EN 3030

CIRCUITS AND SYSTEMS DESIGN

FPGA BASED PROCESSOR IMPLEMENTATION

Group Members

Ekanayake E.M.A.S.	(130150L)
Herath H.M.K.S.	(130203E)
Kuruppu K.A.D.H.K.	(130314U)
Rasanga M.L.W.	(130503A)

Supervisor:
Dr.Jayathu Samarawickrama

A Submitted Project Report In Partial Fulfillment Of The Requirement For The Module EN 3030.

Department of Electronics and Telecommunication,
University Of Moratuwa
August 7, 2016

TABLE OF CONTENTS

1. INTRODUCTION

The objective of this project is to design object specified microprocessor and CPU (Central Processing Unit) model and simulate it using the Verilog hardware description language, and finally to implement it in hardware using programmable logic device such as FPGA (Field Programmable Gate Array). This document describes the microprocessor and CPU design, the test codes used to verify it, and the physical hardware implementation.

1.1.CPU AND MICROPROCESSOR DESIGN

A central processing unit(CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions. Traditionally, the term "CPU" refers to a processor, more specifically to its **processing unit** and **control unit** (CU), distinguishing these core elements of a computer from external components such as main memory and I/O circuitry.

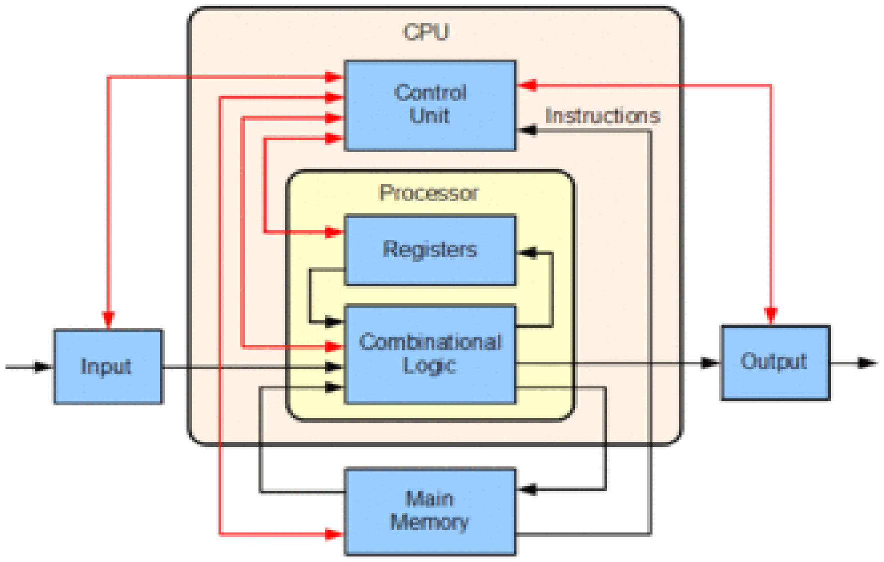


Figure 1.1: Block Diagram of a CPU

A microprocessor is a computer processor which incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits. The microprocessor is a multipurpose, clock driven, register based programmable electronic device which accepts digital or binary data as input, processes it according to instructions stored in its memory, and provides results as output.

Microprocessors contain both combinational logic and sequential digital logic. Microprocessors operate on numbers and symbols represented in the binary numeral system.

Thousands of items that were traditionally not computer-related include microprocessors. These include large and small household appliances, cars (and their accessory equipment units), car keys, tools and test instruments, toys, light switches/dimmers and electrical circuit breakers, smoke alarms, battery packs, and hi-fi audio/visual components (from DVD players to phonograph turntables). Such products as cellular telephones, DVD video system and HDTV broadcast systems fundamentally require consumer devices with powerful, low-cost, microprocessors. Increasingly stringent pollution control standards effectively require automobile manufacturers to use microprocessor engine management systems, to allow optimal control of emissions over widely varying operating conditions of an automobile. Non-programmable controls would require complex, bulky, or costly implementation to achieve the results possible with a microprocessor.

Therefore we can say that microprocessors are very important electronic devices for implement modern world applications. Because all the modern world applications depend on microprocessors, to achieve special tasks various types of microprocessors have been built by different companies. Therefore designing and implementing a special microprocessor to fulfill a given task is very important. In this project we look to design and implement such a microprocessor to achieve given tasks with maximum optimization.

1.2. PROBLEM STATEMENT

In this section we consider about the requirements to be done by the microprocessor and other implementation requirements.

The main requirement is to design a CPU and a microprocessor for filtering and down sampling a given image. The CPU should get data from the serial input and save to a main memory then do the processing part. After that CPU should return the processed data and display the results. This task can be done by dividing the main task for several sub tasks.

- **Control input data signal and store it in a primary memory.**
  This task can be done by UART implementation in the FPGA and send serial data to the UART port by serial data transmitting software. We should implement the UART

receiver using Verilog in FPGA. After receiving the data, received data should be stored in a given primary memory which has been implemented in the FPGA.

- **Filtering the saved image**

  In this section task is to filter the image for reducing the effect of high frequency components in the image. Otherwise high frequency components can be problematic for the accuracy of the image after down sampling. 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.

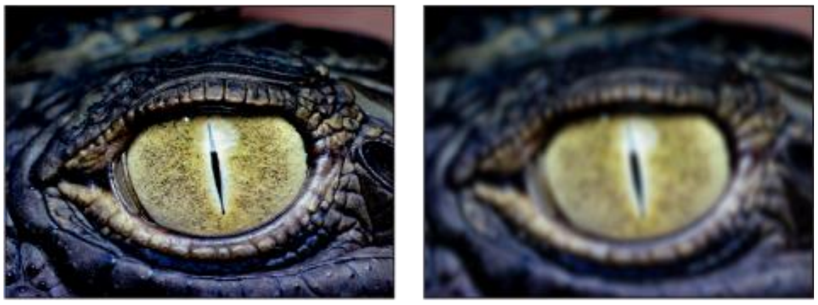


Figure 1.2: Original and Gaussian Filtered image

- **Down Sampling 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. This process also should be implemented on the FPGA and after doing the down sampling process; data should be stored in the primary memory.

- **Control Output data and display the processed image**

  After doing the given tasks to the original image the processed data should be returned and the results should be displayed. UART transmitter should be implemented on the FPGA in order to get the processed data from FPGA to the computer. Serial communication software (or MatLab) can be used to collect the data.

1.3.GENERAL OVERVIEW OF THE SOLUTION

Considering the main tasks discussed in the above section, solution for the given problem can be developed and implemented on FPGA. In this project we used Atlys Spartan-6 FPGA board for the hardware implementation.

First the UART receiving model is implemented and using the UART receiver the raw data can get into the FPGA using serial communication with the computer. In this project we consider a 256x256 image for testing purposes. Therefore we send and store all the 65536 pixel data values in a primary memory in the FPGA. This data is used for processing tasks. This can be done by using USB-UART Bridge. The Atlys board includes a USB-UART Bridge (Exar XR21V1410) which allows us to do serial communication between FPGA and the Host.

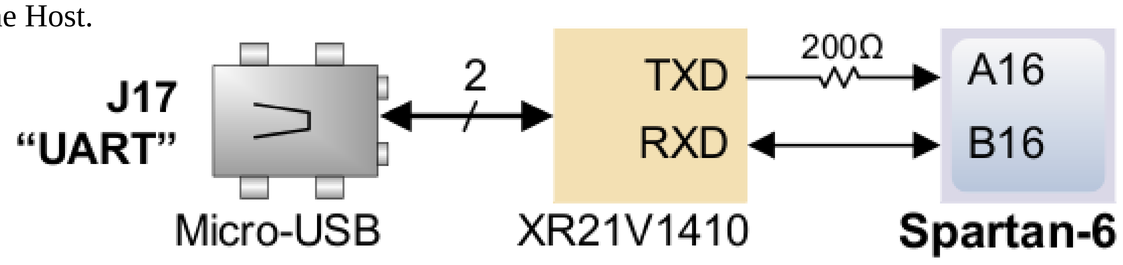


Figure 1.3:Serial Communication with FPGA

After storing the data in the memory we have to do the processing part using microprocessor which we have developed. To complete these tasks we have designed a microprocessor with instruction set architecture which can perform the given task which is filtering and down sampling the image. Using the developed ISA we managed to find micro instructions for each instruction. When the process is running in the FPGA it fetches the instructions from the Instruction memory and does the processing on stored data in the Data memory. All the components in ISA are discussed in detailed in the next section.

Finally we have to give the processed data to the output components. This can also be done using the USB-UART Bridge. We can transmit the processed data through the UART port using serial communication method. After getting the processed data we can analyze the output using different methods. In the last section we have analyzed and verified our results with the software based simulations. (MatLab)

2. INSTRUCTION SET ARCHITECTURE (ISA)

2.1.GENERAL ARCHITECTURE

The goal of this processor is to down sample an image of size 256 x 256. Therefore the data RAM of this architecture has the ability to store 65536 pixels thereby having 65536 memory locations. Based on the above consideration the architecture of this processor is designed as follows.

- **DRAM-** A data RAM which consists of 65536 memory locations with a width of 8 bits to store the pixels of the image
- **IRAM-** An instruction RAM which consists of instructions to be executed with 256 memory locations and a width of 8 bits. This basically contains the assembly code of the algorithm for filtering and down sampling the image.
- **AR** – A 16 it address register which supplies an address to the data RAM
- **PC** – A 16 bit program counter which keeps the address of the next instruction in the instruction RAM to be executed
- **IR** – An 8 bit instruction register which stores the instruction read from the IRAM
- **R1, R2, R3, R-** Four 16 bit general purpose registers
- **AC** – A 16 bit accumulator which has direct access to the ALU. Whenever data is loaded from data memory, it is loaded to AC and data stored to data memory is stored from AC.
- **ALU**–A 16 bit ALU which performs 7 different operations.
- **State Machine**–It generates all the control signals for the processor and makes the decision based on the given instruction. (From IR)
- **BUS** – 16 bit wire which carries the data read from registers, DRAM or IRAM. (This has been implemented using a de-multiplexer)

2.2. DATA PATH

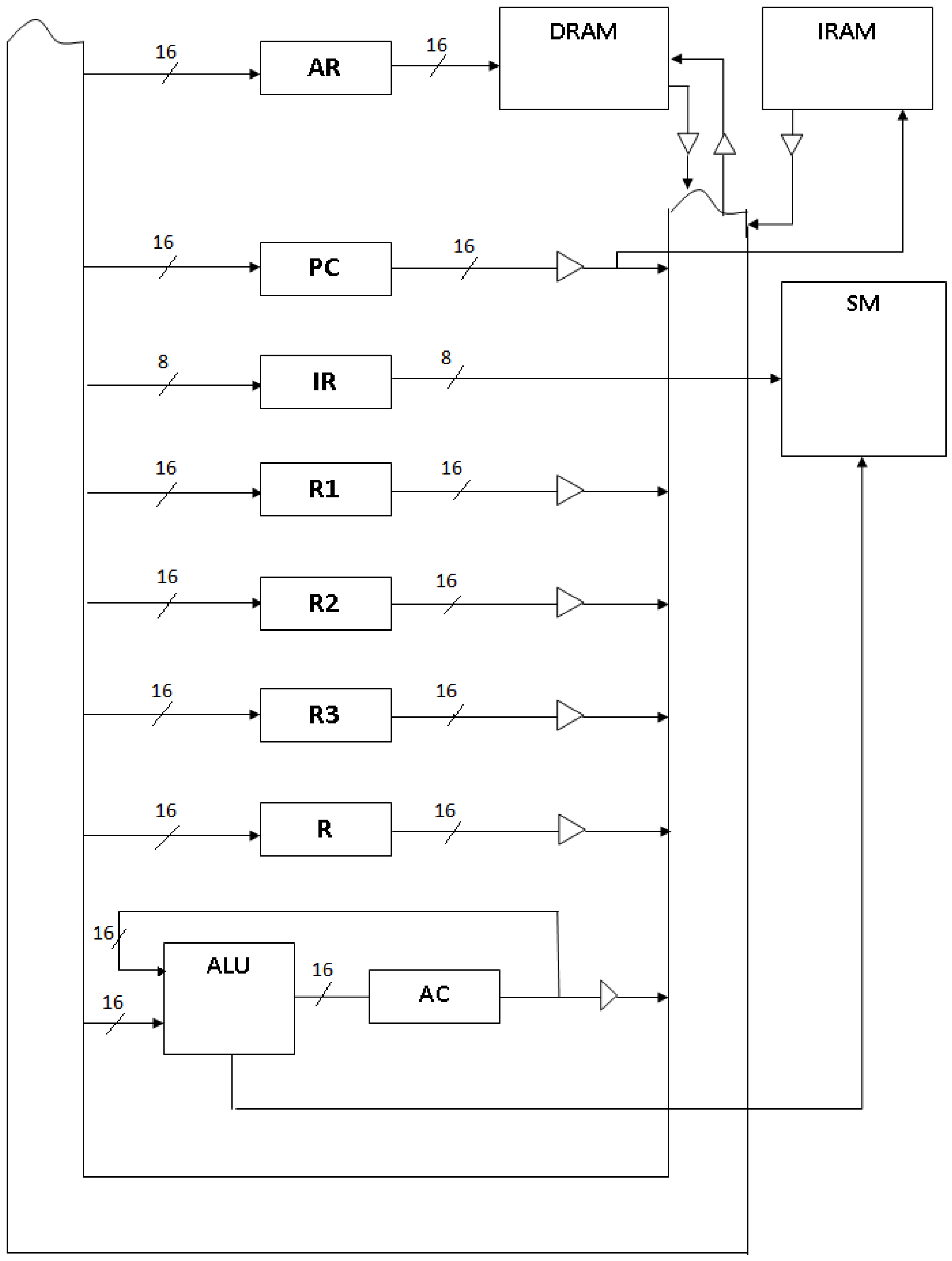


Figure 2.1: Data path of the CPU

2.3. INSTRUCTION SET

The processor consists of 6 bit instructions. The instruction set of the processor is shown in the table.

Instruction	Instruction Code	Operation
CLAC	00000011	AC🡐0
STAC	00000100	DRAM[AR]🡐AC
MVACR1	00000110	R1🡐AC
MVACR2	00000111	R2🡐AC
MVACR3	00001000	R3🡐AC
MVACR	00001001	R🡐AC
MVR1AC	00001010	AC🡐R1
MVR2AC	00001011	AC🡐R2
MVR3AC	00001100	AC🡐R3
MVRAC	00001101	AC🡐R
INCAC	00001110	AC🡐AC+1
LDAC	00001111	AC🡐 DRAM[AC]
SUB	00010001	AC 🡐AC-R
DECAC	00010010	AC🡐AC-1
ADD	00010011	AC🡐AC+R
DIV2	00010100	AC🡐AC/2
MUL4	00010101	AC🡐AC*4
JUMPNZ “α”	00010110	If Z=0; THEN GO TO “α”
ADDM “α”	00011011	AC🡐AC + α
INCR1	00101000	R1🡐R1+1
INCR2	00101001	R2🡐R2+1
INCR3	00101010	R3🡐R3+1
NOP	00111010	No Operation
MVACAR	00111011	AR🡐AC
END	00111100	

Table 2.1: Instruction Set

2.4. FETCH, DECODE, EXECUTION CYCLE

2.4.1.Fetching instructions

Fetch cycle consists of 4 states. Fetch cycle is run by the state machine with FETCH1 being set as the next state at the beginning.

FETCH1: Fetch

FETCH2: IR$\leftarrow$IRAM

FETCH3: PC$\leftarrow$PC+1

FETCH4: Next State$\leftarrow$IR

2.4.2.Decoding instructions

After fetching instructions from the instruction memory, the CPU has to identify which instruction has been fetched thereby invoking the correct execution cycle. This task is done by the state machine. Instruction Register (IR) inputs the fetched instruction to the state machine and the state machine runs the relevant state followed by the next states of the instruction or returns to fetch cycle if the instruction has only one state.

2.4.3.Executing instructions

- **CLAC Instruction**

  CLAC1: AC$\leftarrow$0

  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.

- **STAC Instruction**

  This instruction consists of 2 states.

  STAC1: DRAM[AR]🡐AC

  This involves copying the contents of the AC to the memory address in the data memory pointed by the AR.

  STAC2: WRITE

- **MVACR1, MVACR2, MVACR3, MVACR and MVACAR Instructions**

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

  MVACR11: R1🡐AC

  MVACR21: R2🡐AC

  MVACR31: R3🡐AC

  MVACR1: R🡐AC

  MVACAR: AR🡐AC

- **MVR1AC, MVR2AC, MVR3AC and MVRAC 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.

  MVR1AC1: AC🡐R1

  MVR2AC1: AC🡐R2

  MVR3AC1: AC🡐R3

  MVRAC1: AC🡐R

- **INCAC and DECAC Instructions**

INCAC1: AC←AC+1

INCAC Instruction involves the CPU to add 1 to the contents of AC and write back to AC.

DECAC1: 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.

- **LDAC Instruction**

This instruction consists of two states.

LDAC1: AR←AC

1st state simply copies the contents of AC, which is the memory address to which the data to be written, to AR.

LDAC2: AC←DRAM[AR]

This state involves loading the data from the address given in the previous state (AR) to AC. Then 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.

- **DIV2 and MUL4 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.

DIV21: AC🡐AC/2

MUL41: AC🡐AC*4

- **JUMPNZ Instruction**

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

If Z=1; JUMPNZY1: PC🡐PC+1

If z flag equals to zero three states are involved. In the 2nd state instruction in the memory address is loaded into AC. Then value of AC is copied to PC. Then the CPU moves back to fetch routine.

If Z=0;

JUMPNZN1: FETCH

JUMPNZN2: AC🡐 α (Which stored next to JUMPNZ in IRAM)

JUMPNZN3: PC🡐AC

- **ADDM 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.

ADDM 1: FETCH, PC🡐PC+1

In the 1st state PC is incremented by 1.

ADDM 2: AC$\leftarrow$AC+α (Which stored next to ADDM in IRAM)

In the next state the value in the memory location which is next to "ADDM" is added to AC and written back to AC.

- **INCR1, INCR2, INCR3 Instructions**

These instructions involve only one state which adds 1 to the contents of R1,R2 or R3 and moves back to the fetch routine.

R1: R1$\leftarrow$R1+1

R2: R2$\leftarrow$R2+1

R3: R3$\leftarrow$R3+1

- **NOP Instruction**

By executing this instruction, CPU does nothing and moves back to fetch routine.

NOP1: (NO OPERATION)

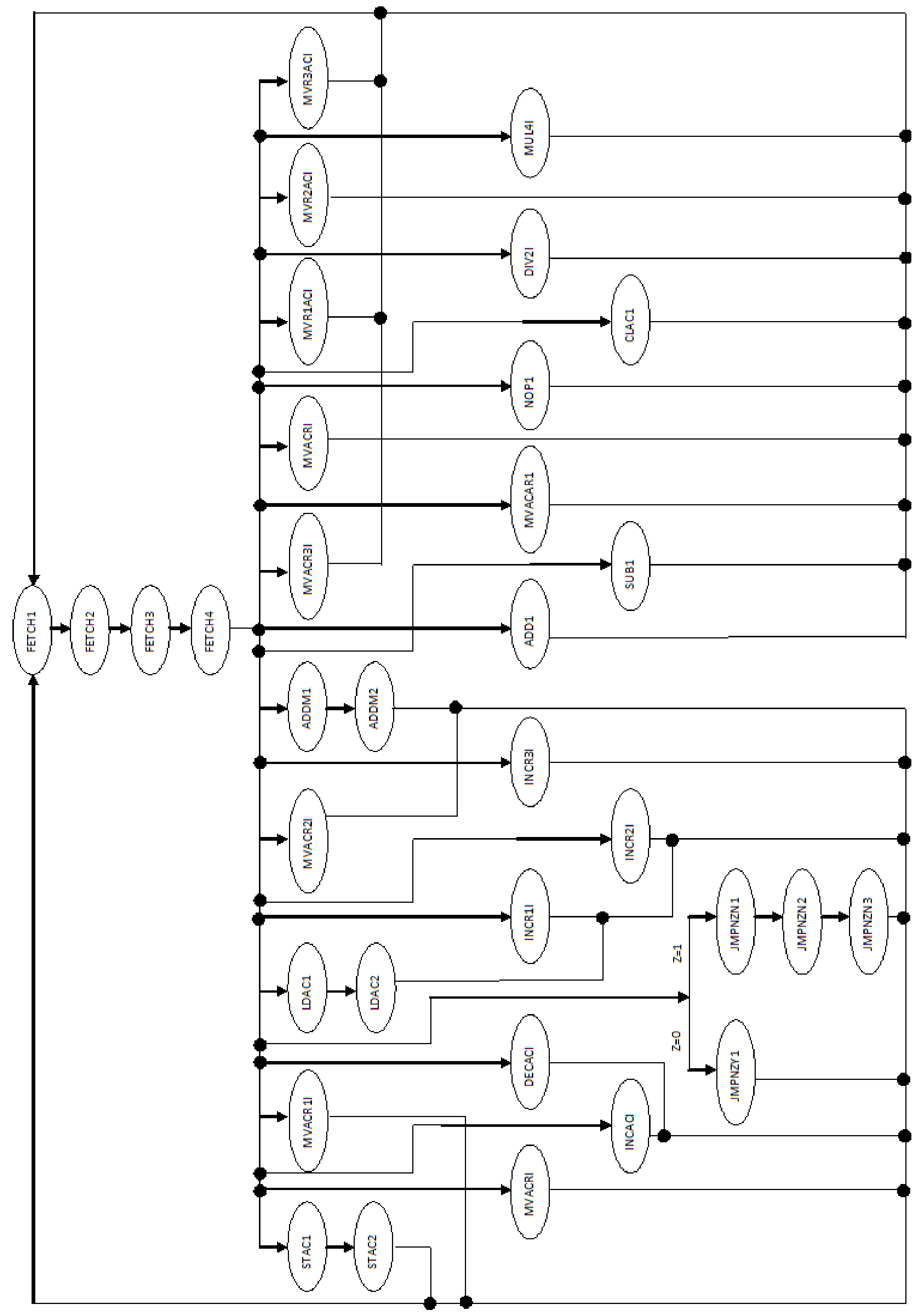


Figure 2.3: State Diagram of the CPU

3.MODULES

3.1.REGISTERS

3.1.1.Register without increment

These are the modules that are used to store data temporally during the process cycle. Each register can store 16 bit data. Data stored in the register is always available at the ‘d_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. Register without increment figure is shown in below,

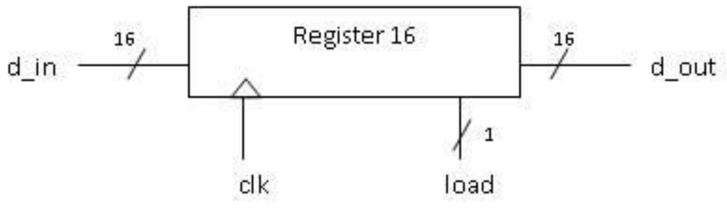


Figure 3.1: Block Diagram of Register 16

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 ‘d_in’ to the register at the positive edge of the clock.

The architecture that has been designed uses a similar type of 8 bit register for the Instruction Register (IR).
(Refer Appendix F for the Verilog code)

3.1.2. Register with increment

Structure of this module is same as the register without 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. Figure of this module is shown below,

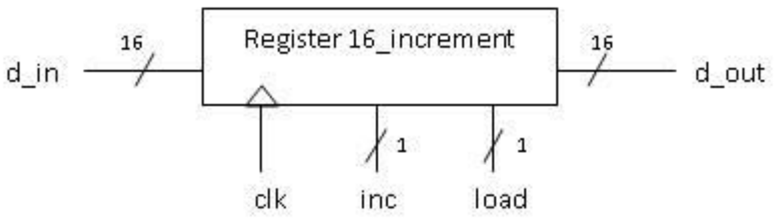


Figure 3.2: Block Diagram of Register 16_increment

(Refer Appendix G for the Verilog code)

3.2. DE-MULTIPLEXER (DEMUX)

In the designed architecture there is only one data bus. So it can only read data from one register at a time to the bus. This implementation has been done by using a de-multiplexer. DEMUX figure is shown in bellow.

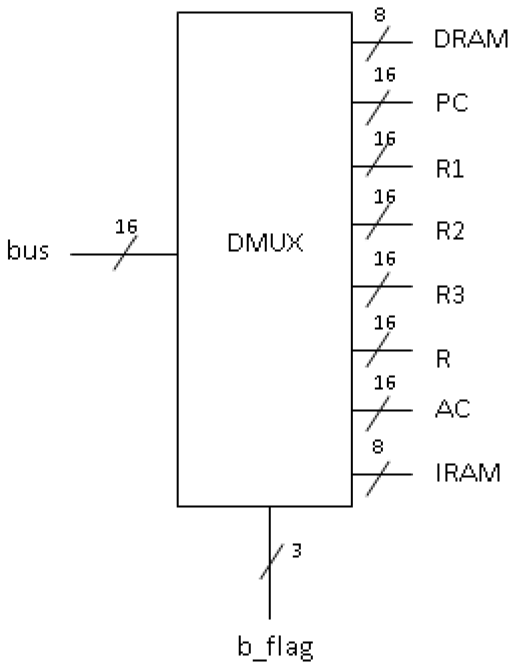


Figure 3.3: Block Diagram of DMUX

In the architecture, registers such as PC, R1, R2, R3, R and AC are connected to the bus and it can read data from those registers. And both IRAM and DRAM are also connected to the bus as well in order to read data from RAM to the bus. According to the architecture, bus is 16 bit but the IRAM and the DRAM is only 8 bit. Therefore additional 8 zeros have been added in front of the both RAMs when reads the data to the bus. In the multiplexer it has been used 3 bit flag to select the register or the RAM that has to be read to the bus. There are only 8 components (Registers, RAMs) connected to the bus therefore 3 bit flag is sufficient for this purpose.

Flag is set as below to read data from registers to the bus,

DRAM	000	R2	011	AC	110
PC	001	R3	100	IRAM	111
R1	010	R	101		

(Refer Appendix H for the Verilog code)

3.3.ALU

All the arithmetic and logical operations are done through this module. It has 2 inputs which are A bus and B bus. Those are 16 bit each. A bus is the output of AC register. Which means AC is directly connected to the ALU. And there is only one output which is C bus. It is also a 16 bit bus. This output is directly connected to the input of the AC register. Therefore any output of an ALU operation can be written only to AC register.
In order to do an ALU operation following step can be taken,

1. Store one operand in register ‘R’
2. Take the other operand into ‘AC’ (Accumulator)
3. Do the ALU operation

Now the operand in the AC register will be the input ALU from the A bus and the value stored in R bus will be the input to the ALU from the B bus. Then the ALU operation will be done according to the ALU op flag in the module. It is a 3 bit flag therefore only 8 operations can be done from this ALU module. These operations are decided by the control unit according to each instruction. Those operations are as follow,

Operation	Op-Flag	
ADD	000	$C \longleftarrow A + B$
SUB	001	$C \longleftarrow A - B$
PASS	010	$C \longleftarrow B$
ZER	011	$C \longleftarrow 0$
DECA	100	$C \longleftarrow A - 1$
MUL4	101	$C \longleftarrow B*4$
DIV2	110	$C \longleftarrow B/2$

In this module z flag is used to indicate whether the output of the ALU is zero or not. This was very useful when we implement JUMPNZ instruction in the control unit. In there before every JUMPNZ instruction we had to use SUB (Subtraction/ AC $\longleftarrow$ AC - R) instruction which writes it output to the AC. In order to keep the z flag high until the next instruction comes, the trick we used was to set the z flag by checking the output of the AC (A_bus). Then if it is zero, z flag will be set to high.

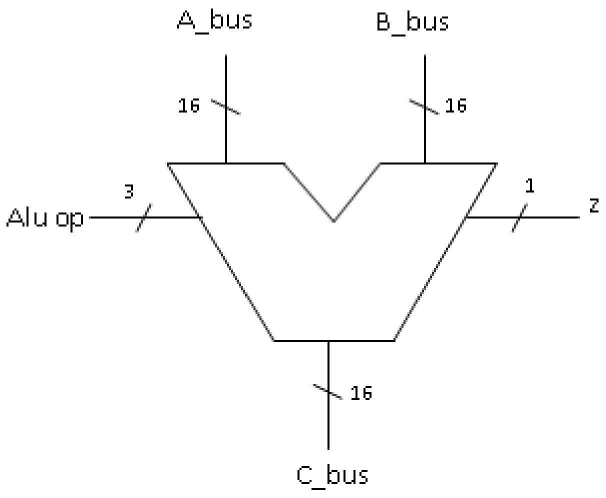


Figure 3.4: Block Diagram of ALU

(Refer Appendix I for the Verilog code)

3.4.CLOCK DIVIDER

This is the module that reduces the frequency of the original clock. We used an inbuilt 100MHz clock in the Spartan 6 FPGA development board. In the processing cycle we had to perform reading data from registers and calculating values trough the ALU in between the negative and positive edges of the each clock cycle. So for a 100MHz clock this gap is 5ns. It might not be sufficient to perform all the tasks required. So it is necessary to widen the gap between positive and negative edge of the clock. For this clock divider module has been implemented. It takes the original clock as the input and returns a divided (frequency reduced) clock as the output. All the components inside the processor use this divided clock as their input so if you want to stop the processing you can do it by setting enable pin as 'Low'.

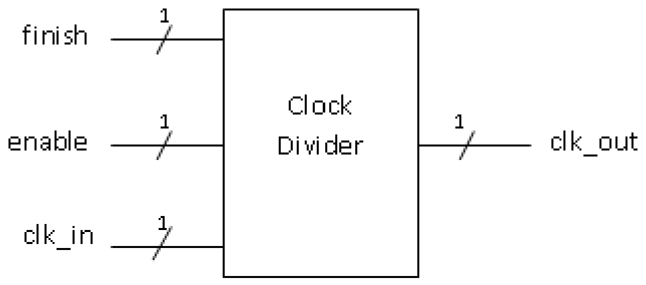


Figure 3.5: Block Diagram of Clock Divider

(Refer Appendix J for the Verilog code)

3.5. STATEMACHINE

This is the heart of the processor. All the controlling signal generating is done by this unit. It has three inputs which is clock, one bit z value, and IR which is 8 bit wide. This module generates 10 different controlling signals to control the processing cycle. In this output signals there are seven 1 bit data paths, two 3 bit data paths and one 8 bit data path. All the output controlling signals are shown in below,

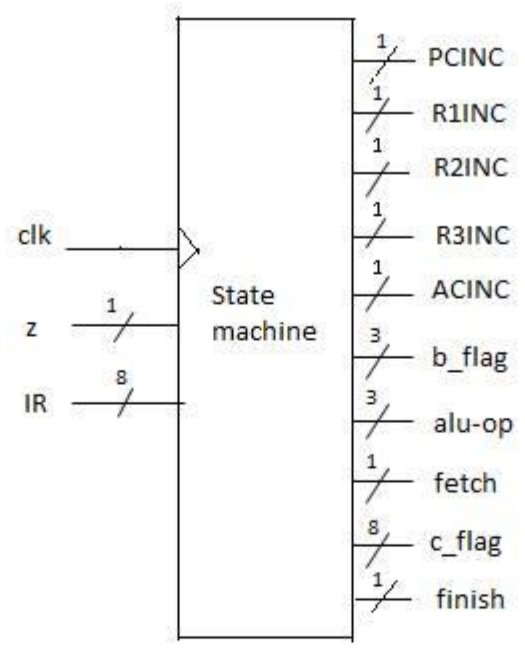


Figure 3.6: Block Diagram of State Machine

- **PCINC** - used to increment value of the saved data in PC register by 1
- **R1INC** - used to increment value of the saved data in R1 register by 1
- **R2INC** - used to increment value of the saved data in R2 register by 1
- **R3INC** - used to increment value of the saved data in R3 register by 1
- **ACINC** - used to increment value of the saved data in AC register by 1
- **b_flag**

  This is 3 bit data path is directly connected to the de-multiplexer which we have mentioned before. Its main task is to differentiate which data has to be gone through the bus from registers and RAMs. As we know 3 bits can represent 8 different combinations so this flag selects one data path from 8 different data paths which are the outputs of registers and RAMs. The list of data paths are given in *Table 3.1*.

Control Signal (b_flag)	**Register which gives Output to B bus**
000	M
001	PC
010	R1
011	R2
100	R3
101	R
110	AC
111	IM

Table 3.1: Data paths of the processor

- **alu-op**

  This is also 3 bit data value which controls the Arithmetic and Logic Unit operations. ALU does its operations according to this control signal. Therefore all the arithmetic and logic operations are controlled by this 3 bit signal. ALU operations and controlling signal which has to be given by the state machine is given in *Table 3.2*

Control Signal (alu-op)	**ALU operation for given signal**
000	ADD
001	SUB
010	PASS
011	ZER
100	DECAC
101	MUL4
110	DIV2
111	-

Table 3.2: ALU operations

- **fetch**– used to enable data write from bus to IR (Instruction register).

- **C_flag**

  This control signal has 8 bit to control all the necessary registers in the processor. This control signal enable writing process into a register from the bus. When some data has to be stored in a register this signal gives writing enabling signal to that register which gives access to get data from the bus to register. These 8 bits are directly connected to each registers' load pin. If this load pin is high register gets the value of the bus to its memory at the positive edge of the clock. Control signal which generates from the state machine for c_flag is given in ***Figure 3.7***.

AR	PC	R1	R2	R3	R	AC	M

Figure 3.7: Control signals of c_flag

Multiple writes to several registers at once can be done by using this control signal.

- **finish**– used to indicate the end of the processing state.

In the above section we have considered about output control signals and now let's take a look at the input data signals. In this module there are three input signals present and the input main data for the module is IR value from the IR register which gives the next instruction to be executed.

All the instructions have one or more microinstructions. In this module every microinstruction is a separate state. We had 25 instructions and altogether around 40 microinstructions. Each of these microinstruction states has its unique 6 bit value. In every state we have assigned how the outputs should be controlled. And also at every state we have assigned the correct next state as well. For an example in every instruction, last microinstruction state assigns its next state as 'FETCH1', which is the first state of the fetch cycle.

We have used the unique number that represents the first microinstruction of a particular instruction as the assembly instructions to our processor.

All the instructions and its representing values are given below. These instructions and values are decoded in the state machine.

Instructions	Encoding Values
CLAC	3
STAC	4
MVACR1	6
MVACR2	7
MVACR3	8
MVACR	9
MVR1AC	10
MVR2AC	11
MVR3AC	12
MVRAC	13
INCAC	14
INCR1	40
INCR2	41
INCR3	42
LDAC	15
SUB	17
DECAC	18
ADD	19
DIV2	20
MUL4	21
JUMPNZ	22
ADDM	27
NOP	58
MVACAR	59
END	60

Table 3.3: Instructions and their encoding values

In each fetching cycle, state machine gets IR value (unique number) from the IR register which has stored the next instructions from the instruction ram. At the last state of the fetch cycle it assigns the next state as the value it took from IR. Then in the next clock cycle that state will be executed. As I mentioned before at the last state of the each instruction it assigns next state as FETCH1 so that we can load the next instruction to be executed.

There is an instruction called 'END' which contains only one microinstruction state. And its task is to set the finish flag high. Then the clock signal to the processor would stop and processing state would be finished.

(Refer Appendix K for the Verilog code)

3.6. PROCESSOR

This module contain the all the instances of the modules used for the processing part and controlling part. This module does not contain instances of memory modules and communication related modules. This is the CPU of our design and this has four inputs which is clock, processor enable, DRAM, and IRAM.

- Clock - gives clock pulse for the synchronization
- Enable – enables the processor and starts the processing
- DRAM – gives dram data to the processor (8 bit)
- IRAM – gives iram data to the processor (8 bit)

For this module it has six output data path which are cout, write, finish, DRAM-addr, IRAM-addr and DRAM-data.

- Cout - output of the clock divider (frequency reduced clock)
- write -Write enable signal to the DRAM
- finish -used to indicate the end of the processing
- DRAM-addr - gives memory location of the DRAM (16 bit)
- IRAM-addr -gives memory location of the IRAM (8 bit)
- DRAM-data -gives the data value which needs to be written in to the DRAM memory (8 bit)

Complete diagram of the processor with inputs and outputs is given below.

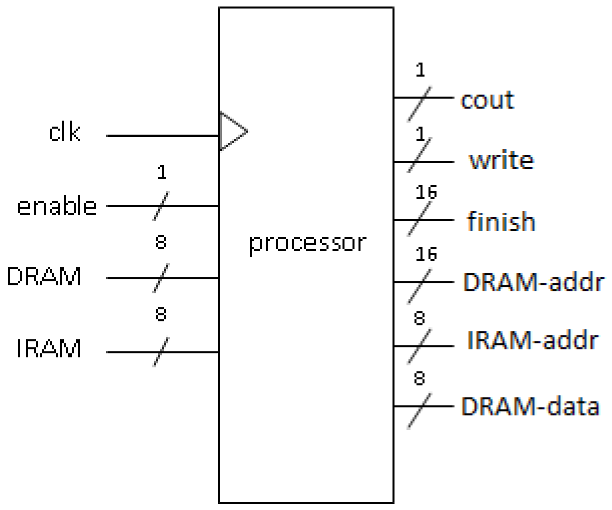


Figure 3.8: Block Diagram of processor

These are the modules integrated in this processor module.

- CLOCK_DIVIDER
- STATE_MACHINE
- DEMUX
- REGISTER(16bit)
- REGISTER_INC(16bit)
- REGISTER(8bit)
- ALU

(Refer Appendix L for the Verilog code)

3.7. COMMUNICATION RELATED MODULES

3.7.1.Board Rate Generator

This module is also operating as a clock divider to match the board rate of the serial communication process. It takes the original clock as the input and returns ‘max_tick’ as the output.

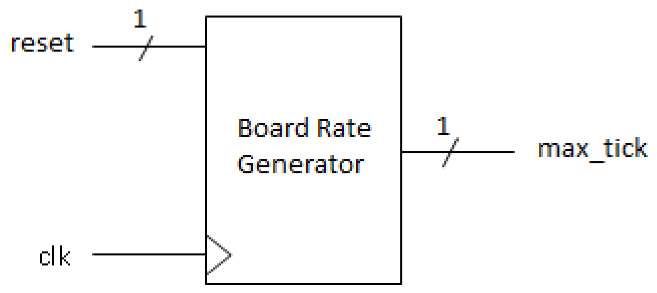


Figure 3.9: Block Diagram of Board Rate Generator

(Refer Appendix M for the Verilog code)

3.7.2.UART_RX

This module was created to receive the serial data and output one byte at a time. By using ‘rx’ pin it receives the data bit by bit and when it receives 8 bits, it generates a word and returns it as the output while setting the ‘rx_done_tick’ High

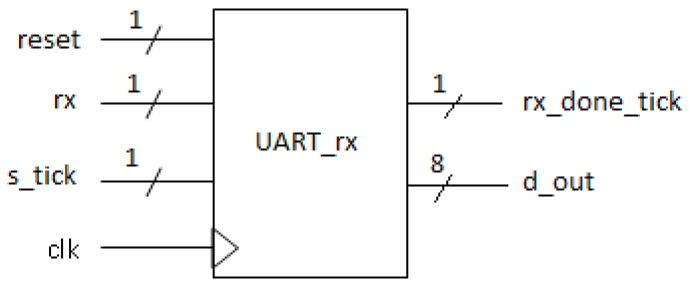


Figure 3.10: Block Diagram of UART_RX

(Refer Appendix N for the Verilog code)

3.7.3.UART_TX

This module was created to transmit datafrom the FPGA to the computer via serial data communication. It takes byte by byte as its input and transmits one bit at a time via 'tx' pin. When it finishes transmitting all the 8 bits of a particular byte it sets the 'tx_done_tick' as '1'.

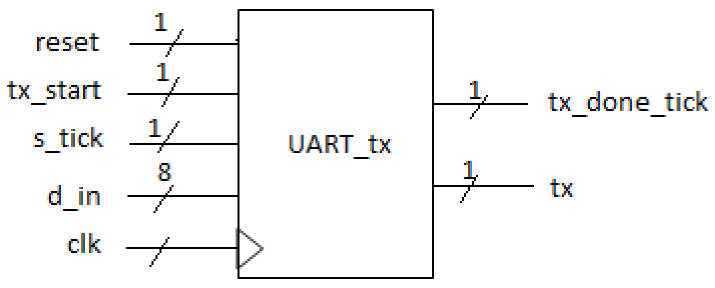


Figure 3.11: Block Diagram of UART_TX

(Refer Appendix O for the Verilog code)

3.7.4.UART_ADDR

This is the module which controls the addresses to the RAM from communication section. It sequentially increases the address when transmitting and receiving data. After receiving is done it sets 'p_enable' (Processor Enable) to high so that the processor can start its task.

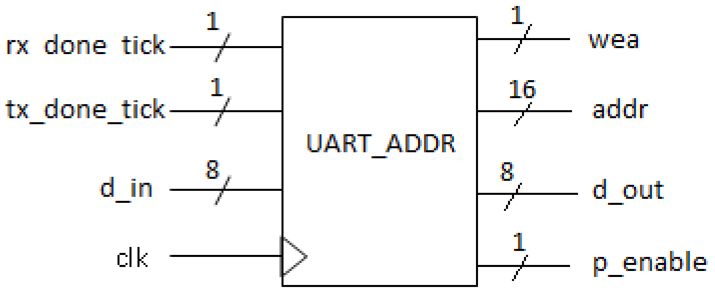


Figure 3.12: Block Diagram of UART_ADDR

(Refer Appendix P for the Verilog code)

3.7.5.Selection

This is the module which controls and connects the communication part and processor. According to architecture processor contains a clock divider module which changes the frequency and all the processing cycle is working according that divided clock. But communication section is handled using the original clock.

Both processing and communicating modules use the same RAM to store and read the data. Since RAM has only one port each for data in, write enable and address it causes problems to select which address should we select while processing and which should be selected while communicating. So to avoid this problem we created this module. By taking ‘p_enable’ (Processor Enable) and ‘p_finish’ (Processing finished) flags as its input it changes its states. (Receiving State/ Processing State/ Transmitting State). According to each state it returns above mentioned outputs in order to maintain and control the entire system.

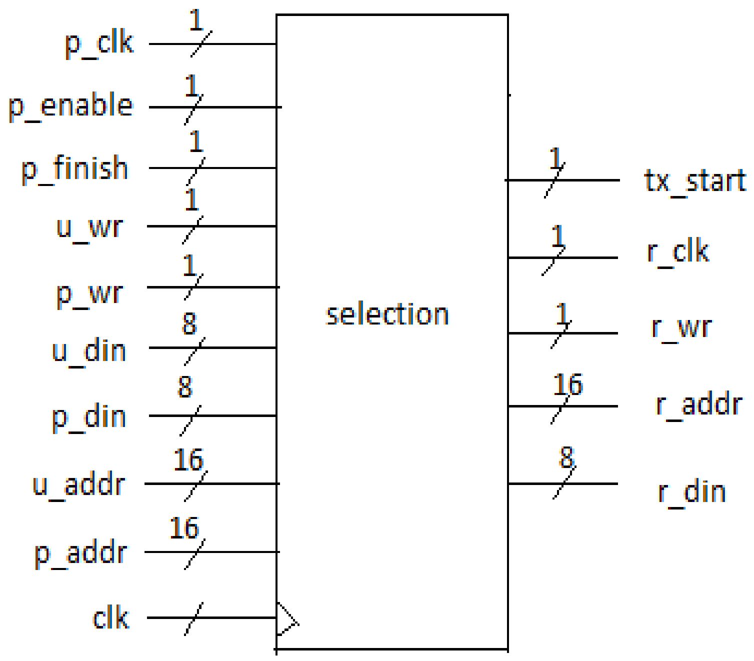


Figure 3.13: Block Diagram of selection

(Refer Appendix Q for the Verilog code)

3.8. MEMORY MODULES

3.8.1.D-Ram

This module is the main data memory which gives space to save image data for processing purposes. After reading data from the UART Bridge and converting serial data to bytes all the data units are saved in the dram module for further operations. This module has 65536 memory locations with a width of 8 bit to store the pixels of the image. Each pixel has integer values which change from 0 to 255. Therefore 8 bit width of a memory unit enough to save image pixels in dram module. This module has four input data paths and one output data path. “d_in” input is used to give input data which has 8 bit data stream and “address” path gives the address value which the dram has to read or write data and it contains a 16 bit data stream. “wrt_en” input which has one bit, gives control signal for dram when control unit needs to write data to the memory from d_in data path and also clk gives the synchronizing signal to do operations in the dram. “d_out” used to give output data to the data bus when request from the control unit. Dram module’s figure is shown below,

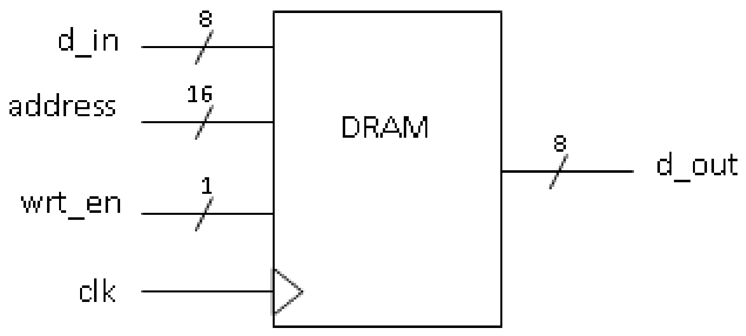


Figure 3.14: Block Diagram of DRAM

In this module 65536 data units (bytes) can be stored and therefore the maximum data pixels can be stored is 65536 pixels of an image. Therefore 16 bit address is needed to represent each data unit and 16 bit data address is used by our design to fulfill this requirements. Internal data storage of dram can be shown as following diagram,

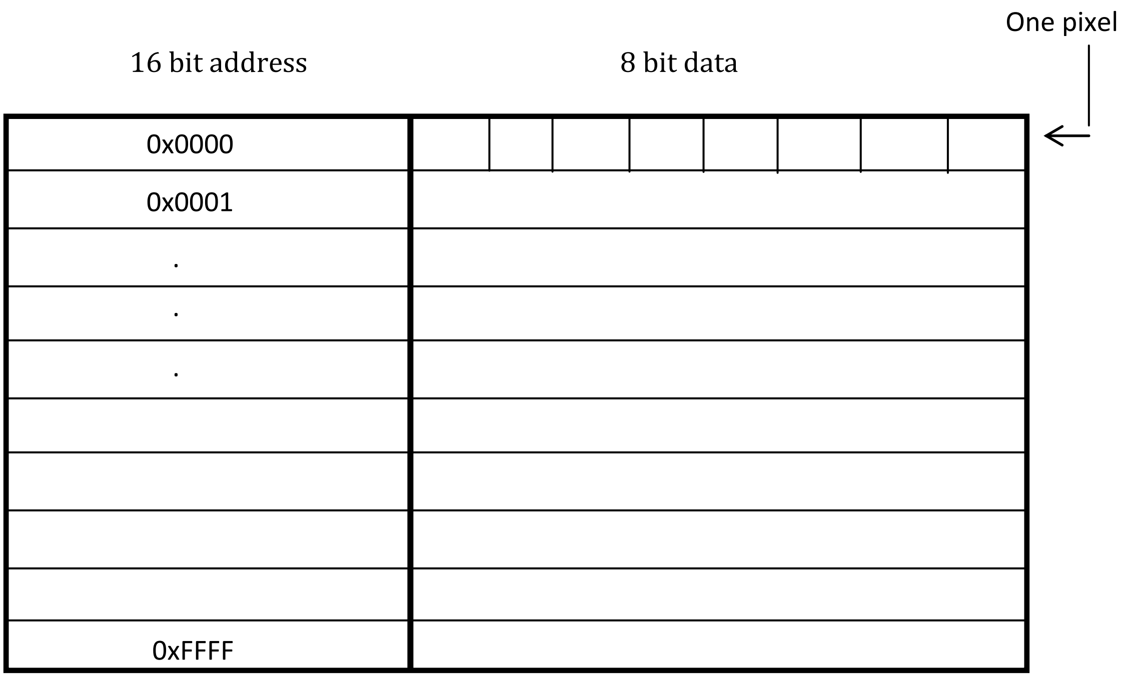


Figure 3.14: Block Diagram of DRAM

In this project Xilinx Memory Block generator is used to generate the memory module in the FPGA.

The Xilinx LogiCORE™ IP Block Memory Generator (BMG) core is an advanced memory constructor that generates area and performance-optimized memories using embedded block RAM resources in Xilinx FPGAs. Users can quickly create optimized memories to leverage the performance and features of block RAMs in Xilinx FPGAs.

The BMG core supports both Native and AXI4 interfaces.

The Native interface BMG core configurations support the same standard BMG functions delivered by previous versions of the Block Memory Generator (up to and including version 4.3). Port interface names are identical.

The AXI4 interface configuration of the BMG core is derived from the Native interface BMG configuration and adds an industry-standard bus protocol interface to the core. Two AXI4 interface styles are available: AXI4 and AXI4-Lite.

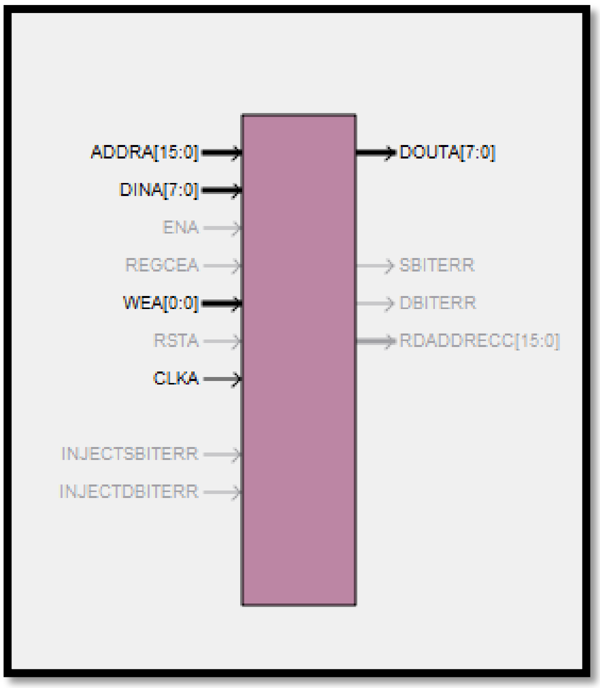


Figure 3.15: Block Memory Diagram for DRAM

3.8.2.I-Ram

The main feature of this module is store instruction in the memory. All the instruction which is coded by assembly language is stored in this memory. When the processor needs instructions it fetches instructions from this instruction ram. This module has saved all the instructions in order to given assembly code. This module consists of instructions to be executed with 256 memory locations and a width of 8 bits. This basically contains the assembly code of the algorithm for filtering and down sampling the image.

This module has only two inputs and one for data input and other one is clock for synchronizing operations. Also instruction ram has one output for data output we needed. In this task we needed about 200 operations. Therefore this instruction ram is designed with 8 bit width data memory and stored one instruction in one memory location. The instruction ram’s figure is show below,

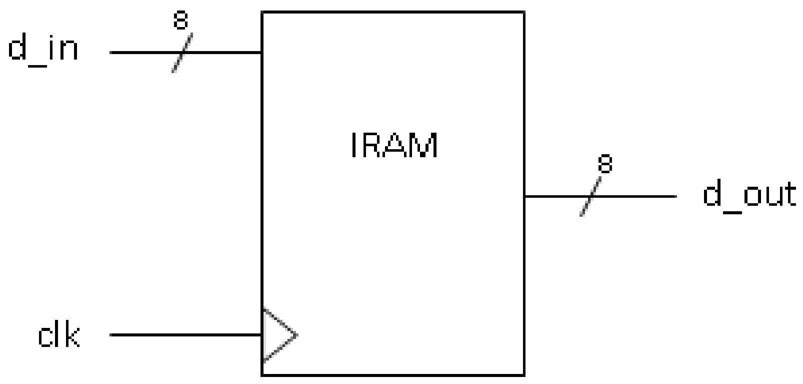
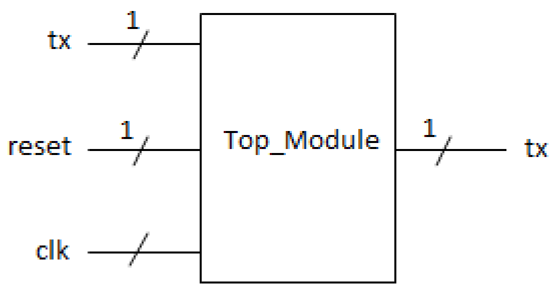


Figure 3.16: Block Diagram of IRAM

We have designed this module also with the Xilinx LogiCORE™ IP Block Memory Generator (BMG) and we store our assembly instructions by loading a 'COE file' to this module.

3.9.TOP MODULE

This is the module which connects communication and processing modules. All the instances have been created inside this module. It has a clock signal and a reset pin as its inputs. And also 'rx' pin for receiving the data coming serially. And it has only one output which is 'tx' to transmit the data serially. It contains instances of Processor, UART_RX, UART_TX, UART_ADDR,



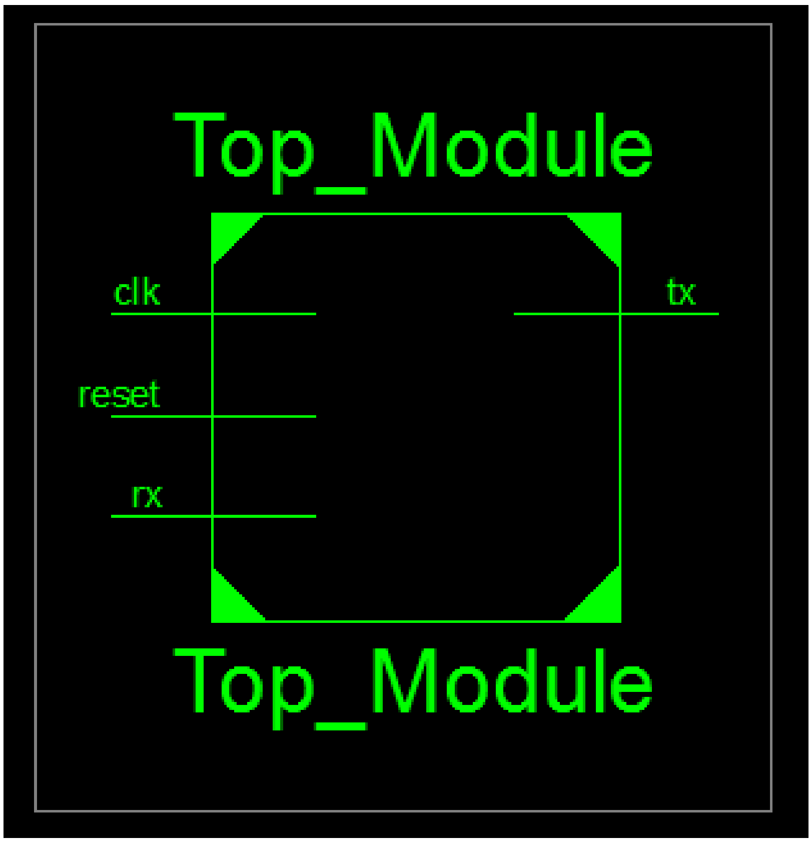


Figure 3.16: Block Diagram of Top Module

(Refer Appendix R for the Verilog code)

3.10. RTL VIEW OF THE PROCESSOR

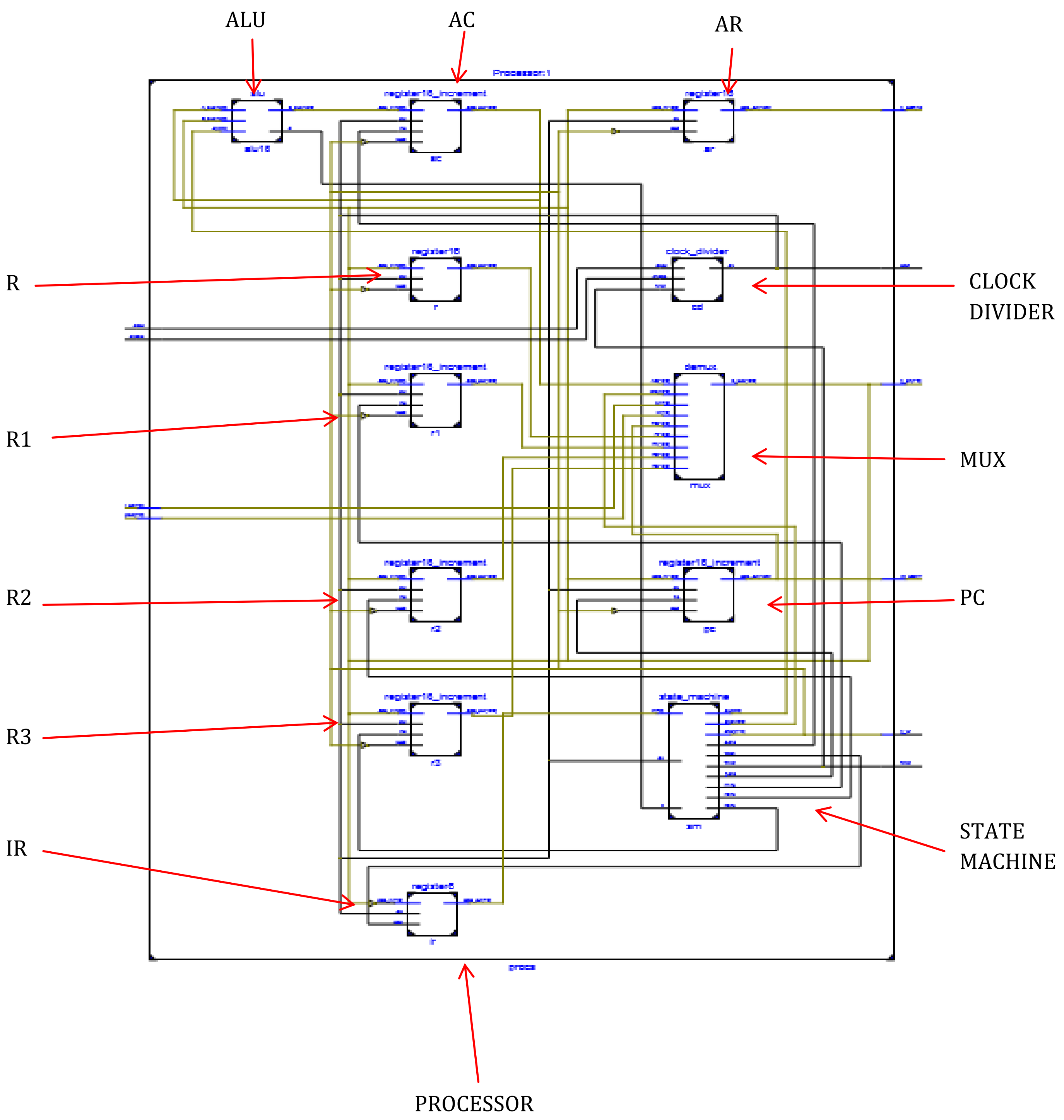


Figure 3.18: RTL view of the processor

3.11. RTL VIEW OF THE TOP MODULE

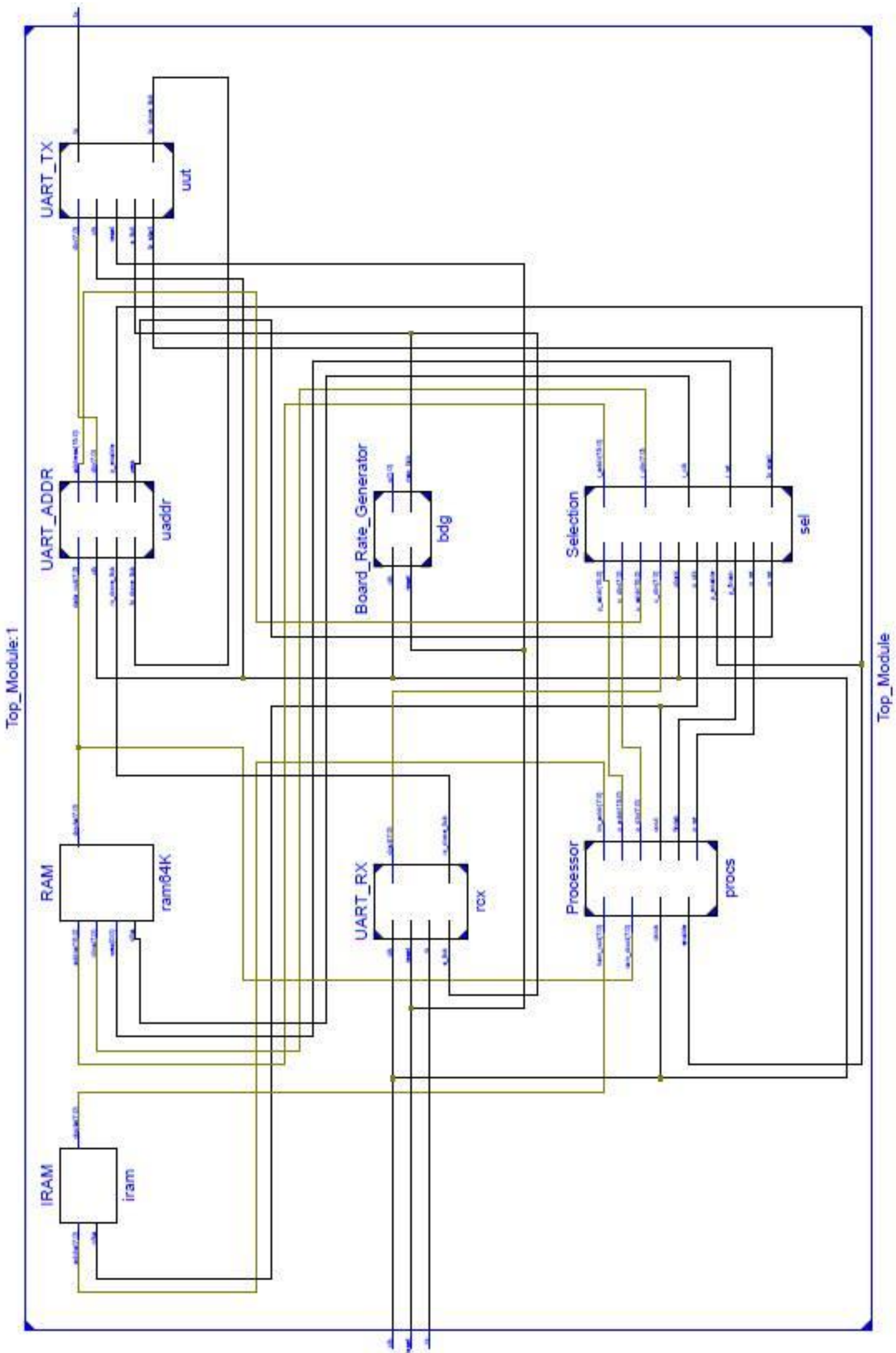


Figure 3.19: RTL view of the top module

4.ALGORITHMS

Algorithms for the processing part of our project is developed based on two main concepts, which are filtering the image and down sampling. In the algorithm it first takes the array of pixel values of the photo and then it will be filtered using a Gaussian filter. Then the filtered photo will be down sampled using a down sampling algorithm. Algorithm was initially generated and tested using Matlab.

4.1. FILTERING ALGORITHM

For the filtering part, 3x3 Gaussian kernel has been used and the weight distribution of the kernel is shown below. (***Figure 4.1***)

1	2	1
2	4	2
1	2	1

Figure 4.1: Gaussian Kernel

After overlapping the center pixel of this kernel with an image pixelwe can calculate a weighted summation value. Then to normalize that value it should be divided by the total weight of the kernel which is 16. Then the average value will be stored in the RAM (at the top left corner pixel location). Graphical illustration of the filter is given below. (Figure 2)

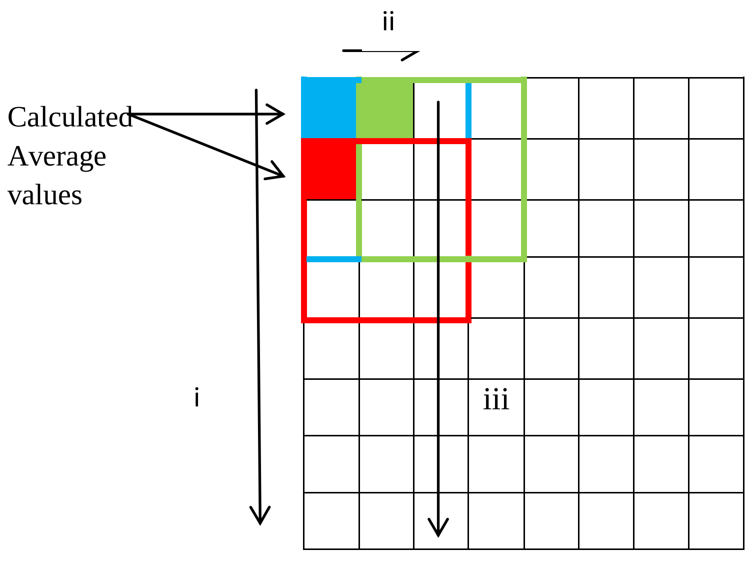


Figure 4.2: Graphical Illustration of the filtering method

As in the figure, first the kernel will calculate average value of that image section and store it in the top left corner pixel. Then the kernel will move downward as shown in the figure 2 and store the average values in incremental manner. After finishing the downward pixels, kernel will move to the next column and store the average value as shown in the figure 2. Likewise the filtered image can be stored in the RAM in sequential manner.

Respective pseudo code (Matlab) for the 256x256 image filtering algorithm is given below,

```
for (j=1 : 254)          // loop going through columns
for(i=1 : 254)           // loop going through rows
x = j*256+i;       //address of the image pixel
total = total+im(x)*4;
total = total+im(x-1)*2;
total = total+im(x+1)*2;
total = total+im(x-255-1)*2;
      total = total+im(x-255-1-1);  //convolved values
      total = total+im(x-255);
total = total+im(x+255+1)*2;
total = total+im(x+255);
total = total+im(x+255+1+1);
total = total/16; //average total convolved value
      k = (j-1)*256+i-1;//address of RAM to store
      im(k) = total;//writing average value to the RAM
total = 0;
end
end
```

In the pseudo it will assign memory address of the image to x and then in the below it will load the respective values from the RAM and multiply it with respective kernel weights. Then the total value will be divided by the total weight of the kernel. Then that value will be stored in the RAM according to memory address ‘k’.

4.2. DOWN SAMPLING ALGORITHM

In order to down sample the image into half the size (height and width) of the original image, this algorithm will take one value per four pixels from the filtered image and store it in the memory to get the output. Values taken from 8x8 filtered image is shown in the Figure 4.3

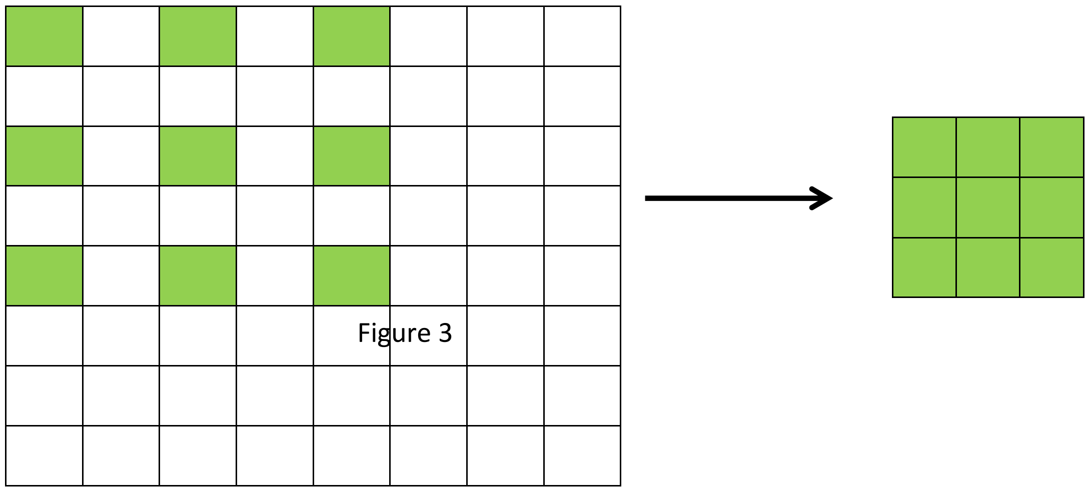


Figure 4.3: Graphical Illustration of the Down-sampling method

Respective pseudo code (Matlab) for down sampling a 256x256 image is given below,

```
k = 0;       // setting writing memory address to zero
for j=0:1:126      // loop going through column
fori=0:1:126// loop going through rows
        y = (2*i+512*j);// taking selected RAM addresses
im(k) = im(y);     // store respective values in RAM
        k = k+1;   // increment writing memory address
end
end
```

4.3. ASSEMBLY CODE FOR THE ALGORITHM

In order to develop the assembly code for the algorithm that we have implemented, following instructions of the processor has been used.

CLAC	MVR3AC	INCR3	JUMPNZ
STAC	MVR2AC	LDAC	ADDM
MVACR3	MVR1AC	SUB	NOP
MVACR2	MVRAC	DECAC	END
MVACR1	INCAC	ADD	
MVACR	INCR1	DIV2	
MVACAR	INCR2	MUL4	

Assembly code for down sample a 256x256 image is given in below.

We set the following registers as the variables for the algorithm,

R1 = i, R2 = x, R3 = total

1. CLAC
2. MVACR — Initiating register values to zero
3. MVACR1
4. MVACR2
5. MVACR3
6. INCR1
7. INCR2
8. MVR2AC
9. MUL4 — Getting equation for variable x
10. MUL4
11. MUL4
12. MUL4
13. MVACR

```
14. MVR1AC
15. ADD
16. MVACR    ┐
17. CLAC     │
18. ADDM     ├        Store variable j in the data memory
19. "255"    │
20. MVACAR   ┘
21. MVR2AC
22. STAC
23. MVRAC    ┐        Store x value in the R2 resistor
24. MVACR2   ┘
25. LDAC
26. MUL4
27. MVACR3
28. MVR2AC
29. DECAC
30. LDAC
31. MUL4
32. DIV2
33. MVACR
34. MVR3AC
35. ADD
36. MVACR3
37. MVR2AC
38. INCAC
39. LDAC
40. MUL4
41. DIV2
42. MVACR
43. MVR3AC
44. ADD
45. MVACR3
46. CLAC
47. ADDM
```

48. “255”
49. MVACR
50. MVR2AC
51. SUB
52. DECAC
53. LDAC
54. MUL4
55. DIV2
56. MVACR
57. MVR3AC
58. ADD
59. MVACR3
60. CLAC
61. ADDM
62. “255”
63. MVACR
64. MVR2AC
65. SUB
66. DECAC
67. DECAC
68. LDAC
69. MVACR
70. MVR3AC
71. ADD
72. MVACR3
73. CLAC
74. ADDM
75. “255”
76. MVACR
77. MVR2AC
78. SUB
79. LDAC
80. MVACR
81. MVR3AC

82. ADD
83. MVACR3
84. CLAC
85. ADDM
86. “255”
87. MVACR
88. MVR2AC
89. ADD
90. INCAC
91. LDAC
92. MUL4
93. DIV2
94. MVACR
95. MVR3AC
96. ADD
97. MVACR3
98. CLAC
99. ADDM
100. “255”
101. MVACR
102. MVR2AC
103. ADD
104. LDAC
105. MVACR
106. MVR3AC
107. ADD
108. MVACR3
109. CLAC
110. ADDM
111. “255”
112. MVACR } Store calculated Gaussian vale in the memory
113. MVR2AC
114. ADD
115. INCAC

116. INCAC
117. LDAC
118. MVACR
119. MVR3AC
120. ADD
121. DIV2
122. DIV2
123. DIV2
124. DIV2
125. MVACR3
126. CLAC
127. ADDM
128. “255”
129. LDAC
130. MVACR2
131. DECAC
132. MUL4
133. MUL4
134. MUL4
135. MUL4
136. MVACR
137. MVR1AC
138. ADD
139. DECAC
140. MVACAR
141. MVR3AC
142. STAC
143. CLAC
144. MVACR3
145. INCR1
146. MVR1AC
147. MVACR
148. CLAC
149. ADDM

150. “255”
151. SUB
152. JUMPNZ
153. “8”
154. INCR2
155. MVR2AC
156. MVACR
157. CLAC
158. MVACR1
159. INCR1
160. ADDM
161. “255”
162. SUB
163. JUMPNZ
164. “8”

Now the image filtering part is done. Filtered image has stored in RAM. Following assembly instructions are for the sampling algorithm.

We set the following registers as the variables for the algorithm,

R1 = i, R2 = j, R3 = k

165. CLAC
166. MVACR1
167. MVACR2
168. MVACR
169. MVACR3
170. MVR1AC
171. MUL4
172. DIV2
173. MVACR
174. MVR2AC
175. MUL4
176. MUL4
177. MUL4

178. MUL4
179. MUL4
180. DIV2
181. ADD
182. LDAC
183. MVACR
184. MVR3AC
185. MVACAR
186. MVRAC
187. STAC
188. INCR3
189. MVR1AC
190. MVACR
191. CLAC
192. ADDM
193. “127”
194. SUB
195. JUMPNZ
196. “170”
197. INCR2
198. MVR2AC
199. MVACR
200. CLAC
201. MVACR1
202. ADDM
203. “127”
204. SUB
205. JUMPNZ
206. “170”
207. END

5. TESTING, SIMULATIONS & MODIFICATIONS

5.1. TESTING & SIMULATIONS

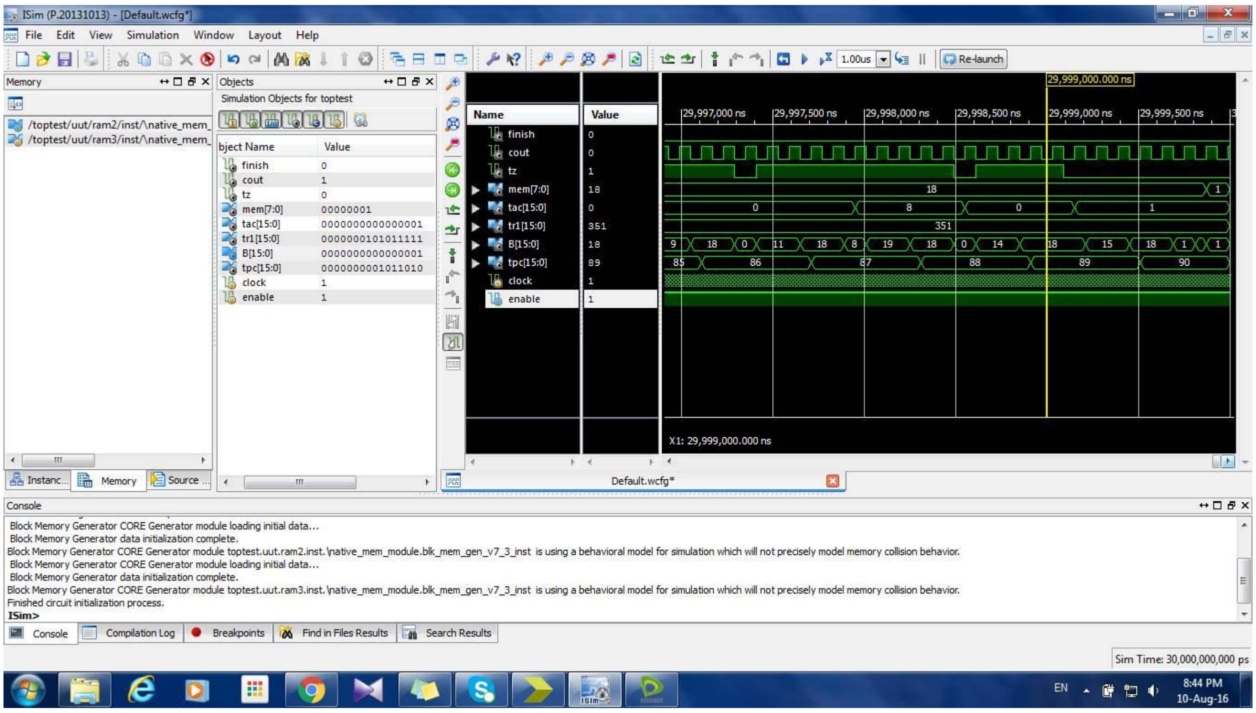
Firstly the modules inside the processor were created. For each module separate test codes were run to check the functionality of the module. (Unit Testing) To simulate the test results, software called 'ISim' was used. To create test modules Verilog syntaxes which are only supported simulation were used. Test code created to make a clock pulse is given below.

```
initial begin
        enable = 1;
        clock = 1;#5;
        while(1) begin
                clock= ~clock;
                #5;
        end
end
```

To verify that our processor and the other modules are working correctly temporary output were taken out for inspection. For an example

- 'tac'- temporary output which indicates the value of AC register
- 'tpc'- temporary output which indicates the value of PC register
- 'cout'- divided clock
- 'tz' – z flag
- 'finish' – finish bit

Some outputs from ISim are shown below.



bject Name	Value
finish	0
cout	1
tz	0
mem[7:0]	00000001
tac[15:0]	0000000000000001
tr1[15:0]	0000000101011111
B[15:0]	0000000000000001
tpc[15:0]	0000000001011010
clock	1
enable	1

Name	Value
finish	0
cout	0
tz	1
mem[7:0]	18
tac[15:0]	0
tr1[15:0]	351
B[15:0]	18
tpc[15:0]	89
clock	1
enable	1

```
Block Memory Generator CORE Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator CORE Generator module toptest.uut.ram2.inst.\native_mem_module.blk_mem_gen_v7_3_inst  is using a behavioral model for simulation which will not precisely model memory collision behavior.
 Block Memory Generator CORE Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator CORE Generator module toptest.uut.ram3.inst.\native_mem_module.blk_mem_gen_v7_3_inst  is using a behavioral model for simulation which will not precisely model memory collision behavior.
Finished circuit initialization process.
ISim>
```

Figure 5.1: Simulated Results

5.2. MODIFICATIONS

We have been asked to modify our processor to do a slightly different task and simulate our results at the evaluation.

Here is an example with simulated results.

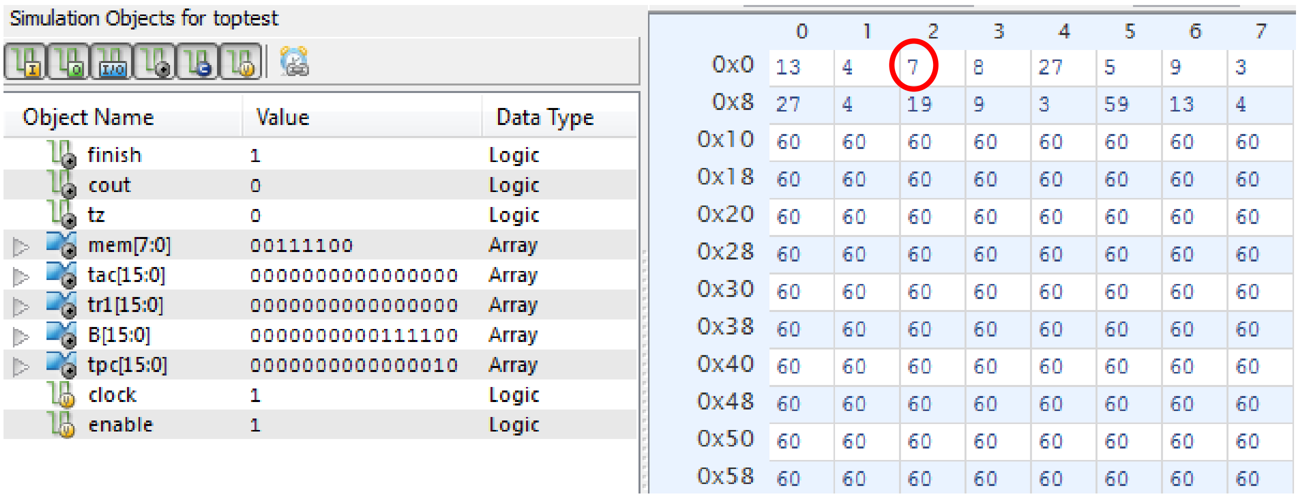
- **Change the assembly instruction 'ADD' to do the 'AND' operation**

  Given task was to change ADD operation to AND operation and simulate it. In order to do that ADD operation in ALU has been changed as followed.

  ADD: C_bus=A_bus + B_bus; ------ > AND: C_bus=A_bus&B_bus;

  To verify the change simple process has been done to calculate the AND value of 13 and 4. In order to do that '13' and '4' have been written in the first two locations of the data ram manually and take those values from the data ram when the process is running. Then the final vale has been stored in the 3rd memory location.

Data ram values before the processing is as follow,



Simulation Objects for toptest

Object Name	Value	Data Type
finish	1	Logic
cout	0	Logic
tz	0	Logic
mem[7:0]	00111100	Array
tac[15:0]	0000000000000000	Array
tr1[15:0]	0000000000000000	Array
B[15:0]	0000000000111100	Array
tpc[15:0]	0000000000000010	Array
clock	1	Logic
enable	1	Logic

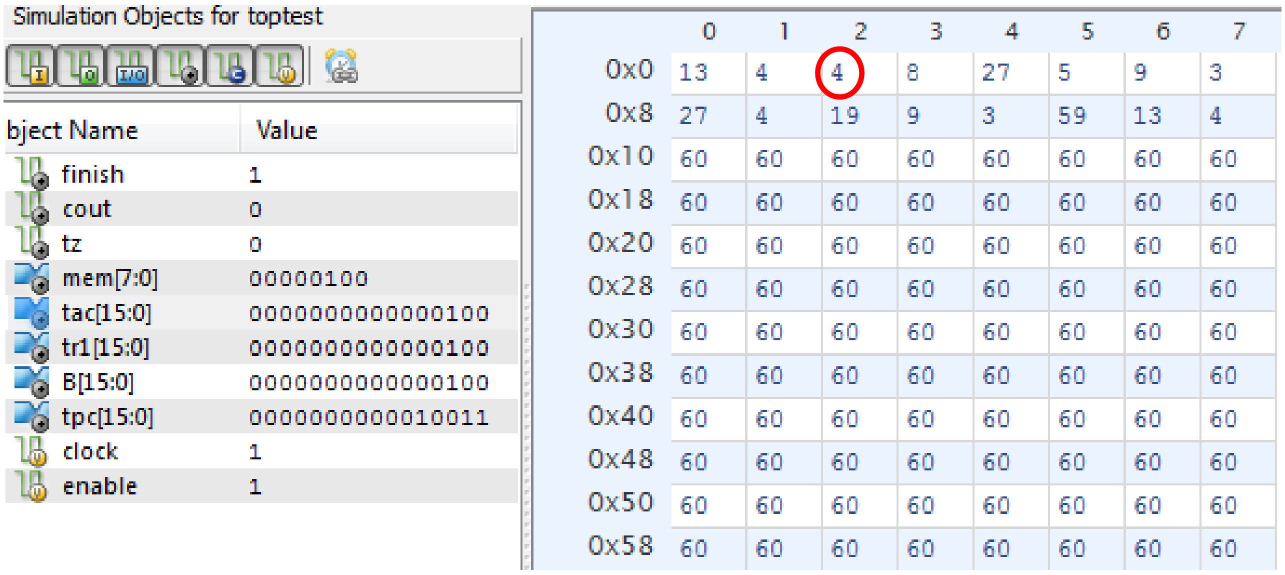
	0	1	2	3	4	5	6	7
0x0	13	4	7	8	27	5	9	3
0x8	27	4	19	9	3	59	13	4
0x10	60	60	60	60	60	60	60	60
0x18	60	60	60	60	60	60	60	60
0x20	60	60	60	60	60	60	60	60
0x28	60	60	60	60	60	60	60	60
0x30	60	60	60	60	60	60	60	60
0x38	60	60	60	60	60	60	60	60
0x40	60	60	60	60	60	60	60	60
0x48	60	60	60	60	60	60	60	60
0x50	60	60	60	60	60	60	60	60
0x58	60	60	60	60	60	60	60	60

Figure 5.2: DRAM before processing

Then the following op code has been written in the IRAM to do the process.

1. CLAC
2. MVACR1
3. MVACR2
4. MVACR3
5. MVACR
6. LDAC
7. MVACR
8. CLAC
9. TNCAC
10. LDAC
11. AND
12. MVACR1
13. CLAC
14. INAC
15. INAC
16. MVACAR
17. MVR1AC
18. STAC
19. END

Then the simulation has been done in the processor. DRAM has been changed as follows after the simulation.


Simulation Objects for toptest

bject Name	Value
finish	1
cout	0
tz	0
mem[7:0]	00000100
tac[15:0]	0000000000000100
tr1[15:0]	0000000000000100
B[15:0]	0000000000000100
tpc[15:0]	0000000000010011
clock	1
enable	1

	0	1	2	3	4	5	6	7
0x0	13	4	4	8	27	5	9	3
0x8	27	4	19	9	3	59	13	4
0x10	60	60	60	60	60	60	60	60
0x18	60	60	60	60	60	60	60	60
0x20	60	60	60	60	60	60	60	60
0x28	60	60	60	60	60	60	60	60
0x30	60	60	60	60	60	60	60	60
0x38	60	60	60	60	60	60	60	60
0x40	60	60	60	60	60	60	60	60
0x48	60	60	60	60	60	60	60	60
0x50	60	60	60	60	60	60	60	60
0x58	60	60	60	60	60	60	60	60

Figure 5.3: DRAM after the Simulation

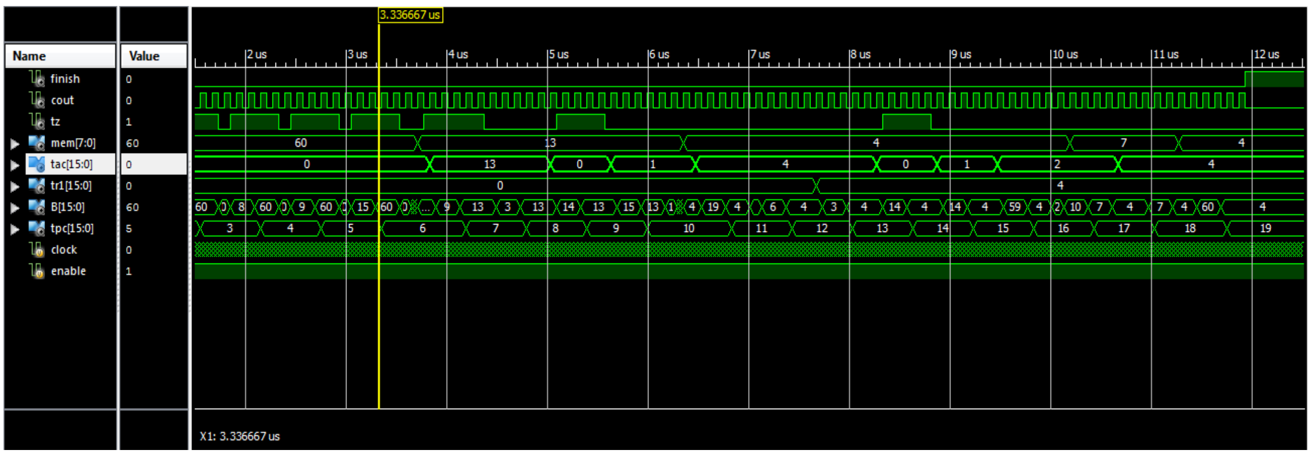

Name	Value
finish	0
cout	0
tz	1
mem[7:0]	60
tac[15:0]	0
tr1[15:0]	0
B[15:0]	60
tpc[15:0]	5
clock	0
enable	1

Figure 5.4: Simulated Results

Binary value of 13 - x - 00001101 Then x & y = 00000100 which 4 in radix 10.

Binary vale of 4 - y - 00000100

As in the above figure, DRAM has been changed after the simulation. 3rd location of the DRAM has been changed according to the answer of the process. AND operation between 13 and 4 should be 4 and that answer has been updated in the DRAM. Also from the time diagram the PC value has been incremented one by one as the AC value has been updated according to the op code that has been written for the process.

This has been proven that the processor that has been designed can be easily modified to do other operations.

6. RESULT ANALYZING AND VERIFICATION

After getting the processed data from the FPGA to the computer, analyzing the result is very important to get an idea about accuracy of the designed ISA and algorithms. To get the error rate and accuracy, first a reference image using MatLab is built and compared with the output of the microprocessor. After comparing the results, a clear idea can be obtained about the designed architecture's efficiency and accuracy.

6.1. GENERATE REFERENCE OUTPUT IMAGE

Using MatLab Software a reference output can be to compare results. Using the algorithms for Gaussian Filtering and down sampling the image, MatLab gives a correct result for the given image after processing. This can be used as our reference for the error detection and verification.

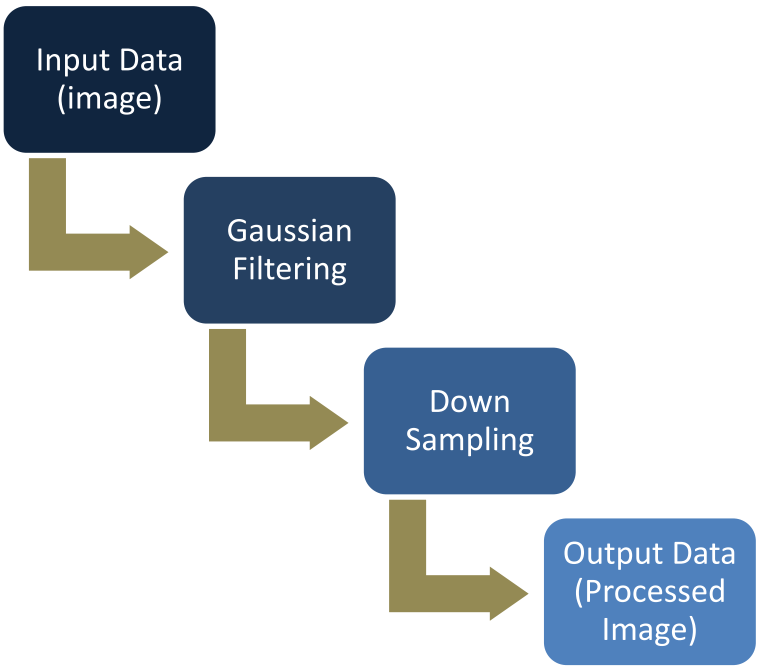


Figure 6.1:Flow Diagram for Making the Reference

In the first step the image is read and a one dimensional pixel array is constructed. This array is used in the processing part. (*Refer Appendix A for the Matlab code for the data input*). In the second step Gaussian Filtering method is used (*Refer Appendix B for the Matlab code*). After Gaussian Filtering the filtered data is used to get the down sampled image by performing the algorithm for down-sampling. (*Refer Appendix C for the Matlab code)* After generating the reference processed data the data are rearranged for display and verification purposes. (*Refer Appendix D for the Matlab code)*

Step by step results of generating the reference image are shown below

- **Original Image**

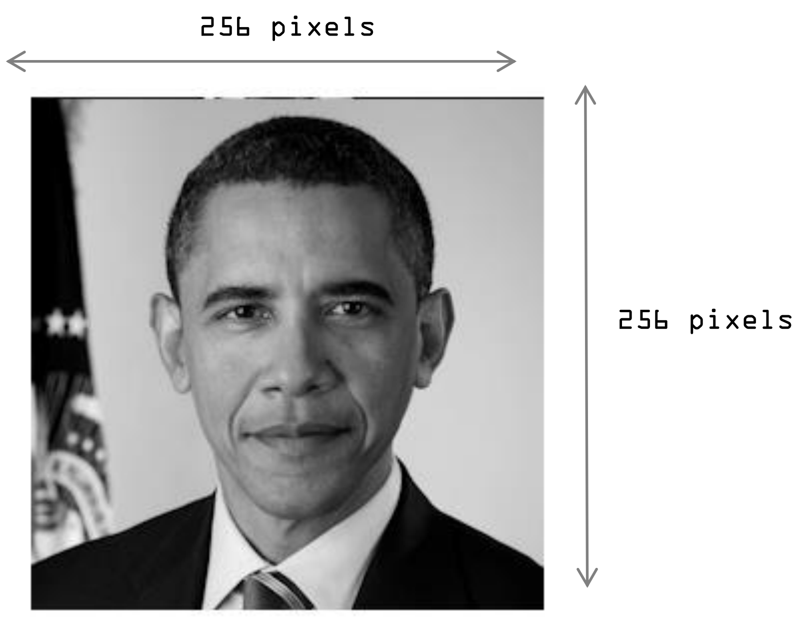


Figure 6.2: Original Image

- **Gaussian Filtered Image (Using MatLab)**

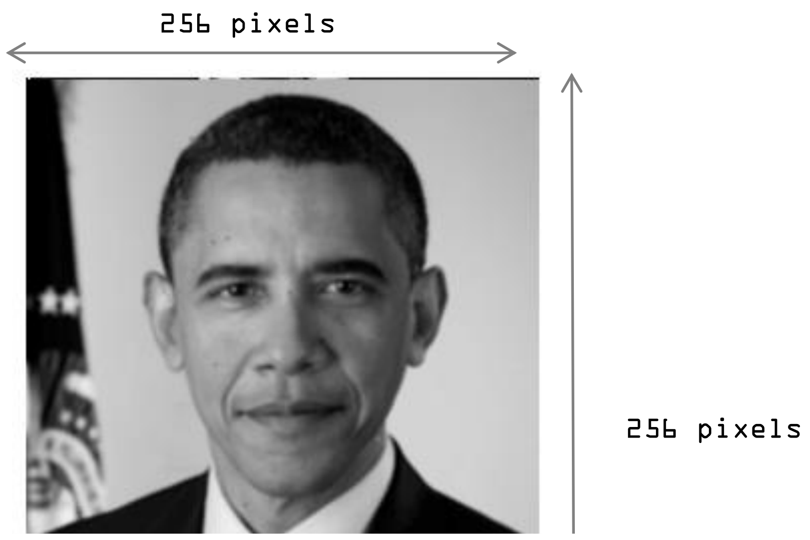


Figure 6.3: Filtered Image Using Matlab

- **Down Sampled Image (Using Matlab)**

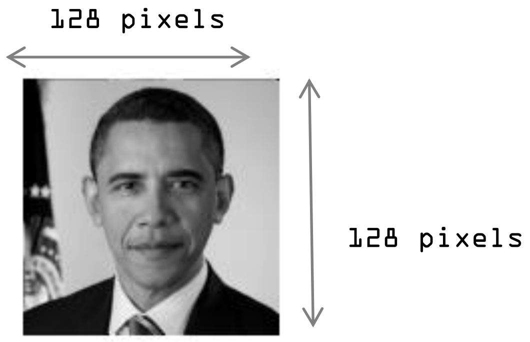


Figure 6.4: Down Sampled Image Using Matlab

6.2. RESULTS VERIFICATION AND ANALYSIS

Using the implemented microprocessor and CPU the given image is processed and output is given as a data array or an image. To compare the accuracy the image s can be compared by using human eye but it is not very accurate for determining the processing accuracy. To get more accurate error the data array of the processed data is compared with the reference image data array.

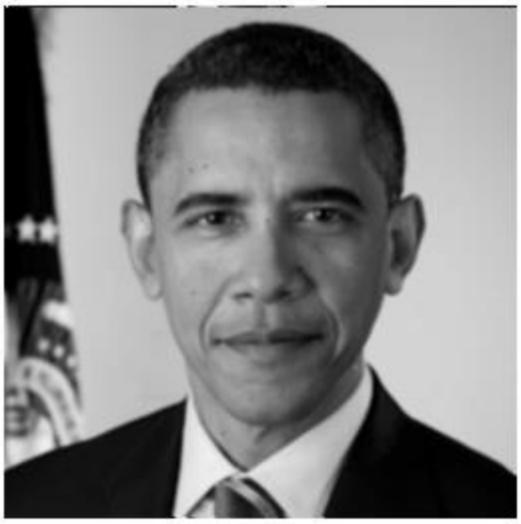


Figure 6.5: FPGA Reference Gaussian Filtered image

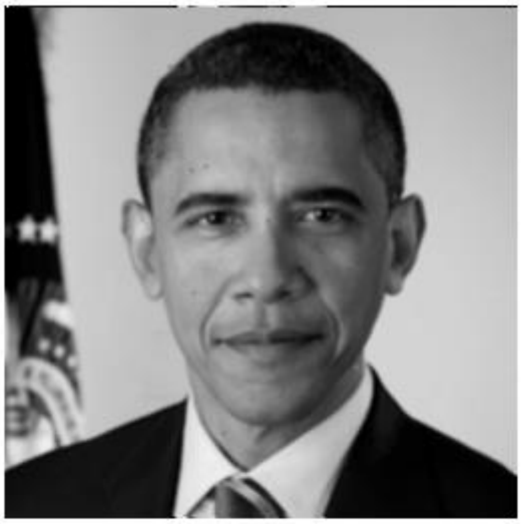


Figure 6.6: FPGA ProcessedGaussian Filtered image

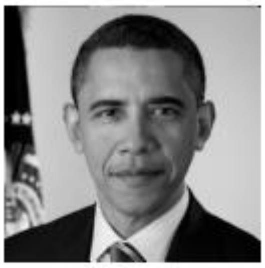


Figure 6.7: ReferenceDown-Sampled image

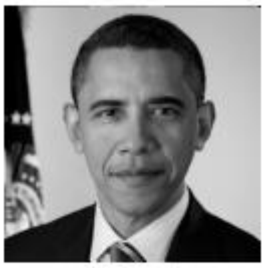


Figure 6.8: FPGA ProcessedDown-Sampled image

Using these data arrays of reference image and the processed image the error array is generated for the given image. (*Refer Appendix E for the Matlab code)*

6.3. ERROR ANALYSIS

Using the reference image and the processed image data array the error array can be generated and using it the accuracy of the result can be calculated.

After generating the error array a very good result was observed because a large amount of zero errors for each pixel were obtained and in the middle of the image some ‘1’s except for zero. It was a very good result and from that error array we can say that our processing method and implementation is working accurately as we expected. That error value ‘1’occurs due to round off done by Matlab. We have done our Matlab simulation by converting the entire image into data type “double” and after converting it to unsigned integer. In this conversion (double -> uint8) Matlab software rounds-off the values considering the decimal places. But when we consider the hardware implementation it does not round-off when we have decimal places. It just takes the floor value. (Because we implement division using right shift method)

Eg: In Matlab:

X = 43/4 = 10.75 (double value)

When converted in to unsigned integer it stores X as ‘11’.

But in the hardware level:

X = 43/4 ➔ 101011 >> 2; answer is 1010 which is '10' in decimal.

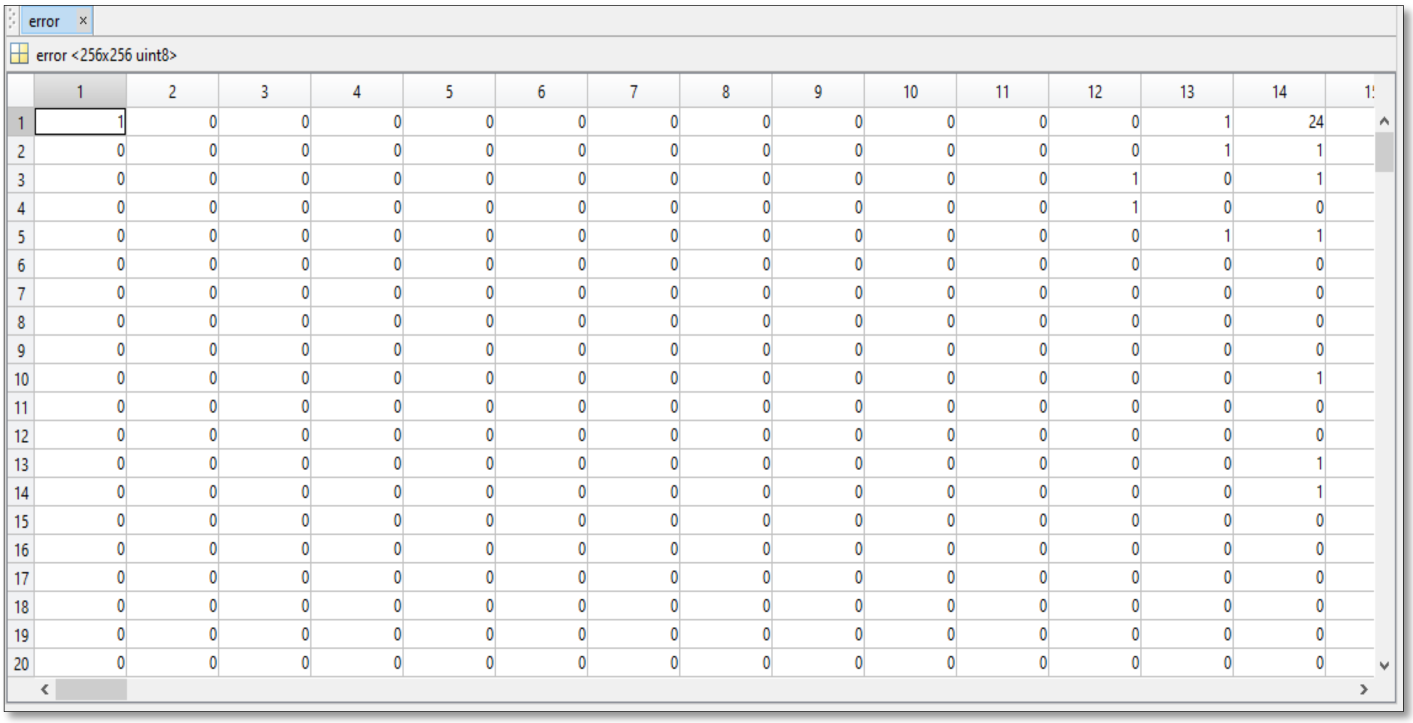
Therefore this is the reason for errors occur in the middle as '1's.

** What we should have done is to add **half of the dividing factor** before we divide some value.

If X= (43 + 2)/4 → 101101 >> 2; answer is 1011 and it is '11' for radix 10.

This is how round-off is performed in hardware. And it is called 'Pre-Filtering'. Unfortunately we were not aware of that matter when we implemented the processor.

However, there are some high valued errors in the edge of the image. This is due to the image array representations of MatLab and FPGA does not match each other's. For Further analysis, error array is given below.


error ×

error <256x256 uint8>

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	0	0	0	0	0	0	0	0	0	0	0	1	24
2	0	0	0	0	0	0	0	0	0	0	0	0	1	1
3	0	0	0	0	0	0	0	0	0	0	0	1	0	1
4	0	0	0	0	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	1	1
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	1
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 6.9: Error Array

Finally we can see that the overall result of the processed data has negligible errors when we compare with the reference image. Therefore we can say that the designed ISA, Algorithms, and FPGA implementation of the Processor works accurately and it fulfills all the given requirements as expected.

7. DESIGN SUMMARY

Top_ModuleProjectStatus(08/09/2016-19:55:07)			
ProjectFile:	FINAL.xise	ParserErrors:	NoErrors
ModuleName:	Top_Module	ImplementationState:	ProgrammingFileGenerated
TargetDevice:	xc6slx45-3csg324	Errors:	NoErrors
Product Version:	ISE14.7	Warnings:	119Warnings(4new)
DesignGoal:	Balanced	RoutingResults:	AllSignalsCompletelyRouted
DesignStrategy:	XilinxDefault(unlocked)	TimingConstraints:	AllConstraintsMet
Environment:	System Settings	FinalTimingScore:	0 (TimingReport)

DeviceUtilizationSummar				[-]
SliceLogicUtilization	Used	Available	Utilization	Note(s)
Number ofSliceRegisters	265	54,57	1%	
Number used asFlipFlops	192			
Number used asLatches	73			
Number used asLatch-thrus	0			
Number used asAND/ORlogics	0			
Number ofSliceLUTs	341	27,28	1%	
Number used aslogic	335	27,28	1%	
Number usingO6outputonly	203			
Number usingO5outputonly	16			
Number usingO5andO6	116			
Number used asROM	0			
Number used asMemory	0	6,40	0%	
Number used exclusivelyasroute-thrus	6			
Number withsame-sliceregisterload	4			
Number withsame-slicecarryload	2			
Number withother load	0			
Number ofoccupiedSlices	119	6,82	1%	
Number ofMUXCYsused	148	13,64	1%	
Number ofLUTFlipFloppairsused	382			
Number withanunusedFlipFlop	128	382	33%	
Number withanunusedLUT	41	382	10%	
Number offullyused LUT-FFpairs	213	382	55%	
Number ofuniquecontrolsets	24			
Number ofsliceregistersiteslost tocontrolset restrictions	55	54,57 6	1%	
Number ofbondedIOBs	4	218	1%	
Number ofLOCedIOBs	4	4	100	
Number ofRAMB16BWERs	32	116	27%	
Number ofRAMB8BWERs	1	232	1%	
Number ofBUFIO2/BUFIO2_2CLKs	0	32	0%	
Number ofBUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number ofBUFG/BUFGMUXs	4	16	25%	
Number used asBUFGs	4			
Number used asBUFGMUX	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	5.08			

PerformanceSumma			[-]
FinalTimingScore:	0(Setup:0,Hold:0)	PinoutData:	PinoutReport
RoutingResults:	AllSignalsCompletelyRouted	ClockData:	ClockReport
TimingConstraints:	AllConstraintsMet		

DetailedReport					[-]
ReportName	Status	Generated	Errors	Warnings	Infos
SynthesisReport	Current	MonAug800:46:512016	0	111Warnings(0new)	4Infos(2new)
TranslationReport	Current	TueAug919:48:192016	0	0	0
MapReport	Current	TueAug919:48:492016	0	4Warnings(2new)	6Infos(0new)
Placeand RouteReport	Current	TueAug919:49:072016	0	0	3Infos(0new)
PowerReport					
Post-PARStaticTimingReport	Current	TueAug919:49:142016	0	0	4Infos(0new)
BitgenReport	Current	TueAug919:49:322016	0	4Warnings(2new)	1Info(0new)

SecondaryReport		[-]
ReportName	Status	Generated
Post-SynthesisSimulationModelReport	Current	TueAug919:55:062016
WebTalkReport	Current	TueAug919:49:332016
WebTalkLogFile	Current	TueAug919:49:432016

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**

9. APPENDIXES

Appendix A

Matlab Code for Read Image and Generate Data Array

```
%code for read and generate data array of the original image..
%ENTC-UOM

data_arry=imread('Pr.png'); %read the image Pr.png and save 2D array in as
data_arry

flat_d_arry=data_arry(:); %make flat data array and save it as flat_d_arry

flat_d_arry=double(flat_d_arry);%to process data make the data type as
double

imshow(data_arry);          %show the read image as a picture....
figure
```

Appendix B

Matlab Code for Gaussian Filtering

```
%code for Gaussian filtering....
%ENTC-UOM

total=0;
total=double(total); %make zero variable
for j=1:1:254   %process Gaussian filter...
for i=1:1:254
       x=j*256+i;
       x=x+1;
       k=(j-1)*256+i-1;
       k=k+1;
total=total+double(flat_d_arry(x)*4);
total=total+double(flat_d_arry(x-1)*2);
total=total+double(flat_d_arry(x+1)*2);
total=total+double(flat_d_arry(x-256)*2);
total=total+double(flat_d_arry(x-257));
total=total+double(flat_d_arry(x-255));
total=total+double(flat_d_arry(x+256)*2);
total=total+double(flat_d_arry(x+255));
total=total+double(flat_d_arry(x+257));
total=total/16;
flat_d_arry(k)=total;
total=0;

end
end

filtered_image=uint8(flat_d_arry);  %save as integers...
c=reshape(filtered_image,256,256);  %make 2d array from vector..
imshow(c);  %display the filtered image....
```

Appendix C

MatLab Code for Down Sampling the Image

```
%code for down sampling....
%ENTC-UOM

c=double(c); %get the filtered image for processing
sampling_image=c(:);
sampling_image=double(sampling_image);
test_image=sampling_image;
value=0;
y=0;
k=1;
for j=0:1:127   %down sampling the filtered image..
for i=0:1:127
            y=2*i+512*j;
%display(y);
            y=y+1;
value=sampling_image(y);
test_image(k)=value;
            y=0;
            k=k+1;
end
end
```

Appendix D

MatLab Code for Display output and Generate Data Array

```
%code for display output and generate data array…
%ENTC-UOM

final_image=uint8(test_image);

final_image=final_image(1:16384);

final_image_pic=reshape(final_image,128,128);%generate data array

imshow(final_image_pic);          %display the down sampled image..
```

Appendix E

MatLab Code for Error Detection

```
%code for error detection..
%ENTC-UOM

error_array=zeros(128,128);     %generate zero value 2d array..

error_array = ( reference_image - final_image_pic );

%get error value for each pixel....
```

Appendix F

Verilog Code for Register without Increment

```
module register16(clk,load,data_in,data_out);
        inputclk,load;
        input  [15:0] data_in;
        output [15:0] data_out;
        reg    [15:0] data_out=16'd99;
        always@(posedgeclk) if(load) data_out<=data_in;
endmodule
```

Appendix G

Verilog Code for Register with Increment

```
module register16_increment(clk,load,inc,data_in,data_out);
  inputclk,load,inc;
  input  [15:0] data_in;
  output [15:0] data_out;
  reg    [15:0] data_out=0;
  always@(posedgeclk)
        begin
              if(load) data_out<=data_in;
              if(inc) data_out<=data_out+16'd1;
        end
endmodule
```

Appendix H

Verilog Code for the De-multiplexer

```
module demux(bflag,M,PC,R1,R2,R3,R,AC,IM,B_bus);
  input [2:0] bflag;
  input [15:0] PC,R1,R2,R3,R,AC;
  input [7:0] M,IM;
  outputreg [15:0] B_bus;
  always @(bflag or M or PC or R1 or R2 or R3 or R or AC or IM)
        begin
        case(bflag)
              3'd0:B_bus={8'b00000000,M};
              3'd1:B_bus=PC;
              3'd2:B_bus=R1;
```

```
            3'd3:B_bus=R2;
            3'd4:B_bus=R3;
            3'd5:B_bus=R;
            3'd6:B_bus=AC;
            3'd7:B_bus={8'b00000000,IM};

        endcase
  end
endmodule
```

Appendix I

Verilog Code for the ALU

```
modulealu(A_bus,B_bus,op,C_bus,Z);
  input [15:0] A_bus;
  input [15:0] B_bus;
  input [2:0]  op;
  output[15:0] C_bus;
  outputreg   Z=0;
  reg   [15:0] C_bus;

  parameter        ADD=3'd0,        SUB=3'd1,        PASS=3'd2,ZER=3'd3,
DECAC=3'd4,BSHIFTL=3'd5, DIV2=3'd6;

  always@(op or A_bus or B_bus)
  begin
        case (op)
            ADD: begin C_bus=A_bus+B_bus;
```

```
                Z=0;end
        SUB: begin C_bus =A_bus-B_bus;
                Z=(A_bus==16'd0)?1'b1:1'b0;end
        PASS: begin C_bus=B_bus;
                if(A_bus==16'd0) Z=1;
                end
        ZER: begin C_bus= 0;
                Z=0;end
        DECAC:beginC_bus=A_bus-16'd1;
                Z=0;end
        BSHIFTL:beginC_bus= A_bus<< 2;
                Z=0;end
        DIV2:beginC_bus = A_bus>> 1;
                Z=0;end

    endcase
 end
endmodule
```

Appendix J

Verilog Code for the Clock Divider

```
moduleclock_divider(clock,enable,finish,clk);
 outputregclk=1;
 inputclock,finish,enable;
 reg     [31:0] count=0;

 always @(posedge clock)
        begin
```

```
                        if(~finish & enable)
                             begin
                             if (count == 32'd5)
                                  begin
                                  clk = ~clk;
                                  count=0;
                                  end
                             else count = count + 32'd1 ;
                             end
                        elseclk=0;
                   end
endmodule
```

Appendix K

Verilog Code for the State Machine

```
module state_machine(clk,z,ir,pcinc,r1inc,r2inc,r3inc,acinc,bflag,alu,fetch,cflag,finish);
  inputclk,z;
  input [7:0] ir;
  outputreg pcinc,r1inc,r2inc,r3inc,acinc,fetch,finish=0;
  outputreg [2:0] alu;
  outputreg [2:0] bflag;
  outputreg [7:0] cflag;

  reg [5:0] PS,NS=FETCH1;

  parameter
  FETCH1          =6'd0,
  FETCH2          =6'd1,
  FETCH3          =6'd2,
  FETCH4          =6'd57,
  CLAC1                   =6'd3,
  MVACAR1         =6'd59,
  STAC1                   =6'd4,
  WRITE           =6'd53,
  MVACRI1         =6'd6,
  MVACRII1        =6'd7,
  MVACRIII1=6'd8,
  MVACR1          =6'd9,
  MVRIAC1         =6'd10,
  MVRIIAC1        =6'd11,
  MVRIIIAC1=6'd12,
  MVRAC1          =6'd13,
  INAC1             =6'd14,
  INCR1                   =6'd40,
  INCR2                   =6'd41,
  INCR3                   =6'd42,
  LDAC1             =6'd15,
  LDAC2             =6'd16,
  LDAC3             =6'd56,
  SUB1              =6'd17,
  DECAC1          =6'd18,
  ADD1            =6'd19,
  DIV21           =6'd20,
  BYTESHIFT1  =6'd21,
  JUMPNZ1     =6'd22,
  JUMPNZY1    =6'd23,
  JUMPNZN1    =6'd24,
  JUMPNZN2    =6'd25,
  JUMPNZN3    =6'd26,
  JUMPNZNSKIP =6'd55,
  ADDM1       =6'd27,
  ADDM2       =6'd28,
  ADDMPC              =6'd54,
  NOP                 =6'd58,
  END                 =6'd60;

  always@(negedgeclk) PS<=NS;
  always@(PS or z or ir)
          case(PS)
                  FETCH1:begin
                          pcinc<=0;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=0;
                          finish<=0;
                          bflag<=3'd7;
                          alu<=3'd2;
                          cflag<=8'b00000000;
                          NS<=FETCH2;
                  end
                  FETCH2:begin
                          pcinc<=0;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=1;
                          finish<=0;
                          bflag<=3'd7;
                          alu<=3'd2;
                          cflag<=8'b00000000;
                          NS<=FETCH3;
                  end
                  FETCH3:begin
                          pcinc<=1;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=0;
                          finish<=0;
                          bflag<=3'd0;
                          alu<=3'd2;
                          cflag<=8'b00000000;
                          NS<=FETCH4;
                  end
                  FETCH4:begin
                          pcinc<=0;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=0;
                          finish<=0;
                          bflag<=3'd0;
                          alu<=3'd2;
                          cflag<=8'b00000000;
                          NS<=ir[5:0];
                  end
                  CLAC1:begin
                          pcinc<=0;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=0;
                          finish<=0;
                          bflag<=3'd0;
                          alu<=3'd3;
                          cflag<=8'b00000010;
                          NS<=FETCH1;
                  end
                  MVACAR1:begin
                          pcinc<=0;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=0;
                          finish<=0;
                          bflag<=3'd6;
                          alu<=3'd0;
                          cflag<=8'b10000000;
                          NS<=FETCH1;
                  end
                  STAC1:begin
                          pcinc<=0;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=0;
                          finish<=0;
                          bflag<=3'd6;
                          alu<=3'd0;
                          cflag<=8'b00000001;
                          NS<=WRITE;
                  end
                  WRITE:begin
                          pcinc<=0;
                          r1inc<=0;
                          r2inc<=0;
                          r3inc<=0;
                          acinc<=0;
                          fetch<=0;
                          finish<=0;
                          bflag<=3'd6;
                          alu<=3'd0;
                          cflag<=8'b00000001;
                          NS<=FETCH1;
                  end
```

```
MVACRI1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd6;
        alu<=3'd0;
        cflag<=8'b00100000;
        NS<=FETCH1;
end
MVACRII1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd6;
        alu<=3'd0;
        cflag<=8'b00010000;
        NS<=FETCH1;
end
MVACRIII1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd6;
        alu<=3'd0;
        cflag<=8'b00001000;
        NS<=FETCH1;
end
MVACR1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd6;
        alu<=3'd0;
        cflag<=8'b00000100;
        NS<=FETCH1;
end
MVRIAC1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd2;
        alu<=3'd2;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
MVRIIAC1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd3;
        alu<=3'd2;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
MVRIIIAC1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd4;
        alu<=3'd2;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
MVRAC1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd5;
        alu<=3'd2;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
INAC1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=1;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=FETCH1;
```

```
end
INCR1:begin
        pcinc<=0;
        r1inc<=1;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=FETCH1;
end
INCR2:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=1;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=FETCH1;
end
INCR3:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=1;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=FETCH1;
end
LDAC1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd6;
        alu<=3'd0;
        cflag<=8'b10000000;
        NS<=LDAC2;
end
LDAC2:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=LDAC3;
end
LDAC3:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd2;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
SUB1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;

        fetch<=0;
        finish<=0;
        bflag<=3'd5;
        alu<=3'd1;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
DECAC1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd4;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
ADD1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd5;
        alu<=3'd0;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
DIV21:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd6;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
BYTESHIFT1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd5;
        cflag<=8'b00000010;
        NS<=FETCH1;
end
```

```
JUMPNZ1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd2;
        cflag<=8'b00000000;
        if(z)NS<=JUMPNZY1;
        else NS<=JUMPNZN1;
end
JUMPNZY1:begin
        pcinc<=1;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd2;
        cflag<=8'b00000000;
        NS<=FETCH1;
end
JUMPNZN1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd7;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=JUMPNZN2;
end
JUMPNZN2:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd7;
        alu<=3'd2;
        cflag<=8'b00000010;
        NS<=JUMPNZN3;
end
JUMPNZN3:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd6;
        alu<=3'd0;
        cflag<=8'b01000000;
        NS<=FETCH1;
end
ADDM1:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd1;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=ADDM2;
end
ADDMPC:begin
        pcinc<=1;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd3;
        alu<=3'd0;
        cflag<=8'b00000000;
        NS<=FETCH1;
end
ADDM2:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd7;
        alu<=3'd0;
        cflag<=8'b00000010;
        NS<=ADDMPC;
end
NOP:begin
        pcinc<=0;
        r1inc<=0;
        r2inc<=0;
        r3inc<=0;
        acinc<=0;
        fetch<=0;
        finish<=0;
        bflag<=3'd0;
        alu<=3'd0;
        cflag<=8'b00000000;
```

```
                NS<=FETCH1;
            end
            END:begin
                pcinc<=0;
                r1inc<=0;
                r2inc<=0;
                r3inc<=0;
                acinc<=0;
                fetch<=0;
                finish<=1;
                bflag<=3'd0;
                alu<=3'd0;
                cflag<=8'b00000000;
                NS<=END;
            end

        endcase
endmodule
```

Appendix L

Verilog Code for the Processor

```
module
Processor(clock,enable,finish,ram_dout,iram_out,cout,p_wr,p_addr,p_din,im_addr);
  inputclock,enable;
  input [7:0] ram_dout,iram_out;
  outputfinish,cout,p_wr;
  output [15:0] p_addr;
  output [7:0] p_din,im_addr;
```

```
    wire clk,z,pcinc,r1inc,r2inc,r3inc,acinc,fetch;
    wire [2:0] bflag,aluop;
    wire [7:0] IR,cflag;
    wire [15:0] bus,PC,R1,R2,R3,R,AC,C_bus;

    assigncout=clk;
    assignp_wr=cflag[0];
    assignp_din=bus[7:0];
    assignim_addr=PC[7:0];

    ///module instances
    clock_divider cd(clock,enable,finish,clk);
    state_machine sm(clk,z,IR,pcinc,r1inc,r2inc,r3inc,acinc,bflag,aluop,fetch,cflag,finish);
    demux mux(bflag,ram_dout,PC,R1,R2,R3,R,AC,iram_out,bus);

    register16ar(clk,cflag[7],bus,p_addr);
    register16_increment pc(clk,cflag[6],pcinc,bus,PC);
    register8ir(clk,fetch,bus[7:0],IR);
    register16_increment r1(clk,cflag[5],r1inc,bus,R1);
    register16_increment r2(clk,cflag[4],r2inc,bus,R2);
    register16_increment r3(clk,cflag[3],r3inc,bus,R3);
    register16 r(clk,cflag[2],bus,R);
    alu alu16(AC,bus,aluop,C_bus,z);
    register16_increment ac(clk,cflag[1],acinc,C_bus,AC);

endmodule
```

Appendix M

Verilog Code for the Board Rate Generator

```
moduleBoard_Rate_Generator
 #(
  parameter     N =2,
                                M =4
                                        )
  (
        input wire clk, reset,
        output wire max_tick,
        output wire [N-1:0] q
        );

  reg [N-1:0] r_reg;
  wire [N-1:0] r_next;

  always@(posedgeclk, posedge reset)
        if(reset)
                r_reg<=0;
        else
                r_reg<= r_next;

  assignr_next = (r_reg==(M-1)) ? 1'b0 :r_reg +1'b1;
  assign q = r_reg;
  assignmax_tick = (r_reg==(M-1)) ? 1'b1 : 1'b0;

endmodule
```

Appendix N

Verilog Code for the UART_RX

```
module UART_RX
  #(parameter DBIT = 8,
                                        SB_TICK = 16
  )
  (
          input wire clk, reset,
          input wire rx, s_tick,
          outputregrx_done_tick,
          output wire [7:0] dout
   );

  localparam [1:0]
          idle = 2'b00,
          start = 2'b01,
          data = 2'b10,
          stop = 2'b11;

  reg [1:0] state_reg, state_next;
  reg [3:0] s_reg, s_next;
  reg [2:0] n_reg, n_next;
  reg [7:0] b_reg, b_next;

  always @(posedgeclk, posedge reset)
          if(reset)
                  begin
                          state_reg<= idle;
```

```
                s_reg<= 0;
                n_reg<= 0;
                b_reg<= 0;
            end
        else
            begin
                state_reg<= state_next;
                s_reg<= s_next;
                n_reg<= n_next;
                b_reg<= b_next;
            end

always @*
begin
        state_next = state_reg;
        rx_done_tick = 1'b0;
        s_next = s_reg;
        n_next = n_reg;
        b_next = b_reg;
        case (state_reg)
            idle:
                if (~rx)
                begin
                    state_next = start;
                    s_next = 0;
                end
            start:
                if (s_tick)
                    if(s_reg == 7)
```

```
                        begin
                                state_next = data;
                                s_next = 0;
                                n_next = 0;
                        end
                else
                        s_next = s_reg + 1'b1;
data:
        if (s_tick)
                if (s_reg == 15)
                        begin
                                s_next = 0;
                                b_next = {rx, b_reg[7:1]};
                                if (n_reg == (DBIT-1))
                                        state_next = stop;
                                else
                                        n_next = n_reg + 1'b1;
                                end
                else
                        s_next = s_reg + 1'b1;
stop:
        if (s_tick)
                if (s_reg == (SB_TICK-1))
                        begin
                                state_next = idle;
                                rx_done_tick = 1'b1;
                        end
                else
                        s_next = s_reg + 1'b1;
        endcase
  end
  assigndout = b_reg;

endmodule
```

Appendix O

Verilog Code for the UART_TX

```
module UART_TX
  #(
        parameter DBIT = 8,
                                SB_TICK = 16
  )
  (
        input   wire clk,reset,
        input   wire tx_start, s_tick,
        input   wire [7:0] din,
        outputregtx_done_tick,
        output wire tx
  );

  localparam [1:0]
  idle   =      2'b00,
  start  =      2'b01,
  data   =      2'b10,
  stop   =      2'b11;

  reg [1:0] state_reg, state_next;
reg [3:0] s_reg, s_next;

reg [2:0] n_reg, n_next;

reg [7:0] b_reg, b_next;

regtx_reg, tx_next;



always@(posedgeclk, posedge reset)
        if (reset)
                begin
                        state_reg<= idle;
                        s_reg<= 4'd0;
                        n_reg<= 3'd0;
                        b_reg<= 8'd0;
                        tx_reg<= 1'b1;
                end
        else
                begin
                        state_reg<= state_next;
                        s_reg<= s_next;
                        n_reg<= n_next;
                        b_reg<= b_next;
                        tx_reg<= tx_next;

                end

always@*
begin
state_next = state_reg;
tx_done_tick = 1'b0;
```

```
s_next = s_reg;
n_next = n_reg;
b_next = b_reg;
tx_next = tx_reg;

case(state_reg)
    idle:
        begin
            tx_next = 1'b1;
            if (tx_start)
                begin
                    state_next = start;
                    s_next = 0;
                    b_next = din;
                end
            end
        start:
            begin
                tx_next = 1'b0;
                if (s_tick)
                    if(s_reg==15)
                        begin
                            state_next = data;
                            s_next=0;
                            n_next=0;
                        end
                    else
                        s_next=s_reg+4'd1;
            end
```

```
data:
        begin
                tx_next = b_reg[0];
                if(s_tick)
                        if(s_reg==15)
                                begin
                                        s_next = 0;
                                        b_next = b_reg>> 1;
                                        if (n_reg==(DBIT-1))
                                                state_next = stop;
                                        else
                                                n_next = n_reg +3'd1;
                                end
                        else
                                s_next = s_reg + 4'd1;
        end
stop:
        begin
                tx_next = 1'b1;
                if (s_tick)
                        if (s_reg==(SB_TICK-1))
                                begin
                                        state_next = idle;
                                        tx_done_tick = 1'b1;
                                end
                        else
                                s_next = s_reg + 4'd1;
        end
```

```
                endcase
        end


        assigntx=tx_reg;


  endmodule
```

Appendix P

Verilog Code for the UART_ADDR

```
module UART_ADDR(clk,rx_done_tick,tx_done_tick,data_out,din,address,wea,p_enable);



  inputclk,rx_done_tick,tx_done_tick;
  input [7:0] data_out;
  output [7:0]din;
  output [15:0] address;
  outputwea,p_enable;

  reg [15:0] address=16'd0;
  regwea=1'b1;
  regp_enable=0;
  reg [7:0] din;

  always@(posedgeclk )
        begin

        if (address==16'd65535 &&rx_done_tick) begin
```

```
            wea<=1'b0;

            address<=16'd0;

            p_enable<=1'b1;

            din<=data_out;

        end


        else if (rx_done_tick&&wea) begin

            address<=address+16'd1;


        end


        else if(tx_done_tick&& address<16'd65535) begin

            address<=address+16'd1;

            din<=data_out;


        end


        end

endmodule
```

Appendix Q

Verilog Code for the module Selection

```
module
Selection(p_enable,p_finish,clock,p_clk,u_wr,p_wr,u_addr,p_addr,u_din,p_din,tx_start,r_c
lk,r_wr,r_addr,r_din);

  inputclock,p_clk,u_wr,p_wr,p_enable,p_finish;

  input [15:0] u_addr,p_addr;

  input [7:0] u_din,p_din;
```

```
outputregtx_start=0;

outputregr_clk,r_wr;

outputreg [15:0] r_addr;

outputreg [7:0] r_din;



reg [1:0] PS=2'b11;


always@(*) begin

        if(~p_enable&& ~p_finish) PS<=2'b00;//receiving data

        if(p_enable&& ~p_finish) PS<=2'b01;//processing data

        if(p_enable&&p_finish) PS<=2'b10;//transmitting data

end

always@(*)

        case(PS)

                2'b00:begin

                                tx_start<=0;

                                r_clk<=clock;

                                r_wr<=u_wr;

                                r_addr<=u_addr;

                                r_din<=u_din;


                        end

                2'b01:begin

                                tx_start<=0;

                                r_clk<=p_clk;

                                r_wr<=p_wr;

                                r_addr<=p_addr;

                                r_din<=p_din;
```

```
                    end
                2'b10:begin
                        tx_start<=1;
                        r_clk<=clock;
                        r_wr<=u_wr;
                        r_addr<=u_addr;
                        r_din<=u_din;

                    end

        endcase

endmodule
```

Appendix R

Verilog Code for the Top Module

```
moduleTop_Module(
        inputclk,reset,rx,
        outputtx
  );

        wire [7:0] data_out;
        wires_tick;
        wire tx_done_tick,p_finish,p_clk,p_wr,tx_start,r_clk,r_wr,wea,p_enable;

        wire [7:0] dout,p_din,r_din,im_addr,iram_out,din;
        wire [15:0] p_addr,r_addr,address
```

```
RAM ram64K(
    .clka(r_clk),
    .wea(r_wr),
    .addra(r_addr),
    .dina(r_din),
    .douta(data_out)
  );

  Board_Rate_Generatorbdg( .clk(clk), .reset(reset), .max_tick(s_tick));

  UART_RX   rcx(.clk(clk),   .reset(reset),   .rx(rx),   .s_tick(s_tick),   .dout(dout),
.rx_done_tick(rx_done_tick));

  UART_TX uut (
                .clk(clk),
                .reset(reset),
                .tx_start(tx_start),
                .s_tick(s_tick),
                .din(din),
                .tx_done_tick(tx_done_tick),
                .tx(tx)
         );

         IRAM iram (
  .clka(p_clk),
  .addra(im_addr),
  .douta(iram_out)
);
```

```
  UART_ADDR  uaddr(clk,rx_done_tick,tx_done_tick,data_out,din,address,wea,p_enable);


  Selection
sel(p_enable,p_finish,clk,p_clk,wea,p_wr,address,p_addr,dout,p_din,tx_start,r_clk,r_wr,r_
addr,r_din);

  Processor
procs(clk,p_enable,p_finish,data_out,iram_out,p_clk,p_wr,p_addr,p_din,im_addr);



endmodule
```

……………………………………..END………………………………………