Department of Electronic & Telecommunication Engineering

University of Moratuwa

EN 3030: Circuits and Systems Design

FPGA BASED PROCESSOR DESIGN

FINAL REPORT

Name	Index number
Kalupahana I.N.	150285X
Warnakula W.D.S.	150665M
Wickramasinghe K.R.B.	150686D
Wijesinghe C.B.	150699U

Supervisor

Dr. Jayathu Samarawickrama

This report is submitted in partial fulfillment of the requirements

for the module EN 3030: Circuits and Systems Design.

20th July 2018

ABSTRACT

Task specific custom processors are popular in todays consumer electronics world ever since its introduction. Factors such as low power consumption, economic feasibility, efficiency, reliability, ability to be enclosed in very small spaces etc, are some of the reasons why this trend has developed and why these kinds of processors play a pivotal role in todays world.

This report contains the detailed discussion of the design and hardware implementation of a custom processor using Field Programmable Gate Array, specifically on a Alterra DE-2 board. The main task of the processor was for filtering an input image and then down sampling it by some factor. Report is furnished with methods and theories used for filtering and down sampling an image. Difference of the output image of the processor and a Matlab code is compared in later part of the report. Verilog code, Assembly code, and ISA are attached as reference.

TABLE OF CONTENTS

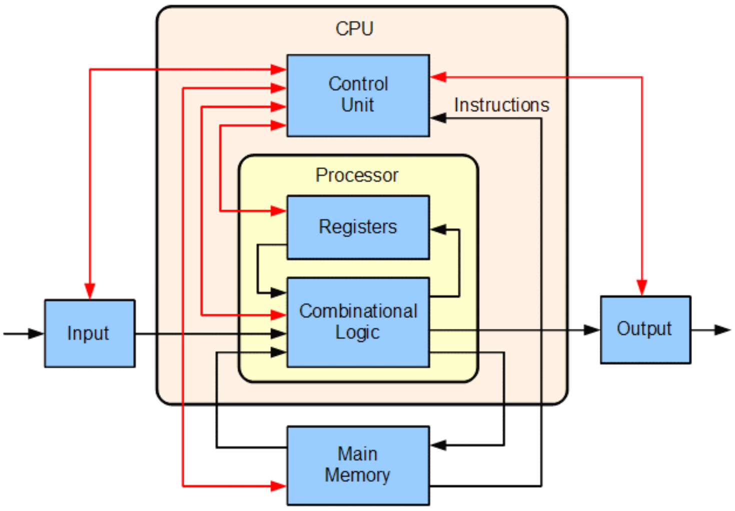
01. INTRODUCTION

1.1 PROCESSOR DESIGN

The objective of this project is to design a Microprocessor and a CPU (Central Processing Unit) which can filter and down sample a given image. The simulations are done using Verilog Hardware Description Language (HDL) and the implementations are done using ‘Quartus’ II web edition version 13.0 along with Altera DE2-115 Education and Development board with Cyclone IV FPGA. This report includes the Microprocessor and CPU design, the test codes that are used, and the physical hardware implementation of it.

1.2 CENTRAL PROCESSING UNIT (CPU)

The basic arithmetic, logical, input/output (I/O) operations specified by the instructions need to be performed to carry out the instructions of a computer program. The electronic circuitry within a computer which performs the above-mentioned task is called the *Central Processing Unit (CPU)*. The *Processing Unit* and the *Control Unit (CU)* are mainly identified as the Processor of a CPU by distinguishing these core elements of a computer from external components such as main memory and I/O circuitry.



Block diagram of a basic uniprocessor - CPU

1.3 MICROPROCESSOR

A *microprocessor* on the other hand, 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. It is a multipurpose, clock driven, register based, digital-integrated circuit which accepts 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.

1.4 PROBLEM STATEMENT

The main task of the given project is to design a processor to down sample an image. The down sampling factor, resolution of the original image, type of the image (colour / gray) were not constrained. According to the application, main requirements for the designing processor are filtering and down sampling a given image. As for the verification of the designed processor, a down sampled image using Matlab and the output image from the processor need to be compared and calculate the standard error.

The followings are the constraints that were selected for the processor design.

- Original image size is 256x256 pixels.
- Down sampling factor is 2.
- Image types are gray and colour.

1.5 PROPOSED SOLUTION BASED ON FPGA

As mentioned in the previous section, the main requirements for the application of the processor are filtering and down sampling an image. Therefore, pixel data of the original image need to be stored in the memory unit of the processor before hand. After completing the down sampling of the image, processed pixel data need to be sent back to the computer and obtain the visualize the output image. Hence, the task flow of the processor can be identified as,

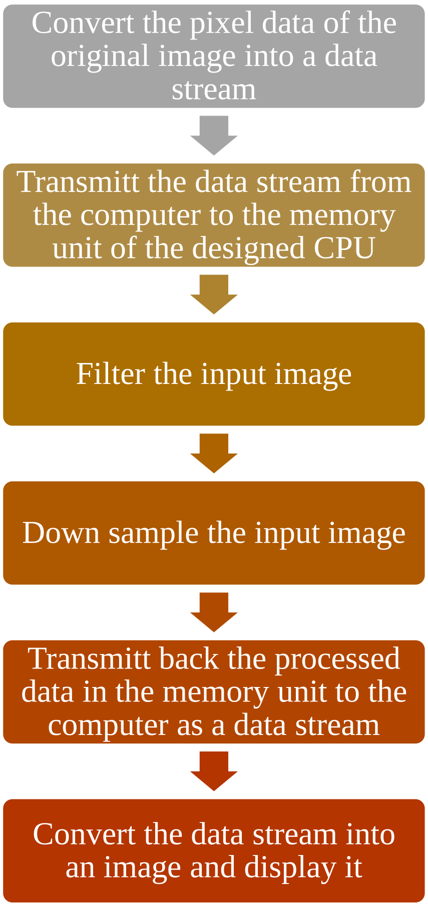

Convert the pixel data of the original image into a data stream
Transmitt the data stream from the computer to the memory unit of the designed CPU
Filter the input image
Down sample the input image
Transmitt back the processed data in the memory unit to the computer as a data stream
Convert the data stream into an image and display it

Figure 1: Task flow

The FPGA development board selected for the implementation of the processor is Altera DE2-115. Therefore, data transmission can be done using a UART module, so that the input data need to be transmitted through the UART module as a data stream. To achieve this, the pixel values of the original image needed to be converted to a data stream. This task can be achieved using Matlab. Then converted data can be transmitted through a COM port of the computer using the 'serial' function of Matlab.

After receiving this data by the processor, it stores data in its memory unit, so that data can be accessed when required. When data reception is completed, the processor starts to filter the pixel values according to a filtering algorithm. Filtered data are again stored in the memory unit. After filtering all the pixel values, the processor starts down sampling pixel by pixel. Output data are again stored in the memory unit.

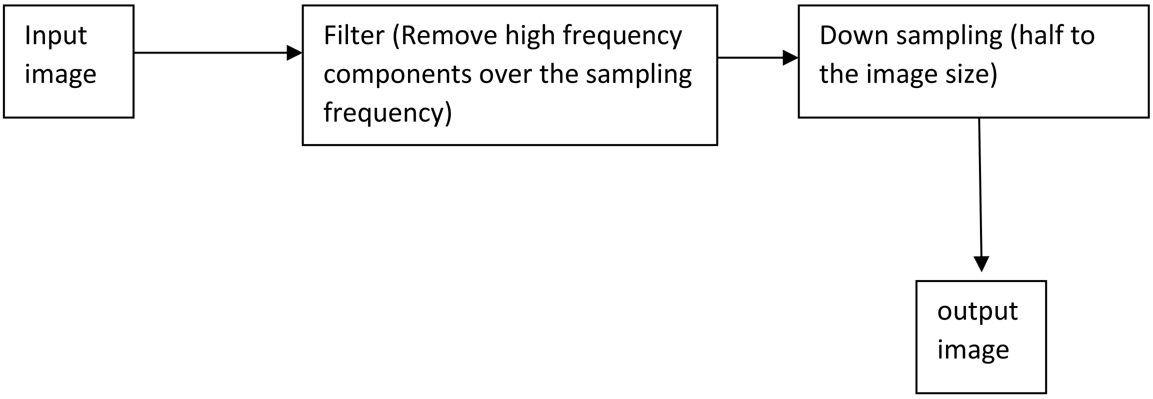
After completing the down sampling process, data need to be transmitted back to the computer for the visualization. Therefore, data stored in the memory transmitted back to the computer through the UART module as a data stream. Then, using Matlab, these data are converted into a matrix of the size of expecting down sampled image. Finally, the matrix is converted to an image and displayed as an image.

After deciding the task flow of the project, core requirements of the designing process can be decided as follows.

- A UART module needs to be implemented to transmit and receive data from the computer.
- To store pixel data of the image in the CPU, the designing memory unit needs to be larger than the total number of pixels.
- The designing ISA for the processor should be capable of handling the filtering and down sampling tasks.

1.6 REQUIRED ALGORITHMS

Project is designed to realize a FPGA processor for down sampling an image.


Input image
Filter (Remove high frequency components over the sampling frequency)
Down sampling (half to the image size)
output image

The processor is mainly based on two algorithms as follows.

- Filtering algorithm
- Down sampling algorithm

Filtering algorithm

The filtering algorithm is based on the 3x3 Gaussian kernel given in Figure 03. The middle pixel location is considered as (0,0) and the standard deviation s 0.6. Used Gaussian function is;

$$Gaussian(x,y) = e^{\frac{(x^2+y^2)}{2\sigma^2}}$$

0.0622	0.2489	0.0622
0.2489	1	0.2489
0.0622	0.2489	0.0622

Figure 2: Gaussian Kernel

But processing decimal values are complex when designing the processor. Therefore, the kernel is first normalized so that the sum of the kernel is a power of 2 and then approximated to integer values.

0.99	3.99	0.99
3.99	16	3.99
0.99	3.99	0.99

Figure 3: Normalized kernel

If the value 3.99 is approximated to 4, the sum of the kernel is 36 which cannot be written as a power of 2. Therefore, it is approximated to 3 which give the sum 32. But the effect of error needs to be considered during the evaluation.

1	3	1
3	16	3
1	3	1

Figure 4: Approximated Kernel

Then the filtering an image can be done using the approximated kernel by moving the kernel along the image. Normalized, weighted sum is taken as the filtered value of a pixel.

Down sampling algorithm

A simple down sampling algorithm is used for this processor to avoid errors. According to this algorithm, if the down sampling factor is k, 1 pixel is selected from adjacent k pixels of the filtered image while maintaining the order. This can be easily visualized as follows where the down sampling factor is 4.

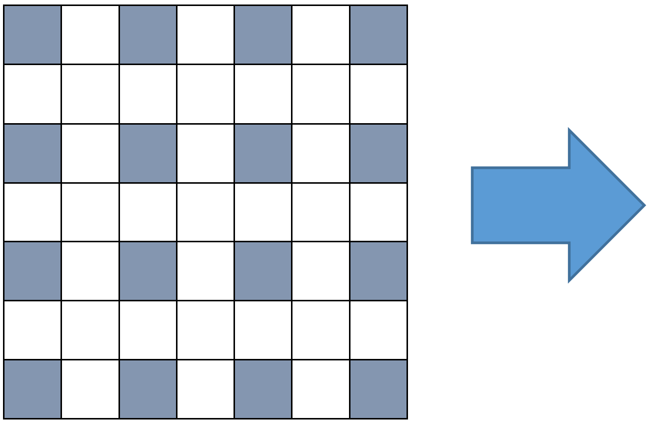


Figure 6: Filtered image

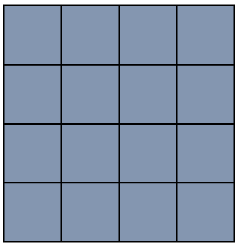


Figure 7: Down sampled image

Implementation of these algorithms in assembly code and Matlab is described in the following sections of the document.

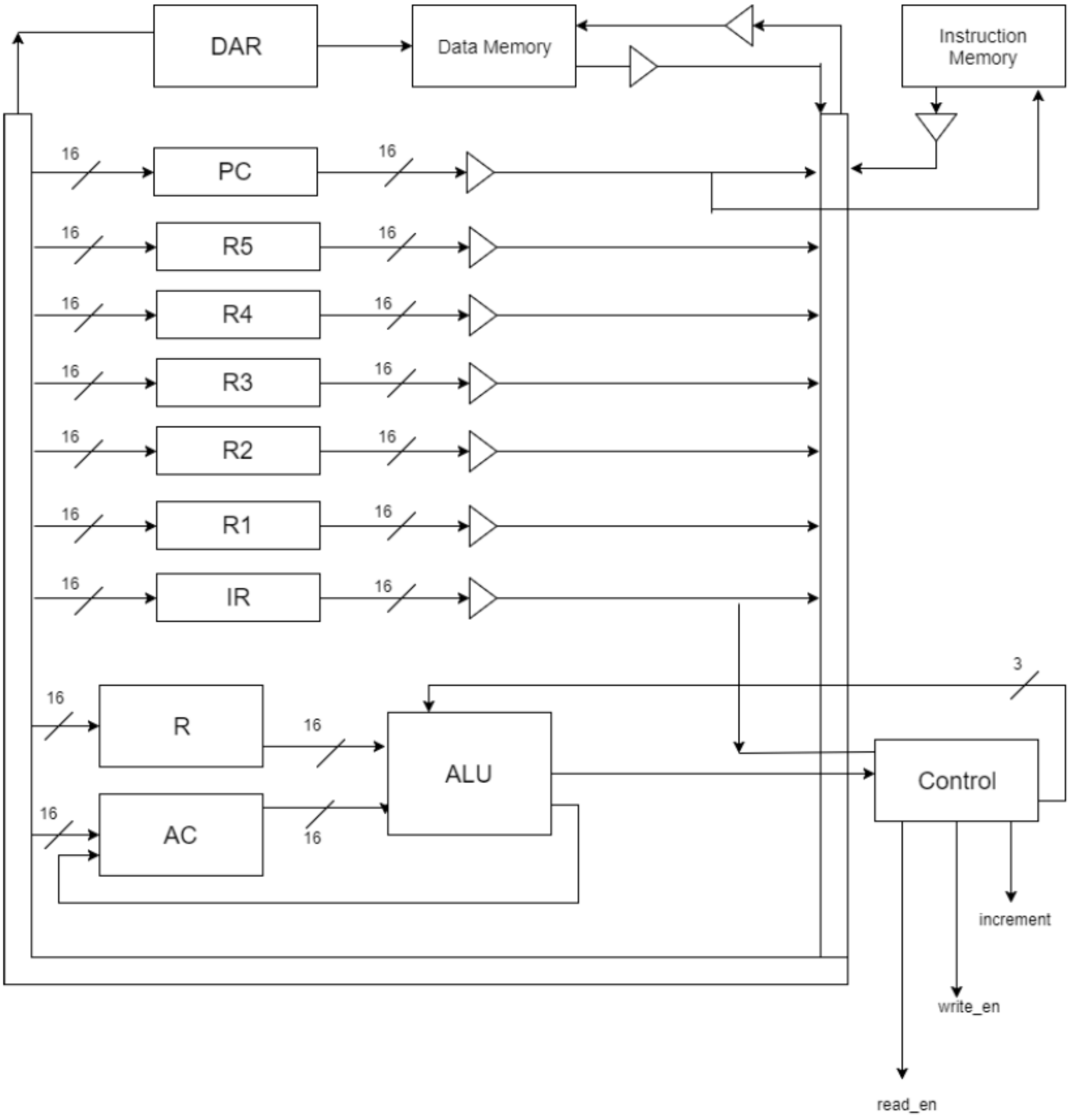
02. DESIGNING A FPGA BASED CUSTOM PROCESSOR

2.1 FPGA – AN OVERVIEW

Field Programmable Logic Array, which is also known as FPGA is a very powerful device used to manipulate digital electronic circuits. It has millions of logic ICs arranged in grid pattern and the user can decide the connections between these ICs according to the requirements. The FPGA board used for this assignment has a chip manufactured by Altera with other peripheral devices. To design the connections between ICs of the FPGA, instructions need to be given from a computer.

2.2 THE ISA – INSTRUCTION SET ARCHITECTURE

2.2.1 THE DATA PATH



2.2.2 INSTRUCTION SET

We were required to custom define an instruction set for this processor, specified towards the task of down sampling and filtering an image. A total of 31 instructions can be seen used in the below table, which was used in our processor design.

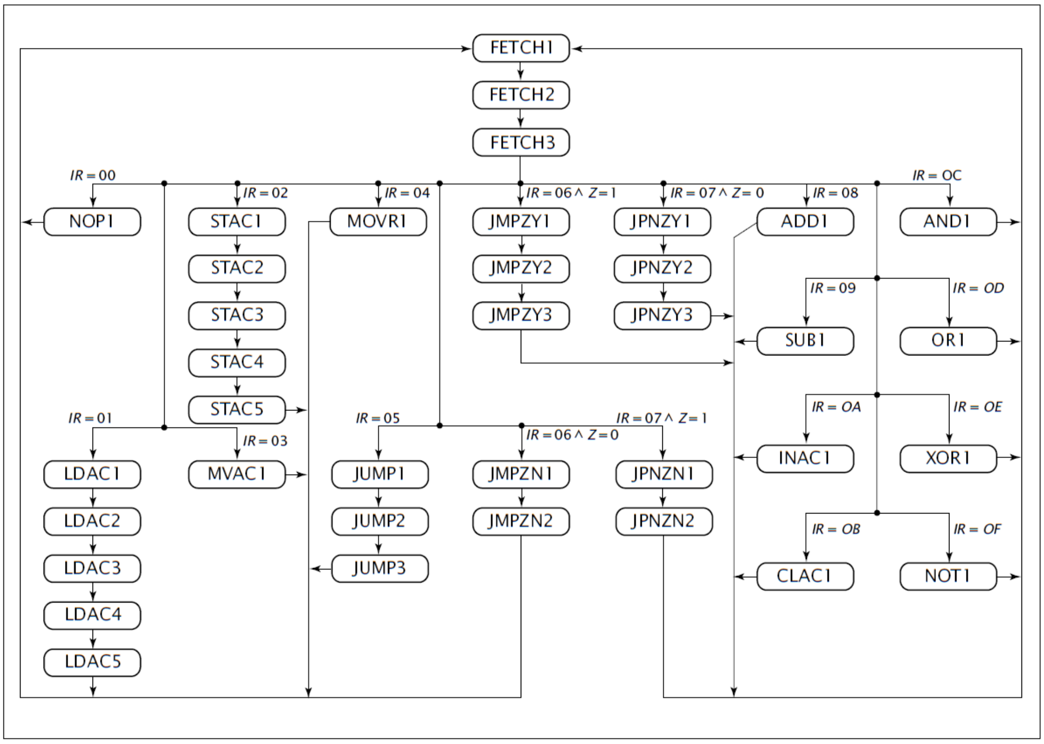
Instruction	Opcode	State	Address	Operation
START		START1		PC <-- 0
		START2		IR <-- 0
FETCH		fetch1	0	AR <-- PC
		fetch2	1	DR <-- M, PC <-- PC+1
		fetch3	2	IR <-- DR, AR <-- PC
NOP	0	nop1	3	NO OPERATION
ADD	1	add1	4	AC <-- AC+R
SUB	2	sub1	5	AC <-- AC-R
LDAC	3	ldac1	6	AR <-- AC
		ldac2	7	READ
		ldac3	8	DR <-- M[τ]
		ldac4	9	AC <-- DR
MOVACR	4	moavacr	10	R <-- AC
MOVACR1	5	movacr1	11	R1 <-- AC
MOVACR2	6	movacr2	12	R2 <-- AC
MOVACR3	7	movacr3	13	R3 <-- AC
MOVACR4	8	movacr4	14	R4 <-- AC
MOVACR5	9	movacr5	15	R5 <-- AC
LDIAC	10	ldiac1	16	READ
		ldiac2	17	AC <-- IM[τ]
		ldiac3	18	PC <-- PC+1
STAC	11	stac1	19	READ AC to BUS
		stac2	20	AR <-- AC
		stac3	21	M <-- AC
MOVRAC	12	movrac	22	AC <-- R
MOVR1AC	13	movr1ac	23	AC <-- R1
MOVR2AC	14	movr2ac	24	AC <-- R2
MOVR3AC	15	movr3ac	25	AC <-- R3
MOVR4AC	16	movr4ac	26	AC <-- R4
MOVR5AC	17	movr5ac	27	AC <-- R5
JUMP	18	jump1	28	READ
		jump2	29	AC <-- IM[τ]
		jump3	30	PC <-- AC
JMPZ	19	jmpzy1	31	READ
		jmpzy2	32	AC <--IM[τ]
		jmpzy3	33	PC <-- AC
		jmpzn1	34	PC <-- PC+1
		jpnzy1	35	READ

JPNZ	20	jpnzy2	36	AC <--IM[τ]
		jpnzy3	37	PC <-- AC
		jpnzn1	38	PC <-- PC+1
MOVAC	21	mvacar1	39	PC <-- AC
		mvacar2	40	AR <-- PC
LSHIFT	22	lshft1	41	AC <--AC<<R
RSHIFT	23	rshft1	42	AC <--AC>>R
CLAC	24	clac1	43	AC<--0 , Z=1
INCPC	25	inpc1	44	PC <-- PC+1
INCAC	26	inac1	45	AC <-- AC+1
INCR1	27	inr1	46	R1<-- R1+1
INCR2	28	inr2	47	R2 <-- R2+1
INCR3	29	inr3	48	R3 <-- R3+1
IDLE	30	idle1	50	
ENDOP	31	endop	51	END OPERATION
			52	
			53	

2.2.3 MICROINSTRUCTION SEQUENCE

The operation of this processor is based on a state machine, which executes one task at any given time. In order to achieve its objective, the machine follows a three component cycle which is repeatedly executed whereas the state in which it is in currently is a direct result of the previous state.

Given below is an example of what such a processor state machine looks like.



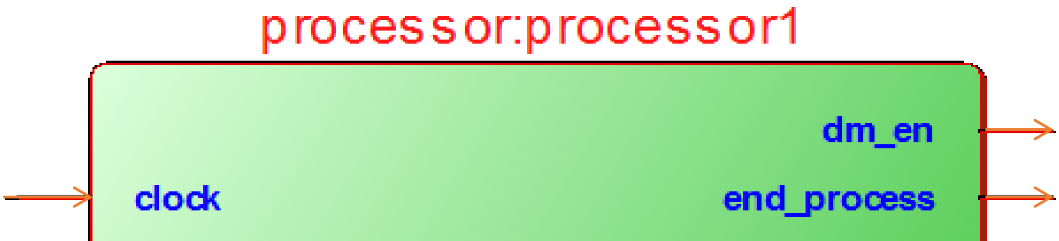
2.3 MODULES AND COMPONENTS

2.3.1 THE PROCESSOR

Instances of other modules used for the processing part and the required controlling instructions are included in this module. However, instances of communication and memory related modules are not included in this module. Inputs and outputs of the *processor module* are stated below.

Inputs	**Outputs**
clock – for synchronization	***dm_en*** – enable pin of data memory
dm_out[7:0] – output of the data memory	***im_en*** – enable pin of instruction memory
im_out[15:0] – output of the instruction memory	***end_process*** – pin that notify the end of the process
status[1:0] – status of transmission	***pc_out[15:0]*** – output of the program counter
	dar_out[15:0] – outputs the DAR value
	bus_out[15:0] – bus which carries the processed data

The diagram of the processor module is as follows:



Modules used for the processing part, and the registers used under this *processor module* are as follows.

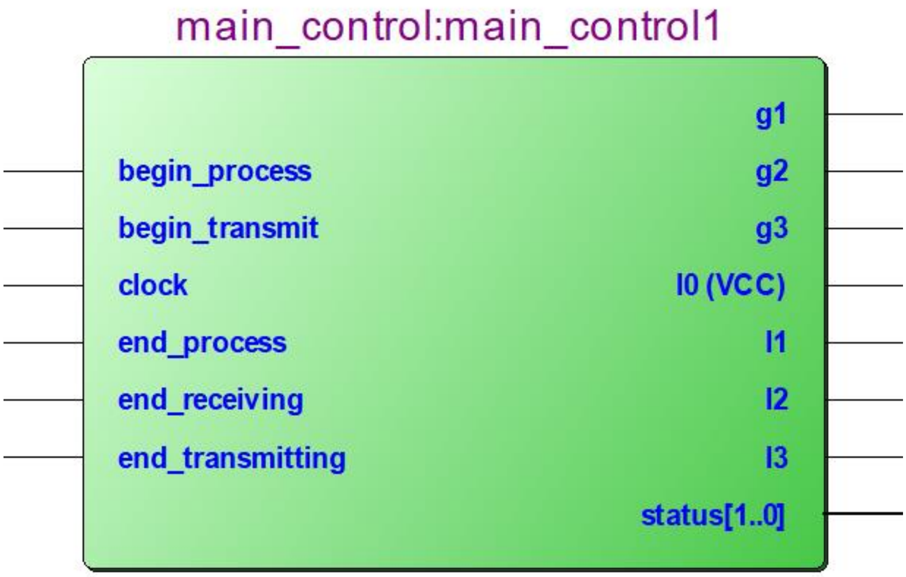
- **ALU** – used for Arithmetic and Logic operations of the processor.
- **AC** – a special purpose register used to store the output of an ALU operation. Further, one input required for an ALU operation is directly taken from AC.
- **Program Counter (PC)** – a special purpose register which keeps track of the next instruction to be fetched.
- **Instruction Register (IR)** – a special purpose register used to store the instructions loaded from *Instruction Memory*.
- **Data Address Register (DAR)** – a special purpose register used to store the address of a memory location in the *Data Memory*. When reading data from or writing data to the data memory, the address stored in DAR is used.
- **General Purpose Registers (R, R1, R2, R3, R4, R5)** – among these registers, R, R1 and R2 are *registers without increment*. Others (R3, R4 and R5) are taken as *registers with increment*.
- **Control Unit** – control signals which control the processing of the processor are given by this module. It is implemented as a state machine.

2.3.2 STATE MACHINE FOR THE DOWNSAMPLING PROCESS

It is used change the status of the process cycle based on the switched connected to it. It has several inputs. Clock, End receiving, End process, End transmit, Begin Process, End Process are in puts.

Last bit of transmission gives the signal that receiving is over. There is a manual switch in processing cycle. It also notifies the last bits of transmission. Main control module catches the combination of these switches and changes in the status of registers.

Structure of the Main Controller is as follows.

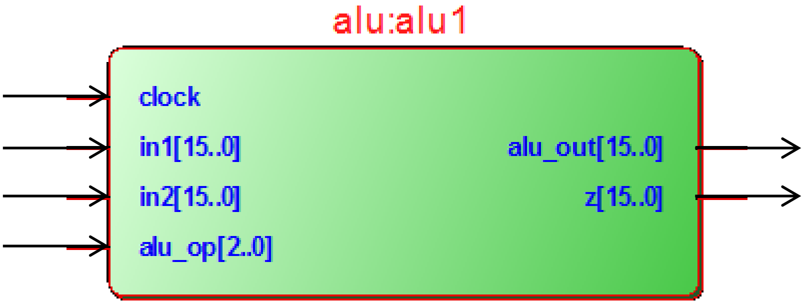


2.3.3 ALU - ARITHMETIC AND LOGIC UNIT

All the *Arithmetic and Logic operations* of the *processor* are handled by ALU. Two inputs are taken from the *register R* and *AC*, and the output is stored back to *AC* as mentioned previously. The *alu_op[1:0]* flag indicates which operation needs to be done in the ALU. Those operations are as follows:

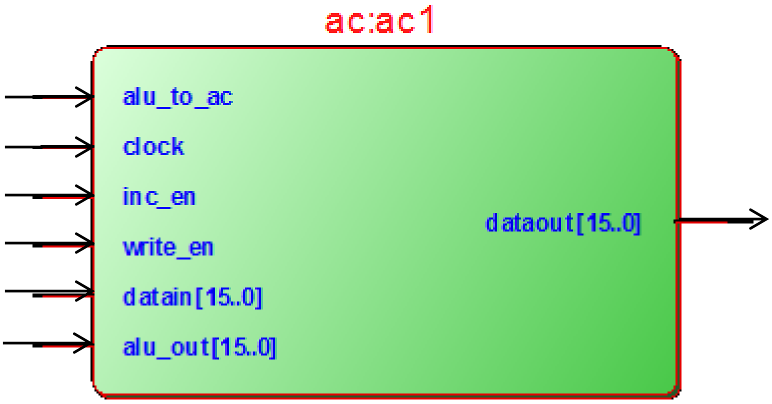
***alu_op[1:0]* Flag**	**Operation**	**Description**
00	ADD	in1 + in2
01	SUB	in1 - in2
10	RIGHT SHIFT	in1 << in2
11	LEFT SHIFT	in1 >> in2

After an ALU operation, the *Z* flag is set to zero ($Z = 0$ when *AC*$\neq$ 0) or one ($Z = 1$ when *AC=0*).



2.3.4 AC - ACCUMULATOR

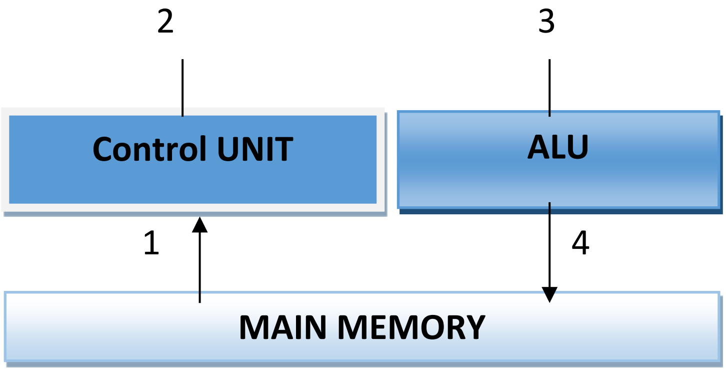
AC is a *special purpose register* which is connected directly to ALU. For any ALU operation, one input is directly taken from the AC; furthermore, the result is also stored to AC after the ALU operation. Therefore, it gets data from both bus and ALU. When *alu_to_ac* is set to 1, 16-bit output of the ALU is loaded to AC; moreover, when *write_en* is set to 1, 16-bit data in the bus is loaded to AC.



2.3.5 CU - CONTROL UNIT

Control Unit (CU) directs the operation of the processor. That means, it guides how to respond to the instructions that flow to memory, arithmetic logic unit, input and output devices.

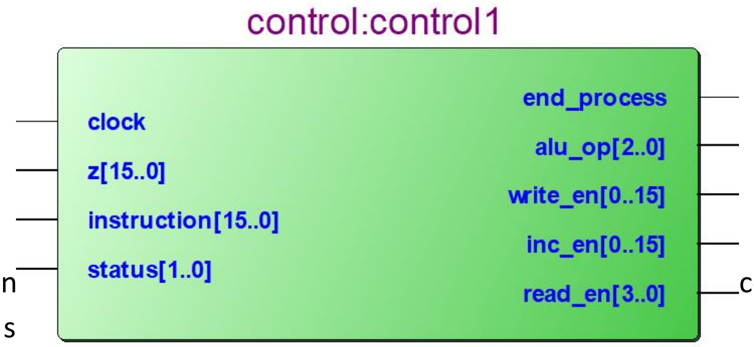
Basic diagram with a flow of instruction can be shown as follows.



1. Fetch instruction from memory
2. Decode instruction into Commands
3. Execute commands
4. Store results in memory

There are different functions of control unit. It controls the data flow inside the processor. Control unit receives external instructions and converts it into sequence of signals. Then CU applies register–transfer level operations for those instructions. It decodes individual instruction in to several sequential steps as fetching addresses, data from registers, memory managing execution, storing data back to memory or registers. Control unit is able to schedule micro instructions between the selected execution units in ALU or other functions. So that control unit can be classified in three ways as instruction unit, scheduling unit and retirement unit. Results that are coming out from instruction pipeline are mainly handled by the retirement unit.

Instruction memory, status, z flag are the inputs of the module and alu_op, write_en, read_en and end process are the outputs. ALU op is 3 bits wide while write_en and read_en are 16 bits wide.



ALU output is zero or on ceiving mode,
transmitting mode are s

ALU Operation

ALU operations are controlled by this unit which is 3 bits wide. It allows for 8 ALU operations. As an instance 000 stands for In1 +In2. 010 stands for In1-In2.

Write_En

This is 16 bits wide signal which is connected to the registers. When write_en =1 , data in the bus is written to the registers. Different registers are connected to 16 bits of wire separately. So that registers can be written parallel with the same value in bus (setting to write _en =1).

Registers and the relevant write_en values are given in the following table.

Bit 16 N/A	Bit 15 N/A	Bit 14 ALU TO AC	Bit 13 INS MEM	Bit 12 DATA MEM	Bit 11 R1	Bit 10 R2	Bit 09 R3	Bit 08 R4	Bit 07 R5	Bit 06 R	Bit 05 AC	Bit 04 IR	Bit 03 N/A	Bit 02 DAR	Bit 01 PC

Read_en

Outputs of all registers are connected to the bus with 16 bits wide signal. Read_en determines the which output should be fetched into the bus. It eliminates the error of writing two registers at the same time.

Table with relevant register values in read_en is shown herewith.

READ_EN VALUE	REGISTER
0000	Unused

0001	PC
0010	DAR
0011	DR
0100	IR
0101	AC
0110	R
0111	R1
1000	R2
1001	R3
1010	R4
1011	R5
1100	DM
1101	IM

Increment_en

This is 16 bits long signal connecting to different registers. When increment_en =1 , values in the registers gets incremented by one.

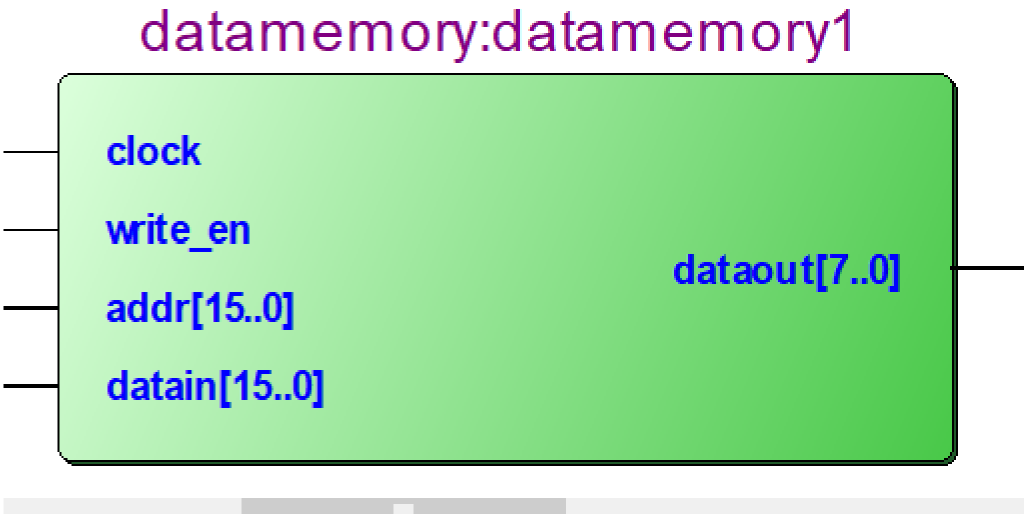
Table with relevant register values in increment_en is shown herewith.

Bit 16 N/A	Bit 15 N/A	Bit 14 ALU TO AC	Bit 13 INS MEM	Bit 12 DATA MEM	Bit 11 R1	Bit 10 R2	Bit 09 R3	Bit 08 R4	Bit 07 R5	Bit 06 R	Bit 05 AC	Bit 04 IR	Bit 03 N/A	Bit 02 DAR	Bit 01 PC

2.3.6 DATA MEMORY (DRAM)

It is the main memory to store data. It stores the data before it transmitted to the computer. UART converts serial data to bytes and Data is sent to DRAM for storing. Memory is 8 bits long and 65536 locations are inside that. Each pixel value can be range between 0-255.

Structure of the data memory is as follows.



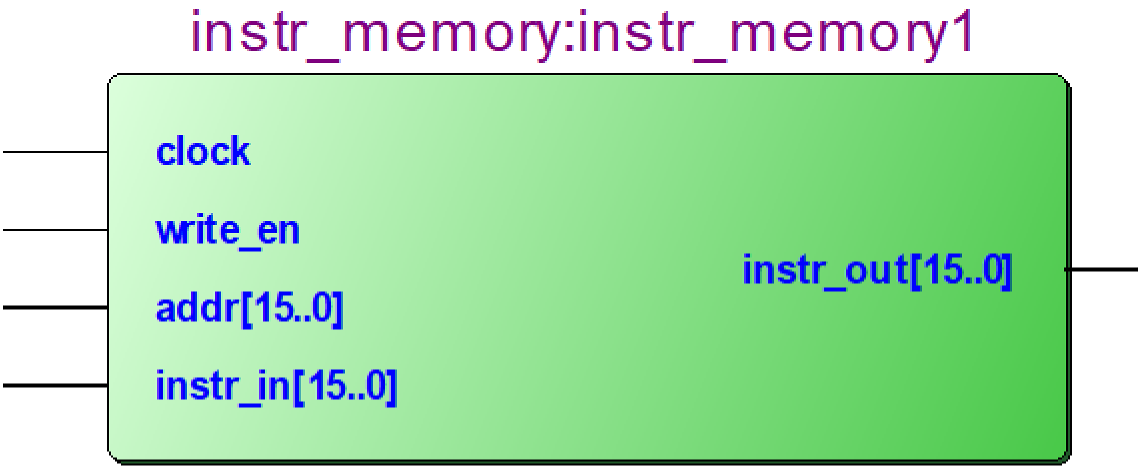
Internal data storage of DRAM is shown as follows.

16 bits address	8 bits address							
0x0000								
0x0001								
0xFFFF								

one pixel

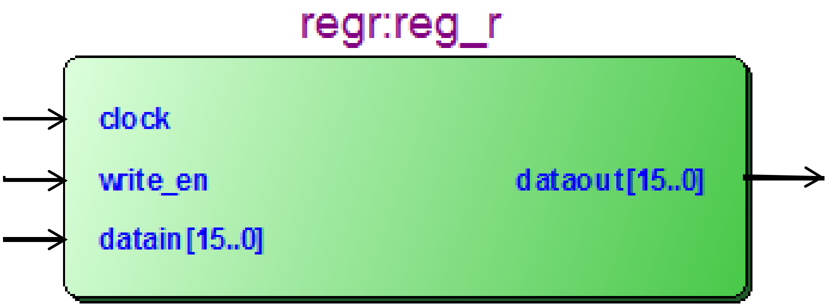
2.3.7 INSTRUCTION MEMORY (IRAM)

Data is used only for read in ROM. It is executed one after the other where instructions are located in the instruction memory location.



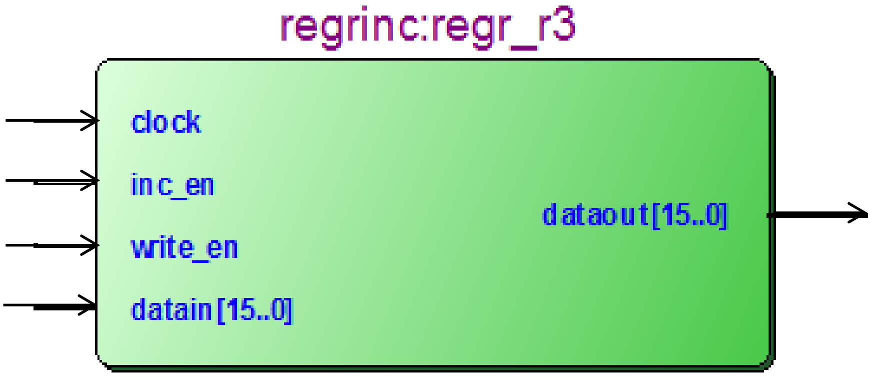
2.3.8 REGISTERS WITHOUT INCREMENT

When data needed for relevant processing tasks needs to be stored temporally, *registers* can be used. In these types of registers, incrementing the register value needs to be done through the ALU. They cannot be directly incremented. Data stored in the register can be accessed through the *dataout[15:0]* pin once the *read_en[3:0]* flag of the bus is enabled. Similarly, when the *write_en* pin is enabled (set to 1) during the positive edge of the clock cycle, data in the bus is read and stored to the register.



2.3.9 REGISTERS WITH INCREMENT

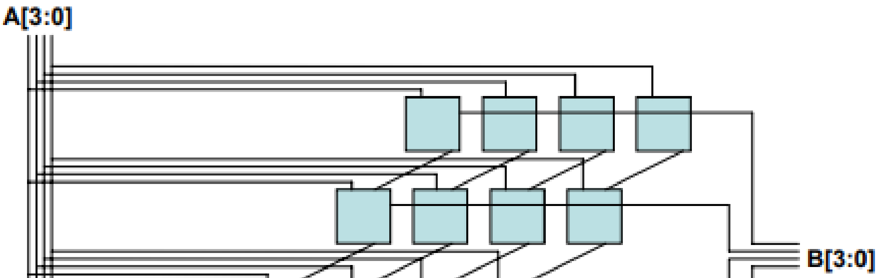
These types of registers can be incremented by 1, after setting the *inc_en* pin to 1 during the positive edge of the clock cycle. There is no need of an ALU operation for incrementing the register value. These registers are used to keep track of loop iterations, without making the process much slower.



2.3.10 BUS

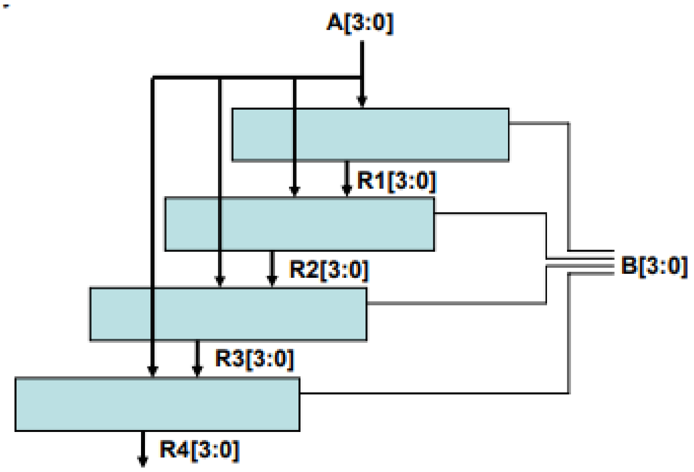
Bus consists of set of wires or connector which transport data. It is allowed to bundle several binary signals and addressed them with single name.

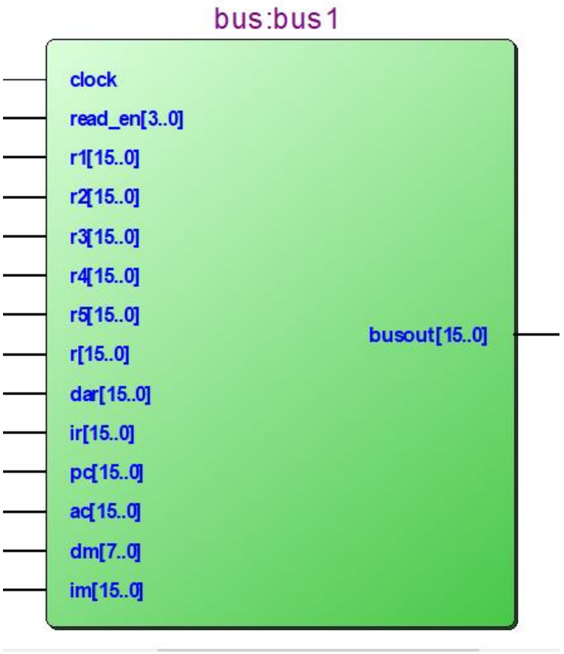
Below is an example of a 4 bit Bus. We used a 16-bit bus for the above processor.



16 blocks will be presented to handle 4 bit data bus.

Rin of one raw is Rout of the previous row. Following diagram shows the relationship.

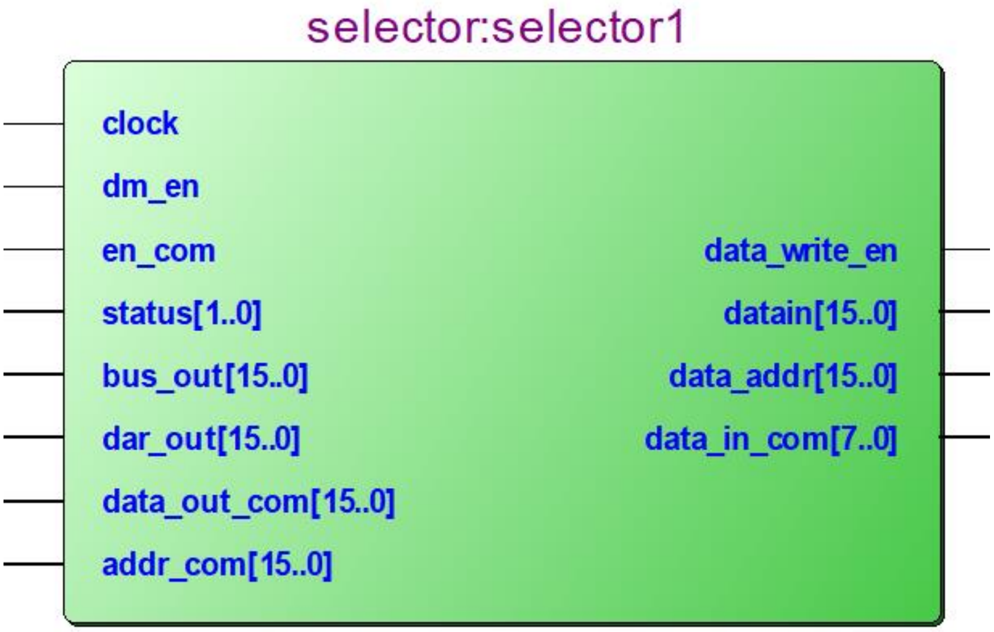




2.3.11 SELECTOR

This module is used to select the output of the processor to the UART and the memory unit depending on the input signals to the processor. This gives commands either to the UART to start receiving data and store them in the DRAM or to the processor to process data according to the requirements or to the UART to start transmitting data stored in the DRAM. Moreover, this module is used to select the DRAM and IRAM addresses according to the requirement and to select accessing modes of RAMs (read / write).

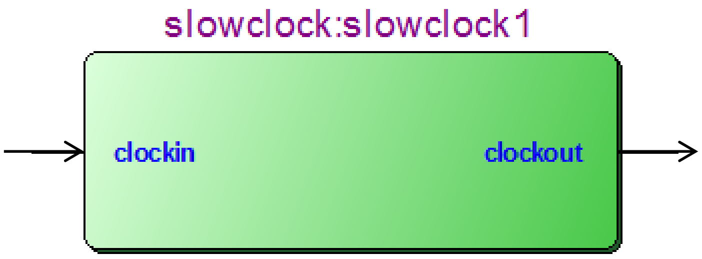
It is used to select from the communication module or the processor. Structure of the selector is as follows.



Selector checks the status from the main controller to selects whether value from processor or receiver should be stored in the memory.

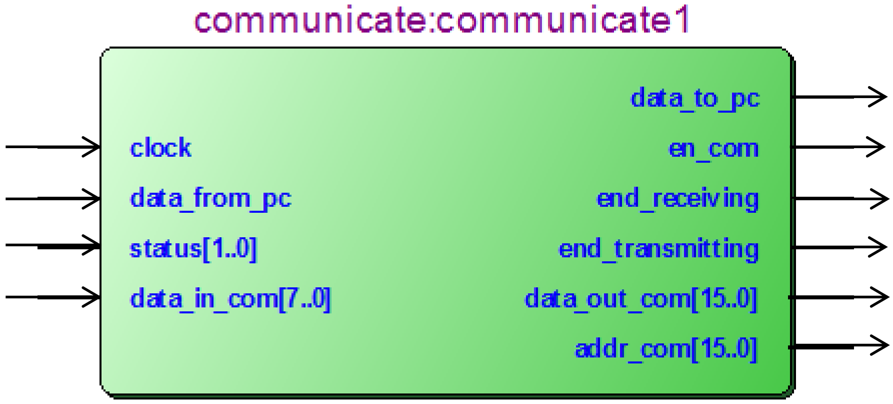
2.3.12 CLOCK DIVIDER

Since the operations of the communication modules tend to become inaccurate at faster clocks, a clock divider which provides the half of the input frequency needed to be used. In this module, the input frequency 50Hz, which is the clock signal produced by the oscillator in *Altera DE2-115* board, is halved and provided at the output.



2.3.13 COMMUNICATION COMPONENTS

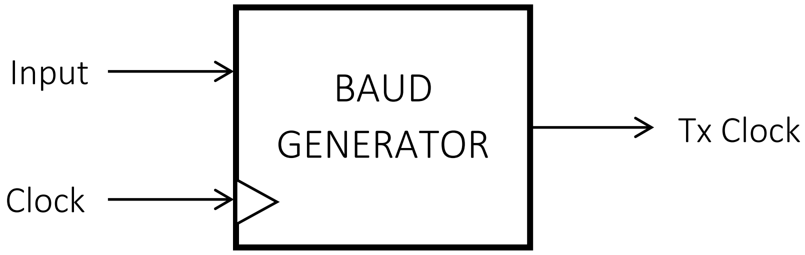
Transmission of data between the computer and the FPGA board is handled by this module. It works under Receiving, Processing, Transmitting and All Done states. The diagram of the *communication module* is as follows:



Under this module, following sub modules are used.

- **Baud Generator:**

When compared to the clock used in the FPGA board (25MHz), the speed of the serial communication (9600 bits/s) is different. This may lead to inaccurate data communication since the original clock rate is not an integer multiple of transmission rate in base 2. Therefore, to synchronize the transmission rate and the original clock rate, a *Baud Generator* is used.

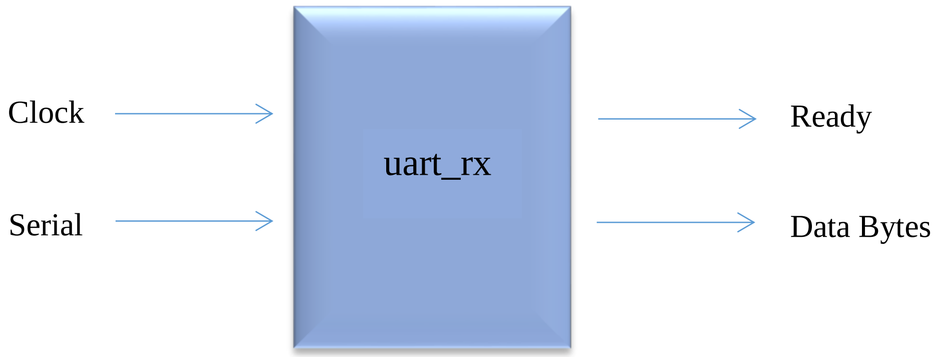


- **Receiver:**

This module is used to receive serial data from a computer and output the received data bytes (8 bits at a time). Data is received through the serial pin with a clock. Register which contains the received data and the ready signal is given as outputs. Several parameters are used to maintain the states of the data receiving process. Following registers are used to store data during the process.

reg [7:0] r_Clock_Count- Number of clocks spent after receiving new data bit

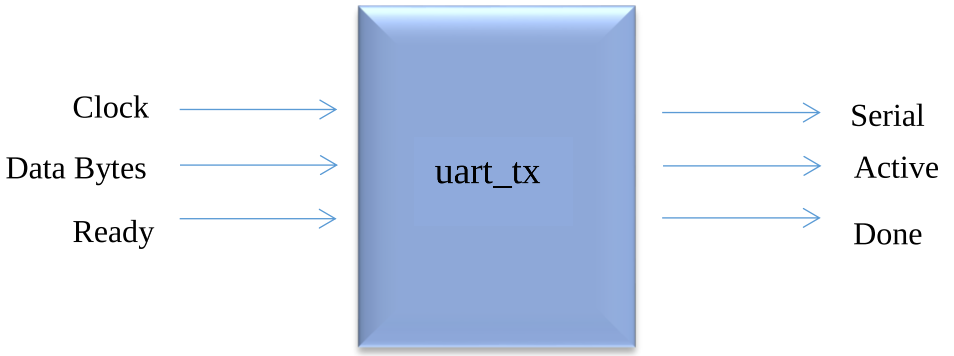
reg [2:0]	r_Bit_Index	- Index of the current bit of the data byte
reg [7:0]	r_Rx_Byte	- Data byte
reg	r_Rx_ready	- Data is ready to output
reg [2:0]	r_state	- Current state of the data receiving process



- **Transmitter:**

  This module is used to transmit serial data from the board to the computer and output the current state of the data transmitting process. Data byte need to be transmitted is taken as the input with a clock and the ready signal. Serial data is given as the output with the done signal. Several parameters are used to maintain the states of the data transmitting process. Following registers are used to store data during the transmitting process.

reg [2:0]	r_state	- Current state of the data transmitting process
reg [7:0]	r_Clock_Count	- Number of clocks spent after transmitting the last bit
reg [2:0]	r_Bit_Index	- Index of the last sent bit
reg [7:0]	r_Tx_Data	- Sending data byte
reg	r_Tx_Done	- Data transmitting state (finished/not finished)
reg	r_Tx_Active	- Current data transmitting state



03. ALGORITHM AND DESIGN CONSIDERATIONS

3.1 DATA PROCESSING TECHNIQUES

How do we use pixel data in the 256x256 image and how we get it into DRAM?

The initial step for the given task is storing the data included in the pixels of the 256x256 image in to a text file. This can be done using MATLAB software. All the pixel values of the image is transformed to an array and written in a text file. The code snippet used for this task is given in the Appendix 'matching number' (01). Data arrangement in the image and the text file is given below.

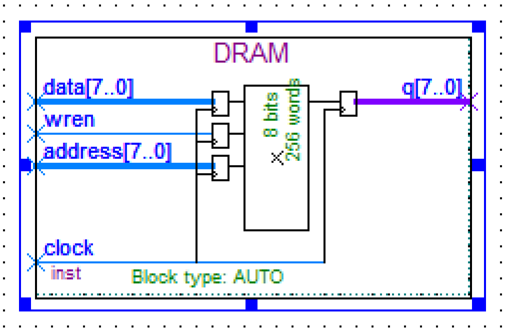
a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y

Image

a
b
c
d
e
f
g
…..
x
y

Text file

Then this data in the text file need to be sent to the serial port of the PC which is connected to the RS232 connector coming from the development board and then transmit to the UART receiver. The code snippet used to send the numerical data in the text to the serial port is given in Appendix 'matching number' (02). After that using the 'UART_RX' module, this data can be stored in the data memory (DRAM). In order to store this in the processor memory, a memory module needs to be initialized. This is done using the 'IP Catalog' of the 'Quartus' Programme. The schematic diagram of the DRAM is given below.



How do we get the final processed data for the output image?

The processed data in the DRAM is transmitted from the development board using the 'UART_TX' transmitter to the serial port. Then using MATLAB, this data can be written to a text file as an array. After that, this array can be transformed in to a matrix corresponding to the down sampled matrix and obtain the final output image. The code snippet used for this task is given in the Appendix 'matching number' (03)

3.2 FILTERING (LOW PASS) ALGORITHM

In order to reduce the effects of aliasing caused by the high frequency components in the image, the image needs to be low pass filtered, prior to down sampling it. This part was implemented in MATLAB by using a Gaussian low pass filter.

The following 256×256 image is used to describe the necessity of low pass filtering the image before down sampling it.

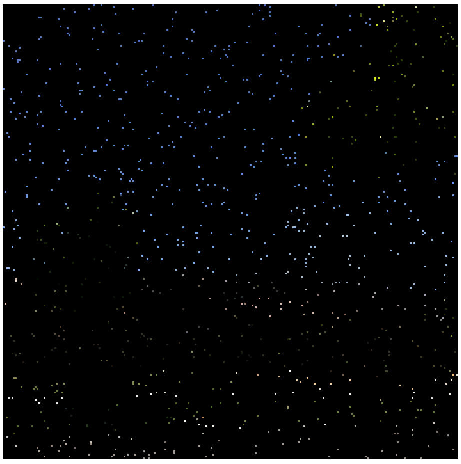


Figure 1: Original RGB Image

After converting the original RGB image in *Figure 1* into a gray scale image by using the "*rgb2gray(image)*" function, white dots which can be considered as high frequency components, are clearly visible. In order to filter this image, a 3×3 kernel that has a half width of size 1 is used. Moreover, the used sigma value that determines the extent of the smoothing is 2/3. The function which was used to create the Gaussian filter Kernel is as follows:

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

By using the above function, following filter kernel was obtained.

0.0377	0.1163	0.0377
0.1163	0.3581	0.1163
0.0377	0.1163	0.0377

Using the above filter kernel, the image was filtered. After down sampling it, following 64×64 image in *Figure 2* was obtained. The image in *Figure 3* is the image which was down sampled without filtering it.

Figure 2: Down Sampled Image after Filtering

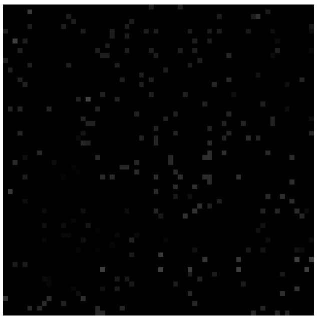


Figure 3: Down Sampled Image without Filtering

By looking at those pictures, it can be clearly seen that the high frequency components in the original image have caused aliasing effects to the image which was down sampled without filtering (*Figure 3*).

3.3 DOWN SAMPLING ALGORITHM

An 256×256 image is down sampled to an 64×64 image. The average intensity value which was computed using 4 nearby pixel values is assigned to the each pixel value in the

down sampled image. The image which was used in this project, Gaussian low pass filtered image and its down sampled image are provided below.

Figure 4: Down Sampled Image

Figure 4: Original Gray Scale Image

Figure 5: Low Pass Filtered Image

3.4 ASS HMS

Low pas filtering

The filtering algorithm is based on the 3x3 Gaussian kernel given in Figure 03. The middle pixel location is considered as (0,0) and the standard deviation s 0.6. Used Gaussian function is;

$$Gaussian(x,y) = e^{\frac{(x^2+y^2)}{2\sigma^2}}$$

0.0622	0.2489	0.0622
0.2489	1	0.2489
0.0622	0.2489	0.0622

Figure 5: Gaussian Kernel

But processing decimal values are complex when designing the processor. Therefore, the kernel is first normalized so that the sum of the kernel is a power of 2 and then approximated to integer values.

0.99	3.99	0.99
3.99	16	3.99
0.99	3.99	0.99

Figure 6: Normalized kernel

If the value 3.99 is approximated to 4, the sum of the kernel is 36 which cannot be written as a power of 2. Therefore, it is approximated to 3 which give the sum 32. But the effect of error needs to be considered during the evaluation.

1	3	1
3	16	3
1	3	1

Figure 7: Approximated Kernel

Then the filtering an image can be done using the approximated kernel by moving the kernel along the image. Normalized, weighted sum is taken as the filtered value of a pixel.

Down sampling algorithm

A simple down sampling algorithm is used for this processor to avoid errors. According to this algorithm, if the down sampling factor is k, 1 pixel is selected from adjacent k pixels of the filtered image while maintaining the order. Following is where the factor is 4.

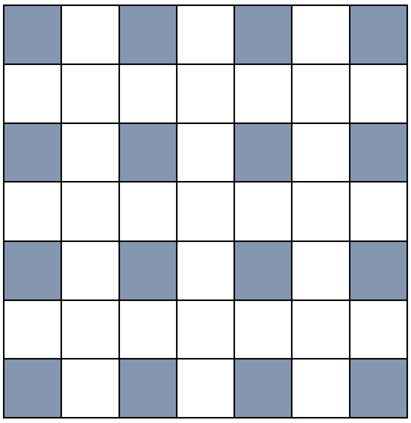


Figure 6: Filtered image

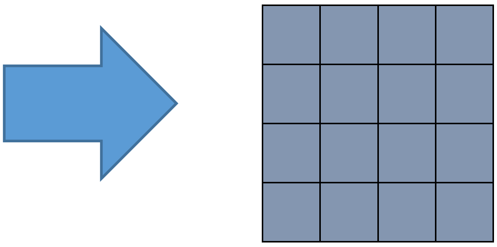


Figure 7: Down sampled image

Implementation of these algorithms in assembly code is described in the following sections of the document.

ASSEMBLY CODE

THE ASSEMBLY CODE IS INITIATED FOR THE FILTERING PROCESS.

```
[0] = LOADIM;
[1] = 16'D257; // STARTING PIXEL 8'D6;
[2] = MOVACR1;
[3] = NOP;
[4] = NOP;
[5] = NOP;
[6] = NOP;
```

```
[7] = NOP;
[8] = NOP;
[9] = MOVR1AC;
[10] = LDAC;
[11] = MOVACR;
[12] = LOADIM;
[13] = 8'D4;
[14] = LSHIFT;
[15] = MOVACR4;
[16] = LOADIM;
[17] = 8'D1;
[18] = MOVACR;
[19] = MOVR1AC;
[20] = ADD;
[21] = LDAC;
[22] = MOVACR5; //R5 HAS RIGHT
[23] = MOVR1AC;
[24] = SUB ;
[25] = LDAC;
[26] = MOVACR;
[27] = MOVR5AC;
[28] = ADD;
[29] = MOVACR5; //R5 HAS RIGHT+LEFT
[30] = LOADIM;
[31] = 16'D256; // ADDED TO GET BOTTOM PIXEL 8'D5;
[32] = MOVACR;
[33] = MOVR1AC;
[34] = ADD;
[35] = LDAC;
[36] = MOVACR;
[37] = MOVR5AC;
[38] = ADD;
[39] = MOVACR5; // R5 HAS RIGHT+LEFT+BOTTOM
[40] = LOADIM;
[41] = 16'D256; // ADDED TO GET TOP PIXEL 8'D5;
[42] = MOVACR;
[43] = MOVR1AC;
[44] = SUB;
[45] = LDAC;
[46] = MOVACR;
[47] = MOVR5AC;
[48] = ADD;
[49] = MOVACR5; // R5 HAS RIGHT+LEFT+BOTTOM +TOP
[50] = MOVACR;
[51] = LOADIM;
[52] = 8'D1;
[53] = LSHIFT;
[54] = MOVACR;
```

```
[55] = MOVR5AC;
[56] = ADD;
[57] = MOVACR;
[58] = MOVR4AC;
[59] = ADD;
[60] = MOVACR4; //R4 HAS 3*(MIDPIXELS)
[61] = LOADIM;
[62] = 16'D257; // TO GET BOTTOM RIGHT AND TOP LEFT 8'D6;
[63] = MOVACR;
[64] = MOVR1AC;
[65] = ADD;
[66] = LDAC;
[67] = MOVACR5;
[68] = MOVR1AC;
[69] = SUB;
[70] = LDAC;
[71] = MOVACR;
[72] = MOVR5AC;
[73] = ADD;
[74] = MOVACR5;
[75] = LOADIM;
[76] = 8'D255; //TO GET BOTTOM LEFT 8'D4;
[77] = MOVACR;
[78] = MOVR1AC;
[79] = ADD;
[80] = LDAC;
[81] = MOVACR;
[82] = MOVR5AC;
[83] = ADD;
[84] = MOVACR5;
[85] = LOADIM;
[86] = 8'D255; //TO GET TOP RIGHT 8'D4;
[87] = MOVACR;
[88] = MOVR1AC;
[89] = SUB;
[90] = LDAC;
[91] = MOVACR;
[92] = MOVR5AC;
[93] = ADD;
[94] = MOVACR5;
[95] = MOVACR;
[96] = MOVR4AC;
[97] = ADD;
[98] = MOVACR;
[99] = LOADIM;
[100] = 8'D5; //TO DIVIDE BY 2^5 = 32
[101] = RSHIFT;
[102] = MOVACR4;
```

```
[103] = LOADIM;
[104] = 16'D257; // TO FIND STORING LOCATION
[105] = MOVACR;
[106] = MOVR1AC;
[107] = SUB;
[108] = MOVACDAR;
[109] = MOVR4AC;
[110] = STAC;
[111] = LOADIM;
[112] = 16'D65278; // 256*254+255-1 TO CHECK END IF ALL PIXEL
[113] = MOVACR;
[114] = MOVR1AC;
[115] = SUB;
[116] = JUMPZ;
[117] = 8'D138; // FINISH CONVOLUTION GO TO DOWNSAMPLE
[118] = LOADIM;
[119] = 8'D253; // TO CHECK END OF ROW 8'D2;
[120] = MOVACR;
[121] = MOVR2AC;
[122] = SUB ;
[123] = JUMPZ;
[124] = 8'D129; // FOR INCREMENTING IF END_OF ROW
[125] = INCR2; // IF NOT END OF ROW
[126] = INCR1;
[127] = JUMP;
[128] = 8'D9; // JUMP TO START
[129] = INCR1;
[130] = INCR1;
[131] = INCR1;
[132] = LOADIM;
[133] = 8'D0;
[134] = MOVACR2;
[135] = JUMP;
[136] = 8'D9;
[137] = ENDOP;
```

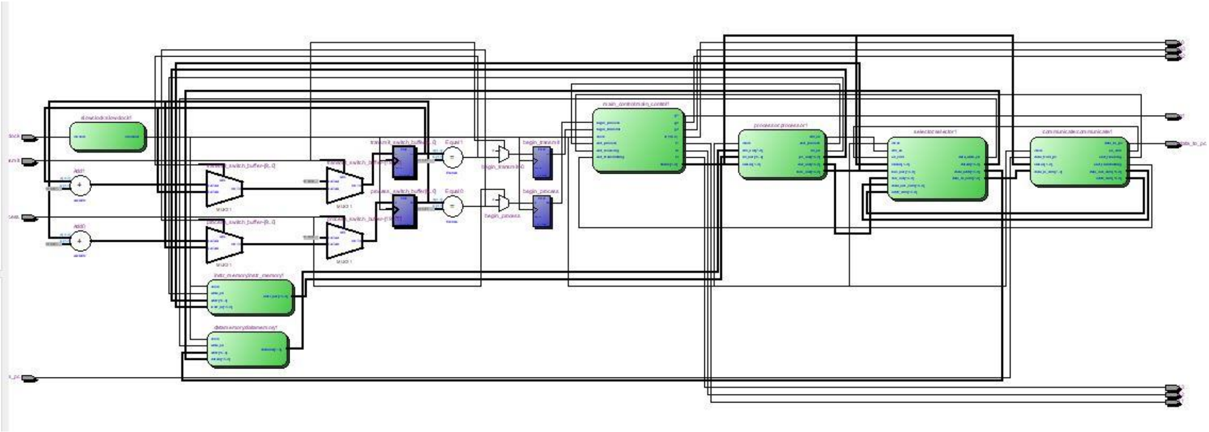
AT THIS POINT, FILTERING ALGORITHM PART IS DONE AND STORED CORRESPONDING VALUE IN THE DRAM. NEXT THING IS TO DOWN SAMPLE THE IMAGE.

```
[138] = LOADIM;
[139] = 8'D0;
[140] = MOVACR1; // SET LOADPIXEL = 0
[141] = MOVACR2; // SET ROWCOUNT = 0
[142] = MOVACR3; // SET SAVEPIXEL = 0
[143] = MOVR1AC; // BEGIN LOOP
[144] = LDAC;
[145] = MOVACR4;
[146] = MOVR3AC;
```

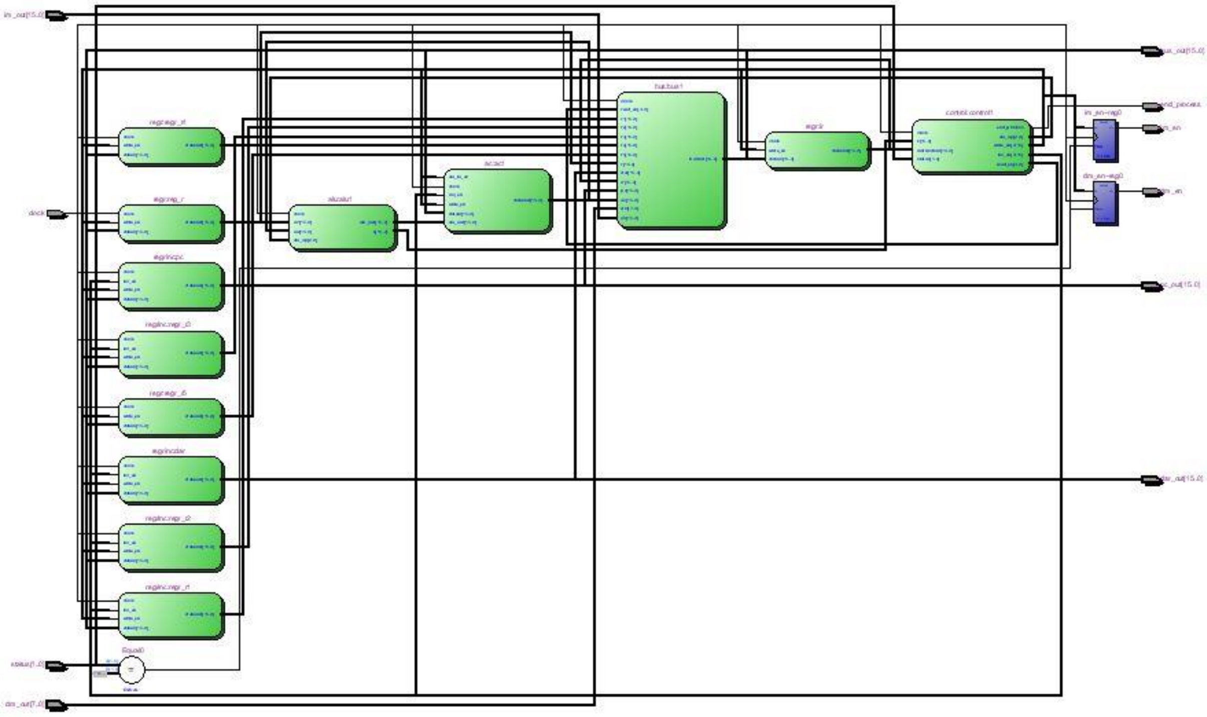
```
[147] = MOVACDAR;
[148] = MOVR4AC;
[149] = STAC;
[150] = MOVR1AC; //CHECK IF LAST PIXEL DONE
[151] = MOVACR;
[152] = LOADIM;
[153] = 16'D64764; // 256*252+253-1
[154] = SUB;
[155] = JUMPZ;
[156] = 8'D186; //ENDOP
[157] = MOVR2AC; // CHECK IF ROW DONE
[158] = MOVACR;
[159] = LOADIM;
[160] = 8'D252; // END OF LINE (ONE PIXEL BEFORE 254)
[161] = SUB;
[162] = JUMPZ;
[163] = 8'D171;// LOOP FOR LINE FINISH
[164] = INCR2;
[165] = INCR2;
[166] = INCR1;
[167] = INCR1;
[168] = INCR3;
[169] = JUMP;
[170] = 8'D143;// LOOP TO LINE BEGIN
[171] = LOADIM; // IF END OF ROW FOLLOW
[172] = 8'D0;
[173] = MOVACR2;
[174] = LOADIM;
[175] = 16'D260; // NEXT_GAP_LINE
[176] = MOVACR;
[177] = MOVR1AC;
[178] = ADD;
[179] = MOVACR1;
[180] = INCR3;
[181] = NOP;
[182] = NOP;
[183] = NOP;
[184] = JUMP;
[185] = 8'D143; // LOOP TO BEGINNING
[186] = ENDOP;
```

04. VERILOG IMPLEMENTATION

4.1 RTL DESIGN SIMULATION



Complete Processor with CPU and Memmory



Central Processing Unit including CU ALU and Registers

4.2 IMPLEMENTATION ON QUARTUS II

Our processor was developed by using ALTERA FPGA board. We chose Verilog HDL for developing the code. According to that we used Quartus Software to compile the Verilog code. After creating all modules inside the processor, we compiled whole Verilog document in Quartus and defined inputs, outputs for either switches or LEDs of each module by changing the pin planner option. Then we again compiled the Verilog document to see the how data paths were created between each module by using RTL viewer option.

Final task was to upload the compiled Verilog document. To do that there is a option in Quartus called Programmer. Then USB blaster was selected for Hardware Setup option and upload the .sof file into the Altera DE2-115 board.

Design Flow Summary

Flow Status	**Successful - Sun Jun 18 03:20:25 2017**
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	Processor
Top-level Entity Name	ultimate
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	1,040 / 114,480 (< 1 %)
Total registers	507
Total pins	12 / 529 (2 %)
Total virtual pins	0
Total memory bits	527,344 / 3,981,312 (13 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Resource Usage Summary

Resource	**Usage**
Total registers	507
-- I/O registers	0
-- Dedicated logic registers	507
Total memory bits	527344
Total fan-out	6766
Total combinational functions	944
Maximum fan-out node	slowclock:slowclock1|clockout
Maximum fan-out	586
Logic elements by mode	
-- normal mode	753
-- arithmetic mode	191
Logic element usage by number of LUT inputs	
-- 3 input functions	221
-- 4 input functions	484
-- <=2 input functions	239
I/O pins	12
Estimated Total logic elements	1,122
Embedded Multiplier 9-bit elements	0
Average fan-out	4.35

Resource Utilization by Entity

Entity	**Combinational ALUTs**	**Dedicated Logic Registers**
|ultimate	944 (29)	507 (22)
|communicate:communicate1|	197 (113)	166 (93)
|async_receiver:rx|	40 (20)	41 (21)
|async_transmitter:tx|	44 (24)	32 (12)
|datamemory:datamemory1|	75 (67)	52 (49)

|instr_memory:instr_memory1|	18 (18)	40 (40)
|main_control:main_control1|	12 (12)	7 (7)
|processor:processor1|	570 (2)	185 (1)
|ac:ac1|	33 (33)	16 (16)
|alu:alu1|	191 (191)	17 (17)
|bus:bus1|	136 (136)	0 (0)
|control:control1|	94 (94)	7 (7)
|regr:ir|	0 (0)	16 (16)
|regr:reg_r|	0 (0)	16 (16)
|regr:regr_r4|	0 (0)	16 (16)
|regr:regr_r5|	0 (0)	16 (16)
|regrinc:dar|	17 (17)	16 (16)
|regrinc:pc|	46 (46)	16 (16)
|regrinc:regr_r1|	17 (17)	16 (16)
|regrinc:regr_r2|	17 (17)	16 (16)
|regrinc:regr_r3|	17 (17)	16 (16)
|selector:selector1|	41 (41)	33 (33)
|slowclock:slowclock1|	2 (2)	2 (2)

05. PERFORMANCE EVALUATION

5.1 COMPARISON WITH MATLAB IMPLEMENTATIONS

In order to evaluate the performance of the processor we evaluated the difference between the outputs of the same algorithm on the same data (image) and compared them. For this task we simulated the algorithm that was input to the instruction memory in Matlab. The Matlab code for this is seen in appendix 1

The image output from the processor and the Matlab code were both in the correct dimensions. Therefore we subtracted the one image from the other and obtained Squared Sum of Differences (SSD) as the measure of accuracy. So lower the SSD the better the performance of the processor.

$$SSD= (Image_{Matlab} - Image_{Processor})^2$$

For the 5 images that we input to the algorithm with varying levels of intensity distribution and intensity values. The processor output image exactly matched with the Matlab image. Therefore the SSD values were 0 consistently.

We investigated the reasons that could have resulted in this perfect result and the following conclusion was reached

1. The reduced baud rate of 9600 bytes per second removes the possibility of having glitches in transmission and receiving data

2. The kernel that we used has a cumulative sum of 32 which implies that the divisions undertaken in the ALU are always a multiple of a power of 2. This reduces the probability of finite precision errors

3. Division by 32 results in a smaller error when rounded of compared to other lower alternatives such as 16 or 8

4. The division was performed after all the computations of the convolution operation were completed as a single step.

5. Implementing the division as divisions by powers of two where the dividend could be easily shifted left or right without using complex division routines in the ALU which could cause finite precision errors

5.2 VERIFICATION OF RESULTS USING ALTERNATE IMPLEMENTATIONS

The processor design that we used was specifically targeted at the purpose of down sampling an image therefore the instructions of the processor could be built in to the design itself. This enabled us to have the instructions to be written in a separate memory module that was initiated as a ROM where writing was not allowed. In a more flexible design of the processor. The memory would have to be implemented such that both the instructions and data can be written from outside

Since the process took less than 1 second to down sample the image the processing part took no significant time at all. However the communication components between the processor and the computer took significant time for implementation. The major reason for this is the unreliability of the communication media with increasing data speeds. A better mode of communication with higher accuracy could speed this process.

Even if the processor that we designed was fully capable of running any algorithm only constrained by the ISA, the speed of much complex algorithms would not be sufficient because of the inefficiencies in the data transfer inside the processor. Therefore the implementation of a cache memory might be advantageous in designing a processor for a more complicated processing task

06. REFERENCES

- https://en.wikipedia.org/wiki/Central_processing_unit
- https://en.wikipedia.org/wiki/Microprocessor
- https://en.wikipedia.org/wiki/Control_unit
- http://www.fpga4student.com/2017/08/verilog-code-for-clock-divider-on-fpga.html
- https://en.wikipedia.org/wiki/Processor_(computing)
- https://en.wikipedia.org/wiki/Processor#Computing

07. APPENDIX

8.1 MATLAB IMPLEMENTATION

The MATLAB code which was written to filter the image is provided below.

```
%% MATLAB Code for Low Pass Filtering an Image

clc;
close all;
clear all;

im = imread('C:\Users\user\Desktop\FPGA\Processor Design Project - Sem
5\MATLAB\im01_256.jpg'); % Open gray scale image

im_double = im2double(im);
[M N] = size(im); % M = number of rows

%% Defining Parameters

hw = 1; % Half width of the Kernel
sigma = 2/3; % Sigma value that determines the extent of smoothing
[x y] = meshgrid([-hw:hw]' , [-hw:hw]');
%% Defining Gausian Kernal

g = exp(-(x.^2 + y.^2)/(2*sigma^2))/(2*pi*sigma^2);

%% Filtering the Image

filtered_Im = zeros(M,N);

for i = 1+hw : M-hw
    for j = 1+hw : N-hw
        filtered_Im(i,j) = sum(sum(g.*im_double(i-hw:i+hw,j-hw:j+hw)));
    end
end

%% Filter Image Using In-built Functions

I1 = imfilter(im, g, 'conv');
I2 = filter2(im,g);
I3 = conv2(im,g);

%% Display Results

figure(1); imshow(im); title('Original Image');
figure(2); imshow(filtered_Im); title('Filtered Image');
figure(3), imshow(I1), title('Filtered Image Using Built-in Function');

%% Saving the Results

imwrite(filtered_Im,'Filtered_Im.png');
```

The MATLAB code which was written to down sample the image is provided below.

```
%% MATLAB Code for Down Sampling an Image

clc;
close all;
clear all;

im = imread('C:\Users\user\Desktop\FPGA\Processor Design Project - Sem
5\MATLAB\Filtered_Im.png'); % Open gray scale image

im_double = im2double(im);
[M N] = size(im); % M = number of rows

%% Down Sampling Code

M_new = 64;
N_new = 64;
Downsampling_Factor = 4;

Downsampled_Im = zeros(M_new,N_new);

for i=1:M_new
    for j=1:N_new
        Downsampled_Im(i,j) = sum(im(Downsampling_Factor*i,
Downsampling_Factor*j-3:Downsampling_Factor*j))/4;
    end
end

Downsampled_Im = uint8(Downsampled_Im);

%% Display Results

figure(1), imshow(im), title('Original Image');
figure(2), imshow(Downsampled_Im), title('Down-sampled Image');

%% Saving the Results

imwrite(Downsampled_Im,'DownSampled_Im.png');
```

In order to compare the differences and similarities of the two images, a MATLAB code was written. In that code, the two histograms of the obtained images are compared. Furthermore, the differences between pixel values of the two images are compared, and the total difference is printed as a percentage. MATLAB code written for the above purpose is provided below.

```
%% MATLAB Code for Comparing Two Images

clc;
close all;
clear all;

im1 = imread('Filtered_DS.png');
im2 = imread('Only_DS.png');

%% Comparison between Two Histograms

figure(1);
```

```
subplot(2,2,1); imshow(im1); title('Image 1');
subplot(2,2,2); imshow(im2); title('Image 2');
subplot(2,2,3); imhist(im1); axis tight; title('Histogram of Image 1');
subplot(2,2,4); imhist(im2); axis tight; title('Histogram of Image 2');

%% Comparison Using the Difference between Pixel Values

[m n] = size(im1);
totPixels = m*n;
similar = 0;
different = 0;

binIm1 = im1 > 125;
binIm2 = im2 > 125;

for i=1:m
    for j=1:n
        if isequal(binIm1(i,j), binIm2(i,j))
            similar = similar + 1;
        else
            different = different + 1;
        end
    end
end

differencePercentage = (different/totPixels)*100;

fprintf('%f%% difference between the compared images \n%d pixels being
different to %d total pixels\n', differencePercentage, different,
totPixels );
```

8.2 ASSEMBLY CODE COMPILER

```
#This is the Assembler for the CPU
#We open the text file containing our algorithm written in assembly
language
assembly_code = open("assembly_code.txt", "r")
decimal_sequence = open("decimal_sequence.txt", "w")
hex_sequence = open("assembly_hex.hex", "w")
#Read assembly code
assembly_text = assembly_code.read()
assembly_list = assembly_text.split('\n')
assembly_code.close()
#print assembly_list
#We require a function to convert our machine code to decimal
def b2d(binary):
      decimal = str(int('0b'+binary,2))
return decimal
#The assembler, which using our custom made ISA to convert assembly code to
machine code
def decode2decimal(instruction_list):
      decimal_list=[]
      ISA = {
                  'IDLE' : '000000', #0
                  'LDAC' : '000011', #3
                  'MOVACR' : '000101', #5
                  'MOVACR1' : '000110', #6
                  'MOVACR2' : '000111', #7
                  'MOVACR3' : '001000', #8
                  'MOVACR4' : '001001', #9
                  'MOVACR5' : '001010', #10
                  'MOVAC' : '001011', #11
                  'MOVRAC' : '001100', #12
                  'MOVR1AC' : '001101', #13
                  'MOVR2AC' : '001110', #14
                  'MOVR3AC' : '001111', #15
                  'MOVR4AC' : '010000', #16
                  'MOVR5AC' : '010001', #17
                  'STAC' : '001011', #19
                  'ADD' : '000001', #20
                  'SUB' : '010110', #22
                  'LSHIFT' : '011000', #24
                  'RSHIFT' : '011010', #26
                  'INCAC' : '011100', #28
                  'INCR1' : '011110', #30
                  'INCR2' : '011111', #31
                  'INCR3' : '100000', #32
                  'LDIAC' : '100001', #33
                  'JMPZ' : '100011', #35
                  'JPNZ' : '100111', #39
                  'JUMP' : '101000', #40
                  'NOP' : '101001', #41
                  'ENDOP' : '101010', #42
                  }
      N = 256 #First pixel for CONV to start is N+1
      F_CONV = (N*N-1)-(N+1) # Final pixel for CONV to end
      # 65278 for N = 256
      F_SAMP = (N*N-1)-(N+1)-(N+1)-(N+1) # Final pixel for DOWN_SAMPLING to
      end
      # 64764 for N = 256
      #Jump commands
```

```
    JUMPZ1 = 129# To loop between rows of CONV
    JUMPZ2 = 138# To finish CONV and goto DOWN_SAMP
    JUMPZ3 = 171# To loop between rows of DOWN_SAMP
    JUMPZ4 = 186# To finish entire operation
    JUMP1 = 9 # To Loop CONV
    JUMP2 = 143 # To Loop DOWN_SAMP
    for instruction in instruction_list:
        if instruction in ISA:
            decimal_list.append("6\'d"+b2d(ISA[instruction]))
        elif instruction == "[Address of N+4]":
            decimal_list.append("16\'d"+str(N+4))
        elif instruction == "[Address of N+1]":
            decimal_list.append("16\'d"+str(N+1))
        elif instruction == "[Address of N]":
            decimal_list.append("16\'d"+str(N))
        elif instruction == "[Address of N-1]":
            decimal_list.append("8\'d"+str(N-1))
        elif instruction == "[Address of N-3]":
            decimal_list.append("8\'d"+str(N-3))
        elif instruction == "[Address of N-4]":
            decimal_list.append("8\'d"+str(N-4))
        elif instruction == "[Address of F_CONV]":
            decimal_list.append("16\'d"+str(F_CONV))
        elif instruction == "[Address of F_SAMP]":
            decimal_list.append("16\'d"+str(F_SAMP))
        else:
            decimal_list.append(str(instruction))
return decimal_list

def decode2hex(instruction_list):
    hex_list=[]
    ISA = {
            'IDLE' : '000000', #0
            'LDAC' : '000011', #3
            'MOVACR' : '000101', #5
            'MOVACR1' : '000110', #6
            'MOVACR2' : '000111', #7
            'MOVACR3' : '001000', #8
            'MOVACR4' : '001001', #9
            'MOVACR5' : '001010', #10
            'MOVAC' : '001011', #11
            'MOVRAC' : '001100', #12
            'MOVR1AC' : '001101', #13
            'MOVR2AC' : '001110', #14
            'MOVR3AC' : '001111', #15
            'MOVR4AC' : '010000', #16
            'MOVR5AC' : '010001', #17
            'STAC' : '001011', #19
            'ADD' : '000001', #20
            'SUB' : '010110', #22
            'LSHIFT' : '011000', #24
            'RSHIFT' : '011010', #26
            'INCAC' : '011100', #28
            'INCR1' : '011110', #30
            'INCR2' : '011111', #31
            'INCR3' : '100000', #32
            'LDIAC' : '100001', #33
            'JMPZ' : '100011', #35
            'JPNZ' : '100111', #39
            'JUMP' : '101000', #40
            'NOP' : '101001', #41
            'ENDOP' : '101010', #42
```

```
                  }
      N = 256 #First pixel for CONV to start is N+1
      F_CONV = (N*N-1)-(N+1) # Final pixel for CONV to end
      # 65278 for N = 256
      F_SAMP = (N*N-1)-(N+1)-(N+1)-(N+1) # Final pixel for DOWN_SAMPLING to
      end
      # 64764 for N = 256
      for instruction in instruction_list:
      if instruction in ISA:
            hex_list.append(hex(int((ISA[instruction]), 2)))
      elif instruction == "[Address of N+4]":
            hex_list.append(hex(N+4))
      elif instruction == "[Address of N+1]":
            hex_list.append(hex(N+1))
      elif instruction == "[Address of N]":
            hex_list.append(hex(N))
      elif instruction == "[Address of N-1]":
            hex_list.append(hex(N-1))
      elif instruction == "[Address of N-3]":
            hex_list.append(hex(N-3))
      elif instruction == "[Address of N-4]":
            hex_list.append(hex(N-4))
      elif instruction == "[Address of F_CONV]":
            hex_list.append(hex(F_CONV))
      elif instruction == "[Address of F_SAMP]":
            hex_list.append(hex(F_SAMP))
      else:
            d = instruction.index('d')
            hex_list.append(hex(int(instruction[d+1:])))
return hex_list

decimal_list = decode2decimal(assembly_list)
hex_list = decode2hex(assembly_list)
len_d = len(decimal_list)
len_h = len(hex_list)

print(len_h)
print(hex_list)

decimal_sequence.writelines(["%s\n"%value for value in decimal_list])
decimal_sequence.close()
hex_sequence.writelines(["%s\n"%value[2:] for value in hex_list])
hex_sequence.close()
```

8.3 VERILOG CODES FOR THE PROCESSOR DESIGN

A. TOP MODULE

```
module top_processor ( input wire data_from_pc ,
input wire fast_clock ,
input wire start_process ,
input wire start_transmit ,
output wire data_to_pc ,
output wire l0,
output wire l1,
output wire l2,
output wire l3,
output wire g1,
output wire g2,
output wire g3);

wire [7:0] dm_out;
wire [15:0] im_out;
wire [15:0] bus_out;
wire dm_en;
wire [15:0] dar_out;
wire im_en;
wire [15:0] pc_out;
wire end_receiving ;
wire end_process ;
wire end_transmitting ;
wire [1:0] status;
wire [15:0] data_out_com ;
wire en_com ;
wire [15:0] addr_com ;
wire [7:0] data_in_com ;
wire clock;

reg begin_process ;
reg begin_transmit ;

wire [15:0] datain;
wire data_write_en ;
wire [15:0] data_addr ;
wire [15:0] instr_in ;
wire instr_write_en ;
wire [15:0] instr_addr ;

reg [9:0] process_switch_buffer = 10'd0;
reg [9:0] transmit_switch_buffer = 10'd0;

always @(posedge clock)
begin
if (start_process )
begin
if (process_switch_buffer == 10'd1023 )
begin
process_switch_buffer <= process_switch_buffer ;
begin_process <=1;
end
else
begin
process_switch_buffer <= process_switch_buffer + 10'd1;
begin_process <=0;
```

```verilog
end
end
else
begin
process_switch_buffer <= 10'd0;
begin_process <= 0;
end
end

always @(posedge clock)
begin
if (start_transmit )
begin
if (transmit_switch_buffer == 10'd1023 )
begin
transmit_switch_buffer <= transmit_switch_buffer ;
begin_transmit <=1;
end
else
begin
transmit_switch_buffer <= transmit_switch_buffer + 10'd1;
begin_transmit <=0;
end
end
else
begin
transmit_switch_buffer <= 10'd0;
begin_transmit <= 0;
end
end

slowclock slowclock1 (.clockin(fast_clock ),
.clockout (clock));

instr_memory instr_memory1 (.clock(clock),
.write_en (im_en),
.addr(pc_out),
.instr_out (im_out),
.instr_in (bus_out));

datamemory datamemory1 ( .clock(clock),
.write_en (data_write_en ),
.addr(data_addr ),
.datain(datain),
.dataout(dm_out));

processor processor1 (.clock(clock),
.dm_out(dm_out),
.im_out(im_out),
.dm_en(dm_en),
.im_en(im_en),
.pc_out(pc_out),
.dar_out(dar_out),
.status(status),
.bus_out(bus_out),
.end_process (end_process ));

selector selector1 ( .clock(clock),
.status(status),
.bus_out(bus_out),
.dm_en(dm_en),
.dar_out(dar_out),
.data_out_com (data_out_com ),
.en_com(en_com),
.addr_com (addr_com ),
.datain(datain),
.data_write_en (data_write_en ),
.data_addr (data_addr ),
.data_in_com (data_in_com ));

communicate communicate1 ( .clock(clock),
.status(status),
.end_receiving (end_receiving ),
.end_transmitting (end_transmitting ),
.data_in_com (data_in_com ),
.data_out_com (data_out_com ),
.addr_com (addr_com ),
.data_to_pc (data_to_pc ),
.data_from_pc (data_from_pc ),
.en_com(en_com));

main_control main_control1 (.clock(clock),
.end_receiving (end_receiving ),
.end_process (end_process ),
.end_transmitting (end_transmitting ),
.begin_process (begin_process ),
.begin_transmit (begin_transmit ),
.status(status),
.l0(l0),
.l1(l1),
.l2(l2),
.l3(l3),
.g1(g1),
.g2(g2),
.g3(g3));

endmodule
```

B. THE PROCESSOR

```
module processor ( input clock,
input [7:0] dm_out,
input [15:0] im_out,
input [1:0] status,

output reg dm_en,
output reg im_en,
output [15:0] pc_out,
output [15:0] dar_out,
output [15:0]bus_out,
output end_process );



wire [2:0] alu_op;
wire [15:0] alu_out;

wire [15:0] regr_out ;
wire [15:0] regr1_out ;
wire [15:0] regr2_out ;
wire [15:0] regr3_out ;
wire [15:0] regr4_out ;
wire [15:0] regr5_out ;
wire [15:0] ac_out;

wire [15:0] ir_out;

wire [15:0] write_en ;
wire [3:0] read_en;
wire [15:0] inc_en;
wire [15:0] mem0;
wire [15:0] mem1;
wire [15:0] mem2;
wire [15:0] mem3;
wire [15:0] mem4;
wire [15:0] mem5;
wire [15:0] mem6;
wire [15:0] mem7;
wire [15:0] mem8;

wire [15:0] z ;

regr reg_r(.clock(clock), .write_en (write_en[6]),.datain(bus_out),.dataout(regr_out ));

regrinc regr_r1(.clock(clock), .write_en (write_en[7]),.datain(bus_out),.dataout(regr1_out ),.inc_en(inc_en[4]));

regrinc regr_r2(.clock(clock), .write_en (write_en[8]),.datain(bus_out),.dataout(regr2_out ),.inc_en(inc_en[5]));

regrinc regr_r3(.clock(clock), .write_en (write_en[9]),.datain(bus_out),.dataout(regr3_out ),.inc_en(inc_en[6]));

regr regr_r4(.clock(clock), .write_en (write_en[10]),.datain(bus_out),.dataout(regr4_out ));

regr regr_r5(.clock(clock), .write_en (write_en[11]),.datain(bus_out),.dataout(regr5_out ));

regrinc dar(.clock(clock), .write_en (write_en[2]),.datain(bus_out),.dataout(dar_out),.inc_en(inc_en[3]));

regr ir(.clock(clock), .write_en (write_en[4]),.datain(bus_out),.dataout(ir_out));
```

```

bus
bus1(.r1(regr1_out ),.r2(regr2_out ),.r3(regr3_out ),.r4(regr4_out ),.r5(regr5_out ),.r(regr_out ),.dar(dar_out),.ir(ir_out),.pc(pc_out),.ac(ac_out),.dm(dm_out),.im(im_out),.busout(bus_out),.read_en(read_en),.clock(clock));

ac ac1(.clock(clock), .write_en (write_en[5]),.datain(bus_out),.dataout(ac_out),.alu_out(alu_out),.alu_to_ac (write_en [14]),.inc_en(inc_en[2]));

```

```
regrinc pc(.clock(clock), .write_en (write_en [1]),.datain(bus_out),.dataout(pc_out),.inc_en(inc_en[1]));

alu

alu1(.clock(clock),.in1(regr_out ),.in2(ac_out),.alu_op(alu_op),.alu_out(alu_out),.z(z));

control

control1 (.clock(clock),.z(z),.instruction (ir_out),.alu_op(alu_op),.write_en (write_en ),.read_en(read_en),.inc_en(inc_en),.end_process (end_process ),.status(status));

always @ (posedge clock)

if (status == 2'b01)begin
dm_en <= write_en [12];
im_en <= write_en [13];

end

endmodule
```

C. STATE MACHINE

```
module main_control ( input clock,
input end_receiving ,
input end_process ,
input end_transmitting ,
input begin_process ,
input begin_transmit ,
output reg [1:0] status,
output reg l0,
output reg l1,
output reg l2,
output reg l3,
output reg g1,
output reg g2,
output reg g3);

reg [1:0] present = 2'b00;
reg [1:0] next = 2'b00;

parameter
receive = 2'b00,
process = 2'b01,
transmit = 2'b10,
alldone = 2'b11;

always @(posedge clock)
begin
if (begin_process )
begin
g1 <= 1;
end
```

```
if (begin_transmit )
begin
g2 <= 1;
end
if (end_receiving )
begin
g3 <= 1;
end
end

initial
begin
g1 <= 0;
g2 <= 0;
l0 <= 0;
l1 <= 0;
l2 <= 0;
l3 <= 0;
g3 <= 0;
end

always @(posedge clock)
present <= next;

always @(present or begin_transmit or begin_process or
end_receiving or end_process or
end_transmitting )
case(present)
receive: begin
status <= 2'b00;
l0 <=1;
l1 <=0;
l2 <=0;
```

```
l3 <=0;

if (end_receiving && !end_process && !end_transmitting )
next<=process;
else
next<=receive;
end

process: begin
status <= 2'b01;
l0 <=1;
l1 <=1;
l2 <=0;
l3 <=0;

if (end_receiving && end_process && !end_transmitting &&
begin_transmit )
next<=transmit ;
else
next<=process;
end

transmit : begin
status <= 2'b10;
l0 <=1;
l1 <=1;
l2 <=1;
```

```
l3 <=0;

if (end_receiving && end_process && end_transmitting )
next<=alldone;
else
next<=transmit ;
end

alldone: begin
status <= 2'b11;
l0 <=1;
l1 <=1;
l2 <=1;
l3 <=1;
next<=alldone;
end
endcase

endmodule
```

D. ALU - ARITHMETIC AND LOGIC UNIT

```
module alu( input clock,
input [15:0] in1,
input [15:0] in2,
input [2:0] alu_op,
output reg [15:0] alu_out,
output reg [15:0] z );

always @(posedge clock)
begin
case(alu_op)
3'd1: alu_out <= in1 + in2;
3'd2: alu_out <= in2 - in1;
3'd3: alu_out <= in1 << in2;
3'd4: alu_out <= in1 >> in2;
endcase

if (alu_out==0)
z <= 1;
else
z <= 0;
end

endmodule
```

E. AC - ACCUMULATOR

```
module ac( input clock,
input write_en ,
input [15:0] datain,
output reg [15:0] dataout = 16'd0,
input [15:0] alu_out,
input alu_to_ac ,
input inc_en);

always @(posedge clock)
```

```
begin
if (inc_en == 1)
dataout <= dataout + 16'd1;
if (write_en == 1)
dataout <= datain;
if (alu_to_ac == 1)
dataout <= alu_out;
end

endmodule
```

F. CU - CONTROL UNIT

```
module control(input clock,
input [15:0] z,
input [15:0] instruction ,
output reg [2:0] alu_op,
output reg [15:0] write_en ,
output reg [15:0] inc_en,
output reg [3:0] read_en,
output reg end_process ,
input [1:0] status );

reg [5:0] present = 6'd0;
reg [5:0] next = 6'd0;

parameter
fetch1 = 6'd1,
fetch2 = 6'd2,
loadac1 = 6'd3,
loadac2 = 6'd4,
loadac1x = 6'd50,
loadac2x = 6'd51,
movacr = 6'd5,
movacr1 = 6'd6,
movacr2 = 6'd7,
movacr3 = 6'd8,
movacr4 = 6'd9,
movacr5 = 6'd10,
movacdar = 6'd11,
movrac = 6'd12,
movr1ac = 6'd13,
movr2ac = 6'd14,
movr3ac = 6'd15,
movr4ac = 6'd16,
movr5ac = 6'd17,
movdarac = 6'd18,
stac = 6'd19,
stacx = 6'd49,
stacy = 6'd52,
add1 = 6'd20,
add2 = 6'd21,
sub1 = 6'd22,
sub2 = 6'd23,
lshift1 = 6'd24,
lshift2 = 6'd25,
rshift1 = 6'd26,
rshift2 = 6'd27,
incac = 6'd28,
incdar = 6'd29,
incr1 = 6'd30,
incr2 = 6'd31,
incr3 = 6'd32,
loadim1 = 6'd33,
loadim2 = 6'd34,
loadimx = 6'd45,
jumpz1 = 6'd35,
jumpz2 = 6'd36,
jumpz3 = 6'd37,
jumpz4 = 6'd38,
jumpz2x = 6'd46,
jumpz3x = 6'd47,
jumpz4x = 6'd48,
jumpnz1 = 6'd39,
jump1 = 6'd40,
```

Date: July 19, 2018 control.v Project: processor_design
Page 2 of 9 Revision: processor_design

```
nop = 6'd41,
endop = 6'd42,
fetchx = 6'd44,
stac2 = 6'd43,
idle = 6'd0;

always @(posedge clock)
present <= next;

always @(posedge clock)
begin
if (present == endop)
end_process <= 1'd1;
else
end_process <= 1'd0;
end

always @(present or z or instruction or status)
case(present)
idle: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
if (status == 2'b01)
next <= fetch1;
else
next <= idle;
end

fetch1: begin
read_en <= 4'd13;
write_en <= 16'b0000000000010000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= fetchx;
end

fetchx: begin
read_en <= 4'd13;
write_en <= 16'b0000000000010000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= fetch2;
end
```

```

fetch2: begin
read_en <= 4'd13;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= instruction [5:0];
end

loadac1: begin
read_en <= 4'd5;
write_en <= 16'b0000000000000100 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= loadac1x ;
end

loadac1x : begin
read_en <= 4'd5;
write_en <= 16'b0000000000000100 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= loadac2x ;
end

loadac2x : begin
read_en <= 4'd12;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= loadac2;
end

loadac2: begin
read_en <= 4'd12;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movacr: begin
read_en <= 4'd5;
write_en <= 16'b0000000001000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movacr1: begin
read_en <= 4'd5;
write_en <= 16'b0000000010000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movacr2: begin
read_en <= 4'd5;
write_en <= 16'b0000000100000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
```

```
next <= fetch1;
end

read_en <= 4'd5;
write_en <= 16'b0000001000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movacr4: begin
read_en <= 4'd5;
write_en <= 16'b0000010000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movacr5: begin
read_en <= 4'd5;
write_en <= 16'b0000100000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movacdar : begin
read_en <= 4'd5;
write_en <= 16'b0000000000000100 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movrac: begin
read_en <= 4'd6;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movr1ac: begin
read_en <= 4'd7;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movr2ac: begin
read_en <= 4'd8;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movr3ac: begin
read_en <= 4'd9;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
```

```
alu_op <= 3'd0;
next <= fetch1;
end

movr4ac: begin
read_en <= 4'd10;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movr5ac: begin
read_en <= 4'd11;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

movdarac : begin
read_en <= 4'd2;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

stac: begin
read_en <= 4'd5;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= stac2;
end

stac2: begin
read_en <= 4'd5;
write_en <= 16'b0001000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= stacx;
end

stacx: begin
read_en <= 4'd5;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

add1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd1;
next <= add2;
end

add2: begin
read_en <= 4'd0;
```

```
write_en <= 16'b1100000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd1;
next <= fetch1;
end

sub1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd2;
next <= sub2;
end

sub2: begin
read_en <= 4'd0;
write_en <= 16'b1100000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd2;
next <= fetch1;
end

lshift1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd3;
next <= lshift2;
end

lshift2: begin
read_en <= 4'd0;
write_en <= 16'b1100000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd3;
next <= fetch1;
end

rshift1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd4;
next <= rshift2;
end

rshift2: begin
read_en <= 4'd0;
write_en <= 16'b1100000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd4;
next <= fetch1;
end

incac: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000110 ;
alu_op <= 3'd0;
next <= fetch1;
end

```

```
incdar: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000001010 ;
alu_op <= 3'd0;
next <= fetch1;
end

incr1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000010010 ;
alu_op <= 3'd0;
next <= fetch1;
end

incr2: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000100010 ;
alu_op <= 3'd0;
next <= fetch1;
end

incr3: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000001000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

loadim1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= loadimx;
end

loadimx: begin
read_en <= 4'd13;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= loadim2;
end

loadim2: begin
read_en <= 4'd13;
write_en <= 16'b0000000000100000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

jump1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= jumpz2x;
```

```
end

jumpz1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
if (z == 1)
next <= jumpz2x;
else
next <= jumpz4x;
end

jumpnz1: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
if (z == 0 )
next <= jumpz2;
else
next <= jumpz4;
end

jumpz2x: begin
read_en <= 4'd13;
write_en <= 16'b0000000000000010 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= jumpz2;
end

jumpz2: begin
read_en <= 4'd13;
write_en <= 16'b0000000000000010 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= jumpz3x;
end

jumpz3x: begin
read_en <= 4'd13;
write_en <= 16'b0000000000010000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= jumpz3;
end

jumpz3: begin
read_en <= 4'd13;
write_en <= 16'b0000000000010000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= fetch2;
end

jumpz4x: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= jumpz4;
```

```
end

jumpz4: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= jumpz3x;
end

nop: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000010 ;
alu_op <= 3'd0;
next <= fetch1;
end

endop: begin
read_en <= 4'd12;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= endop;
end

default: begin
read_en <= 4'd0;
write_en <= 16'b0000000000000000 ;
inc_en <= 16'b0000000000000000 ;
alu_op <= 3'd0;
next <= fetch1;
end
endcase

endmodule
```

G. DATA MEMORY (DRAM)

```
module datamemory (input clock,
input write_en ,
input [15:0] addr,
input [15:0] datain,
output reg [7:0] dataout );

reg [7:0] ram [65535:0];

always @(posedge clock)
begin
if (write_en == 1)
ram[addr] <= datain[7:0];
else
dataout <= ram[addr];
end

endmodule
```

H. INSTRUCTION MEMORY (IRAM)

```verilog
module instr_memory ( input clock,
input write_en ,
input [15:0] addr,
input [15:0] instr_in ,
output reg [15:0] instr_out );


reg [15:0] ram [190:0];

parameter ldac = 8'd3;
parameter movacr = 8'd5;
parameter movacr1 = 8'd6;
parameter movacr2 = 8'd7;
parameter movacr3 = 8'd8;
parameter movacr4 = 8'd9;
parameter movacr5 = 8'd10;
parameter movacdar = 8'd11;
parameter movrac = 8'd12;
parameter movr1ac = 8'd13;
parameter movr2ac = 8'd14;
parameter movr3ac = 8'd15;
parameter movr4ac = 8'd16;
parameter movr5ac = 8'd17;
parameter movdarac = 8'd18;
parameter stac = 8'd19;
parameter add = 8'd20;
parameter sub = 8'd22;
parameter lshift = 8'd24;
parameter rshift = 8'd26;
parameter incac = 8'd28;
parameter incdar = 8'd29;
parameter incr1 = 8'd30;
parameter incr2 = 8'd31;
parameter incr3 = 8'd32;
parameter loadim = 8'd33;
parameter jumpz = 8'd35;
parameter jumpnz = 8'd39;
parameter jump = 8'd40;
parameter nop = 8'd41;
parameter endop = 8'd42;

initial begin
ram[0] = loadim;
ram[1] = 16'd257;
ram[2] = movacr1;
ram[3] = nop;
ram[4] = nop;
ram[5] = nop;
ram[6] = nop;
ram[7] = nop;
ram[8] = nop;
ram[9] = movr1ac;
ram[10] = ldac;
ram[11] = movacr;
ram[12] = loadim;
ram[13] = 8'd4;
ram[14] = lshift;
ram[15] = movacr4;
ram[16] = loadim;
ram[17] = 8'd1;
ram[18] = movacr;
ram[19] = movr1ac;
```

```
ram[20] = add;
ram[21] = ldac;
ram[22] = movacr5;
ram[23] = movr1ac;
ram[24] = sub ;
ram[25] = ldac;
ram[26] = movacr;
ram[27] = movr5ac;
ram[28] = add;
ram[29] = movacr5;
ram[30] = loadim;
ram[31] = 16'd256;
ram[32] = movacr;
ram[33] = movr1ac;
ram[34] = add;
ram[35] = ldac;
ram[36] = movacr;
ram[37] = movr5ac;
ram[38] = add;
ram[39] = movacr5;
ram[40] = loadim;
ram[41] = 16'd256;
ram[42] = movacr;
ram[43] = movr1ac;
ram[44] = sub;
ram[45] = ldac;
ram[46] = movacr;
ram[47] = movr5ac;
ram[48] = add;
ram[49] = movacr5;
ram[50] = movacr;
ram[51] = loadim;
ram[52] = 8'd1;
ram[53] = lshift;
ram[54] = movacr;
ram[55] = movr5ac;
ram[56] = add;
ram[57] = movacr;
ram[58] = movr4ac;
ram[59] = add;
ram[60] = movacr4;
ram[61] = loadim;
ram[62] = 16'd257;
ram[63] = movacr;
ram[64] = movr1ac;
ram[65] = add;
ram[66] = ldac;
ram[67] = movacr5;
ram[68] = movr1ac;
ram[69] = sub;
ram[70] = ldac;
ram[71] = movacr;
ram[72] = movr5ac;
ram[73] = add;
ram[74] = movacr5;
ram[75] = loadim;
ram[76] = 8'd255;
ram[77] = movacr;
ram[78] = movr1ac;
ram[79] = add;
ram[80] = ldac;
ram[81] = movacr;
```

```
ram[82] = movr5ac;
ram[83] = add;
ram[84] = movacr5;
ram[85] = loadim;
ram[86] = 8'd255;
ram[87] = movacr;
ram[88] = movr1ac;
ram[89] = sub;
ram[90] = ldac;
ram[91] = movacr;
ram[92] = movr5ac;
ram[93] = add;
ram[94] = movacr5;
ram[95] = movacr;
ram[96] = movr4ac;
ram[97] = add;
ram[98] = movacr;
ram[99] = loadim;
ram[100] = 8'd5;
ram[101] = rshift;
ram[102] = movacr4;
ram[103] = loadim;
ram[104] = 16'd257;
ram[105] = movacr;
ram[106] = movr1ac;
ram[107] = sub;
ram[108] = movacdar ;
ram[109] = movr4ac;
ram[110] = stac;
ram[111] = loadim;
ram[112] = 16'd65278 ;
ram[113] = movacr;
ram[114] = movr1ac;
ram[115] = sub;
ram[116] = jumpz;
ram[117] = 8'd138;
ram[118] = loadim;
ram[119] = 8'd253;
ram[120] = movacr;
ram[121] = movr2ac;
ram[122] = sub ;
ram[123] = jumpz;
ram[124] = 8'd129;
ram[125] = incr2;
ram[126] = incr1;
ram[127] = jump;
ram[128] = 8'd9;
ram[129] = incr1;
ram[130] = incr1;
ram[131] = incr1;
ram[132] = loadim;
ram[133] = 8'd0;
ram[134] = movacr2;
ram[135] = jump;
ram[136] = 8'd9;
ram[137] = endop;
ram[138] = loadim;
ram[139] = 8'd0;
ram[140] = movacr1;
ram[141] = movacr2;
ram[142] = movacr3;
ram[143] = movr1ac;
```

```
ram[144] = ldac;
ram[145] = movacr4;
ram[146] = movr3ac;
ram[147] = movacdar ;
ram[148] = movr4ac;
ram[149] = stac;
ram[150] = movr1ac;
ram[151] = movacr;
ram[152] = loadim;
ram[153] = 16'd64764 ;
ram[154] = sub;
ram[155] = jumpz;
ram[156] = 8'd186;
ram[157] = movr2ac;
ram[158] = movacr;
ram[159] = loadim;
ram[160] = 8'd252;
ram[161] = sub;
ram[162] = jumpz;
ram[163] = 8'd171;
ram[164] = incr2;
ram[165] = incr2;
ram[166] = incr1;
ram[167] = incr1;
ram[168] = incr3;
ram[169] = jump;
ram[170] = 8'd143;
ram[171] = loadim;
ram[172] = 8'd0;
ram[173] = movacr2;
ram[174] = loadim;
ram[175] = 16'd260;
ram[176] = movacr;
ram[177] = movr1ac;
ram[178] = add;
ram[179] = movacr1;
ram[180] = incr3;
ram[181] = nop;
ram[182] = nop;
ram[183] = nop;
ram[184] = jump;
ram[185] = 8'd143;
ram[186] = endop;
end

always @(posedge clock) begin
if (write_en == 1)
ram[addr] <= instr_in [7:0];
else
instr_out <= ram[addr];
end

endmodule
```

I. REGISTERS WITHOUT INCREMENT

```
module regr(input clock,
input write_en ,
input [15:0] datain,
output reg [15:0] dataout );

always @(posedge clock)
begin
if (write_en == 1)
dataout <= datain;
end

endmodule
```

J. REGISTERS WITH INCREMENT

```
module regrinc(input clock,
input write_en ,
input [15:0] datain,
output reg [15:0] dataout = 16'd0,
input inc_en);

always @(posedge clock)
begin
if (write_en == 1)
dataout <= datain;
if (inc_en == 1)
dataout <= dataout + 16'd1;
end

endmodule
```

K. BUS

```
module bus( input clock,
input [3:0] read_en,
input [15:0] r1,
input [15:0] r2,
input [15:0] r3,
input [15:0] r4,
input [15:0] r5,
input [15:0] r,
input [15:0] dar,
input [15:0] ir,
input [15:0] pc,
input [15:0] ac,
input [7:0] dm,
input [15:0] im,
output reg [15:0] busout ) ;
always @(r1 or r2 or r3 or r4 or r5 or r or dar or ir or pc or ac or im or read_en or dm)

begin
case(read_en)
4'd1: busout <= pc;
4'd2: busout <= dar;
4'd4: busout <= ir;
4'd5: busout <= ac;
4'd6: busout <= r;
4'd7: busout <= r1;
4'd8: busout <= r2;
```

```
4'd9: busout <= r3;
4'd10: busout <= r4;
4'd11: busout <= r5;
4'd12: busout <= dm + 16'd0;
4'd13: busout <= im ;
default: busout <= 16'd0;
endcase
end
endmodule
```

L. SELECTOR

```
module selector ( input clock,
input [1:0] status;
input [15:0] bus_out,
input dm_en,
input [15:0] dar_out,
input [15:0] data_out_com ,
input en_com,
input [15:0] addr_com ,
output reg [15:0] datain,
output reg data_write_en ,
output reg [15:0] data_addr ,
output reg [7:0] data_in_com );

always @(posedge clock)
case (status)
2'b00:begin
datain <= data_out_com ;
data_write_en <= en_com;
data_addr <= addr_com ;
end

2'b01:begin
datain <= bus_out;
data_write_en <=dm_en;
data_addr <= dar_out;
end

2'b10:begin
data_in_com <= bus_out[7:0];
data_write_en <= en_com;
data_addr <= addr_com ;
end
endcase

endmodule
```

M. CLOCK DIVIDER

```
module slowclock ( clockin,clockout );

input clockin;
output clockout ;

reg clockout = 1'b0;
reg counter = 1'b0;

always @ (posedge clockin)
begin
```

```
counter <= counter + 1'b1;
if(counter == 1'b1)
clockout <= ~clockout ;
end
endmodule
```

N. BAUD RATE GENERATOR

```
module BaudTickGen (
input clk, enable,
output tick // generate a tick at the specified baud rate *
oversampling
);
parameter ClkFrequency = 25000000 ;
parameter Baud = 115200;
parameter Oversampling = 1;

function integer log2(input integer v); begin log2=0;
while(v>>log2) log2=log2+1; end
endfunction
localparam AccWidth = log2(ClkFrequency /Baud)+8; // +/- 2% max
timing error over a byte
reg [AccWidth :0] Acc = 0;
localparam ShiftLimiter = log2(Baud*Oversampling >> (31-
AccWidth )); // this makes sure Inc
calculation doesn't overflow
localparam Inc = ((Baud*Oversampling << (AccWidth -
ShiftLimiter ))+(ClkFrequency >>(
ShiftLimiter +1)))/(ClkFrequency >>ShiftLimiter );
always @(posedge clk) if(enable) Acc <= Acc[AccWidth -1:0] +
Inc[AccWidth :0]; else Acc <= Inc
[AccWidth :0];
assign tick = Acc[AccWidth ];
endmodule
```

O. TRANSMITTER

```
module uart_tx
#(parameter CLKS_PER_BIT = 217)
(
input Clock,
input Tx_DV,
input [7:0] Tx_Byte,
output Tx_Active ,
output reg Tx_Serial ,
output Tx_Done
);

parameter IDLE = 3'b000;
parameter TX_START_BIT = 3'b001;
parameter TX_DATA_BITS = 3'b010;
parameter TX_STOP_BIT = 3'b011;
parameter CLEANUP = 3'b100;

reg [2:0] r_state = 0;
reg [7:0] r_Clock_Count = 0;
reg [2:0] r_Bit_Index = 0;
reg [7:0] r_Tx_Data = 0;
reg r_Tx_Done = 0;
reg r_Tx_Active = 0;

always @(posedge Clock)
begin

case (r_state)
IDLE :
begin
Tx_Serial <= 1'b1;
r_Tx_Done <= 1'b0;
r_Clock_Count <= 0;
r_Bit_Index <= 0;

if (i_Tx_DV == 1'b1)
begin
r_Tx_Active <= 1'b1;
r_Tx_Data <= Tx_Byte;
r_state <= TX_START_BIT ;
end
else
r_state <= IDLE;
end

TX_START_BIT :
begin
Tx_Serial <= 1'b0;

// Wait CLKS_PER_BIT-1 clock cycles for start bit to finish
if (r_Clock_Count < CLKS_PER_BIT -1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_state <= TX_START_BIT ;
end
else
begin
r_Clock_Count <= 0;
r_state <= TX_DATA_BITS ;
end
end

// Wait CLKS_PER_BIT-1 clock cycles for data bits to finish
TX_DATA_BITS :
begin
Tx_Serial <= r_Tx_Data [r_Bit_Index ];

if (r_Clock_Count < CLKS_PER_BIT -1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_state <= TX_DATA_BITS ;
end
else
begin
r_Clock_Count <= 0;

// Check if we have sent out all bits
if (r_Bit_Index < 7)
begin
r_Bit_Index <= r_Bit_Index + 1;
r_state <= TX_DATA_BITS ;
end
else
```

```
begin
r_Bit_Index <= 0;
r_state <= TX_STOP_BIT ;
end
end
end

// Send out Stop bit.
TX_STOP_BIT :
begin
Tx_Serial <= 1'b1; //Stop bit = 1
if (r_Clock_Count < CLKS_PER_BIT -1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_state <= TX_STOP_BIT ;
end
else
begin
r_Tx_Done <= 1'b1;
r_Clock_Count <= 0;
r_state <= CLEANUP;
r_Tx_Active <= 1'b0;
end
end

s_CLEANUP :
begin
r_Tx_Done <= 1'b1;
r_state <= IDLE;
end

default :
r_state <= IDLE;

endcase
end

assign Tx_Active = r_Tx_Active ;
assign Tx_Done = r_Tx_Done ;

endmodule
```

P. RECEIVER

```
module uart_rx
#(parameter CLKS_PER_BIT = 217)
(
input Clock,
input Rx_Serial ,
output Rx_DV,
output [7:0] Rx_Byte
);

parameter IDLE = 3'b000;
parameter RX_START_BIT = 3'b001;
parameter RX_DATA_BITS = 3'b010;
parameter RX_STOP_BIT = 3'b011;
parameter CLEANUP = 3'b100;

reg r_Rx_Data_R = 1'b1;
reg r_Rx_Data = 1'b1;
reg [7:0] r_Clock_Count = 0;
reg [2:0] r_Bit_Index = 0; //8 bits total
reg [7:0] r_Rx_Byte = 0;
reg r_Rx_ready = 0;
reg [2:0] r_state = 0;

// Purpose: Double-register the incoming data.
// This allows it to be used in the UART RX Clock Domain.
// (It removes problems caused by metastability)
always @(posedge i_Clock)
begin
r_Rx_Data_R <= Rx_Serial ;
r_Rx_Data <= r_Rx_Data_R ;
end

always @(posedge i_Clock)
begin
case (r_state)
IDLE :
begin
r_Rx_ready <= 1'b0;
r_Clock_Count <= 0;
r_Bit_Index <= 0;

if (r_Rx_Data == 1'b0) // Start bit detected
r_state <= RX_START_BIT ;
else
r_state <= IDLE;
end

// Check middle of start bit to make sure it's still low
RX_START_BIT :
begin
if (r_Clock_Count == (CLKS_PER_BIT -1)/2)
begin
if (r_Rx_Data == 1'b0)
begin
r_Clock_Count <= 0; // reset counter, found the middle
r_state <= RX_DATA_BITS ;
end
else
r_state <= IDLE;
end
else
begin
r_Clock_Count <= r_Clock_Count + 1;
r_state <= RX_START_BIT ;
end
end

// Wait CLKS_PER_BIT-1 clock cycles to sample serial data
RX_DATA_BITS :
begin
if (r_Clock_Count < CLKS_PER_BIT -1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_state <= RX_DATA_BITS ;
end
else
begin
r_Clock_Count <= 0;
```

```
r_Rx_Byte [r_Bit_Index ] <= r_Rx_Data ;

// Check if we have received all bits
if (r_Bit_Index < 7)
begin
r_Bit_Index <= r_Bit_Index + 1;
r_state <= RX_DATA_BITS ;
end
else
begin
r_Bit_Index <= 0;
r_state <= RX_STOP_BIT ;
end
end
end

// Receive Stop bit.
RX_STOP_BIT :
begin
// Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
if (r_Clock_Count < CLKS_PER_BIT -1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_state <= RX_STOP_BIT ;
end
else
begin
r_Rx_DV <= 1'b1; //Stop bit = 1
r_Clock_Count <= 0;
r_state <= CLEANUP;
end
end

CLEANUP :
begin
r_state <= IDLE;
r_Rx_ready <= 1'b0;
end

default :
r_state <= s_IDLE;

endcase
end

assign Rx_DV = r_Rx_DV;
assign Rx_Byte = r_Rx_Byte ;

endmodule
```

Q. COMMUNICATION

```
module communicate ( input clock,
input [1:0] status,
input data_from_pc ,
input [7:0] data_in_com ,

output reg end_receiving ,
output reg end_transmitting ,
output [15:0] data_out_com ,
output reg [15:0] addr_com ,
 output data_to_pc ,
 output reg en_com);

 reg tx_enable ;
 wire tx_busy;
 wire rx_ready ;

 reg [15:0] trans_addr =16'b0;
 reg [15:0] receive_addr =16'b0;
 reg [15:0] next_trans_addr =16'b0;
 reg [15:0] next_receive_addr =16'b0;

 reg [3:0] present = 4'b0000;
 reg [3:0] next = 4'b0000;

 initial begin
 end_transmitting = 0;
 end_receiving = 0;
 end


 parameter
 txrx_idle = 4'b0000,
 tx_send = 4'b0001,
 tx_send_wait = 4'b0010,
 tx_update = 4'b0011,
 tx_end = 4'b0100,


 rx_start = 4'b0101,
 rx_get = 4'b0110,
 rx_update = 4'b0111,
 rx_end = 4'b1000;

 always @(posedge clock)
 begin
 trans_addr <= next_trans_addr ;
 receive_addr <= next_receive_addr ;
 end

 always @(posedge clock) begin
 case(next)
 txrx_idle : begin
 tx_enable = 0;


 addr_com <= 16'b0;
 en_com <=0;
 next_trans_addr <= 16'b0;
 next_receive_addr <= 16'b0;

 if (status == 2'b10)
 next <= tx_send;
 else if (status == 2'b00)
 next<= rx_start ; //rx_start; //
 else
 next<= txrx_idle ;
 end

 tx_send:begin
 tx_enable = 1;
```

```

end_receiving = 0;
end_transmitting = 0;
addr_com <= trans_addr ;
en_com <=0;
next <= tx_send_wait ;
end

tx_send_wait :begin
if (~tx_busy) begin
next <= tx_update ;
end

end

tx_update :begin

tx_enable <=0;
if (trans_addr > 16128)begin
end_transmitting <= 1;
next<= tx_end;
next_trans_addr <= trans_addr ;
next_receive_addr <= receive_addr ;
end


else begin
end_transmitting <= 0;
next<=tx_send;
next_trans_addr <= trans_addr + 16'b1;
next_receive_addr <= receive_addr ;
end

end


tx_end: begin

tx_enable = 0;
end_receiving = 1;
end_transmitting = 1;
addr_com <= 16'b0;
en_com <=0;
next_trans_addr <= 16'b0;
next_receive_addr <= 16'b0;

next<=tx_end;
end

rx_start : begin

tx_enable = 0;
end_receiving = 0;
end_transmitting = 0;
addr_com <= 16'b0;
en_com <=0;
next_trans_addr <= trans_addr ;
next_receive_addr <= receive_addr ;

next<=rx_get;
end

```

```

rx_get:begin

tx_enable = 0;
end_receiving = 0;
end_transmitting = 0;
addr_com <= receive_addr ;

if (rx_ready ) begin
en_com <=1;

if (receive_addr == 65535) begin
end_receiving <= 1;
next<= rx_end;
end

else begin
next <= rx_update ;
end
end

else begin
next <= rx_get;
end


end

rx_update : begin

next_receive_addr <= receive_addr + 16'b1;
next_trans_addr <= trans_addr ;
end_receiving <= 0;
next<=rx_get;
end

rx_end: begin
tx_enable = 0;
end_receiving = 1;
end_transmitting = 0;
addr_com <= 16'b0;
en_com <=0;
next_trans_addr <= 16'b0;
next_receive_addr <= 16'b0;

next<=txrx_idle ;
end

endcase

end

async_transmitter #(25000000 ,115200) tx( .clk(clock),
.TxD_start (tx_enable ),
.TxD_data (data_in_com ),
.TxD(data_to_pc ),
.TxD_busy (tx_busy)
);

async_receiver #(25000000 ,115200) rx( .clk(clock),
.RxD(data_from_pc ),
.RxD_data_ready (rx_ready ),
.RxD_data (data_out_com )
);

endmodule
```