University of Moratuwa

Faculty of Engineering

EN3030: Circuit and System Design

FPGA based Multi-Core Processor Design

Name	Index number
R.T. Hithanadura	170227R
K.H.M.T.M.S. Samarakoon	170538V
D.T.D. Vithanage	170656F
M.N.C. Waas	170661P

Supervisor
Dr. Jayathu Samarawickrama

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

2021-07-07

Abstract

Custom processors are popular in electronic industry today ever since its introduction. A custom processor mean a processor designed for a specific task. The reasons for this popularity are high efficiency, low risk way to implement a task, low power consumption, economic feasibility, ability to be enclosed in small places etc.

In this report we discuss about our custom processor design including both simulation and hardware implementation using Field Programmable Logic Array (FPGA). The designed multi-core processor is optimized for matrix multiplication.

We have design not only a SIMD multi-core processor but also a full system which can generate random matrices on a text file in the laptop and send the matrices to the FPGA board through UART communication, then multiply the matrices using the custom made processor then send the answer matrix to the laptop again. The received answer matrix is then saved in a text file. The whole process is *fully automated* and *100% accurate*.

This report includes methods and theories used for above task. The Instruction Set Architecture (ISA), Verilog, SystemVerilog codes, Assembly code and the python code can be found in Appendix area.

A detail demonstration and explanation of the functionality implementation of the FPGA design can be found below. This will show a working processor design for matrix multiplication in a many core architecture and a fully functional with a working condition design in a FPGA Board. **Click here to open the demo**.

Contents

Acronyms **vi**

List of Figures **vii**

1 Introduction **1**

1.1 Single-Core processor 1

1.2 Multi-Core processor 1

1.3 Problem statement 2

1.4 Proposed solution 2

1.5 Processor design flow 3

1.6 Processing order 4

2 Algorithm **5**

2.1 Matrix calculation parallelism among cores 5

2.2 Memory access handling 6

2.2.1 Possible data memory access methods by multiple processor cores . 6

3 Instruction Set Architecture **9**

3.1 Instruction Set 9

3.2 Assembly Code 10

3.3 Data path (Hardware Architecture) 10

3.4 Micro-instructions 13

3.5 State Diagram 15

4 Modules **16**

4.1 Top Module 16

4.2 Data Memory 17
4.3 Instruction Memory 17
4.4 Multi-Core Processor 18
4.5 Processor 19
4.6 Control Unit 20
4.7 ALU 22
4.8 Register 22
4.9 Inc_Register 23
4.10 Z Register 23
4.11 Multiplexer (Data-Bus) 24

5 Simulation, Testing & Debugging 26

6 Results Analyzing and Verification 29

6.1 Comparison of Speeds of Multicore Processor Designs 36

7 Design Summary 40

7.1 Flow Summary 40
7.2 Analysis & Synthesis Summary 40
7.3 Filter Summary 41
7.4 Assembler Summary 41
7.5 Timing Analyzer Summary 42
7.6 EDA Netlist Writer Summary 42

8 References 43

9 Appendix 44

9.1 Multi-core processor related SystemVerilog modules 44
9.1.1 Common details for modules - details.sv 44
9.1.2 Top module - toFpga.sv 45

9.1.3 Memory . . . 48
9.1.4 Multi-core processor . . . 49
9.1.5 Single-core processor . . . 53
9.1.6 Control Unit . . . 56
9.1.7 Arithmetic and Logic Unit (ALU) . . . 67
9.1.8 Incrementable Registers . . . 69
9.1.9 Register . . . 70
9.1.10 Multiplexer (System Bus) . . . 71
9.1.11 Z register . . . 72
9.2 UART communication related SystemVerilog modules . . . 73
9.2.1 UART communication interface for the memory . . . 73
9.2.2 Data word size handler between memory and UART system . . . 78
9.2.3 UART system . . . 81
9.2.4 UART baud rate generator . . . 83
9.2.5 UART transmitter . . . 84
9.2.6 UART receiver . . . 86
9.3 Count and display the number of clock cycles for matrix multiplication process . . . 88
9.3.1 Clock cycle counter - timeCounter.sv . . . 88
9.3.2 Display on 8 seven segments on the FPGA board . . . 89
9.4 Testbenches for top level module . . . 92
9.5 Python code for implementation and verification . . . 96
9.5.1 Python code . . . 96
9.5.2 Python functions . . . 96
9.6 Text files used during synthesis and Verification . . . 101
9.6.1 Assembly code . . . 101
9.6.2 Example Input random matrix file . . . 102

9.6.3 Example answer matrix given by the multi-core processor in FPGA board . . . 103
9.6.4 Example answer matrix given by the python calculation on PC . . . 103
9.7 Multi-core processor related Verilog modules . . . 103
9.7.1 Top module - toFpga.v . . . 103
9.7.2 Memory . . . 106
9.7.3 Multi-core processor . . . 106
9.7.4 Single-core processor . . . 109
9.7.5 Control Unit . . . 112
9.7.6 Arithmetic and Logic Unit (ALU) . . . 124
9.7.7 Incrementable Registers . . . 125
9.7.8 Register . . . 126
9.7.9 Multiplexer (System Bus) . . . 128
9.7.10 Z register . . . 129
9.8 UART communication related Verilog modules . . . 130
9.8.1 UART communication interface for the memory . . . 130
9.8.2 Data word size handler between memory and UART system . . . 132
9.8.3 UART system . . . 134
9.8.4 UART baud rate generator . . . 135
9.8.5 UART transmitter . . . 135
9.8.6 UART receiver . . . 137
9.9 Count and display the number of clock cycles for matrix multiplication process . . . 138
9.9.1 Clock cycle counter - timeCounter.v . . . 138
9.9.2 Display on 8 seven segments on the FPGA board . . . 139
9.10 Testbenches for top level module . . . 142

Acronyms

ALU Arithmetic and Logic Unit.

CPU Central Processing Unit.

FPGA Field Programmable Logic Array.

HDL Hardware Descriptive Language.

I/O input/output.

ip-core interlectual property-core.

ISA Instruction Set Architecture.

PC Personal Computer.

RAM Random Access Memory.

SIMD Single Instruction - Multiple Data.

UART Universal Asynchronous Receiver-Transmitter.

List of Figures

1.1 Block diagram of a SIMD Multi-core Processor 2

1.2 Processor Design 3

1.3 Process Flow 4

2.1 Single core multiplication algorithm 5

2.2 Dual-core multiplication algorithm 6

2.3 Three-core multiplication algorithm 6

3.1 Naming convention of matrices and their dimensions 10

3.2 Data path of the CPU 12

3.3 State diagram of the CPU 15

4.1 System Hierarchy 16

4.2 Block diagram of Top Module 17

4.3 Block diagram of Data Memory 17

4.4 Block diagram of Instruction Memory 18

4.5 Block diagram of Muli-core processor 19

4.6 Block diagram of Processor 20

4.7 Block diagram of Control Unit 21

4.8 Block diagram of ALU 22

4.9 Block diagram of Register 23

4.10 Block diagram of Inc_Register 23

4.11 Block diagram of Z Register 24

4.12 Block diagram of Multiplexer 25

5.1 ALU Simulation 26

5.2 Register Simulation 26

5.3 Inc Register Simulation 27

5.4 Control Unit Simulation 27

5.5 UART Receiver Simulation 27

5.6 Signal Tap Logic Analyser Usage 27

5.7 In System Memory Content Editor Usage 28

6.1 Simulation of memory access of the processor 29

6.2 Comparison of Two 5x5 Matrices Multiplication Results of a Single-Core Processor & Python Script 29

6.3 Comparison of 8x9 Matrix & 9x10 Matrix Multiplication Results of a Single-Core Processor & Python Script 30

6.4 Comparison of 5x6 Matrix & 6x7 Matrix Multiplication Results of a Dual-Core Processor & Python Script 30

6.5 Comparison of 7x12 Matrix & 12x9 Matrix Multiplication Results of a Single-Core Processor (received using UART) & Python Script 31

6.6 Comparison of 10x7 Matrix & 7x9 Matrix Multiplication Results of a Dual-Core Processor (received using UART) & Python Script 32

6.7 Comparison of 10x8 Matrix & 8x12 Matrix Multiplication Results of a 3-Core Processor (received using UART) & Python Script 33

6.8 Comparison of 5x7 Matrix & 7x4 Matrix Multiplication Results of a 4-Core Processor (received using UART) & Python Script 33

6.9 Single-Core Processor 34

6.10 Dual-Core Processor 34

6.11 Three-Core Processor 35

6.12 Four-Core Processor 35

6.13 Speed Comparison of Processors with Different Number of Cores 37

7.1 Flow Summary 40

7.2 Analysis & Synthesis Summary 40

7.3 Filter Summary 41

7.4 Assembler Summary 41

7.5 Timing Analyzer Summary 42

7.6 EDA Netlist Writer Summary 42

List of Tables

3.1 Instruction Set 9

3.2 Memory Locations of Some Necessary Data Values 11

3.3 A sample long table. 14

4.1 Operations of ALU 22

4.2 Flag value to select register 25

6.1 Speed Comparison of Processors with Different Number of Cores 36

6.2 Speed Comparison for Single-Core Processor 38

6.3 Speed Comparison for Dual-Core Processor 38

6.4 Speed Comparison for Three-Core Processor 39

6.5 Speed Comparison for Four-Core Processor 39

1 Introduction

The objective of this project is to design a Multi-Core Processor which is optimized for matrix multiplication.

We have design a full system which can generate random matrices on a text file in the laptop and send the matrices to the FPGA board, then multiply the matrices using the custom made multi-core processor then send the answer matrix to the laptop again. The received answer matrix is then saved in a text file. The whole process is fully automated and 100% accurate.

We have implemented the same design twice, first using **Verilog HDL** and second using **SystemVerilog HDL**. Both designs have the same capabilities. For the simulation we have designed test benches for each module and they are designed using both languages. The part of the system which runs in the laptop is designed using **Python 3.8**.

Quartus Prime 18.1 Lite edition along with ALTERA DE2-115 Education and Development FPGA board with Cyclone IV softcore processor is used for the implementations.

1.1 Single-Core processor

The Processing unit is the electronic circuitry within a computer that executes instructions that set up a computer program. The processor performs basic arithmetic, logic, controlling and input/output (I/O) operations determined by the instructions of the program. This consists of a control unit to control every single task within the processor, an Arithmetic and Logic Unit (ALU) to do arithmetic and logical operations and registers to keep data temporally. The processor needs external components such as data memory, instruction memory for data storage and I/O circuitry for user interface.

1.2 Multi-Core processor

A multi-core processor is a processor which contains more than one processor core which have similar capabilities and characteristics as mentioned in the previous topic. Each core has its own control unit, registers and ALU. Therefore, they can work independently. Time consumption for the process can be drastically reduced by dividing the process among each core in an intelligent manner. Single Instruction - Multiple Data (SIMD) is a one method which can be used to divide a large process into similar smaller sub-processes and use each individual core for one sub-process. In this document we have shown how to do a matrix multiplication process using SIMD method, with the help of a multi-core processor.

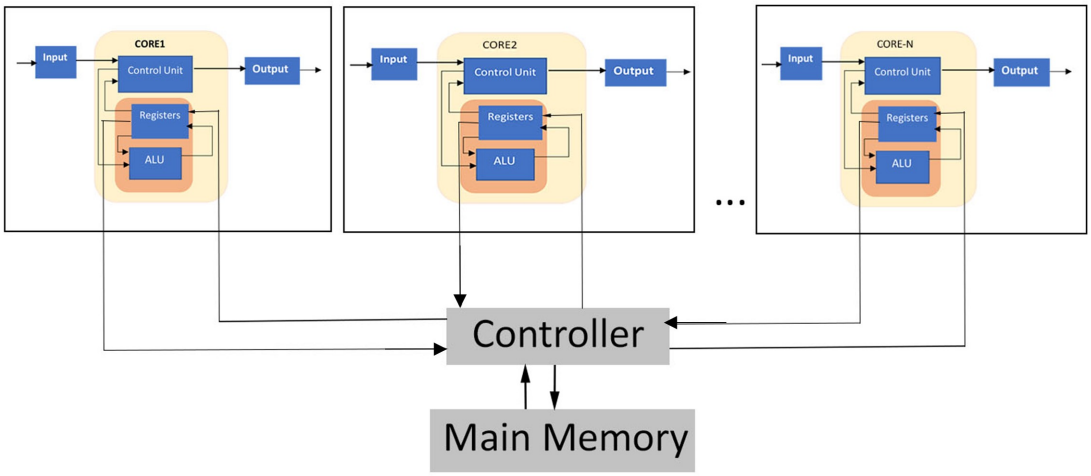


Figure 1.1: Block diagram of a SIMD Multi-core Processor

1.3 Problem statement

The task of the given project is to design a **single instruction multiple data multi-core processor** to multiply 2 matrices. There are no constrains or limits like shape of the matrices, size of matrices, signed or unsigned or variable type like floating point numbers.

The secondary necessity is to develop a system to verify the design's correctness.

1.4 Proposed solution

The main requirement is to design a multi-core processor optimized for matrix multiplication. The task can be divided into parts as follows.

1. Generate random matrices with proper dimensions.
2. Convert matrices into binary data stream considering the number of cores that is going to be used.
3. Convert instructions (Assembly code) into binary data stream.
4. Transmit the instructions to the Instruction memory module in FPGA board from PC.
5. Transmit the data (input matrices and initialization data) from PC to the data memory module FPGA board.
6. Multiply 2 matrices using designed multi-core processor in the FPGA and store the answer matrix in the data memory.

7. Send the matrix multiply answer from the FPGA to the PC and display it and validate the correctness.

The proposed solution is implemented on a ALTERA DE2-115 FPGA board which has a EP4CE115F29C7 Cyclone IV E soft-core processor. The designing process can be briefly described as followed.

1.5 Processor design flow

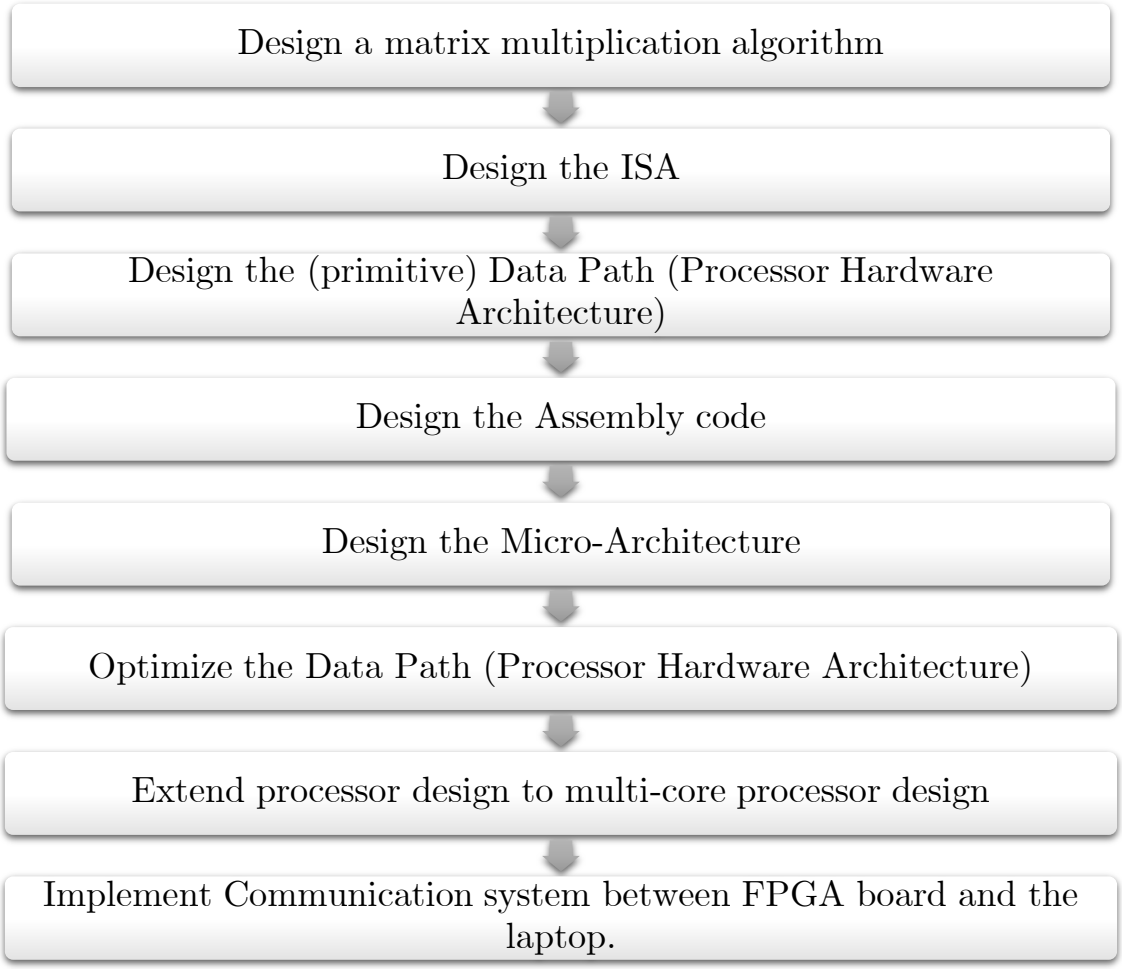


Figure 1.2: Processor Design

1.6 Processing order

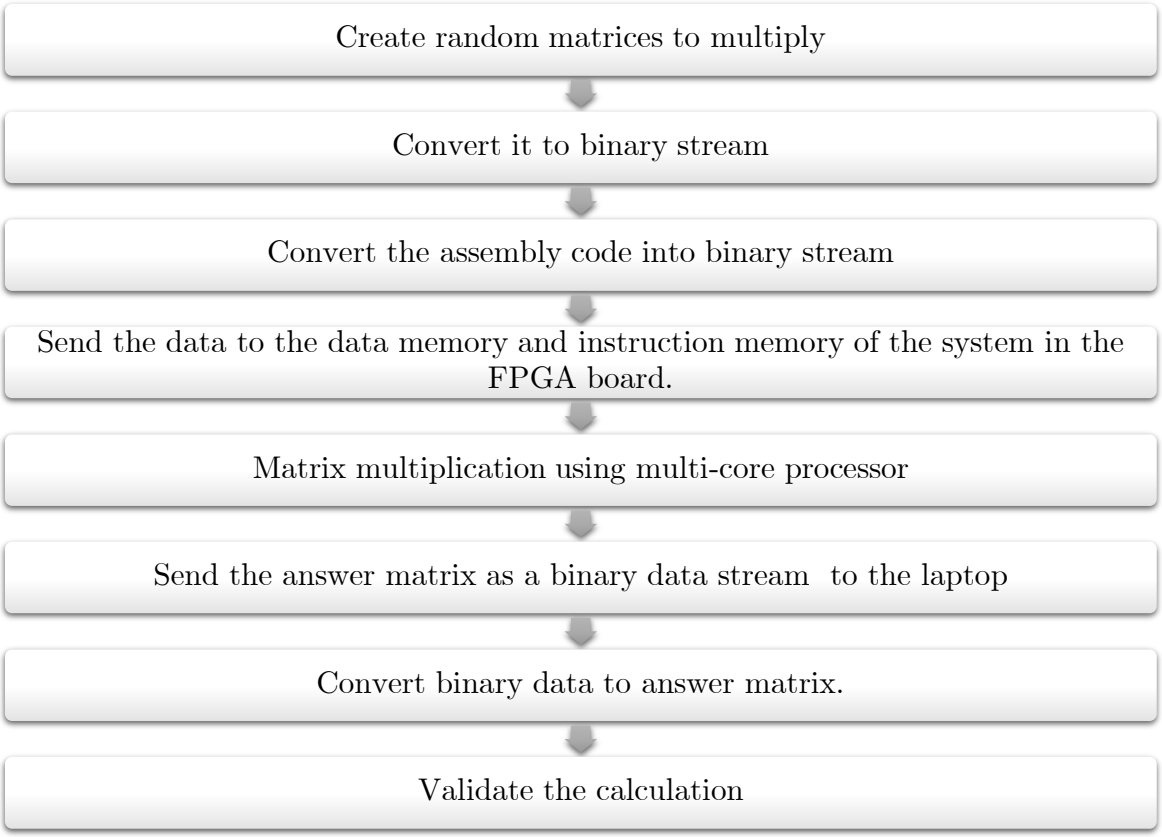


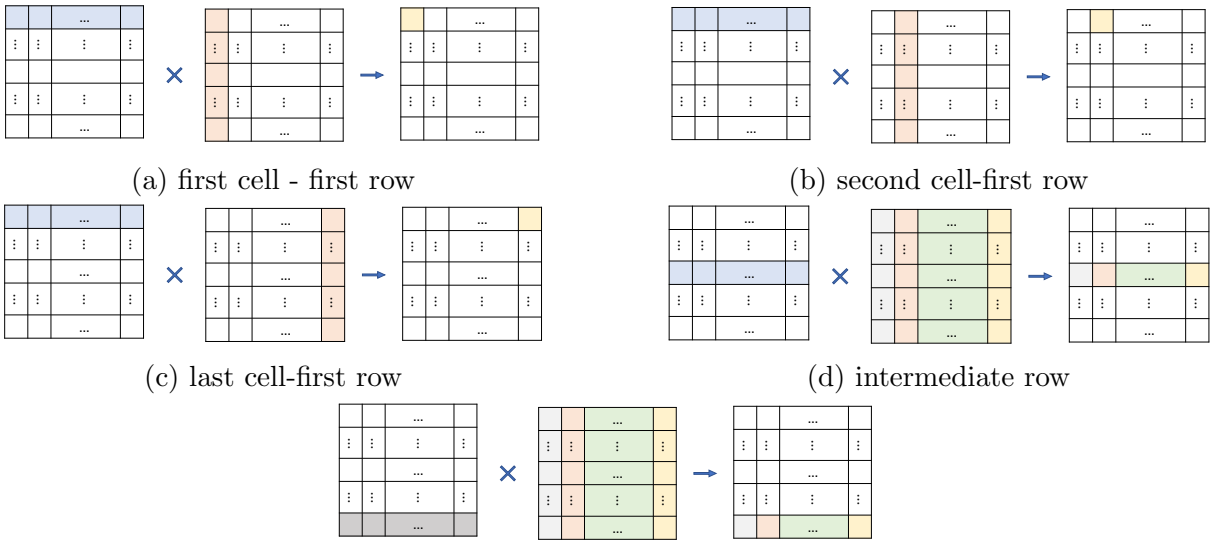
Figure 1.3: Process Flow

2 Algorithm

The first and the most important part of the project is developing a proper algorithm for Matrix calculation. We designed an algorithm which is very simple yet efficiently implementable with SIMD multi-core processor. Also the algorithm was developed such that the number of processor cores can be easily changed (Hardware level changes) without no issues happening to the software level implementation.

2.1 Matrix calculation parallelism among cores

According to the algorithm, the answer matrix is calculated row by row. The calculation of rows of the answer matrix are divided among the processor cores. Each core do the calculation parallely and independently. The process is explained with images belongs to some steps as follows.



(a) first cell - first row

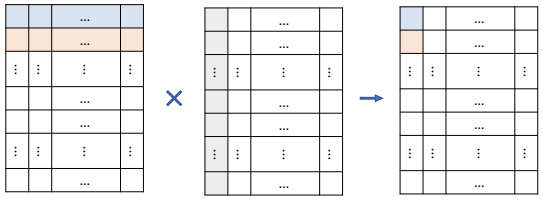
(b) second cell-first row

(c) last cell-first row

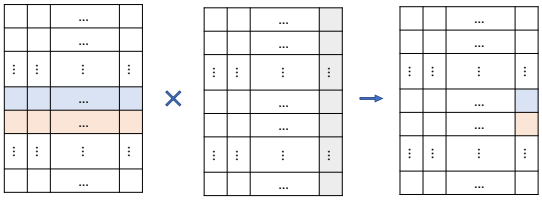
(d) intermediate row

(e) last row

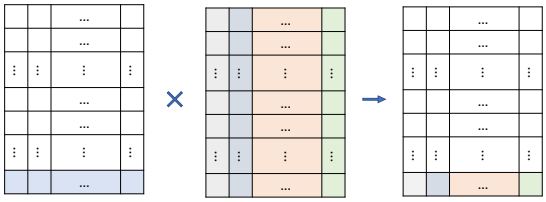
Figure 2.1: Single core multiplication algorithm



(a) first cell - first 2 rows

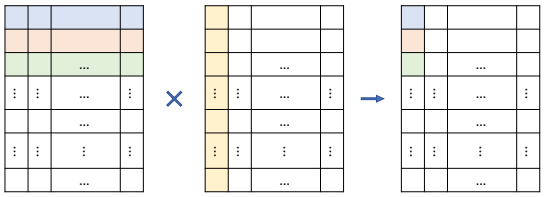


(b) last cell-intermediate 2 rows

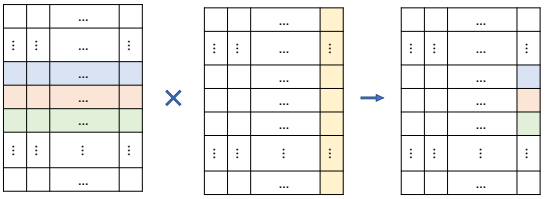


(c) last row

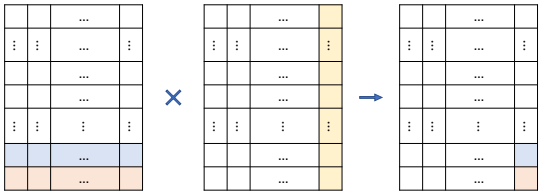
Figure 2.2: Dual-core multiplication algorithm



(a) first cell - first 3 rows



(b) last cell-intermediate 3 rows



(c) last cell-last 2 row

Figure 2.3: Three-core multiplication algorithm

2.2 Memory access handling

As the design is a SIMD multi-core processor, at a given time each core executes the same instruction. Therefore each core should receive the same instruction but different data. Though sending the same instruction to each core is not a problem, sending different data to different cores simultaneously, was a challenging task.

2.2.1 Possible data memory access methods by multiple processor cores

At the beginning, different memory accessing methods were studied and finally came to a simple, yet efficient algorithm and memory architecture. Some of the possible memory

access methods are discussed below. The task was to multiply 2 matrices which can be implemented such that same memory location will not be updated by different cores. (No interference among core.) This fact was considered when selecting the suitable memory accessing method.

- **Accessing multiple memory addresses by each core simultaneously**
  This method is the most direct method. In order to do this, one way is dividing the memory into equal parts (number of parts in the memory is equal to number of processor cores) and each core can access only to the memory partition which belongs to it. In this method an algorithm should be designed to send different addresses to the memory by each core even though they executes the same instruction. Also the memory should have number of inputs and outputs equal to the number of cores. Though this multiple input multiple output memory can be designed using HDLs, as the memory blocks in the FPGA board have only 1 input and 1 output, this memory design will be realized using registers in the FPGA board. Therefore, if the data memory is considerably large, this design may not be realizable as the number of registers on the FPGA board are very low with respect to the size of memory blocks. Also design compilation might take a long time to map the designed memory to a large number of registers in different locations on the FPGA board. In order to solve this issue, instead of accessing the main memory, small memory units (level-1 cache) for each core can be implemented. But before the process (matrix multiplication) start each cache should be filled with its relevant data.

- **Access multiple memory address by each core sequencially**
  In this method too, the memory can be partitioned to equal to number of cores equally and each core can access only to the memory part belongs to it. But now the memory need only a single input and single output port. Therefore this memory design can be implemented on a FPGA board easily with existing RAM blocks. But there should be a control unit in between data memory and multi-core processor to handle memory access by each core. But as the design is SIMD when memory accessing happens, all the cores will have to hold their process until each core access the memory location related to the current instruction.

- **Divide memory words into number of cores**
  In this method each memory word is divided into number cores. Each core will access the same memory word. But access only to the part of the memory word which belongs to that core. As the width of the data of each processor core is constant, the width of the memory word will increase with the number of cores. As an example if the width of the data required for a core is 12, and if there are 4 cores, then the width of the memory word will be 48. First core will access [0:11] bits of the memory word, second core will access [12:23] of the memory word and so on. This method can be easily implemented. It does not require extra control unit to control memory access by each core and each core will access its related area of memory simultaneously. Furthermore this memory is realizable using the existing RAM blocks on the FPGA board.

Among the above mentioned methods, the 3rd method was selected for memory accessing for the design. This method performed considerably well. We **could test with**

multi-core processor designs which includes from 1 to 6 cores with this method and this memory access method gave correct results.

3 Instruction Set Architecture

After designing a proper algorithm the next step was to design a proper ISA, assembly code and suitable hardware structure.

3.1 Instruction Set

In order to do the matrix multiplication we defined an suitable ISA. A total of 23 instructions are there. (Even though a instruction called "NOP" is defined to use for idle clock cycles it was not used within the matrix multiplication.)

INSTRUCTION	OPCODE	OPERATION
NOP	0	No operation
ENDOP	1	End operation
CLAC	2	AC ← 0
LDIAC	3	AC ← dataMem{addr}
LDAC	4	AC ← dataMem{AC}
STR	5	dataMem{AC} ← R
STIR	6	dataMem{addr} ← R
JUMP	7	PC ← instructionMem{addr}
JMPNZ	8	Jump if z!=0
JMPZ	9	Jump if z==0
MUL	10	AC ← AC*R1
ADD	11	AC ← AC+R
SUB	12	AC ← AC-RC
INCAC	13	AC ← AC+1
MV_RL_AC	14	RL ← AC
MV_RP_AC	15	RP ← AC
MV_RQ_AC	16	RQ ← AC
MV_RC_AC	17	RC ← AC
MV_R_AC	18	R ← AC
MV_R1_AC	19	R1 ← AC
MV_AC_RP	20	AC ← RP
MV_AC_RQ	21	AC ← RQ
MV_AC_RL	22	AC ← RL

Table 3.1: Instruction Set

3.2 Assembly Code

In order to do the matrix multiplication, an assembly code was developed using the designed ISA. The assembly code can be found on section - 9.6.1. It does not depend on the following factors and will remain same for all the cases.

- The number of cores in the multi-core processor.
- The size of the matrices.

In order to tackle the above factors and setup the process, the basic details (core count, matrix dimensions etc.) should be given to the multi-core processor at the beginning of the process. Therefore these details should be stored in the known addresses (first 12 memory locations) in the data memory. (Table - 3.2) The naming conventions used for matrices are given in figure - 3.1.

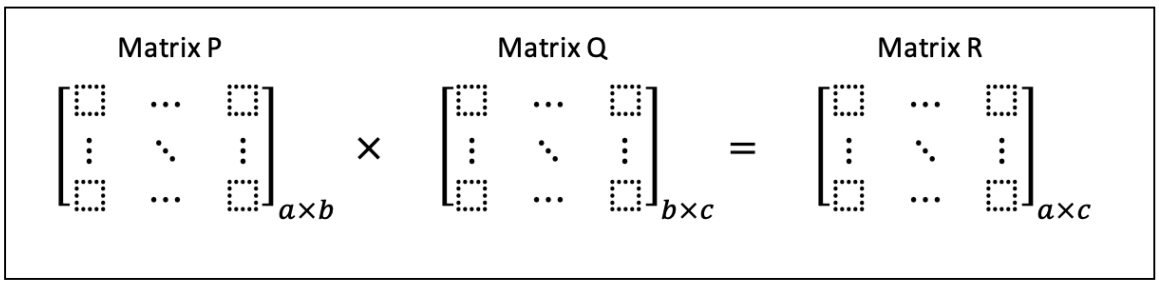


Figure 3.1: Naming convention of matrices and their dimensions

3.3 Data path (Hardware Architecture)

After designing the ISA and the assembly code, the (primitive) data path was designed. At first, a system with a single-core processor was designed. It mainly includes the processor, data memory and the instruction memory. After that the micro architecture could be developed based on this primitive data path. Then by looking at the micro-instructions the data path could be optimized by reducing redundant paths. The basic units in the data path are briefly described below.

- **Data memory** – This contains 4096 memory locations. Width of the memory is a multiple of 12 (It depends on the number of processor cores as described in memory access method selection section.
- **Instruction memory** - This contains 256 memory locations. The width of the memory is 8 bits.
- **AR** - (12-bit) Address register holds the address of the current location in the data memory.

Address	Data Value	Meaning
0	a	No. of rows of matrix P
1	b	No. of columns of matrix P (No. of rows of matrix Q)
2	c	No. of columns of matrix Q
3	start_addr_P	Start address of matrix P
4	start_addr_Q	Start address of matrix Q
5	start_addr_R	Start address of matrix R
6	end_addr_P	End address of matrix P
7	end_addr_Q	End address of matrix Q
8	current_addr_P	The address of currently processing data of matrix P
9	current_addr_Q	The address of currently processing data of matrix Q
10	current_addr_R	The address of currently processing data of matrix R
11	current_c	Current row count
12	current_a	Current column count

Table 3.2: Memory Locations of Some Necessary Data Values

- **PC** – An 8-bit program counter which keeps the address of current/next instruction in the instruction memory.
- **IR** – An 8-bit instruction register which stores the instruction read from the instruction memory.
- **R, RL, RC, RP, RQ, R1** – (12-bit) general purpose registers.
- **AC** – (12-bit) Accumulator which has direct access to the ALU. Whenever data is loaded from data memory, it is loaded to AC and data stored to data memory is stored from AC.
- **Z-register** – (1-bit) zero flag register.
- **ALU** – A 12-bit Arithmetic and Logic Unit which performs mainly 3 different operations. Those are Multiplication, Addition and Subtraction. Other than them it can pass one of the inputs to the output or set the output to zero.
- **Control Unit** – It generates all the control signals for the components of the processor based on the acquired current instruction from Instruction memory.
- **System bus** – (12-bit) wire which carries the data among registers and data memory

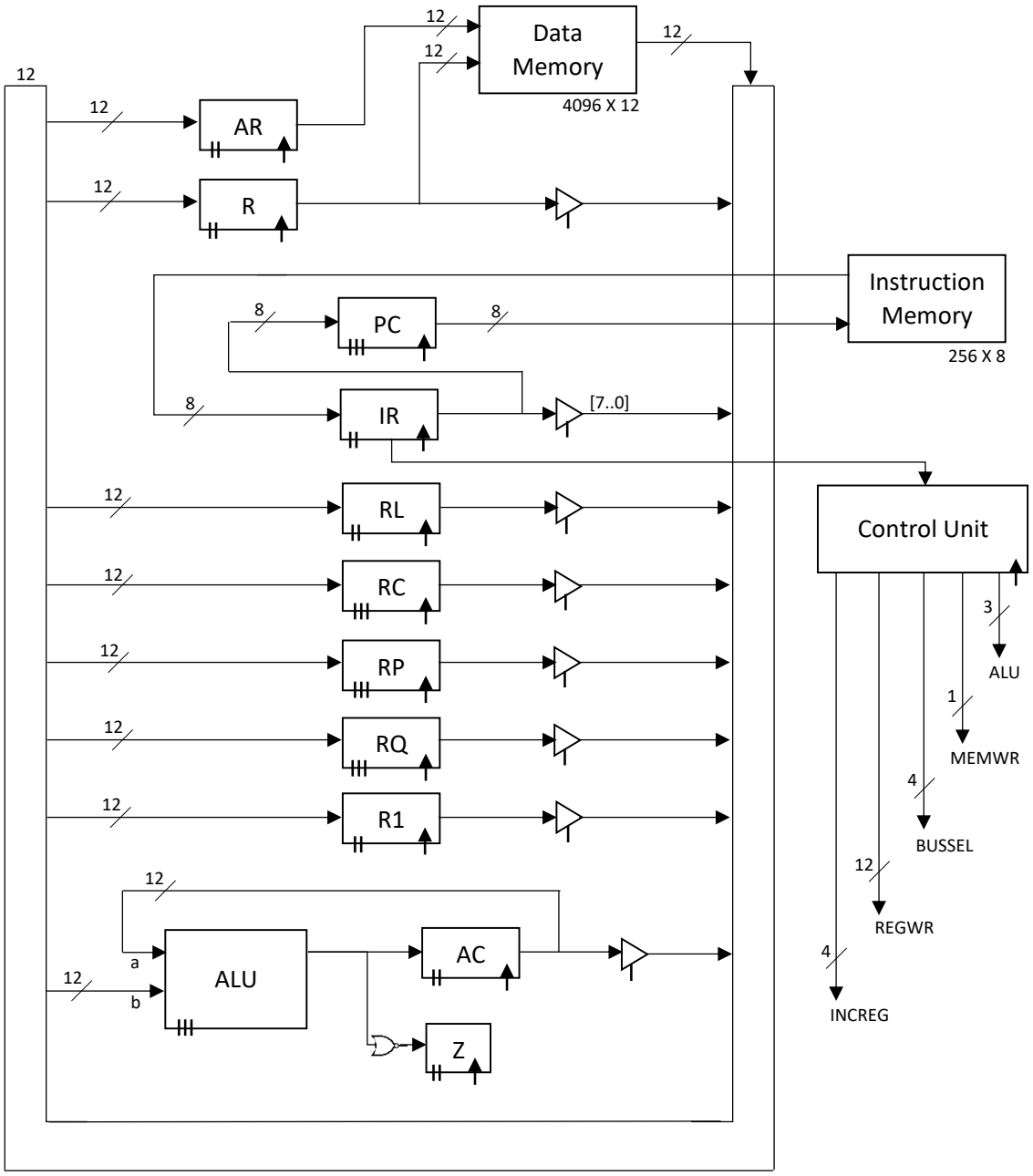


Figure 3.2: Data path of the CPU

MEMWR	-	Memory Write
BUSSEL	-	Bus Select
REGWR	-	Register Write Enable
INCREG	-	Register Increment Enable

3.4 Micro-instructions

INSTRUCTION	STATE	MICROINSTRUCTION
FETCH	FETCH_DELAY1	IR ← Instruction memory, Instruction read
	FETCH1	IR ← Instruction memory, Instruction read
	FETCH2	PC ← PC+1
NOP	NOP1	No operation
LDAC	LDAC1	AR ← AC, Data read
	LDAC_DELAY1	R ← Data Memory
	LDAC2	R ← Data Memory
LDIAC	LDIAC_DELAY1	IR ← Instruction Memory
	LDIAC1	IR ← Instruction Memory
	LDIAC2	AR ← IR, PC ← PC+1, Data read
	LDIAC_DELAY2	AC ← Data Memory
	LDIAC3	AC ← Data Memory
STR	STR1	AR ← AC, Data write
	STR_DELAY1	Data Memory ← R
	STR2	Data Memory ← R
STIR	STIR_DELAY1	IR ← Instruction Memory
	STIR1	IR ← Instruction Memory
	STIR2	AR ← IR, PC ← PC+1, Data write
	STIR_DELAY2	Data Memory ← R
	STIR3	Data Memory ← R
MV_RL_AC	MV_RL_AC1	RL ← AC
MV_R_AC	MV_R_AC1	R ← AC
MV_RP_AC	MV_RP_AC1	RP ← AC
MV_RQ_AC	MV_RQ_AC1	RQ ← AC
MV_RC_AC	MV_RC_AC1	RC ← AC
MV_R1_AC	MV_R1_AC1	R1 ← AC
MV_AC_RP	MV_AC_RP1	AC ← RP
MV_AC_RQ	MV_AC_RQ1	AC ← RQ
MV_AC_RL	MV_AC_RL1	AC ← RL
JUMP	JUMP_DELAY1	IR ← Instruction Memory
	JUMP1	IR ← Instruction Memory
	JUMP2	PC ← IR
JMPZ	JMPZY_DELAY1	IR ← Instruction Memory
	JMPZY1	IR ← Instruction Memory
	JMPZY2	PC ← IR
	JMPZN1	PC ← PC+1
JMPNZ	JMPNZY_DILAY1	IR ← Instruction Memory
	JMPNZY1	IR ← Instruction Memory
	JMPNZY2	PC ← IR
	JMPNZN1	PC ← PC+1
CLAC	CLAC1	AC ← 0
ADD	ADD1	AC ← AC+R

Continued on next page

Table 3.3: A sample long table.

Table 3.3 – continued from previous page

INSTRUCTION	STATE	MICROINSTRUCTION
SUB	SUB1	AC ← AC-RC
MUL	MUL1	AC ← AC*R1, RP ← RP+1, RQ ← RQ+1, RC ← RC+1
INCAC	INCAC1	AC ← AC+1
ENDOP	ENDOP1	End Operation

3.5 State Diagram

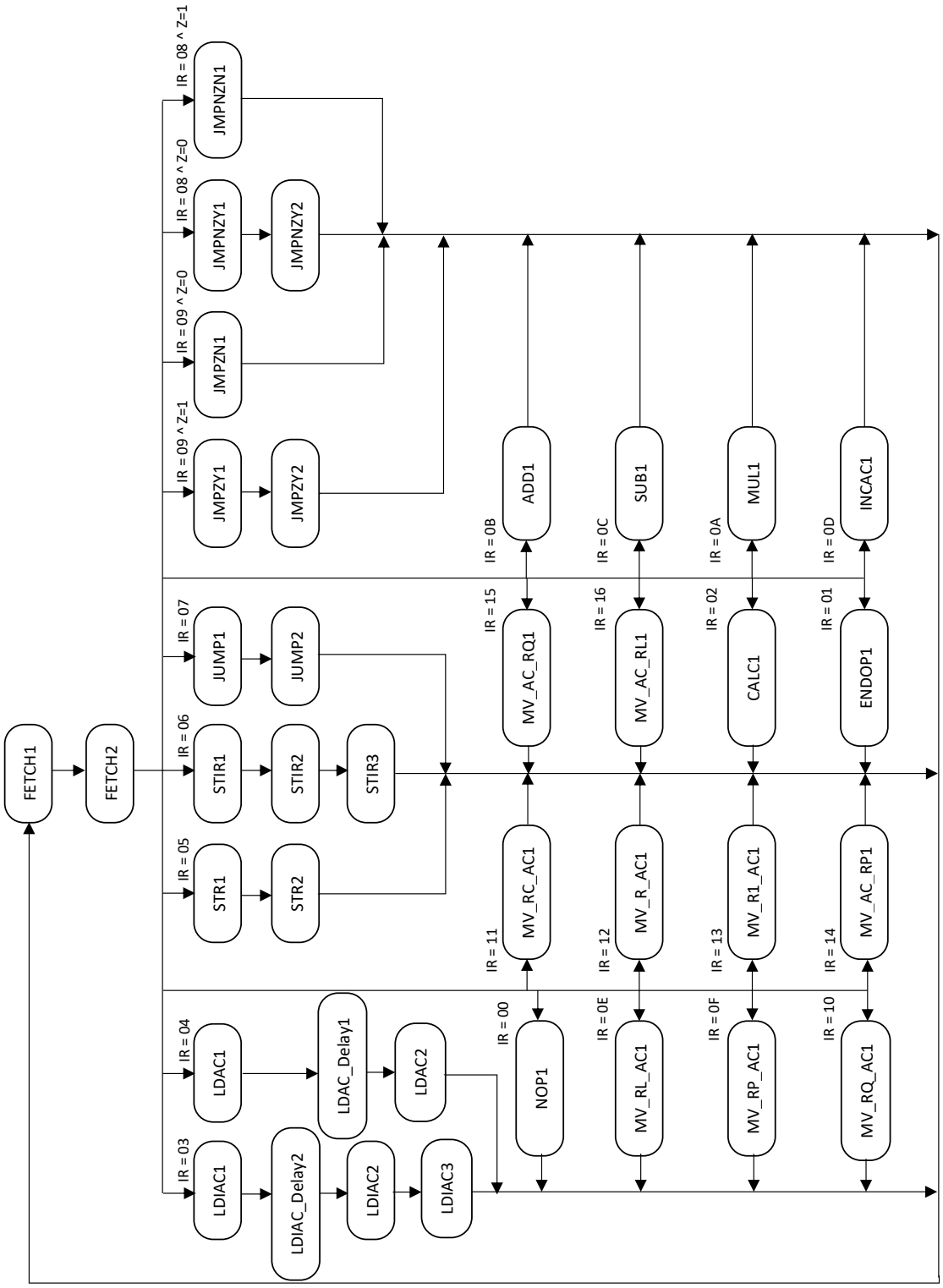


Figure 3.3: State diagram of the CPU

4 Modules

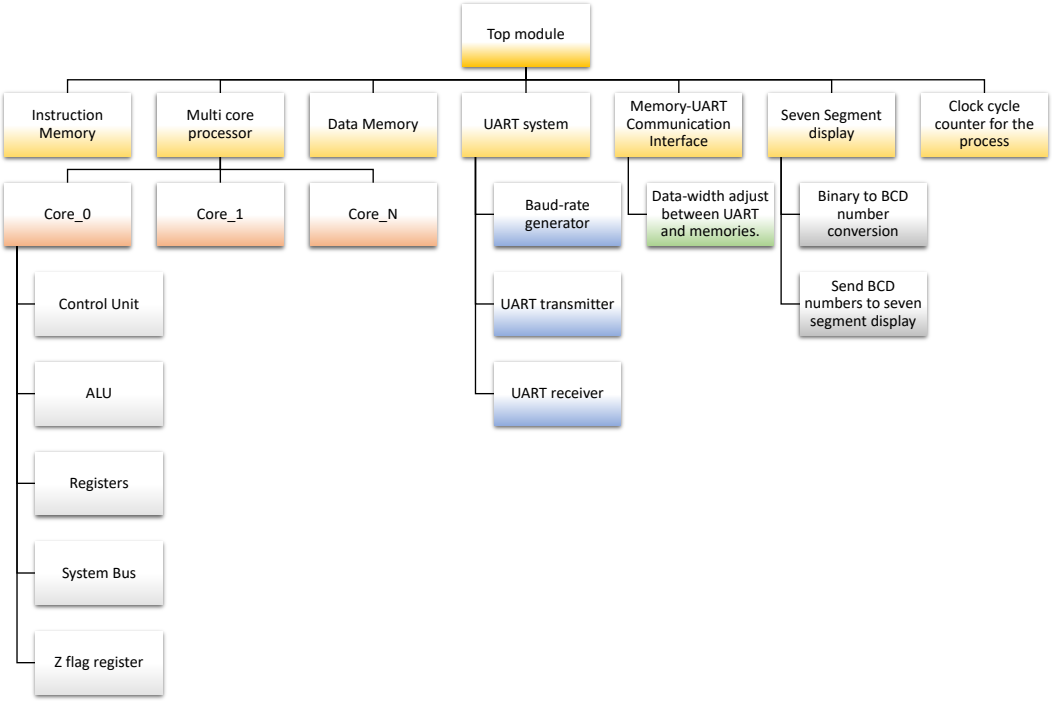


Figure 4.1: System Hierarchy

4.1 Top Module

This is the sum of all the modules connected with each other. All the instances have been created inside this module. This includes,

- Multi-core Processor
- Data memory (instance of RAM module)
- Instruction memory (instance of RAM module)
- UART-memory communication interfaces for data-memory & instruction-memory
- UART system (to communicate with a PC)
- Time counter (to count the clock cycles required for the matrix multiplication)

- HEX-display (8 Seven-segments) driver (To indicate current state of the system and show the clock count for the process after the calculation.)

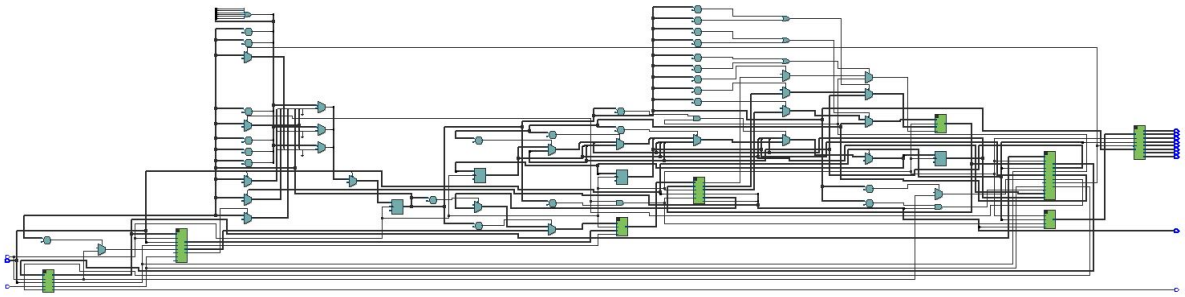


Figure 4.2: Block diagram of Top Module

4.2 Data Memory

This module is the main memory that stores data for processing purposes. The data is sent to data memory for storing after reading data from UART. Data memory module has 4096 locations with width of 12 bits. This module has four inputs and one output. The inputs are clock, write enable, a 12-bit address and a 12-bit data input. The output is a 12-bit data output (q). Above 12-bit data input and output are only for a single core. When number of cores is increased, the bit length will be increased proportional to the core count as mentioned in this 3rd method of section 2.2. For example, data width will become 24 bits when core count is two, data width will become 36 bits when core count is 3.

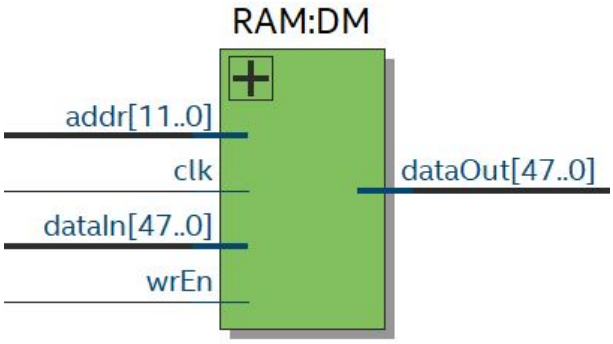


Figure 4.3: Block diagram of Data Memory

Internal data storage of data memory is shown below.

4.3 Instruction Memory

This is used for storing instructions used during the process. All instructions which is coded by assembly language is stored as a machine code in the instruction memory.

12-bit Address	12-bit data
0x000	
0x001	
. . .	. . .
0xFFF	

The processor fetches instructions from this module when it needs them to be executed. This memory stores instructions in the order of the given assembly code. This module has 256 locations with width of 8 bits. This has four inputs. Those are clock, wrEn, 8-bit address and 8-bit data. This module has an 8-bit output (q).

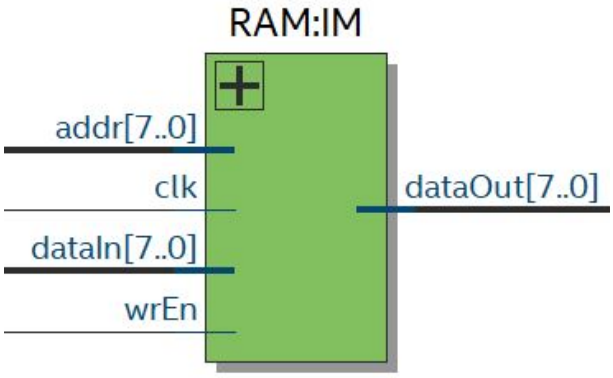


Figure 4.4: Block diagram of Instruction Memory

4.4 Multi-Core Processor

This module is the multi-core processor which includes multiple instances of single-core processor module. The data buses from data-memory and instruction memory are properly divided among each single-core processor. The number of cores are easily changed from the top module as every thing is parameterized.

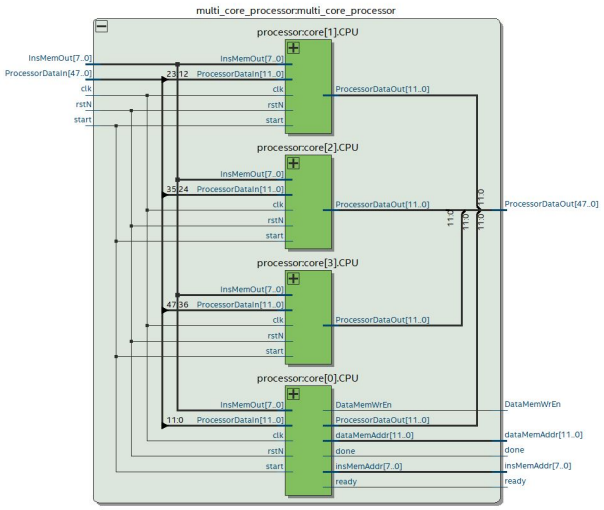


Figure 4.5: Block diagram of Muli-core processor

4.5 Processor

This is the module which contains all the instances of other modules that used for processing controlling. But instances of memory modules (data memory and instruction memory) are not included in this module.

There are five inputs for the processor.

- **clk** – This gives clock pulses for the synchronization.
- **fromDataMem** – This give data from data-memory to the processor. This has width of 12 bits.
- **fromInsMem** – An 8-bit signal that gives instructions from the instruction-memory to the processor.
- **rst** – Resets the processor to the initial state.
- **start** – give command to starts the process.

The processor module generates six outputs.

- **dMemWrEn** – This enables write to the data-memory.
- **dataMemAddr** – This gives the address of a location in the data-memory. This is 12-bit wide.
- **insMemAddr** – This gives the memory location of instruction memory. This is 8-bit wide.
- **toDataMem** – This gives the data value which needs to be written into the data memory. This has width of 12 bits.

- **ready** – Indicate processor is ready to start the process.
- **done** – This is used to indicate the end of the process.

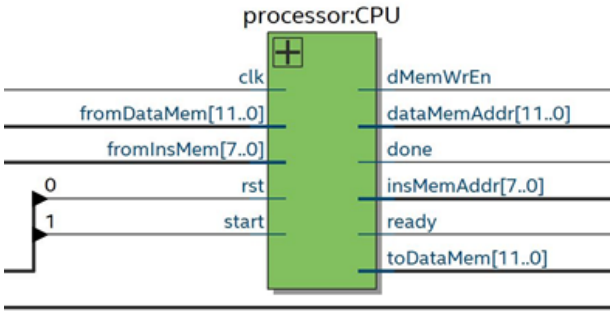


Figure 4.6: Block diagram of Processor

Followings are the modules that includes in the processor module.

- ALU
- Control Unit
- AC (Accumulator)
- PC (Program counter)
- AR (Address register)
- IR (Instruction register)
- General purpose Registers (R, RL, RC, RP, RQ, R1)
- Z (zero flag register)

4.6 Control Unit

Control unit controls the behaviour of all the components of the processor core by sending the corresponding control signals to each of them. The tasks that are controlled by the control unit are as follows.

1. Select ALU operation.
2. Select bus input from the registers and data-memory output.
3. Increase the value by 1 which are stored in ‘inc_registers‘.
4. Enable register write (including Z-register write enable).

5. Enable Data memory write.
6. Send signals (ready & done) to the user when the processor is ready to start and when the process has done the process.

All control signals are generated in this module. It has five inputs. Those are clock, reset, start, 1-bit Z-register value and 8-bit instruction. It generates six controlling signal sets and 2 indication signals.

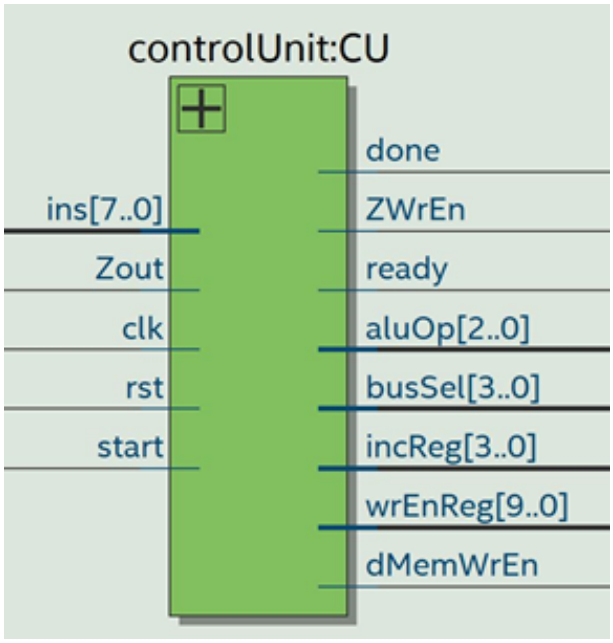


Figure 4.7: Block diagram of Control Unit

- **aluOp** – This controls all arithmetic and logic operations in ALU. This is 3-bit wide.
- **dMemWrEn** – A 1-bit signal that controls the memory write operation.
- **busSel** – Select the bus input from the register outputs and data-memory output.
- **wrEnReg** - A 10-bit signal that uses to enable write data into registers.
- **incReg** – A 4-bit signal that uses to increment value of the saved data in 'inc_reg' type registers by 1.
- **ZWrEn** – A single bit signal to enable write data into Z-register (zero flag register).
- **Ready** – A signal to indicate that the processor is ready to start the operation (matrix multiplication).
- **Done** – A signal to indicate that the process (matrix multiplication) is over.

4.7 ALU

All arithmetic and logic operations are done using this module. It has two 12-bit inputs that are 'a' and 'b'. Input 'a' is taken from the output of the AC register and input 'b' is taken from the 'data-bus' of the processor. It has one 12-bit output that is 'dataOut'. It is directly connected to the input of the AC register. It is a 12-bit bus. When an ALU operation happens, the Z flag is set to zero (if ALU output $\neq 0$) or one (if ALU output == 0).

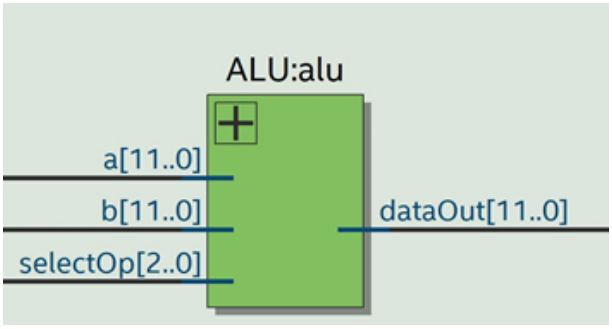


Figure 4.8: Block diagram of ALU

ALU operation are done according to the 'selectOp' 3-bit control signal. Those operations are as follows.

selectOp	Operation	Description
000	clr	dataOut ← 0
001	pass	dataOut ← b
010	add	dataOut ← a+b
011	sub	dataOut ← a-b
100	mul	dataOut ← a * b
101	inc	dataOut ← dataOut +1
110/111	idle	No operation

Table 4.1: Operations of ALU

4.8 Register

Registers are used to store data temporarily within the processor when data needed for processing tasks. These can store 12-bit data. This type of registers does not have increment flag. Therefore, incrementing the register value must be done through the ALU. Data stored in a register is available at 'dataout' and it is connected to the data-bus of the processor (except IR). So, it can select which data should be read to the bus. This module has four inputs and one output.

- When 'wrEn' is HIGH, data available at 'dataIn' is written to the register at the positive edge of clock cycle.

- 'rst' is used to clear the data within the register.
- 'clk' is the clock input used for synchronizing.

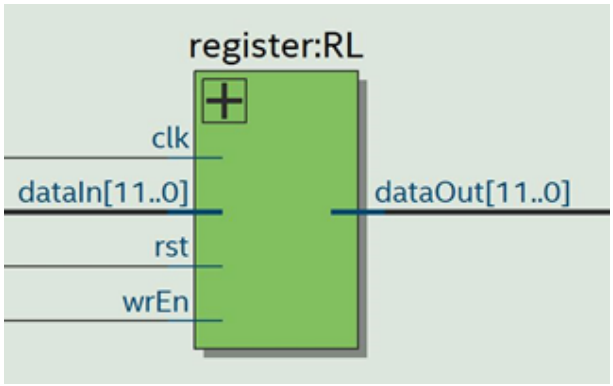


Figure 4.9: Block diagram of Register

4.9 Inc_Register

This has all the capabilities that a normal register has. More than that these registers can be incremented by 1 directly. Therefore, no need to go through an ALU operation for incrementing. This can be done by enabling the 'incEn' input. Then at the positive edge of clock cycle the value will be incremented.

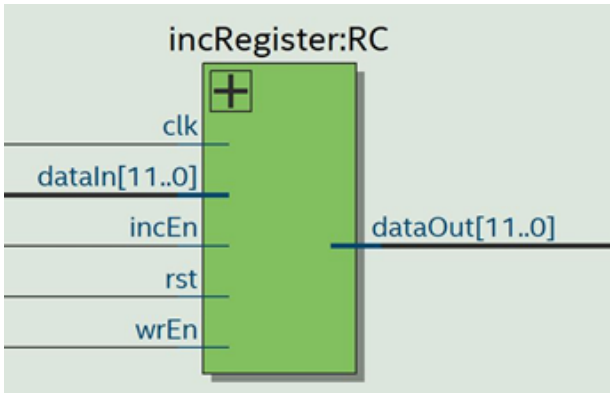


Figure 4.10: Block diagram of Inc_Register

4.10 Z Register

Z register is used as a zero-flag register to find whether the ALU operation produces a zero or not. If the ALU output is zero z-register value will be 1. Otherwise, it will be 0.

- 'clk' used for synchronization

- ‘wrEn’ used to enable write z- register only after an ALU operation happens.
- ‘rst’ is used to reset the z-register value.
- ‘Zout’ is directly connected to the Control Unit of the processor.

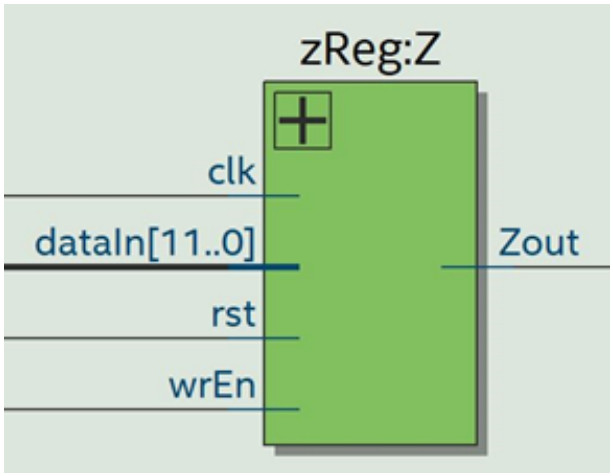


Figure 4.11: Block diagram of Z Register

4.11 Multiplexer (Data-Bus)

DATA-BUS consists of set of 12 wires which transport data among registers in the processor. In the architecture, inputs and outputs of the registers such as R, RL, RC, RP, RQ, R1 and AC are connected to the bus. Therefore, bus can read data from and write to those registers. Also, Data Memory is connected to this data-bus.

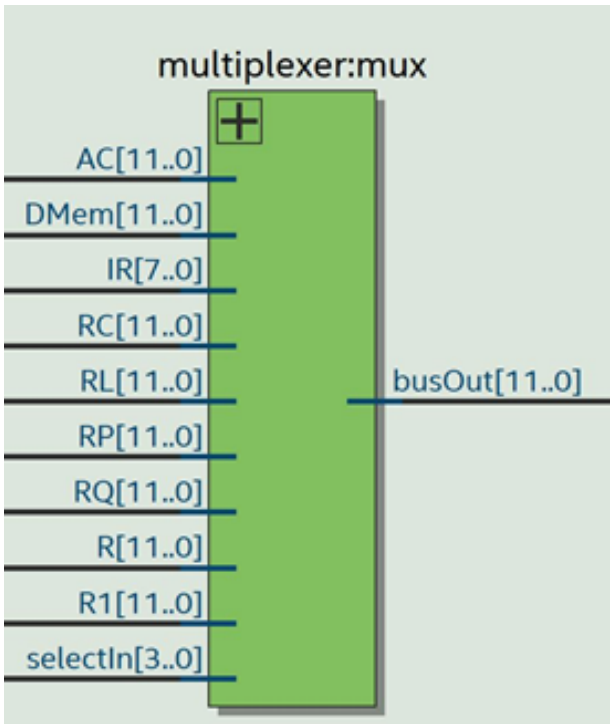


Figure 4.12: Block diagram of Multiplexer

In the multiplexer there is a 4-bit control signal ('selectIn') to select the register or the memory-output that needed to be read to the bus. Control signal is set as below to read data from the registers to the bus.

Input of the bus	**'selectIn' control signal**
Data Memory	0000
R	0001
IR	0010
RL	0011
RC	0100
RP	0101
RQ	0110
R1	0111
AC	1000
idle	1001..1111

Table 4.2: Flag value to select register

5 Simulation, Testing & Debugging

- Simulation was done using “Questasim 10.6” software.
- Each of the modules related to matrix multiplication are tested with simulations using proper test benches.
- Modules related to UART communication are also simulated and tested with test benches in SystemVerilog design.
- In order to simulate top level modules in the hierarchy (ex:- top_module, multi-core_processor, processor) random matrices were generated using a python script and used them to calculate their multiplication using the processor design and validate the calculated answer using another python script.

After the design was uploaded to the FPGA board, the following tesing and debugging methods were used to validate the performance and accuracy of the system.

- To test the design after uploaded to the FPGA board, used ”1-PORT RAM ip-core” and used the “In system memory content editor” tool of Quartus Prime software to access the memory content from the memory real time.
- “Signal Tap Logic Analyzer” is used to find the values of the wires, registers, inputs and outputs for debugging purposes.

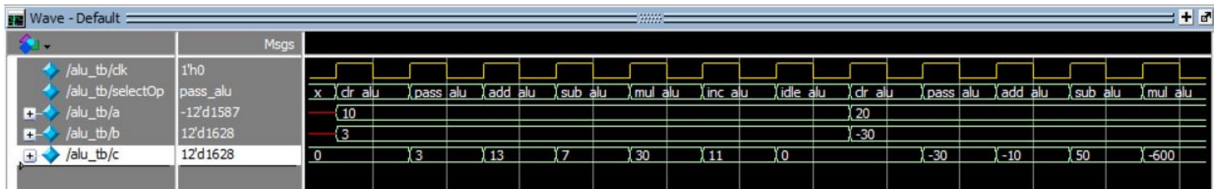


Figure 5.1: ALU Simulation

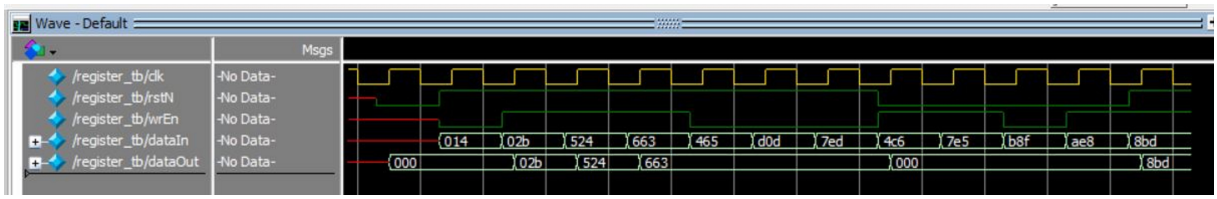


Figure 5.2: Register Simulation

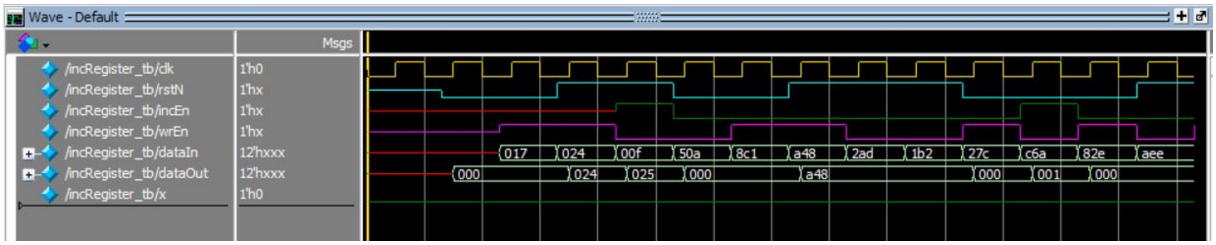


Figure 5.3: Inc Register Simulation

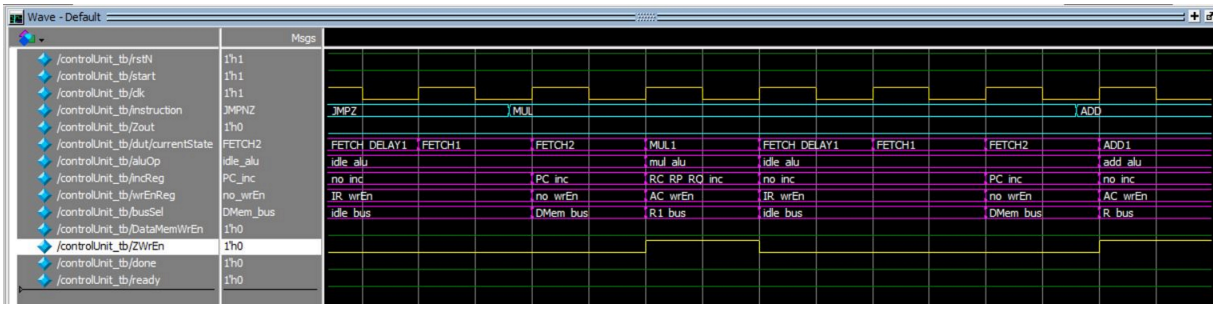


Figure 5.4: Control Unit Simulation

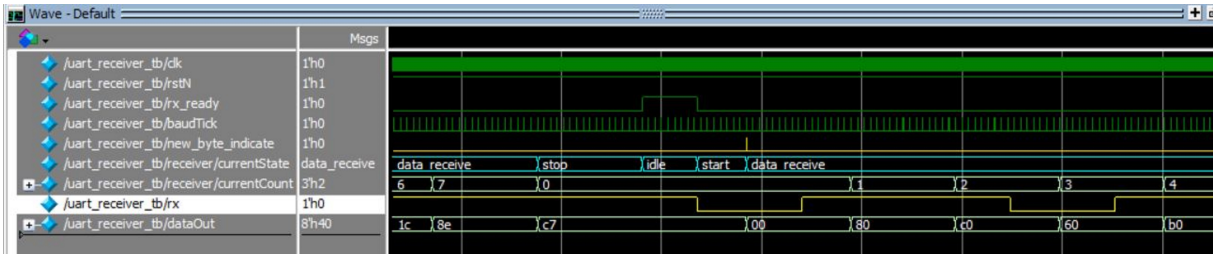


Figure 5.5: UART Receiver Simulation

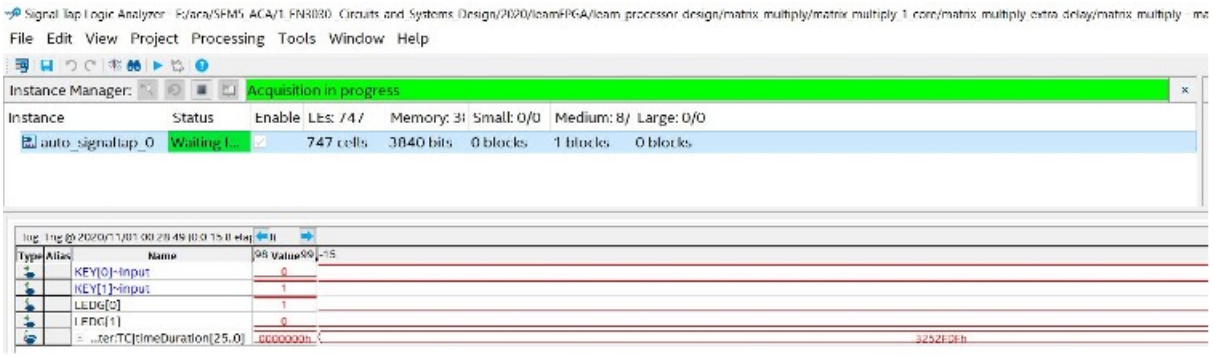


Figure 5.6: Signal Tap Logic Analyser Usage

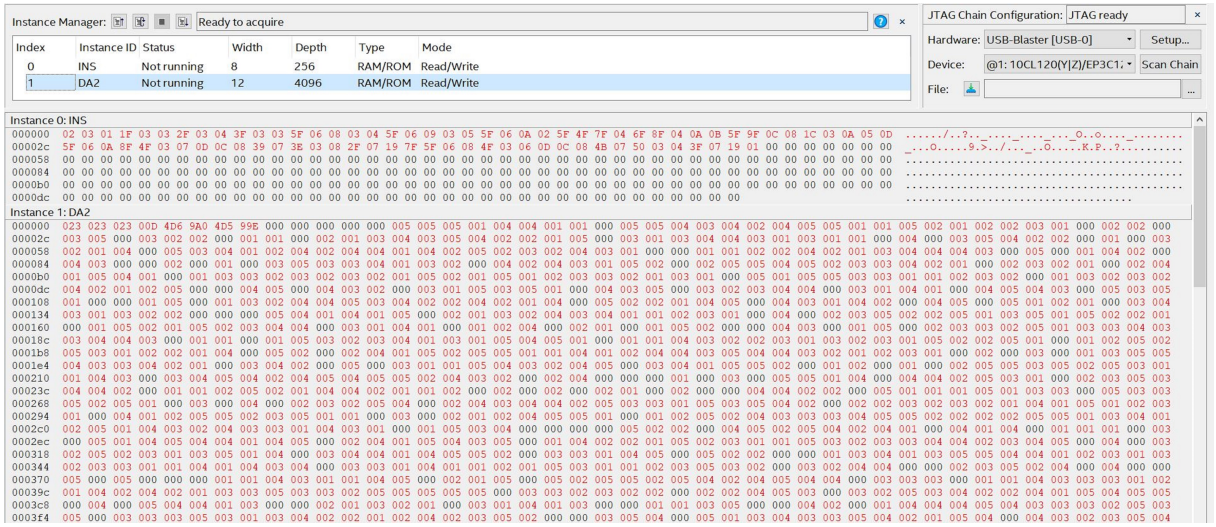


Figure 5.7: In System Memory Content Editor Usage

6 Results Analyzing and Verification

Analyzing and verifying results from the processor is very important since it gives an idea about the accuracy and the error rate. The methods that we used are as follows.

- Simulating the processor using RTL simulation using Modelsim software. The test benches (memory initialization values) used for the simulations are in the appendix O and P.
- Comparing outputs from the processer with the outputs from a python code for same set of input matrices, used to check the correctness of the answer matrix. (Python codes are given in appendix A, B, C, D)
- RTL (Netlist) viewer is used to find whether the overall design is the one that we expected.

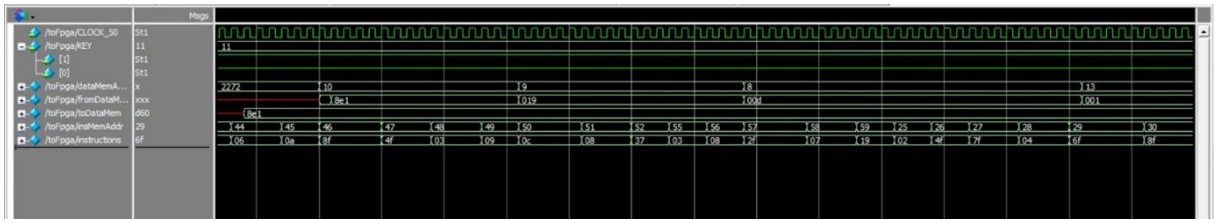


Figure 6.1: Simulation of memory access of the processor

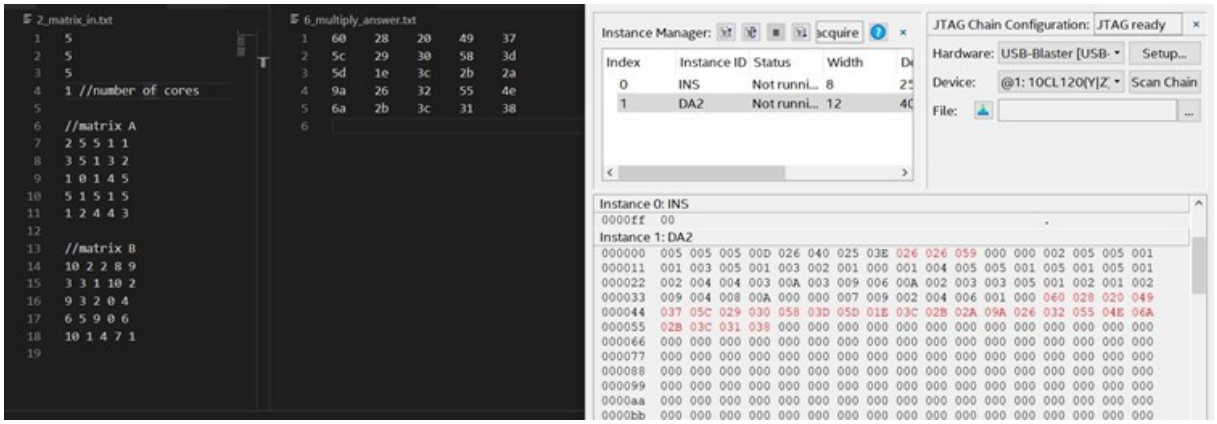


Figure 6.2: Comparison of Two 5x5 Matrices Multiplication Results of a Single-Core Processor & Python Script

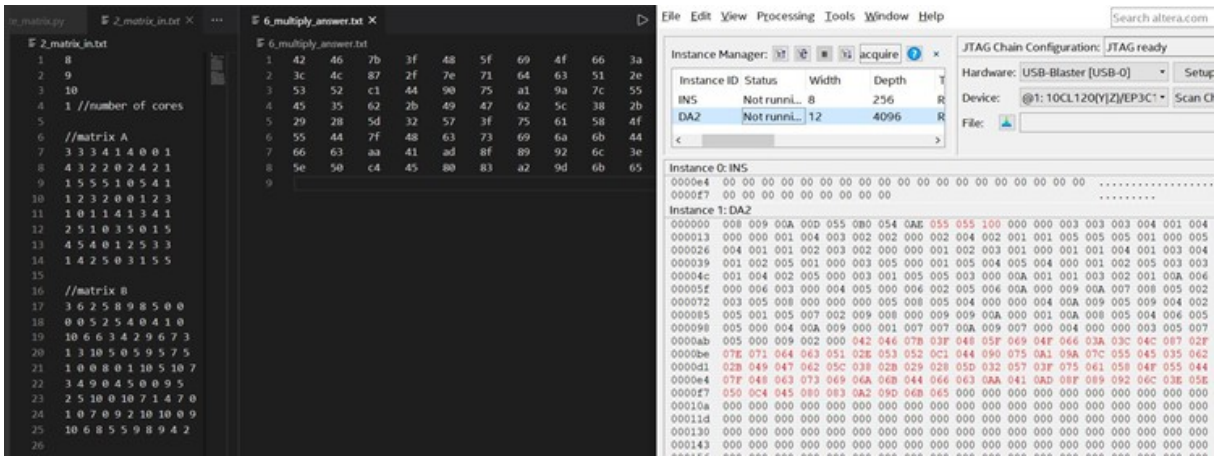


Figure 6.3: Comparison of 8x9 Matrix & 9x10 Matrix Multiplication Results of a Single-Core Processor & Python Script

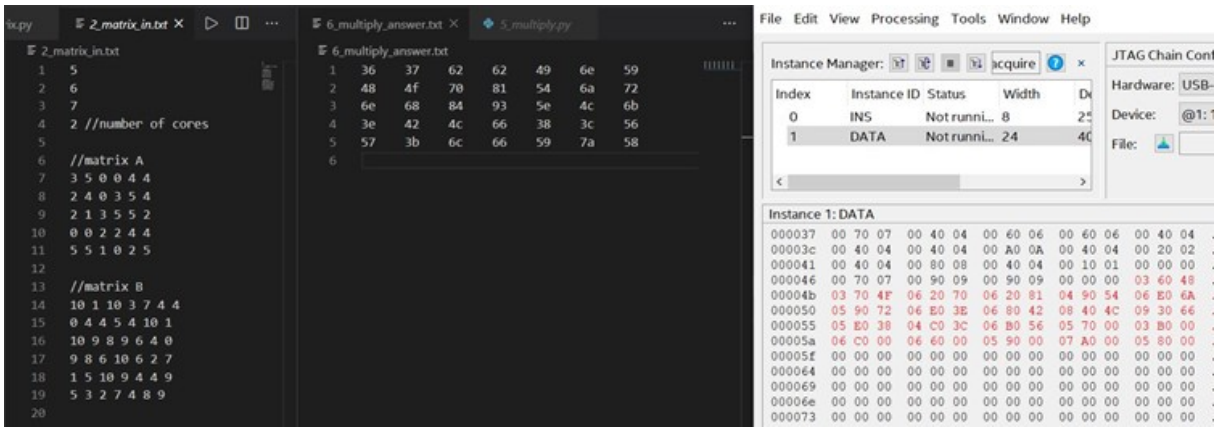


Figure 6.4: Comparison of 5x6 Matrix & 6x7 Matrix Multiplication Results of a Dual-Core Processor & Python Script

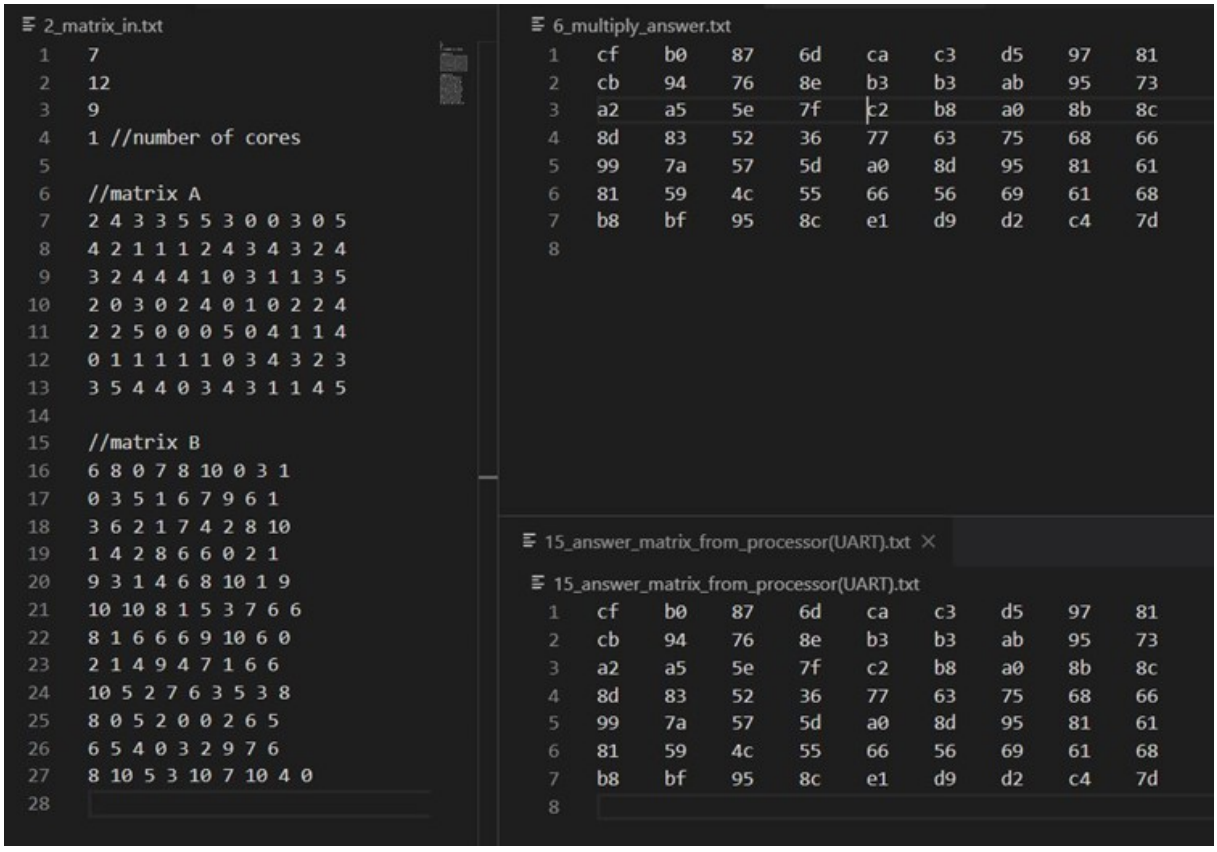
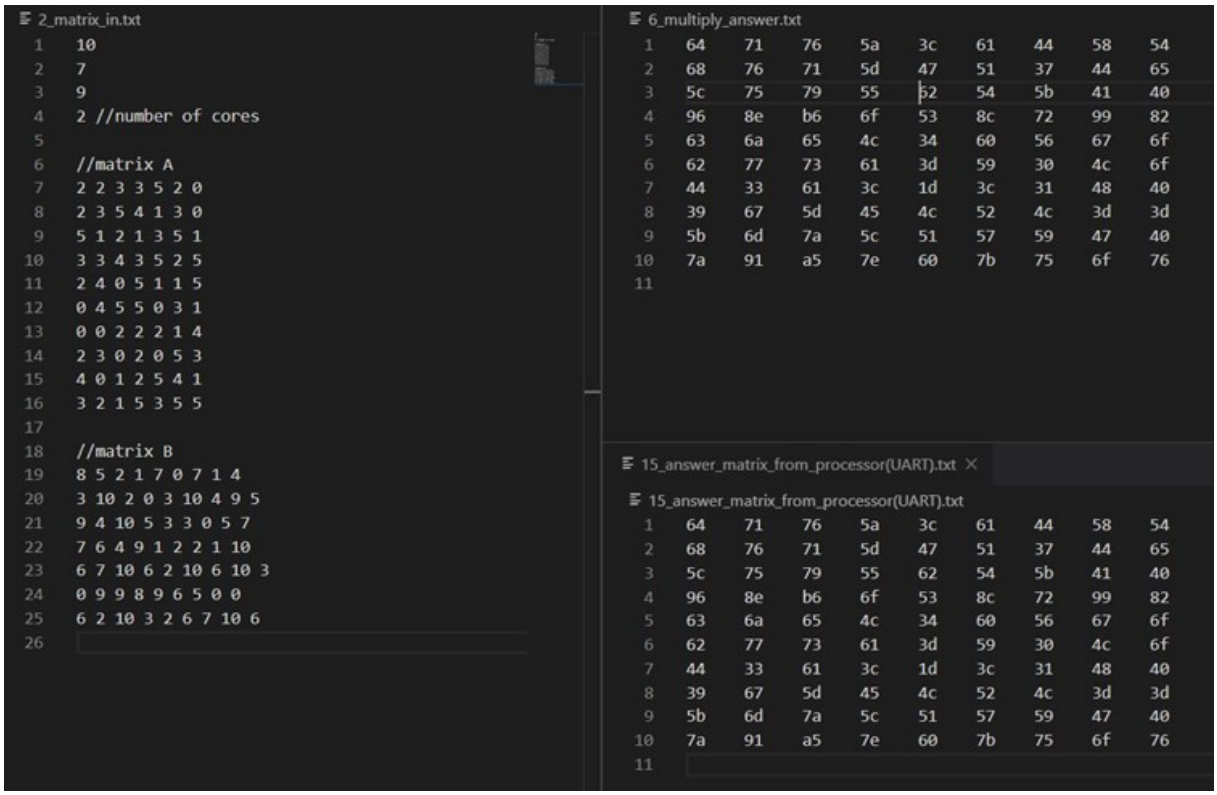


Figure 6.5: Comparison of 7x12 Matrix & 12x9 Matrix Multiplication Results of a Single-Core Processor (received using UART) & Python Script



2_matrix_in.txt

```
10
7
9
2 //number of cores

//matrix A
2 2 3 3 5 2 0
2 3 5 4 1 3 0
5 1 2 1 3 5 1
3 3 4 3 5 2 5
2 4 0 5 1 1 5
0 4 5 5 0 3 1
0 0 2 2 2 1 4
2 3 0 2 0 5 3
4 0 1 2 5 4 1
3 2 1 5 3 5 5

//matrix B
8 5 2 1 7 0 7 1 4
3 10 2 0 3 10 4 9 5
9 4 10 5 3 3 0 5 7
7 6 4 9 1 2 2 1 10
6 7 10 6 2 10 6 10 3
0 9 9 8 9 6 5 0 0
6 2 10 3 2 6 7 10 6
```

6_multiply_answer.txt

```
64 71 76 5a 3c 61 44 58 54
68 76 71 5d 47 51 37 44 65
5c 75 79 55 62 54 5b 41 40
96 8e b6 6f 53 8c 72 99 82
63 6a 65 4c 34 60 56 67 6f
62 77 73 61 3d 59 30 4c 6f
44 33 61 3c 1d 3c 31 48 40
39 67 5d 45 4c 52 4c 3d 3d
5b 6d 7a 5c 51 57 59 47 40
7a 91 a5 7e 60 7b 75 6f 76
```

15_answer_matrix_from_processor(UART).txt

15_answer_matrix_from_processor(UART).txt

```
64 71 76 5a 3c 61 44 58 54
68 76 71 5d 47 51 37 44 65
5c 75 79 55 62 54 5b 41 40
96 8e b6 6f 53 8c 72 99 82
63 6a 65 4c 34 60 56 67 6f
62 77 73 61 3d 59 30 4c 6f
44 33 61 3c 1d 3c 31 48 40
39 67 5d 45 4c 52 4c 3d 3d
5b 6d 7a 5c 51 57 59 47 40
7a 91 a5 7e 60 7b 75 6f 76
```


Figure 6.6: Comparison of 10x7 Matrix & 7x9 Matrix Multiplication Results of a Dual-Core Processor (received using UART) & Python Script

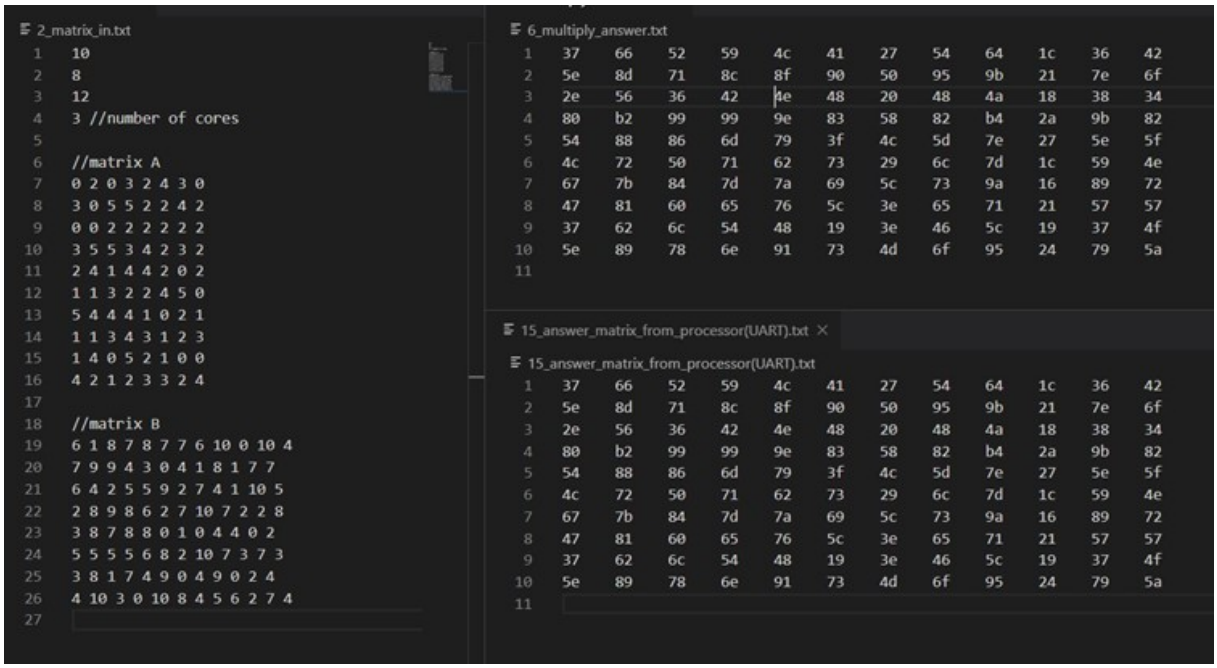


Figure 6.7: Comparison of 10x8 Matrix & 8x12 Matrix Multiplication Results of a 3-Core Processor (received using UART) & Python Script

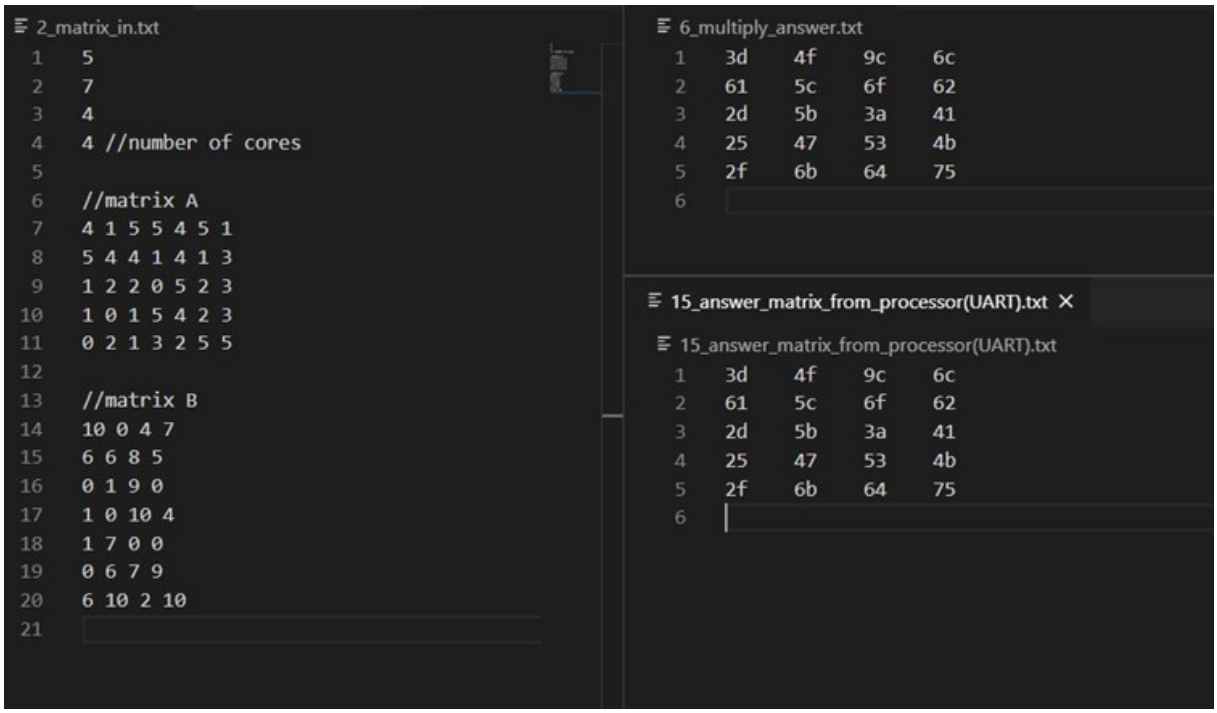


Figure 6.8: Comparison of 5x7 Matrix & 7x4 Matrix Multiplication Results of a 4-Core Processor (received using UART) & Python Script

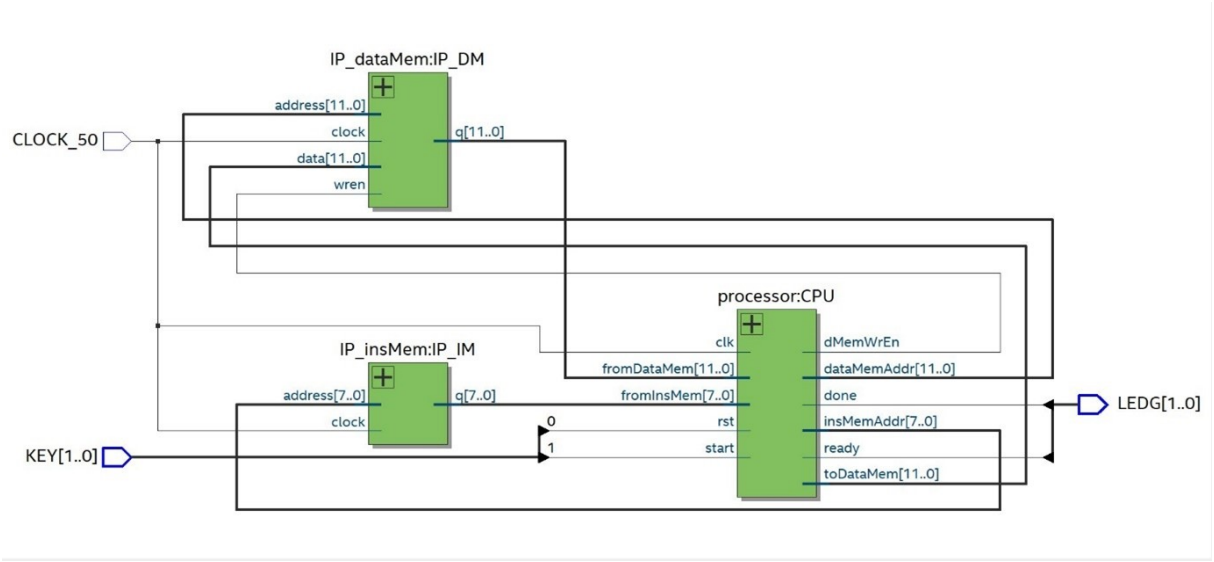


Figure 6.9: Single-Core Processor

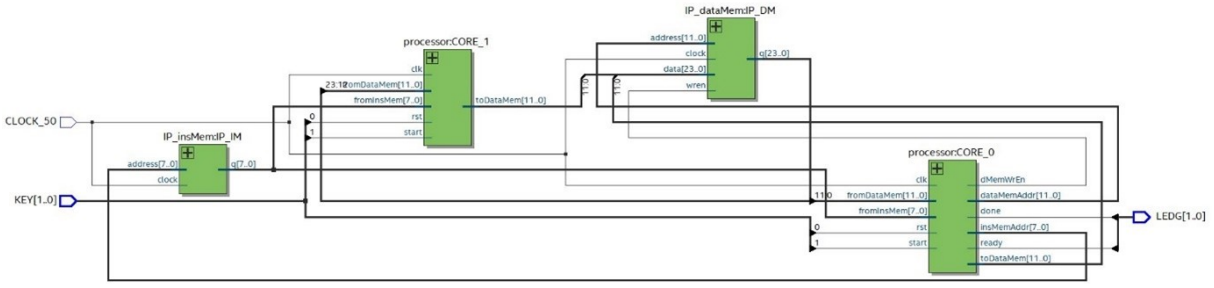


Figure 6.10: Dual-Core Processor

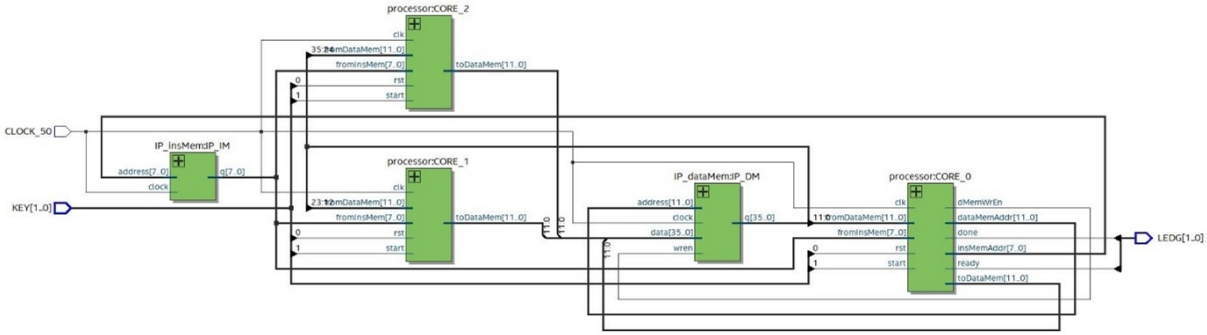


Figure 6.11: Three-Core Processor

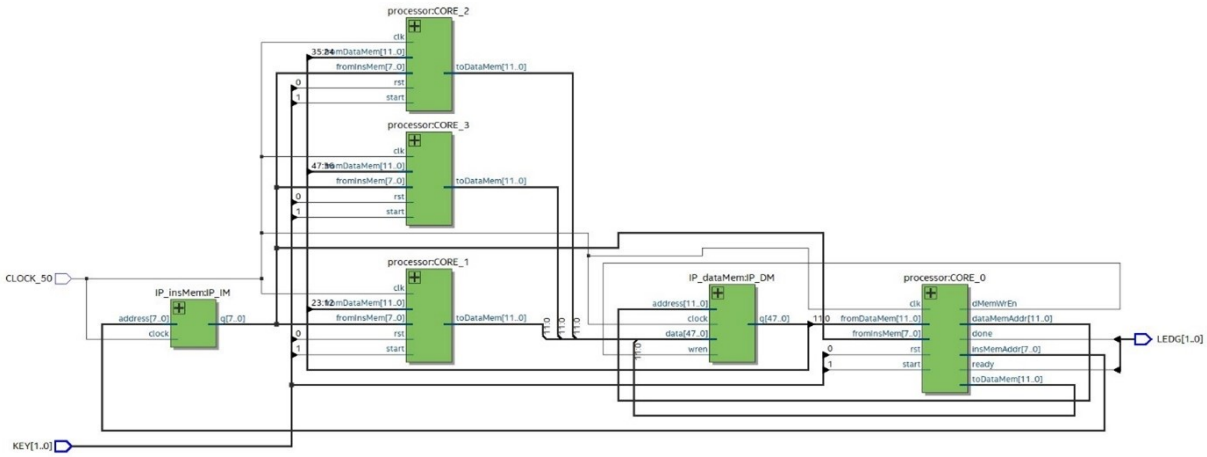


Figure 6.12: Four-Core Processor

6.1 Comparison of Speeds of Multicore Processor Designs

When the number of clock cycles needed for the multiplication process increases with the dimensions of the matrices and reduces significantly when number of processor cores increased. Different test cases are given below. The symbols 'a', 'b', 'c' refers to the dimensions of two input matrices described as in Figure 4.1'.

No.	Matrix dimensions			No. of clock cycles for the process			
	a	b	c	Single-Core	Dual-Core	Three-Core	Four-Core
1	15	15	15	143209	76408	47779	38236
2	15	15	25	238309	127128	79479	63596
3	15	15	35	333409	177848	111179	88956
4	15	25	15	228909	122008	76279	61036
5	15	25	25	380809	203128	126979	101596
6	15	25	35	532909	284248	177679	142156
7	15	35	15	314209	167608	104779	83836
8	15	35	25	523309	279128	174479	139596
9	15	35	35	732409	390648	244179	195356
10	25	15	15	238639	124123	85951	66865
11	25	15	25	397139	206543	143011	111245
12	25	15	35	555639	288963	200071	155625
13	25	25	15	381139	198223	137251	106765
14	25	25	25	634639	330043	228511	177745
15	25	25	35	888139	461863	319771	248725
16	25	35	15	523639	272323	188551	146665
17	25	35	25	872139	453543	314011	244245
18	25	35	35	1220639	634763	439471	341825
19	35	15	15	334069	171838	114580	85951
20	35	15	25	555969	285958	190660	143011
21	35	15	35	777869	400078	266740	200071
22	35	25	15	533569	274438	182980	137251
23	35	25	25	888469	456958	304660	228511
24	35	25	35	1243369	639478	426340	319771
25	35	35	15	733069	377038	251380	188551
26	35	35	25	1220969	627958	418660	314011
27	35	35	35	1708869	878878	585940	439471

Table 6.1: Speed Comparison of Processors with Different Number of Cores

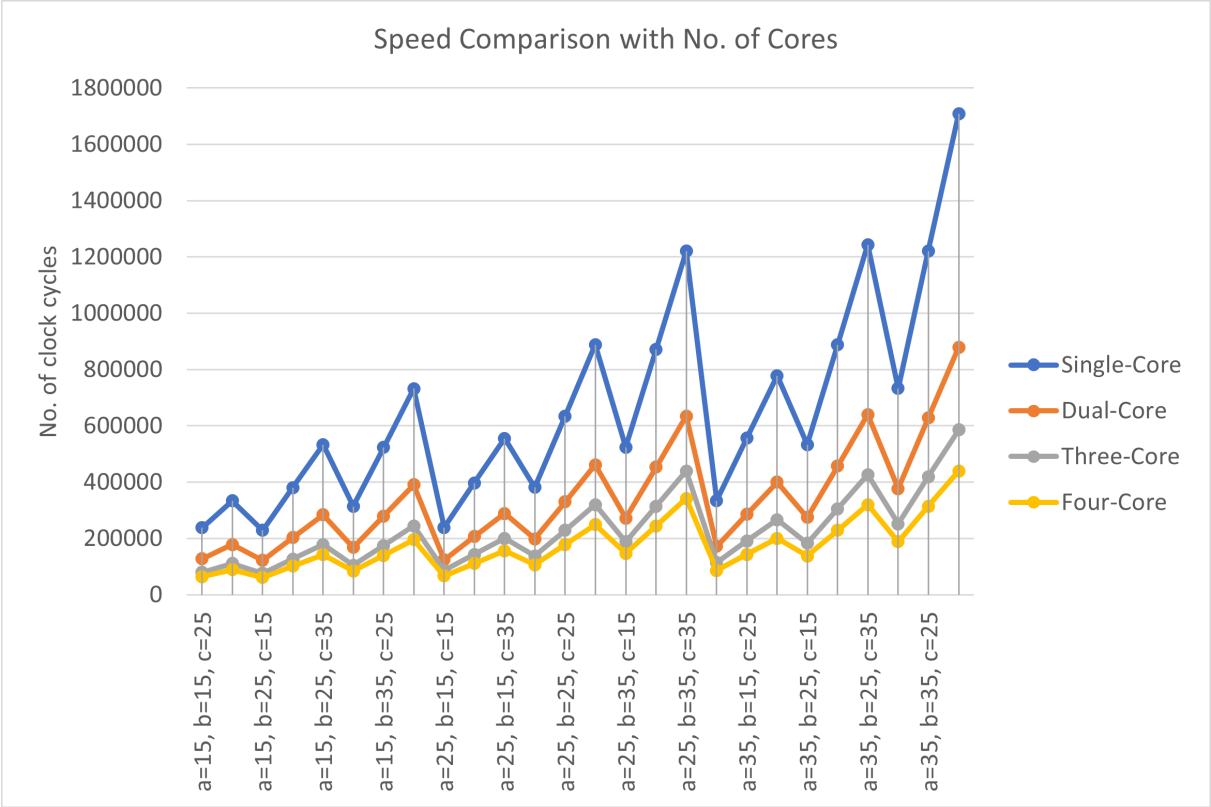


Figure 6.13: Speed Comparison of Processors with Different Number of Cores

Some extra test cases for single-core dual-core, three-core, four-core processors are given below.

Single-core Processor			
a	**b**	**c**	**No. of clock cycles**
2	2	2	690
2	3	4	1554
4	3	2	1620
5	5	5	6579
5	5	15	19279
5	5	25	31979
5	15	5	16079
5	15	15	47779
5	15	25	79479
5	25	5	25579
5	25	15	76279
5	25	25	126979
15	5	5	19609
15	5	15	57709
15	5	25	95809
15	15	5	48109

Table 6.2: Speed Comparison for Single-Core Processor

Dual-core Processor			
a	**b**	**c**	**No. of clk cycles**
35	35	40	1004338
35	40	35	998578
35	40	40	1141138
40	35	35	976524
40	35	40	1115924
40	40	35	1109524
40	40	40	1267924
40	40	45	1426324
40	45	40	1419924
40	45	45	1597324
45	40	40	1458103
45	40	45	1640263
45	45	40	1632903
45	45	44	1796111

Table 6.3: Speed Comparison for Dual-Core Processor

Three-core Processor			
a	**b**	**c**	**No. of clock cycles**
35	35	50	836860
35	50	35	825340
35	50	50	1178860
50	35	35	830055
50	35	50	1185525
50	50	35	1169205
49	49	49	1604983

Table 6.4: Speed Comparison for Three-Core Processor

Four-core Processor			
a	**b**	**c**	**No. of clk cycles**
35	35	45	564931
35	35	55	690391
35	45	35	559171
35	45	45	718831
35	45	55	878491
35	55	35	678871
35	55	45	872731
35	55	55	1066591
45	35	35	585940
45	35	45	753220
45	35	55	920500
45	45	35	745540
45	45	45	958420
45	45	55	1171300
45	55	35	905140
45	55	45	1163620
55	35	35	683586
55	35	45	878746
55	35	55	1073906
55	45	35	869786
55	45	45	1118146
55	45	55	1366506
55	55	35	1055986
55	55	45	1357546
52	52	52	1379533

Table 6.5: Speed Comparison for Four-Core Processor

7 Design Summary

7.1 Flow Summary

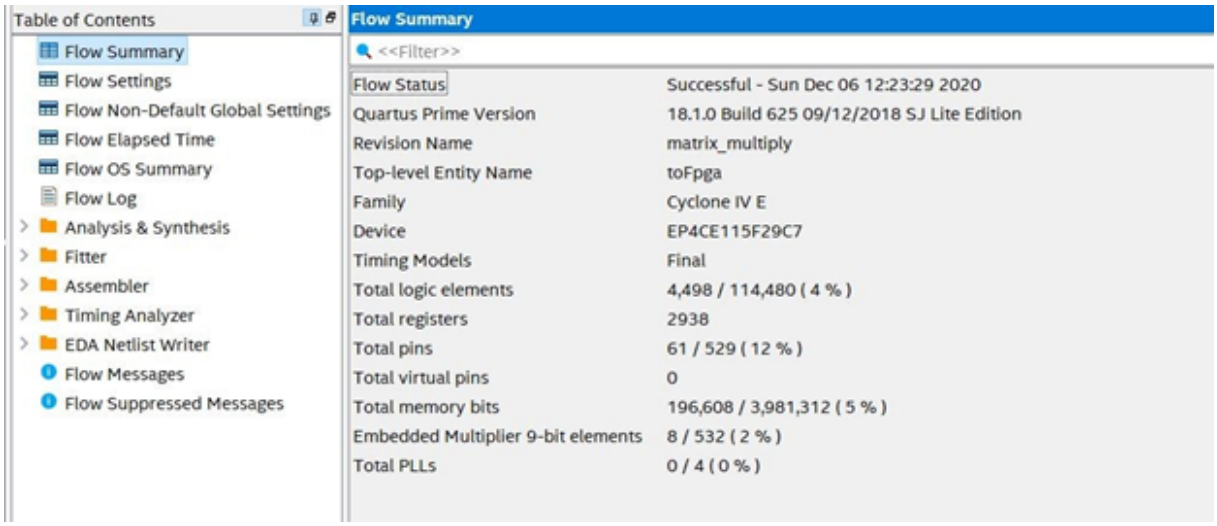


Figure 7.1: Flow Summary

7.2 Analysis & Synthesis Summary

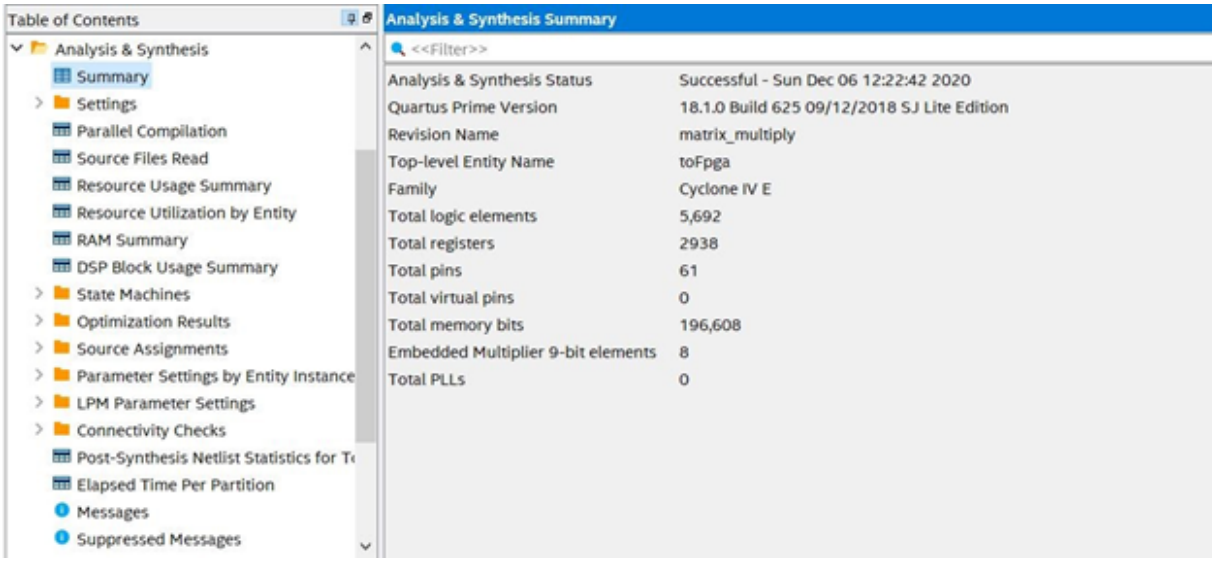


Figure 7.2: Analysis & Synthesis Summary

7.3 Filter Summary

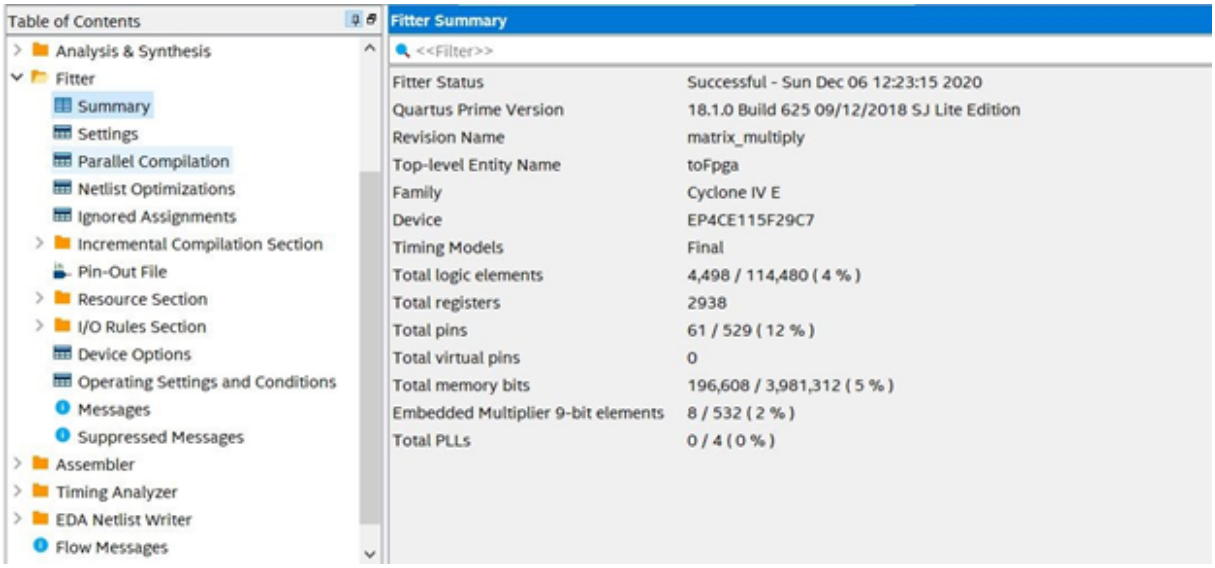


Figure 7.3: Filter Summary

7.4 Assembler Summary

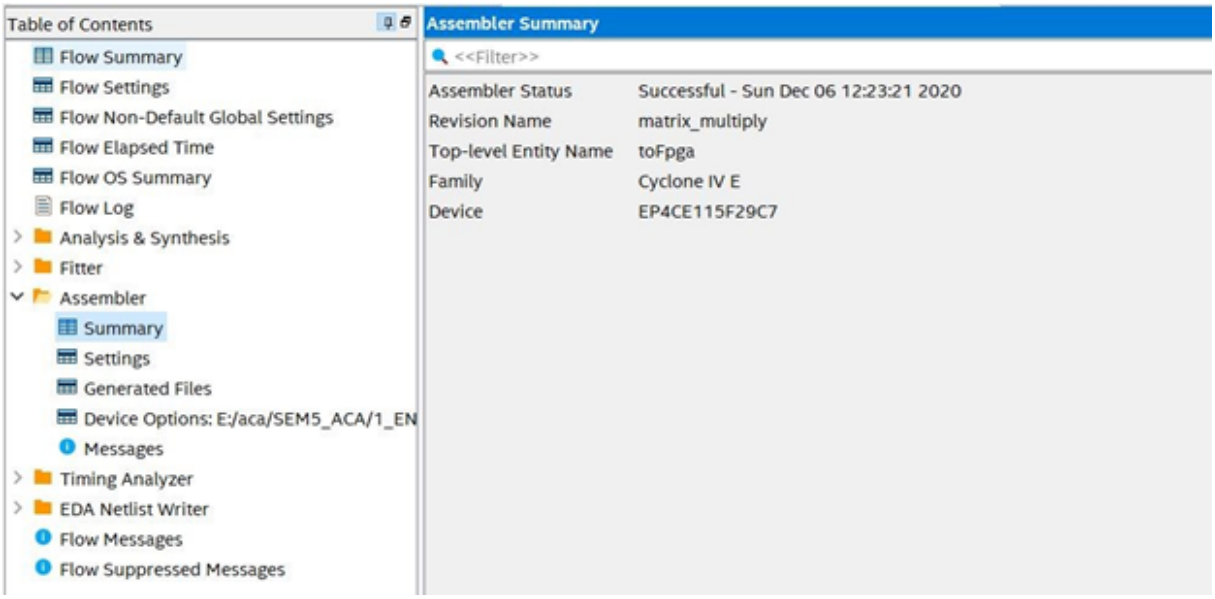


Figure 7.4: Assembler Summary

7.5 Timing Analyzer Summary

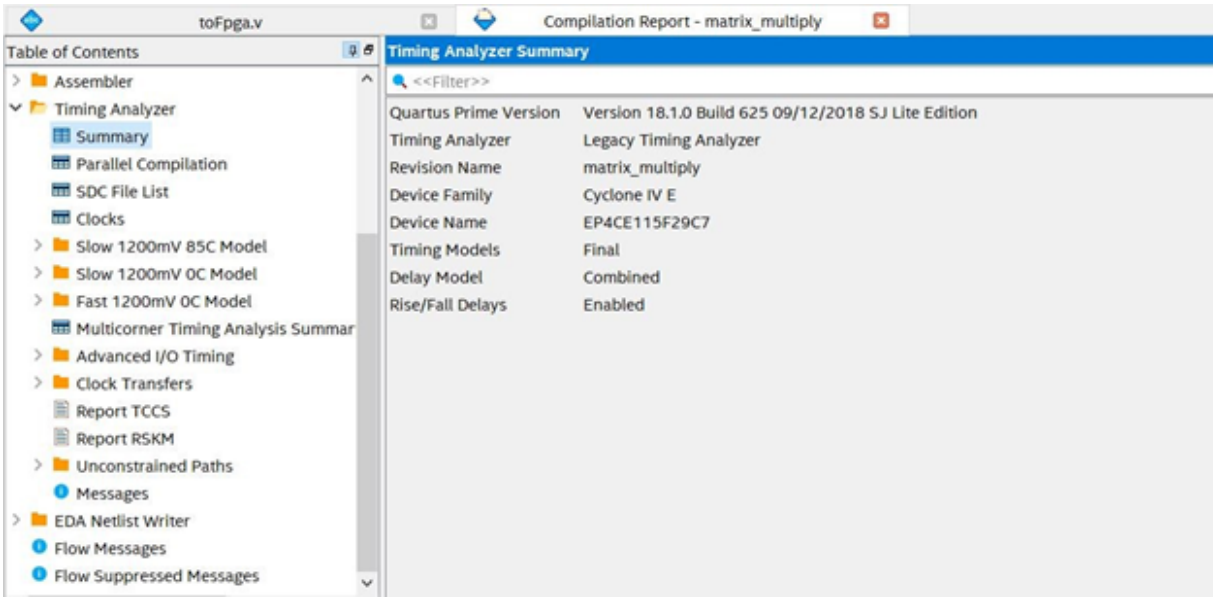


Figure 7.5: Timing Analyzer Summary

7.6 EDA Netlist Writer Summary

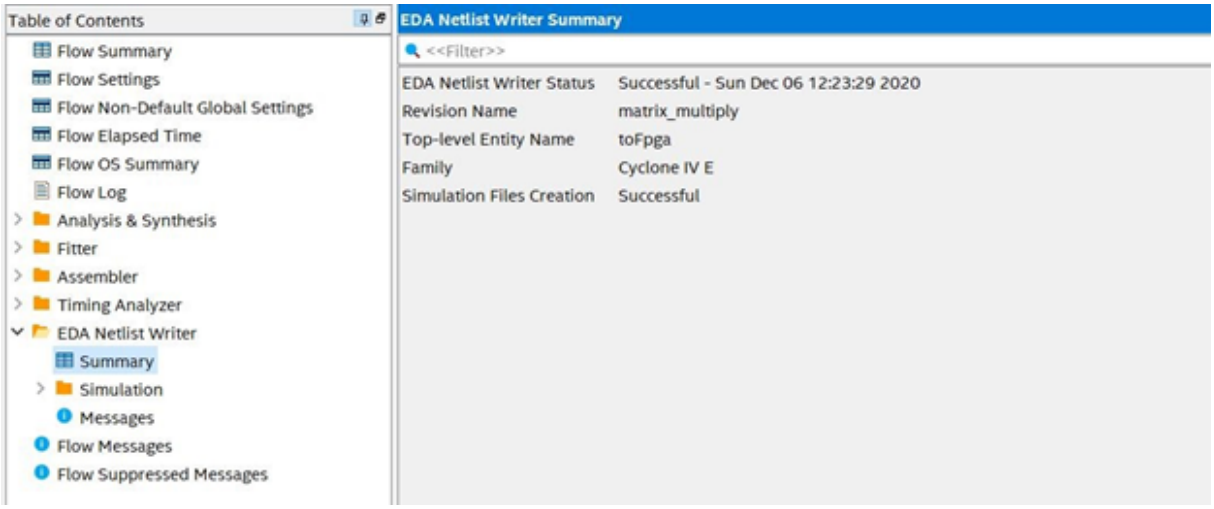


Figure 7.6: EDA Netlist Writer Summary

8 References

- FPGA PROTOTYPING BY VERILOG EXAMPLES – PONG P. CHU
- The Verilog® Hardware Description Language, Fifth Edition - Donald E. Thomas ECE Department Carnegie Mellon University Pittsburgh, PA
- Embedded Core Design with FPGAs – Zainalabedin Navabi, PH.D.
- Computer System Organization & Architecture – JOHN D. CARPINELLI
- Digital Design and Computer Architecture – David Money Harris & Sarah L Harris
- DE2_115_User_manual
- 1364-2001 - IEEE Standard Verilog® Hardware Description Language
- 1800-2009 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language
- RTL Modeling with SystemVerilog for Simulation and Synthesis Using SystemVerilog for ASIC and FPGA Design - Stuart Sutherland
- SystemVerilog for Verification, A Guide to Learning the Testbench Language Features, Third Edition - Chris spear & Greg Tumbush
- My First FPGA for Altera DE2- 115 Board.pdf
- Introduction to FPGA Design for Embedded Systems - https://www.coursera.org/learn/intro-fpga-design-embedded-systems
- FPGA Softcore Processors and IP Acquisition - https://www.coursera.org/learn/fpga-softcore-proccessors-ip
- In system memory content editor tutorial - https://www.youtube.com/watch?v=_VcVtFvJnuY&ab_channel=IntelFPGA

9 Appendix

9.1 Multi-core processor related SystemVerilog modules

9.1.1 Common details for modules - details.sv

```
package details;
localparam IR_WIDTH = 8;

typedef enum logic [2:0]{
    clr_alu = 3'd0,   //alu operations
    pass_alu = 3'd1,
    add_alu =  3'd2,
    sub_alu =  3'd3,
    mul_alu =  3'd4,
    inc_alu =  3'd5,
    idle_alu = 3'd6
} alu_op_t;

typedef enum logic [3:0]{
    DMem_bus = 4'b0,   //multiplexer
    R_bus = 4'd1,
    IR_bus = 4'd2,
    RL_bus = 4'd3,
    RC_bus = 4'd4,
    RP_bus = 4'd5,
    RQ_bus = 4'd6,
    R1_bus = 4'd7,
    AC_bus = 4'd8,
    idle_bus = 4'd9
} bus_in_sel_t;

typedef enum logic [IR_WIDTH-1:0]{
    NOP       =   8'd0,
    ENDOP       = 8'd1,
    CLAC        = 8'd2,
    LDIAC       = 8'd3,
    LDAC        = 8'd4,
    STR         = 8'd5,
    STIR        = 8'd6,
    JUMP        = 8'd7,
    JMPNZ       = 8'd8,
    JMPZ        = 8'd9,
    MUL         =   8'd10,
    ADD         =   8'd11,
    SUB         =   8'd12,
    INCAC     =   8'd13,
    MV_RL_AC    = {4'd1,4'd15},
    MV_RP_AC    = {4'd2,4'd15},
    MV_RQ_AC    = {4'd3,4'd15},
    MV_RC_AC    = {4'd4,4'd15},
    MV_R_AC     = {4'd5,4'd15},
    MV_R1_AC    = {4'd6,4'd15},
    MV_AC_RP    = {4'd7,4'd15},
    MV_AC_RQ    = {4'd8,4'd15},
    MV_AC_RL    = {4'd9,4'd15}
} ISA_t;

typedef enum logic [3:0] {
    no_inc = 4'b0000,
    PC_inc = 4'b1000,
    RC_inc = 4'b0100,
    RP_inc = 4'b0010,
    RQ_inc = 4'b0001,
    RC_RP_RQ_inc = 4'b0111
} inc_reg_t;

typedef enum logic [9:0] {
    no_wrEn = 10'b0000000000,
```

```
    AR_wrEn = 10'b1000000000,
    R_wrEn  = 10'b0100000000,
    PC_wrEn = 10'b0010000000,
    IR_wrEn = 10'b0001000000,
    RL_wrEn = 10'b0000100000,
    RC_wrEn = 10'b0000010000,
    RP_wrEn = 10'b0000001000,
    RQ_wrEn = 10'b0000000100,
    R1_wrEn = 10'b0000000010,
    AC_wrEn = 10'b0000000001
} wrEnReg_t;


endpackage
```

9.1.2 Top module - toFpga.sv

```
module toFpga (
    input logic CLOCK_50,
    input logic [1:0]KEY,
    output logic [1:0]LEDG,
    output logic [6:0]HEX0,HEX1,HEX2, HEX3, HEX4,HEX5, HEX6, HEX7,  // for checking perposes
    input logic UART_RXD,
  output logic UART_TXD
);

localparam CORE_COUNT = 4;
localparam REG_WIDTH = 12;
localparam DATA_MEM_WIDTH = CORE_COUNT * REG_WIDTH;
localparam INS_WIDTH = 8;
localparam INS_MEM_DEPTH = 256;
localparam DATA_MEM_DEPTH = 4096;
localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);

localparam BAUD_RATE = 115200;
localparam UART_WIDTH = 8;

////// logic related to data memory ///////////
logic [DATA_MEM_WIDTH-1:0]DataMemOut, DataMemIn, processor_DataOut, uart_DataOut;
logic [DATA_MEM_ADDR_WIDTH-1:0] processor_dataMemAddr;
logic [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr, uart_dataMemAddr;

////// logic related to instruction memory ////////////
logic [INS_WIDTH-1:0]InsMemOut, InsMemIn;
logic [INS_MEM_ADDR_WIDTH-1:0] processor_InsMemAddr;
logic [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,uart_InsMemAddr;

////// other logics ////////////
logic dataMemWrEn, processor_DataMemWrEn, uart_dataMemWrEn;
logic uart_InsMemWrEn;
logic clk, rstN, startN, processStart;
logic processDone, processor_ready;

/// logic related to communication unit ///////////////
logic rxByteReady, txByteReady;
logic rx_new_byte_indicate, new_ins_byte_indicate, new_data_byte_indicate;
logic txByteStart;

logic [7:0]byteForTx, byteFromRx;

logic uart_DataMem_transmitted, uart_DataMem_received, uart_InsMem_received;
logic uart_dmem_start_transmitN;
//////////////////////////////////////
assign LEDG[1] = processDone;
assign LEDG[0] = processor_ready;



///////////////// state change logic

typedef enum logic [2:0] {
    idle = 3'd0,
    uart_receive_Imem = 3'd1,
    uart_receive_dmem = 3'd2,
    process_ready = 3'd3,
    process_exicute = 3'd4,
```

```
    uart_transmit_dmem = 3'd5,
    finish = 3'd6
} state_t;

state_t currentState, nextState;

always @(posedge clk) begin
    if (~rstN) begin
        currentState <= idle;
    end
    else begin
        currentState <= nextState;
    end
end

always_comb begin
    nextState = currentState;

    case (currentState)
        idle: begin     // start state
            if (~startN) begin
                nextState = uart_receive_Imem;
            end
        end

        uart_receive_Imem: begin  // send the instructions (machine_code) from PC through UART
            if (uart_InsMem_received) begin
                nextState = uart_receive_dmem;
            end
        end

        uart_receive_dmem: begin  //send the data memory values from PC through UART
            if (uart_DataMem_received) begin
                nextState = process_exicute;
            end
        end

        process_exicute: begin  // processor exicute program (matrix multiplication)
            if (processDone) begin
                nextState = uart_transmit_dmem;
            end
        end

        uart_transmit_dmem: begin   // send the answer matrix to PC through UART
            if (uart_DataMem_transmitted) begin
                nextState = finish;
            end
        end

        finish: begin  //End of the process

        end

        default : nextState = idle;

    endcase
end

assign clk = CLOCK_50;
assign rstN = KEY[0];
assign startN = KEY[1];
assign processStart = ((currentState == uart_receive_dmem) && (uart_DataMem_received))? 1'b1: 1'b0;
assign uart_dmem_start_transmitN = ((currentState == process_exicute) && (processDone))? 1'b0: 1'b1;

assign dataMemWrEn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )?
    uart_dataMemWrEn:
                    (currentState == process_exicute)? processor_DataMemWrEn:
                    1'b0;

assign dataMemAddr = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )?
    uart_dataMemAddr:
                    (currentState == process_exicute)? processor_dataMemAddr:
                    {DATA_MEM_ADDR_WIDTH{1'b0}};

assign DataMemIn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )?
    uart_DataOut:
                    (currentState == process_exicute)? processor_DataOut:
                    {DATA_MEM_WIDTH{1'b0}};

```

```
assign insMemAddr = (currentState == uart_receive_Imem)? uart_InsMemAddr:
                    (currentState == process_exicute)? processor_InsMemAddr:
                    {INS_MEM_ADDR_WIDTH{1'b0}};

assign new_ins_byte_indicate = (currentState == uart_receive_Imem)? rx_new_byte_indicate: 1'b0;
assign new_data_byte_indicate = (currentState == uart_receive_dmem)? rx_new_byte_indicate: 1'b0;


//////////////////////////////////////////////////////////////////////
multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
                    .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
                    multi_core_processor(.clk,.rstN,.start(processStart), .ProcessorDataIn(DataMemOut),
                    .InsMemOut, .ProcessorDataOut(processor_DataOut), .insMemAddr(processor_InsMemAddr),
                    .dataMemAddr(processor_dataMemAddr), .DataMemWrEn(processor_DataMemWrEn),
                    .done(processDone), .ready(processor_ready));


 RAM #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH)) IM(.clk(CLOCK_50), .wrEn(uart_InsMemWrEn),
             .dataIn(InsMemIn), .addr(insMemAddr), .dataOut(InsMemOut));

 RAM #(.WIDTH(DATA_MEM_WIDTH), .DEPTH(DATA_MEM_DEPTH)) DM(.clk(CLOCK_50), .wrEn(dataMemWrEn),
             .dataIn(DataMemIn), .addr(dataMemAddr), .dataOut(DataMemOut));

////////////// communication system

mem_communication_interface #(.MEM_WORD_LENGTH(DATA_MEM_WIDTH), .MEM_DEPTH(DATA_MEM_DEPTH), .UART_WIDTH(
    UART_WIDTH))
             dMem_com_interface(
                 .clk, .rstN, .txStartN(uart_dmem_start_transmitN),
                 .mem_transmitted(uart_DataMem_transmitted), .mem_received(uart_DataMem_received),
                 /////inputs outputs with memory
                 .dataFromMem(DataMemOut), .memWrEn(uart_dataMemWrEn) , .mem_address(uart_dataMemAddr),
                 .dataToMem(uart_DataOut),
                 ////////inputs outputs with uart system
                 .rxByteReady, .rx_new_byte_indicate(new_data_byte_indicate), .txByteReady,
                 .byteFromRx, .txByteStart, .byteForTx,
                 ///////// select start end mem addresses of tx and rx
                 .tx_start_addr_in(uartMemory[1]), .tx_end_addr_in(uartMemory[2]),
                 .rx_end_addr_in(uartMemory[0]), .toggle_addr_range(1'b1)
             );

mem_communication_interface #(.MEM_WORD_LENGTH(INS_WIDTH), .MEM_DEPTH(INS_MEM_DEPTH), .UART_WIDTH(UART_WIDTH
    ))
             insMem_com_interface(
                 .clk, .rstN, .txStartN(1'b1),
                 .mem_transmitted(), .mem_received(uart_InsMem_received),
                 /////inputs outputs with memory
                 .dataFromMem(), .memWrEn(uart_InsMemWrEn), .mem_address(uart_InsMemAddr), .dataToMem(
    InsMemIn),
                 ////////inputs outputs with uart system
                 .rxByteReady, .rx_new_byte_indicate(new_ins_byte_indicate), .txByteReady, .byteFromRx,
                 .txByteStart(), .byteForTx(),
                 ////////// //select start end mem addresses of tx and rx
                 .tx_start_addr_in(), .tx_end_addr_in(), .rx_end_addr_in(),
                 .toggle_addr_range(1'b0)
             );

uart_system #(.DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE)) uart_system (
    .clk, .rstN,.txByteStart,.rx(UART_RXD), .byteForTx, .tx(UART_TXD), .tx_ready(txByteReady),
    .rx_ready(rxByteReady), .rx_new_byte_indicate, .byteFromRx
);

////////////// memory addesses identification to find end of receiving dataMem and answer matrix transmission
    start & end addresses //////////
localparam  Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
             R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
             R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);

logic [REG_WIDTH-1:0]uartMemory[2:0]; //0- end address of Q, 1- start address of R, 2- end address of R

always_ff @(posedge clk) begin
    if (~rstN) begin
        uartMemory <= '{default:'0};
    end
    else if (uart_dataMemWrEn & (currentState == uart_receive_dmem)) begin
        if (uart_dataMemAddr == Q_end_addr_location)
            uartMemory[0] <= uart_DataOut[REG_WIDTH-1:0];
        else if (uart_dataMemAddr == R_start_addr_location)
            uartMemory[1] <= uart_DataOut[REG_WIDTH-1:0];
```

```
        else if (uart_dataMemAddr == R_end_addr_location)
            uartMemory[2] <= uart_DataOut[REG_WIDTH-1:0];
    end
end

//////////////to count the time taken to the process
logic [25:0]currentTime;
timeCounter TC(.clk(CLOCK_50), .rstN(rstN), .start(processStart), .stop(processDone),
                .timeDuration(currentTime));

logic [6:0]hex_display_value[7:0];
assign '{HEX7,HEX6,HEX5, HEX4, HEX3,HEX2, HEX1, HEX0} = hex_display_value;

hex_display HD(.clk, .rstN, .state(currentState),
            .start_timeValue_convetion(~uart_DataMem_transmitted), .binary_time_value(currentTime),
            .hex_display_value
            );

endmodule :toFpga
```

9.1.3 Memory

Module - RAM.sv

```
module RAM
#(
    parameter WIDTH = 12,
    parameter DEPTH = 256,
    parameter ADDR_WIDTH = $clog2(DEPTH)
)
(
    input logic clk, wrEn,
    input logic [WIDTH-1:0]dataIn,
    input logic [ADDR_WIDTH-1:0]addr,
    output logic [WIDTH-1:0]dataOut
);

logic [ADDR_WIDTH-1:0]addr_reg;

logic [WIDTH-1:0]memory[0:DEPTH-1] ;

always_ff @(posedge clk) begin
    addr_reg <= addr;
    if (wrEn) begin
        memory[addr] <= dataIn;   // write requires only 1 clk cyle.
    end
end
assign dataOut = memory[addr_reg];   // address is registered. Need 2 clk cycles to read.

endmodule : RAM
```

Testbench - RAM_tb.sv

```
module RAM_tb();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 20;
logic clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WIDTH = 12;
localparam DEPTH = 8;
localparam ADDR_WIDTH = $clog2(DEPTH);

logic wrEn;
typedef logic [WIDTH-1:0]data_t;
```

```
data_t dataIn, dataOut;
typedef logic [ADDR_WIDTH-1:0]addr_t;
addr_t addr;

RAM #(.WIDTH(WIDTH), .DEPTH(DEPTH), .ADDR_WIDTH(ADDR_WIDTH)) dut(.*);

initial begin
    @(posedge clk);
    #(CLK_PERIOD*4/5);
    wrEn <= 1'b1;
    addr <= 3;
    dataIn <= 100;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    wrEn <= 1'b0;
    addr <= 20;
    dataIn <=30;

    repeat (20) begin
        @(posedge clk);
        #(CLK_PERIOD*4/5);    // to give data litte bit earlier to the posedge of the clk

        dataIn = $random();
        addr = $random();
        wrEn = $random();
    end

    $stop;
end

typedef logic [WIDTH-1:0]test_memory_t[0:DEPTH-1];
test_memory_t test_memory;

function test_memory_t update (test_memory_t test_memory,
                    logic wrEn, addr_t addr, data_t dataIn, data_t dataOut);

  if (wrEn) begin
      test_memory[addr] = dataIn;
  end

  return test_memory;

endfunction

function void check (test_memory_t test_memory, data_t dataOut, addr_t addr);

    if (dataOut != test_memory[addr]) begin
        $display("dataOut = %p, memory_value = %p, addr = %p", dataOut, test_memory[addr], addr);
    end

endfunction

initial begin
    forever begin
        @(posedge clk);
        test_memory = update (test_memory, wrEn, addr, dataIn, dataOut);

        @(negedge clk);
        #(CLK_PERIOD*1/10);
        check (test_memory, dataOut, addr);
    end
end

endmodule:RAM_tb
```

9.1.4 Multi-core processor

Module - multi_core_processor.sv

```
module multi_core_processor #(
    parameter REG_WIDTH = 12,
    parameter INS_WIDTH = 8,
    parameter CORE_COUNT = 4,
    parameter DATA_MEM_ADDR_WIDTH = 12,
```

```
    parameter INS_MEM_ADDR_WIDTH = 8
)
(
    input logic clk,rstN,start,
    input logic [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataIn,
    input logic [INS_WIDTH-1:0]InsMemOut,
    output logic [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataOut,
    output logic [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,
    output logic [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr,
    output logic DataMemWrEn, done,ready
);

logic core_DataMemWrEn[0:CORE_COUNT-1];
logic core_done[0:CORE_COUNT-1];
logic core_ready[0:CORE_COUNT-1];
logic [DATA_MEM_ADDR_WIDTH-1:0] core_dataMemAddr[0:CORE_COUNT-1];
logic [INS_MEM_ADDR_WIDTH-1:0] core_InsMemAddr[0:CORE_COUNT-1];

genvar i;
generate
    for (i=0;i<CORE_COUNT; i=i+1) begin:core
        processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH),
                    .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH)) CPU
                    (.clk, .rstN, .start, .ProcessorDataIn(ProcessorDataIn[REG_WIDTH*i+:REG_WIDTH]),
                    .InsMemOut, .dataMemAddr(core_dataMemAddr[i]),
                    .ProcessorDataOut(ProcessorDataOut[REG_WIDTH*i+:REG_WIDTH]),
                    .insMemAddr(core_InsMemAddr[i]), .DataMemWrEn(core_DataMemWrEn[i]),
                    .done(core_done[i]), .ready(core_ready[i]) );
    end
endgenerate

assign dataMemAddr = core_dataMemAddr[0];
assign DataMemWrEn = core_DataMemWrEn[0];
assign insMemAddr = core_InsMemAddr[0];
assign ready = core_ready[0];
assign done = core_done[0];

endmodule : multi_core_processor
```

Testbench - multi_core_processor_tb.sv

```
class RAM_class #(parameter WIDTH, DEPTH);

localparam ADDR_WIDTH = $clog2(DEPTH);

typedef logic [WIDTH-1:0]  data_t;
typedef logic [ADDR_WIDTH-1:0] addr_t;

logic [WIDTH-1:0] RAM [0:DEPTH-1];
// logic [ADDR_WIDTH-1:0] addr;


function void initialize_full_memory(input logic [WIDTH-1:0]initial_vals[0:DEPTH-1]);
    RAM = initial_vals;
endfunction

function void initialize_singal_memory_location(input data_t value, input addr_t addr);
    RAM[addr] = value;
endfunction

task Read_memory(input addr_t addr, output data_t value, ref logic clk);        // if needed read time delay
     can be added
    value = RAM[addr];
endtask

task Write_memory(input addr_t addr, input data_t data, input logic wrEn, ref logic clk); // if needed write
     time delay can be added
    @(posedge clk);
    if (wrEn) begin
        RAM[addr] = data;
    end
endtask

function void display_RAM();
    foreach(this.RAM[i])
        $display(i,this.RAM[i]);

endfunction
```

```
36
37 function data_t get_value(input addr_t addr);
38     return RAM[addr];
39 endfunction
40
41 endclass
42
43
44 module multi_core_processor_tb import details::*;();
45
46 timeunit 1ns;
47 timeprecision 1ps;
48 localparam CLK_PERIOD = 20;
49 logic clk;
50 initial begin
51     clk <= 0;
52     forever begin
53         #(CLK_PERIOD/2);
54         clk <= ~clk;
55     end
56 end
57
58 localparam CORE_COUNT = 4;
59 localparam REG_WIDTH = 12;
60 localparam INS_WIDTH = 8;
61 localparam INS_MEM_DEPTH = 256;
62 localparam DATA_MEM_DEPTH = 4096;
63 localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
64 localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);
65 localparam DATA_MEM_WIDTH = REG_WIDTH*CORE_COUNT;
66
67 logic rstN,start;
68 logic [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataIn;
69 logic [INS_WIDTH-1:0]InsMemOut;
70 logic [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataOut;
71 logic [INS_MEM_ADDR_WIDTH-1:0]insMemAddr;
72 logic [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr;
73 logic DataMemWrEn, done,ready;
74
75 multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
76                        .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
    dut(.*);
77
78 ///// initialize instruction and data memory /////////
79 RAM_class #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH)) ins_mem = new;
80 RAM_class #(.WIDTH(DATA_MEM_WIDTH), .DEPTH(DATA_MEM_DEPTH)) data_mem = new;
81
82 logic [INS_WIDTH-1:0]temp_ins_mem[0:INS_MEM_DEPTH-1];
83 logic [DATA_MEM_WIDTH-1:0]temp_data_mem[0:DATA_MEM_DEPTH-1];
84
85 initial begin
86     $readmemb("../../9_ins_mem_tb.txt", temp_ins_mem);
87     $readmemb("../../4_data_mem_tb.txt", temp_data_mem);
88     ins_mem.initialize_full_memory(temp_ins_mem);
89     data_mem.initialize_full_memory(temp_data_mem);
90 end
91
92 initial begin
93     @(posedge clk);
94     rstN <= 1'b0;
95     start <= 1'b0;
96     @(posedge clk);
97     rstN <= 1'b1;
98     start <= 1'b1;
99 end
100
101 always_ff @(posedge clk) begin
102     ins_mem.Read_memory(.addr(insMemAddr), .value(InsMemOut), .clk(clk));
103 end
104
105 always_ff @(posedge clk) begin
106     data_mem.Read_memory(.addr(dataMemAddr), .value(ProcessorDataIn), .clk(clk));
107 end
108
109 always_ff @(posedge clk) begin
110     data_mem.Write_memory(.addr(dataMemAddr), .data(ProcessorDataOut), .wrEn(DataMemWrEn), .clk(clk));
111 end
112
113 ////////////// verification of the simulation correctness /////////
```

```
114
115 localparam  Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
116           R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
117           R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);
118 logic [REG_WIDTH-1:0] a, b, c, P_start_addr, Q_start_addr, R_start_addr, P_end_addr, Q_end_addr, R_end_addr;
119
120 logic [DATA_MEM_WIDTH-1:0] temp_data_mem_2[0:DATA_MEM_DEPTH-1];
121
122 always_ff @(posedge clk) begin
123     if (done) begin
124         a = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd0))[REG_WIDTH-1:0];
125         b = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd1))[REG_WIDTH-1:0];
126         c = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd2))[REG_WIDTH-1:0];
127         P_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd3))[REG_WIDTH-1:0];
128         Q_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd4))[REG_WIDTH-1:0];
129         R_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd5))[REG_WIDTH-1:0];
130         P_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd6))[REG_WIDTH-1:0];
131         Q_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd7))[REG_WIDTH-1:0];
132         R_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd8))[REG_WIDTH-1:0];
133
134
135         temp_data_mem_2 = data_mem.RAM;
136         $writememh("../../7_multiply_answer.txt", temp_data_mem_2, R_start_addr,R_end_addr); // write answer
     matrix to a file
137
138         $display("\nMatrix P\n");
139         print_matrix_P(data_mem.RAM, a,b,P_start_addr, P_end_addr, CORE_COUNT);
140
141         $display("\nMatrix Q\n");
142         print_matrix_Q(data_mem.RAM,b,c,Q_start_addr,Q_end_addr);
143
144         $display("\nMatrix R\n");
145         print_matrix_R(data_mem.RAM,a,c,R_start_addr,R_end_addr,CORE_COUNT);
146
147         repeat(5) @(posedge clk);  // end of the simulation
148         $stop;
149     end
150 end
151
152 function automatic void print_matrix_P(input logic [DATA_MEM_WIDTH-1:0]DMEM[0:DATA_MEM_DEPTH-1],
153                                        logic [REG_WIDTH-1:0]a,b,P_start_addr,P_end_addr, int CORE_COUNT);
154
155     int d = (a%CORE_COUNT == 0)? a/CORE_COUNT : a/CORE_COUNT+1;
156
157     for (int x=0;x<d;x++) begin
158         for (int y=CORE_COUNT;y>0;y--) begin
159             if ((x+1)*CORE_COUNT-y>= a) begin
160                 break;
161             end
162             for (int z=0;z<b;z++) begin
163                 logic [DATA_MEM_WIDTH-1:0]temp_1 = DMEM[(P_start_addr + x*b+z)];
164                 logic [REG_WIDTH-1:0]temp_2 = temp_1[(y*REG_WIDTH-1) -:REG_WIDTH];
165                 $write("%h ", temp_2);
166             end
167             $write("\n");
168         end
169     end
170 endfunction
171
172 function automatic void print_matrix_Q(input logic [DATA_MEM_WIDTH-1:0]DMEM[0:DATA_MEM_DEPTH-1],
173                                        logic [REG_WIDTH-1:0]b,c,Q_start_addr,Q_end_addr);
174
175     for (int i=Q_start_addr;i<Q_start_addr+b;i++) begin
176         for (int j=i;j<=Q_end_addr;j=j+b) begin
177             logic [DATA_MEM_WIDTH-1:0] temp_1 = DMEM[j];
178             $write("%h ", temp_1[REG_WIDTH-1:0]);
179         end
180         $write("\n");
181     end
182 endfunction
183
184 function automatic void print_matrix_R(input logic [DATA_MEM_WIDTH-1:0]DMEM[0:DATA_MEM_DEPTH-1],
185                                        logic [REG_WIDTH-1:0]a,c,R_start_addr,R_end_addr, int CORE_COUNT);
186
187     int d = (a%CORE_COUNT == 0)? a/CORE_COUNT : a/CORE_COUNT+1;
188
189     for (int x=0;x<d;x++) begin
190         for (int y=CORE_COUNT;y>0;y--) begin
191             if ((x+1)*CORE_COUNT-y>= a) begin
```

```
                break;
            end
            for (int z=0;z<c;z++) begin
                logic [DATA_MEM_WIDTH-1:0]temp_1 = DMEM[(R_start_addr + x*c+z)];
                logic [REG_WIDTH-1:0]temp_2 = temp_1[(y*REG_WIDTH-1) -:REG_WIDTH];
                $write("%h ", temp_2);
            end
            $write("\n");
        end
    end
endfunction


endmodule : multi_core_processor_tb
```

9.1.5 Single-core processor

Module - processor.sv

```
module processor import details::*;
#(
    parameter REG_WIDTH = 12,
    parameter INS_WIDTH = 8,
    parameter DATA_MEM_ADDR_WIDTH = 12,
    parameter INS_MEM_ADDR_WIDTH = 8
)
(
    input  logic clk,rstN,start,
    input  logic [REG_WIDTH-1:0]ProcessorDataIn,
    input  logic [INS_WIDTH-1:0]InsMemOut,
    output logic [REG_WIDTH-1:0]ProcessorDataOut,
    output logic [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr,
    output logic [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,
    output logic DataMemWrEn,
    output logic done,ready
);

logic [REG_WIDTH-1:0]alu_a, alu_b, alu_out;
alu_op_t select_alu_op;
bus_in_sel_t busSel;
logic [REG_WIDTH-1:0]busOut;
logic [INS_WIDTH-1:0]IRout;
logic [REG_WIDTH-1:0] Rout, RLout, RCout, RPout, RQout, R1out, ACout;
logic Zout, ZWrEn;
inc_reg_t incReg;    // {PC, RC, RP, RQ}
logic PC_inc, RC_inc, RP_inc, RQ_inc;
wrEnReg_t wrEnReg;   // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
logic AR_wrEn,R_wrEn,PC_wrEn,IR_wrEn,RL_wrEn,RC_wrEn,RP_wrEn,RQ_wrEn,R1_wrEn,AC_wrEn;

assign {PC_inc, RC_inc, RP_inc, RQ_inc} = incReg;
assign {AR_wrEn,R_wrEn,PC_wrEn,IR_wrEn,RL_wrEn,RC_wrEn,RP_wrEn,RQ_wrEn,R1_wrEn,AC_wrEn} = wrEnReg;

controlUnit CU(.clk, .rstN, .start, .Zout, .instruction(ISA_t'(IRout)), .aluOp(select_alu_op), .incReg,
                .wrEnReg, .busSel, .DataMemWrEn, .ZWrEn, .done, .ready);

alu #(.WIDTH(REG_WIDTH)) alu(.a(alu_a), .b(alu_b), .selectOp(select_alu_op), .c(alu_out));

multiplexer #(.WIDTH(REG_WIDTH), .IR_WIDTH(INS_WIDTH)) mux(.selectIn(busSel), .DMem(ProcessorDataIn),
            .R(Rout), .RL(RLout), .RC(RCout), .RP(RPout), .RQ(RQout), .R1(R1out),
            .AC(ACout), .IR(IRout), .busOut(busOut));

register #(.WIDTH(REG_WIDTH)) AR(.dataIn(busOut), .wrEn(AR_wrEn), .rstN(rstN), .clk(clk),
            .dataOut(dataMemAddr));

register #(.WIDTH(REG_WIDTH)) R(.dataIn(busOut), .wrEn(R_wrEn), .rstN(rstN), .clk(clk), .dataOut(Rout));

incRegister #(.WIDTH(INS_WIDTH)) PC(.dataIn(IRout), .wrEn(PC_wrEn), .rstN(rstN), .clk(clk), .incEn(PC_inc),
            .dataOut(insMemAddr));

register #(.WIDTH(INS_WIDTH)) IR(.dataIn(InsMemOut), .wrEn(IR_wrEn), .rstN(rstN), .clk(clk), .dataOut(IRout)
    );

register #(.WIDTH(REG_WIDTH)) RL(.dataIn(busOut), .wrEn(RL_wrEn), .rstN(rstN), .clk(clk), .dataOut(RLout));

incRegister #(.WIDTH(REG_WIDTH)) RC(.dataIn(busOut), .wrEn(RC_wrEn), .rstN(rstN), .clk(clk), .incEn(RC_inc),
```

```
            .dataOut(RCout));

incRegister #(.WIDTH(REG_WIDTH)) RP(.dataIn(busOut), .wrEn(RP_wrEn), .rstN(rstN), .clk(clk), .incEn(RP_inc),
            .dataOut(RPout));

incRegister #(.WIDTH(REG_WIDTH)) RQ(.dataIn(busOut), .wrEn(RQ_wrEn), .rstN(rstN), .clk(clk), .incEn(RQ_inc),
            .dataOut(RQout));

register #(.WIDTH(REG_WIDTH)) R1(.dataIn(busOut), .wrEn(R1_wrEn), .rstN(rstN), .clk(clk), .dataOut(R1out));

register #(.WIDTH(REG_WIDTH)) AC(.dataIn(alu_out), .wrEn(AC_wrEn), .rstN(rstN), .clk(clk), .dataOut(ACout));

zReg #(.WIDTH(REG_WIDTH)) Z(.dataIn(alu_out), .clk(clk), .rstN(rstN), .wrEn(ZWrEn), .Zout(Zout));

assign ProcessorDataOut = Rout;

assign alu_a = ACout;
assign alu_b = busOut;

endmodule:processor
```

Testbench - processor_tb.sv

```
class RAM_class #(parameter WIDTH, DEPTH);

localparam ADDR_WIDTH = $clog2(DEPTH);

typedef logic [WIDTH-1:0]  data_t;
typedef logic [ADDR_WIDTH-1:0] addr_t;

logic [WIDTH-1:0] RAM [0:DEPTH-1];
// logic [ADDR_WIDTH-1:0] addr;


function void initialize_full_memory(input logic [WIDTH-1:0]initial_vals[0:DEPTH-1]);
    RAM = initial_vals;
endfunction

function void initialize_singal_memory_location(input data_t value, input addr_t addr);
    RAM[addr] = value;
endfunction

task Read_memory(input addr_t addr, output data_t value, ref logic clk);
    // if needed read time delay can be added here
    value = RAM[addr];
endtask

task Write_memory(input addr_t addr, input data_t data, input logic wrEn, ref logic clk);
    // if needed write time delay can be added here
    @(posedge clk);
    if (wrEn) begin
        RAM[addr] = data;
    end
endtask

function void display_RAM();
    foreach(this.RAM[i])
        $display(i,this.RAM[i]);

endfunction

function data_t get_value(input addr_t addr);
    return RAM[addr];
endfunction

endclass


module processor_tb import details::*;();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 20;
logic clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
```

```
        clk <= ~clk;
    end
end

localparam REG_WIDTH = 12;
localparam INS_WIDTH = 8;
localparam INS_MEM_DEPTH = 256;
localparam DATA_MEM_DEPTH = 4096;
localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);

logic rstN,start;
logic [REG_WIDTH-1:0]ProcessorDataIn;
logic [INS_WIDTH-1:0]InsMemOut;
logic  [REG_WIDTH-1:0]dataMemAddr,ProcessorDataOut;
logic  [INS_WIDTH-1:0]insMemAddr;
logic  DataMemWrEn;
logic  done,ready;

processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH),
            .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH))dut(.*);

///// initialize instruction and data memory /////////
RAM_class #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH)) ins_mem = new;
RAM_class #(.WIDTH(REG_WIDTH), .DEPTH(DATA_MEM_DEPTH)) data_mem = new;

logic [INS_WIDTH-1:0]temp_ins_mem[0:INS_MEM_DEPTH-1];
logic [REG_WIDTH-1:0]temp_data_mem[0:DATA_MEM_DEPTH-1];

initial begin
    $readmemb("../../9_ins_mem_tb.txt", temp_ins_mem);
    $readmemb("../../4_data_mem_tb.txt", temp_data_mem);
    ins_mem.initialize_full_memory(temp_ins_mem);
    data_mem.initialize_full_memory(temp_data_mem);
end


initial begin
    @(posedge clk);
    rstN <= 1'b0;
    start <= 1'b0;
    @(posedge clk);
    rstN <= 1'b1;
    start <= 1'b1;
end

always_ff @(posedge clk) begin
    ins_mem.Read_memory(.addr(insMemAddr), .value(InsMemOut), .clk(clk));
end

always_ff @(posedge clk) begin
    data_mem.Read_memory(.addr(dataMemAddr), .value(ProcessorDataIn), .clk(clk));
end

always_ff @(posedge clk) begin
    data_mem.Write_memory(.addr(dataMemAddr), .data(ProcessorDataOut), .wrEn(DataMemWrEn), .clk(clk));
end


////////////// verification of the simulation correctness /////////

localparam  Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
            R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
            R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);
logic [REG_WIDTH-1:0] a, b, c, P_start_addr, Q_start_addr, R_start_addr, P_end_addr, Q_end_addr, R_end_addr;

logic [REG_WIDTH-1:0] temp_data_mem_2[0:DATA_MEM_DEPTH-1];

always_ff @(posedge clk) begin
    if (done) begin
        a = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd0));
        b = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd1));
        c = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd2));
        P_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd3));
        Q_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd4));
        R_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd5));
        P_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd6));
        Q_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd7));
        R_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd8));
```

```
135
136
137         temp_data_mem_2 = data_mem.RAM;
138         $writememh("../../7_multiply_answer.txt", temp_data_mem_2, R_start_addr,R_end_addr); // write answer
        matrix to a file
139
140         $display("\nMatrix P\n");
141         print_matrix_P(data_mem.RAM, a,b,P_start_addr, P_end_addr);
142
143         $display("\nMatrix Q\n");
144         print_matrix_Q(data_mem.RAM,b,c,Q_start_addr,Q_end_addr);
145
146         $display("\nMatrix R\n");
147         print_matrix_R(data_mem.RAM,a,c,R_start_addr,R_end_addr);
148
149         $stop;
150     end
151 end
152
153 function void print_matrix_P(input logic [REG_WIDTH-1:0]DMEM[0:DATA_MEM_DEPTH-1],
154                              logic [REG_WIDTH-1:0]a,b,P_start_addr,P_end_addr);
155
156     for (int i=P_start_addr;i<P_end_addr;i=i+b) begin
157         for (int j=i;j<i+b;j++) begin
158             $write("%h ", DMEM[j]);
159         end
160         $write("\n");
161     end
162 endfunction
163
164 function void print_matrix_Q(input logic [REG_WIDTH-1:0]DMEM[0:DATA_MEM_DEPTH-1],
165                              logic [REG_WIDTH-1:0]b,c,Q_start_addr,Q_end_addr);
166
167     for (int i=Q_start_addr;i<Q_start_addr+b;i++) begin
168         for (int j=i;j<=Q_end_addr;j=j+b) begin
169             $write("%h ", DMEM[j]);
170         end
171         $write("\n");
172     end
173 endfunction
174
175 function void print_matrix_R(input logic [REG_WIDTH-1:0]DMEM[0:DATA_MEM_DEPTH-1],
176                              logic [REG_WIDTH-1:0]a,c,R_start_addr,R_end_addr);
177
178     for (int i=R_start_addr;i<R_end_addr;i=i+c) begin
179         for (int j=i;j<i+c;j++) begin
180             $write("%h ", DMEM[j]);
181         end
182         $write("\n");
183     end
184 endfunction
185
186 endmodule : processor_tb
```

9.1.6 Control Unit

Module - controlUnit.sv

```
1  module controlUnit import details::*;
2  #(
3      parameter IR_WIDTH = 8
4  )
5  (
6      input  logic clk,rstN,start,Zout,
7      input  ISA_t instruction,
8      output alu_op_t aluOp,
9      output inc_reg_t incReg,    // {PC, RC, RP, RQ}
10     output wrEnReg_t wrEnReg,   // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
11     output bus_in_sel_t busSel,
12     output logic DataMemWrEn,ZWrEn,
13     output logic done,ready
14 );
15
16 typedef enum logic [5:0] {
17     IDLE = 6'd0,  //states
```

```
18
19      NOP1 = 6'd1,
20
21      ENDOP1 = 6'd2,
22
23      CLAC1 = 6'd3,
24
25      FETCH_DELAY1 = 6'd37,          ///// (extra_delay for memory read)
26      FETCH1 = 6'd4,
27      FETCH2 = 6'd5,
28
29      LDIAC_DELAY1 =  6'd38,          ///// (extra_delay for memory read)
30      LDIAC1 = 6'd6,
31      LDIAC2 = 6'd7,
32      LDIAC_DELAY2 = 6'd39,          ///// (extra_delay for memory read)
33      LDIAC3 = 6'd8,
34
35      LDAC1 = 6'd9,
36      LDAC_DELAY1 = 6'd40,          ///// (extra_delay for memory read)
37      LDAC2 = 6'd10,
38
39      STIR_DELAY1 = 6'd41,          ///// (extra_delay for memory read)
40      STIR1 = 6'd11,
41      STIR2 = 6'd12,
42      STIR_DELAY2 = 6'd42,          ///// (extra_delay for memory write)
43      STIR3 = 6'd13,
44
45      STR1 = 6'd14,
46      STR_DELAY1 = 6'd43,          ///// (extra_delay for memory write)
47      STR2 = 6'd15,
48
49      JUMP_DELAY1 = 6'd44,          ///// (extra_delay for memory read)
50      JUMP1 = 6'd16,
51      JUMP2 = 6'd17,
52
53      JMPNZY_DILAY1 = 6'd45,          ///// (extra_delay for memory read)
54      JMPNZY1 = 6'd18,
55      JMPNZY2 = 6'd19,
56      JMPNZN1 = 6'd20,
57
58      JMPZY_DELAY1 = 6'd46,          ///// (extra_delay for memory read)
59      JMPZY1 = 6'd21,
60      JMPZY2 = 6'd22,
61      JMPZN1 = 6'd23,
62
63      MUL1 = 6'd24,
64
65      ADD1 = 6'd25,
66
67      SUB1 = 6'd26,
68
69      INCAC1 = 6'd27,
70
71      MV_RL_AC1 = 6'd28,
72
73      MV_RP_AC1 = 6'd29,
74
75      MV_RQ_AC1 = 6'd30,
76
77      MV_RC_AC1 = 6'd31,
78
79      MV_R_AC1 = 6'd32,
80
81      MV_R1_AC1 = 6'd33,
82
83      MV_AC_RP1 = 6'd34,
84
85      MV_AC_RQ1 = 6'd35,
86
87      MV_AC_RL1 = 6'd36
88  } states_t;
89
90  states_t currentState, nextState;
91
92  always_ff @(posedge clk) begin
93      if (~rstN) begin
94          currentState <= IDLE;
95      end
96      else begin
```

```
        currentState <= nextState;
    end
end

////////////next state calculation
always_comb begin
    nextState = currentState;
    unique case(currentState)
        IDLE : begin
            if (start)
                nextState <= FETCH1;
            else
                nextState <= IDLE;
        end

        NOP1 : nextState <= FETCH1;

        ENDOP1 : nextState <= ENDOP1;

        CLAC1 : nextState <= FETCH1;

        FETCH_DELAY1 : nextState <= FETCH1;        // extra_delay for memory read
        FETCH1 : nextState <= FETCH2;
        FETCH2 : begin
            unique case(instruction)
                NOP: nextState <= NOP1;
                ENDOP: nextState <= ENDOP1;
                CLAC: nextState <= CLAC1;
                LDIAC: nextState <= LDIAC_DELAY1;
                LDAC: nextState <= LDAC1;
                STR: nextState <= STR1;
                STIR: nextState <= STIR_DELAY1;  // extra_delay for memory read
                JUMP: nextState <= JUMP_DELAY1;  // extra_delay for memory read
                JMPNZ: nextState <= (Zout == 0)? JMPNZY_DILAY1 : JMPNZN1; // extra_delay for memory read
                JMPZ: nextState <= (Zout == 1)? JMPZY_DELAY1 : JMPZN1;  // extra_delay for memory read
                MUL: nextState <= MUL1;
                ADD: nextState <= ADD1;
                SUB: nextState <= SUB1;
                INCAC: nextState <= INCAC1;
                MV_RL_AC: nextState <= MV_RL_AC1;
                MV_RP_AC: nextState <= MV_RP_AC1;
                MV_RQ_AC: nextState <= MV_RQ_AC1;
                MV_RC_AC: nextState <= MV_RC_AC1;
                MV_R_AC: nextState <= MV_R_AC1;
                MV_R1_AC: nextState <= MV_R1_AC1;
                MV_AC_RP: nextState <= MV_AC_RP1;
                MV_AC_RQ: nextState <= MV_AC_RQ1;
                MV_AC_RL: nextState <= MV_AC_RL1;
                default: nextState <= IDLE;
            endcase
        end

        LDIAC_DELAY1 : nextState <= LDIAC1;        // extra_delay for memory read
        LDIAC1 : nextState <= LDIAC2;
        LDIAC2 : nextState <= LDIAC_DELAY2;
        LDIAC_DELAY2 : nextState <= LDIAC3;       // extra_delay for memory read
        LDIAC3 : nextState <= FETCH_DELAY1;

        LDAC1 : nextState <= LDAC_DELAY1;
        LDAC_DELAY1 : nextState <= LDAC2;        // extra_delay for memory read
        LDAC2 : nextState <= FETCH_DELAY1;

        STIR_DELAY1 : nextState <= STIR1;       // extra_delay for memory read
        STIR1 : nextState <= STIR2;
        STIR2 : nextState <= STIR_DELAY2;
        STIR_DELAY2 : nextState <= STIR3;         // extra_delay for memory write
        STIR3 : nextState <= FETCH_DELAY1;

        STR1 : nextState <= STR_DELAY1;
        STR_DELAY1 : nextState <= STR2;         // extra_delay for memory write
        STR2 : nextState <= FETCH1;

        JUMP_DELAY1 : nextState <= JUMP1;        // extra_delay for memory read
        JUMP1 : nextState <= JUMP2;
        JUMP2 : nextState <= FETCH_DELAY1;

        JMPNZY_DILAY1 : nextState <= JMPNZY1;        // extra_delay for memory read
        JMPNZY1 : nextState <= JMPNZY2;
        JMPNZY2 : nextState <= FETCH_DELAY1;
```

```
        JMPNZN1 : nextState <= FETCH_DELAY1;

        JMPZY_DELAY1 : nextState <= JMPZY1;         // extra_delay for memory read
        JMPZY1 : nextState <= JMPZY2;
        JMPZY2 : nextState <= FETCH_DELAY1;
        JMPZN1 : nextState <= FETCH_DELAY1;

        MUL1 : nextState <= FETCH_DELAY1;

        ADD1 : nextState <= FETCH_DELAY1;

        SUB1 : nextState <= FETCH_DELAY1;

        INCAC1 : nextState <= FETCH_DELAY1;

        MV_RL_AC1 : nextState <= FETCH_DELAY1;

        MV_RP_AC1 : nextState <= FETCH_DELAY1;

        MV_RQ_AC1 : nextState <= FETCH_DELAY1;

        MV_RC_AC1 : nextState <= FETCH_DELAY1;

        MV_R_AC1 : nextState <= FETCH_DELAY1;

        MV_R1_AC1 : nextState <= FETCH_DELAY1;

        MV_AC_RP1 : nextState <= FETCH_DELAY1;

        MV_AC_RQ1 : nextState <= FETCH_DELAY1;

        MV_AC_RL1 : nextState <= FETCH_DELAY1;
        default  : nextState <= IDLE;
    endcase
end

//////////logic within a state
always_comb begin
    unique case(currentState)
        IDLE: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        NOP1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        ENDOP1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        CLAC1: begin
            aluOp <= clr_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        FETCH_DELAY1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
```

```
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        FETCH1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        FETCH2: begin
            aluOp <= idle_alu;
            incReg <= PC_inc;
            wrEnReg <= no_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        LDIAC_DELAY1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        LDIAC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        LDIAC2: begin
            aluOp <= idle_alu;
            incReg <= PC_inc;
            wrEnReg <= AR_wrEn;
            busSel <= IR_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        LDIAC_DELAY2: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        LDIAC3: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        LDAC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= AR_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        LDAC_DELAY1: begin
            aluOp <= pass_alu;
```

```
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        LDAC2: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        STR1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= AR_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        STR_DELAY1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b1;
            ZWrEn <= 1'b0;
        end

        STR2: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b1;
            ZWrEn <= 1'b0;
        end

        STIR_DELAY1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        STIR1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        STIR2: begin
            aluOp <= idle_alu;
            incReg <= PC_inc;
            wrEnReg <= AR_wrEn;
            busSel <= IR_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        STIR_DELAY2: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b1;
            ZWrEn <= 1'b0;
        end

```

```
413         STIR3: begin
414             aluOp <= idle_alu;
415             incReg <= no_inc;
416             wrEnReg <= no_wrEn;
417             busSel <= idle_bus;
418             DataMemWrEn <= 1'b1;
419             ZWrEn <= 1'b0;
420         end
421
422         JUMP_DELAY1: begin
423             aluOp <= idle_alu;
424             incReg <= no_inc;
425             wrEnReg <= IR_wrEn;
426             busSel <= DMem_bus;
427             DataMemWrEn <= 1'b0;
428             ZWrEn <= 1'b0;
429         end
430
431         JUMP1: begin
432             aluOp <= idle_alu;
433             incReg <= no_inc;
434             wrEnReg <= IR_wrEn;
435             busSel <= DMem_bus;
436             DataMemWrEn <= 1'b0;
437             ZWrEn <= 1'b0;
438         end
439
440         JUMP2: begin
441             aluOp <= idle_alu;
442             incReg <= no_inc;
443             wrEnReg <= PC_wrEn;
444             busSel <= IR_bus;
445             DataMemWrEn <= 1'b0;
446             ZWrEn <= 1'b0;
447         end
448
449         JMPNZY_DILAY1: begin
450             aluOp <= idle_alu;
451             incReg <= no_inc;
452             wrEnReg <= IR_wrEn;
453             busSel <= DMem_bus;
454             DataMemWrEn <= 1'b0;
455             ZWrEn <= 1'b0;
456         end
457
458         JMPNZY1: begin
459             aluOp <= idle_alu;
460             incReg <= no_inc;
461             wrEnReg <= IR_wrEn;
462             busSel <= DMem_bus;
463             DataMemWrEn <= 1'b0;
464             ZWrEn <= 1'b0;
465         end
466
467         JMPNZY2: begin
468             aluOp <= idle_alu;
469             incReg <= no_inc;
470             wrEnReg <= PC_wrEn;
471             busSel <= IR_bus;
472             DataMemWrEn <= 1'b0;
473             ZWrEn <= 1'b0;
474         end
475
476         JMPNZN1: begin
477             aluOp <= idle_alu;
478             incReg <= PC_inc;
479             wrEnReg <= no_wrEn;
480             busSel <= idle_bus;
481             DataMemWrEn <= 1'b0;
482             ZWrEn <= 1'b0;
483         end
484
485         JMPZY_DELAY1: begin
486             aluOp <= idle_alu;
487             incReg <= no_inc;
488             wrEnReg <= IR_wrEn;
489             busSel <= DMem_bus;
490             DataMemWrEn <= 1'b0;
491             ZWrEn <= 1'b0;
```

```
492         end
493
494         JMPZY1: begin
495             aluOp <= idle_alu;
496             incReg <= no_inc;
497             wrEnReg <= IR_wrEn;
498             busSel <= DMem_bus;
499             DataMemWrEn <= 1'b0;
500             ZWrEn <= 1'b0;
501         end
502
503         JMPZY2: begin
504             aluOp <= idle_alu;
505             incReg <= no_inc;
506             wrEnReg <= PC_wrEn;
507             busSel <= IR_bus;
508             DataMemWrEn <= 1'b0;
509             ZWrEn <= 1'b0;
510         end
511
512         JMPZN1: begin
513             aluOp <= idle_alu;
514             incReg <= PC_inc;
515             wrEnReg <= no_wrEn;
516             busSel <= idle_bus;
517             DataMemWrEn <= 1'b0;
518             ZWrEn <= 1'b0;
519         end
520
521         MUL1: begin
522             aluOp <= mul_alu;
523             incReg <= RC_RP_RQ_inc;
524             wrEnReg <= AC_wrEn;
525             busSel <= R1_bus;
526             DataMemWrEn <= 1'b0;
527             ZWrEn <= 1'b1;
528         end
529
530         ADD1: begin
531             aluOp <= add_alu;
532             incReg <= no_inc;
533             wrEnReg <= AC_wrEn;
534             busSel <= R_bus;
535             DataMemWrEn <= 1'b0;
536             ZWrEn <= 1'b1;
537         end
538
539         SUB1: begin
540             aluOp <= sub_alu;
541             incReg <= no_inc;
542             wrEnReg <= AC_wrEn;
543             busSel <= RC_bus;
544             DataMemWrEn <= 1'b0;
545             ZWrEn <= 1'b1;
546         end
547
548         INCAC1: begin
549             aluOp <= inc_alu;
550             incReg <= no_inc;
551             wrEnReg <= AC_wrEn;
552             busSel <= idle_bus;
553             DataMemWrEn <= 1'b0;
554             ZWrEn <= 1'b1;
555         end
556
557         MV_RL_AC1: begin
558             aluOp <= idle_alu;
559             incReg <= no_inc;
560             wrEnReg <= RL_wrEn;
561             busSel <= AC_bus;
562             DataMemWrEn <= 1'b0;
563             ZWrEn <= 1'b0;
564         end
565
566         MV_RP_AC1: begin
567             aluOp <= idle_alu;
568             incReg <= no_inc;
569             wrEnReg <= RP_wrEn;
570             busSel <= AC_bus;
```

```
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        MV_RQ_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= RQ_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        MV_RC_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= RC_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        MV_R_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= R_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        MV_R1_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= R1_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end

        MV_AC_RP1: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= RP_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        MV_AC_RQ1: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= RQ_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        MV_AC_RL1: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= RL_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
        end

        default : begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
        end
    endcase
end

assign done = (currentState == ENDOP1)?1'b1:1'b0;
```

```
assign ready = (currentState == IDLE)? 1'b1:1'b0;

endmodule :controlUnit
```

Testbench - controlUnit_tb.sv

```
module controlUnit_tb import details::*;();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 10;
logic clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam IR_WIDTH = 8;

logic rstN,start,Zout;
ISA_t instruction;
alu_op_t aluOp;
inc_reg_t incReg;    // {PC, RC, RP, RQ}
wrEnReg_t wrEnReg;   // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
bus_in_sel_t busSel;
logic DataMemWrEn,ZWrEn;
logic done,ready;


localparam // time duration for each instruction to exicute
        NOP_time_duration       =   4,
        ENDOP_time_duration     =   4,
        CLAC_time_duration      =   4,
        LDIAC_time_duration     =   8,
        LDAC_time_duration      =   6,
        STR_time_duration       =   6,
        STIR_time_duration      =   8,
        JUMP_time_duration      =   6,
        JMPNZ_Y_time_duration   =   6,
        JMPNZ_N_time_duration   =   4,
        JMPZ_Y_time_duration    =   6,
        JMPZ_N_time_duration    =   4,
        MUL_time_duration     =   4,
        ADD_time_duration     =   4,
        SUB_time_duration     =   4,
        INCAC_time_duration     =   4,
        MV_RL_AC_time_duration  =   4,
        MV_RP_AC_time_duration  =   4,
        MV_RQ_AC_time_duration  =   4,
        MV_RC_AC_time_duration  =   4,
        MV_R_AC_time_duration   =   4,
        MV_R1_AC_time_duration  =   4,
        MV_AC_RP_time_duration  =   4,
        MV_AC_RQ_time_duration  =   4,
        MV_AC_RL_time_duration  =   4;


controlUnit #(.IR_WIDTH(IR_WIDTH)) dut(.*);

task automatic test_instruction(
    input int duration,
    input ISA_t ins,
    input logic Z_value,
    ref ISA_t instruction,
    ref logic Zout
);

    @(posedge clk);
    instruction = ins;
    Zout = Z_value;

    #(duration * CLK_PERIOD);
    // repeat(duration)@(posedge clk);

endtask
```

```
73
74
75 initial begin
76     @(posedge clk);
77     rstN <= 1'b0;
78     @(posedge clk);
79     rstN <= 1'b1;
80     start <= 1'b1;
81
82     ////// test NOP
83     test_instruction(.duration(NOP_time_duration), .ins(NOP), .Z_value(1'bX), .instruction(instruction), .
       Zout(Zout));
84
85     ///// test CLAC
86     test_instruction(.duration(CLAC_time_duration), .ins(CLAC),  .Z_value(1'bX), .instruction(instruction),
       .Zout(Zout));
87
88     ///// test LDIAC
89     test_instruction(.duration(LDIAC_time_duration), .ins(LDIAC), .Z_value(1'bX), .instruction(instruction),
        .Zout(Zout));
90
91     ////// test LDAC
92     test_instruction(.duration(LDAC_time_duration), .ins(LDAC), .Z_value(1'bX), .instruction(instruction), .
       Zout(Zout));
93
94     ////// test STR
95     test_instruction(.duration(STR_time_duration), .ins(STR), .Z_value(1'bX), .instruction(instruction), .
       Zout(Zout));
96
97     ////// test STIR
98     test_instruction(.duration(STIR_time_duration), .ins(STIR), .Z_value(1'bX), .instruction(instruction), .
       Zout(Zout));
99
100    ////// test JUMP
101    test_instruction(.duration(JUMP_time_duration), .ins(JUMP), .Z_value(1'bX), .instruction(instruction), .
       Zout(Zout));
102
103    ////// test JMPNZ_Y
104    test_instruction(.duration(JMPNZ_Y_time_duration), .ins(JMPNZ), .Z_value(1'b0), .instruction(instruction
       ), .Zout(Zout));
105
106    ////// test JMPNZ_N
107    test_instruction(.duration(JMPNZ_N_time_duration), .ins(JMPNZ), .Z_value(1'b1), .instruction(instruction
       ), .Zout(Zout));
108
109    ////// test JMPZ_Y
110    test_instruction(.duration(JMPZ_Y_time_duration), .ins(JMPZ), .Z_value(1'b1), .instruction(instruction),
        .Zout(Zout));
111
112    ////// test JMPZ_N
113    test_instruction(.duration(JMPZ_N_time_duration), .ins(JMPZ), .Z_value(1'b0), .instruction(instruction),
        .Zout(Zout));
114
115    ////// test MUL
116    test_instruction(.duration(MUL_time_duration), .ins(MUL), .Z_value(1'b0), .instruction(instruction), .
       Zout(Zout));
117
118    ////// test ADD
119    test_instruction(.duration(ADD_time_duration), .ins(ADD), .Z_value(1'b0), .instruction(instruction), .
       Zout(Zout));
120
121    ////// test SUB
122    test_instruction(.duration(SUB_time_duration), .ins(SUB), .Z_value(1'b0), .instruction(instruction), .
       Zout(Zout));
123
124    ////// test INCAC
125    test_instruction(.duration(INCAC_time_duration), .ins(INCAC), .Z_value(1'b0), .instruction(instruction),
        .Zout(Zout));
126
127    ////// test MV_RL_AC
128    test_instruction(.duration(MV_RL_AC_time_duration), .ins(MV_RL_AC), .Z_value(1'b0), .instruction(
       instruction), .Zout(Zout));
129
130    ////// test MV_RP_AC
131    test_instruction(.duration(MV_RP_AC_time_duration), .ins(MV_RP_AC), .Z_value(1'b0), .instruction(
       instruction), .Zout(Zout));
132
133    ////// test MV_RQ_AC
134    test_instruction(.duration(MV_RQ_AC_time_duration), .ins(MV_RQ_AC), .Z_value(1'b0), .instruction(
```

```
    instruction), .Zout(Zout));

    ////// test MV_RC_AC
    test_instruction(.duration(MV_RC_AC_time_duration), .ins(MV_RC_AC), .Z_value(1'b0), .instruction(
    instruction), .Zout(Zout));

    ////// test MV_R_AC
    test_instruction(.duration(MV_R_AC_time_duration), .ins(MV_R_AC), .Z_value(1'b0), .instruction(
    instruction), .Zout(Zout));

    ////// test MV_R1_AC
    test_instruction(.duration(MV_R1_AC_time_duration), .ins(MV_R1_AC), .Z_value(1'b0), .instruction(
    instruction), .Zout(Zout));

    ////// test MV_AC_RP
    test_instruction(.duration(MV_AC_RP_time_duration), .ins(MV_AC_RP), .Z_value(1'b0), .instruction(
    instruction), .Zout(Zout));

    ////// test MV_AC_RQ
    test_instruction(.duration(MV_AC_RQ_time_duration), .ins(MV_AC_RQ), .Z_value(1'b0), .instruction(
    instruction), .Zout(Zout));

    ////// test MV_AC_RL
    test_instruction(.duration(MV_AC_RL_time_duration), .ins(MV_AC_RL), .Z_value(1'b0), .instruction(
    instruction), .Zout(Zout));

    ///// test ENDOP
    test_instruction(.duration(ENDOP_time_duration), .ins(ENDOP), .Z_value(1'bX), .instruction(instruction),
     .Zout(Zout));

end

initial begin         // simulation stop condition
    forever begin
        @(posedge clk);
        if (done) begin  // identify the end of the simulation
            #(5*CLK_PERIOD);
            $stop;
        end
    end
end

endmodule : controlUnit_tb
```

9.1.7 Arithmetic and Logic Unit (ALU)

Module - alu.sv

```
module alu import details::*;
#(parameter WIDTH = 12)
(
    input logic signed [WIDTH-1:0]a,b,
    input alu_op_t selectOp,
    output logic signed [WIDTH-1:0]c
);

always_comb begin
    unique case(selectOp)
        clr_alu : c = 'd0;
        pass_alu: c = b;
        add_alu : c = a+b;
        sub_alu : c = a-b;
        mul_alu : c = a*b;
        inc_alu : c = a+WIDTH'(signed'(1));
        default: c = '0;
    endcase
end

endmodule:alu
```

Testbench - alu_tb.sv

```
module alu_tb import details::*;();
```

```
2
3  timeunit 1ns;
4  timeprecision 1ps;
5
6  localparam CLK_PERIOD = 20;
7
8  logic clk;
9  initial begin
10     clk = 0;
11     forever #(CLK_PERIOD/2) clk <= ~clk;
12 end
13
14 localparam WIDTH = 12;
15
16 logic signed [WIDTH-1:0]a,b,c;
17 alu_op_t selectOp;
18
19 alu #(.WIDTH(WIDTH))dut(.*);
20
21 initial begin
22     @(posedge clk);
23     a <= 10;
24     b <= 3;
25     selectOp <= clr_alu;
26
27     @(posedge clk);
28     selectOp <= pass_alu;
29
30     @(posedge clk);
31     selectOp <= add_alu;
32
33     @(posedge clk);
34     selectOp <= sub_alu;
35
36     @(posedge clk);
37     selectOp <= mul_alu;
38
39     @(posedge clk);
40     selectOp <= inc_alu;
41
42     @(posedge clk);
43     selectOp <= idle_alu;
44
45     @(posedge clk);
46     a <= 20;
47     b <= -30;
48     selectOp <= clr_alu;
49
50     @(posedge clk);
51     selectOp <= pass_alu;
52
53     @(posedge clk);
54     selectOp <= add_alu;
55
56     @(posedge clk);
57     selectOp <= sub_alu;
58
59     @(posedge clk);
60     selectOp <= mul_alu;
61
62     @(posedge clk);
63     selectOp <= inc_alu;
64
65     @(posedge clk);
66     selectOp <= idle_alu;
67
68     @(posedge clk);
69     repeat(10) begin
70         @(posedge clk);
71         a = $random();
72         b = $random();
73         selectOp = alu_op_t'($random());
74     end
75     $stop;
76 end
77 endmodule:alu_tb
```

9.1.8 Incrementable Registers

Module - incRegister.sv

```
module incRegister
#(parameter WIDTH = 12)
(
    input logic [WIDTH-1:0]dataIn,
    input logic wrEn, rstN, clk, incEn,
    output logic [WIDTH-1:0]dataOut
);

logic [WIDTH-1:0] value;

always_ff @(posedge clk) begin
    if (~rstN) value <= '0;
    else if (wrEn) value <= dataIn;
    if (incEn) value <= value+ WIDTH'(1);
end

assign dataOut = value;

endmodule:incRegister
```

Testbench - incRegister_tb.sv

```
module incRegister_tb();

timeunit 1ns;
timeprecision 1ps;

localparam CLK_PERIOD = 10;
logic clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WIDTH = 12;

logic unsigned [WIDTH-1:0]dataIn, dataOut;
logic wrEn, incEn, rstN;

incRegister #(.WIDTH(WIDTH)) dut(.*);

initial begin
    @(posedge clk);
    #(CLK_PERIOD*4/5);
    rstN <= 0;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 23;
    wrEn <= 1;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 36;
    wrEn <= 1;
    rstN <= 1;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 15;
    wrEn <= 0;
    incEn <= 1;

    repeat (10) begin
        @(posedge clk);
        #(CLK_PERIOD*4/5);
        dataIn = $random();
        wrEn = $random();
```

```
        incEn = $random();
        rstN = $random();
    end

    $stop;
end

initial begin
    forever begin
       @(posedge clk);
       #(CLK_PERIOD*4/5);
        $display("dataIn = %d   rstN = %b    wrEn = %b   incEn = %b    dataOut = %d", dataIn, rstN, wrEn,
    incEn, dataOut);
    end

end

endmodule:incRegister_tb
```

9.1.9 Register

Module - register.sv

```
module register
#(parameter WIDTH = 12)
(
    input logic [WIDTH-1:0] dataIn,
    input logic clk,rstN,wrEn,
    output logic [WIDTH-1:0]dataOut
);

logic [WIDTH-1:0] value;

always_ff @( posedge clk ) begin
    if (~rstN) value <= '0;
    else if (wrEn) value <= dataIn;
end

assign dataOut = value;

endmodule : register
```

Testbench - register_tb.sv

```
module register_tb();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 20;

logic clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WIDTH = 12;
logic [WIDTH-1:0]dataIn, dataOut;
logic rstN, wrEn;

register #(.WIDTH(WIDTH)) dut(.*);

initial begin
    @(posedge clk);
    #(CLK_PERIOD*4/5);
    rstN <= 0;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    rstN <= 1;
```

```
    dataIn <= 20;
    wrEn <= 0;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 43;
    wrEn <= 1;

    repeat(10) begin
        @(posedge clk);
        #(CLK_PERIOD*4/5);
        dataIn = $random();
        wrEn = $random();
        rstN = $random();
    end

    repeat(2) @(posedge clk);
    $stop;
end

endmodule: register_tb
```

9.1.10 Multiplexer (System Bus)

Module - multiplexer.sv

```
module multiplexer import details::*;
#(
    parameter WIDTH = 12,
    parameter IR_WIDTH = 8
)
(
    input  bus_in_sel_t selectIn,
    input  logic [WIDTH-1:0]DMem, R, RL, RC, RP, RQ, R1, AC,
    input  logic [IR_WIDTH-1:0]IR,
    output logic [WIDTH-1:0]busOut
);

always_comb begin
    unique case(selectIn)
        DMem_bus: busOut = DMem;
        R_bus: busOut = R;
        IR_bus: busOut = WIDTH'(IR);
        RL_bus: busOut = RL;
        RC_bus: busOut = RC;
        RP_bus: busOut = RP;
        RQ_bus: busOut = RQ;
        R1_bus: busOut = R1;
        AC_bus: busOut = AC;
        idle_bus: busOut = '0;
        default: busOut = '0;
    endcase
end

endmodule:multiplexer
```

Testbench - multiplexer_tb.sv

```
module multiplexer_tb import details::*;
();

timeunit 1ns;
timeprecision 1ps;

localparam CLK_PERIOD = 10;
logic clk;
initial begin
    clk = 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end
```

```

localparam  DMem_sel = 4'b0,
            R_sel = 4'd1,
            IR_sel = 4'd2,
            RL_sel = 4'd3,
            RC_sel = 4'd4,
            RP_sel = 4'd5,
            RQ_sel = 4'd6,
            R1_sel = 4'd7,
            AC_sel = 4'd8,
            idle = 4'd9;

localparam WIDTH = 12;
localparam IR_WIDTH = 8;

bus_in_sel_t selectIn;
logic [WIDTH-1:0]DMem, R, RL, RC, RP, RQ, R1, AC;
logic [IR_WIDTH-1:0]IR;
logic [WIDTH-1:0]busOut;

multiplexer #(.WIDTH(WIDTH), .IR_WIDTH(IR_WIDTH)) dut(.*);

assign DMem = 10;
assign R = 11;
assign RL = 12;
assign RC = 13;
assign RP = 14;
assign RQ = 15;
assign R1 = 16;
assign AC = 17;
assign IR = 18;


initial begin
    for (int i = 0; i < 10 ;i++ ) begin
        @(posedge clk);
        selectIn <= bus_in_sel_t'(i);
        $display("selectIn = %d busOut = %d", i, busOut);
    end

    repeat(10) begin
        @(posedge clk);
        void'(std::randomize(selectIn));
        $display("selectIn = %d busOut = %d", selectIn, busOut);
    end

    $stop;
end

endmodule:multiplexer_tb
```

9.1.11 Z register

Module - zReg.sv

```
module zReg
#(parameter WIDTH = 12)
(
    input logic [WIDTH-1:0]dataIn,
    input logic clk, rstN, wrEn,
    output logic Zout
);

logic value;

always_ff @( posedge clk ) begin
    if (~rstN) value <= 1'b0;
    else if (wrEn) begin
        if (dataIn == 0) value <= 1'b1;
        else value <= 1'b0;
    end
end

assign Zout = value;

```

```
endmodule:zReg
```

Testbench - zReg_tb.sv

```
module zReg_tb();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 20;

logic clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WIDTH = 12;
logic [WIDTH-1:0]dataIn;
logic rstN, wrEn, Zout;

zReg #(.WIDTH(WIDTH)) dut(.*);

initial begin
    @(posedge clk);
    rstN <= 0;

    @(posedge clk);
    rstN <= 1;
    dataIn <= 0;
    wrEn <= 0;

    @(posedge clk);
    dataIn <= 0;
    wrEn <= 1;

    @(posedge clk);
    dataIn <= 4;
    wrEn <= 1;

    repeat(10) begin
        @(posedge clk);
        dataIn = $urandom();
        wrEn = $urandom();
        rstN = $urandom();
    end

    $stop;
end

endmodule: zReg_tb
```

9.2 UART communication related SystemVerilog modules

9.2.1 UART communication interface for the memory

Module - mem_communication_interface.sv

```
module mem_communication_interface
#(
    parameter MEM_WORD_LENGTH = 12,
    parameter MEM_DEPTH = 4096,
    parameter MEM_ADDR_LENGTH = $clog2(MEM_DEPTH),
    parameter UART_WIDTH = 8
)
 (
     ///////////////input output with main program
    input  logic clk,rstN,txStartN,
```

```
    output logic  mem_transmitted, mem_received,

    ////////////////inputs outputs with memory
    input  logic [MEM_WORD_LENGTH-1:0]dataFromMem,
    output logic memWrEn,
    output logic [MEM_ADDR_LENGTH-1:0]mem_address,
    output logic [MEM_WORD_LENGTH-1:0]dataToMem,

    ////////////////////inputs outputs with uart system
    input logic rxByteReady, rx_new_byte_indicate, txByteReady,
    input logic [UART_WIDTH-1:0] byteFromRx,
    output logic txByteStart,
    output logic [UART_WIDTH-1:0] byteForTx,

    /////////// select start end mem addresses of tx and rx
    input logic [MEM_ADDR_LENGTH-1:0]tx_start_addr_in,tx_end_addr_in,rx_end_addr_in,
    input logic toggle_addr_range   // 0 - go untill the last address of the memory, 1 - consider inputted
    start, end addresses
);

localparam logic [MEM_ADDR_LENGTH-1:0] LAST_MEM_ADDR = '1; //all bits = 1

logic startTranmit,txReady,rxDone;
logic new_rx_data_indicate;

typedef enum logic [3:0] {
    idle       = 4'b0,
    transmit_0 = 4'd1,
    transmit_1 = 4'd2,
    transmit_2 = 4'd3,
    transmit_3 = 4'd4,
    receive_0  = 4'd5,
    receive_1  = 4'd6,
    receive_2  = 4'd7,
    receive_1_1= 4'd8,
    receive_1_2= 4'd9
} state_t;

state_t currentState, nextState;
logic [MEM_ADDR_LENGTH-1:0]currentAddress, nextAddress;
logic currentStartTransmit, nextStartTranmit;

logic [MEM_ADDR_LENGTH-1:0] tx_start_addr, tx_end_addr, rx_end_addr;

always_ff @(posedge clk) begin
    if (~rstN) begin
        currentState <= idle;
        currentAddress <= '0;
        currentStartTransmit <= 1'b0;
    end
    else begin
        currentState <= nextState;
        currentAddress <= nextAddress;
        currentStartTransmit <= nextStartTranmit;
    end
end

always_comb begin
    nextState = currentState;
    nextAddress = currentAddress;
    nextStartTranmit = currentStartTransmit;

    case (currentState)
        idle: begin
            nextAddress = '0;
            nextStartTranmit = 1'b0;    //to transmit this should be zero
            if (new_rx_data_indicate)
                nextState = receive_0;
            else if (~txStartN) begin
                nextState = transmit_0;
                nextAddress = tx_start_addr;
            end
       end


        //transmission process starts here
        transmit_0: begin         // to give the address to memory
            nextState = transmit_1;
        end
```

```
89
90          transmit_1: begin       // extra delay for ip core memory
91              nextStartTranmit = 1'b1;
92              nextState = transmit_2;
93          end
94
95          transmit_2: begin         // start uart transmitter
96              nextStartTranmit = 1'b1;
97              nextState = transmit_3;
98          end
99
100         transmit_3: begin        // find end of the uart transmission
101             nextStartTranmit = 1'b0;
102             if (txReady == 1'b1) begin
103                 if (currentAddress == tx_end_addr) begin
104                     nextState = idle;
105                 end
106                 else begin
107                     nextAddress = currentAddress + MEM_ADDR_LENGTH'(1'b1);
108                     nextState = transmit_0;
109                 end
110             end
111         end
112         //receiving process starts here
113         receive_0: begin            // to find the end of the uart receiving
114             if (rxDone) begin
115                 nextState = receive_1;
116             end
117         end
118
119         receive_1: begin           // store in the memory (no need extra delay as it is explicitly given in
    receive_0)
120             nextState = receive_1_1;
121         end
122
123         receive_1_1: begin
124             nextState = receive_1_2;
125         end
126
127         receive_1_2: begin
128             nextState = receive_2;
129         end
130
131         receive_2: begin            //to find the end of the receiving process
132             if (currentAddress == rx_end_addr) begin
133                 nextState = idle;
134             end
135             else begin
136                 nextAddress = currentAddress + MEM_ADDR_LENGTH'(1'b1);
137                 nextState = receive_0;
138             end
139         end
140
141     endcase
142 end
143
144 assign startTranmit = currentStartTransmit;
145 assign memWrEn = ((currentState == receive_1) || (currentState == receive_1_1))? 1'b1: 1'b0;
146
147 assign mem_address = currentAddress;
148
149 assign mem_received = ((currentState == receive_2) && (currentAddress == rx_end_addr))? 1'b1: 1'b0;
150 assign mem_transmitted = ((currentState == transmit_3) && (txReady == 1'b1) && (currentAddress ==
    tx_end_addr))? 1'b1: 1'b0;
151
152 assign tx_start_addr = (toggle_addr_range == 1'b0)? '0: tx_start_addr_in;
153 assign tx_end_addr = (toggle_addr_range == 1'b0)? LAST_MEM_ADDR: tx_end_addr_in;
154
155 assign rx_end_addr = (toggle_addr_range == 1'b0)? LAST_MEM_ADDR:
156                      (rx_end_addr_in == 0)? {{MEM_ADDR_LENGTH-4{1'b0}}, {4{1'b1}}}: //give 31 as the end
    rx address if input addr is small
157                      rx_end_addr_in;
158
159 data_encoder_decoder #(.WORD_SIZE(MEM_WORD_LENGTH), .UART_WIDTH(UART_WIDTH)) data_encoder_decoder
160                     (
161                         .dataFromMem,
162                         .clk, .rstN, .txStart(startTranmit),
163                         .txReady, .rxDone,
164                         .dataToMem,
```

```
                        ////////////////// uart ports
                        .new_rx_data_indicate,
                        .rxByteReady, .rx_new_byte_indicate, .txByteReady,
                        .byteFromRx,
                        .txByteStart,
                        .byteForTx
                    );

endmodule:mem_communication_interface
```

Testbench - mem_communication_interface_tb.sv

```
class Memory #(parameter DEPTH , WIDTH);

    localparam ADDR_WIDTH = $clog2(DEPTH);
    typedef logic [WIDTH-1:0] mem_t [0:DEPTH-1];
    typedef logic [WIDTH-1:0]  data_t;
    typedef logic [ADDR_WIDTH-1:0] addr_t;

    rand mem_t memory;
    // addr_t addr;

    function new();
      void'(this.randomize(memory));
    endfunction

    function logic[WIDTH-1:0] getData(input addr_t addr);
        return this.memory[addr];
    endfunction

    function void storeData(input addr_t addr, input data_t data);
        this.memory[addr] = data;
    endfunction

    task Read_memory(input addr_t addr, output data_t value, ref logic clk);
        // if needed read time delay can be added here
        value = memory[addr];
    endtask

    task Write_memory(input addr_t addr, input data_t data, input logic wrEn, ref logic clk);
        // if needed write time delay can be added
        @(posedge clk);
        if (wrEn) begin
            memory[addr] = data;
        end
    endtask

endclass

module mem_communication_interface_tb ();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 20;
logic clk;
initial begin
    clk = 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end
localparam MEM_WORD_LENGTH = 12;
localparam MEM_DEPTH = 64;
localparam MEM_ADDR_LENGTH = $clog2(MEM_DEPTH);
localparam BAUD_RATE = 115200;
localparam UART_WIDTH = 8;
localparam BAUD_TIME_PERIOD = 10**9 / BAUD_RATE;
localparam WORD_2_UART_COUNT = MEM_WORD_LENGTH/UART_WIDTH + ((MEM_WORD_LENGTH % UART_WIDTH == 0)? 0:1);

localparam [MEM_ADDR_LENGTH-1:0]
    tx_start_addr_val = 0,
    tx_end_addr_val = {MEM_ADDR_LENGTH{1'b1}},
    rx_end_addr_val = {MEM_ADDR_LENGTH{1'b1}};
localparam bit toggle_addr_range_val = 1'b0;

//////////////input output with main program
logic rstN,txStartN;
```

```
67  logic mem_transmitted, mem_received;
68
69  ////////////////inputs outputs with memory
70  logic [MEM_WORD_LENGTH-1:0]dataFromMem;
71  logic memWrEn;
72  logic [MEM_ADDR_LENGTH-1:0]mem_address;
73  logic [MEM_WORD_LENGTH-1:0]dataToMem;
74
75  ////////////////////inputs outputs with uart system
76  logic rxByteReady, rx_new_byte_indicate, txByteReady;
77  logic [UART_WIDTH-1:0] byteFromRx;
78  logic txByteStart;
79  logic [UART_WIDTH-1:0] byteForTx;
80
81  //// select start end mem addresses of tx and rx
82  logic [MEM_ADDR_LENGTH-1:0]tx_start_addr_in,tx_end_addr_in,rx_end_addr_in;
83  logic toggle_addr_range;   // 0 - go untill the last address of the memory, 1 - consider inputted start, end
        addresses
84
85  //// logic belongs to uart_system
86  logic tx,rx;
87
88  Memory #(.DEPTH(MEM_DEPTH), .WIDTH(MEM_WORD_LENGTH)) Memory_a;
89  initial begin
90      Memory_a = new();
91  end
92
93
94  mem_communication_interface #(.MEM_WORD_LENGTH(MEM_WORD_LENGTH), .MEM_DEPTH(MEM_DEPTH),
95                  .MEM_ADDR_LENGTH(MEM_ADDR_LENGTH), .UART_WIDTH(UART_WIDTH)) mem_communication_interface (.*)
        ;
96
97  uart_system #(.DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE)) uart_system(.clk, .rstN,.txByteStart,.rx,.
        byteForTx,
98                  .tx, .tx_ready(txByteReady), .rx_ready(rxByteReady), .rx_new_byte_indicate, .byteFromRx);
99
100 ////////////memory read and write
101 always_ff @(posedge clk) begin
102     Memory_a.Read_memory(.addr(mem_address), .value(dataFromMem), .clk(clk));
103 end
104
105 always_ff @(posedge clk) begin
106     Memory_a.Write_memory(.addr(mem_address), .data(dataToMem), .wrEn(memWrEn), .clk(clk));
107 end
108
109
110 // test transmission
111 initial begin
112
113     @(posedge clk);   // initialize values
114     rstN <= 1'b0;
115     rx <= 1'b1;
116     txStartN <= 1'b1;
117     tx_start_addr_in <= tx_start_addr_val;
118     tx_end_addr_in <= tx_end_addr_val;
119     rx_end_addr_in <= rx_end_addr_val;
120     toggle_addr_range <= toggle_addr_range_val;
121
122     //////// transmission testing starts here
123     @(posedge clk);
124     rstN <= 1'b1;
125     txStartN <= 1'b0;
126     @(posedge clk);
127     txStartN <= 1'b1;
128     wait(mem_transmitted);
129
130     @(posedge clk);
131     repeat(5) @(posedge clk);
132     $display("transmission is done \n");
133     ////////////////// receiver testing starts here
134
135     @(posedge clk);
136     rstN <= 1'b0;
137     @(posedge clk);
138     rstN <= 1'b1;
139
140     @(posedge clk);
141     for (int i=0; i <= rx_end_addr_val;i++) begin
142         for (int j=0; j< WORD_2_UART_COUNT;j++) begin
```

```
            @(posedge clk);  //starting delimiter
            rx <= 1'b0;
            #(BAUD_TIME_PERIOD);
            for (int i=0;i<UART_WIDTH;i++) begin:data  //data
                @(posedge clk);
                void'(std::randomize(rx));
                #(BAUD_TIME_PERIOD);
            end
            @(posedge clk);  // end delimiter
            rx <= 1'b1;
            #(BAUD_TIME_PERIOD);
        end
        $display("dataToMem = %d", dataToMem);
    end

    $display("receiving is done %p",Memory_a.memory);

    $stop;  // end of the simulation
end

endmodule:mem_communication_interface_tb
```

9.2.2 Data word size handler between memory and UART system

Module - data_encoder_decoder.sv

```
module data_encoder_decoder
#(
    parameter WORD_SIZE = 12, // width of memory word
    parameter UART_WIDTH = 8
)
(
    input  logic [WORD_SIZE-1:0]dataFromMem,
    input  logic clk,rstN,txStart,
    output logic txReady,rxDone,
    output logic [WORD_SIZE-1:0]dataToMem,
    output logic new_rx_data_indicate,

    ////////////// uart ports
    input logic rxByteReady, rx_new_byte_indicate, txByteReady,
    input logic [UART_WIDTH-1:0] byteFromRx,
    output logic txByteStart,
    output logic [UART_WIDTH-1:0] byteForTx


);

localparam EXTRA = ((WORD_SIZE % UART_WIDTH) == 0)?0:1;
localparam COUNT = (WORD_SIZE/UART_WIDTH) + EXTRA;
localparam BUFFER_WIDTH = COUNT * UART_WIDTH;
localparam COUNTER_LENGTH = (COUNT == 1)? 1:$clog2(COUNT);

typedef enum logic [2:0]{
    idle = 3'd0,
    transmit_1,
    transmit_2,
    receive_0 ,
    receive_1 ,
    receive_2 ,
    receive_3
}state_t;


state_t currentState, nextState;
logic [BUFFER_WIDTH-1:0]currentTxBuffer, nextTxBuffer;
logic [BUFFER_WIDTH-1:0]currentRxBuffer, nextRxBuffer;
logic [COUNTER_LENGTH-1:0]currentTxCount, nextTxCount;
logic [COUNTER_LENGTH-1:0]currentRxCount, nextRxCount;

always_ff @(posedge clk) begin
    if (!rstN) begin
        currentTxBuffer <= '0;
        currentRxBuffer <= '0;
        currentTxCount <= '0;
        currentRxCount <= '0;
```

```
        currentState <= idle;
    end
    else begin
        currentTxBuffer <= nextTxBuffer;
        currentRxBuffer <= nextRxBuffer;
        currentTxCount <= nextTxCount;
        currentRxCount <= nextRxCount;
        currentState <= nextState;
    end
end

always_comb begin
    nextState = currentState;
    nextTxCount = currentTxCount;
    nextRxCount = currentRxCount;
    nextTxBuffer = currentTxBuffer;
    nextRxBuffer = currentRxBuffer;

    case (currentState)
        idle: begin
            nextTxCount = '0;
            nextRxCount = '0;
            if (rx_new_byte_indicate) begin  //receiver indicates a arrival of new byte
                nextState = receive_1;
                nextRxBuffer = '0;
            end
            else if (txStart) begin
                nextState = transmit_1;
                if ((WORD_SIZE % UART_WIDTH) == 0)
                    nextTxBuffer = dataFromMem;
                else
                    nextTxBuffer = {'0,dataFromMem};
            end
        end

        transmit_1: begin           //start byte transmission
            nextState = transmit_2;
        end


        transmit_2: begin           // end of the byte transmission
            if (txByteReady) begin
                if (BUFFER_WIDTH == UART_WIDTH) begin
                    nextState = idle;
                end
                else begin
                    nextTxCount = currentTxCount + COUNTER_LENGTH'(1'b1);
                    if (currentTxCount == COUNT-1 )
                        nextState = idle;
                    else begin
                        nextTxBuffer = currentTxBuffer >> UART_WIDTH;
                        nextState = transmit_1;
                    end
                end
            end
        end

        receive_0: begin           // to give a EXTRA time to make rxByteReady to become 1'b0
            nextState = receive_1;
        end

        receive_1 : begin
            if(rxByteReady) begin
                if (BUFFER_WIDTH == UART_WIDTH) begin
                    nextRxBuffer = byteFromRx;
                    nextState = idle;
                end
                else begin
                    nextRxCount = currentRxCount + COUNTER_LENGTH'(1'b1);
                    nextRxBuffer = {byteFromRx,currentRxBuffer[BUFFER_WIDTH-1 -: (BUFFER_WIDTH - UART_WIDTH)
    ]};
                    if (currentRxCount == (COUNT-1))
                        nextState = idle;
                    else
                        nextState = receive_2;
                end
            end
        end

```

```
        receive_2: begin
            if(rx_new_byte_indicate)  //receiver indicates a arrival of new byte
                nextState = receive_3;
        end

        receive_3: begin              // to give a EXTRA time to make rxByteReady to become 1'b0
            nextState = receive_1;
        end

    endcase
end


assign txByteStart = (currentState == transmit_1)? 1'b1: 1'b0;
assign txReady = (currentState == idle)? 1'b1 : 1'b0;
assign byteForTx = currentTxBuffer[UART_WIDTH-1:0];
assign rxDone = ((currentState == receive_1) && (rxByteReady == 1'b1) && (currentRxCount == COUNT-1 ))? 1'b1
     : 1'b0;
assign dataToMem = currentRxBuffer[WORD_SIZE-1:0];
assign new_rx_data_indicate = ((currentState == idle) && (rx_new_byte_indicate))? 1'b1: 1'b0; //arrival of
    new data set

endmodule //data_encoder_decoder
```

Testbench - data_encoder_decoder_tb.sv

```
module data_encoder_decoder_tb();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 20;
logic clk;

initial begin
    clk <= 1'b0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WORD_SIZE = 24;
localparam UART_WIDTH = 8;
localparam BAUD_RATE = 19200;
localparam BAUD_TIME_PERIOD = 10**9 / BAUD_RATE;
localparam WORD_2_UART_COUNT = WORD_SIZE/UART_WIDTH + ((WORD_SIZE % UART_WIDTH == 0)? 0:1);

//////// logic related to data_encoder_decoder
logic [WORD_SIZE-1:0]dataFromMem;
logic rstN,txStart;
logic txReady,rxDone;
logic [WORD_SIZE-1:0]dataToMem;
logic new_rx_data_indicate;

/////////////// logic related to uart_system
logic rxByteReady, rx_new_byte_indicate, txByteReady;
logic [UART_WIDTH-1:0] byteFromRx;
logic txByteStart;
logic [UART_WIDTH-1:0] byteForTx;
logic rx, tx;

logic tx_ready,rx_ready;

assign txByteReady = tx_ready;
assign rxByteReady = rx_ready;

data_encoder_decoder #(.WORD_SIZE(WORD_SIZE), .UART_WIDTH(UART_WIDTH)) dut(.*);

uart_system #(.DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE)) uart_system (.*);

initial begin

    @(posedge clk);
    rstN <= 1'b0;
    rx = 1'b1;
    txStart <= 1'b0;

    @(posedge clk);
```

```
    rstN <= 1'b1;

    //////////transmission check
    repeat(5) begin
        @(posedge clk);
        wait(txReady);
        @(posedge clk);
        dataFromMem = $random();
        txStart <= 1'b1;

        @(posedge clk);
        txStart <= 1'b0;
    end

    ////////////// reset before receiver check begin
    @(posedge clk);
    wait(txReady);  // wait until transmission is over
    @(posedge clk);
    rstN <= 1'b0;
    @(posedge clk);
    rstN <= 1'b1;

    ////////////////receiver check
    @(posedge clk);
    repeat(5) begin
        for (int j=0; j< WORD_2_UART_COUNT;j++) begin:rx_check
            @(posedge clk);  //starting delimiter
            rx <= 1'b0;
            #(BAUD_TIME_PERIOD);
            for (int i=0;i<UART_WIDTH;i++) begin:data  //data
                @(posedge clk);
                rx = $random();
                #(BAUD_TIME_PERIOD);
            end
            @(posedge clk);  // end delimiter
            rx <= 1'b1;
            #(BAUD_TIME_PERIOD);

        end
    end
    $stop;
end

endmodule:data_encoder_decoder_tb
```

9.2.3 UART system

Module - uart_system.sv

```
module uart_system
#(
    parameter DATA_WIDTH = 8,
    parameter BAUD_RATE = 19200
)(
    input  logic clk, rstN,txByteStart,rx,
    input  logic [DATA_WIDTH-1:0]byteForTx,
    output logic tx,tx_ready,rx_ready,rx_new_byte_indicate,
    output logic [DATA_WIDTH-1:0]byteFromRx
);

logic baudTick;

uart_baudRateGen #(.BAUD_RATE(BAUD_RATE)) baudRateGen(.clk, .rstN, .baudTick);

uart_transmitter #(.DATA_WIDTH(DATA_WIDTH)) transmitter(
                    .dataIn(byteForTx), .clk, .baudTick, .rstN, .txStart(txByteStart), .tx, .tx_ready
                    );

uart_receiver #(.DATA_WIDTH(DATA_WIDTH)) receiver (
                .rx, .clk, .rstN, .baudTick, .rx_ready, .dataOut(byteFromRx), .new_byte_indicate(
    rx_new_byte_indicate)
                );

endmodule:uart_system
```

Testbench - uart_system_tb.sv

```
module uart_system_tb();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 20;
logic clk;

initial begin
    clk <=  0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam DATA_WIDTH = 8;
localparam BAUD_RATE = 19200;
localparam BAUD_TIME_PERIOD = 10**9 / BAUD_RATE;

logic rstN,txByteStart,rx;
logic [DATA_WIDTH-1:0]byteForTx;
logic tx,tx_ready,rx_ready,rx_new_byte_indicate;
logic [DATA_WIDTH-1:0]byteFromRx;

uart_system #(.DATA_WIDTH(DATA_WIDTH), .BAUD_RATE(BAUD_RATE)) dut(.*);

///////// initial reset
initial begin
    @(posedge clk);
    rstN <= 1'b0;
    @(posedge clk);
    rstN <= 1'b1;
end


///////// Trasmitter test
initial begin
    #(CLK_PERIOD*2); // to initialize

    repeat(10) begin
        @(posedge clk);
        wait(tx_ready);
        @(posedge clk);
        byteForTx = $urandom();
        txByteStart = 1'b1;

        @(posedge clk);
        txByteStart = 1'b0;
    end
    @(posedge clk);
    wait(tx_ready);
    $stop;
end

/////////  Receier test
initial begin
    #(CLK_PERIOD*2); // to initialize

    repeat(10) begin
        @(posedge clk);  //starting delimiter
        rx <= 1'b0;
        #(BAUD_TIME_PERIOD);
        for (int i=0;i<DATA_WIDTH;i++) begin:data  //data
            @(posedge clk);
            rx = $urandom();
            #(BAUD_TIME_PERIOD);
        end
        @(posedge clk);  // end delimiter
        rx <= 1'b1;
        #(BAUD_TIME_PERIOD);
    end

    $stop;
end
endmodule:uart_system_tb
```

9.2.4 UART baud rate generator

Module - uart_baudRateGen.sv

```
module uart_baudRateGen
#(
    parameter BAUD_RATE = 19200
)
(
    input logic clk,rstN,
    output logic baudTick
);
localparam CLK_RATE = 50*(10**6);
localparam RESOLUTION = 16;  // samples per 1 baud
localparam int MAX_COUNT = (CLK_RATE/BAUD_RATE/RESOLUTION); //round to smaller integer
localparam WIDTH = $clog2(MAX_COUNT);

logic [WIDTH-1:0]count;

always_ff @(posedge clk) begin
    if (~rstN)
        count <= '0;
    else if (count < MAX_COUNT)
        count <= count + WIDTH'(1'b1);
    else
        count <= '0;
end

assign  baudTick = (count==MAX_COUNT)? 1'b1:1'b0;


endmodule:uart_baudRateGen
```

Testbench - uart_baudRateGen_tb.sv

```
module uart_baudRateGen_tb();

timeunit 1ns;
timeprecision 1ps;

localparam CLK_PERIOD = 20;
logic clk;
initial begin
    clk = 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam BAUD_RATE = 115200;
localparam CLK_RATE = 50*(10**6);
localparam RESOLUTION = 16;  // samples per 1 baud
localparam MAX_COUNT = (CLK_RATE/BAUD_RATE/RESOLUTION);
initial begin
    $display("max_count = %d", MAX_COUNT);
end

logic rstN,baudTick;
uart_baudRateGen #(.BAUD_RATE(BAUD_RATE)) dut (.*);

initial begin
    @(posedge clk);
    rstN <= 1'b0;

    @(posedge clk);
    rstN <= 1'b1;
end

int clk_count = 0;
initial begin
    @(posedge clk);
    repeat(MAX_COUNT *5) begin
        @(posedge clk);
        if (baudTick == 1'b1) begin
```

```
            $display("clk_count = %d", clk_count);
            clk_count = 0;
        end
        else begin
            clk_count = clk_count + 1;
        end
    end
    $stop;
end

endmodule:uart_baudRateGen_tb
```

9.2.5 UART transmitter

Module - uart_transmitter.sv

```
module uart_transmitter
#(
    parameter DATA_WIDTH = 8
)
(
    input logic [DATA_WIDTH-1:0]dataIn,
    input logic clk, baudTick, rstN, txStart,
    output logic tx,tx_ready
);

typedef enum logic [1:0]{
    idle = 2'd0,
    start = 2'd1,
    data_transmit = 2'd2,
    stop = 2'd3
} state_t;

localparam COUNTER_WIDTH = $clog2(DATA_WIDTH);

state_t currentState, nextState;
logic [DATA_WIDTH-1:0]currentData, nextData;
logic currentBit, nextBit;
logic [COUNTER_WIDTH-1:0]currentCount, nextCount;
logic [3:0]currentTick, nextTick;


always_ff @(posedge clk) begin
    if (~rstN) begin
        currentTick <= 4'b0;
        currentCount <= '0;
        currentState <= idle;
        currentData <= '0;
        currentBit <= 1'b1;
    end
    else begin
        currentTick <= nextTick;
        currentCount <= nextCount;
        currentState <= nextState;
        currentData <= nextData;
        currentBit <= nextBit;
    end
end

always_comb begin
    nextState = currentState;
    nextTick = currentTick;
    nextCount = currentCount;
    nextData = currentData;
    nextBit = currentBit;

    case (currentState)
        idle: begin
            if (txStart) begin
                nextTick = 4'd0;
                nextBit = 1'b0;   //start of the data byte
                nextData = dataIn;
                nextState = start;
            end
            else
```

```
                nextBit = 1'b1;
        end

        start: begin
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 4'd15) begin
                    nextCount = '0;
                    nextBit = currentData[0];
                    nextState = data_transmit;
                end
            end
        end

        data_transmit: begin
            nextBit = currentData[0];
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 4'd15) begin
                    nextCount = currentCount + COUNTER_WIDTH'(1'b1);
                    nextData = currentData >>1;
                    if (currentCount == (DATA_WIDTH-1)) begin
                        nextBit = 1'b1;
                        nextState = stop;
                    end
                end
            end
        end

        stop: begin
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 4'd15)
                    nextState = idle;
            end
        end

    endcase
end

assign tx = currentBit;
assign  tx_ready = (currentState == idle)? 1'b1:1'b0;

endmodule //uart_transmitter
```

Testbench - uart_transmitter_tb.sv

```
module uart_transmitter_tb();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 10;
logic clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam DATA_WIDTH = 8;
localparam BAUD_RATE = 19200;

logic rstN,baudTick;
logic [DATA_WIDTH-1:0]dataIn;
logic txStart;
logic tx,tx_ready;

uart_baudRateGen #(.BAUD_RATE(BAUD_RATE)) baudRateGen(.*);
uart_transmitter #(.DATA_WIDTH(DATA_WIDTH)) transmitter(.*);

initial begin
    @(posedge clk);
    rstN <= 1'b0;
    @(posedge clk);
    rstN <= 1'b1;

```

```
    repeat(10) begin
        @(posedge clk);
        wait(tx_ready);
        @(posedge clk);
        dataIn = $urandom();
        txStart = 1'b1;

        @(posedge clk);
        txStart = 1'b0;
    end
    @(posedge clk);
    wait(tx_ready);
    $stop;
end

endmodule:uart_transmitter_tb
```

9.2.6 UART receiver

Module - uart_receiver.sv

```
module uart_receiver
#(
    parameter DATA_WIDTH = 8
)
(
    input  logic rx, clk, rstN, baudTick,
    output logic rx_ready,
    output logic [DATA_WIDTH-1:0]dataOut,
    output logic new_byte_indicate
);

typedef enum logic [1:0]{
    idle = 2'd0,
    start = 2'd1,
    data_receive = 2'd2,
    stop = 2'd3
} state_t;

localparam COUNTER_WIDTH = $clog2(DATA_WIDTH);

state_t currentState, nextState;
logic [COUNTER_WIDTH-1:0]currentCount,nextCount;
logic [DATA_WIDTH-1:0]currentData, nextData;
logic [3:0]currentTick, nextTick;

always_ff @(posedge clk) begin
    if(~rstN) begin
        currentState <= idle;
        currentCount <= '0;
        currentTick <= 4'b0;
        currentData <= '0;
    end
    else begin
        currentState <= nextState;
        currentCount <= nextCount;
        currentTick <= nextTick;
        currentData <= nextData;
    end
end

always_comb begin
    nextState = currentState;
    nextCount = currentCount;
    nextTick = currentTick;
    nextData = currentData;

    case (currentState )
        idle: begin
            nextTick = 4'b0;
            nextCount = '0;
            if (rx == 1'b0) begin
                nextState = start;
            end
        end
```

```
55
56          start: begin
57             if (baudTick) begin
58                 nextTick = currentTick + 4'b1;
59                 if (currentTick == 4'd7) begin
60                     if (~rx) begin
61                          nextState = data_receive;
62                          nextCount = '0;
63                          nextTick = 4'b0;
64                          nextData = '0;
65                     end
66                     else begin
67                         nextState = idle;
68                     end
69                 end
70             end
71          end
72
73          data_receive: begin
74              if (baudTick) begin
75                  nextTick = currentTick + 4'b1;
76                  if (currentTick == 4'd15) begin
77                      nextData = {rx,currentData[DATA_WIDTH-1:1]};
78                      nextCount = currentCount + COUNTER_WIDTH'(1'b1);
79                      if (currentCount == (DATA_WIDTH-1)) begin
80                          nextState = stop;
81                      end
82                  end
83              end
84          end
85
86          stop: begin
87              if (baudTick) begin
88                  nextTick = currentTick + 4'b1;
89                  if (currentTick == 4'd15) begin
90                      nextState = idle;
91                  end
92              end
93          end
94
95      endcase
96  end
97
98  assign dataOut = currentData;
99  assign rx_ready = (currentState == idle)? 1'b1: 1'b0;
100 assign new_byte_indicate = ((currentState == start) && (baudTick) && (currentTick == 4'd7) && (~rx))? 1'b1
        :1'b0; //start of new data_receive byte
101
102 endmodule //uart_receiver
```

Testbench - uart_receiver_tb.sv

```
1  module uart_receiver_tb();
2
3  timeunit 1ns;
4  timeprecision 1ps;
5  localparam CLK_PERIOD = 20;
6  logic clk;
7
8  initial begin
9      clk <= 1'b0;
10     forever begin
11         #(CLK_PERIOD/2);
12         clk <= ~clk;
13     end
14 end
15
16 localparam DATA_WIDTH = 8;
17 localparam BAUD_RATE = 19200;
18 localparam BAUD_TIME_PERIOD = 10**9 / BAUD_RATE;
19
20 logic rstN,baudTick;
21 logic rx, rx_ready;
22 logic [DATA_WIDTH-1:0]dataOut;
23 logic new_byte_indicate;
24
25 uart_baudRateGen #(.BAUD_RATE(BAUD_RATE)) baudRateGen(.*);
26 uart_receiver #(.DATA_WIDTH(DATA_WIDTH)) receiver(.*);
```

```
27
28 initial begin
29     @(posedge clk);
30     rstN <= 1'b0;
31     rx <= 1'b1;
32     @(posedge clk);
33     rstN <= 1'b1;
34
35     repeat(10) begin
36         @(posedge clk);  //starting delimiter
37         rx <= 1'b0;
38         #(BAUD_TIME_PERIOD);
39         for (int i=0;i<DATA_WIDTH;i++) begin:data  //data
40             @(posedge clk);
41             rx = $urandom();
42             #(BAUD_TIME_PERIOD);
43         end
44         @(posedge clk);  // end delimiter
45         rx <= 1'b1;
46         #(BAUD_TIME_PERIOD);
47     end
48     $stop;
49 end
50
51 endmodule:uart_receiver_tb
```

9.3 Count and display the number of clock cycles for matrix multiplication process

9.3.1 Clock cycle counter - timeCounter.sv

```
1  module timeCounter (
2      input logic clk, rstN, start, stop,
3      output logic [25:0]timeDuration
4  );
5  localparam  idle = 2'b0,
6              counting = 2'b1,
7              countEnd = 2'd2;
8
9  reg [25:0]currentTime, nextTime;
10 reg [1:0]currentState, nextState;
11
12 always @(posedge clk) begin
13     if (~rstN) begin
14         currentTime <= 26'b0;
15         currentState <= idle;
16     end
17     else begin
18         currentTime <= nextTime;
19         currentState <= nextState;
20     end
21 end
22
23 always_comb begin
24     nextTime = currentTime;
25     nextState = currentState;
26
27     case (currentState)
28         idle: begin
29             nextTime = 26'b0;
30             if (start) begin
31                 nextState = counting;
32             end
33         end
34
35         counting: begin
36             if (stop)begin
37                 nextState = countEnd;
38             end
39             else begin
40                 nextTime = currentTime + 26'b1;
41             end
42         end
```

```
43
44          countEnd: begin
45              //wait until reset (keep the same count)
46          end
47
48      endcase
49  end
50
51  assign  timeDuration = currentTime;
52
53  endmodule //timeCounter
```

9.3.2 Display on 8 seven segments on the FPGA board

Module - hex_display.sv

```
1  module hex_display (
2      input logic clk, rstN,
3      input logic [2:0]state,
4      input logic start_timeValue_convetion,
5      input logic [25:0]binary_time_value,
6      // output logic  [6:0]out0, out1,out2,out3,out4,out5,out6,out7
7      output logic [6:0]hex_display_value[7:0]
8  );
9
10 localparam  //lettes needed to be shown on seven segments
11             a = 5'd10,
12             b = 5'd11,
13             c = 5'd12,
14             d = 5'd13,
15             e = 5'd14,
16             f = 5'd15,
17             i = 5'd18,
18             n = 5'd19,
19             o = 5'd20,
20             p = 5'd21,
21             r = 5'd22,
22             s = 5'd23,
23             t = 5'd24,
24             u = 5'd25,
25             y = 5'd26,
26             off = 5'd27;
27
28 typedef enum logic [2:0] {
29     idle = 3'd0,
30     uart_receive_Imem = 3'd1,
31     uart_receive_dmem = 3'd2,
32     process_ready = 3'd3,
33     process_exicute = 3'd4,
34     uart_transmit_dmem = 3'd5,
35     finish = 3'd6
36 } state_t;
37
38 state_t currentState, nextState;
39
40 logic done1;
41 logic [3:0]BCD_time_value[7:0];
42 logic [4:0]letters[7:0];
43
44 always_comb begin
45     unique case (state)
46         idle :
47             letters = '{off,off,off,r,e,a,d,y};
48
49         uart_receive_Imem :
50             letters = '{u,a,r,t,off,i,n,s};
51
52         uart_receive_dmem :
53             letters = '{u,a,r,t,d,a,t,a};
54
55         process_exicute :
56             letters = '{off,p,r,o,c,e,s,s};
57
58         uart_transmit_dmem :
59             letters = '{u,a,r,t,off,a,n,s};
```

```
60
61          finish :
62              for (int i=0;i<8;i++) begin
63                  letters[i] = {1'b0, BCD_time_value[i]};
64              end
65
66          default:
67              letters = '{off,off,off,off,off,off,off,off};
68      endcase
69  end
70
71  convertToHEX BtH(.value(letters), .Hex_value(hex_display_value));
72
73
74  binaryToBCD timeConverter(.binary_value(binary_time_value),.clk(clk),.rstN(rstN),
75                            .start(start_timeValue_convetion), .done(),.ready(),
76                            .BCD_value(BCD_time_value)
77                            );
78
79  endmodule //hex_display
```

Module - convertToHEX.sv

```
1  module convertToHEX(
2      input logic [4:0]value[7:0],
3      output logic [6:0]Hex_value[7:0]
4  );
5
6  genvar i;
7  generate
8      for (i=0;i<8;i++) begin :SSeg
9          SSeg SSeg(.in(value[i]), .out(Hex_value[i]));
10     end
11 endgenerate
12
13 endmodule :convertToHEX
```

Module - SSeg.sv

```
1  module SSeg(
2      input logic [4:0]in,
3      output logic [6:0]out
4  );
5
6  localparam
7              a = 5'd10,
8              b = 5'd11,
9              c = 5'd12,
10             d = 5'd13,
11             e = 5'd14,
12             f = 5'd15,
13             i = 5'd18,
14             n = 5'd19,
15             o = 5'd20,
16             p = 5'd21,
17             r = 5'd22,
18             s = 5'd23,
19             t = 5'd24,
20             u = 5'd25,
21             y = 5'd26,
22             off = 5'd27;
23
24
25 always_comb begin
26     case (in)
27         5'd0:out = 7'b1000000;
28         5'd1:out = 7'b1111001;
29         5'd2:out = 7'b0100100;
30         5'd3:out = 7'b0110000;
31         5'd4:out = 7'b0011001;
32         5'd5:out = 7'b0010010;
33         5'd6:out = 7'b0000010;
34         5'd7:out = 7'b1111000;
35         5'd8:out = 7'b0000000;
36         5'd9:out = 7'b0011000;
```

```
        5'ha:out = 7'b0001000;
        5'hb:out = 7'b0000011;
        5'hc:out = 7'b1000110;
        5'hd:out = 7'b0100001;
        5'he:out = 7'b0000110;
        5'hf:out = 7'b0001110;
        i   :out = 7'b1110000;
        n   :out = 7'b0001011;
        o   :out = 7'b1000000;
        p   :out = 7'b0001100;
        r   :out = 7'b1001110;
        s   :out = 7'b0010010;
        t   :out = 7'b0000111;
        u   :out = 7'b1000001;
        y   :out = 7'b0010001;
        off :out = 7'b1111111;

        default: out = 7'b1111111;
    endcase
end

endmodule // SSeg
```

Module - binaryToBCD.sv

```
module binaryToBCD(
    input logic [25:0]binary_value, // to get 8 digit output need 26.57 input bits (26)
    input logic clk,rstN,start,
    output logic ready, done,
    output logic [3:0]BCD_value[7:0]
);

typedef enum logic [1:0] {
    idle,
    subtract,
    shift,
    over
  } state_t;

state_t currentState,nextState;

logic [4:0]currentCount,nextCount;        //count upto 26
logic [31:0]currentValue,nextValue;
logic [25:0]currentInput,nextInput;



always @(posedge clk) begin
    if (!rstN) begin
        currentCount <= 5'b0;
        currentValue <= 32'b0;
        currentState <= idle;
        currentInput <= 26'b0;
    end else begin
        currentValue <= nextValue;
        currentCount <= nextCount;
        currentState <= nextState;
        currentInput <= nextInput;
    end
end

////next state logic
always_comb begin
    nextState = currentState;
    case (currentState)
        idle:
            if (!start) nextState = subtract;

        subtract:
            nextState = shift;

        shift:
            if (currentCount<5'd25)
                nextState = subtract;
            else
                nextState = over;

        over:
```

```
            nextState = idle;

    endcase
end

always_comb begin
    nextCount = currentCount;
    nextValue = currentValue;
    nextInput = currentInput;
    BCD_value = '{default:'0};

    unique case (currentState)
        idle: begin
            for (int j=0;j<8;j++) begin :digit_set
                BCD_value[j] = currentValue[4*j+:4];
            end

            if (!start) begin        // KEY is push button stay at 1 normally
                nextCount = 5'b0;
                nextInput = binary_value;
        nextValue = 32'b0;
            end
        end

        subtract: begin
            for (int i=0;i<32;i= i+4) begin: sub
                if (currentValue[i+:4] >4)
                    nextValue[i+:4] = currentValue[i+:4] + 4'b0011;
            end
        end

        shift: begin
            nextInput[25:1] = currentInput[24:0];
            nextValue = {currentValue[30:0],currentInput[25]};
            nextCount = currentCount+5'b1;
        end

        over: begin
            for (int k=0;k<8;k++) begin :digit_set_2
                BCD_value[k] = currentValue[4*k+:4];
            end
        end
    endcase
end

assign done = (currentState != over);  // actives the BCDtoHEX by the negedge
assign ready = (currentState == idle);

endmodule: binaryToBCD
```

9.4 Testbenches for top level module

In order to do the simulation for the full design, the top level module (Section - 9.1.2) is changed appropriately. Also instead of using a single module RAM (Section - 9.1.3) for both data-memory and instruction-memory we used 2 separate modules in order to memory initialization using external text file and write the memory content to a text file after the process is finished in order to validate by comparison the answer matrix with Python3.8 based calculation.

Module - simulation_top_tb.sv

```
module simulation_top_tb();

timeunit 1ns;
timeprecision 1ps;
localparam CLK_PERIOD = 10;

logic clk;
initial begin
    clk <= 0;
    forever begin
```

```
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam CORE_COUNT = 1;

logic rstN, startN;
logic processor_ready, processDone;

simulation_top #(.CORE_COUNT(CORE_COUNT)) simulation_top(.*);

initial begin
    @(posedge clk);
    rstN = 1'b0;
    startN = 1'b1;

    @(posedge clk);
    rstN = 1'b1;
    startN = 1'b0;

    @(posedge clk);
    startN = 1'b1;

    wait(processDone);

    repeat(10) @(posedge clk);
    $stop;
end

endmodule : simulation_top_tb
```

Module - simulation_top.sv

```
module simulation_top import details::*;
#(
    parameter CORE_COUNT = 1
)
(
    input logic clk, rstN, startN,
    output logic processor_ready, processDone
);

// localparam CORE_COUNT = 2;
localparam REG_WIDTH = 12;
localparam DATA_MEM_WIDTH = CORE_COUNT * REG_WIDTH;
localparam INS_WIDTH = 8;
localparam INS_MEM_DEPTH = 256;
localparam DATA_MEM_DEPTH = 4096;
localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);


////// logic related to data memory //////////
logic [DATA_MEM_WIDTH-1:0]DataMemOut, DataMemIn, processor_DataOut, uart_DataOut;
logic [DATA_MEM_ADDR_WIDTH-1:0] processor_dataMemAddr;
logic [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr, uart_dataMemAddr;

////// logic related to instruction memory ///////////
logic [INS_WIDTH-1:0]InsMemOut, InsMemIn;
logic [INS_MEM_ADDR_WIDTH-1:0] processor_InsMemAddr;
logic [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,uart_InsMemAddr;

////// other logics ////////////
logic dataMemWrEn, processor_DataMemWrEn, uart_dataMemWrEn;
logic uart_InsMemWrEn;
logic processStart;



///////////////// state change logic

typedef enum logic [2:0] {
    idle = 3'd0,
    process_exicute = 3'd4,
    finish = 3'd6
} state_t;

```

```
state_t currentState, nextState;

always @(posedge clk) begin
    if (~rstN) begin
        currentState <= idle;
    end
    else begin
        currentState <= nextState;
    end
end

always_comb begin
    nextState = currentState;

    case (currentState)
        idle: begin     // start state
            if (~startN) begin
                nextState = process_exicute;
            end
        end

        process_exicute: begin  // processor exicute program (matrix multiplication)
            if (processDone) begin
                nextState = finish;
            end
        end


        finish: begin  //End of the process

        end

        default : nextState = idle;

    endcase
end

assign processStart = ((currentState == idle) && (~startN))? 1'b1: 1'b0;

assign dataMemWrEn = (currentState == process_exicute)? processor_DataMemWrEn : 1'b0;

assign dataMemAddr = (currentState == process_exicute)? processor_dataMemAddr: {DATA_MEM_ADDR_WIDTH{1'b0}};

assign DataMemIn = (currentState == process_exicute)? processor_DataOut: {DATA_MEM_WIDTH{1'b0}};

assign insMemAddr = (currentState == process_exicute)? processor_InsMemAddr: {INS_MEM_ADDR_WIDTH{1'b0}};

////////////////////////////////////////////////////////////////////////
multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
                    .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
                    multi_core_processor
                    (.clk,.rstN,.start(processStart), .ProcessorDataIn(DataMemOut), .InsMemOut,
                    .ProcessorDataOut(processor_DataOut), .insMemAddr(processor_InsMemAddr),
                    .dataMemAddr(processor_dataMemAddr), .DataMemWrEn(processor_DataMemWrEn),
                    .done(processDone), .ready(processor_ready));


 INS_RAM #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH), .mem_init(yes))
        IM(.clk(clk), .wrEn(uart_InsMemWrEn), .dataIn(InsMemIn), .addr(insMemAddr), .dataOut(InsMemOut));

 DATA_RAM #(.WIDTH(DATA_MEM_WIDTH), .DEPTH(DATA_MEM_DEPTH), .mem_init(yes))
        DM(.clk(clk), .wrEn(dataMemWrEn), .dataIn(DataMemIn), .addr(dataMemAddr), .dataOut(DataMemOut),
        .processDone(processDone));


endmodule :simulation_top
```

Module - INS_RAM.sv

```
module INS_RAM import details::*;
#(
    parameter mem_init_t mem_init = no,
    parameter WIDTH = 8,
    parameter DEPTH = 256,
    parameter ADDR_WIDTH = $clog2(DEPTH)
)
(
    input logic clk, wrEn,
```

```
    input logic [WIDTH-1:0]dataIn,
    input logic [ADDR_WIDTH-1:0]addr,
    output logic [WIDTH-1:0]dataOut
);


logic [ADDR_WIDTH-1:0]addr_reg;

logic [WIDTH-1:0]memory[0:DEPTH-1] ;

/// initialize memory for simulation  /////////
initial begin
    if (mem_init == yes) begin
        $readmemb("../../9_ins_mem_tb.txt", memory);
    end
end
////////////////////////////////////////////////

always_ff @(posedge clk) begin
    addr_reg <= addr;
    if (wrEn) begin
        memory[addr] <= dataIn;   // write requires only 1 clk cyle.
    end
end
assign dataOut = memory[addr_reg];   // address is registered. Need 2 clk cycles to read.

endmodule : INS_RAM
```

Module - DATA_RAM.sv

```
module DATA_RAM import details::*;
#(
    parameter mem_init_t mem_init = no,
    parameter WIDTH = 12,
    parameter DEPTH = 4096,
    parameter ADDR_WIDTH = $clog2(DEPTH)
)
(
    input logic clk, wrEn,
    input logic [WIDTH-1:0]dataIn,
    input logic [ADDR_WIDTH-1:0]addr,
    output logic [WIDTH-1:0]dataOut,

    input logic processDone   // need only for simulation (to get the memory at the end)
);


logic [ADDR_WIDTH-1:0]addr_reg;

logic [WIDTH-1:0]memory[0:DEPTH-1] ;

/// initialize memory for simulation  /////////
initial begin
    if (mem_init == yes) begin
        $readmemb("../../4_data_mem_tb.txt", memory);
    end
end

always_comb begin
    if (processDone) begin
        $writememb("../../11_data_mem_out.txt", memory);
    end
end

////////////////////////////////////////////////

always_ff @(posedge clk) begin
    addr_reg <= addr;
    if (wrEn) begin
        memory[addr] <= dataIn;   // write requires only 1 clk cyle.
    end
end
assign dataOut = memory[addr_reg];   // address is registered. Need 2 clk cycles to read.

endmodule : DATA_RAM
```

9.5 Python code for implementation and verification

9.5.1 Python code

```
import math
import time

from processor_matrix_multiplication_functions import *
baud_rate = 115200

# matrix A - a x b
# matrix B - b x c
a = int(input("give a: "))
b = int(input("give b: "))
c = int(input("give c: "))
core_count = int(input("give number of cores: "))
d = math.ceil(a/core_count)  #(number of raws per one core)

create_random_matrices(a,b,c,core_count)

decoded_code = read_assembly_code()
send_instructions_memory_UART(decoded_code,baud_rate)

time.sleep(1) ## time gap between insMem, dataMem transmission

arrange_and_send_data_memory_to_FPGA(a,b,c,d,core_count,baud_rate)

receive_answer_matrix_from_FPGA(a,b,c,d,core_count,baud_rate)

calculate_answer_matrix_in_PC()

compare_answer_matrix_FPGA_and_PC()
```

9.5.2 Python functions

```
import random
import serial
import math
import struct
import time

def create_random_matrices(a,b,c,core_count):
    writeFile  = open('1_matrix_in.txt','w')

    writeFile.write(str(a)+"\n"+str(b)+"\n"+str(c)+"\n"+str(core_count)+" //number of cores"+"\n")

    writeFile.write("\n//matrix A\n")
    out = ""
    for x in range(a):
        temp = ""
        for y in range (b):
            temp+=(str(random.randint(-5,5))+" ")
        out+=(temp+"\n")

    print (out)
    writeFile.write(out)

    writeFile.write("\n//matrix B\n")

    out = ""
    for x in range(b):
        temp = ""
        for y in range (c):
            temp+=(str(random.randint(-5,5))+" ")
        out+=(temp+"\n")

    print (out)
    writeFile.write(out)
    writeFile.close()

    return


```

```
def decode(instruction):
    isa = {
        'NOP': '00000000',
        'ENDOP': '00000001',
        'CLAC': '00000010',
        'LDIAC': '00000011',
        'LDAC': '00000100',
        'STR': '00000101',
        'STIR': '00000110',
        'JUMP': '00000111',
        'JMPNZ': '00001000',
        'JMPZ': '00001001',
        'MUL': '00001010',
        'ADD': '00001011',
        'SUB': '00001100',
        'INCAC': '00001101',
        'MV_RL_AC': '00011111',
        'MV_RP_AC': '00101111',
        'MV_RQ_AC': '00111111',
        'MV_RC_AC': '01001111',
        'MV_R_AC': '01011111',
        'MV_R1_AC': '01101111',
        'MV_AC_RP': '01111111',
        'MV_AC_RQ': '10001111',
        'MV_AC_RL': '10011111',
    }
    if instruction in isa:
        return isa[instruction]
    else:
        return instruction


def read_assembly_code():

    code = open('2_assembly_code.txt','r')
    txt_code = code.read().strip().split('\n')

    decoded_code = []
    i = 0
    while(i<len(txt_code)):
        if txt_code[i][0]=='/':
            i+=1
            continue
        codewithoutcomments = ''
        for j in txt_code[i]:
            if j!='/':
                codewithoutcomments+=j
            else:
                break
        assert codewithoutcomments[-1]==';','Error! missing ; at the end of the line'
        temp = codewithoutcomments[:-1]
        decoded_int = decode(temp)
        if(temp=='LDIAC'or temp=='STIR' or temp=='JMPNZ' or temp=='JUMP' or temp == 'JMPZ'):
            decoded_code.append(decoded_int)
            address = txt_code[i+1]
            addresswithoutcomments = ''
            for j in address:
                if j!='/':
                    addresswithoutcomments+=j
                else:
                    break
            assert addresswithoutcomments[-1]==';','Error! missing ; at the end of the line'
            decoded_code.append('{0:08b}'.format(int(addresswithoutcomments[:-1])))
            i+=1
        else:
            decoded_code.append(decoded_int)
        i+=1

    return decoded_code

def send_instructions_memory_UART(decoded_code,baud_rate):
    uart_list = []
    for i in range(255,-1,-1):
        if (i<len(decoded_code)):
            value = (128)*int(decoded_code[i][0]) + (64)*int(decoded_code[i][1]) + (32)*int(decoded_code[i
    ][2]) + (16)*int(decoded_code[i][3]) + (8)*int(decoded_code[i][4]) + (4)*int(decoded_code[i][5]) + (2)*
    int(decoded_code[i][6]) + (1)*int(decoded_code[i][7])
            uart_list.append(value)
        else:
```

```
            uart_list.append(0)

    ser = serial.Serial('COM3',baud_rate,bytesize=8)

    ser.write(uart_list[::-1])
    ser.close()


def arrange_and_send_data_memory_to_FPGA(a,b,c,d,core_count,baud_rate):
    data = open('1_matrix_in.txt','r')
    txt_code = data.read().strip().split('\n')

    out = b''

    memWordLength = core_count*12
    byte_count = math.ceil(memWordLength/8)
    full_length = 8*byte_count
    extra_length = full_length - memWordLength

    start_addr_P = 14
    start_addr_Q = start_addr_P + d*b
    start_addr_R = start_addr_Q + b*c + 1 #with extra space
    end_addr_P = start_addr_P + d*b -1
    end_addr_Q = start_addr_Q + b*c -1
    end_addr_R = start_addr_R + d*c - 1

    setup_values = [a, b, c, start_addr_P, start_addr_Q, start_addr_R, end_addr_P, end_addr_Q, end_addr_R,
    0, 0, 0, 0, 0]

    memLength = 4096
    filled_memLength = len(setup_values) + (a//core_count + (a % core_count != 0))*b + b*c
    empty_memLength = memLength-filled_memLength

    ############################### to send setup values ##############################

    for i in setup_values:
        # print(i)
        temp = '{0:012b}'.format(i)*core_count
        temp = extra_length*'0' + temp
        for x in range(byte_count,0,-1):
            temp2 = int (("0b" + temp[(x-1)*8:x*8]), 2)
            out += struct.pack('!B', temp2)


    ############################### to send matrix A values

    i = 4
    while (txt_code[i] == '' or  txt_code[i][0] == '/'): #to find the start of matrix A
        i = i+1

    A = []
    for x in range (i,i+a):
        temp2 = []
        temp = txt_code[x].strip().split(" ")
        for j in temp:
            if (int(j)>=0):
                temp2.append('{0:012b}'.format(int(j)))
            else:
                temp2.append(bin((int('1'*12,2)^abs(int(j)))+1)[2:]) ## 2's complement
        A.append(temp2)

    if len(A)%core_count!=0:
        for k in range(core_count-len(A)%core_count):
            temp = ['{0:012b}'.format(int(0)) for i in range(len(A[0]))]
            A.append(temp)

    for x in range(0,len(A),core_count):
        for y in range(len(A[0])):
            temporary_bin = ''
            for k in range(core_count):
                temporary_bin+=A[x+k][y]

            temporary_bin = extra_length*'0' + temporary_bin
            for j in range(byte_count,0,-1):
                temp2 = int (("0b" + temporary_bin[(j-1)*8:j*8]), 2)
                out += struct.pack('!B', temp2)

    ############ to send matrix B values ##########

```

```
    i = i + a
    while (txt_code[i] == '' or  txt_code[i][0] == '/'): #to find the start of matrix B
        i = i+1

    matrix_B = []
    for x in range (i,i+b):
        temp = txt_code[x].strip().split(" ")
        matrix_B.append(temp)

    for y in range(c):
        for x in range(b):
            temporary_bin = ''
            for k in range(core_count):
                if (int(matrix_B[x][y]) < 0):
                    temporary_bin+= bin((int('1'*12,2)^abs(int(matrix_B[x][y])))+1)[2:] ## 2's complement
                else:
                    temporary_bin+='{0:012b}'.format(int(matrix_B[x][y]))

            temporary_bin = extra_length*'0' + temporary_bin
            for j in range(byte_count,0,-1):
                temp2 = int (("0b" + temporary_bin[(j-1)*8:j*8]), 2)
                out += struct.pack('!B', temp2)

    ############ append 0s for last left words

    ser = serial.Serial('COM3',baud_rate,bytesize=8)
    ser.write(out)

def decodeCombinedValues(valueIn,no_of_cores):
    if (no_of_cores == 1):
        return [valueIn]

    out = []
    temp = bin(valueIn)[2:]
    temp = format(int(temp), ('0'+str(12*no_of_cores)+'d'))
    for x in range(no_of_cores):
        out.append(int(('0b' + temp[x*12:x*12+12]), 2))
    return out

def convertToCorrectForm(inputMatrix, no_of_cores, a,c,d):

    OutputMatrix = [[0 for x in range(c)] for y in range(a)]

    for x in range(d):
        for y in range(c):
            for z in range(no_of_cores):
                if (x*no_of_cores+z) < a:
                    val = inputMatrix[x*c+y][z]
                    if (val > 2047):
                        val = -4096 + val
                        OutputMatrix[x*no_of_cores+z][y] = "-"+hex(val)[3:]
                    else:
                        OutputMatrix[x*no_of_cores+z][y] = hex(val)[2:]
    return (OutputMatrix)

def receive_answer_matrix_from_FPGA(a,b,c,d,core_count,baud_rate):
    memWordLength = core_count*12
    byte_count = math.ceil(memWordLength/8)
    full_length = 8*byte_count
    extra_length = full_length - memWordLength

    start_addr_P = 14
    start_addr_Q = start_addr_P + d*b
    start_addr_R = start_addr_Q + b*c + 1 #with extra space
    end_addr_P = start_addr_P + d*b -1
    end_addr_Q = start_addr_Q + b*c -1
    end_addr_R = start_addr_R + d*c - 1

    ser = serial.Serial('COM3',baud_rate,bytesize=8)

    ################################# receive dmemory after calculation

    DMem = []
    R_matrix_lenght = end_addr_R - start_addr_R + 1

    for x in range(R_matrix_lenght):
        temp = ''
        for x in range (byte_count):
            in_bin = ser.read()
```

```
            temp = '{0:08b}'.format((int.from_bytes(in_bin,byteorder='little'))) + temp
        # print (hex(int("0b"+temp[extra_length:],2)))
        temp = int("0b"+temp[extra_length:],2)
        DMem.append(temp)

    ################################## find R matrix

    matrix_initial = DMem
    matrix_second = []

    for x in matrix_initial:
        matrix_second.append(decodeCombinedValues(x, core_count))

    matrix_R = convertToCorrectForm(matrix_second, core_count, a,c,d)

    writeFile = open('3_answer_matrix_FPGA.txt', 'w')

    for x in (matrix_R):
        for y in (x):
            writeFile.write((6-len(y))*" " + y)
        writeFile.write('\n')
    writeFile.close()

def calculate_answer_matrix_in_PC():

    data = open('1_matrix_in.txt','r')
    txt_code = data.read().strip().split('\n')
    writeFile  = open('4_answer_matrix_PC.txt','w')
    out = []

    a = int(txt_code[0])
    b = int(txt_code[1])
    c = int(txt_code[2])

    # matrix A - a x b
    # matrix B - b x c
    i = 4
    while (txt_code[i] == '' or  txt_code[i][0] == '/'): #to find the start of matrix A
        i = i+1

    A = []
    for x in range (i,i+a):
        temp = txt_code[x].strip().split(" ")
        A.append(temp)
    i = i + a
    while (txt_code[i] == '' or  txt_code[i][0] == '/'): #to find the start of matrix B
        i = i+1

    matrix_B = []
    for x in range (i,i+b):
        temp = txt_code[x].strip().split(" ")
        matrix_B.append(temp)

    mul = []

    # begin_time = time.time()

    for i in range(int(a)):
        temp_list= []
        for k in range(int(c)):
            temp_ans = 0
            for j in range(int(b)):
                temp_ans += int(A[i][j])*int(matrix_B[j][k])
                if (temp_ans<0):
                    val = "-"+hex(temp_ans)[3:]
                else:
                    val = hex(temp_ans)[2:]
            temp_list.append(" "*(5-len(val))+ val)
        mul.append(temp_list)
    out = ' '
    for i in mul:
        temp = ''
        for j in i:
            temp+=(str(j)+' ')
        out+=(temp+'\n ')
    # print(out)
    writeFile.write(out)
    writeFile.close()

```

```
    # end_time = time.time()
    # print ((end_time - begin_time)*(2.5*(10**9)))

def compare_answer_matrix_FPGA_and_PC():

    python_anwer_file = open('4_answer_matrix_PC.txt','r')
    fpga_answer_file = open('3_answer_matrix_FPGA.txt','r')

    python_anwer_matrix_temp = python_anwer_file.read().strip().split('\n')
    python_anwer_matrix = []
    for line in python_anwer_matrix_temp:
        python_anwer_matrix.append(line.split())
    python_anwer_file.close()

    fpga_answer_matrix_temp = fpga_answer_file.read().strip().split('\n')
    fpga_answer_matrix = []
    for line in fpga_answer_matrix_temp:
        fpga_answer_matrix.append(line.split())
    fpga_answer_file.close()

    dim_a = len(python_anwer_matrix)
    dim_c = len(python_anwer_matrix[0])

    not_similar = 0
    for x in range (dim_a):
        for y in range (dim_c):
            if (python_anwer_matrix[x][y] != fpga_answer_matrix[x][y]):
                print ("[",x,",",y,"] fpga value = ",fpga_answer_matrix[x][y]," python value = ",
    python_anwer_matrix[x][y])
                not_similar = 1

    if (not_similar):
        print ("wrong answer")
    else:
        print ("correct answer")
```

9.6 Text files used during synthesis and Verification

9.6.1 Assembly code

```
CLAC;
LDIAC;
1;//addresss of b
MV_RL_AC;
LDIAC;
3;//start address of P
MV_RP_AC;
LDIAC;
4;//start address of Q
MV_RQ_AC;
LDIAC;//set current P,Q,R
3;//start address of P
MV_R_AC;
STIR;
9;//current address of P
LDIAC;
4;//start address of Q
MV_R_AC;
STIR;
10;//current address of Q
LDIAC;
5;//start address of R
MV_R_AC;
STIR;
11;//current address of R
CLAC;
MV_R_AC;
MV_RC_AC;
MV_AC_RP;//loop starts
LDAC;
MV_R1_AC;
MV_AC_RQ;
LDAC;
MUL;
```

```
ADD;
MV_R_AC;
MV_AC_RL;
SUB;
JMPNZ;
28;//loop again
LDIAC;
11;//current address of R
STR;
INCAC;
MV_R_AC;
STIR;
11;//current addressof R
MV_AC_RQ;
MV_RC_AC;
LDIAC;
7;// end address of Q
INCAC;//this is because current Q is increased by one
SUB;
JMPNZ;
57;// b != end
JUMP;
62;//b == end
LDIAC;//b!=end starts here
9;// current address of P
MV_RP_AC;
JUMP;
25;// jump to CLAC before loop begin
MV_AC_RP;//b==end starts here
MV_R_AC;//to store current P value
STIR;
9;// current address of P
MV_RC_AC;//to subtract
LDIAC;
6;//end address P
INCAC;
SUB;
JMPNZ;
75;// jump to a != end
JUMP;
80;// jump to a == end
LDIAC;//a!=end starts here
4;//start address Q
MV_RQ_AC;
JUMP;
25;// jump to CLAC before loop begin
ENDOP;//a==end starts here
```

9.6.2 Example Input random matrix file

```
10
12
4
5 //number of cores

//matrix A
0 4 4 0 -4 -2 3 -3 3 -3 -5 -2
5 5 -4 5 -3 -3 5 5 5 -1 5 5
4 4 5 5 0 4 -3 -4 -3 -2 1 4
4 -3 0 -2 1 2 -2 -2 0 -1 -4 -4
0 5 2 -2 2 4 -1 3 4 -5 5 5
-5 -3 -3 -2 4 -1 -4 -5 -1 5 0 -5
-3 -3 5 -3 4 2 -5 -5 -3 -5 -3 2
-4 -4 0 -2 5 1 2 -1 -1 1 0 3
4 -1 1 -5 5 1 3 -4 3 5 -3 2
0 2 -1 -3 5 -3 -4 -5 4 4 1 3

//matrix B
-4 -1 -1 5
-2 -5 -4 0
0 0 2 -5
-2 4 4 1
2 0 -1 2
-4 4 -5 4
-2 -2 1 -4
-4 0 2 3
-5 3 5 -3
```

```
0 2 -3 0
-4 4 3 -4
0 -5 3 1
```

9.6.3 Example answer matrix given by the multi-core processor in FPGA board

```
     3   -29     6   -30
   -6d   -18    4e    -3
   -11    -b    -f    19
    10    11   -26    2a
   -44   -12    18    -6
    4b    30   -2d    -9
    51   -11   -13     1
    27     0     3    -d
     9   -18   -24     5
    1c   -11    -6    -c
```

9.6.4 Example answer matrix given by the python calculation on PC

```
     3   -29     6   -30
   -6d   -18    4e    -3
   -11    -b    -f    19
    10    11   -26    2a
   -44   -12    18    -6
    4b    30   -2d    -9
    51   -11   -13     1
    27     0     3    -d
     9   -18   -24     5
    1c   -11    -6    -c
```

9.7 Multi-core processor related Verilog modules

9.7.1 Top module - toFpga.v

```
module toFpga (
    input CLOCK_50,
    input [1:0]KEY,
    input UART_RXD,
  output UART_TXD,
    output [1:0]LEDG,
    output [6:0]HEX0,HEX1,HEX2, HEX3, HEX4,HEX5, HEX6, HEX7
);

localparam CORE_COUNT = 2;
localparam REG_WIDTH = 12;
localparam DATA_MEM_WIDTH = CORE_COUNT * REG_WIDTH;
localparam INS_WIDTH = 8;
localparam INS_MEM_DEPTH = 256;
localparam DATA_MEM_DEPTH = 4096;
localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);

localparam BAUD_RATE = 115200;
localparam UART_WIDTH = 8;

localparam  idle = 3'd0,
            uart_receive_Imem = 3'd1,
            uart_receive_dmem = 3'd2,
            process_ready = 3'd3,
            process_exicute = 3'd4,
            uart_transmit_dmem = 3'd5,
            finish = 3'd6;

////// logic related to data memory //////////
wire [DATA_MEM_WIDTH-1:0]DataMemOut,DataMemIn, processor_DataOut, uart_DataOut;
wire [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr, uart_dataMemAddr, processor_dataMemAddr;

```

```
////// logic related to instruction memory ////////////
wire [INS_WIDTH-1:0]InsMemOut, InsMemIn;
wire [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,uart_InsMemAddr, processor_InsMemAddr;

////// other logics ////////////
wire dataMemWrEn,processor_DataMemWrEn, uart_dataMemWrEn;
wire uart_InsMemWrEn;
wire rstN, clk, start;
wire processStartN, processDone, processor_ready;

wire txReady, rxReady;
wire new_byte_indicate, new_ins_byte_indicate, new_data_byte_indicate;
wire txByteStart;

wire [UART_WIDTH-1:0]byteForTx, byteFromRx;

wire uart_DataMem_transmitted, uart_DataMem_received, uart_InsMem_received;
wire uart_dmem_start_transmit;

////////////////////////////////////
assign LEDG[1] = processDone;
assign LEDG[0] = processor_ready;

reg [2:0]currentState, nextState;

always @(posedge clk) begin
    if (~rstN) begin
        currentState <= idle;
    end
    else begin
        currentState <= nextState;
    end
end

always @(*) begin
    nextState = currentState;

    case (currentState)
        idle: begin     // start state
            if (~start) begin
                nextState = uart_receive_Imem;
            end
        end

        uart_receive_Imem: begin  // send the instructions (machine_code) from PC through UART
            if (uart_InsMem_received) begin
                nextState = uart_receive_dmem;
            end
        end

        uart_receive_dmem: begin  //send the data memory values from PC through UART
            if (uart_DataMem_received) begin
                nextState = process_exicute;
            end
        end

        process_exicute: begin  // processor exicute program (matrix multiplication)
            if (processDone) begin
                nextState = uart_transmit_dmem;
            end
        end

        uart_transmit_dmem: begin   // send the answer matrix to PC through UART
            if (uart_DataMem_transmitted) begin
                nextState = finish;
            end
        end

        finish: begin  //End of the process

        end

        default : nextState = idle;

    endcase
end

assign clk = CLOCK_50;
assign rstN = KEY[0];
```

```
assign start = KEY[1];
assign processStartN = ((currentState == uart_receive_dmem) && (uart_DataMem_received))? 1'b0: 1'b1;
assign uart_dmem_start_transmit = ((currentState == process_exicute) && (processDone))? 1'b0: 1'b1;

assign dataMemWrEn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )? uart_dataMemWrEn:
                    (currentState == process_exicute)? processor_DataMemWrEn:
                    1'b0;

assign dataMemAddr = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )? uart_dataMemAddr:
                    (currentState == process_exicute)? processor_dataMemAddr:
                    {DATA_MEM_ADDR_WIDTH{1'b0}};

assign DataMemIn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )? uart_DataOut:
                    (currentState == process_exicute)? processor_DataOut:
                    {DATA_MEM_WIDTH{1'b0}};

assign insMemAddr = (currentState == uart_receive_Imem)? uart_InsMemAddr:
                    (currentState == process_exicute)? processor_InsMemAddr:
                    {INS_MEM_ADDR_WIDTH{1'b0}};

assign new_ins_byte_indicate = (currentState == uart_receive_Imem)? new_byte_indicate: 1'b0;
assign new_data_byte_indicate = (currentState == uart_receive_dmem)? new_byte_indicate: 1'b0;


multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
                    .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
                    multi_core_processor(.clk(clk),.rstN(rstN),.startN(processStartN),
                    .ProcessorDataIn(DataMemOut), .InsMemOut(InsMemOut), .ProcessorDataOut(processor_DataOut),
                    .insMemAddr(processor_InsMemAddr), .dataMemAddr(processor_dataMemAddr),
                    .DataMemWrEn(processor_DataMemWrEn), .done(processDone), .ready(processor_ready));


////////////instantiation of memory modules for data and instruction memory

RAM #(.DATA_WIDTH(INS_WIDTH), .ADDR_WIDTH(INS_MEM_ADDR_WIDTH), .DEPTH(INS_MEM_DEPTH))
                IM(.address(insMemAddr), .clk(clk), .dataIn(InsMemIn), .wrEn(uart_InsMemWrEn),
                .dataOut(InsMemOut)); // size = (256 x 8)

RAM #(.DATA_WIDTH(DATA_MEM_WIDTH), .ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .DEPTH(DATA_MEM_DEPTH))
                DM(.address(dataMemAddr), .clk(clk), .dataIn(DataMemIn), .wrEn(dataMemWrEn),
                .dataOut(DataMemOut)); // size = (4096 x 48)

uart_mem_interface #(.MEM_WORD_LENGTH(DATA_MEM_WIDTH), .MEM_ADDR_LENGTH(DATA_MEM_ADDR_WIDTH),
            .UART_WIDTH(UART_WIDTH)) dataMemInterface
            (.clk(clk), .rstN(rstN), .txStart(uart_dmem_start_transmit),
            .mem_transmitted(uart_DataMem_transmitted), .mem_received(uart_DataMem_received),
            ///input outputs with memory
            .dataFromMem(DataMemOut), .memWrEn(uart_dataMemWrEn), .mem_address(uart_dataMemAddr),
            .dataToMem(uart_DataOut),
            ///inputs outputs with uart system
            .txByteReady(txReady), .rxByteReady(rxReady), .new_rx_byte_indicate(new_data_byte_indicate),
            .ByteFromUart(byteFromRx), .uartTxStart(txByteStart), .byteToUart(byteForTx),
            //select start end mem addresses of tx and rx
            .tx_start_addr_in(uartMemory[1]), .tx_end_addr_in(uartMemory[2]),
            .rx_end_addr_in(uartMemory[0]), .toggle_addr_range(1'b1));

uart_mem_interface #(.MEM_WORD_LENGTH(INS_WIDTH), .MEM_ADDR_LENGTH(INS_MEM_ADDR_WIDTH),
            .UART_WIDTH(UART_WIDTH)) ImemInterface
            (.clk(clk), .rstN(rstN), .txStart(1'b1), .mem_transmitted(),
            .mem_received(uart_InsMem_received),
            ///input outputs with memory
            .dataFromMem(), .memWrEn(uart_InsMemWrEn), .mem_address(uart_InsMemAddr),
            .dataToMem(InsMemIn),
            ///inputs outputs with uart system
            .txByteReady(txReady), .rxByteReady(rxReady), .new_rx_byte_indicate(new_ins_byte_indicate),
            .ByteFromUart(byteFromRx), .uartTxStart(), .byteToUart(),
            //select start end mem addresses of tx and rx
            .tx_start_addr_in(), .tx_end_addr_in(), .rx_end_addr_in(),
            .toggle_addr_range(1'b0));

uart_system #(.DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE))
            US(.clk(clk), .rstN(rstN),.txByteStart(txByteStart), .rx(UART_RXD),
                .byteForTx(byteForTx), .tx(UART_TXD), .txReady(txReady) ,.rxReady(rxReady),
                .new_byte_indicate(new_byte_indicate), .byteFromRx(byteFromRx));

```

```
localparam  Q_end_addr_location = 12'd7,
            R_start_addr_location = 12'd5,
            R_end_addr_location = 12'd8;

reg [REG_WIDTH-1:0]uartMemory[2:0]; //0- end address of Q, 1- start address of R, 2- end address of R

always @(posedge clk) begin
    if (uart_dataMemWrEn) begin
        if (uart_dataMemAddr == Q_end_addr_location)
            uartMemory[0] <= uart_DataOut[REG_WIDTH-1:0];
        else if (uart_dataMemAddr == R_start_addr_location)
            uartMemory[1] <= uart_DataOut[REG_WIDTH-1:0];
        else if (uart_dataMemAddr == R_end_addr_location)
            uartMemory[2] <= uart_DataOut[REG_WIDTH-1:0];
    end
end

///////////////to count the time taken to the process
wire [25:0]currentTime;
timeCounter TC(.clk(clk), .rstN(rstN), .startN(processStartN), .stop(processDone),
                .timeDuration(currentTime));

////////////////////// to show current state & no. of clock cycles on the 8 seven segments
hex_display HD(.clk(clk), .rstN(rstN), .state(currentState),
            .start_timeValue_convetion(~uart_DataMem_transmitted), .timeValue(currentTime),
            .out0(HEX0), .out1(HEX1), .out2(HEX2),.out3(HEX3),.out4(HEX4),
            .out5(HEX5), .out6(HEX6), .out7(HEX7));

endmodule //toFpga
```

9.7.2 Memory

Module - RAM.v

```
module RAM
#(
    parameter DATA_WIDTH = 12,
    parameter DEPTH = 256,
    parameter ADDR_WIDTH = $clog2(DEPTH)  // 4096 locations
)

(
    input clk,wrEn,
    input [DATA_WIDTH-1:0]dataIn,
    input [ADDR_WIDTH-1:0]address,
    output [DATA_WIDTH-1:0]dataOut
);

reg [DATA_WIDTH-1:0] memory [0:DEPTH-1];
reg [ADDR_WIDTH-1:0] addr_reg;

always @(posedge clk) begin
    addr_reg <= address;
    if (wrEn) begin
        memory[address] <= dataIn;
    end
end

assign dataOut = memory[addr_reg];

endmodule //RAM
```

Testbench - RAM_tb.v

9.7.3 Multi-core processor

Module - multi_core_processor.v

```
module multi_core_processor #(
```

```
    parameter REG_WIDTH = 12,
    parameter INS_WIDTH = 8,
    parameter CORE_COUNT = 4,
    parameter DATA_MEM_ADDR_WIDTH = 12,
    parameter INS_MEM_ADDR_WIDTH = 8
)
(
    input clk,rstN,startN,
    input [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataIn,
    input [INS_WIDTH-1:0]InsMemOut,
    output [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataOut,
    output [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,
    output [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr,
    output DataMemWrEn, done,ready
);

wire core_DataMemWrEn[0:CORE_COUNT-1];
wire core_done[0:CORE_COUNT-1];
wire core_ready[0:CORE_COUNT-1];
wire [DATA_MEM_ADDR_WIDTH-1:0] core_dataMemAddr[0:CORE_COUNT-1];
wire [INS_MEM_ADDR_WIDTH-1:0] core_InsMemAddr[0:CORE_COUNT-1];

genvar i;
generate
    for (i=0;i<CORE_COUNT; i=i+1) begin:core
        processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH),
            .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH)) CPU
            (.clk(clk), .rstN(rstN), .startN(startN), .ProcessorDataIn(ProcessorDataIn[REG_WIDTH*i+:
    REG_WIDTH]),
            .InsMemOut(InsMemOut), .dataMemAddr(core_dataMemAddr[i]),
            .ProcessorDataOut(ProcessorDataOut[REG_WIDTH*i+:REG_WIDTH]), .insMemAddr(core_InsMemAddr[i]),
            .DataMemWrEn(core_DataMemWrEn[i]), .done(core_done[i]), .ready(core_ready[i]) );
    end
endgenerate

assign dataMemAddr = core_dataMemAddr[0];
assign DataMemWrEn = core_DataMemWrEn[0];
assign insMemAddr = core_InsMemAddr[0];
assign ready = core_ready[0];
assign done = core_done[0];

endmodule //multi_core_processor
```

Testbench - multi_core_processor_tb.v

```
`timescale 1ns/1ns
module multi_core_processor_tb ();

localparam CLK_PERIOD = 20;
reg clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam CORE_COUNT = 3;
localparam REG_WIDTH = 12;
localparam INS_WIDTH = 8;
localparam INS_MEM_DEPTH = 256;
localparam DATA_MEM_DEPTH = 4096;
localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);
localparam DATA_MEM_WIDTH = REG_WIDTH*CORE_COUNT;

reg rstN,startN;
reg [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataIn;
reg [INS_WIDTH-1:0]InsMemOut;
wire [REG_WIDTH*CORE_COUNT-1:0]ProcessorDataOut;
wire [INS_MEM_ADDR_WIDTH-1:0]insMemAddr;
wire [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr;
wire DataMemWrEn, done,ready;

multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
                .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))dut
                (.clk(clk), .rstN(rstN), .startN(startN), .ProcessorDataIn(ProcessorDataIn),
```

```
                .InsMemOut(InsMemOut), .ProcessorDataOut(ProcessorDataOut), .insMemAddr(insMemAddr),
                .dataMemAddr(dataMemAddr), .DataMemWrEn(DataMemWrEn), .done(done), .ready(ready));

///// initialize instruction and data memory /////////

reg [INS_WIDTH-1:0]ins_mem[0:INS_MEM_DEPTH-1];
reg [DATA_MEM_WIDTH-1:0]data_mem[0:DATA_MEM_DEPTH-1];

initial begin
    $readmemb("../../9_ins_mem_tb.txt", ins_mem);
    $readmemb("../../4_data_mem_tb.txt", data_mem);
end
////////////// read data and instruction memory /////////
always @(posedge clk) begin
    ProcessorDataIn  <= data_mem[dataMemAddr];
    InsMemOut <= ins_mem[insMemAddr];
end

////////// write data memory //////////
always @(posedge clk) begin
    if (DataMemWrEn) begin
        data_mem[dataMemAddr] <= ProcessorDataOut;
    end
end


///// start the simulation

initial begin
    @(posedge clk);
    rstN <= 1'b0;
    startN <= 1'b1;
    @(posedge clk);
    rstN <= 1'b1;
    startN <= 1'b0;
    @(posedge clk);
    startN <= 1'b1;
end



////////////// verification of the simulation correctness /////////

localparam  Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
            R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
            R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);

reg [REG_WIDTH-1:0] a, b, c, P_start_addr, Q_start_addr, R_start_addr, P_end_addr, Q_end_addr, R_end_addr;

reg disp_temp;
always @(posedge clk) begin
    if (done) begin
        a = data_mem[12'd0][REG_WIDTH-1:0];
        b = data_mem[12'd1][REG_WIDTH-1:0];
        c = data_mem[12'd2][REG_WIDTH-1:0];
        P_start_addr = data_mem[12'd3][REG_WIDTH-1:0];
        Q_start_addr = data_mem[12'd4][REG_WIDTH-1:0];
        R_start_addr = data_mem[12'd5][REG_WIDTH-1:0];
        P_end_addr = data_mem[12'd6][REG_WIDTH-1:0];
        Q_end_addr = data_mem[12'd7][REG_WIDTH-1:0];
        R_end_addr = data_mem[12'd8][REG_WIDTH-1:0];

        $writememb("../../11_data_mem_out.txt", data_mem); // write data memory to file

        $display("\nMatrix P\n");
        disp_temp = print_matrix_P(a, b, P_start_addr, P_end_addr, CORE_COUNT);

        $display("\nMatrix Q\n");
        disp_temp = print_matrix_Q(b, c, Q_start_addr, Q_end_addr);

        $display("\nMatrix R\n");
        disp_temp = print_matrix_R(a, c, R_start_addr, R_end_addr, CORE_COUNT);

        repeat(5) @(posedge clk);  // end of the simulation
        $stop;
    end
end
```

```
function automatic print_matrix_P (input [REG_WIDTH-1:0]a,b,P_start_addr,P_end_addr,
                                   input integer CORE_COUNT);
    integer d = (a%CORE_COUNT == 0)? a/CORE_COUNT : a/CORE_COUNT+1;
    integer x,y,z;
    for (x=0;x<d;x=x+1)  begin
        for (y=CORE_COUNT;y>0;y=y-1) begin :core_count_loop
            if ((x+1)*CORE_COUNT-y>= a) begin
                disable core_count_loop;
            end
            for (z=0;z<b;z=z+1) begin : print_val_P
                reg [DATA_MEM_WIDTH-1:0]temp_1 = data_mem[(P_start_addr + x*b+z)];
                reg [REG_WIDTH-1:0]temp_2 = temp_1[(y*REG_WIDTH-1) -:REG_WIDTH];
                $write("%h ", temp_2);
            end
            $write("\n");
        end
    end
endfunction

function automatic print_matrix_Q(input [REG_WIDTH-1:0]b,c,Q_start_addr,Q_end_addr);
    integer i,j;
    for (i=Q_start_addr;i<Q_start_addr+b;i=i+1) begin
        for (j=i;j<=Q_end_addr;j=j+b) begin :print_val_Q
            reg [DATA_MEM_WIDTH-1:0] temp_1 = data_mem[j];
            $write("%h ", temp_1[REG_WIDTH-1:0]);
        end
        $write("\n");
    end
endfunction

function automatic print_matrix_R(input [REG_WIDTH-1:0]a,c,R_start_addr,R_end_addr,
                                  input integer CORE_COUNT);
    integer d = (a%CORE_COUNT == 0)? a/CORE_COUNT : a/CORE_COUNT+1;

    integer x,y,z;
    for (x=0;x<d;x=x+1) begin
        for (y=CORE_COUNT;y>0;y=y-1) begin : core_count_loop
            if ((x+1)*CORE_COUNT-y>= a) begin
                disable core_count_loop;
            end
            for (z=0;z<c;z=z+1) begin : print_val_R
                reg [DATA_MEM_WIDTH-1:0]temp_1 = data_mem[(R_start_addr + x*c+z)];
                reg [REG_WIDTH-1:0]temp_2 = temp_1[(y*REG_WIDTH-1) -:REG_WIDTH];
                $write("%h ", temp_2);
            end
            $write("\n");
        end
    end
endfunction




endmodule //multi_core_processor_tb
```

9.7.4 Single-core processor

Module - processor.v

```
module processor
#(
    parameter REG_WIDTH = 12,
    parameter INS_WIDTH = 8,
    parameter DATA_MEM_ADDR_WIDTH = 12,
    parameter INS_MEM_ADDR_WIDTH = 8
)
(
    input clk,rstN,startN,
    input [REG_WIDTH-1:0]ProcessorDataIn,
    input [INS_WIDTH-1:0]InsMemOut,
    output [REG_WIDTH-1:0]ProcessorDataOut,
    output [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr,
    output [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,
    output DataMemWrEn,
    output done,ready
```

```
);

wire [REG_WIDTH-1:0]alu_a, alu_b, alu_out;
wire [2:0]select_alu_op;
wire [3:0]busSel;
wire [REG_WIDTH-1:0]busOut;
wire [INS_WIDTH-1:0]IRout;
wire [REG_WIDTH-1:0] Rout, RLout, RCout, RPout, RQout, R1out, ACout;
wire Zout, ZWrEn;
wire [3:0]incReg;     // {PC, RC, RP, RQ}
wire [9:0]wrEnReg;    // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}

//instantiation of each module within the processor are as follows.

controlUnit #(.INS_WIDTH(INS_WIDTH)) CU(.clk(clk), .rstN(rstN), .startN(startN), .Zout(Zout), .ins(IRout),
                .aluOp(select_alu_op), .incReg(incReg), .wrEnReg(wrEnReg), .busSel(busSel),
                .DataMemWrEn(DataMemWrEn), .ZWrEn(ZWrEn), .done(done), .ready(ready));

ALU #(.WIDTH(REG_WIDTH)) alu(.a(alu_a), .b(alu_b), .selectOp(select_alu_op), .dataOut(alu_out));

multiplexer #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH)) mux(.selectIn(busSel), .DMem(ProcessorDataIn),
                .R(Rout), .RL(RLout), .RC(RCout), .RP(RPout), .RQ(RQout), .R1(R1out), .AC(ACout),
                .IR(IRout), .busOut(busOut));

register #(.WIDTH(REG_WIDTH)) AR(.dataIn(busOut), .wrEn(wrEnReg[9]), .rstN(rstN), .clk(clk),
                .dataOut(dataMemAddr));

register #(.WIDTH(REG_WIDTH)) R(.dataIn(busOut), .wrEn(wrEnReg[8]), .rstN(rstN), .clk(clk),
                .dataOut(Rout));

incRegister #(.WIDTH(INS_WIDTH)) PC(.dataIn(IRout), .wrEn(wrEnReg[7]), .rstN(rstN), .clk(clk),
                .incEn(incReg[3]), .dataOut(insMemAddr));

register #(.WIDTH(INS_WIDTH)) IR(.dataIn(InsMemOut), .wrEn(wrEnReg[6]), .rstN(rstN),
                .clk(clk), .dataOut(IRout));

register #(.WIDTH(REG_WIDTH)) RL(.dataIn(busOut), .wrEn(wrEnReg[5]), .rstN(rstN), .clk(clk),
                .dataOut(RLout));

incRegister #(.WIDTH(REG_WIDTH)) RC(.dataIn(busOut), .wrEn(wrEnReg[4]), .rstN(rstN),
                .clk(clk), .incEn(incReg[2]), .dataOut(RCout));

incRegister #(.WIDTH(REG_WIDTH)) RP(.dataIn(busOut), .wrEn(wrEnReg[3]), .rstN(rstN),
                .clk(clk), .incEn(incReg[1]), .dataOut(RPout));

incRegister #(.WIDTH(REG_WIDTH)) RQ(.dataIn(busOut), .wrEn(wrEnReg[2]), .rstN(rstN),
                .clk(clk), .incEn(incReg[0]), .dataOut(RQout));

register #(.WIDTH(REG_WIDTH)) R1(.dataIn(busOut), .wrEn(wrEnReg[1]), .rstN(rstN), .clk(clk),
                .dataOut(R1out));

register #(.WIDTH(REG_WIDTH)) AC(.dataIn(alu_out), .wrEn(wrEnReg[0]), .rstN(rstN), .clk(clk),
                .dataOut(ACout));

zReg #(.WIDTH(REG_WIDTH)) Z(.dataIn(alu_out), .clk(clk), .rstN(rstN), .wrEn(ZWrEn), .Zout(Zout));

// necessary wires are routed as follows among modules as below.
assign ProcessorDataOut = Rout;
assign alu_a = ACout;
assign alu_b = busOut;

endmodule //processor
```

Testbench - processor_tb.v

```
`timescale 1ns/1ps
module processor_tb ();

localparam CLK_PERIOD = 20;
reg clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

```

```
localparam REG_WIDTH = 12;
localparam INS_WIDTH = 8;
localparam INS_MEM_DEPTH = 256;
localparam DATA_MEM_DEPTH = 4096;
localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);

reg rstN,startN;
reg [REG_WIDTH-1:0]ProcessorDataIn;
reg [INS_WIDTH-1:0]InsMemOut;
wire  [REG_WIDTH-1:0]dataMemAddr;
wire [REG_WIDTH-1:0] ProcessorDataOut;
wire  [INS_WIDTH-1:0]insMemAddr;
wire  DataMemWrEn;
wire  done,ready;

processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH),
            .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH))dut(
            .clk(clk), .rstN(rstN), .startN(startN), .ProcessorDataIn(ProcessorDataIn),
            .InsMemOut(InsMemOut), .ProcessorDataOut(ProcessorDataOut), .dataMemAddr(dataMemAddr),
            .insMemAddr(insMemAddr), .DataMemWrEn(DataMemWrEn), .done(done), .ready(ready));

///// initialize instruction and data memory /////////

reg [INS_WIDTH-1:0]ins_mem[0:INS_MEM_DEPTH-1];
reg [REG_WIDTH-1:0]data_mem[0:DATA_MEM_DEPTH-1];

initial begin
    $readmemb("../../9_ins_mem_tb.txt", ins_mem);
    $readmemb("../../4_data_mem_tb.txt", data_mem);
end

////////////// read data and instruction memory /////////
always @(posedge clk) begin
    ProcessorDataIn  <= data_mem[dataMemAddr];
    InsMemOut <= ins_mem[insMemAddr];
end

/////////// write data memory //////////
always @(posedge clk) begin
    if (DataMemWrEn) begin
        data_mem[dataMemAddr] <= ProcessorDataOut;
    end
end

///// start the simulation

initial begin
    @(posedge clk);
    rstN <= 1'b0;
    startN <= 1'b1;
    @(posedge clk);
    rstN <= 1'b1;
    startN <= 1'b0;
    @(posedge clk);
    startN <= 1'b1;
end

////////////// verification of the simulation correctness /////////

localparam  Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
            R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
            R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);

reg [REG_WIDTH-1:0] a, b, c, P_start_addr, Q_start_addr, R_start_addr, P_end_addr, Q_end_addr, R_end_addr;

reg disp_temp;
always @(posedge clk) begin
    if (done) begin
        a = data_mem[12'd0][REG_WIDTH-1:0];
        b = data_mem[12'd1][REG_WIDTH-1:0];
        c = data_mem[12'd2][REG_WIDTH-1:0];
        P_start_addr = data_mem[12'd3][REG_WIDTH-1:0];
        Q_start_addr = data_mem[12'd4][REG_WIDTH-1:0];
        R_start_addr = data_mem[12'd5][REG_WIDTH-1:0];
        P_end_addr = data_mem[12'd6][REG_WIDTH-1:0];
        Q_end_addr = data_mem[12'd7][REG_WIDTH-1:0];
        R_end_addr = data_mem[12'd8][REG_WIDTH-1:0];

```

```
        $writememb("../../11_data_mem_out.txt", data_mem); // write data memory to a text file

        $display("\nMatrix P\n");
        disp_temp = print_matrix_P(a, b, P_start_addr, P_end_addr);

        $display("\nMatrix Q\n");
        disp_temp = print_matrix_Q(b, c, Q_start_addr, Q_end_addr);

        $display("\nMatrix R\n");
        disp_temp = print_matrix_R(a, c, R_start_addr, R_end_addr);

        repeat(5) @(posedge clk);  // end of the simulation
        $stop;
    end
end

function automatic print_matrix_P(input [REG_WIDTH-1:0]a,b,P_start_addr,P_end_addr);
    integer i,j;
    for (i=P_start_addr;i<P_end_addr;i=i+b) begin
        for (j=i;j<i+b;j=j+1) begin
            $write("%h ", data_mem[j]);
        end
        $write("\n");
    end
endfunction

function automatic print_matrix_Q(input [REG_WIDTH-1:0]b,c,Q_start_addr,Q_end_addr);
    integer i,j;
    for (i=Q_start_addr;i<Q_start_addr+b;i=i+1) begin
        for (j=i;j<=Q_end_addr;j=j+b) begin
            $write("%h ", data_mem[j]);
        end
        $write("\n");
    end
endfunction

function automatic print_matrix_R(input [REG_WIDTH-1:0]a,c,R_start_addr,R_end_addr);
    integer i,j;
    for (i=R_start_addr;i<R_end_addr;i=i+c) begin
        for (j=i;j<i+c;j=j+1) begin
            $write("%h ", data_mem[j]);
        end
        $write("\n");
    end
endfunction

endmodule // processor_tb
```

9.7.5 Control Unit

Module - controlUnit.v

```
module controlUnit
#(
    parameter INS_WIDTH = 8
)
(
    input clk,rstN,startN,Zout,
    input [INS_WIDTH-1:0]ins,
    output reg [2:0]aluOp,
    output reg [3:0]incReg,    // {PC, RC, RP, RQ}
    output reg [9:0]wrEnReg,   // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
    output reg [3:0]busSel,
    output reg DataMemWrEn,ZWrEn,
    output done,ready
);

// all the states (47) for the instructions are as follows

localparam  IDLE = 6'd0,  //states

            NOP1 = 6'd1,

            ENDOP1 = 6'd2,

```

```
            CLAC1 = 6'd3,

            FETCH_DELAY1 = 6'd37,     /////(extra_delay for memory read)
            FETCH1 = 6'd4,
            FETCH2 = 6'd5,

            LDIAC_DELAY1 =  6'd38,     ///// (extra_delay for memory read)
            LDIAC1 = 6'd6,
            LDIAC2 = 6'd7,
            LDIAC_DELAY2 = 6'd39,       //// (extra_delay for memory read)
            LDIAC3 = 6'd8,

            LDAC1 = 6'd9,
            LDAC_DELAY1 = 6'd40,       ///// (extra_delay for memory read)
            LDAC2 = 6'd10,

            STIR_DELAY1 = 6'd41,       ///// (extra_delay for memory read)
            STIR1 = 6'd11,
            STIR2 = 6'd12,
            STIR_DELAY2 = 6'd42,       ///// (extra_delay for memory write)
            STIR3 = 6'd13,

            STR1 = 6'd14,
            STR_DELAY1 = 6'd43,         ///// (extra_delay for memory write)
            STR2 = 6'd15,

            JUMP_DELAY1 = 6'd44,        ///// (extra_delay for memory read)
            JUMP1 = 6'd16,
            JUMP2 = 6'd17,

            JMPNZY_DILAY1 = 6'd45,      ///// (extra_delay for memory read)
            JMPNZY1 = 6'd18,
            JMPNZY2 = 6'd19,
            JMPNZN1 = 6'd20,

            JMPZY_DELAY1 = 6'd46,       ///// (extra_delay for memory read)
            JMPZY1 = 6'd21,
            JMPZY2 = 6'd22,
            JMPZN1 = 6'd23,

            MUL1 = 6'd24,

            ADD1 = 6'd25,

            SUB1 = 6'd26,

            INCAC1 = 6'd27,

            MV_RL_AC1 = 6'd28,

            MV_RP_AC1 = 6'd29,

            MV_RQ_AC1 = 6'd30,

            MV_RC_AC1 = 6'd31,

            MV_R_AC1 = 6'd32,

            MV_R1_AC1 = 6'd33,

            MV_AC_RP1 = 6'd34,

            MV_AC_RQ1 = 6'd35,

            MV_AC_RL1 = 6'd36;

localparam  clr_alu  =  3'd0,   //alu operations
            pass_alu =  3'd1,
            add_alu  =  3'd2,
            sub_alu  =  3'd3,
            mul_alu  =  3'd4,
            inc_alu  =  3'd5,
            idle_alu =  3'd6;

localparam  DMem_bus = 4'b0,   //multiplexer
            R_bus    = 4'd1,
            IR_bus   = 4'd2,
            RL_bus   = 4'd3,
            RC_bus   = 4'd4,
```

```
            RP_bus   = 4'd5,
            RQ_bus   = 4'd6,
            R1_bus   = 4'd7,
            AC_bus   = 4'd8,
            idle_bus = 4'd9;

localparam   //instruction set (assemly instructions and their machine codes)
            NOP      =  8'd0,
            ENDOP    =  8'd1,
            CLAC     =  8'd2,
            LDIAC    =  8'd3,
            LDAC     =  8'd4,
            STR      =  8'd5,
            STIR     =  8'd6,
            JUMP     =  8'd7,
            JMPNZ    =  8'd8,
            JMPZ     =  8'd9,
            MUL      =  8'd10,
            ADD      =  8'd11,
            SUB      =  8'd12,
            INCAC    =  8'd13,
            MV_RL_AC =  {4'd1,4'd15},
            MV_RP_AC =  {4'd2,4'd15},
            MV_RQ_AC =  {4'd3,4'd15},
            MV_RC_AC =  {4'd4,4'd15},
            MV_R_AC  =  {4'd5,4'd15},
            MV_R1_AC =  {4'd6,4'd15},
            MV_AC_RP =  {4'd7,4'd15},
            MV_AC_RQ =  {4'd8,4'd15},
            MV_AC_RL =  {4'd9,4'd15};

localparam [9:0]
    no_wrEn = 10'b0000000000,
    AR_wrEn = 10'b1000000000,
    R_wrEn  = 10'b0100000000,
    PC_wrEn = 10'b0010000000,
    IR_wrEn = 10'b0001000000,
    RL_wrEn = 10'b0000100000,
    RC_wrEn = 10'b0000010000,
    RP_wrEn = 10'b0000001000,
    RQ_wrEn = 10'b0000000100,
    R1_wrEn = 10'b0000000010,
    AC_wrEn = 10'b0000000001;

localparam [3:0]
    no_inc = 4'b0000,
    PC_inc = 4'b1000,
    RC_inc = 4'b0100,
    RP_inc = 4'b0010,
    RQ_inc = 4'b0001,
    RC_RP_RQ_inc = 4'b0111;

reg [5:0]currentState, nextState;

always @(posedge clk) begin
    if (~rstN) begin
        currentState <= IDLE;
    end
    else begin
        currentState <= nextState;
    end
end

//all the control signals for each state are set as below
always @(startN, Zout, ins, currentState) begin
    case (currentState)
        IDLE: begin
            aluOp <= idle_alu;
            incReg <= 4'd0;
            wrEnReg <= 10'd0;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;

            if (~startN)
                nextState <= FETCH1;
            else
                nextState <= IDLE;
        end
```

```
182
183         NOP1: begin
184             aluOp <= idle_alu;
185             incReg <= 4'd0;
186             wrEnReg <= 10'd0;
187             busSel <= idle_bus;
188             DataMemWrEn <= 1'b0;
189             ZWrEn <= 1'b0;
190             nextState <= FETCH1;
191         end
192
193         ENDOP1: begin
194             aluOp <= idle_alu;
195             incReg <= 4'd0;
196             wrEnReg <= 10'd0;
197             busSel <= DMem_bus;
198             DataMemWrEn <= 1'b0;
199             ZWrEn <= 1'b0;
200             nextState <= ENDOP1;
201         end
202
203         CLAC1: begin
204             aluOp <= clr_alu;
205             incReg <= 4'd0;
206             wrEnReg <= 10'd1;
207             busSel <= idle_bus;
208             DataMemWrEn <= 1'b0;
209             ZWrEn <= 1'b1;
210             nextState <= FETCH1;
211         end
212
213         FETCH_DELAY1: begin
214             aluOp <= idle_alu;
215             incReg <= 4'd0;
216             wrEnReg <= IR_wrEn;
217             busSel <= idle_bus;
218             DataMemWrEn <= 1'b0;
219             ZWrEn <= 1'b0;
220             nextState <= FETCH1;
221         end
222
223         FETCH1: begin
224             aluOp <= idle_alu;
225             incReg <= 4'd0;
226             wrEnReg <= IR_wrEn;
227             busSel <= idle_bus;
228             DataMemWrEn <= 1'b0;
229             ZWrEn <= 1'b0;
230             nextState <= FETCH2;
231         end
232
233         FETCH2: begin
234             aluOp <= idle_alu;
235             incReg <= PC_inc;
236             wrEnReg <= 10'd0;
237             busSel <= 4'd0;
238             DataMemWrEn <= 1'b0;
239             ZWrEn <= 1'b0;
240             case (ins)   //has to deside what is the next state
241
242                 NOP: nextState <= NOP1;
243                 ENDOP: nextState <= ENDOP1;
244                 CLAC: nextState <= CLAC1;
245                 LDIAC: nextState <= LDIAC_DELAY1;
246                 LDAC: nextState <= LDAC1;
247                 STR: nextState <= STR1;
248                 STIR: nextState <= STIR_DELAY1;
249                 JUMP: nextState <= JUMP_DELAY1;
250                 JMPNZ: nextState <= (Zout == 0)? JMPNZY_DILAY1 : JMPNZN1;
251                 JMPZ: nextState <= (Zout == 1)? JMPZY_DELAY1 : JMPZN1;
252                 MUL: nextState <= MUL1;
253                 ADD: nextState <= ADD1;
254                 SUB: nextState <= SUB1;
255                 INCAC: nextState <= INCAC1;
256                 MV_RL_AC: nextState <= MV_RL_AC1;
257                 MV_RP_AC: nextState <= MV_RP_AC1;
258                 MV_RQ_AC: nextState <= MV_RQ_AC1;
259                 MV_RC_AC: nextState <= MV_RC_AC1;
260                 MV_R_AC: nextState <= MV_R_AC1;
```

```
                MV_R1_AC: nextState <= MV_R1_AC1;
                MV_AC_RP: nextState <= MV_AC_RP1;
                MV_AC_RQ: nextState <= MV_AC_RQ1;
                MV_AC_RL: nextState <= MV_AC_RL1;
                default : nextState <= IDLE;

            endcase
        end

        LDIAC_DELAY1: begin
            aluOp <= idle_alu;
            incReg <= 4'd0;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= LDIAC1;
        end

        LDIAC1: begin
            aluOp <= idle_alu;
            incReg <= 4'd0;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= LDIAC2;
        end

        LDIAC2: begin
            aluOp <= idle_alu;
            incReg <= PC_inc;
            wrEnReg <= AR_wrEn;
            busSel <= IR_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= LDIAC_DELAY2;
        end

        LDIAC_DELAY2: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
            nextState <= LDIAC3;
        end

        LDIAC3: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
            nextState <= FETCH_DELAY1;
        end

        LDAC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= AR_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= LDAC_DELAY1;
        end

        LDAC_DELAY1: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
            nextState <= LDAC2;
        end
```

```
        LDAC2: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= DMem_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
            nextState <= FETCH_DELAY1;
        end

        STR1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= AR_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= STR_DELAY1;
        end

        STR_DELAY1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b1;
            ZWrEn <= 1'b0;
            nextState <= STR2;
        end

        STR2: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b1;
            ZWrEn <= 1'b0;
            nextState <= FETCH1;
        end

        STIR_DELAY1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= STIR1;
        end

        STIR1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= IR_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= STIR2;
        end

        STIR2: begin
            aluOp <= idle_alu;
            incReg <= PC_inc;
            wrEnReg <= AR_wrEn;
            busSel <= IR_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= STIR_DELAY2;
        end

        STIR_DELAY2: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= no_wrEn;
            busSel <= idle_bus;
            DataMemWrEn <= 1'b1;
            ZWrEn <= 1'b0;
            nextState <= STIR3;
        end
```

```
419
420         STIR3: begin
421             aluOp <= idle_alu;
422             incReg <= no_inc;
423             wrEnReg <= no_wrEn;
424             busSel <= idle_bus;
425             DataMemWrEn <= 1'b1;
426             ZWrEn <= 1'b0;
427             nextState <= FETCH_DELAY1;
428         end
429
430         JUMP_DELAY1: begin
431             aluOp <= idle_alu;
432             incReg <= 4'd0;
433             wrEnReg <= IR_wrEn;
434             busSel <= 4'd0;
435             DataMemWrEn <= 1'b0;
436             ZWrEn <= 1'b0;
437             nextState <= JUMP1;
438         end
439
440         JUMP1: begin
441             aluOp <= idle_alu;
442             incReg <= 4'd0;
443             wrEnReg <= IR_wrEn;
444             busSel <= 4'd0;
445             DataMemWrEn <= 1'b0;
446             ZWrEn <= 1'b0;
447             nextState <= JUMP2;
448         end
449
450         JUMP2: begin
451             aluOp <= idle_alu;
452             incReg <= 4'd0;
453             wrEnReg <= PC_wrEn;
454             busSel <= IR_bus;
455             DataMemWrEn <= 1'b0;
456             ZWrEn <= 1'b0;
457             nextState <= FETCH_DELAY1;
458         end
459
460         JMPNZY_DILAY1: begin
461             aluOp <= idle_alu;
462             incReg <= 4'd0;
463             wrEnReg <= IR_wrEn;
464             busSel <= 4'd0;
465             DataMemWrEn <= 1'b0;
466             ZWrEn <= 1'b0;
467             nextState <= JMPNZY1;
468         end
469
470         JMPNZY1: begin
471             aluOp <= idle_alu;
472             incReg <= 4'd0;
473             wrEnReg <= IR_wrEn;
474             busSel <= 4'd0;
475             DataMemWrEn <= 1'b0;
476             ZWrEn <= 1'b0;
477             nextState <= JMPNZY2;
478         end
479
480         JMPNZY2: begin
481             aluOp <= idle_alu;
482             incReg <= 4'd0;
483             wrEnReg <= PC_wrEn;
484             busSel <= IR_bus;
485             DataMemWrEn <= 1'b0;
486             ZWrEn <= 1'b0;
487             nextState <= FETCH_DELAY1;
488         end
489
490         JMPNZN1: begin
491             aluOp <= idle_alu;
492             incReg <= PC_inc;
493             wrEnReg <= no_wrEn;
494             busSel <= idle_bus;
495             DataMemWrEn <= 1'b0;
496             ZWrEn <= 1'b0;
497             nextState <= FETCH_DELAY1;
```

```
498         end
499
500         JMPZY_DELAY1: begin
501             aluOp <= idle_alu;
502             incReg <= 4'd0;
503             wrEnReg <= IR_wrEn;
504             busSel <= 4'd0;
505             DataMemWrEn <= 1'b0;
506             ZWrEn <= 1'b0;
507             nextState <= JMPZY1;
508         end
509
510         JMPZY1: begin
511             aluOp <= idle_alu;
512             incReg <= 4'd0;
513             wrEnReg <= IR_wrEn;
514             busSel <= 4'd0;
515             DataMemWrEn <= 1'b0;
516             ZWrEn <= 1'b0;
517             nextState <= JMPZY2;
518         end
519
520         JMPZY2: begin
521             aluOp <= idle_alu;
522             incReg <= 4'd0;
523             wrEnReg <= PC_wrEn;
524             busSel <= IR_bus;
525             DataMemWrEn <= 1'b0;
526             ZWrEn <= 1'b0;
527             nextState <= FETCH_DELAY1;
528         end
529
530         JMPZN1: begin
531             aluOp <= idle_alu;
532             incReg <= PC_inc;
533             wrEnReg <= no_wrEn;
534             busSel <= idle_bus;
535             DataMemWrEn <= 1'b0;
536             ZWrEn <= 1'b0;
537             nextState <= FETCH_DELAY1;
538         end
539
540         MUL1: begin
541             aluOp <= mul_alu;
542             incReg <= RC_RP_RQ_inc;
543             wrEnReg <= AC_wrEn;
544             busSel <= R1_bus;
545             DataMemWrEn <= 1'b0;
546             ZWrEn <= 1'b1;
547             nextState <= FETCH_DELAY1;
548         end
549
550         ADD1: begin
551             aluOp <= add_alu;
552             incReg <= no_inc;
553             wrEnReg <= AC_wrEn;
554             busSel <= R_bus;
555             DataMemWrEn <= 1'b0;
556             ZWrEn <= 1'b1;
557             nextState <= FETCH_DELAY1;
558         end
559
560         SUB1: begin
561             aluOp <= sub_alu;
562             incReg <= no_inc;
563             wrEnReg <= AC_wrEn;
564             busSel <= RC_bus;
565             DataMemWrEn <= 1'b0;
566             ZWrEn <= 1'b1;
567             nextState <= FETCH_DELAY1;
568         end
569
570         INCAC1: begin
571             aluOp <= inc_alu;
572             incReg <= no_inc;
573             wrEnReg <= AC_wrEn;
574             busSel <= idle_bus;
575             DataMemWrEn <= 1'b0;
576             ZWrEn <= 1'b1;
```

```
            nextState <= FETCH_DELAY1;
        end

        MV_RL_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= RL_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= FETCH_DELAY1;
        end

        MV_RP_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= RP_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= FETCH_DELAY1;
        end

        MV_RQ_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= RQ_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= FETCH_DELAY1;
        end

        MV_RC_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= RC_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= FETCH_DELAY1;
        end

        MV_R_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= R_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= FETCH_DELAY1;
        end

        MV_R1_AC1: begin
            aluOp <= idle_alu;
            incReg <= no_inc;
            wrEnReg <= R1_wrEn;
            busSel <= AC_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= FETCH_DELAY1;
        end

        MV_AC_RP1: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= RP_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
            nextState <= FETCH_DELAY1;
        end

        MV_AC_RQ1: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= RQ_bus;
            DataMemWrEn <= 1'b0;
```

```
            ZWrEn <= 1'b1;
            nextState <= FETCH_DELAY1;
        end

        MV_AC_RL1: begin
            aluOp <= pass_alu;
            incReg <= no_inc;
            wrEnReg <= AC_wrEn;
            busSel <= RL_bus;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b1;
            nextState <= FETCH_DELAY1;
        end

        default : begin
            aluOp <= idle_alu;
            incReg <= 4'd0;
            wrEnReg <= 10'd0;
            busSel <= 4'd0;
            DataMemWrEn <= 1'b0;
            ZWrEn <= 1'b0;
            nextState <= IDLE;
        end

    endcase

end

// signals to the user whether processor is ready or the process is over
assign done = (currentState == ENDOP1)?1'b1:1'b0;
assign ready = (currentState == IDLE)? 1'b1:1'b0;

endmodule //controlUnit
```

Testbench - controlUnit_tb.sv

```
`timescale 1ns/1ps

module controlUnit_tb ();

localparam CLK_PERIOD = 20;
reg clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam INS_WIDTH = 8;

reg rstN,startN,Zout;
reg [INS_WIDTH-1:0] ins;
wire [2:0] aluOp;
wire [3:0] incReg;    // {PC, RC, RP, RQ}
wire [9:0] wrEnReg;   // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
wire [3:0] busSel;
wire DataMemWrEn,ZWrEn;
wire done,ready;

localparam [INS_WIDTH-1:0]
    NOP       =   8'd0,
    ENDOP       = 8'd1,
    CLAC        = 8'd2,
    LDIAC       = 8'd3,
    LDAC        = 8'd4,
    STR         = 8'd5,
    STIR        = 8'd6,
    JUMP        = 8'd7,
    JMPNZ       = 8'd8,
    JMPZ        = 8'd9,
    MUL         =   8'd10,
    ADD         =   8'd11,
    SUB         =   8'd12,
    INCAC     =   8'd13,
    MV_RL_AC    = {4'd1,4'd15},
    MV_RP_AC    = {4'd2,4'd15},
```

```
    MV_RQ_AC    = {4'd3,4'd15},
    MV_RC_AC    = {4'd4,4'd15},
    MV_R_AC     = {4'd5,4'd15},
    MV_R1_AC    = {4'd6,4'd15},
    MV_AC_RP    = {4'd7,4'd15},
    MV_AC_RQ    = {4'd8,4'd15},
    MV_AC_RL    = {4'd9,4'd15};

localparam [31:0]// time duration for each instruction to exicute
        NOP_time_duration       =   4,
        ENDOP_time_duration     =   4,
        CLAC_time_duration      =   4,
        LDIAC_time_duration     =   8,
        LDAC_time_duration      =   6,
        STR_time_duration       =   6,
        STIR_time_duration      =   8,
        JUMP_time_duration      =   6,
        JMPNZ_Y_time_duration   =   6,
        JMPNZ_N_time_duration   =   4,
        JMPZ_Y_time_duration    =   6,
        JMPZ_N_time_duration    =   4,
        MUL_time_duration     =   4,
        ADD_time_duration     =   4,
        SUB_time_duration     =   4,
        INCAC_time_duration     =   4,
        MV_RL_AC_time_duration  =   4,
        MV_RP_AC_time_duration  =   4,
        MV_RQ_AC_time_duration  =   4,
        MV_RC_AC_time_duration  =   4,
        MV_R_AC_time_duration   =   4,
        MV_R1_AC_time_duration  =   4,
        MV_AC_RP_time_duration  =   4,
        MV_AC_RQ_time_duration  =   4,
        MV_AC_RL_time_duration  =   4;


controlUnit #(.INS_WIDTH(INS_WIDTH)) dut(.clk(clk), .rstN(rstN), .startN(startN),
            .Zout(Zout), .ins(ins), .aluOp(aluOp), .incReg(incReg), .wrEnReg(wrEnReg),
            .busSel(busSel), .DataMemWrEn(DataMemWrEn), .ZWrEn(ZWrEn),
            .done(done), .ready(ready));

task automatic test_instruction(
    input [31:0] duration,
    input Z_value,
    input [INS_WIDTH-1:0] instruction
); begin

    @(posedge clk);
    ins <= instruction;
    Zout <= Z_value;

    #(duration * CLK_PERIOD);
end

endtask


initial begin
    @(posedge clk);
    rstN <= 1'b0;
    @(posedge clk);
    rstN <= 1'b1;
    startN <= 1'b0;
    @(posedge clk);
    startN <= 1'b1;

    ////// test NOP
    test_instruction(NOP_time_duration, 1'bX, NOP);

    ///// test CLAC
    test_instruction(CLAC_time_duration, 1'bX, CLAC);

    ///// test LDIAC
    test_instruction(LDIAC_time_duration, 1'bX, LDIAC);

    ////// test LDAC
    test_instruction(LDAC_time_duration, 1'bX, LDAC);

    ////// test STR
```

```
122     test_instruction(STR_time_duration, 1'bX, STR);
123 
124     ////// test STIR
125     test_instruction(STIR_time_duration, 1'bX, STIR);
126 
127     ////// test JUMP
128     test_instruction(JUMP_time_duration, 1'bX, JUMP);
129 
130     ////// test JMPNZ_Y
131     test_instruction(JMPNZ_Y_time_duration, 1'b0, JMPNZ);
132 
133     ////// test JMPNZ_N
134     test_instruction(JMPNZ_N_time_duration, 1'b1, JMPNZ);
135 
136     ////// test JMPZ_Y
137     test_instruction(JMPZ_Y_time_duration, 1'b1, JMPZ);
138 
139     ////// test JMPZ_N
140     test_instruction(JMPZ_N_time_duration, 1'b0, JMPZ);
141 
142     ////// test MUL
143     test_instruction(MUL_time_duration, 1'b0, MUL);
144 
145     ////// test ADD
146     test_instruction(ADD_time_duration, 1'b0, ADD);
147 
148     ////// test SUB
149     test_instruction(SUB_time_duration, 1'b0, SUB);
150 
151     ////// test INCAC
152     test_instruction(INCAC_time_duration, 1'b0, INCAC);
153 
154     ////// test MV_RL_AC
155     test_instruction(MV_RL_AC_time_duration, 1'b0, MV_RL_AC);
156 
157     ////// test MV_RP_AC
158     test_instruction(MV_RP_AC_time_duration, 1'b0, MV_RP_AC);
159 
160     //// test MV_RQ_AC
161     test_instruction(MV_RQ_AC_time_duration, 1'b0, MV_RQ_AC);
162 
163     ////// test MV_RC_AC
164     test_instruction(MV_RC_AC_time_duration, 1'b0, MV_RC_AC);
165 
166     ////// test MV_R_AC
167     test_instruction(MV_R_AC_time_duration, 1'b0, MV_R_AC);
168 
169     ////// test MV_R1_AC
170     test_instruction(MV_R1_AC_time_duration, 1'b0, MV_R1_AC);
171 
172     ////// test MV_AC_RP
173     test_instruction(MV_AC_RP_time_duration, 1'b0, MV_AC_RP);
174 
175     ////// test MV_AC_RQ
176     test_instruction(MV_AC_RQ_time_duration, 1'b0, MV_AC_RQ);
177 
178     ////// test MV_AC_RL
179     test_instruction(MV_AC_RL_time_duration, 1'b0, MV_AC_RL);
180 
181     ///// test ENDOP
182     test_instruction(ENDOP_time_duration, 1'bX, ENDOP);
183 
184 end
185 
186 initial begin          // simulation stop condition
187     forever begin
188         @(posedge clk);
189         if (done) begin
190             #(5*CLK_PERIOD);
191             $stop;
192         end
193     end
194 
195 end
196 
197 endmodule : controlUnit_tb
```

9.7.6 Arithmetic and Logic Unit (ALU)

Module - alu.v

```
module ALU
#(parameter WIDTH = 12)
(
    input signed [WIDTH-1:0]a,b,  // a - from Accumulator, b - from data-bus
    input [2:0]selectOp,
    output signed [WIDTH-1:0]dataOut
);

localparam [2:0]
            clr  =  3'd0,  // ALU operations and their control signals
            pass =  3'd1,
            add  =  3'd2,
            sub  =  3'd3,
            mul  =  3'd4,
            inc  =  3'd5,
            idle =  3'd6;

//output of the ALU is decided as below
assign dataOut =    (selectOp == clr)? {WIDTH{1'b0}}:
                    (selectOp == pass)? b:
                    (selectOp == add)? a+b:
                    (selectOp == sub)? a-b:
                    (selectOp == mul)? a*b:
                    (selectOp == inc)? a+1'b1:
                    (selectOp == idle)? {WIDTH{1'b0}}:
                    {WIDTH{1'b0}};  // default value is ZERO

endmodule //ALU
```

Testbench - alu_tb.v

```
`timescale 1ns/1ps
module ALU_tb();

localparam CLK_PERIOD = 20;

reg clk;
initial begin
    clk = 0;
    forever #(CLK_PERIOD/2) clk <= ~clk;
end

localparam [2:0]
            clr  =  3'd0,  // ALU operations and their control signals
            pass =  3'd1,
            add  =  3'd2,
            sub  =  3'd3,
            mul  =  3'd4,
            inc  =  3'd5,
            idle =  3'd6;

localparam WIDTH = 12;

reg signed [WIDTH-1:0]a,b;
reg [2:0] selectOp;
wire signed [WIDTH-1:0] dataOut;


ALU #(.WIDTH(WIDTH))dut(.a(a), .b(b), .selectOp(selectOp), .dataOut(dataOut));

initial begin
    @(posedge clk);
    a <= 10;
    b <= 3;
    selectOp <= clr;

    @(posedge clk);
    selectOp <= pass;

    @(posedge clk);
    selectOp <= add;
```

```
41
42      @(posedge clk);
43      selectOp <= sub;
44
45      @(posedge clk);
46      selectOp <= mul;
47
48      @(posedge clk);
49      selectOp <= inc;
50
51      @(posedge clk);
52      selectOp <= idle;
53
54      @(posedge clk);
55      a <= 20;
56      b <= -30;
57      selectOp <= clr;
58
59      @(posedge clk);
60      selectOp <= pass;
61
62      @(posedge clk);
63      selectOp <= add;
64
65      @(posedge clk);
66      selectOp <= sub;
67
68      @(posedge clk);
69      selectOp <= mul;
70
71      @(posedge clk);
72      selectOp <= inc;
73
74      @(posedge clk);
75      selectOp <= idle;
76
77      @(posedge clk);
78      repeat(10) begin
79          @(posedge clk);
80          a = $random();
81          b = $random();
82          selectOp = $random();
83      end
84      $stop;
85  end
86  endmodule //ALU_tb
```

9.7.7 Incrementable Registers

Module - incRegister.v

```
1  module incRegister
2  #(parameter WIDTH = 12 )  //default size of a register is 12
3  (
4      input [WIDTH-1:0]dataIn,
5      input wrEn,rstN,clk,incEn,
6      output [WIDTH-1:0]dataOut
7  );
8
9  reg [WIDTH-1:0]value;
10
11 always @(posedge clk) begin
12     if (~rstN)
13         value <= 0;
14     else if (wrEn)
15         value <= dataIn;
16     else if (incEn)
17         value <= value + 1'b1;
18 end
19
20 assign dataOut = value;
21
22 endmodule //incRegister
```

Testbench - incRegister_tb.v

```
`timescale 1ns/1ps

module incRegister_tb();

localparam CLK_PERIOD = 20;
reg clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WIDTH = 12;

reg [WIDTH-1:0]dataIn;
reg wrEn, incEn, rstN;
wire [WIDTH-1:0]dataOut;

incRegister #(.WIDTH(WIDTH)) dut(.dataIn(dataIn), .wrEn(wrEn), .rstN(rstN),
                            .clk(clk), .incEn(incEn), .dataOut(dataOut));

initial begin
    @(posedge clk);
    #(CLK_PERIOD*4/5);
    rstN <= 0;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 23;
    wrEn <= 1;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 36;
    wrEn <= 1;
    rstN <= 1;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 15;
    wrEn <= 0;
    incEn <= 1;

    repeat (10) begin
        @(posedge clk);
        #(CLK_PERIOD*4/5);
        wrEn = $random();
        incEn = $random();
        rstN = $random();
        dataIn = $random();
    end

    $stop;
end

initial begin
    forever begin
       @(posedge clk);
       #(CLK_PERIOD*1/5);
        $display("dataIn = %d   rstN = %b    wrEn = %b   incEn = %b    dataOut = %d",
                dataIn, rstN, wrEn, incEn, dataOut);
    end

end

endmodule:incRegister_tb
```

9.7.8 Register

Module - register.v

```
module register
#(parameter WIDTH = 12)  //default size of a register is 12
(
    input [WIDTH-1:0]dataIn,
    input wrEn,rstN,clk,
    output [WIDTH-1:0]dataOut
);

reg [WIDTH-1:0]value;

always @(posedge clk) begin
    if (~rstN)
        value <= 0;
    else if (wrEn)
        value <= dataIn;
end

assign dataOut = value;

endmodule //register
```

Testbench - register_tb.v

```
`timescale 1ns/1ps

module register_tb();

localparam CLK_PERIOD = 20;

reg clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WIDTH = 12;
reg [WIDTH-1:0]dataIn;
reg rstN, wrEn;
wire [WIDTH-1:0] dataOut;

register #(.WIDTH(WIDTH)) dut(.clk(clk), .wrEn(wrEn), .rstN(rstN), .dataIn(dataIn), .dataOut(dataOut));

initial begin
    @(posedge clk);
    rstN <= 0;

    @(posedge clk);
    rstN <= 1;
    dataIn <= 20;
    wrEn <= 0;

    @(posedge clk);
    dataIn <= 43;
    wrEn <= 1;

    repeat(10) begin
        @(posedge clk);
        dataIn = $random();
        wrEn = $random();
        rstN = $random();
    end

    @(posedge clk);
    $stop;
end

endmodule // register_tb
```

9.7.9 Multiplexer (System Bus)

Module - multiplexer.v

```
module multiplexer
#(
    parameter REG_WIDTH = 12,
    parameter INS_WIDTH = 8
)
(
    input [3:0]selectIn,
    input [REG_WIDTH-1:0]DMem, R, RL, RC, RP, RQ, R1, AC,
    input [INS_WIDTH-1:0]IR,
    output [REG_WIDTH-1:0]busOut
);

//control signals to select the bus input
localparam  [3:0]
    DMem_sel = 4'b0,
    R_sel    = 4'd1,
    IR_sel   = 4'd2,
    RL_sel   = 4'd3,
    RC_sel   = 4'd4,
    RP_sel   = 4'd5,
    RQ_sel   = 4'd6,
    R1_sel   = 4'd7,
    AC_sel   = 4'd8,
    idle     = 4'd9;


//input of the data-bus is selected as below
assign busOut = (selectIn == DMem_sel)? DMem:
                (selectIn == R_sel) ? R:
                (selectIn == IR_sel)? {{(REG_WIDTH-INS_WIDTH){1'b0}},IR}: //first 3 bits are zero
                (selectIn == RL_sel)? RL:
                (selectIn == RC_sel)? RC:
                (selectIn == RP_sel)? RP:
                (selectIn == RQ_sel)? RQ:
                (selectIn == R1_sel)? R1:
                (selectIn == AC_sel)? AC:
                {REG_WIDTH{1'b0}};

endmodule //multiplexer
```

Testbench - multiplexer_tb.v

```
`timescale 1ns/1ps
module multiplexer_tb();

localparam CLK_PERIOD = 10;

reg clk;
initial begin
    clk = 0;
    forever #(CLK_PERIOD/2) clk <= ~clk;
end

//control signals to select the bus input
localparam  [3:0]
    DMem_sel = 4'b0,
    R_sel    = 4'd1,
    IR_sel   = 4'd2,
    RL_sel   = 4'd3,
    RC_sel   = 4'd4,
    RP_sel   = 4'd5,
    RQ_sel   = 4'd6,
    R1_sel   = 4'd7,
    AC_sel   = 4'd8,
    idle     = 4'd9;

localparam REG_WIDTH = 12;
localparam INS_WIDTH = 8;

reg [3:0] selectIn;
wire [REG_WIDTH-1:0]DMem, R, RL, RC, RP, RQ, R1, AC;
```

```
wire [INS_WIDTH-1:0]IR;
wire [REG_WIDTH-1:0]busOut;

assign DMem = 20;
assign R = 21;
assign RL = 22;
assign RC = 23;
assign RP = 24;
assign RQ = 25;
assign R1 = 26;
assign AC = 27;
assign IR = 28;

multiplexer #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH)) mux(.selectIn(selectIn), .DMem(DMem),
                .R(R), .RL(RL), .RC(RC), .RP(RP), .RQ(RQ), .R1(R1), .AC(AC),
                .IR(IR), .busOut(busOut));
initial begin
    @(posedge clk);
    selectIn <= R_sel;
    $display("selectIn = %d busOut = %d", selectIn, busOut);

    @(posedge clk);
    selectIn <= RL_sel;
    $display("selectIn = %d busOut = %d", selectIn, busOut);

     @(posedge clk);
    selectIn <= RP_sel;
    $display("selectIn = %d busOut = %d", selectIn, busOut);

    @(posedge clk);
    selectIn <= RQ_sel;
    $display("selectIn = %d busOut = %d", selectIn, busOut);

    @(posedge clk);
    selectIn <= IR_sel;
    $display("selectIn = %d busOut = %d", selectIn, busOut);

    @(posedge clk);
    repeat(10) begin
        @(posedge clk);
        selectIn <= $urandom()%10;
        $display("selectIn = %d busOut = %d", selectIn, busOut);
    end
    $stop;
end

endmodule
```

9.7.10 Z register

Module - zReg.v

```
module zReg
#(parameter WIDTH = 12)
(
    input [WIDTH-1:0]dataIn,
    input clk,rstN,wrEn,
    output Zout
);

reg value;

always @(posedge clk) begin
    if (~rstN)
        value <= 1'b0;
    else if (wrEn)
        if (dataIn == 1'b0)
            value <= 1'b1;
        else
            value <= 1'b0;
end

assign  Zout = value;

endmodule //zReg
```

Testbench - zReg_tb.v

```
`timescale 1ns/1ps
module zReg_tb();

localparam CLK_PERIOD = 20;

reg clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam WIDTH = 12;
reg [WIDTH-1:0]dataIn;
reg rstN, wrEn;
wire Zout;

zReg #(.WIDTH(WIDTH)) dut(.dataIn(dataIn), .clk(clk), .rstN(rstN), .wrEn(wrEn), .Zout(Zout));

initial begin
    @(posedge clk);
    #(CLK_PERIOD*4/5);
    rstN <= 0;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    rstN <= 1;
    dataIn <= 0;
    wrEn <= 0;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 0;
    wrEn <= 1;

    @(posedge clk);
    #(CLK_PERIOD*4/5);
    dataIn <= 4;
    wrEn <= 1;

    repeat(10) begin
        @(posedge clk);
        #(CLK_PERIOD*4/5);
        dataIn = $random();
        wrEn = $urandom();
        rstN = $urandom();
    end

    $stop;
end

endmodule// zReg_tb
```

9.8 UART communication related Verilog modules

9.8.1 UART communication interface for the memory

Module - mem_communication_interface.v

```
module uart_mem_interface
#(
    parameter MEM_WORD_LENGTH = 12,
    parameter MEM_ADDR_LENGTH = 12,
    parameter UART_WIDTH = 8
)
(
     //////////////input output with main program
    input clk,rstN,txStart,
```

```
    output mem_transmitted, mem_received,

    //select start end mem addresses of tx and rx
    input [MEM_ADDR_LENGTH-1:0]tx_start_addr_in,tx_end_addr_in,rx_end_addr_in,
    input toggle_addr_range,   // 0 - go untill the last address of the memory,
                               //1 - consider inputted start, end addresses

    ////////////////inputs outputs with memory
    input [MEM_WORD_LENGTH-1:0]dataFromMem,
    output memWrEn,
    output [MEM_ADDR_LENGTH-1:0]mem_address,
    output [MEM_WORD_LENGTH-1:0]dataToMem,

    ////////////////////inputs outputs with uart system

    input txByteReady,rxByteReady,new_rx_byte_indicate,
    input [UART_WIDTH-1:0]ByteFromUart,
    output uartTxStart,
    output [UART_WIDTH-1:0]byteToUart
);

wire startTranmit,txReady,rxDone;
wire new_rx_data_indicate;

wire [MEM_ADDR_LENGTH-1:0]tx_start_addr, tx_end_addr, rx_end_addr;

localparam [2:0]
                idle = 3'b0,
                transmit_0 = 3'd1,
                transmit_1 = 3'd2,
                transmit_2 = 3'd3,
                transmit_3 = 3'd4,
                receive_0 = 3'd5,
                receive_1 = 3'd6,
                receive_2 = 3'd7;


reg [MEM_ADDR_LENGTH-1:0]currentAddress, nextAddress;
reg [2:0]currentState, nextState;
reg currentStartTransmit, nextStartTranmit;


always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        currentState <= idle;
        currentAddress <= {MEM_ADDR_LENGTH{1'b0}};
        currentStartTransmit <= 1'b1; //starts if value is zero
    end
    else begin
        currentState <= nextState;
        currentAddress <= nextAddress;
        currentStartTransmit <= nextStartTranmit;
    end
end

always @(*) begin
    nextState = currentState;
    nextAddress = currentAddress;
    nextStartTranmit = currentStartTransmit;

    case (currentState)
        idle: begin
            nextAddress = {MEM_ADDR_LENGTH{1'b0}};
            nextStartTranmit = 1'b1;    //to transmit this should be zero
            if (new_rx_data_indicate)
                nextState = receive_0;
            else if (~txStart) begin
                nextState = transmit_0;
                nextAddress = tx_start_addr;
            end
        end
        //transmission process starts here
        transmit_0: begin         // to give the address to memory
            nextState = transmit_1;
        end

        transmit_1: begin       // extra delay for ip core memory
            nextStartTranmit = 1'b0;
            nextState = transmit_2;
```

```
        end

        transmit_2: begin          // start uart transmitter
            nextStartTranmit = 1'b0;
            nextState = transmit_3;
        end

        transmit_3: begin         // find end of the uart transmission
            nextStartTranmit = 1'b1;
            if (txReady == 1'b1) begin
                if (currentAddress == tx_end_addr) begin
                    nextState = idle;
                end
                else begin
                    nextAddress = currentAddress + {{MEM_ADDR_LENGTH-1{1'b0}} , 1'b1};
                    nextState = transmit_0;
                end
            end
        end
        //receiving process starts here
        receive_0: begin            // to find the end of the uart receiving
            if (rxDone) begin
                nextState = receive_1;
            end
        end

        receive_1: begin // store in the memory (no need extra delay as it is explicitly given in receive_0)
            nextState = receive_2;
        end

        receive_2: begin            //to find the end of the receiving process
            if (currentAddress == rx_end_addr) begin
                nextState = idle;
            end
            else begin
                nextAddress = currentAddress + {{MEM_ADDR_LENGTH-1{1'b0}} , 1'b1};
                nextState = receive_0;
            end
        end
    endcase
end

assign startTranmit = currentStartTransmit;
assign memWrEn = (currentState == receive_1)? 1'b1: 1'b0;
assign mem_received = ((currentState == receive_2) && (currentAddress == rx_end_addr))? 1'b1: 1'b0;
assign mem_transmitted = ((currentState == transmit_3) && (txReady == 1'b1) && (currentAddress ==
    tx_end_addr))?
                         1'b1: 1'b0;

assign mem_address = currentAddress;

uart_encoder_decoder #(.DATA_WIDTH(MEM_WORD_LENGTH), .UART_WIDTH(UART_WIDTH)) ED(.dataFromMem(dataFromMem),
                    .clk(clk), .rstN(rstN), .txStart(startTranmit), .txReady(txReady), .rxDone(rxDone),
                    .dataToMem(dataToMem), .new_rx_data_indicate(new_rx_data_indicate),
                    .txByteReady(txByteReady), .txByteStart(uartTxStart), .byteForTx(byteToUart),
                    .byteFromRx(ByteFromUart), .rxByteReady(rxByteReady),
                    .new_rx_byte_indicate(new_rx_byte_indicate));

assign tx_start_addr = (toggle_addr_range == 1'b0)? {MEM_ADDR_LENGTH{1'b0}}: tx_start_addr_in;
assign tx_end_addr = (toggle_addr_range == 1'b0)? {MEM_ADDR_LENGTH{1'b1}}: tx_end_addr_in;

assign rx_end_addr = (toggle_addr_range == 1'b0)? {MEM_ADDR_LENGTH{1'b1}}:
                        (rx_end_addr_in == 0)? {{MEM_ADDR_LENGTH-4{1'b0}}, {4{1'b1}}}:
                        rx_end_addr_in;        //give 31 as the end rx address if input addr is small

endmodule //uart_mem_interface
```

9.8.2 Data word size handler between memory and UART system

Module - data_encoder_decoder.v

```
module uart_encoder_decoder
#(
    parameter DATA_WIDTH = 12, // width of memory word
    parameter UART_WIDTH = 8
```

```
)
(
    input [DATA_WIDTH-1:0]dataFromMem,
    input clk,rstN,txStart,
    output txReady,rxDone,
    output [DATA_WIDTH-1:0]dataToMem,
    output new_rx_data_indicate,

    ////////////////////// inputs outputs for the UART system
    input txByteReady,
    output txByteStart,
    output [UART_WIDTH-1:0]byteForTx,

    input [UART_WIDTH-1:0]byteFromRx,
    input rxByteReady,new_rx_byte_indicate
);

localparam EXTRA = ((DATA_WIDTH % UART_WIDTH) == 0)?0:1;
localparam COUNT = (DATA_WIDTH/UART_WIDTH) + EXTRA;
localparam BUFFER_WIDTH = COUNT * UART_WIDTH;
localparam COUNTER_LENGTH = (COUNT == 1)? 1:$clog2(COUNT);

localparam [2:0]
            idle = 3'd0,
            transmit_1 = 3'd1,
            transmit_2 = 3'd2,
            receive_0 = 3'd3,
            receive_1 = 3'd4,
            receive_2 = 3'd5,
            receive_3 = 3'd6;

reg [2:0]currentState, nextState;
reg [BUFFER_WIDTH-1:0]currentTxBuffer, nextTxBuffer;
reg [BUFFER_WIDTH-1:0]currentRxBuffer, nextRxBuffer;
reg [COUNTER_LENGTH-1:0]currentTxCount, nextTxCount;
reg [COUNTER_LENGTH-1:0]currentRxCount, nextRxCount;

always @(posedge clk or negedge rstN) begin
    if (!rstN) begin
        currentTxBuffer <= {BUFFER_WIDTH{1'b0}};
        currentRxBuffer <= {BUFFER_WIDTH{1'b0}};
        currentTxCount <= {COUNTER_LENGTH{1'b0}};
        currentRxCount <= {COUNTER_LENGTH{1'b0}};
        currentState <= idle;
    end
    else begin
        currentTxBuffer <= nextTxBuffer;
        currentRxBuffer <= nextRxBuffer;
        currentTxCount <= nextTxCount;
        currentRxCount <= nextRxCount;
        currentState <= nextState;
    end
end

always @(*) begin
    nextState = currentState;
    nextTxCount = currentTxCount;
    nextRxCount = currentRxCount;
    nextTxBuffer = currentTxBuffer;
    nextRxBuffer = currentRxBuffer;

    case (currentState)
        idle: begin
            nextTxCount = {COUNTER_LENGTH{1'b0}};
            nextRxCount = {COUNTER_LENGTH{1'b0}};
            if (new_rx_byte_indicate) begin  //receiver indicates a arrival of new byte
                nextState = receive_1;
                nextRxBuffer = {BUFFER_WIDTH{1'b0}};
            end
            else if (txStart == 1'b0) begin
                nextState = transmit_1;
                nextTxBuffer = dataFromMem;
            end
        end

        transmit_1: begin           //start byte transmission
            nextState = transmit_2;
        end

```

```
84
85          transmit_2: begin           // end of the byte transmission
86              if (txByteReady) begin
87                  if (BUFFER_WIDTH == UART_WIDTH) begin
88                      nextState = idle;
89                  end
90                  else begin
91                      nextTxCount = currentTxCount + 1'b1;
92                      if (currentTxCount == COUNT-1 )
93                          nextState = idle;
94                      else begin
95                          nextTxBuffer = currentTxBuffer >> UART_WIDTH;
96                          nextState = transmit_1;
97                      end
98                  end
99              end
100         end
101
102         receive_0: begin    // to give a EXTRA time to make rxByteReady to become 1'b0
103             nextState = receive_1;
104         end
105
106         receive_1 : begin
107             if(rxByteReady) begin
108                 nextRxCount = currentRxCount + 1'b1;
109                 if (BUFFER_WIDTH == 8) begin
110                     nextRxBuffer = byteFromRx;
111                     nextState = idle;
112                 end
113                 else begin
114                     nextRxBuffer = {byteFromRx,currentRxBuffer[BUFFER_WIDTH-1:8]};
115                     if (currentRxCount == (COUNT-1))
116                         nextState = idle;
117                     else
118                         nextState = receive_2;
119                 end
120             end
121         end
122
123         receive_2: begin
124             if(new_rx_byte_indicate)  //receiver indicates a arrival of new byte
125                 nextState = receive_3;
126         end
127
128         receive_3: begin     // to give a EXTRA time to make rxByteReady to become 1'b0
129             nextState = receive_1;
130         end
131
132     endcase
133 end
134
135
136 assign txByteStart = (currentState == transmit_1)? 1'b0: 1'b1;
137 assign txReady = (currentState == idle)? 1'b1 : 1'b0;
138 assign byteForTx = currentTxBuffer[UART_WIDTH-1:0];
139 assign rxDone = ((currentState == receive_1) && (rxByteReady == 1'b1) && (currentRxCount == COUNT-1 ))?
140                 1'b1 : 1'b0;
141 assign dataToMem = currentRxBuffer[DATA_WIDTH-1:0];
142 assign new_rx_data_indicate = ((currentState == idle) && (new_rx_byte_indicate))? 1'b1: 1'b0; //arrival of
    new data set
143
144 endmodule //uart_encoder_decoder
```

9.8.3 UART system

Module - uart_system.v

```
1 module uart_system
2 #(
3     parameter DATA_WIDTH = 8,
4     parameter BAUD_RATE = 19200
5 )
6 (
7     input clk, rstN,txByteStart,rx,
8     input [DATA_WIDTH-1:0]byteForTx,
```

```
    output tx,txReady,rxReady,new_byte_indicate,
    output [DATA_WIDTH-1:0]byteFromRx
);

wire baudTick;

uart_transmitter #(.DATA_WIDTH(DATA_WIDTH)) Tx(.dataIn(byteForTx), .clk(clk), .baudTick(baudTick),
                .rstN(rstN), .txStart(txByteStart), .tx(tx), .TxReady(txReady));

uart_receiver #(.DATA_WIDTH(DATA_WIDTH)) RX(.rx(rx), .clk(clk), .rstN(rstN), .baudTick(baudTick),
                .dataOut(byteFromRx), .ready(rxReady), .new_byte_indicate(new_byte_indicate));

uart_boudRateGen #(.BAUD_RATE(BAUD_RATE)) BRG(.clk(clk), .rstN(rstN), .baudTick(baudTick));


endmodule //uart_system
```

9.8.4 UART baud rate generator

Module - uart_baudRateGen.v

```
module uart_boudRateGen
#(
    parameter BAUD_RATE = 19200
)
(
    input clk,rstN,
    output baudTick
);

localparam CLK_RATE = 50*(10**6);
localparam RESOLUTION = 16;  // samples per 1 baud
localparam MAX_COUNT = (CLK_RATE/BAUD_RATE/RESOLUTION); //round to smaller integer
localparam WIDTH = $clog2(MAX_COUNT);

reg [WIDTH-1:0]count;

always @(posedge clk) begin
    if (~rstN)
        count <= 1'b0;
    else if (count < MAX_COUNT)
        count <= count + 1'b1;
    else
        count <= 1'b0;
end

assign  baudTick = (count==MAX_COUNT)? 1'b1:1'b0;


endmodule //uart_boudRateGen
```

9.8.5 UART transmitter

Module - uart_transmitter.v

```
module uart_transmitter
#(
    parameter DATA_WIDTH = 8
)
(
    input [DATA_WIDTH-1:0]dataIn,
    input clk, baudTick,rstN,txStart,
    output tx,TxReady
);

localparam [1:0]
                idle = 2'd0,
                start = 2'd1,
                data = 2'd2,
                stop = 2'd3;

```

```
reg [1:0]currentState, nextState;
reg [DATA_WIDTH-1:0]currentData, nextData;
reg currentBit, nextBit;
reg [2:0]currentCount, nextCount;
reg [3:0]currentTick, nextTick;


always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        currentTick <= 4'b0;
        currentCount <= 3'b0;
        currentState <= idle;
        currentData <= 0;
        currentBit <= 1'b1;
    end
    else begin
        currentTick <= nextTick;
        currentCount <= nextCount;
        currentState <= nextState;
        currentData <= nextData;
        currentBit <= nextBit;
    end
end

always @(*) begin
    nextTick = currentTick;
    nextCount = currentCount;
    nextState = currentState;
    nextData = currentData;
    nextBit = currentBit;

    case (currentState)
        idle: begin
            if (~txStart) begin
                nextState = start;
                nextTick = 4'b0;
                nextBit = 1'b0;
                nextData = dataIn;
            end
            else
                nextBit = 1'b1;
        end

        start: begin
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 4'd15) begin
                    nextState = data;
                    nextCount = 3'b0;
                    nextBit = currentData[0];
                end
            end
        end

        data: begin
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 15) begin
                    nextCount = currentCount + 3'b1;
                    nextBit = currentData[nextCount];
                    if (currentCount ==3'd7) begin
                        nextState = stop;
                        nextBit = 1'b1;
                    end
                end
            end
        end

        stop: begin
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 4'd15)
                    nextState = idle;
            end
        end

    endcase
end

```

```
assign tx = currentBit;
assign  TxReady = (currentState == idle)? 1'b1:1'b0;

endmodule //uart_transmitter
```

9.8.6 UART receiver

Module - uart_receiver.v

```
module uart_receiver
#(
    parameter DATA_WIDTH = 8
)
(
    input rx, clk, rstN, baudTick,
    output ready,
    output [DATA_WIDTH-1:0]dataOut,
    output new_byte_indicate
);

localparam [1:0]
            idle = 2'b0,
            start = 2'b1,
            data = 2'd2,
            stop = 2'd3;

reg [1:0]currentState, nextState;
reg [2:0]currentCount,nextCount;
reg [3:0]currentTick, nextTick;
reg [DATA_WIDTH-1:0]currentData, nextData;

always @(posedge clk or negedge rstN) begin
    if(~rstN) begin
        currentState <= idle;
        currentCount <= 3'b0;
        currentTick <= 4'b0;
        currentData <= 0;
    end
    else begin
        currentState <= nextState;
        currentCount <= nextCount;
        currentTick <= nextTick;
        currentData <= nextData;
    end
end

always @(*) begin
    nextState = currentState;
    nextCount = currentCount;
    nextTick = currentTick;
    nextData = currentData;

    case (currentState )
        idle: begin
            nextTick = 4'b0;
            nextCount = 3'b0;
            if (rx == 1'b0) begin
                nextState = start;
            end
        end

        start: begin
           if (baudTick) begin
               nextTick = currentTick + 4'b1;
               if (currentTick == 4'd7) begin
                   if (~rx) begin
                        nextState = data;
                        nextCount = 3'b0;
                        nextTick = 4'b0;
                        nextData = 8'b0;
                   end
                   else begin
                       nextState = idle;

                   end
```

```
                end
            end
        end

        data: begin
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 4'd15) begin
                    nextData[currentCount] = rx;
                    nextCount = currentCount + 3'b1;
                    if (currentCount == 3'd7) begin
                        nextState = stop;
                    end
                end
            end
        end

        stop: begin
            if (baudTick) begin
                nextTick = currentTick + 4'b1;
                if (currentTick == 4'd15) begin
                    nextState = idle;
                end
            end
        end

    endcase
end

assign dataOut = currentData;
assign ready = (currentState == idle)? 1'b1: 1'b0;
assign new_byte_indicate = ((currentState == start) && (baudTick) && (currentTick == 4'd7) && (~rx))?
                           1'b1:1'b0; //start of new data byte

endmodule //uart_receiver
```

9.9 Count and display the number of clock cycles for matrix multiplication process

9.9.1 Clock cycle counter - timeCounter.v

```
module timeCounter (
    input clk, rstN, startN, stop,
    output [25:0]timeDuration
);
localparam  idle = 2'b0,
            counting = 2'b1,
            countEnd = 2'd2;

reg [25:0]currentTime, nextTime;
reg [1:0]currentState, nextState;

always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        currentTime <= 26'b0;
        currentState <= idle;
    end
    else begin
        currentTime <= nextTime;
        currentState <= nextState;
    end
end

always @( *) begin
    nextTime = currentTime;
    nextState = currentState;

    case (currentState)
        idle: begin
            nextTime = 26'b0;
            if (~startN) begin
                nextState = counting;
```

```
            end
        end

        counting: begin
            if (stop)begin
                nextState = countEnd;
            end
            else begin
                nextTime = currentTime + 26'b1;
            end
        end

        countEnd: begin
            //wait until reset (keep the same count)
        end

    endcase
end

assign  timeDuration = currentTime;

endmodule //timeCounter
```

9.9.2 Display on 8 seven segments on the FPGA board

Module - hex_display.v

```
module hex_display (
    input clk, rstN,
    input [2:0]state,
    input start_timeValue_convetion,
    input [25:0]timeValue,
    output [6:0]out0, out1,out2,out3,out4,out5,out6,out7
);

localparam  //lettes needed to be shown on seven segments
            a = 5'd10,
            b = 5'd11,
            c = 5'd12,
            d = 5'd13,
            e = 5'd14,
            f = 5'd15,
            i = 5'd18,
            n = 5'd19,
            o = 5'd20,
            p = 5'd21,
            r = 5'd22,
            s = 5'd23,
            t = 5'd24,
            u = 5'd25,
            y = 5'd26,
            off = 5'd27;

localparam  uart_ready = 3'd0,
            uart_receive_Imem = 3'd1,
            uart_receive_dmem = 3'd2,
            process_ready = 3'd3,
            process_exicute = 3'd4,
            uart_transmit_dmem = 3'd5,
            finish = 3'd6;

reg [2:0]currentState, nextState;

wire done1;
wire [4:0]x0,x1,x2,x3,x4,x5,x6,x7;
wire [3:0]y0,y1,y2,y3,y4,y5,y6,y7;

assign {x7,x6,x5,x4,x3,x2,x1,x0} =
        (state == uart_ready)?          {off,off,off,r,e,a,d,y}:
        (state == uart_receive_Imem)?   {u,a,r,t,off,i,n,s}:
        (state == uart_receive_dmem)?   {u,a,r,t,d,a,t,a}:
        (state == process_exicute)?     {off,p,r,o,c,e,s,s}:
        (state == uart_transmit_dmem)?  {u,a,r,t,off,a,n,s}:
        (state == finish)?              {1'b0,y7,1'b0,y6,1'b0,y5,1'b0,y4,1'b0,y3,1'b0,y2,1'b0,y1,1'b0,y0}:
        {off,off,off,off,off,off,off,off};
```

```
49
50 BCDtoHEX BtH(.in0(x0), .in1(x1), .in2(x2),.in3(x3),.in4(x4),.in5(x5),.in6(x6),.in7(x7),
51             .out0(out0), .out1(out1), .out2(out2),.out3(out3),.out4(out4),.out5(out5),
52             .out6(out6),.out7(out7));
53
54 binaryToBCD timeConverter(.binaryValue(timeValue),.clk(clk),.rstN(rstN),
55                           .start(start_timeValue_convetion), .done(),.ready(),
56                           .digit7(y7),.digit6(y6),.digit5(y5),.digit4(y4),
57                           .digit3(y3),.digit2(y2),.digit1(y1),.digit0(y0));
58
59 endmodule //hex_display
```

Module - BCDtoHEX.v

```
1  module BCDtoHEX(
2      input [4:0]in0,in1,in2,in3,in4,in5,in6,in7,
3      output [6:0]out0,out1,out2,out3,out4,out5,out6,out7
4  );
5
6  SSeg  s0(.in(in0),.out(out0));
7  SSeg  s1(.in(in1),.out(out1));
8  SSeg  s2(.in(in2),.out(out2));
9  SSeg  s3(.in(in3),.out(out3));
10 SSeg  s4(.in(in4),.out(out4));
11 SSeg  s5(.in(in5),.out(out5));
12 SSeg  s6(.in(in6),.out(out6));
13 SSeg  s7(.in(in7),.out(out7));
14
15
16 endmodule // BCDtoHEX
```

Module - SSeg.v

```
1  module SSeg(
2      input [4:0]in,
3      output reg [6:0]out
4  );
5
6  localparam
7              a = 5'd10,
8              b = 5'd11,
9              c = 5'd12,
10             d = 5'd13,
11             e = 5'd14,
12             f = 5'd15,
13             i = 5'd18,
14             n = 5'd19,
15             o = 5'd20,
16             p = 5'd21,
17             r = 5'd22,
18             s = 5'd23,
19             t = 5'd24,
20             u = 5'd25,
21             y = 5'd26,
22             off = 5'd27;
23
24
25 always @(*) begin
26     case (in)
27         5'd0:out = 7'b1000000;
28         5'd1:out = 7'b1111001;
29         5'd2:out = 7'b0100100;
30         5'd3:out = 7'b0110000;
31         5'd4:out = 7'b0011001;
32         5'd5:out = 7'b0010010;
33         5'd6:out = 7'b0000010;
34         5'd7:out = 7'b1111000;
35         5'd8:out = 7'b0000000;
36         5'd9:out = 7'b0011000;
37         5'ha:out = 7'b0001000;
38         5'hb:out = 7'b0000011;
39         5'hc:out = 7'b1000110;
40         5'hd:out = 7'b0100001;
41         5'he:out = 7'b0000110;
42         5'hf:out = 7'b0001110;
```

```
        i   :out = 7'b1110000;
        n   :out = 7'b0001011;
        o   :out = 7'b1000000;
        p   :out = 7'b0001100;
        r   :out = 7'b1001110;
        s   :out = 7'b0010010;
        t   :out = 7'b0000111;
        u   :out = 7'b1000001;
        y   :out = 7'b0010001;
        off :out = 7'b1111111;

        default: out = 7'b1111111;
    endcase
end


endmodule // SSeg
```

Module - binaryToBCD.v

```
module binaryToBCD(
    input [25:0]binaryValue,// to get 8 bit value 26.57 (26) bit value needed
    input clk,rstN,start,
    output reg ready, done,
    output reg [3:0]digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0
);

localparam [1:0]
    idle = 2'b00,
    subtract = 2'b01,
    shift = 2'b10,
    over = 2'b11;

reg [4:0]currentCount,nextCount;            //count upto 26
reg [31:0]currentValue,nextValue;
reg [1:0]currentState,nextState;
reg [25:0]currentInput,nextInput;

always @(posedge clk,negedge rstN) begin
    if (~rstN) begin
        currentCount <= 5'b0;
        currentValue <= 32'b0;
        currentState <= idle;
        currentInput <= 26'b0;
    end else begin
        currentValue <= nextValue;
        currentCount <= nextCount;
        currentState <= nextState;
        currentInput <= nextInput;
    end
end

always @(*) begin
    nextCount = currentCount;
    nextValue = currentValue;
    nextState = currentState;
    nextInput = currentInput;
    {digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0} = 32'b0;
    ready = 1'b0;
    done = 1'b1;     // actives the BCDtoHEX by the negedge

    case (currentState)
        idle: begin
            {digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0} = currentValue;
            ready = 1'b1;
            if (!start) begin        // KEY is push button stay at 1 normally
                nextState = subtract;
                ready = 1'b0;
                nextCount = 5'b0;
                nextInput = binaryValue;
        nextValue = 32'b0;
            end
        end

        subtract: begin
            if (currentValue[3:0]>4)
                nextValue[3:0] = currentValue[3:0]+4'b0011;
            if (currentValue[7:4]>4)
```

```
                nextValue[7:4] = currentValue[7:4]+4'b0011;
            if (currentValue[11:8]>4)
                nextValue[11:8] = currentValue[11:8]+4'b0011;
            if (currentValue[15:12]>4)
                nextValue[15:12] = currentValue[15:12]+4'b0011;
            if (currentValue[19:16]>4)
                nextValue[19:16] = currentValue[19:16]+4'b0011;
            if (currentValue[23:20]>4)
                nextValue[23:20] = currentValue[23:20]+4'b0011;
            if (currentValue[27:24]>4)
                nextValue[27:24] = currentValue[27:24]+4'b0011;
            if (currentValue[31:28]>4)
                nextValue[31:28] = currentValue[31:28]+4'b0011;

            nextState = shift;
        end

        shift: begin
            nextInput[25:1] = currentInput[24:0];
            nextValue = {currentValue[30:0],currentInput[25]};
            nextCount = currentCount+5'b1;

            if (currentCount<5'd25)
                nextState = subtract;
            else
                nextState = over;
        end

        over: begin
            nextState = idle;
            done = 1'b0;
        {digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0} = currentValue;
        end
    endcase
end

endmodule // binaryToBCD
```

9.10 Testbenches for top level module

In order to do the simulation for the full design, the top level module (Section - 9.1.2) is changed appropriately. Also instead of using a single module RAM (Section - 9.1.3) for both data-memory and instruction-memory we used 2 separate modules in order to memory initialization using external text file and write the memory content to a text file after the process is finished in order to validate by comparison the answer matrix with Python3.8 based calculation.

Module - simulation_top_tb.sv

```
`timescale 1ns/1ps

module simulation_top_tb();

localparam CLK_PERIOD = 20;

reg clk;
initial begin
    clk <= 0;
    forever begin
        #(CLK_PERIOD/2);
        clk <= ~clk;
    end
end

localparam CORE_COUNT = 2;

reg rstN, startN;
wire processor_ready, processDone;

simulation_top #(.CORE_COUNT(CORE_COUNT)) simulation_top(.clk(clk), .rstN(rstN), .startN(startN),
                .processor_ready(processor_ready), .processDone(processDone));
```

```
23
24 initial begin
25     @(posedge clk);
26     rstN = 1'b0;
27     startN = 1'b1;
28
29     @(posedge clk);
30     rstN = 1'b1;
31     startN = 1'b0;
32
33     @(posedge clk);
34     startN = 1'b1;
35
36     wait(processDone);
37
38     repeat(10) @(posedge clk);
39     $stop;
40 end
41
42 endmodule // simulation_top_tb
```

Module - simulation_top.v

```
1  module simulation_top
2  #(
3      parameter CORE_COUNT = 1
4  )
5  (
6      input wire clk, rstN, startN,
7      output wire processor_ready, processDone
8  );
9
10 // localparam CORE_COUNT = 2;
11 localparam REG_WIDTH = 12;
12 localparam DATA_MEM_WIDTH = CORE_COUNT * REG_WIDTH;
13 localparam INS_WIDTH = 8;
14 localparam INS_MEM_DEPTH = 256;
15 localparam DATA_MEM_DEPTH = 4096;
16 localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
17 localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);
18
19
20 ////// logic related to data memory ///////////
21 wire [DATA_MEM_WIDTH-1:0]DataMemOut, DataMemIn, processor_DataOut, uart_DataOut;
22 wire [DATA_MEM_ADDR_WIDTH-1:0] processor_dataMemAddr;
23 wire [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr, uart_dataMemAddr;
24
25 ////// logic related to instruction memory ////////////
26 wire [INS_WIDTH-1:0]InsMemOut, InsMemIn;
27 wire [INS_MEM_ADDR_WIDTH-1:0] processor_InsMemAddr;
28 wire [INS_MEM_ADDR_WIDTH-1:0]insMemAddr,uart_InsMemAddr;
29
30 ////// other logics ////////////
31 wire dataMemWrEn, processor_DataMemWrEn, uart_dataMemWrEn;
32 wire processStartN;
33
34
35
36 ///////////////// state change logic
37
38 localparam [2:0]   //states
39     idle = 3'd0,
40     process_exicute = 3'd4,
41     finish = 3'd6;
42
43
44 reg [2:0] currentState, nextState;
45
46 always @(posedge clk) begin
47     if (~rstN) begin
48         currentState <= idle;
49     end
50     else begin
51         currentState <= nextState;
52     end
53 end
54
55 always @(*) begin
```

```
56
57      nextState = currentState;
58
59      case (currentState)
60          idle: begin     // start state
61              if (~startN) begin
62                  nextState = process_exicute;
63              end
64          end
65
66          process_exicute: begin  // processor exicute program (matrix multiplication)
67              if (processDone) begin
68                  nextState = finish;
69              end
70          end
71
72
73          finish: begin  //End of the process
74
75          end
76
77          default : nextState = idle;
78
79      endcase
80  end
81
82  assign processStartN = ((currentState == idle) && (~startN))? 1'b0: 1'b1;
83
84  assign dataMemWrEn = (currentState == process_exicute)? processor_DataMemWrEn : 1'b0;
85
86  assign dataMemAddr = (currentState == process_exicute)? processor_dataMemAddr: {DATA_MEM_ADDR_WIDTH{1'b0}};
87
88  assign DataMemIn = (currentState == process_exicute)? processor_DataOut: {DATA_MEM_WIDTH{1'b0}};
89
90  assign insMemAddr = (currentState == process_exicute)? processor_InsMemAddr: {INS_MEM_ADDR_WIDTH{1'b0}};
91
92  ////////////////////////////////////////////////////////////////////////
93  multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
94                      .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
95                      multi_core_processor
96                      (.clk(clk), .rstN(rstN), .startN(processStartN), .ProcessorDataIn(DataMemOut),
97                      .InsMemOut(InsMemOut), .ProcessorDataOut(processor_DataOut), .insMemAddr(
    processor_InsMemAddr),
98                      .dataMemAddr(processor_dataMemAddr), .DataMemWrEn(processor_DataMemWrEn), .done(
    processDone),
99                      .ready(processor_ready));
100
101
102  INS_RAM #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH), .mem_init(1))
103         IM(.clk(clk), .wrEn(1'b0), .dataIn(InsMemIn), .addr(insMemAddr), .dataOut(InsMemOut));
104
105  DATA_RAM #(.WIDTH(DATA_MEM_WIDTH), .DEPTH(DATA_MEM_DEPTH), .mem_init(1)) DM
106         (.clk(clk), .wrEn(dataMemWrEn), .dataIn(DataMemIn), .addr(dataMemAddr), .dataOut(DataMemOut),
107         .processDone(processDone));
108
109
110 endmodule //simulation_top
```

Module - INS_RAM.v

```
1  module INS_RAM
2  #(
3      parameter mem_init = 0,
4      parameter WIDTH = 8,
5      parameter DEPTH = 256,
6      parameter ADDR_WIDTH = $clog2(DEPTH)
7  )
8  (
9      input wire clk, wrEn,
10     input wire [WIDTH-1:0]dataIn,
11     input wire [ADDR_WIDTH-1:0]addr,
12     output wire [WIDTH-1:0]dataOut
13 );
14
15
16 reg [ADDR_WIDTH-1:0]addr_reg;
17
18 reg [WIDTH-1:0]memory[0:DEPTH-1] ;
```

```
19
20 /// initialize memory for simulation  /////////
21 initial begin
22     if (mem_init == 1) begin
23         $readmemb("../../9_ins_mem_tb.txt", memory);
24     end
25 end
26 ////////////////////////////////////////////////
27
28 always @(posedge clk) begin
29     addr_reg <= addr;
30     if (wrEn) begin
31         memory[addr] <= dataIn;   // write requires only 1 clk cyle.
32     end
33 end
34 assign dataOut = memory[addr_reg];   // address is registered. Need 2 clk cycles to read.
35
36 endmodule // INS_RAM
```

Module - DATA_RAM.v

```
1  module DATA_RAM
2  #(
3      parameter mem_init = 0,
4      parameter WIDTH = 12,
5      parameter DEPTH = 4096,
6      parameter ADDR_WIDTH = $clog2(DEPTH)
7  )
8  (
9      input wire clk, wrEn,
10     input wire [WIDTH-1:0]dataIn,
11     input wire [ADDR_WIDTH-1:0]addr,
12     output wire [WIDTH-1:0]dataOut,
13
14     input wire processDone   // need only for simulation (to get the memory at the end)
15 );
16
17
18 reg [ADDR_WIDTH-1:0]addr_reg;
19
20 reg [WIDTH-1:0]memory[0:DEPTH-1] ;
21
22 /// initialize memory for simulation  /////////
23 initial begin
24     if (mem_init == 1) begin
25         $readmemb("../../4_data_mem_tb.txt", memory);
26     end
27 end
28
29 always @(processDone) begin
30     if (processDone) begin
31         $writememb("../../11_data_mem_out.txt", memory);
32     end
33 end
34
35 ////////////////////////////////////////////////
36
37 always @(posedge clk) begin
38     addr_reg <= addr;
39     if (wrEn) begin
40         memory[addr] <= dataIn;   // write requires only 1 clk cyle.
41     end
42 end
43 assign dataOut = memory[addr_reg];   // address is registered. Need 2 clk cycles to read.
44
45 endmodule // DATA_RAM
```