

Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN3030-Circuits and Systems Design



# DPUT Processor

Report on Verilog based design and implementation of an image  
down-sampling processor on FPGA

A.G.D.K. Anhettigama	160026P
S.P. Dissanayake	160134U
M. N. J. U. Indrajith	160221J
W. W. A. T. E. Weerasinghe	160680M

16th July, 2019

## Contents

1.0 Introduction .....	3
2.0 Instruction Set Architecture of DPUT processor .....	3
2.1 Instruction types .....	4
2.2 Registers & Flags .....	5
2.3 Instruction Set.....	5
2.4 Datapath .....	7
3.0 Modules .....	8
3.1 PROGRAM_COUNTER .....	8
3.2 AC_ALU.....	8
3.3 UART.....	9
3.3.1 Baudrate Generator .....	11
3.3.2 Transmitter .....	11
3.3.3 Receiver.....	12
3.4 INSTRUCTION_DECODER .....	13
3.5 LOOP_REGISTER .....	15
3.6 MEM_REGISTERS .....	15
3.7 REG_BANK.....	16
3.8 IRAM and DRAM .....	17
3.9 PROCESSOR_TOP_MODULE.....	17
3.10 TEST_BENCH.....	17
4.0 Design Summary .....	18
4.1 Resource utilization.....	18
4.2 Timing Summary .....	20
4.3 Technical Aspects.....	20
5.0 Simulation .....	21
6.0 Supplementary Tools .....	22
6.1 Compiler.....	22
6.2 Simulator .....	23
7.0 Down-sampling algorithm.....	24
7.1 Filtering .....	24
7.2 Down-sampling .....	27
8.0 Results and Verification .....	29

8.1 Transmitting and receiving down-sampled image.....	29
8.2 Comparison between images down-sampled by the computer and FPGA.....	31
9.0 Reference .....	32
10.0 Appendix .....	32
10.1 Verilog codes.....	32
10.1.1 PROGRAM_COUNTER .....	32
10.1.2 AC_ALU.....	33
10.1.3 Baudrate Generator .....	35
10.1.4 Transmitter .....	36
10.1.5 Receiver.....	37
10.1.6 UART.....	39
10.1.7 INSTRUCTION_DECODER .....	40
10.1.8 LOOP_REGISTER .....	43
10.1.9 MEM_REGISTERS .....	44
10.1.10 REG_BANK.....	45
10.1.11 PROCESSOR_TOP_MODULE.....	46
10.1.12 TEST_BENCH.....	50
10.2 Python codes.....	51
10.2.1 Compiler.....	51
10.2.2 Simulator .....	59
10.3 Assembly code for down-sampling.....	64
10.4 Schematics .....	68

## 1.0 Introduction

When a processor for a custom ISA is designed, we should have a tradeoff in hardware complexity, user/ programmer friendliness, performance, size and cost. Specially when your processor supposes to do a specific task, your instruction set and register pool should be able to perform that task efficiently using available resource/for a given cost.

For us our specific task was to down sample an image by a factor of two. Available resources were **Xilinx Spartan - 6 FPGA** with UART interface to communicate between Processor and Computer. We were able to achieve our task by optimized processor for looping operations, a memory which can be used to store two 256x256 images at one time and UART transceiver which can be used to send/ receive data from memory. Verilog HDL was used to implement modules and Xilinx ISE Design Suite as our programming environment.

DPUT Processor is an 8 – bit processor with 64kb (4096x16 bit) instruction memory and 1024kb (131072x8 bit) data memory. Even though this is an 8-bit processor (which has 8-bit dram) all the data registers inside the processor are 16-bit registers. This enables programmer to multiply 2 8-bit values without overflow. Excluding special purpose registers DPUT has 16 general purpose registers which reduces frequent memory access which improves the performance of the processor.

Since lot of looping operations are involved in image processing algorithms, we have introduced a new instruction with a special register and a flag which can reduce no of clock cycles w.r.t normal ISA. Also, instruction set with only 15 instructions have reduced the programming complexity. Since instructions are categorized into 5 instruction types with a meaningful way, instruction set is simple and easily understandable by the programmer.

We have also implemented a python compiler which reads a .txt file into a binary machine code which can be imported to instruction memory. In addition to compiler we also implemented a python-based simulator which reads an instruction txt file and displays the values of the registers when an instruction is getting executed. This can be used to debug your algorithm without repeatedly synthesizing your modules. Also, OUT port can be connected to any inside bus and it can also be used to debug your modules/algorithms.

## 2.0 Instruction Set Architecture of DPUT processor

The design objective of our ISA is to optimize the ISA for image processing applications and Serial Communication through UART interface. Since the image is a 2D-array of data and requires loops to go through pixels and execute operations, we have introduced an instruction called JUMPDEC which can be used to execute loop operations very fast.

DPUT ISA belongs to the RISC category of instruction sets. ISA contains only 15 simple instructions of equal length and all instructions requires only one clock cycle to execute. As in RISC, each instruction performs a specific task and there are separate instructions to access data from memory (Load/Store Architecture).

#### Key features of ISA

- No of instructions = 15
- No of instruction types = 5
- Avg. clock cycles per instruction (CPI) = 1
- Memory used per instruction = 2 Bytes

## 2.1 Instruction types

We have divided instructions into 5 main types based on their operation.

- **I-type** – When this type of instruction is executed processor is mainly in idle state (i.e. no internal operation is done.) UART module control instructions also fall into this category because in our processor UART module is operated as a separate module.
- **A-type** – These instructions are used to perform ALU operations
- **S-type** – These instructions perform shift register operations. Here AC register acts as a shift register
- **T-type** – These instructions are used to move data in one register to another.
- **M-type** – Deal with Memory. Contains two sub categories
  - **M1-type** – deal with data memory. LOAD/STORE instructions.
  - **M2-type** – deal with instruction memory. These instructions are used to perform branching operations

Instruction Type	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
type - I	OPCODE				x	x	x	x	x	x	x	x	x	x	x	x
type - A	OPCODE				R						CONST					
type - S	OPCODE				x	x	x	x	x	x	x	x	N			
type - T	OPCODE				x	S					x	D				
type - M1	OPCODE				M											
type - M2	OPCODE				INST											

*Instruction Types*

- **OPCODE (4 bits)** - Specifies the operation to be performed.
- **R, D, S (5 bits)** - Specifies Register address
- **CONST (7 bits)** - unsigned integer (0-127)
- **N (4 bits)** - unsigned integer (0-15) specifies amount of bit shift to be performed
- **M (12 bits)** - Data memory offset
- **INST (12 bits)** - Jump instruction address (0-4095)
- **X (1 bit)** - Unused bit

## 2.2 Registers & Flags

There are 19 registers (including user inaccessible registers) in DPUT Processor. Most of them are 16-bit registers.

Register	Name	Size (bits)	Address	Description
PC	PROGRAM COUNTER	12	-	Contains the location of the next instruction to be executed
IR	INSTRUCTION REGISTER	16	-	Holds the current instruction being executed.
MBR	MEMORY BASE REGISTER	16	00001	Contains Base ([15:1] bits) of the dram address
MDR	MEMORY DATA REGISTER	8	00010	Stores the data being transferred in and from dram
UARTTX	UART TX REGISTER	8	00011	Contains data to be sent through UART interface
UARTRX	UART RX REGISTER	8	00100	Contains data came through UART interface
LR	LOOP REGISTER	16	00110	Keep the current cycle of the loop
ZR	ZERO REGISTER	16	00000	Contains constant zero value
AC	ACCUMULATOR	16	00101	Stores results of ALU
R0 - R15	GP REGISTERS	16	10000 - 11111	General purpose registers - can be used to store values
Z	ZERO FLAG	1	-	Set to one if AC is zero
LRZ	LOOP OVERFLOW FLAG	1	-	Set to one if LR is zero
TXBUSY	TX BUSY FLAG	1	-	Indicate UART module is transmitting data
RXREADY	RX READY FLAG	1	-	Indicate UART module is being receiving data

*Registers and Flags*

## 2.3 Instruction Set

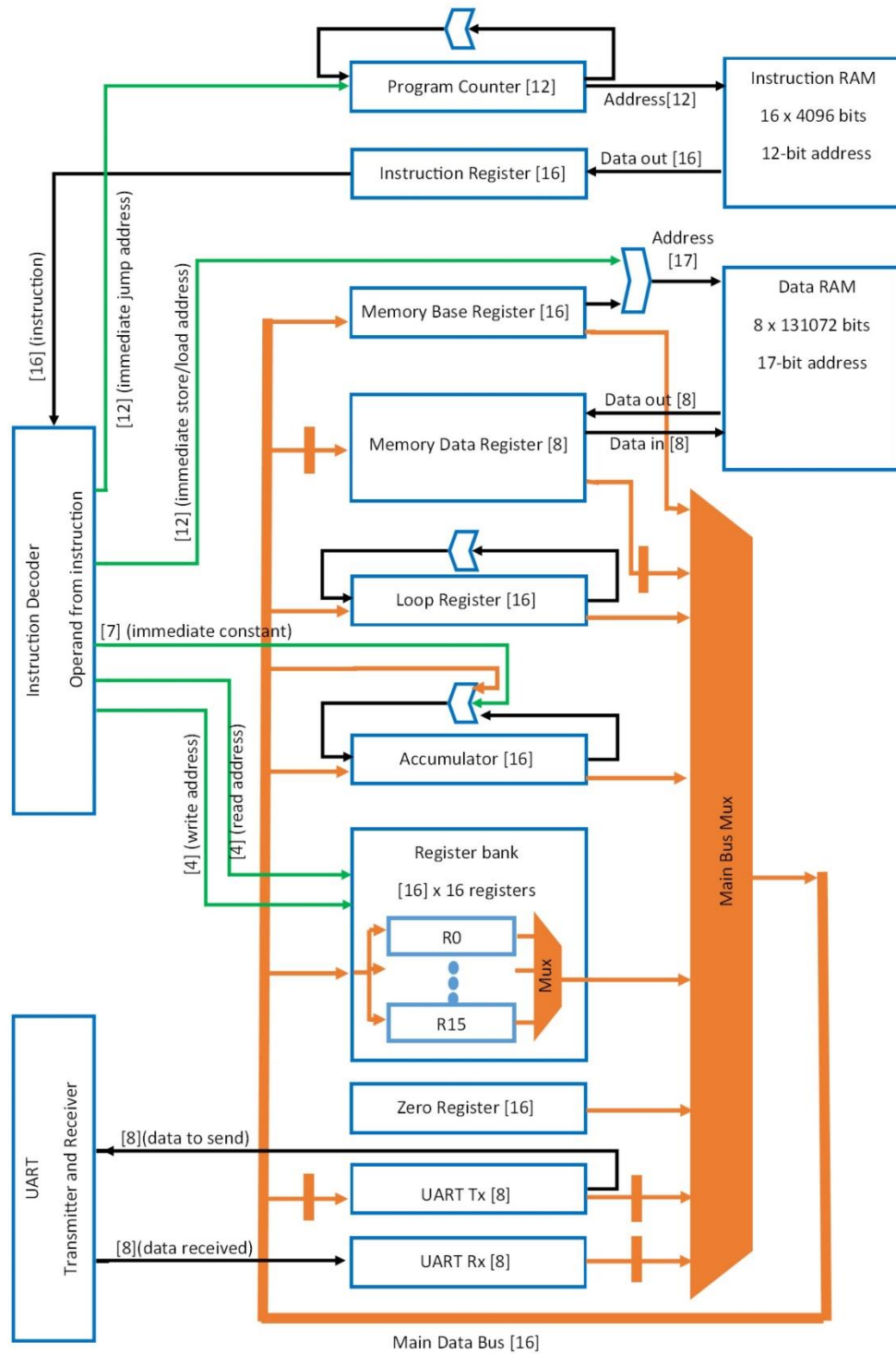
Typically, all instructions are executed within one clock cycle. But for synchronization purposes we advise to add NOP instruction after UARTTX/UARTRX is used. Since the block RAM modules

of the FPGA can be accessed within one clock cycle, LOAD/STORE instructions are also executed within one clock cycle.

Instruction	Type	Opcode	Description
NOP	type - I	0000	No Operation
UARTSEND	type - I	1101	Wait for UART output to complete (followed by a NOP)
UARTREAD	type - I	1110	Wait for UART input to complete (followed by a NOP)
ADD	type - A	0001	$AC \leftarrow AC + ([R] + [CONST])$
SUB	type - A	0010	$AC \leftarrow AC - ([R] + [CONST])$
MUL	type - A	0010	$AC \leftarrow AC * ([R] + [CONST])$
DIV	type - A	0100	$AC \leftarrow AC / ([R] + [CONST])$
SHR	type - S	0101	AC Shift right N bits
SHL	type - S	0110	AC Shift left N bits
LOAD	type - M1	0111	$[M + 2*MBR] \leftarrow MDR$
STORE	type - M1	1000	$MDR \leftarrow [M + 2*MBR]$
JUMP	type - M2	1001	Jump to [INST] in IRAM
JUMPZ	type - M2	1010	Jump to [INST] in IRAM IF Z FLAG =1
JUMPDEC	type - M2	1011	Decrement LR by ONE. Jump to [INST] in IRAM if LRZ = 0
MOVE	type - T	1100	$[D] \leftarrow [S]$

*Instruction Set of DPUT ISA*

## 2.4 Datapath

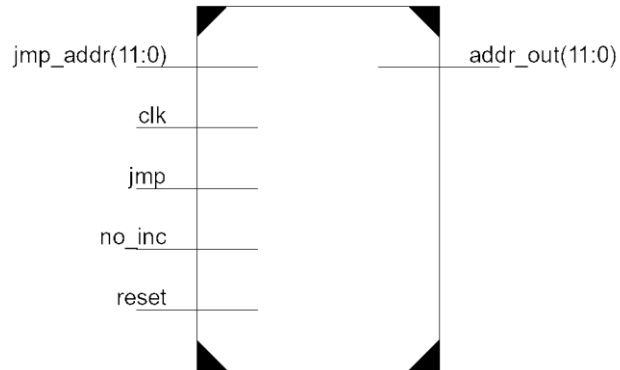


Datapath



## 3.0 Modules

### 3.1 PROGRAM\_COUNTER



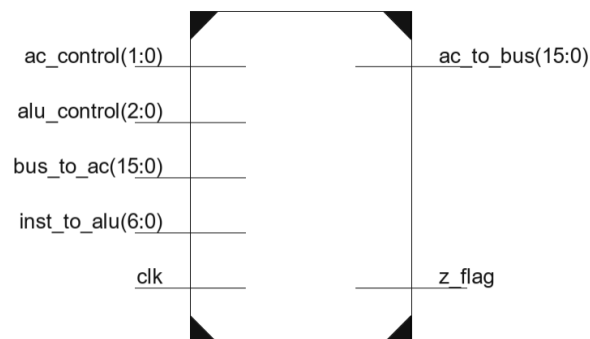
Program Counter is a 12-bit register which is sensitive to positive edge of the clock pulse.

Control Signals	Description
<code>Jump</code>	if ( <code>jump</code> ) : <code>addr_out</code> = <code>jump_addr</code>
<code>no_inc</code>	If ( <code>no_inc</code> ) : <code>addr_out</code> will not increment
<code>Reset</code>	If ( <code>reset</code> ) : <code>addr_out</code> set to zero

In default configuration **jump**, **no\_inc** & **reset** signals are set to zero. Therefore, at every positive edge of the clock pulse **addr\_out** will be incremented by 1. i.e it is pointed to next instruction to be fetched.

When type - M2 instructions use **jump** signal to perform branching. If you want to set the processor into idle state **no\_inc** signal should be set. **reset** pin is used to restart the program.

### 3.2 AC\_ALU



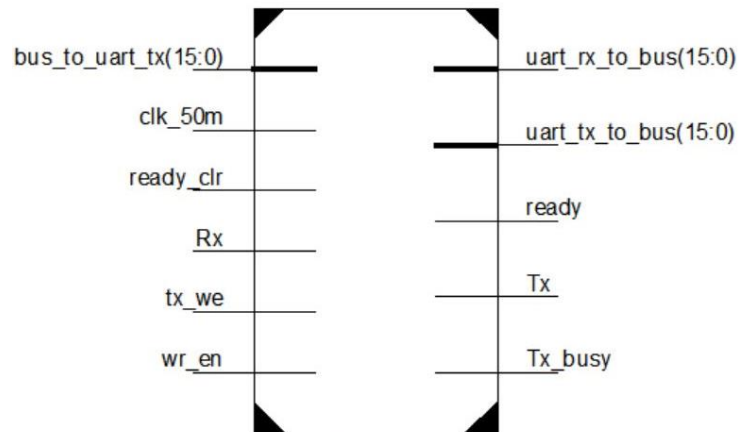
This module includes Accumulator (AC) and Arithmetic and Logic Unit (ALU). ALU can perform 4 basic arithmetic operations: Addition, Subtraction, Division and Multiplication. AC can be used as a shift register which can shift 0-15 positions left/right (But it also done through ALU). **Z\_flag** set whenever AC is zero.

Main bus is connected to module through **bus\_to\_ac** port and constants from instruction (**CONST/N**) is connected via **inst\_to\_alu** port. AC's value is always given out through **ac\_to\_bus** port.

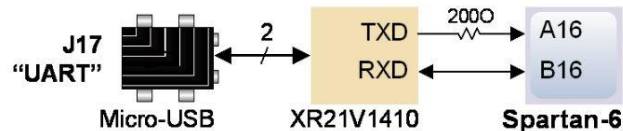
control Signal		Selection	Operation	Description
alu_control (2:0)		000	Addition	Data in Main bus and Constant from instruction is added to AC and stored back in AC
		001	Subtraction	Data in Main bus and Constant from instruction is added together and subtracted from AC and stored back in AC
		010	Multiplication	Data in Main bus and Constant from instruction is added together and multiplied by AC and stored back in AC
		011	Division	Data in AC is divided by the summation of Main bus value and Constant
		100	Shift Left	Data in AC is shifted to N bits left (N is given as Constant)
		101	Shift Right	Data in AC is shifted to N bits right ((N is given as Constant))
ac_control (1:0)	ac_control[0]	0	AC = ALU output	write ALU output to AC
		1	AC = Main bus input	write data in main bus to AC
	ac_control[1]	0	AC write disabled	don't write input data to AC register
		1	AC write enabled	write input data to AC register

### 3.3 UART

UART stands for *Universal Asynchronous Receiver and Transmitter*. This 'Asynchronous' term conveys that this protocol does not have a clock signal to synchronize output bits of the transmitter to the input bits of receiver. UART adds start bit and a stop bit at the beginning and at the end of each data packet to distinguish the data accurately. The UART module transmits data which is received by a data bus. In this implementation, the data which comes in a parallel form gets added the start and stop bits and sent to the Tx pin bit by bit serially. At the other end; the receiver collects the serially incoming bits and converts the data into parallel form and then removes the start and stop bit.



Control Signals	Description
Clk_50m	Clock signal for the UART module
ready_clr	Receiving process starting logic for Receiver
Rx	Receiver pin in FPGA
tx_we	Data is assigned from bus to data_in of uart_tx if high
wr_en Tx_busy	If <i>not</i> (Tx_busy) & wr_en: enabling the writing process of uart_tx
Tx	Transmitter pin in FPGA
ready	This flag is high before receiving process. When it has started the flag becomes low.
bus_to_uart_tx	Carries the data which is sent to uart_tx
uart_rx_to_bus	Serial to parallel converted data of uart_rx to is added to this bus
uart_tx_to_bus	



*Interface between PC USB port and Rx,Tx of FPGA*

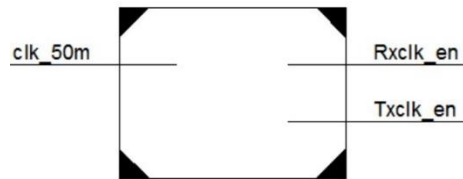
In the FPGA, there is a EXAR USB-UART bridge which can enable communication with the board using COM ports. EXAR part delivers the data to the FPGA using software flow control and 2 wires.

The UART module consists of 3 sub modules

- i. Baud rate generator
- ii. Transmitter
- iii. Receiver

### 3.3.1 Baudrate Generator

This module generates separate clocks for Transmitter and Receiver modules. The input clock signal for this is a 6.25MHz clock which is generated by dividing the main clock (100MHz) by 16.

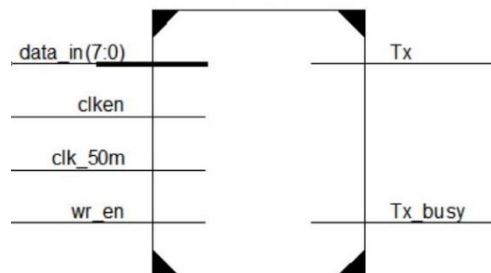


Control Signals	Description
Clk_50m	Clock signal for enabling the baudrate generator
Rxclk_en	Divided clock signal for the transmitter
Txclk_en	Divided clock signal for the receiver

### 3.3.2 Transmitter

Transmitter consists of 4 main stages in transmitting process.

- i. TX\_STATE\_IDLE - Transmitter is idle and Tx pin is set to high
- ii. TX\_STATE\_START - Transmission starts
- iii. TX\_STATE\_DATA - Parallel data is sampled and sent through the Tx serially
- iv. TX\_STATE\_STOP - Stops the transmission and sets the state back to idle



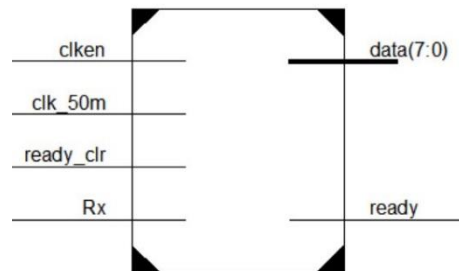
Control Signals	Description
data_in	Input data to the transmitter which is to be sent through Tx
clk_en	Divided clock for the Transmitter generated by baud rate generator
clk_50m	Input clock for enabling the transmitter
wr_en	A flag which should be high to start transmission process

Tx	Wire to transmitter pin
Tx_busy	Checks whether transmitter is in IDLE state

### 3.3.3 Receiver

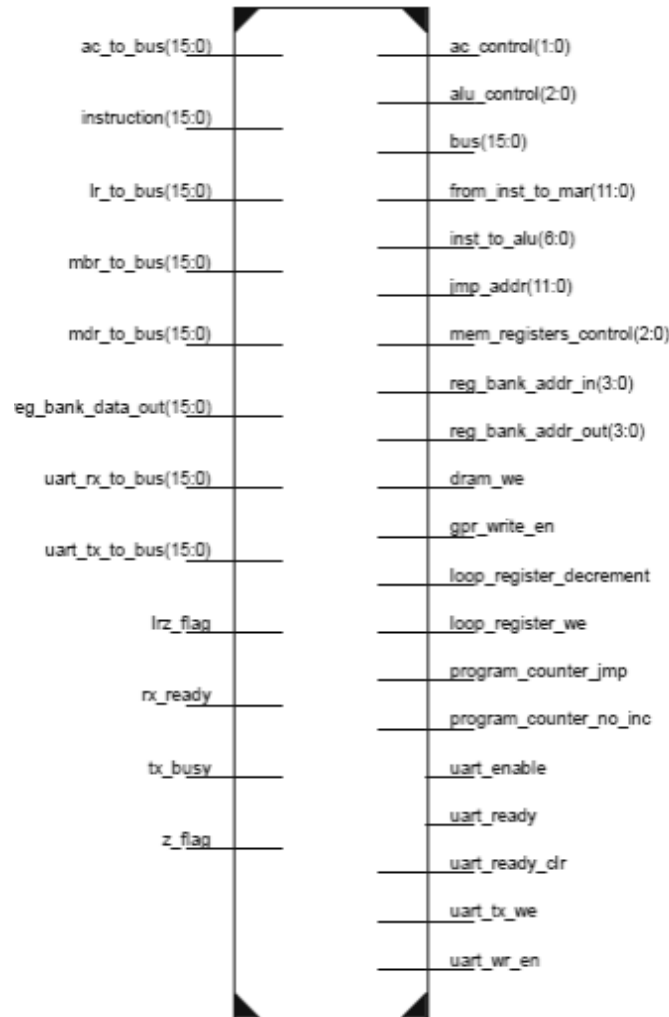
There are 3 states in the receiver

- i. RX\_STATE\_START - Starting the receiving process. The *ready* flag is set from high to low
- ii. RX\_STATE\_DATA - Collecting the incoming serial data and converting into parallel form
- iii. RX\_STATE\_STOP - Ends the receiving process. The *ready* flag is set to high again



Control Signals	Description
clk_en	Divide clock generated by baud rate generator to run the states in receiver
Clk_50m	The main clock to enable all processes of receiver
Ready_clr	Flag to enable the transmitter
Rx	Wire connected to receiver pin
data	Serially incoming data is converted into parallel form and assigned to this wire
ready	When ready clear is set to high this becomes low and starts transmitting. At the end again this becomes high

### 3.4 INSTRUCTION\_DECODER



Instruction Decoder receives the binary instruction from the Instruction Register and issues all the control signals required for executing the instruction. Other than control signals it also issues immediate constants extracted from the instruction to other modules. It also includes the Main Bus multiplexer which drives the Main Bus. In total, Instruction Decoder issues 19 control signals and 6 immediate constant outputs. Other than the instruction itself, outputs of AC, LR, MBR, MDR, UARTRX, UARTTX and Register Bank (registers which will be driving the bus) and the 4 flags (Z flag, LRZ flag, TXBUSY flag, RXREADY flag) are given as inputs to this module. Instruction Decoder is a combinational module. A summary of the outputs is given below.

### Immediate constants:

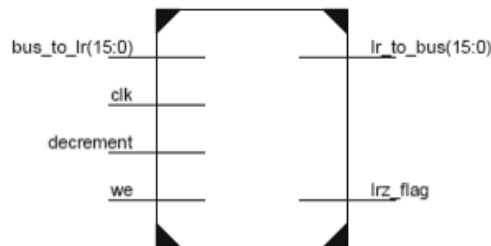
Reg_bank_addr_out [4 bits]	Reg_bank_addr_in [4 bits]	Inst_to_alu [7 bits]	Jmp_addr [12 bits]	From_inst_to_mar [12 bits]
Address of the general purpose register which is read into the main bus	Address of the general purpose register which is written from the main bus	Immediate constant from the instruction to the ALU	Jump address from the instruction to the Program Counter	Immediate LOAD/STORE address offset from the instruction

### Control signals:

INSTRUCTION	ACCUMULATOR		ALU			MEMORY REGISTERS			REGISTER BANK	PROGRAM COUNTER		LOOP REGISTER		UART					DATA RAM
	AC WRITE ENABLE	AC INPUT SELECT (0:BUS, 1:ALU)	ALU OPERATION SELECTION (0:ADD, 1:SUB, 2:MUL, 3:DIV, 4:SHR, 5:SHL)	MBR WRITE ENABLE	MDR WRITE ENABLE	MDR INPUT SELECT (0:BUS, 1:MEMORY)	REGISTER BANK WRITE ENABLE	PROGRAM COUNTER JMP	PROGRAM COUNTER NO_INCREMENT	LOOP REGISTER DECREMENT	LOOP REGISTER WRITE ENABLE	UART READY	UART READY_CLR	UART WR_EN	UART ENABLE	UARTTX WRITE ENABLE	DATA RAM WRITE ENABLE		
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
UARTSEND TXBUSY = 1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
UARTSEND TXBUSY = 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
UARTREAD RXREADY = 0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
UARTREAD RXREADY = 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
ADD	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SUB	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SHR	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SHL	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOAD	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
STORE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
JUMP	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
JMPZ Z=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMPZ Z=1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

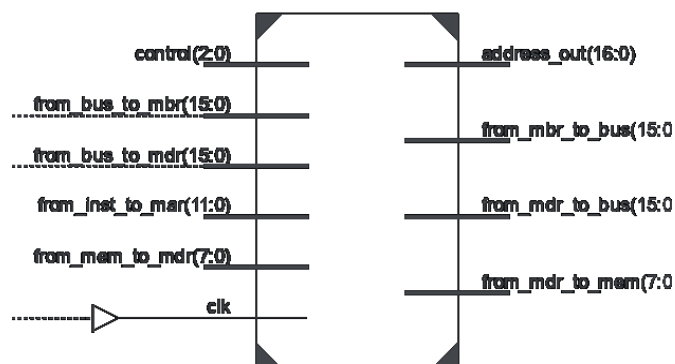
JMPDEC LRZ=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMPDEC LRZ=1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
MOVE	WRITE ENABLE SIGNAL OF THE CORRESPONDING REGISTER WILL BE 1 DEPENDING ON THE DESTINATION. ALL OTHER SIGNALS WILL REMAIN 0																			

### 3.5 LOOP\_REGISTER



Loop register is a normal 16-bit data register except it has its own subtractor and associated LRZ (Loop Register Zero) flag. This register is sensitive to the positive edge of the **clk**. The included subtractor will decrement the value in the register in each clock cycle until the **decrement** signal remains high. **bus\_to\_lr** is the input from the Main Bus and when **we** is high, the register value will be replaced by this value.

### 3.6 MEM\_REGISTERS

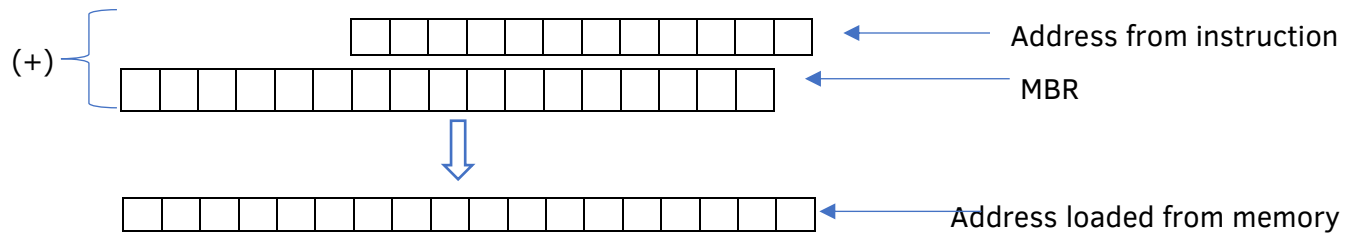


MBR and MDR are the two registers come under this. MBR consists of the base address which is used in construction of the address and the data in that address will be loaded to the MDR replacing any value which it contains. MDR is an 8-bit register. We cannot directly connect the 16-bit bus to MDR. Therefore, we add 8 zero bits before MDR when connecting to the bus.

The address to be stored or loaded isn't stored in a specific register. In the LOAD or STORE instruction, the address is given as a 12-bit address. It is added to the twice of the value in the MBR to get the address to be loaded from the data memory. That is, the value in the MBR is

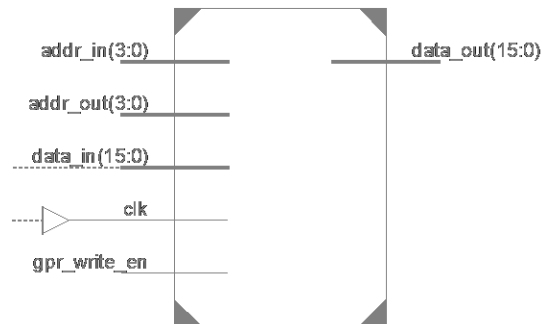


shifted 1 bit left and added to the given address in the instruction. Therefore, if we increment MBR by 1, it increases the address by two. When we run the down sampling loop, it is easy to skip pixels using this way.



Control signal	operation		description
Control[0]	Input selecting	0	Input is taken from the bus
		1	Input is taken from the memory
Control[1]	MDR write enable		if(control[1]): mdr<=mdr_input_wire at the next positive edge of clock
Control[2]	Mbr write enable		if(control[2]): mbr<=from_bus_to_mbr at the next positive edge of clock

### 3.7 REG\_BANK



There are 16 general purpose registers. Data of the register specified by the address given through 'addr\_out' will always available at the 'data\_out'.

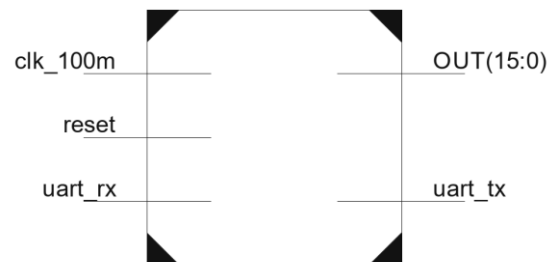
Control signal	description
gpr_write_en	When high, writes data from 'data_in' to the register given by the address 'addr_in', at the positive edge of clock

### 3.8 IRAM and DRAM

IRAM (instruction memory) and the DRAM (data memory) were constructed using the in-built Ip Core Generator tool of the Xilinx ISE Design Suite. These were constructed as block memory modules. Each module has its **clk (clock)**, **we (write enable)**, **addr (address)**, **din (data in)** and **dout (data out)** pins. The on-board 100MHz clock was directly given to the clock pins of these RAMs resulting a low-latency data in/out cycle.

**Dimensions:** IRAM – 16 bits x 4096 words, DRAM – 8 bits x 131072 words

### 3.9 PROCESSOR\_TOP\_MODULE



This module wraps all sub modules (processor, RAM and UART transceiver) together. **clk\_100m** pin is connected to internal built-in clock of FPGA and internally divided into 6.25MHz clock because timing analysis indicates that maximum clock speed we can use is 8.602MHz. It is done by incrementing **clkreg** a 4-bit register at positive edge of **clk\_100m** and by taking 4<sup>th</sup> bit as divided clock signal. **reset** pin is connected to reset button in order to restart the program. Data communication between PC and DPUT Processor is done through **uart\_tx** and **uart\_rx** pins of this module. These modules are connected to UART interface of the development board. **OUT** is a 16-bit port which is connected to 8 LEDs of the development board. it can be connected to any internal wire/register and can be used to observe states of connected wire/registers. By default, it is connected to wire **ac\_to\_bus**. [Appendix I](#) shows the internal wiring and sub modules of this module.

### 3.10 TEST\_BENCH


Test Bench was used for the simulation and debugging process of the processor in 'ISim', a Verilog simulation software. This module includes an instance of the **processor\_top\_module** and an instance of the **UART** module placed externally. This allowed to carry out a complete

test regarding the functionality of all the modules comprising the processor. Test Bench does not have any inputs or outputs.

## 4.0 Design Summary

### 4.1 Resource utilization

processor_top_module Project Status (05/03/2019 - 15:50:36)			
Project File:	processor_top_module.xise	Parser Errors:	No Errors
Module Name:	processor_top_module	Implementation State:	Placed and Routed
Target Device:	xc6slx45-2csg324	• Errors:	No Errors
Product Version:	ISE 14.7	• Warnings:	<a href="#">32 Warnings (1 new)</a>
Design Goal:	Balanced	• Routing Results:	<a href="#">All Signals Completely Routed</a>
Design Strategy:	<a href="#">Xilinx Default (unlocked)</a>	• Timing Constraints:	<a href="#">All Constraints Met</a>
Environment:	<a href="#">System Settings</a>	• Final Timing Score:	0 ( <a href="#">Timing Report</a> )

Device Utilization Summary					
Slice Logic Utilization	Used	Available	Utilization	Note(s )	
Number of Slice Registers	202	54,576	1%		
Number used as Flip Flops	202				
Number used as Latches	0				
Number used as Latch-thrus	0				
Number used as AND/OR logics	0				
Number of Slice LUTs	843	27,288	3%		
Number used as logic	827	27,288	3%		
Number using O6 output only	651				
Number using O5 output only	24				
Number using O5 and O6	152				
Number used as ROM	0				
Number used as Memory	12	6,408	1%		
Number used as Dual Port RAM	12				

Number using O6 output only	0			
Number using O5 output only	0			
Number using O5 and O6	12			
Number used as Single Port RAM	0			
Number used as Shift Register	0			
Number used exclusively as route-thrus	4			
Number with same-slice register load	0			
Number with same-slice carry load	4			
Number with other load	0			
Number of occupied Slices	295	6,822	4%	
Number of MUXCYs used	336	13,644	2%	
Number of LUT Flip Flop pairs used	881			
Number with an unused Flip Flop	693	881	78%	
Number with an unused LUT	38	881	4%	
Number of fully used LUT-FF pairs	150	881	17%	
Number of unique control sets	19			
Number of slice register sites lost to control set restrictions	46	54,576	1%	
Number of bonded <a href="#">IOBs</a>	20	218	9%	
Number of LOCed IOBs	11	20	55%	
Number of RAMB16BWERs	68	116	58%	
Number of RAMB8BWERs	0	232	0%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	2	16	12%	
Number used as BUFGs	2			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	8	0%	
Number of ILOGIC2/ISERDES2s	0	376	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	376	0%	
Number of OLOGIC2/OSERDES2s	0	376	0%	

Number of BSCANs	0	4	0%
Number of BUFHs	0	256	0%
Number of BUFPLLs	0	8	0%
Number of BUFPLL_MCBs	0	4	0%
Number of DSP48A1s	1	58	1%
Number of ICAPs	0	1	0%
Number of MCBs	0	2	0%
Number of PCILOGICSEs	0	2	0%
Number of PLL_ADVs	0	4	0%
Number of PMVs	0	1	0%
Number of STARTUPs	0	1	0%
Number of SUSPEND_SYNCs	0	1	0%
Average Fanout of Non-Clock Nets	5.35		

## 4.2 Timing Summary

Speed Grade: -2

Minimum period: 119.206ns (Maximum Frequency: 8.389MHz)

Minimum input arrival time before clock: 5.158ns

Maximum output required time after clock: 4.240ns

Maximum combinational path delay: No path found

## 4.3 Technical Aspects

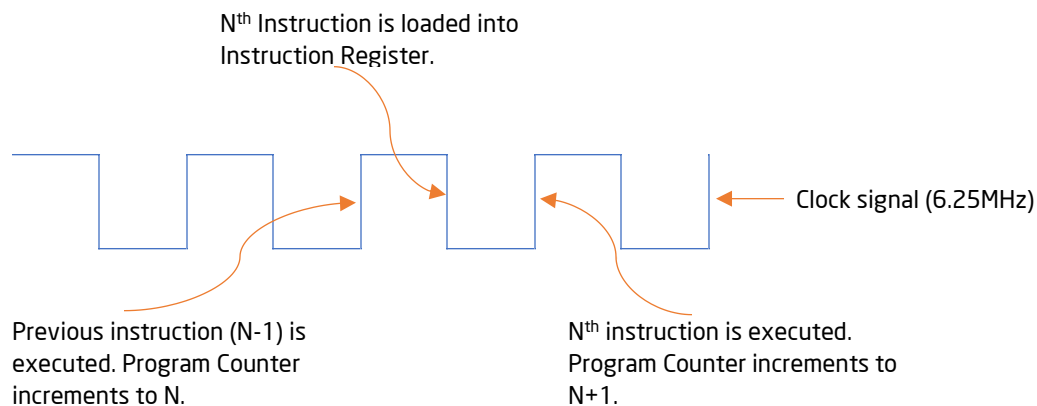
Aspect	Value	Units
Input clock speed	100	MHz
Processor clock speed	6.25	MHz
Average clock cycles per instruction	1	Clock cycles
Instruction memory size	16 x 4096	Bits
Data memory size	8 x 131072	Bits
Register width	16	Bits
I/O method	UART	-
UART speed	9600	Bauds
UART flow control	None	-
UART parity	None	-

UART stop bits	1	-
UART no of data bits	8	Bits
USB-UART bridge model	XR21V1410	-

Additional software required for communication:

- XR21V1410 USB-UART bridge driver – can be downloaded from [www.exar.com](http://www.exar.com).
- A serial communication terminal (Eg: Putty)

### Typical instruction cycle:



## 5.0 Simulation

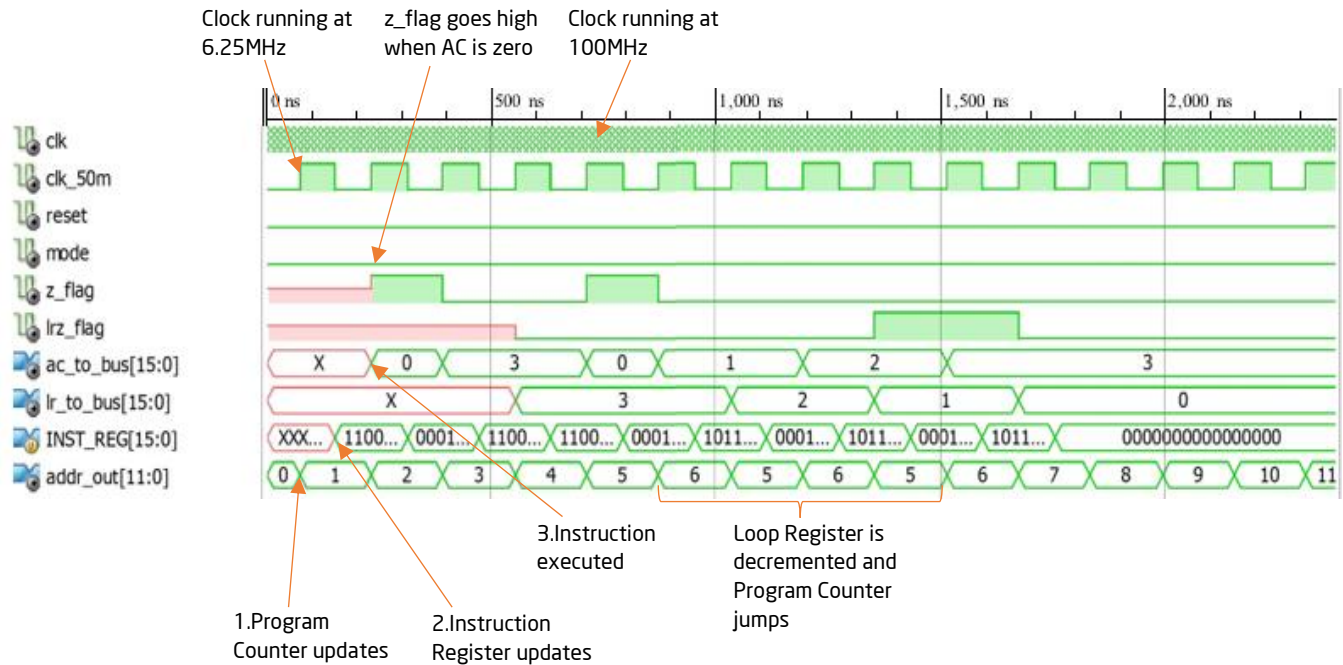
Simulations of the processor were carried out using the ISim simulation tool which is packaged up with Xilinx ISE 14.7.

Simulation result of executing the program given below is shown in the diagram:

```

nop
move zr ac
add zr 3
move ac lr
move zr ac
add zr 1
jmpdec 5

```



## 6.0 Supplementary Tools

### 6.1 Compiler

An assembly-to-machine code compiler (compileme.py) was written using Python (version 3) in order to convert human readable instructions into machine code. This compiler includes features such as label resolution, immediate addressing resolution for LOAD and STORE instructions, string resolution for UARTSEND and immediate constant resolution for ALU instructions. A file including instructions written in assembly is given as the input. Output is two files, one in assembly itself but with resolved labels, addresses, strings and constants, and the other includes instructions in machine code. An automatic commenting and error detection mechanism is also included. A compilation example is given below.

```

1  load 4096
2  :apple add r1 129
3  jumpdec :apple
4  uartsend :hi

```

Assembly code with incorrect instruction

```

1  load 4096
2  :apple add r1 129
3  jmpdec :apple
4  uartsend :hi

```

Assembly code with corrected instruction

```

1  move mbr r15 //0.load 4096
2  move ac r14
3  move zr ac
4  add zr 1
5  shl 12
6  move ac mbr
7  load 0
8  move r15 mbr
9  move r14 ac //end
10 add r1 2 //1.:apple add r1 129
11 add zr 127 //end
12 jmpdec 9
13 move ac r15 //3.uartsend :hi
14 move zr ac
15 add zr 104
16 move ac uarttx
17 uartsend
18 nop
19 move zr ac
20 add zr 105
21 move ac uarttx
22 uartsend
23 nop
24 move r15 ac //end

```

Output file 1: in assembly (with resolved labels, addresses, strings and constants)

```

(base) C:\Users\pasan\Desktop>python compileme.py in.txt
output files named as precompiled.txt and compiled.coe
0->
move mbr r15 //0.load 4096
move ac r14
move zr ac
add zr 1
shl 12
move ac mbr
load 0
move r15 mbr
move r14 ac //end

1->
add r1 2 //1.:apple add r1 129
add zr 127 //end

Error: 2 : jumpdec is not a valid instruction
precompilation interrupted

```

Compiler reporting about the incorrect instruction

```

memory_initialization_radix = 2;
memory_initialization_vector =
1100000001011111,
1100000101011110,
1100000000000101,
0001000000000001,
0110000000001100,
1100000101000001,
0111000000000000,
1100011111000001,
1100011110000101,
0001100010000010,
0001000001111111,
101100000001001,
1100000101011111,
1100000000000101,
0001000001101000,
1100000101000011,
1101000000000000,
0000000000000000,
1100000000000101,
0001000001101001,
1100000101000011,
1101000000000000,
0000000000000000,
1100011111000101,
0000000000000000;

```

Output file 2: in machine code

## 6.2 Simulator

A simulator (tryme.py) was written in Python (version 3) for the purpose of testing and verifying the algorithms. This was based on the compiler mentioned above. The processor architecture was implemented using a dictionary resembling the register stack. Memory was also implemented using a dictionary. Values of these dictionaries were written into an output file by the simulator after the execution of each instruction. This allowed to investigate the values of each register and memory location through each step of the algorithm. Following image shows a part of an output file generated.



```

***** simulation results - generated on Mon May 6 21:21:32 2019*****

simulation start

1. move mbr r15 //0.load 4096
{'zr': 0, 'mbr': 0, 'mdr': 0, 'uarttx': 0, 'uartrx': 0, 'ac': 0, 'lr': 0}
gpr:{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0}

2. move ac r14
{'zr': 0, 'mbr': 0, 'mdr': 0, 'uarttx': 0, 'uartrx': 0, 'ac': 0, 'lr': 0}
gpr:{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0}

3. move zr ac
{'zr': 0, 'mbr': 0, 'mdr': 0, 'uarttx': 0, 'uartrx': 0, 'ac': 0, 'lr': 0}
gpr:{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0}

4. add zr 1
{'zr': 0, 'mbr': 0, 'mdr': 0, 'uarttx': 0, 'uartrx': 0, 'ac': 1, 'lr': 0}
gpr:{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0}

```

## 7.0 Down-sampling algorithm

The down-sampling algorithm consists mainly two stages. If we directly down sample the original image, we might not be able to obtain an image equal to the original image because it will skip some pixels which contribute for the details in the image by a considerable amount. Therefore, we first filter the image using a suitable filter in order to smooth the image. After that we pick the relevant pixels to make the resulting image.

### 7.1 Filtering

In this we use a 3\*3 kernel with the weights of each position as given below.

1	2	1
2	4	2
1	2	1

Applying the kernel for each and every pixel is not needed as we are going to skip one pixel after the other each time. Therefore, we can skip the even numbered rows and columns in both directions when applying the kernel. Let the pixels of the image have values as shown below.

	0	1	2	3	4	.	.	.		
0	aa	ab	ac	ad	ae	.	.	.		
1	ba	bb	bc	bd	be					
2	ca	cb	cc	cd	ce					
3	da	db	dc	dd	de					

4	ea	eb	ec	ed	ef					
.	.									
.	.									
.	.									

If we apply the above-mentioned kernel to the (1,1) cell and average by dividing by the sum of the kernel we obtain the result as shown below.

$$R_{1,1} = \frac{1 * aa + 2 * ab + 1 * ac + 2 * ba + 4 * bb + 2 * bc + 1 * ca + 2 * cb + 1 * cc}{1 + 2 + 1 + 2 + 4 + 2 + 1 + 2 + 1}$$

$$R_{1,1} = \frac{aa + 2ab + ac + 2ba + 4bb + 2bc + ca + 2cb + cc}{16}$$

After obtaining the result we are going to replace the value of (1,1) cell with the result 'r'. We do this averaging only for the cells with both coordinates are odd numbers as we pick only those at the down sampling stage.

We store the original values of each pixel that came through UART in the data memory as a stack of pixel values.

0	aa
1	ab
2	ac
	...
256	ba
257	bb
	...
65535	

Applying of above calculation for the values in the stack is little bit difficult and the code may be longer. Therefore, we implement the same thing by applying two kernels of 1\*3 and 3\*1 in the rows and columns respectively.

1	2	1
---	---	---

1
2
1

By applying the first filter along each row for the odd numbered pixels and replacing the middle cell we obtain the grid as below.

$$r_{0,1} = \frac{aa + 2ab + ac}{4}$$

$$r_{0,3} = \frac{ac + 2ad + ae}{4}$$

...

	0	1	2	3	4	.	.	.		
0	aa	$r_{0,1}$	ac	$r_{0,3}$	ae	.	.	.		
1	ba	$r_{1,1}$	bc	$r_{1,3}$	be					
2	ca	$r_{2,1}$	cc	$r_{2,3}$	ce					
3	da	$r_{3,1}$	dc	$r_{3,3}$	de					
4	ea	$r_{4,1}$	ec	$r_{4,3}$	ef					
.	.									
.	.									
.	.									

After that we apply the vertical filter for the modified columns (odd numbered columns).

1
2
1

$$R_{1,1} = \frac{r_{0,1} + 2r_{1,1} + r_{2,1}}{4}$$

$$R_{1,1} = \frac{\left(\frac{aa + 2ab + ac}{4}\right) + 2\left(\frac{ba + 2bb + bc}{4}\right) + \left(\frac{ca + 2cb + cc}{4}\right)}{4}$$

$$R_{1,1} = \frac{aa + 2ab + ac + 2ba + 4bb + 2bc + ca + 2cb + cc}{16}$$

Therefore, we can see the result obtained by this method is same as the previously obtained result.

Since the image contain an even number of rows and columns, we are unable to do this calculation for the last column and last row. For that issue we add an extra row and a column at the end containing values as the same as of the last row and last column.

## 7.2 Down-sampling

After replacing the pixels with obtained results of the image will be as shown below.

	0	1	2	3	4	5	.	.	.		
0							.	.	.		
1		$R_{1,1}$		$R_{1,3}$							
2											
3		$R_{3,1}$		$R_{3,3}$							
4											
5											
.	.										.
.	.										.
.	.										.

In the data memory stack, the result will be as shown below.

0	
1	$R_{1,1}$
2	
3	$R_{1,3}$
4	
5	$R_{1,5}$
	...

253	
254	
255	
256	
257	
258	
...	
509	
510	
511	
512	
513	
514	
515	
516	
...	
65535	

Now we have to select these cells and store them separately in the correct order. The initial image stored in the address range from 0 to 65535. Therefore, we store the result from 65537 to 81920.

	0	1	2	3	4	5	.	.	.		
0							.	.	.		
1		$R_{1,1}$		$R_{1,3}$							
2											
3		$R_{3,1}$		$R_{3,3}$							
4											
5											
.	.										.
.	.										.
.	.										.
							.	.	.		



$R_{1,1}$	$R_{1,3}$	...		
$R_{3,1}$	$R_{3,3}$			
...				

## 8.0 Results and Verification

### 8.1 Transmitting and receiving down-sampled image

For the transmitting and receiving process at the PC end, *Pycharm* IDE was used. Reasons for using *Pycharm* was, image processing sections and serial communication process could be easily implemented with it.

The ***Pyserial Library*** is included for serial communication with COM ports and FPGA. Since the data which should be transmitted and received was integers between [0-255], ASCII values could not be used in the code (ASCII values are only up to 127 starting from 0). A built-in library in Python could address this issue. By the method ***pack*** included in ***struct class***, integers could be encoded (similar to Unsigned char in C) and sent and received through COM ports.

```
import numpy as np
import cv2 as cv
import serial
import struct

port = "COM6"
baud = 9600

img = cv.imread('girl.jpg',cv.IMREAD_GRAYSCALE)
reconstruct = np.zeros((128,128))
w = img.shape[1]
h = img.shape[0]
ser = serial.Serial(port, baud, timeout=None)
print('Port Opened.....')
print('Writing Image')

for i in range(h):
    for j in range(w):
        ser.write(struct.pack('B', int(img[j][i])))
        print(i,j)
        sync = ser.read()[0]
print('Reading Image')

for i in range(128):
    for j in range(128):
        reconstruct[j,i]=ser.read()[0]
        print(i,j)
        ser.write(struct.pack('B',int(reconstruct[j][i])))
print('Image reading completed')
```



*Original Image (256x256) sent to FPGA*



*Down sampled Image  
(128x128) by FPGA*

## 8.2 Comparison between images down-sampled by the computer and FPGA

```
import cv2 as cv
import numpy as np
import math

imageIn='girl.jpg'
imageOut='girl_out.jpg'

img = cv.imread(imageIn,cv.IMREAD_GRAYSCALE)
cv.imshow('Original',img.astype('uint8'))
image_size=128*128
img1=np.zeros((128,128),np.float)

#running horizontal kernel along the image
for y in range(0,256):
    for x in range(1,256,2):          #y:0-255, x0,2,...254
        if (x==255):
            img[y, x] = (img[y, x-1] + (img[y, x]*2) + img[y, x+1]) /4
        else:
            img[y, x] = (img[y, x-1] + (img[y, x]*2) + img[y, x+1])/4
#running vertical kernel along the image
for y in range(1,256,2):
    for x in range(1,256,2):          #y:0,2,4-254, x0,2,...254
        if (y==255):
            img[y, x] = (img[y-1, x] + (img[y, x]*2)+img[y, x+1]) /4
        else:
            img[y,x]=(img[y-1,x]+(img[y,x]*2)+img[y+1,x])/4

downsamplePC = img1
downsampleFPGA = cv.imread(imageOut,cv.IMREAD_GRAYSCALE)

#calculating the squared error
sqerror =0
for i in range(128):
    for j in range(128):
        dif=abs(int(downsamplePC[j,i])-int(downsampleFPGA[j,i]))
        sqerror=sqerror+dif**2

print('sde:',round(math.sqrt(sqerror)/(image_size),1))
cv.imshow('DownsampleFPGA',downsampleFPGA.astype('uint8'))
cv.imshow('DownsamplePC',downsamplePC.astype('uint8'))

cv.waitKey(0)
```

The algorithm which was implemented in the processor to down sample images was implemented in python to compare the difference between expected accuracy and actual accuracy.



The square root of sum of squared error was zero.

```
G:\Anaconda\envs\opencv\python.exe C:/Users/Thilina/Desktop/amoeba/erroranalysis.py  
sde: 0.0
```

## 9.0 Reference

- [1] Atlys FPGA Board Reference Manual -  
[https://reference.digilentinc.com/\\_media/atlys:atlys:atlys\\_rm.pdf?\\_ga=2.122795851.1975576103.1563257482-996813489.1548924697](https://reference.digilentinc.com/_media/atlys:atlys:atlys_rm.pdf?_ga=2.122795851.1975576103.1563257482-996813489.1548924697)
- [2] Atlys Spartan-6 FPGA Trainer Board schematics -  
[https://reference.digilentinc.com/\\_media/atlys:atlys:atlys\\_sch.pdf?\\_ga=2.122795851.1975576103.1563257482-996813489.1548924697](https://reference.digilentinc.com/_media/atlys:atlys:atlys_sch.pdf?_ga=2.122795851.1975576103.1563257482-996813489.1548924697)
- [3] Xilinx ISE 14.7 In-Depth Tutorial -  
[https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx13\\_2/ise\\_tutorial\\_ug695.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/ise_tutorial_ug695.pdf)
- [4] Designing Digital Computer Systems With Verilog by D. Lilja and S. Sapatnekar (2005)

## 10.0 Appendix

### 10.1 Verilog codes

#### 10.1.1 PROGRAM\_COUNTER

```
`timescale 1ns / 1ps  
////////////////////////////////////  
// Company:  
// Engineer:  
//  
// Create Date:    11:41:24 03/31/2019  
// Design Name:  
// Module Name:    program_counter  
// Project Name:  
// Target Devices:  
// Tool versions:  
// Description:  
//  
// Dependencies:  
//  
// Revision:  
// Revision 0.01 - File Created
```

```

// Additional Comments:
//
/////////////////////////////////////////////////////////////////
module program_counter(
    input clk,
    input reset,
    input no_inc,
    input [11:0] jmp_addr,
    input jmp, //jump enable control signal
    output reg [11:0] addr_out
);

initial addr_out<=12'b000000000000;

always @(posedge clk) begin
    if (~reset) begin
        if (jmp)
            addr_out<=jmp_addr;
        else
            addr_out<=addr_out+{11'b000000000000,~no_inc};
        end
    else
        addr_out<=12'b000000000000;
end
endmodule

```

### 10.1.2 AC\_ALU

```

`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:38:15 03/14/2019
// Design Name:
// Module Name:    ac_alu
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////
module ac_alu(
    input clk,
    input [1:0] ac_control,
    input [2:0] alu_control,
    input [15:0] bus_to_ac,
    input [6:0] inst_to_alu,

```

```

    output [15:0] ac_to_bus,
    output z_flag
);

// ac_control //
// 0-> ac input select 0-bus 1-alu
// 1-> ac write enable

// alu_control //
// 0->ADD
// 1->SUB
// 2->MUL
// 3->DIV
// 4->SHR
// 5->SHL

reg [15:0] ac;

wire [15:0] ac_input;
wire [15:0] alu_out;
wire [15:0] const_from_inst;

assign const_from_inst[6:0]=inst_to_alu;
assign const_from_inst[15:7]=0;
assign ac_input=ac_control[0]? alu_out:bus_to_ac;
assign ac_to_bus=ac;

assign z_flag=ac? 1'b0:1'b1;

assign alu_out=mux(bus_to_ac,ac,const_from_inst,alu_control);
function [15:0] mux(
    input [15:0]bus_to_ac,
    input [15:0] ac,
    input [15:0] const_from_inst,
    input [3:0] alu_control
);

    case(alu_control)
        3'b000: begin mux=ac+(bus_to_ac+const_from_inst); end //add
        3'b001: begin mux=ac-(bus_to_ac+const_from_inst); end //sub
        3'b010: begin mux=ac*(bus_to_ac+const_from_inst); end //mul
        3'b011: begin mux=ac/(bus_to_ac+const_from_inst); end //div
        3'b100: begin //shr
            case (const_from_inst[3:0])
                0:begin mux=ac; end
                1:begin mux=ac>>1; end
                2:begin mux=ac>>2; end
                3:begin mux=ac>>3; end
                4:begin mux=ac>>4; end
                5:begin mux=ac>>5; end
                6:begin mux=ac>>6; end
                7:begin mux=ac>>7; end
                8:begin mux=ac>>8; end
                9:begin mux=ac>>9; end
            endcase
        end
    endcase
endfunction

```

```

        10:begin mux=ac>>10; end
        11:begin mux=ac>>11; end
        12:begin mux=ac>>12; end
        13:begin mux=ac>>13; end
        14:begin mux=ac>>14; end
        default:begin mux=ac>>15; end
    endcase
end
3'b101: begin //shl
    case (const_from_inst[3:0])
        0:begin mux=ac; end
        1:begin mux=ac<<1; end
        2:begin mux=ac<<2; end
        3:begin mux=ac<<3; end
        4:begin mux=ac<<4; end
        5:begin mux=ac<<5; end
        6:begin mux=ac<<6; end
        7:begin mux=ac<<7; end
        8:begin mux=ac<<8; end
        9:begin mux=ac<<9; end
        10:begin mux=ac<<10; end
        11:begin mux=ac<<11; end
        12:begin mux=ac<<12; end
        13:begin mux=ac<<13; end
        14:begin mux=ac<<14; end
        default:begin mux=ac<<15; end
    endcase
end
    default: mux=ac;
endcase
endfunction

always @(posedge clk) begin
    if(ac_control[1]) ac<=ac_input;
end

endmodule

```

### 10.1.3 Baudrate Generator

```

//TestBench_clk = 50MHz
//BuadRate = 9600
module baudrate (input wire clk_50m,
                 output wire Rxclk_en,
                 output wire Txclk_en
                 );
    parameter RX_MAX = 6250000 / (9600 * 16);
    parameter TX_MAX = 6250000 / 9600;
    parameter RX_WIDTH = $clog2(RX_MAX);
    parameter TX_WIDTH = $clog2(TX_MAX);
    reg [RX_WIDTH - 1:0] rx_acc = 0;
    reg [TX_WIDTH - 1:0] tx_acc = 0;

    assign Rxclk_en = (rx_acc == 5'd0);

```

```

assign Txclk_en = (tx_acc == 9'd0);

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

always @(posedge clk_50m) begin
    if (tx_acc == TX_MAX[TX_WIDTH - 1:0])
        tx_acc <= 0;
    else
        tx_acc <= tx_acc + 9'b1; //+=000000001
    end
endmodule

```

### 10.1.4 Transmitter

```

module transmitter( input wire [7:0] data_in, //input data as an 8-bit register/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
    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

```

### 10.1.5 Receiver

```

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

initial begin
    ready = 1'b1; // initialize ready = 0
    data = 8'b0000_0000; // 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;

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 (ready_clr)
        ready <= 1'b0; // This resets ready to 0

    if (clken) begin
        case (state) // Let us consider the 3 states of the receiver
        RX_STATE_START: begin // We define conditions 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 collecting
            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 think 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;
                ready <= 1'b1;
                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

```

### 10.1.6 UART

```
module uart(input [15:0] bus_to_uart_tx, //input data
            input clk_50m,
            input Rx,
            input wr_en,
            input ready_clr,
            input tx_we,
            output Tx,
            output ready,
            output Tx_busy,
            output [15:0] uart_tx_to_bus,
            output [15:0] uart_rx_to_bus
            );

reg [7:0] data_in;
assign uart_tx_to_bus={8'b00000000,data_in};
always @(posedge clk_50m) begin
    if(tx_we) begin
        data_in<=bus_to_uart_tx[7:0];
    end
end

wire [7:0] data_out;
assign uart_rx_to_bus={8'b00000000,data_out};

wire wr_en_mod;
assign wr_en_mod=(~Tx_busy) & wr_en;

wire Txclk_en, Rxclk_en;
baudrate uart_baud( .clk_50m(clk_50m),
                   .Rxclk_en(Rxclk_en),
                   .Txclk_en(Txclk_en)
                   );
transmitter uart_Tx( .data_in(data_in),
                   .wr_en(wr_en_mod),
                   .clk_50m(clk_50m),
                   .clken(Txclk_en), //We assign Tx clock to enable clock
                   .Tx(Tx),
                   .Tx_busy(Tx_busy)
                   );
receiver uart_Rx( .Rx(Rx),
                 .ready(ready),
                 .ready_clr(ready_clr),
                 .clk_50m(clk_50m),
                 .clken(Rxclk_en), //We assign Tx clock to enable clock
                 .data(data_out)
                 );
endmodule
```



## 10.1.7 INSTRUCTION\_DECODER

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:36:40 03/31/2019
// Design Name:
// Module Name:    instruction_decoder
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module instruction_decoder(
    input [15:0] instruction,

    //main bus drivers
    input [15:0] mbr_to_bus,
    input [15:0] mdr_to_bus,
    input [15:0] uart_tx_to_bus,
    input [15:0] uart_rx_to_bus,
    input [15:0] ac_to_bus,
    input [15:0] lr_to_bus,
    input [15:0] reg_bank_data_out,

    //flags
    input z_flag,
    input lrz_flag,
    input tx_busy,
    input rx_ready,

    //instruction operand digestion
    output [15:0] bus,
    output [3:0] reg_bank_addr_out,
    output [6:0] inst_to_alu,
    output [11:0] jmp_addr,
    output [11:0] from_inst_to_mar,
    output [3:0] reg_bank_addr_in,

    //control signals
    output [1:0] ac_control,
    output [2:0] alu_control,
```

```

output [2:0] mem_registers_control,

output gpr_write_en,

output program_counter_jump,

output loop_register_decrement,
output loop_register_we,

output uart_ready,
output uart_ready_clr,
output uart_wr_en,
output uart_enable,
output uart_tx_we,

output dram_we,

//set outside the LUT
output program_counter_no_inc
);

//main bus mux
wire [4:0] bus_mux_select;
assign bus=bus_mux(
    bus_mux_select,
    mbr_to_bus,
    mdr_to_bus,
    uart_tx_to_bus,
    uart_rx_to_bus,
    ac_to_bus,
    lr_to_bus,
    reg_bank_data_out
);
function [15:0] bus_mux(
    input [4:0] bus_mux_select,
    input [15:0] mbr_to_bus,
    input [15:0] mdr_to_bus,
    input [15:0] uart_tx_to_bus,
    input [15:0] uart_rx_to_bus,
    input [15:0] ac_to_bus,
    input [15:0] lr_to_bus,
    input [15:0] reg_bank_data_out
);
case(bus_mux_select)
    5'b00000:begin bus_mux=16'b0000000000000000; end
    5'b00001:begin bus_mux=mbr_to_bus; end
    5'b00010:begin bus_mux=mdr_to_bus; end
    5'b00011:begin bus_mux=uart_tx_to_bus; end
    5'b00100:begin bus_mux=uart_rx_to_bus; end
    5'b00101:begin bus_mux=ac_to_bus; end
    5'b00110:begin bus_mux=lr_to_bus; end
    default:begin bus_mux=reg_bank_data_out; end

```

```

        endcase
    endfunction

//operand digestion
wire reg_addr_mux_select;
assign bus_mux_select=reg_addr_mux_select? instruction[10:6] : instruction[11:7];
assign reg_bank_addr_out=reg_addr_mux_select? instruction[9:6] : instruction[10:7];
assign inst_to_alu=instruction[6:0];
assign jmp_addr=instruction[11:0];
assign from_inst_to_mar=instruction[11:0];
assign reg_bank_addr_in=instruction[3:0];

//lookup table for control signals
wire [18:0] decoder_out;
wire [4:0] reg_addr;
assign reg_addr=instruction[4:0];
assign {
    ac_control[1:0],
    alu_control[2:0],

    mem_registers_control[2:0],

    gpr_write_en,

    program_counter_jmp,

    loop_register_decrement,
    loop_register_we,

    uart_ready,
    uart_ready_clr,
    uart_wr_en,
    uart_enable,
    uart_tx_we,

    reg_addr_mux_select,

    dram_we} = decoder_out;

assign decoder_out = (instruction[15:12]==4'b0001)?19'b11000000000000000000:
    (instruction[15:12]==4'b0010)?19'b11001000000000000000:
    (instruction[15:12]==4'b0011)?19'b11010000000000000000:
    (instruction[15:12]==4'b0100)?19'b11011000000000000000:
    (instruction[15:12]==4'b0101)?19'b11100000000000000000:
    (instruction[15:12]==4'b0110)?19'b11101000000000000000:
    (instruction[15:12]==4'b0111)?19'b00000001100000000000:
    (instruction[15:12]==4'b1000)?19'b00000000000000000001:
    (instruction[15:12]==4'b1001)?19'b00000000010000000000:

    (instruction[15:12]==4'b1010 &
z_flag==1'b0)?19'b00000000000000000000:
    (instruction[15:12]==4'b1010 &
z_flag==1'b1)?19'b00000000010000000000:

```

```

                (instruction[15:12]==4'b1011 &
lrz_flag==1'b0)?19'b0000000001100000000:
                (instruction[15:12]==4'b1011 &
lrz_flag==1'b1)?19'b000000000100000000:

                (instruction[15:12]==4'b1100 &
reg_addr==5'b00001)?19'b0000010000000000010:
                (instruction[15:12]==4'b1100 &
reg_addr==5'b00010)?19'b0000001000000000010:
                (instruction[15:12]==4'b1100 &
reg_addr==5'b00011)?19'b00000000000000000110:
                (instruction[15:12]==4'b1100 &
reg_addr==5'b00101)?19'b1000000000000000010:
                (instruction[15:12]==4'b1100 &
reg_addr==5'b00110)?19'b0000000000010000010:
                (instruction[15:12]==4'b1100 &
reg_addr[4]==1'b1)?19'b0000000010000000010:

                (instruction[15:12]==4'b1101)?19'b00000000000000010000:
                (instruction[15:12]==4'b1110)?19'b0000000000000100000:
                19'b0000000000000000000;

assign program_counter_no_inc=tx_busy | (~rx_ready);
endmodule

```

### 10.1.8 LOOP\_REGISTER

```

`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:20:18 03/31/2019
// Design Name:
// Module Name:    loop_register
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////
module loop_register(
    input clk,
    input [15:0] bus_to_lr,
    input decrement,
    input we,
    output [15:0] lr_to_bus,

```

```

        output lrz_flag
    );

    reg [15:0] lr;

    assign lrz_flag=(lr==16'b0000_0000_0000_0001)? 1'b1:1'b0;
    assign lr_to_bus=lr;

    always @(posedge clk) begin
        if(decrement) lr<=lr-16'b0000000000000001;
        if(we) lr<=bus_to_lr;
    end

endmodule

```

### 10.1.9 MEM\_REGISTERS

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:54:52 03/09/2019
// Design Name:
// Module Name:    mem_registers
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module mem_registers(
    input clk,
    input [2:0] control,
    // 0=> mdr input select 0-from bus, 1-from mem
    // 1=> mdr write enable
    // 2=> mbr write enable

    input [11:0] from_inst_to_mar,
    input [15:0] from_bus_to_mbr,
    input [15:0] from_bus_to_mdr,
    input [7:0] from_mem_to_mdr,

    output [16:0] address_out,
    output [15:0] from_mbr_to_bus,
    output [15:0] from_mdr_to_bus,
    output [7:0] from_mdr_to_mem
);

```

```

reg [15:0] mbr;
reg [7:0] mdr;

wire [16:0] mbr_alu_1;
wire [16:0] mbr_alu_2;
wire [7:0] mdr_input_wire;

assign from_mbr_to_bus=mbr;
assign mbr_alu_1[11:0]=from_inst_to_mar;
assign mbr_alu_1[16:12]=5'b00000;
assign mbr_alu_2[16:1]=mbr[15:0];
assign mbr_alu_2[0]=1'b0;
assign address_out=mbr_alu_1 + mbr_alu_2;

assign mdr_input_wire = control[0]? from_mem_to_mdr : from_bus_to_mdr[7:0];
assign from_mdr_to_bus = {8'b00000000,mdr[7:0]};
assign from_mdr_to_mem = mdr;

always @(posedge clk) begin
    if(control[1]) begin
        mdr<=mdr_input_wire;
    end
    if(control[2]) begin
        mbr<=from_bus_to_mbr;
    end
end

endmodule

```

### 10.1.10 REG\_BANK

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:05:33 03/31/2019
// Design Name:
// Module Name:    reg_bank
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module reg_bank(
    input [15:0] data_in,

```

```

    input clk,
    input [3:0] addr_in,
    input gpr_write_en,
    input [3:0] addr_out,
    output [15:0] data_out
);

//REG bank
reg [15:0] R [15:0];

//set data out
assign data_out=R[addr_out];

//assign data to a register
always @(posedge clk) begin
    if (gpr_write_en) begin
        R[addr_in]<=data_in;
    end
end

endmodule

```

### 10.1.11 PROCESSOR\_TOP\_MODULE

```

`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    03:36:55 04/05/2019
// Design Name:
// Module Name:    processor_top_module
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////
module processor_top_module(
    input clk_100m,
    input reset,
    input uart_rx,
    output uart_tx,
    output [15:0] OUT
);

wire clk;

```

```

wire [15:0] bus;

wire [1:0] ac_control;
wire [2:0] alu_control;
wire [6:0] inst_to_alu;
wire [15:0] ac_to_bus;
wire z_flag;
assign OUT=ac_to_bus; //connect to top module output
ac_alu AC_ALU(
    .clk(clk),
    .ac_control(ac_control),
    .alu_control(alu_control),
    .bus_to_ac(bus),
    .inst_to_alu(inst_to_alu),
    .ac_to_bus(ac_to_bus),
    .z_flag(z_flag)
);

wire loop_register_decrement;
wire loop_register_we;
wire [15:0] lr_to_bus;
wire lrz_flag;
loop_register LOOP_REGISTER(
    .clk(clk),
    .bus_to_lr(bus),
    .decrement(loop_register_decrement),
    .we(loop_register_we),
    .lr_to_bus(lr_to_bus),
    .lrz_flag(lrz_flag)
);

wire [2:0] mem_registers_control;
wire [11:0] from_inst_to_mar;
wire [7:0] from_mem_to_mdr; //from dram - data
wire [16:0] address_out; //to dram - address
wire [15:0] from_mbr_to_bus;
wire [15:0] from_mdr_to_bus;
wire [7:0] from_mdr_to_mem; //to dram - data
mem_registers MEM_REGISTERS(
    .clk(clk),
    .control(mem_registers_control),
    // 0=> mdr input select 0-from bus, 1-from mem
    // 1=> mdr write enable
    // 2=> mbr write enable

    .from_inst_to_mar(from_inst_to_mar),
    .from_bus_to_mbr(bus),
    .from_bus_to_mdr(bus),
    .from_mem_to_mdr(from_mem_to_mdr),

    .address_out(address_out),
    .from_mbr_to_bus(from_mbr_to_bus),
    .from_mdr_to_bus(from_mdr_to_bus),
    .from_mdr_to_mem(from_mdr_to_mem)
);

```



```

    );

    wire program_counter_no_inc;
    wire [11:0] jmp_addr;
    wire program_counter_jump;
    wire [11:0] addr_out; //to iram - address
    program_counter PROGRAM_COUNTER(
        .clk(clk),
        .reset(reset),
        .no_inc(program_counter_no_inc),
        .jmp_addr(jmp_addr),
        .jump(program_counter_jump), //jump enable control signal
        .addr_out(addr_out)
    );

    wire [3:0] reg_bank_addr_in;
    wire gpr_write_en;
    wire [3:0] reg_bank_addr_out;
    wire [15:0] reg_bank_data_out;
    reg_bank REG_BANK(
        .data_in(bus),
        .clk(clk),
        .addr_in(reg_bank_addr_in),
        .gpr_write_en,
        .addr_out(reg_bank_addr_out),
        .data_out(reg_bank_data_out)
    );

    wire dram_we;
    dram DRAM(
        .clka(clk_100m), // input clka
        .wea(dram_we), // input [0 : 0] wea
        .addra(address_out), // input [16 : 0] addra
        .dina(from_mdr_to_mem), // input [7 : 0] dina
        .douta(from_mem_to_mdr) // output [7 : 0] douta
    );

    wire [15:0] iram_dout;
    iram IRAM (
        .clka(clk_100m), // input clka
        .wea(1'b0), // input [0 : 0] wea
        .addra(addr_out), // input [11 : 0] addra
        .dina(16'b0000000000000000), // input [15 : 0] dina
        .douta(iram_dout) // output [15 : 0] douta
    );

    wire tx_busy;
    wire rx_ready;
    wire uart_ready; //control signal never used
    wire uart_ready_clr;
    wire uart_wr_en;
    wire uart_enable;
    wire uart_tx_we;
    wire [15:0] uart_tx_to_bus;

```

```

wire [15:0] uart_rx_to_bus;
uart UART(
    .bus_to_uart_tx(bus), //input data
    .clk_50m(clk),
    .Rx(uart_rx),
    .wr_en(uart_wr_en),
    .ready_clr(uart_ready_clr),
    .tx_we(uart_tx_we),
    .Tx(uart_tx),
    .ready(rx_ready),
    .Tx_busy(tx_busy),
    .uart_tx_to_bus(uart_tx_to_bus),
    .uart_rx_to_bus(uart_rx_to_bus)
);

wire [15:0] instruction;
instruction_decoder INSTRUCTION_DECODER(
    .instruction(instruction),

    //main bus drivers
    .mbr_to_bus(from_mbr_to_bus),
    .mdr_to_bus(from_mdr_to_bus),
    .uart_tx_to_bus(uart_tx_to_bus),
    .uart_rx_to_bus(uart_rx_to_bus),
    .ac_to_bus(ac_to_bus),
    .lr_to_bus(lr_to_bus),
    .reg_bank_data_out(reg_bank_data_out),

    //flags
    .z_flag(z_flag),
    .lrz_flag(lrz_flag),
    .tx_busy(tx_busy),
    .rx_ready(rx_ready),

    //instruction operand digestion
    .bus(bus),
    .reg_bank_addr_out(reg_bank_addr_out),
    .inst_to_alu(inst_to_alu),
    .jmp_addr(jmp_addr),
    .from_inst_to_mar(from_inst_to_mar),
    .reg_bank_addr_in(reg_bank_addr_in),

    //control signals
    .ac_control(ac_control),
    .alu_control(alu_control),

    .mem_registers_control(mem_registers_control),

    .gpr_write_en(gpr_write_en),

    .program_counter_jmp(program_counter_jmp),

```

```

        .loop_register_decrement(loop_register_decrement),
        .loop_register_we(loop_register_we),

        .uart_ready(uart_ready),
        .uart_ready_clr(uart_ready_clr),
        .uart_wr_en(uart_wr_en),
        .uart_enable(uart_enable),
        .uart_tx_we(uart_tx_we),

        .dram_we(dram_we),

        .program_counter_no_inc(program_counter_no_inc)
    );

reg [15:0] INST_REG;
initial INST_REG<=16'b0000000000000000;
assign instruction=INST_REG;
always @(negedge clk) begin
    INST_REG<=iram_dout;
end

reg [3:0] clkreg;
initial clkreg=0;
always @(posedge clk_100m) clkreg=clkreg+1;
assign clk=clkreg[3]; //running @ 100MHz/16=6.25MHz
//assign clk=clk_100m;
endmodule

```

### 10.1.12 TEST\_BENCH

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:24:27 04/08/2019
// Design Name:
// Module Name:    test_bench
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module test_bench(
);

```

```

wire clk, reset, rx, tx;
wire [15:0] out;

reg [3:0] clkreg;
initial clkreg=0;
always @(posedge clk) clkreg<=clkreg+1;
wire clk_50m;
assign clk_50m=clkreg[3];

wire [15:0] out_uart_tx_to_bus;
wire [15:0] out_uart_rx_to_bus;
reg [15:0] out_bus_to_uart_tx;
reg send,wr_en;
wire tx_busy;

always @(send,posedge tx_busy) wr_en<=~tx_busy;

uart UART(
    .bus_to_uart_tx(out_bus_to_uart_tx), //input data
    .clk_50m(clk_50m), //running @ 6.25MHz
    .Rx(rx),
    .wr_en(wr_en),
    .ready_clr(),
    .tx_we(1'b1),
    .Tx(tx),
    .ready(),
    .Tx_busy(tx_busy),
    .uart_tx_to_bus(out_uart_tx_to_bus),
    .uart_rx_to_bus(out_uart_rx_to_bus)
);

processor_top_module PROCESSOR(
    .clk_100m(clk),
    .reset(reset),
    .uart_rx(tx),
    .uart_tx(rx),
    .OUT(out)
);
endmodule

```

## 10.2 Python codes

### 10.2.1 Compiler

```

import re
import sys

ops = {
    'nop': 0b0000,
    'add': 0b0001,

```

```

        'sub': 0b0010,
        'mul': 0b0011,
        'div': 0b0100,
        'shr': 0b0101,
        'shl': 0b0110,
        'load': 0b0111,
        'store': 0b1000,
        'jump': 0b1001,
        'jmpz': 0b1010,
        'jmpdec': 0b1011,
        'move': 0b1100,
        'uartsend': 0b1101,
        'uartread': 0b1110
    }
    reg = {
        'zr': 0b00000,
        'mbr': 0b00001,
        'mdr': 0b00010,
        'uarttx': 0b00011,
        'uartrx': 0b00100,
        'ac': 0b00101,
        'lr': 0b00110
    }

```

```

def getValToAc(x):
    num = '{:016b}'.format(x)
    acommand = ''
    zeros = 0
    ones = False
    for i in range(0,16):
        c = num[i]
        if c == '1':
            if zeros and ones: acommand = acommand + 'shl ' + str(zeros) + '\n'
            acommand = acommand + 'add zr 1\n'
            zeros = 0
            if i == 15: ones = False
            else: ones = True
        zeros += 1
    if zeros and ones: acommand = acommand + 'shl ' + str(zeros) + '\n'
    return acommand

```

```

argCount = len(sys.argv)
infileName = ''

```

```

oufileName = 'precompiled.txt'
compfileName = 'compiled.coe'
if argCount < 2:
    print("no input file\nusage:\nchill.py <input file name(required)> <output file
names(optional)>")
    exit()
elif argCount == 2:
    infileName = sys.argv[1]
    print('output files named as precompiled.txt and compiled.coe')
elif argCount == 3:
    infileName = sys.argv[1]
    oufileName = sys.argv[2]
    print('output file named as compiled.coe')
else:
    infileName = sys.argv[1]
    oufileName = sys.argv[2]
    compfileName = sys.argv[2]

# precompiling

infile = open(infileName, 'r')
oufile = open('temp.txt', 'w')
lineNo = 0
command = ''
jumpLines={}
writeLineCount=0
alllines=infile.readlines()

try:
    while(lineNo<len(alllines)):
        line=alllines[lineNo]
        line = line.lower().strip()
        x = (re.split("\s+", line))
        command=''

        if len(x[0])<1: lineNo += 1
        elif x[0][0]==':':
            jumpLines[x[0][1:]] = writeLineCount
            x=x[1:]
        else:
            # warn if x[0] is not in ops
            if not (x[0] in ops.keys()):
                print('Error:', lineNo, ':', x[0], ' is not a valid instruction')
                break

        if (x[0] == 'nop'):
            command = x[0] + '\n'

```

```

elif (x[0] in ['add', 'sub', 'mul', 'div']):
    if x[1][0] == 'r':
        # warn if index is not numeric
        if not x[1][1:].isdecimal():
            print('Error:', lineNo, ':', x[1], 'is not a valid register')
            break

        regNum = int(x[1][1:])

        # warn if index>number of registers
        if regNum < 0 or regNum > 15:
            print('Error:', lineNo, ':', x[1], 'exceeds valid register index range')
            break

    else:
        # warn if not a valid registers
        if not x[1] in reg.keys():
            print('Error:', lineNo, ':', x[1], 'is not a valid register')
            break

    if int(x[2])<128:
        command = x[0] + ' ' + x[1] + ' ' + x[2] + '\n'
    else:
        num = int(x[2])
        command = x[0] + ' ' + x[1] + ' ' + str(num % 127) + ' //' + str(lineNo) + '.'
+ line + '\n'
        for kk in range(0, int(num / 127)):
            command = command + x[0] + ' zr ' + str(127) + '\n'
        command = command[:-1] + ' //end\n'

elif (x[0] in ['shr', 'shl']):
    # warn for shift range
    if int(x[1]) < 0 or int(x[1]) > 15:
        print('Error:', lineNo, ':', x[1], 'exceeds valid shift range')
        break
    command = x[0] + ' ' + x[1] + '\n'

elif (x[0] in ['load', 'store']):
    num = int(x[1])
    #warn for address range
    if num < 0 or num > 131071:
        print('Error:', lineNo, ':', x[1], 'exceeds valid address range')
        break

```

```

if num>4095:
    command = 'move mbr r15 //' + str(lineNo) + '.' + line + '\nmove ac r14\nmove
zr ac\n'

    command = command + getValToAc(int(num / 2))
    command = command + 'move ac mbr\n'
    command = command + x[0] + ' ' + str(num%2) + '\n'
    command = command + 'move r15 mbr\nmove r14 ac //end\n'
else:
    command = x[0] + ' ' + x[1] + '\n' #TODO: correct getValToAc()

elif (x[0] in ['jump', 'jmpz', 'jmpdec']):
    # warn for address range
    if not x[1][0]==':':
        if not x[1].isdecimal():
            print('Error:', lineNo, ':', x[1], 'is not a valid immediate accessible
address')

            break
        num = int(x[1])
        if num < 0 or num > 4095:
            print('Error:', lineNo, ':', x[1], 'exceeds valid immediate accessible
address range')

            break

    command = x[0] + ' ' + x[1] + '\n'

elif (x[0] == 'move'):
    breakAll = False
    for i in range(0, 2):
        if (x[i + 1][0] == 'r'):
            # warn if index is not numeric
            if not x[i + 1][1:].isdecimal():
                print('Error:', lineNo, ':', x[i + 1], 'is not a valid register')
                breakAll = True
                break

        regNum = int(x[i + 1][1:])

        # warn if index>number of registers
        if regNum < 0 or regNum > 15:
            print('Error:', lineNo, ':', x[i + 1], 'exceeds valid register index
range')

            breakAll = True
            break

    else:
        # warn if not a valid registers

```



```

        if not x[i + 1] in reg.keys():
            print('Error:', lineNo, ':', x[i + 1], 'is not a valid register')
            breakAll = True
            break

    if breakAll:
        break

    command = x[0] + ' ' + x[1] + ' ' + x[2] + '\n'

elif (x[0] in ['uartsend', 'uartread']):
    if x[0]=='uartsend' and len(x)>1 and x[1][0]==':':
        string = x[1]
        command = 'move ac r15 //' + str(lineNo) + '.' + line + '\n'
        for char in string[1:]:
            asciival = ord(char)
            command = command + 'move zr ac\nadd zr ' + str(asciival) + '\nmove ac
uarttx\nuartsend\nnop\n'
        command = command + 'move r15 ac //end\n'
    else:
        command = x[0] + '\n'

else:
    # warn about errors
    print('Error:', lineNo, ': \'', alllines[lineNo].strip(), '\': invalid command.')
    break

#print(thisLineNo, '.', x, REGS, GPR, MEM)
#dd=str(thisLineNo)+'.\t'+line+'\n\t'+str(REGS)+' \n\tgpr:'+str(GPR)+'
\n\tmem:'+str(MEM)+'\n\n'

dd=command
writelineCount=writeLineCount+len(dd.strip().split('\n'))
oufile.write(dd)
print(str(lineNo) + '->\n' + dd)
lineNo = lineNo + 1

except IndexError:
    print('Error:', lineNo, ': \'', alllines[lineNo].strip(), '\': invalid command.')
    exit()

```

```

# precompilation ends

if lineNo==len(alllines):
    outfile.close()
    infile.close()

# replaceing labels
infile = open('temp.txt', 'r')
outfile = open(outfileName, 'w')
for line in infile:
    line = line.lower().strip()
    y = (re.split("\s+", line))
    if (y[0] in ['jump', 'jmpz', 'jmpdec']):
        if y[1][0]==':':
            if not y[1][1:] in jumpLines.keys():
                print('label', y[1][1:], 'not found.')
                print('precompilation interrupted')
                exit()
            command = y[0] + ' ' + str(jumpLines[y[1][1:]]) + '\n'
        else:
            command = y[0] + ' ' + y[1] + '\n'
    else:
        command = line + '\n'

    outfile.write(command)
outfile.close()
infile.close()

print('precompilation end\n')
outfile.close()
infile.close()

# compiling

infile = open(outfileName, 'r')
outfile = open(compfileName, 'w')
outfile.write("memory_initialization_radix = 2;\nmemory_initialization_vector =\n")
lineNo = 0
for line in infile:
    lineNo = lineNo + 1
    line = line.lower()
    x = (re.split("\s+", line))

    command = '{:04b}'.format(ops[x[0]])
    if (x[0] == 'nop'):
        command = command + '000000000000'

```

```

elif (x[0] in ['add', 'sub', 'mul', 'div']):
    if (x[1][0] == 'r'):
        regNum = int(x[1][1:])
        command = command + '1' + '{:04b}'.format(regNum)
    else:
        command = command + '0' + '{:04b}'.format(reg[x[1]])
        command = command + ' ' + '{:07b}'.format(int(x[2]))

elif (x[0] in ['shr', 'shl']):
    num = '{:012b}'.format(int(x[1]))
    command = command + ' ' + num

elif (x[0] in ['load', 'store']):
    command = command + ' ' + '{:012b}'.format(int(x[1]))

elif (x[0] in ['jump', 'jmpz', 'jmpdec']):
    command = command + ' ' + '{:012b}'.format(int(x[1]))

elif (x[0] == 'move'):
    for i in range(0, 2):
        if (x[i + 1][0] == 'r'):
            regNum = int(x[i + 1][1:])
            command = command + '01' + '{:04b}'.format(regNum)
        else:
            command = command + '00' + '{:04b}'.format(reg[x[i + 1]])

elif (x[0] in ['uartsend', 'uartread']):
    command = command + '000000000000'

else:
    # warn about errors
    print('unknown compile error!')
    break

print(lineNo, '.', x, command)
outfile.write(command + ",\n")

if not infile.readline():
    print('compilation end')
    outfile.write("000000000000000;\n")

else:
    print('compilation interrupted')

# compilation ends

else:
    print('precompilation interrupted\n')

```

## 10.2.2 Simulator

```
import re
import sys
import time

ops = {
    'nop': 0b0000,
    'add': 0b0001,
    'sub': 0b0010,
    'mul': 0b0011,
    'div': 0b0100,
    'shr': 0b0101,
    'shl': 0b0110,
    'load': 0b0111,
    'store': 0b1000,
    'jump': 0b1001,
    'jmpz': 0b1010,
    'jmpdec': 0b1011,
    'move': 0b1100,
    'uartsend': 0b1101,
    'uartread': 0b1110
}

reg = {
    'zr': 0b000000,
    'mbr': 0b000001,
    'mdr': 0b000010,
    'uarttx': 0b000011,
    'uartrx': 0b000100,
    'ac': 0b000101,
    'lr': 0b000110
}

argCount = len(sys.argv)
infileName = ''
oufileName = 'results.txt'
if argCount < 2:
    print("no input file\nusage:\ncompileme.py <input file name(required)> <output file\nname(optional)>")
    exit()
elif argCount == 2:
    infileName = sys.argv[1]
    print('output file named as results.txt')
else:
    infileName = sys.argv[1]
    oufileName = sys.argv[2]

localtime = time.asctime( time.localtime(time.time()) )
infile = open(infileName, 'r')
outfile = open(oufileName, 'w')
outfile.write("***** simulation results - generated on "+localtime+"*****\n\nsimulation\nstart\n\n")
lineNo = 0

#registers
```

```

REGS = {
    'zr': 0,
    'mbr': 0,
    'mdr': 0,
    'uarttx': 0,
    'uartrx': 0,
    'ac': 0,
    'lr': 0
}
GPR = {0:0,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:0,11:0,12:0,13:0,14:0,15:0}
#GPR = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
MEM = {}

alllines=infile.readlines()
round=0
while(lineNo<len(alllines)):
    line=alllines[lineNo]
    thisLineNo=lineNo+1
    line = line.lower().strip()
    x = (re.split("\s+", line))

    # warn if x[0] is not in ops
    if not (x[0] in ops.keys()):
        print('Error at line', lineNo+1, ':', x[0], ' is not a valid instruction')
        break

    command = '{:04b}'.format(ops[x[0]])
    if (x[0] == 'nop'):
        command = command + '000000000000'
        lineNo = lineNo + 1

    elif (x[0] in ['add', 'sub', 'mul', 'div']):
        operand=0
        toac=0
        if (x[1][0] == 'r'):
            # warn if index is not numeric
            if not x[1][1:].isdecimal():
                print('Error at line', lineNo+1, ':', x[1], 'is not a valid register')
                break

            regNum = int(x[1][1:])

            # warn if index>number of registers
            if regNum < 0 or regNum > 15:
                print('Error at line', lineNo+1, ':', x[1], 'exceeds valid register index
range')

                break

            command = command + '1' + '{:04b}'.format(regNum)
            operand=GPR[regNum]
        else:

```

```

        # warn if not a valid registers
        if not x[1] in reg.keys():
            print('Error at line', lineNo+1, ':', x[1], 'is not a valid register')
            break
        command = command + '0' + '{:04b}'.format(reg[x[1]])
        operand=REGS[x[1]]
        command = command + ' ' + '{:07b}'.format(int(x[2]))
        if x[0]=='add':
            toac=REGS['ac']+(operand+int(x[2]))
        elif x[0]=='sub':
            toac = REGS['ac'] - (operand + int(x[2]))
        elif x[0]=='mul':
            toac = REGS['ac'] * (operand + int(x[2]))
        elif x[0]=='div':
            toac = REGS['ac'] / (operand + int(x[2]))
        else:
            toac = REGS['ac']
        REGS['ac']=int(toac)
        lineNo = lineNo + 1

elif (x[0] in ['shr', 'shl']):
    # warn for shift range
    num = '{:012b}'.format(int(x[1]))
    if int(x[1]) < 0 or int(x[1]) > 15:
        print('Error at line', lineNo+1, ':', x[1], 'exceeds valid shift range')
        break
    command = command + ' ' + num
    if x[0]=='shr': REGS['ac']=REGS['ac']>>int(x[1])
    else: REGS['ac']=REGS['ac']<<int(x[1])
    lineNo = lineNo + 1

elif (x[0] in ['load', 'store']):
    # TODO: warn for address range
    num = int(x[1])
    if num < 0 or num > 4095:
        print('Error at line', lineNo+1, ':', x[1], 'exceeds valid immediate accessible
address range')
        break
    command = command + ' ' + '{:012b}'.format(int(x[1]))
    mar = num + REGS['mbr'] * 2
    if x[0]=='store':
        MEM[mar]=REGS['mdr']
    #     outfile.write('storing mem data at address ' + str(mar) + '.\n')
    #     print('storing mem data at address', mar, '.')
    else:
        if mar in MEM.keys():
            REGS['mdr']=MEM[mar]
        #     outfile.write('reading mem data at address ' + str(mar) + '.\n')
        #     print('reading mem data at address', mar, '.')

```

```

        else:
            REGS['mdr']=7
#            outfile.write('no mem data at address '+str(mar)+' . so 0 assumed.\n')
#            print('no mem data at address',mar,'. so 0 assumed.')

lineNo = lineNo + 1

elif (x[0] in ['jump', 'jmpz', 'jmpdec']):
    # warn for address range
    num = int(x[1])
    if num < 0 or num > 4095:
        print('Error at line', lineNo+1, ':', x[1], 'exceeds valid immediate accessible
address range')
        break
    command = command + ' ' + '{:012b}'.format(int(x[1]))
    if x[0]=='jump': lineNo=num; #outfile.write('jump to '+str(num)+'.\n')
    elif x[0]=='jmpz':
        if REGS['ac']==0: lineNo=num; #outfile.write('jmpz to '+str(num)+'.\n')
        else:lineNo=lineNo+1
    else:
        oldLineNo=lineNo
        if REGS['lr']>0:
            REGS['lr']-=1
            lineNo=num
            #outfile.write('jmpdec to ' + str(num) + '.\n')
        if REGS['lr']==0: lineNo=oldLineNo+1

elif (x[0] == 'move'):
    breakAll = False
    sourceVal=0
    isGpr=[False, False]
    for i in range(0, 2):
        if (x[i + 1][0] == 'r'):
            # warn if index is not numeric
            if not x[i + 1][1:].isdecimal():
                print('Error at line', lineNo+1, ':', x[i + 1], 'is not a valid register')
                breakAll = True
                break

        regNum = int(x[i + 1][1:])

        # warn if index>number of registers
        if regNum < 0 or regNum > 15:
            print('Error at line', lineNo+1, ':', x[i + 1], 'exceeds valid register
index range')

            breakAll = True

```

```

        break

        command = command + '01' + '{:04b}'.format(regNum)
        isGpr[i]=True
    else:
        # warn if not a valid registers
        if not x[i + 1] in reg.keys():
            print('Error at line', lineNo+1, ':', x[i + 1], 'is not a valid register')
            breakAll = True
            break

        command = command + '00' + '{:04b}'.format(reg[x[i + 1]])

if breakAll:
    break

if isGpr[0]: sourceVal = GPR[int(x[1][1:])]
else: sourceVal=REGS[x[1]]
if isGpr[1]: GPR[int(x[2][1:])] = sourceVal
else: REGS[x[2]] = sourceVal
lineNo=lineNo+1

elif (x[0] in ['uartsend', 'uartread']):
    command = command + '000000000000'
    lineNo=lineNo+1
    if x[0]=='uartsend':
        print('uartsend:', REGS['uarttx'])
        outfile.write('uart output sent. value:'+str(REGS['uarttx'])+'.\n')
    else:
        done=False
        while(not done):
            uinput=input('enter uart read number within range:0 to 255\n')
            #uinput=str(70)
            if uinput.isdecimal() and int(uinput)>-1 and int(uinput)<256:
                uinput=int(uinput)
                REGS['uartrx']=uinput
                outfile.write('uart input given. value:'+str(uinput)+'.\n')
                done=True
            else:
                print('invalid uart input!\n')

else:
    # warn about errors
    print('unknown compile error!')
    break

```



```

        #print(thisLineNo, '.', x, REGS, GPR, MEM)
        #dd=str(thisLineNo)+'.\t'+line+'\n\t'+str(REGS)+' \n\tgpr:'+str(GPR)+'
\n\tmem:'+str(MEM)+'\n\n'
        dd=str(thisLineNo)+'.\t'+line+'\n\t'+str(REGS)+' \n\tgpr:'+str(GPR)+'\n\n'
        if round%(5000)==0:outfile.write(dd);print(dd)
        if lineNo>(len(alllines)-1):pp=str(thisLineNo)+'.\t'+line+'\n\t'+str(REGS)+'
\n\tgpr:'+str(GPR)+' \n\tmem:'+str(MEM)+'\n\n'; outfile.write(pp)
        round=round+1

if lineNo==len(alllines):
    print('simulation end')
    outfile.write('simulation end.')
else:
    print('simulation interrupted')
    outfile.write('simulation interrupted.')

```

### 10.3 Assembly code for down-sampling

```

nop
move zr mbr
move zr mdr
move zr ac
add zr 127
add zr 1
move ac r1
mul r1 0
mul zr 2
move ac lr
move zr uarttx
:thilalpha uartread
nop
move uartrx mdr
store 0
uartsend
nop
uartread
nop
move uartrx mdr
store 1
uartsend
nop
move mbr ac
add zr 1
move ac mbr
jmpdec :thilalpha
move zr ac //processing image
add zr 64
shl 2
move ac lr
:chanbeta move lr r1
move zr ac

```

```

add zr 127
move ac lr
:chanalpha move zr ac
add zr 64
shl 2
sub r1 0
shl 8
move ac r2
move zr ac
add zr 127
sub lr 0
shl 1
add r2 0
shr 1
move ac mbr
move zr ac
load 1
add mdr 0
shl 1
load 0
add mdr 0
load 2
add mdr 0
shr 2
move ac mdr
store 1
jmpdec :chanalpha
move zr ac
add r2 127
add zr 127
shr 1
move ac mbr
move zr ac
load 1
add mdr 0
shl 1
add mdr 0
load 0
add mdr 0
shr 2
move ac mdr
store 1
move r1 lr
jmpdec :chanbeta
move zr ac
add zr 64
shl 1
move ac lr
:chandelta move lr r1
move zr ac
add zr 127
move ac lr
:changamma move zr ac
add zr 2
add zr 127
sub r1 0
shl 1
sub zr 1
move ac r2
move zr ac
add zr 127
sub lr 0
shl 9
add r2 0

```

```

shr 1
move ac mbr
move zr ac
load 257
add mdr 0
shl 1
load 1
add mdr 0
load 513
add mdr 0
shr 2
move ac mdr
store 257
jmpdec :changamma
move zr ac
add zr 127
shl 9
add r2 0
shr 1
move ac mbr
move zr ac
load 257
add mdr 0
shl 1
add mdr 0
load 1
add mdr 0
shr 2
move ac mdr
store 257
move r1 lr
jmpdec :chandelta
move zr ac
add zr 64
shl 1
move ac lr
:chantheta move lr r1
move zr ac
add zr 64
move ac lr
:chaneeta move zr ac //load address
add zr 64
sub lr 0
shl 1
move ac r2
move zr ac
add zr 127
add zr 2
sub r1 0
shl 8
sub zr 127
sub zr 1
add r2 0
move ac mbr
load 1
move mdr r3
load 3
move zr ac //store address
add zr 127
add zr 1
sub r1 0
shl 6
move ac r2
move zr ac

```

```

add zr 1
shl 15
add zr 64
sub lr 0
add r2 0
move ac mbr
store 2
move r3 mdr
store 1
jmpdec :chaneeta
move r1 lr
jmpdec :chantheta
move zr ac //sending image
add zr 1
shl 15
move ac mbr
add zr 64
move ac r1
mul r1 0
mul zr 2
move ac lr
:thilibeta load 1
move mdr uarttx
uartsend
nop
uartread
nop
load 2
move mdr uarttx
uartsend
nop
uartread
nop
move mbr ac
add zr 1
move ac mbr
jmpdec :thilibeta

```

10.4 Schematics

