EN3030- CIRCUITS AND SYSTEMS DESIGN

FPGA BASED PROCESSOR DESIGN
FINAL REPORT

Group members

Shan Anjana	160027U
Nipuna Leelarathne	160348E
Isuru Rajapakshe	160505J
Mevan Wijewardena	160716G

Supervisor
Dr. Jayathu Samarawickrama

This report is submitted in partial fulfillment of the requirements
for the module EN 3030: Circuits and Systems Design.

ABSTRACT

Custom processors are widely used in the modern technical world. Even if a general CPU in a computer could perform the tasks of a custom processor the speed and the power efficiency of custom processors have marked their significance in electronics.

In this report we describe our method of designing a custom processor for image processing. We used Field-Programmable Gate Array (FPGA) for our task. The report contains detailed descriptions on our implementation, design procedure and the functionality of our processor.

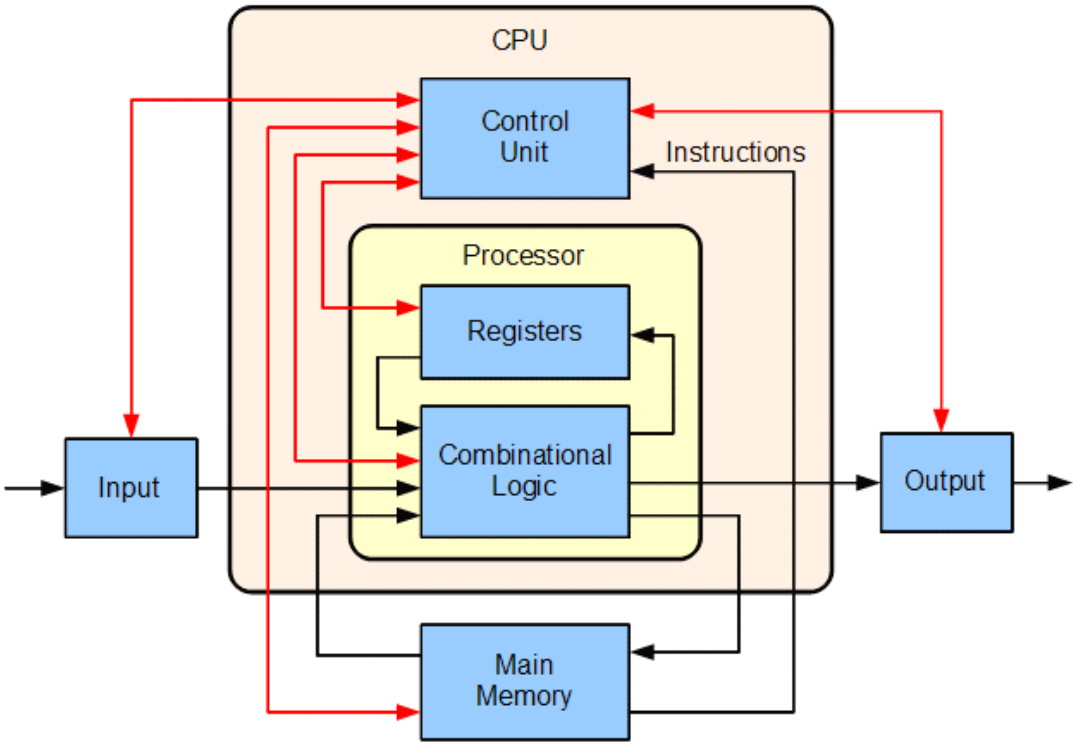
TABLE OF CONTENTS

INTRODUCTION

In this project we designed a custom processor to down sample a given image. The processor is implemented in Verilog which is a Hardware Description Language. The implementation of the processor is done using Quartz Prime Lite Edition along with Altera DE2-115 Education and Development board with Cyclone IV FPGA.

About CPUs

CPU carries out the instructions of a given computer program by performing the arithmetic, logic, control, input and output operations which are specified by the instructions in the program.



Block diagram of a CPU

PROBLEM STATEMENT

The task of this project is to down sample a given image. The down sampling factor is two and following are some of the parameters,

- Down sampling factor- 2
- Image dimensions- 500×500 pixels
- Color/ Black and white

OUR SOLUTION

We came up with a solution to the above problem using the tools mentioned in the INTRODUCTION. Before moving onto implementing the processor we decided the basic steps in the algorithm we were to implement. Accordingly we listed the basic steps to achieve out task.

1. Transmitting the image from PC to FPGA.
2. Storing the image in FPGA.
3. Processing the image / down sampling and storing the down sampled image.
4. Sending the down sampled image back to the PC and displaying it.

We identified above as the steps which could be handled separately and we divided them among our members.

For each of the above steps we identified many solutions.

Transmitting the image from PC to FPGA- The first decision we had to make was the language we are using for handling the image, receiving and sending tasks from the PC side. We had two options, Python and MATLAB. Since Python is open source and PySerial package offers easier ways for serial communication we preferred python over MATLAB.

Storing the image in FPGA- Again we had several solutions. First was to store the whole image in the FPGA first and then processing it. The next was to obtain the required pixels from the PC, process them and send them back to the computer. The second method consumes much less storage space in FPGA compared to the first. But it is implementation wise complex. Moreover we will have to transmit the same pixel multiple times for processing. Compared to the other operations transmission takes much longer time. Hence the second method is more time consuming. Hence we adopted the first.

Processing the image / down sampling and storing the down sampled image.

Sending the down sampled image back to the PC and displaying it - For these two steps (3 and 4) we didn't come up with multiple solutions.

Above mentioned are just the basic summary of different approaches we took for each process. Our method will be explained in detail.

DETAILED DESCRIPTION OF THE PROCESS

Transmitting and storing the image

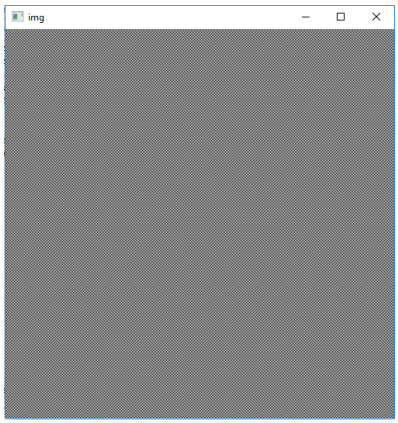
We will consider the case of down sampling a color image so that steps for black and white images can be described similarly. First the image is loaded in python using opencv package. Then each one of the color planes is taken. First all the 25000(500×500) pixels in that plane are sent to the FPGA. We use UART for the communication process and the package we use in python is PySerial. PySerial allows the transmission of bytes and hence conversion of pixel values to bit stream is not necessary.

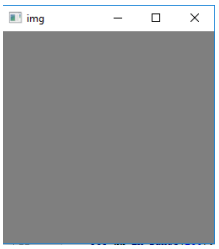
The transmitter receiver unit in the processor receives the data and it stores each pixel in its memory (For this purpose the memory unit of the processor is enabled).

Processing the stored data

Processing the data is mainly done under three steps.

1. Filtering the image to smoothen the image and to remove random noise
   This is necessary before down sampling. Following is a gray scale image which comprises of alternate 255's and 0's. Hence it is gray in color to a human eye. But if we down sample this image without filtering the resulting image will either be completely black or white. The next image is obtained by first filtering and then down sampling the first image**.**





2. After filtering the next step is down sampling the image. Since our goal is to down sample by a factor of two and since we already filtered the image, in this step we simply chose alternate bits.
3. Final step is storing the down sampled image. In step 1 we did not calculate the filtered value of each pixel. Hence storing the final image became much easier and efficient. The down sampling algorithm will be explained later.

After the above three steps we have the down sampled image stored in the memory unit.

Sending the down sampled image back to the PC.

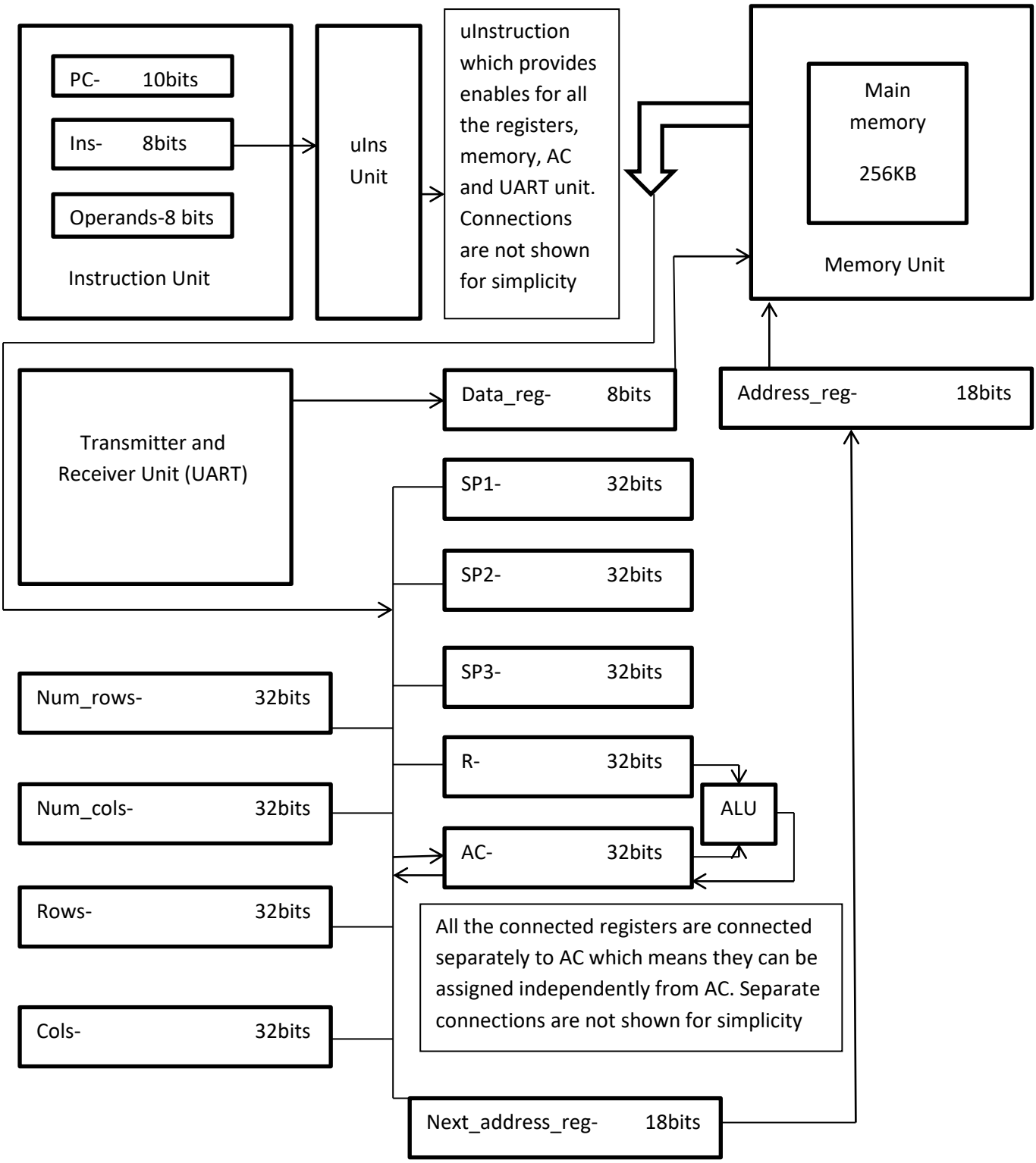
In this step we again use the UART transmitter unit of the processor to send the down sampled image back to the PC.

The above steps are repeated three times for the three color planes of the image and then the final image is reproduced and displayed in the PC using opencv.

IMPORTANT FACTS ABOUT OUR DESIGN

- We used 50MHz built in clock for synchronization
- Most of the time is taken for transmission of the image (UART). In fact a very small time is taken for processing.
- The memory unit has enough memory to store a 512×512 pixel image (256kB) at once.

THE PROCESSOR DATAPATH



THE INSTRUCTION SET

To suit our purpose we created our own instruction set which comprised of 39 instructions. Following are the instructions and there description.

No	OpCode		Instruction Code/No operands	Instruction Description
1	8'b	0	NOP /0	No operation
2	8'b	1	RCV /3	Receive t number of bytes where t is specified by the operands
3	8'b	10	TRN /3	Transmit t number of bytes where t is specified by the operands
4	8'b	11	LOADAC /3	Load the byte in the memory location t (t specified by operands) to AC
5	8'b	100	STOREAC /3	Store the byte in AC to the memory location t (t specified by operands)
6	8'b	101	SHIFTR8 /0	Shift AC by one byte to the right
7	8'b	110	SHIFTR1 /0	Shift AC by one bit to the right
8	8'b	111	SHIFTL8 /0	Shift AC by one byte to the left
9	8'b	1000	SHIFTL1 0	Shift AC by one bit to the left
10	8'b	1001	RAC /0	R<=AC
11	8'b	1010	ROWAC /0	Rows<=AC
12	8'b	1011	ACROW /0	AC<=Rows
13	8'b	1100	COLAC /0	Cols<=AC
14	8'b	1101	ACCOL /0	AC<=Cols
15	8'b	1110	ACNCOL /0	AC<=Ncols
16	8'b	1111	NCOLAC /0	Ncols<=AC
17	8'b	10000	ACNROW /0	AC<=Nrows
18	8'b	10001	NROWAC /0	Nrows<=AC
19	8'b	10010	ACNXTADD /0	AC<=NxtAddress
20	8'b	10011	STONXTADD /0	Store the last byte of AC to the memory location specified by NxtAddress
21	8'b	10100	LOADNXTADD /0	Load the byte in the memory location specified by NextAddress to AC
22	8'b	10101	ACXOR /0	AC<=AC xor R
23	8'b	10110	ACR /0	AC<=R
24	8'b	10111	ACINC /0	AC<=AC+1
25	8'b	11000	ACDEC /0	AC<=AC-1
26	8'b	11001	ACZERO /0	AC<=0
27	8'b	11010	JMPNZ /2	if(Z=0) Jump to instruction t specified by operand
28	8'b	11011	JMP /2	Jump to instruction t specified by operands

29	8'b	11100	JMPZ /2	if(Z=1) Jump to instruction t specified by operand
30	8'b	11101	ACADD /0	AC<=AC+R
31	8'b	11110	ACSUB /0	AC<=AC-R
32	8'b	11111	SP1AC /0	SP1<=AC
33	8'b	100000	ACSP1 /0	AC<=SP1
34	8'b	100001	NXTADDAC /0	NextAddress<=AC
35	8'b	100010	ACSP2 /0	AC<=SP2
36	8'b	100011	ACSP3 /0	AC<=SP3
37	8'b	100100	SP2AC /0	SP2<=AC
38	8'b	100101	SP3AC /0	SP3<=AC
39	8'b	100110	STOP /0	stop process

Here most of the instructions take single clock cycle except for a few. Instructions 2, 3, 4, 5, 20, 21, 27, 28 and 29 are the only ones which consume more than one clock cycle.

RCV- This instruction consists of 4 bytes three of which are operands. This instruction is to receive Ω number of bytes from the PC where Ω is specified by the operands. Due to the limitations of the memory Ω can go up to 512×512. We designed a careful protocol for transmission. In order to minimize errors we are first sending 11110000 from UART to PC. PC is only allowed to transmit after receiving this. This is why our transmission is bit slower. But on the other hand it ensures error free communication with perfect synchronization between PC and processor.

TRN- This is similar to RCV but we do not specify a specific protocol here. Once the processor starts transmitting the PC should be ready to receive from the other side.

Other instructions do not have specific details and the explanation in the table explain their functionality.

MICRO INSTRUCTIONS

We implemented a structure similar to CISC architecture. Hence each of the above instruction comprises of several Micro-Instructions. Following is the Micro-Instruction format.

SP3 28	SP2 27	IU 26	25	24	SP1 23	Tr 22	R_Next 21	20	T_Next 19	18	Baud rate 17	RAM 16	15

NextAddress 14	AC 13	12	11	10	9	Address_reg 8	7	R 6	Rows 5	Cols 4	Num_rows 3	Num_cols 2	Data_reg 1	0

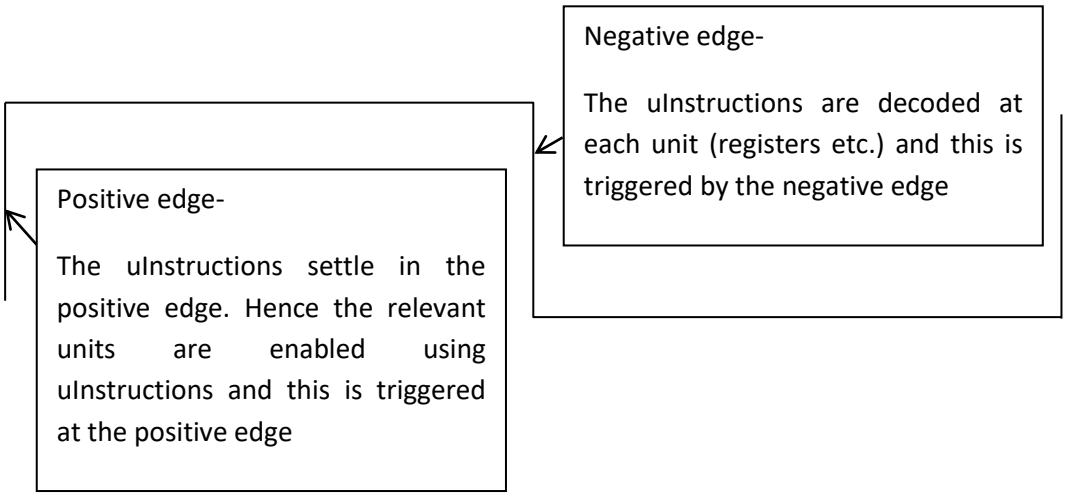
A single micro instruction is 29 bits and shown above are each of the 29 bits. A Microinstruction is a state of the processor in a given clock cycle. It is important to note that a Micro-Instruction is a single clock cycle operation. A Microinstruction defines the set of enables to each of the registers and units including Instruction Unit and Memory Unit.

A single instruction is a collection of Microinstructions. For example STONXTADD comprises of,

```
1. 29'b00000000000010000000010000001
2. 29'b00000000000000000000000000000
3. 29'b00000000000000000000000000000
4. 29'b00000000000000000000000000000
```

HOW THE CLOCK IS UTILIZED IN OUR DESIGN

We used 50MHz clock throughout our design. Only for transmitter and receiver we had to sample the clock through a different module.



THE MAIN MEMORY

We used the altsyncram megafunction to create an SDRAM. One byte of memory can be stored in each location and there are 262,144 such locations making it easily possible to process a 500×500 image.

The memory we used is synchronous. In case of a write operation when the enables are provided at the positive edge of one clock cycle the value on the data register is written to the location specified by the address register at the positive edge of the next clock cycle.

In case of a read operation when the enables are provided at the positive edge of one clock cycle the value on the memory location specified by the address register is written to the q bus(data bus) at the positive edge of the next clock cycle. For safety we are providing one more clock cycle for memory read and write operations.

There are four instructions for memory operations. Namely STOREC, LOADAC, STONXTADD, LOADNXTADD. NextAddress register is used when a sequence of bytes are stored in or retrieved from the memory. All instructions are used to store or update the value of AC but NXTADD instructions use the NxtAddress register as the address register whereas in other two the address can be provided as an operand.

THE TRANSMITTER AND THE RECEIVER

Transmission is done using UART protocol. The transmission is governed by the instruction TRN and the receiver by RCV the functionalities of which are explained above. Even if all the other modules use the 50MHz clock for transmitter and the receiver we sampled the clock.

For receiver the 50MHz clock is divided by 2^5 = 32 whereas for transmitter it is divided by 2^9= 512. We are using the baud rate 115200 which means there are 5000000/115200 =435 clocks for one bit. Due to this when the TRN instruction is called, when the transmitter clock enabled the transmitter goes through the four states.

- TX_STATE_IDLE
- TX_STATE_START
- TX_STATE_DATA
- TX_STATE_STOP

AT the stage TX_STATE_DATA bits are transmitted during each transmitter clock cycle (which is 1/512 of the processor clock).

The receiver's functionality is different. When the RCV instruction is called, when the receiver clock is enabled receiver goes through four states again.

- RX_STATE_START
- RX_STATE_DATA
- RX_STATE_STOP
- RX_STATE_IDLE

In RX_STATE_DATA we do not receive a bit in each receiver clock (which is 1/32 of the processor clock). If this was to be done the baud rates of the two sides will not match. Hence one bit is received for 16 receiver clock cycles (Now 16×32=512 clocks per bit which will match with the baud rate of 435 clocks per bit). Moreover for the error free receiving of the bit, the value of the receiver after the 8^{th} receiver clock cycle is considered.

The objective of the TRN and RCV instructions is to transmit or receive a given number of bytes to or from the PC which will be necessary when transferring an image to or from the PC. RX_TX registers are used to keep track of the number of bytes transferred.

THE INSTRUCTION UNIT

The instruction unit consists of an instruction memory, PC, Instruction register and the operands register. The processor goes through the three cycles fetch, decode and execute. The instruction unit comes into play in the fetch cycle. Once an instruction is completed a new instruction is fetched (Instruction indicated by PC) and the PC is incremented by one. In Jump instructions (JMP, JMPZ, JMPNZ) the value of the PC is updated during the execution cycle of the instruction. In fact these are the only instructions in which the PC can be controlled externally rather than being increased by one. JMP is unconditional jump which is used for loops and JMZ and JMPNZ are conditional jumps which are used for if clauses.

Instruction memory stores all the instructions to be executed in order. Our instruction memory is 1024 bytes in size which means our processor can handle any task with less than 1024 instructions and which could be done using our instruction set.

Some of the instruction memory locations may not be actual Instructions but can be operands of another instruction. There is an instruction register to store the currently executing instruction and a separate operands register to store such operands.

THE GENERAL PURPOSE AND SPECIAL PURPOSE REGISTERS

Due to the implementation of our processor we have much general purpose and special purpose registers. AC is the accumulator which is responsible for all the ALU instructions. There are 3 registers SP1, SP2, SP3 which are 32 bits in size and can be loaded directly to/ from AC. There are also 32 bit registers Num_cols, Num_rows, Rows and Cols which are again same as the previous ones. The names are used to distinguish there purpose which is to handle row and column operations during image processing. R is also like the above registers except for which it can combine with AC in ALU operations (ACADD, ACSUB etc.)

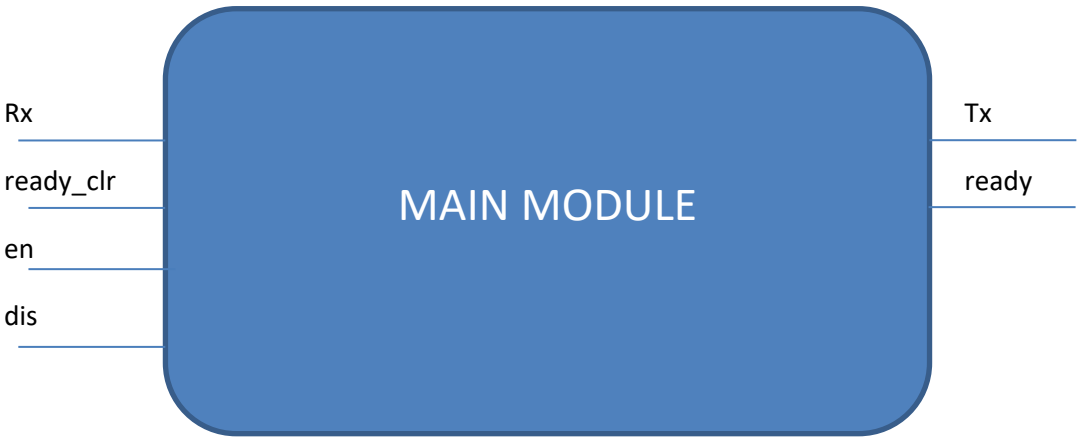
Apart from the above we implemented PC, Instruction register, Address register, Next_address register, RX_TX registers whose functionalities are explained in previous sections.

ALU OPERATIONS

5'b1	AC<=R_to_AC	Assigns R to AC
5'b10010	AC<=ncols_to_AC	Assigns Num_cols to AC
5'b10	AC<=nrows_to_AC	Assigns Num_rows to AC
5'b11	AC<=cols_to_AC	Assigns cols to AC
5'b100	AC<=rows_to_AC	Assigns rows to AC
5'b101	AC<=(AC<<8)	Shifts AC to left by one byte
5'b110	AC<=(AC>>8)	Shifts AC to right by one byte
5'b111	AC<=(AC<<1)	Shifts AC to left by one bit
5'b1000	AC<=(AC>>1)	Shifts AC to right by one byte
5'b1001	AC<=AC+R_to_AC	Adds R to AC and stores the result in AC
5'b1010	AC<=AC^R_to_AC	Bitwise XORS R to AC and stores the result in AC
5'b1011	AC<=0	Sets AC to zero
5'b1100	AC<=AC+1	Increments AC by 1
5'b1101	AC<=AC-1	Decrements AC by 1
5'b1110	AC[7:0]<=memory_to_AC	Assigns the value in memory bus to lower 8 bits of AC
5'b1111	AC[7:0]<=Ins_to_AC	Assigns the value in instruction register to Lower 8 bits of AC
5'b10000	AC<=SP1_to_AC	Assigns SP1 to AC
5'b10001	AC[17:0]<=NA_to_AC	Assigns NextAddress reg to lower 18 bits of AC
5'b10011	AC<=AC-R_to_AC	Subtracts R from AC and stores in AC
5'b10100	AC<=SP2_to_AC	Assigns SP2 to AC
5'b10101	AC<=SP3_to_AC	Assigns SP3 to AC

THE DIFFERENT MODULES IN OUR DESIGN

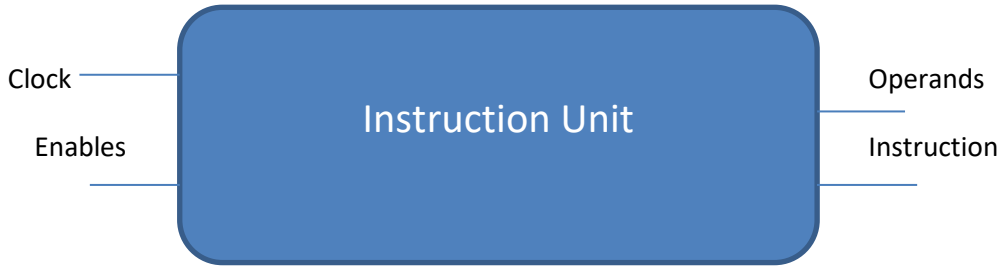
Main module



This is the main module which combines all the other modules such as the Instruction Unit, Memory Unit and the registers. This contains,

- AC- Connects directly to ALU and performs all the arithmetic operations. There are many instructions to control this register. For example we can store or load directly into this register from the memory.
- R- This is the other register which is directly connected to ALU.
- SP1, SP2, SP3- special purpose registers which are directly connected to AC through separate buses
- Rows, Cols, Nrows, Ncols- These registers are specifically designed to keep track of row numbers and column numbers of images in image processing.
  (All the above mentioned registers are 32 bit)
- NextAddress- Used to keep track of the next address when addresses are accessed sequentially.(This is 18 bit)
- Address reg- Used for memory operations. (18 bits)
- Data reg- Used for memory operations. (8 bits)

Instruction Unit



This unit handles all the instruction related tasks. It takes the enables as the input and sends out the Instruction or the Operands depending on whether the respective location of the Instruction memory is an Instruction or an operand. The module contains,

- PC (10 bits) - Program counter. Stores the address of the next instruction to be performed.
- Instruction memory (1kB) Stores the instructions to be performed.

Memory Unit

This is the main memory inside the processor. The memory is 256kB in size. When the address is written into the address bus and when the write is enabled the value in the Data bus is written to the respective memory location. Similarly when read is enabled the value in the memory location is read to the output bus.

Transmitter

This is the module which is used to transmit data to the PC. When transmitter clock is enabled the data in Data_in is sent to Tx using UART communication protocol. This data is received by the PC. The instruction TRN is used to transmit data to the PC. Ω number of bytes can be transmitted where Ω is specified by the operands.

Receiver

This module is used to receive data from the PC. When receiver clock is enabled this module receives data from the PC from Rx and once one byte arrives it is written into data. The instruction RCV is used to receive data. Ω number of bytes can be received where Ω is specified by the operands.

Baud Rate Controller

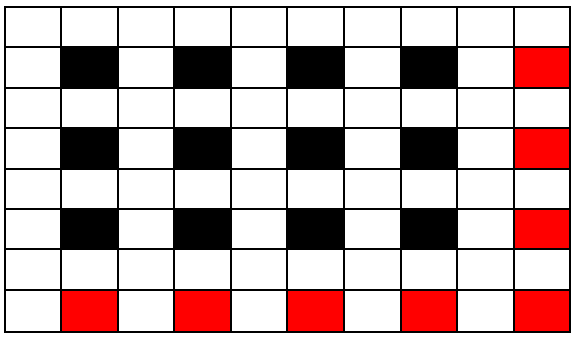
This module is used to provide the clock enable for the transmitter and receiver modules. Since they should enable at different times and should operate much slower than the original clock rate this module enables the clock in two different ways.

THE DOWNSAMPLING ALGORITHM

A sketch of the algorithm we used for downsampling is provided at the beginning. In this section we provide a detailed description of the algorithm we used for downsampling.

After loading the image pixels are stored sequentially in the memory.

1) Filtering the image and downsampling.



Consider the above image (8×10) for an example. On each of the black squares we used a Gaussian filer. Consider the Gaussian kernel.

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

We used a 3×3 kernel for filtering. The shown filter is a close approximation of a 3×3 Gaussian kernel.

$$f(x,y) = ce^{\frac{(x^2+y^2)}{2\Omega^2}}$$

Index the above image staring from the top left corner square as (0, 0) and top right as (0, 9). Now first we use the Gaussian filter on (1, 1) and store it in the location where we had (0, 0). This won't be a problem since we will never need (0, 0) again. Next we use Gaussian filter centering (1, 3) and store the result in (0, 1). We cannot use Gaussian filter on (1, 9). Hence we use that pixel as it is and store it in (0, 4). Next we use a Gaussian filter centered at (3, 1) and store it at (1, 0). The only exception is we do not use Gaussian filter on red squares and we use them as they are. This way we won't we do not have to downsample again. The downsampled image is stored sequentially in the memory now. Now we just have to transmit the image to PC.

Doing the above for all the three color planes and reconstructing the image in opencv in python we can display the downsampled image in python.

The method we used is both efficient in memory and time.

- We are only using filters on the required pixels. This saves time.
- We are not using extra storage to store the downsampled image. This way we maximized the size of the image we chose.
- We did not have to use division due to the nature of the kernel. It contains powers of two as divisors which could be handled using shift operations which is much faster.

THE ASSEMBLY CODE FOR DOWNSAMPING

```
ACZERO
SP1AC
NXTADDAC
RCV
0
0
100
ACZERO
NXTADDAC
ACZERO
LOADNXTADD
SHIFTR8
RAC
ACZERO
ACINC
NXTADDAC
ACR
LOADNXTADD
NCOLAC
ACZERO
ACINC
SHIFTR1
NXTADDAC
ACZERO
LOADNXTADD
SHIFTR8
RAC
ACZERO
ACINC
ACINC
ACINC
NXTADDAC
ACR
LOADNXTADD
NROWAC
ACZERO
STOREAC
11
11010000
10010001
ACZERO
ACINC
```

```
ACINC
ACINC
RAC
LOADAC
11
11010000
10010001
SP2AC
ACXOR
JMPZ
1
101011
ACSP2
ACINC
STOREAC
11
11010000
10010001
ACZERO
ACINC
ROWAC
COLAC
ACZERO
SP1AC
SP2AC
SP3AC
NXTADDAC
RCV
11
11010000
10010000
ACZERO
NXTADDAC
ACNXTADD
ACINC
RAC
ACNCOL
ACADD
NXTADDAC
ACZERO
SP2AC
```

ACNROW
ACDEC
RAC
ACROW
ACXOR
JMPZ
0
11111111
ACNCOL
ACDEC
RAC
ACCOL
ACXOR
JMPZ
0
11100010
ACNCOL
RAC
ACZERO
ACNXTADD
ACDEC
ACSUB
NXTADDAC
ACZERO
LOADNXTADD
SP1AC
ACZERO
ACNXTADD
ACINC
NXTADDAC
ACZERO
LOADNXTADD
SHIFTR1
RAC
ACSP1
ACADD
SP1AC
ACZERO
ACNXTADD
ACINC
NXTADDAC
ACZERO

LOADNXTADD
RAC
ACSP1
ACADD
SP1AC
ACZERO
ACNXTADD
RAC
ACNCOL
ACADD
NXTADDAC
ACZERO
LOADNXTADD
SHIFTR1
RAC
ACSP1
ACADD
SP1AC
ACZERO
ACNXTADD
ACDEC
NXTADDAC
ACZERO
LOADNXTADD
SHIFTR1
SHIFTR1
RAC
ACSP1
ACADD
SP1AC
ACZERO
ACNXTADD
ACDEC
NXTADDAC
ACZERO
LOADNXTADD
SHIFTR1
RAC
ACSP1
ACADD
SP1AC
ACZERO

ACNXTADD
RAC
ACNCOL
ACADD
NXTADDAC
ACZERO
LOADNXTADD
RAC
ACSP1
ACADD
SP1AC
ACZERO
ACNXTADD
ACINC
NXTADDAC
ACZERO
LOADNXTADD
SHIFTR1
RAC
ACSP1
ACADD
SP1AC
ACZERO
ACNXTADD
ACINC
NXTADDAC
ACZERO
LOADNXTADD
RAC
ACSP1
ACADD
SHIFTL1
SHIFTL1
SHIFTL1
SHIFTL1
SP1AC
ACZERO
ACNXTADD
SP3AC
ACSP2
NXTADDAC
ACINC

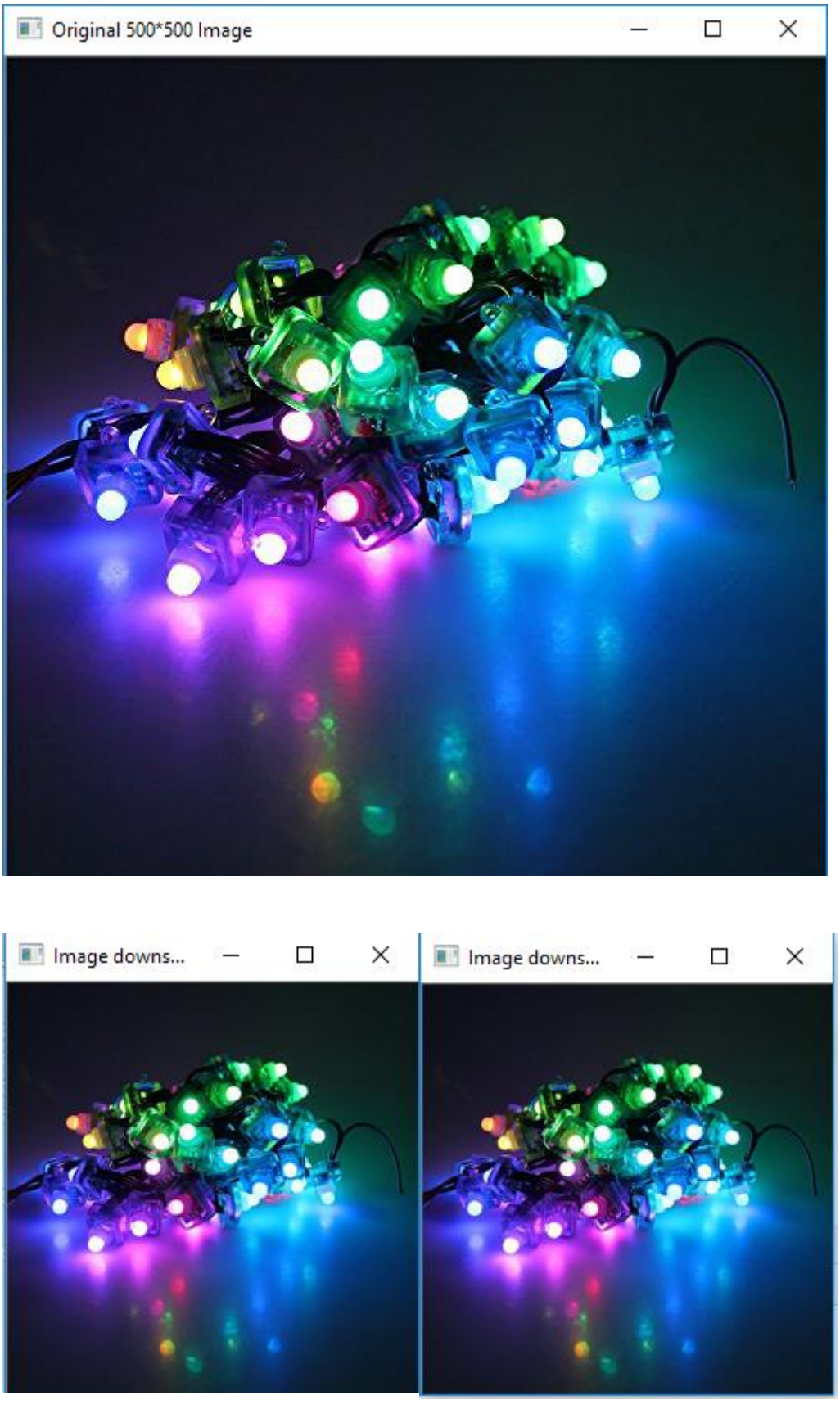
SP2AC
ACSP1
STONXTADD
ACNCOL
RAC
ACZERO
ACSP3
ACSUB
ACINC
NXTADDAC
ACCOL
ACINC
ACINC
COLAC
JMP
0
1011011
ACZERO
ACINC
COLAC
ACROW
ACINC
ACINC
ROWAC
ACZERO
LOADNXTADD
RAC
ACZERO
ACNXTADD
SP3AC
ACSP2
NXTADDAC
ACINC
SP2AC
ACR
STONXTADD
ACNCOL
RAC
ACSP3
ACADD
ACINC
ACINC

NXTADDAC
JMP
0
1010011
ACZERO
ACINC
COLAC
ACCOL
RAC
ACNCOL
ACINC
ACSUB
JMPZ
1
100010
ACZERO
LOADNXTADD
RAC
ACZERO
ACNXTADD
SP3AC
ACSP2
NXTADDAC
ACINC
SP2AC

ACR
STONXTADD
ACZERO
ACSP3
ACINC
ACINC
NXTADDAC
ACCOL
ACINC
ACINC
COLAC
JMP
1
10
ACZERO
NXTADDAC
TRN
0
11110100
100100
JMP
0
101000
STOP

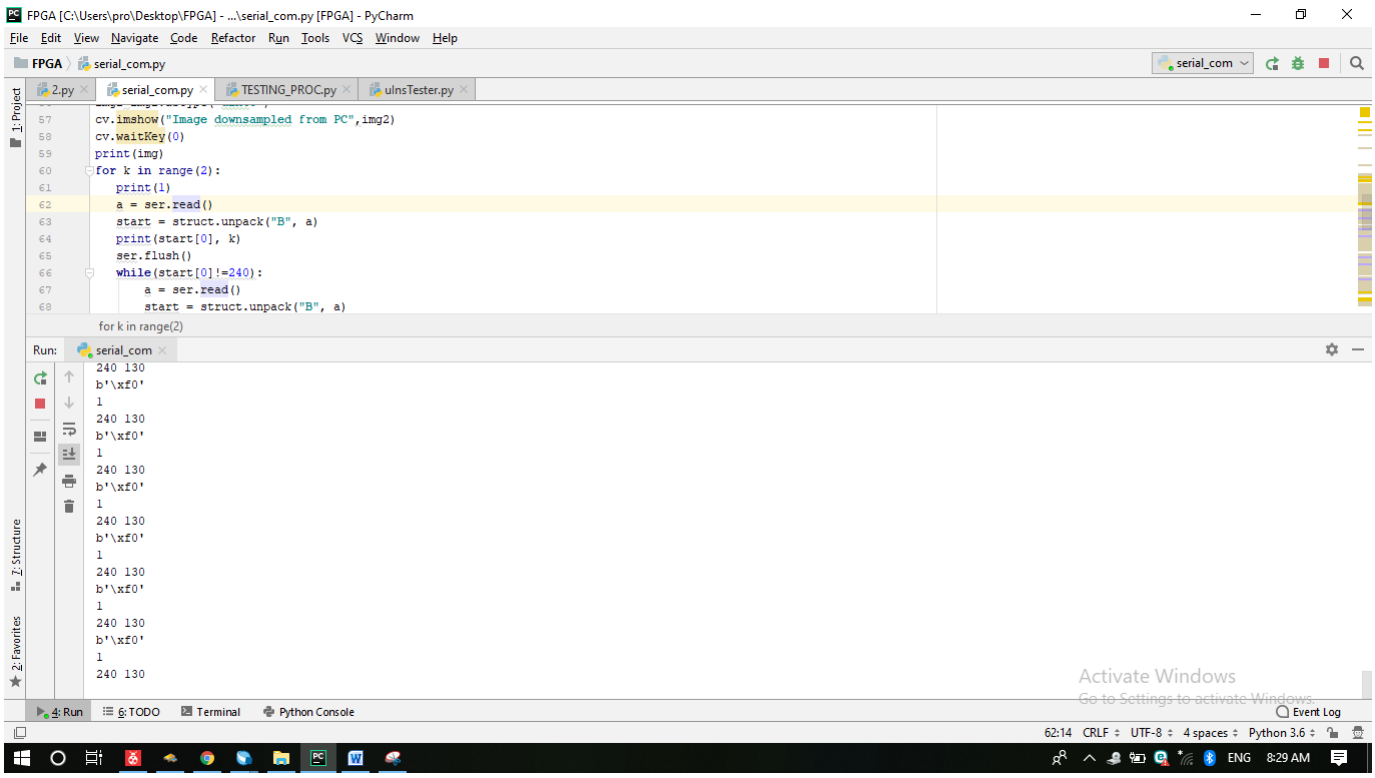
THE RESULTS OF OUR DOWNSAMPLING ALGORITHM

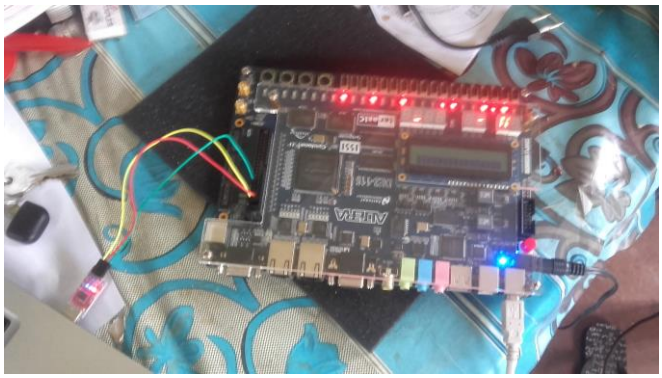
Following are the results after downsampling a 500×500 image from our processor.



Original image top, Image downsampled from PC bottom left and Image downsampled from FPGA bottom right

It is notable that most of the time for processing is taken for the transmission of the image. To transmit all the three color planes and to process them it took around 7 minutes but for processing it took only 1 second. This is due to the protocol we used in transmitting. In PC to downsample the image it took around 3 seconds.





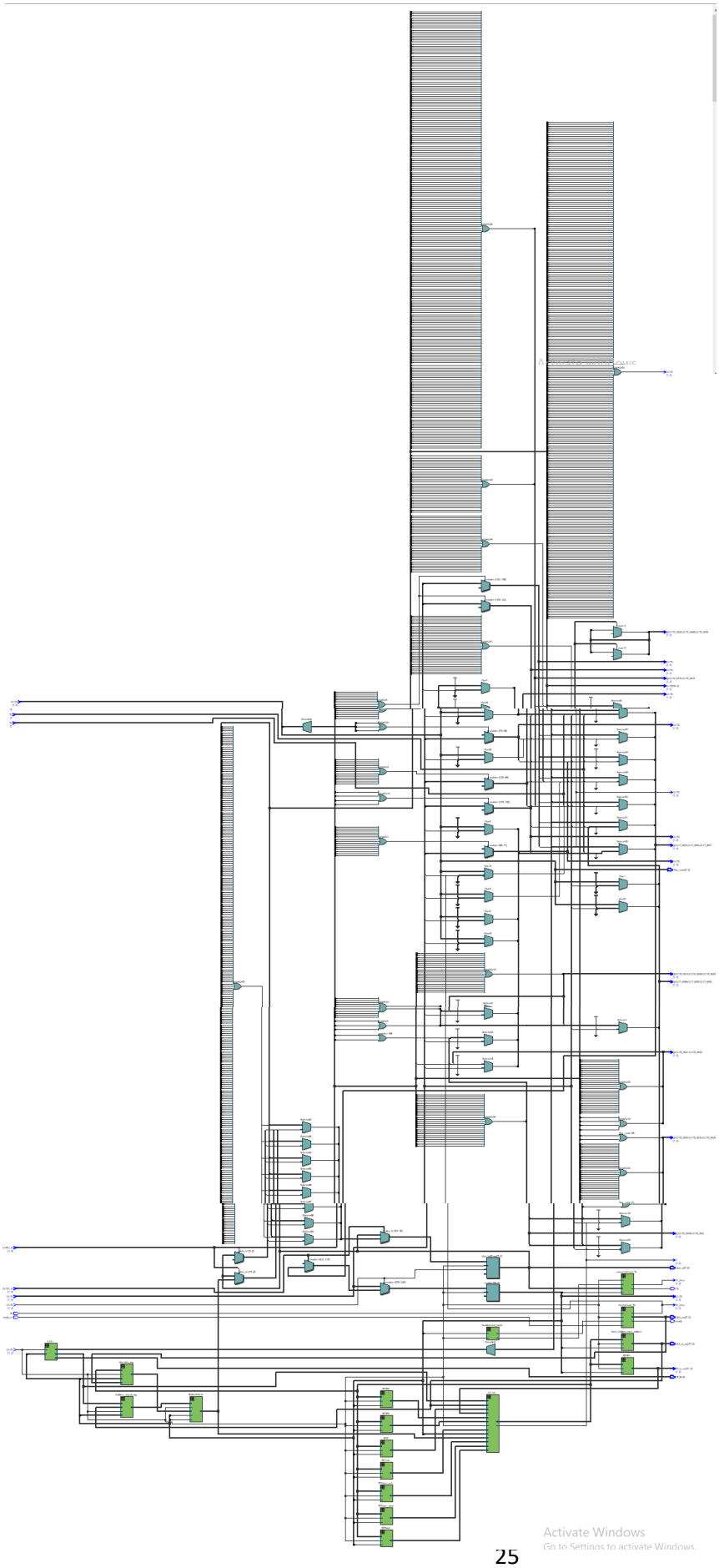
```
Processing time: 1.2815794944763184
computer time 2.989008665084839
Error in processing 0

Process finished with exit code 0
```

Screenshot of python IDE during transmission top, Picture of the FPGA during transmission bottom left and statistics of processing bottom right

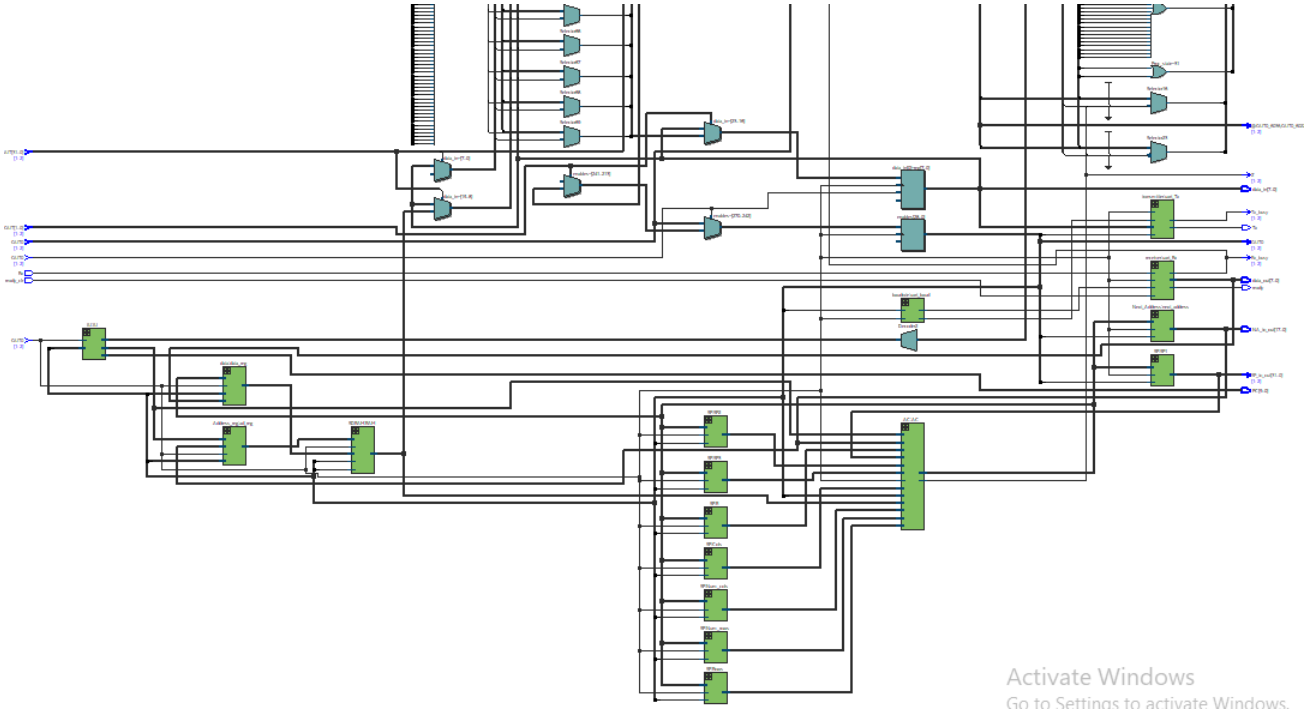
Observe the stats of the processing. The FPGA was much faster in processing the image. More importantly the error between the image downsampled from the PC and from the FPGA is zero which means they are identical. Here we used

$$\text{Error} = \sum_{over\ all\ the\ pixels\ of\ the\ downsampled\ images} |\text{python pixel value} - \text{FPGA processor donsampled image pixel value}|$$

THE COMPLETE RTL VIEW OF THE PROCESSOR

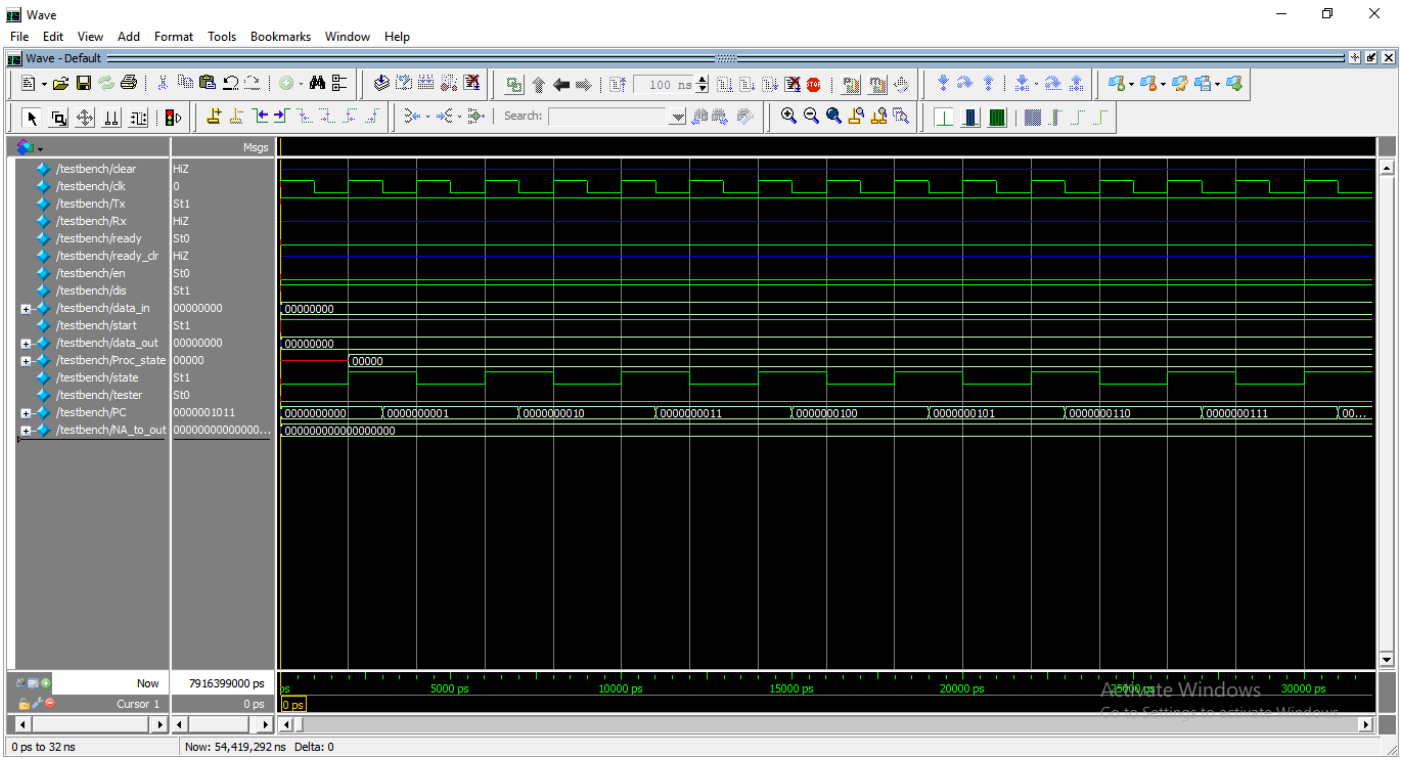


THE VIEW OF THE DIFFERENT MODULES



Shown in green is the RTL view of different modules

TIMING ANALYSIS FOR A SAMPLE PROGRAMME



The sample program is given below,

ACZERO
NOP
ACINC
ACINC
RAC
ACZERO
ACINC
ACINC
SHIFTL1
ACADD

Due to the complexity of the original program we did not use it for timing analysis. Instead we used the above program for it. Diagram is a simulation of the above program. The test bench used is provided in the appendix. We can see how the PC and the other signals change with the clock.

THE COMPILATION REPORT IN QUARTUS PRIME

Flow Status	Successful - Mon May 27 08:03:57 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	uart
Top-level Entity Name	uart
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2,197 / 114,480 (2 %)
Total registers	537
Total pins	60 / 529 (11 %)
Total virtual pins	0
Total memory bits	2,097,152 / 3,981,312 (53 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

REFERENCES

- Ecs.umass.edu. (2019). [online] Available at: http://www.ecs.umass.edu/ece354/ECE354HomePageFiles/Labs_files/Quartus_II_Introduction.pdf [Accessed 28 May 2019].
- Asic-world.com. (2019). *Verilog Tutorial*. [online] Available at: http://www.asic-world.com/verilog/veritut.html [Accessed 28 May 2019].
- Alchitry. (2019). *Basic CPU*. [online] Available at: https://alchitry.com/blogs/tutorials/basic-cpu [Accessed 28 May 2019].

APPENDIX

Python codes

1) Serial communication

```
 import serial
import sys
import glob
import serial
import struct
import cv2 as cv
import time
import numpy as np
import time
def serial_ports():
    """ Lists serial port names

        :raises EnvironmentError:
            On unsupported or unknown platforms
        :returns:
            A list of the serial ports available on the system
    """
    if sys.platform.startswith('win'):
        ports = ['COM%s' % (i + 1) for i in range(256)]
    elif sys.platform.startswith('linux') or
sys.platform.startswith('cygwin'):
        # this excludes your current terminal "/dev/tty"
        ports = glob.glob('/dev/tty[A-Za-z]*')
    elif sys.platform.startswith('darwin'):
        ports = glob.glob('/dev/tty.*')
    else:
        raise EnvironmentError('Unsupported platform')

    result = []
    for port in ports:
        try:
            s = serial.Serial(port)
            s.close()
            result.append(port)
        except (OSError, serial.SerialException):
            pass
    return result
#print(serial_ports())
#initialize serial object
ser = serial.Serial('COM30',115200,timeout=10,bytesize=8,stopbits=1)
#loading the image
img1=cv.imread("7.jpg")
img=img1
#dispalying the original image
cv.imshow("Original 500*500 Image",img)
cv.waitKey(0)
print(img.shape)
#downsampling the original image using pyhton
img2=np.zeros((250,250,3))
t3=time.time()
for i in range(3):
```

```
  for t in range(1,500,2):
    for tt in range(1,500,2):
      if(t!=499 and tt!=499):
        g=(img[t,tt,i]*4)+(img[t-1,tt,i]*2)+(img[t,tt-
1,i]*2)+(img[t+1,tt,i]*2)+(img[t,tt+1,i]*2)+(img[t-1,tt-1,i])+(img[t-
1,tt+1,i])+(img[t+1,tt-1,i])+(img[t+1,tt+1,i])
        g=g//16
        img2[t//2,tt//2,i]=g
      else:
        img2[t//2,tt//2,i]=img[t,tt,i]
t4=time.time()
img2=img2.astype('uint8')
# dispalying the image downsampled from python
cv.imshow("Image downsampled from PC",img2)
cv.waitKey(0)
print(img)
#starting serial communication with FPGA
for k in range(2):
   print(1)
   a = ser.read()
   start = struct.unpack("B", a)
   print(start[0], k)
   ser.flush()
   while(start[0]!=240):
       a = ser.read()
       start = struct.unpack("B", a)
       ser.flush()
   print(a,k)
   if(start[0]==240):
    ser.write(struct.pack('>B', 1))
   ser.flush()
   print(1)
   a = ser.read()
   start = struct.unpack("B", a)
   print(start[0], k)
   ser.flush()
   while (start[0] != 240):
       a = ser.read()
       start = struct.unpack("B", a)
       ser.flush()
   print(a, k)
   if (start[0] == 240):
       ser.write(struct.pack('>B', 244))
   ser.flush()
s=0
tx=0

picture=np.zeros((250,250,3))
for i in range(3):
    # Sending each color plane of the original image to FPGA
    for k in range(500):
      for kk in range(500):
       print(1)
       a = ser.read()
       start = struct.unpack("B", a)
       print(start[0], k)
       ser.flush()
       while(start[0]!=240):
```

```
            a = ser.read()
            start = struct.unpack("B", a)
            ser.flush()
        print(a)
        if(start[0]==240):
         ser.write(struct.pack('>B', img[k,kk,i]))
        ser.flush()
    t0=time.time()
    #Receiving each color plane of image downsampled form FPGA
    for k in range(250):
      for kk in range(250):
          b=ser.read()
          g=struct.unpack("B", b)
          if(k+kk==0):
              t1=time.time()
          print(g[0])
          if(g[0]!=k%256):
              s=s+1
          picture[k,kk,i]=g[0]
          ser.flush()
    tx=tx+t1-t0
picture=picture.astype('uint8')
ser.close()
print(picture.shape)
#displaying the image downsampled from FPGA
cv.imshow("Image downsampled from FPGA",picture)
cv.waitKey(0)
cv.destroyAllWindows()
print("Processing time:",tx)
print("computer time",t4-t3)
print("Error in processing",sum(sum(sum(abs(picture-img2)))))
```

2) uInstruction tester

```
a=int(input("Enter the number of Uinstrauctions to be
tested:").strip())
for k in range(a):
    b=input("Enter the  uInstruction:").strip()
    c=b[4:]
    print("IU",c[2:5])
    print("SP1",c[5])
    print("Transmit",c[6])
    print("rnext",c[7:9])
    print("tnext",c[9:11])
    print("Rxallowed",c[11])
    print("wren",c[12])
    print("rden",c[13])
    print("Next Add",c[14])
    print("AC",c[15:20])
    print("Address",c[20:22])
    print("Rreg",c[22])
    print("rows",c[23])
    print("cols",c[24])
    print("next row",c[25])
    print("next col",c[26])
    print("data",c[27:29])
```

3)Compiler

```
import pandas as pd
filename="Instructions_edited_final.csv"
data=pd.read_csv(filename,encoding='utf-8')
a=list(map(int,(data['Unnamed: 2'][:39]).values.tolist()))
b=(data['Instruction Code'][:39]).values.tolist()
d={}
for g in range(len(a)):
    d[b[g].strip()]="8'b"+str(a[g])
num=int(input("Number of Instructions:").strip())
for k in range(num):
    p=input().strip()
    if(p in d.keys()):
        print("ins_mem["+str(k)+"]="+d[p]+";")
    else:
        print("ins_mem["+str(k)+"]=8'b"+str(p)+";")
```

TestBench

```
`timescale 1ns/1ps
module testbench();
      wire clear;
      reg clk;
      wire Tx;
      wire Rx;
      wire ready;
      wire ready_clr;
      wire en=0;
      wire dis=1;
      wire [7:0] data_in;
      wire start;
      wire[7:0] data_out;
      wire [4:0] Proc_state;
      wire state;
      wire tester;
      wire [9:0] PC;
      wire[17:0] NA_to_out;
   uart uart1(clear,clk,Tx,Rx,ready,ready_clr,en,dis,
data_in,start,data_out, Proc_state,state, tester,PC,NA_to_out);
  always // no sensitivity list, so it always executes
      begin
        clk = 1; #1; clk = 0; #1; // 10ns period
      end
endmodule
```

Verilog modules

1)Top module- uart.v

```
module uart(input wire clear,
                        input wire clk_50m,
                        output wire Tx,
                        input wire Rx,
                        output wire ready,
                        input wire ready_clr,
                        input wire en,
                        input wire dis,
                        output reg [7:0] data_in,
                        output reg start,
                        output wire[7:0] data_out,
                        output reg [4:0] Proc_state,
                        output reg state,
                        output reg tester,
                        output wire[9:0] PC,
                        output wire[17:0] NA_to_out
                        );
initial begin
  start=1'b0;
  data_in=8'b0;
  state=1'b0;
  tester=0;
end
wire [31:0] rows_to_out;
wire[7:0] Ins_to_out;
wire [31:0] SP1_to_out;
wire [31:0] ncols_to_out;
wire [31:0] AC_to_out;
wire [31:0] R_to_out;
wire [31:0] nrows_to_out;
wire [31:0] RNext_to_out;
wire [31:0] TNext_to_out;
wire [31:0] cols_to_out;
wire [31:0] SP2_to_out;
wire [31:0] SP3_to_out;
wire [17:0] Ad_to_out;
wire [ 7:0] data_to_out;
wire [ 7:0] operands;
wire [ 7:0] q;
wire        Z;
wire        Tx_busy;
wire        Rx_busy;
wire        Rxclk_en;
wire        Txclk_en;
reg  [17:0] address;
reg  [28:0] enables=29'b0;
wire clk;
clock clk1(clk,clk_50m);
AC AC(enables[13:9] ,AC_to_out,
Z,R_to_out,ncols_to_out,nrows_to_out,rows_to_out,cols_to_out,SP1_to_out
,SP2_to_out,SP3_to_out,q,operands,NA_to_out,clk_50m);
SP        Num_cols(clk_50m,enables[2] ,AC_to_out,ncols_to_out);
SP        Num_rows(clk_50m,enables[3] ,AC_to_out,nrows_to_out);
SP        Rows(clk_50m, enables[5],AC_to_out,rows_to_out);
```

```
SP        Cols(clk_50m, enables[4],AC_to_out,cols_to_out);
Next_Address next_address(clk_50m, enables[14], AC_to_out, NA_to_out);
Rx_TX_regs r_next(clk_50m,enables[21:20], RNext_to_out);
Rx_TX_regs t_next(clk_50m,enables[19:18], TNext_to_out);
IU IU(clk_50m,Ins_to_out,operands,enables[26:24],PC);
SP SP1(clk_50m,enables[23],AC_to_out,SP1_to_out);
SP SP2(clk_50m,enables[27],AC_to_out,SP2_to_out);
SP SP3(clk_50m,enables[28],AC_to_out,SP3_to_out);
SP R(clk_50m,enables[6],AC_to_out,R_to_out);
baudrate uart_baud(clk_50m,enables[17],Rxclk_en, Txclk_en);
transmitter uart_Tx(   data_in,enables[22],clk_50m,Txclk_en,
Tx,Tx_busy);
receiver uart_Rx(Rx,ready,ready_clr,clk_50m,Rxclk_en,
data_out,Rx_busy);
Address_reg ad_reg(clk_50m,operands,NA_to_out,enables[8:7], Ad_to_out);
SDRAM RAM(Ad_to_out,clk_50m,data_to_out,enables[16],enables[15],q);
data data_reg(clk_50m,AC_to_out,data_out,enables[1:0],data_to_out);
always@ (posedge clk_50m) begin
 if(~dis) begin
   start<=0;
      enables<=29'b00100000000000000000000000000;
      state<=0;
 end
 if(~en) begin
   start<=1'b1;
      enables<=29'b00000000000000000000000000000;
 end
 if(start==1'b1) begin
   case(state)
      1'b0: begin
        enables<=29'b00001000000000000000000000000;
        Proc_state<=0;
        state<=1'b1;
      end
      1'b1: begin
        case(Ins_to_out)
        0:begin
          enables<=29'b00000000000000000000000000000;
             state<=1'b0;
        end
        8'b1: begin
              case(Proc_state)
              5'b0: begin
               enables<=29'b00011001000000001011000000000;
                  Proc_state<=5'b1;
              end
              5'b1: begin
               enables<=29'b00000000000000001111000000000;
                  Proc_state<=5'b10;
              end
              5'b10: begin
               enables<=29'b00000000000000000101000000000;
                  Proc_state<=5'b11;
              end
              5'b11: begin
               enables<=29'b00011000000000000000000000000;
                  Proc_state<=5'b100;
              end
```

```
5'b100: begin
 enables<=29'b00000000000000001111000000000;
    Proc_state<=5'b101;
end
5'b101: begin
 enables<=29'b00000000000000001010000000000;
    Proc_state<=5'b110;
end
5'b110: begin
 enables<=29'b00011000000000000000000000000;
    Proc_state<=5'b111;
end
5'b111: begin
 enables<=29'b00000000000000001111000000000;
    Proc_state<=5'b1000;
end
5'b1000: begin
 enables<=29'b00000100000000000000000000000;
    Proc_state<=5'b1001;
end
5'b1001: begin
 if(SP1_to_out>RNext_to_out) begin
      Proc_state<=5'b1010;
      enables<=29'b00000000000000000000000000000;
     end
     else begin
      enables<=29'b00000001000000000000000000000;
      Proc_state<=5'b10011;
     end
    end
5'b1010: begin
     data_in<=8'd240;
     enables<=29'b00000010100000000000000000000;
     Proc_state<=5'b1011;
end
5'b1011: begin
  enables<=29'b00000010000000000000000000000;
     if(Tx_busy==1'b1)
        Proc_state<=5'b1100;
end
5'b1100: begin
  enables<=29'b00000000000000000000000000000;
     if(Tx_busy==0)
          Proc_state<=5'b1101;
end
5'b1101: begin
     enables<=29'b00000000000100000000000000000;
     Proc_state<=5'b1110;
end
5'b1110:begin
    if(Rx_busy==1'b1) begin
            Proc_state<=5'b1111;
            enables<=29'b00000000000100001011000000000;
          end
          else
            enables<=29'b00000000000100000000000000000;

end
```

```
        5'b1111:begin
            if(Rx_busy==0) begin
             enables<=29'b00000000000010010001010000010;
             Proc_state<=5'b10000;
            end
            else
             enables<=29'b00000000000100000000000000000;
        end
        5'b10000:begin
          enables<=29'b00000000000000001100000000000;
             Proc_state<=5'b10001;
        end
        5'b10001: begin
             enables<=29'b00000000000000100000000000000;
             Proc_state<=5'b10010;
        end
        5'b10010: begin
          enables<=29'b00000000000000000000000000000;
             Proc_state<=5'b1001;
        end
        5'b10011: begin
             enables<=29'b00000000000000000000000000000;
             state<=0;
        end
       endcase
end
8'b10: begin
  case(Proc_state)
        5'b0: begin
         enables<=29'b00011000010000001011000000000;
            Proc_state<=5'b1;
        end
        5'b1: begin
         enables<=29'b00000000000000001111000000000;
            Proc_state<=5'b10;
        end
        5'b10: begin
         enables<=29'b00000000000000000101000000000;
            Proc_state<=5'b11;
        end
        5'b11: begin
         enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b100;
        end
        5'b100: begin
         enables<=29'b00000000000000001111000000000;
            Proc_state<=5'b101;
        end
        5'b101: begin
         enables<=29'b00000000000000000101000000000;
            Proc_state<=5'b110;
        end
        5'b110: begin
         enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b111;
        end
        5'b111: begin
         enables<=29'b00000000000000001111000000000;
```

```
            Proc_state<=5'b1000;
          end
          5'b1000: begin
           enables<=29'b00000100000000000000000000000;
            Proc_state<=5'b1001;
          end
          5'b1001: begin
            if(SP1_to_out>TNext_to_out) begin
              Proc_state<=5'b1010;
              enables<=29'b00000000000000000000000000000;
             end
             else begin
              enables<=29'b00000000010000001011000000000;
              Proc_state<=5'b10000;
             end
          end
          5'b1010: begin
            enables<=29'b00000000000001010001010000000;
              Proc_state<=5'b1011;
          end
          5'b1011: begin
            enables<=29'b00000000000000001100000000000;
              Proc_state<=5'b1100;
          end
          5'b1100:begin
            enables<=29'b00000000000001000000000000000;
              Proc_state<=5'b1101;
          end
          5'b1101: begin
            enables<=29'b00000010001000000000000000000;
              data_in<=q;
              Proc_state<=5'b1110;
          end
          5'b1110: begin
            enables<=29'b00000010000000000000000000000;
              if(Tx_busy==1'b1)
                Proc_state<=5'b1111;
          end
          5'b1111: begin
            enables<=29'b00000000000000000000000000000;
              if(Tx_busy==0)
                 Proc_state<=5'b1001;
          end
          5'b10000:begin
            enables<=29'b00000000000000000000000000000;
              state<=0;
          end
        endcase
end
8'b11: begin
   case(Proc_state)
          5'b0: begin
           enables<=29'b00011000000000001011000000000;
              Proc_state<=5'b1;
          end
          5'b1: begin
           enables<=29'b00000000000000000000100000000;
              Proc_state<=5'b10;
```

```
        end
        5'b10: begin
         enables<=29'b00000000000000000000110000000;
            Proc_state<=5'b11;
        end
        5'b11: begin
         enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b100;
        end
        5'b100: begin
         enables<=29'b00000000000000000000100000000;
            Proc_state<=5'b101;
        end
        5'b101: begin
         enables<=29'b00000000000000000000110000000;
            Proc_state<=5'b110;
        end
        5'b110: begin
         enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b111;
        end
        5'b111: begin
         enables<=29'b00000000000000000000100000000;
            Proc_state<=5'b1000;
        end
        5'b1000: begin
         enables<=29'b00000000000001000000000000000;
            Proc_state<=5'b1001;
        end
        5'b1001: begin
         enables<=29'b00000000000000000000000000000;
            Proc_state<=5'b1111;
        end
        5'b1111:begin
         enables<=29'b00000000000000000000000000000;
            Proc_state<=5'b1010;
        end
        5'b1010:begin
         enables<=29'b00000000000000001110000000000;
            Proc_state<=5'b1011;
        end
        5'b1011:begin
         enables<=29'b00000000000000000000000000000;
            state<=0;
        end
      endcase
end
8'b100: begin
  case(Proc_state)
        5'b0: begin
         enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b1;
        end
        5'b1: begin
         enables<=29'b00000000000000000000100000000;
            Proc_state<=5'b10;
        end
        5'b10: begin
```

```
         enables<=29'b00000000000000000000110000000;
            Proc_state<=5'b11;
         end
         5'b11: begin
          enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b100;
         end
         5'b100: begin
          enables<=29'b00000000000000000000100000000;
            Proc_state<=5'b101;
         end
         5'b101: begin
          enables<=29'b00000000000000000000110000000;
            Proc_state<=5'b110;
         end
         5'b110: begin
          enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b111;
         end
         5'b111: begin
          enables<=29'b00000000000000000000100000000;
            Proc_state<=5'b1000;
         end
         5'b1000: begin
          enables<=29'b00000000000010000000000000001;
            Proc_state<=5'b1001;
         end
         5'b1001: begin
          enables<=29'b00000000000000000000000000000;
            Proc_state<=5'b1010;
         end
         5'b1010:begin
          enables<=29'b00000000000000000000000000000;
            Proc_state<=5'b1011;
         end
         5'b1011:begin
          enables<=29'b00000000000000000000000000000;
            state<=0;
          end
        endcase
end
8'b101: begin
   enables<=29'b00000000000000000101000000000;
      state<=0;
end
8'b110: begin
   enables<=29'b00000000000000000111000000000;
      state<=0;
end
8'b111: begin
   enables<=29'b00000000000000000110000000000;
      state<=0;
end
8'b1000:begin
   enables<=29'b00000000000000001000000000000;
      state<=0;
end
8'b1001: begin
```

```
   enables<=29'b00000000000000000000001000000;
      state<=0;
end
8'b1010: begin
   enables<=29'b00000000000000000000000100000;
      state<=0;
end
8'b1011: begin
   enables<=29'b00000000000000000100000000000;
      state<=0;
end
8'b1100: begin
   enables<=29'b00000000000000000000000010000;
      state<=0;
end
8'b1101: begin
   enables<=29'b00000000000000000011000000000;
      state<=0;
end
8'b1110: begin
   enables<=29'b00000000000000010010000000000;
      state<=0;
end
8'b1111: begin
   enables<=29'b00000000000000000000000000100;
      state<=0;
end
8'b10000: begin
   enables<=29'b00000000000000000010000000000;
      state<=0;
end
8'b10001:begin
   enables<=29'b00000000000000000000000001000;
      state<=0;
end
8'b10010:begin
   enables<=29'b00000000000000010001000000000;
      state<=0;
end
8'b10011:begin
 case(Proc_state)
     5'b0: begin
        enables<=29'b00000000000010000000010000001;
           Proc_state<=5'b1;
     end
  5'b1: begin
        enables<=29'b00000000000000000000000000000;
        Proc_state<=5'b10;
     end
     5'b10: begin
        enables<=29'b00000000000000000000000000000;
        Proc_state<=5'b11;
     end
     5'b11:begin
        enables<=29'b00000000000000000000000000000;
        state<=0;
     end
endcase
```

```
end
8'b10100:begin
   case(Proc_state)
    5'b0: begin
        enables<=29'b00000000000001000000010000000;
           Proc_state<=5'b1;
       end
       5'b1: begin
        enables<=29'b00000000000000000000000000000;
           Proc_state<=5'b10;
       end
       5'b10: begin
        enables<=29'b00000000000000000000000000000;
           Proc_state<=5'b11;
       end
       5'b11:begin
        enables<=29'b00000000000000001110000000000;
           Proc_state<=5'b100;
       end
       5'b100:begin
        enables<=29'b00000000000000000000000000000;
           state<=0;
       end
      endcase
end
8'b10101:begin
   enables<=29'b00000000000000001010000000000;
      state<=0;
end
8'b10110:begin
   enables<=29'b00000000000000000001000000000;
      state<=0;
end
8'b10111: begin
   enables<=29'b00000000000000001100000000000;
      state<=0;
end
8'b11000: begin
   enables<=29'b00000000000000001101000000000;
      state<=0;

end
8'b11001:begin
   enables<=29'b00000000000000001011000000000;
      state<=0;
end
8'b11010:begin
 case(Proc_state)
 5'b0: begin
       if(Z==0) begin
            enables<=29'b00010000000000000000000000000;
            Proc_state<=5'b1;
          end
          else begin
            enables<=29'b00011000000000000000000000000;
            Proc_state<=5'b101;
          end
 end
```

```
 5'b1:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b10;
 end
 5'b10:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b11;
 end
 5'b11:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b100;
 end
 5'b100: begin
   enables<=29'b00000000000000000000000000000;
      state<=0;
 end
 5'b101:begin
   enables<=29'b00011000000000000000000000000;
      Proc_state<=5'b100;
 end
 endcase
end
8'b11011:begin
 case(Proc_state)
 5'b0:begin
   enables<=29'b00010000000000000000000000000;
      Proc_state<=5'b1;
 end
 5'b1:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b10;
 end
 5'b10:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b11;
 end
 5'b11:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b100;
 end
 5'b100: begin
   enables<=29'b00000000000000000000000000000;
      state<=0;
 end
 endcase
end
8'b11100:begin
 case(Proc_state)
 5'b0:begin
   if(Z==1'b1) begin
     enables<=29'b00010000000000000000000000000;
          Proc_state<=5'b1;
      end
      else begin
        enables<=29'b00011000000000000000000000000;
          Proc_state<=5'b101;
      end
```

```
 end
 5'b1:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b10;
 end
 5'b10:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b11;
 end
 5'b11:begin
   enables<=29'b00000000000000000000000000000;
      Proc_state<=5'b100;
 end
 5'b100: begin
   enables<=29'b00000000000000000000000000000;
      state<=0;
 end
 5'b101:begin
   enables<=29'b00011000000000000000000000000;
      Proc_state<=5'b100;
 end
 endcase
end
8'b11101: begin
  enables<=29'b00000000000000001001000000000;
     state<=0;
end
8'b11110: begin
  enables<=29'b00000000000000010011000000000;
     state<=0;
end
8'b11111: begin
  enables<=29'b00000100000000000000000000000;
     state<=0;
end
8'b100000:begin
  enables<=29'b00000000000000010000000000000;
     state<=0;
end
8'b100001:begin
  enables<=29'b00000000000000100000000000000;
     state<=0;
end
8'b100010:begin
  enables<=29'b00000000000000010100000000000;
     state<=0;
end
8'b100011:begin
  enables<=29'b00000000000000010101000000000;
     state<=0;
end
8'b100100:begin
  enables<=29'b01000000000000000000000000000;
     state<=0;
end
8'b100101:begin
  enables<=29'b10000000000000000000000000000;
     state<=0;
```

```
       end
       8'b100110:begin
         enables<=29'b00000000000000000000000000000;
            start<=0;
       end
       endcase
   end
  endcase
 end
end
endmodule
```

2) AC

```
module AC (input wire  [ 4:0] enables,
           output wire [31:0] AC_to_out,
                   output wire        Z,
                   input wire  [31:0] R_to_AC,
                   input wire  [31:0] ncols_to_AC,
                   input wire  [31:0] nrows_to_AC,
                   input wire  [31:0] rows_to_AC,
                   input wire  [31:0] cols_to_AC,
                   input wire  [31:0] SP1_to_AC,
                   input wire  [31:0] SP2_to_AC,
                   input wire  [31:0] SP3_to_AC,
                   input wire  [7:0 ] memory_to_AC,
                   input wire  [7:0 ] Ins_to_AC,
                   input wire  [17:0] NA_to_AC,
                   input wire         clk
                   );
reg [31:0] AC;
always@(negedge clk) begin
  case(enables)
  5'b1: begin
    AC<=R_to_AC;
  end
  5'b10010: begin
    AC<=ncols_to_AC;
  end
  5'b10: begin
    AC<=nrows_to_AC;
  end
  5'b11: begin
    AC<=cols_to_AC;
  end
  5'b100:begin
    AC<=rows_to_AC;
  end
  5'b101:begin
    AC<=(AC<<8);
  end
  5'b110:begin
    AC<=(AC>>8);
  end
  5'b111:begin
    AC<=(AC<<1);
```

```
  end
  5'b1000:begin
    AC<=(AC>>1);
  end
  5'b1001:begin
    AC<=AC+R_to_AC;
  end
  5'b1010:begin
    AC<=AC^R_to_AC;
  end
  5'b1011:begin
    AC<=0;
  end
  5'b1100:begin
    AC<=AC+1;
  end
  5'b1101:begin
    AC<=AC-1;
  end
  5'b1110:begin
    AC[7:0]<=memory_to_AC;
  end
  5'b1111:begin
    AC[7:0]<=Ins_to_AC;
  end
  5'b10000:begin
    AC<=SP1_to_AC;
  end
  5'b10001:begin
    AC[17:0]<=NA_to_AC;
  end
  5'b10011:begin
    AC<=AC-R_to_AC;
  end
  5'b10100:begin
    AC<=SP2_to_AC;
  end
  5'b10101:begin
    AC<=SP3_to_AC;
  end
  default: begin

  end
  endcase
end
assign AC_to_out=AC;
assign Z=(AC==0);
endmodule
```

3)Receiver

```
module receiver  (input wire Rx,
                                    output reg ready,
                                    input wire ready_clr,
                                    input wire clk_50m,
                                    input wire clken,
                                    output reg [7:0] data,
                                    output wire Rx_busy
                                    );
initial begin
      ready = 1'b0; // initialize ready = 0
      data = 8'b0; // initialize data as 00000000
end
// Define the 4 states using 00,01,10 signals
parameter RX_STATE_START     = 2'b00;
parameter RX_STATE_DATA           = 2'b01;
parameter RX_STATE_STOP           = 2'b10;
parameter RX_STATE_IDLE    = 2'b11;


 // This is a 4-bit register
reg [3:0] bit_pos = 0; // bit position is a 4-bit register/vector,
initially equal to 000
reg [7:0] scratch = 8'b0; // An 8-bit register assigned to 00000000
reg [3:0] sample=0;



reg [1:0] state=2'b11;
always @(posedge clk_50m) begin
      if (ready_clr)
            ready <= 1'b0; // This resets ready to 0

      if (clken ) begin
            case (state) // Let us consider the 4 states of the receiver
            RX_STATE_IDLE: begin
               state<=RX_STATE_START;
            end
            RX_STATE_START: begin
            // We define condtions for starting the receiver
                  if (!Rx || sample != 0) // start counting from the
first low sample
                        sample <= sample + 4'b1; // increment by 0001
                  if (sample == 15) begin // once a full bit has been
sampled
                        state <= RX_STATE_DATA; //   start collecting
data bits

                        bit_pos <= 0;
                        sample <= 0;
                        scratch <= 0;
                  end
            end
            RX_STATE_DATA: begin // We define conditions for starting
the data colleting
                  sample <= sample + 4'b1; // increment by 0001
                  if (sample == 4'h8) begin // we keep assigning Rx data
until all bits have 01 to 7
```

```
                        scratch[bit_pos[2:0]] <= Rx;
                        bit_pos <= bit_pos + 4'b1; // increment by 0001
                  end
                  if (bit_pos == 8 && sample == 15) // when a full bit
has been sampled and
                        state <= RX_STATE_STOP; // bit position has
finally reached 7, assign state to stop
            end
            RX_STATE_STOP: begin
                  /*
                   * Our baud clock may not be running at exactly the
                   * same rate as the transmitter.  If we thing that
                   * we're at least half way into the stop bit, allow
                   * transition into handling the next start bit.
                   */
                  if (sample == 15 || (sample >= 8 && !Rx)) begin
                        state <= RX_STATE_IDLE;
                        data <= scratch;
                        ready <= 1'b1;
                        sample <= 0;
                  end
                  else begin
                        sample <= sample + 4'b1;
                  end
            end
            default: begin
                  state <= RX_STATE_IDLE; // always begin with state
assigned to START
            end
            endcase
      end
end
assign Rx_busy=(state!=RX_STATE_IDLE);
endmodule
```

4)Transmitter

```
module transmitter(    input wire [7:0] data_in, //input data as an 8-
bit regsiter/vector
                                    input wire wr_en,
//enable wire to start
                                       input wire clk_50m,
                                       input wire clken, //clock
signal for the transmitter
                                       output reg Tx, //a single 1-bit
register variable to hold transmitting bit
                                       output wire Tx_busy
//transmitter is busy signal

                                       );

initial begin
       Tx = 1'b1; //initialize Tx = 1 to begin the transmission
end
//Define the 4 states using 00,01,10,11 signals
parameter TX_STATE_IDLE      = 2'b00;
```

```
parameter TX_STATE_START     = 2'b01;
parameter TX_STATE_DATA      = 2'b10;
parameter TX_STATE_STOP      = 2'b11;

reg [7:0] data = 8'h00; //set an 8-bit register/vector as
data,initially equal to 00000000
reg [2:0] bit_pos = 3'h0; //bit position is a 3-bit register/vector,
initially equal to 000
reg [1:0] state=2'b00;
//state is a 2 bit register/vector,initially equal to 00

always @(posedge clk_50m) begin
      case (state) //Let us consider the 4 states of the transmitter
      TX_STATE_IDLE: begin //We define the conditions for idle  or NOT-
BUSY state
            if(wr_en) begin
                  state <= TX_STATE_START; //assign the start signal to
state
                  data <= data_in; //we assign input data vector to the
current data
                  bit_pos <= 3'h0; //we assign the bit position to zero
            end

      end
      TX_STATE_START: begin //We define the conditions for the
transmission start state
            if (clken) begin
                  Tx <= 1'b0; //set Tx = 0 after transmission has
started
                  state <= TX_STATE_DATA;
            end
      end
      TX_STATE_DATA: begin
            if (clken) begin
                  if (bit_pos == 3'h7) begin//we keep assigning Tx with
the data until all bits have been transmitted from 0 to 7
                        state <= TX_STATE_STOP; // when bit position has
finally reached 7, assign state to stop transmission
                  end
                  else
                        bit_pos <= bit_pos + 3'h1; //increment the bit
position by 001
                  Tx <= data[bit_pos]; //Set Tx to the data value of the
bit position ranging from 0-7
            end
      end
      TX_STATE_STOP: begin
            if (clken) begin
                  Tx <= 1'b1; //set Tx = 1 after transmission has ended
                  state <= TX_STATE_IDLE; //Move to IDLE state once a
transmission has been completed

            end
      end
      default: begin
            Tx <= 1'b1; // always begin with Tx = 1 and state assigned
to IDLE
            state <= TX_STATE_IDLE;
```

```
      end
      endcase
end

assign Tx_busy = (state != TX_STATE_IDLE); //We assign the BUSY signal
when the transmitter is not idle

endmodule
```

5)SP

```
module SP(input wire clk,
          input wire    enable,
                  input wire [31:0]   AC_to_SP,
                  output wire [31:0] SP_to_out
                  );
reg [31:0] SP=0;
always@(negedge clk) begin
      if(enable==1'b1)
        SP<=AC_to_SP;
end
assign SP_to_out=SP;
endmodule
```

6)RX_TX regs

```
module Rx_TX_regs(input wire clk,
                   input wire [1:0] enables,
                                    output wire [31:0] RT_to_out
                                    );
reg [31:0] RT=0;
always@(negedge clk) begin
 case(enables)
  2'b1: begin
   RT<=RT+1;
  end
  2'b10: begin
   RT<=0;
  end
  default: begin

  end
 endcase
end
assign RT_to_out=RT;
endmodule
```

7)Baudrate

```
//This is a baud rate generator to divide a 50MHz clock into a 115200
baud Tx/Rx pair.
//The Rx clock oversamples by 16x.

module baudrate  (input wire clk_50m,
                                  input wire Rx_allowed,
                                 output wire Rxclk_en,
                                 output wire Txclk_en
                                 );
//Our Testbench uses a 50 MHz clock.
//Want to interface to 115200 baud UART for Tx/Rx pair
//Hence, 50000000 / 115200 = 435 Clocks Per Bit.
parameter RX_ACC_MAX = 50000000 / (115200 * 16);
parameter TX_ACC_MAX = 50000000 / 115200;
parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);
parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);
reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;
reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;

assign Rxclk_en = (rx_acc == 5'd0) & Rx_allowed;
assign Txclk_en = (tx_acc == 9'd0) ;

always @(posedge clk_50m) begin
      if (rx_acc == RX_ACC_MAX[RX_ACC_WIDTH - 1:0])
            rx_acc <= 0;
      else
            rx_acc <= rx_acc + 5'b1; //increment by 00001
end

always @(posedge clk_50m) begin
      if (tx_acc == TX_ACC_MAX[TX_ACC_WIDTH - 1:0])
            tx_acc <= 0;
      else
            tx_acc <= tx_acc + 9'b1; //increment by 000000001
end

endmodule
```

8)Address regs

```
module Address_reg(input wire clk,
                    input wire [7:0] Ins,
                                  input wire [17:0] Na,
                                  input wire[1:0] enables,
                                  output wire[17:0] Add_to_out
                                  );
reg [17:0] Address_reg;
always@(negedge clk) begin
  case(enables)
  2'b1: begin
    Address_reg<=Na;
  end
  2'b10:begin
    Address_reg[7:0]<=Ins;
  end
```

```
  2'b11:begin
     Address_reg<=(Address_reg<<8);
  end
  default: begin
  end
  endcase
end
assign Add_to_out=Address_reg;
endmodule
```

9)Next Address reg

```
module Next_Address(input wire clk,
                    input wire enables,
                                        input wire[31:0] AC_to_NA,
                                        output wire[17:0] NA_to_out
                                        );
reg [17:0]  NA=0;
always@(negedge clk) begin
 case(enables)
   1'b1: begin
    NA<=AC_to_NA[17:0];
      end
      default: begin

      end
  endcase
end
assign NA_to_out=NA;
endmodule
```

10)Instruction Unit

```
module IU(input wire clk,
                  output wire [7:0] Ins_to_out,
                  output reg [7:0] Operands,
                  input wire  [2:0] enable,
                  output reg[9:0] PC
                 );
initial begin
  PC=0;
end
reg [9:0] temp;
reg [7:0] Ins=8'b0;
reg [7:0] ins_mem[1023:0];
initial begin
   ins_mem[0]=8'b11001;
      ins_mem[1]=8'b11111;
      ins_mem[2]=8'b100001;
      ins_mem[3]=8'b1;
      ins_mem[4]=8'b0;
      ins_mem[5]=8'b0;
      ins_mem[6]=8'b100;
      ins_mem[7]=8'b11001;
      ins_mem[8]=8'b100001;
```

```
ins_mem[9]=8'b11001;
ins_mem[10]=8'b10100;
ins_mem[11]=8'b101;
ins_mem[12]=8'b1001;
ins_mem[13]=8'b11001;
ins_mem[14]=8'b10111;
ins_mem[15]=8'b100001;
ins_mem[16]=8'b10110;
ins_mem[17]=8'b10100;
ins_mem[18]=8'b1111;
ins_mem[19]=8'b11001;
ins_mem[20]=8'b10111;
ins_mem[21]=8'b110;
ins_mem[22]=8'b100001;
ins_mem[23]=8'b11001;
ins_mem[24]=8'b10100;
ins_mem[25]=8'b101;
ins_mem[26]=8'b1001;
ins_mem[27]=8'b11001;
ins_mem[28]=8'b10111;
ins_mem[29]=8'b10111;
ins_mem[30]=8'b10111;
ins_mem[31]=8'b100001;
ins_mem[32]=8'b10110;
ins_mem[33]=8'b10100;
ins_mem[34]=8'b10001;
ins_mem[35]=8'b11001;
ins_mem[36]=8'b100;
ins_mem[37]=8'b11;
ins_mem[38]=8'b11010000;
ins_mem[39]=8'b10010001;
ins_mem[40]=8'b11001;
ins_mem[41]=8'b10111;
ins_mem[42]=8'b10111;
ins_mem[43]=8'b10111;
ins_mem[44]=8'b1001;
ins_mem[45]=8'b11;
ins_mem[46]=8'b11;
ins_mem[47]=8'b11010000;
ins_mem[48]=8'b10010001;
ins_mem[49]=8'b100100;
ins_mem[50]=8'b10101;
ins_mem[51]=8'b11100;
ins_mem[52]=8'b1;
ins_mem[53]=8'b101011;
ins_mem[54]=8'b100010;
ins_mem[55]=8'b10111;
ins_mem[56]=8'b100;
ins_mem[57]=8'b11;
ins_mem[58]=8'b11010000;
ins_mem[59]=8'b10010001;
ins_mem[60]=8'b11001;
ins_mem[61]=8'b10111;
ins_mem[62]=8'b1010;
ins_mem[63]=8'b1100;
ins_mem[64]=8'b11001;
ins_mem[65]=8'b11111;
ins_mem[66]=8'b100100;
```

```
ins_mem[67]=8'b100101;
ins_mem[68]=8'b100001;
ins_mem[69]=8'b1;
ins_mem[70]=8'b11;
ins_mem[71]=8'b11010000;
ins_mem[72]=8'b10010000;
ins_mem[73]=8'b11001;
ins_mem[74]=8'b100001;
ins_mem[75]=8'b10010;
ins_mem[76]=8'b10111;
ins_mem[77]=8'b1001;
ins_mem[78]=8'b1110;
ins_mem[79]=8'b11101;
ins_mem[80]=8'b100001;
ins_mem[81]=8'b11001;
ins_mem[82]=8'b100100;
ins_mem[83]=8'b10000;
ins_mem[84]=8'b11000;
ins_mem[85]=8'b1001;
ins_mem[86]=8'b1011;
ins_mem[87]=8'b10101;
ins_mem[88]=8'b11100;
ins_mem[89]=8'b0;
ins_mem[90]=8'b11111111;
ins_mem[91]=8'b1110;
ins_mem[92]=8'b11000;
ins_mem[93]=8'b1001;
ins_mem[94]=8'b1101;
ins_mem[95]=8'b10101;
ins_mem[96]=8'b11100;
ins_mem[97]=8'b0;
ins_mem[98]=8'b11100010;
ins_mem[99]=8'b1110;
ins_mem[100]=8'b1001;
ins_mem[101]=8'b11001;
ins_mem[102]=8'b10010;
ins_mem[103]=8'b11000;
ins_mem[104]=8'b11110;
ins_mem[105]=8'b100001;
ins_mem[106]=8'b11001;
ins_mem[107]=8'b10100;
ins_mem[108]=8'b11111;
ins_mem[109]=8'b11001;
ins_mem[110]=8'b10010;
ins_mem[111]=8'b10111;
ins_mem[112]=8'b100001;
ins_mem[113]=8'b11001;
ins_mem[114]=8'b10100;
ins_mem[115]=8'b110;
ins_mem[116]=8'b1001;
ins_mem[117]=8'b100000;
ins_mem[118]=8'b11101;
ins_mem[119]=8'b11111;
ins_mem[120]=8'b11001;
ins_mem[121]=8'b10010;
ins_mem[122]=8'b10111;
ins_mem[123]=8'b100001;
ins_mem[124]=8'b11001;
```

```
ins_mem[125]=8'b10100;
ins_mem[126]=8'b1001;
ins_mem[127]=8'b100000;
ins_mem[128]=8'b11101;
ins_mem[129]=8'b11111;
ins_mem[130]=8'b11001;
ins_mem[131]=8'b10010;
ins_mem[132]=8'b1001;
ins_mem[133]=8'b1110;
ins_mem[134]=8'b11101;
ins_mem[135]=8'b100001;
ins_mem[136]=8'b11001;
ins_mem[137]=8'b10100;
ins_mem[138]=8'b110;
ins_mem[139]=8'b1001;
ins_mem[140]=8'b100000;
ins_mem[141]=8'b11101;
ins_mem[142]=8'b11111;
ins_mem[143]=8'b11001;
ins_mem[144]=8'b10010;
ins_mem[145]=8'b11000;
ins_mem[146]=8'b100001;
ins_mem[147]=8'b11001;
ins_mem[148]=8'b10100;
ins_mem[149]=8'b110;
ins_mem[150]=8'b110;
ins_mem[151]=8'b1001;
ins_mem[152]=8'b100000;
ins_mem[153]=8'b11101;
ins_mem[154]=8'b11111;
ins_mem[155]=8'b11001;
ins_mem[156]=8'b10010;
ins_mem[157]=8'b11000;
ins_mem[158]=8'b100001;
ins_mem[159]=8'b11001;
ins_mem[160]=8'b10100;
ins_mem[161]=8'b110;
ins_mem[162]=8'b1001;
ins_mem[163]=8'b100000;
ins_mem[164]=8'b11101;
ins_mem[165]=8'b11111;
ins_mem[166]=8'b11001;
ins_mem[167]=8'b10010;
ins_mem[168]=8'b1001;
ins_mem[169]=8'b1110;
ins_mem[170]=8'b11101;
ins_mem[171]=8'b100001;
ins_mem[172]=8'b11001;
ins_mem[173]=8'b10100;
ins_mem[174]=8'b1001;
ins_mem[175]=8'b100000;
ins_mem[176]=8'b11101;
ins_mem[177]=8'b11111;
ins_mem[178]=8'b11001;
ins_mem[179]=8'b10010;
ins_mem[180]=8'b10111;
ins_mem[181]=8'b100001;
ins_mem[182]=8'b11001;
```

```
ins_mem[183]=8'b10100;
ins_mem[184]=8'b110;
ins_mem[185]=8'b1001;
ins_mem[186]=8'b100000;
ins_mem[187]=8'b11101;
ins_mem[188]=8'b11111;
ins_mem[189]=8'b11001;
ins_mem[190]=8'b10010;
ins_mem[191]=8'b10111;
ins_mem[192]=8'b100001;
ins_mem[193]=8'b11001;
ins_mem[194]=8'b10100;
ins_mem[195]=8'b1001;
ins_mem[196]=8'b100000;
ins_mem[197]=8'b11101;
ins_mem[198]=8'b1000;
ins_mem[199]=8'b1000;
ins_mem[200]=8'b1000;
ins_mem[201]=8'b1000;
ins_mem[202]=8'b11111;
ins_mem[203]=8'b11001;
ins_mem[204]=8'b10010;
ins_mem[205]=8'b100101;
ins_mem[206]=8'b100010;
ins_mem[207]=8'b100001;
ins_mem[208]=8'b10111;
ins_mem[209]=8'b100100;
ins_mem[210]=8'b100000;
ins_mem[211]=8'b10011;
ins_mem[212]=8'b1110;
ins_mem[213]=8'b1001;
ins_mem[214]=8'b11001;
ins_mem[215]=8'b100011;
ins_mem[216]=8'b11110;
ins_mem[217]=8'b10111;
ins_mem[218]=8'b100001;
ins_mem[219]=8'b1101;
ins_mem[220]=8'b10111;
ins_mem[221]=8'b10111;
ins_mem[222]=8'b1100;
ins_mem[223]=8'b11011;
ins_mem[224]=8'b0;
ins_mem[225]=8'b1011011;
ins_mem[226]=8'b11001;
ins_mem[227]=8'b10111;
ins_mem[228]=8'b1100;
ins_mem[229]=8'b1011;
ins_mem[230]=8'b10111;
ins_mem[231]=8'b10111;
ins_mem[232]=8'b1010;
ins_mem[233]=8'b11001;
ins_mem[234]=8'b10100;
ins_mem[235]=8'b1001;
ins_mem[236]=8'b11001;
ins_mem[237]=8'b10010;
ins_mem[238]=8'b100101;
ins_mem[239]=8'b100010;
ins_mem[240]=8'b100001;
```

```
ins_mem[241]=8'b10111;
ins_mem[242]=8'b100100;
ins_mem[243]=8'b10110;
ins_mem[244]=8'b10011;
ins_mem[245]=8'b1110;
ins_mem[246]=8'b1001;
ins_mem[247]=8'b100011;
ins_mem[248]=8'b11101;
ins_mem[249]=8'b10111;
ins_mem[250]=8'b10111;
ins_mem[251]=8'b100001;
ins_mem[252]=8'b11011;
ins_mem[253]=8'b0;
ins_mem[254]=8'b1010011;
ins_mem[255]=8'b11001;
ins_mem[256]=8'b10111;
ins_mem[257]=8'b1100;
ins_mem[258]=8'b1101;
ins_mem[259]=8'b1001;
ins_mem[260]=8'b1110;
ins_mem[261]=8'b10111;
ins_mem[262]=8'b11110;
ins_mem[263]=8'b11100;
ins_mem[264]=8'b1;
ins_mem[265]=8'b100010;
ins_mem[266]=8'b11001;
ins_mem[267]=8'b10100;
ins_mem[268]=8'b1001;
ins_mem[269]=8'b11001;
ins_mem[270]=8'b10010;
ins_mem[271]=8'b100101;
ins_mem[272]=8'b100010;
ins_mem[273]=8'b100001;
ins_mem[274]=8'b10111;
ins_mem[275]=8'b100100;
ins_mem[276]=8'b10110;
ins_mem[277]=8'b10011;
ins_mem[278]=8'b11001;
ins_mem[279]=8'b100011;
ins_mem[280]=8'b10111;
ins_mem[281]=8'b10111;
ins_mem[282]=8'b100001;
ins_mem[283]=8'b1101;
ins_mem[284]=8'b10111;
ins_mem[285]=8'b10111;
ins_mem[286]=8'b1100;
ins_mem[287]=8'b11011;
ins_mem[288]=8'b1;
ins_mem[289]=8'b10;
ins_mem[290]=8'b11001;
ins_mem[291]=8'b100001;
ins_mem[292]=8'b10;
ins_mem[293]=8'b0;
ins_mem[294]=8'b11110100;
ins_mem[295]=8'b100100;
ins_mem[296]=8'b11011;
ins_mem[297]=8'b0;
ins_mem[298]=8'b101000;
```

```
      ins_mem[299]=8'b100110;
end
assign Ins_to_out=Ins;
always@(negedge clk) begin
      case(enable)
      3'b1: begin
        Ins=ins_mem[PC];
        PC<=PC+1;
      end
      3'b10: begin
        temp=PC;
        PC[7:0]=ins_mem[temp];
        temp=temp+1;
        PC=PC<<8;
        PC[7:0]<=ins_mem[temp];
      end
      3'b11: begin
        Operands=ins_mem[PC];
        PC<=PC+1;
      end
      3'b100:begin
        PC<=0;
      end
      default: begin
      end
      endcase
end
endmodule
```

11)Data reg

```
module data(input wire clk,
            input wire[31:0] AC_to_data,
                        input wire[7:0]  tr_to_data,
                  input wire[1:0]  enables,
                        output wire[7:0] data_to_out
                        );
reg [7:0] data;
always@ (negedge clk) begin
  case(enables)
   2'b1: begin
     data<=AC_to_data[7:0];
      end
   2'b10:begin
     data<=tr_to_data;
      end
      default:begin
      end
  endcase
end
assign data_to_out=data;
endmodule
```

12)SDRAM

```
// megafunction wizard: %RAM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: SDRAM.v
// Megafunction Name(s):
//               altsyncram
//
// Simulation Library Files(s):
//               altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 18.1.0 Build 625 09/12/2018 SJ Lite Edition
// ************************************************************


//Copyright (C) 2018  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel FPGA IP License Agreement, or other applicable license
//agreement, including, without limitation, that your use is for
//the sole purpose of programming logic devices manufactured by
//Intel and sold by Intel or its authorized distributors.  Please
//refer to the applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module SDRAM (
      address,
      clock,
      data,
      wren,
      rden,
      q);

      input [17:0]  address;
      input   clock;
      input [7:0]  data;
      input   wren;
      input rden;
      output      [7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
      tri1    clock;
```

```
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

      wire [7:0] sub_wire0;
      wire [7:0] q = sub_wire0[7:0];

      altsyncram altsyncram_component (
                  .address_a (address),
                  .clock0 (clock),
                  .data_a (data),
                  .wren_a (wren),
                  .q_a (sub_wire0),
                  .aclr0 (1'b0),
                  .aclr1 (1'b0),
                  .address_b (1'b1),
                  .addressstall_a (1'b0),
                  .addressstall_b (1'b0),
                  .byteena_a (1'b1),
                  .byteena_b (1'b1),
                  .clock1 (1'b1),
                  .clocken0 (1'b1),
                  .clocken1 (1'b1),
                  .clocken2 (1'b1),
                  .clocken3 (1'b1),
                  .data_b (1'b1),
                  .eccstatus (),
                  .q_b (),
                  .rden_a (rden),
                  .rden_b (1'b1),
                  .wren_b (1'b0));
      defparam
            altsyncram_component.clock_enable_input_a = "BYPASS",
            altsyncram_component.clock_enable_output_a = "BYPASS",
            altsyncram_component.intended_device_family = "Cyclone IV
E",
            altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
            altsyncram_component.lpm_type = "altsyncram",
            altsyncram_component.numwords_a = 262144,
            altsyncram_component.operation_mode = "SINGLE_PORT",
            altsyncram_component.outdata_aclr_a = "NONE",
            altsyncram_component.outdata_reg_a = "CLOCK0",
            altsyncram_component.power_up_uninitialized = "FALSE",
            altsyncram_component.read_during_write_mode_port_a =
"NEW_DATA_NO_NBE_READ",
            altsyncram_component.widthad_a = 18,
            altsyncram_component.width_a = 8,
            altsyncr
```

```
// Retrieval info: PRIVATE: AclrData NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone IV
E"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING ""
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "262144"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
// Retrieval info: PRIVATE: RegAddr NUMERIC "1"
// Retrieval info: PRIVATE: RegData NUMERIC "1"
// Retrieval info: PRIVATE: RegOutput NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "1"
// Retrieval info: PRIVATE: SingleClock NUMERIC "1"
// Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
// Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: WidthAddr NUMERIC "18"
// Retrieval info: PRIVATE: WidthData NUMERIC "8"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf
altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone IV
E"
// Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "262144"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "CLOCK0"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A STRING
"NEW_DATA_NO_NBE_READ"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "18"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 18 0 INPUT NODEFVAL
"address[17..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: data 0 0 8 0 INPUT NODEFVAL "data[7..0]"
// Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren"
// Retrieval info: CONNECT: @address_a 0 0 18 0 address 0 0 18 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 8 0 data 0 0 8 0
```

```
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: q 0 0 8 0 @q_a 0 0 8 0
// Retrieval info: GEN_FILE: TYPE_NORMAL SDRAM.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL SDRAM.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL SDRAM.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL SDRAM.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL SDRAM_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL SDRAM_bb.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL SDRAM_syn.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```