Abstract

Application specific custom processors are used widely in electronic devices which are optimized for a given task. These designs are implemented, tested and debugged first prior to the fabrication process of the system on a chip (SoC) and field programmable gate arrays (FPGA) provide an efficient way of doing so. This report describes the design and implementation of a custom processor for image down sampling using Altera DE2-115 board and Cyclone IV EP4CE115F29 FPGA.

The designed custom processor is capable of downsampling a grayscale or RGB image of 256x256 pixels by any factor up to 15 using an average filter or a Gaussian filter. However the aliasing effect increases with the downsampling factor and downsampling using factors 2 and 3 are only discussed. In addition to that, the processor can be used to implement any linear separable filter and Gaussian smoothing is implemented as an example. The unique instruction set contains 14 instructions of the same length of 16 bits.

MATLAB is used to transmit and receive the image using UART and a python based compiler is used to decode the instructions in human language to binary bit patterns. The implementation of the downsampling algorithm using the processor was compared with the same implementation in MATLAB and zero sum of squared differences (SSD) could be obtained. This report looks at the instruction set, registers, data path, processor architecture, algorithms, compiler, timing diagrams and the obtained results. Implemented codes in Verilog, MATLAB and python are attached to the end as appendices.

Table of Contents

1. Introduction

Processors can be categorized into two different types as general purpose processors and application specific processors. The processor that we are designing belongs to the second type since it is developed for the specific purpose of low pass filtering and down sampling a given image. An instruction set architecture which suits to achieve these tasks is defined. For the design, Verilog hardware description language is used and for the implementation, Quartus Prime 18.1 Light Edition along with Altera DE2-115 education and development board with Cyclone IV FPGA (Field Programmable Gate Array) are used.

1.1 Introduction to the DE2-115 Board

Following hardware is provided on the DE2-115 board;

- Altera Cyclone® IV 4CE115 FPGA device
- Altera Serial Configuration device – EPCS64
- USB Blaster (on board) for programming; both JTAG and Active Serial (AS) programming modes are supported
- 2MB SRAM
- Two 64MB SDRAM
- 8MB Flash memory
- SD Card socket
- 4 Push-buttons
- 18 Slide switches
- 18 Red user LEDs
- 9 Green user LEDs
- 50MHz oscillator for clock sources
- 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (8-bit high-speed triple DACs) with VGA-out connector
- TV Decoder (NTSC/PAL/SECAM) and TV-in connector
- 2 Gigabit Ethernet PHY with RJ45 connectors
- USB Host/Slave Controller with USB type A and type B connectors
- RS-232 transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- IR Receiver
- 2 SMA connectors for external clock input/output
- One 40-pin Expansion Header with diode protection
- One High Speed Mezzanine Card (HSMC) connector
- 16x2 LCD module

In addition to these hardware features, the DE2-115 board has software support for standard I/O interfaces and a control panel facility for accessing various components. Also, the software is provided for supporting a number of demonstrations that illustrate the advanced capabilities of the DE2-115 board.

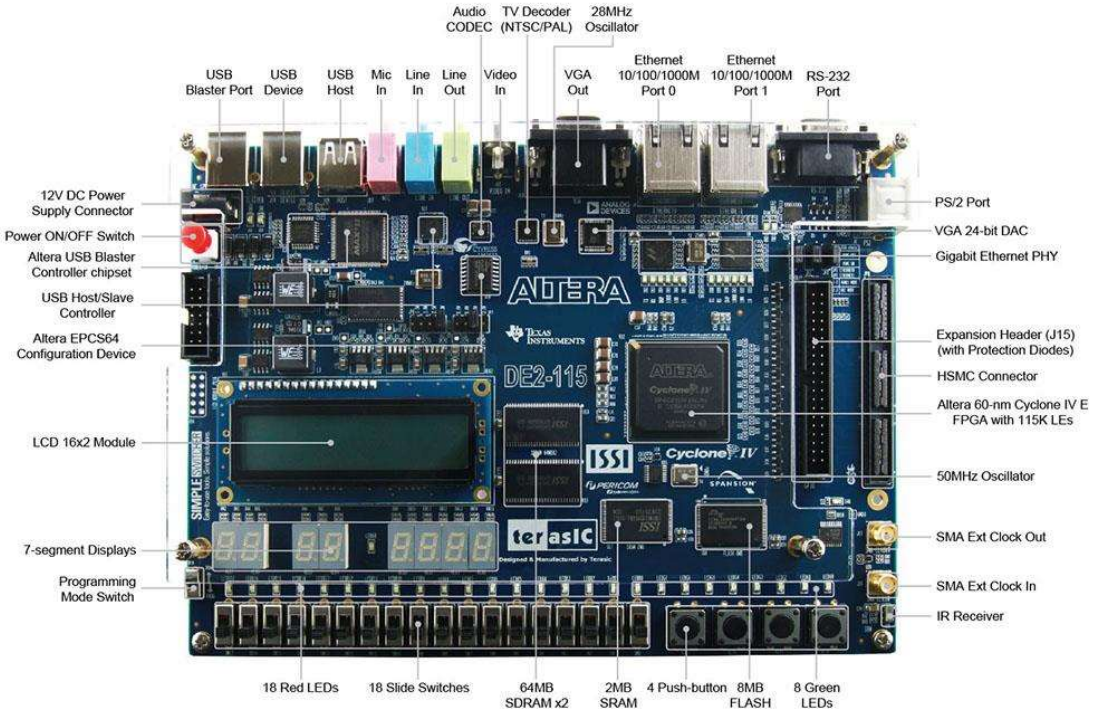


Figure 1.1 – Altera DE2-115 Board

1.2 Central Processing Unit (CPU)

The CPU carries out the instruction cycle (fetch, decode and execute) for the instructions written in the program by performing the basic arithmetic, logical, control and I/O (Input/Output) operations specified by the instructions. The CPU comprises of the processing unit and the control unit. The CPU has a control unit in order to decode instructions and generate control signals. CPUs may or may not have multiple processors to carry out multiple tasks simultaneously.

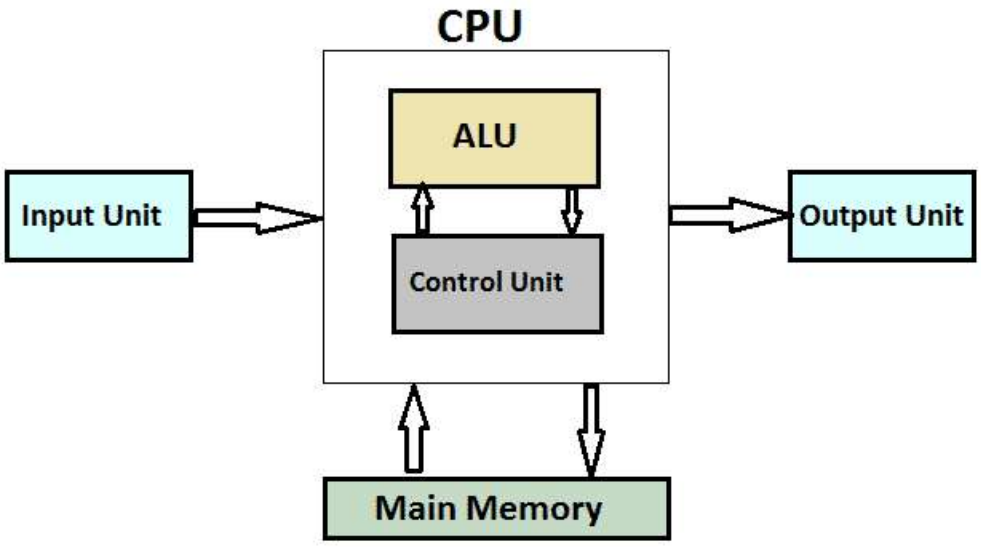


Figure 1.2 – Block Diagram of a CPU

1.3 Problem Statement

Designing a custom processor which is capable of down sampling a given image is the main objective of the given task. There are no constraints imposed upon the specifications like the down sampling factor, resolution of the original image and whether the original image is color or greyscale. When studying the specified task, it becomes clear that, filtering and down sampling are the main requirements of the processor. In addition to that, down sampled output image of our processor should be compared with a down sampled output image obtained by MATLAB and the standard error should be calculated in order to verify the implementation.

1.4 General Overview of the Solution

This main task can be divided into several sub tasks as follows;

1. Converting the pixel data of the given image in to a data stream.
   This can be done using Matlab

2. Transmission of the data stream from the computer to the designed memory unit in the FPGA.
   With the help of 'serial' objects in MATLAB, the obtained data stream can be transmitted through a COM port of the computer to the FPGA. To receive the data, UART receiver module should be implemented using Verilog in FPGA. After receiving the data, they should be stored in a memory unit that has been implemented in FPGA.

3. Filtering the stored image
   Data is stored in the memory unit, so that the data can be accessed when required. So the processor starts its work according to the available instructions based on a filtering algorithm. This process of filtering is mainly done to get rid of the higher frequency components in the image as they can cause aliasing effects in the down sampled output image. After filtering, the filtered data is again stored to the memory unit.

4. Down sampling the stored image
   Here the filtered image is down sampled by a given factor. As mentioned above there are no constraints on the down sampling factor, so the capabilities of our processor decide the extent of down sampling factors that can be achieved. Then the down sampled data is also again stored to the same memory unit.

5. Transmission of the processed data from the memory unit of the FPGA back to the computer as a data stream.
   For the visualization of the image, processed data stored in the memory unit should be sent back to the computer as a data stream. For this a UART transmitter module should be implemented in FPGA.

6. Converting the received data stream into an image.
   Received pixel data is converted to a matrix of the size of the expected down sampled image and then displayed using Matlab.

7. Verification of the results

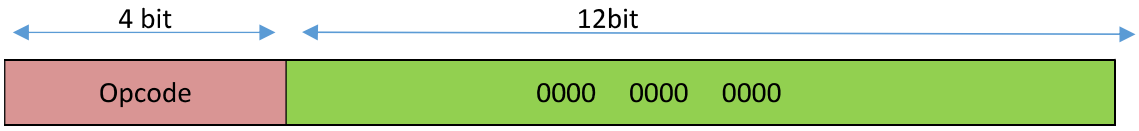
   Original image is processed through the same process of filtering and down sampling by using a MATLAB code and the down sampled output image is obtained. Then the two output images from the two procedures are compared considering the sum of squared error.

The above procedure describes the downsampling process of a grayscale image. Downsampling of a RGB image can be carried out by splitting the image into three planes and carrying out the same process for the three planes separately. According to the task flow described above, core requirements of the design process can be wrapped up as follows;

- Matlab codes for the necessary transmission and verification tasks.
- UART transceiver module implementation in FPGA.
- Memory unit implementation for the storage of pixel data in FPGA.
- Instruction set architecture for the processor which is capable of handling the selected filtering and down sampling algorithms.
- Processor implementation in FPGA.

2. Instruction Set

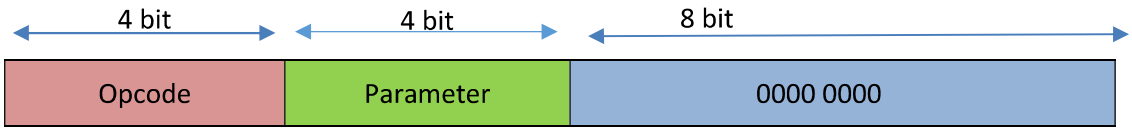
1. NOP



0000

No operation. No. of clock cycles = 1.

2. LOAD

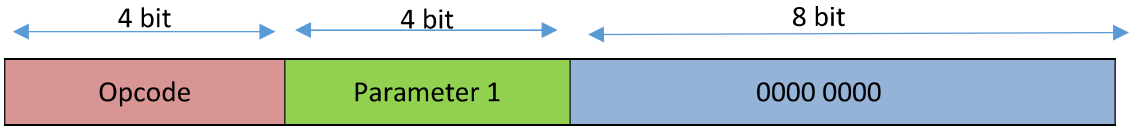


0001

0001 - AC
0010 - RR

Loads data from data memory (DRAM) into MDR. Field, parameter of the instruction takes one of the 2 above mentioned parameters which specifies the source of memory address. Simply it loads the value in the data memory that corresponds to the address in the parameter register, to MDR. No. of clock cycles = 2.

3. STORE

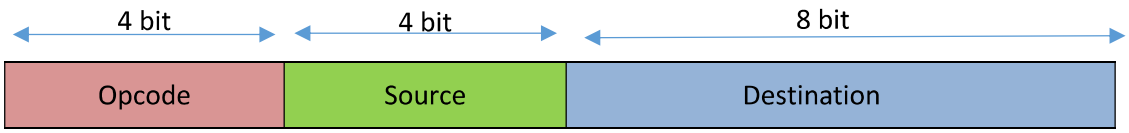


0001

0001 - AC
0010 - WR

Stores data in MDR to data memory. Field, parameter of the instruction takes one of the 2 above mentioned parameters which specifies the source of memory address. Simply it stores the value in MDR to the location in data memory that corresponds to the address in the parameter register. No. of clock cycles = 2.

4. COPY

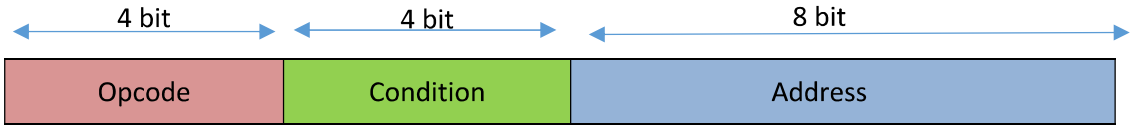


Opcode	Source	Destination
0001	0001 - AC 0010 - R1 0011 - R2 0101 – MDR 0110 – SR1 0111 – SR2 1000 – SR3 1001 – RRR 1010 – CRR	0000 0001 – AC 0000 0101 – MDR 0000 0110 – SR1 0000 0010 – R1 0000 0011 - R2

Copy instruction;

- Copies an 8-bit data from any one of the 8 bit registers presented under source field to 8/16 bit registers presented under destination field.
- Copies a 16-bit data from any one of the 16 bit registers presented under source field to 16 bit registers presented under destination field.
- No. of clock cycles = 2.

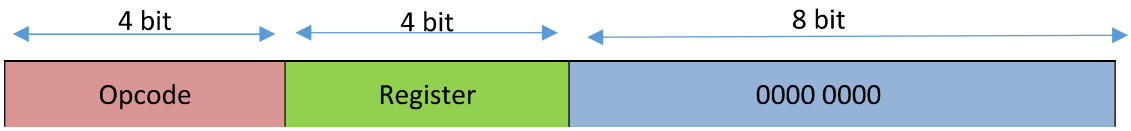
5. JUMP



Opcode	Condition	Address
0100	0000 – Unconditional 0001 – (z=0) (AC≠0) 0010 – (z≠0) (AC=0)	

Jump instruction is used in looping. There are 3 conditions presented under condition filed. Here if the condition comes under the condition field is true a jump is made to the location in the instruction memory that corresponds to the address in the address field of the instruction. No. of clock cycles = 1.

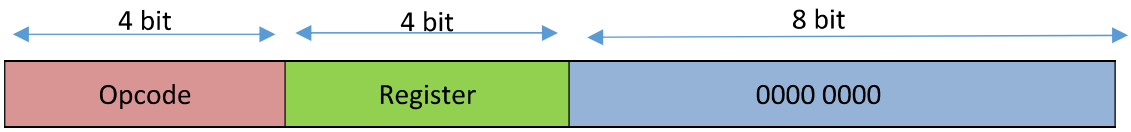
6. ADD



0001 - AC
0010 - R1
0011 - R2
0101 – MDR
0101 0110 – SR1
0111 – SR2
1000 – SR3
1001 – RRR
1010 – CRR

Adds the value in any one of the registers presented under register field of the instruction to the value in AC and the result is stored into AC. No. of clock cycles = 2.

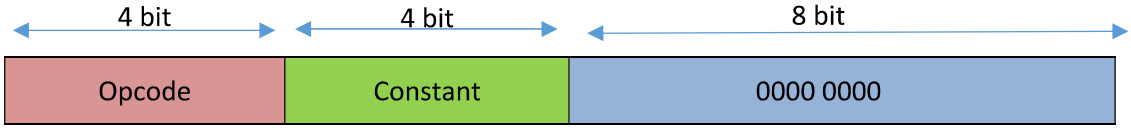
7. SUB



0001 - AC
0010 - R1
0011 - R2
0101 – MDR
0110 0110 – SR1
0111 – SR2
1000 – SR3
1001 – RRR
1010 – CRR

Subtracts the value in any one of the registers presented under register field of the instruction from the value in AC and the result is stored into AC. No. of clock cycles = 2.

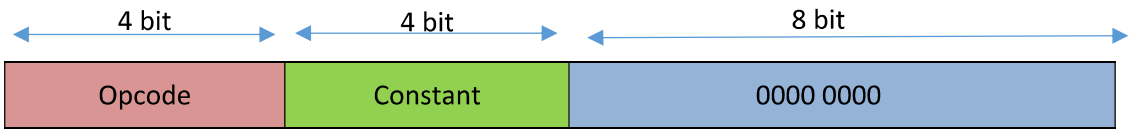
8. MUL



0111

Multiplies the value in AC by a 4 bit constant presented in the constant field of the instruction. No. of clock cycles = 2.

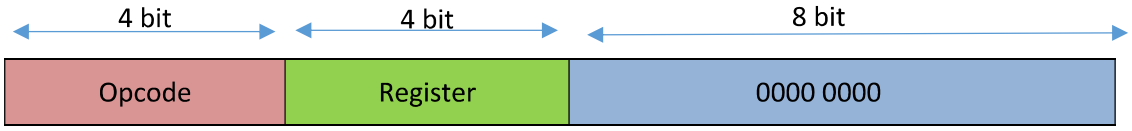
9. DIV



1000

Divides the value in AC by a 4 bit constant presented in the constant field of the instruction. No. of clock cycles = 2.

10. CLR

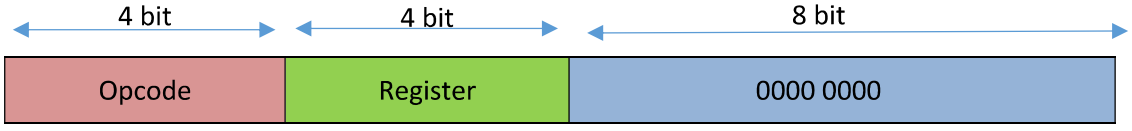


1001

1001 – RRR
1010 – CRR
1011 – RWR
1100 – CWR
0001 - AC

Clears the value in any one of the presented registers under the register field in the instruction. No. of clock cycles = 1.

11. INC

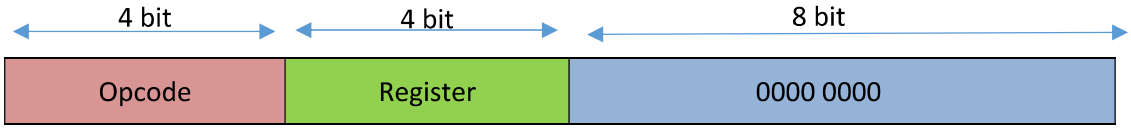


1010

1001 – RRR
1010 – CRR
1011 – RWR
1100 – CWR
0001 - AC

Increments the value by one in any one of the registers presented under the register field in the instruction. No. of clock cycles = 1.

12. DEC

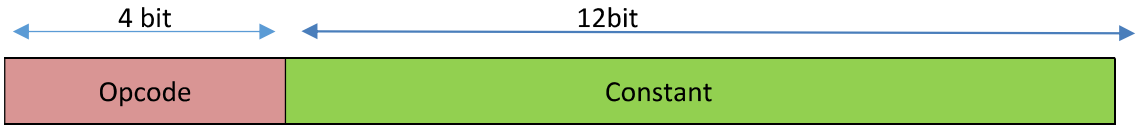


1011

1001 – RRR
1010 – CRR
1011 – RWR
1100 – CWR
0001 - AC

Decrements the value by one in any one of the registers presented under the register field in the instruction. No. of clock cycles = 1.

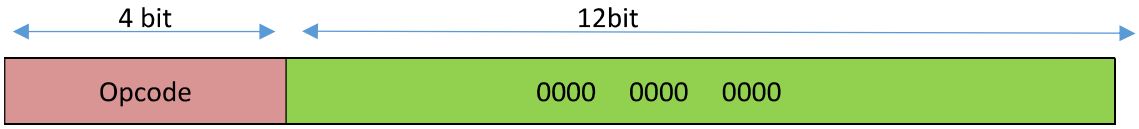
13. LOADK



1100

Loads the 12 bit constant presented in the constant field of the instruction to AC. No. of clock cycles = 1.

14. END



1111

Resets the processor. No. of clock cycles = 1.

3. Registers

Register	Name of the register	Size in bits	Purpose
MAR	Memory address register	16	The memory address which the processor is going to reach next is stored here.
MDR	Memory data register	8	Data read from (or data ready to be written in) the memory is stored here.
IR	Instruction register	16	The instruction retrieved from instruction memory is stored here.
PC	Program counter	8	Stores the address of the next instruction in IRAM.
AC	Accumulator	16	Stores the output of ALU.
Z	Zero flag register	1	Z=1 when AC = 0. Z=0 otherwise.
R1	Register 1	16	General purpose registers.
R2	Register 2	16	General purpose registers
SR1	Shift register 1	8	Shift register with shifting ability.
SR2	Shift register 2	8	Shift register with shifting ability.
SR3	Shift register 3	8	Shift register with shifting ability.
RRR	Row read register	8	Stores the row of the reading pixel. 15 – 8 bits of the pixel address.
CRR	Column read register	8	Stores the column of the reading pixel. 7 – 0 bits of the pixel address.
RWR	Row write register	8	Stores the row of the writing pixel. 15 – 8 bits of the pixel address.
CWR	Column write register	8	Stores the column of the writing pixel. 7 – 0 bits of the pixel address.

Table 3.1 – Details of the Registers

4. Datapath

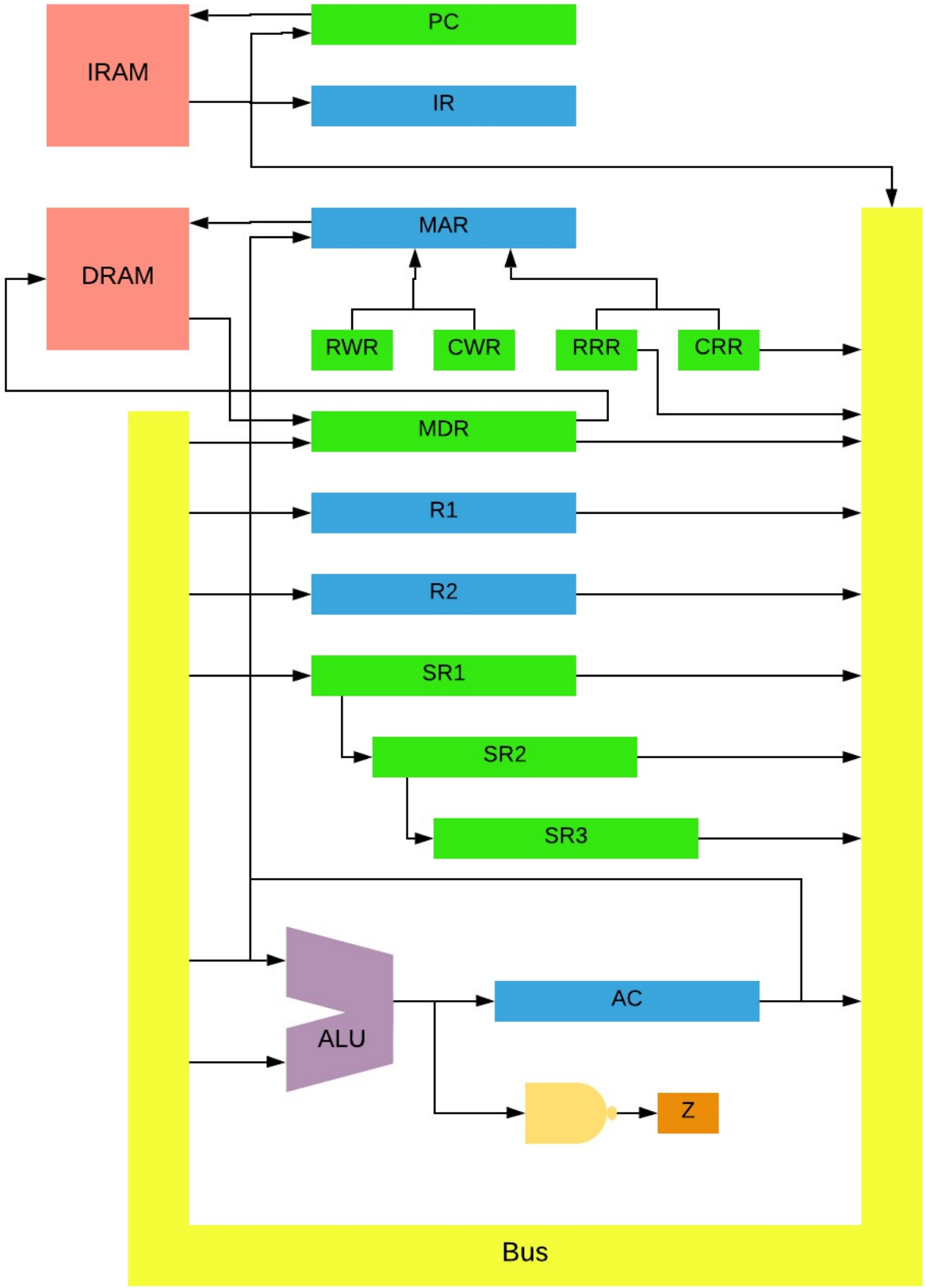


Figure 4.1 – Datapath of the Processor

5. System Architecture

5.1 State Diagram

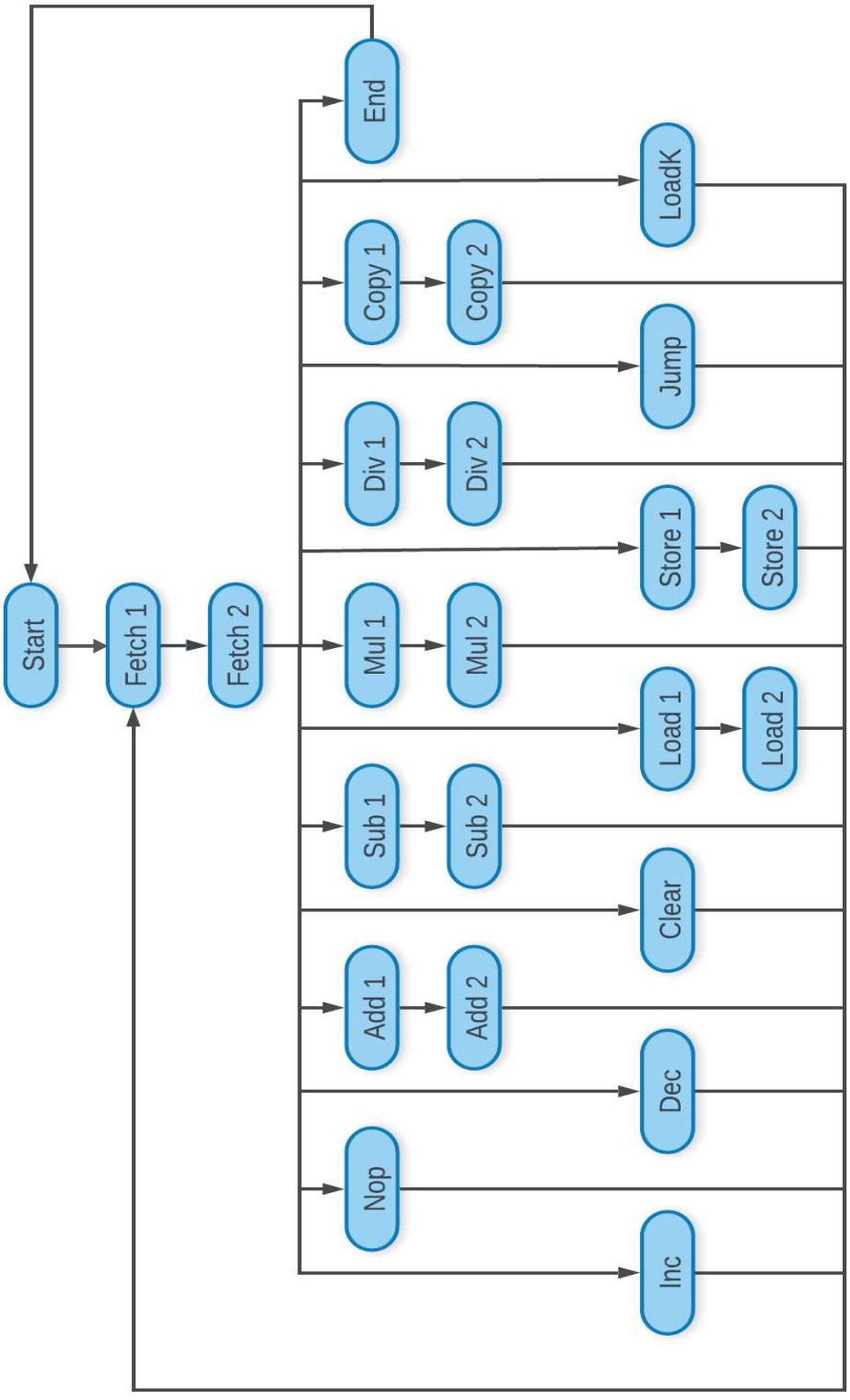


Figure 5.1 – State Diagram of the Processor

5.2 Micro Instructions

Instruction	Opcode	Micro Instructions	State	Operation
-	-	START	16	State <= Fetch 1 (if enable processor is high) State <= Start (if enable processor is low)
-	-	FETCH1	17	Read IRAM (load instruction) State <= Fetch 2
-	-	FETCH2	18	PC <= PC + 1 State <= {4'b0000, opcode}
NOP	0000	NOP	0	No operation State <= Fetch 1
LOAD	0001	LOAD1	1	MAR <= AC or RR (from instruction) State <= Load 2
		LOAD2	24	MDR <= DRAM State <= Fetch 1
STORE	0010	STORE1	2	MAR <= AC or WR (from instruction) State <= Store 2
		STORE2	25	DRAM <= MDR State <= Fetch 1
COPY	0011	COPY1	3	Bus <= Selected source register State <= Copy 2
		COPY2	23	Selected destination register <= Bus State <= Fetch 1
JUMP	0100	JUMP	4	PC <= Address given in instruction (If the condition is satisfied) PC <= PC + 1 (If the condition is not satisfied) State <= Fetch 1
ADD	0101	ADD1	5	Bus <= Selected register State <= Add 2
		ADD2	19	AC <= AC + Bus State <= Fetch 1
SUB	0110	SUB1	6	Bus <= Selected register State <= Sub 2
		SUB2	20	AC <= AC – Bus State <= Fetch 1

MUL	0111	MUL1	7	Bus <= Multiplying constant (from instruction) State <= Mul 2
		MUL2	21	AC <= AC * Bus State <= Fetch 1
DIV	1000	DIV1	8	Bus <= Dividing constant (from instruction) State <= Div 2
		DIV2	22	AC <= AC / Bus State <= Fetch 1
CLR	1001	CLR	9	Selected register <= 0 State <= Fetch 1
INC	1010	INC	10	Selected register <= Selected register + 1 State <= Fetch 1
DEC	1011	DEC	11	Selected register <= Selected register – 1 State <= Fetch 1
LOADK	1100	LOADK	12	AC <= Constant (from instruction) State <= Fetch 1
END	1111	END	15	State <= Start

Table 5.1 – Micro Instructions

5.3 System Architecture

5.3.1 Top Level Module

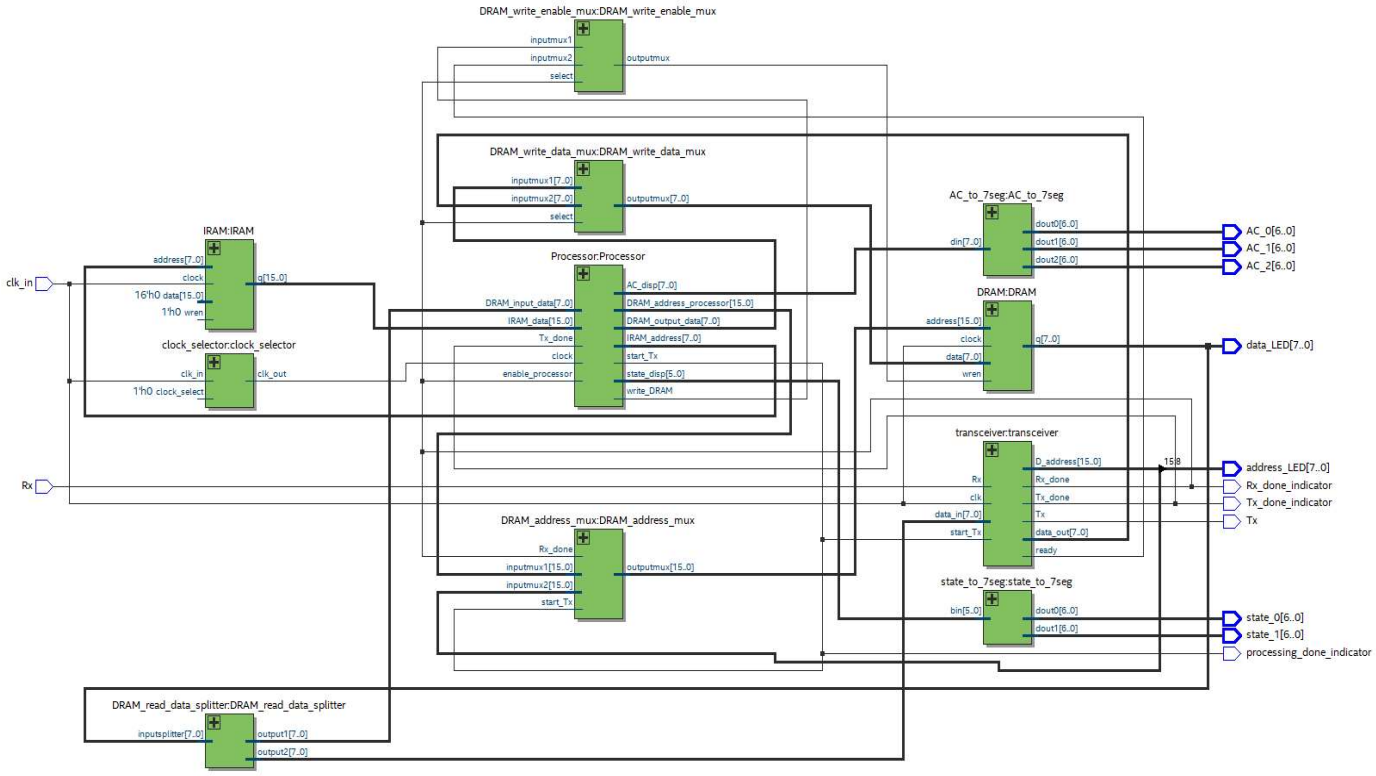


Figure 5.2 – Top Level Module

The top level module combines all the important modules of our implementation which includes the processor, the UART transceiver, data and instruction memories, clock modules, decoders for 7 segment displays and multiplexers. It takes Rx and clk_in as its inputs. The Rx pin connects with the UART interface in the Altera DE2-115 board. The clk_in is connected to the 50 MHz clock available in the board.

It has Tx pin as an output which is also connected to the UART interface of the board. This Rx and Tx pins are connected to the computer through a USB to serial cable which can be used to transmit the image to the FPGA and vice versa. It has 3 output LEDs to display the operation status of the module. These 3 LEDs indicate receiving done, processing done and transmitting done respectively. Two 7 segment modules are connected to the top level module to indicate the state in debugging mode. Three 7 segment modules are used to indicate the value in AC as well.

A set of 8 LEDs are used to indicate the higher 8 bits of the 16 bit data memory address which is currently accessed by the UART transceiver. Another set of 8 LEDs are used to indicate the value stored in that DRAM address. These LED sets are used to debug the UART transceiver module and show the transmission progress in normal operating conditions. The operation of each module inside the top level module are described separately as follows.

5.3.2 Processor

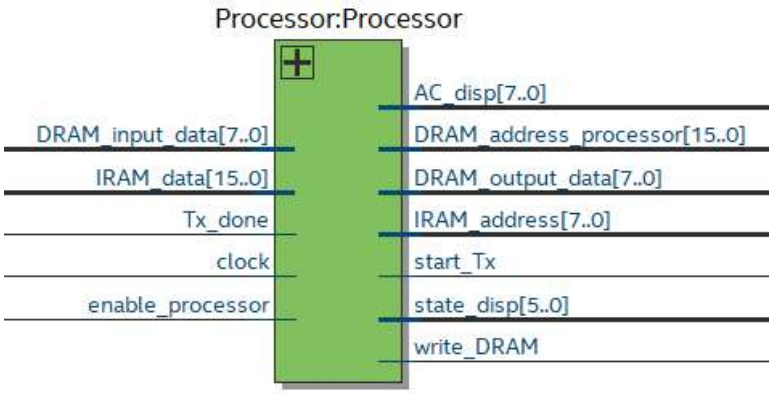


Figure 5.3 – Processor

The processor is the main unit in our implementation which can be used to downsampling and several other operations using the instruction set defined. It is connected to the instruction memory using the IRAM data and IRAM address pins which can be used to fetch instructions. It can read and write data from and to the data memory using DRAM input data and DRAM output data pins respectively and the DRAM address is given by the DRAM address processor pin. Writing to the DRAM is enabled by the write DRAM pin.

It takes the 25 MHz clocked for its operations which is provided by a PLL clock module using the clock pin. Processor only becomes active when the enable processor pin is set to high. Tx done pin is used to identify whether the resultant image is successfully transmitted to the computer. AC disp and state disp pins are used to send the current value in AC and the current state to 7 segment displays through decoders which can be used for debugging purposes. Start Tx pin is used to indicate the processing is completed thus the transceiver can start transmitting the image.

The processor comprises of a state machine which implements the fetch decode and execute cycle. Several registers are used to store values and memory addresses in intermediate operations which together forms the data path. An arithmetic and logic unit is used to perform mathematical operations. State machine and other control modules ensure that each instruction is executed in the desired manner.

5.3.2.1 State Machine

The state machine is the main control unit of the processor which ensures the correct control signals are given to the each unit inside the processor. The state machine switches its state at each negative edge of the clock cycle depending on the instruction to realize the fetch decode and execute cycle depending on the state diagram. In each state the necessary control signals are provided for the each unit. State machine takes four inputs; clock, instruction, Tx done and enable processor. State machine is synchronized with the 25 MHz clock. Fetched instruction is read to determine the next state. At the beginning stating machine is at the start state and jumps to fetch 1 state when enable processor is set to high. Tx done is used to reset the processor back to the start state.

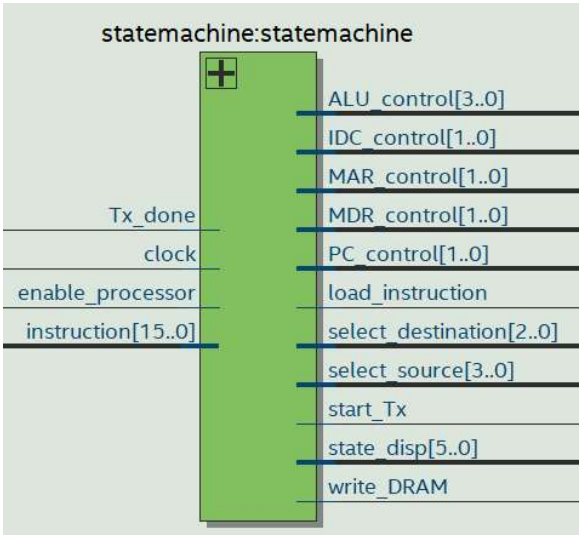


Figure 5.4 – State Machine

State machine outputs several control signals to the other modules inside the processor. ALU control is used to select the operation of the Arithmetic and the Logic Unit depending on the current state. IDC control determines the control signals sent to the IDC controller to execute increment decrement and clear instructions. MAR control and MDR control signals are sent to MAR and MDR respectively to select the writing source of the contents of the register. PC control is used to increment PC or change PC in jump instructions subjected to the jump condition.

Load instruction is used to read IRAM and write the instruction register. Select destination and select source outputs are used to select the input to the bus and the register that will be written from the bus. Start Tx control signal is given to the UART transceiver to start transmission when the processing is completed. State disp is used to display the current state in 7 segments which is used for debugging purposes. Write DRAM signal is used to enable writing operation in the DRAM.

5.3.2.2 Arithmetic and Logic Unit (ALU)

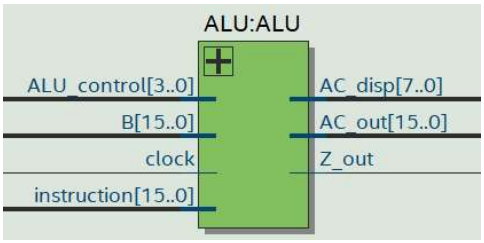


Figure 5.5 – ALU

Arithmetic and Logic Unit performs all arithmetic and logical operations inside the processor. In our instruction set, it is possible to perform addition, subtraction, multiplication and division which is sufficient to perform downsampling and other filtering operations. The 16 bit AC register is inside the same module which stores the result of the ALU. The ALU control decides the operation of the ALU or it is used to change the value in the AC register. The B input provides the data bus as the second input the ALU and the first input is the AC register. It also reads the instruction to load constant values to the AC directly from the instruction.

The 1 bit Z register is also inside the same module which flags whether the result obtained by the ALU is zero or not. Z flag is high when the result is zero and low otherwise. It is used by the PC module to check the jump condition in jump instructions. AC out provides the value in the AC to other required modules and AC disp is used to display the lower 8 bits stored in AC in 7 segment display for debugging purposes. The subtraction operation performed in the ALU provides the modulus difference of the two values since signed integers are not used in our implementation.

5.3.2.3 Bus

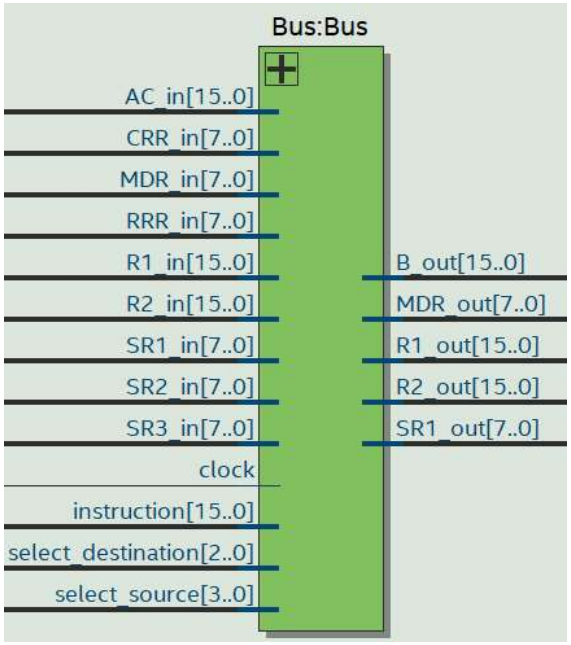


Figure 5.5 – Bus

The bus module is used for transferring data between the registers in the processor. It also provides the second input to the ALU through the B out. The data bus can be loaded from many registers including AC, CRR, MDR, RRR, R1, R2, SR1, SR2 and SR3. Data bus is 16 bits and when 8 bit registers are loaded into the bus the first 8 bits are set to zero. MDR, R1, R2 and SR1 registers can be written from the bus. The source and the destination registers of the bus is selected from the state machine using the select source and select destination pins. Bus module also reads the instruction to separate 4 bit constant values which are provided into the ALU in multiply and divide instructions.

5.3.2.4 Increment, Decrement and Clear (IDC) Controller

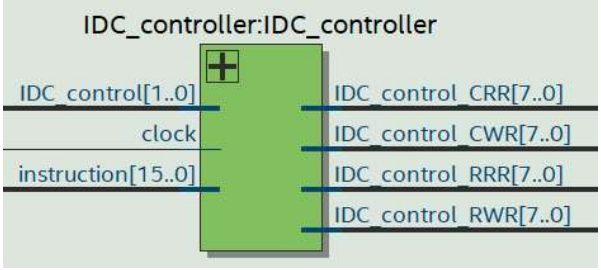


Figure 5.6 – IDC controller

The IDC controller is a control unit which is used to increment, decrement and clear CRR, CWR, RRR and RWR registers. These four registers determine the pixel which is being read or written. Iterating over the image is done through by incrementing, decrementing and clearing these four registers. The IDC control specify which operation should be performed and the register is selected from reading the instruction. It output four wires to CRR, CWR, RRR and RWR which use the updated value by the IDC controller to modify their contents.

5.3.2.5 Instruction Register (IR)

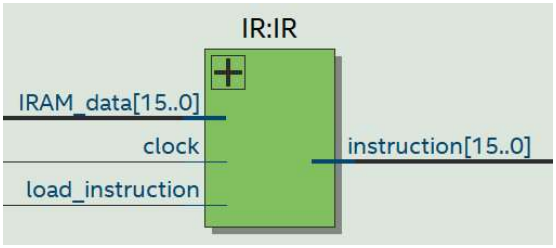


Figure 5.7 – Instruction Register

The 16 bit instruction register stores the instruction loaded from the IRAM in the fetch cycle. IRAM data changes the contents of the instruction register when load instruction is set to high and the module is synchronized with the clock. Output of the instruction register is used by many modules to extract the opcode, registers, constants and other required parameters from the instruction.

5.3.2.6 Program Counter (PC)

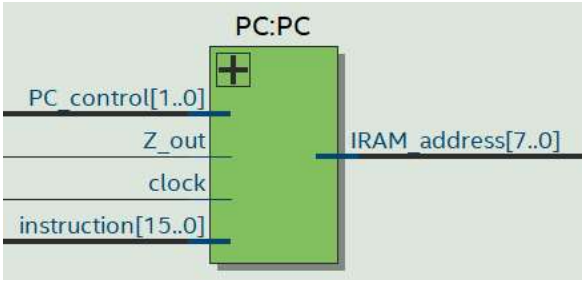


Figure 5.8 – Program Counter

The 8 bit program counter points to the next IRAM address which should be loaded into the instruction register in the next fetch cycle. The operation of the PC is decided by the PC control. In normal instructions PC increments by 1 whenever an instruction is read from IRAM and points to the instructions. In jump instructions, PC is modified according to the jump address subjected to the jump condition. Both parameters jump address and jump condition is extracted by the instruction.

It also takes the value of the Z flag as an input to determine whether the jump condition is met. The module is synchronized with the clock. The output of the PC is connected directly to the instruction memory and by changing the value in the PC, the required instruction can be loaded into the instruction register from IRAM in the next fetch cycle.

5.3.2.7 Memory Address Register (MAR)

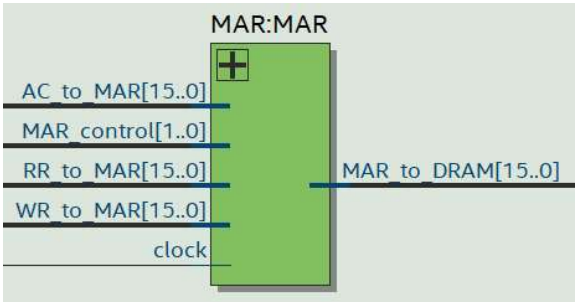


Figure 5.9 – Memory Address Register

The 16 bit memory address register stores the memory address of the DRAM which should be loaded into the memory data register or which should be written from the contents in the MDR. The contents of MAR can be updated from three sources; AC, read registers and write registers. The module is synchronized with the clock and the MAR control decides the source to modify the value stored in MAR. The output of MAR is directly connected to the address pin of the DRAM.

5.3.2.8 Memory Data Register (MDR)

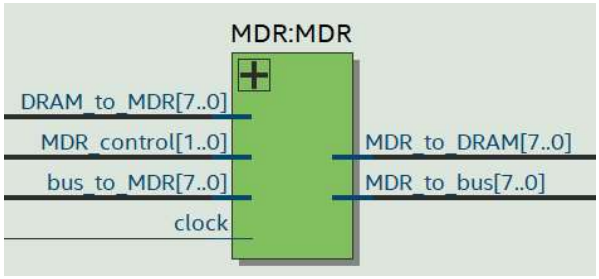


Figure 5.10 – Memory Data Register

The 8 bit memory data register stores the data loaded by the DRAM or data which should be written to the DRAM. The contents of MDR can be updated from two sources; DRAM and the data bus. The module is synchronized with the clock and the MDR control decides the source to modify the value stored in MDR. The output of MDR is connected to the data in pin of the DRAM and it can also be used as an input to the data bus.

5.3.2.9 Shift Registers (SR1, SR2 and SR3)

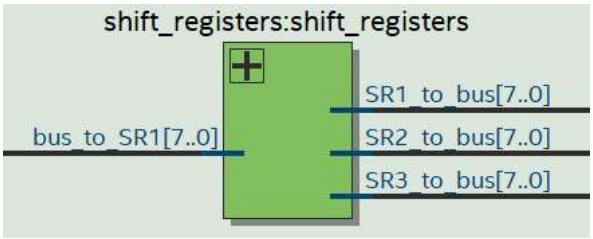


Figure 5.11 – Shift Registers

The 8 bit shift registers are used to store pixel values from the image and it becomes important to store original pixel values inside the processor when pixels are modified with processed values when applying filters. Since we are overwriting the original image from the resultant image there will be occasions where data needed for the processor has been overwritten by the new values. Shift registers address this problem. The SR1 can be written from the data bus and when it is overwritten the previous value stored in the SR1 will be stored in SR2. Similarly SR3 will be updated by the previous value stored in SR2. All three 8 bit registers can be loaded into the bus separately.

5.3.2.10 General Purpose Registers (R1 and R2)

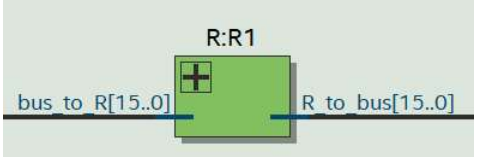


Figure 5.12 – General Purpose Register

The 16 bit general purpose registers R1 and R2 can be used to store 16 bit values. The content in R1 and R2 can be modified by the bus and the contents can be loaded into the data bus as well.

5.3.2.11 Read Registers (RR)

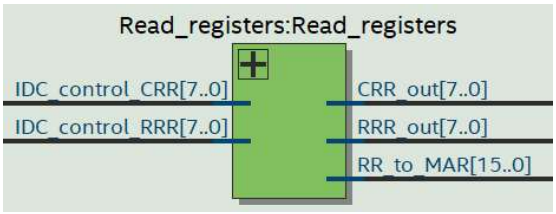


Figure 5.13 – Read Registers

The read registers module contains the column read register (CRR) and row read register (RRR). The contents of the two registers can be only modified by the IDC control unit i.e. these registers can only be incremented, decremented or cleared. The combined output of CRR and RRR is given to the MAR and the data bus can be loaded by CRR or RRR separately.

5.3.2.12 Write Registers (WR)

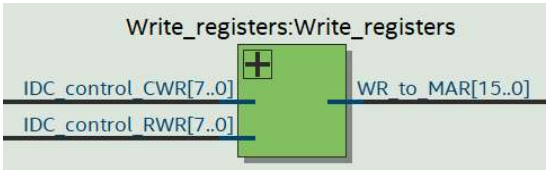


Figure 5.14 – Write Registers

The write registers module contains the column write register (CWR) and row write register (RWR). The contents of the two registers can be incremented, decremented or cleared by the IDC controller. The combined output of CWR and RWR is given to the MAR.

5.3.3 Memory

In our implementation we use separate instruction and data memories to store instructions and data separately.

5.3.3.1 Instruction Memory

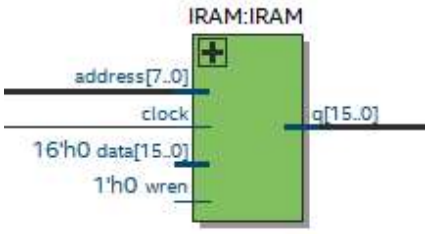


Figure 5.15 – Instruction Memory

Instruction memory have 256 memory locations and each memory location can store a 16 bit value. The instruction memory can only be read by the processor thus the data out pin is connected to the instruction register and the 8 bit address pin is connected to the program counter. Values in the instruction memory is initialized by a memory initialization file (mif format) and cannot be modified later in the operation. Thus the data in pin and write enable pin are not connected. Instruction memory is created using IP Catalog.

5.3.3.2 Data Memory

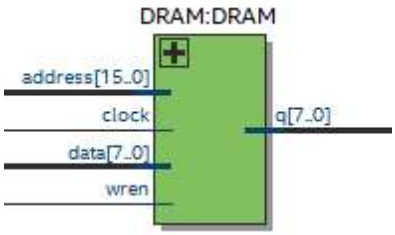


Figure 5.16 – Data Memory

Data memory have 65,536 memory locations and each memory location can store an 8 bit value. The data memory is used to store the 256*256 image loaded by the UART transceiver and the modified image is also stored in the same memory overriding the previous contents. The data memory can be read by both the processor and the UART transceiver. Thus the data out pin is connected to a splitter and the 16 bit address pin is connected to a multiplexer which determines the source.

Values in the data memory is first initialized by the received image by the UART transceiver. Then depending on the processor operation values in the data memory get modified. After the processor operation data memory is read by the UART transceiver to transmit the processed image to the computer. Thus the data in pin and write enable pin are also connected to multiplexers which selects the source processor or the transceiver accordingly. Data memory is also created using IP Catalog.

5.3.4 Communication Modules

Since our processor was designed with minimum possible memory resources, the three planes of the RGB image are loaded into the memory one after the other and the receiving, processing and transmitting cycle was executed for the three planes separately. To process RGB images, a reliable communication method was crucial to transmit and receive the three planes to and from the FPGA development board. Accuracy of this communication method is critical since, even though the processor performed perfectly, performance of the processor is evaluated using output data from the communication module and can be affected by bit errors occurred in transmission.

In our implementation we have used the UART protocol for the communication between the processor and the computer. As the communication medium we have used a USB to RS232 cable to connect the FPGA development board to the computer. In the FPGA we have designed a UART transceiver module to communicate with the computer. In the computer we have used a MATLAB program to access the serial link.

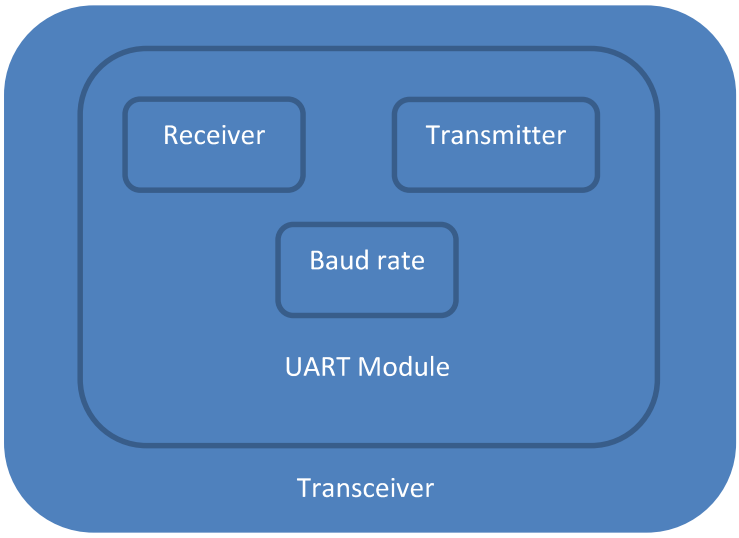


Figure 5.17 – Communication Modules

In our implementation, there are two low level modules for the receiver and the transmitter. Baud rate module is responsible for giving timing signals to the receiver and transmitter according to the desired baud rate. In our implementation, we have used 115,200 as the baud rate. In the transmitter, there are 4 states and one bit is transmitted for 434 clock cycles (50 MHz clock). After each byte the output is kept high for 1 bit period before transmitting the next byte. At the end of the transmission of each byte there is an indicator to indicate successful transmission. The receiver also has 4 operating states and it samples each incoming bit 16 times and takes the 8^{th} sample. This is done to remove errors like false 1 -> 0 transitions in the beginning.

UART module consists of all the above 3 modules and it has input and output buses and serial Rx and Tx wires with Tx busy flag and Rx available flag. The transceiver module is responsible for storing received data in the memory in a systematical way and transmitting the processed image to the computer in the end.

5.3.4.1 Receiver

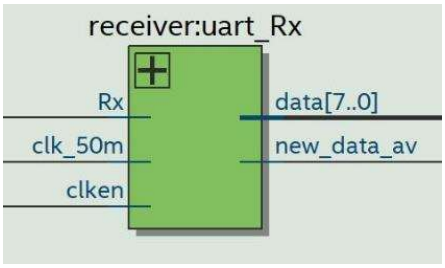


Figure 5.18 – Receiver

This module takes bits serially communicated through the Rx pin and output a byte through the data pin when 8 bits are received. Then new data available flag is changed to high indicating that a new byte is available at the data pin. Clock enable is the output from the baud-rate module and clock 50m is the 50 MHz clock signal.

5.3.4.2 Transmitter

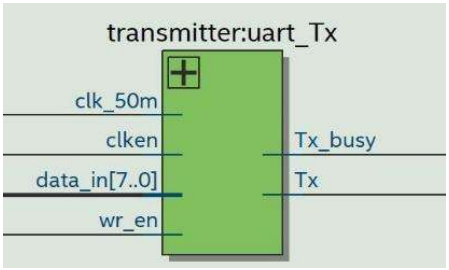


Figure 5.19 – Transmitter

This module transmits the received byte from data in pin serially through the Tx pin. Tx busy pin indicates whether the module is currently transmitting a byte (turns to logic high). Clock enable pin is connected to the signaling wire from the baud rate module. Start transmission signal is given by changing the write enable pin from high to low for a single clock cycle.

5.3.4.3 Baud Rate

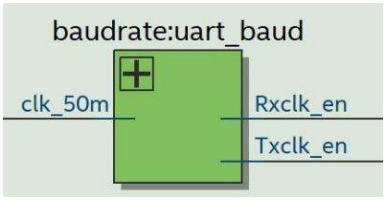


Figure 5.20 – Baud Rate

This module divides the main clock according to the desired baud rate using counters to give timing signals to the receiver and the transmitter. For the Rx clock enable, the input 50 MHz clock is divided by a factor of 27 and for the Tx clock enable the input 50 MHz clock is divided by a factor of 434.

5.3.4.4 UART Module

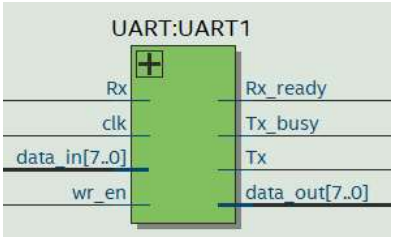


Figure 5.21 – UART Module

UART module interconnects the above three modules. (Transmitter, receiver and baud rate modules)

5.3.4.5 Transceiver

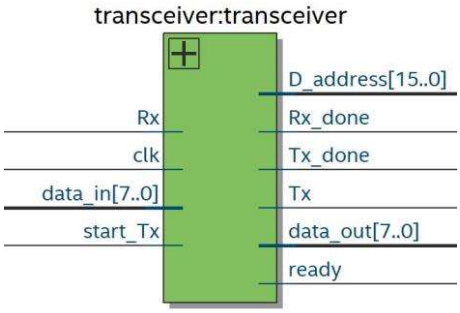


Figure 5.22 – Transceiver

This module controls the UART module and it stores the data received in the data memory and transmit the stored data after processing is completed. Transceiver module was designed in a particular manner that after each transmission of a 256*256 grayscale image or a single plane from a 256*256 RGB image, it could be reset to the original state by sending a byte from the computer thus it can receive the next image or the next plane.

When transmitting and storing the image pixels are stored by iterating through the first row and then going to the next row format. Advantage of this system was, easily callable pixel address structure in memory. As an example, pixel with the row address M (8 bits) and column address N (8 bits) is stored in the memory location with the {M, N} address.

	1	2	3	4	5	6	7
1							
2							
3							
4							
5							

00000011 00000011

Figure 5.23 – Memory Organization

5.3.5 Other Modules

5.3.5.1 Clock Selector

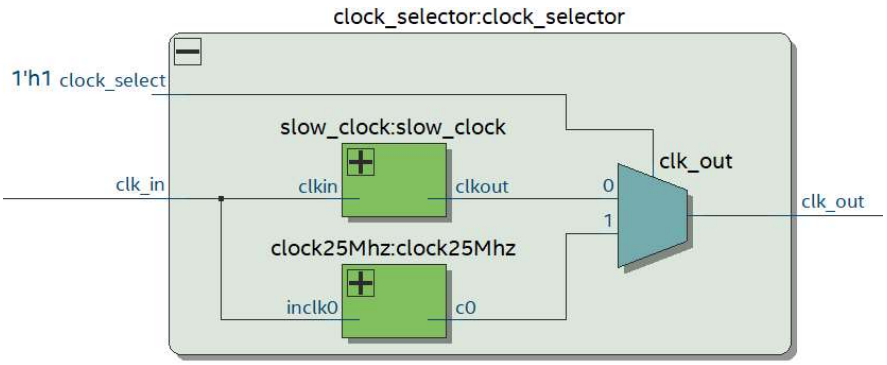


Figure 5.24 – Clock selector

Clock selector module is used to select the clock for the processor. In normal operating conditions 25 MHz clock is used and when debugging a slower clock (around 1Hz) is used. The 25 MHz clock is generated by phase locked loops using IP Catalog. Slow clock is generated by incrementing counters. Both clocks take the 50 MHz clock as their input clock. Clock select pin can be set to high or low to select the required clock.

5.3.5.2 DRAM Address Mux

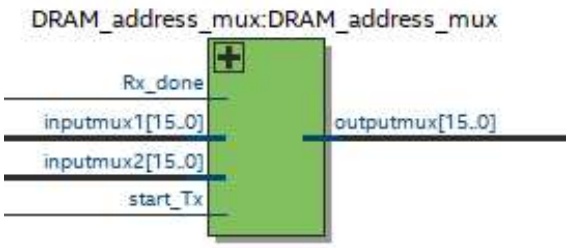


Figure 5.25 – DRAM Address Mux

DRAM address multiplexer is used to select the source of the address pin of the DRAM. Address can be given by the processor or the transceiver. In image receiving stage and image transmitting stage transceiver should be the source. In processing stage processor should be the source. Rx done and start Tx pins are used to identify the stage since Rx done will change from low to high when stage transits from receiving to processing and start Tx will change from low to high when stage transits from processing to transmitting.

5.3.5.3 DRAM Read Data Splitter

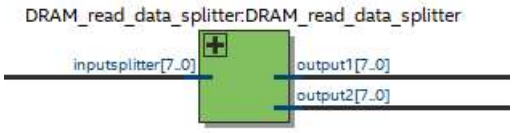


Figure 5.26 – DRAM Read Data Splitter

DRAM read data splitter is used to split the data out wire of the DRAM to the processor and the transceiver.

5.3.5.4 DRAM Write Data Mux

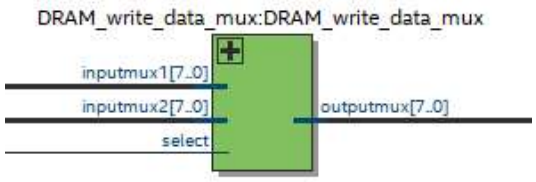


Figure 5.27 – DRAM Write Data Mux

DRAM write data multiplexer is used to select the source of the data in pin of the DRAM. Data in can be given by the processor or the transceiver. In image receiving stage transceiver should be the source. In processing stage processor should be the source. Rx done pin is used to identify the stage since Rx done will change from low to high when stage transits from receiving to processing.

5.3.5.5 DRAM Write Enable Mux

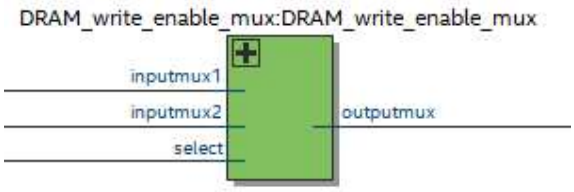


Figure 5.28 – DRAM Write Enable Mux

DRAM write enable multiplexer is used to select the source of the write enable pin of the DRAM. Write enable can be given by the processor or the transceiver. In image receiving stage transceiver should be the source. In processing stage processor should be the source. Rx done pin is used to identify the stage since Rx done will change from low to high when stage transits from receiving to processing.

5.3.5.6 State to 7 Segment Display Module

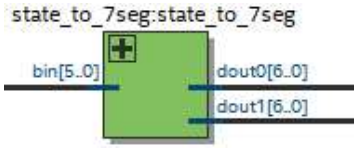


Figure 5.29 – State to 7 seg Module

This module is used to decode the last 6 bits of the state of the processor to display in the 7 segment displays. Two 7 segment displays are used since the decimal value of the state doesn't exceed 25.

5.3.5.7 AC to 7 Segment Display Module

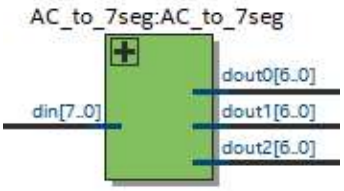


Figure 5.30 – AC to 7 seg Module

This module is used to decode the last 8 bits of the value in the AC register to display in the 7 segment displays. Three 7 segment displays are used since the decimal value of the state doesn't exceed 255.

6. Timing Analysis

The processer architecture is designed to output all control signals to other modules by the state machine at the negative clock edge. The data transmission between modules (read/write operations) occur at the positive clock edge. Therefore no conflicts will arise since control signals are given to the modules when data is available. The following timing diagrams visualize the execution of each instruction including start and fetch cycles.

1. Start and Fetch Cycle

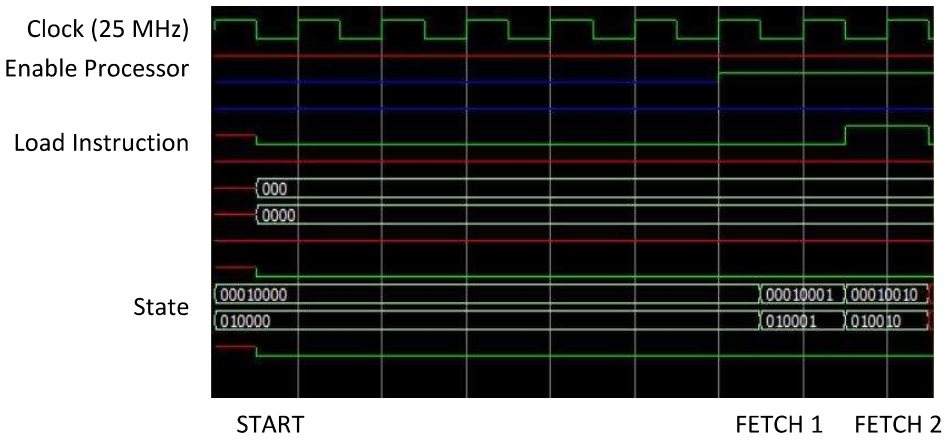


Figure 6.1 – Timing Diagram of Start and Fetch Cycle

As is figure 6.1, in the beginning processor is in the start state, where it is waiting until enable processor signal to jump from low to high. When enable processor signal is changed to high, the processor jumps to fetch 1 state in the negative edge of the next clock pulse and it jumps to fetch 2 state in the negative clock edge of the next clock pulse.

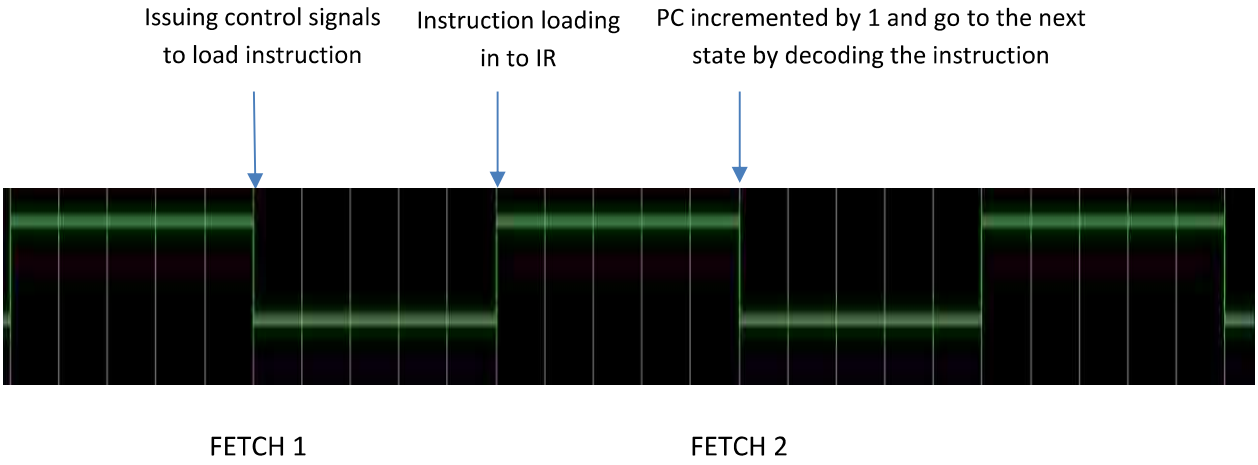


Figure 6.2 – Simplified Timing Diagram of Fetch Cycle

2. NOP

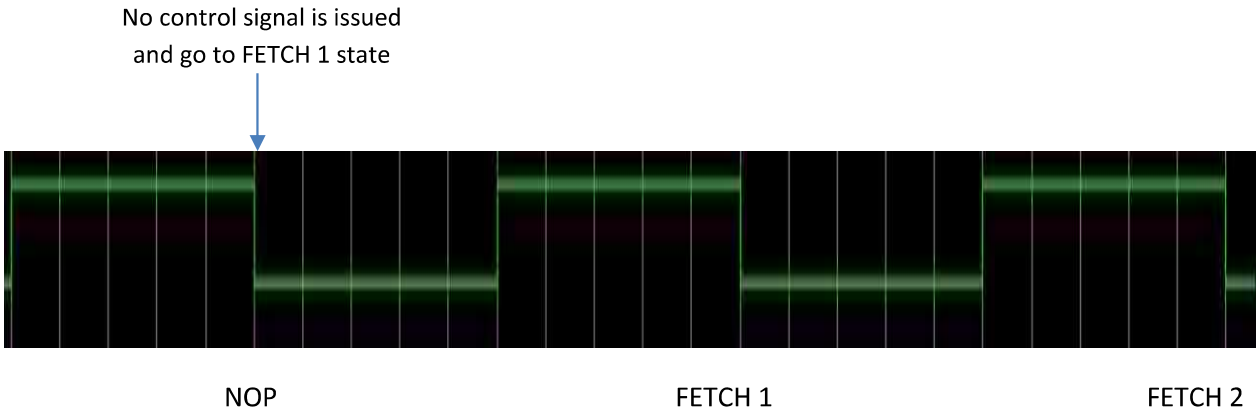


Figure 6.3 – Timing Diagram of Nop Instruction

3. ADD

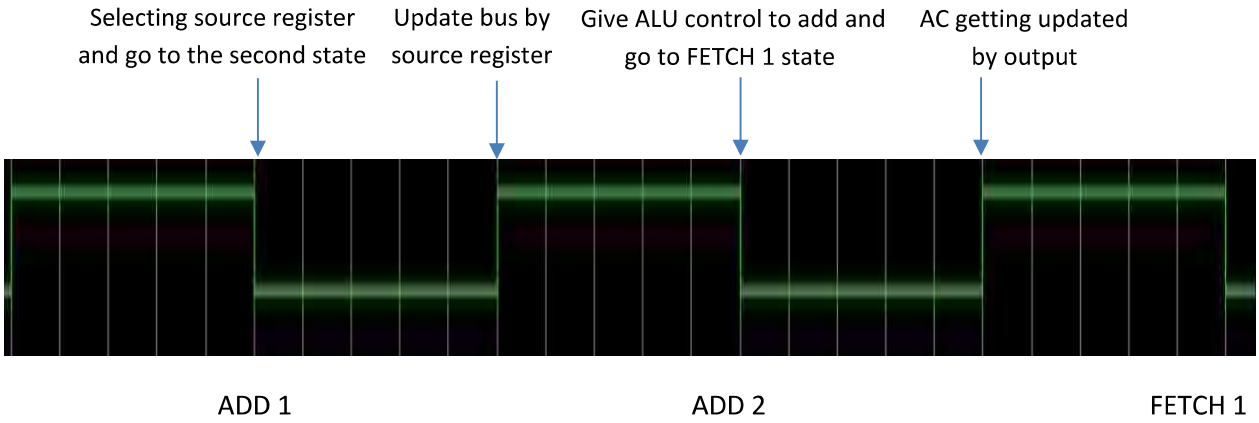


Figure 6.4 – Timing Diagram of Add Instruction

4. SUB

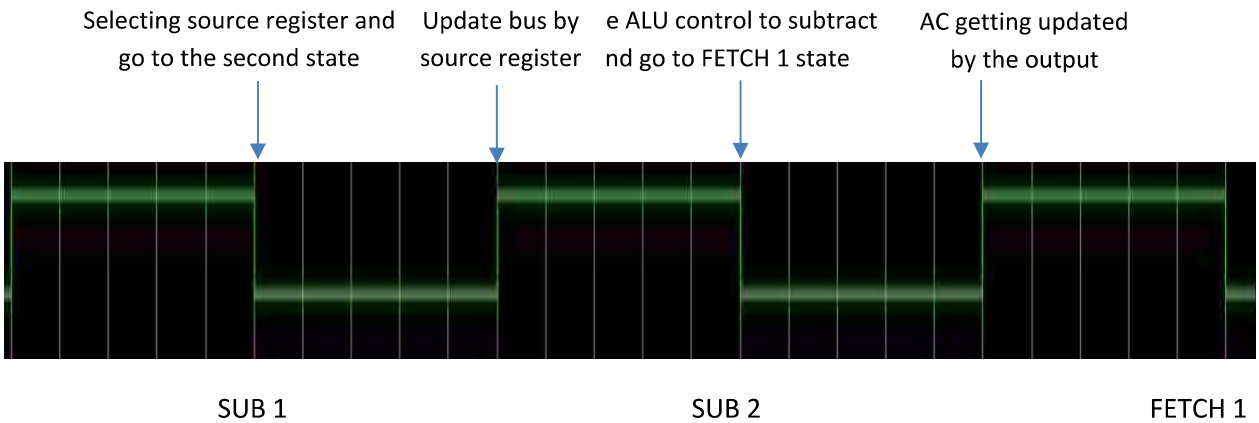


Figure 6.5 – Timing Diagram of Sub Instruction

5. MUL

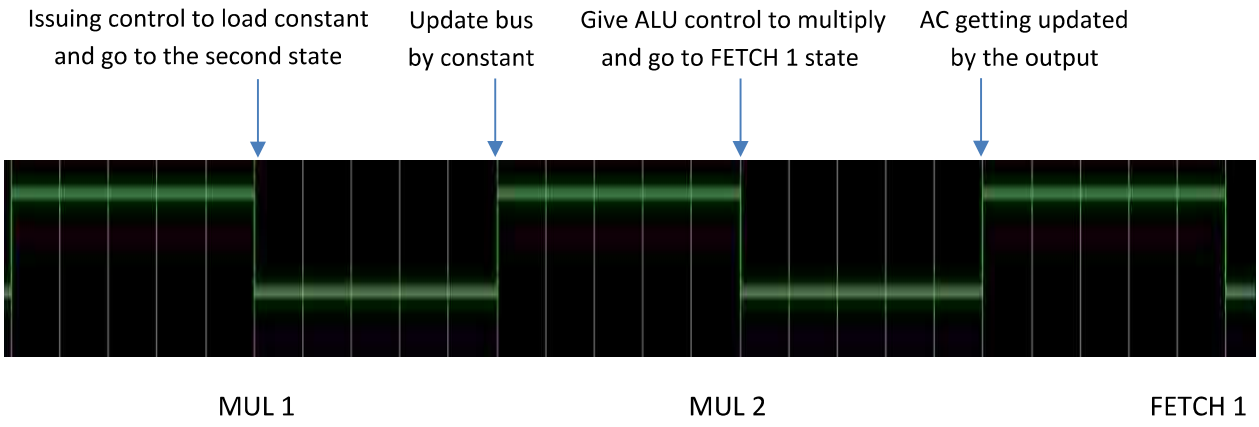


Figure 6.6 – Timing Diagram of Mul Instruction

6. DIV

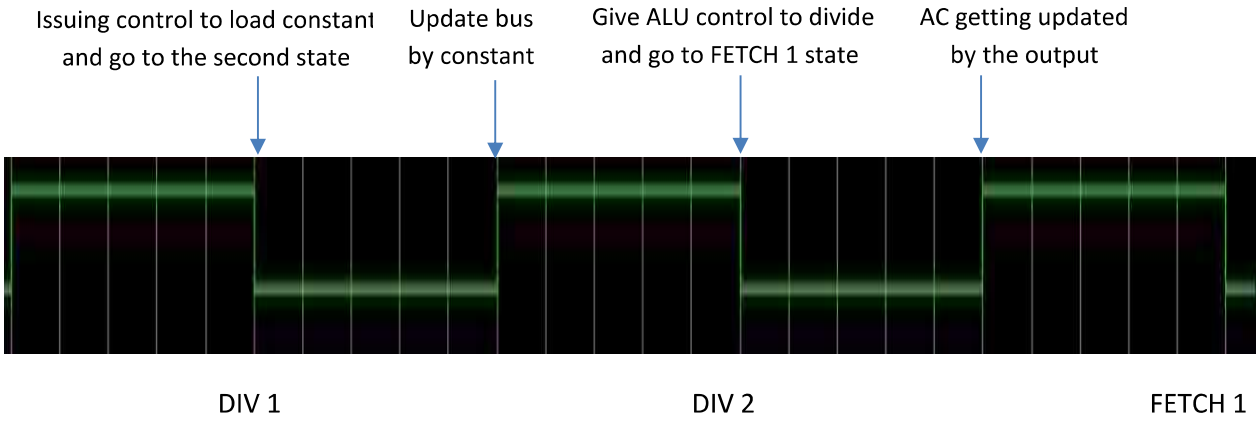


Figure 6.7 – Timing Diagram of Div Instruction

7. COPY

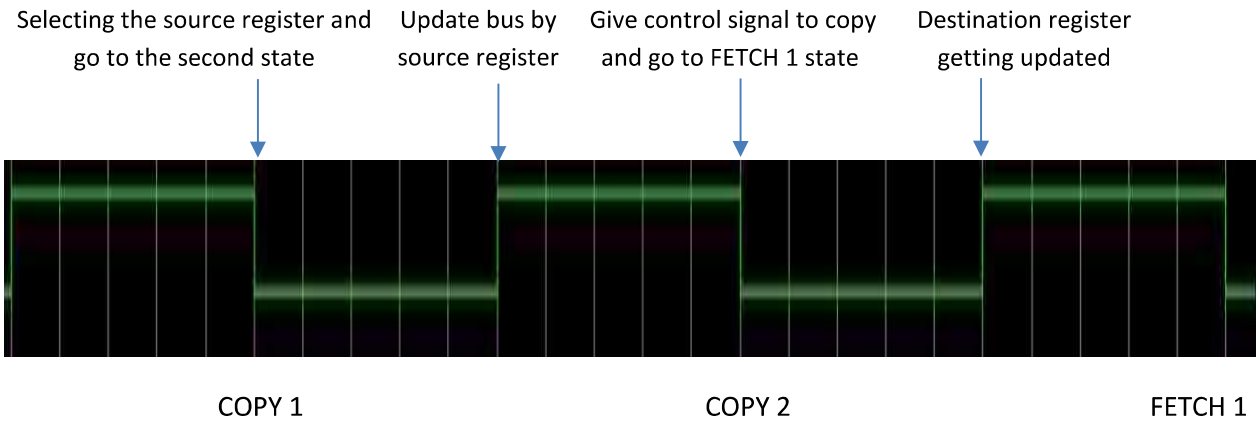


Figure 6.8 – Timing Diagram of Copy Instruction

8. LOADK

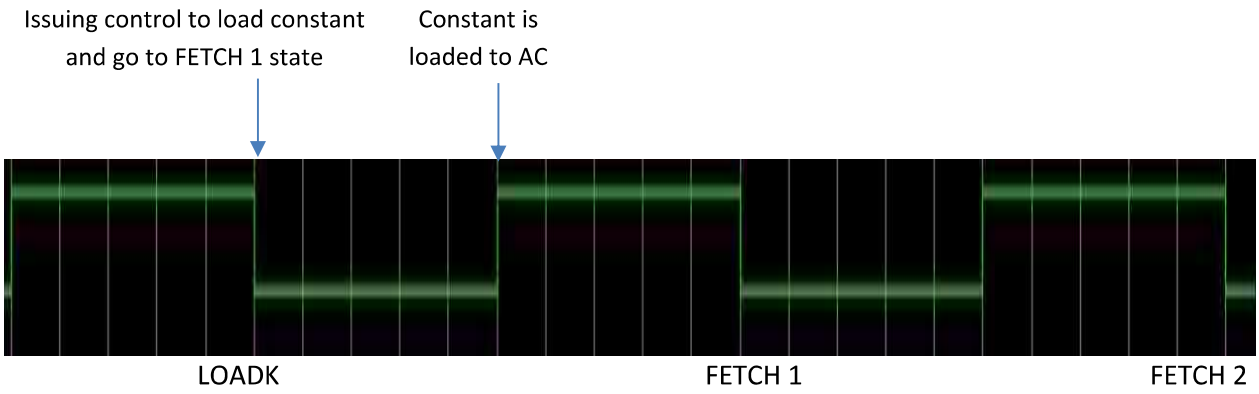


Figure 6.9 – Timing Diagram of LoadK Instruction

9. JUMP

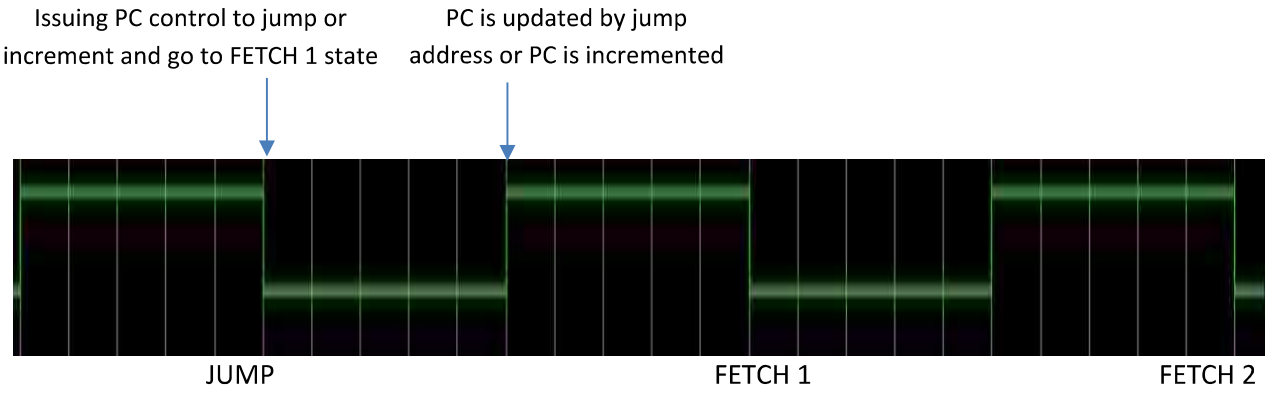


Figure 6.10 – Timing Diagram of Jump Instruction

10. INC

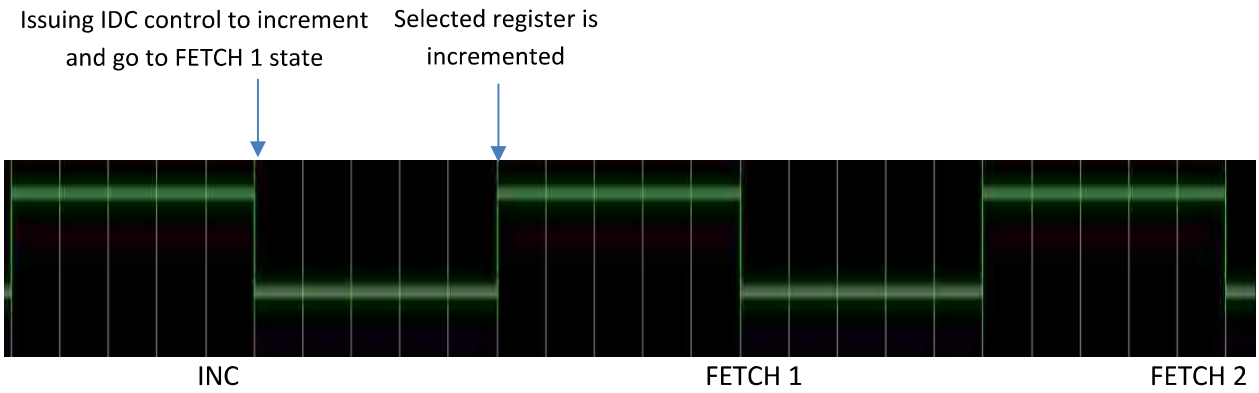


Figure 6.11 – Timing Diagram of Inc Instruction

11. DEC

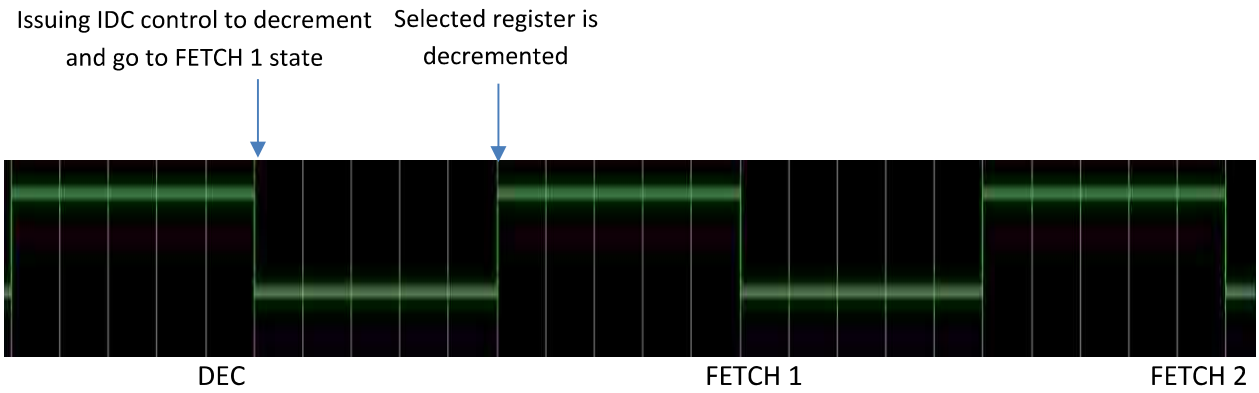


Figure 6.12 – Timing Diagram of Dec Instruction

12. CLR

Figure 6.13 – Timing Diagram of Clr Instruction

13. LOAD

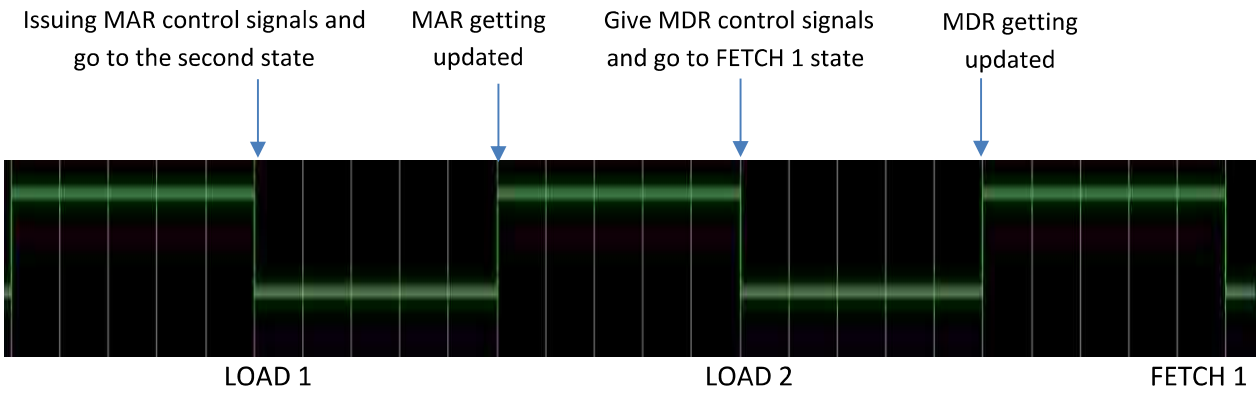


Figure 6.14 – Timing Diagram of Load Instruction

14. STORE

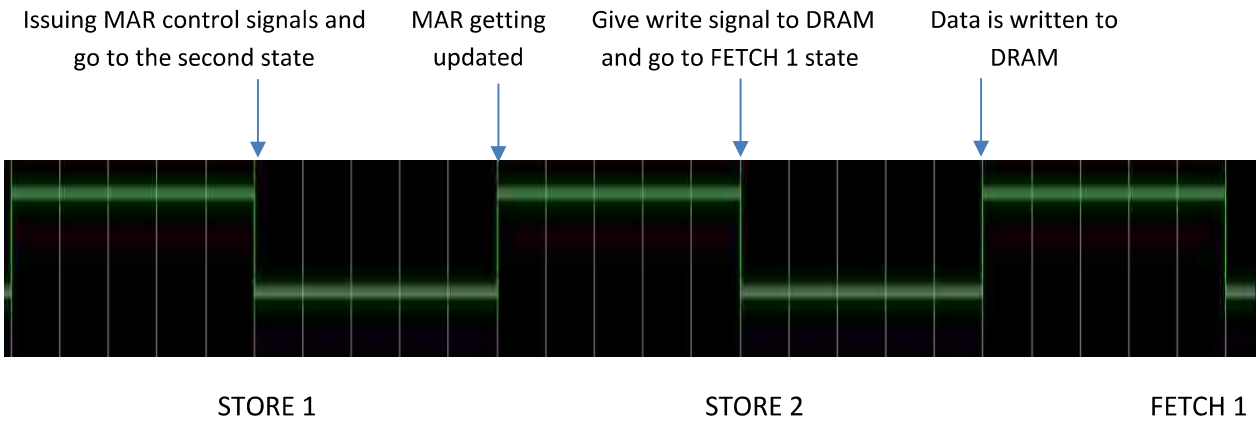


Figure 6.15 – Timing Diagram of Store Instruction

15. END

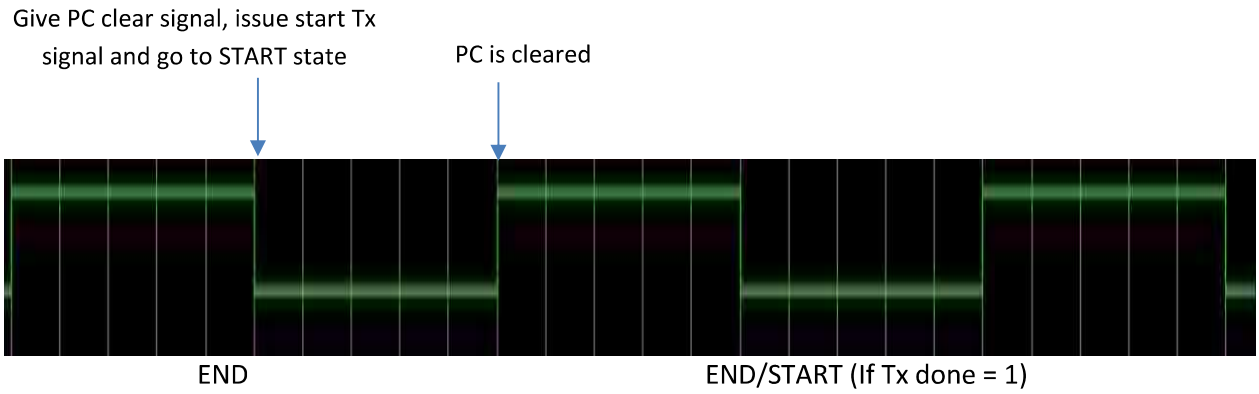


Figure 6.16 – Timing Diagram of End Instruction

The Start Tx signal is given at the END state and it waits in the same state until Tx done signal to become high to go back to the START state.

7. Algorithms

7.1 Downsampling using an Average Filter

The down sampling process must inherit the properties of the original image to the final image. For this to happen, the high frequency components of the original image should be addressed. Therefore a low pass filtering will rectify the effect of high frequencies in the image. The low pass filter we are using under this method is an average filter.

The average filter is a linearly separable filter which means that the filter can be first implemented row wise and then column wise. Four pixels in a neighborhood square are averaged to get the final value of the top left most pixel in the square. To use the linear separability property, the columns are chosen alternatively and the column next to them is added on to them. Then the average is taken and the chosen column is replaced by this averaged column.

The same procedure is then carried out row wise. Since only the averaged rows and columns are selected and saved, the original image size is halved. Therefore both the low pass filtering and the down sampling objectives have been merged into a single code. In our implementation this method is only used to down sample by a factor of two and for higher down sampling factors the Gaussian filter is used. The down sampling process using an average filter is illustrated in the following diagram.

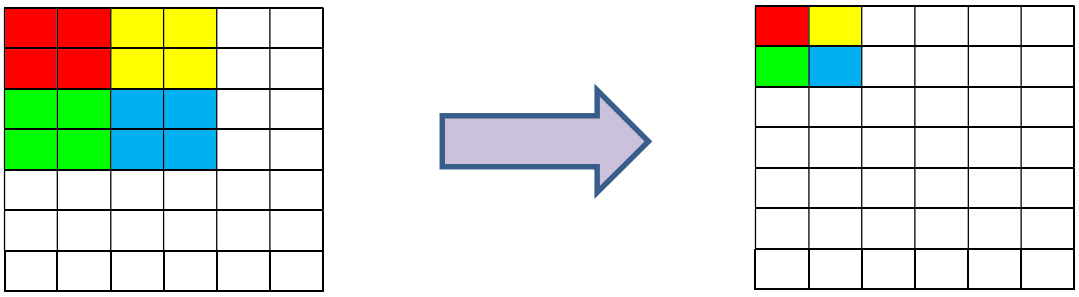


Figure 7.1 – Downsampling using an Average Filter

7.1.1 Assembly Code for Downsampling using an Average Filter

```
(1)  LOADK 255
(2)  COPY AC R1
(3)  CLR RRR
(4)  CLR RWR
(5)  CLR CRR
(6)  CLR CWR
(7)  LOAD RR
(8)  COPY MDR RR
(9)  INC CRR
(10) LOAD RR
(11) COPY MDR AC
(12) ADD R2
(13) DIV 2
```

(14) COPY AC MDR
(15) STORE WR
(16) COPY R1 AC
(17) SUB CRR
(18) JUMP Z != 21
(19) INC CWR
(20) INC CRR
(21) JUMPU 6
(22) CLR CRR
(23) CLR CWR
(24) COPY R1 AC
(25) SUB RRR
(26) JUMP X!=0 29
(27) INC RWR
(28) INC RRR
(29) JUMPU 6
(30) CLR RRR
(31) CLR RWR
(32) CLR CRR
(33) CLR CWR
(34) LOADK 255
(35) COPY AC R1
(36) LOAD RR
(37) COPY MDR R2
(38) INC RRR
(39) LOAD RR
(40) COPY MDR AC
(41) ADD R2
(42) DIV 2
(43) COPY AC MDR
(44) STORE WR
(45) COPY R1 AC
(46) SUB RRR
(47) JUMP X!=0
(48) INC RRR
(49) INC RWR
(50) JUMPU 35
(51) CLR RRR
(52) CLR RWR
(53) COPY R1 AC
(54) SUB CRR
(55) JUMP Z!=0 58
(56) INC CWR
(57) INC CRR
(58) JUMP U 35
(59) END

7.2 Downsampling using a Gaussian Filter

The 3x3 Gaussian kernel used for Gaussian smoothing is shown below in the figure 7.2. However since our processor does not support floating point arithmetic an approximated Gaussian kernel of size 3×3 as given in the figure 7.3 is used and the resultant value is divided by the total. This kernel is equivalent to two linear separable kernels of size 1×3 and 3×1 as shown in figure 7.4. The image is filtered using these two linear separable kernels in vertical and horizontal directions respectively. This creates the overall effect of the Gaussian kernel applied in 2D.

After smoothing the image by the Gaussian filter to remove the high frequency components, the relevant pixels are picked to generate the down sampled image according to the specified down sampling factor. This method can be used to down sample the given image up to a down sampling factor of 15 and assembly codes for down sampling factors 2 and 3 are shown below.

0.0622	0.2489	0.0622
0.2489	1	0.2489
0.0622	0.2489	0.0622

Figure 7.2 – 3x3 Gaussian Kernel

1	2	1
2	4	2
1	2	1

Figure 7.3 – 3x3 Approximated Gaussian Kernel

1			*	1	2	1
2					
1					

Figure 7.4 – Linear Seprability of the Gaussian Kerne

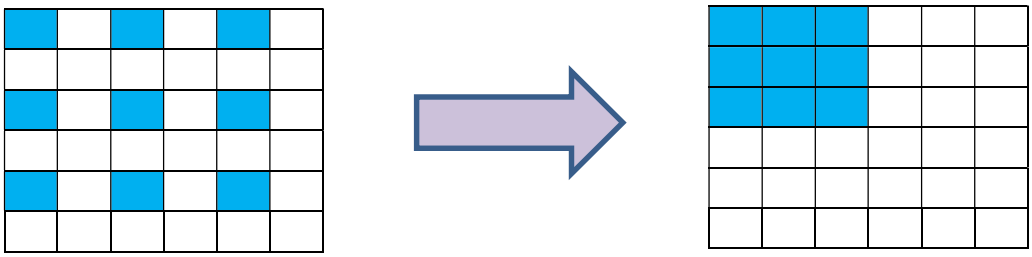


Figure 7.5 – Downsampling using a Gaussian Filter

7.2.1 Assembly Code for Downsampling using a Gaussian Filter (Downsampling factor = 2)

1. LOADK 255
2. COPY AC R1
3. CLR CRR
4. CLR CWR
5. CLR RRR
6. CLR RWR
7. LOAD RR
8. COPY MDR SR1
9. INC CRR
10. LOAD RR
11. COPY MDR SR1
12. INC CRR
13. LOAD RR
14. COPY MDR SR1
15. COPY SR2 AC
16. MUL 2
17. ADD SR3
18. ADD SR1
19. DIV 4
20. INC CWR
21. COPY AC MDR
22. STORE WR
23. COPY R1 AC
24. SUB CRR
25. JUMP Z=0 11
26. CLR CRR
27. CLR CWR
28. COPY RRR AC
29. SUB R1
30. JUMP Z!=0 33
31. INC RWR
32. INC RRR
33. JUMP U 6
34. NOP

35. CLR CRR
36. CLR CWR
37. CLR RRR
38. CLR RWR
39. LOAD RR
40. COPY MDR SR1
41. INC RRR
42. LOAD RR
43. COPY MDR SR1
44. INC RRR
45. LOAD RR
46. COPY MDR SR1
47. COPY SR2 AC
48. MUL 2
49. ADD SR3
50. ADD SR1
51. DIV 4
52. INC RWR
53. COPY AC MDR
54. STORE WR
55. COPY RRR AC
56. SUB R1
57. JUMP Z=0 43
58. CLR RRR
59. CLR RWR
60. COPY CRR AC
61. SUB R1
62. JUMP Z!=0 65
63. INC CWR
64. INC CRR
65. JUMP U 38
66. NOP
67. CLR CWR
68. CLR RWR
69. CLR CRR
70. CLR RRR
71. LOADK 2
72. COPY AC R2
73. LOADK 254
74. COPY AC R1
75. LOAD RR
76. STORE WR
77. COPY CRR AC
78. SUB R1
79. INC CWR
80. JUMP Z!=0 83

81. INC CRR
82. INC CRR
83. JUMP U 74
84. CLR CRR
85. CLR CWR
86. INC RWR
87. COPY RRR AC
88. SUB R1
89. JUMP Z!=0 92
90. INC RRR
91. INC RRR
92. JUMP U 74
93. END

7.2.2 Assembly Code for Downsampling using a Gaussian Filter (Downsampling factor = 3)

1. LOADK 255
2. COPY AC R1
3. CLR CRR
4. CLR CWR
5. CLR RRR
6. CLR RWR
7. LOAD RR
8. COPY MDR SR1
9. INC CRR
10. LOAD RR
11. COPY MDR SR1
12. INC CRR
13. LOAD RR
14. COPY MDR SR1
15. COPY SR2 AC
16. MUL 2
17. ADD SR3
18. ADD SR1
19. DIV 4
20. INC CWR
21. COPY AC MDR
22. STORE WR
23. COPY R1 AC
24. SUB CRR
25. JUMP Z=0 11
26. CLR CRR
27. CLR CWR
28. COPY RRR AC
29. SUB R1
30. JUMP Z!=0 33
31. INC RWR
32. INC RRR

33. JUMP U 6
34. NOP
35. CLR CRR
36. CLR CWR
37. CLR RRR
38. CLR RWR
39. LOAD RR
40. COPY MDR SR1
41. INC RRR
42. LOAD RR
43. COPY MDR SR1
44. INC RRR
45. LOAD RR
46. COPY MDR SR1
47. COPY SR2 AC
48. MUL 2
49. ADD SR3
50. ADD SR1
51. DIV 4
52. INC RWR
53. COPY AC MDR
54. STORE WR
55. COPY RRR AC
56. SUB R1
57. JUMP Z=0 43
58. CLR RRR
59. CLR RWR
60. COPY CRR AC
61. SUB R1
62. JUMP Z!=0 65
63. INC CWR
64. INC CRR
65. JUMP U 38
66. NOP
67. CLR CWR
68. CLR RWR
69. CLR CRR
70. CLR RRR
71. LOADK 3
72. COPY AC R2
73. LOADK 255
74. COPY AC R1
75. LOAD RR
76. STORE WR
77. COPY CRR AC
78. SUB R1
79. INC CWR
80. JUMP Z!=0 84
81. INC CRR

82. INC CRR
83. INC CRR
84. JUMP U 74
85. CLR CRR
86. CLR CWR
87. INC RWR
88. COPY RRR AC
89. SUB R1
90. JUMP Z!=0 94
91. INC RRR
92. INC RRR
93. INC RRR
94. JUMP U 74
95. END

7.3 Gaussian Smoothing

The same Gaussian kernel used in down sampling using a Gaussian filter method can be used to implement Gaussian smoothing. The kernel is applied as two linear separable kernels for the vertical and horizontal directions. The assembly code used for Gaussian smoothing a given image is shown below.

7.3.1 Assembly Code for Gaussian Smoothing

1. LOADK 255
2. COPY AC R1
3. CLR CRR
4. CLR CWR
5. CLR RRR
6. CLR RWR
7. LOAD RR
8. COPY MDR SR1
9. INC CRR
10. LOAD RR
11. COPY MDR SR1
12. INC CRR
13. LOAD RR
14. COPY MDR SR1
15. COPY SR2 AC
16. MUL 2
17. ADD SR3
18. ADD SR1
19. DIV 4
20. INC CWR
21. COPY AC MDR
22. STORE WR
23. COPY R1 AC

24. SUB CRR
25. JUMP Z=0 11
26. CLR CRR
27. CLR CWR
28. COPY RRR AC
29. SUB R1
30. JUMP Z!=0 33
31. INC RWR
32. INC RRR
33. JUMP U 6
34. NOP
35. CLR CRR
36. CLR CWR
37. CLR RRR
38. CLR RWR
39. LOAD RR
40. COPY MDR SR1
41. INC RRR
42. LOAD RR
43. COPY MDR SR1
44. INC RRR
45. LOAD RR
46. COPY MDR SR1
47. COPY SR2 AC
48. MUL 2
49. ADD SR3
50. ADD SR1
51. DIV 4
52. INC RWR
53. COPY AC MDR
54. STORE WR
55. COPY RRR AC
56. SUB R1
57. JUMP Z=0 43
58. CLR RRR
59. CLR RWR
60. COPY CRR AC
61. SUB R1
62. JUMP Z!=0 65
63. INC CWR
64. INC CRR
65. JUMP U 38

8. Compiler

A compiler is used to translate a program written in a high level language into binary code. In our implementation, the compiler converts the assembly code which humans can understand into binary code that the processor can understand. The compiler is written in python. It takes a text file as the input in which the program is written in assembly code using the instructions from the pre-defined instruction set. The compiler converts the each line in assembly code into binary code and saves in another file.

This file is then modified into the .mif (memory initialization file) format which can be used to initialize the instruction memory in FPGA. The python code for compiler is attached as an appendix. The opcodes and the format of each instruction are given in the table under the section instruction set. The operands of the instructions can be either constants or registers or any other specific information depending on the instruction. The assembly code, the binary code obtained by the compiler and the mif format for an example set of instructions are given below.

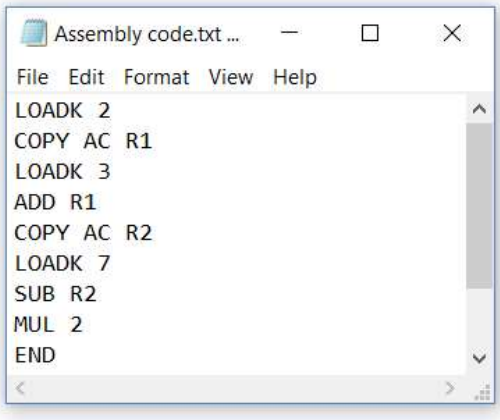


Figure 8.1 – Assembly Code

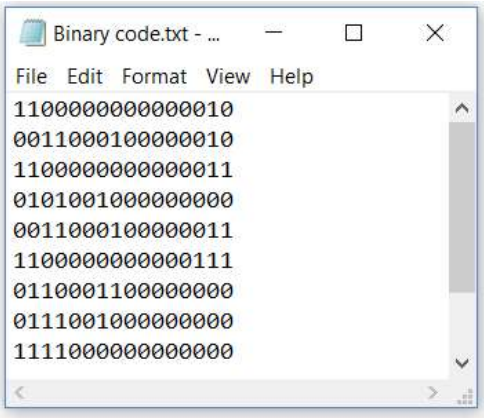
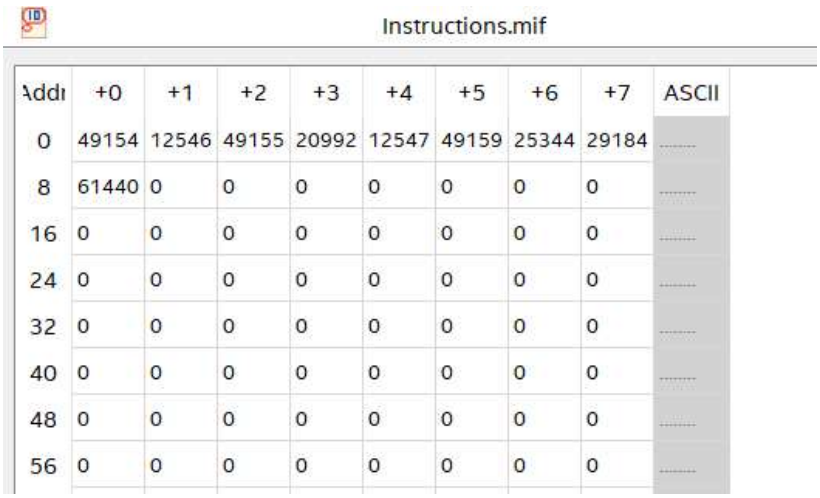


Figure 8.2 – Binary Code



Addı	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	49154	12546	49155	20992	12547	49159	25344	29184	.....
8	61440	0	0	0	0	0	0	0	.....
16	0	0	0	0	0	0	0	0	.....
24	0	0	0	0	0	0	0	0	.....
32	0	0	0	0	0	0	0	0	.....
40	0	0	0	0	0	0	0	0	.....
48	0	0	0	0	0	0	0	0	.....
56	0	0	0	0	0	0	0	0	.....

Figure 8.3 – Memory Initialization File

9. Debugging

In order to verify the accurate operation of processor several debugging features have been incorporated. In normal operation processor works with a 25MHz clock speed and completes the operation in few seconds. However for testing, a separate module called slow clock has been used which works at a much lower speed around 1Hz can be used and step by step changes can be clearly visualized. Three LEDs are used to indicate the status of receiving done, processing done and transmission done.

For visualization purposes two 7 segment displays have been used to display the state. With the help of them it can be ensured that the processor switches to the correct states in each instruction. In addition to that three 7 segment displays have been used to display the value in AC. Using them arithmetic accuracy of the operations can be verified. For visualizing the UART transmission process, two sets of LEDs are used. The following diagram illustrates the 7 segment display and LED assignments.

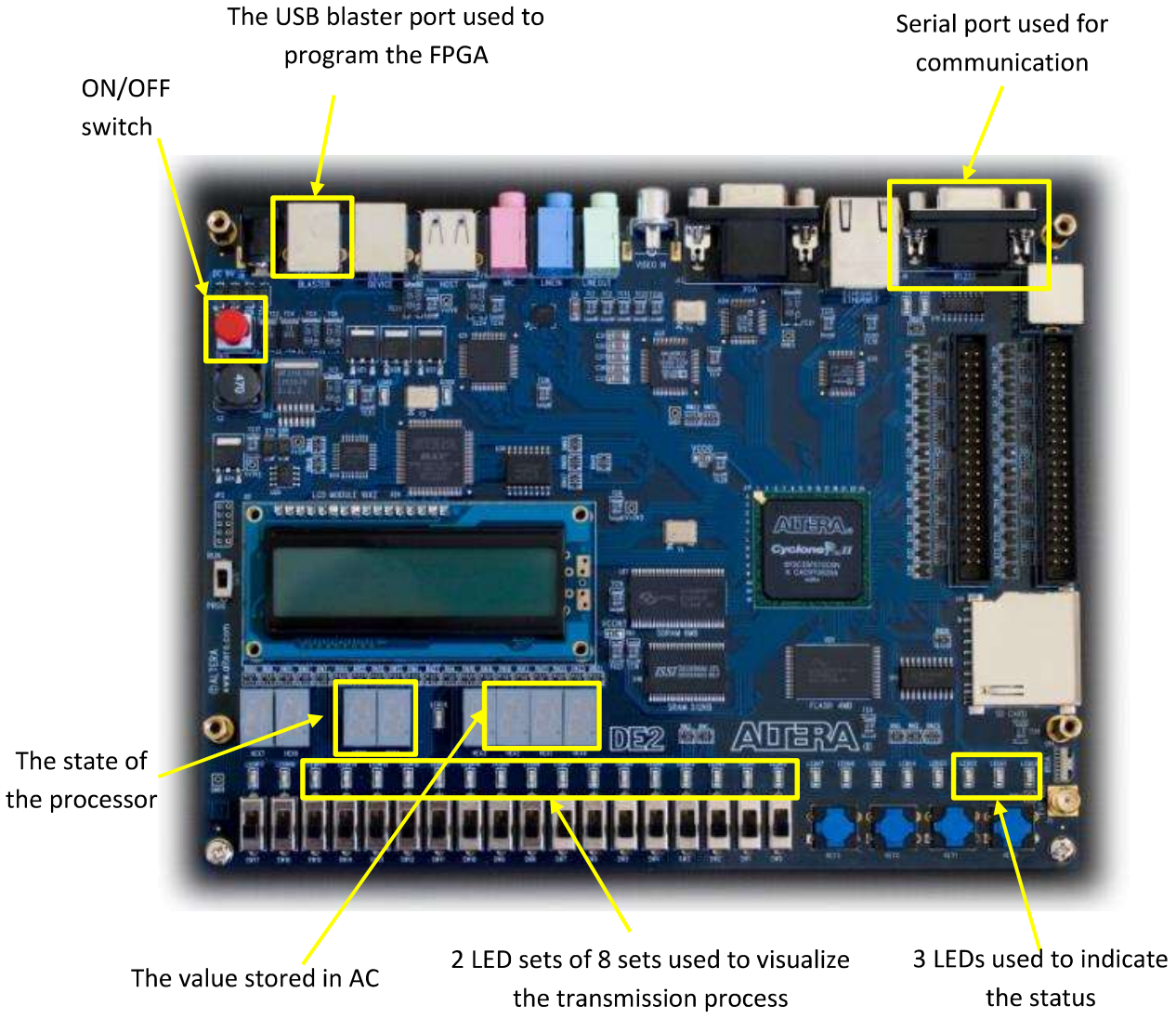


Figure 9.1 – Usage of the Components of the DE2-115 Board

10. Results

10.1 Downsampling using an Average Filter (Grayscale Image)

e

Figure 10.1 – Received Image

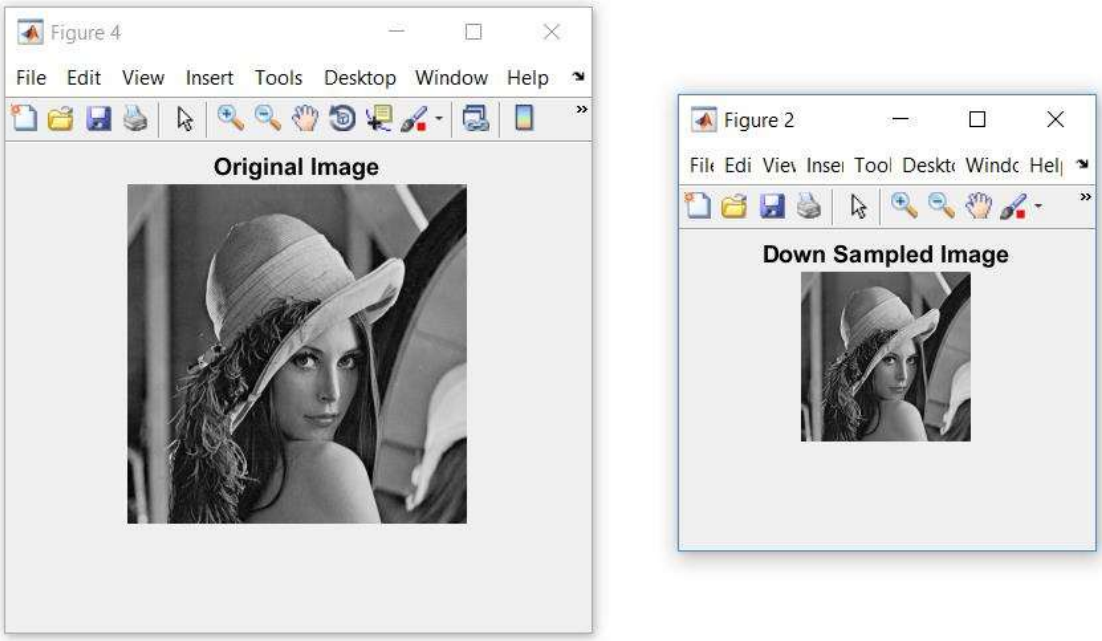


Figure 10.2 – Original Image vs Downsampled Image

10.2 Downsampling using an Average Filter (RGB Image)

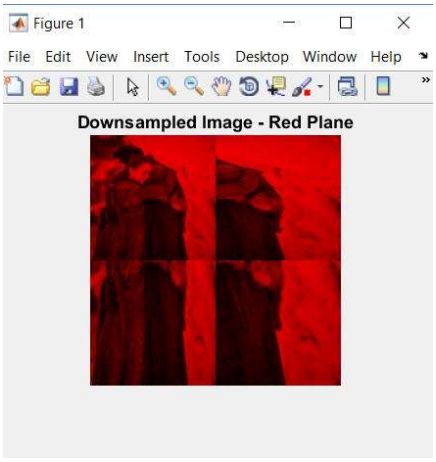


Figure 10.3 – Received Image (R)

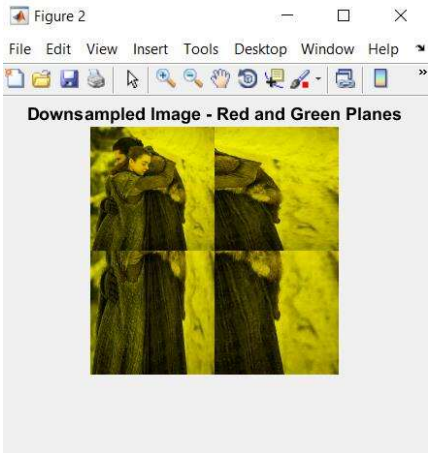


Figure 10.4 – Received Image (R & G)

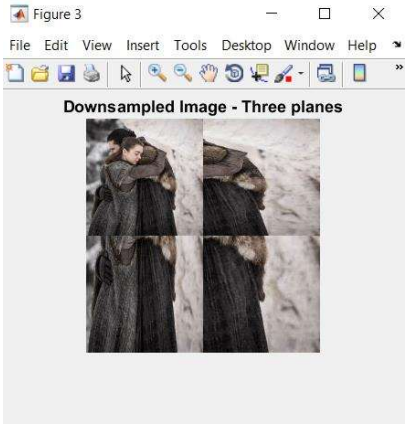


Figure 10.5 – Received Image (R,G & B)

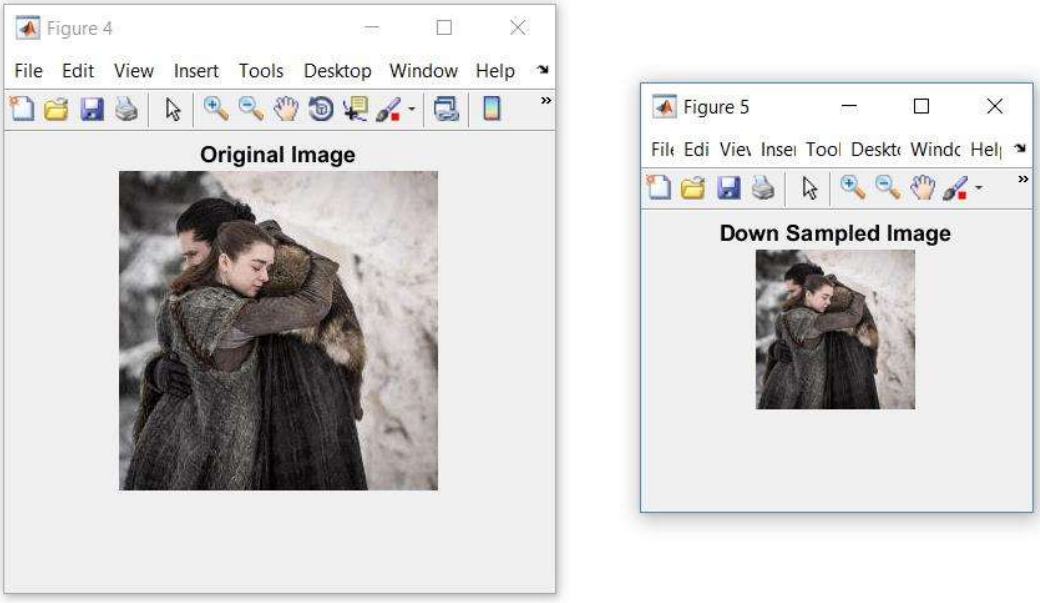


Figure 10.6 – Original Image vs Downsampled Image

10.3 Downsampling using a Gaussian Filter (Grayscale Image)

Figure 10.7 – Received Image

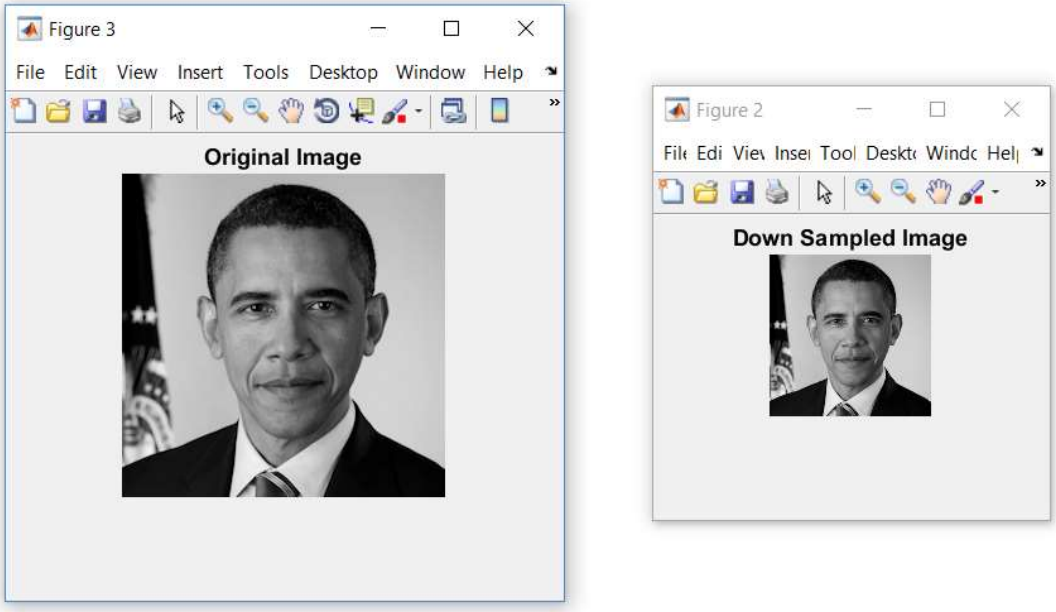


Figure 10.8 – Original Image vs Downsampled Image

10.4 Downsampling using a Gaussian Filter (RGB Image)

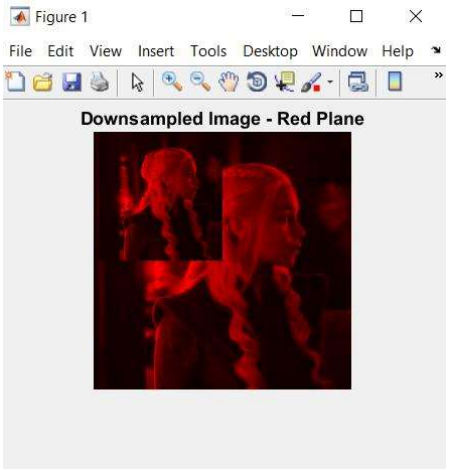


Figure 10.9 – Received Image (R)

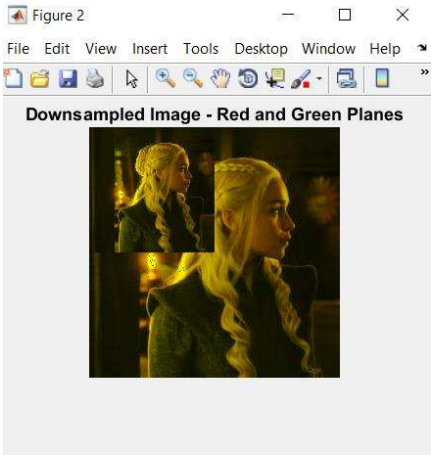


Figure 10.10 – Received Image (R & G)

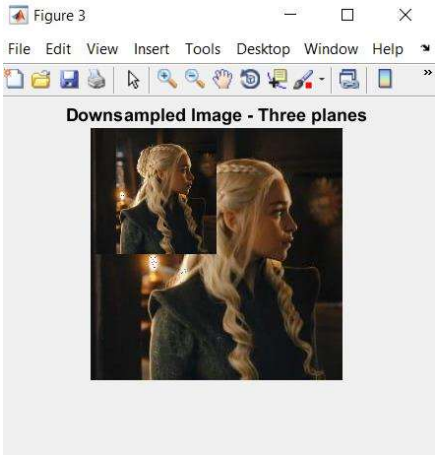


Figure 10.11 – Received Image (R,G & B)

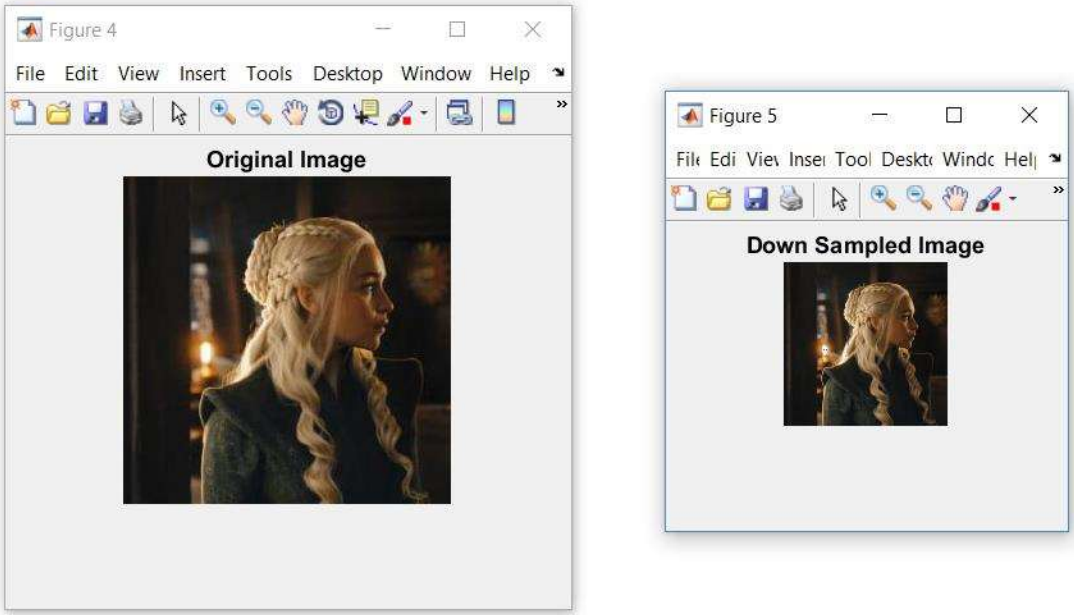


Figure 10.12 – Original Image vs Downsampled Image

10.5 Downsampling by a factor of 3

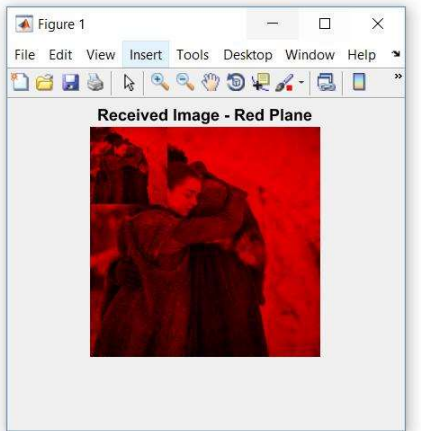


Figure 10.13 – Received Image (R)

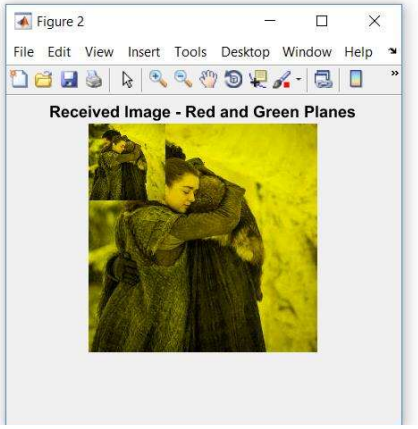


Figure 10.14 – Received Image (R & G)

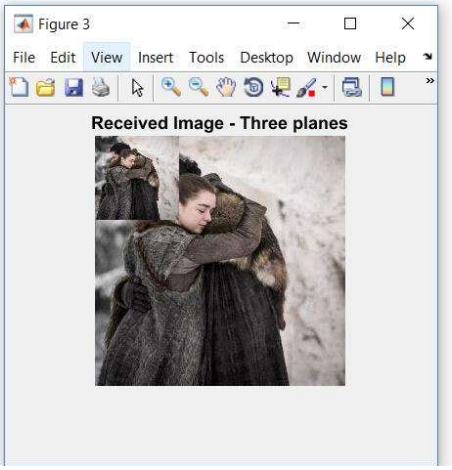


Figure 10.15 – Received Image (R,G & B)

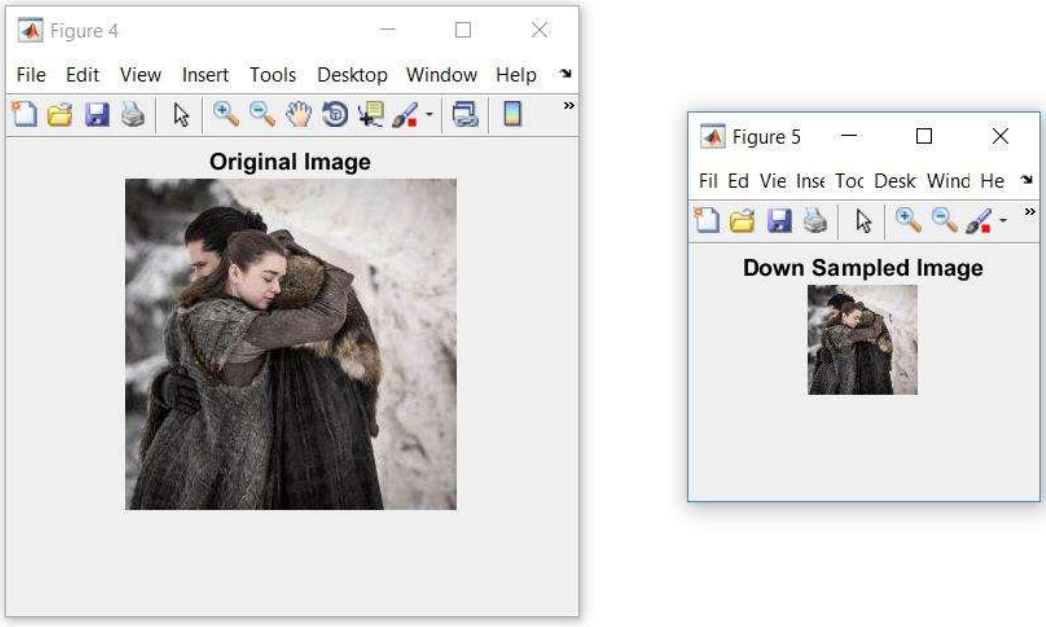


Figure 10.16 – Original Image vs Downsampled Image

10.6 Gaussian Smoothing

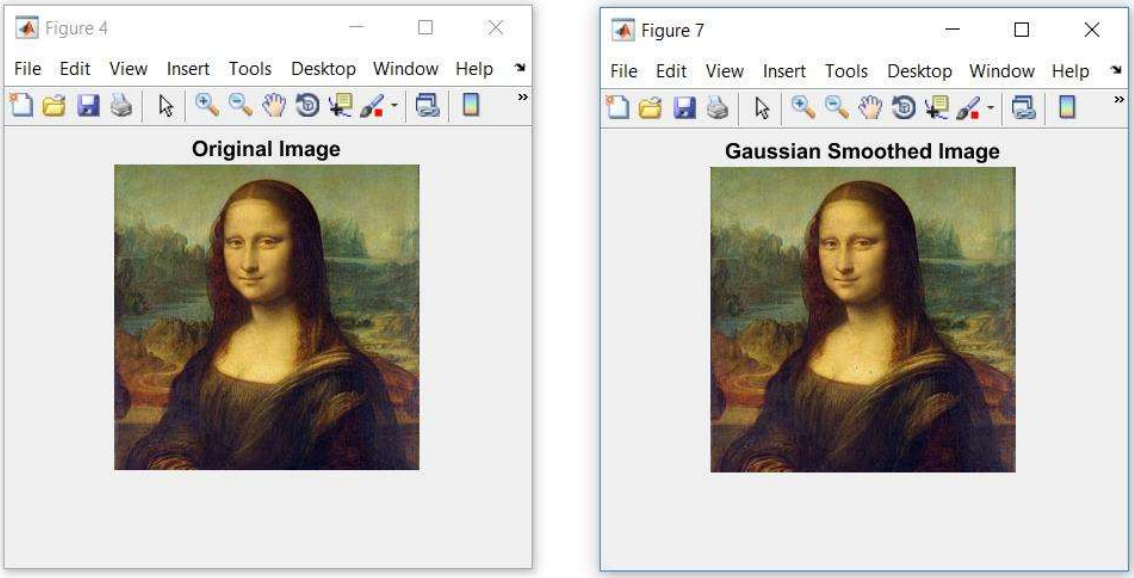


Figure 10.17 – Original Image vs Gaussian Smoothed Image

Figure 10.18 – Original Image – Gaussian Smoothed Image

11. Conclusion

At the end of the project we were able to successfully implement an application specific custom processor for image downsampling with following features.

- Ability to use both Gaussian and average filtering.
- Both grayscale and RGB images can be processed.
- Images could be down-sampled with any factor up to 15.
- Ability to perform other linear separable kernels like Gaussian smoothing.
- Ability to process images continuously without manually resetting or re-uploading the code.
- Use of minimum possible resources.

Flow Status	Successful - Sat Jul 13 14:05:50 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	Top_Level_Module
Top-level Entity Name	Top_Level_Module
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	1,165 / 114,480 (1 %)
Total registers	294
Total pins	57 / 529 (11 %)
Total virtual pins	0
Total memory bits	528,384 / 3,981,312 (13 %)
Embedded Multiplier 9-bit elements	2 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

Figure 11.1 – Usage of Resources

For verifying the successful implementation, we calculated the pixel wise error between the received image from the FPGA board and the processed image using MATLAB codes which include the same operations. WE could obtain zero SSE (Squared Sum of Errors) for both methods, downsampling using a Gaussian filter and downsampling using an average filter.

Figure 11.2 – Comparison for Grayscale Image

Figure 11.2 – Comparison for RGBImage

When considering about the time consumption, it took around three minutes to transmit a 256x256 RGB image and return back the down-sampled image. From the total time it took around 4 seconds for processing and the rest was expended on transmission.

12. References

1. Altera DE2-115 User Manual
2. https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html
3. https://www.mathworks.com/help/matlab/ref/serial.html
4. https://www.dsprelated.com/freebooks/sasp/Downsampling_Aliasing.html
5. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ram_rom.pdf
6. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altpll.pdf
7. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_tq_tutorial.pdf
8. http://referencedesigner.com/blog/clock-divider-in-spartixed-using-verilog/2939/

13. Appendices

13.1 Verilog Codes

13.1.1 Top Level Module

```
//Top Level Module used for interconnecting all modules

module Top_Level_Module(input clk_in,
                        input Rx,
                        input clock_select,
                        output Tx,
                        output [7:0] address_LED,
                        output [7:0] data_LED,
                        output [6:0] state_0,
                        output [6:0] state_1,
                        output [6:0] AC_0,
                        output [6:0] AC_1,
                        output [6:0] AC_2,
                        output processing_done_indicator,
                        output Rx_done_indicator,
                        output Tx_done_indicator);

wire [7:0] IRAM_address;
wire [15:0] IRAM_to_processor;

wire [15:0] DRAM_address;
wire [15:0] DRAM_address_processor;
wire [15:0] DRAM_address_receiver;

wire [7:0] DRAM_write_data;
wire [7:0] DRAM_data_processor;
wire [7:0] DRAM_data_receiver;

wire [7:0] DRAM_read_data;
wire [7:0] DRAM_to_processor;
wire [7:0] DRAM_to_transmitter;

wire [5:0] state_disp;
wire [7:0] AC_disp;

wire write_DRAM;
wire write_DRAM_processor;
wire write_DRAM_receiver;

wire enable_processor;
wire start_Tx;

wire Rx_done;
wire Tx_done;

wire clk_out;

assign Rx_done_indicator = Rx_done;
assign processing_done_indicator = start_Tx;
assign Tx_done_indicator = Tx_done;

assign address_LED = DRAM_address_receiver[15:8];
assign data_LED = DRAM_read_data;


IRAM IRAM(.address(IRAM_address),
          .clock(clk_in),
          .data(16'b0),
          .wren(1'b0),
          .q(IRAM_to_processor));

DRAM DRAM(.address(DRAM_address),
          .clock(clk_in),
          .data(DRAM_write_data),
          .wren(write_DRAM),
          .q(DRAM_read_data));

```

```
68  Processor Processor(.IRAM_address(IRAM_address),
69                      .IRAM_data(IRAM_to_processor),
70                      .clock(clk_out),
71                      .DRAM_address_processor(DRAM_address_processor),
72                      .DRAM_output_data(DRAM_data_processor),
73                      .DRAM_input_data(DRAM_to_processor),
74                      .write_DRAM(write_DRAM_processor),
75                      .enable_processor(Rx_done),
76                      .Tx_done(Tx_done),
77                      .start_Tx(start_Tx),
78                      .state_disp(state_disp),
79                      .AC_disp(AC_disp));
80
81  transceiver transceiver(.Rx(Rx),
82                          .Tx(Tx),
83                          .start_Tx(start_Tx),
84                          .clk(clk_in),
85                          .data_in(DRAM_to_transmitter),
86                          .data_out(DRAM_data_receiver),
87                          .ready(write_DRAM_receiver),
88                          .D_address(DRAM_address_receiver),
89                          .Rx_done(Rx_done),
90                          .Tx_done(Tx_done));
91
92  DRAM_write_data_mux DRAM_write_data_mux(.outputmux(DRAM_write_data),
93                                          .inputmux1(DRAM_data_processor),
94                                          .inputmux2(DRAM_data_receiver),
95                                          .select(Rx_done));
96
97  DRAM_write_enable_mux DRAM_write_enable_mux(.outputmux(write_DRAM),
98                                              .inputmux1(write_DRAM_processor),
99                                              .inputmux2(write_DRAM_receiver),
100                                             .select(Rx_done));
101
102 DRAM_read_data_splitter DRAM_read_data_splitter(.output1(DRAM_to_processor),
103                                                 .output2(DRAM_to_transmitter),
104                                                 .inputsplitter(DRAM_read_data));
105
106 DRAM_address_mux DRAM_address_mux(.outputmux(DRAM_address),
107                                   .inputmux1(DRAM_address_processor),
108                                   .inputmux2(DRAM_address_receiver),
109                                   .Rx_done(Rx_done),
110                                   .start_Tx(start_Tx));
111
112 clock_selector clock_selector(.clk_in(clk_in),
113                               .clock_select(clock_select),
114                               .clk_out(clk_out));
115
116 state_to_7seg state_to_7seg(.bin(state_disp),
117                             .dout0(state_0),
118                             .dout1(state_1));
119
120 AC_to_7seg AC_to_7seg(.din(AC_disp),
121                       .dout2(AC_2),
122                       .dout1(AC_1),
123                       .dout0(AC_0));
124
125 endmodule
```

13.1.2 Processor

```
//Processor for downsampling

module Processor(input wire [7:0] DRAM_input_data,
                 input wire [15:0] IRAM_data,
                 input wire enable_processor,
                 input wire Tx_done,
                 input wire clock,
                 output wire [7:0] IRAM_address,
                 output wire [15:0] DRAM_address_processor,
                 output wire [7:0] DRAM_output_data,
                 output wire write_DRAM,
                 output wire start_Tx,
                 output wire [5:0] state_disp,
                 output wire [7:0] AC_disp);

wire [15:0] instruction;

wire [3:0] ALU_control;
wire [3:0] select_source;
wire [2:0] select_destination;
wire [1:0] IDC_control;
wire [1:0] PC_control;
wire [1:0] MDR_control;
wire [1:0] MAR_control;
wire load_instruction;

wire [15:0] bus_to_R1;
wire [15:0] bus_to_R2;
wire [15:0] bus_to_B;
wire [7:0] bus_to_SR1;
wire [7:0] bus_to_MDR;

wire [15:0] AC_to_Bus;
wire [15:0] R1_to_bus;
wire [15:0] R2_to_bus;
wire [7:0] SR1_to_bus;
wire [7:0] SR2_to_bus;
wire [7:0] SR3_to_bus;
wire [7:0] MDR_to_bus;

wire [7:0] IDC_control_RRR;
wire [7:0] IDC_control_RWR;
wire [7:0] IDC_control_CRR;
wire [7:0] IDC_control_CWR;

wire [7:0] RRR_out;
wire [7:0] CRR_out;

wire [15:0] RR_to_MAR;
wire [15:0] WR_to_MAR;

wire Z_out;

statemachine statemachine(.instruction(instruction),
                          .clock(clock),
                          .enable_processor(enable_processor),
                          .Tx_done(Tx_done),
                          .load_instruction(load_instruction),
                          .ALU_control(ALU_control),
                          .select_source(select_source),
                          .select_destination(select_destination),
                          .PC_control(PC_control),
                          .IDC_control(IDC_control),
                          .MDR_control(MDR_control),
                          .MAR_control(MAR_control),
                          .write_DRAM(write_DRAM),
                          .start_Tx(start_Tx),
                          .state_disp(state_disp));

```

```
IR IR(.clock(clock),
      .IRAM_data(IRAM_data),
      .instruction(instruction),
      .load_instruction(load_instruction));

ALU ALU(.B(bus_to_B),
        .ALU_control(ALU_control),
        .AC_out(AC_to_Bus),
        .Z_out(Z_out),
        .instruction(instruction),
        .AC_disp(AC_disp),
        .clock(clock));

Bus Bus(.AC_in(AC_to_Bus),
        .SR1_in(SR1_to_bus),
        .SR2_in(SR2_to_bus),
        .SR3_in(SR3_to_bus),
        .R1_in(R1_to_bus),
        .R2_in(R2_to_bus),
        .RRR_in(RRR_out),
        .CRR_in(CRR_out),
        .MDR_in(MDR_to_bus),
        .SR1_out(bus_to_SR1),
        .R1_out(bus_to_R1),
        .R2_out(bus_to_R2),
        .MDR_out(bus_to_MDR),
        .B_out(bus_to_B),
        .select_source(select_source),
        .select_destination(select_destination),
        .instruction(instruction),
        .clock(clock));

PC PC(.Z_out(Z_out),
      .PC_control(PC_control),
      .IRAM_address(IRAM_address),
      .clock(clock),
      .instruction(instruction));

IDC_controller IDC_controller(.IDC_control(IDC_control),
                              .instruction(instruction),
                              .IDC_control_RRR(IDC_control_RRR),
                              .IDC_control_RWR(IDC_control_RWR),
                              .IDC_control_CRR(IDC_control_CRR),
                              .IDC_control_CWR(IDC_control_CWR),
                              .clock(clock));

R R1(.R_to_bus(R1_to_bus),
     .bus_to_R(bus_to_R1));

R R2(.R_to_bus(R2_to_bus),
     .bus_to_R(bus_to_R2));

shift_registers shift_registers(.bus_to_SR1(bus_to_SR1),
                                .SR1_to_bus(SR1_to_bus),
                                .SR2_to_bus(SR2_to_bus),
                                .SR3_to_bus(SR3_to_bus));

Read_registers Read_registers(.IDC_control_RRR(IDC_control_RRR),
                              .IDC_control_CRR(IDC_control_CRR),
                              .RR_to_MAR(RR_to_MAR),
                              .RRR_out(RRR_out),
                              .CRR_out(CRR_out));

Write_registers Write_registers(.IDC_control_RWR(IDC_control_RWR),
                                .IDC_control_CWR(IDC_control_CWR),
                                .WR_to_MAR(WR_to_MAR));

MDR MDR(.DRAM_to_MDR(DRAM_input_data),
        .bus_to_MDR(bus_to_MDR),
        .clock(clock),
        .MDR_to_bus(MDR_to_bus),
        .MDR_to_DRAM(DRAM_output_data),
        .MDR_control(MDR_control));
```

```
MAR MAR(.AC_to_MAR(AC_to_Bus),
        .RR_to_MAR(RR_to_MAR),
        .WR_to_MAR(WR_to_MAR),
        .MAR_to_DRAM(DRAM_address_processor),
        .clock(clock),
        .MAR_control(MAR_control));

endmodule
```

13.1.3 State Machine

```
//State Machine of the processor

module statemachine(input wire [15:0] instruction,
                    input wire clock,
                    input wire enable_processor,
                    input wire Tx_done,
                    output reg load_instruction,
                    output reg [3:0] ALU_control,
                    output reg [3:0] select_source,
                    output reg [2:0] select_destination,
                    output reg [1:0] IDC_control,
                    output reg [1:0] MDR_control,
                    output reg [1:0] MAR_control,
                    output reg [1:0] PC_control,
                    output reg write_DRAM,
                    output reg start_Tx,
                    output wire [5:0] state_disp);


reg [7:0] state;
reg [3:0] source_register;
reg [2:0] destination_register;
reg [3:0] opcode;

assign state_disp = state[5:0];

initial
begin
    state = 8'b00010000; //start
end

always@(instruction)
begin
    opcode = instruction[15:12];
    source_register = instruction[11:8];
    destination_register = instruction[2:0];
end

always@(negedge clock)
begin
    case(state)

        8'b00010000: //start
            begin
                load_instruction <= 0;
                ALU_control <= 4'b0000;
                select_source <= 4'b0000;
                select_destination <= 3'b000;
                PC_control <= 2'b00;
                IDC_control <= 2'b00;
                MDR_control <= 2'b00;
                MAR_control <= 2'b00;
                write_DRAM <= 0;
                start_Tx <= 0;
                if (enable_processor)
                begin
                    state <= 8'b00010001; //fetch1
                end
                else
                begin
                    state <= 8'b00010000; //start
                end
            end

```

```
65          8'b00010001: //fetch1
66              begin
67                  load_instruction <= 1;
68                  ALU_control <= 4'b0000;
69                  select_source <= 4'b0000;
70                  select_destination <= 3'b000;
71                  PC_control <= 2'b00;
72                  IDC_control <= 2'b00;
73                  MDR_control <= 2'b00;
74                  MAR_control <= 2'b00;
75                  write_DRAM <= 0;
76                  start_Tx <= 0;
77                  state <= 8'b00010010; //fetch2
78              end
79
80          8'b00010010: //fetch2
81              begin
82                  load_instruction <= 0;
83                  ALU_control <= 4'b0000;
84                  select_source <= 4'b0000;
85                  select_destination <= 3'b000;
86                  PC_control <= 2'b01;
87                  IDC_control <= 2'b00;
88                  MDR_control <= 2'b00;
89                  MAR_control <= 2'b00;
90                  write_DRAM <= 0;
91                  start_Tx <= 0;
92                  state <= {4'b0000, opcode}; //loaded instruction
93              end
94
95          8'b00000000: //nop
96              begin
97                  load_instruction <= 0;
98                  ALU_control <= 4'b0000;
99                  select_source <= 4'b0000;
100                 select_destination <= 3'b000;
101                 PC_control <= 2'b00;
102                 IDC_control <= 2'b00;
103                 MDR_control <= 2'b00;
104                 MAR_control <= 2'b00;
105                 write_DRAM <= 0;
106                 start_Tx <= 0;
107                 state <= 8'b00010001; //fetch1
108             end
109
110         8'b00000101: //add1
111             begin
112                 load_instruction <= 0;
113                 ALU_control <= 4'b0000;
114                 select_source <= source_register;
115                 select_destination <= 3'b000;
116                 PC_control <= 2'b00;
117                 IDC_control <= 2'b00;
118                 MDR_control <= 2'b00;
119                 MAR_control <= 2'b00;
120                 write_DRAM <= 0;
121                 start_Tx <= 0;
122                 state <= 8'b00010011; //add2
123             end
124
125         8'b00010011: //add2
126             begin
127                 load_instruction <= 0;
128                 ALU_control <= 4'b0001;
129                 select_source <= 4'b0000;
130                 select_destination <= 3'b000;
131                 PC_control <= 2'b00;
132                 IDC_control <= 2'b00;
133                 MDR_control <= 2'b00;
134                 MAR_control <= 2'b00;
135                 write_DRAM <= 0;
136                 start_Tx <= 0;
137                 state <= 8'b00010001; //fetch1
138             end
139
```

```
140             8'b00000110: //sub1
141                 begin
142                     load_instruction <= 0;
143                     ALU_control <= 4'b0000;
144                     select_source <= source_register;
145                     select_destination <= 3'b000;
146                     PC_control <= 2'b00;
147                     IDC_control <= 2'b00;
148                     MDR_control <= 2'b00;
149                     MAR_control <= 2'b00;
150                     write_DRAM <= 0;
151                     start_Tx <= 0;
152                     state <= 8'b00010100; //sub2
153                 end
154 
155             8'b00010100: //sub2
156                 begin
157                     load_instruction <= 0;
158                     ALU_control <= 4'b0010;
159                     select_source <= 4'b0000;
160                     select_destination <= 3'b000;
161                     PC_control <= 2'b00;
162                     IDC_control <= 2'b00;
163                     MDR_control <= 2'b00;
164                     MAR_control <= 2'b00;
165                     write_DRAM <= 0;
166                     start_Tx <= 0;
167                     state <= 8'b00010001; //fetch1
168                 end
169 
170             8'b00000111: //mul1
171                 begin
172                     load_instruction <= 0;
173                     ALU_control <= 4'b0000;
174                     select_source <= 4'b1011;
175                     select_destination <= 3'b000;
176                     PC_control <= 2'b00;
177                     IDC_control <= 2'b00;
178                     MDR_control <= 2'b00;
179                     MAR_control <= 2'b00;
180                     write_DRAM <= 0;
181                     start_Tx <= 0;
182                     state <= 8'b00010101; //mul2
183                 end
184 
185             8'b00010101: //mul2
186                 begin
187                     load_instruction <= 0;
188                     ALU_control <= 4'b0011;
189                     select_source <= 4'b0000;
190                     select_destination <= 3'b000;
191                     PC_control <= 2'b00;
192                     IDC_control <= 2'b00;
193                     MDR_control <= 2'b00;
194                     MAR_control <= 2'b00;
195                     write_DRAM <= 0;
196                     start_Tx <= 0;
197                     state <= 8'b00010001; //fetch1
198                 end
199 
200             8'b00001000: //div1
201                 begin
202                     load_instruction <= 0;
203                     ALU_control <= 4'b0000;
204                     select_source <= 4'b1011;
205                     select_destination <= 3'b000;
206                     PC_control <= 2'b00;
207                     IDC_control <= 2'b00;
208                     MDR_control <= 2'b00;
209                     MAR_control <= 2'b00;
210                     write_DRAM <= 0;
211                     start_Tx <= 0;
212                     state <= 8'b00010110; //div2
213                 end
214 
```

```
215         8'b00010110: //div2
216             begin
217                 load_instruction <= 0;
218                 ALU_control <= 4'b0100;
219                 select_source <= 4'b0000;
220                 select_destination <= 3'b000;
221                 PC_control <= 2'b00;
222                 IDC_control <= 2'b00;
223                 MDR_control <= 2'b00;
224                 MAR_control <= 2'b00;
225                 write_DRAM <= 0;
226                 start_Tx <= 0;
227                 state <= 8'b00010001; //fetch1
228             end
229 
230         8'b00000011: //copy1
231             begin
232                 load_instruction <= 0;
233                 ALU_control <= 4'b0000;
234                 select_source <= source_register;
235                 select_destination <= 3'b000;
236                 PC_control <= 2'b00;
237                 IDC_control <= 2'b00;
238                 MDR_control <= 2'b00;
239                 MAR_control <= 2'b00;
240                 write_DRAM <= 0;
241                 start_Tx <= 0;
242                 state <= 8'b00010111; //copy2
243             end
244 
245         8'b00010111: //copy2
246             begin
247                 load_instruction <= 0;
248                 if (destination_register == 3'b001)
249                     ALU_control <= 4'b0101;
250                 else
251                     ALU_control <= 4'b0000;
252                 select_source <= 4'b0000;
253                 if (destination_register == 3'b001)
254                     select_destination <= 3'b000;
255                 else
256                     select_destination <= destination_register;
257                 if (destination_register == 3'b101)
258                     MDR_control <= 2'b10;
259                 else
260                     MDR_control <= 2'b00;
261                 PC_control <= 2'b00;
262                 IDC_control <= 2'b00;
263                 MAR_control <= 2'b00;
264                 write_DRAM <= 0;
265                 start_Tx <= 0;
266                 state <= 8'b00010001; //fetch1
267             end
268 
269         8'b00001100: //loadk
270             begin
271                 load_instruction <= 0;
272                 ALU_control <= 4'b0110;
273                 select_source <= 4'b0000;
274                 select_destination <= 3'b000;
275                 PC_control <= 2'b00;
276                 IDC_control <= 2'b00;
277                 MDR_control <= 2'b00;
278                 MAR_control <= 2'b00;
279                 write_DRAM <= 0;
280                 start_Tx <= 0;
281                 state <= 8'b00010001; //fetch1
282             end
283 
```

```
284         8'b00000100: //jump
285             begin
286                 load_instruction <= 0;
287                 ALU_control <= 4'b0000;
288                 select_source <= 4'b0000;
289                 select_destination <= 3'b000;
290                 PC_control <= 2'b10;
291                 IDC_control <= 2'b00;
292                 MDR_control <= 2'b00;
293                 MAR_control <= 2'b00;
294                 write_DRAM <= 0;
295                 start_Tx <= 0;
296                 state <= 8'b00010001; //fetch1
297             end
298 
299         8'b00001010: //inc
300             begin
301                 load_instruction <= 0;
302                 if (source_register == 4'b0001)
303                     ALU_control <= 4'b0111;
304                 else
305                     ALU_control <= 4'b0000;
306                 if (source_register == 4'b0001)
307                     IDC_control <= 2'b00;
308                 else
309                     IDC_control <= 2'b01;
310                 select_source <= 4'b0000;
311                 select_destination <= 3'b000;
312                 PC_control <= 2'b00;
313                 MDR_control <= 2'b00;
314                 MAR_control <= 2'b00;
315                 write_DRAM <= 0;
316                 start_Tx <= 0;
317                 state <= 8'b00010001; //fetch1
318             end
319 
320         8'b00001011: //dec
321             begin
322                 load_instruction <= 0;
323                 if (source_register == 4'b0001)
324                     ALU_control <= 4'b1000;
325                 else
326                     ALU_control <= 4'b0000;
327                 if (source_register == 4'b0001)
328                     IDC_control <= 2'b00;
329                 else
330                     IDC_control <= 2'b10;
331                 select_source <= 4'b0000;
332                 select_destination <= 3'b000;
333                 PC_control <= 2'b00;
334                 MDR_control <= 2'b00;
335                 MAR_control <= 2'b00;
336                 write_DRAM <= 0;
337                 start_Tx <= 0;
338                 state <= 8'b00010001; //fetch1
339             end
340 
341         8'b00001001: //clr
342             begin
343                 load_instruction <= 0;
344                 if (source_register == 4'b0001)
345                     ALU_control <= 4'b1001;
346                 else
347                     ALU_control <= 4'b0000;
348                 if (source_register == 4'b0001)
349                     IDC_control <= 2'b00;
350                 else
351                     IDC_control <= 2'b11;
352                 select_source <= 4'b0000;
353                 select_destination <= 3'b000;
```

```
                PC_control <= 2'b00;
                MDR_control <= 2'b00;
                MAR_control <= 2'b00;
                write_DRAM <= 0;
                start_Tx <= 0;
                state <= 8'b00010001; //fetch1
            end

        8'b00000001: //load1
            begin
                load_instruction <= 0;
                ALU_control <= 4'b0000;
                select_source <= 4'b0000;
                select_destination <= 3'b000;
                PC_control <= 2'b00;
                IDC_control <= 2'b00;
                MDR_control <= 2'b00;
                if (source_register == 4'b0001)
                    MAR_control <= 2'b01;
                else if (source_register == 4'b0010)
                    MAR_control <= 2'b10;
                else
                    MAR_control <= 2'b00;
                write_DRAM <= 0;
                start_Tx <= 0;
                state <= 8'b00011000; //load2
            end

        8'b00011000: //load2
            begin
                load_instruction <= 0;
                ALU_control <= 4'b0000;
                select_source <= 4'b0000;
                select_destination <= 3'b000;
                PC_control <= 2'b00;
                IDC_control <= 2'b00;
                MDR_control <= 2'b01;
                MAR_control <= 2'b00;
                write_DRAM <= 0;
                start_Tx <= 0;
                state <= 8'b00010001; //fetch1
            end

        8'b00000010: //store1
            begin
                load_instruction <= 0;
                ALU_control <= 4'b0000;
                select_source <= 4'b0000;
                select_destination <= 3'b000;
                PC_control <= 2'b00;
                IDC_control <= 2'b00;
                MDR_control <= 2'b00;
                if (source_register == 4'b0001)
                    MAR_control <= 2'b01;
                else if (source_register == 4'b0010)
                    MAR_control <= 2'b11;
                else
                    MAR_control <= 2'b00;
                write_DRAM <= 0;
                start_Tx <= 0;
                state <= 8'b00011001; //store2
            end

        8'b00011001: //store2
            begin
                load_instruction <= 0;
                ALU_control <= 4'b0000;
                select_source <= 4'b0000;
                select_destination <= 3'b000;
                PC_control <= 2'b00;
                IDC_control <= 2'b00;
                MDR_control <= 2'b00;
                MAR_control <= 2'b00;
                write_DRAM <= 1;
                start_Tx <= 0;
                state <= 8'b00010001; //fetch1
            end
```

```
        8'b00001111: //end
            begin
                load_instruction <= 0;
                ALU_control <= 4'b0000;
                select_source <= 4'b0000;
                select_destination <= 3'b000;
                PC_control <= 2'b11;
                IDC_control <= 2'b00;
                MDR_control <= 2'b00;
                MAR_control <= 2'b00;
                write_DRAM <= 0;
                start_Tx <= 1;
                if (Tx_done)
                begin
                    state <= 8'b00010000; //start
                end
                else
                begin
                    state <= 8'b00001111; //end
                end
            end

        default: state <= 8'b00010000; //start
      endcase
    end
endmodule
```

13.1.4 Arithmetic and Logic Unit (ALU)

```
//Arithmetic and logic unit for arithmetic and logical operations

module ALU(input wire [15:0] B,
           input wire [3:0] ALU_control,
           input wire [15:0] instruction,
           input wire clock,
           output wire [15:0] AC_out,
           output wire [7:0] AC_disp,
           output wire Z_out);

reg [15:0] AC;
reg [11:0] load_constant;
reg Z;

assign AC_out = AC;
assign AC_disp = AC[7:0];
assign Z_out = Z;

initial
begin
    AC = 16'b0;
end

always@(instruction)
begin
    load_constant = instruction[11:0];
end

always @(AC)
    begin
        if(AC==16'b0)
            Z <= 1;
        else
            Z <= 0;
    end

always @(posedge clock)
begin
    case(ALU_control)

    4'b0000: //no change
    begin
        AC <= AC;
    end
```

```
46          4'b0001: //addition
47          begin
48              AC <= AC+B;
49          end
50
51          4'b0010: //substraction
52          begin
53              if(AC>B)
54                  AC <= AC-B;
55              else
56                  AC <= B-AC;
57          end
58
59          4'b0011: //multiplication
60          begin
61              AC <= AC*B;
62          end
63
64          4'b0100: //division
65          begin
66              AC <= AC/B;
67          end
68
69          4'b0101: //substitution
70          begin
71              AC <= B;
72          end
73
74          4'b0110: //loading constant value
75          begin
76              AC <= {4'b0,load_constant};
77          end
78
79          4'b0111: //increment
80          begin
81              AC <= AC + 16'd1;
82          end
83
84          4'b1000: //decrement
85          begin
86              AC <= AC - 16'd1;
87          end
88
89          4'b1001: //clear
90          begin
91              AC <= 16'b0;
92          end
93
94          default: //clear
95          begin
96              AC <= AC;
97          end
98          endcase
99      end
100
101     endmodule
```

13.1.5 Data Bus

```
//Data bus used to transfer data between registers

module Bus(input wire [15:0] AC_in,
                input wire [7:0] SR1_in,
                input wire [7:0] SR2_in,
                input wire [7:0] SR3_in,
                input wire [15:0] R1_in,
                input wire [15:0] R2_in,
                input wire [7:0] RRR_in,
                input wire [7:0] CRR_in,
                input wire [7:0] MDR_in,
                input wire [3:0] select_source,
                input wire [2:0] select_destination,
                input wire [15:0] instruction,
                input wire clock,
                output reg [7:0] SR1_out,
                output reg [15:0] R1_out,
                output reg [15:0] R2_out,
                output wire [7:0] MDR_out,
                output wire [15:0] B_out);

reg [15:0] value;
reg [3:0] multiply_divide_constant;
assign B_out = value;

assign MDR_out = value[7:0];

initial
begin
    value = 16'b0;
end

always@(instruction)
begin
    multiply_divide_constant <= instruction[11:8];
end

//reading data ino the bus

always@(posedge clock)
    begin
        case(select_source)

        4'b0000:
            begin
                value <= value;
            end

        4'b0001:
            begin
                value <= AC_in;
            end

        4'b0101:
            begin
                value <= {8'b0,MDR_in};
            end

        4'b0110:
            begin
                value <= {8'b0,SR1_in};
            end

```

```
64          4'b0111:
65              begin
66                  value <= {8'b0,SR2_in};
67              end
68
69          4'b1000:
70              begin
71                  value <= {8'b0,SR3_in};
72              end
73
74          4'b0010:
75              begin
76                  value <= R1_in;
77              end
78
79          4'b0011:
80              begin
81                  value <= R2_in;
82              end
83
84          4'b1001:
85              begin
86                  value <= {8'b0,RRR_in};
87              end
88
89          4'b1010:
90              begin
91                  value <= {8'b0,CRR_in};
92              end
93
94          4'b1011:
95              begin
96                  value <= {12'b0,multiply_divide_constant};
97              end
98
99          endcase
100     end
101
102 //writing data from the bus
103
104 always@(posedge clock)
105     begin
106         case(select_destination)
107
108         3'b000:
109             begin
110                 value <= value;
111             end
112
113         3'b010:
114             begin
115                 R1_out <= value;
116             end
117
118         3'b011:
119             begin
120                 R2_out <= value;
121             end
122
123         3'b110:
124             begin
125                 SR1_out <= value[7:0];
126             end
127
128         endcase
129     end
130
131 endmodule
```

13.1.6 Increment, Decrement and Clear Control Unit (IDC Controller)

```
//IDC control unit to increment, decrement and clear registers

module IDC_controller(input wire [1:0] IDC_control,
                      input wire [15:0] instruction,
                      input wire clock,
                      output reg [7:0] IDC_control_RRR,
                      output reg [7:0] IDC_control_RWR,
                      output reg [7:0] IDC_control_CRR,
                      output reg [7:0] IDC_control_CWR);

reg [3:0] IDC_register;

always@(instruction)
begin
    IDC_register <= instruction[11:8];
end

always@(posedge clock)
begin
    case(IDC_control)
    2'b00: //do nothing
        begin
            IDC_control_RRR <= IDC_control_RRR;
            IDC_control_RWR <= IDC_control_RWR;
            IDC_control_CRR <= IDC_control_CRR;
            IDC_control_CWR <= IDC_control_CWR;
        end

    2'b01: //increment
        begin
            case(IDC_register)
            4'b1001:
                begin
                    IDC_control_RRR <= IDC_control_RRR + 8'd1;
                end

            4'b1010:
                begin
                    IDC_control_CRR <= IDC_control_CRR + 8'd1;
                end

            4'b1011:
                begin
                    IDC_control_RWR <= IDC_control_RWR + 8'd1;
                end

            4'b1100:
                begin
                    IDC_control_CWR <= IDC_control_CWR + 8'd1;
                end

            default:
                begin
                    IDC_control_RRR <= IDC_control_RRR;
                    IDC_control_RWR <= IDC_control_RWR;
                    IDC_control_CRR <= IDC_control_CRR;
                    IDC_control_CWR <= IDC_control_CWR;
                end
            endcase
        end

    2'b10: //decrement
        begin
            case(IDC_register)
            4'b1001:
                begin
                    IDC_control_RRR <= IDC_control_RRR - 8'd1;
                end

```

```
70                  4'b1010:
71                      begin
72                          IDC_control_CRR <= IDC_control_CRR - 8'd1;
73                      end
74
75                  4'b1011:
76                      begin
77                          IDC_control_RWR <= IDC_control_RWR - 8'd1;
78                      end
79
80                  4'b1100:
81                      begin
82                          IDC_control_CWR <= IDC_control_CWR - 8'd1;
83                      end
84
85                  default:
86                      begin
87                          IDC_control_RRR <= IDC_control_RRR;
88                          IDC_control_RWR <= IDC_control_RWR;
89                          IDC_control_CRR <= IDC_control_CRR;
90                          IDC_control_CWR <= IDC_control_CWR;
91                      end
92                  endcase
93              end
94
95          2'b11: //clear
96              begin
97                  case(IDC_register)
98                  4'b1001:
99                      begin
100                         IDC_control_RRR <= 8'b0;
101                     end
102
103                 4'b1010:
104                     begin
105                         IDC_control_CRR <= 8'b0;
106                     end
107
108                 4'b1011:
109                     begin
110                         IDC_control_RWR <= 8'b0;
111                     end
112
113                 4'b1100:
114                     begin
115                         IDC_control_CWR <= 8'b0;
116                     end
117
118                 default:
119                     begin
120                         IDC_control_RRR <= IDC_control_RRR;
121                         IDC_control_RWR <= IDC_control_RWR;
122                         IDC_control_CRR <= IDC_control_CRR;
123                         IDC_control_CWR <= IDC_control_CWR;
124                     end
125                 endcase
126             end
127
128         default:
129             begin
130                 IDC_control_RRR <= IDC_control_RRR;
131                 IDC_control_RWR <= IDC_control_RWR;
132                 IDC_control_CRR <= IDC_control_CRR;
133                 IDC_control_CWR <= IDC_control_CWR;
134             end
135     endcase
136 end
137
138 endmodule
```

13.1.7 Instruction Register (IR)

```
//Instruction Register used to store loaded instructions

module IR(input wire [15:0] IRAM_data,
          input wire load_instruction,
          input wire clock,
          output reg [15:0] instruction);

initial
begin
    instruction=16'b0;
end

always @(posedge load_instruction)
begin
    instruction <= IRAM_data;
end

endmodule
```

13.1.8 Memory Address Register (MAR)

```
//Memory Address Register used to store the DRAM address

module MAR(input wire [15:0] AC_to_MAR,
           input wire [15:0] RR_to_MAR,
           input wire [15:0] WR_to_MAR,
           input wire clock,
           input wire [1:0] MAR_control,
           output wire [15:0] MAR_to_DRAM);

reg [15:0] MAR;

assign MAR_to_DRAM = MAR;

initial
begin
    MAR = 16'b0;
end

always@(posedge clock)
begin
    case(MAR_control)
    2'b00:
        begin
            MAR <= MAR;
        end

    2'b01:
        begin
            MAR <= AC_to_MAR;
        end

    2'b10:
        begin
            MAR <= RR_to_MAR;
        end

    2'b11:
        begin
            MAR <= WR_to_MAR;
        end

    default:
        begin
            MAR <= MAR;
        end
    endcase
end

endmodule
```

13.1.9 Memory Data Register (MDR)

```
1  //Memory Data Register used to store the DRAM data
2
3  module MDR(input wire [7:0] DRAM_to_MDR,
4             input wire [7:0] bus_to_MDR,
5             input wire [1:0] MDR_control,
6             input wire clock,
7             output wire [7:0] MDR_to_bus,
8             output wire [7:0] MDR_to_DRAM);
9
10 reg [7:0] MDR;
11
12 assign MDR_to_bus = MDR;
13 assign MDR_to_DRAM = MDR;
14
15 initial
16 begin
17     MDR=8'b0;
18 end
19
20 always@(posedge clock)
21 begin
22     case(MDR_control)
23     2'b00:
24         begin
25         MDR <= MDR;
26         end
27
28     2'b01:
29         begin
30         MDR <= DRAM_to_MDR;
31         end
32
33     2'b10:
34         begin
35         MDR <= bus_to_MDR;
36         end
37
38     default:
39         begin
40         MDR <= MDR;
41         end
42     endcase
43 end
44
45 endmodule
```

13.1.10 Program Counter (PC)

```
//Program Counter used to store the IRAM address to load instructions

module PC(input wire Z_out,
          input wire [1:0] PC_control,
          input wire [15:0] instruction,
          input wire clock,
          output wire [7:0] IRAM_address);

reg [7:0] PC;
reg [7:0] jump_address;
reg [3:0] jump_condition;

assign IRAM_address = PC;

initial
begin
    PC <= 8'b0;
end

always@(instruction)
begin
    jump_address <= instruction[7:0];
    jump_condition <= instruction[11:8];
end

always@(posedge clock)
begin
    if (PC_control==2'b10) //jump
    begin
        case(jump_condition)

        4'b0000: //unconditional jump
            begin
            PC <= jump_address;
            end

        4'b0001: //jump when z=0
            if (Z_out==0)
            begin
                PC <= jump_address;
            end
            else
                PC <= PC;

        4'b0010: //jump when z=1
            if (Z_out==1)
            begin
                PC <= jump_address;
            end
            else
                PC <= PC;


        default: PC <= PC;
        endcase
    end

    else if (PC_control==2'b01) //increment
        PC <= PC + 8'd1;

    else if (PC_control==2'b11) //clear
        PC <= 0;

    else
        PC <= PC;

end

endmodule
```

13.1.11 General Purpose Registers (R1 & R2)

```
//General purpose registers used as R1 and R2

module R(input [15:0] bus_to_R,
         output [15:0] R_to_bus);

reg [15:0] value;

assign R_to_bus = value;

initial
begin
    value <= 16'b0;
end

always@(bus_to_R)
begin
    value = bus_to_R;
end

endmodule
```

13.1.12 Shift Registers (SR1, SR2 & SR3)

```
//Shift register bank used to store pixel data

module shift_registers(input wire [7:0] bus_to_SR1,
                       output wire [7:0] SR1_to_bus,
                       output wire [7:0] SR2_to_bus,
                       output wire [7:0] SR3_to_bus);

reg [7:0] SR1;
reg [7:0] SR2;
reg [7:0] SR3;

assign SR1_to_bus = SR1;
assign SR2_to_bus = SR2;
assign SR3_to_bus = SR3;

initial
begin
    SR1 <= 8'b0;
    SR2 <= 8'b0;
    SR3 <= 8'b0;
end


always@(bus_to_SR1 or SR1 or SR2)
begin
        SR3 <= SR2;
        SR2 <= SR1;
        SR1 <= bus_to_SR1;
end

endmodule
```

13.1.13 Read Registers (CRR & RRR)

```
//Read registers RRR and CRR used to read image pixels

module Read_registers(input [7:0] IDC_control_RRR,
                      input [7:0] IDC_control_CRR,
                      output [15:0] RR_to_MAR,
                      output [7:0] RRR_out,
                      output [7:0] CRR_out);

reg [7:0] RRR;
reg [7:0] CRR;

assign RRR_out = RRR;
assign CRR_out = CRR;
assign RR_to_MAR = {RRR,CRR};

initial
begin
    RRR <= 8'b0;
    CRR <= 8'b0;
end

always@(IDC_control_RRR)
begin
    RRR <= IDC_control_RRR;
end

always@(IDC_control_CRR)
begin
    CRR <= IDC_control_CRR;
end

endmodule
```

13.1.14 Write Registers (CWR & RWR)

```
//Write registers RWR and CWR used to write the image pixels

module Write_registers(input [7:0] IDC_control_RWR,
                       input [7:0] IDC_control_CWR,
                       output [15:0] WR_to_MAR);

reg [7:0] RWR;
reg [7:0] CWR;

assign WR_to_MAR = {RWR,CWR};

initial
begin
    RWR <= 8'b0;
    CWR <= 8'b0;
end

always@(IDC_control_RWR)
begin
    RWR <= IDC_control_RWR;
end

always@(IDC_control_CWR)
begin
    CWR <= IDC_control_CWR;
end

endmodule
```

13.1.15 Instruction Memory (IRAM)

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

// ============================================================
// File Name: IRAM.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 IRAM (
    address,
    clock,
    data,
    wren,
    q);

    input   [7:0]  address;
    input     clock;
    input   [15:0]  data;
    input     wren;
    output  [15:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

```

```
    wire [15:0] sub_wire0;
    wire [15:0] q = sub_wire0[15:0];

    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 (1'b1),
                .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.init_file = "Instructions.mif",
        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 = 256,
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
        altsyncram_component.widthad_a = 8,
        altsyncram_component.width_a = 16,
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// 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 "0"
// 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 "Instructions.mif"
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "256"
// 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 "0"
// 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 "8"
// Retrieval info: PRIVATE: WidthData NUMERIC "16"
// 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: INIT_FILE STRING "Instructions.mif"
// 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 "256"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
// 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 "8"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "16"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 8 0 INPUT NODEFVAL "address[7..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: data 0 0 16 0 INPUT NODEFVAL "data[15..0]"
// Retrieval info: USED_PORT: q 0 0 16 0 OUTPUT NODEFVAL "q[15..0]"
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren"
// Retrieval info: CONNECT: @address_a 0 0 8 0 address 0 0 8 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 16 0 data 0 0 16 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: q 0 0 16 0 @q_a 0 0 16 0
// Retrieval info: GEN_FILE: TYPE_NORMAL IRAM.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL IRAM.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL IRAM.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL IRAM.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL IRAM_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL IRAM_bb.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL IRAM_syn.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```

13.1.16 Data Memory (DRAM)

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

// ============================================================
// File Name: DRAM.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 DRAM (
    address,
    clock,
    data,
    wren,
    q);

    input   [15:0]  address;
    input     clock;
    input   [7:0]  data;
    input     wren;
    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 (1'b1),
                .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.init_file = "Data.mif",
        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 = 65536,
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
        altsyncram_component.widthad_a = 16,
        altsyncram_component.width_a = 8,
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// 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 "0"
// 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 "Data.mif"
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "65536"
// 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 "0"
// 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 "16"
// 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: INIT_FILE STRING "Data.mif"
// 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 "65536"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
// 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 "16"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 16 0 INPUT NODEFVAL "address[15..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 16 0 address 0 0 16 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 DRAM.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL DRAM.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL DRAM.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL DRAM.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL DRAM_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL DRAM_bb.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL DRAM_syn.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```

13.1.17 DRAM Address Multiplexer

```
//DRAM_address_mux for setting DRAM address using processor and transceiver

module DRAM_address_mux(input wire [15:0] inputmux1,
                        input wire [15:0] inputmux2,
                        input wire start_Tx,
                        input wire Rx_done,
                        output wire [15:0] outputmux);

assign outputmux = start_Tx ? (Rx_done ? inputmux2 : inputmux2) : (Rx_done ? inputmux1 : inputmux2);

endmodule
```

13.1.18 DRAM Write Data Multiplexer

```
//DRAM_write_data_mux for setting DRAM writing data using processor and transceiver

module DRAM_write_data_mux(input wire [7:0] inputmux1,
                           input wire [7:0] inputmux2,
                           input wire select,
                           output wire [7:0] outputmux);

assign outputmux = select ? inputmux1:inputmux2;

endmodule
```

13.1.19 DRAM Write Enable Multiplexer

```
//DRAM_write_data_mux for enabling DRAM writing using processor and transceiver

module DRAM_write_enable_mux(input wire inputmux1,
                             input wire inputmux2,
                             input wire select,
                             output wire outputmux);

assign outputmux = select ? inputmux1:inputmux2;

endmodule
```

13.1.20 DRAM Read Data Splitter

```
//DRAM_read_data_splitter for splitting DRAM data into processor and transceiver

module DRAM_read_data_splitter(input wire [7:0] inputsplitter,
                               output wire [7:0] output1,
                               output wire [7:0] output2);

assign output1 = inputsplitter;
assign output2 = inputsplitter;

endmodule
```

13.1.21 25MHz PLL Clock

```
// megafunction wizard: %ALTPLL%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altpll

// ============================================================
// File Name: pll_25mhz.v
// Megafunction Name(s):
//          altpll
//
// Simulation Library Files(s):
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 15.0.0 Build 145 04/22/2015 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2015 Altera Corporation. All rights reserved.
//Your use of Altera 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 Altera Program License
//Subscription Agreement, the Altera Quartus II License Agreement,
//the Altera MegaCore Function License Agreement, or other
//applicable license agreement, including, without limitation,
//that your use is for the sole purpose of programming logic
//devices manufactured by Altera and sold by Altera 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 pll_25mhz (
    areset,
    inclk0,
    c0,
    locked);

    input     areset;
    input     inclk0;
    output    c0;
    output    locked;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri0      areset;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [0:0] sub_wire2 = 1'h0;
    wire [4:0] sub_wire3;
    wire  sub_wire5;
    wire  sub_wire0 = inclk0;
    wire [1:0] sub_wire1 = {sub_wire2, sub_wire0};
    wire [0:0] sub_wire4 = sub_wire3[0:0];
    wire  c0 = sub_wire4;
    wire  locked = sub_wire5;
```

```
66
67      altpll  altpll_component (
68                  .areset (areset),
69                  .inclk (sub_wire1),
70                  .clk (sub_wire3),
71                  .locked (sub_wire5),
72                  .activeclock (),
73                  .clkbad (),
74                  .clkena ({6{1'b1}}),
75                  .clkloss (),
76                  .clkswitch (1'b0),
77                  .configupdate (1'b0),
78                  .enable0 (),
79                  .enable1 (),
80                  .extclk (),
81                  .extclkena ({4{1'b1}}),
82                  .fbin (1'b1),
83                  .fbmimicbidir (),
84                  .fbout (),
85                  .fref (),
86                  .icdrclk (),
87                  .pfdena (1'b1),
88                  .phasecounterselect ({4{1'b1}}),
89                  .phasedone (),
90                  .phasestep (1'b1),
91                  .phaseupdown (1'b1),
92                  .pllena (1'b1),
93                  .scanaclr (1'b0),
94                  .scanclk (1'b0),
95                  .scanclkena (1'b1),
96                  .scandata (1'b0),
97                  .scandataout (),
98                  .scandone (),
99                  .scanread (1'b0),
100                 .scanwrite (1'b0),
101                 .sclkout0 (),
102                 .sclkout1 (),
103                 .vcooverrange (),
104                 .vcounderrange ());
105     defparam
106         altpll_component.bandwidth_type = "AUTO",
107         altpll_component.clk0_divide_by = 2,
108         altpll_component.clk0_duty_cycle = 50,
109         altpll_component.clk0_multiply_by = 1,
110         altpll_component.clk0_phase_shift = "0",
111         altpll_component.compensate_clock = "CLK0",
112         altpll_component.inclk0_input_frequency = 20000,
113         altpll_component.intended_device_family = "Cyclone IV E",
114         altpll_component.lpm_hint = "CBX_MODULE_PREFIX=pll_25mhz",
115         altpll_component.lpm_type = "altpll",
116         altpll_component.operation_mode = "NORMAL",
117         altpll_component.pll_type = "AUTO",
118         altpll_component.port_activeclock = "PORT_UNUSED",
119         altpll_component.port_areset = "PORT_USED",
120         altpll_component.port_clkbad0 = "PORT_UNUSED",
121         altpll_component.port_clkbad1 = "PORT_UNUSED",
122         altpll_component.port_clkloss = "PORT_UNUSED",
123         altpll_component.port_clkswitch = "PORT_UNUSED",
124         altpll_component.port_configupdate = "PORT_UNUSED",
```

```
        altpll_component.port_fbin = "PORT_UNUSED",
        altpll_component.port_inclk0 = "PORT_USED",
        altpll_component.port_inclk1 = "PORT_UNUSED",
        altpll_component.port_locked = "PORT_USED",
        altpll_component.port_pfdena = "PORT_UNUSED",
        altpll_component.port_phasecounterselect = "PORT_UNUSED",
        altpll_component.port_phasedone = "PORT_UNUSED",
        altpll_component.port_phasestep = "PORT_UNUSED",
        altpll_component.port_phaseupdown = "PORT_UNUSED",
        altpll_component.port_pllena = "PORT_UNUSED",
        altpll_component.port_scanaclr = "PORT_UNUSED",
        altpll_component.port_scanclk = "PORT_UNUSED",
        altpll_component.port_scanclkena = "PORT_UNUSED",
        altpll_component.port_scandata = "PORT_UNUSED",
        altpll_component.port_scandataout = "PORT_UNUSED",
        altpll_component.port_scandone = "PORT_UNUSED",
        altpll_component.port_scanread = "PORT_UNUSED",
        altpll_component.port_scanwrite = "PORT_UNUSED",
        altpll_component.port_clk0 = "PORT_USED",
        altpll_component.port_clk1 = "PORT_UNUSED",
        altpll_component.port_clk2 = "PORT_UNUSED",
        altpll_component.port_clk3 = "PORT_UNUSED",
        altpll_component.port_clk4 = "PORT_UNUSED",
        altpll_component.port_clk5 = "PORT_UNUSED",
        altpll_component.port_clkena0 = "PORT_UNUSED",
        altpll_component.port_clkena1 = "PORT_UNUSED",
        altpll_component.port_clkena2 = "PORT_UNUSED",
        altpll_component.port_clkena3 = "PORT_UNUSED",
        altpll_component.port_clkena4 = "PORT_UNUSED",
        altpll_component.port_clkena5 = "PORT_UNUSED",
        altpll_component.port_extclk0 = "PORT_UNUSED",
        altpll_component.port_extclk1 = "PORT_UNUSED",
        altpll_component.port_extclk2 = "PORT_UNUSED",
        altpll_component.port_extclk3 = "PORT_UNUSED",
        altpll_component.self_reset_on_loss_lock = "OFF",
        altpll_component.width_clock = 5;

endmodule
```

13.1.22 Slow Clock

```
//slow clock used for debugging

module slow_clock(clkin,clkout);

reg [24:0] counter;
output reg clkout;
input clkin;

initial begin
    counter = 0;
    clkout = 0;
end

always @(posedge clkin) begin
    if (counter == 0) begin
        counter <= 24999999;
        clkout <= ~clkout;
    end else begin
        counter <= counter - 25'd1;
    end
end
endmodule
```

13.1.23 Clock Selector

```
//Clock selecter module for switching between slow_clock and fast_clock

module clock_selector(input clk_in,
                      input clock_select,
                      output wire clk_out);

wire c0;
wire c1;

pll_25mhz fast_clock(.inclk0(clk_in), .c0(c0));
slow_clock slow_clock(.clkin(clk_in), .clkout(c1));

assign clk_out = clock_select ? c0 : c1;

endmodule
```

13.1.24 State to 7 Segment Display Module

```
//state_to_7seg module used do display the state number on 7 segments for debugging

module state_to_7seg(bin,dout0,dout1);

input [5:0] bin;
reg [5:0] D0;
reg [5:0] D1;

output [6:0] dout0;
output [6:0] dout1;

decoder d0(.din(D0[3:0]),.dout(dout0));
decoder d1(.din(D1[3:0]),.dout(dout1));

always@(bin)
begin
    if (bin < 6'd10)
        begin
        D0 = bin;
        D1 = 6'd0;
        end
    else
        if(bin < 6'd20)
            begin
            D0 = bin - 6'd10;
            D1 = 6'd1;
            end
        else
            if(bin < 6'd30)
                begin
                D0 = bin - 6'd20;
                D1 = 6'd2;
                end
            else
                if(bin < 6'd40)
                    begin
                    D0 = bin - 6'd30;
                    D1 = 6'd3;
                    end
                else
                    if(bin < 6'd50)
                    begin
                        D0 = bin - 6'd40;
                        D1 = 6'd5;
                    end
                    else
                        begin
                        D0 = 6'd0;
                        D1 = 6'd0;
                        end
end

endmodule
```

13.1.25 AC to 7 Segment Display Module

```
//AC_to_7seg module used do display the value of AC on 7 segments for debugging

module AC_to_7seg(din,dout2,dout1,dout0);

input  [7:0] din;
output [6:0] dout0;
output [6:0] dout1;
output [6:0] dout2;

reg [3:0]  counter=3'b0;
reg [19:0] shifter=20'd0;

decoder d0(.din(shifter[11:8]),.dout(dout0));
decoder d1(.din(shifter[15:12]),.dout(dout1));
decoder d2(.din(shifter[19:16]),.dout(dout2));

always @(din)
begin
    shifter[7:0]=din;
    shifter[19:8]=12'b0;
    for (counter=4'd0;counter<4'd8;counter=counter+4'd1)
    begin
        if(shifter[11:8]>4'd4)
            begin
                shifter[11:8]=shifter[11:8]+4'd3;
            end
        if(shifter[15:12]>4'd4)
            begin
                shifter[15:12]=shifter[15:12]+4'd3;
            end
        if(shifter[19:16]>4'd4)
            begin
                shifter[19:16]=shifter[19:16]+4'd3;
            end
        shifter=shifter<<1;
    end
end

endmodule
```

13.1.26 BCD into 7 Segment Display Decoder

```
 //decoder module used for decoding BCD into 7 segment display

module decoder(input wire [3:0] din,
               output reg [6:0] dout);

 always@(din)
case(din)
     4'd0:dout<=7'b1000000;
     4'd1:dout<=7'b1111001;
     4'd2:dout<=7'b0100100;
     4'd3:dout<=7'b0110000;
     4'd4:dout<=7'b0011001;
     4'd5:dout<=7'b0010010;
     4'd6:dout<=7'b0000010;
     4'd7:dout<=7'b1111000;
     4'd8:dout<=7'b0000000;
     4'd9:dout<=7'b0011000;
 default:dout<=7'b0000000;
endcase

 endmodule
```

13.1.27 Transceiver

```
//Transceiver module for governing the communication process through UART

module transceiver(input wire Rx,
                   input wire start_Tx,
                   input wire clk,
                   input wire [7:0] data_in,
                   output wire Tx,
                   output wire ready,
                   output wire Rx_done,
                   output wire Tx_done,
                   output wire [7:0] data_out,
                   output wire [15:0] D_address);

UART UART(.data_in(data_in),
          .wr_en(wr_en),
          .clk(clk),
          .Tx(Tx),
          .Rx(Rx),
          .Tx_busy(tb),
          .Rx_ready(Rx_ready),
          .data_out(data_out));

 wire tb;
 wire wr_en;
 wire Rx_ready;

 reg en;
 reg busy;
 reg status1; //1st bit state
 reg status2; //Rx done state
 reg status3; //Tx done state
 reg ready1;
 reg ready2;
 reg counter2;

 reg[15:0] W_address;    //write address to dram
 reg [15:0] R_address;   //read address from dram

 assign D_address = status2 ? R_address : W_address; //mux to select address
 assign Rx_done = status2;
 assign wr_en = en;
 assign ready=(~ready2) && ready1; //positive edge detection
 assign Tx_done=status3;

 initial
begin
    status1=0;
    status2=0;
    status3=0;
    W_address=0;
    R_address=0;
    counter2=0;
    en=1;
end

 always@(posedge ready)
begin
    if(~status1 && ~status2)
    begin
        status1=1;
    end

    else if(status1 && ~status2 && W_address < 65535)
        W_address=W_address+1;

```

```
    if(W_address == 65535)
        begin
            status2=1;
            W_address=0;
        end
end

always@(posedge clk)
begin
    busy = tb;
    ready1 <= Rx_ready;
    ready2 <= ready1;
    if (start_Tx && status2 && ~busy && R_address < 65535 && counter2 < 2)
    begin
        en = 0;
        counter2 = counter2 + 1;
    end

    else if (start_Tx && status2 && ~busy && R_address < 65535 && counter2 == 2)
    begin
        R_address = R_address + 1;
        en = 0;
    end

    else if (R_address == 65535 && status2 && ~busy)
    begin
        status3 = 1;
        en = 1;
    end
end

endmodule
```

13.1.28 UART Module

```
//UART module used for combining transmitter, receiver and baudrate modules

module UART(input wire [7:0] data_in, //input data
            input wire wr_en, //enable tx
            input wire clk,
            output wire Tx,
            input wire Rx,
            output wire Tx_busy,
            output wire Rx_ready,
            output wire [7:0] data_out);

wire Txclk_en, Rxclk_en;
wire Tx_en;

baudrate uart_baud( .clk_50m(clk),
                    .Rxclk_en(Rxclk_en),
                    .Txclk_en(Txclk_en));

transmitter uart_Tx(.data_in(data_in),
                    .wr_en(wr_en),
                    .clk_50m(clk),
                    .clken(Txclk_en), //We assign Tx clock to enable clock
                    .Tx(Tx),
                    .Tx_busy(Tx_busy));

receiver uart_Rx(.Rx(Rx),
                 .new_data_av(Rx_ready),
                 .clk_50m(clk),
                 .clken(Rxclk_en), //We assign Tx clock to enable clock
                 .data(data_out));

endmodule
```

13.1.29 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 = TX_STATE_IDLE; //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) //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
            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
```

13.1.30 Receiver

```
module receiver  (input wire Rx,
                  input wire clk_50m,
                  input wire clken,
                  output reg [7:0] data,
                  output wire new_data_av);

initial begin
    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;

assign new_data_av= (state==2'b10)? 1'b1:1'b0;

reg [1:0] state = RX_STATE_START; // state is a 2-bit register/vector,initially equal to 00
reg [3:0] sample = 0; // 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

always @(posedge clk_50m) begin

    if (clken) begin
        case (state) // Let us consider the 3 states of the receiver
        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_START;
                data <= scratch;
                sample <= 0;
            end
            else begin
                sample <= sample + 4'b1;
            end
        end
        default: begin
            state <= RX_STATE_START; // always begin with state assigned to START
        end
        endcase
    end
end

endmodule
```

13.1.31 Baud Rate

```
1  //This is a baud rate generator to divide a 50MHz clock into a 115200 baud Tx/Rx pair.
2  //The Rx clock oversamples by 16x.
3
4  module baudrate  (input wire clk_50m,
5                    output wire Rxclk_en,
6                    output wire Txclk_en);
7
8  //Our Testbench uses a 50 MHz clock.
9   //Want to interface to 115200 baud UART for Tx/Rx pair
10  //Hence, 50000000 / 115200 = 435 Clocks Per Bit.
11  parameter RX_ACC_MAX = 50000000 / (115200 * 16);
12  parameter TX_ACC_MAX = 50000000 / 115200;
13  parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);
14  parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);
15  reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;
16  reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;
17
18  assign Rxclk_en = (rx_acc == 5'd0);
19  assign Txclk_en = (tx_acc == 9'd0);
20
21 always @(posedge clk_50m) begin
22      if (rx_acc == RX_ACC_MAX[RX_ACC_WIDTH - 1:0])
23          rx_acc <= 0;
24      else
25          rx_acc <= rx_acc + 5'b1; //increment by 00001
26 end
27
28 always @(posedge clk_50m) begin
29      if (tx_acc == TX_ACC_MAX[TX_ACC_WIDTH - 1:0])
30          tx_acc <= 0;
31      else
32          tx_acc <= tx_acc + 9'b1; //increment by 000000001
33 end
34
35  endmodule
36
```

13.2 MATLAB Codes

13.2.1 Downsampling using an Average Filter (Grayscale Image)

```
delete(instrfind);
clear all;
close all;
clc;

image=imread('lena.bmp');
dest_image=zeros(256,256,'uint8');
sObject=serial('COM3','BaudRate',115200,'Terminator','LF','TimeOut',20);

N = 2;
K=floor(256/N);

fopen(sObject);
for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,1));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j)=fread(sObject,1,'uint8');
    end
end

fclose(sObject);

dest_image=uint8(dest_image);
figure;
imshow(dest_image);
title('Received Image');

down_sampled_image=zeros(128,128,'uint8');
for i=1:K
    for j=1:K
        down_sampled_image(i,j)=dest_image(i,j);
    end
end

figure;
imshow(down_sampled_image);
title('Down Sampled Image');

figure;
imshow(image);
title('Original Image');

image_MATLAB=zeros(128,128);
image_MATLAB=uint8(image_MATLAB);

for i=1:128
    for j=1:128
        image_MATLAB(i,j)=idivide(idivide(image(2*i-1,2*j-1)+image(2*i-1,2*j),2)+idivide(image(2*i,2*j-1)+image(2*i,2*j),2),2)
    end
end

figure;
imshow(image_MATLAB);
title('Down Sampled Image using MATLAB');

den=(uint8(image_MATLAB)-down_sampled_image).*(uint8(image_MATLAB)-down_sampled_image);
SSE=sum(sum(den))/(256*256/(N*N));

disp(strcat('Sum of Squared Errors =  ',num2str(SSE)));

fclose(sObject);
```

13.2.2 Downsampling using an Average Filter (RGB Image)

```
delete(instrfind);
clear all;
close all;
clc;

N=2; %down sampling factor
image=imread('got.jpg');
dest_image=zeros(256,256,3,'uint8');
sObject=serial('COM6','BaudRate',115200,'Terminator','LF','TimeOut',20);

fopen(sObject);
for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,1));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,1)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Red Plane');

fwrite(sObject,0);

for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,2));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');
dest_image(1,1,2)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,2)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Red and Green Planes');

fwrite(sObject,0);

for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,3));
    end
end

```

```
fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');
dest_image(1,1,2)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,3)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Three planes');

K=floor(256/N);
down_sampled_image=zeros(K,K,3,'uint8');

for i=1:K
    for j=1:K
        for k=1:3
            down_sampled_image(i,j,k)=dest_image(i,j,k);
        end
    end
end

figure;
imshow(image);
title('Original Image');

figure;
imshow(down_sampled_image);
title('Down Sampled Image');

image_MATLAB=zeros(128,128,3);
image_MATLAB=uint8(image_MATLAB);

for k=1:3
    for i=1:128
        for j=1:128
            image_MATLAB(i,j,k)=idivide(idivide(image(2*i-1,2*j-1,k)+image(2*i-1,2*j,k),2)+idivide(image(2*i,2*j-1,k)+image(2*i,2*j,k),2),2);
        end
    end
end

figure;
imshow(image_MATLAB);
title('Down Sampled Image using MATLAB');

den=(uint8(image_MATLAB)-down_sampled_image).*(uint8(image_MATLAB)-down_sampled_image);
SSE=sum(sum(sum(den)))/(256*256*3/(N^2));

disp(strcat('Sum of Squared Errors = ',num2str(SSE)));

fclose(sObject);
```

13.2.3 Downsampling using a Gaussian Filter (Grayscale Image)

```
delete(instrfind);
clear all;
close all;
clc;

image=imread('lena.bmp');
dest_image=zeros(256,256,'uint8');
sObject=serial('COM3','BaudRate',115200,'Terminator','LF','TimeOut',20);

N = 2;
K=floor(256/N);

fopen(sObject);
for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,1));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j)=fread(sObject,1,'uint8');
    end
end

fclose(sObject);

dest_image=uint8(dest_image);
figure;
imshow(dest_image);
title('Received Image');

down_sampled_image=zeros(128,128,'uint8');
for i=1:K
    for j=1:K
        down_sampled_image(i,j)=dest_image(i,j);
    end
end

figure;
imshow(down_sampled_image);
title('Down Sampled Image');

figure;
imshow(image);
title('Original Image');

fclose(sObject);
```

13.2.4 Downsampling using a Gaussian Filter (RGB Image)

```
delete(instrfind);
clear all;
close all;
clc;

N=3; %down sampling factor
image=imread('got.jpg');
dest_image=zeros(256,256,3,'uint8');
sObject=serial('COM6','BaudRate',115200,'Terminator','LF','TimeOut',20);

fopen(sObject);
for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,1));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,1)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Red Plane');

fwrite(sObject,0);

for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,2));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');
dest_image(1,1,2)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,2)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Red and Green Planes');

fwrite(sObject,0);

for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,3));
    end
end

```

```
fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');
dest_image(1,1,2)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,3)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Three planes');

K=floor(256/N);
down_sampled_image=zeros(K,K,3,'uint8');

for i=1:K
    for j=1:K
        for k=1:3
            down_sampled_image(i,j,k)=dest_image(i,j,k);
        end
    end
end

figure;
imshow(image);
title('Original Image');

figure;
imshow(down_sampled_image);
title('Down Sampled Image');

fclose(sObject);
```

13.2.5 Gaussian Smoothing (Grayscale Image)

```
delete(instrfind);
clear all;
close all;
clc;

image=imread('lena.bmp');
dest_image=zeros(256,256,'uint8');
sObject=serial('COM3','BaudRate',115200,'Terminator','LF','TimeOut',20);

fopen(sObject);
for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,1));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j)=fread(sObject,1,'uint8');
    end
end

fclose(sObject);

dest_image=uint8(dest_image);
figure;
imshow(dest_image);
title('Gaussian Smoothed Image');

figure;
imshow(image);
title('Original Image');

fclose(sObject);
```

13.2.6 Gaussian Smoothing (RGB Image)

```
delete(instrfind);
clear all;
close all;
clc;

image=imread('monalisa.jpg');
dest_image=zeros(256,256,3,'uint8');
sObject=serial('COM3','BaudRate',115200,'Terminator','LF','TimeOut',20);

fopen(sObject);
for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,1));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,1)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Red Plane');

fwrite(sObject,0);

for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,2));
    end
end

fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');
dest_image(1,1,2)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,2)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Received Image - Red and Green Planes');

fwrite(sObject,0);

for i=1:256
    for j=1:256
       fwrite(sObject,image(i,j,3));
    end
end

```

```
fwrite(sObject,0);
fwrite(sObject,0);

dest_image(1,1,1)=fread(sObject,1,'uint8');
dest_image(1,1,2)=fread(sObject,1,'uint8');

for i=1:256
    for j=1:256
        dest_image(i,j,3)=fread(sObject,1,'uint8');
    end
end

figure;
imshow(dest_image);
title('Gaussian Smoothed Image');

figure;
imshow(image);
title('Original Image');

fclose(sObject);
```

13.3 Python Codes

13.3.1 Compiler

```
def compiler(instruction):

    parts = instruction.split()

    instructions = ['NOP','LOAD','STORE','COPY','JUMP','ADD','SUB','MUL','DIV','CLR','INC','DEC','LOADK','END']
    inscode = ['0000','0001','0010','0011','0100','0101','0110','0111','1000','1001','1010','1011','1100','1111']

    registers= ['AC','R1','R2','MAR','MDR','SR1','SR2','SR3','RRR','CRR','RWR','CWR','RR','WR']
    registercode=['0001','0010','0011','0100','0101','0110','0111','1000','1001','1010','1011','1100','0010','0010']

    opcode = inscode[instructions.index(parts[0])]
    restricted = ['LOADK','JUMP','MUL','DIV']

    if parts[0] not in restricted :
        if len(parts)>1:
            opcode = opcode + registercode[registers.index(parts[1])]
            if len(parts)==2:
                opcode = opcode + '00000000'
            else:
                opcode = opcode + '0000'+registercode[registers.index(parts[2])]
        else:
            opcode = opcode + '000000000000'
        return opcode

    elif parts[0]=='IDLE':
        opcode='0000000000000000'

    else:
        if parts[0]=='LOADK':
            k = int(parts[1])
            kbin = bin(k)[2:]
            opcode = opcode+ (12-len(kbin))*'0' +kbin
        else:
            if parts[0]=='JUMP':
                jmpcond = ['Z=0','Z!=0','U']
                jmpcondcode = ['0001','0010', '0000']
                opcode = opcode+jmpcondcode[jmpcond.index(parts[1])]
                addr = int(parts[2])
                addrbin = bin(addr)[2:]
                opcode = opcode + (8-len(addrbin))*'0' + addrbin

            if parts[0]=='DIV' or parts[0]=='MUL':
                param =  bin(int(parts[1]))[2:]
                param = (4-len(param))*'0'+param
                opcode = opcode + param +'00000000'

    return opcode

file = open('Assembly code.txt','r')
hexfile = open('hexfile.txt','w')
lines =file.readlines()
opcodes = []

for i in range(len(lines)):
    opcodes.append(compiler(lines[i]))

for l in opcodes:
    print(l)
    hexfile.write(str(int(l,2)))
    hexfile.write("\n")
```