

# 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

<b>Acronyms</b>	<b>vi</b>
<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
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
<b>2 Algorithm</b>	<b>5</b>
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
<b>3 Instruction Set Architecture</b>	<b>9</b>
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
<b>4 Modules</b>	<b>16</b>
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
<b>5</b>	<b>Simulation, Testing &amp; Debugging</b>	<b>26</b>
<b>6</b>	<b>Results Analyzing and Verification</b>	<b>29</b>
6.1	Comparison of Speeds of Multicore Processor Designs	36
<b>7</b>	<b>Design Summary</b>	<b>40</b>
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
<b>8</b>	<b>References</b>	<b>43</b>
<b>9</b>	<b>Appendix</b>	<b>44</b>
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 Multi-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.

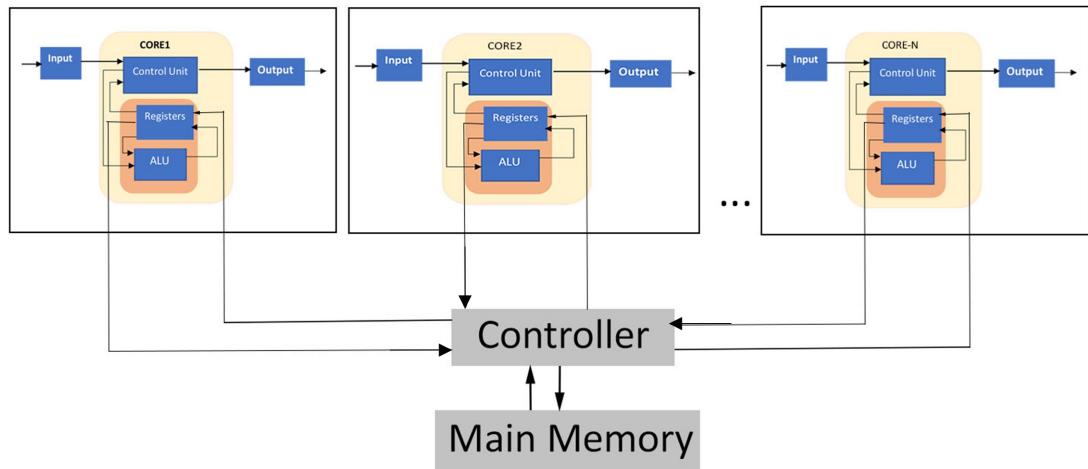


Figure 1.1: Block diagram of a SIMD Multi-core Processor

### 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 constraints 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

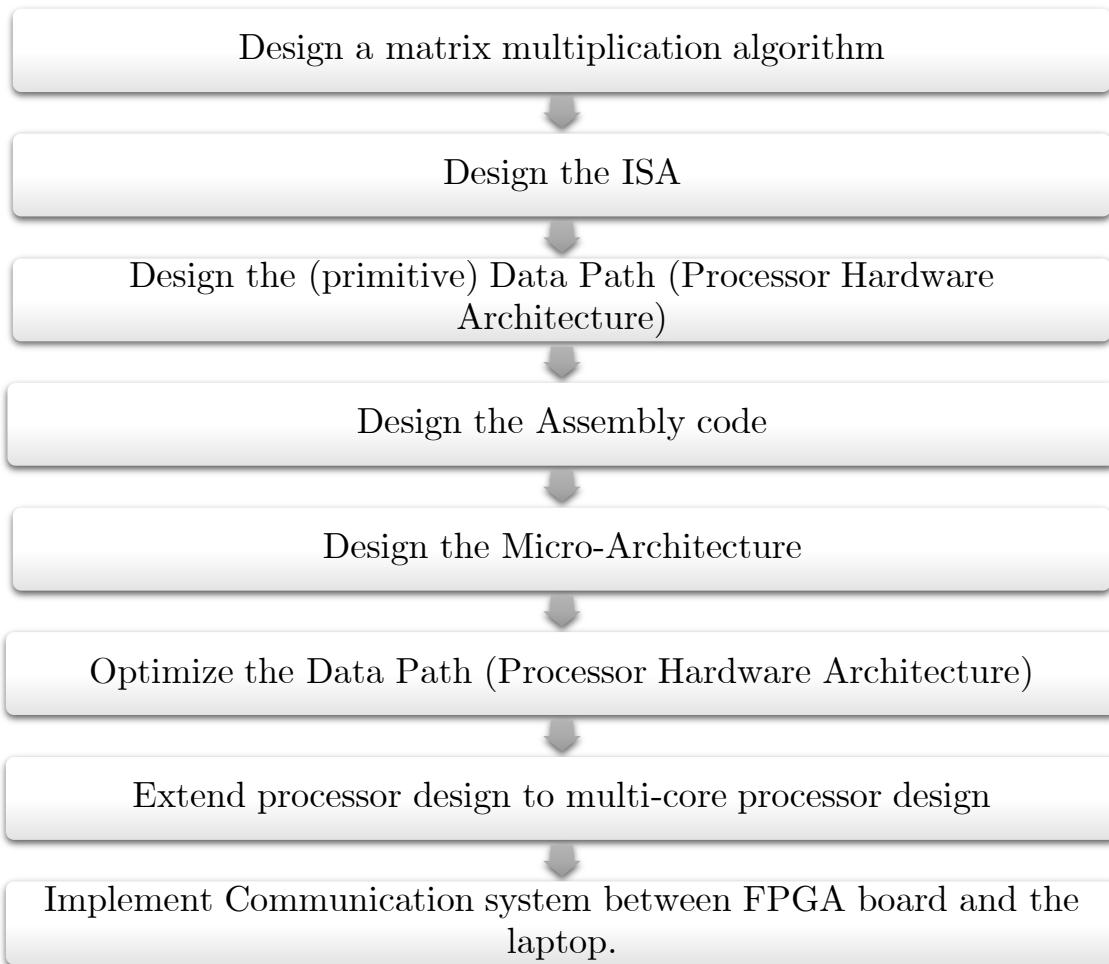


Figure 1.2: Processor Design

## 1.6 Processing order

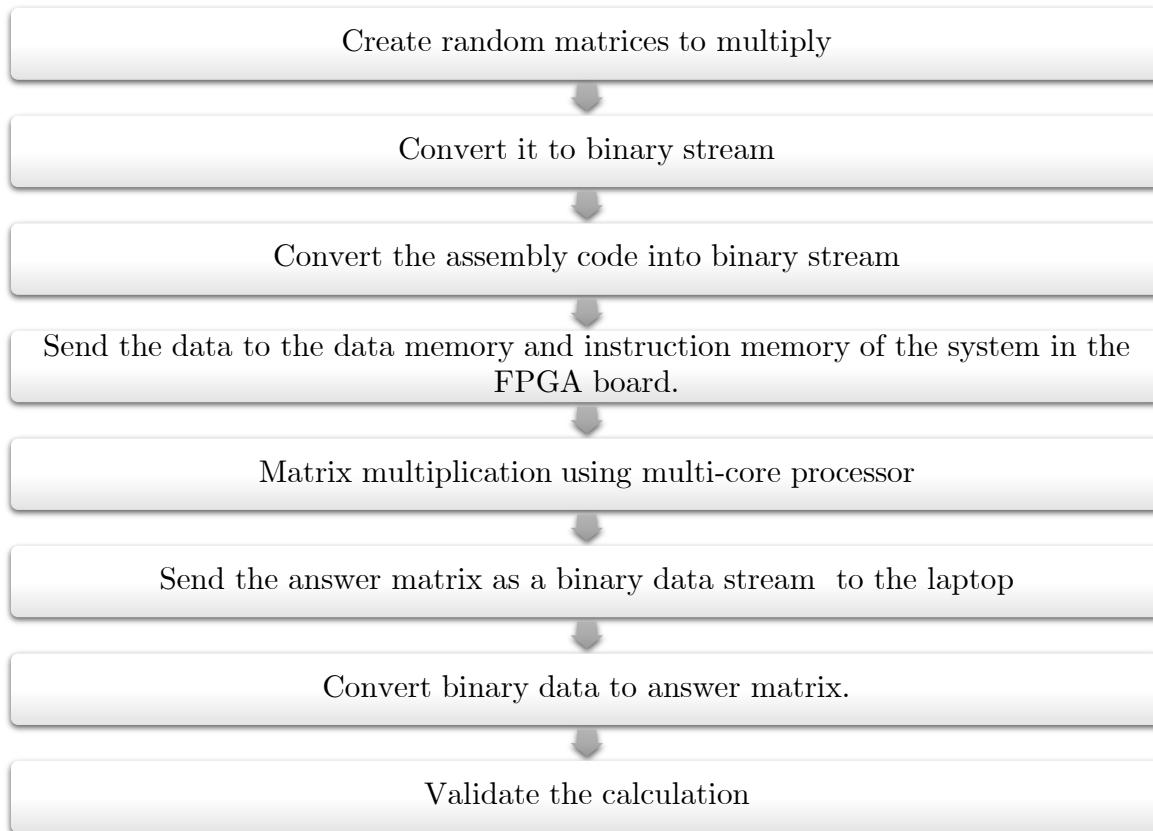


Figure 1.3: Process Flow

# 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 parallelly and independently. The process is explained with images belongs to some steps as follows.

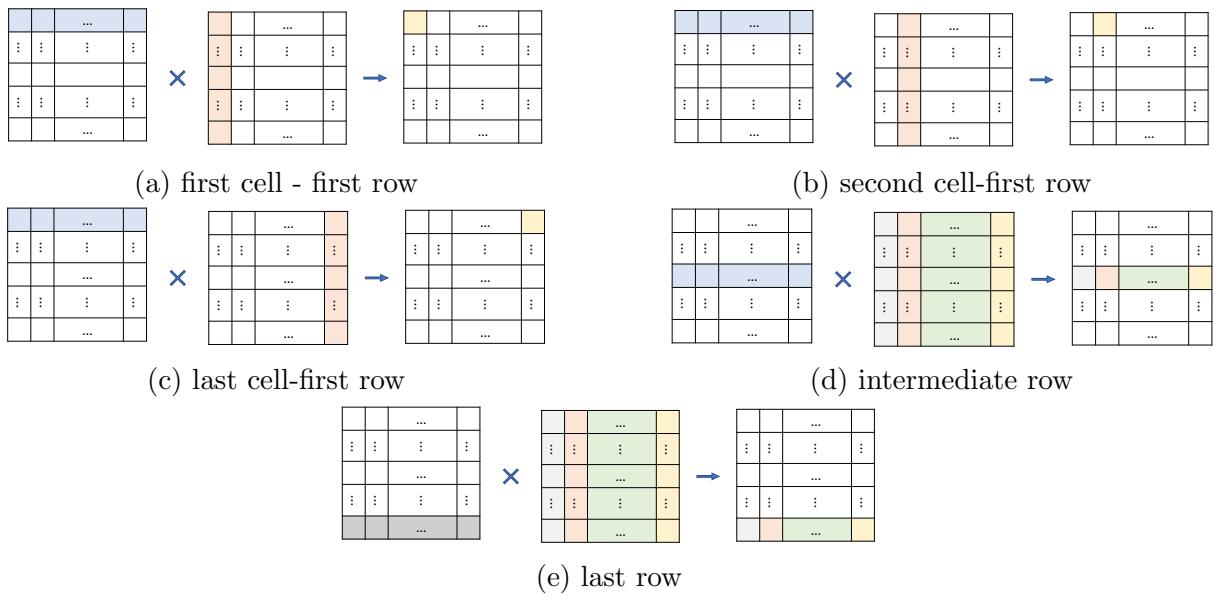


Figure 2.1: Single core multiplication algorithm

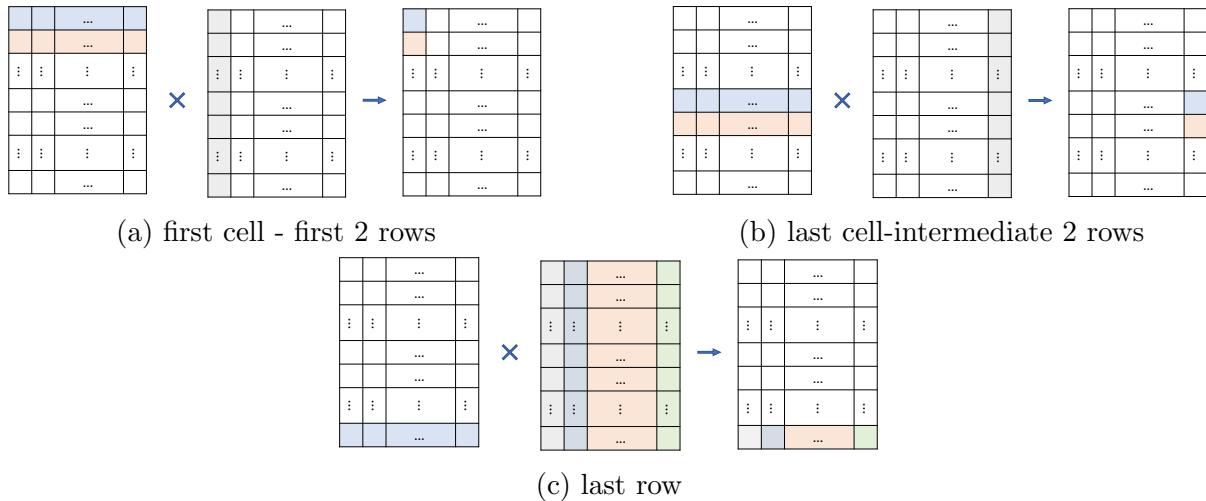


Figure 2.2: Dual-core multiplication algorithm

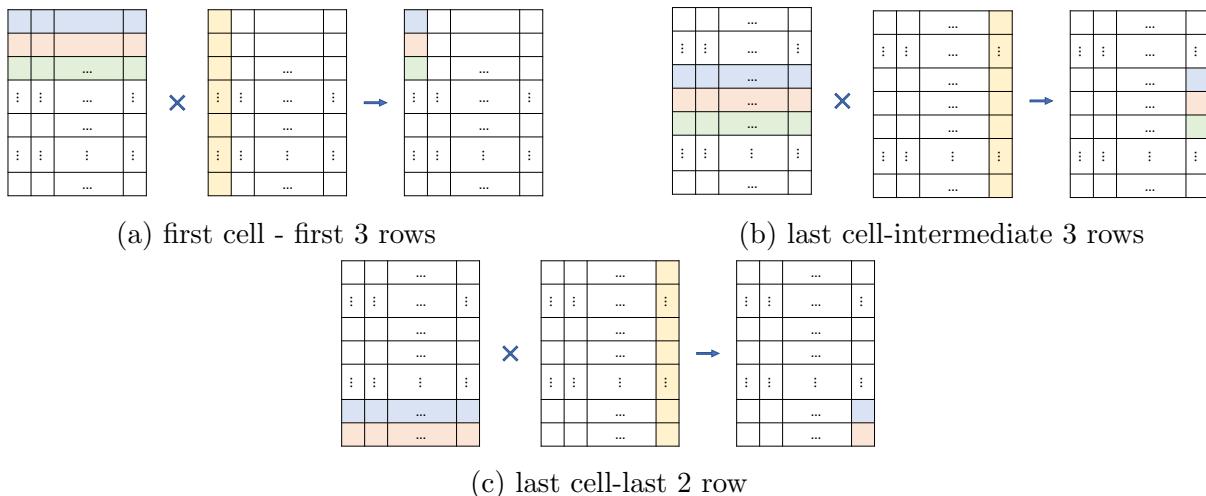


Figure 2.3: Three-core multiplication algorithm

## 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 sequentially**

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	$AC \leftarrow 0$
LDIAC	3	$AC \leftarrow \text{dataMem}\{\text{addr}\}$
LDAC	4	$AC \leftarrow \text{dataMem}\{AC\}$
STR	5	$\text{dataMem}\{AC\} \leftarrow R$
STIR	6	$\text{dataMem}\{\text{addr}\} \leftarrow R$
JUMP	7	$PC \leftarrow \text{instructionMem}\{\text{addr}\}$
JMPNZ	8	Jump if $z \neq 0$
JMPZ	9	Jump if $z == 0$
MUL	10	$AC \leftarrow AC * R1$
ADD	11	$AC \leftarrow AC + R$
SUB	12	$AC \leftarrow AC - RC$
INCAC	13	$AC \leftarrow AC + 1$
MV_RL_AC	14	$RL \leftarrow AC$
MV_RP_AC	15	$RP \leftarrow AC$
MV_RQ_AC	16	$RQ \leftarrow AC$
MV_RC_AC	17	$RC \leftarrow AC$
MV_R_AC	18	$R \leftarrow AC$
MV_R1_AC	19	$R1 \leftarrow AC$
MV_AC_RP	20	$AC \leftarrow RP$
MV_AC_RQ	21	$AC \leftarrow RQ$
MV_AC_RL	22	$AC \leftarrow RL$

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.

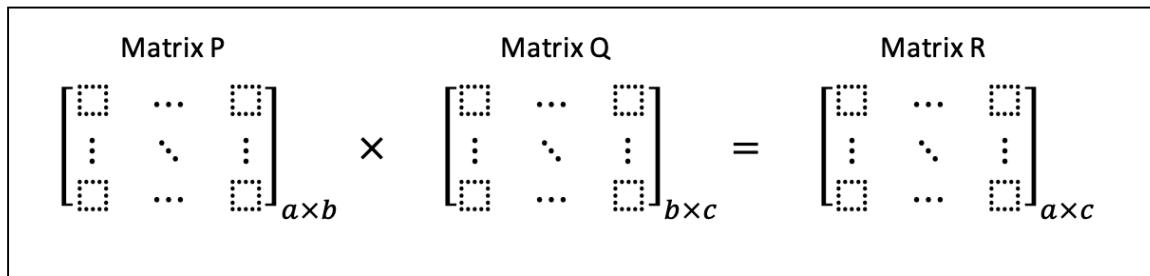


Figure 3.1: Naming convention of matrices and their dimensions

## 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

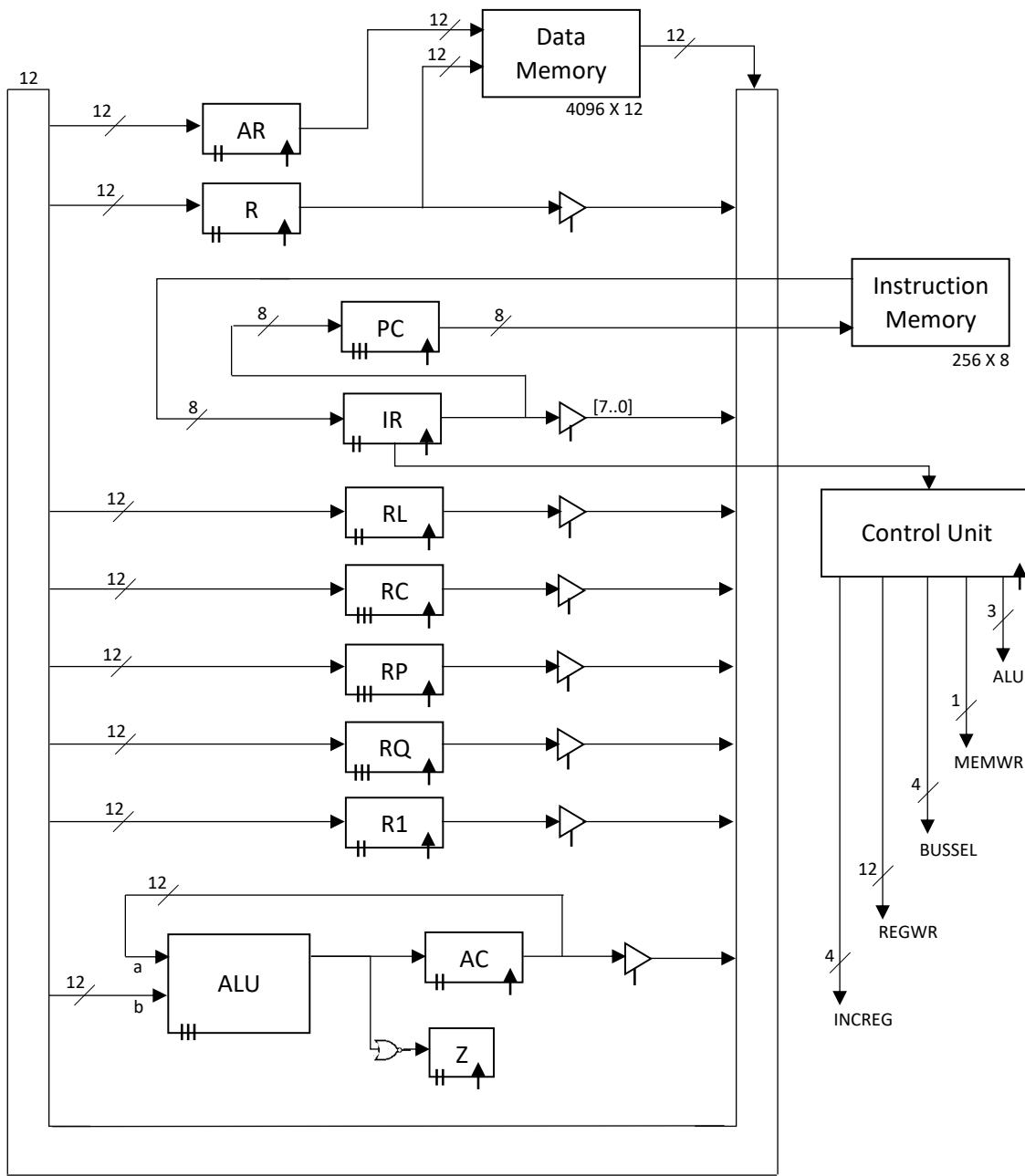


Figure 3.2: Data path of the CPU

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

### 3.4 Micro-instructions

INSTRUCTION	STATE	MICROINSTRUCTION
FETCH	FETCH_DELAY1 FETCH1 FETCH2	IR $\leftarrow$ Instruction memory, Instruction read IR $\leftarrow$ Instruction memory, Instruction read PC $\leftarrow$ PC+1
NOP	NOP1	No operation
LDAC	LDAC1 LDAC_DELAY1 LDAC2	AR $\leftarrow$ AC, Data read R $\leftarrow$ Data Memory R $\leftarrow$ Data Memory
LDIAC	LDIAC_DELAY1 LDIAC1 LDIAC2 LDIAC_DELAY2 LDIAC3	IR $\leftarrow$ Instruction Memory IR $\leftarrow$ Instruction Memory AR $\leftarrow$ IR, PC $\leftarrow$ PC+1, Data read AC $\leftarrow$ Data Memory AC $\leftarrow$ Data Memory
STR	STR1 STR_DELAY1 STR2	AR $\leftarrow$ AC, Data write Data Memory $\leftarrow$ R Data Memory $\leftarrow$ R
STIR	STIR_DELAY1 STIR1 STIR2 STIR_DELAY2 STIR3	IR $\leftarrow$ Instruction Memory IR $\leftarrow$ Instruction Memory AR $\leftarrow$ IR, PC $\leftarrow$ PC+1, Data write Data Memory $\leftarrow$ R Data Memory $\leftarrow$ R
MV_RL_AC	MV_RL_AC1	RL $\leftarrow$ AC
MV_R_AC	MV_R_AC1	R $\leftarrow$ AC
MV_RP_AC	MV_RP_AC1	RP $\leftarrow$ AC
MV_RQ_AC	MV_RQ_AC1	RQ $\leftarrow$ AC
MV_RC_AC	MV_RC_AC1	RC $\leftarrow$ AC
MV_R1_AC	MV_R1_AC1	R1 $\leftarrow$ AC
MV_AC_RP	MV_AC_RP1	AC $\leftarrow$ RP
MV_AC_RQ	MV_AC_RQ1	AC $\leftarrow$ RQ
MV_AC_RL	MV_AC_RL1	AC $\leftarrow$ RL
JUMP	JUMP_DELAY1 JUMP1 JUMP2	IR $\leftarrow$ Instruction Memory IR $\leftarrow$ Instruction Memory PC $\leftarrow$ IR
JMPZ	JMPZY_DELAY1 JMPZY1 JMPZY2 JMPZN1	IR $\leftarrow$ Instruction Memory IR $\leftarrow$ Instruction Memory PC $\leftarrow$ IR PC $\leftarrow$ PC+1
JMPNZ	JMPNZY_DELAY1 JMPNZY1 JMPNZY2 JMPNZN1	IR $\leftarrow$ Instruction Memory IR $\leftarrow$ Instruction Memory PC $\leftarrow$ IR PC $\leftarrow$ PC+1
CLAC	CLAC1	AC $\leftarrow$ 0
ADD	ADD1	AC $\leftarrow$ 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	$AC \leftarrow AC - RC$
MUL	MUL1	$AC \leftarrow AC * R1$ , $RP \leftarrow RP + 1$ , $RQ \leftarrow RQ + 1$ , $RC \leftarrow RC + 1$
INCAC	INCAC1	$AC \leftarrow AC + 1$
ENDOP	ENDOP1	End Operation

### 3.5 State Diagram

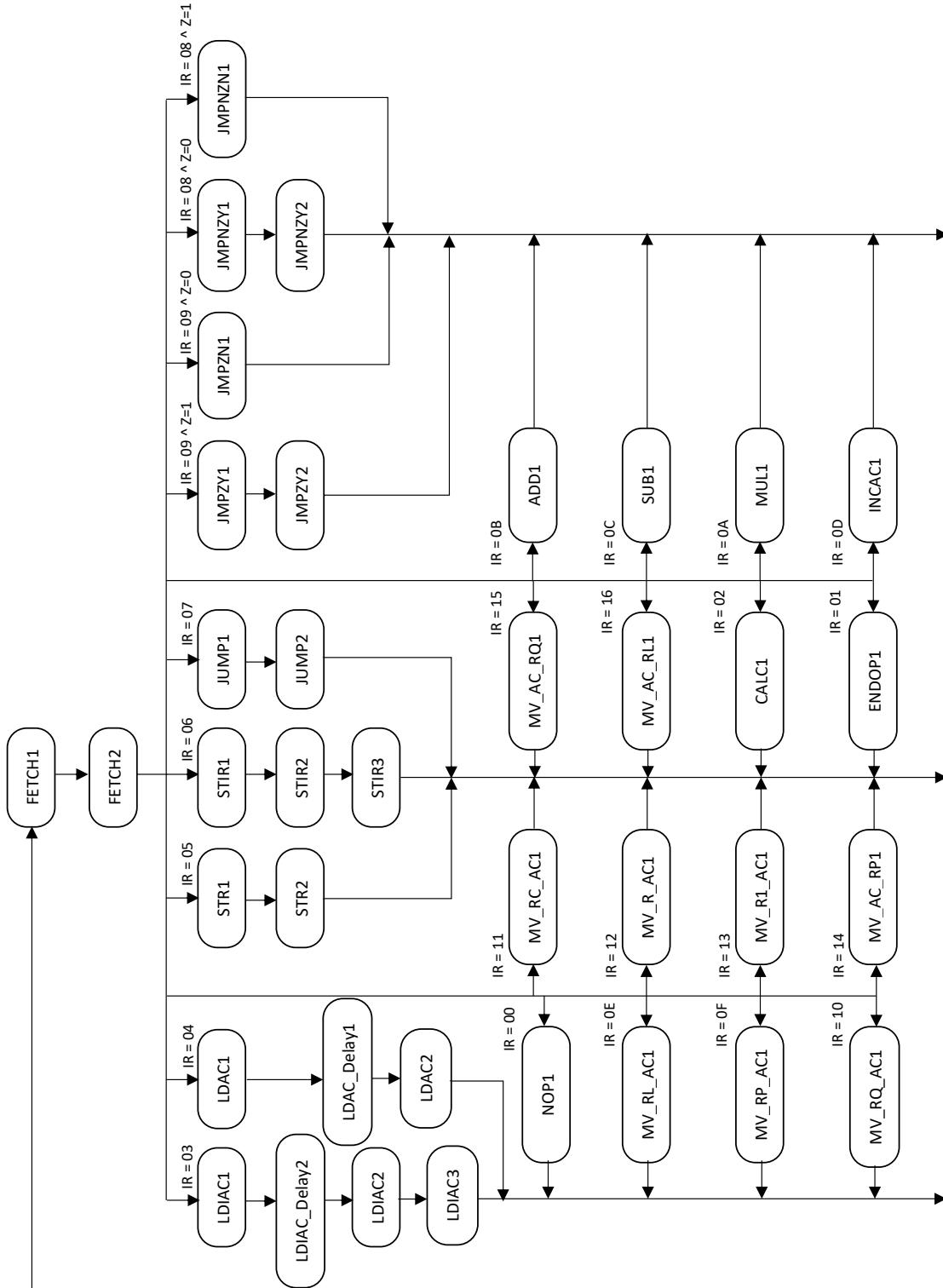


Figure 3.3: State diagram of the CPU

# 4 Modules

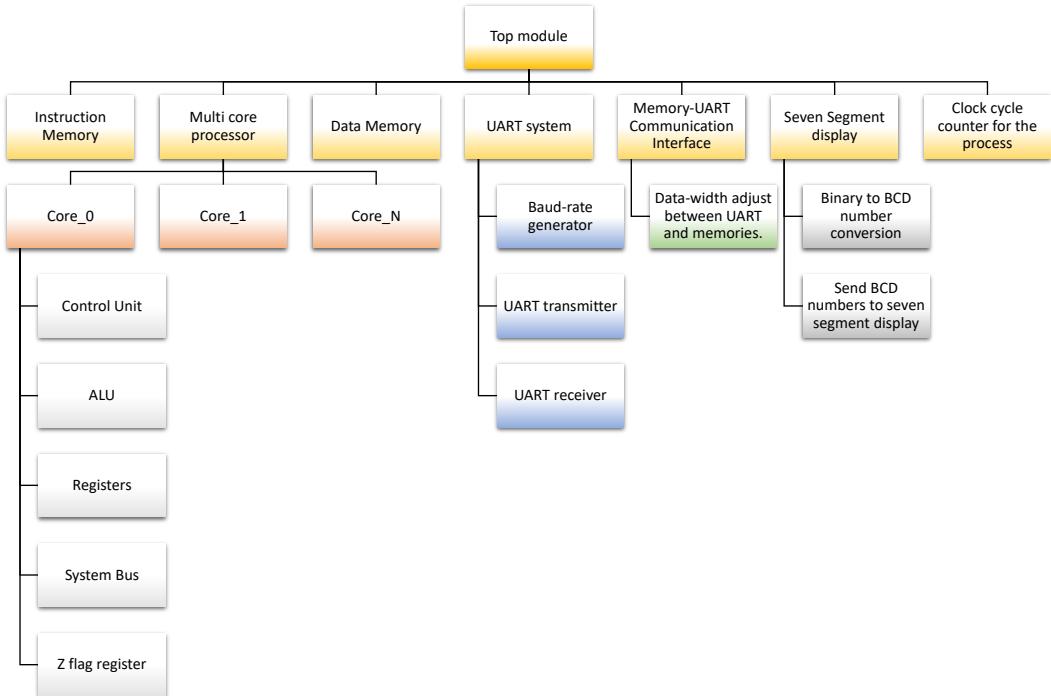


Figure 4.1: System Hierarchy

## 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.)

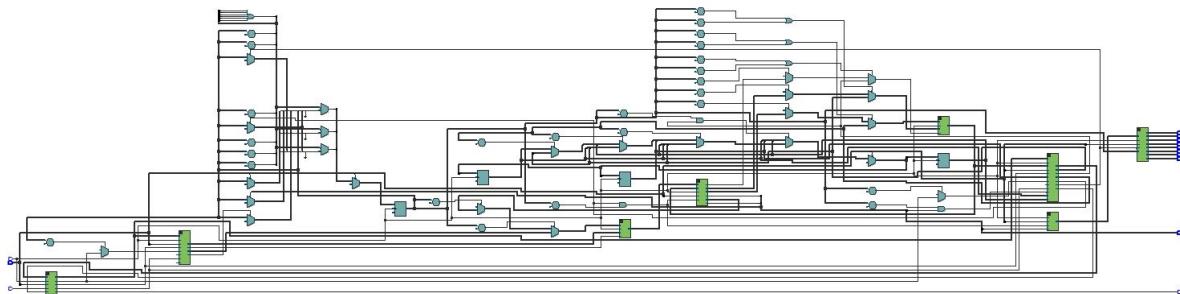


Figure 4.2: Block diagram of Top Module

## 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.

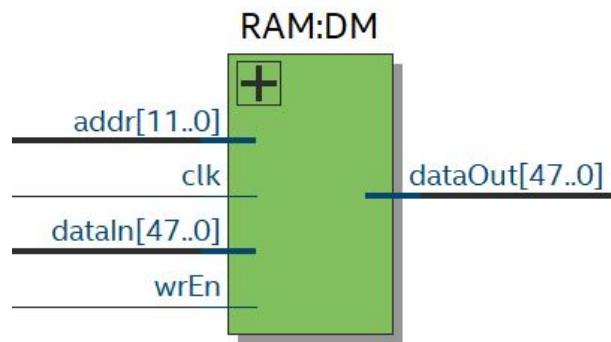


Figure 4.3: Block diagram of Data Memory

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								
.	.				.			.
.	.				.			.
.	.				.			.
0xFFFF								

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).

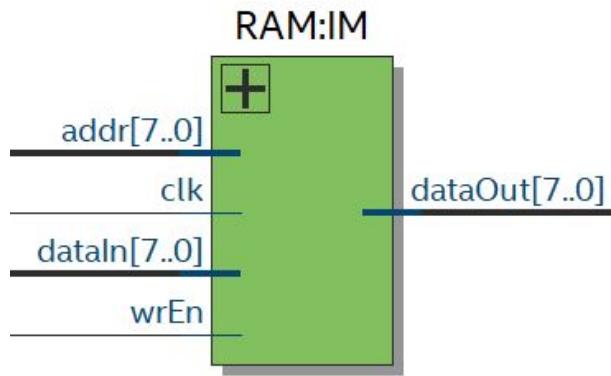


Figure 4.4: Block diagram of Instruction Memory

## 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.

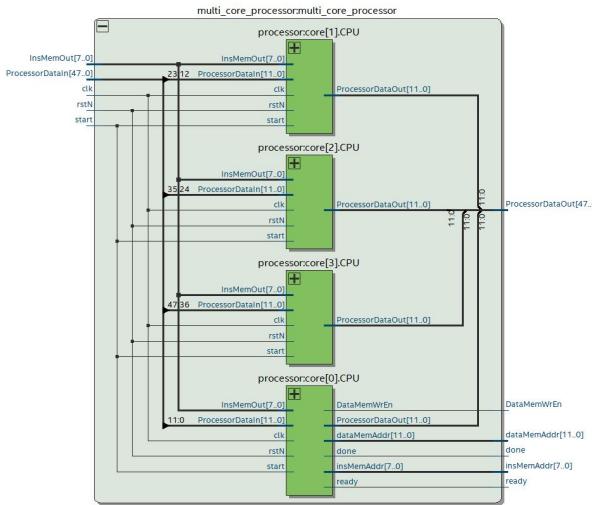


Figure 4.5: Block diagram of Multi-core processor

## 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.

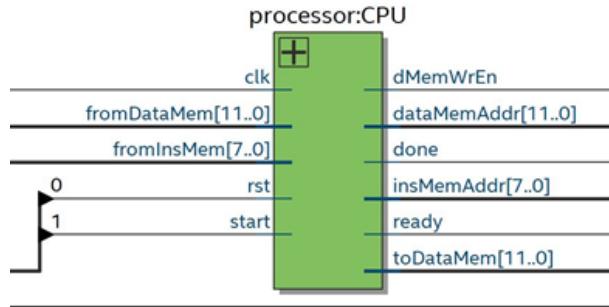


Figure 4.6: Block diagram of Processor

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.

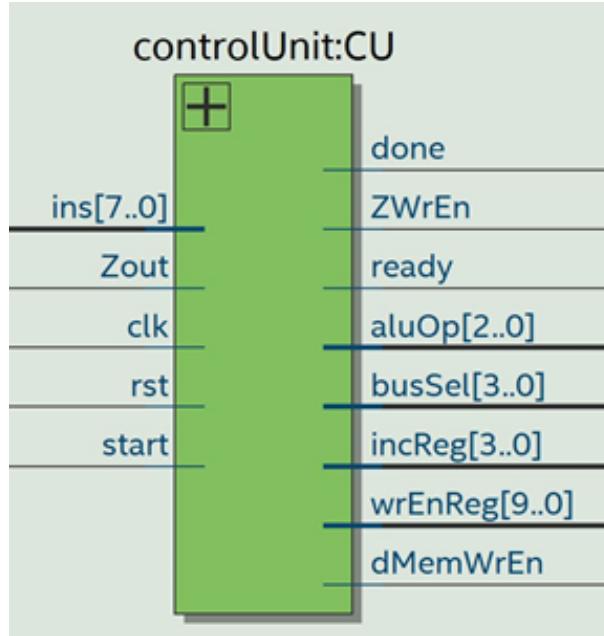


Figure 4.7: Block diagram of Control Unit

- **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).

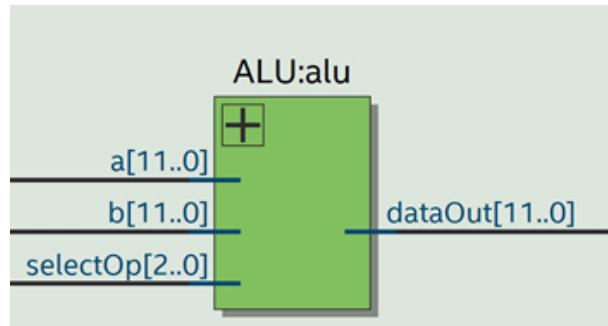


Figure 4.8: Block diagram of ALU

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

selectOp	Operation	Description
000	clr	$\text{dataOut} \leftarrow 0$
001	pass	$\text{dataOut} \leftarrow b$
010	add	$\text{dataOut} \leftarrow a+b$
011	sub	$\text{dataOut} \leftarrow a-b$
100	mul	$\text{dataOut} \leftarrow a * b$
101	inc	$\text{dataOut} \leftarrow \text{dataOut} + 1$
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.

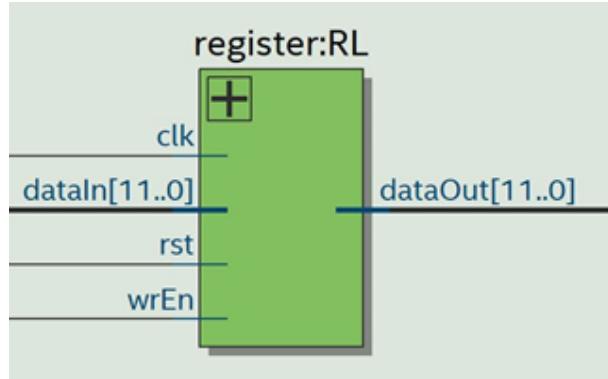


Figure 4.9: Block diagram of Register

## 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.

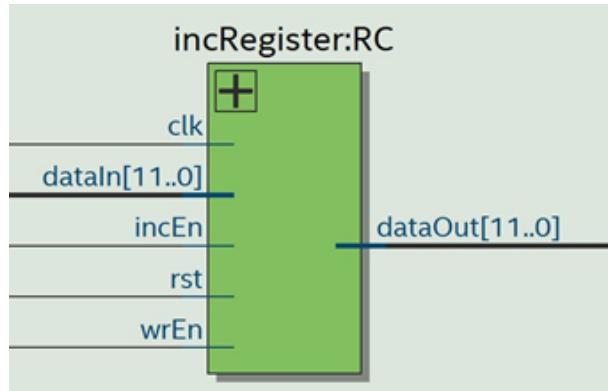


Figure 4.10: Block diagram of Inc\_Register

## 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.

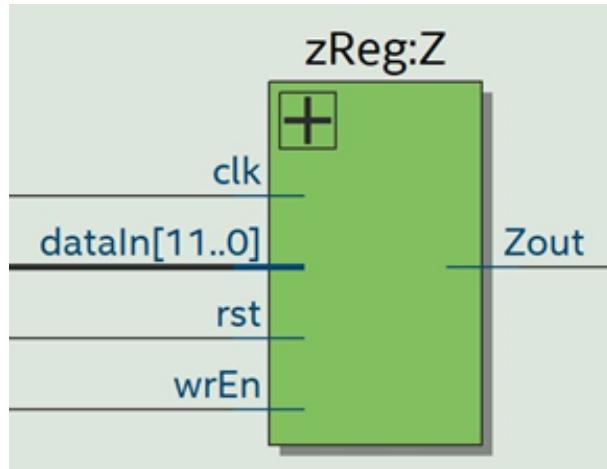


Figure 4.11: Block diagram of Z Register

## 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.

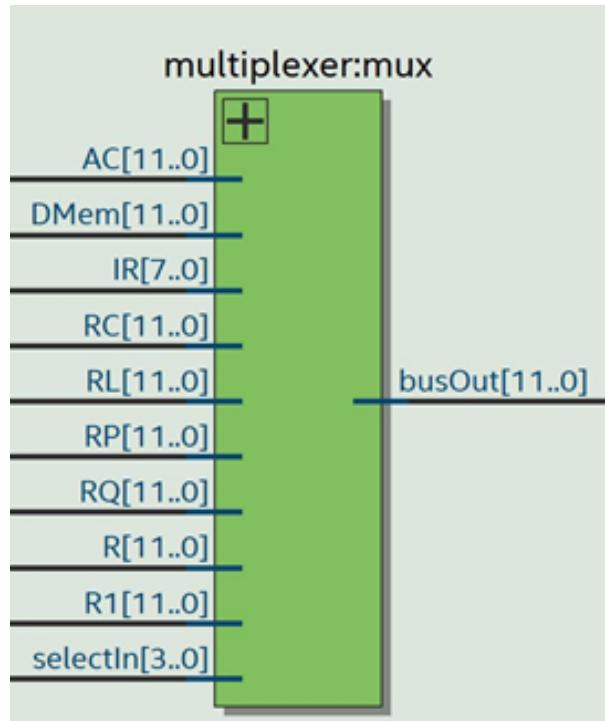


Figure 4.12: Block diagram of Multiplexer

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 testing 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

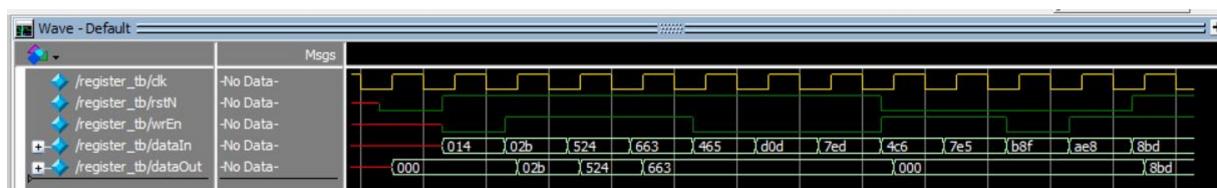


Figure 5.2: Register Simulation

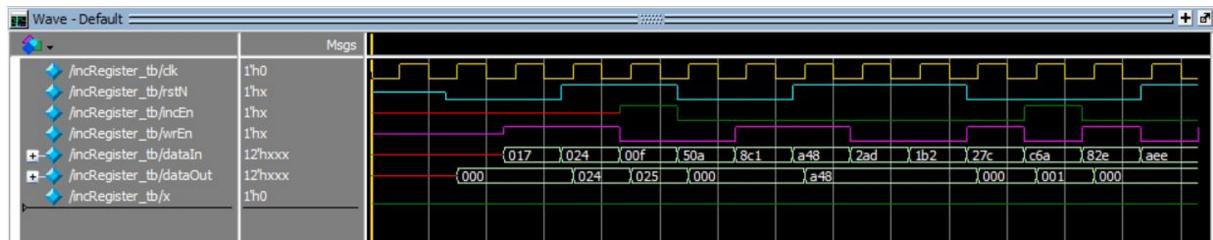


Figure 5.3: Inc Register Simulation

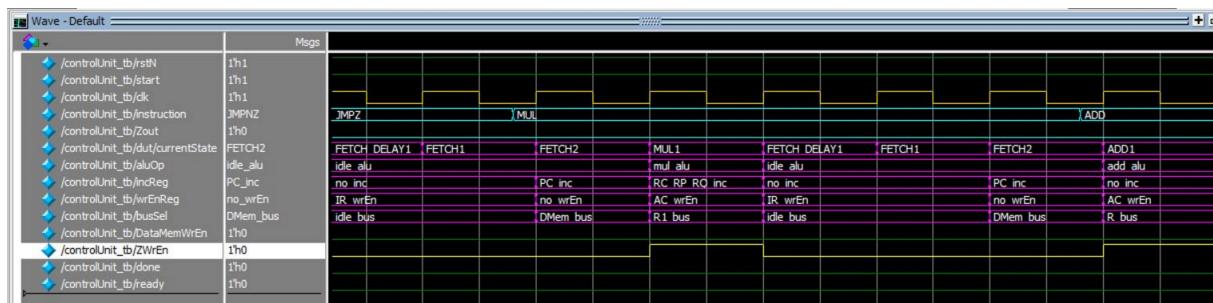


Figure 5.4: Control Unit Simulation

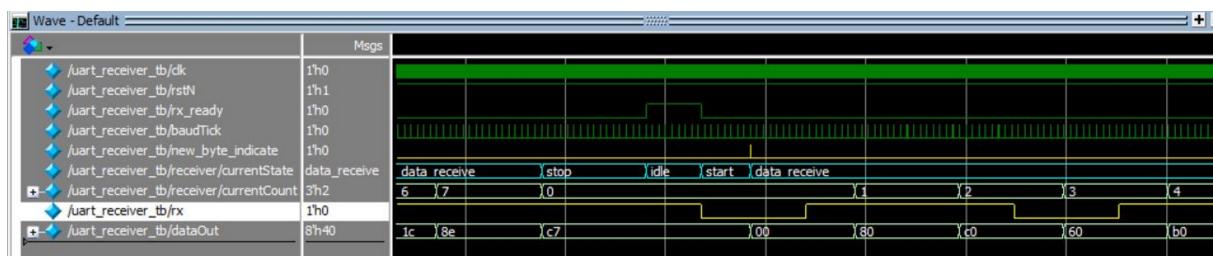


Figure 5.5: UART Receiver Simulation

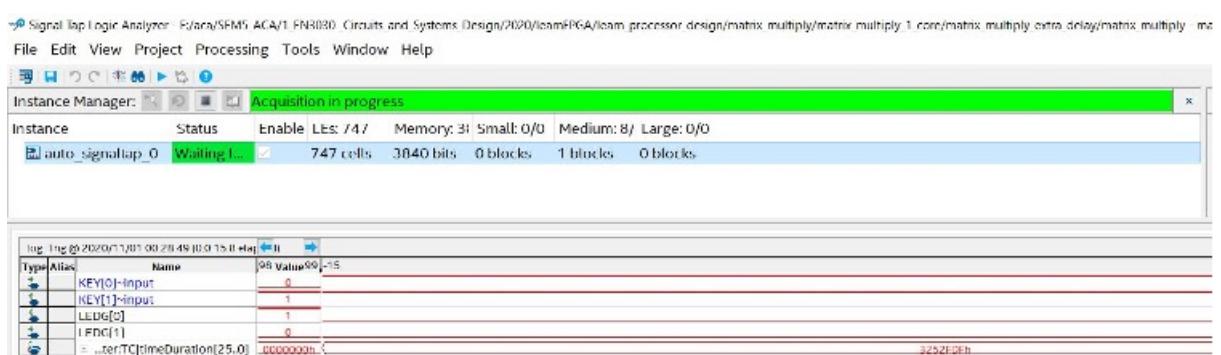


Figure 5.6: Signal Tap Logic Analyser Usage

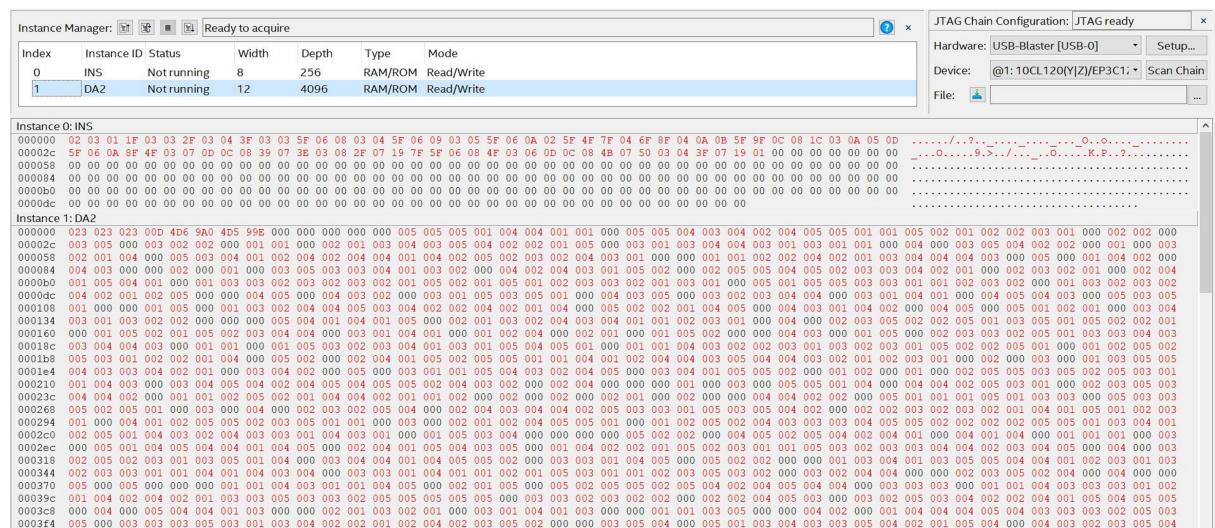


Figure 5.7: In System Memory Content Editor Usage

# 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 processor 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.

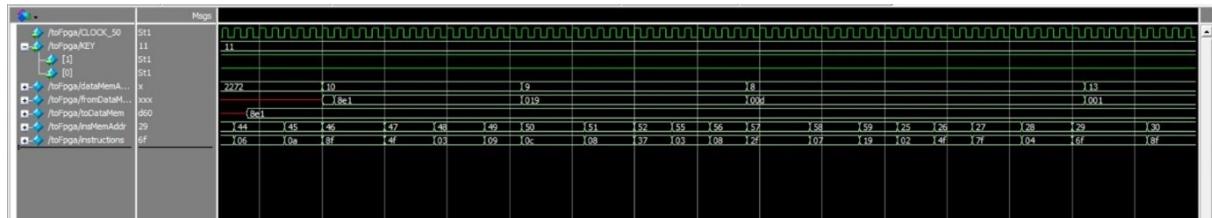


Figure 6.1: Simulation of memory access of the processor

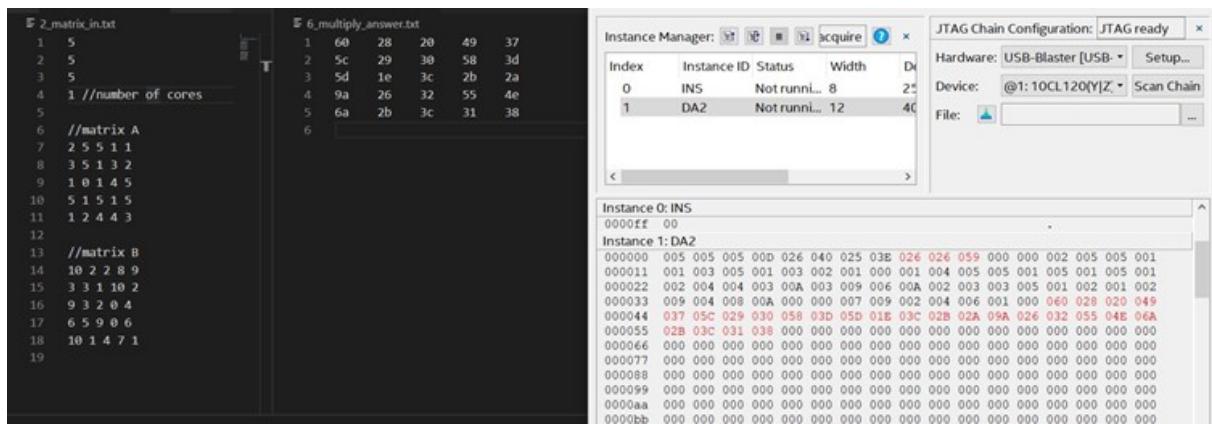


Figure 6.2: Comparison of Two 5x5 Matrices Multiplication Results of a Single-Core Processor & Python Script

```

2_matrix_in.txt
1 8
2 9
3 10
4 1 //number of cores
5
6 //matrix A
7 3 3 3 4 1 4 0 0 1
8 4 3 2 2 0 2 4 2 1
9 1 5 5 5 1 0 5 4 1
10 1 2 3 2 0 0 1 2 3
11 1 0 1 1 4 1 3 4 1
12 2 5 1 0 3 5 0 1 5
13 4 5 4 0 1 2 5 3 3
14 1 4 2 5 0 3 1 5 5
15
16 //matrix B
17 3 6 2 5 8 9 8 5 0 0
18 0 0 5 2 5 4 0 4 1 0
19 10 6 6 3 4 2 9 6 7 3
20 1 3 10 5 0 5 9 5 7 5
21 1 0 0 8 0 1 10 5 10 7
22 3 4 9 0 4 5 0 0 9 5
23 2 5 10 0 10 7 1 4 7 0
24 1 0 7 0 9 2 10 10 0 9
25 10 6 8 5 5 9 8 9 4 2
26

```

```

6.multiply_answer.txt
1 42 46 7b 3f 48 5f 69 4f 66 3a
2 3c 4c 87 2f 7e 71 64 63 51 2e
3 53 52 c1 44 90 75 a1 9a 7c 55
4 45 35 62 2b 49 47 62 5c 38 2b
5 29 28 5d 32 57 3f 75 61 58 4f
6 55 44 7f 48 63 73 69 6a 6b 44
7 66 63 aa 41 ad 8f 89 92 6c 3e
8 5e 50 c4 45 80 83 a2 9d 6b 65
9

```

```

6.multiply.py
# 2_matrix_in.txt
1 5
2 6
3 7
4 2 //number of cores
5
6 //matrix A
7 3 5 0 0 4 4
8 2 4 0 3 5 4
9 2 1 3 5 5 2
10 0 0 2 2 4 4
11 5 5 1 0 2 5
12
13 //matrix B
14 10 1 10 3 7 4 4
15 0 4 4 5 4 10 1
16 10 9 8 9 6 4 0
17 9 8 6 10 6 2 7
18 1 5 10 9 4 4 9
19 5 3 2 7 4 8 9
20

```

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

```

2_matrix_in.txt
1 5
2 6
3 7
4 2 //number of cores
5
6 //matrix A
7 3 5 0 0 4 4
8 2 4 0 3 5 4
9 2 1 3 5 5 2
10 0 0 2 2 4 4
11 5 5 1 0 2 5
12
13 //matrix B
14 10 1 10 3 7 4 4
15 0 4 4 5 4 10 1
16 10 9 8 9 6 4 0
17 9 8 6 10 6 2 7
18 1 5 10 9 4 4 9
19 5 3 2 7 4 8 9
20

```

```

6.multiply_answer.txt
1 36 37 62 62 49 6e 59
2 48 4f 78 81 54 6a 72
3 66 68 84 93 5e 4c 6b
4 3e 42 4c 66 38 3c 56
5 57 3b 6c 66 59 7a 58
6

```

```

6.multiply.py
# 2_matrix_in.txt
1 5
2 6
3 7
4 2 //number of cores
5
6 //matrix A
7 3 5 0 0 4 4
8 2 4 0 3 5 4
9 2 1 3 5 5 2
10 0 0 2 2 4 4
11 5 5 1 0 2 5
12
13 //matrix B
14 10 1 10 3 7 4 4
15 0 4 4 5 4 10 1
16 10 9 8 9 6 4 0
17 9 8 6 10 6 2 7
18 1 5 10 9 4 4 9
19 5 3 2 7 4 8 9
20

```

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

<pre> 2_matrix_in.txt 1 7 2 12 3 9 4 1 //number of cores 5 6 //matrix A 7 2 4 3 3 5 5 3 0 0 3 0 5 8 4 2 1 1 1 2 4 3 4 3 2 4 9 3 2 4 4 4 1 0 3 1 1 3 5 10 2 0 3 0 2 4 0 1 0 2 2 4 11 2 2 5 0 0 0 5 0 4 1 1 4 12 0 1 1 1 1 1 0 3 4 3 2 3 13 3 5 4 4 0 3 4 3 1 1 4 5 14 15 //matrix B 16 6 8 0 7 8 10 0 3 1 17 0 3 5 1 6 7 9 6 1 18 3 6 2 1 7 4 2 8 10 19 1 4 2 8 6 6 0 2 1 20 9 3 1 4 6 8 10 1 9 21 10 10 8 1 5 3 7 6 6 22 8 1 6 6 6 9 10 6 0 23 2 1 4 9 4 7 1 6 6 24 10 5 2 7 6 3 5 3 8 25 8 0 5 2 0 0 2 6 5 26 6 5 4 0 3 2 9 7 6 27 8 10 5 3 10 7 10 4 0 28 </pre>	<pre> 6_multiply_answer.txt 1 cf b0 87 6d ca c3 d5 97 81 2 cb 94 76 8e b3 b3 ab 95 73 3 a2 a5 5e 7f c2 b8 a0 8b 8c 4 8d 83 52 36 77 63 75 68 66 5 99 7a 57 5d a0 8d 95 81 61 6 81 59 4c 55 66 56 69 61 68 7 b8 bf 95 8c e1 d9 d2 c4 7d 8 </pre>	<pre> 15_answer_matrix_from_processor(UART).txt × 15_answer_matrix_from_processor(UART).txt 1 cf b0 87 6d ca c3 d5 97 81 2 cb 94 76 8e b3 b3 ab 95 73 3 a2 a5 5e 7f c2 b8 a0 8b 8c 4 8d 83 52 36 77 63 75 68 66 5 99 7a 57 5d a0 8d 95 81 61 6 81 59 4c 55 66 56 69 61 68 7 b8 bf 95 8c e1 d9 d2 c4 7d 8 </pre>
--	---	---

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

<pre> <b>2_matrix_in.txt</b> 1 10 2 7 3 9 4 2 //number of cores 5 6 //matrix A 7 2 2 3 3 5 2 0 8 2 3 5 4 1 3 0 9 5 1 2 1 3 5 1 10 3 3 4 3 5 2 5 11 2 4 0 5 1 1 5 12 0 4 5 5 0 3 1 13 0 0 2 2 2 1 4 14 2 3 0 2 0 5 3 15 4 0 1 2 5 4 1 16 3 2 1 5 3 5 5 17 18 //matrix B 19 8 5 2 1 7 0 7 1 4 20 3 10 2 0 3 10 4 9 5 21 9 4 10 5 3 3 0 5 7 22 7 6 4 9 1 2 2 1 10 23 6 7 10 6 2 10 6 10 3 24 0 9 9 8 9 6 5 0 0 25 6 2 10 3 2 6 7 10 6 26 </pre>	<pre> <b>6_multiply_answer.txt</b> 1 64 71 76 5a 3c 61 44 58 54 2 68 76 71 5d 47 51 37 44 65 3 5c 75 79 55 52 54 5b 41 40 4 96 8e b6 6f 53 8c 72 99 82 5 63 6a 65 4c 34 60 56 67 6f 6 62 77 73 61 3d 59 30 4c 6f 7 44 33 61 3c 1d 3c 31 48 40 8 39 67 5d 45 4c 52 4c 3d 3d 9 5b 6d 7a 5c 51 57 59 47 40 10 7a 91 a5 7e 60 7b 75 6f 76 11 </pre>
<pre> <b>15_answer_matrix_from_processor(UART).txt</b> <b>15_answer_matrix_from_processor(UART).txt</b> 1 64 71 76 5a 3c 61 44 58 54 2 68 76 71 5d 47 51 37 44 65 3 5c 75 79 55 62 54 5b 41 40 4 96 8e b6 6f 53 8c 72 99 82 5 63 6a 65 4c 34 60 56 67 6f 6 62 77 73 61 3d 59 30 4c 6f 7 44 33 61 3c 1d 3c 31 48 40 8 39 67 5d 45 4c 52 4c 3d 3d 9 5b 6d 7a 5c 51 57 59 47 40 10 7a 91 a5 7e 60 7b 75 6f 76 11 </pre>	

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

```

2_matrix_in.txt
1 10
2 8
3 12
4 3 //number of cores
5
6 //matrix A
7 0 2 0 3 2 4 3 0
8 3 0 5 5 2 2 4 2
9 0 0 2 2 2 2 2 2
10 3 5 5 3 4 2 3 2
11 2 4 1 4 4 2 0 2
12 1 1 3 2 2 4 5 0
13 5 4 4 4 1 0 2 1
14 1 1 3 4 3 1 2 3
15 1 4 0 5 2 1 0 0
16 4 2 1 2 3 3 2 4
17
18 //matrix B
19 6 1 8 7 8 7 7 6 10 0 10 4
20 7 9 9 4 3 0 4 1 8 1 7 7
21 6 4 2 5 5 9 2 7 4 1 10 5
22 2 8 9 8 6 2 7 10 7 2 2 8
23 3 8 7 8 8 0 1 0 4 4 0 2
24 5 5 5 5 6 8 2 10 7 3 7 3
25 3 8 1 7 4 9 0 4 9 0 2 4
26 4 10 3 0 10 8 4 5 6 2 7 4
27

6_multiply_answer.txt
1 37 66 52 59 4c 41 27 54 64 1c 36 42
2 5e 8d 71 8c 8f 90 50 95 9b 21 7e 6f
3 2e 56 36 42 4e 48 20 48 4a 18 38 34
4 80 b2 99 99 9e 83 58 82 b4 2a 9b 82
5 54 88 86 6d 79 3f 4c 5d 7e 27 5e 5f
6 4c 72 50 71 62 73 29 6c 7d 1c 59 4e
7 67 7b 84 7d 7a 69 5c 73 9a 16 89 72
8 47 81 60 65 76 5c 3e 65 71 21 57 57
9 37 62 6c 54 48 19 3e 46 5c 19 37 4f
10 5e 89 78 6e 91 73 4d 6f 95 24 79 5a
11

15_answer_matrix_from_processor(UART).txt ×
15_answer_matrix_from_processor(UART).txt
1 37 66 52 59 4c 41 27 54 64 1c 36 42
2 5e 8d 71 8c 8f 90 50 95 9b 21 7e 6f
3 2e 56 36 42 4e 48 20 48 4a 18 38 34
4 80 b2 99 99 9e 83 58 82 b4 2a 9b 82
5 54 88 86 6d 79 3f 4c 5d 7e 27 5e 5f
6 4c 72 50 71 62 73 29 6c 7d 1c 59 4e
7 67 7b 84 7d 7a 69 5c 73 9a 16 89 72
8 47 81 60 65 76 5c 3e 65 71 21 57 57
9 37 62 6c 54 48 19 3e 46 5c 19 37 4f
10 5e 89 78 6e 91 73 4d 6f 95 24 79 5a
11

```

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

```

2_matrix_in.txt
1 5
2 7
3 4
4 4 //number of cores
5
6 //matrix A
7 4 1 5 5 4 5 1
8 5 4 4 1 4 1 3
9 1 2 2 0 5 2 3
10 1 0 1 5 4 2 3
11 0 2 1 3 2 5 5
12
13 //matrix B
14 10 0 4 7
15 6 6 8 5
16 0 1 9 0
17 1 0 10 4
18 1 7 0 0
19 0 6 7 9
20 6 10 2 10
21

6_multiply_answer.txt
1 3d 4f 9c 6c
2 61 5c 6f 62
3 2d 5b 3a 41
4 25 47 53 4b
5 2f 6b 64 75
6

15_answer_matrix_from_processor(UART).txt ×
15_answer_matrix_from_processor(UART).txt
1 3d 4f 9c 6c
2 61 5c 6f 62
3 2d 5b 3a 41
4 25 47 53 4b
5 2f 6b 64 75
6

```

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

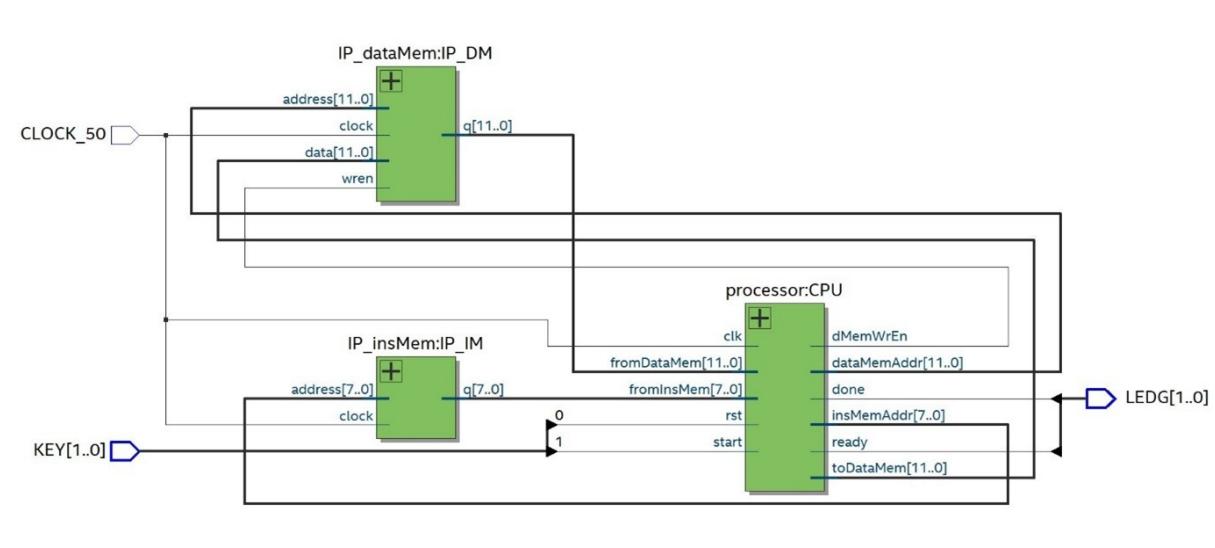


Figure 6.9: Single-Core Processor

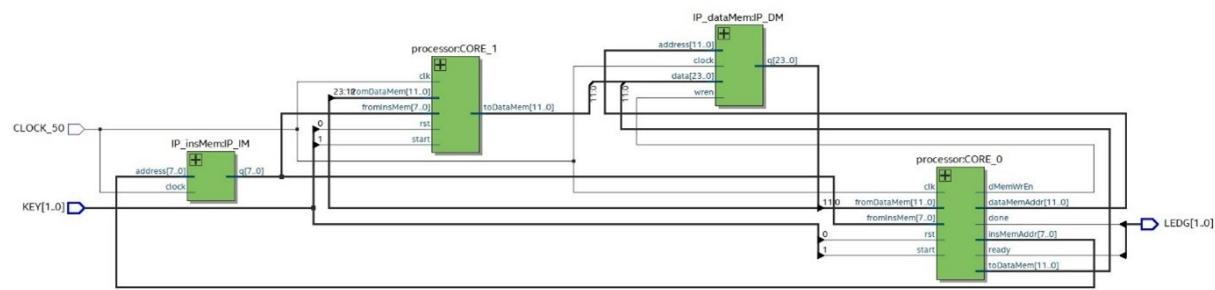


Figure 6.10: Dual-Core Processor

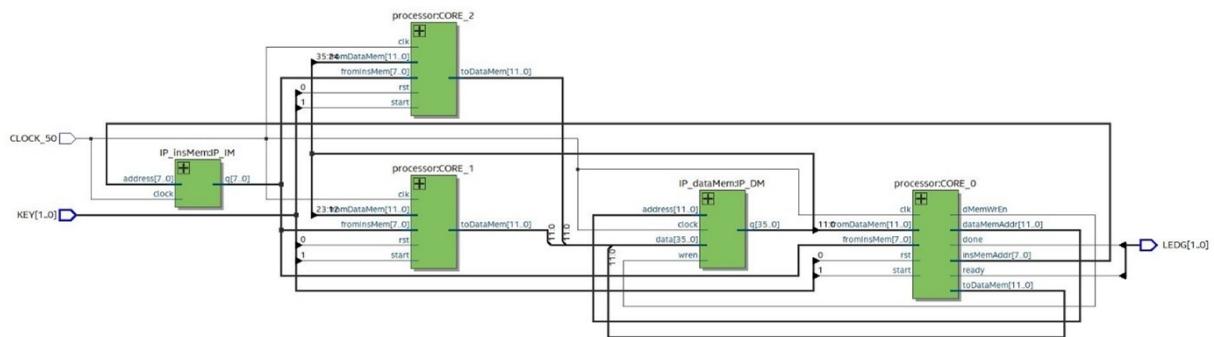


Figure 6.11: Three-Core Processor

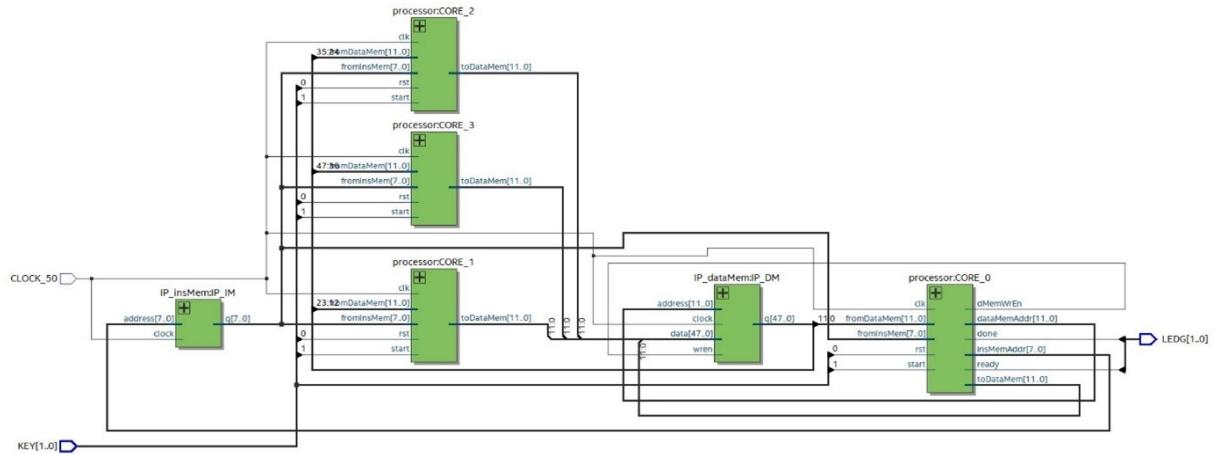


Figure 6.12: Four-Core Processor

## 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

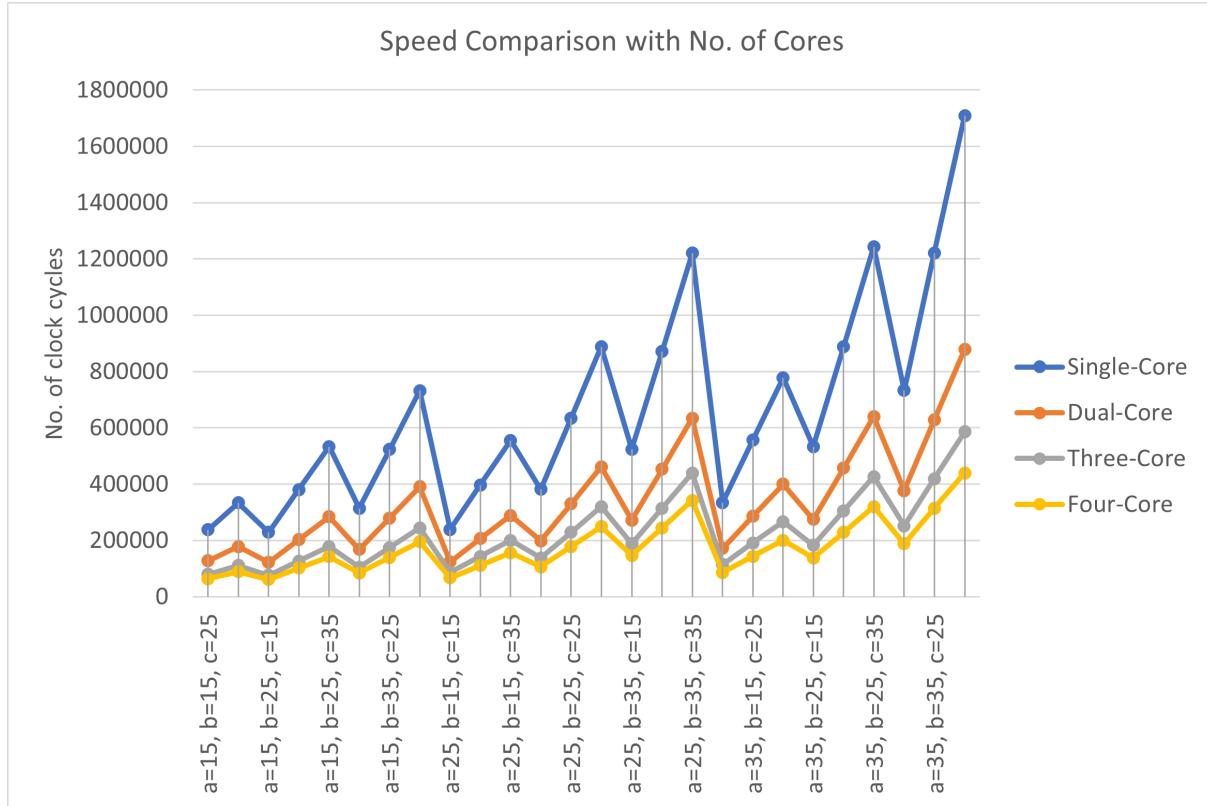


Figure 6.13: Speed Comparison of Processors with Different Number of Cores

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

Single-core Processor			
<b>a</b>	<b>b</b>	<b>c</b>	<b>No. of clock cycles</b>
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			
<b>a</b>	<b>b</b>	<b>c</b>	<b>No. of clk cycles</b>
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			
<b>a</b>	<b>b</b>	<b>c</b>	<b>No. of clock cycles</b>
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			
<b>a</b>	<b>b</b>	<b>c</b>	<b>No. of clk cycles</b>
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

Table of Contents		Flow Summary																												
<ul style="list-style-type: none"><li>Flow Summary</li><li>Flow Settings</li><li>Flow Non-Default Global Settings</li><li>Flow Elapsed Time</li><li>Flow OS Summary</li><li>Flow Log</li><li>Analysis &amp; Synthesis</li><li>Fitter</li><li>Assembler</li><li>Timing Analyzer</li><li>EDA Netlist Writer</li><li>Flow Messages</li><li>Flow Suppressed Messages</li></ul>		<input type="text"/> <<Filter>>																												
		<table><thead><tr><th>Flow Status</th><th>Successful - Sun Dec 06 12:23:29 2020</th></tr></thead><tbody><tr><td>Quartus Prime Version</td><td>18.1.0 Build 625 09/12/2018 SJ Lite Edition</td></tr><tr><td>Revision Name</td><td>matrix_multiply</td></tr><tr><td>Top-level Entity Name</td><td>toFpga</td></tr><tr><td>Family</td><td>Cyclone IV E</td></tr><tr><td>Device</td><td>EP4CE115F29C7</td></tr><tr><td>Timing Models</td><td>Final</td></tr><tr><td>Total logic elements</td><td>4,498 / 114,480 ( 4 % )</td></tr><tr><td>Total registers</td><td>2938</td></tr><tr><td>Total pins</td><td>61 / 529 ( 12 % )</td></tr><tr><td>Total virtual pins</td><td>0</td></tr><tr><td>Total memory bits</td><td>196,608 / 3,981,312 ( 5 % )</td></tr><tr><td>Embedded Multiplier 9-bit elements</td><td>8 / 532 ( 2 % )</td></tr><tr><td>Total PLLs</td><td>0 / 4 ( 0 % )</td></tr></tbody></table>	Flow Status	Successful - Sun Dec 06 12:23:29 2020	Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition	Revision Name	matrix_multiply	Top-level Entity Name	toFpga	Family	Cyclone IV E	Device	EP4CE115F29C7	Timing Models	Final	Total logic elements	4,498 / 114,480 ( 4 % )	Total registers	2938	Total pins	61 / 529 ( 12 % )	Total virtual pins	0	Total memory bits	196,608 / 3,981,312 ( 5 % )	Embedded Multiplier 9-bit elements	8 / 532 ( 2 % )	Total PLLs	0 / 4 ( 0 % )
Flow Status	Successful - Sun Dec 06 12:23:29 2020																													
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition																													
Revision Name	matrix_multiply																													
Top-level Entity Name	toFpga																													
Family	Cyclone IV E																													
Device	EP4CE115F29C7																													
Timing Models	Final																													
Total logic elements	4,498 / 114,480 ( 4 % )																													
Total registers	2938																													
Total pins	61 / 529 ( 12 % )																													
Total virtual pins	0																													
Total memory bits	196,608 / 3,981,312 ( 5 % )																													
Embedded Multiplier 9-bit elements	8 / 532 ( 2 % )																													
Total PLLs	0 / 4 ( 0 % )																													

Figure 7.1: Flow Summary

## 7.2 Analysis & Synthesis Summary

Table of Contents		Analysis & Synthesis Summary																								
<ul style="list-style-type: none"><li>Analysis &amp; Synthesis<ul style="list-style-type: none"><li>Summary</li><li>Settings</li><li>Parallel Compilation</li><li>Source Files Read</li><li>Resource Usage Summary</li><li>Resource Utilization by Entity</li><li>RAM Summary</li><li>DSP Block Usage Summary</li><li>State Machines</li><li>Optimization Results</li><li>Source Assignments</li><li>Parameter Settings by Entity Instance</li><li>LPM Parameter Settings</li><li>Connectivity Checks</li><li>Post-Synthesis Netlist Statistics for T</li><li>Elapsed Time Per Partition</li><li>Messages</li><li>Suppressed Messages</li></ul></li></ul>		<input type="text"/> <<Filter>>																								
		<table><thead><tr><th>Analysis &amp; Synthesis Status</th><th>Successful - Sun Dec 06 12:22:42 2020</th></tr></thead><tbody><tr><td>Quartus Prime Version</td><td>18.1.0 Build 625 09/12/2018 SJ Lite Edition</td></tr><tr><td>Revision Name</td><td>matrix_multiply</td></tr><tr><td>Top-level Entity Name</td><td>toFpga</td></tr><tr><td>Family</td><td>Cyclone IV E</td></tr><tr><td>Total logic elements</td><td>5,692</td></tr><tr><td>Total registers</td><td>2938</td></tr><tr><td>Total pins</td><td>61</td></tr><tr><td>Total virtual pins</td><td>0</td></tr><tr><td>Total memory bits</td><td>196,608</td></tr><tr><td>Embedded Multiplier 9-bit elements</td><td>8</td></tr><tr><td>Total PLLs</td><td>0</td></tr></tbody></table>	Analysis & Synthesis Status	Successful - Sun Dec 06 12:22:42 2020	Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition	Revision Name	matrix_multiply	Top-level Entity Name	toFpga	Family	Cyclone IV E	Total logic elements	5,692	Total registers	2938	Total pins	61	Total virtual pins	0	Total memory bits	196,608	Embedded Multiplier 9-bit elements	8	Total PLLs	0
Analysis & Synthesis Status	Successful - Sun Dec 06 12:22:42 2020																									
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition																									
Revision Name	matrix_multiply																									
Top-level Entity Name	toFpga																									
Family	Cyclone IV E																									
Total logic elements	5,692																									
Total registers	2938																									
Total pins	61																									
Total virtual pins	0																									
Total memory bits	196,608																									
Embedded Multiplier 9-bit elements	8																									
Total PLLs	0																									

Figure 7.2: Analysis & Synthesis Summary

## 7.3 Filter Summary

Table of Contents		Filter Summary
>	Analysis & Synthesis	
>	Filter	
	Summary	
	Settings	
	Parallel Compilation	
	Netlist Optimizations	
	Ignored Assignments	
>	Incremental Compilation Section	
	Pin-Out File	
>	Resource Section	
>	I/O Rules Section	
	Device Options	
	Operating Settings and Conditions	
	Messages	
	Suppressed Messages	
>	Assembler	
>	Timing Analyzer	
>	EDA Netlist Writer	
	Flow Messages	

Figure 7.3: Filter Summary

## 7.4 Assembler Summary

Table of Contents		Assembler Summary
>	Flow Summary	
	Flow Settings	
	Flow Non-Default Global Settings	
	Flow Elapsed Time	
	Flow OS Summary	
	Flow Log	
>	Analysis & Synthesis	
>	Filter	
>	Assembler	
	Summary	
	Settings	
	Generated Files	
	Device Options: E:/aca/SEM5_ACA/1_EN	
	Messages	
>	Timing Analyzer	
>	EDA Netlist Writer	
	Flow Messages	
	Flow Suppressed Messages	

Figure 7.4: Assembler Summary

## 7.5 Timing Analyzer Summary

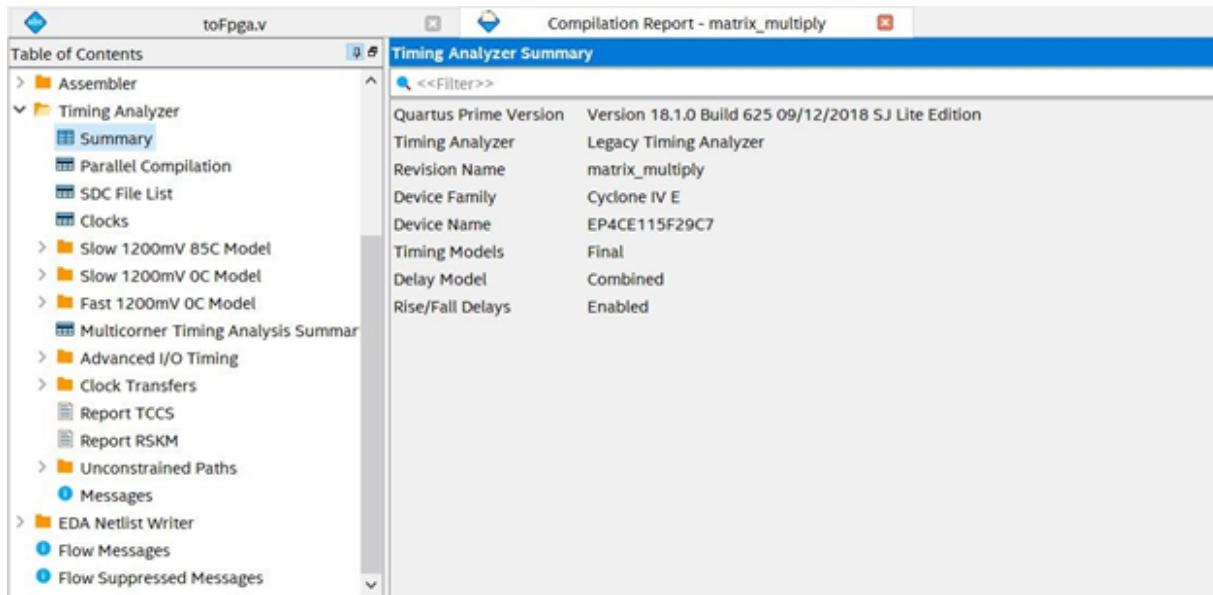


Figure 7.5: Timing Analyzer Summary

## 7.6 EDA Netlist Writer Summary

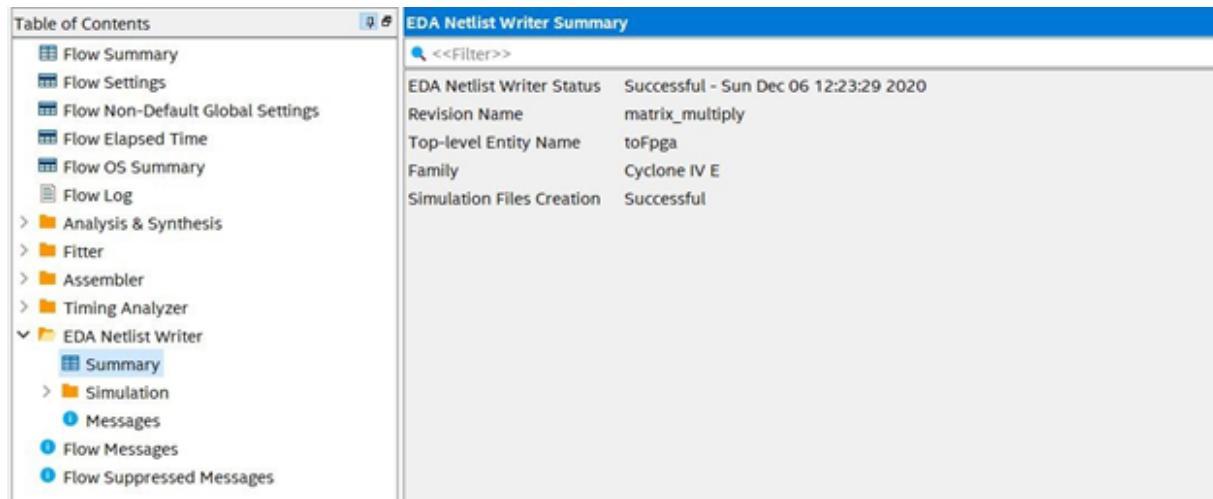


Figure 7.6: EDA Netlist Writer Summary

# 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](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

```
1 package details;
2 localparam IR_WIDTH = 8;
3
4 typedef enum logic [2:0]{
5     clr_alu = 3'd0,      //alu operations
6     pass_alu = 3'd1,
7     add_alu = 3'd2,
8     sub_alu = 3'd3,
9     mul_alu = 3'd4,
10    inc_alu = 3'd5,
11    idle_alu = 3'd6
12 } alu_op_t;
13
14 typedef enum logic [3:0]{
15     DMem_bus = 4'b0,    //multiplexer
16     R_bus = 4'd1,
17     IR_bus = 4'd2,
18     RL_bus = 4'd3,
19     RC_bus = 4'd4,
20     RP_bus = 4'd5,
21     RQ_bus = 4'd6,
22     R1_bus = 4'd7,
23     AC_bus = 4'd8,
24     idle_bus = 4'd9
25 } bus_in_sel_t;
26
27 typedef enum logic [IR_WIDTH-1:0]{
28     NOP          = 8'd0,
29     ENDOP        = 8'd1,
30     CLAC         = 8'd2,
31     LDIAC        = 8'd3,
32     LDAC         = 8'd4,
33     STR          = 8'd5,
34     STIR         = 8'd6,
35     JUMP         = 8'd7,
36     JMPNZ        = 8'd8,
37     JMPZ         = 8'd9,
38     MUL          = 8'd10,
39     ADD          = 8'd11,
40     SUB          = 8'd12,
41     INCAC        = 8'd13,
42     MV_RL_AC    = {4'd1,4'd15},
43     MV_RP_AC    = {4'd2,4'd15},
44     MV_RQ_AC    = {4'd3,4'd15},
45     MV_RC_AC    = {4'd4,4'd15},
46     MV_R_AC     = {4'd5,4'd15},
47     MV_R1_AC    = {4'd6,4'd15},
48     MV_AC_RP    = {4'd7,4'd15},
49     MV_AC_RQ    = {4'd8,4'd15},
50     MV_AC_RL    = {4'd9,4'd15}
51 } ISA_t;
52
53 typedef enum logic [3:0] {
54     no_inc = 4'b0000,
55     PC_inc = 4'b1000,
56     RC_inc = 4'b0100,
57     RP_inc = 4'b0010,
58     RQ_inc = 4'b0001,
59     RC_RP_RQ_inc = 4'b0111
60 } inc_reg_t;
61
62 typedef enum logic [9:0] {
63     no_wrEn = 10'b0000000000,
```

```

64     AR_wrEn = 10'b1000000000,
65     R_wrEn = 10'b0100000000,
66     PC_wrEn = 10'b0010000000,
67     IR_wrEn = 10'b0001000000,
68     RL_wrEn = 10'b0000100000,
69     RC_wrEn = 10'b0000010000,
70     RP_wrEn = 10'b0000001000,
71     RQ_wrEn = 10'b0000000100,
72     R1_wrEn = 10'b0000000010,
73     AC_wrEn = 10'b0000000001
74 } wrEnReg_t;
75
76
77 endpackage

```

### 9.1.2 Top module - toFpga.sv

```

1 module toFpga (
2     input logic CLOCK_50,
3     input logic [1:0] KEY,
4     output logic [1:0] LEDG,
5     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7, // for checking perposes
6     input logic UART_RXD,
7     output logic UART_TXD
8 );
9
10 localparam CORE_COUNT = 4;
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 localparam BAUD_RATE = 115200;
20 localparam UART_WIDTH = 8;
21
22 ////////////// logic related to data memory //////////////
23 logic [DATA_MEM_WIDTH-1:0] DataMemOut, DataMemIn, processor_DataOut, uart_DataOut;
24 logic [DATA_MEM_ADDR_WIDTH-1:0] processor_dataMemAddr;
25 logic [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr, uart_dataMemAddr;
26
27 ////////////// logic related to instruction memory //////////////
28 logic [INS_WIDTH-1:0] InsMemOut, InsMemIn;
29 logic [INS_MEM_ADDR_WIDTH-1:0] processor_InsMemAddr;
30 logic [INS_MEM_ADDR_WIDTH-1:0] insMemAddr, uart_InsMemAddr;
31
32 ////////////// other logics ///////////////
33 logic dataMemWrEn, processor_DataMemWrEn, uart_dataMemWrEn;
34 logic uart_InsMemWrEn;
35 logic clk, rstN, startN, processStart;
36 logic processDone, processor_ready;
37
38 ////////////// logic related to communication unit ///////////////
39 logic rxByteReady, txByteReady;
40 logic rx_new_byte_indicate, new_ins_byte_indicate, new_data_byte_indicate;
41 logic txByteStart;
42
43 logic [7:0] byteForTx, byteFromRx;
44
45 logic uart_DataMem_transmitted, uart_DataMem_received, uart_InsMem_received;
46 logic uart_dmem_start_transmitN;
47 //////////////// state change logic
48 assign LEDG[1] = processDone;
49 assign LEDG[0] = processor_ready;
50
51
52
53 //////////////// state change logic
54
55 typedef enum logic [2:0] {
56     idle = 3'd0,
57     uart_receive_Imem = 3'd1,
58     uart_receive_dmem = 3'd2,
59     process_ready = 3'd3,
60     process_execute = 3'd4,

```

```

61     uart_transmit_dmem = 3'd5,
62     finish = 3'd6
63 } state_t;
64
65 state_t currentState, nextState;
66
67 always @(posedge clk) begin
68     if (~rstN) begin
69         currentState <= idle;
70     end
71     else begin
72         currentState <= nextState;
73     end
74 end
75
76 always_comb begin
77     nextState = currentState;
78
79     case (currentState)
80         idle: begin // start state
81             if (~startN) begin
82                 nextState = uart_receive_Imem;
83             end
84         end
85
86         uart_receive_Imem: begin // send the instructions (machine_code) from PC through UART
87             if (uart_InsMem_received) begin
88                 nextState = uart_receive_dmem;
89             end
90         end
91
92         uart_receive_dmem: begin //send the data memory values from PC through UART
93             if (uart_DataMem_received) begin
94                 nextState = process_exicute;
95             end
96         end
97
98         process_exicute: begin // processor exicute program (matrix multiplication)
99             if (processDone) begin
100                 nextState = uart_transmit_dmem;
101             end
102         end
103
104         uart_transmit_dmem: begin // send the answer matrix to PC through UART
105             if (uart_DataMem_transmitted) begin
106                 nextState = finish;
107             end
108         end
109
110         finish: begin //End of the process
111     end
112
113     default : nextState = idle;
114
115     endcase
116 end
117
118 assign clk = CLOCK_50;
119 assign rstN = KEY[0];
120 assign startN = KEY[1];
121 assign processStart = ((currentState == uart_receive_dmem) && (uart_DataMem_received))? 1'b1: 1'b0;
122 assign uart_dmem_start_transmitN = ((currentState == process_exicute) && (processDone))? 1'b0: 1'b1;
123
124 assign dataMemWrEn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem))?
125     uart_dataMemWrEn:
126         (currentState == process_exicute)? processor_DataMemWrEn:
127             1'b0;
128
129 assign dataMemAddr = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem))?
130     uart_dataMemAddr:
131         (currentState == process_exicute)? processor_dataMemAddr:
132             {DATA_MEM_ADDR_WIDTH{1'b0}};
133
134 assign DataMemIn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem))?
135     uart_DataOut:
136         (currentState == process_exicute)? processor_DataOut:
             {DATA_MEM_WIDTH{1'b0}};

```

```

137 assign insMemAddr = (currentState == uart_receive_Imem)? uart_InsMemAddr:
138     (currentState == process_execute)? processor_InsMemAddr:
139     {INS_MEM_ADDR_WIDTH{1'b0}};
140
141 assign new_ins_byte_indicate = (currentState == uart_receive_Imem)? rx_new_byte_indicate: 1'b0;
142 assign new_data_byte_indicate = (currentState == uart_receive_dmem)? rx_new_byte_indicate: 1'b0;
143
144 //////////////////////////////////////////////////////////////////
145 multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
146   .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
147   multi_core_processor(.clk,.rstN,.start(processStart), .ProcessorDataIn(DataMemOut),
148   .InsMemOut, .ProcessorDataOut(processor_DataOut), .insMemAddr(processor_InsMemAddr),
149   .dataMemAddr(processor_dataMemAddr), .DataMemWrEn(processor_DataMemWrEn),
150   .done(processDone), .ready(processor_ready));
151
152
153
154 RAM #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH)) IM(.clk(CLOCK_50), .wrEn(uart_InsMemWrEn),
155   .dataIn(InsMemIn), .addr(insMemAddr), .dataOut(InsMemOut));
156
157 RAM #(.WIDTH(DATA_MEM_WIDTH), .DEPTH(DATA_MEM_DEPTH)) DM(.clk(CLOCK_50), .wrEn(dataMemWrEn),
158   .dataIn(DataMemIn), .addr(dataMemAddr), .dataOut(DataMemOut));
159
160 ////////////// communication system
161
162 mem_communication_interface #(.MEM_WORD_LENGTH(DATA_MEM_WIDTH), .MEM_DEPTH(DATA_MEM_DEPTH), .UART_WIDTH(
163   UART_WIDTH))
164   dMem_com_interface(
165     .clk, .rstN, .txStartN(uart_dmem_start_transmitN),
166     .mem_transmitted(uart_DataMem_transmitted), .mem_received(uart_DataMem_received),
167     ////inputs outputs with memory
168     .dataFromMem(DataMemOut), .memWrEn(uart_dataMemWrEn), .mem_address(uart_dataMemAddr),
169     .dataToMem(uart_DataOut),
170     ////inputs outputs with uart system
171     .rxByteReady, .rx_new_byte_indicate(new_data_byte_indicate), .txByteReady,
172     .byteFromRx, .txByteStart, .byteForTx,
173     ////////////// select start end mem addresses of tx and rx
174     .tx_start_addr_in(uartMemory[1]), .tx_end_addr_in(uartMemory[2]),
175     .rx_end_addr_in(uartMemory[0]), .toggle_addr_range(1'b1)
176   );
177
178 mem_communication_interface #(.MEM_WORD_LENGTH(INS_WIDTH), .MEM_DEPTH(INS_MEM_DEPTH), .UART_WIDTH(UART_WIDTH))
179   insMem_com_interface(
180     .clk, .rstN, .txStartN(1'b1),
181     .mem_transmitted(), .mem_received(uart_InsMem_received),
182     ////inputs outputs with memory
183     .dataFromMem(), .memWrEn(uart_InsMemWrEn), .mem_address(uart_InsMemAddr), .dataToMem(
184       InsMemIn),
185     ////inputs outputs with uart system
186     .rxByteReady, .rx_new_byte_indicate(new_ins_byte_indicate), .txByteReady, .byteFromRx,
187     .txByteStart(), .byteForTx(),
188     ////////////// //select start end mem addresses of tx and rx
189     .tx_start_addr_in(), .tx_end_addr_in(), .rx_end_addr_in(),
190     .toggle_addr_range(1'b0)
191 );
192
193 uart_system #(.DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE)) uart_system (
194   .clk, .rstN,.txByteStart,.rx(UART_RXD), .byteForTx, .tx(UART_TXD), .tx_ready(txByteReady),
195   .rx_ready(rxByteReady), .rx_new_byte_indicate, .byteFromRx
196 );
197
198 ////////////// memory addesses identification to find end of receiving dataMem and answer matrix transmission
199 // start & end addresses //////////////
200 localparam Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
201   R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
202   R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);
203
204 logic [REG_WIDTH-1:0]uartMemory[2:0]; //0- end address of Q, 1- start address of R, 2- end address of R
205
206 always_ff @(posedge clk) begin
207   if (~rstN) begin
208     uartMemory <= '{default:'0};
209   end
210   else if (uart_dataMemWrEn & (currentState == uart_receive_dmem)) begin
211     if (uart_dataMemAddr == Q_end_addr_location)
212       uartMemory[0] <= uart_DataOut[REG_WIDTH-1:0];
213     else if (uart_dataMemAddr == R_start_addr_location)
214       uartMemory[1] <= uart_DataOut[REG_WIDTH-1:0];

```

```

212     else if (uart_dataMemAddr == R_end_addr_location)
213         uartMemory[2] <= uart_DataOut[REG_WIDTH-1:0];
214     end
215 end
216
217 ///////////////to count the time taken to the process
218 logic [25:0]currentTime;
219 timeCounter TC(.clk(CLOCK_50), .rstN(rstN), .start(processStart), .stop(processDone),
220                 .timeDuration(currentTime));
221
222 logic [6:0]hex_display_value[7:0];
223 assign '{HEX7,HEX6,HEX5, HEX4, HEX3,HEX2, HEX1, HEX0} = hex_display_value;
224
225 hex_display HD(.clk, .rstN, .state(currentState),
226                  .start_timeValue_convetion(~uart_DataMem_transmitted), .binary_time_value(currentTime),
227                  .hex_display_value
228 );
229
230 endmodule :toFpga

```

### 9.1.3 Memory

#### Module - RAM.sv

```

1 module RAM
2 #(
3     parameter WIDTH = 12,
4     parameter DEPTH = 256,
5     parameter ADDR_WIDTH = $clog2(DEPTH)
6 )
7 (
8     input logic clk, wrEn,
9     input logic [WIDTH-1:0]dataIn,
10    input logic [ADDR_WIDTH-1:0]addr,
11    output logic [WIDTH-1:0]dataOut
12 );
13
14 logic [ADDR_WIDTH-1:0]addr_reg;
15
16 logic [WIDTH-1:0]memory[0:DEPTH-1] ;
17
18 always_ff @(posedge clk) begin
19     addr_reg <= addr;
20     if (wrEn) begin
21         memory[addr] <= dataIn; // write requires only 1 clk cyle.
22     end
23 end
24 assign dataOut = memory[addr_reg]; // address is registered. Need 2 clk cycles to read.
25
26 endmodule : RAM

```

#### Testbench - RAM\_tb.sv

```

1 module RAM_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5 localparam CLK_PERIOD = 20;
6 logic clk;
7 initial begin
8     clk <= 0;
9     forever begin
10        #(CLK_PERIOD/2);
11        clk <= ~clk;
12    end
13 end
14
15 localparam WIDTH = 12;
16 localparam DEPTH = 8;
17 localparam ADDR_WIDTH = $clog2(DEPTH);
18
19 logic wrEn;
20 typedef logic [WIDTH-1:0]data_t;

```

```

21 data_t dataIn, dataOut;
22 typedef logic [ADDR_WIDTH-1:0] addr_t;
23 addr_t addr;
24
25 RAM #( .WIDTH(WIDTH), .DEPTH(DEPTH), .ADDR_WIDTH(ADDR_WIDTH)) dut(.*);
26
27 initial begin
28   @(posedge clk);
29   #(CLK_PERIOD*4/5);
30   wrEn <= 1'b1;
31   addr <= 3;
32   dataIn <= 100;
33
34   @(posedge clk);
35   #(CLK_PERIOD*4/5);
36   wrEn <= 1'b0;
37   addr <= 20;
38   dataIn <=30;
39
40   repeat (20) begin
41     @(posedge clk);
42     #(CLK_PERIOD*4/5); // to give data little bit earlier to the posedge of the clk
43
44     dataIn = $random();
45     addr = $random();
46     wrEn = $random();
47   end
48
49   $stop;
50 end
51
52 typedef logic [WIDTH-1:0] test_memory_t [0:DEPTH-1];
53 test_memory_t test_memory;
54
55 function test_memory_t update (test_memory_t test_memory,
56                                 logic wrEn, addr_t addr, data_t dataIn, data_t dataOut);
57
58   if (wrEn) begin
59     test_memory[addr] = dataIn;
60   end
61
62   return test_memory;
63
64 endfunction
65
66 function void check (test_memory_t test_memory, data_t dataOut, addr_t addr);
67
68   if (dataOut != test_memory[addr]) begin
69     $display("dataOut = %p, memory_value = %p, addr = %p", dataOut, test_memory[addr], addr);
70   end
71
72 endfunction
73
74 initial begin
75   forever begin
76     @(posedge clk);
77     test_memory = update (test_memory, wrEn, addr, dataIn, dataOut);
78
79     @(negedge clk);
80     #(CLK_PERIOD*1/10);
81     check (test_memory, dataOut, addr);
82   end
83 end
84
85 endmodule:RAM_tb

```

### 9.1.4 Multi-core processor

#### Module - multi\_core\_processor.sv

```

1 module multi_core_processor #(
2   parameter REG_WIDTH = 12,
3   parameter INS_WIDTH = 8,
4   parameter CORE_COUNT = 4,
5   parameter DATA_MEM_ADDR_WIDTH = 12,

```

```

6     parameter INS_MEM_ADDR_WIDTH = 8
7 )
8 (
9     input logic clk,rstN,start,
10    input logic [REG_WIDTH*CORE_COUNT-1:0] ProcessorDataIn ,
11    input logic [INS_WIDTH-1:0] InsMemOut ,
12    output logic [REG_WIDTH*CORE_COUNT-1:0] ProcessorDataOut ,
13    output logic [INS_MEM_ADDR_WIDTH-1:0] insMemAddr ,
14    output logic [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr ,
15    output logic DataMemWrEn , done,ready
16 );
17
18 logic core_DataMemWrEn [0:CORE_COUNT-1];
19 logic core_done [0:CORE_COUNT-1];
20 logic core_ready [0:CORE_COUNT-1];
21 logic [DATA_MEM_ADDR_WIDTH-1:0] core_dataMemAddr [0:CORE_COUNT-1];
22 logic [INS_MEM_ADDR_WIDTH-1:0] core_InsMemAddr [0:CORE_COUNT-1];
23
24 genvar i;
25 generate
26     for (i=0;i<CORE_COUNT; i=i+1) begin:core
27         processor #(.REG_WIDTH(REG_WIDTH) , .INS_WIDTH(INS_WIDTH) , .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH) ,
28                     .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH)) CPU
29             (.clk, .rstN, .start, .ProcessorDataIn(ProcessorDataIn[REG_WIDTH*i+:REG_WIDTH]),
30             .InsMemOut, .dataMemAddr(core_dataMemAddr[i]),
31             .ProcessorDataOut(ProcessorDataOut[REG_WIDTH*i+:REG_WIDTH]),
32             .insMemAddr(core_InsMemAddr[i]), .DataMemWrEn(core_DataMemWrEn[i]),
33             .done(core_done[i]), .ready(core_ready[i]) );
34     end
35 endgenerate
36
37 assign dataMemAddr = core_dataMemAddr[0];
38 assign DataMemWrEn = core_DataMemWrEn[0];
39 assign insMemAddr = core_InsMemAddr[0];
40 assign ready = core_ready[0];
41 assign done = core_done[0];
42
43 endmodule : multi_core_processor

```

## Testbench - multi\_core\_processor\_tb.sv

```

1 class RAM_class #(parameter WIDTH, DEPTH);
2
3 localparam ADDR_WIDTH = $clog2(DEPTH);
4
5 typedef logic [WIDTH-1:0] data_t;
6 typedef logic [ADDR_WIDTH-1:0] addr_t;
7
8 logic [WIDTH-1:0] RAM [0:DEPTH-1];
9 // logic [ADDR_WIDTH-1:0] addr;
10
11
12 function void initialize_full_memory(input logic [WIDTH-1:0] initial_vals[0:DEPTH-1]);
13     RAM = initial_vals;
14 endfunction
15
16 function void initialize_singal_memory_location(input data_t value, input addr_t addr);
17     RAM[addr] = value;
18 endfunction
19
20 task Read_memory(input addr_t addr, output data_t value, ref logic clk); // if needed read time delay
21     value = RAM[addr];
22 endtask
23
24 task Write_memory(input addr_t addr, input data_t data, input logic wrEn, ref logic clk); // if needed write
25     time delay can be added
26     @(posedge clk);
27     if (wrEn) begin
28         RAM[addr] = data;
29     end
30 endtask
31
32 function void display_RAM();
33     foreach(this.RAM[i])
34         $display(i,this.RAM[i]);
35 endfunction

```



```

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

```

```

192         break;
193     end
194     for (int z=0;z<c;z++) begin
195       logic [DATA_MEM_WIDTH-1:0] temp_1 = DMEM[(R_start_addr + x*c+z)];
196       logic [REG_WIDTH-1:0] temp_2 = temp_1[(y*REG_WIDTH-1) -:REG_WIDTH];
197       $write("%h ", temp_2);
198     end
199   end
200 end
201 endfunction
202
203
204 endmodule : multi_core_processor_tb

```

### 9.1.5 Single-core processor

#### Module - processor.sv

```

1 module processor import details::*;
2 #(
3   parameter REG_WIDTH = 12,
4   parameter INS_WIDTH = 8,
5   parameter DATA_MEM_ADDR_WIDTH = 12,
6   parameter INS_MEM_ADDR_WIDTH = 8
7 )
8 (
9   input  logic clk,rstN,start,
10  input  logic [REG_WIDTH-1:0]ProcessorDataIn ,
11  input  logic [INS_WIDTH-1:0]InsMemOut ,
12  output logic [REG_WIDTH-1:0]ProcessorDataOut ,
13  output logic [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr ,
14  output logic [INS_MEM_ADDR_WIDTH-1:0]insMemAddr ,
15  output logic DataMemWrEn ,
16  output logic done,ready
17 );
18
19 logic [REG_WIDTH-1:0]alu_a, alu_b, alu_out;
20 alu_op_t select_alu_op;
21 bus_in_sel_t busSel;
22 logic [REG_WIDTH-1:0]busOut;
23 logic [INS_WIDTH-1:0]IRout;
24 logic [REG_WIDTH-1:0] Rout, RLout, RCout, RPout, RQout, R1out, ACout;
25 logic Zout, ZWrEn;
26 inc_reg_t incReg; // {PC, RC, RP, RQ}
27 logic PC_inc, RC_inc, RP_inc, RQ_inc;
28 wrEnReg_t wrEnReg; // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
29 logic AR_wrEn,R_wrEn,PC_wrEn,IR_wrEn,RL_wrEn,RC_wrEn,RP_wrEn,RQ_wrEn,R1_wrEn,AC_wrEn;
30
31 assign {PC_inc, RC_inc, RP_inc, RQ_inc} = incReg;
32 assign {AR_wrEn,R_wrEn,PC_wrEn,IR_wrEn,RL_wrEn,RC_wrEn,RP_wrEn,RQ_wrEn,R1_wrEn,AC_wrEn} = wrEnReg;
33
34 controlUnit CU(.clk, .rstN, .start, .Zout, .instruction(ISAt'(IRout)), .aluOp(select_alu_op), .incReg ,
35 .wrEnReg, .busSel, .DataMemWrEn, .ZWrEn, .done, .ready);
36
37 alu #(WIDTH(REG_WIDTH)) alu(.a(alu_a), .b(alu_b), .selectOp(select_alu_op), .c(alu_out));
38
39 multiplexer #(WIDTH(REG_WIDTH), .IR_WIDTH(INS_WIDTH)) mux(.selectIn(busSel), .DMem(ProcessorDataIn),
40 .R(Rout), .RL(RLout), .RC(RCout), .RP(RPout), .RQ(RQout), .R1(R1out),
41 .AC(ACout), .IR(IRout), .busOut(busOut));
42
43 register #(WIDTH(REG_WIDTH)) AR(.dataIn(busOut), .wrEn(AR_wrEn), .rstN(rstN), .clk(clk),
44 .dataOut(dataMemAddr));
45
46 register #(WIDTH(REG_WIDTH)) R(.dataIn(busOut), .wrEn(R_wrEn), .rstN(rstN), .clk(clk), .dataOut(Rout));
47
48 incRegister #(WIDTH(INS_WIDTH)) PC(.dataIn(IRout), .wrEn(PC_wrEn), .rstN(rstN), .clk(clk), .incEn(PC_inc),
49 .dataOut(insMemAddr));
50
51 register #(WIDTH(INS_WIDTH)) IR(.dataIn(InsMemOut), .wrEn(IR_wrEn), .rstN(rstN), .clk(clk), .dataOut(IRout));
52
53 register #(WIDTH(REG_WIDTH)) RL(.dataIn(busOut), .wrEn(RL_wrEn), .rstN(rstN), .clk(clk), .dataOut(RLout));
54
55 incRegister #(WIDTH(REG_WIDTH)) RC(.dataIn(busOut), .wrEn(RC_wrEn), .rstN(rstN), .clk(clk), .incEn(RC_inc),

```

```

56         .dataOut(RCout));
57
58 incRegister #(.WIDTH(REG_WIDTH)) RP(.dataIn(busOut), .wrEn(RP_wrEn), .rstN(rstN), .clk(clk), .incEn(RP_inc),
59         .dataOut(RPout));
60
61 incRegister #(.WIDTH(REG_WIDTH)) RQ(.dataIn(busOut), .wrEn(RQ_wrEn), .rstN(rstN), .clk(clk), .incEn(RQ_inc),
62         .dataOut(RQout));
63
64 register #(.WIDTH(REG_WIDTH)) R1(.dataIn(busOut), .wrEn(R1_wrEn), .rstN(rstN), .clk(clk), .dataOut(R1out));
65
66 register #(.WIDTH(REG_WIDTH)) AC(.dataIn(alu_out), .wrEn(AC_wrEn), .rstN(rstN), .clk(clk), .dataOut(ACout));
67
68 zReg #(.WIDTH(REG_WIDTH)) Z(.dataIn(alu_out), .clk(clk), .rstN(rstN), .wrEn(ZWrEn), .Zout(Zout));
69
70 assign ProcessorDataOut = Rout;
71
72 assign alu_a = ACout;
73 assign alu_b = busOut;
74
75 endmodule:processor

```

## Testbench - processor\_tb.sv

```

1  class RAM_class #(parameter WIDTH, DEPTH);
2
3  localparam ADDR_WIDTH = $clog2(DEPTH);
4
5  typedef logic [WIDTH-1:0] data_t;
6  typedef logic [ADDR_WIDTH-1:0] addr_t;
7
8  logic [WIDTH-1:0] RAM [0:DEPTH-1];
9 // logic [ADDR_WIDTH-1:0] addr;
10
11
12 function void initialize_full_memory(input logic [WIDTH-1:0] initial_vals[0:DEPTH-1]);
13     RAM = initial_vals;
14 endfunction
15
16 function void initialize_singal_memory_location(input data_t value, input addr_t addr);
17     RAM[addr] = value;
18 endfunction
19
20 task Read_memory(input addr_t addr, output data_t value, ref logic clk);
21     // if needed read time delay can be added here
22     value = RAM[addr];
23 endtask
24
25 task Write_memory(input addr_t addr, input data_t data, input logic wrEn, ref logic clk);
26     // if needed write time delay can be added here
27     @(posedge clk);
28     if (wrEn) begin
29         RAM[addr] = data;
30     end
31 endtask
32
33 function void display_RAM();
34     foreach(this.RAM[i])
35         $display(i,this.RAM[i]);
36
37 endfunction
38
39 function data_t get_value(input addr_t addr);
40     return RAM[addr];
41 endfunction
42
43 endclass
44
45
46 module processor_tb import details::*;();
47
48 timeunit 1ns;
49 timeprecision 1ps;
50 localparam CLK_PERIOD = 20;
51 logic clk;
52 initial begin
53     clk <= 0;
54     forever begin
55         #(CLK_PERIOD/2);

```

```

56     clk <= ~clk;
57   end
58 end
59
60 localparam REG_WIDTH = 12;
61 localparam INS_WIDTH = 8;
62 localparam INS_MEM_DEPTH = 256;
63 localparam DATA_MEM_DEPTH = 4096;
64 localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
65 localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);
66
67 logic rstN,start;
68 logic [REG_WIDTH-1:0] ProcessorDataIn;
69 logic [INS_WIDTH-1:0] InsMemOut;
70 logic [REG_WIDTH-1:0] dataMemAddr,ProcessorDataOut;
71 logic [INS_WIDTH-1:0] insMemAddr;
72 logic DataMemWrEn;
73 logic done,ready;
74
75 processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH),
76             .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH)) dut(.*);
77
78 //////////////////////////////////////////////////////////////////
79 RAM_class #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH)) ins_mem = new;
80 RAM_class #(.WIDTH(REG_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 [REG_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
93 initial begin
94   @(posedge clk);
95   rstN <= 1'b0;
96   start <= 1'b0;
97   @(posedge clk);
98   rstN <= 1'b1;
99   start <= 1'b1;
100 end
101
102 always_ff @(posedge clk) begin
103   ins_mem.Read_memory(.addr(insMemAddr), .value(InsMemOut), .clk(clk));
104 end
105
106 always_ff @(posedge clk) begin
107   data_mem.Read_memory(.addr(dataMemAddr), .value(ProcessorDataIn), .clk(clk));
108 end
109
110 always_ff @(posedge clk) begin
111   data_mem.Write_memory(.addr(dataMemAddr), .data(ProcessorDataOut), .wrEn(DataMemWrEn), .clk(clk));
112 end
113
114
115 ////////////////////////////////////////////////////////////////// verification of the simulation correctness //////////////////////////////////////////////////////////////////
116
117 localparam Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
118       R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
119       R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);
120 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;
121
122 logic [REG_WIDTH-1:0] temp_data_mem_2[0:DATA_MEM_DEPTH-1];
123
124 always_ff @(posedge clk) begin
125   if (done) begin
126     a = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd0));
127     b = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd1));
128     c = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd2));
129     P_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd3));
130     Q_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd4));
131     R_start_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd5));
132     P_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd6));
133     Q_end_addr = data_mem.get_value(DATA_MEM_ADDR_WIDTH'(12'd7));
134     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
139     //matrix to a file
140
141     $display("\nMatrix P\n");
142     print_matrix_P(data_mem.RAM, a, b, P_start_addr, P_end_addr);
143
144     $display("\nMatrix Q\n");
145     print_matrix_Q(data_mem.RAM, b, c, Q_start_addr, Q_end_addr);
146
147     $display("\nMatrix R\n");
148     print_matrix_R(data_mem.RAM, a, c, R_start_addr, R_end_addr);
149
150     $stop;
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    NOP1  = 6'd1,
19
20    ENDOP1 = 6'd2,
21
22    CLAC1  = 6'd3,
23
24    FETCH_DELAY1 = 6'd37,           // //// (extra_delay for memory read)
25    FETCH1  = 6'd4,
26    FETCH2  = 6'd5,
27
28    LDIAC_DELAY1 = 6'd38,           // //// (extra_delay for memory read)
29    LDIAC1  = 6'd6,
30    LDIAC2  = 6'd7,
31    LDIAC_DELAY2 = 6'd39,           // //// (extra_delay for memory read)
32    LDIAC3  = 6'd8,
33
34    LDAC1  = 6'd9,
35    LDAC_DELAY1 = 6'd40,           // //// (extra_delay for memory read)
36    LDAC2  = 6'd10,
37
38    STIR_DELAY1 = 6'd41,           // //// (extra_delay for memory read)
39    STIR1  = 6'd11,
40    STIR2  = 6'd12,
41    STIR_DELAY2 = 6'd42,           // //// (extra_delay for memory write)
42    STIR3  = 6'd13,
43
44    STR1  = 6'd14,
45    STR_DELAY1 = 6'd43,           // //// (extra_delay for memory write)
46    STR2  = 6'd15,
47
48    JUMP_DELAY1 = 6'd44,           // //// (extra_delay for memory read)
49    JUMP1  = 6'd16,
50    JUMP2  = 6'd17,
51
52    JMPNZY_DELAY1 = 6'd45,           // //// (extra_delay for memory read)
53    JMPNZY1  = 6'd18,
54    JMPNZY2  = 6'd19,
55    JMPZN1  = 6'd20,
56
57    JMPZY_DELAY1 = 6'd46,           // //// (extra_delay for memory read)
58    JMPZY1  = 6'd21,
59    JMPZY2  = 6'd22,
60    JMPZN1  = 6'd23,
61
62    MUL1  = 6'd24,
63
64    ADD1  = 6'd25,
65
66    SUB1  = 6'd26,
67
68    INCAC1 = 6'd27,
69
70    MV_RL_AC1 = 6'd28,
71
72    MV_RP_AC1 = 6'd29,
73
74    MV_RQ_AC1 = 6'd30,
75
76    MV_RC_AC1 = 6'd31,
77
78    MV_R_AC1 = 6'd32,
79
80    MV_R1_AC1 = 6'd33,
81
82    MV_AC_RP1 = 6'd34,
83
84    MV_AC_RQ1 = 6'd35,
85
86    MV_AC_RL1 = 6'd36
87 } states_t;
88
89 states_t currentState, nextState;
90
91 always_ff @(posedge clk) begin
92     if (~rstN) begin
93         currentState <= IDLE;
94     end
95     else begin
96

```

```

97     currentState <= nextState;
98   end
99 end
100
101 ///////////////next state calculation
102 always_comb begin
103   nextState = currentState;
104   unique case(currentState)
105     IDLE : begin
106       if (start)
107         nextState <= FETCH1;
108       else
109         nextState <= IDLE;
110     end
111
112   NOP1 : nextState <= FETCH1;
113
114   ENDOP1 : nextState <= ENDOP1;
115
116   CLAC1 : nextState <= FETCH1;
117
118   FETCH_DELAY1 : nextState <= FETCH1;           // extra_delay for memory read
119   FETCH1 : nextState <= FETCH2;
120   FETCH2 : begin
121     unique case(instruction)
122       NOP: nextState <= NOP1;
123       ENDOP: nextState <= ENDOP1;
124       CLAC: nextState <= CLAC1;
125       LDIAC: nextState <= LDIAC_DELAY1;
126       LDAC: nextState <= LDAC1;
127       STR: nextState <= STR1;
128       STIR: nextState <= STIR_DELAY1; // extra_delay for memory read
129       JUMP: nextState <= JUMP_DELAY1; // extra_delay for memory read
130       JMPNZ: nextState <= (Zout == 0)? JMPNZ_DELAY1 : JMPNZN1; // extra_delay for memory read
131       JMPZ: nextState <= (Zout == 1)? JMPZ_DELAY1 : JMPZN1; // extra_delay for memory read
132       MUL: nextState <= MUL1;
133       ADD: nextState <= ADD1;
134       SUB: nextState <= SUB1;
135       INCAC: nextState <= INCAC1;
136       MV_RL_AC: nextState <= MV_RL_AC1;
137       MV_RP_AC: nextState <= MV_RP_AC1;
138       MV_RQ_AC: nextState <= MV_RQ_AC1;
139       MV_RC_AC: nextState <= MV_RC_AC1;
140       MV_R_AC: nextState <= MV_R_AC1;
141       MV_R1_AC: nextState <= MV_R1_AC1;
142       MV_AC_RP: nextState <= MV_AC_RP1;
143       MV_AC_RQ: nextState <= MV_AC_RQ1;
144       MV_AC_RL: nextState <= MV_AC_RL1;
145       default: nextState <= IDLE;
146     endcase
147   end
148
149   LDIAC_DELAY1 : nextState <= LDIAC1;           // extra_delay for memory read
150   LDIAC1 : nextState <= LDIAC2;
151   LDIAC2 : nextState <= LDIAC_DELAY2;
152   LDIAC_DELAY2 : nextState <= LDIAC3;           // extra_delay for memory read
153   LDIAC3 : nextState <= FETCH_DELAY1;
154
155   LDAC1 : nextState <= LDAC_DELAY1;
156   LDAC_DELAY1 : nextState <= LDAC2;             // extra_delay for memory read
157   LDAC2 : nextState <= FETCH_DELAY1;
158
159   STIR_DELAY1 : nextState <= STIR1;             // extra_delay for memory read
160   STIR1 : nextState <= STIR2;
161   STIR2 : nextState <= STIR_DELAY2;
162   STIR_DELAY2 : nextState <= STIR3;             // extra_delay for memory write
163   STIR3 : nextState <= FETCH_DELAY1;
164
165   STR1 : nextState <= STR_DELAY1;
166   STR_DELAY1 : nextState <= STR2;               // extra_delay for memory write
167   STR2 : nextState <= FETCH1;
168
169   JUMP_DELAY1 : nextState <= JUMP1;             // extra_delay for memory read
170   JUMP1 : nextState <= JUMP2;
171   JUMP2 : nextState <= FETCH_DELAY1;
172
173   JMPNZ_DELAY1 : nextState <= JMPNZY1;           // extra_delay for memory read
174   JMPNZY1 : nextState <= JMPNZY2;
175   JMPNZY2 : nextState <= FETCH_DELAY1;

```

```

176     JMPZN1 : nextState <= FETCH_DELAY1;
177
178     JMPZY_DELAY1 : nextState <= JMPZY1;           // extra_delay for memory read
179     JMPZY1 : nextState <= JMPZY2;
180     JMPZY2 : nextState <= FETCH_DELAY1;
181     JMPZN1 : nextState <= FETCH_DELAY1;
182
183     MUL1 : nextState <= FETCH_DELAY1;
184
185     ADD1 : nextState <= FETCH_DELAY1;
186
187     SUB1 : nextState <= FETCH_DELAY1;
188
189     INCAC1 : nextState <= FETCH_DELAY1;
190
191     MV_RL_AC1 : nextState <= FETCH_DELAY1;
192
193     MV_RP_AC1 : nextState <= FETCH_DELAY1;
194
195     MV_RQ_AC1 : nextState <= FETCH_DELAY1;
196
197     MV_RC_AC1 : nextState <= FETCH_DELAY1;
198
199     MV_R_AC1 : nextState <= FETCH_DELAY1;
200
201     MV_R1_AC1 : nextState <= FETCH_DELAY1;
202
203     MV_AC_RP1 : nextState <= FETCH_DELAY1;
204
205     MV_AC_RQ1 : nextState <= FETCH_DELAY1;
206
207     MV_AC_RL1 : nextState <= FETCH_DELAY1;
208     default : nextState <= IDLE;
209   endcase
210 end
211
212 ///////////////logic within a state
213 always_comb begin
214   unique case(currentState)
215     IDLE: begin
216       aluOp <= idle_alu;
217       incReg <= no_inc;
218       wrEnReg <= no_wrEn;
219       busSel <= idle_bus;
220       DataMemWrEn <= 1'b0;
221       ZWrEn <= 1'b0;
222     end
223
224     NOP1: begin
225       aluOp <= idle_alu;
226       incReg <= no_inc;
227       wrEnReg <= no_wrEn;
228       busSel <= idle_bus;
229       DataMemWrEn <= 1'b0;
230       ZWrEn <= 1'b0;
231     end
232
233     ENDOP1: begin
234       aluOp <= idle_alu;
235       incReg <= no_inc;
236       wrEnReg <= no_wrEn;
237       busSel <= DMem_bus;
238       DataMemWrEn <= 1'b0;
239       ZWrEn <= 1'b0;
240     end
241
242     CLAC1: begin
243       aluOp <= clr_alu;
244       incReg <= no_inc;
245       wrEnReg <= AC_wrEn;
246       busSel <= idle_bus;
247       DataMemWrEn <= 1'b0;
248       ZWrEn <= 1'b1;
249     end
250
251     FETCH_DELAY1: begin
252       aluOp <= idle_alu;
253       incReg <= no_inc;
254       wrEnReg <= IR_wrEn;

```

```

255         busSel <= idle_bus;
256         DataMemWrEn <= 1'b0;
257         ZWrEn <= 1'b0;
258     end
259
260     FETCH1: begin
261         aluOp <= idle_alu;
262         incReg <= no_inc;
263         wrEnReg <= IR_wrEn;
264         busSel <= idle_bus;
265         DataMemWrEn <= 1'b0;
266         ZWrEn <= 1'b0;
267     end
268
269     FETCH2: begin
270         aluOp <= idle_alu;
271         incReg <= PC_inc;
272         wrEnReg <= no_wrEn;
273         busSel <= DMem_bus;
274         DataMemWrEn <= 1'b0;
275         ZWrEn <= 1'b0;
276     end
277
278     LDIACT_DELAY1: begin
279         aluOp <= idle_alu;
280         incReg <= no_inc;
281         wrEnReg <= IR_wrEn;
282         busSel <= idle_bus;
283         DataMemWrEn <= 1'b0;
284         ZWrEn <= 1'b0;
285     end
286
287     LDIACT1: begin
288         aluOp <= idle_alu;
289         incReg <= no_inc;
290         wrEnReg <= IR_wrEn;
291         busSel <= idle_bus;
292         DataMemWrEn <= 1'b0;
293         ZWrEn <= 1'b0;
294     end
295
296     LDIACT2: begin
297         aluOp <= idle_alu;
298         incReg <= PC_inc;
299         wrEnReg <= AR_wrEn;
300         busSel <= IR_bus;
301         DataMemWrEn <= 1'b0;
302         ZWrEn <= 1'b0;
303     end
304
305     LDIACT_DELAY2: begin
306         aluOp <= pass_alu;
307         incReg <= no_inc;
308         wrEnReg <= AC_wrEn;
309         busSel <= DMem_bus;
310         DataMemWrEn <= 1'b0;
311         ZWrEn <= 1'b1;
312     end
313
314     LDIACT3: begin
315         aluOp <= pass_alu;
316         incReg <= no_inc;
317         wrEnReg <= AC_wrEn;
318         busSel <= DMem_bus;
319         DataMemWrEn <= 1'b0;
320         ZWrEn <= 1'b1;
321     end
322
323     LDAC1: begin
324         aluOp <= idle_alu;
325         incReg <= no_inc;
326         wrEnReg <= AR_wrEn;
327         busSel <= AC_bus;
328         DataMemWrEn <= 1'b0;
329         ZWrEn <= 1'b0;
330     end
331
332     LDAC_DELAY1: begin
333         aluOp <= pass_alu;

```

```
334         incReg <= no_inc;
335         wrEnReg <= AC_wrEn;
336         busSel <= DMem_bus;
337         DataMemWrEn <= 1'b0;
338         ZWrEn <= 1'b1;
339     end
340
341     LDAC2: begin
342         aluOp <= pass_alu;
343         incReg <= no_inc;
344         wrEnReg <= AC_wrEn;
345         busSel <= DMem_bus;
346         DataMemWrEn <= 1'b0;
347         ZWrEn <= 1'b1;
348     end
349
350     STR1: begin
351         aluOp <= idle_alu;
352         incReg <= no_inc;
353         wrEnReg <= AR_wrEn;
354         busSel <= AC_bus;
355         DataMemWrEn <= 1'b0;
356         ZWrEn <= 1'b0;
357     end
358
359     STR_DELAY1: begin
360         aluOp <= idle_alu;
361         incReg <= no_inc;
362         wrEnReg <= no_wrEn;
363         busSel <= idle_bus;
364         DataMemWrEn <= 1'b1;
365         ZWrEn <= 1'b0;
366     end
367
368     STR2: begin
369         aluOp <= idle_alu;
370         incReg <= no_inc;
371         wrEnReg <= no_wrEn;
372         busSel <= idle_bus;
373         DataMemWrEn <= 1'b1;
374         ZWrEn <= 1'b0;
375     end
376
377     STIR_DELAY1: begin
378         aluOp <= idle_alu;
379         incReg <= no_inc;
380         wrEnReg <= IR_wrEn;
381         busSel <= idle_bus;
382         DataMemWrEn <= 1'b0;
383         ZWrEn <= 1'b0;
384     end
385
386     STIR1: begin
387         aluOp <= idle_alu;
388         incReg <= no_inc;
389         wrEnReg <= IR_wrEn;
390         busSel <= idle_bus;
391         DataMemWrEn <= 1'b0;
392         ZWrEn <= 1'b0;
393     end
394
395     STIR2: begin
396         aluOp <= idle_alu;
397         incReg <= PC_inc;
398         wrEnReg <= AR_wrEn;
399         busSel <= IR_bus;
400         DataMemWrEn <= 1'b0;
401         ZWrEn <= 1'b0;
402     end
403
404     STIR_DELAY2: begin
405         aluOp <= idle_alu;
406         incReg <= no_inc;
407         wrEnReg <= no_wrEn;
408         busSel <= idle_bus;
409         DataMemWrEn <= 1'b1;
410         ZWrEn <= 1'b0;
411     end
412
```

```

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_DELAY1: 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      JMPZN1: 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;
```

```

571         DataMemWrEn <= 1'b0;
572         ZWrEn <= 1'b0;
573     end
574
575     MV_RQ_AC1: begin
576         aluOp <= idle_alu;
577         incReg <= no_inc;
578         wrEnReg <= RQ_wrEn;
579         busSel <= AC_bus;
580         DataMemWrEn <= 1'b0;
581         ZWrEn <= 1'b0;
582     end
583
584     MV_RC_AC1: begin
585         aluOp <= idle_alu;
586         incReg <= no_inc;
587         wrEnReg <= RC_wrEn;
588         busSel <= AC_bus;
589         DataMemWrEn <= 1'b0;
590         ZWrEn <= 1'b0;
591     end
592
593     MV_R_AC1: begin
594         aluOp <= idle_alu;
595         incReg <= no_inc;
596         wrEnReg <= R_wrEn;
597         busSel <= AC_bus;
598         DataMemWrEn <= 1'b0;
599         ZWrEn <= 1'b0;
600     end
601
602     MV_R1_AC1: begin
603         aluOp <= idle_alu;
604         incReg <= no_inc;
605         wrEnReg <= R1_wrEn;
606         busSel <= AC_bus;
607         DataMemWrEn <= 1'b0;
608         ZWrEn <= 1'b0;
609     end
610
611     MV_AC_RP1: begin
612         aluOp <= pass_alu;
613         incReg <= no_inc;
614         wrEnReg <= AC_wrEn;
615         busSel <= RP_bus;
616         DataMemWrEn <= 1'b0;
617         ZWrEn <= 1'b1;
618     end
619
620     MV_AC_RQ1: begin
621         aluOp <= pass_alu;
622         incReg <= no_inc;
623         wrEnReg <= AC_wrEn;
624         busSel <= RQ_bus;
625         DataMemWrEn <= 1'b0;
626         ZWrEn <= 1'b1;
627     end
628
629     MV_AC_RL1: begin
630         aluOp <= pass_alu;
631         incReg <= no_inc;
632         wrEnReg <= AC_wrEn;
633         busSel <= RL_bus;
634         DataMemWrEn <= 1'b0;
635         ZWrEn <= 1'b1;
636     end
637
638     default : begin
639         aluOp <= idle_alu;
640         incReg <= no_inc;
641         wrEnReg <= no_wrEn;
642         busSel <= idle_bus;
643         DataMemWrEn <= 1'b0;
644         ZWrEn <= 1'b0;
645     end
646 endcase
647 end
648
649 assign done = (currentState == ENDOP1)?1'b1:1'b0;

```

```

650 assign ready = (currentState == IDLE)? 1'b1:1'b0;
651
652 endmodule :controlUnit

```

## Testbench - controlUnit\_tb.sv

```

1 module controlUnit_tb import details::*;();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5 localparam CLK_PERIOD = 10;
6 logic clk;
7 initial begin
8     clk <= 0;
9     forever begin
10         #(CLK_PERIOD/2);
11         clk <= ~clk;
12     end
13 end
14
15 localparam IR_WIDTH = 8;
16
17 logic rstN,start,Zout;
18 ISA_t instruction;
19 alu_op_t aluOp;
20 inc_reg_t incReg;    // {PC, RC, RP, RQ}
21 wrEnReg_t wrEnReg;  // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
22 bus_in_sel_t busSel;
23 logic DataMemWrEn,ZWrEn;
24 logic done,ready;
25
26
27 localparam // time duration for each instruction to execute
28     NOP_time_duration      = 4,
29     ENDOP_time_duration    = 4,
30     CLAC_time_duration    = 4,
31     LDIAIC_time_duration  = 8,
32     LDAC_time_duration    = 6,
33     STR_time_duration     = 6,
34     STIR_time_duration    = 8,
35     JUMP_time_duration    = 6,
36     JMPNZ_Y_time_duration = 6,
37     JMPNZ_N_time_duration = 4,
38     JMPZ_Y_time_duration  = 6,
39     JMPZ_N_time_duration  = 4,
40     MUL_time_duration     = 4,
41     ADD_time_duration     = 4,
42     SUB_time_duration     = 4,
43     INCAC_time_duration   = 4,
44     MV_RL_AC_time_duration= 4,
45     MV_RP_AC_time_duration= 4,
46     MV_RQ_AC_time_duration= 4,
47     MV_RC_AC_time_duration= 4,
48     MV_R_AC_time_duration = 4,
49     MV_R1_AC_time_duration= 4,
50     MV_AC_RP_time_duration= 4,
51     MV_AC_RQ_time_duration= 4,
52     MV_AC_RL_time_duration= 4;
53
54
55 controlUnit #(.IR_WIDTH(IR_WIDTH)) dut(.*);
56
57 task automatic test_instruction(
58     input int duration,
59     input ISA_t ins,
60     input logic Z_value,
61     ref ISA_t instruction,
62     ref logic Zout
63 );
64
65     @(posedge clk);
66     instruction = ins;
67     Zout = Z_value;
68
69     #(duration * CLK_PERIOD);
70     // repeat(duration)@(posedge clk);
71
72 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), .
84     Zout(Zout));
85
86     ////// test CLAC
87     test_instruction(.duration(CLAC_time_duration), .ins(CLAC), .Z_value(1'bX), .instruction(instruction), .
88     Zout(Zout));
89
90     ////// test LDIAC
91     test_instruction(.duration(LDIAC_time_duration), .ins(LDIAC), .Z_value(1'bX), .instruction(instruction), .
92     Zout(Zout));
93
94     ////// test LDAC
95     test_instruction(.duration(LDAC_time_duration), .ins(LDAC), .Z_value(1'bX), .instruction(instruction), .
96     Zout(Zout));
97
98     ////// test STR
99     test_instruction(.duration(STR_time_duration), .ins(STR), .Z_value(1'bX), .instruction(instruction), .
100    Zout(Zout));
101
102    ////// test STIR
103    test_instruction(.duration(STIR_time_duration), .ins(STIR), .Z_value(1'bX), .instruction(instruction), .
104    Zout(Zout));
105
106    ////// test JUMP
107    test_instruction(.duration(JUMP_time_duration), .ins(JUMP), .Z_value(1'bX), .instruction(instruction), .
108    Zout(Zout));
109
110    ////// test JMPNZ_Y
111    test_instruction(.duration(JMPNZ_Y_time_duration), .ins(JMPNZ), .Z_value(1'b0), .instruction(instruction),
112    .Zout(Zout));
113
114    ////// test JMPNZ_N
115    test_instruction(.duration(JMPNZ_N_time_duration), .ins(JMPNZ), .Z_value(1'b1), .instruction(instruction),
116    .Zout(Zout));
117
118    ////// test JMPZ_Y
119    test_instruction(.duration(JMPZ_Y_time_duration), .ins(JMPZ), .Z_value(1'b1), .instruction(instruction),
120    .Zout(Zout));
121
122    ////// test JMPZ_N
123    test_instruction(.duration(JMPZ_N_time_duration), .ins(JMPZ), .Z_value(1'b0), .instruction(instruction),
124    .Zout(Zout));
125
126    ////// test MUL
127    test_instruction(.duration(MUL_time_duration), .ins(MUL), .Z_value(1'b0), .instruction(instruction), .
128    Zout(Zout));
129
130    ////// test ADD
131    test_instruction(.duration(ADD_time_duration), .ins(ADD), .Z_value(1'b0), .instruction(instruction), .
132    Zout(Zout));
133
134    ////// test SUB
135    test_instruction(.duration(SUB_time_duration), .ins(SUB), .Z_value(1'b0), .instruction(instruction), .
136    Zout(Zout));
137
138    ////// test INCAC
139    test_instruction(.duration(INCAC_time_duration), .ins(INCAC), .Z_value(1'b0), .instruction(instruction),
140    .Zout(Zout));
141
142    ////// test MV_RL_AC
143    test_instruction(.duration(MV_RL_AC_time_duration), .ins(MV_RL_AC), .Z_value(1'b0), .instruction(instruction),
144    .Zout(Zout));
145
146    ////// test MV_RP_AC
147    test_instruction(.duration(MV_RP_AC_time_duration), .ins(MV_RP_AC), .Z_value(1'b0), .instruction(instruction),
148    .Zout(Zout));
149
150    ////// test MV_RQ_AC
151    test_instruction(.duration(MV_RQ_AC_time_duration), .ins(MV_RQ_AC), .Z_value(1'b0), .instruction(instruction),
152    .Zout(Zout));
```

```

135     instruction), .Zout(Zout));
136
137     ////// test MV_RC_AC
138     test_instruction(.duration(MV_RC_AC_time_duration), .ins(MV_RC_AC), .Z_value(1'b0), .instruction(
139     instruction), .Zout(Zout));
140
141     ////// test MV_R_AC
142     test_instruction(.duration(MV_R_AC_time_duration), .ins(MV_R_AC), .Z_value(1'b0), .instruction(
143     instruction), .Zout(Zout));
144
145     ////// test MV_R1_AC
146     test_instruction(.duration(MV_R1_AC_time_duration), .ins(MV_R1_AC), .Z_value(1'b0), .instruction(
147     instruction), .Zout(Zout));
148
149     ////// test MV_AC_RP
150     test_instruction(.duration(MV_AC_RP_time_duration), .ins(MV_AC_RP), .Z_value(1'b0), .instruction(
151     instruction), .Zout(Zout));
152
153     ////// test MV_AC_RQ
154     test_instruction(.duration(MV_AC_RQ_time_duration), .ins(MV_AC_RQ), .Z_value(1'b0), .instruction(
155     instruction), .Zout(Zout));
156
157     ////// test MV_AC_RL
158     test_instruction(.duration(MV_AC_RL_time_duration), .ins(MV_AC_RL), .Z_value(1'b0), .instruction(
159     instruction), .Zout(Zout));
160
161     ////// test ENDOP
162     test_instruction(.duration(ENDOP_time_duration), .ins(ENDOP), .Z_value(1'bX), .instruction(instruction),
163     .Zout(Zout));
164
165   end
166
167 initial begin          // simulation stop condition
168   forever begin
169     @(posedge clk);
170     if (done) begin // identify the end of the simulation
171       #(5*CLK_PERIOD);
172       $stop;
173     end
174   end
175 end
176
177 endmodule : controlUnit_tb

```

### 9.1.7 Arithmetic and Logic Unit (ALU)

#### Module - alu.sv

```

1 module alu import details::*;
2 #(
3   parameter WIDTH = 12
4   (
5     input logic signed [WIDTH-1:0]a,b,
6     input alu_op_t selectOp,
7     output logic signed [WIDTH-1:0]c
8   );
9
10  always_comb begin
11    unique case(selectOp)
12      clr_alu : c = 'd0;
13      pass_alu: c = b;
14      add_alu : c = a+b;
15      sub_alu : c = a-b;
16      mul_alu : c = a*b;
17      inc_alu : c = a+WIDTH'(signed'(1));
18      default: c = '0;
19    endcase
20  end
21 endmodule:alu

```

#### Testbench - alu\_tb.sv

```

1 module alu_tb import details::*;()

```

```
2
3 timeunit 1ns;
4 timprecision 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

```

1 module incRegister
2 #(parameter WIDTH = 12)
3 (
4     input logic [WIDTH-1:0] dataIn,
5     input logic wrEn, rstN, clk, incEn,
6     output logic [WIDTH-1:0] dataOut
7 );
8
9 logic [WIDTH-1:0] value;
10
11 always_ff @(posedge clk) begin
12     if (~rstN) value <= '0;
13     else if (wrEn) value <= dataIn;
14     if (incEn) value <= value+ WIDTH'(1);
15 end
16
17 assign dataOut = value;
18
19 endmodule:incRegister

```

### Testbench - incRegister\_tb.sv

```

1 module incRegister_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5
6 localparam CLK_PERIOD = 10;
7 logic clk;
8 initial begin
9     clk <= 0;
10    forever begin
11        #(CLK_PERIOD/2);
12        clk <= ~clk;
13    end
14 end
15
16 localparam WIDTH = 12;
17
18 logic unsigned [WIDTH-1:0] dataIn, dataOut;
19 logic wrEn, incEn, rstN;
20
21 incRegister #(.WIDTH(WIDTH)) dut(.*);
22
23 initial begin
24     @(posedge clk);
25     #(CLK_PERIOD*4/5);
26     rstN <= 0;
27
28     @(posedge clk);
29     #(CLK_PERIOD*4/5);
30     dataIn <= 23;
31     wrEn <= 1;
32
33     @(posedge clk);
34     #(CLK_PERIOD*4/5);
35     dataIn <= 36;
36     wrEn <= 1;
37     rstN <= 1;
38
39     @(posedge clk);
40     #(CLK_PERIOD*4/5);
41     dataIn <= 15;
42     wrEn <= 0;
43     incEn <= 1;
44
45     repeat (10) begin
46         @(posedge clk);
47         #(CLK_PERIOD*4/5);
48         dataIn = $random();
49         wrEn = $random();

```

```

50      incEn = $random();
51      rstN = $random();
52
53      $stop;
54 end
55
56
57 initial begin
58   forever begin
59     @(posedge clk);
60     #(CLK_PERIOD*4/5);
61     $display("dataIn = %d      rstN = %b      wrEn = %b      incEn = %b      dataOut = %d", dataIn, rstN, wrEn,
62     incEn, dataOut);
63   end
64 end
65
66 endmodule:incRegister_tb

```

## 9.1.9 Register

### Module - register.sv

```

1 module register
2 #(parameter WIDTH = 12)
3 (
4   input logic [WIDTH-1:0] dataIn,
5   input logic clk,rstN,wrEn,
6   output logic [WIDTH-1:0] dataOut
7 );
8
9 logic [WIDTH-1:0] value;
10
11 always_ff @( posedge clk ) begin
12   if (~rstN) value <= '0;
13   else if (wrEn) value <= dataIn;
14 end
15
16 assign dataOut = value;
17
18 endmodule : register

```

### Testbench - register\_tb.sv

```

1 module register_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5 localparam CLK_PERIOD = 20;
6
7 logic clk;
8 initial begin
9   clk <= 0;
10  forever begin
11    #(CLK_PERIOD/2);
12    clk <= ~clk;
13  end
14 end
15
16 localparam WIDTH = 12;
17 logic [WIDTH-1:0] dataIn, dataOut;
18 logic rstN, wrEn;
19
20 register #(WIDTH(WIDTH)) dut(.*);
21
22 initial begin
23   @(posedge clk);
24   #(CLK_PERIOD*4/5);
25   rstN <= 0;
26
27   @(posedge clk);
28   #(CLK_PERIOD*4/5);
29   rstN <= 1;

```

```

30     dataIn <= 20;
31     wrEn <= 0;
32
33     @(posedge clk);
34     #(CLK_PERIOD*4/5);
35     dataIn <= 43;
36     wrEn <= 1;
37
38     repeat(10) begin
39         @(posedge clk);
40         #(CLK_PERIOD*4/5);
41         dataIn = $random();
42         wrEn = $random();
43         rstN = $random();
44     end
45
46     repeat(2) @(posedge clk);
47     $stop;
48 end
49
50 endmodule: register_tb

```

## 9.1.10 Multiplexer (System Bus)

### Module - multiplexer.sv

```

1 module multiplexer import details::*;
2 #(
3     parameter WIDTH = 12,
4     parameter IR_WIDTH = 8
5 )
6 (
7     input bus_in_sel_t selectIn,
8     input logic [WIDTH-1:0] DMem, R, RL, RC, RP, RQ, R1, AC,
9     input logic [IR_WIDTH-1:0] IR,
10    output logic [WIDTH-1:0] busOut
11 );
12
13 always_comb begin
14     unique case(selectIn)
15         DMem_bus: busOut = DMem;
16         R_bus: busOut = R;
17         IR_bus: busOut = WIDTH'(IR);
18         RL_bus: busOut = RL;
19         RC_bus: busOut = RC;
20         RP_bus: busOut = RP;
21         RQ_bus: busOut = RQ;
22         R1_bus: busOut = R1;
23         AC_bus: busOut = AC;
24         idle_bus: busOut = '0;
25         default: busOut = '0;
26     endcase
27 end
28
29 endmodule:multiplexer

```

### Testbench - multiplexer\_tb.sv

```

1 module multiplexer_tb import details::*;
2 ();
3
4 timeunit 1ns;
5 timeprecision 1ps;
6
7 localparam CLK_PERIOD = 10;
8 logic clk;
9 initial begin
10     clk = 0;
11     forever begin
12         #(CLK_PERIOD/2);
13         clk <= ~clk;
14     end
15 end

```

```

16
17 localparam DMem_sel = 4'b0,
18      R_sel = 4'd1,
19      IR_sel = 4'd2,
20      RL_sel = 4'd3,
21      RC_sel = 4'd4,
22      RP_sel = 4'd5,
23      RQ_sel = 4'd6,
24      R1_sel = 4'd7,
25      AC_sel = 4'd8,
26      idle = 4'd9;
27
28 localparam WIDTH = 12;
29 localparam IR_WIDTH = 8;
30
31 bus_in_sel_t selectIn;
32 logic [WIDTH-1:0] DMem, R, RL, RC, RP, RQ, R1, AC;
33 logic [IR_WIDTH-1:0] IR;
34 logic [WIDTH-1:0] busOut;
35
36 multiplexer #(.WIDTH(WIDTH), .IR_WIDTH(IR_WIDTH)) dut(.*);
37
38 assign DMem = 10;
39 assign R = 11;
40 assign RL = 12;
41 assign RC = 13;
42 assign RP = 14;
43 assign RQ = 15;
44 assign R1 = 16;
45 assign AC = 17;
46 assign IR = 18;
47
48
49 initial begin
50   for (int i = 0; i < 10 ;i++ ) begin
51     @(posedge clk);
52     selectIn <= bus_in_sel_t'(i);
53     $display("selectIn = %d busOut = %d", i, busOut);
54   end
55
56   repeat(10) begin
57     @(posedge clk);
58     void'(std::randomize(selectIn));
59     $display("selectIn = %d busOut = %d", selectIn, busOut);
60   end
61
62   $stop;
63 end
64
65 endmodule: multiplexer_tb

```

## 9.1.11 Z register

### Module - zReg.sv

```

1 module zReg
2 #(parameter WIDTH = 12)
3 (
4   input logic [WIDTH-1:0] dataIn,
5   input logic clk, rstN, wrEn,
6   output logic Zout
7 );
8
9 logic value;
10
11 always_ff @(posedge clk) begin
12   if (~rstN) value <= 1'b0;
13   else if (wrEn) begin
14     if (dataIn == 0) value <= 1'b1;
15     else value <= 1'b0;
16   end
17 end
18
19 assign Zout = value;
20

```

```
21 endmodule : zReg
```

### Testbench - zReg\_tb.sv

```

1 module zReg_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5 localparam CLK_PERIOD = 20;
6
7 logic clk;
8 initial begin
9     clk <= 0;
10    forever begin
11        #(CLK_PERIOD/2);
12        clk <= ~clk;
13    end
14 end
15
16 localparam WIDTH = 12;
17 logic [WIDTH-1:0] dataIn;
18 logic rstN, wrEn, Zout;
19
20 zReg #(WIDTH(WIDTH)) dut(.*);
21
22 initial begin
23     @(posedge clk);
24     rstN <= 0;
25
26     @(posedge clk);
27     rstN <= 1;
28     dataIn <= 0;
29     wrEn <= 0;
30
31     @(posedge clk);
32     dataIn <= 0;
33     wrEn <= 1;
34
35     @(posedge clk);
36     dataIn <= 4;
37     wrEn <= 1;
38
39     repeat(10) begin
40         @(posedge clk);
41         dataIn = $urandom();
42         wrEn = $urandom();
43         rstN = $urandom();
44     end
45
46     $stop;
47 end
48
49 endmodule: zReg_tb

```

## 9.2 UART communication related SystemVerilog modules

### 9.2.1 UART communication interface for the memory

#### Module - mem\_communication\_interface.sv

```

1 module mem_communication_interface
2 #(
3     parameter MEM_WORD_LENGTH = 12,
4     parameter MEM_DEPTH = 4096,
5     parameter MEM_ADDR_LENGTH = $clog2(MEM_DEPTH),
6     parameter UART_WIDTH = 8
7 )
8 (
9     ///////////////////input output with main program
10    input  logic clk,rstN,txStartN,

```

```

11     output logic mem_transmitted, mem_received,
12
13     ////////////////////inputs outputs with memory
14     input logic [MEM_WORD_LENGTH-1:0] dataFromMem,
15     output logic memWrEn,
16     output logic [MEM_ADDR_LENGTH-1:0] mem_address,
17     output logic [MEM_WORD_LENGTH-1:0] dataToMem,
18
19     ////////////////////inputs outputs with uart system
20     input logic rxByteReady, rx_new_byte_indicate, txByteReady,
21     input logic [UART_WIDTH-1:0] byteFromRx,
22     output logic txByteStart,
23     output logic [UART_WIDTH-1:0] byteForTx,
24
25     /////////// select start end mem addresses of tx and rx
26     input logic [MEM_ADDR_LENGTH-1:0] tx_start_addr_in, tx_end_addr_in, rx_end_addr_in,
27     input logic toggle_addr_range // 0 - go untill the last address of the memory, 1 - consider inputted
28     start, end addresses
29 );
30
31 localparam logic [MEM_ADDR_LENGTH-1:0] LAST_MEM_ADDR = '1; //all bits = 1
32
33 logic startTransmit, txReady, rxDone;
34 logic new_rx_data_indicate;
35
36 typedef enum logic [3:0] {
37     idle      = 4'b0,
38     transmit_0 = 4'd1,
39     transmit_1 = 4'd2,
40     transmit_2 = 4'd3,
41     transmit_3 = 4'd4,
42     receive_0  = 4'd5,
43     receive_1  = 4'd6,
44     receive_2  = 4'd7,
45     receive_1_1= 4'd8,
46     receive_1_2= 4'd9
47 } state_t;
48
49 state_t currentState, nextState;
50 logic [MEM_ADDR_LENGTH-1:0] currentAddress, nextAddress;
51 logic currentStartTransmit, nextStartTransmit;
52
53 logic [MEM_ADDR_LENGTH-1:0] tx_start_addr, tx_end_addr, rx_end_addr;
54
55 always_ff @(posedge clk) begin
56     if (~rstN) begin
57         currentState <= idle;
58         currentAddress <= '0;
59         currentStartTransmit <= 1'b0;
60     end
61     else begin
62         currentState <= nextState;
63         currentAddress <= nextAddress;
64         currentStartTransmit <= nextStartTransmit;
65     end
66 end
67
68 always_comb begin
69     nextState = currentState;
70     nextAddress = currentAddress;
71     nextStartTransmit = currentStartTransmit;
72
73     case (currentState)
74         idle: begin
75             nextAddress = '0;
76             nextStartTransmit = 1'b0; //to transmit this should be zero
77             if (new_rx_data_indicate)
78                 nextState = receive_0;
79             else if (~txStartN) begin
80                 nextState = transmit_0;
81                 nextAddress = tx_start_addr;
82             end
83         end
84
85         //transmission process starts here
86         transmit_0: begin // to give the address to memory
87             nextState = transmit_1;
88         end

```

```

89      transmit_1: begin      // extra delay for ip core memory
90          nextStartTransmit = 1'b1;
91          nextState = transmit_2;
92      end
93
94      transmit_2: begin      // start uart transmitter
95          nextStartTransmit = 1'b1;
96          nextState = transmit_3;
97      end
98
99      transmit_3: begin      // find end of the uart transmission
100         nextStartTransmit = 1'b0;
101         if (txReady == 1'b1) begin
102             if (currentAddress == tx_end_addr) begin
103                 nextState = idle;
104             end
105             else begin
106                 nextAddress = currentAddress + MEM_ADDR_LENGTH'(1'b1);
107                 nextState = transmit_0;
108             end
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
120     receive_0)
121         nextState = receive_1_1;
122     end
123
124     receive_1_1: begin
125         nextState = receive_1_2;
126     end
127
128     receive_1_2: begin
129         nextState = receive_2;
130     end
131
132     receive_2: begin          //to find the end of the receiving process
133         if (currentAddress == rx_end_addr) begin
134             nextState = idle;
135         end
136         else begin
137             nextAddress = currentAddress + MEM_ADDR_LENGTH'(1'b1);
138             nextState = receive_0;
139         end
140     end
141 endcase
142 end
143
144 assign startTransmit = 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 ==
151     tx_end_addr))? 1'b1: 1'b0;
152
153 assign tx_start_addr = (toggle_addr_range == 1'b0)? '0: tx_start_addr_in;
154 assign tx_end_addr = (toggle_addr_range == 1'b0)? LAST_MEM_ADDR: tx_end_addr_in;
155
156 assign rx_end_addr = (toggle_addr_range == 1'b0)? LAST_MEM_ADDR:
157     (rx_end_addr_in == 0)? {{MEM_ADDR_LENGTH-4{1'b0}}, {4{1'b1}}}: //give 31 as the end
158     rx address if input addr is small
159     rx_end_addr_in;
160
161 data_encoder_decoder #( .WORD_SIZE(MEM_WORD_LENGTH), .UART_WIDTH(UART_WIDTH)) data_encoder_decoder
162 (
163     .dataFromMem,
164     .clk, .rstN, .txStart(startTransmit),
165     .txReady, .rxDone,
166     .dataToMem,

```

```

165          /////////////////// uart ports
166          .new_rx_data_indicate,
167          .rxByteReady, .rx_new_byte_indicate, .txByteReady,
168          .byteFromRx,
169          .txByteStart,
170          .byteForTx
171      );
172
173 endmodule:mem_communication_interface

```

## Testbench - mem\_communication\_interface\_tb.sv

```

1  class Memory #(parameter DEPTH , WIDTH);
2
3     localparam ADDR_WIDTH = $clog2(DEPTH);
4     typedef logic [WIDTH-1:0] mem_t [0:DEPTH-1];
5     typedef logic [WIDTH-1:0] data_t;
6     typedef logic [ADDR_WIDTH-1:0] addr_t;
7
8     rand mem_t memory;
9     // addr_t addr;
10
11    function new();
12        void'(this.randomize(memory));
13    endfunction
14
15    function logic[WIDTH-1:0] getData(input addr_t addr);
16        return this.memory[addr];
17    endfunction
18
19    function void storeData(input addr_t addr, input data_t data);
20        this.memory[addr] = data;
21    endfunction
22
23    task Read_memory(input addr_t addr, output data_t value, ref logic clk);
24        // if needed read time delay can be added here
25        value = memory[addr];
26    endtask
27
28    task Write_memory(input addr_t addr, input data_t data, input logic wrEn, ref logic clk);
29        // if needed write time delay can be added
30        @(posedge clk);
31        if (wrEn) begin
32            memory[addr] = data;
33        end
34    endtask
35
36 endclass
37
38 module mem_communication_interface_tb ();
39
40     timeunit 1ns;
41     timeprecision 1ps;
42     localparam CLK_PERIOD = 20;
43     logic clk;
44     initial begin
45         clk = 0;
46         forever begin
47             #(CLK_PERIOD/2);
48             clk <= ~clk;
49         end
50     end
51     localparam MEM_WORD_LENGTH = 12;
52     localparam MEM_DEPTH = 64;
53     localparam MEM_ADDR_LENGTH = $clog2(MEM_DEPTH);
54     localparam BAUD_RATE = 115200;
55     localparam UART_WIDTH = 8;
56     localparam BAUD_TIME_PERIOD = 10**9 / BAUD_RATE;
57     localparam WORD_2_UART_COUNT = MEM_WORD_LENGTH/UART_WIDTH + ((MEM_WORD_LENGTH % UART_WIDTH == 0)? 0:1);
58
59     localparam [MEM_ADDR_LENGTH-1:0]
60         tx_start_addr_val = 0,
61         tx_end_addr_val = {MEM_ADDR_LENGTH{1'b1}},
62         rx_end_addr_val = {MEM_ADDR_LENGTH{1'b1}};
63     localparam bit toggle_addr_range_val = 1'b0;
64
65     /////////////////// input output with main program
66     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
84 addresses
85
86 ////////////// logic belongs to uart_system
87 logic tx,rx;
88
89 Memory #(.DEPTH(MEM_DEPTH), .WIDTH(MEM_WORD_LENGTH)) Memory_a;
90 initial begin
91     Memory_a = new();
92 end
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
98 uart_system #(.DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE)) uart_system(.clk, .rstN,.txByteStart,.rx,.
99 byteForTx,
100 .tx, .tx_ready(txByteReady), .rx_ready(rxByteReady), .rx_new_byte_indicate, .byteFromRx);
101
102 ///////////////memory read and write
103 always_ff @(posedge clk) begin
104     Memory_a.Read_memory(.addr(mem_address), .value(dataFromMem), .clk(clk));
105 end
106
107 always_ff @(posedge clk) begin
108     Memory_a.Write_memory(.addr(mem_address), .data(dataToMem), .wrEn(memWrEn), .clk(clk));
109 end
110
111 // test transmission
112 initial begin
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

```

```

143     @(posedge clk); //starting delimiter
144     rx <= 1'b0;
145     #(BAUD_TIME_PERIOD);
146     for (int i=0;i<UART_WIDTH;i++) begin: data //data
147         @(posedge clk);
148         void'(std::randomize(rx));
149         #(BAUD_TIME_PERIOD);
150     end
151     @(posedge clk); // end delimiter
152     rx <= 1'b1;
153     #(BAUD_TIME_PERIOD);
154 end
155 $display("dataToMem = %d", dataToMem);
156 end
157
158 $display("receiving is done %p",Memory_a.memory);
159
160 $stop; // end of the simulation
161 end
162
163 endmodule:mem_communication_interface_tb

```

## 9.2.2 Data word size handler between memory and UART system

### Module - data\_encoder\_decoder.sv

```

1 module data_encoder_decoder
2 #(
3     parameter WORD_SIZE = 12, // width of memory word
4     parameter UART_WIDTH = 8
5 )
6 (
7     input  logic [WORD_SIZE-1:0]dataFromMem,
8     input  logic clk,rstN,txStart,
9     output logic txReady,rxDone,
10    output logic [WORD_SIZE-1:0]dataToMem,
11    output logic new_rx_data_indicate,
12
13    /////////////////// uart ports
14    input  logic rxByteReady, rx_new_byte_indicate, txByteReady,
15    input  logic [UART_WIDTH-1:0] byteFromRx,
16    output logic txByteStart,
17    output logic [UART_WIDTH-1:0] byteForTx
18
19
20 );
21
22 localparam EXTRA = ((WORD_SIZE % UART_WIDTH) == 0)?0:1;
23 localparam COUNT = (WORD_SIZE/UART_WIDTH) + EXTRA;
24 localparam BUFFER_WIDTH = COUNT * UART_WIDTH;
25 localparam COUNTER_LENGTH = (COUNT == 1)? 1:$clog2(COUNT);
26
27 typedef enum logic [2:0]{
28     idle = 3'd0,
29     transmit_1,
30     transmit_2,
31     receive_0 ,
32     receive_1 ,
33     receive_2 ,
34     receive_3
35 }state_t;
36
37
38 state_t currentState, nextState;
39 logic [BUFFER_WIDTH-1:0]currentTxBuffer, nextTxBuffer;
40 logic [BUFFER_WIDTH-1:0]currentRxBuffer, nextRxBuffer;
41 logic [COUNTER_LENGTH-1:0]currentTxCount, nextTxCount;
42 logic [COUNTER_LENGTH-1:0]currentRxCount, nextRxCount;
43
44 always_ff @(posedge clk) begin
45     if (!rstN) begin
46         currentTxBuffer <= '0;
47         currentRxBuffer <= '0;
48         currentTxCount <= '0;
49         currentRxCount <= '0;

```

```

50     currentState <= idle;
51 end
52 else begin
53     currentTxBuffer <= nextTxBuffer;
54     currentRxBuffer <= nextRxBuffer;
55     currentTxCount <= nextTxCount;
56     currentRxCount <= nextRxCount;
57     currentState <= nextState;
58 end
59 end
60
61 always_comb begin
62     nextState = currentState;
63     nextTxCount = currentTxCount;
64     nextRxCount = currentRxCount;
65     nextTxBuffer = currentTxBuffer;
66     nextRxBuffer = currentRxBuffer;
67
68     case (currentState)
69     idle: begin
70         nextTxCount = '0;
71         nextRxCount = '0;
72         if (rx_new_byte_indicate) begin //receiver indicates a arrival of new byte
73             nextState = receive_1;
74             nextRxBuffer = '0;
75         end
76         else if (txStart) begin
77             nextState = transmit_1;
78             if ((WORD_SIZE % UART_WIDTH) == 0)
79                 nextTxBuffer = dataFromMem;
80             else
81                 nextTxBuffer = {'0,dataFromMem};
82         end
83     end
84
85     transmit_1: begin           //start byte transmission
86         nextState = transmit_2;
87     end
88
89
90     transmit_2: begin           // end of the byte transmission
91         if (txByteReady) begin
92             if (BUFFER_WIDTH == UART_WIDTH) begin
93                 nextState = idle;
94             end
95             else begin
96                 nextTxCount = currentTxCount + COUNTER_LENGTH'(1'b1);
97                 if (currentTxCount == COUNT-1)
98                     nextState = idle;
99                 else begin
100                     nextTxBuffer = currentTxBuffer >> UART_WIDTH;
101                     nextState = transmit_1;
102                 end
103             end
104         end
105     end
106
107     receive_0: begin           // to give a EXTRA time to make rxByteReady to become 1'b0
108         nextState = receive_1;
109     end
110
111     receive_1 : begin
112         if(rxByteReady) begin
113             if (BUFFER_WIDTH == UART_WIDTH) begin
114                 nextRxBuffer = byteFromRx;
115                 nextState = idle;
116             end
117             else begin
118                 nextRxCount = currentRxCount + COUNTER_LENGTH'(1'b1);
119                 nextRxBuffer = {byteFromRx,currentRxBuffer[BUFFER_WIDTH-1 -: (BUFFER_WIDTH - UART_WIDTH)}
120             ];
121             if (currentRxCount == (COUNT-1))
122                 nextState = idle;
123             else
124                 nextState = receive_2;
125             end
126         end
127     end

```

```

128     receive_2: begin
129         if(rx_new_byte_indicate) //receiver indicates a arrival of new byte
130             nextState = receive_3;
131     end
132
133     receive_3: begin           // to give a EXTRA time to make rxByteReady to become 1'b0
134         nextState = receive_1;
135     end
136
137     endcase
138 end
139
140
141 assign txByteStart = (currentState == transmit_1)? 1'b1: 1'b0;
142 assign txReady = (currentState == idle)? 1'b1 : 1'b0;
143 assign byteForTx = currentTxBuffer[UART_WIDTH-1:0];
144 assign rxDone = ((currentState == receive_1) && (rxByteReady == 1'b1) && (currentRxCount == COUNT-1 ))? 1'b1
145 : 1'b0;
146 assign dataToMem = currentRxBuffer[WORD_SIZE-1:0];
147 assign new_rx_data_indicate = ((currentState == idle) && (rx_new_byte_indicate))? 1'b1: 1'b0; //arrival of
148     new data set
149
150 endmodule //data_encoder_decoder

```

## Testbench - data\_encoder\_decoder\_tb.sv

```

1 module data_encoder_decoder_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 WORD_SIZE = 24;
17 localparam UART_WIDTH = 8;
18 localparam BAUD_RATE = 19200;
19 localparam BAUD_TIME_PERIOD = 10**9 / BAUD_RATE;
20 localparam WORD_2_UART_COUNT = WORD_SIZE/UART_WIDTH + ((WORD_SIZE % UART_WIDTH == 0)? 0:1);
21
22 ////////////// logic related to data_encoder_decoder
23 logic [WORD_SIZE-1:0] dataFromMem;
24 logic rstN,txStart;
25 logic txReady,rxDone;
26 logic [WORD_SIZE-1:0] dataToMem;
27 logic new_rx_data_indicate;
28
29 /////////////// logic related to uart_system
30 logic rxByteReady, rx_new_byte_indicate, txByteReady;
31 logic [UART_WIDTH-1:0] byteFromRx;
32 logic txByteStart;
33 logic [UART_WIDTH-1:0] byteForTx;
34 logic rx, tx;
35
36 logic tx_ready,rx_ready;
37
38 assign txByteReady = tx_ready;
39 assign rxByteReady = rx_ready;
40
41 data_encoder_decoder #(WORD_SIZE(WORD_SIZE), .UART_WIDTH(UART_WIDTH)) dut(.*);
42
43 uart_system #(DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE)) uart_system (.*);
44
45 initial begin
46
47     @(posedge clk);
48     rstN <= 1'b0;
49     rx = 1'b1;
50     txStart <= 1'b0;
51
52     @(posedge clk);

```

```

53     rstN <= 1'b1;
54
55     ///////////////transmission check
56     repeat(5) begin
57         @(posedge clk);
58         wait(txReady);
59         @(posedge clk);
60         dataFromMem = $random();
61         txStart <= 1'b1;
62
63         @(posedge clk);
64         txStart <= 1'b0;
65     end
66
67     /////////////// reset before receiver check begin
68     @(posedge clk);
69     wait(txReady); // wait until transmission is over
70     @(posedge clk);
71     rstN <= 1'b0;
72     @(posedge clk);
73     rstN <= 1'b1;
74
75     ///////////////receiver check
76     @(posedge clk);
77     repeat(5) begin
78         for (int j=0; j< WORD_2_UART_COUNT;j++) begin:rx_check
79             @(posedge clk); //starting delimiter
80             rx <= 1'b0;
81             #(BAUD_TIME_PERIOD);
82             for (int i=0;i<UART_WIDTH;i++) begin:data //data
83                 @(posedge clk);
84                 rx = $random();
85                 #(BAUD_TIME_PERIOD);
86             end
87             @(posedge clk); // end delimiter
88             rx <= 1'b1;
89             #(BAUD_TIME_PERIOD);
90
91         end
92     end
93     $stop;
94 end
95
96 endmodule: data_encoder_decoder_tb

```

### 9.2.3 UART system

#### Module - uart\_system.sv

```

1 module uart_system
2 #(
3     parameter DATA_WIDTH = 8,
4     parameter BAUD_RATE = 19200
5 )(
6     input  logic clk, rstN,txByteStart ,rx ,
7     input  logic [DATA_WIDTH-1:0]byteForTx ,
8     output logic tx,tx_ready ,rx_ready ,rx_new_byte_indicate ,
9     output logic [DATA_WIDTH-1:0]byteFromRx
10 );
11
12 logic baudTick;
13
14 uart_baudRateGen #(.BAUD_RATE(BAUD_RATE)) baudRateGen(.clk, .rstN, .baudTick);
15
16 uart_transmitter #(.DATA_WIDTH(DATA_WIDTH)) transmitter(
17     .dataIn(byteForTx), .clk, .baudTick, .rstN, .txStart(txByteStart), .tx, .tx_ready
18 );
19
20 uart_receiver #(.DATA_WIDTH(DATA_WIDTH)) receiver (
21     .rx, .clk, .rstN, .baudTick, .rx_ready, .dataOut(byteFromRx), .new_byte_indicate(
22     rx_new_byte_indicate
23 );
24 endmodule:uart_system

```

## Testbench - uart\_system\_tb.sv

```

1 module uart_system_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5 localparam CLK_PERIOD = 20;
6 logic clk;
7
8 initial begin
9   clk <= 0;
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,txByteStart,rx;
21 logic [DATA_WIDTH-1:0]byteForTx;
22 logic tx,tx_ready,rx_ready,rx_new_byte_indicate;
23 logic [DATA_WIDTH-1:0]byteFromRx;
24
25 uart_system #(.DATA_WIDTH(DATA_WIDTH), .BAUD_RATE(BAUD_RATE)) dut(.*);
26
27 ////////// initial reset
28 initial begin
29   @(posedge clk);
30   rstN <= 1'b0;
31   @(posedge clk);
32   rstN <= 1'b1;
33 end
34
35
36 ////////// Transmitter test
37 initial begin
38   #(CLK_PERIOD*2); // to initialize
39
40   repeat(10) begin
41     @(posedge clk);
42     wait(tx_ready);
43     @(posedge clk);
44     byteForTx = $urandom();
45     txByteStart = 1'b1;
46
47     @(posedge clk);
48     txByteStart = 1'b0;
49   end
50   @(posedge clk);
51   wait(tx_ready);
52   $stop;
53 end
54
55 ////////// Receiver test
56 initial begin
57   #(CLK_PERIOD*2); // to initialize
58
59   repeat(10) begin
60     @(posedge clk); //starting delimiter
61     rx <= 1'b0;
62     #(BAUD_TIME_PERIOD);
63     for (int i=0;i<DATA_WIDTH;i++) begin: data //data
64       @(posedge clk);
65       rx = $urandom();
66       #(BAUD_TIME_PERIOD);
67     end
68     @(posedge clk); // end delimiter
69     rx <= 1'b1;
70     #(BAUD_TIME_PERIOD);
71   end
72
73   $stop;
74 end
75 endmodule:uart_system_tb

```

## 9.2.4 UART baud rate generator

### Module - uart\_baudRateGen.sv

```

1 module uart_baudRateGen
2 #(
3     parameter BAUD_RATE = 19200
4 )
5 (
6     input logic clk,rstN,
7     output logic baudTick
8 );
9 localparam CLK_RATE = 50*(10**6);
10 localparam RESOLUTION = 16; // samples per 1 baud
11 localparam int MAX_COUNT = (CLK_RATE/BAUD_RATE/RESOLUTION); //round to smaller integer
12 localparam WIDTH = $clog2(MAX_COUNT);
13
14 logic [WIDTH-1:0] count;
15
16 always_ff @(posedge clk) begin
17     if (~rstN)
18         count <= '0;
19     else if (count < MAX_COUNT)
20         count <= count + WIDTH'(1'b1);
21     else
22         count <= '0;
23 end
24
25 assign baudTick = (count==MAX_COUNT)? 1'b1:1'b0;
26
27
28 endmodule:uart_baudRateGen

```

### Testbench - uart\_baudRateGen\_tb.sv

```

1 module uart_baudRateGen_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5
6 localparam CLK_PERIOD = 20;
7 logic clk;
8 initial begin
9     clk = 0;
10    forever begin
11        #(CLK_PERIOD/2);
12        clk <= ~clk;
13    end
14 end
15
16 localparam BAUD_RATE = 115200;
17 localparam CLK_RATE = 50*(10**6);
18 localparam RESOLUTION = 16; // samples per 1 baud
19 localparam MAX_COUNT = (CLK_RATE/BAUD_RATE/RESOLUTION);
20 initial begin
21     $display("max_count = %d", MAX_COUNT);
22 end
23
24 logic rstN,baudTick;
25 uart_baudRateGen #(.BAUD_RATE(BAUD_RATE)) dut (.*);
26
27 initial begin
28     @(posedge clk);
29     rstN <= 1'b0;
30
31     @(posedge clk);
32     rstN <= 1'b1;
33 end
34
35 int clk_count = 0;
36 initial begin
37     @(posedge clk);
38     repeat(MAX_COUNT *5) begin
39         @(posedge clk);
40         if (baudTick == 1'b1) begin

```

```

41         $display("clk_count = %d", clk_count);
42         clk_count = 0;
43     end
44     else begin
45         clk_count = clk_count + 1;
46     end
47 end
48 $stop;
49 end
50
51 endmodule:uart_baudRateGen_tb

```

## 9.2.5 UART transmitter

### Module - uart\_transmitter.sv

```

1 module uart_transmitter
2 #(
3     parameter DATA_WIDTH = 8
4 )
5 (
6     input logic [DATA_WIDTH-1:0] dataIn,
7     input logic clk, baudTick, rstN, txStart,
8     output logic tx, tx_ready
9 );
10
11 typedef enum logic [1:0]{
12     idle = 2'd0,
13     start = 2'd1,
14     data_transmit = 2'd2,
15     stop = 2'd3
16 } state_t;
17
18 localparam COUNTER_WIDTH = $clog2(DATA_WIDTH);
19
20 state_t currentState, nextState;
21 logic [DATA_WIDTH-1:0] currentData, nextData;
22 logic currentBit, nextBit;
23 logic [COUNTER_WIDTH-1:0] currentCount, nextCount;
24 logic [3:0] currentTick, nextTick;
25
26
27 always_ff @(posedge clk) begin
28     if (~rstN) begin
29         currentTick <= 4'b0;
30         currentCount <= '0;
31         currentState <= idle;
32         currentData <= '0;
33         currentBit <= 1'b1;
34     end
35     else begin
36         currentTick <= nextTick;
37         currentCount <= nextCount;
38         currentState <= nextState;
39         currentData <= nextData;
40         currentBit <= nextBit;
41     end
42 end
43
44 always_comb begin
45     nextState = currentState;
46     nextTick = currentTick;
47     nextCount = currentCount;
48     nextData = currentData;
49     nextBit = currentBit;
50
51     case (currentState)
52         idle: begin
53             if (txStart) begin
54                 nextTick = 4'd0;
55                 nextBit = 1'b0; //start of the data byte
56                 nextData = dataIn;
57                 nextState = start;
58             end
59         else

```

```

60         nextBit = 1'b1;
61     end
62
63     start: begin
64         if (baudTick) begin
65             nextTick = currentTick + 4'b1;
66             if (currentTick == 4'd15) begin
67                 nextCount = '0;
68                 nextBit = currentData[0];
69                 nextState = data_transmit;
70             end
71         end
72     end
73
74     data_transmit: begin
75         nextBit = currentData[0];
76         if (baudTick) begin
77             nextTick = currentTick + 4'b1;
78             if (currentTick == 4'd15) begin
79                 nextCount = currentCount + COUNTER_WIDTH'(1'b1);
80                 nextData = currentData >>1;
81                 if (currentCount == (DATA_WIDTH-1)) begin
82                     nextBit = 1'b1;
83                     nextState = stop;
84                 end
85             end
86         end
87     end
88
89     stop: begin
90         if (baudTick) begin
91             nextTick = currentTick + 4'b1;
92             if (currentTick == 4'd15)
93                 nextState = idle;
94         end
95     end
96
97 endcase
98 end
99
100 assign tx = currentBit;
101 assign tx_ready = (currentState == idle)? 1'b1:1'b0;
102
103 endmodule //uart_transmitter

```

## Testbench - uart\_transmitter\_tb.sv

```

1 module uart_transmitter_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5 localparam CLK_PERIOD = 10;
6 logic clk;
7 initial begin
8     clk <= 0;
9     forever begin
10         #(CLK_PERIOD/2);
11         clk <= ~clk;
12     end
13 end
14
15 localparam DATA_WIDTH = 8;
16 localparam BAUD_RATE = 19200;
17
18 logic rstN,baudTick;
19 logic [DATA_WIDTH-1:0]dataIn;
20 logic txStart;
21 logic tx,tx_ready;
22
23 uart_baudRateGen #(.BAUD_RATE(BAUD_RATE)) baudRateGen(.*);
24 uart_transmitter #(.DATA_WIDTH(DATA_WIDTH)) transmitter(.*);
25
26 initial begin
27     @(posedge clk);
28     rstN <= 1'b0;
29     @(posedge clk);
30     rstN <= 1'b1;
31

```

```

32     repeat(10) begin
33         @(posedge clk);
34         wait(tx_ready);
35         @(posedge clk);
36         dataIn = $urandom();
37         txStart = 1'b1;
38
39         @(posedge clk);
40         txStart = 1'b0;
41     end
42     @(posedge clk);
43     wait(tx_ready);
44     $stop;
45 end
46
47 endmodule:uart_transmitter_tb

```

## 9.2.6 UART receiver

### Module - uart\_receiver.sv

```

1 module uart_receiver
2 #(
3     parameter DATA_WIDTH = 8
4 )
5 (
6     input  logic rx, clk, rstN, baudTick,
7     output logic rx_ready,
8     output logic [DATA_WIDTH-1:0]dataOut,
9     output logic new_byte_indicate
10 );
11
12 typedef enum logic [1:0]{
13     idle = 2'd0,
14     start = 2'd1,
15     data_receive = 2'd2,
16     stop = 2'd3
17 } state_t;
18
19 localparam COUNTER_WIDTH = $clog2(DATA_WIDTH);
20
21 state_t currentState, nextState;
22 logic [COUNTER_WIDTH-1:0]currentCount,nextCount;
23 logic [DATA_WIDTH-1:0]currentData, nextData;
24 logic [3:0]currentTick, nextTick;
25
26 always_ff @(posedge clk) begin
27     if(~rstN) begin
28         currentState <= idle;
29         currentCount <= '0;
30         currentTick <= 4'b0;
31         currentData <= '0;
32     end
33     else begin
34         currentState <= nextState;
35         currentCount <= nextCount;
36         currentTick <= nextTick;
37         currentData <= nextData;
38     end
39 end
40
41 always_comb begin
42     nextState = currentState;
43     nextCount = currentCount;
44     nextTick = currentTick;
45     nextData = currentData;
46
47     case (currentState)
48         idle: begin
49             nextTick = 4'b0;
50             nextCount = '0;
51             if (rx == 1'b0) begin
52                 nextState = start;
53             end
54         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,currentTime[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 = currentState;
99 assign rx_ready = (currentState == idle)? 1'b1: 1'b0;
100 assign new_byte_indicate = ((currentState == start) && (baudTick) && (currentTick == 4'd7) && (~rx))? 1'b1
101 :1'b0; //start of new data_receive byte
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;

```

```

37      5'ha:out = 7'b00001000;
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;
43      i   :out = 7'b1110000;
44      n   :out = 7'b0001011;
45      o   :out = 7'b1000000;
46      p   :out = 7'b0001100;
47      r   :out = 7'b1001110;
48      s   :out = 7'b0010010;
49      t   :out = 7'b0000111;
50      u   :out = 7'b1000001;
51      y   :out = 7'b0010001;
52      off :out = 7'b1111111;
53
54      default: out = 7'b1111111;
55  endcase
56 end
57
58 endmodule // SSeg

```

## Module - binaryToBCD.sv

```

1 module binaryToBCD(
2     input logic [25:0]binary_value, // to get 8 digit output need 26.57 input bits (26)
3     input logic clk,rstN,start,
4     output logic ready, done,
5     output logic [3:0]BCD_value[7:0]
6 );
7
8 typedef enum logic [1:0] {
9     idle,
10    subtract,
11    shift,
12    over
13 } state_t;
14
15 state_t currentState,nextState;
16
17 logic [4:0]currentCount,nextCount;           //count upto 26
18 logic [31:0]currentValue,nextValue;
19 logic [25:0]currentInput,nextInput;
20
21
22
23 always @(posedge clk) begin
24     if (!rstN) begin
25         currentCount <= 5'b0;
26         currentValue <= 32'b0;
27         currentState <= idle;
28         currentInput <= 26'b0;
29     end else begin
30         currentValue <= nextValue;
31         currentCount <= nextCount;
32         currentState <= nextState;
33         currentInput <= nextInput;
34     end
35 end
36
37 ////next state logic
38 always_comb begin
39     nextState = currentState;
40     case (currentState)
41         idle:
42             if (!start) nextState = subtract;
43
44         subtract:
45             nextState = shift;
46
47         shift:
48             if (currentCount<5'd25)
49                 nextState = subtract;
50             else
51                 nextState = over;
52
53         over:

```

```

54         nextState = idle;
55
56     endcase
57 end
58
59 always_comb begin
60     nextCount = currentCount;
61     nextValue = currentValue;
62     nextInput = currentInput;
63     BCD_value = '{default:'0};
64
65     unique case (currentState)
66         idle: begin
67             for (int j=0;j<8;j++) begin :digit_set
68                 BCD_value[j] = currentValue[4*j+:4];
69             end
70
71             if (!start) begin // KEY is push button stay at 1 normally
72                 nextCount = 5'b0;
73                 nextInput = binary_value;
74             nextValue = 32'b0;
75             end
76         end
77
78         subtract: begin
79             for (int i=0;i<32;i= i+4) begin: sub
80                 if (currentValue[i+:4] >4)
81                     nextValue[i+:4] = currentValue[i+:4] + 4'b0011;
82             end
83         end
84
85         shift: begin
86             nextInput[25:1] = currentInput[24:0];
87             nextValue = {currentValue[30:0],currentInput[25]};
88             nextCount = currentCount+5'b1;
89         end
90
91         over: begin
92             for (int k=0;k<8;k++) begin :digit_set_2
93                 BCD_value[k] = currentValue[4*k+:4];
94             end
95         end
96     endcase
97 end
98
99 assign done = (currentState != over); // actives the BCDtoHEX by the negedge
100 assign ready = (currentState == idle);
101
102 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

```

1 module simulation_top_tb();
2
3 timeunit 1ns;
4 timeprecision 1ps;
5 localparam CLK_PERIOD = 10;
6
7 logic clk;
8 initial begin
9     clk <= 0;
10    forever begin

```

```

11      #(CLK_PERIOD/2);
12      clk <= ~clk;
13  end
14 end
15
16 localparam CORE_COUNT = 1;
17
18 logic rstN, startN;
19 logic processor_ready, processDone;
20
21 simulation_top #(.CORE_COUNT(CORE_COUNT)) simulation_top(.*);
22
23 initial begin
24     @(posedge clk);
25     rstN = 1'b0;
26     startN = 1'b1;
27
28     @(posedge clk);
29     rstN = 1'b1;
30     startN = 1'b0;
31
32     @(posedge clk);
33     startN = 1'b1;
34
35     wait(processDone);
36
37     repeat(10) @(posedge clk);
38     $stop;
39 end
40
41 endmodule : simulation_top_tb

```

## Module - simulation\_top.sv

```

1 module simulation_top import details::*;
2 #(
3     parameter CORE_COUNT = 1
4 )
5 (
6     input logic clk, rstN, startN,
7     output logic 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 logic [DATA_MEM_WIDTH-1:0] DataMemOut, DataMemIn, processor_DataOut, uart_DataOut;
22 logic [DATA_MEM_ADDR_WIDTH-1:0] processor_dataMemAddr;
23 logic [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr, uart_dataMemAddr;
24
25 ////////////// logic related to instruction memory ///////////////
26 logic [INS_WIDTH-1:0] InsMemOut, InsMemIn;
27 logic [INS_MEM_ADDR_WIDTH-1:0] processor_InsMemAddr;
28 logic [INS_MEM_ADDR_WIDTH-1:0] insMemAddr, uart_InsMemAddr;
29
30 ////////////// other logics ///////////////
31 logic dataMemWrEn, processor_DataMemWrEn, uart_dataMemWrEn;
32 logic uart_InsMemWrEn;
33 logic processStart;
34
35
36
37 ///////////////// state change logic
38
39 typedef enum logic [2:0] {
40     idle = 3'd0,
41     process_execute = 3'd4,
42     finish = 3'd6
43 } state_t;
44

```

```

45 state_t currentState, nextState;
46
47 always @ (posedge clk) begin
48   if (~rstN) begin
49     currentState <= idle;
50   end
51   else begin
52     currentState <= nextState;
53   end
54 end
55
56 always_comb begin
57   nextState = currentState;
58
59   case (currentState)
60     idle: begin // start state
61       if (~startN) begin
62         nextState = process_execute;
63       end
64     end
65
66     process_execute: begin // processor execute program (matrix multiplication)
67       if (processDone) begin
68         nextState = finish;
69       end
70     end
71
72     finish: begin //End of the process
73   end
74
75   default : nextState = idle;
76
77 endcase
78 end
79
80 assign processStart = ((currentState == idle) && (~startN)) ? 1'b1: 1'b0;
81
82 assign dataMemWrEn = (currentState == process_execute) ? processor_DataMemWrEn : 1'b0;
83
84 assign dataMemAddr = (currentState == process_execute) ? processor_dataMemAddr: {DATA_MEM_ADDR_WIDTH{1'b0}};
85
86 assign DataMemIn = (currentState == process_execute) ? processor_DataOut: {DATA_MEM_WIDTH{1'b0}};
87
88 assign insMemAddr = (currentState == process_execute) ? processor_InsMemAddr: {INS_MEM_ADDR_WIDTH{1'b0}};
89
90 assign /////////////////////////////////
91 multi_core_processor #( .REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
92   .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
93   multi_core_processor
94   (.clk,.rstN,.start(processStart), .ProcessorDataIn(DataMemOut), .InsMemOut,
95   .ProcessorDataOut(processor_DataOut), .insMemAddr(processor_InsMemAddr),
96   .dataMemAddr(processor_dataMemAddr), .DataMemWrEn(processor_DataMemWrEn),
97   .done(processDone), .ready(processor_ready));
98
99
100
101 INS_RAM #( .WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH), .mem_init(yes))
102   IM(.clk(clk), .wrEn(uart_InsMemWrEn), .dataIn(InsMemIn), .addr(insMemAddr), .dataOut(InsMemOut));
103
104 DATA_RAM #( .WIDTH(DATA_MEM_WIDTH), .DEPTH(DATA_MEM_DEPTH), .mem_init(yes))
105   DM(.clk(clk), .wrEn(dataMemWrEn), .dataIn(DataMemIn), .addr(dataMemAddr), .dataOut(DataMemOut),
106   .processDone(processDone));
107
108
109
110 endmodule :simulation_top

```

## Module - INS\_RAM.sv

```

1 module INS_RAM import details::*;
2 #(
3   parameter mem_init_t mem_init = no,
4   parameter WIDTH = 8,
5   parameter DEPTH = 256,
6   parameter ADDR_WIDTH = $clog2(DEPTH)
7 )
8 (
9   input logic clk, wrEn,

```

```

10     input logic [WIDTH-1:0]dataIn ,
11     input logic [ADDR_WIDTH-1:0]addr ,
12     output logic [WIDTH-1:0]dataOut
13 );
14
15
16 logic [ADDR_WIDTH-1:0]addr_reg;
17
18 logic [WIDTH-1:0]memory[0:DEPTH-1] ;
19
20 /// initialize memory for simulation //////////////
21 initial begin
22   if (mem_init == yes) begin
23     $readmemb("../9_ins_mem_tb.txt", memory);
24   end
25 end
26 /////////////////////////////////
27
28 always_ff @(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.sv

```

1 module DATA_RAM import details::*;
2 #(
3   parameter mem_init_t mem_init = no,
4   parameter WIDTH = 12,
5   parameter DEPTH = 4096,
6   parameter ADDR_WIDTH = $clog2(DEPTH)
7 )
8 (
9   input logic clk, wrEn,
10  input logic [WIDTH-1:0]dataIn ,
11  input logic [ADDR_WIDTH-1:0]addr ,
12  output logic [WIDTH-1:0]dataOut ,
13
14  input logic processDone // need only for simulation (to get the memory at the end)
15 );
16
17
18 logic [ADDR_WIDTH-1:0]addr_reg;
19
20 logic [WIDTH-1:0]memory[0:DEPTH-1] ;
21
22 /// initialize memory for simulation //////////////
23 initial begin
24   if (mem_init == yes) begin
25     $readmemb("../4_data_mem_tb.txt", memory);
26   end
27 end
28
29 always_comb begin
30   if (processDone) begin
31     $writememb("../11_data_mem_out.txt", memory);
32   end
33 end
34
35 /////////////////////////////////
36
37 always_ff @(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

```

## 9.5 Python code for implementation and verification

### 9.5.1 Python code

```

1 import math
2 import time
3
4 from processor_matrix_multiplication_functions import *
5 baud_rate = 115200
6
7 # matrix A - a x b
8 # matrix B - b x c
9 a = int(input("give a: "))
10 b = int(input("give b: "))
11 c = int(input("give c: "))
12 core_count = int(input("give number of cores: "))
13 d = math.ceil(a/core_count) # (number of rows per one core)
14
15 create_random_matrices(a,b,c,core_count)
16
17 decoded_code = read_assembly_code()
18 send_instructions_memory_UART(decoded_code,baud_rate)
19
20 time.sleep(1) ## time gap between insMem, dataMem transmission
21
22 arrange_and_send_data_memory_to_FPGA(a,b,c,d,core_count,baud_rate)
23
24 receive_answer_matrix_from_FPGA(a,b,c,d,core_count,baud_rate)
25
26 calculate_answer_matrix_in_PC()
27
28 compare_answer_matrix_FPGA_and_PC()

```

### 9.5.2 Python functions

```

1 import random
2 import serial
3 import math
4 import struct
5 import time
6
7 def create_random_matrices(a,b,c,core_count):
8     writeFile = open('1_matrix_in.txt','w')
9
10    writeFile.write(str(a)+"\n"+str(b)+"\n"+str(c)+"\n"+str(core_count)+" //number of cores"\n")
11
12    writeFile.write("\n//matrix A\n")
13    out = ""
14    for x in range(a):
15        temp = ""
16        for y in range (b):
17            temp+=(str(random.randint(-5,5))+" ")
18        out+=(temp+"\n")
19
20    print (out)
21    writeFile.write(out)
22
23    writeFile.write("\n//matrix B\n")
24
25    out = ""
26    for x in range(b):
27        temp = ""
28        for y in range (c):
29            temp+=(str(random.randint(-5,5))+" ")
30        out+=(temp+"\n")
31
32    print (out)
33    writeFile.write(out)
34    writeFile.close()
35
36    return
37
38

```

```

39 def decode(instruction):
40     isa = {
41         'NOP': '00000000',
42         'ENDOP': '00000001',
43         'CLAC': '00000010',
44         'LDIAC': '00000011',
45         'LDAC': '00000100',
46         'STR': '00000101',
47         'STIR': '00000110',
48         'JUMP': '00000111',
49         'JMPNZ': '00001000',
50         'JMPZ': '00001001',
51         'MUL': '00001010',
52         'ADD': '00001011',
53         'SUB': '00001100',
54         'INCAC': '00001101',
55         'MV_RL_AC': '00011111',
56         'MV_RP_AC': '00101111',
57         'MV_RQ_AC': '00111111',
58         'MV_RC_AC': '01001111',
59         'MV_R_AC': '01011111',
60         'MV_R1_AC': '01101111',
61         'MV_AC_RP': '01111111',
62         'MV_AC_RQ': '10001111',
63         'MV_AC_RL': '10011111',
64     }
65     if instruction in isa:
66         return isa[instruction]
67     else:
68         return instruction
69
70
71 def read_assembly_code():
72
73     code = open('2_assembly_code.txt', 'r')
74     txt_code = code.read().strip().split('\n')
75
76     decoded_code = []
77     i = 0
78     while(i<len(txt_code)):
79         if txt_code[i][0]=='/':
80             i+=1
81             continue
82         codewithoutcomments = ''
83         for j in txt_code[i]:
84             if j!= '/':
85                 codewithoutcomments+=j
86             else:
87                 break
88         assert codewithoutcomments[-1]==';', 'Error! missing ; at the end of the line'
89         temp = codewithoutcomments[:-1]
90         decoded_int = decode(temp)
91         if(temp=='LDIAC' or temp=='STIR' or temp=='JMPNZ' or temp=='JUMP' or temp == 'JMPZ'):
92             decoded_code.append(decoded_int)
93             address = txt_code[i+1]
94             addresswithoutcomments = ''
95             for j in address:
96                 if j!= '/':
97                     addresswithoutcomments+=j
98                 else:
99                     break
100            assert addresswithoutcomments[-1]==';', 'Error! missing ; at the end of the line'
101            decoded_code.append('{0:08b}'.format(int(addresswithoutcomments[:-1])))
102            i+=1
103        else:
104            decoded_code.append(decoded_int)
105            i+=1
106
107     return decoded_code
108
109 def send_instructions_memory_UART(decoded_code, baud_rate):
110     uart_list = []
111     for i in range(255,-1,-1):
112         if (i<len(decoded_code)):
113             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])
114             uart_list.append(value)
115         else:

```

```

116         uart_list.append(0)
117
118     ser = serial.Serial('COM3',baud_rate,bytesize=8)
119
120     ser.write(uart_list[::-1])
121     ser.close()
122
123
124 def arrange_and_send_data_memory_to_FPGA(a,b,c,d,core_count,baud_rate):
125     data = open('1_matrix_in.txt','r')
126     txt_code = data.read().strip().split('\n')
127
128     out = b''
129
130     memWordLength = core_count*12
131     byte_count = math.ceil(memWordLength/8)
132     full_length = 8*byte_count
133     extra_length = full_length - memWordLength
134
135     start_addr_P = 14
136     start_addr_Q = start_addr_P + d*b
137     start_addr_R = start_addr_Q + b*c + 1 #with extra space
138     end_addr_P = start_addr_P + d*b - 1
139     end_addr_Q = start_addr_Q + b*c - 1
140     end_addr_R = start_addr_R + d*c - 1
141
142     setup_values = [a, b, c, start_addr_P, start_addr_Q, start_addr_R, end_addr_P, end_addr_Q, end_addr_R,
143     0, 0, 0, 0]
144
145     memLength = 4096
146     filled_memLength = len(setup_values) + (a//core_count + (a % core_count != 0))*b + b*c
147     empty_memLength = memLength-filled_memLength
148
149     ##### to send setup values #####
150
151     for i in setup_values:
152         # print(i)
153         temp = '{0:012b}'.format(i)*core_count
154         temp = extra_length*'0' + temp
155         for x in range(byte_count,0,-1):
156             temp2 = int(("0b" + temp[(x-1)*8:x*8]), 2)
157             out += struct.pack('!B', temp2)
158
159     ##### to send matrix A values
160
161     i = 4
162     while (txt_code[i] == ' ' or txt_code[i][0] == '/'): #to find the start of matrix A
163         i = i+1
164
165     A = []
166     for x in range (i,i+a):
167         temp2 = []
168         temp = txt_code[x].strip().split(" ")
169         for j in temp:
170             if (int(j)>=0):
171                 temp2.append('{0:012b}'.format(int(j)))
172             else:
173                 temp2.append(bin((int('1'*12,2)^abs(int(j)))+1)[2:]) ## 2's complement
174         A.append(temp2)
175
176     if len(A)%core_count!=0:
177         for k in range(core_count-len(A)%core_count):
178             temp = ['{0:012b}'.format(int(0)) for i in range(len(A[0]))]
179             A.append(temp)
180
181     for x in range(0,len(A),core_count):
182         for y in range(len(A[0])):
183             temporary_bin = ''
184             for k in range(core_count):
185                 temporary_bin+=A[x+k][y]
186
187             temporary_bin = extra_length*'0' + temporary_bin
188             for j in range(byte_count,0,-1):
189                 temp2 = int(("0b" + temporary_bin[(j-1)*8:j*8]), 2)
190                 out += struct.pack('!B', temp2)
191
192     ##### to send matrix B values #####
193

```

```

194
195     i = i + a
196     while (txt_code[i] == ' ' or txt_code[i][0] == '/'): #to find the start of matrix B
197         i = i+1
198
199     matrix_B = []
200     for x in range (i,i+b):
201         temp = txt_code[x].strip().split(" ")
202         matrix_B.append(temp)
203
204     for y in range(c):
205         for x in range(b):
206             temporary_bin = ''
207             for k in range(core_count):
208                 if (int(matrix_B[x][y]) < 0):
209                     temporary_bin+= bin((int('1'*12,2)^abs(int(matrix_B[x][y])))+1)[2:] ## 2's complement
210                 else:
211                     temporary_bin+='{0:012b}'.format(int(matrix_B[x][y]))
212
213             temporary_bin = extra_length*'0' + temporary_bin
214             for j in range(byte_count,0,-1):
215                 temp2 = int(("0b" + temporary_bin[(j-1)*8:j*8]), 2)
216                 out += struct.pack('!B', temp2)
217
218 ##### append 0s for last left words
219
220 ser = serial.Serial('COM3',baud_rate,bytesize=8)
221 ser.write(out)
222
223 def decodeCombinedValues(valueIn,no_of_cores):
224     if (no_of_cores == 1):
225         return [valueIn]
226
227     out = []
228     temp = bin(valueIn)[2:]
229     temp = format(int(temp), ('0'+str(12*no_of_cores)+'d'))
230     for x in range(no_of_cores):
231         out.append(int('0b' + temp[x*12:x*12+12]), 2)
232     return out
233
234 def convertToCorrectForm(inputMatrix, no_of_cores, a,c,d):
235
236     OutputMatrix = [[0 for x in range(c)] for y in range(a)]
237
238     for x in range(d):
239         for y in range(c):
240             for z in range(no_of_cores):
241                 if (x*no_of_cores+z) < a:
242                     val = inputMatrix[x*c+y][z]
243                     if (val > 2047):
244                         val = -4096 + val
245                         OutputMatrix[x*no_of_cores+z][y] = "--"+hex(val)[3:]
246                     else:
247                         OutputMatrix[x*no_of_cores+z][y] = hex(val)[2:]
248     return (OutputMatrix)
249
250 def receive_answer_matrix_from_FPGA(a,b,c,d,core_count,baud_rate):
251     memWordLength = core_count*12
252     byte_count = math.ceil(memWordLength/8)
253     full_length = 8*byte_count
254     extra_length = full_length - memWordLength
255
256     start_addr_P = 14
257     start_addr_Q = start_addr_P + d*b
258     start_addr_R = start_addr_Q + b*c + 1 #with extra space
259     end_addr_P = start_addr_P + d*b -1
260     end_addr_Q = start_addr_Q + b*c -1
261     end_addr_R = start_addr_R + d*c - 1
262
263     ser = serial.Serial('COM3',baud_rate,bytesize=8)
264
265 ##### receive dmemory after calculation
266
267     DMem = []
268     R_matrix_lenght = end_addr_R - start_addr_R + 1
269
270     for x in range(R_matrix_lenght):
271         temp = ''
272         for x in range (byte_count):
273             in_bin = ser.read()

```

```

273         temp = '{0:08b}'.format((int.from_bytes(in_bin,byteorder='little')) + temp
274 # print (hex(int("0b"+temp[extra_length:],2)))
275 temp = int("0b"+temp[extra_length:],2)
276 DMem.append(temp)
277
278 ##### find R matrix
279
280 matrix_initial = DMem
281 matrix_second = []
282
283 for x in matrix_initial:
284     matrix_second.append(decodeCombinedValues(x, core_count))
285
286 matrix_R = convertToCorrectForm(matrix_second, core_count, a,c,d)
287
288 writeFile = open('3_answer_matrix_FPGA.txt', 'w')
289
290 for x in (matrix_R):
291     for y in (x):
292         writeFile.write((6-len(y))*" " + y)
293     writeFile.write('\n')
294 writeFile.close()
295
296 def calculate_answer_matrix_in_PC():
297
298     data = open('1_matrix_in.txt','r')
299     txt_code = data.read().strip().split('\n')
300     writeFile = open('4_answer_matrix_PC.txt','w')
301     out = []
302
303     a = int(txt_code[0])
304     b = int(txt_code[1])
305     c = int(txt_code[2])
306
307     # matrix A - a x b
308     # matrix B - b x c
309     i = 4
310     while (txt_code[i] == '' or txt_code[i][0] == '/'): #to find the start of matrix A
311         i = i+1
312
313     A = []
314     for x in range (i,i+a):
315         temp = txt_code[x].strip().split(" ")
316         A.append(temp)
317     i = i + a
318     while (txt_code[i] == '' or txt_code[i][0] == '/'): #to find the start of matrix B
319     i = i+1
320
321     matrix_B = []
322     for x in range (i,i+b):
323         temp = txt_code[x].strip().split(" ")
324         matrix_B.append(temp)
325
326     mul = []
327
328     # begin_time = time.time()
329
330     for i in range(int(a)):
331         temp_list= []
332         for k in range(int(c)):
333             temp_ans = 0
334             for j in range(int(b)):
335                 temp_ans += int(A[i][j])*int(matrix_B[j][k])
336                 if (temp_ans<0):
337                     val = "-" + hex(temp_ans)[3:]
338                 else:
339                     val = hex(temp_ans)[2:]
340                 temp_list.append(" "*(5-len(val))+ val)
341         mul.append(temp_list)
342     out = ','
343     for i in mul:
344         temp = ''
345         for j in i:
346             temp+=(str(j)+', ')
347         out+=(temp+'\n ')
348     # print(out)
349     writeFile.write(out)
350     writeFile.close()
351

```

```

352     # end_time = time.time()
353     # print ((end_time - begin_time)*(2.5*(10**9)))
354
355 def compare_answer_matrix_FPGA_and_PC():
356
357     python_answer_file = open('4_answer_matrix_PC.txt', 'r')
358     fpga_answer_file = open('3_answer_matrix_FPGA.txt', 'r')
359
360     python_answer_matrix_temp = python_answer_file.read().strip().split('\n')
361     python_answer_matrix = []
362     for line in python_answer_matrix_temp:
363         python_answer_matrix.append(line.split())
364     python_answer_file.close()
365
366     fpga_answer_matrix_temp = fpga_answer_file.read().strip().split('\n')
367     fpga_answer_matrix = []
368     for line in fpga_answer_matrix_temp:
369         fpga_answer_matrix.append(line.split())
370     fpga_answer_file.close()
371
372     dim_a = len(python_answer_matrix)
373     dim_c = len(python_answer_matrix[0])
374
375     not_similar = 0
376     for x in range(dim_a):
377         for y in range(dim_c):
378             if (python_answer_matrix[x][y] != fpga_answer_matrix[x][y]):
379                 print ("[",x,",",",",y,"] fpga value = ",fpga_answer_matrix[x][y]," python value = ",python_answer_matrix[x][y])
380                 not_similar = 1
381
382     if (not_similar):
383         print ("wrong answer")
384     else:
385         print ("correct answer")

```

## 9.6 Text files used during synthesis and Verification

### 9.6.1 Assembly code

```

1 CLAC;
2 LDIAC;
3 1;//addresss of b
4 MV_RL_AC;
5 LDIAC;
6 3;//start address of P
7 MV_RP_AC;
8 LDIAC;
9 4;//start address of Q
10 MV_RQ_AC;
11 LDIAC;//set current P,Q,R
12 3;//start address of P
13 MV_R_AC;
14 STIR;
15 9;//current address of P
16 LDIAC;
17 4;//start address of Q
18 MV_R_AC;
19 STIR;
20 10;//current address of Q
21 LDIAC;
22 5;//start address of R
23 MV_R_AC;
24 STIR;
25 11;//current address of R
26 CLAC;
27 MV_R_AC;
28 MV_RC_AC;
29 MV_AC_RP;//loop starts
30 LDAC;
31 MV_R1_AC;
32 MV_AC_RQ;
33 LDAC;
34 MUL;

```

```

35 ADD;
36 MV_R_AC;
37 MV_AC_RL;
38 SUB;
39 JMPNZ;
40 28;//loop again
41 LDIAC;
42 11;//current address of R
43 STR;
44 INCAC;
45 MV_R_AC;
46 STIR;
47 11;//current address of R
48 MV_AC_RQ;
49 MV_RC_AC;
50 LDIAC;
51 7;// end address of Q
52 INCAC;//this is because current Q is increased by one
53 SUB;
54 JMPNZ;
55 57;// b != end
56 JUMP;
57 62;//b == end
58 LDIAC;//b!=end starts here
59 9;// current address of P
60 MV_RP_AC;
61 JUMP;
62 25;// jump to CLAC before loop begin
63 MV_AC_RP;//b==end starts here
64 MV_R_AC;//to store current P value
65 STIR;
66 9;// current address of P
67 MV_RC_AC;//to subtract
68 LDIAC;
69 6;//end address P
70 INCAC;
71 SUB;
72 JMPNZ;
73 75;// jump to a != end
74 JUMP;
75 80;// jump to a == end
76 LDIAC;//a!=end starts here
77 4;//start address Q
78 MV_RQ_AC;
79 JUMP;
80 25;// jump to CLAC before loop begin
81 ENDOP;//a==end starts here

```

## 9.6.2 Example Input random matrix file

```

1 10
2 12
3 4
4 5 //number of cores
5
6 //matrix A
7 0 4 4 0 -4 -2 3 -3 3 -3 -5 -2
8 5 5 -4 5 -3 -3 5 5 5 -1 5 5
9 4 4 5 5 0 4 -3 -4 -3 -2 1 4
10 4 -3 0 -2 1 2 -2 -2 0 -1 -4 -4
11 0 5 2 -2 2 4 -1 3 4 -5 5 5
12 -5 -3 -3 -2 4 -1 -4 -5 -1 5 0 -5
13 -3 -3 5 -3 4 2 -5 -5 -3 -5 -3 2
14 -4 -4 0 -2 5 1 2 -1 -1 1 0 3
15 4 -1 1 -5 5 1 3 -4 3 5 -3 2
16 0 2 -1 -3 5 -3 -4 -5 4 4 1 3
17
18 //matrix B
19 -4 -1 -1 5
20 -2 -5 -4 0
21 0 0 2 -5
22 -2 4 4 1
23 2 0 -1 2
24 -4 4 -5 4
25 -2 -2 1 -4
26 -4 0 2 3
27 -5 3 5 -3

```

```

28 0 2 -3 0
29 -4 4 3 -4
30 0 -5 3 1

```

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

```

1   3  -29     6   -30
2  -6d -18     4e   -3
3  -11  -b     -f   19
4   10  11    -26   2a
5  -44  -12     18   -6
6   4b  30    -2d   -9
7   51  -11    -13   1
8   27  0     3   -d
9   9   -18    -24   5
10  1c  -11    -6   -c

```

### 9.6.4 Example answer matrix given by the python calculation on PC

```

1   3  -29     6   -30
2  -6d -18     4e   -3
3  -11  -b     -f   19
4   10  11    -26   2a
5  -44  -12     18   -6
6   4b  30    -2d   -9
7   51  -11    -13   1
8   27  0     3   -d
9   9   -18    -24   5
10  1c  -11    -6   -c

```

## 9.7 Multi-core processor related Verilog modules

### 9.7.1 Top module - toFpga.v

```

1 module toFpga (
2   input CLOCK_50,
3   input [1:0]KEY,
4   input UART_RXD,
5   output UART_TXD,
6   output [1:0]LEDG,
7   output [6:0]HEX0,HEX1,HEX2, HEX3, HEX4,HEX5, HEX6, HEX7
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 localparam BAUD_RATE = 115200;
20 localparam UART_WIDTH = 8;
21
22 localparam idle = 3'd0,
23       uart_receive_Imem = 3'd1,
24       uart_receive_dmem = 3'd2,
25       process_ready = 3'd3,
26       process_execute = 3'd4,
27       uart_transmit_dmem = 3'd5,
28       finish = 3'd6;
29
30 ////////////// logic related to data memory //////////////////
31 wire [DATA_MEM_WIDTH-1:0]DataMemOut,DataMemIn, processor_DataOut, uart_DataOut;
32 wire [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr, uart_dataMemAddr, processor_dataMemAddr;
33

```

```

34 //////////////////////////////////////////////////////////////////
35 wire [INS_WIDTH-1:0] InsMemOut, InsMemIn;
36 wire [INS_MEM_ADDR_WIDTH-1:0] insMemAddr, uart_InsMemAddr, processor_InsMemAddr;
37
38 //////////////////////////////////////////////////////////////////
39 wire dataMemWrEn, processor_DataMemWrEn, uart_dataMemWrEn;
40 wire uart_InsMemWrEn;
41 wire rstN, clk, start;
42 wire processStartN, processDone, processor_ready;
43
44 wire txReady, rxReady;
45 wire new_byte_indicate, new_ins_byte_indicate, new_data_byte_indicate;
46 wire txByteStart;
47
48 wire [UART_WIDTH-1:0] byteForTx, byteFromRx;
49
50 wire uart_DataMem_transmitted, uart_DataMem_received, uart_InsMem_received;
51 wire uart_dmem_start_transmit;
52
53 //////////////////////////////////////////////////////////////////
54 assign LEDG[1] = processDone;
55 assign LEDG[0] = processor_ready;
56
57 reg [2:0] currentState, nextState;
58
59 always @(posedge clk) begin
60   if (~rstN) begin
61     currentState <= idle;
62   end
63   else begin
64     currentState <= nextState;
65   end
66 end
67
68 always @(*) begin
69   nextState = currentState;
70
71   case (currentState)
72     idle: begin // start state
73       if (~start) begin
74         nextState = uart_receive_Imem;
75       end
76     end
77
78     uart_receive_Imem: begin // send the instructions (machine_code) from PC through UART
79       if (uart_InsMem_received) begin
80         nextState = uart_receive_dmem;
81       end
82     end
83
84     uart_receive_dmem: begin // send the data memory values from PC through UART
85       if (uart_DataMem_received) begin
86         nextState = process_execute;
87       end
88     end
89
90     process_execute: begin // processor execute program (matrix multiplication)
91       if (processDone) begin
92         nextState = uart_transmit_dmem;
93       end
94     end
95
96     uart_transmit_dmem: begin // send the answer matrix to PC through UART
97       if (uart_DataMem_transmitted) begin
98         nextState = finish;
99       end
100    end
101
102    finish: begin //End of the process
103
104    end
105
106    default : nextState = idle;
107
108  endcase
109 end
110
111 assign clk = CLOCK_50;
112 assign rstN = KEY[0];

```

```

113 assign start = KEY[1];
114 assign processStartN = ((currentState == uart_receive_dmem) && (uart_DataMem_received))? 1'b0: 1'b1;
115 assign uart_dmem_start_transmit = ((currentState == process_exicute) && (processDone))? 1'b0: 1'b1;
116
117 assign dataMemWrEn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )?
118     uart_dataMemWrEn:
119         (currentState == process_exicute)? processor_DataMemWrEn:
120             1'b0;
121
122 assign dataMemAddr = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )?
123     uart_dataMemAddr:
124         (currentState == process_exicute)? processor_dataMemAddr:
125             {DATA_MEM_ADDR_WIDTH{1'b0}};
126
127 assign DataMemIn = ((currentState == uart_receive_dmem) || (currentState == uart_transmit_dmem) )?
128     uart_DataOut:
129         (currentState == process_exicute)? processor_DataOut:
130             {DATA_MEM_WIDTH{1'b0}};
131
132 assign insMemAddr = (currentState == uart_receive_Imem)? uart_InsMemAddr:
133     (currentState == process_exicute)? processor_InsMemAddr:
134         {INS_MEM_ADDR_WIDTH{1'b0}};
135
136
137 multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
138     .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))
139     multi_core_processor(.clk(clk),.rstN(rstN),.startN(processStartN),
140         .ProcessorDataIn(DataMemOut), .InsMemOut(InsMemOut), .ProcessorDataOut(processor_DataOut
141     ),
142         .insMemAddr(processor_InsMemAddr), .dataMemAddr(processor_dataMemAddr),
143         .DataMemWrEn(processor_DataMemWrEn), .done(processDone), .ready(processor_ready));
144
145 ////////////// instantiation of memory modules for data and instruction memory
146
147 RAM #(.DATA_WIDTH(INS_WIDTH), .ADDR_WIDTH(INS_MEM_ADDR_WIDTH), .DEPTH(INS_MEM_DEPTH))
148     IM(.address(insMemAddr), .clk(clk), .dataIn(InsMemIn), .wrEn(uart_InsMemWrEn),
149         .dataOut(InsMemOut)); // size = (256 x 8)
150
151 RAM #(.DATA_WIDTH(DATA_MEM_WIDTH), .ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .DEPTH(DATA_MEM_DEPTH))
152     DM(.address(dataMemAddr), .clk(clk), .dataIn(DataMemIn), .wrEn(dataMemWrEn),
153         .dataOut(DataMemOut)); // size = (4096 x 48)
154
155 uart_mem_interface #(.MEM_WORD_LENGTH(DATA_MEM_WIDTH), .MEM_ADDR_LENGTH(DATA_MEM_ADDR_WIDTH),
156     .UART_WIDTH(UART_WIDTH)) dataMemInterface
157     (.clk(clk), .rstN(rstN), .txStart(uart_dmem_start_transmit),
158     .mem_transmitted(uart_DataMem_transmitted), .mem_received(uart_DataMem_received),
159     //input outputs with memory
160     .dataFromMem(DataMemOut), .memWrEn(uart_dataMemWrEn), .mem_address(uart_dataMemAddr),
161     .dataToMem(uart_DataOut),
162     //inputs outputs with uart system
163     .txByteReady(txReady), .rxByteReady(rxReady), .new_rx_byte_indicate(new_data_byte_indicate),
164     .ByteFromUart(byteFromRx), .uartTxStart(txByteStart), .byteToUart(byteForTx),
165     //select start end mem addresses of tx and rx
166     .tx_start_addr_in(uartMemory[1]), .tx_end_addr_in(uartMemory[2]),
167     .rx_end_addr_in(uartMemory[0]), .toggle_addr_range(1'b1));
168
169 uart_mem_interface #(.MEM_WORD_LENGTH(INS_WIDTH), .MEM_ADDR_LENGTH(INS_MEM_ADDR_WIDTH),
170     .UART_WIDTH(UART_WIDTH)) ImemInterface
171     (.clk(clk), .rstN(rstN), .txStart(1'b1), .mem_transmitted(),
172     .mem_received(uart_InsMem_received),
173     //input outputs with memory
174     .dataFromMem(), .memWrEn(uart_InsMemWrEn), .mem_address(uart_InsMemAddr),
175     .dataToMem(InsMemIn),
176     //inputs outputs with uart system
177     .txByteReady(txReady), .rxByteReady(rxReady), .new_rx_byte_indicate(new_ins_byte_indicate),
178     .ByteFromUart(byteFromRx), .uartTxStart(), .byteToUart(),
179     //select start end mem addresses of tx and rx
180     .tx_start_addr_in(), .tx_end_addr_in(), .rx_end_addr_in(),
181     .toggle_addr_range(1'b0));
182
183 uart_system #(.DATA_WIDTH(UART_WIDTH), .BAUD_RATE(BAUD_RATE))
184     US(.clk(clk), .rstN(rstN), .txByteStart(txByteStart), .rx(UART_RXD),
185         .byteForTx(byteForTx), .tx(UART_TXD), .txReady(txReady), .rxReady(rxReady),
186         .new_byte_indicate(new_byte_indicate), .byteFromRx(byteFromRx));
187

```

```

188 localparam Q_end_addr_location = 12'd7,
189      R_start_addr_location = 12'd5,
190      R_end_addr_location = 12'd8;
191
192 reg [REG_WIDTH-1:0]uartMemory[2:0]; //0- end address of Q, 1- start address of R, 2- end address of R
193
194 always @(posedge clk) begin
195   if (uart_dataMemWrEn) begin
196     if (uart_dataMemAddr == Q_end_addr_location)
197       uartMemory[0] <= uart_DataOut[REG_WIDTH-1:0];
198     else if (uart_dataMemAddr == R_start_addr_location)
199       uartMemory[1] <= uart_DataOut[REG_WIDTH-1:0];
200     else if (uart_dataMemAddr == R_end_addr_location)
201       uartMemory[2] <= uart_DataOut[REG_WIDTH-1:0];
202   end
203 end
204
205 ///////////////////to count the time taken to the process
206 wire [25:0]currentTime;
207 timeCounter TC(.clk(clk), .rstN(rstN), .startN(processStartN), .stop(processDone),
208 .timeDuration(currentTime));
209
210 /////////////////// to show current state & no. of clock cycles on the 8 seven segments
211 hex_display HD(.clk(clk), .rstN(rstN), .state(currentState),
212 .start_timeValue_convetion(~uart_DataMem_transmitted), .timeValue(currentTime),
213 .out0(HEX0), .out1(HEX1), .out2(HEX2), .out3(HEX3), .out4(HEX4),
214 .out5(HEX5), .out6(HEX6), .out7(HEX7));
215
216 endmodule //toFpga

```

## 9.7.2 Memory

### Module - RAM.v

```

1 module RAM
2 #(
3   parameter DATA_WIDTH = 12,
4   parameter DEPTH = 256,
5   parameter ADDR_WIDTH = $clog2(DEPTH) // 4096 locations
6 )
7
8 (
9   input clk,wrEn,
10  input [DATA_WIDTH-1:0]dataIn,
11  input [ADDR_WIDTH-1:0]address,
12  output [DATA_WIDTH-1:0]dataOut
13 );
14
15 reg [DATA_WIDTH-1:0] memory [0:DEPTH-1];
16 reg [ADDR_WIDTH-1:0] addr_reg;
17
18 always @(posedge clk) begin
19   addr_reg <= address;
20   if (wrEn) begin
21     memory[address] <= dataIn;
22   end
23 end
24
25 assign dataOut = memory[addr_reg];
26
27 endmodule //RAM

```

### Testbench - RAM\_tb.v

## 9.7.3 Multi-core processor

### Module - multi\_core\_processor.v

```

1 module multi_core_processor #(

```

```

2     parameter REG_WIDTH = 12,
3     parameter INS_WIDTH = 8,
4     parameter CORE_COUNT = 4,
5     parameter DATA_MEM_ADDR_WIDTH = 12,
6     parameter INS_MEM_ADDR_WIDTH = 8
7 )
8 (
9     input clk,rstN,startN,
10    input [REG_WIDTH*CORE_COUNT-1:0] ProcessorDataIn ,
11    input [INS_WIDTH-1:0] InsMemOut ,
12    output [REG_WIDTH*CORE_COUNT-1:0] ProcessorDataOut ,
13    output [INS_MEM_ADDR_WIDTH-1:0] insMemAddr ,
14    output [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr ,
15    output DataMemWrEn , done,ready
16 );
17
18 wire core_DataMemWrEn [0:CORE_COUNT-1];
19 wire core_done [0:CORE_COUNT-1];
20 wire core_ready [0:CORE_COUNT-1];
21 wire [DATA_MEM_ADDR_WIDTH-1:0] core_dataMemAddr [0:CORE_COUNT-1];
22 wire [INS_MEM_ADDR_WIDTH-1:0] core_InsMemAddr [0:CORE_COUNT-1];
23
24 genvar i;
25 generate
26     for (i=0;i<CORE_COUNT; i=i+1) begin:core
27         processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH),
28                     .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH)) CPU
29                     (.clk(clk), .rstN(rstN), .startN(startN), .ProcessorDataIn(ProcessorDataIn[REG_WIDTH*i+:
30                         REG_WIDTH]), ,
31                         .InsMemOut(InsMemOut), .dataMemAddr(core_dataMemAddr[i]),
32                         .ProcessorDataOut(ProcessorDataOut[REG_WIDTH*i+:REG_WIDTH]), .insMemAddr(core_InsMemAddr[i]),
33                         .DataMemWrEn(core_DataMemWrEn[i]), .done(core_done[i]), .ready(core_ready[i]) );
34     end
35 endgenerate
36
37 assign dataMemAddr = core_dataMemAddr[0];
38 assign DataMemWrEn = core_DataMemWrEn[0];
39 assign insMemAddr = core_InsMemAddr[0];
40 assign ready = core_ready[0];
41 assign done = core_done[0];
42 endmodule //multi_core_processor

```

## Testbench - multi\_core\_processor\_tb.v

```

1 'timescale 1ns/1ns
2 module multi_core_processor_tb ();
3
4 localparam CLK_PERIOD = 20;
5 reg clk;
6 initial begin
7     clk <= 0;
8     forever begin
9         #(CLK_PERIOD/2);
10        clk <= ~clk;
11    end
12 end
13
14 localparam CORE_COUNT = 3;
15 localparam REG_WIDTH = 12;
16 localparam INS_WIDTH = 8;
17 localparam INS_MEM_DEPTH = 256;
18 localparam DATA_MEM_DEPTH = 4096;
19 localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
20 localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);
21 localparam DATA_MEM_WIDTH = REG_WIDTH*CORE_COUNT;
22
23 reg rstN,startN;
24 reg [REG_WIDTH*CORE_COUNT-1:0] ProcessorDataIn;
25 reg [INS_WIDTH-1:0] InsMemOut;
26 wire [REG_WIDTH*CORE_COUNT-1:0] ProcessorDataOut;
27 wire [INS_MEM_ADDR_WIDTH-1:0] insMemAddr;
28 wire [DATA_MEM_ADDR_WIDTH-1:0] dataMemAddr;
29 wire DataMemWrEn , done,ready;
30
31 multi_core_processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .CORE_COUNT(CORE_COUNT),
32                     .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH))dut
33                     (.clk(clk), .rstN(rstN), .startN(startN), .ProcessorDataIn(ProcessorDataIn),

```

```

34         .InsMemOut(InsMemOut), .ProcessorDataOut(ProcessorDataOut), .insMemAddr(insMemAddr),
35         .dataMemAddr(dataMemAddr), .DataMemWrEn(DataMemWrEn), .done(done), .ready(ready));
36
37 ///// initialize instruction and data memory /////
38
39 reg [INS_WIDTH-1:0] ins_mem [0:INS_MEM_DEPTH-1];
40 reg [DATA_MEM_WIDTH-1:0] data_mem [0:DATA_MEM_DEPTH-1];
41
42 initial begin
43     $readmemb("../9_ins_mem_tb.txt", ins_mem);
44     $readmemb("../4_data_mem_tb.txt", data_mem);
45 end
46 ///// read data and instruction memory /////
47 always @(posedge clk) begin
48     ProcessorDataIn  <= data_mem[dataMemAddr];
49     InsMemOut <= ins_mem[insMemAddr];
50 end
51
52 ///// write data memory /////
53 always @(posedge clk) begin
54     if (DataMemWrEn) begin
55         data_mem[dataMemAddr] <= ProcessorDataOut;
56     end
57 end
58
59
60 ///// start the simulation
61
62 initial begin
63     @(posedge clk);
64     rstN <= 1'b0;
65     startN <= 1'b1;
66     @(posedge clk);
67     rstN <= 1'b1;
68     startN <= 1'b0;
69     @(posedge clk);
70     startN <= 1'b1;
71 end
72
73
74
75 ///// verification of the simulation correctness /////
76
77 localparam Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
78       R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
79       R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);
80
81 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;
82
83 reg disp_temp;
84 always @(posedge clk) begin
85     if (done) begin
86         a = data_mem[12'd0][REG_WIDTH-1:0];
87         b = data_mem[12'd1][REG_WIDTH-1:0];
88         c = data_mem[12'd2][REG_WIDTH-1:0];
89         P_start_addr = data_mem[12'd3][REG_WIDTH-1:0];
90         Q_start_addr = data_mem[12'd4][REG_WIDTH-1:0];
91         R_start_addr = data_mem[12'd5][REG_WIDTH-1:0];
92         P_end_addr = data_mem[12'd6][REG_WIDTH-1:0];
93         Q_end_addr = data_mem[12'd7][REG_WIDTH-1:0];
94         R_end_addr = data_mem[12'd8][REG_WIDTH-1:0];
95
96         $writememb("../11_data_mem_out.txt", data_mem); // write data memory to file
97
98         $display("\nMatrix P\n");
99         disp_temp = print_matrix_P(a, b, P_start_addr, P_end_addr, CORE_COUNT);
100
101        $display("\nMatrix Q\n");
102        disp_temp = print_matrix_Q(b, c, Q_start_addr, Q_end_addr);
103
104        $display("\nMatrix R\n");
105        disp_temp = print_matrix_R(a, c, R_start_addr, R_end_addr, CORE_COUNT);
106
107        repeat(5) @(posedge clk); // end of the simulation
108        $stop;
109    end
110 end
111
112

```

```

113 function automatic print_matrix_P (input [REG_WIDTH-1:0]a,b,P_start_addr,P_end_addr,
114                                     input integer CORE_COUNT);
115   integer d = (a%CORE_COUNT == 0)? a/CORE_COUNT : a/CORE_COUNT+1;
116   integer x,y,z;
117   for (x=0;x<d;x=x+1) begin
118     for (y=CORE_COUNT;y>0;y=y-1) begin :core_count_loop
119       if ((x+1)*CORE_COUNT-y>= a) begin
120         disable core_count_loop;
121       end
122       for (z=0;z<b;z=z+1) begin : print_val_P
123         reg [DATA_MEM_WIDTH-1:0]temp_1 = data_mem[(P_start_addr + x*b+z)];
124         reg [REG_WIDTH-1:0]temp_2 = temp_1[(y*REG_WIDTH-1) -:REG_WIDTH];
125         $write("%h ", temp_2);
126       end
127       $write("\n");
128     end
129   end
130 endfunction
131
132 function automatic print_matrix_Q(input [REG_WIDTH-1:0]b,c,Q_start_addr,Q_end_addr);
133   integer i,j;
134   for (i=Q_start_addr;i<Q_start_addr+b;i=i+1) begin
135     for (j=i;j<=Q_end_addr;j=j+b) begin :print_val_Q
136       reg [DATA_MEM_WIDTH-1:0] temp_1 = data_mem[j];
137       $write("%h ", temp_1[REG_WIDTH-1:0]);
138     end
139     $write("\n");
140   end
141 endfunction
142
143 function automatic print_matrix_R(input [REG_WIDTH-1:0]a,c,R_start_addr,R_end_addr,
144                                     input integer CORE_COUNT);
145   integer d = (a%CORE_COUNT == 0)? a/CORE_COUNT : a/CORE_COUNT+1;
146
147   integer x,y,z;
148   for (x=0;x<d;x=x+1) begin
149     for (y=CORE_COUNT;y>0;y=y-1) begin : core_count_loop
150       if ((x+1)*CORE_COUNT-y>= a) begin
151         disable core_count_loop;
152       end
153       for (z=0;z<c;z=z+1) begin : print_val_R
154         reg [DATA_MEM_WIDTH-1:0]temp_1 = data_mem[(R_start_addr + x*c+z)];
155         reg [REG_WIDTH-1:0]temp_2 = temp_1[(y*REG_WIDTH-1) -:REG_WIDTH];
156         $write("%h ", temp_2);
157       end
158       $write("\n");
159     end
160   end
161 endfunction
162
163
164
165
166 endmodule //multi_core_processor_tb

```

### 9.7.4 Single-core processor

#### Module - processor.v

```

1 module processor
2 #(
3   parameter REG_WIDTH = 12,
4   parameter INS_WIDTH = 8,
5   parameter DATA_MEM_ADDR_WIDTH = 12,
6   parameter INS_MEM_ADDR_WIDTH = 8
7 )
8 (
9   input clk,rstN,startN,
10  input [REG_WIDTH-1:0]ProcessorDataIn ,
11  input [INS_WIDTH-1:0]InsMemOut ,
12  output [REG_WIDTH-1:0]ProcessorDataOut ,
13  output [DATA_MEM_ADDR_WIDTH-1:0]dataMemAddr ,
14  output [INS_MEM_ADDR_WIDTH-1:0]insMemAddr ,
15  output DataMemWrEn ,
16  output done,ready

```

```

17 );
18
19 wire [REG_WIDTH-1:0] alu_a, alu_b, alu_out;
20 wire [2:0] select_alu_op;
21 wire [3:0] busSel;
22 wire [REG_WIDTH-1:0] busOut;
23 wire [INS_WIDTH-1:0] IROUT;
24 wire [REG_WIDTH-1:0] Rout, RLout, RCout, RPout, RQout, R1out, ACout;
25 wire Zout, ZWrEn;
26 wire [3:0] incReg; // {PC, RC, RP, RQ}
27 wire [9:0] wrEnReg; // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
28
29 //instantiation of each module within the processor are as follows.
30
31 controlUnit #(.INS_WIDTH(INS_WIDTH)) CU(.clk(clk), .rstN(rstN), .startN(startN), .Zout(Zout), .ins(IROUT),
32 .aluOp(select_alu_op), .incReg(incReg), .wrEnReg(wrEnReg), .busSel(busSel),
33 .DataMemWrEn(DataMemWrEn), .ZWrEn(ZWrEn), .done(done), .ready(ready));
34
35 ALU #(.WIDTH(REG_WIDTH)) alu(.a(alu_a), .b(alu_b), .selectOp(select_alu_op), .dataOut(alu_out));
36
37 multiplexer #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH)) mux(.selectIn(busSel), .DMem(ProcessorDataIn),
38 .R(Rout), .RL(RLout), .RC(RCout), .RP(RPout), .RQ(RQout), .R1(R1out), .AC(ACout),
39 .IR(IROUT), .busOut(busOut));
40
41 register #(.WIDTH(REG_WIDTH)) AR(.dataIn(busOut), .wrEn(wrEnReg[9]), .rstN(rstN), .clk(clk),
42 .dataOut(dataMemAddr));
43
44 register #(.WIDTH(REG_WIDTH)) R(.dataIn(busOut), .wrEn(wrEnReg[8]), .rstN(rstN), .clk(clk),
45 .dataOut(Rout));
46
47 incRegister #(.WIDTH(INS_WIDTH)) PC(.dataIn(IROUT), .wrEn(wrEnReg[7]), .rstN(rstN), .clk(clk),
48 .incEn(incReg[3]), .dataOut(insMemAddr));
49
50 register #(.WIDTH(INS_WIDTH)) IR(.dataIn(InsMemOut), .wrEn(wrEnReg[6]), .rstN(rstN),
51 .clk(clk), .dataOut(IROUT));
52
53 register #(.WIDTH(REG_WIDTH)) RL(.dataIn(busOut), .wrEn(wrEnReg[5]), .rstN(rstN), .clk(clk),
54 .dataOut(RLout));
55
56 incRegister #(.WIDTH(REG_WIDTH)) RC(.dataIn(busOut), .wrEn(wrEnReg[4]), .rstN(rstN),
57 .clk(clk), .incEn(incReg[2]), .dataOut(RCout));
58
59 incRegister #(.WIDTH(REG_WIDTH)) RP(.dataIn(busOut), .wrEn(wrEnReg[3]), .rstN(rstN),
60 .clk(clk), .incEn(incReg[1]), .dataOut(RPout));
61
62 incRegister #(.WIDTH(REG_WIDTH)) RQ(.dataIn(busOut), .wrEn(wrEnReg[2]), .rstN(rstN),
63 .clk(clk), .incEn(incReg[0]), .dataOut(RQout));
64
65 register #(.WIDTH(REG_WIDTH)) R1(.dataIn(busOut), .wrEn(wrEnReg[1]), .rstN(rstN), .clk(clk),
66 .dataOut(R1out));
67
68 register #(.WIDTH(REG_WIDTH)) AC(.dataIn(alu_out), .wrEn(wrEnReg[0]), .rstN(rstN), .clk(clk),
69 .dataOut(ACout));
70
71 zReg #(.WIDTH(REG_WIDTH)) Z(.dataIn(alu_out), .clk(clk), .rstN(rstN), .wrEn(ZWrEn), .Zout(Zout));
72
73 // necessary wires are routed as follows among modules as below.
74 assign ProcessorDataOut = Rout;
75 assign alu_a = ACout;
76 assign alu_b = busOut;
77
78 endmodule //processor

```

## Testbench - processor\_tb.v

```

1 `timescale 1ns/1ps
2 module processor_tb ();
3
4 localparam CLK_PERIOD = 20;
5 reg clk;
6 initial begin
7   clk <= 0;
8   forever begin
9     #(CLK_PERIOD/2);
10    clk <= ~clk;
11  end
12 end
13

```

```

14 localparam REG_WIDTH = 12;
15 localparam INS_WIDTH = 8;
16 localparam INS_MEM_DEPTH = 256;
17 localparam DATA_MEM_DEPTH = 4096;
18 localparam DATA_MEM_ADDR_WIDTH = $clog2(DATA_MEM_DEPTH);
19 localparam INS_MEM_ADDR_WIDTH = $clog2(INS_MEM_DEPTH);
20
21 reg rstN,startN;
22 reg [REG_WIDTH-1:0] ProcessorDataIn;
23 reg [INS_WIDTH-1:0] InsMemOut;
24 wire [REG_WIDTH-1:0] dataMemAddr;
25 wire [REG_WIDTH-1:0] ProcessorDataOut;
26 wire [INS_WIDTH-1:0] insMemAddr;
27 wire DataMemWrEn;
28 wire done,ready;
29
30 processor #(.REG_WIDTH(REG_WIDTH), .INS_WIDTH(INS_WIDTH), .INS_MEM_ADDR_WIDTH(INS_MEM_ADDR_WIDTH),
31 .DATA_MEM_ADDR_WIDTH(DATA_MEM_ADDR_WIDTH))dut(
32 .clk(clk), .rstN(rstN), .startN(startN), .ProcessorDataIn(ProcessorDataIn),
33 .InsMemOut(InsMemOut), .ProcessorDataOut(ProcessorDataOut), .dataMemAddr(dataMemAddr),
34 .insMemAddr(insMemAddr), .DataMemWrEn(DataMemWrEn), .done(done), .ready(ready));
35
36 //////////////////////////////////////////////////////////////////
37
38 reg [INS_WIDTH-1:0] ins_mem[0:INS_MEM_DEPTH-1];
39 reg [REG_WIDTH-1:0] data_mem[0:DATA_MEM_DEPTH-1];
40
41 initial begin
42 $readmemb("../9_ins_mem_tb.txt", ins_mem);
43 $readmemb("../4_data_mem_tb.txt", data_mem);
44 end
45
46 ////////////////////////////////////////////////////////////////// read data and instruction memory ///////////////
47 always @(posedge clk) begin
48 ProcessorDataIn <= data_mem[dataMemAddr];
49 InsMemOut <= ins_mem[insMemAddr];
50 end
51
52 ////////////////////////////////////////////////////////////////// write data memory ///////////////
53 always @(posedge clk) begin
54 if (DataMemWrEn) begin
55 data_mem[dataMemAddr] <= ProcessorDataOut;
56 end
57 end
58
59 ////////////////////////////////////////////////////////////////// start the simulation
60
61 initial begin
62 @(posedge clk);
63 rstN <= 1'b0;
64 startN <= 1'b1;
65 @(posedge clk);
66 rstN <= 1'b1;
67 startN <= 1'b0;
68 @(posedge clk);
69 startN <= 1'b1;
70 end
71
72 ////////////////////////////////////////////////////////////////// verification of the simulation correctness ///////////////
73
74 localparam Q_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd7),
75 R_start_addr_location = DATA_MEM_ADDR_WIDTH'(12'd5),
76 R_end_addr_location = DATA_MEM_ADDR_WIDTH'(12'd8);
77
78 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;
79
80 reg disp_temp;
81 always @(posedge clk) begin
82 if (done) begin
83 a = data_mem[12'd0][REG_WIDTH-1:0];
84 b = data_mem[12'd1][REG_WIDTH-1:0];
85 c = data_mem[12'd2][REG_WIDTH-1:0];
86 P_start_addr = data_mem[12'd3][REG_WIDTH-1:0];
87 Q_start_addr = data_mem[12'd4][REG_WIDTH-1:0];
88 R_start_addr = data_mem[12'd5][REG_WIDTH-1:0];
89 P_end_addr = data_mem[12'd6][REG_WIDTH-1:0];
90 Q_end_addr = data_mem[12'd7][REG_WIDTH-1:0];
91 R_end_addr = data_mem[12'd8][REG_WIDTH-1:0];
92

```

```

93     $writememb("../11_data_mem_out.txt", data_mem); // write data memory to a text file
94
95     $display("\nMatrix P\n");
96     disp_temp = print_matrix_P(a, b, P_start_addr, P_end_addr);
97
98     $display("\nMatrix Q\n");
99     disp_temp = print_matrix_Q(b, c, Q_start_addr, Q_end_addr);
100
101    $display("\nMatrix R\n");
102    disp_temp = print_matrix_R(a, c, R_start_addr, R_end_addr);
103
104    repeat(5) @(posedge clk); // end of the simulation
105    $stop;
106  end
107 end
108
109 function automatic print_matrix_P(input [REG_WIDTH-1:0]a,b,P_start_addr,P_end_addr);
110   integer i,j;
111   for (i=P_start_addr;i<P_end_addr;i=i+b) begin
112     for (j=i;j<i+b;j=j+1) begin
113       $write("%h ", data_mem[j]);
114     end
115     $write("\n");
116   end
117 endfunction
118
119 function automatic print_matrix_Q(input [REG_WIDTH-1:0]b,c,Q_start_addr,Q_end_addr);
120   integer i,j;
121   for (i=Q_start_addr;i<Q_start_addr+b;i=i+1) begin
122     for (j=i;j<=Q_end_addr;j=j+b) begin
123       $write("%h ", data_mem[j]);
124     end
125     $write("\n");
126   end
127 endfunction
128
129 function automatic print_matrix_R(input [REG_WIDTH-1:0]a,c,R_start_addr,R_end_addr);
130   integer i,j;
131   for (i=R_start_addr;i<R_end_addr;i=i+c) begin
132     for (j=i;j<i+c;j=j+1) begin
133       $write("%h ", data_mem[j]);
134     end
135     $write("\n");
136   end
137 endfunction
138
139 endmodule // processor_tb

```

## 9.7.5 Control Unit

### Module - controlUnit.v

```

1 module controlUnit
2 #(
3   parameter INS_WIDTH = 8
4 )
5 (
6   input clk,rstN,startN,Zout,
7   input [INS_WIDTH-1:0]ins,
8   output reg [2:0]aluOp,
9   output reg [3:0]incReg,    // {PC, RC, RP, RQ}
10  output reg [9:0]wrEnReg,   // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
11  output reg [3:0]busSel,
12  output reg DataMemWrEn,ZWrEn,
13  output done,ready
14 );
15
16 // all the states (47) for the instructions are as follows
17
18 localparam IDLE = 6'd0, //states
19
20   NOP1 = 6'd1,
21
22   ENDOP1 = 6'd2,
23

```

```

24      CLAC1 = 6'd3,
25
26      FETCH_DELAY1 = 6'd37,      //////(extra_delay for memory read)
27      FETCH1 = 6'd4,
28      FETCH2 = 6'd5,
29
30      LDIAC_DELAY1 = 6'd38,      ////// (extra_delay for memory read)
31      LDIAC1 = 6'd6,
32      LDIAC2 = 6'd7,
33      LDIAC_DELAY2 = 6'd39,      ///// (extra_delay for memory read)
34      LDIAC3 = 6'd8,
35
36      LDAC1 = 6'd9,
37      LDAC_DELAY1 = 6'd40,      ////// (extra_delay for memory read)
38      LDAC2 = 6'd10,
39
40      STIR_DELAY1 = 6'd41,      ////// (extra_delay for memory read)
41      STIR1 = 6'd11,
42      STIR2 = 6'd12,
43      STIR_DELAY2 = 6'd42,      ////// (extra_delay for memory write)
44      STIR3 = 6'd13,
45
46      STR1 = 6'd14,
47      STR_DELAY1 = 6'd43,      ////// (extra_delay for memory write)
48      STR2 = 6'd15,
49
50      JUMP_DELAY1 = 6'd44,      ////// (extra_delay for memory read)
51      JUMP1 = 6'd16,
52      JUMP2 = 6'd17,
53
54      JMPNZY_DELAY1 = 6'd45,      ////// (extra_delay for memory read)
55      JMPNZY1 = 6'd18,
56      JMPNZY2 = 6'd19,
57      JMPZN1 = 6'd20,
58
59      JMPZY_DELAY1 = 6'd46,      ////// (extra_delay for memory read)
60      JMPZY1 = 6'd21,
61      JMPZY2 = 6'd22,
62      JMPZN1 = 6'd23,
63
64      MUL1 = 6'd24,
65
66      ADD1 = 6'd25,
67
68      SUB1 = 6'd26,
69
70      INCAC1 = 6'd27,
71
72      MV_RL_AC1 = 6'd28,
73
74      MV_RP_AC1 = 6'd29,
75
76      MV_RQ_AC1 = 6'd30,
77
78      MV_RC_AC1 = 6'd31,
79
80      MV_R_AC1 = 6'd32,
81
82      MV_R1_AC1 = 6'd33,
83
84      MV_AC_RP1 = 6'd34,
85
86      MV_AC_RQ1 = 6'd35,
87
88      MV_AC_RL1 = 6'd36;
89
90 localparam clr_alu = 3'd0,      //alu operations
91      pass_alu = 3'd1,
92      add_alu = 3'd2,
93      sub_alu = 3'd3,
94      mul_alu = 3'd4,
95      inc_alu = 3'd5,
96      idle_alu = 3'd6;
97
98 localparam DMem_bus = 4'b0,      //multiplexer
99      R_bus = 4'd1,
100     IR_bus = 4'd2,
101     RL_bus = 4'd3,
102     RC_bus = 4'd4,

```

```

103      RP_bus    = 4'd5,
104      RQ_bus    = 4'd6,
105      R1_bus    = 4'd7,
106      AC_bus    = 4'd8,
107      idle_bus  = 4'd9;
108
109 localparam  //instruction set (assembly instructions and their machine codes)
110   NOP        = 8'd0,
111   ENDOP     = 8'd1,
112   CLAC      = 8'd2,
113   LDIAC     = 8'd3,
114   LDAC      = 8'd4,
115   STR       = 8'd5,
116   STIR      = 8'd6,
117   JUMP      = 8'd7,
118   JMPNZ    = 8'd8,
119   JMPZ     = 8'd9,
120   MUL       = 8'd10,
121   ADD       = 8'd11,
122   SUB       = 8'd12,
123   INCAC    = 8'd13,
124   MV_RL_AC = {4'd1,4'd15},
125   MV_RP_AC = {4'd2,4'd15},
126   MV_RQ_AC = {4'd3,4'd15},
127   MV_RC_AC = {4'd4,4'd15},
128   MV_R_AC  = {4'd5,4'd15},
129   MV_R1_AC = {4'd6,4'd15},
130   MV_AC_RP = {4'd7,4'd15},
131   MV_AC_RQ = {4'd8,4'd15},
132   MV_AC_RL = {4'd9,4'd15};
133
134 localparam [9:0]
135   no_wrEn = 10'b0000000000,
136   AR_wrEn = 10'b1000000000,
137   R_wrEn  = 10'b0100000000,
138   PC_wrEn = 10'b0010000000,
139   IR_wrEn = 10'b0001000000,
140   RL_wrEn = 10'b0000100000,
141   RC_wrEn = 10'b0000010000,
142   RP_wrEn = 10'b0000001000,
143   RQ_wrEn = 10'b0000000100,
144   R1_wrEn = 10'b0000000010,
145   AC_wrEn = 10'b0000000001;
146
147 localparam [3:0]
148   no_inc = 4'b0000,
149   PC_inc = 4'b1000,
150   RC_inc = 4'b0100,
151   RP_inc = 4'b0010,
152   RQ_inc = 4'b0001,
153   RC_RP_RQ_inc = 4'b0111;
154
155 reg [5:0] currentState, nextState;
156
157 always @ (posedge clk) begin
158   if (~rstN) begin
159     currentState <= IDLE;
160   end
161   else begin
162     currentState <= nextState;
163   end
164 end
165
166 //all the control signals for each state are set as below
167 always @ (startN, Zout, ins, currentState) begin
168   case (currentState)
169     IDLE: begin
170       aluOp <= idle_alu;
171       incReg <= 4'd0;
172       wrEnReg <= 10'd0;
173       busSel <= idle_bus;
174       DataMemWrEn <= 1'b0;
175       ZWrEn <= 1'b0;
176
177       if (~startN)
178         nextState <= FETCH1;
179       else
180         nextState <= IDLE;
181   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)? JMPNZ_DELAY1 : JMPZN1;
251             JMPZ: nextState <= (Zout == 1)? JMPZ_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;

```

```

261             MV_R1_AC:  nextState  <= MV_R1_AC1;
262             MV_AC_RP:  nextState  <= MV_AC_RP1;
263             MV_AC_RQ:  nextState  <= MV_AC_RQ1;
264             MV_AC_RL:  nextState  <= MV_AC_RL1;
265             default   :  nextState  <= IDLE;
266
267         endcase
268     end
269
270     LDIAC_DELAY1: begin
271         aluOp  <= idle_alu;
272         incReg <= 4'd0;
273         wrEnReg <= IR_wrEn;
274         busSel  <= idle_bus;
275         DataMemWrEn <= 1'b0;
276         ZWrEn  <= 1'b0;
277         nextState <= LDIAC1;
278     end
279
280     LDIAC1: begin
281         aluOp  <= idle_alu;
282         incReg <= 4'd0;
283         wrEnReg <= IR_wrEn;
284         busSel  <= idle_bus;
285         DataMemWrEn <= 1'b0;
286         ZWrEn  <= 1'b0;
287         nextState <= LDIAC2;
288     end
289
290     LDIAC2: begin
291         aluOp  <= idle_alu;
292         incReg <= PC_inc;
293         wrEnReg <= AR_wrEn;
294         busSel  <= IR_bus;
295         DataMemWrEn <= 1'b0;
296         ZWrEn  <= 1'b0;
297         nextState <= LDIAC_DELAY2;
298     end
299
300     LDIAC_DELAY2: begin
301         aluOp  <= pass_alu;
302         incReg <= no_inc;
303         wrEnReg <= AC_wrEn;
304         busSel  <= DMem_bus;
305         DataMemWrEn <= 1'b0;
306         ZWrEn  <= 1'b1;
307         nextState <= LDIAC3;
308     end
309
310     LDIAC3: begin
311         aluOp  <= pass_alu;
312         incReg <= no_inc;
313         wrEnReg <= AC_wrEn;
314         busSel  <= DMem_bus;
315         DataMemWrEn <= 1'b0;
316         ZWrEn  <= 1'b1;
317         nextState <= FETCH_DELAY1;
318     end
319
320     LDAC1: begin
321         aluOp  <= idle_alu;
322         incReg <= no_inc;
323         wrEnReg <= AR_wrEn;
324         busSel  <= AC_bus;
325         DataMemWrEn <= 1'b0;
326         ZWrEn  <= 1'b0;
327         nextState <= LDAC_DELAY1;
328     end
329
330     LDAC_DELAY1: begin
331         aluOp  <= pass_alu;
332         incReg <= no_inc;
333         wrEnReg <= AC_wrEn;
334         busSel  <= DMem_bus;
335         DataMemWrEn <= 1'b0;
336         ZWrEn  <= 1'b1;
337         nextState <= LDAC2;
338     end
339

```

```

340      LDAC2: begin
341          aluOp <= pass_alu;
342          incReg <= no_inc;
343          wrEnReg <= AC_wrEn;
344          busSel <= DMem_bus;
345          DataMemWrEn <= 1'b0;
346          ZWrEn <= 1'b1;
347          nextState <= FETCH_DELAY1;
348      end
349
350      STR1: begin
351          aluOp <= idle_alu;
352          incReg <= no_inc;
353          wrEnReg <= AR_wrEn;
354          busSel <= AC_bus;
355          DataMemWrEn <= 1'b0;
356          ZWrEn <= 1'b0;
357          nextState <= STR_DELAY1;
358      end
359
360      STR_DELAY1: begin
361          aluOp <= idle_alu;
362          incReg <= no_inc;
363          wrEnReg <= no_wrEn;
364          busSel <= idle_bus;
365          DataMemWrEn <= 1'b1;
366          ZWrEn <= 1'b0;
367          nextState <= STR2;
368      end
369
370      STR2: begin
371          aluOp <= idle_alu;
372          incReg <= no_inc;
373          wrEnReg <= no_wrEn;
374          busSel <= idle_bus;
375          DataMemWrEn <= 1'b1;
376          ZWrEn <= 1'b0;
377          nextState <= FETCH1;
378      end
379
380      STIR_DELAY1: begin
381          aluOp <= idle_alu;
382          incReg <= no_inc;
383          wrEnReg <= IR_wrEn;
384          busSel <= idle_bus;
385          DataMemWrEn <= 1'b0;
386          ZWrEn <= 1'b0;
387          nextState <= STIR1;
388      end
389
390      STIR1: begin
391          aluOp <= idle_alu;
392          incReg <= no_inc;
393          wrEnReg <= IR_wrEn;
394          busSel <= idle_bus;
395          DataMemWrEn <= 1'b0;
396          ZWrEn <= 1'b0;
397          nextState <= STIR2;
398      end
399
400      STIR2: begin
401          aluOp <= idle_alu;
402          incReg <= PC_inc;
403          wrEnReg <= AR_wrEn;
404          busSel <= IR_bus;
405          DataMemWrEn <= 1'b0;
406          ZWrEn <= 1'b0;
407          nextState <= STIR_DELAY2;
408      end
409
410      STIR_DELAY2: begin
411          aluOp <= idle_alu;
412          incReg <= no_inc;
413          wrEnReg <= no_wrEn;
414          busSel <= idle_bus;
415          DataMemWrEn <= 1'b1;
416          ZWrEn <= 1'b0;
417          nextState <= STIR3;
418      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_DELAY1: 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     JMPZN1: 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;
```

```
577     nextState <= FETCH_DELAY1;
578 end

579 MV_RL_AC1: begin
580     aluOp <= idle_alu;
581     incReg <= no_inc;
582     wrEnReg <= RL_wrEn;
583     busSel <= AC_bus;
584     DataMemWrEn <= 1'b0;
585     ZWrEn <= 1'b0;
586     nextState <= FETCH_DELAY1;
587 end

588 MV_RP_AC1: begin
589     aluOp <= idle_alu;
590     incReg <= no_inc;
591     wrEnReg <= RP_wrEn;
592     busSel <= AC_bus;
593     DataMemWrEn <= 1'b0;
594     ZWrEn <= 1'b0;
595     nextState <= FETCH_DELAY1;
596 end

597 MV_RQ_AC1: begin
598     aluOp <= idle_alu;
599     incReg <= no_inc;
600     wrEnReg <= RQ_wrEn;
601     busSel <= AC_bus;
602     DataMemWrEn <= 1'b0;
603     ZWrEn <= 1'b0;
604     nextState <= FETCH_DELAY1;
605 end

606 MV_RC_AC1: begin
607     aluOp <= idle_alu;
608     incReg <= no_inc;
609     wrEnReg <= RC_wrEn;
610     busSel <= AC_bus;
611     DataMemWrEn <= 1'b0;
612     ZWrEn <= 1'b0;
613     nextState <= FETCH_DELAY1;
614 end

615 MV_R_AC1: begin
616     aluOp <= idle_alu;
617     incReg <= no_inc;
618     wrEnReg <= R_wrEn;
619     busSel <= AC_bus;
620     DataMemWrEn <= 1'b0;
621     ZWrEn <= 1'b0;
622     nextState <= FETCH_DELAY1;
623 end

624 MV_R1_AC1: begin
625     aluOp <= idle_alu;
626     incReg <= no_inc;
627     wrEnReg <= R1_wrEn;
628     busSel <= AC_bus;
629     DataMemWrEn <= 1'b0;
630     ZWrEn <= 1'b0;
631     nextState <= FETCH_DELAY1;
632 end

633 MV_AC_RP1: begin
634     aluOp <= pass_alu;
635     incReg <= no_inc;
636     wrEnReg <= AC_wrEn;
637     busSel <= RP_bus;
638     DataMemWrEn <= 1'b0;
639     ZWrEn <= 1'b1;
640     nextState <= FETCH_DELAY1;
641 end

642 MV_AC_RQ1: begin
643     aluOp <= pass_alu;
644     incReg <= no_inc;
645     wrEnReg <= AC_wrEn;
646     busSel <= RQ_bus;
647     DataMemWrEn <= 1'b0;
648 end

649 MV_AC_R1: begin
650     aluOp <= pass_alu;
651     incReg <= no_inc;
652     wrEnReg <= AC_wrEn;
653     busSel <= R1_bus;
654     DataMemWrEn <= 1'b0;
655 end
```

```

656         ZWrEn <= 1'b1;
657         nextState <= FETCH_DELAY1;
658     end
659
660     MV_AC_RL1: begin
661         aluOp <= pass_alu;
662         incReg <= no_inc;
663         wrEnReg <= AC_wrEn;
664         busSel <= RL_bus;
665         DataMemWrEn <= 1'b0;
666         ZWrEn <= 1'b1;
667         nextState <= FETCH_DELAY1;
668     end
669
670     default : begin
671         aluOp <= idle_alu;
672         incReg <= 4'd0;
673         wrEnReg <= 10'd0;
674         busSel <= 4'd0;
675         DataMemWrEn <= 1'b0;
676         ZWrEn <= 1'b0;
677         nextState <= IDLE;
678     end
679
680 endcase
681
682 end
683
684 // signals to the user whether processor is ready or the process is over
685 assign done = (currentState == ENDOP1)?1'b1:1'b0;
686 assign ready = (currentState == IDLE)? 1'b1:1'b0;
687
688 endmodule //controlUnit

```

## Testbench - controlUnit\_tb.sv

```

1 `timescale 1ns/1ps
2
3 module controlUnit_tb ();
4
5 localparam CLK_PERIOD = 20;
6 reg clk;
7 initial begin
8     clk <= 0;
9     forever begin
10         #(CLK_PERIOD/2);
11         clk <= ~clk;
12     end
13 end
14
15 localparam INS_WIDTH = 8;
16
17 reg rstN,startN,Zout;
18 reg [INS_WIDTH-1:0] ins;
19 wire [2:0] aluOp;
20 wire [3:0] incReg;      // {PC, RC, RP, RQ}
21 wire [9:0] wrEnReg;    // {AR, R, PC, IR, RL, RC, RP, RQ, R1, AC}
22 wire [3:0] busSel;
23 wire DataMemWrEn,ZWrEn;
24 wire done,ready;
25
26 localparam [INS_WIDTH-1:0]
27     NOP      = 8'd0,
28     ENDOP   = 8'd1,
29     CLAC    = 8'd2,
30     LDIAC   = 8'd3,
31     LDAC    = 8'd4,
32     STR     = 8'd5,
33     STIR    = 8'd6,
34     JUMP    = 8'd7,
35     JMPNZ   = 8'd8,
36     JMPZ    = 8'd9,
37     MUL     = 8'd10,
38     ADD     = 8'd11,
39     SUB     = 8'd12,
40     INCAC   = 8'd13,
41     MV_RL_AC = {4'd1,4'd15},
42     MV_RP_AC = {4'd2,4'd15},

```

```

43     MV_RQ_AC      = {4'd3,4'd15},
44     MV_RC_AC      = {4'd4,4'd15},
45     MV_R_AC       = {4'd5,4'd15},
46     MV_R1_AC      = {4'd6,4'd15},
47     MV_AC_RP      = {4'd7,4'd15},
48     MV_AC_RQ      = {4'd8,4'd15},
49     MV_AC_RL      = {4'd9,4'd15};
50
51 localparam [31:0]// time duration for each instruction to execute
52     NOP_time_duration      = 4,
53     ENDOP_time_duration    = 4,
54     CLAC_time_duration     = 4,
55     LDIAC_time_duration    = 8,
56     LDAC_time_duration     = 6,
57     STR_time_duration      = 6,
58     STIR_time_duration     = 8,
59     JUMP_time_duration     = 6,
60     JMPNZ_Y_time_duration = 6,
61     JMPNZ_N_time_duration = 4,
62     JMPZ_Y_time_duration  = 6,
63     JMPZ_N_time_duration  = 4,
64     MUL_time_duration      = 4,
65     ADD_time_duration      = 4,
66     SUB_time_duration      = 4,
67     INCAC_time_duration    = 4,
68     MV_RL_AC_time_duration = 4,
69     MV_RP_AC_time_duration = 4,
70     MV_RQ_AC_time_duration = 4,
71     MV_RC_AC_time_duration = 4,
72     MV_R_AC_time_duration  = 4,
73     MV_R1_AC_time_duration = 4,
74     MV_AC_RP_time_duration = 4,
75     MV_AC_RQ_time_duration = 4,
76     MV_AC_RL_time_duration = 4;
77
78
79 controlUnit #(INS_WIDTH(INS_WIDTH)) dut(.clk(clk), .rstN(rstN), .startN(startN),
80           .Zout(Zout), .ins(ins), .aluOp(aluOp), .incReg(incReg), .wrEnReg(wrEnReg),
81           .busSel(busSel), .DataMemWrEn(DataMemWrEn), .ZWrEn(ZWrEn),
82           .done(done), .ready(ready));
83
84 task automatic test_instruction(
85     input [31:0] duration,
86     input Z_value,
87     input [INS_WIDTH-1:0] instruction
88 ); begin
89
90     @(posedge clk);
91     ins <= instruction;
92     Zout <= Z_value;
93
94     #(duration * CLK_PERIOD);
95 end
96
97 endtask
98
99
100 initial begin
101     @(posedge clk);
102     rstN <= 1'b0;
103     @(posedge clk);
104     rstN <= 1'b1;
105     startN <= 1'b0;
106     @(posedge clk);
107     startN <= 1'b1;
108
109     ////// test NOP
110     test_instruction(NOP_time_duration, 1'bX, NOP);
111
112     ////// test CLAC
113     test_instruction(CLAC_time_duration, 1'bX, CLAC);
114
115     ////// test LDIAC
116     test_instruction(LDIAC_time_duration, 1'bX, LDIAC);
117
118     ////// test LDAC
119     test_instruction(LDAC_time_duration, 1'bX, LDAC);
120
121     ////// 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

```

1 module ALU
2 #(parameter WIDTH = 12)
3 (
4     input signed [WIDTH-1:0]a,b, // a - from Accumulator, b - from data-bus
5     input [2:0]selectOp,
6     output signed [WIDTH-1:0]dataOut
7 );
8
9 localparam [2:0]
10    clr = 3'd0, // ALU operations and their control signals
11    pass = 3'd1,
12    add = 3'd2,
13    sub = 3'd3,
14    mul = 3'd4,
15    inc = 3'd5,
16    idle = 3'd6;
17
18 //output of the ALU is decided as below
19 assign dataOut = (selectOp == clr)? {WIDTH{1'b0}}:
20                 (selectOp == pass)? b:
21                 (selectOp == add)? a+b:
22                 (selectOp == sub)? a-b:
23                 (selectOp == mul)? a*b:
24                 (selectOp == inc)? a+1'b1:
25                 (selectOp == idle)? {WIDTH{1'b0}}:
26                 {WIDTH{1'b0}}; // default value is ZERO
27
28 endmodule //ALU

```

### Testbench - alu\_tb.v

```

1 'timescale 1ns/1ps
2 module ALU_tb();
3
4 localparam CLK_PERIOD = 20;
5
6 reg clk;
7 initial begin
8     clk = 0;
9     forever #(CLK_PERIOD/2) clk <= ~clk;
10 end
11
12 localparam [2:0]
13    clr = 3'd0, // ALU operations and their control signals
14    pass = 3'd1,
15    add = 3'd2,
16    sub = 3'd3,
17    mul = 3'd4,
18    inc = 3'd5,
19    idle = 3'd6;
20
21 localparam WIDTH = 12;
22
23 reg signed [WIDTH-1:0]a,b;
24 reg [2:0] selectOp;
25 wire signed [WIDTH-1:0] dataOut;
26
27
28 ALU #(WIDTH(WIDTH))dut(.a(a), .b(b), .selectOp(selectOp), .dataOut(dataOut));
29
30 initial begin
31     @(posedge clk);
32     a <= 10;
33     b <= 3;
34     selectOp <= clr;
35
36     @(posedge clk);
37     selectOp <= pass;
38
39     @(posedge clk);
40     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

```

1  `timescale 1ns/1ps
2
3  module incRegister_tb();
4
5  localparam CLK_PERIOD = 20;
6  reg clk;
7  initial begin
8      clk <= 0;
9      forever begin
10         #(CLK_PERIOD/2);
11         clk <= ~clk;
12     end
13 end
14
15 localparam WIDTH = 12;
16
17 reg [WIDTH-1:0] dataIn;
18 reg wrEn, incEn, rstN;
19 wire [WIDTH-1:0] dataOut;
20
21 incRegister #(WIDTH(WIDTH)) dut(.dataIn(dataIn), .wrEn(wrEn), .rstN(rstN),
22                               .clk(clk), .incEn(incEn), .dataOut(dataOut));
23
24 initial begin
25     @(posedge clk);
26     #(CLK_PERIOD*4/5);
27     rstN <= 0;
28
29     @(posedge clk);
30     #(CLK_PERIOD*4/5);
31     dataIn <= 23;
32     wrEn <= 1;
33
34     @(posedge clk);
35     #(CLK_PERIOD*4/5);
36     dataIn <= 36;
37     wrEn <= 1;
38     rstN <= 1;
39
40     @(posedge clk);
41     #(CLK_PERIOD*4/5);
42     dataIn <= 15;
43     wrEn <= 0;
44     incEn <= 1;
45
46     repeat (10) begin
47         @(posedge clk);
48         #(CLK_PERIOD*4/5);
49         wrEn = $random();
50         incEn = $random();
51         rstN = $random();
52         dataIn = $random();
53     end
54
55     $stop;
56 end
57
58 initial begin
59     forever begin
60         @(posedge clk);
61         #(CLK_PERIOD*1/5);
62         $display("dataIn = %d    rstN = %b    wrEn = %b    incEn = %b    dataOut = %d",
63                  dataIn, rstN, wrEn, incEn, dataOut);
64     end
65
66 end
67
68 endmodule:incRegister_tb

```

## 9.7.8 Register

### Module - register.v

```

1 module register
2 #(parameter WIDTH = 12) //default size of a register is 12
3 (
4     input [WIDTH-1:0] dataIn,
5     input wrEn,rstN,clk,
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 end
17
18 assign dataOut = value;
19
20 endmodule //register

```

### Testbench - register\_tb.v

```

1 'timescale 1ns/1ps
2
3 module register_tb();
4
5 localparam CLK_PERIOD = 20;
6
7 reg clk;
8 initial begin
9     clk <= 0;
10    forever begin
11        #(CLK_PERIOD/2);
12        clk <= ~clk;
13    end
14 end
15
16 localparam WIDTH = 12;
17 reg [WIDTH-1:0] dataIn;
18 reg rstN, wrEn;
19 wire [WIDTH-1:0] dataOut;
20
21 register #(.WIDTH(WIDTH)) dut(.clk(clk), .wrEn(wrEn), .rstN(rstN), .dataIn(dataIn), .dataOut(dataOut));
22
23 initial begin
24     @(posedge clk);
25     rstN <= 0;
26
27     @(posedge clk);
28     rstN <= 1;
29     dataIn <= 20;
30     wrEn <= 0;
31
32     @(posedge clk);
33     dataIn <= 43;
34     wrEn <= 1;
35
36     repeat(10) begin
37         @(posedge clk);
38         dataIn = $random();
39         wrEn = $random();
40         rstN = $random();
41     end
42
43     @(posedge clk);
44     $stop;
45 end
46
47 endmodule // register_tb

```

## 9.7.9 Multiplexer (System Bus)

### Module - multiplexer.v

```

1 module multiplexer
2 #(
3     parameter REG_WIDTH = 12,
4     parameter INS_WIDTH = 8
5 )
6 (
7     input [3:0] selectIn,
8     input [REG_WIDTH-1:0] DMem, R, RL, RC, RP, RQ, R1, AC,
9     input [INS_WIDTH-1:0] IR,
10    output [REG_WIDTH-1:0] busOut
11 );
12
13 //control signals to select the bus input
14 localparam [3:0]
15     DMem_sel = 4'b0,
16     R_sel    = 4'd1,
17     IR_sel   = 4'd2,
18     RL_sel   = 4'd3,
19     RC_sel   = 4'd4,
20     RP_sel   = 4'd5,
21     RQ_sel   = 4'd6,
22     R1_sel   = 4'd7,
23     AC_sel   = 4'd8,
24     idle     = 4'd9;
25
26
27 //input of the data-bus is selected as below
28 assign busOut = (selectIn == DMem_sel)? DMem:
29             (selectIn == R_sel)? R:
30             (selectIn == IR_sel)? {{(REG_WIDTH-INS_WIDTH){1'b0}},IR}: //first 3 bits are zero
31             (selectIn == RL_sel)? RL:
32             (selectIn == RC_sel)? RC:
33             (selectIn == RP_sel)? RP:
34             (selectIn == RQ_sel)? RQ:
35             (selectIn == R1_sel)? R1:
36             (selectIn == AC_sel)? AC:
37             {REG_WIDTH{1'b0}};
38
39 endmodule //multiplexer

```

### Testbench - multiplexer\_tb.v

```

1 `timescale 1ns/1ps
2 module multiplexer_tb();
3
4 localparam CLK_PERIOD = 10;
5
6 reg clk;
7 initial begin
8     clk = 0;
9     forever #(CLK_PERIOD/2) clk <= ~clk;
10 end
11
12 //control signals to select the bus input
13 localparam [3:0]
14     DMem_sel = 4'b0,
15     R_sel    = 4'd1,
16     IR_sel   = 4'd2,
17     RL_sel   = 4'd3,
18     RC_sel   = 4'd4,
19     RP_sel   = 4'd5,
20     RQ_sel   = 4'd6,
21     R1_sel   = 4'd7,
22     AC_sel   = 4'd8,
23     idle     = 4'd9;
24
25 localparam REG_WIDTH = 12;
26 localparam INS_WIDTH = 8;
27
28 reg [3:0] selectIn;
29 wire [REG_WIDTH-1:0] DMem, R, RL, RC, RP, RQ, R1, AC;

```

```

30 wire [INS_WIDTH-1:0] IR;
31 wire [REG_WIDTH-1:0] busOut;
32
33 assign DMem = 20;
34 assign R = 21;
35 assign RL = 22;
36 assign RC = 23;
37 assign RP = 24;
38 assign RQ = 25;
39 assign R1 = 26;
40 assign AC = 27;
41 assign IR = 28;
42
43 multiplexer #(REG_WIDTH, INS_WIDTH) mux(.selectIn(selectIn), .DMem(DMem),
44 .R(R), .RL(RL), .RC(RC), .RP(RP), .RQ(RQ), .R1(R1), .AC(AC),
45 .IR(IR), .busOut(busOut));
46 initial begin
47   @(posedge clk);
48   selectIn <= R_sel;
49   $display("selectIn = %d busOut = %d", selectIn, busOut);
50
51   @(posedge clk);
52   selectIn <= RL_sel;
53   $display("selectIn = %d busOut = %d", selectIn, busOut);
54
55   @(posedge clk);
56   selectIn <= RP_sel;
57   $display("selectIn = %d busOut = %d", selectIn, busOut);
58
59   @(posedge clk);
60   selectIn <= RQ_sel;
61   $display("selectIn = %d busOut = %d", selectIn, busOut);
62
63   @(posedge clk);
64   selectIn <= IR_sel;
65   $display("selectIn = %d busOut = %d", selectIn, busOut);
66
67   @(posedge clk);
68   repeat(10) begin
69     @(posedge clk);
70     selectIn <= $urandom()%10;
71     $display("selectIn = %d busOut = %d", selectIn, busOut);
72   end
73   $stop;
74 end
75
76 endmodule

```

### 9.7.10 Z register

#### Module - zReg.v

```

1 module zReg
2 #(parameter WIDTH = 12)
3 (
4   input [WIDTH-1:0] dataIn,
5   input clk, rstN, wrEn,
6   output Zout
7 );
8
9 reg value;
10
11 always @(posedge clk) begin
12   if (~rstN)
13     value <= 1'b0;
14   else if (wrEn)
15     if (dataIn == 1'b0)
16       value <= 1'b1;
17     else
18       value <= 1'b0;
19 end
20
21 assign Zout = value;
22
23 endmodule //zReg

```

## Testbench - zReg\_tb.v

```

1  `timescale 1ns/1ps
2  module zReg_tb();
3
4  localparam CLK_PERIOD = 20;
5
6  reg clk;
7  initial begin
8      clk <= 0;
9      forever begin
10         #(CLK_PERIOD/2);
11         clk <= ~clk;
12     end
13 end
14
15 localparam WIDTH = 12;
16 reg [WIDTH-1:0]dataIn;
17 reg rstN, wrEn;
18 wire Zout;
19
20 zReg #(WIDTH(WIDTH)) dut(.dataIn(dataIn), .clk(clk), .rstN(rstN), .wrEn(wrEn), .Zout(Zout));
21
22 initial begin
23     @(posedge clk);
24     #(CLK_PERIOD*4/5);
25     rstN <= 0;
26
27     @(posedge clk);
28     #(CLK_PERIOD*4/5);
29     rstN <= 1;
30     dataIn <= 0;
31     wrEn <= 0;
32
33     @(posedge clk);
34     #(CLK_PERIOD*4/5);
35     dataIn <= 0;
36     wrEn <= 1;
37
38     @(posedge clk);
39     #(CLK_PERIOD*4/5);
40     dataIn <= 4;
41     wrEn <= 1;
42
43     repeat(10) begin
44         @(posedge clk);
45         #(CLK_PERIOD*4/5);
46         dataIn = $random();
47         wrEn = $urandom();
48         rstN = $urandom();
49     end
50
51     $stop;
52 end
53
54 endmodule// zReg_tb

```

## 9.8 UART communication related Verilog modules

### 9.8.1 UART communication interface for the memory

#### Module - mem\_communication\_interface.v

```

1  module uart_mem_interface
2  #(
3      parameter MEM_WORD_LENGTH = 12,
4      parameter MEM_ADDR_LENGTH = 12,
5      parameter UART_WIDTH = 8
6  )
7  (
8      ////////////////////input output with main program
9      input clk,rstN,txStart,

```

```

10    output mem_transmitted, mem_received,
11
12    //select start end mem addresses of tx and rx
13    input [MEM_ADDR_LENGTH-1:0] tx_start_addr_in, tx_end_addr_in, rx_end_addr_in,
14    input toggle_addr_range, // 0 - go untill the last address of the memory,
15                                //1 - consider inputted start, end addresses
16
17    ////////////////////inputs outputs with memory
18    input [MEM_WORD_LENGTH-1:0] dataFromMem,
19    output memWrEn,
20    output [MEM_ADDR_LENGTH-1:0] mem_address,
21    output [MEM_WORD_LENGTH-1:0] dataToMem,
22
23    ////////////////////inputs outputs with uart system
24
25    input txByteReady, rxByteReady, new_rx_byte_indicate,
26    input [UART_WIDTH-1:0] ByteFromUart,
27    output uartTxStart,
28    output [UART_WIDTH-1:0] byteToUart
29 );
30
31 wire startTranmit, txReady, rxDone;
32 wire new_rx_data_indicate;
33
34 wire [MEM_ADDR_LENGTH-1:0] tx_start_addr, tx_end_addr, rx_end_addr;
35
36 localparam [2:0]
37     idle = 3'b0,
38     transmit_0 = 3'd1,
39     transmit_1 = 3'd2,
40     transmit_2 = 3'd3,
41     transmit_3 = 3'd4,
42     receive_0 = 3'd5,
43     receive_1 = 3'd6,
44     receive_2 = 3'd7;
45
46
47 reg [MEM_ADDR_LENGTH-1:0] currentAddress, nextAddress;
48 reg [2:0] currentState, nextState;
49 reg currentStartTransmit, nextStartTranmit;
50
51
52 always @(posedge clk or negedge rstN) begin
53     if (~rstN) begin
54         currentState <= idle;
55         currentAddress <= {MEM_ADDR_LENGTH{1'b0}};
56         currentStartTransmit <= 1'b1; //starts if value is zero
57     end
58     else begin
59         currentState <= nextState;
60         currentAddress <= nextAddress;
61         currentStartTransmit <= nextStartTranmit;
62     end
63 end
64
65 always @(*) begin
66     nextState = currentState;
67     nextAddress = currentAddress;
68     nextStartTranmit = currentStartTransmit;
69
70     case (currentState)
71         idle: begin
72             nextAddress = {MEM_ADDR_LENGTH{1'b0}};
73             nextStartTranmit = 1'b1; //to transmit this should be zero
74             if (new_rx_data_indicate)
75                 nextState = receive_0;
76             else if (~txStart) begin
77                 nextState = transmit_0;
78                 nextAddress = tx_start_addr;
79             end
80         end
81         //transmission process starts here
82         transmit_0: begin // to give the address to memory
83             nextState = transmit_1;
84         end
85
86         transmit_1: begin // extra delay for ip core memory
87             nextStartTranmit = 1'b0;
88             nextState = transmit_2;

```

```

89
90
91     end
92
93     transmit_2: begin          // start uart transmitter
94         nextStartTranmit = 1'b0;
95         nextState = transmit_3;
96     end
97
98     transmit_3: begin          // find end of the uart transmission
99         nextStartTranmit = 1'b1;
100        if (txReady == 1'b1) begin
101            if (currentAddress == tx_end_addr) begin
102                nextState = idle;
103            end
104            else begin
105                nextAddress = currentAddress + {{MEM_ADDR_LENGTH-1{1'b0}} , 1'b1};
106                nextState = transmit_0;
107            end
108        end
109    end
110
111    //receiving process starts here
112    receive_0: begin          // to find the end of the uart receiving
113        if (rxDone) begin
114            nextState = receive_1;
115        end
116    end
117
118    receive_1: begin // store in the memory (no need extra delay as it is explicitly given in receive_0)
119        nextState = receive_2;
120    end
121
122    receive_2: begin          //to find the end of the receiving process
123        if (currentAddress == rx_end_addr) begin
124            nextState = idle;
125        end
126        else begin
127            nextAddress = currentAddress + {{MEM_ADDR_LENGTH-1{1'b0}} , 1'b1};
128            nextState = receive_0;
129        end
130    end
131 endcase
132 end
133
134 assign startTranmit = currentStartTransmit;
135 assign memWrEn = (currentState == receive_1)? 1'b1: 1'b0;
136 assign mem_received = ((currentState == receive_2) && (currentAddress == rx_end_addr))? 1'b1: 1'b0;
137 assign mem_transmitted = ((currentState == transmit_3) && (txReady == 1'b1) && (currentAddress == tx_end_addr))?
138                                         1'b1: 1'b0;
139
140 assign mem_address = currentAddress;
141
142 uart_encoder_decoder #( .DATA_WIDTH(MEM_WORD_LENGTH), .UART_WIDTH(UART_WIDTH)) ED(.dataFromMem(dataFromMem),
143 .clk(clk), .rstN(rstN), .txStart(startTranmit), .txReady(txReady), .rxDone(rxDone),
144 .dataToMem(dataToMem), .new_rx_data_indicate(new_rx_data_indicate),
145 .txByteReady(txByteReady), .txByteStart(uartTxStart), .byteForTx(byteToUart),
146 .byteFromRx(ByteFromUart), .rxByteReady(rxByteReady),
147 .new_rx_byte_indicate(new_rx_byte_indicate));
148
149 assign tx_start_addr = (toggle_addr_range == 1'b0)? {MEM_ADDR_LENGTH{1'b0}}: tx_start_addr_in;
150 assign tx_end_addr = (toggle_addr_range == 1'b0)? {MEM_ADDR_LENGTH{1'b1}}: tx_end_addr_in;
151
152 assign rx_end_addr = (toggle_addr_range == 1'b0)? {MEM_ADDR_LENGTH{1'b1}}:
153                                         (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
154
155 endmodule //uart_mem_interface

```

## 9.8.2 Data word size handler between memory and UART system

### Module - data\_encoder\_decoder.v

```

1 module uart_encoder_decoder
2 #(
3     parameter DATA_WIDTH = 12, // width of memory word
4     parameter UART_WIDTH = 8

```

```

5 )
6 (
7     input [DATA_WIDTH-1:0] dataFromMem ,
8     input clk,rstN,txStart ,
9     output txReady,rxDone ,
10    output [DATA_WIDTH-1:0] dataToMem ,
11    output new_rx_data_indicate ,
12
13    //////////////////// inputs outputs for the UART system
14    input txByteReady ,
15    output txByteStart ,
16    output [UART_WIDTH-1:0] byteForTx ,
17
18    input [UART_WIDTH-1:0] byteFromRx ,
19    input rxByteReady ,new_rx_byte_indicate
20 );
21
22 localparam EXTRA = ((DATA_WIDTH % UART_WIDTH) == 0)?0:1;
23 localparam COUNT = (DATA_WIDTH/UART_WIDTH) + EXTRA;
24 localparam BUFFER_WIDTH = COUNT * UART_WIDTH;
25 localparam COUNTER_LENGTH = (COUNT == 1)? 1:$clog2(COUNT);
26
27 localparam [2:0]
28     idle = 3'd0 ,
29     transmit_1 = 3'd1 ,
30     transmit_2 = 3'd2 ,
31     receive_0 = 3'd3 ,
32     receive_1 = 3'd4 ,
33     receive_2 = 3'd5 ,
34     receive_3 = 3'd6 ;
35
36 reg [2:0] currentState , nextState ;
37 reg [BUFFER_WIDTH-1:0] currentTxBuffer , nextTxBuffer ;
38 reg [BUFFER_WIDTH-1:0] currentRxBuffer , nextRxBuffer ;
39 reg [COUNTER_LENGTH-1:0] currentTxCount , nextTxCount ;
40 reg [COUNTER_LENGTH-1:0] currentRxCount , nextRxCount ;
41
42 always @(posedge clk or negedge rstN) begin
43     if (!rstN) begin
44         currentTxBuffer <= {BUFFER_WIDTH{1'b0}} ;
45         currentRxBuffer <= {BUFFER_WIDTH{1'b0}} ;
46         currentTxCount <= {COUNTER_LENGTH{1'b0}} ;
47         currentRxCount <= {COUNTER_LENGTH{1'b0}} ;
48         currentState <= idle ;
49     end
50     else begin
51         currentTxBuffer <= nextTxBuffer ;
52         currentRxBuffer <= nextRxBuffer ;
53         currentTxCount <= nextTxCount ;
54         currentRxCount <= nextRxCount ;
55         currentState <= nextState ;
56     end
57 end
58
59 always @(*) begin
60     nextState = currentState ;
61     nextTxCount = currentTxCount ;
62     nextRxCount = currentRxCount ;
63     nextTxBuffer = currentTxBuffer ;
64     nextRxBuffer = currentRxBuffer ;
65
66     case (currentState)
67         idle: begin
68             nextTxCount = {COUNTER_LENGTH{1'b0}} ;
69             nextRxCount = {COUNTER_LENGTH{1'b0}} ;
70             if (new_rx_byte_indicate) begin //receiver indicates a arrival of new byte
71                 nextState = receive_1 ;
72                 nextRxBuffer = {BUFFER_WIDTH{1'b0}} ;
73             end
74             else if (txStart == 1'b0) begin
75                 nextState = transmit_1 ;
76                 nextTxBuffer = dataFromMem ;
77             end
78         end
79
80         transmit_1: begin //start byte transmission
81             nextState = transmit_2 ;
82         end
83

```

```

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
143     new data set
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 ,

```

```

9      output tx,txReady,rxReady,new_byte_indicate,
10     output [DATA_WIDTH-1:0]byteFromRx
11 );
12
13 wire baudTick;
14
15 uart_transmitter #(.DATA_WIDTH(DATA_WIDTH)) Tx(.dataIn(byteForTx), .clk(clk), .baudTick(baudTick),
16           .rstN(rstN), .txStart(txByteStart), .tx(tx), .TxReady(txReady));
17
18 uart_receiver #(.DATA_WIDTH(DATA_WIDTH)) RX(.rx(rx), .clk(clk), .rstN(rstN), .baudTick(baudTick),
19           .dataOut(byteFromRx), .ready(rxReady), .new_byte_indicate(new_byte_indicate));
20
21 uart_boudRateGen #(.BAUD_RATE(BAUD_RATE)) BRG(.clk(clk), .rstN(rstN), .baudTick(baudTick));
22
23
24 endmodule //uart_system

```

## 9.8.4 UART baud rate generator

### Module - uart\_baudRateGen.v

```

1 module uart_boudRateGen
2 #(
3     parameter BAUD_RATE = 19200
4 )
5 (
6     input clk,rstN,
7     output baudTick
8 );
9
10 localparam CLK_RATE = 50*(10**6);
11 localparam RESOLUTION = 16; // samples per 1 baud
12 localparam MAX_COUNT = (CLK_RATE/BAUD_RATE/RESOLUTION); //round to smaller integer
13 localparam WIDTH = $clog2(MAX_COUNT);
14
15 reg [WIDTH-1:0]count;
16
17 always @(posedge clk) begin
18     if (~rstN)
19         count <= 1'b0;
20     else if (count < MAX_COUNT)
21         count <= count + 1'b1;
22     else
23         count <= 1'b0;
24 end
25
26 assign baudTick = (count==MAX_COUNT)? 1'b1:1'b0;
27
28
29 endmodule //uart_boudRateGen

```

## 9.8.5 UART transmitter

### Module - uart\_transmitter.v

```

1 module uart_transmitter
2 #(
3     parameter DATA_WIDTH = 8
4 )
5 (
6     input [DATA_WIDTH-1:0]dataIn,
7     input clk, baudTick,rstN,txStart,
8     output tx,TxReady
9 );
10
11 localparam [1:0]
12     idle = 2'd0,
13     start = 2'd1,
14     data = 2'd2,
15     stop = 2'd3;
16

```

```
17 reg [1:0] currentState, nextState;
18 reg [DATA_WIDTH-1:0] currentData, nextData;
19 reg currentBit, nextBit;
20 reg [2:0] currentCount, nextCount;
21 reg [3:0] currentTick, nextTick;
22
23
24 always @(posedge clk or negedge rstN) begin
25   if (~rstN) begin
26     currentTick <= 4'b0;
27     currentCount <= 3'b0;
28     currentState <= idle;
29     currentData <= 0;
30     currentBit <= 1'b1;
31   end
32   else begin
33     currentTick <= nextTick;
34     currentCount <= nextCount;
35     currentState <= nextState;
36     currentData <= nextData;
37     currentBit <= nextBit;
38   end
39 end
40
41 always @(*) begin
42   nextTick = currentTick;
43   nextCount = currentCount;
44   nextState = currentState;
45   nextData = currentData;
46   nextBit = currentBit;
47
48   case (currentState)
49     idle: begin
50       if (~txStart) begin
51         nextState = start;
52         nextTick = 4'b0;
53         nextBit = 1'b0;
54         nextData = dataIn;
55       end
56       else
57         nextBit = 1'b1;
58     end
59
60     start: begin
61       if (baudTick) begin
62         nextTick = currentTick + 4'b1;
63         if (currentTick == 4'd15) begin
64           nextState = data;
65           nextCount = 3'b0;
66           nextBit = currentData[0];
67         end
68       end
69     end
70
71     data: begin
72       if (baudTick) begin
73         nextTick = currentTick + 4'b1;
74         if (currentTick == 15) begin
75           nextCount = currentCount + 3'b1;
76           nextBit = currentData[nextCount];
77           if (currentCount == 3'd7) begin
78             nextState = stop;
79             nextBit = 1'b1;
80           end
81         end
82       end
83     end
84
85     stop: begin
86       if (baudTick) begin
87         nextTick = currentTick + 4'b1;
88         if (currentTick == 4'd15)
89           nextState = idle;
90       end
91     end
92
93   endcase
94 end
95
```

```

96 assign tx = currentBit;
97 assign TxReady = (currentState == idle)? 1'b1:1'b0;
98
99 endmodule //uart_transmitter

```

## 9.8.6 UART receiver

### Module - uart\_receiver.v

```

1 module uart_receiver
2 #(
3     parameter DATA_WIDTH = 8
4 )
5 (
6     input rx, clk, rstN, baudTick,
7     output ready,
8     output [DATA_WIDTH-1:0] dataOut,
9     output new_byte_indicate
10 );
11
12 localparam [1:0]
13     idle = 2'b0,
14     start = 2'b1,
15     data = 2'd2,
16     stop = 2'd3;
17
18 reg [1:0] currentState, nextState;
19 reg [2:0] currentCount, nextCount;
20 reg [3:0] currentTick, nextTick;
21 reg [DATA_WIDTH-1:0] currentData, nextData;
22
23 always @ (posedge clk or negedge rstN) begin
24     if (~rstN) begin
25         currentState <= idle;
26         currentCount <= 3'b0;
27         currentTick <= 4'b0;
28         currentData <= 0;
29     end
30     else begin
31         currentState <= nextState;
32         currentCount <= nextCount;
33         currentTick <= nextTick;
34         currentData <= nextData;
35     end
36 end
37
38 always @(*) begin
39     nextState = currentState;
40     nextCount = currentCount;
41     nextTick = currentTick;
42     nextData = currentData;
43
44     case (currentState)
45         idle: begin
46             nextTick = 4'b0;
47             nextCount = 3'b0;
48             if (rx == 1'b0) begin
49                 nextState = start;
50             end
51         end
52
53         start: begin
54             if (baudTick) begin
55                 nextTick = currentTick + 4'b1;
56                 if (currentTick == 4'd7) begin
57                     if (~rx) begin
58                         nextState = data;
59                         nextCount = 3'b0;
60                         nextTick = 4'b0;
61                         nextData = 8'b0;
62                     end
63                     else begin
64                         nextState = idle;
65                     end
66                 end
67             end
68         end
69     endcase
70 end
71
72 endmodule

```

```

67
68         end
69     end
70 end
71
72 data: begin
73     if (baudTick) begin
74         nextTick = currentTick + 4'b1;
75         if (currentTick == 4'd15) begin
76             nextData[currentCount] = rx;
77             nextCount = currentCount + 3'b1;
78             if (currentCount == 3'd7) begin
79                 nextState = stop;
80             end
81         end
82     end
83 end
84
85 stop: begin
86     if (baudTick) begin
87         nextTick = currentTick + 4'b1;
88         if (currentTick == 4'd15) begin
89             nextState = idle;
90         end
91     end
92 end
93
94 endcase
95 end
96
97 assign dataOut = currentData;
98 assign ready = (currentState == idle)? 1'b1: 1'b0;
99 assign new_byte_indicate = ((currentState == start) && (baudTick) && (currentTick == 4'd7) && (~rx))?
100                                         1'b1:1'b0; //start of new data byte
101
102 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

```

1 module timeCounter (
2     input clk, rstN, startN, stop,
3     output [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 or negedge rstN) 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 @(*) begin
24     nextTime = currentTime;
25     nextState = currentState;
26
27     case (currentState)
28         idle: begin
29             nextTime = 26'b0;
30             if (~startN) begin
31                 nextState = counting;
32             end
33         end
34     endcase
35 end

```

```

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

```

## 9.9.2 Display on 8 seven segments on the FPGA board

### Module - hex\_display.v

```

1 module hex_display (
2     input clk, rstN,
3     input [2:0]state,
4     input start_timeValue_convetion,
5     input [25:0]timeValue,
6     output [6:0]out0, out1,out2,out3,out4,out5,out6,out7
7 );
8
9 localparam //lettes needed to be shown on seven segments
10    a = 5'd10,
11    b = 5'd11,
12    c = 5'd12,
13    d = 5'd13,
14    e = 5'd14,
15    f = 5'd15,
16    i = 5'd18,
17    n = 5'd19,
18    o = 5'd20,
19    p = 5'd21,
20    r = 5'd22,
21    s = 5'd23,
22    t = 5'd24,
23    u = 5'd25,
24    y = 5'd26,
25    off = 5'd27;
26
27 localparam uart_ready = 3'd0,
28     uart_receive_Imem = 3'd1,
29     uart_receive_dmem = 3'd2,
30     process_ready = 3'd3,
31     process_exicute = 3'd4,
32     uart_transmit_dmem = 3'd5,
33     finish = 3'd6;
34
35 reg [2:0]currentState, nextState;
36
37 wire done1;
38 wire [4:0]x0,x1,x2,x3,x4,x5,x6,x7;
39 wire [3:0]y0,y1,y2,y3,y4,y5,y6,y7;
40
41 assign {x7,x6,x5,x4,x3,x2,x1,x0} =
42     (state == uart_ready)? {off,off,off,r,e,a,d,y}:
43     (state == uart_receive_Imem)? {u,a,r,t,off,i,n,s}:
44     (state == uart_receive_dmem)? {u,a,r,t,d,a,t,a}:
45     (state == process_exicute)? {off,p,r,o,c,e,s,s}:
46     (state == uart_transmit_dmem)? {u,a,r,t,off,a,n,s}:
47     (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}:
48     {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_convention), .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'b00110000;
31         5'd4:out = 7'b0011001;
32         5'd5:out = 7'b0010010;
33         5'd6:out = 7'b00000010;
34         5'd7:out = 7'b1111000;
35         5'd8:out = 7'b00000000;
36         5'd9:out = 7'b00011000;
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;

```

```

43      i  :out = 7'b1110000;
44      n  :out = 7'b0001011;
45      o  :out = 7'b1000000;
46      p  :out = 7'b0001100;
47      r  :out = 7'b1001110;
48      s  :out = 7'b0010010;
49      t  :out = 7'b0000111;
50      u  :out = 7'b1000001;
51      y  :out = 7'b0010001;
52      off :out = 7'b1111111;
53
54      default: out = 7'b1111111;
55  endcase
56 end
57
58
59 endmodule // SSeg

```

## Module - binaryToBCD.v

```

1 module binaryToBCD(
2     input [25:0]binaryValue,// to get 8 bit value 26.57 (26) bit value needed
3     input clk,rstN,start,
4     output reg ready, done,
5     output reg [3:0]digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0
6 );
7
8 localparam [1:0]
9     idle = 2'b00,
10    subtract = 2'b01,
11    shift = 2'b10,
12    over = 2'b11;
13
14 reg [4:0]currentCount,nextCount;           //count upto 26
15 reg [31:0]currentValue,nextValue;
16 reg [1:0]currentState,nextState;
17 reg [25:0]currentInput,nextInput;
18
19 always @ (posedge clk, negedge rstN) begin
20     if (~rstN) begin
21         currentCount <= 5'b0;
22         currentValue <= 32'b0;
23         currentState <= idle;
24         currentInput <= 26'b0;
25     end else begin
26         currentValue <= nextValue;
27         currentCount <= nextCount;
28         currentState <= nextState;
29         currentInput <= nextInput;
30     end
31 end
32
33 always @(*) begin
34     nextCount = currentCount;
35     nextValue = currentValue;
36     nextState = currentState;
37     nextInput = currentInput;
38     {digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0} = 32'b0;
39     ready = 1'b0;
40     done = 1'b1;      // actives the BCDtoHEX by the negedge
41
42     case (currentState)
43         idle: begin
44             {digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0} = currentValue;
45             ready = 1'b1;
46             if (!start) begin      // KEY is push button stay at 1 normally
47                 nextState = subtract;
48                 ready = 1'b0;
49                 nextCount = 5'b0;
50                 nextInput = binaryValue;
51             nextValue = 32'b0;
52             end
53         end
54
55         subtract: begin
56             if (currentValue[3:0]>4)
57                 nextValue[3:0] = currentValue[3:0]+4'b0011;
58             if (currentValue[7:4]>4)

```

```

59         nextValue[7:4] = currentValue[7:4]+4'b0011;
60     if (currentValue[11:8]>4)
61         nextValue[11:8] = currentValue[11:8]+4'b0011;
62     if (currentValue[15:12]>4)
63         nextValue[15:12] = currentValue[15:12]+4'b0011;
64     if (currentValue[19:16]>4)
65         nextValue[19:16] = currentValue[19:16]+4'b0011;
66     if (currentValue[23:20]>4)
67         nextValue[23:20] = currentValue[23:20]+4'b0011;
68     if (currentValue[27:24]>4)
69         nextValue[27:24] = currentValue[27:24]+4'b0011;
70     if (currentValue[31:28]>4)
71         nextValue[31:28] = currentValue[31:28]+4'b0011;
72
73     nextState = shift;
74 end
75
76 shift: begin
77     nextInput[25:1] = currentInput[24:0];
78     nextValue = {currentValue[30:0],currentInput[25]};
79     nextCount = currentCount+5'b1;
80
81     if (currentCount<5'd25)
82         nextState = subtract;
83     else
84         nextState = over;
85 end
86
87 over: begin
88     nextState = idle;
89     done = 1'b0;
90     {digit7,digit6,digit5,digit4,digit3,digit2,digit1,digit0} = currentValue;
91 end
92 endcase
93 end
94 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

```

1 `timescale 1ns/1ps
2
3 module simulation_top_tb();
4
5 localparam CLK_PERIOD = 20;
6
7 reg clk;
8 initial begin
9     clk <= 0;
10    forever begin
11        #(CLK_PERIOD/2);
12        clk <= ~clk;
13    end
14 end
15
16 localparam CORE_COUNT = 2;
17
18 reg rstN, startN;
19 wire processor_ready, processDone;
20
21 simulation_top #(CORE_COUNT(CORE_COUNT)) simulation_top(.clk(clk), .rstN(rstN), .startN(startN),
22 .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_execute;
63             end
64         end
65
66         process_execute: begin // processor execute program (matrix multiplication)
67             if (processDone) begin
68                 nextState = finish;
69             end
70         end
71
72         finish: begin //End of the process
73
74     end
75
76     default : nextState = idle;
77
78 endcase
79
80 end
81
82 assign processStartN = ((currentState == idle) && (~startN))? 1'b0: 1'b1;
83
84 assign dataMemWrEn = (currentState == process_execute)? processor_DataMemWrEn : 1'b0;
85
86 assign dataMemAddr = (currentState == process_execute)? processor_dataMemAddr: {DATA_MEM_ADDR_WIDTH{1'b0}};
87
88 assign DataMemIn = (currentState == process_execute)? processor_DataOut: {DATA_MEM_WIDTH{1'b0}};
89
90 assign insMemAddr = (currentState == process_execute)? 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(
98                         processor_InsMemAddr),
99                         .dataMemAddr(processor_dataMemAddr), .DataMemWrEn(processor_DataMemWrEn), .done(
100                         processDone),
101                         .ready(processor_ready));
102
103 INS_RAM #(.WIDTH(INS_WIDTH), .DEPTH(INS_MEM_DEPTH), .mem_init(1))
104     IM(.clk(clk), .wrEn(1'b0), .dataIn(InsMemIn), .addr(insMemAddr), .dataOut(InsMemOut));
105
106 DATA_RAM #(.WIDTH(DATA_MEM_WIDTH), .DEPTH(DATA_MEM_DEPTH), .mem_init(1)) DM
107     (.clk(clk), .wrEn(dataMemWrEn), .dataIn(DataMemIn), .addr(dataMemAddr), .dataOut(DataMemOut),
108     .processDone(processDone));
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 //////////////////////////////////////////////////////////////////
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 cycle.
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 //////////////////////////////////////////////////////////////////
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 cycle.
41   end
42 end
43 assign dataOut = memory [addr_reg]; // address is registered. Need 2 clk cycles to read.
44
45 endmodule // DATA_RAM

```