

PROCESSOR IMPLEMENTATION

130221G- G.O. Ishendra
130364V-T. N. Malawaraarachchi
130656T- K. S. Wijayasekara
130678L- D. R. H. Witharane

EN3030 - Circuits and Systems Design

Contents

1 OBJECTIVE	3
1.1 What is a CPU?	3
1.2 Then what is a Microprocessor?	4
1.3 What should be achieved at the end?	5
1.4 How to achieve the given target?	6
2 Designing an object specified processor architecture	7
2.1 ISA (Instruction Set Architecture)	7
2.1.1 Components.....	7
2.1.2 Data Path.....	11
2.1.3 Instruction Set.....	12
2.1.4 Fetch, Decode and Execution Cycle	16
3 Modeling the above designed architecture using Verilog.	21
3.1 Verilog implementation	21
3.2 Memory implementation.....	21
3.2.1 Data Memory	22
3.2.2 Instruction Memory	23
4 Implementing and testing the modules on FPGA board.....	24
4.1 Implemented architecture on FPGA board.....	24
4.1.2 B_bus.....	26
4.1.3 ALU	27
4.1.4 Control store	28
4.1.5 Processor.....	30
4.1.6 RamMem.....	31
4.1.7 RomIns	31
4.1.8 realCPU.....	31
4.1.9 Clock divider.....	32
4.2 Testing.....	32
4.2.1 Testing and Simulations	32
4.2.2 Assembly code	34
5 UART communication	36
5.1 Baud rate generator.....	36
5.2 Receiving a data stream through UART	36

5.3 Sending a data stream which is in the ram using UART.....	37
6 Algorithm	38
6.1 Filtering the image	38
6.2 Down sampling the image	39
8 Result Analyzing and Verification	40
8.1 Generate Reference Output Image.....	40
8.2 Results Verification and Analysis	42
8.3 Error Analysis	43
8.4 Test Results	49
Appendix A	57
APPENDIX B	65
APPENDIX C	74
APPENDIX D.....	78
APPENDIX E	79
References	86

1 OBJECTIVE

The task which was allocated was to design a processor architecture to make a processor design. The processor should be capable of performing a specific task. In this project the allocated task was to filter and down sample an image. For that first of all we had to design an architecture. And then model it using Verilog and implement and test it on an FPGA development board(in this case Spartan 6). So the followings contain a detailed report of the project.

1.1 What is a CPU?

Many people consider the CPU (Central Processing Unit) to be the brains of the computer. This analogy is very loose because, for the most part, the CPU cannot keep data stored inside it like a brain. In contrast, it is used to process much of the information needed by the computer, just like our brain thinks and processes information and gives orders to our other body parts.

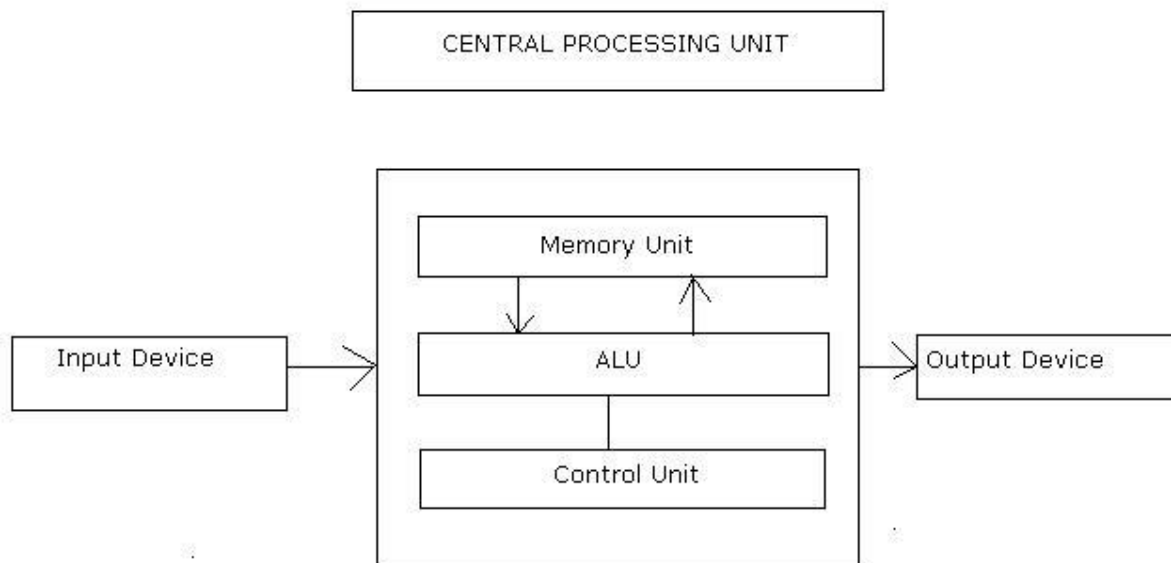
Over the past few years, we have seen the CPU MHz speeds go from 100 MHz to over 2 GHz (1000 MHz = 1 GHz). This is one reason that people need to learn about a CPU. Many people would expect a 1.8 GHz Intel Pentium 4 to be much faster than a 1.4 GHz AMD Athlon because its speed is 0.4 GHz faster. In truth, not only is the Intel Pentium 4 up to three times more expensive than the AMD Athlon, it is either much slower or neck to neck in most "Real world tests", which compares the amount of times that it takes each CPU to perform a certain task. With this information, you know that you should not judge a computer by the "speed ratings". But if one CPU goes at a faster MHz rate and is slower, what does determine the speed of the CPU? There are a variety of factors, but we will show you the main parts of a CPU, and what they are used for.

When looking at a CPU, there are a few basic things that we should know about it: • A CPU has four basic tasks that it performs. They are Fetch, Decode, Manipulate and Output.

- Speed rating, although not accurate, is almost always measured by MHz.
- The CPU speed is determined by a combination of raw MHz as well as design and other features such as the FPU of the chip.

Before we get into details about how it works, we must remember that, like many other computer parts, the CPU is comprised of millions of logic gates embedded into it which then are used to complete a variety of different operations. The size of the CPU core, the part with the logic gates, can be as small as the size of a smaller coin. The gates are used with a clock that regulates the speed at which the CPU is fed data. The speed at which it does this is measured in Hz (amount of clock pulses in one second), MHz (about 1 million Hz) and GHz (about 1000 MHz). If there was no clock to regulate the data flow, the CPU would be unorganized and useless. The clock does a similar thing for the CPU as traffic lights

do for the traffic. It makes everything organized and tells when the data should pass through, and when it should not.



1.2 Then what is a Microprocessor?

A microprocessor is a tiny electronic chip found inside a computer's central processing unit and other electronic devices. Its basic function is to take input, process it and then provide appropriate output.

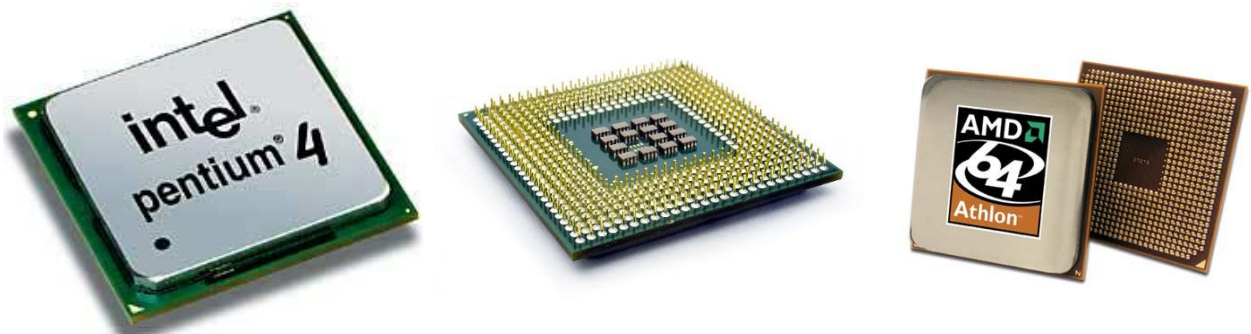
On the surface, a microprocessor's job may seem like an easy task, but modern microprocessors perform trillions of instructions per second.

A computer's central processing unit handles all the processing functions of a computer including processing instructions from peripheral devices as well as input from running programs.

From the time a computer is turned on to the time it is shut down, a microprocessor will have performed millions of logic and arithmetic operations. These operations utilize tiny number holding spaces called registers. Typical arithmetic operations include adding, subtracting and comparing two or more values.

To perform the operations, a microprocessor has to receive specific instructions as part of its design. For instance, when a computer is started, the microprocessor receives its first set of instructions from the basic input-output system.

A microprocessor's speed is measured in megahertz. It is common to associate a higher megahertz with better performance but this is not always true. A computer's overall performance is influenced by several factors such as the amount of available memory, the bus architecture, the applications running on the computer and the efficiency of the processor.



1.3 What should be achieved at the end?

1. **Generate a byte array of a black and white image and sending it through UART using Matlab.**

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 the appendix A for Matlab code) After generating the array we should find a way to send that array to our FPGA board. For that we used UART which is one of the most common communication protocols nowadays. After generating an one dimensional pixel array sending it through UART was also implemented in the same code (appendix A).

2. **Receive an array of data from UART and store it on the RAM on FPGA board.**

Once we implement the sending part in Matlab, we should implement the receiving part in the FPGA board using Verilog. After receiving the data, received data should be stored in a given primary memory which has been implemented in the FPGA.

3. **Filter the image and rewrite to the RAM.**

As we have mentioned above our target is to generate a down sampled image using the processor we implemented using our ISA(Instruction set architecture). Before down sampling an image usually we should perform a filtering process to reduce the sharpness between down sampled image's pixels. We should write the filtered image back to the RAM too.

4. Down sample the image and rewrite to the RAM.

After filtering the image saved in the RAM, we should implement a program to our ISA to perform down sampling using its instructions.

5. Send an array of data which is on RAM through UART

Once we are done with the processing part and the down sampled image is successfully saved on RAM, we should send the down sampled image to the Personal computer via UART to show it. For that we should implement a UART transmitting code in Verilog.

6. Regenerate the down sampled image using Matlab.

Finally when we successfully receive the one dimensional pixel array via UART we must regenerate the image and show it on the computer screen.

1.4 How to achieve the given target?

Above we have identified the problem and have identified the different blocks of the problem. Now it is much easier to look into these blocks and solve them individually. Once we implement solutions for all these blocks we should combine them too. First of all let's implement solutions for above blocks.

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.

1. Designing an object specified processor architecture
2. Modeling the above designed architecture using Verilog.
3. Implementing and testing the Verilog code on FPGA board.

4. Implementing the below sections on FPGA using Verilog
 - Receiving a data stream through UART and save it on RAM
 - Filtering the image
 - Down sampling the image
 - Sending a data stream which is in the ram using UART

2 Designing an object specified processor architecture

2.1 ISA (Instruction Set Architecture)

2.1.1 Components

- **Data Memory-**

Since we are planning to filter and down sample an image, we must have a memory allocated to store the image we are going to process and also the filtered and down sampled image also should be saved in to a memory. In this case we have we haven't designed this to process a fixed sized image. So we must have some memory to store image size and other details too. To implement this we have used a data RAM with 65536 memory locations and 8 bit(which can accommodate greyscale image pixel) data size. (ideally to put a 256x256 image, but since we are taking some memory locations to store required external data the maximum size of the image is 255x255)

- **Program Memory**

To perform the processing we must have a sequence of instructions stored in the FPGA board for that we implemented a ROM(Read-only-Memory) since the program is not going to change. After making our algorithm we realized it is better to have 512 memory locations with a data width of 8 bits.

- **Memory Address Register**

Memory Address register is 16 bit register which we use store and provide an address to RAM when we are fetching from RAM or writing to the RAM. Basically MAR takes 3 inputs which are the clock, Ld(write enable signal) and the C Bus. And it outputs 16bit address.

- **Memory Data Register**

Memory Data Register(MDR) is a 8 bit register which we use to store data only before WRITING to the RAM. When we are fetching we can take it directly to the B bus. Basically MDR takes 3 inputs which are the clock, Ld(write enable signal) and the C Bus. And it outputs 8bit data to the RAM.

- **Program Counter**

Program counter is a 9bit register which keeps the address of the next instruction that in the instruction memory. PC takes 4 inputs which are the clock, Ld(write enable signal), Inc(a control signal which increments the address by 1 when it is 1) and the C Bus. And it directly outputs 16bit address to the Instruction memory.

- **Instruction Register**

Instruction register is an 8 bit register which stores the instruction to be executed. takes 3 inputs which are the clock, fetch(fetch signal) and input directly from the instruction memory. And it outputs 8bit instruction to the B bus.

- **General Purpose Registers R,R1, R2**

In our Instruction Set Architecture we have 3 16bit general purpose registers. Even though there are 3 general purpose registers, the widely used one is the R. the basic purpose of having those general purpose registers is to use them as intermediate storing elements. A general purpose register takes 3 inputs which are the clock, Ld(write enable signal) and the C Bus. And it outputs 16bit data to the B bus.

- **Accumulator**

The Accumulator is a 16 bit register which is directly connected to the ALU(via A bus). Whenever data is loaded from data memory, it is loaded to AC and data stored to data memory is stored from AC. AC takes 3 inputs which are the clock, Ld(write enable signal), a clear signal to clear data on itself and the C Bus. And it outputs 16bit data to the ALU via A bus.

- **Arithmetic and Logic Unit**

ALU is like the heart of the processor. All the mathematical and logical functions are carried out by this component. ALU has 3 busses connected to it which are A(Accumulator), B(R,R2,R3,Data Memory) and C(output to all other registers except for IR)

ALU is capable of performing 11 different functions which are,

MUL:	$C=A*B;$
DIV:	$C=A/B;$
AND:	$C=A \& B;$
OR:	$C=A B;$
NOT:	$C=\sim A;$
INC:	$C=A+16'd1;$
DEC:	$C=A-16'd1;$
LSHIFT8:	$C=B<<8;$
AtoC:	$C=A;$
BtoC:	$C=B;$
setZ:	$Z=0;$

Depending on 4 inputs which are clock, A bus, B bus and 3 bit control signal. After performing the operation it will output 16 bit data to C bus and set z flag depending on the operation results.

- **BUS**

A Bus:-

A 16 bit wire which connects the Accumulator and the ALU.

B Bus:-

A multiplexer which connects Instruction register, R, R2, R3 and Data Memory. The input is controlled by a 3 bit control signal given by the control store and outputs 16bit data to the ALU via a wire.

C Bus:-

C bus is a 16 bit wire which connects the output of ALU and all the other registers except for the Instruction register.

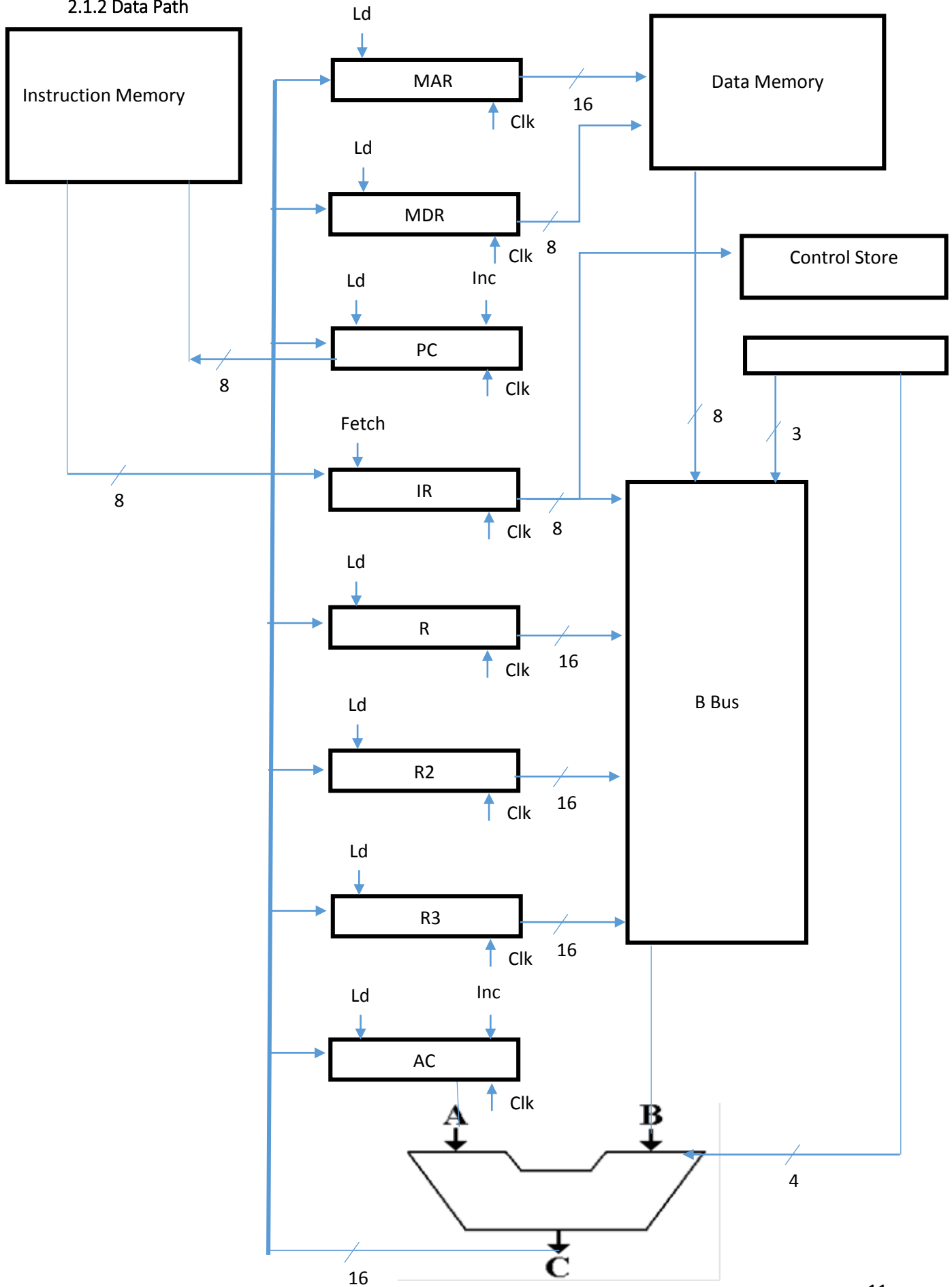
- **Control Store**

Control store is the most complex part in the ISA. Our ISA consists of 28 instructions which have been implemented using 53 micro instructions. Each micro instruction contains 22 control information bits and 8 next address bits.

Micro Instruction Format

		JUMP		ALU			CLEAR		Increment	CBUS write enable signals					
Z_en	JMPC	JMNZ					AC	END	PC	AC	R	R1	R2	PC	MDR

2.1.2 Data Path



2.1.3 Instruction Set

In the below table we have illustrated our instruction, microinstruction and what actually happens there in the process.

Instruction	Microinstruction	Command
FETCH	FETCH1	NULL
	FETCH2	$IR \leftarrow IMM$
	FETCH3	$PC \leftarrow PC + 1$
ADD	ADD1	$AC \leftarrow AC + R$
NOP	NOP1	FETCT
SUB	SUB1	$AC \leftarrow AC - R$
MUL	MUL1	$AC \leftarrow AC * R$
DIV	DIV1	$AC \leftarrow AC / R$
AND	AND1	$AC \leftarrow AC \wedge R$
OR	OR1	$AC \leftarrow AC \vee R$

NOT	NOT1	AC <= !AC
INC	INC1	AC <= AC+1
DEC	DEC1	AC <= AC^R
CLAC	CLAC1	AC <= 0
LDAC	LDAC1	NULL
	LDAC2	IR<=IMM
	LDAC3	PC<=PC+1, AC<=IR<<8
	LDAC4	NULL
	LDAC5	IR<=IMM
	LDAC6	PC<=PC+1, AC<=AC IR
JMNZ	JMNZZ1	NULL
	JMNZZ2	IR << IMM
	JMNZN1	PC <= PC + 1
STAC	STAC1	NULL
	STAC2	IR <= IMM, R <= AC
	STAC3	PC<=PC+1, AC<=IR<<8

	STAC4	NULL
	STAC5	IR<=IMM
	STAC6	PC<=PC+1, AC<=AC IR
	STAC7	MAR<=AC
LDADD	LDADD1	MAR <= AC
	LDADD2	NULL
STADD	STADD1	MAR <= R2
	STADD2	MDR <= AC
	STADD3	WRITE
MVACR	MVACR1	R <= AC
MVACR2	MVACR21	R2 <= AC
MVACR3	MVACR31	R3 <= AC
MVR	MVR1	AC <= R
MVR2	MVR21	AC <= R2
MVR3	MVR31	AC <= R3

JMNZ	JMNZZ3	PC<=PC+1, AC<=IR<<8
	JMNZZ4	NULL
	JMNZZ5	IR<=IMM
	JMNZZ6	PC<=PC+1, AC<=AC IR
	JMNZZ7	PC <= AC
	JMNZN2	PC <= PC + 1
	JMNZN3	CLRZ
LDAC	LDAC7	MAR<=AC
	LDAC8	NULL
	LDAC9	AC<=DMM
STAC	STAC8	MDR <= R
	STAC9	WRITE
LDADD	LDADD3	AC <= DMM
FINISH	FINISH1	MAR=1

2.1.4 Fetch, Decode and Execution Cycle

Fetching

In the fetching cycle of our ISA there are 3 stages. Fetching is also implemented in our control store and here are the microinstructions for the fetch cycle,

FETCH	FETCH1	NULL
	FETCH2	IR<=IMM
	FETCH3	PC<=PC+1

Decoding

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.

Executing

1. ADD

$$AC \leftarrow AC + R$$

This instructions plays a huge role in the ISA. In this instruction it adds the values in the AC which we get to the ALU via A bus and the value of R general purpose register which we get via the multiplexer in B bus. The final value is pushed to the C bus and written to AC again. This instruction has only one micro instruction.

2. NOP

Does nothing. We use this to when we need to skip a clock cycle.

3. SUB

$$AC \leq AC - R$$

This instruction it subtracts the values in the AC which we get to the ALU via A bus and the value of R general purpose register which we get via the multiplexer in B bus. The final value is pushed to the C bus written to AC again. This instruction has only one micro instruction

4. MUL

$$AC \leq AC * R$$

This instruction it multiplies the values in the AC which we get to the ALU via A bus and the value of R general purpose register which we get via the multiplexer in B bus. The final value is pushed to the C bus. This instruction has only one micro instruction.

5. DIV

$$AC \leq AC / R$$

This instruction it divides the values in the AC which we get to the ALU via A bus and the value of R general purpose register which we get via the multiplexer in B bus. The final value is pushed to the C bus. This instruction has only one micro instruction.

6. AND

$$AC \leq AC \wedge R$$

This instruction performs the bit wise AND operations to values in the AC which we get to the ALU via A bus and the value of R general purpose register which we get via the multiplexer in B bus. The final value is pushed to the C bus. This instruction has only one micro instruction.

7. OR

$$AC \leq AC \vee R$$

This instruction performs the bit wise OR operations to values in the AC which we get to the ALU via A bus and the value of R general purpose register which we get via the multiplexer in B bus. The final value is pushed to the C bus. This instruction has only one micro instruction.

8. NOT

$AC \leftarrow !AC$

This instruction inverts the value(bit wise) of Accumulator and put it back to the Accumulator. Again this is also an instruction with single micro instruction

9. INC

$AC \leftarrow AC+1$

INC instruction increments the value in AC by 1.

10. DEC

$AC \leftarrow AC-1$

DEC instruction decrements the value in AC by 1.

11. CLAC

$AC \leftarrow 0$

CLAC stands for clear Accumulator. So from this instruction 0 is assigned to the Accumulator.

12. LDAC

LDAC1 NULL

LDAC2 $IR \leftarrow IMM$

LDAC3 $PC \leftarrow PC+1, AC \leftarrow IR \ll 8$

LDAC4 NULL

LDAC5 $IR \leftarrow IMM$

LDAC6 $PC \leftarrow PC+1, AC \leftarrow AC \mid IR$

This instruction contains 6 micro instructions. The ultimate goal of this instruction is to Load a data in the location which has the address of MAR value to the Accumulator.

13. JMNZ

JMNZZ1	NULL
JMNZZ2	IR << IMM
JMNZN3	PC <= PC + 1

14. STAC

STAC1	NULL
STAC2	IR <= IMM, R <= AC
STAC3	PC <= PC+1, AC <= IR << 8
STAC4	NULL
STAC5	IR <= IMM
STAC6	PC <= PC+1, AC <= AC IR
STAC7	MAR <= AC

The purpose of STAC is to put the value in Accumulator to the Data memory which the location is specified by the address in the MAR. this instruction needs to perform 7 micro instructions to perform the instruction.

15. LDADD

LDADD1	MAR <= AC
LDADD2	NULL

This instruction loads the value in Accumulator to the MAR. 2 micro instructions needed to perform the instruction.

16. STADD

STADD1	MAR <= R2
STADD2	MDR <= AC
STADD3	WRITE

From this instruction first it puts the address in R2 general purpose register to the MAR and then puts the value in Accumulator to the MDR. Once we have those values in MAR and MDR we can perform a write. Then it will write value in MDR to the memory location specified by the value in MAR. This Instruction needs 3 micro instructions to complete this task.

17. MVACR

MVACR1 R \leq AC

MVACR copies the value in AC to R. This is a very important instruction since the primary purpose of having general purpose registers is to store temporary values, we should be able to get the values in Accumulator to the these general purpose registers.

18. MVACR2

MVACR21 R2 \leq AC

MVACR2 copies the value in AC to R2.

19. MVACR3

MVACR31 R3 \leq AC

MVACR3 copies the value in AC to R3.

20. MVR

MVR1 AC \leq R

MVR copies the value in R general purpose register to Accumulator. This instruction helps to restore the temporally stored values in general purpose registers.

21. MVR2

MVR21 AC <= R2

MVR2 copies the value in R2 general purpose register to Accumulator.

22. MVR3

MVR21 AC <= R3

MVR3 copies the value in R3 general purpose register to Accumulator.

23. FINISH

FINISH1 MAR=1

The use of this instruction is to mark the end of the program

3 Modeling the above designed architecture using Verilog.

3.1 Verilog implementation

Now we have an Instruction Set Architecture specially made to give a solution to our filtering and down sampling process. Even though it is a customized processor we have included some other general instructions too. Once the architecture is done we should implement that using Verilog hardware descriptive language to set up the processor on our FPGA board. We used Verilog modules to implement our components except for data memory and instruction memory. All the Verilog codes are attached here with Appendix A.

3.2 Memory implementation

As described above we need to have two separate memory blocks according to our ISA. So we thought of using the Xilinx LogiCORE™ IP Block Memory Generator to generate our memory blocks.

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.

Memory Types

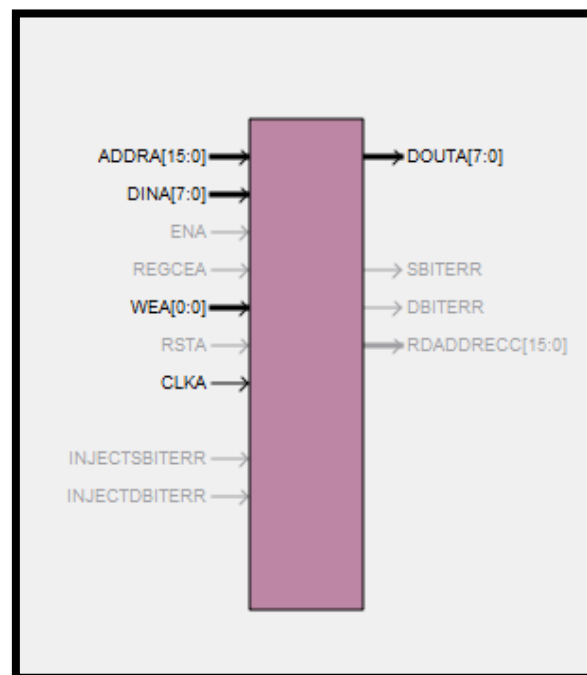
The Block Memory Generator core uses embedded block RAM to generate five types of memories:

- Single-port RAM
- Simple Dual-port RAM
- True Dual-port RAM
- Single-port ROM
- Dual-port ROM

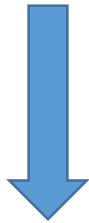
For dual-port memories, each port operates independently. Operating mode, clock frequency, optional output registers, and optional pins are selectable per port. For Simple Dual-port RAM, the operating modes are not selectable.

3.2.1 Data Memory

We used a 8 bit wide 65536 deep single port RAM to implement it.



Address	Data bits							
0x0000								
0x0001								
0x0010								
0x0011								



0xFFFF (65536)

3.2.2 Instruction Memory

We used a 8 bit wide 512 deep **single port ROM** to implement it.

Address	Data bits							
0b000000000								
0b000000001								
0b000000010								
0b000000011								

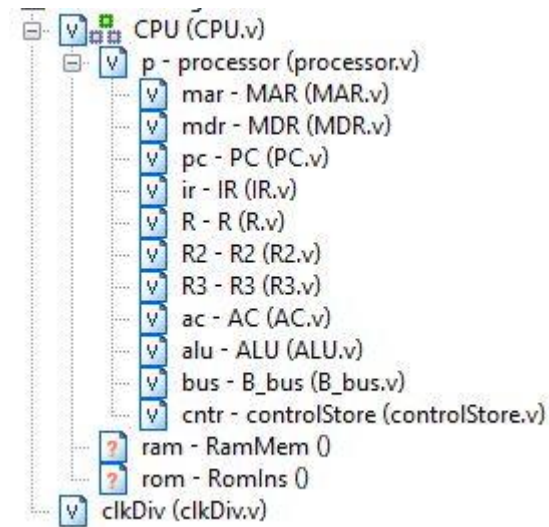


0b111111111 (512)

4 Implementing and testing the modules on FPGA board

4.1 Implemented architecture on FPGA board

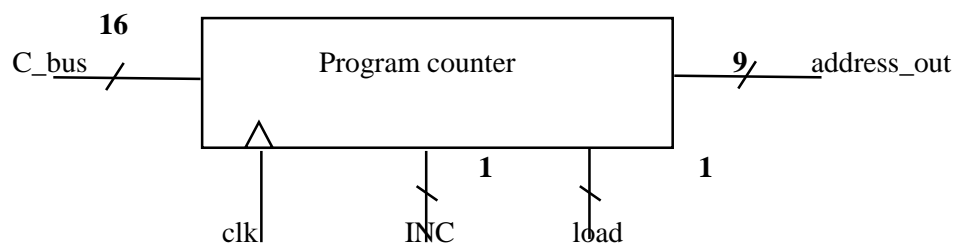
Modules



4.1.1 Registers

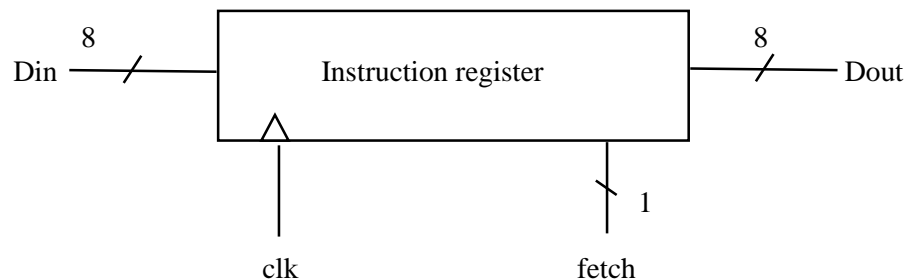
4.1.1.1 PC

The program counter is the register which keeps the address of the instruction to be executed next. Despite all the other registers and busses are either 8 bits or 16 bits long, the size of the PC is 9 bits since the program memory has only 512 memory locations and it would be a waste to have 16 bits instead. This is the only register to have an increment signal. This increment signal is used to increment the value of this register by “1” without needing to send its value through the ALU. Had the INC signal been high at the positive edge of the clock, the register value would be incremented. Since the incrementing of PC is very often used, adding of this INC signal saves lots of clock cycles.



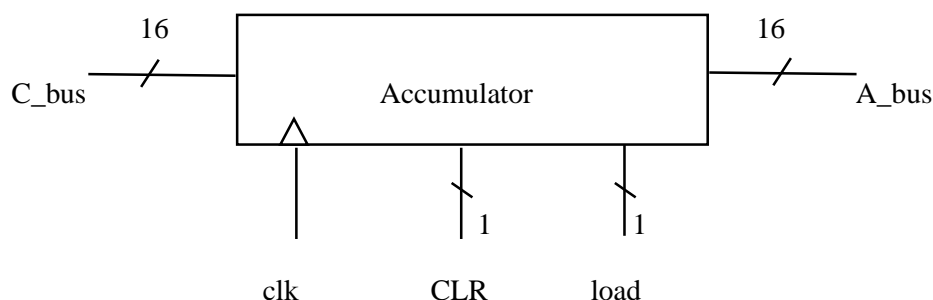
4.1.1.2 IR

Instruction Register reads instructions directly from the program memory where the PC points to, at the positive edge of the clock if the fetch command is high at that time and then it outputs this instruction directly to the control store and the B_bus. Therefore IR is also slightly different from the other registers since it is the only register to have a fetch command.



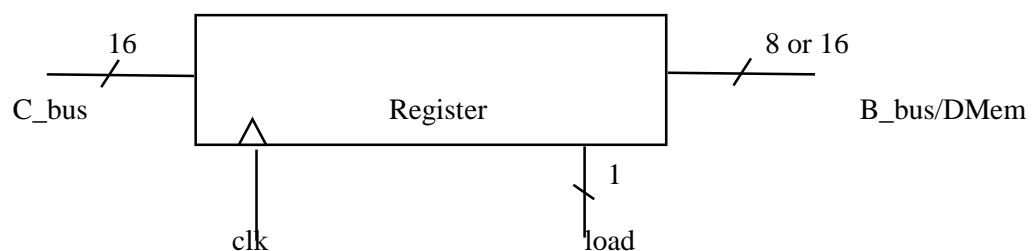
4.1.1.3 AC

Accumulator is another special register which 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 to zero as and when required. AC is widely used in almost all the operations carried out through the ALU and therefore it is very effective to have a special clear command dedicated to itself.



4.1.1.4 MAR, MDR and other general purpose registers

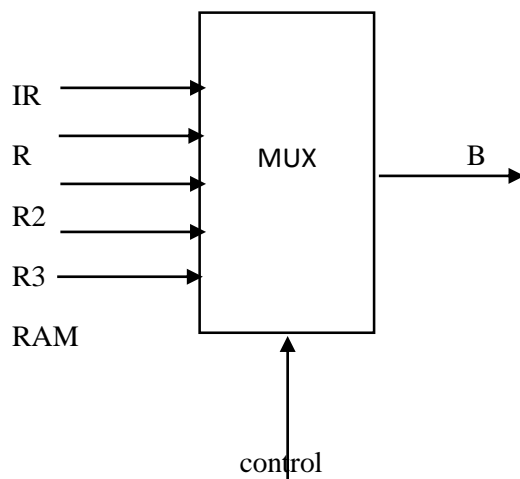
MAR and all the other 3 general purpose registers are 16 bit registers that has input from C bus whereas MDR is a 8 bit register. All the general purpose registers outputs to the B_bus module which is a multiplexer whereas MAR and MDR outputs are directly connected to the data memory as MAR points to 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.



4.1.2 B_bus

B bus is a multiplexer which has 6 inputs, 4 from registers (3 general purpose registers and IR) one from the control store as control signal and the other input directly from the RAM. Conceptually this B_bus is a bus connecting these general purpose registers, IR and RAM to the ALU's 2nd input. To control the access to the bus from these modules there needs to be a control signal coming from the control store.

This functionality has been achieved by connecting the output of the multiplexer to its inputs according to the control signal given from the control store which is 3 bits. The output of this B_bus is connected to the ALU. All the busses are 16 bits to avoid overflowing when doing addition and multiplication. Inputs from IR and RAM are only 8 bits and therefore zero padding is done by adding redundant 8 zeros in front to the values of these inputs.

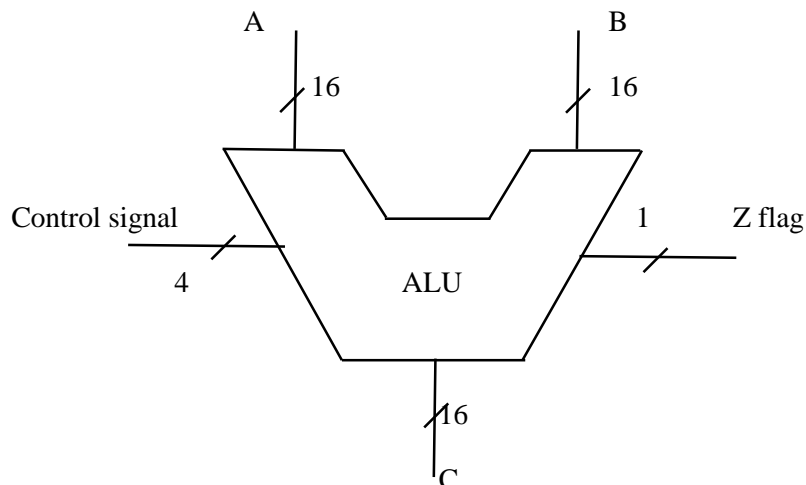


Control signal	Access given to
000	IR
001	R
010	R2
011	R3
100	RAM
default	High impedance

4.1.3 ALU

This is the module through which all the arithmetic and logic operations of the processor is done. This is a combinational logic circuit which is implemented using blocking statements. It has 2 16 bit inputs, one from each AC register and B_bus. The output of the ALU is the 16 bit C bus which connects to all the registers except IR. The load signal of each register will decide whether the data from C bus be taken in or not. In addition to those ports the ALU also has a 4 bit control signal from control store and a special flag called Z flag. The Z flag is used to indicate whether the result of a SUB operation is zero or not and it is used inside the control store to implement jump statements.

The Z flag is in reality a register and therefore the value of the Z flag is only changed at positive edge of the clock cycle. If the output of the SUB operation is equal to zero the value of the Z flag will be set to 1 whereas in all the other occasions it is 0.



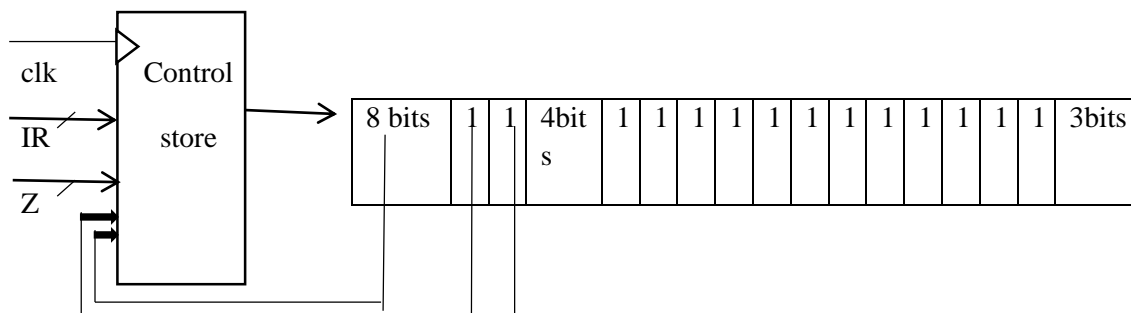
These are the available operations and their description

operation	Control signal	description
ADD	0000	$C=A+B$
SUB	0001	$C=A-B$
MUL	0010	$C=A*B$
DIV	0011	$C=A/B$
AND	0100	$C=A \& B$
OR	0101	$C=A B$
NOT	0110	$C=\sim A$

INC	0111	C=A+1
DEC	1000	C=A-1
LSHIFT8	1001	C=A<<8
AtoC	1010	C=A
BtoC	1011	C=B

4.1.4 Control store

This is the unit where all the control signals for this processor are stored. This module has 5 inputs and an output. Clock, the 8 bit address of the next micro instruction to be executed, the instruction which was read from the instruction memory, JMPC and JMNZ bits and Z flag are the inputs to this module and the only output is the 29 bits long control signal which is taken out at the negative edge of each clock cycles.



The format of the micro instruction is give below.

- **JMPC and JMNZ**

First 8 bits altogether gives the address of the next micro instruction to be executed in the next clock cycle. The next 2 bits are JMPC and JMNZ out of which JMPC is used to identify which address should the control store output in the next negative edge of the clock. If JMPC is 1 then IR is used as the next address and else the next address from the output is used as the next address. If the last micro instruction executed was the last of the FETCH command, the JMPC is set to “1” and otherwise zero.

JMNZ bit is used to implement the jump statement. If JMNZ is 1, that means the next address is neither of IR or next address from output. The next address is determined after checking the Z flag value. If Z flag is 1, the next address is 21 and if the Z flag is 0 the next address is 40.

Therefore inside the control store, the JMNZ bit is first checked, and then if it is high Z flag will be checked before determining the next address and if JMNZ is 0, then JMPC bit is checked and the output address will be determined accordingly.

Therefore the 8 bit next address and the next JMPC and JMNZ bits are fed back into the control store as its inputs.

- **ALU operation** - This is 4 bits long and directly connected to the ALU to control its operation.
- **CLR** – This one bit control signal is directly connected to the clear pin of the AC register. If CLR is 1 the value of AC will be set to 0, and otherwise there will be no effect.
- **FINISH** – If this bit is set to high, the processor will stop working immediately. This bit used only in one instruction which is used as the finish command to stop the processor.
- **PC_INC** – This bit is used to increment the PC register value by 1.
- **LOAD bits** – bits from 18th to 24th are used as load commands to registers AC, R, R1, R2, PC, MDR and MAR respectively. These bits will decide which registers are permitted to load the value from C bus enabling multiple load operations to several registers at the same time.
- **Memory control bits (WRITE and FETCH)**

These 2 bits are used to control the writing process of the data to the RAM and to control the fetching of the instructions from instruction memory. The WRITE bit is connected to the RAM and the value in the MDR register will be written to it only if the WRITE bit is 1 at the corresponding positive edge of the clock cycle.

The FETCH bit is connected to the IR register. The instruction to which the PC is pointed will be fetched in to the IR register only if the FETCH command is 1 at the corresponding positive edge.

- **B-bus control** - These last 3 bits are connected to the B_bus to control its output.

There are 25 instructions in our ISA and each instruction is comprised of one or more micro instructions from a set of 53 micro instructions. These 53 micro instructions are stored in the ROM of the control store. The memory location of the first micro instruction of a full instruction is called the opcode and it is a number. Following is the brief description of all instructions and its opcode (location where the first micro instruction of this instruction in the ROM)

Instruction	Opcode
FETCH – 3 micro instructions	0
ADD	2
NOP	3
SUB	4
MUL	5
DIV	6
AND	7
OR	8
NOT	9

INC	10
DEC	11
CLAC	12
LDAC	13
JMNZ	19
STAC	22
LDADD	29
STADD	31
MVACR	34
MVACR2	35
MVACR3	36
MVR	37
MVR2	38
MVR3	39
FINISH	53

4.1.5 Processor

This is the outer module which contains the instances of all the above mentioned modules and this can be considered as the processing and controlling unit of the processor. This module contains the instances of control store, ALU, B_bus, and all the registers MAR, MDR, PC, IR, R, R2, R3, and AC. The Memory modules, other communication modules and this module itself are instantiated inside the outmost wrapper module realCPU.

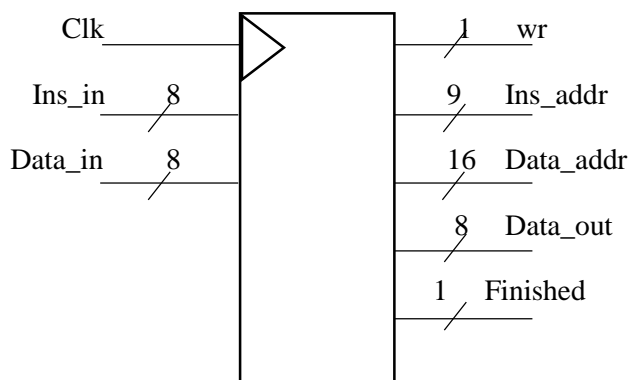
This module has 4 inputs and 5 outputs. Description about the inputs and the outputs are summarized below.

Inputs-

- clock/clk – clock input.
- Ins_in – this wire is to connect the IR register inside the processor with instruction memory to read instructions from the rom.
- Data_in – this wire connects data memory and B_bus inside the processor and used to read data from the memory.

Outputs –

- wr – WRITE command from control store to the RAM.
- Ins_addr – gives the instruction memory address to the ROM from PC.
- Data_addr – gives the data memory address to the RAM from MAR.
- Data_out – take data from MDR to RAM.
- Finished – FINISH command from the control store is taken out through this.



4.1.6 RamMem

This is the Ram module for the processor having storage of 65536 of 8 bit words. This single port RAM is created using block memory generator of the Spartan 6 board. This has 4 inputs and an output. Clock, 16 bit memory address, 8 bit data in and WRITE signal as inputs and data out as the output.

4.1.7 RomIns

This is the instruction memory of the processor having storage of 512, 8 bit words. This is a single port ROM module created using block memory generator of the Spartan 6 board. This has 2 inputs, clock and instruction address and an instruction output.

4.1.8 realCPU

This is the outmost wrapper module of this processor. This contains instances of processor module, RAM, ROM along with the communication modules selection, baud_rate_generator, and uart transmitter and receiver. This has 3 inputs reset, clock and transmitter wire out of which clock is connected to L15 pin of the Spartan 6 board which is the clock of the board.

This module connects the processor to the memory modules in turn making the complete processor. The clock of the processor and memory modules is provided through a clock divider which is inside the realCPU module.

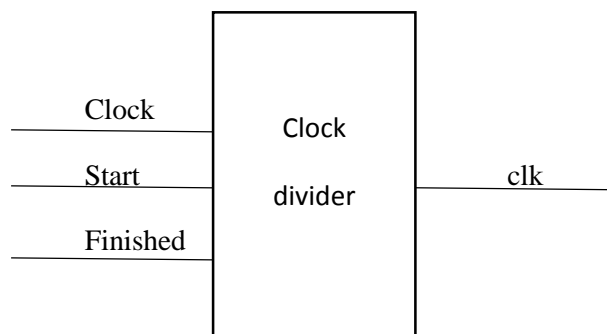
4.1.9 Clock divider

The processor module and the memory modules will receive the clock through this module. This module will take the inbuilt clock of the Spartan 6 FPGA board as its input and will output a delayed clock which will be given as the input to those aforementioned modules.

The main use of this clock divider is to create a way to switch off the processor after the processing is done. If the finish command is given that means if the FINISH bit is HIGH then this clock divider module will stop working in turn stopping the processor as it will no longer receive any clock. This is how we achieve to having a FINISH command in our Instruction set Architecture. This FINISH command is essential to have the UART communication with the computer.

Another use of this clock divider module is to reduce the frequency of the clock module given to the processor. The inbuilt clock of the Spartan 6 FPGA board is 100MHz which is very high frequency which means the execution of the instruction has to be done in very small time. Since there is only half cycle time in between the generating of the micro instruction and executing it, this frequency seemed a bit too much when implementing on the board even though it worked fine on simulation. The output of this clock module is a 20MHz clock which is fast enough to do the processing quickly enough while having no errors.

Even though the clock divider is implemented inside the real CPU module itself, the structure of this clock divider can be illustrated in a block diagram as follows.



4.2 Testing

4.2.1 Testing and Simulations

We first started creating module by module which contains inside the processor. For each module we made separate test codes to check the functionality of the module. (Unit Testing) To simulate the test results, software called 'ISim' have been used. In Verilog there are some syntaxes only supports for simulations we have used them to create test modules. Test code created to make a clock pulse is given below.

```

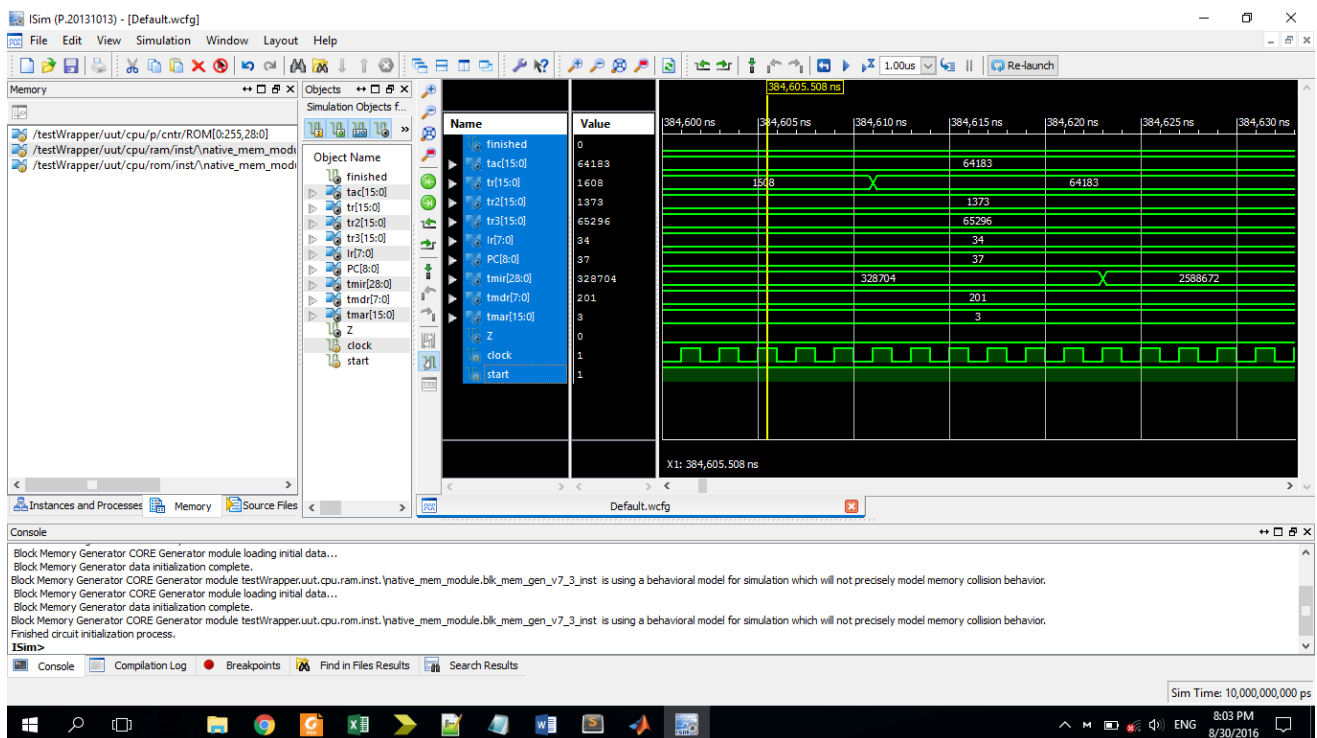
initial begin
    enable = 1;
    clock = 1;#5;
    while(1) begin
        clock= ~clock;
        #5;
    end
end
end

```

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.



Simulated Results

In the testing process first we ran simple programs on processor for test the functionality and timing of each instruction.

Following shows an example, which stores numbers from 0 to 9 in contiguous memory locations, by using a loop structure.

4.2.2 Assembly code

1. CLAC
2. STAC I
3. LDAC adr
4. MVACR2
5. LDAC I
6. STADD
7. LDAC I
8. INC
9. STAC I
10. MVR2
11. INC
12. STAC adr
13. LDAC I
14. MVACR
15. LDAC N
16. SUB
17. JMNZ LOOP
18. FINISH

Simulation Objects f...		
Object Name		
finished		
tac[15:0]		
tr[15:0]		
tr2[15:0]		
tr3[15:0]		
lr[7:0]		
PC[8:0]		
tmir[28:0]		
tmdr[7:0]		
tmar[15:0]		
Z		
clock		
start		

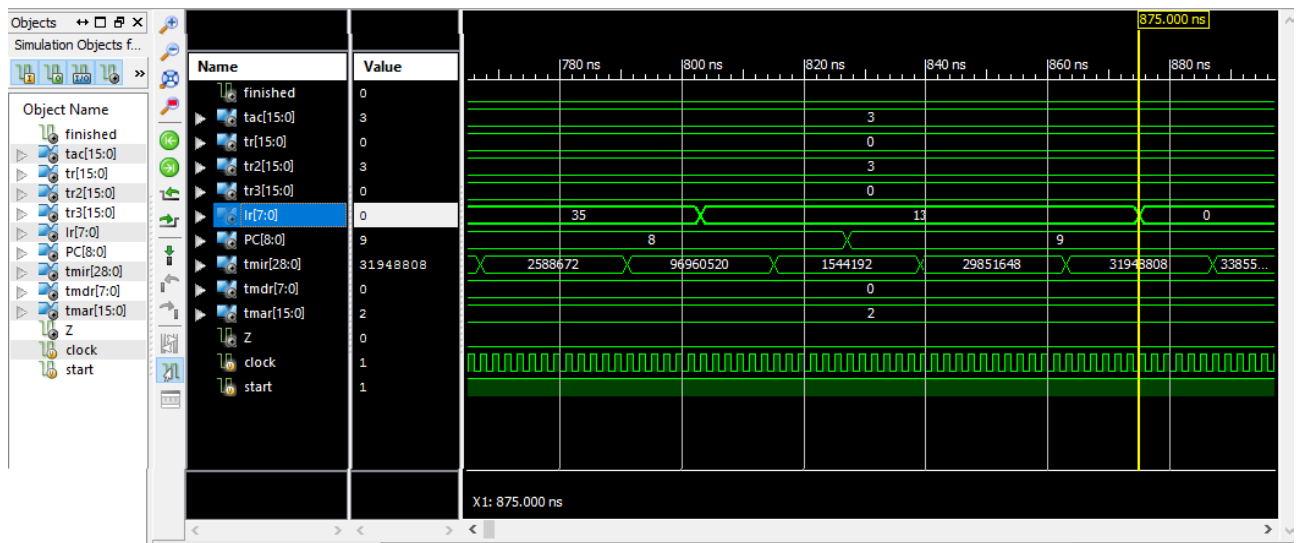
0	10
1	0
2	3
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0

DATA RAM before processing

Simulation Objects f...		
Object Name		
finished		
tac[15:0]		
tr[15:0]		
tr2[15:0]		
tr3[15:0]		
lr[7:0]		
PC[8:0]		
tmir[28:0]		
tmdr[7:0]		
tmar[15:0]		
Z		
clock		
start		

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

DATA RAM after processing



Simulated Results

5 UART communication

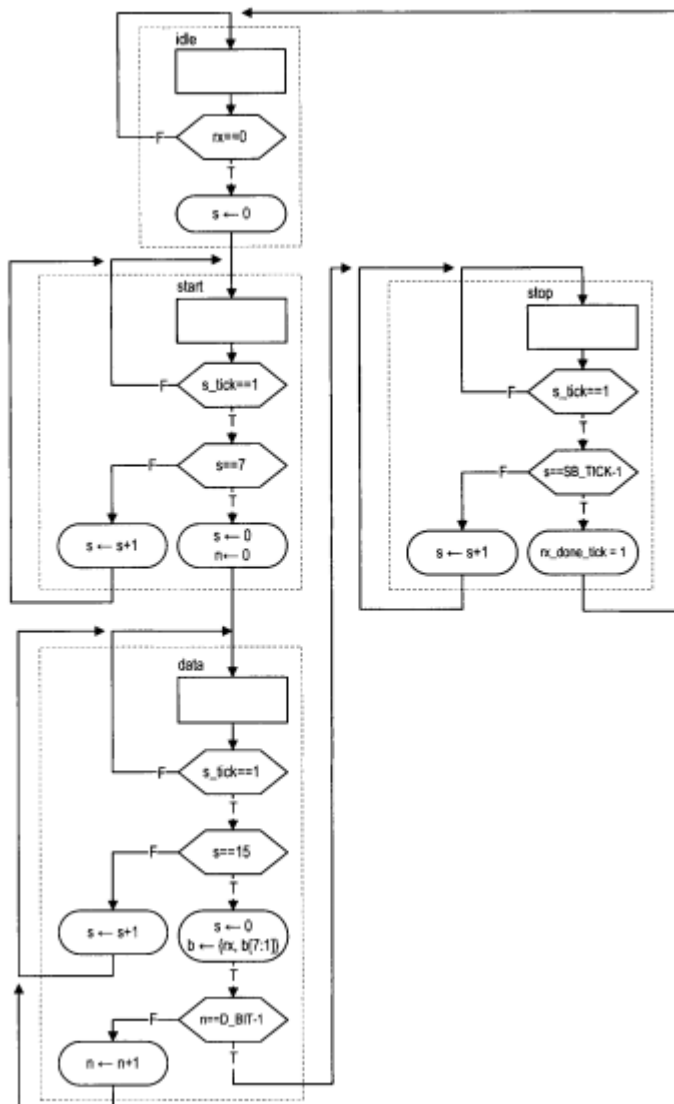
5.1 Baud rate generator

The baud rate generator generates a sampling signal whose frequency is exactly 16 times the UART's designated baud rate. To avoid creating a new clock domain and violating the synchronous design principle, the sampling signal should function as enable ticks rather than the clock signal to the UART receiver.

5.2 Receiving a data stream through UART

With an understanding of the oversampling procedure, we can derive the ASMD chart accordingly, as shown in Figure. To accommodate future modification, two constants are used in the description. The DBIT constant indicates the number of data bits, and the SB-TICK constant indicates the number of ticks needed for the stop bits, which is 16, 24, and 32 for 1, 1.5, and 2 stop bits, respectively. DBIT and SB-TICK are assigned to 8 and 16 in this design. The chart includes three major states, start, data, and stop, which represent the processing of the start bit, data bits, and stop bit. The s-tick signal is the enable tick from the baud rate generator and there are 16 ticks in a bit interval. Note that the FSM stays in the same state unless the s-tick signal is asserted. There are two counters, represented by the s and n registers. The s register keeps track of the number of sampling ticks and counts to 7 in the start state, to 15 in the data state, and to SB-TICK in the stop state. The n register keeps track of the number of data bits received in the data state. The retrieved bits are shifted into and reassembled in the b Figure ASMD chart of a UART receiver. Register, A status signal, rx-done-tick, is included. It is asserted for one clock cycle after the receiving process is completed.

The Verilog implemented code is in Appendix



5.3 Sending a data stream which is in the ram using UART

The organization of a UART transmitting subsystem is similar to that of the receiving subsystem. It consists of a UART transmitter, baud rate generator, and interface circuit. The interface circuit is similar to that of the receiving subsystem except that the main system sets the flag FF or writes the FIFO buffer, and the UART transmitter clears the flag FF or reads the FIFO buffer. The UART transmitter is essentially a shift register that shifts out data bits at a specific rate. The rate can be controlled by one-clock-cycle enable ticks generated by the baud rate generator. Because no oversampling is involved, the frequency of the ticks is 16 times slower than that of the UART receiver. Instead of introducing a new counter, the UART transmitter usually shares the baud rate generator of the UART receiver and

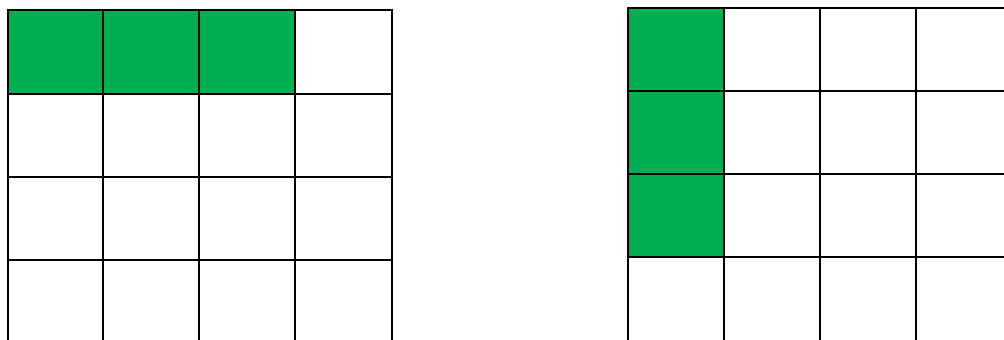
uses an internal counter to keep track of the number of enable ticks. A bit is shifted out every 16 enable ticks. The ASMD chart of the UART transmitter is similar to that of the UART receiver. After assertion of the tx-start signal, the FSMD loads the data word and then gradually progresses through the start, data, and stop states to shift out the corresponding bits. It signals completion by asserting the tx-done-tick signal for one clock cycle. A 1-bit buffer, tx-reg, is used to filter out any potential glitch. UART source codes are attached with the appendix E.

6 Algorithm

We used two main concepts when developing our algorithm for the project. We broke down the algorithm to two parts where first we filtered the image with a Gaussian filter and then we downsampled the image. In the filtering part we used two kernels 1x3 and 3x1 for 'X' axis and 'Y' axis respectively. A c pseudo code for the algorithm is attached with Appendix D.

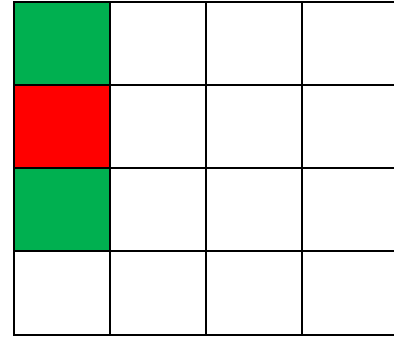
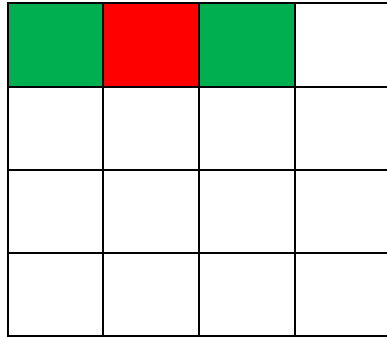
6.1 Filtering the image

For the filtering part, 1x3 Gaussian Kernel and 3x1 Gaussian Kernel has been used and the weights of those kernels are shown below.



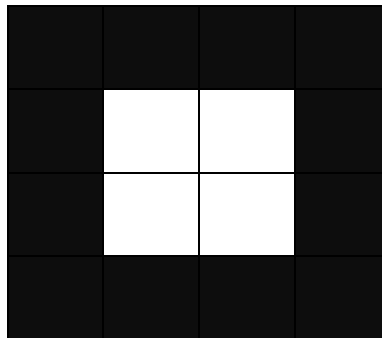
Gaussian Kernel

First we did filtering for the 'X' axis. Therefore, when the 1x3 kernel overlaps on 3 pixels, we multiply the values of these 3 pixels with the respective value of the overlapped pixel of the Gaussian kernel. After the multiplication, we put the sum to the center pixel as shown below. Finally, the values are normalized by dividing the sum of the values in the kernel.



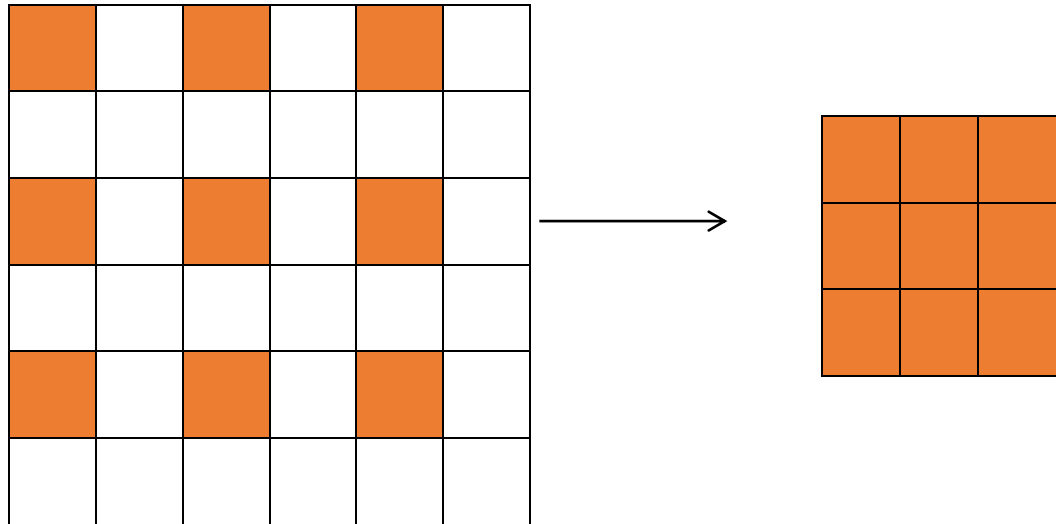
filtering method

Since the kernel can't overlap completely with the first row, last row, first column and the last column, we padded the image with 0 valued bits around for sides of the image using matlab before passing the image to the FPGA.



6.2 Down sampling the image

In the down sampling algorithm, what we did was to start from the cell (0,0) and take the left upper corner pixel's value of every 4 pixel block. The values are stored overlapping the values of the filtered image to overcome the memory issues. The algorithm will be graphically explained below.



Down-sampling method

8 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.

8.1 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.

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**

252 pixels



252 pixels

Original Image

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

252 pixels



252 pixels

Filtered Image Using Matlab

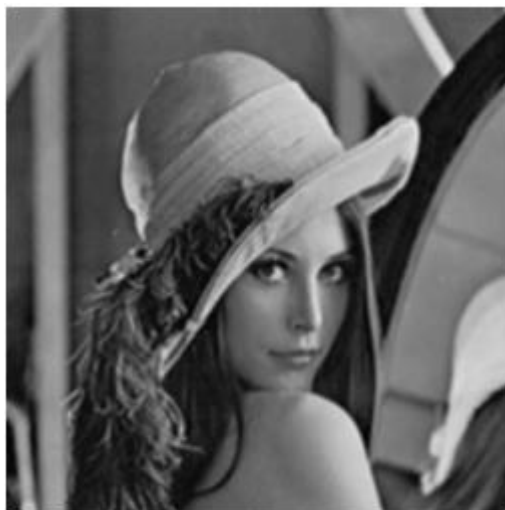
Down Sampled Image (Using Matlab)



Down Sampled Image Using Matlab

8.2 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.



FPGA Reference Gaussian Filtered image



FPGA Processed Gaussian Filtered image



Reference Down-Sampled image



FPGA Processed Down-Sampled image

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

8.3 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 we convert it to unsigned integer it stores X as '11'.

But in the hardware level:

$X = 43/4 \square 101011 \gg 2$; answer is 1010 which is '10' in decimal.

So 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 $X = (43 + 2)/4 \square 101101 \gg 2$; 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


	1	2	3	4	5	6	7	8	9	10	11	12	13	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	0	0	0	0	0	0	0	0	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	0	0	0	0	0	
8	0	0	0	0	0	0	0	0	0	0	0	0	0	
9	0	0	0	0	0	0	0	0	0	0	0	0	0	
10	0	0	0	0	0	0	0	0	0	0	0	0	0	
11	0	0	0	0	0	0	0	0	0	0	0	0	0	
12	0	0	0	0	0	0	0	0	0	0	0	0	0	
13	0	0	0	0	0	0	1	1	1	0	0	0	0	2
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	4	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
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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

realCPU Project Status (09/03/2016 - 17:19:32)			
Project File:	processor7777.xise	Parser Errors:	No Errors
Module Name:	realCPU	Implementation State:	Programming File Generated

Target Device:	xc6slx45-3csg324	• Errors:	No Errors
Product Version:	ISE 14.7	• Warnings:	<u>55 Warnings (0 new)</u>
Design Goal:	Balanced	• Routing Results:	<u>All Signals Completely Routed</u>
Design Strategy:	<u>Xilinx Default (unlocked)</u>	• Timing Constraints:	<u>All Constraints Met</u>
Environment:	<u>System Settings</u>	• Final Timing Score:	0 <u>(Timing Report)</u>

Device Utilization Summary				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	191	54,576	1%	
Number used as Flip Flops	190			
Number used as Latches	1			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	339	27,288	1%	
Number used as logic	332	27,288	1%	
Number using O6 output only	264			
Number using O5 output only	16			
Number using O5 and O6	52			
Number used as ROM	0			
Number used as Memory	0	6,408	0%	

Number used exclusively as route-thrus	7			
Number with same-slice register load	4			
Number with same-slice carry load	3			
Number with other load	0			
Number of occupied Slices	118	6,822	1%	
Number of MUXCYs used	64	13,644	1%	
Number of LUT Flip Flop pairs used	350			
Number with an unused Flip Flop	171	350	48%	
Number with an unused LUT	11	350	3%	
Number of fully used LUT-FF pairs	168	350	48%	
Number of unique control sets	22			
Number of slice register sites lost to control set restrictions	57	54,576	1%	
Number of bonded <u>IOBs</u>	12	218	5%	
Number of LOCed IOBs	12	12	100%	
Number of RAMB16BWERs	32	116	27%	
Number of RAMB8BWERs	1	232	1%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	2	16	12%	
Number used as BUFGs	2			
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	1	58	1%	
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	4.81			

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	Sat Sep 3 17:18:19 2016	0	49 Warnings (0 new)	6 Infos (0 new)
Translation Report	Current	Sat Sep 3 17:18:27 2016	0	0	0
Map Report	Current	Sat Sep 3 17:18:50 2016	0	3 Warnings (0 new)	6 Infos (0 new)
Place and Route Report	Current	Sat Sep 3 17:19:05 2016	0	0	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Sat Sep 3 17:19:13 2016	0	0	4 Infos (0 new)
Bitgen Report	Current	Sat Sep 3 17:19:30 2016	0	3 Warnings (0 new)	1 Info (0 new)

Secondary Reports			[-]
Report Name	Status	Generated	
WebTalk Report	Current	Sat Sep 3 17:19:31 2016	
WebTalk Log File	Current	Sat Sep 3 17:19:32 2016	

8.4 Test Results

Original



Matlab Filterd



FPGA Filtered



MatLab Downsampled



FPGA DOWNSAMPLED



Total Pixels (Original Image)	$252 \times 252 = 63504$
Total Pixels (Down sampled Image)	$126 \times 126 = 15876$
Error 0 Pixels	15822
Error non zero Pixels	54
Error Percentage	0.34%

Original



Matlab Filterd



FPGA Filtered



MatLab Downsampled



FPGA DOWNSAMPLED



Total Pixels (Original Image)	$252 \times 252 = 63504$
Total Pixels (Down sampled Image)	$126 \times 126 = 15876$
Error 0 Pixels	15810
Error non zero Pixels	66
Error Percentage	0.42%

Original



Matlab Filterd



FPGA Filtered



MatLab Downsampled

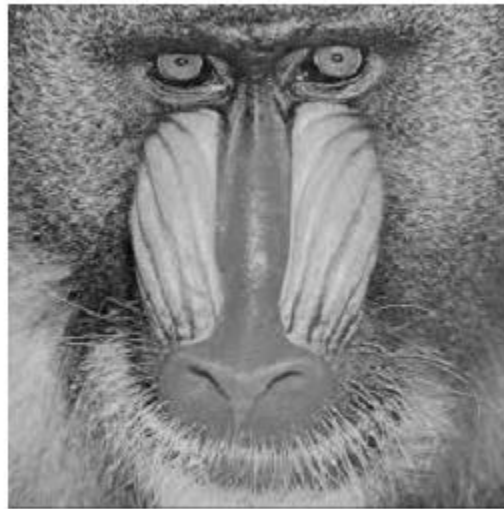


FPGA DOWNSAMPLED



Total Pixels (Original Image)	$252 \times 252 = 63504$
Total Pixels (Down sampled Image)	$126 \times 126 = 15876$
Error 0 Pixels	15798
Error non zero Pixels	54
Error Percentage	0.49%

Original



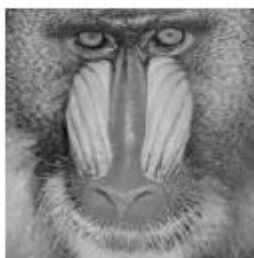
Matlab Filterd



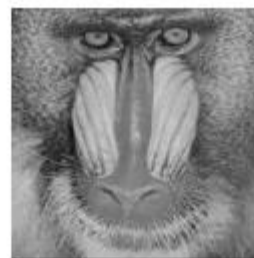
FPGA Filtered



MatLab Downsampled



FPGA DOWNSAMPLED



Total Pixels (Original Image)	252 x 252 = 63504
Total Pixels (Down sampled Image)	126x126= 15876
Error 0 Pixels	15761
Error non zero Pixels	115
Error Percentage	0.72%

Original



Matlab Filterd



FPGA Filtered



MatLab Downsampled



FPGA DOWNSAMPLED



Total Pixels (Original Image)	$252 \times 252 = 63504$
Total Pixels (Down sampled Image)	$126 \times 126 = 15876$
Error 0 Pixels	15743
Error non zero Pixels	133
Error Percentage	0.84%

Original



Matlab Filterd



FPGA Filtered



MatLab Downsampled



FPGA DOWNSAMPLED



Total Pixels (Original Image)	$252 \times 252 = 63504$
Total Pixels (Down sampled Image)	$126 \times 126 = 15876$
Error 0 Pixels	15847
Error non zero Pixels	29
Error Percentage	0.18%

Appendix A

Verilog implementation of ISA

MAR.v

```
`timescale 1ns / 1ps
module MAR(clk,Ld,C,addr);

    input clk;
    input Ld;
    input [15:0] C;
    output [15:0] addr;

    reg [15:0] addr=0;

    always@(posedge clk)
    begin
        if(Ld)
        begin
            addr<=C;
        end
    end

end

endmodule
```

MDR.v

```
`timescale 1ns / 1ps
module MDR(clk,Ld,C,Dout);

    input clk;
    input Ld;
    input [15:0] C;
    output [7:0] Dout;

    reg [7:0] Dout=0;

    always@(posedge clk)
    begin
        if(Ld)
        begin
            Dout<=C[7:0];
        end
    end

end

endmodule
```

PC.v

```
`timescale 1ns / 1ps
module PC(clk, Ld, C, INC, addr );
```

```

input clk;
input Ld;
input [15:0] C;
input INC;
output [8:0] addr;

reg [8:0] addr=9'd0;

always@(posedge clk)
begin

    if(INC) addr<=addr+9'b000000001;
    if(Ld) addr<=C[8:0];

end

endmodule

```

IR.v

```

`timescale 1ns / 1ps
module IR(clk,fetch,Din,Dout);
    input clk;
    input fetch;
    input [7:0] Din;
    output [7:0] Dout;

    reg [7:0] Dout=0;

    always@(posedge clk)
    begin
        if(fetch)
        begin
            Dout<=Din;
        end
    end
endmodule

```

R.v

```

`timescale 1ns / 1ps
module R(clk,Ld,C,Dout);

    input clk;
    input Ld;
    input [15:0] C;
    output [15:0] Dout;

    reg [15:0] Dout=0;

    always@(posedge clk)
    begin
        if(Ld)
        begin

```

```

        Dout<=C;
    end
end

endmodule

```

R2.v

```

`timescale 1ns / 1ps
module R2(clk,Ld,C,Dout);

    input clk;
    input Ld;
    input [15:0] C;
    output [15:0] Dout;

    reg [15:0] Dout=0;

    always@(posedge clk)
    begin
        if(Ld)
        begin
            Dout<=C;
        end
    end

endmodule

```

R3.v

```

`timescale 1ns / 1ps
module R3(clk,Ld,C,Dout);

    input clk;
    input Ld;
    input [15:0] C;
    output [15:0] Dout;

    reg [15:0] Dout=0;

    always@(posedge clk)
    begin
        if(Ld)
        begin
            Dout<=C;
        end
    end

endmodule

```

AC.v

```

`timescale 1ns / 1ps
module AC(clk,Ld,CLR,C,A);

```

```

input CLR;
input clk;
input Ld;
input [15:0] C;
output [15:0] A;

reg [15:0] A=0;

always@(posedge clk)
begin
    if(Ld)
    begin
        A<=C;
    end
    else if(CLR) begin
        A<=0;
    end
end

endmodule

ALU.v

`timescale 1ns / 1ps
module ALU(clk,control,A,B,C,Z);

    input clk;
    input [15:0] A,B;
    input [3:0] control;
    output [15:0] C;
    reg [15:0] C;

    output Z;
    reg Z=0;

    reg z_assign = 1'b0;

    parameter ADD=4'd0, SUB=4'd1, MUL=4'd2, DIV=4'd3, AND=4'd4, OR=4'd5, NOT=4'd6,
    INC=4'd7, DEC=4'd8, LSHIFT8=4'd9, AtoC=4'd10, BtoC=4'd11;//setZ=14;

    always@(B or control)
    begin
        case(control)
            ADD:C=A+B;
            SUB: begin
                C=A-B;
                if(C==16'd0) z_assign=1'd1;
                else z_assign=1'd0;
            end
            MUL:C=A*B;
            DIV:C=A/B;
            AND:C=A & B;
            OR:C=A|B;

```

```

        NOT:C=~A;
        INC:C=A+16'd1;
        DEC:C=A-16'd1;
        LSHIFT8:C=B<<8;
        AtoC:C=A;
        BtoC:C=B;
        default: C=16'd0;
        //setZ:Z=0;
    endcase

end

/*always@(posedge clk)
begin
    if(C==0) Z=1;
    else Z=0;
end*/

always@(posedge clk) begin
    if(control==SUB)
        Z=z_assign;
    else
        Z=Z;
end

endmodule

```

B_bus.v

```

`timescale 1ns / 1ps
module B_bus(mem,R,R2,R3,IR,control,bus);

    input [7:0] mem;
    input [7:0] IR;
    input [15:0] R;
    input [15:0] R2;
    input [15:0] R3;
    input [2:0] control;
    output [15:0] bus;

    reg [15:0] bus;

    always@(mem or IR or R or R2 or R3 or control)
    begin
        case(control)
            3'b000:bus<={8'b00000000,IR};
            3'b001:bus<=R;
            3'b010:bus<=R2;
            3'b011:bus<=R3;
            3'b100:bus<={8'b00000000,mem};
            default bus<=16'bz;
        endcase
    end

end

```

```
endmodule
```

controlStore.v

```
`timescale 1ns / 1ps
```

```
module controlStore(clk,addr,IR,MIR,JMP,Z);
```

```
    input clk;
    input [7:0] addr;
    input [7:0] IR;
    input [1:0] JMP;
    input Z;
    output [28:0] MIR;
```

```
    reg [7:0] address;
    reg [28:0] MIR;
```

```
    reg [28:0] ROM [0:255];
```

```
    ///////////////////////////////////////////////////
    initial
    begin
```

```
        ROM[0]=2588672;
        ROM[1]=96960520;
        ROM[46]=1544192;
        ROM[2]=2049;
        ROM[3]=491520;
        ROM[4]=34817;
        ROM[5]=67585;
        ROM[6]=100353;
        ROM[7]=133121;
        ROM[8]=165889;
        ROM[9]=198657;
        ROM[10]=231425;
        ROM[11]=264193;
        ROM[12]=507904;
        ROM[13]=29851648;
        ROM[14]=31948808;
        ROM[15]=33855488;
        ROM[16]=36143104;
        ROM[17]=38240264;
        ROM[18]=98736128;
        ROM[19]=42434560;
        ROM[20]=84901896;
        ROM[21]=94867456;
        ROM[22]=48726016;
        ROM[23]=50660360;
        ROM[24]=52729856;
        ROM[25]=55017472;
        ROM[26]=57114632;
        ROM[27]=58890240;
        ROM[28]=105185312;
```

```

ROM[29]=63242272;
ROM[30]=109543424;
ROM[31]=67469346;
ROM[32]=69533760;
ROM[33]=491536;
ROM[34]=328704;
ROM[35]=328192;
ROM[36]=327936;
ROM[37]=362497;
ROM[38]=362498;
ROM[39]=362499;
ROM[40]=86284288;
ROM[41]=88571904;
ROM[42]=90669064;
ROM[43]=92444672;
ROM[44]=327808;
ROM[45]=495616;//113741824
ROM[47]=100991008;
ROM[48]=103251968;
ROM[49]=362500;
ROM[50]=107315265;
ROM[51]=491536;
ROM[52]=362500;
ROM[53]=499712;
//ROM[54]=458752;

```

```
end
```

```
////////////////////////////////////
```

```
always@(negedge clk)
```

```
begin
```

```
    if(JMP[0]==0)
```

```
        begin
```

```
            if(JMP[1]==0)
```

```
                begin
```

```
                    address<=addr;
```

```
                end
```

```
            else if(JMP[1])
```

```
                begin
```

```
                    address<=IR;
```

```
                end
```

```
        end
```

```
    else if(JMP[0])
```

```
        begin
```

```
            if(Z)
```

```
                begin
```

```
                    address<=8'd21;
```

```
                end
```

```
            else if(Z==0)
```

```
                begin
```

```
                    address<=8'd40;
```

```
                end
```

```
        end
```

```
end
```



```

        else begin address<=8'd0; end
    end

    always@(address)
    begin
        MIR<=ROM[address];
    end

endmodule

processor.v

`timescale 1ns / 1ps
module processor(clock,clk,wr,Ins_addr,Ins_in,Data_addr,Data_in,Data_out,finished);

    input clock;
    input clk;
    output wr;
    output [8:0] Ins_addr;
    input [7:0] Ins_in;
    output [15:0] Data_addr;
    input [7:0] Data_in;
    output [7:0] Data_out;
    output finished;

    ////////////////////////////////////////////////// temporary ports
    /*output [15:0]tac;
    output [15:0]tr;
    output [15:0]tr2;
    output [15:0]tr3;
    output [7:0]Ir;
    output [8:0]tIAddr;
    output [28:0]tmir;
    output Z;*/
    ////////////////////////////////////////////////// assigning temp ports

    /*assign tac=a;
    assign tr=r;
    assign tr2=r2;
    assign tr3=r3;
    assign Ir=Ins;
    assign tIAddr=w;
    assign tmir=mir;
    assign Z=z;*/
    //////////////////////////////////////////////////

    assign wr=mir[4];
    assign Ins_addr=w;
    assign finished=mir[13];

    wire [28:0] mir;
    wire [7:0] Ins;
    wire [15:0] c;

```

```

wire [15:0] r;
wire [15:0] r2;
wire [15:0] r3;
wire [15:0] b;
wire [15:0] a;
wire z;
wire [8:0] w;

MAR mar(clk,mir[5],c,Data_addr);
MDR mdr(clk,mir[6],c,Data_out);
PC pc(clk,mir[7],c,mir[12],w);
IR ir(clk,mir[3],Ins_in,Ins);

R R(clk,mir[10],c,r);
R2 R2(clk,mir[9],c,r2);
R3 R3(clk,mir[8],c,r3);

AC ac(clk,mir[11],mir[14],c,a);

ALU alu(clk,mir[18:15],a,b,c,z);

B_bus bus(Data_in,r,r2,r3,Ins,mir[2:0],b);

controlStore cntr(clk,mir[28:21],Ins,mir,mir[20:19],z);

endmodule

```

APPENDIX B

Other Verilog implementations

Selection.v

```

`timescale 1ns / 1ps
module selection(
    input pro_work,
    input pro_finish,

    input real_clk,
    input proc_clk,

    input uart_write_EN,
    input proc_write_EN,

    input [15:0] uart_addr,
    input [15:0] proc_addr,

    input [7:0] uart_din,
    input [7:0] proc_din,

```

```

output reg tx_start,
output reg ram_clk,
output reg ram_write_EN,
output reg [15:0] ram_addr,
output reg [7:0] ram_din
);

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

always@(*) begin
    if(~pro_work && ~pro_finish) PC_state<=2'b00;//receiving data
    else if(pro_work && ~pro_finish) PC_state<=2'b01;//processing data
    else if(pro_work && pro_finish) PC_state<=2'b10;//transmitting data
    else PC_state<=2'b11;          //null state
end

always@(*)
    case(PC_state)
        2'b00:begin
            tx_start<=0;
            ram_clk<=real_clk;
            ram_write_EN<=uart_write_EN;
            ram_addr<=uart_addr;
            ram_din<={8'd0,uart_din};

            end
        2'b01:begin
            tx_start<=0;
            ram_clk<=proc_clk;
            ram_write_EN<=proc_write_EN;
            ram_addr<=proc_addr;
            ram_din<=proc_din;

            end
        2'b10:begin
            tx_start<=1;
            ram_clk<=real_clk;
            ram_write_EN<=uart_write_EN;
            ram_addr<=uart_addr;
            ram_din<={8'd0,uart_din};

            end
        default:begin
            tx_start<=0;
            ram_clk<=1'd0;
            ram_write_EN<=1'd0;
            ram_addr<=16'd0;
            ram_din<=16'd0;

            end
    endcase

endmodule

baud_rate_generator.v

```

```

`timescale 1ns / 1ps
module baud_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@(posedge clk, posedge reset)
        if(reset
            r_reg <=0;
        else
            r_reg <= r_next;

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

endmodule

```

uart_rx.v

```

`timescale 1ns / 1ps
module uart_rx
    #(parameter DBIT = 8,
        SB_TICK = 16
    )
    (
        input wire clk, reset,
        input wire rx, s_tick,
        output reg rx_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 @(posedge clk, 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
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
                            assign dout = b_reg;
endmodule

uart_tx.v

`timescale 1ns / 1ps
module uart_tx
    #(
        parameter DBIT = 8,
                                SB_TICK = 16
    )
    (
        input    wire clk,reset,
        input    wire tx_start, s_tick,
        input    wire [7:0] din,
        output reg tx_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;
    reg tx_reg, tx_next;

    always@(posedge clk, 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;
            end
        end
endmodule

```

```

        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
                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
                            end
                        else
                            s_next = s_reg + 4'd1;
                        end
                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
    end
endcase
end

assign tx=tx_reg;

endmodule

```

Top Module

realCPU.v

```

`timescale 1ns / 1ps
module realCPU(clk,reset,rx,tx,led);

    input clk,reset;
    input rx;
    output tx;
    output reg [7:0] led;

    reg [15:0] address;
    reg wea;
    wire [7:0] data_out;
    reg [7:0] count;
    wire s_tick;
    wire tx_done_tick;
    reg [7:0] din;
    //reg tx_start;
    wire [7:0] dout;

    initial begin
        enable=1'b1;
        //ena=1'b1;
        wea=1'b1;
        led=8'b00000000;
        address=16'd0;
        count=8'd0;
        //tx_start=1'b0;
    end

    //-----to turn off the processor when finished and turn on when tx finished

```



```

        wire finished;
        reg start_processor=1'b0;

//----- clock divider-----
reg clk_D=1'd0;
reg [31:0] counter=32'd0;

always @(posedge clk)
    begin
        if(~finished & start_processor)
            begin
                if (counter == 32'd5)//25000000
                    begin
                        clk_D = ~clk_D;
                        counter=0;
                    end
                else counter = counter + 32'd1 ;
            end
        else clk_D=0;
    end

//processor wires into selection module
    wire wr;
    wire [15:0] dataAddr;
    wire [7:0] dataOut;

//instruction memory wires
    wire [8:0] InsAddr;
    wire [7:0] InsIn;

//ram memory wires used in selection module
    wire ram_clk;
    wire ram_wr;
    wire tx_start;
    wire [15:0] ram_addr;
    wire [7:0] ram_data;

//-----
//DMem ram(clk,wr,dataAddr,dataOut,dataIn);

    RamMem ram (
        .clka(ram_clk), // input clka
        .wea(ram_wr), // input [0 : 0] wea
        .addra(ram_addr), // input [15 : 0] addra
        .dina(ram_data), // input [7 : 0] dina
        .douta(data_out) // output [7 : 0] douta
    );

    //InstructionMemory rom(InsAddr,InsIn);

    RomIns rom (
        .clka(~clk), // input clka /*****THERE IS SOME DIFFERENCE IN HERE WITH
DILSHAN'S PROCESSOR
        .addra(InsAddr), // input [8 : 0] addra

```

```

        .douta(InsIn) // output [7 : 0] douta
    );

    processor p(clk,clk_D,wr,InsAddr,InsIn,dataAddr,data_out,dataOut,finished);

    //-----
    selection selection(
        .pro_work(start_processor),
        .pro_finish(finished),

        .real_clk(clk),
        .proc_clk(clk_D),

        .uart_write_EN(wea),
        .proc_write_EN(wr),

        .uart_addr(address),
        .proc_addr(dataAddr),

        .uart_din(dout),
        .proc_din(dataOut),

        .tx_start(tx_start),
        .ram_clk(ram_clk),
        .ram_write_EN(ram_wr),
        .ram_addr(ram_addr),
        .ram_din(ram_data)
    );

    //-----

    baud_rate_generator bdg(
        .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)
    );

```

```

always@(posedge clk )
begin

    if (address==16'd16395 && rx_done_tick) begin////////*****16'b65535 is the last address of
the memory
        led<=8'd255;
        wea<=1'b0;
        address<=16'd0;
        //count<=count+8'd1;
        start_processor<=1'b1;
        din<=data_out[7:0];
    end

    else if (rx_done_tick && wea) begin
        address<=address+16'd1;
        led<=data_out[7:0];
    end

    else if(tx_done_tick && address<16'd16395) begin
        address<=address+16'd1;
        din<=data_out[7:0];
        led<=8'b10101010;//address[13:6];
    end
    //else if(count>8'd5 && rx_done_tick) begin
    //led<=data_out;
    //address<=address+16'd1;
    //end
end

endmodule

```

APPENDIX C

MatLab one dimensional pixel array generator and UART communication

communicator.m

```

s = serial('COM16', 'BaudRate', 1562500);
% 1562500

%% Input Image

% InImage_color=imread('cameraman.png');
% InImage=rgb2gray(InImage_color);
% InImage=imread('cameraman.png');
InImage=[10,30,4,2,20,20,6,8;18,16,120,140,16,18,140,120;10,30,4,2,20,20,6,8;18,16,120,140,16,18,
140,120;10,30,4,2,20,20,6,8;18,16,120,140,16,18,140,120;10,30,4,2,20,20,6,8;18,16,120,140,16,18,1
40,120];
imshow(InImage);
InImage_matlab=double(InImage);

% Prepare Input Image for transmission
InImageD=double(InImage);

```

```

InImageArray=InImageD(:);

%Preparing appending dataset
[row,col]=size(InImage);
i=0;j=0;
wo=12;
col_2=row/2;
offset=12;
k1=1;k2=2;k3=1;
div=4;
temp=0;
data=double([row;col;i;j;col_2;wo;offset;k1;k2;k3;div;temp]);

%Transmitting dataset
x=[data;InImageArray];

N=length(x);
M=(row/2)*(col/2)+12;
a=zeros(N,1);

fopen(s);
disp('Port Open');

for i=1:N
    fwrite(s,x(i),'uint8');
end

disp('Write Done');
for i=1:N
    a(i,1)=fread(s,1);
end

fclose(s);

%Processing for out Image
OutImage=[a(14:M+1)];
OutImage=OutImage';
outImage=OutImage;

Result=reshape(uint8(OutImage),[row/2,col/2]);

figure,imshow(Result);

MatLab_OutImage=zeros(row/2,col/2);
%MatLab_FilterImage=zeros(row,col);

for j=2:col-1
    temp=InImage_matlab(1,j);
    for i=2:row-1
        total=0;

```

```

        total=total+temp*1;
        total=total+InImage_matlab(i,j)*2;
        total=total+InImage_matlab(i+1,j)*1;
        temp=InImage_matlab(i,j);
        InImage_matlab(i,j)=floor(total/4);
    end
end

for i=2:row-1
    temp=InImage_matlab(i,1);
    for j=2:col-1
        total=0;
        total=total+temp*1;
        total=total+InImage_matlab(i,j)*2;
        total=total+InImage_matlab(i,j+1)*1;
        temp=InImage_matlab(i,j);
        InImage_matlab(i,j)=floor(total/4);
    end
end

MatLab_FilterImage=InImage_matlab;

% wo=1;
%
% for i=1:2:row
%     for j=1:2:col
%         MatLab_OutImage(wo)=InImage_matlab(i,j);
%         wo=wo+1;
%     end
% end
MatLab_OutImge=MatLab_FilterImage(1:2:end,1:2:end);
MatLab_Reshaped_Result=reshape(uint8(MatLab_OutImge),[row/2,col/2]);
% Difference=MatLab_FilterImage-Result;
Error=MatLab_Reshaped_Result-Result;
Squared_Error=Error.*Error;

Sum_Square_error=sum(sum(Squared_Error));
mean_sqrt_error=sqrt(Sum_Square_error*(row*col));

mean_square_error_presentage=mean_sqrt_error*(100/255);

disp('mean square error ');
mean_square_error_presentage

disp('Zero Error Pixels :-')
Myzero=zeros(row/2,col/2);
No_Error_zero=sum(sum(Error==Myzero))/(row*col/4)

disp('One Error Pixels :- ')
Myone=ones(row/2,col/2);
No_Error_one=sum(sum(Error==Myone))/(row*col/4)

disp('Total Error : -')
No_Error_one+No_Error_zero

```

```
disp('Error greater than 1')
sum(sum(Myone<Error))
```

MatLab Filtering and down sampling code

MatLab_Downsampling.m

```
% % Input Image

% InImage_color=imread('Fruit.png');
% InImage=rgb2gray(InImage_color);
InImage=imread('Lora.png');
figure,imshow(InImage);
InImage_matlab=double(InImage);

% Prepare Input Image for transmission
InImageD=double(InImage);
InImageArray=InImageD(:);

[row,col]=size(InImage);

MatLab_OutImage=zeros(row/2,col/2);

% [1,2,1] gaussian kernal y direction filtering

for j=2:col-1
    temp=InImage_matlab(1,j);
    for i=2:row-1
        total=0;
        total=total+temp*1;
        total=total+InImage_matlab(i,j)*2;
        total=total+InImage_matlab(i+1,j)*1;
        temp=InImage_matlab(i,j);
        InImage_matlab(i,j)=floor(total/4);
    end
end
% [1,2,1] gaussian kernal x direction filtering

for i=2:row-1
    temp=InImage_matlab(i,1);
    for j=2:col-1
        total=0;
        total=total+temp*1;
        total=total+InImage_matlab(i,j)*2;
        total=total+InImage_matlab(i,j+1)*1;
        temp=InImage_matlab(i,j);
        InImage_matlab(i,j)=floor(total/4);
    end
end
MatLab_FilterImage=InImage_matlab; % Matlab Filtered Image
```

```
MatLab_OutImage=MatLab_FilterImage(1:2:end,1:2:end); %Matlab downsampling
```

```
MatLab_Reshaped_Result=reshape(uint8(MatLab_OutImage),[row/2,col/2]);  
figure, imshow(MatLab_Reshaped_Result);
```

APPENDIX D

C code for algorithm

```
#include<stdio.h>  
#include<math.h>  
  
int main()  
{  
  
int row=128,col=128;  
int k1=1;k2=2;k3=1;  
int div=(k1+k2+k3);  
int in_image[row][col];  
int out_image[row/2][col/2];  
int i;  
int j;  
int temp;  
int total;  
  
for(j=1;j<col-1;j++)  
{  
temp=in_image[1][j];  
for(i=1;j<row-1;i++)  
{  
total=0;  
total=total+temp*k1;  
total=total+in_image[i][j]*k2;  
total=total+in_image[i+1][j]*k3;  
temp=in_image[i][j];  
in_image[i][j]=total/div;  
}  
}  
  
for(i=1;j<row-1;i++)  
{  
temp=in_image[i][1];  
for(j=1;j<col-1;j++)  
{
```

```

        total=0;
        total=total+temp*k1;
        total=total+in_image[i][j]*k2;
        total=total+in_image[i][j+1]*k3;
        temp=in_image[i][j];
        in_image[i][j]=total/div;
    }
}

int wo=0;

for(i=1;j<row-1;i+=2)
{
    for(j=1;j<col-1;j+=2)
    {
        out_image[wo]=in_image[i][j];
        wo=wo+1;
    }
}
}
return 0;
}

```

APPENDIX E

UART communication

baud_rate_generator.v

```

`timescale 1ns / 1ps
module baud_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@(posedge clk, posedge reset)
    if(reset)
        r_reg <=0;
    else
        r_reg <= r_next;

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

endmodule

```

uart_rx.v

```

`timescale 1ns / 1ps
module uart_rx
    #(parameter DBIT = 8,
        SB_TICK = 16
    )
    (
        input wire clk, reset,
        input wire rx, s_tick,
        output reg rx_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 @(posedge clk, 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
                                assign dout = b_reg;

endmodule

uart_tx.v

`timescale 1ns / 1ps
module uart_tx
    #(
        parameter DBIT = 8,
                                SB_TICK = 16
    )
    (
        input  wire clk,reset,
        input  wire tx_start, s_tick,
        input  wire [7:0] din,
        output reg tx_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;
    reg tx_reg, tx_next;

    always@(posedge clk, 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

```

```

        end
    else
        begin
            state_reg <= state_next;
            s_reg <= s_next;
            n_reg <= n_next;
            b_reg <= b_next;
            tx_reg <= tx_next;
        end
    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
                                    end
                                else
                                    s_next = s_reg + 4'd1;
                                end
                            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
                                end
                            endcase
                        end

                        assign tx=tx_reg;

                    endmodule

```

selection.v

```

`timescale 1ns / 1ps
module selection(
    input pro_work,
    input pro_finish,

    input real_clk,
    input proc_clk,

    input uart_write_EN,
    input proc_write_EN,

    input [15:0] uart_addr,
    input [15:0] proc_addr,

```

```

input [7:0] uart_din,
input [7:0] proc_din,

output reg tx_start,
output reg ram_clk,
output reg ram_write_EN,
output reg [15:0] ram_addr,
output reg [7:0] ram_din
);

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

always@(*) begin
    if(~pro_work && ~pro_finish) PC_state<=2'b00;//receiving data
    else if(pro_work && ~pro_finish) PC_state<=2'b01;//processing data
    else if(pro_work && pro_finish) PC_state<=2'b10;//transmitting data
    else PC_state<=2'b11;          //null state
end

always@(*)
    case(PC_state)
        2'b00:begin
            tx_start<=0;
            ram_clk<=real_clk;
            ram_write_EN<=uart_write_EN;
            ram_addr<=uart_addr;
            ram_din<={8'd0,uart_din};

            end
        2'b01:begin
            tx_start<=0;
            ram_clk<=proc_clk;
            ram_write_EN<=proc_write_EN;
            ram_addr<=proc_addr;
            ram_din<=proc_din;

            end
        2'b10:begin
            tx_start<=1;
            ram_clk<=real_clk;
            ram_write_EN<=uart_write_EN;
            ram_addr<=uart_addr;
            ram_din<={8'd0,uart_din};

            end
        default:begin
            tx_start<=0;
            ram_clk<=1'd0;

```

```

                                ram_write_EN<=1'd0;
                                ram_addr<=16'd0;
                                ram_din<=16'd0;
                                end
                                endcase
                                endmodule

```

References

1.CPU basics

<http://file.seekpart.com/keywordpdf/2011/5/19/2011519124210902.pdf>

2. Block Memory Generator

http://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_2/pg058-blk-mem-gen.pdf

3 FPGA PROTOTYPINGBY VERILOG EXAMPLES

<http://www2.dc.ufscar.br/~marcondes/netfpga/FPGAPrototypingByVerilogExamples.pdf>