <= 1;
STATE_RX <= WAIT_RX;
end
else begin
reset_cpu <= 0;
//select <= select;
STATE_RX <= WAIT_RX;
end
end
WAIT_RX:    //    wait until rx_done goes high
begin
//reset_cpu <= 0;

if(~rx_done) begin
STATE_RX <= IDLE_RX;
end
else begin
STATE_RX <= WAIT_RX;
end

if(reset) begin
write_addr <= 0;
select <= 0;
end
else begin
write_addr <= write_addr;
//select <= select;
end
end

default:
STATE_RX <= IDLE_RX;
```

```
endcase

case(STATE_TX)

IDLE_TX:
begin
if(start) begin
read_addr <= 0;
select <= 0;
STATE_TX <= TX_1;
end
else begin
read_addr <= 0;
//select <= select;
STATE_TX <= IDLE_TX;
end
end
TX_1:
begin
ram_addr <= read_addr;
STATE_TX <= TX_2;
end
TX_2:
begin
tx_start <= 1;
STATE_TX <= TX_3;
end
TX_3:
begin
tx_start <= 0;
STATE_TX <= TX_4;
end
TX_4:
begin
if(tx_done) begin
if(read_addr < RAM_LEN - 1) begin
read_addr <= read_addr + 1;
STATE_TX <= TX_1;
end
else begin
read_addr <= 0;
STATE_TX <= WAIT_TX;
end
end
else begin
STATE_TX <= TX_4;
end
end
WAIT_TX:
begin
```

```
if(~start) begin
STATE_TX <= IDLE_TX;
end
else begin
STATE_TX <= WAIT_TX;
end
end

default:
STATE_TX <= IDLE_TX;
endcase
end
endmodule
```

APPENDIX III: PROCESSOR-I

APPENDIX III-A: General Purpose Register

```
module GPR(
input clk, load,
input [23:0] C_bus,
output reg [23:0] d_out
);

always@(posedge clk)
begin
if(load)    d_out <= C_bus;
end

end module
```

APPENDIX III-B: Memory Address Register (MAR)

```
module MAR(
input clk, load,
input [23:0] C_bus,
output reg [23:0] data_addr
);

always@(posedge clk)
begin
if(load) data_addr <= C_bus;
end
endmodule
```

APPENDIX III-C: Memory Data Register (MDR)

```
module MDR(

input clk, load, read,
input [7:0] C_bus,
input [7:0] data_in,
output reg [7:0] data_out
);

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

if(read)    data_out <= data_in;

end
endmodule
```

APPENDIX IID: Programme Counter (PC)

```
module PC(
input enable, clk, load, inc,
input [8:0] C_bus,
output reg [8:0] ins_addr
);

reg state = 0;

initial
begin
ins_addr <= 0;
end

always @(posedge enable)
begin
state <= 1;
//ins_addr <= 9'b0;
end

always@(posedge clk)
begin
if (state)
begin
//if(enable) ins_addr <= 9'b0;
if(load) ins_addr <= C_bus[8:0];
else if(inc) ins_addr <= ins_addr + 9'b000000001;
end
end
```

```
endmodule
```

APPENDIX III-E: Memory Buffer Register Unit (MBRU)

```
module MBRU(
input clk, fetch,
input [7:0] ins_in,
output reg [7:0] ins_out
);

always@(posedge clk)
begin
if(fetch)
begin
ins_out <= ins_in;
end
end
endmodule
```

APPENDIX IIF: 4to16 Decoder

```
module decoder (
input [23:0] R1, R2, R3, R4, R,
input [8:0] PC,
input [7:0] MBRU, MDR,
input [3:0] b_control,
output reg [23:0] B_bus
);

always @(b_control or R1 or R2 or R3 or R4 or R or PC or MBRU or MDR)
case(b_control)

4'd1:       B_bus <= {16'b0, MDR};

4'd2:       B_bus <= {15'b0, PC};

4'd3:       B_bus <= {16'b0, MBRU};

4'd4:       B_bus <= R1;

4'd5:       B_bus <= R2;

4'd6:       B_bus <= R3;

4'd7:       B_bus <= R4;

4'd8:       B_bus <= R;

default: B_bus <= 24'b0;
```

```
endcase
endmodule
```

APPENDIX III-G: Control Store

```
module microProcessor (

input enable, clk, Z_flag, addr_sel, JUMP,
input [7:0] addr, MBRU,
output reg [30:0] MIR
);


reg [1:0] state = 2'b00;
reg start = 1'b0;
reg [7:0] next_addr = 0;
reg [30:0] ROM[0:57];

parameter JMNZ1 = 8'd22,      JMNZY1 = 8'd23,   JMNZN1 = 8'd25,   FETCH2 =
8'd1;

//parameter JMPZY1 = 9'd50, JMPZN1 = 9'd48;
/*
always @(posedge enable)
begin
next_addr <= 8'b0;
end
*/

initial begin
MIR = 31'b0;
end

always @(posedge enable)
begin
start = 1'b1;
//MIR = 31'b0;
//state = 2'b00;
end

always @(posedge clk)
if(start)
begin
case(state)
2'b00:      state = 2'b01;
2'b01:      state = 2'b10;
2'b10:      state = 2'b11;
2'b11:      state = state;
default: state = state;
```

```
endcase
end

always @(negedge clk)
begin
if(state == 2'b11)
begin
case(addr)
FETCH2: MIR = {MBRU, ROM[FETCH2][22:0]};
JMNZ1:      if(Z_flag == 1'b0) MIR = ROM[JMNZN1];
else  MIR = ROM[JMNZY1];
default:    MIR = ROM[addr];
endcase
end
end

/*
always @(negedge clk)
begin
if(addr_sel)      next_addr = MBRU;
else  next_addr = addr;
end

always @(next_addr)
begin
if(enable == 1'b1)
begin
case(next_addr)
JMNZ1:      if(Z_flag == 1'b0) MIR = ROM[JMNZN1];
else  MIR = ROM[JMNZY1];
default:    MIR = ROM[next_addr];
endcase
end
else
begin
MIR = ROM[0];
end
end
*/
initial
begin
ROM[0] = 31'b00000001_00_0000_000000000_100_1_1010;
ROM[1] = 31'bXXXXXXXX_01_0000_000000000_000_0_1010;

ROM[2] = 31'b00111001_10_0000_000000000_000_0_0000;

ROM[3] = 31'b00000100_00_0000_000000000_010_0_0000;
ROM[4] = 31'b00000000_00_1000_000000001_000_0_0001;
ROM[5] = 31'b00000110_00_0000_000000000_000_1_0000;
ROM[6] = 31'b00000111_00_1000_000000001_000_0_0011;
```

```
ROM[7] = 31'b00001000_00_0101_000000001_100_1_0000;
ROM[8] = 31'b00000000_00_0001_000000001_000_0_0011;
ROM[9] = 31'b00001010_00_0111_010000000_000_0_0000;
ROM[10] = 31'b00000000_00_0000_000000000_001_0_0000;
ROM[11] = 31'b00000000_00_0111_000100000_000_0_0000;
ROM[12] = 31'b00000000_00_0111_000010000_000_0_0000;
ROM[13] = 31'b00000000_00_0111_000001000_000_0_0000;
ROM[14] = 31'b00000000_00_0111_000000100_000_0_0000;
ROM[15] = 31'b00000000_00_0111_000000010_000_0_0000;
ROM[16] = 31'b00000000_00_0111_100000000_000_0_0000;
ROM[17] = 31'b00000000_00_1000_000000001_000_0_0100;
ROM[18] = 31'b00000000_00_1000_000000001_000_0_0101;
ROM[19] = 31'b00000000_00_1000_000000001_000_0_0110;
ROM[20] = 31'b00000000_00_1000_000000001_000_0_0111;
ROM[21] = 31'b00000000_00_1000_000000001_000_0_1000;
ROM[22] = 31'bXXXXXXXX_00_0000_000000000_000_0_0000;
ROM[23] = 31'b00011000_00_0000_000000000_000_1_0000;
ROM[24] = 31'b00111000_00_0000_000000000_000_1_0000;
      //00111000_00_0001_001000000_000_0_0011
      00000000_00_0001_001000000_000_0_0011
ROM[25] = 31'b00011010_00_0000_000000000_100_1_0000;
ROM[26] = 31'b00011011_00_1000_000000001_000_0_0011;
ROM[27] = 31'b00011100_00_0101_000000001_100_0_0000;
ROM[28] = 31'b00111000_00_0001_001000000_000_0_0011;
      //00111000_00_0001_001000000_000_0_0011
      00000000_00_0001_001000000_000_0_0011
//
ROM[56] = 31'b00000000_00_0000_000000000_000_0_0000;
//
ROM[29] = 31'b00000000_00_1001_000000001_000_0_0000;
ROM[30] = 31'b00000000_00_1010_000000001_000_0_0000;
ROM[31] = 31'b00000000_00_0001_000000001_000_0_1000;
ROM[32] = 31'b00000000_00_0010_000000001_000_0_1000;
ROM[33] = 31'b00100010_00_0111_000000010_100_1_0000;
ROM[34] = 31'b00100011_00_1000_000000001_000_0_0011;
ROM[35] = 31'b00100100_00_0101_000000001_100_1_0000;
ROM[36] = 31'b00100101_00_1000_000000001_000_0_0011;
ROM[37] = 31'b00000000_00_0001_000000001_000_0_1000;
ROM[38] = 31'b00100111_00_0111_000000010_100_1_0000;
ROM[39] = 31'b00101000_00_1000_000000001_000_0_0011;
ROM[40] = 31'b00101001_00_0101_000000001_100_1_0000;
ROM[41] = 31'b00101010_00_1000_000000001_000_0_0011;
ROM[42] = 31'b00000000_00_0010_000000001_000_0_1000;
ROM[43] = 31'b00000000_00_0110_000000001_000_0_0000;
ROM[44] = 31'b00000000_00_0011_000000001_000_0_0000;
ROM[45] = 31'b00000000_00_0100_000000001_000_0_0000;
ROM[46] = 31'b00101111_00_0101_000000001_000_0_0000;
ROM[47] = 31'b00000000_00_0011_000000001_000_0_0000;
ROM[48] = 31'b00110001_00_1000_000000001_000_0_0100;
ROM[49] = 31'b00110010_00_1001_000000001_000_0_0000;
```

```
ROM[50] = 31'b00000000_00_0111_000100000_000_0_0000;
ROM[51] = 31'b00110100_00_1000_000000001_000_0_0101;
ROM[52] = 31'b00110101_00_1001_000000001_000_0_0000;
ROM[53] = 31'b00000000_00_0111_000010000_000_0_0000;
ROM[54] = 31'b00000000_00_1011_000000001_000_0_0000;
ROM[55] = 31'b00000000_00_0001_000001001_000_0_0110;

ROM[57] = 31'b00111001_00_0000_000000000_000_0_0000;
end
endmodule
```

APPENDIX III-H: Arithmetic Logic Unit (ALU)

```
module ALU(
input [23:0] A_bus,
input [23:0] B_bus,
input [3:0] oper,
output reg [23:0] C_bus,
output reg Z_flag
);

parameter   ADD = 4'b0001,    SUB = 4'b0010, PASSATOC = 4'b0111, PASSBTOC
= 4'b1000,
INCAC = 4'b1001,  DECAC = 4'b1010,  LSHFT1 = 4'b0011, LSHFT2 = 4'b0100,
LSHFT8 = 4'b0101, RSHFT4 = 4'b0110, RESET = 4'b1011;

always@(B_bus or oper or A_bus)
case(oper)
ADD:              C_bus = A_bus + B_bus;
            //1

SUB:              begin
C_bus = A_bus - B_bus;
Z_flag = (C_bus == 16'b0)? 1'b1 : 1'b0;   //2
end

LSHFT1:           C_bus = A_bus << 1;                //3

LSHFT2:           C_bus = A_bus << 2;                //4

LSHFT8:           C_bus = A_bus << 8;                //5

RSHFT4:           C_bus = A_bus >> 4;                //6

PASSATOC:   C_bus = A_bus;                           //7

PASSBTOC:   C_bus = B_bus;                           //8

INCAC:            C_bus = A_bus + 24'b1;        //9
```

```
DECAC:            C_bus = A_bus - 24'b1;        //10

RESET:            C_bus = 24'b0;

default:          begin
C_bus = 24'b0;
end

endcase
endmodule
```

APPENDIX III-I: Clock Generator

```
module CLOCK_GEN(
input clk_in,
output clk_out
);

parameter factor = 4;

reg [7:0] counter = 0;
reg out = 0;

assign clk_out = out;

always @(posedge clk_in) begin

if(counter < factor - 1) begin
counter <= counter + 1;
end
else begin
out <= ~out;
counter <= 0;
end

end
endmodule
```

APPENDIX III-J: Central Processing Unit (CPU)

```
module CPU(

input clk,
input reset,
input rx,
output tx,
output status
);
/*
```

```
input enable, clk;
output finish;

wire write;
wire [7:0] data_out, DRAM_data_write, ins_out;
wire [8:0] ins_addr;
wire [23:0] data_addr;

*/

// Instruction ROM
wire [7:0] instruction;
wire [8:0] pc_addr;

// Instruction RAM CPU Related
wire [7:0] ram_out_cpu, ram_in_cpu;
wire [23:0] ram_addr_cpu;

//
wire [7:0] ram_out_tx, ram_in_rx;
wire [15:0] ram_addr_io;

// TO the RAM module
wire [7:0] ram_out, ram_in;
wire [15:0] ram_addr;

wire write_cpu, write_io;
wire reset_cpu, finish;

wire sel;
wire clk2;

assign status = ~sel;

CLOCK_GEN clk_gen (
.clk_in(clk),
.clk_out(clk2)
);



Processor pentium4 (
.enable(reset_cpu),
.clk(clk2),
.data_in(ram_out_cpu),
.DRAM_addr(ram_addr_cpu),
.DRAM_data_write(ram_in_cpu),
.write(write_cpu),
.IROM_ins_fetch(instruction),
.IROM_addr(pc_addr),
```

```
.finish(finish)
);

/*
CPU cpu (
.clk(clk2),
.INSTRUCTION(instruction),
.RAM_IN(ram_out_cpu),
.reset(reset_cpu),
.RAM_OUT(ram_in_cpu),
.RAM_ADDR(ram_addr_cpu),
.PC_ADDR(pc_addr),
.WRITE(write_cpu),
.FINISH(finish)
);
*/

//----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
DataRAM dram216 (
.clka(clk2), // input clka
.wea(write), // input [0 : 0] wea
.addra(ram_addr), // input [15 : 0] addra
.dina(ram_in), // input [7 : 0] dina
.douta(ram_out) // output [7 : 0] douta
);

/*
RAM Ram (
.clka(clk2), // input clka
.wea(write), // input [0 : 0] wea
.addra(ram_addr), // input [15 : 0] addra
.dina(ram_in), // input [7 : 0] dina
.douta(ram_out) // output [7 : 0] douta
);
*/

//----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
InsROM irom512 (
.clka(clk2), // input clka
.addra(pc_addr), // input [8 : 0] addra
.douta(instruction) // output [7 : 0] douta
);

/*
ROM rom (
.address(pc_addr),
.out(instruction)
);
*/
```

```
IO_MODULE IO (
.clk(clk2),
.reset(reset),
.start(finish),
.rx_done(rx_done),
.tx_start(tx_start),
.tx_done(tx_done),
.ram_addr(ram_addr_io),
.write(write_io),
.reset_cpu(reset_cpu),
.sel(sel)
);

UART_TX Tx (
.clk(clk2),
.tx_start(tx_start),
.in_data_byte(ram_out_tx),
.tx_out(tx),
.tx_done(tx_done)
);

UART_RX Rx (
.clk(clk2),
.rx_in(rx),
.rx_byte(ram_in_rx),
.rx_done(rx_done)
);

MUX2to1_16bit mux1 (
.A(ram_addr_io),
.B(ram_addr_cpu[15:0]),
.sel(sel),
.OUT(ram_addr)
);

MUX2to1 mux2 (
.A(write_io),
.B(write_cpu),
.sel(sel),
.OUT(write)
);

MUX2to1_8bit mux3 (
.A(ram_in_rx),
.B(ram_in_cpu),
.sel(sel),
.OUT(ram_in)
);
```

```
DMUX1to2_8bit dmux1 (
.IN(ram_out),
.sel(sel),
.A(ram_out_tx),
.B(ram_out_cpu)
);

endmodule
```

APPENDIX IV: PROCESSOR-II

APPENDIX IV-A: Register (24 bit)

```
module REG24bit(
input clk, load,
input [23:0] IN,
output reg [23:0] OUT
);

always @(posedge clk) begin

if(load) begin
OUT <= IN;
end

end

endmodule
```

APPENDIX IV-B: Program Counter (PC)

```
module PC(
input clk,
input reset,inc,load,
input [23:0] IN,
output reg [23:0] OUT
);

always @(posedge clk)
begin
if(inc) begin
OUT = OUT + 1;
end

if(load) begin
OUT = IN;
end
```

```
if(reset) begin
OUT = 0;
end
end

endmodule
```

APPENDIX IV-C: 8to1 Mux

```
module MUX8to1(
input [23:0] A,B,C,D,E,F,G,H,
input [2:0] sel,
output reg [23:0] OUT
);

always @(A or B or C or D or E or F or G or H or sel) begin
case(sel)
3'b000: OUT = A;
3'b001: OUT = B;
3'b010: OUT = C;
3'b011: OUT = D;
3'b100: OUT = E;
3'b101: OUT = F;
3'b110: OUT = G;
3'b111: OUT = H;
endcase
end

endmodule
```

APPENDIX IV-D: 3to1 Mux

```
module MUX3to1(
input [23:0] A,B,C,
input [1:0] sel,
output reg [23:0] OUT
);

always @(A or B or C or sel) begin
OUT = 0;
case(sel)
2'b00: OUT = A;
2'b01: OUT = B;
2'b10: OUT = C;
endcase
end

endmodule
```

APPENDIX IV-E: 1to2 Demux

```
module DMUX1to2(
input [23:0] IN,
input sel,
output reg [23:0] A,B
);

always @(IN or sel) begin

case(sel)
1'b0: A = IN;
1'b1: B = IN;
endcase

end

endmodule
```

APPENDIX IV-F: 1to10 Demux

```
module DMUX1to10(
input IN,
input [3:0] sel,
output reg A,B,C,D,E,F,G,H,I,J
);

always @(IN or sel) begin

A = 0;
B = 0;
C = 0;
D = 0;
E = 0;
F = 0;
G = 0;
H = 0;
I = 0;
J = 0;

case(sel)
4'b0000: A = IN;
4'b0001: B = IN;
4'b0010: C = IN;
4'b0011: D = IN;
4'b0100: E = IN;
4'b0101: F = IN;
4'b0110: G = IN;
4'b0111: H = IN;
```

```
4'b1000: I = IN;
4'b1001: J = IN;
endcase

end

endmodule
```

APPENDIX IV-G: Arithmetic & Logic Unit (ALU)

```
module ALU24bit(
input [23:0] A,B,
input [3:0] sel,
output reg [23:0] OUT,
output reg Z,N
);

parameter NOP     = 4'b0000;
parameter ADD     = 4'b0001;
parameter SUB     = 4'b0010;
parameter MUL     = 4'b0011;
parameter DIV     = 4'b0100;
parameter INC     = 4'b0101;
parameter DEC     = 4'b0110;
parameter NEG     = 4'b0111;
parameter NOT     = 4'b1000;
parameter AND     = 4'b1001;
parameter OR      = 4'b1010;
parameter XOR     = 4'b1011;


always @(A or B or sel) begin

Z = 0;
N = 0;

case(sel)
NOP: OUT = A;
ADD: OUT = A+B;
SUB: OUT = A-B;
MUL: OUT = A<<B;
DIV: OUT = A>>B;
INC: OUT = A+1;
DEC: OUT = A-1;
NEG: OUT = -A;
NOT: OUT = ~A;
AND: OUT = A&B;
OR:  OUT = A|B;
XOR: OUT = A^B;
```

```
endcase

if(OUT==0) begin
Z=1;
end
else if(OUT[23]==1) begin
N=1;
end
end

endmodule
```

APPENDIX IV-H: Clock Generator

```
module CLOCK_GEN(
input clk_in,
output clk_out
);

parameter factor = 4;

reg [7:0] counter = 0;
reg out = 0;

assign clk_out = out;

always @(posedge clk_in) begin

if(counter < factor - 1) begin
counter <= counter + 1;
end
else begin
out <= ~out;
counter <= 0;
end

end
endmodule
```

APPENDIX IV-I: Decoder

```
module DECODER(
input clk,
input [23:0] INSTRUCTION,
input Z,N,
output reg INC_PC, LOAD_REG,
output reg [2:0] MUX_A_SEL, MUX_B_SEL,
output reg [1:0] MUX_C_SEL,
output reg DMUX_A_SEL,
output reg [3:0] DMUX_B_SEL,
```

```
output reg [3:0] ALU_CONTROL,
output reg WRITE,
output reg [23:0] IMMEDIATE,
output reg FINISH
);

parameter NOP           = 5'b00000;       // No Operation
parameter LOAD    = 5'b00001;       // RegA = RAM[MAR]
parameter STORE   = 5'b00010;       // RAM[MAR] = RegA
parameter MOVE    = 5'b00011;       // RegA = RegB
parameter LDMAR   = 5'b00100;       // MAR = RegA
parameter LDMARI  = 5'b00101;       // MAR = signed immediate (19-bit)
parameter LOADI   = 5'b00110;       // RegA = signed immediate (16-bit)
parameter LDACI   = 5'b00111;       // AC = signed immediate (19-bit)

parameter ADD           = 5'b01000;       // RegA = RegB + RegC
parameter SUB           = 5'b01001;       // RegA = RegB - RegC
parameter MUL           = 5'b01010;       // RegA = RegB << RegC
parameter DIV           = 5'b01011;       // RegA = RegB >> RegC
parameter INC           = 5'b01100;       // RegA = RegB + 1
parameter DEC           = 5'b01101;       // RegA = RegB - 1
parameter NEG           = 5'b01110;       // RegA = -RegB
parameter NOT           = 5'b01111;       // RegA = !RegB
parameter AND           = 5'b10000;       // RegA = RegB & RegC
parameter OR            = 5'b10001;       // RegA = RegB | RegC
parameter XOR           = 5'b10010;       // RegA = RegB ^ RegC

parameter JGT           = 5'b10011;       // If ALU out > 0 then PC = Reg
A else PC ? PC + 1
parameter JEQ           = 5'b10100;       // If ALU out = 0 then PC = Reg
A else PC ? PC + 1
parameter JGE           = 5'b10101;       // If ALU out >= 0 then PC = Reg
A else PC ? PC + 1
parameter JLT           = 5'b10110;       // If ALU out < 0 then PC = Reg
A else PC ? PC + 1
parameter JNE           = 5'b10111;       // If ALU out != 0 then PC = Reg
A else PC ? PC + 1
parameter JLE           = 5'b11000;       // If ALU out <= 0 then PC = Reg
A else PC ? PC + 1
parameter JMP           = 5'b11001;       // PC ? Reg A (Unconditional
Jump)

parameter FIN           = 5'b11010;       // FINISH = 1

reg [4:0] OPCODE  = 0;
reg [2:0] RegA          = 0;
reg [2:0] RegB    = 0;
reg [2:0] RegC    = 0;
reg [15:0] IMM16  = 0;
reg [18:0] IMM19  = 0;
```

```
always @(negedge clk) begin

// decode instruction
OPCODE      = INSTRUCTION[23:19];
RegA        = INSTRUCTION[18:16];
RegB        = INSTRUCTION[15:13];
RegC        = INSTRUCTION[12:10];
IMM16       = INSTRUCTION[15:0];
IMM19       = INSTRUCTION[18:0];

// set control signal to default values (0)
INC_PC = 1; // increment PC by default
LOAD_REG = 0;
MUX_A_SEL = 0;
MUX_B_SEL = 0;
MUX_C_SEL = 0;
DMUX_A_SEL = 0;
DMUX_B_SEL = 0;
ALU_CONTROL = 0;
WRITE = 0;
IMMEDIATE = 0;
FINISH = 0;

case(OPCODE)
NOP:
begin
INC_PC = 1; // increment pc
end
LOAD:
begin
MUX_C_SEL = 1;    // select RAM
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
STORE:
begin
MUX_B_SEL = RegA;
DMUX_A_SEL = 1; // select RAM
WRITE = 1;
end
MOVE:
begin
MUX_A_SEL = RegB;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
LDMAR:
begin
MUX_A_SEL = RegA;
```

```
DMUX_B_SEL = 8;   // select MAR
LOAD_REG = 1;
end
LDMARI:
begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 8; // select MAR
LOAD_REG = 1;
end
LOADI:
begin
IMMEDIATE = IMM16;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
LDACI:
begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 0; // select AC
LOAD_REG = 1;
end


ADD:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 1;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
SUB:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 2;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
MUL:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 3;
```

```
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
DIV:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 4;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
INC:
begin
MUX_A_SEL = RegA;
ALU_CONTROL = 5;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
DEC:
begin
MUX_A_SEL = RegA;
ALU_CONTROL = 6;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
NEG:
begin
MUX_A_SEL = RegA;
ALU_CONTROL = 7;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
NOT:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 8;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
AND:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 9;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
```

```
end
OR:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 10;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end
XOR:
begin
MUX_A_SEL = RegB;
MUX_B_SEL = RegC;
DMUX_A_SEL = 0;
ALU_CONTROL = 11;
DMUX_B_SEL = RegA;
LOAD_REG = 1;
end

JGT:
begin
if(Z==0 & N==0) begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 9; // select PC
LOAD_REG = 1;
INC_PC = 0; // do not increment PC
end
end
JEQ:
begin
if(Z==1) begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 9; // select PC
LOAD_REG = 1;
INC_PC = 0; // do not increment PC
end
end
JGE:
begin
if(N==0) begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 9; // select PC
LOAD_REG = 1;
INC_PC = 0; // do not increment PC
end
end
```

```
JLT:
begin
if(Z==0 & N==1) begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 9; // select PC
LOAD_REG = 1;
INC_PC = 0; // do not increment PC
end
end
JNE:
begin
if(Z==0) begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 9; // select PC
LOAD_REG = 1;
INC_PC = 0; // do not increment PC
end
end
JLE:
begin
if(Z|N) begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 9; // select PC
LOAD_REG = 1;
INC_PC = 0; // do not increment PC
end
end
JMP:
begin
IMMEDIATE = IMM19;
MUX_C_SEL = 2; // select immediate value from decoder
DMUX_B_SEL = 9; // select PC
LOAD_REG = 1;
INC_PC = 0; // do not increment PC
end

FIN:
begin
FINISH = 1;
end

endcase

end

endmodule
```

APPENDIX IV-J: Root System

```
module ROOT_SYSTEM(
input clk,
input reset,
input rx,
output tx,
output status
);

wire [23:0] instruction, pc_addr;

wire [7:0] ram_out_cpu, ram_in_cpu;
wire [15:0] ram_addr_cpu;

wire [7:0] ram_out_tx, ram_in_rx;
wire [15:0] ram_addr_io;

wire [7:0] ram_out, ram_in;
wire [15:0] ram_addr;

wire write_cpu, write_io;
wire reset_cpu, finish;

wire sel;
wire clk2;

assign status = ~sel;

CLOCK_GEN clk_gen (
.clk_in(clk),
.clk_out(clk2)
);

CPU cpu (
.clk(clk2),
.INSTRUCTION(instruction),
.RAM_IN(ram_out_cpu),
.reset(reset_cpu),
.RAM_OUT(ram_in_cpu),
.RAM_ADDR(ram_addr_cpu),
.PC_ADDR(pc_addr),
.WRITE(write_cpu),
.FINISH(finish)
);

RAM Ram (
.clka(clk2), // input clka
.wea(write), // input [0 : 0] wea
```

```
.addra(ram_addr), // input [15 : 0] addra
.dina(ram_in), // input [7 : 0] dina
.douta(ram_out) // output [7 : 0] douta
);

ROM rom (
.address(pc_addr),
.out(instruction)
);

IO_MODULE IO (
.clk(clk2),
.reset(reset),
.start(finish),
.rx_done(rx_done),
.tx_start(tx_start),
.tx_done(tx_done),
.ram_addr(ram_addr_io),
.write(write_io),
.reset_cpu(reset_cpu),
.sel(sel)
);

UART_TX Tx (
.clk(clk2),
.tx_start(tx_start),
.in_data_byte(ram_out_tx),
.tx_out(tx),
.tx_done(tx_done)
);

UART_RX Rx (
.clk(clk2),
.rx_in(rx),
.rx_byte(ram_in_rx),
.rx_done(rx_done)
);

MUX2to1_16bit mux1 (
.A(ram_addr_io),
.B(ram_addr_cpu),
.sel(sel),
.OUT(ram_addr)
);

MUX2to1 mux2 (
.A(write_io),
.B(write_cpu),
.sel(sel),
.OUT(write)
```

```
);

MUX2to1_8bit mux3 (
.A(ram_in_rx),
.B(ram_in_cpu),
.sel(sel),
.OUT(ram_in)
);

DMUX1to2_8bit dmux1 (
.IN(ram_out),
.sel(sel),
.A(ram_out_tx),
.B(ram_out_cpu)
);

endmodule
```