Department of Electronic & Telecommunication Engineering University of Moratuwa

EN3030 - Circuits and Systems Design

Multi-Core Processor Design for Matrix Multiplication using FPGA

Group 26

Authors:
H.U.D.B. Haputhanthri
D.B. Ishtaweera
S.N. Liyanagoonawardena
R.M.S.N.Rajapakshe

Index Number:
170208K
170242H
170348M
170476D

This report is submitted as a partial fulfillment for the EN3030 - Circuits and Systems Design
December 29, 2021

Contents

List of Figures

List of Tables

1 Introduction

1.1 Processor Design & Central Processing Unit (CPU)

The goal of this project is to implement a simple processor that is capable of handling single instruction and multiple data processing in order to compute and output the results of a matrix multiplication. The processor is specifically designed and optimized to do matrix multiplication of square matrices of $n \times n$. However, the proposed solution also facilitates the multiplication of matrices with different shapes.

The proposed solution is implemented in Verilog Hardware Description Language using the QuartusLite 18.1.0.625.

The CPU can be seen as the brain of the computer that has the ability to retrieve and execute instructions. This processing is done using Control Unit (CU) and the Arithmetic and Logic Unit (ALU) where the CU retrieves the instructions in the correct order and interprets them while performing the logical tasks where the ALU conducts the necessary mathematical operations [1]. CPU consists of different types of registers and is directly associated with the main memory, input and output externally.

1.2 Microprocessor

The microprocessor is a small electronic device that directs current through the desired paths to achieve the specific outputs. While it shares many of its functions with the CPU, it includes additional functionalities using other processing units. These control and data processing logic are integrated in a single or lesser number of integrated circuits (ICs). While microprocessor contains both combinational and sequential digital logic it conducts its operations using binary number system representations.

1.3 Problem Statement

Matrix multiplication is an essential function used in the linear algebra and is frequently used in applications of statistics, applied mathematics, physics, engineering and economics [2].

This project aims to implement a multi-core processor which multiply two $p \times q$ and $q \times r$ matrices and obtain the results in the form of a $p \times r$ matrix. It is specifically designed and optimized for square matrix multiplication with shape $(n \times n)$ where $n = p = q = r$.

1.4 General Overview of the Proposed Solution

The proposed solution facilitates the multiplication of two $p \times q$ and $q \times r$ matrices using a number of cores as defined by the user as per his requirement.

Initially, following the input of the two matrices, along with the number of cores and matrix sizes, the implemented Python code will rearrange the matrices to be stored appropriately in the memory as per the proposed memory allocation defined in the algorithm. Next, the Verilog testbench will read the rearranged data which will be stored in the Data Memory (DM). Following the storage, the cores will initialize the operation and complete the multiplication of the two input matrices. After this realization, the Verilog testbench will write the results of the process stored in the DM to an external text file. Finally, the implemented Python code will rearrange the data stored in that text file in a more user readable format and outputs the results of the matrix multiplication while informing the latency of the computation. This is achieved after comparing the obtained results with expected outcome calculated from a matrix multiplication in Python.

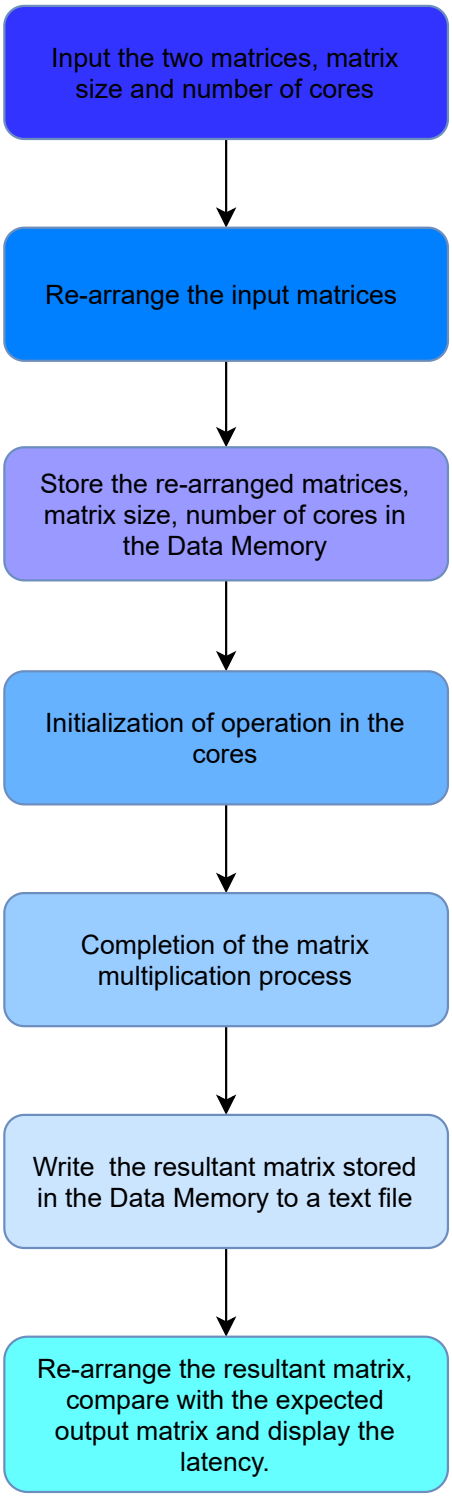

Figure 1: Flow of the general overview of the proposed solution.

2 Processor Design

2.1 Field Programmable Gate Array (FPGA) Overview

FPGA can be seen as a semiconductor Integrated Circuit (IC) which has mutable electrical functionalities. The programmable connectivity between such millions of ICs can be implemented by the user according to the desired tasks. FPGAs provide several benefits such as; flexibility, acceleration of system performance and integration of multiple functions leading to comparatively fewer device failures due to less number of external devices [3]. At present, FPGAs are utilized in numerous field such as; ASIC prototyping, aerospace and defense, automotive, consumer electronics, industrial, medical, video & image processing and wired and wireless communications [4].

2.2 Instruction Set Architecture (ISA)

ISA can be seen as the programmers' view of the microprocessor which includes the information required to interact with it such as the information needed by the programmer to write a program for the microprocessor, the specifications needed by the compiler to compile the programs written in high level language.

When designing an ISA, it is crucial to focus on few important factors such as [5];

- The expected operation of the ISA and if the designed Instruction set includes all the instructions required to perform that task.
- The orthogonality of the instructions in order to minimize the overlapping which can lead to inefficient resource allocation and thus, higher cost.
- Trade offs that are related to performance, size and cost. Here it is important to note that, few registers would lead to more accessing of the memory which in turn increases the time consumption and decreases the performance but more registers would lead to a higher cost.
- The backward compatibility if a second design is expected to be released.
- Floating point data processing
- The requirement of interruptions
- The requirement of conditional instructions

2.2.1 Data Path

Figure 2 illustrates the optimized final data path diagram of the proposed solution.

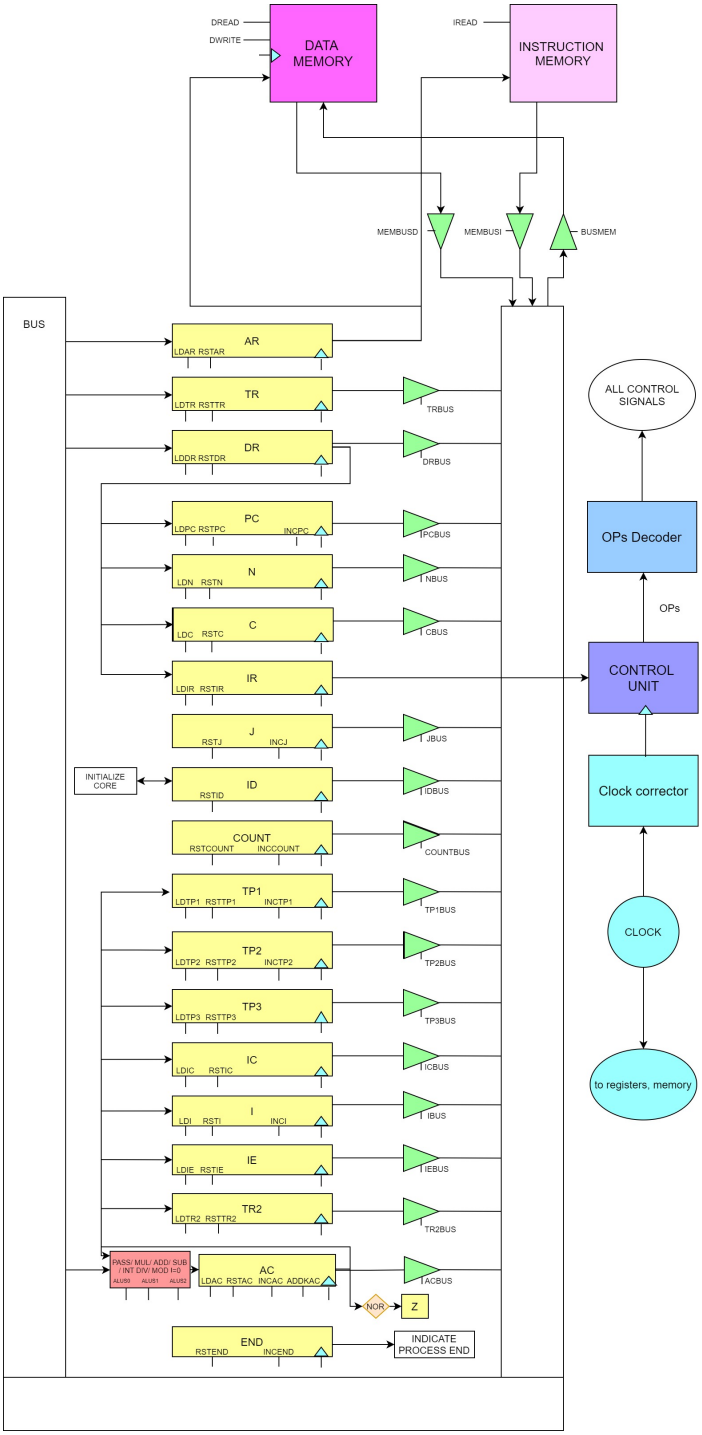

Figure 2: Data path of the proposed solution.

2.2.2 Instructions

Considering the task is not general purpose but specific for matrix multiplication, we have mainly focused on developing Instruction Set which can allocate resources much faster with minimal number of clock cycles in order to do the calculations need to do the matrix multiplication. Maintaining orthogonality was also focused in order to reduce the overlapping of instructions.

Table 1 summarizes all the instructions, their opcodes and the operations used in the proposed project.

Table 1: Instruction Set

Instruction	Opcode	Operation
SETN	0	$n \leftarrow M[\alpha_n]$
SETC	1	$c \leftarrow M[\alpha_c]$
SETTP1	2	$TP1 \leftarrow n * i$
SETTP2	3	$TP2 \leftarrow n * j + k$
SETTP3	4	$TP3 \leftarrow (n * i + j) + 2k$ and reset *count*
RNGI	5	$i_{count} = n//c + n\%c! = 0$ and reset $TR2$
STRTI	6	$i_{start} = i_{count} \times core\_id$
ENDI	7	$i_{end} = i_{count} \times (core\_id + 1)$
LDACTP1	8	$AC \leftarrow M[TP1]$ and increment *count*
LDACTP2	9	$AC \leftarrow M[TP2]$
LDACTP3	10	$AC \leftarrow M[TP3]$
LDN	11	$AC \leftarrow n$
MVACTR	12	$AC \leftarrow TR$
MVCOUNT	13	$TR \leftarrow count$, increment $TP1$, increment $TP2$
MVJ	14	$TR \leftarrow j$
MVI	15	$TR \leftarrow i$
MVIE	16	$AC \leftarrow i_{end}$ and increment i
MUL	17	$AC \leftarrow AC * TR$
ADD	18	$AC \leftarrow AC + TR$
SUB	19	$AC \leftarrow AC - TR$ if AC=0 then Z=1 else Z=0
STAC	20	$M[TP3] \leftarrow AC$, reset $TR2$, increment j
JNPZ	21	if Z=0 then jump to α
JNPZY	22	jump to α
JNPZN	23	continue without jump
RSTJ	24	$j \leftarrow 0$
END	25	$END \leftarrow 1$
MVTR2	26	$TR \leftarrow TR2$
MVACTR2	27	$TR2 \leftarrow AC$
LDTR2	28	$AC \leftarrow TR2$

2.2.3 Micro-instructions

Table 2 presents the micro-instructions for each instruction in the proposed solution.

Table 2: Micro-instructions

Instruction	Micro-instruction
FETCH	ARPC DRMI, PCINC IRDR, ARPC
SETN	DRMI, PCINC ARDR DRMD NDR
SETC	DRMI, PCINC ARDR DRMD CDR
SETTP1	ACN TRI MUL TP1AC
SETTP2	ACN TRJ MUL ADDK TP2AC
SETTP3	ACN TRI MUL TRJ ADD ADDK ADDK TP3AC, COUNTZ0
RNGI	ACN TRC INTDIV TR2AC ACN ISMOD TRTR2 ADD ICAC, TR2Z0
STRTI	TRIC ACID

Instruction	Micro-instruction
	MUL IAC
ENDI	ACID ACINC, TRIC MUL IEAC
LDACTP1	ARTP1,COUNTINC DRMD ACDR
LDACTP2	ARTP2 DRMD ACDR
LDACTP3	ARTP3 DRMD ACDR
LDN	ACN
MVACTR	TRAC
MVCOUNT	TRCOUNT, TP1INC, TP2INC
MVJ	TRJ
MVI	TRI
MVIE	ACIE, IINC
MUL	MUL
ADD	ADD
SUB	SUB
STAC	ARTP3, TR2Z0 DRAC MDDR, JINC
JNPZ	
JNPZY	DRMI PCDR
JNPZN	PCINC
RSTJ	JZ0
END	ENDINC
MVTR2	TRTR2
MVACTR2	TR2AC
LDTR2	ACTR2

Figure 3 depicts the state machine of the proposed solution.

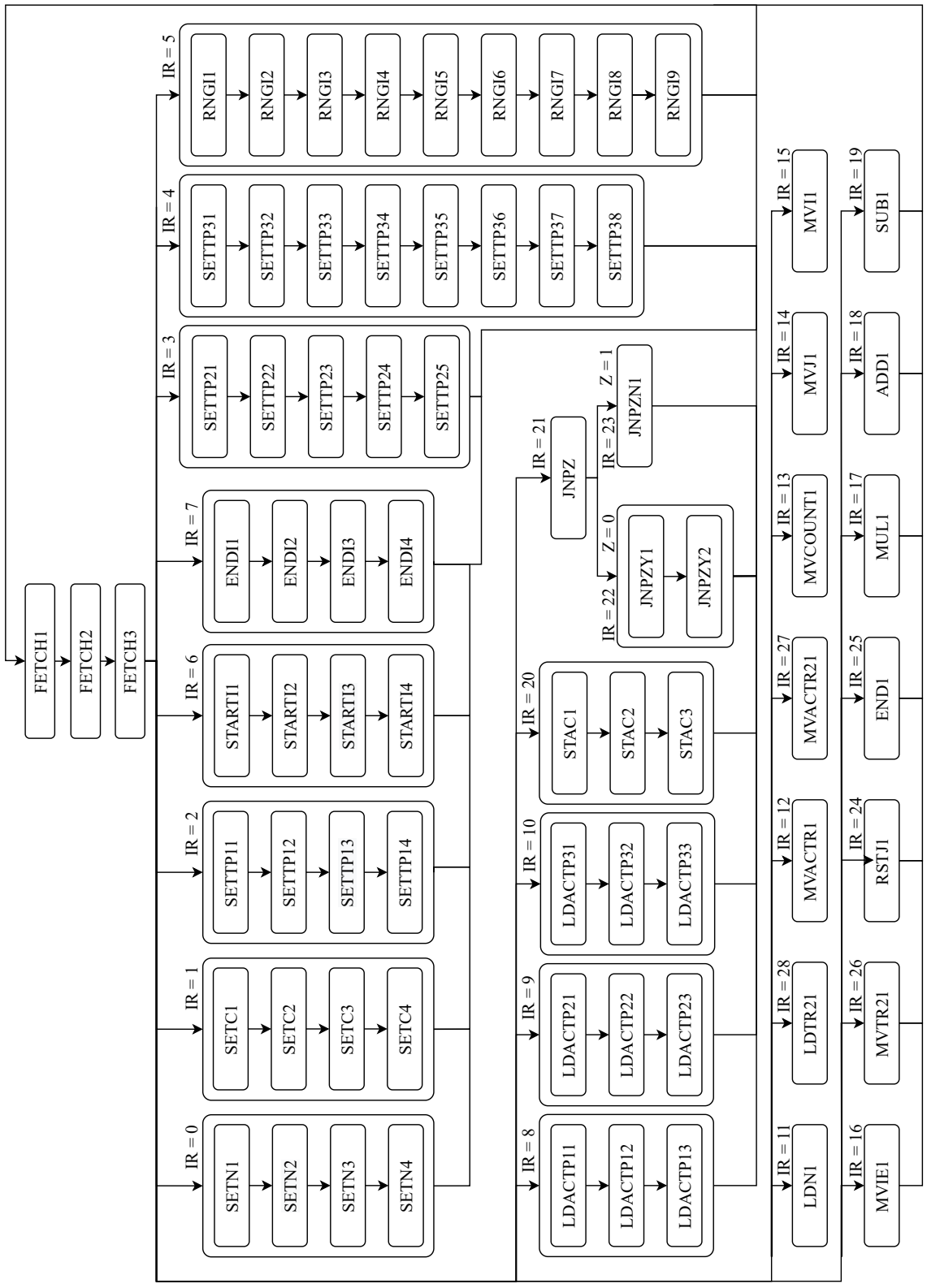

Figure 3: State machine

2.3 Components & Modules

2.3.1 Processor

The processor contains modules and registers used to conduct all processing operations and the controlling and managing of the related instructions. However memory related modules such as Data Memory (DM) and Instruction memory (IM) are situated outside the processor. The inputs and outputs for the processor are as shown in the Table 3;

Table 3: Inputs and Outputs of the processor

Input	DM out	Data coming from the data memory
	IM out	Instructions coming from the Instruction memory
	Start	Signal given to indicate the start of the process
	Clock	Clock pulse given for synchronization
	Reset	Signal given to clear all the registers except register in Control unit
Output	DM in	Data recieved to Data memory
	AR out	Output from the Address Register (AR) goes to DM and IM to point memory locations.
	Read MI	Read Signal for reading Instruction Memory
	Write MD	Write signal for writing to Data Memory
	Read MD	Read Signal for reading Data the memory

The processor consists of several main modules listed below;

- CU - Is responsible for managing and controlling all the processes in the processor.
- ALU - Is responsible for conducting all arithmetic and logical operations.
- AC - A special purpose register directly connected to the ALU and is used to store ALU outputs immediately.
- AR - A special purpose register to point to the address of the data/instruction to be read.
- TR - A general purpose register for temporary storage which is utilized for computations in ALU.
- DR - A special purpose register to store both instruction and data which will be written to/ read from instruction/ data memories.
- PC - A special purpose register to point to the next instruction to be fetched.
- N - A special purpose register to store the dimensions of the input matrices (n).
- C - A special purpose register to store the number of cores (c).

- IR - A special purpose register to store current instruction.
- I - A special purpose register to store the row index of the position of output matrix which is currently being computed (More details can be found in Fig. 17, Algorithm 1).
- J - A special purpose register to store the column index of the position of the output matrix which is currently being computed (More details can be found in Fig. 17, Algorithm 1).
- ID - A special purpose register to store the core identity ($core\_id$).
- COUNT - A special purpose register to store the the variable *count* which indicates the current position of row-I in first matrix and column-J in second matrix (More details can be found in Fig. 17, Algorithm 1).
- TP1 - A special purpose register to store the memory location of the first matrix which is being currently computed.
- TP2 - A special purpose register to store the memory location of the second matrix which is being currently computed.
- TP3 - A special purpose register to store the memory location of the output matrix which will be computed next.
- IC - A special purpose register to indicate the number of rows of output matrix which should be calculated by a particular core (i_c)
- IE - A special purpose register to indicate the last row of the output matrix which should be calculated by a particular core (i_{end})
- TR2 - A general purpose register to store the intermediate values of computations done by ALU
- END - A special purpose register to indicate the end of matrix multiplication process of a particular core.

2.3.2 Arithmetic and Logic Unit (ALU)

Figure 4 illustrates the Arithmetic and Logic Unit and Accumulator used in the proposed solution.

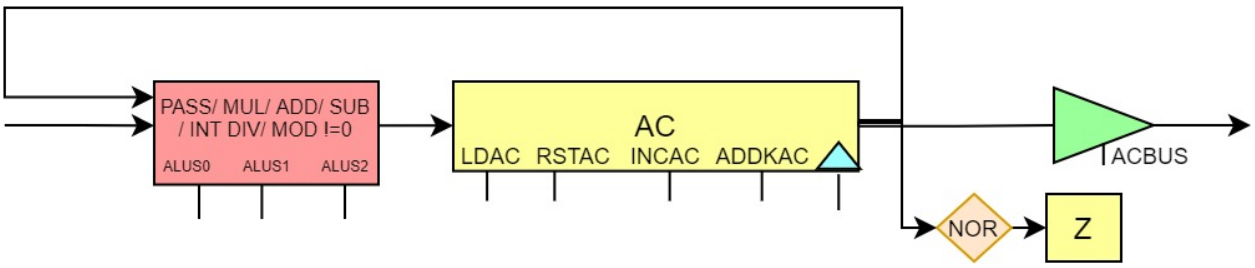

Figure 4: Arithmetic and Logic Unit and Accumulator

ALU is responsible for conduction of all arithmetic and logic functions in the processor. As the goal of this project is to multiply two matrices, ALU plays a vital role in the proposed solution.

ALU is located inside the processor where it takes 3 main inputs;

- in-bus - 16 bit input from the bus
- in-AC - 16 bit input from the AC
- Operation - 3 bit control signal from the OPs Decoder (which will be explained in detailed in the section 2.3.8) to select the required specific function. The utilization of 3 bits is to facilitate the 6 different functions of the proposed ALU.

ALU is directly connected with the AC register and AC is connected to Z register through a NOR gate and to the bus via a buffer.

The value of Z will be utilized in the Algorithm 1 of the proposed solution and explained in the section 2.3.7.

In the proposed solution, ALU conducts 6 main functions which are mentioned in Table 4;

Table 4: Functions of the ALU

Function	Description
Mul	data-out = in-AC * in-bus;
Add	data-out = in-AC + in-bus;
Sub	data-out = in-AC - in-bus;
Int-div	data-out= (in-AC - in-AC % in-bus)/in-bus
Is-mod	data-out= ((in-AC % in-bus) != 0)
Pass	data-out = in-bus;

2.3.3 Register

When designing the ISA, we have defined the registers in a way that it would bring maximum efficiency to the desired task and that it would need minimum possible references to main memory in between operations in order to avoid the inefficient time consumption.

The registers utilized in the proposed solution and their sizes are shown in the table 5 .

Table 5: Registers and their sizes

Register	**Size (bits)**
AR	16
TR	16
TR2	16
DR	16
IR	16
PC	16
AC	16
N	16
C	16
J	16
ID	16
COUNT	16
TP1	16
TP2	16
TP3	16
IC	16
I	16
IE	16

Registers in general can mainly be divided into two groups that are registers with and without increment. The importance of those are explained below.

Registers without increment - Figure 5 shows an example for registers without increment used in the proposed solution.

Figure 5: An example for registers without increment

The purpose of these registers is to temporarily store data during each processing tasks for a short period of time. These registers are not given an increment control signal and thus any increment related to these must be done by passing the register data through the ALU and feeding it back to the same register through the bus.

They consists of following control signals;

- Load enable - load the data to register
- Reset - Reset the data of register to 0

The data stored in these registers are always available at the input of buffer as shown in Figure 5 and will fed into the bus whenever a buffer control signal is issued. In the proposed solution the registers can be written in each positive edge if the load enable control signal specific to each register is given.

Registers with increment Figure 6- shows an example for registers with increment used in the proposed solution.

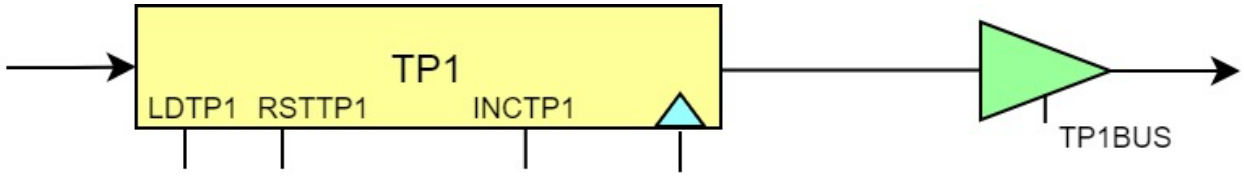

Figure 6: An example for registers with increment

While these registers are similar to registers without increment they have the capability to increment its value by 1 without passing the values through the ALU. This is especially useful in tracking the interactions in a loop while maintaining efficiency. The registers TP1, TP2, AC, J, I, COUNT are such examples.

Apart from that, we have utilized registers with several other types of control signals which helps to improve the performance of the proposed hardware solution. All the registers consists of one or more following control signals (More details can be found in Fig. 2);

- Load enable - Load the data to register
- Reset - Reset the data of register to 0
- Increment - Increment the data in register by 1
- Increment-K - Increment the data in register by K. This is a special type of register only used for AC to reduce the number of clock cycles.

2.3.4 Accumulator (AC)

The Accumulator is a register with increment, which has a unique combination of connections where it receives its input from the ALU and sends its output to both back to the ALU and its connected buffer, after conducting the necessary arithmetic and logical operations.

AC also passes its output into the Z flag through a NOT gate. This mainly utilized to find the end of a loop by comparing the current position with the maximum length of the particular loop. (More details can be found in the section (1) where the ALU output will be equal to 0 which will in turn make the Z flag 1.

It is also given the previously mentioned control signals such as LDAC, RSTAC and INCAC in addition to the ADDKAC which increment the data in the register by a value K.

2.3.5 Buffer

Buffer is a single input single output unit which passes the input with/without the enabling of the buffer as per its implementation. This is used select the which register output should be passed to bus. In the proposed solution the buffer is mainly used to control the flow of 16 bits input to the bus from the registers, Data Memory and the Instruction Memory. These buffers are enabled using control signals issued from the OPs Decoder as explained below.

2.3.6 Bus

Bus refers to the set of wires connecting all registers and modules within the processor except the CU. It transfers data to and from all other modules inside the processor and can even exchange data with the Instruction Memory (IM) and Data Memory (DM).

In the proposed solution the LDx (load enable of x) control signal given to registers allows the data in the bus to be written into the registers and the control signal xBUS given to the Buffers allows the data sent out from the registers to pass through the buffer and to be fed into the bus.

2.3.7 Control Unit (CU)

Figure 7 shows the CU, OPs Decoder, Clock Corrector and Clock of the proposed solution.

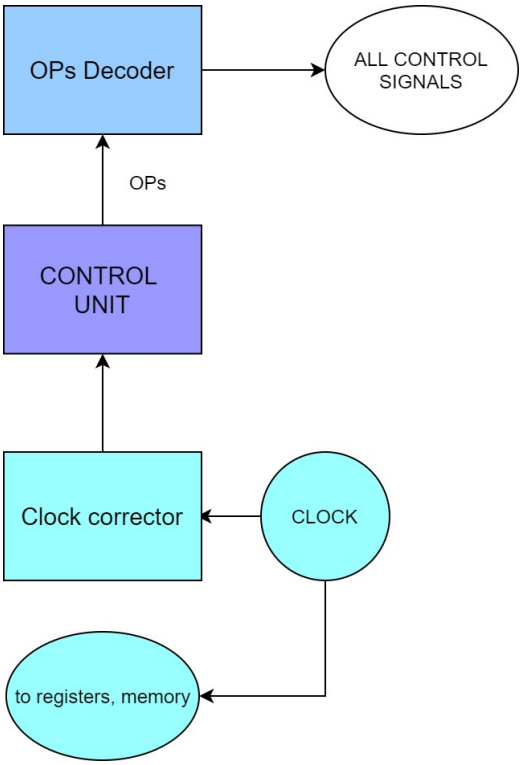

Figure 7: CU, OPs Decoder, Clock Corrector and Clock

CU generates the control signals and handles all the operations within the processor and their respective instructions.The main source of input for the CU is the IR, in addition to the corrected clock signal, where it fetches the respective instructions in addition to the Z flag. Finally, it outputs the command signals(OPs) in the negative edge of the clock. This implementation is influenced from [5].

After obtaining the appropriate instruction, it decodes them into commands(OPs), conditions for the multiplexer within the CU, BT and the next Address. Finally, it outputs the uOPS into the OPs Decoder unit to issue the respective control signals to perform ALU operations, load registers, write register content into the bus, memory read and memory write.

The Z flag denotes the status of the ALU output where Z=1 in case where the ALU output equals zero. Inside the CU, after selecting the relevant micro instruction from the microcode memory, CU issues another 2 bit conditional signal and a 1 bit BT signal which will use to get the output from the two multiplexers inside the CU. One multiplexer will use those values to select 1, Z, or Z' as its input and the other one will use it to select the correct addresses to be use in the next step. This can either lead to the incremented next location of the micro code memory (current address +1), to the address from the microcode memory (address specified by ADDR) or to a new instruction address (MAP address) from the IR.

Overall flow of the CU is shown in Fig. 8.

Note that, the clock received for the CU is returned from Clock Corrector module explained below. The release of uOPs from control unit is done at negative clock edge cycles to avoid the conflicts with operations in the other modules.

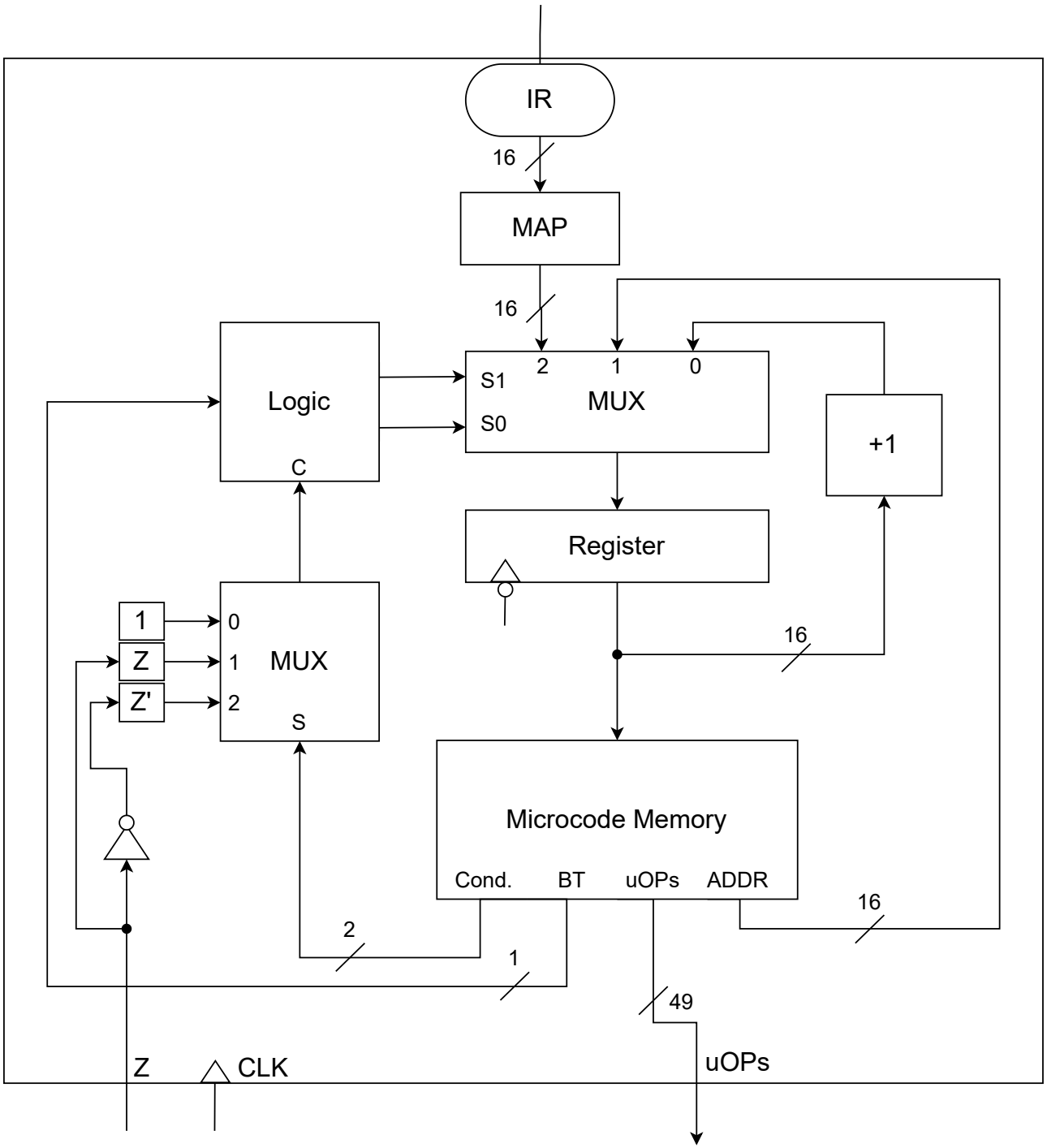

Figure 8: Implementation of Control Unit (Logic can be found in Fig. 9)

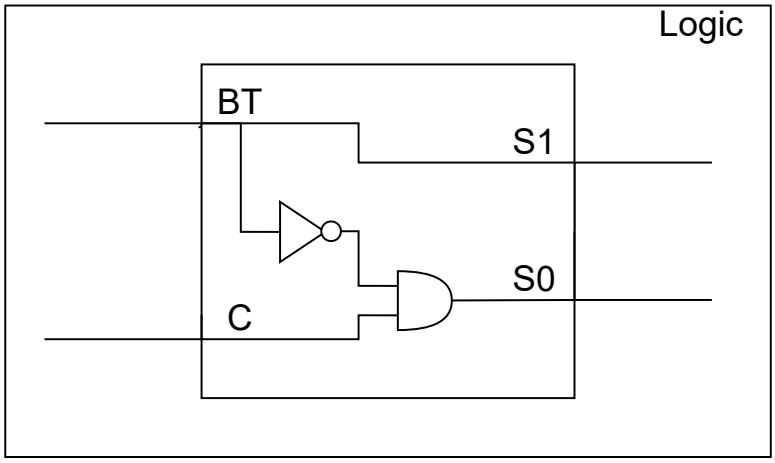

Figure 9: Logic block in Control Unit in Fig. 8

Mapping Logic in Control Unit In order to obtain the corresponding starting microcode memory address given a particular instruction, mapping logic is introduced. This is implemented as a look-up table which contains the mapping between IR and MAP address. Implementation details can be found in Appendix (Fig. 35, 36).

2.3.8 OPs Decoder

This unit generates the control signals using vertical microcodes methods [5]. The implementation of a OPs Decoder instead of using horizontal microcodes method is due to;

Horizontal microcodes consists majorly of zeros. Thus, more than 85 percent bits are inactive. However, vertical microcode reduces this number of bits.

These control signals are used to manage several processes such as;

- ALU operations - 3 bits signals to facilitate up to 8 arithmetic and logical functions. The proposed solution utilizes 6 different functions. The functions are summarized in the table 4
- LDx - Signals to load the registers with the content in the bus or another register, as per their connectivity.
- xBUS - Control signal sent to each buffer to feed the 16 bit output from each register to be written into the bus.
- RSTx - Control signal to clear and reset the values in a register.
- INCx - Control signal to increment the values in a register.

The table 6 summarizes the decoding of the uOPs and their respective control signals.

Table 6: Control signals decoded from the uOPs

Control Signal	**uOPs**
DREAD	DRMD
IREAD	DRMI
DWRITE	MDDR
BUSMEM	MDDR
MEMBUSD	DRMD
MEMBUSI	DRMI
TRBUS	ISMOD, INTDIV, ADD, MUL, SUB
DRBUS	ARDR, ACDR, MDDR
PCBUS	ARPC
NBUS	ACN
CBUS	TRC
JBUS	TRJ
IDBUS	ACID
COUNTBUS	TRCOUNT
TP1BUS	ARTP1
TP2BUS	ARTP2
TP3BUS	ARTP3
ICBUS	TRIC
IBUS	TRI
IEBUS	ACIE
TR2BUS	ACTR2, TRTR2
ACBUS	TRAC, DRAC
LDAR	ARDR, ARPC, ARTP1, ARTP2, ARTP3
LDTR	TRAC, TRJ, TRTR2, TRIC, TRC, TRI, TRCOUNT
LDDR	DRAC, DRMI, DRMD
LDPC	PCDR
LDN	NDR
LDC	CDR
LDIR	IRDR
LDTP1	TP1AC
LDTP2	TP2AC
LDTP3	TP3AC
LDIC	ICAC
LDI	IAC
LDIE	IEAC
LDTR2	TR2AC
LDAC	ISMOD, INTDIV, ACN, ACIE, ACTR2, ADD, ACID, ACDR, MUL, SUB
RSTAR	RESET
RSTTR	RESET
RSTDR	RESET
RSTPC	RESET
RSTN	RESET

Control Signal	uOPs
RSTC	RESET
RSTIR	RESET
RSTJ	JZ0, RESET
RSTID	RESET
RSTCOUNT	COUNTZ0, RESET
RSTTP1	RESET
RSTTP2	RESET
RSTTP3	RESET
RSTIC	RESET
RSTI	RESET
RSTIE	RESET
RSTTR2	TR2Z0, RESET
RSTAC	RESET
RSTEND	START, RESET
INCPC	PCINC
INCJ	JINC
INCCOUNT	COUNTINC
INCTP1	TP1INC
INCTP2	TP2INC
INCI	IINC
INCAC	ACINC
INCEND	ENDINC
ALU0	MUL, SUB, ISMOD
ALU1	ADD, SUB
ALU2	INTDIV, ISMOD
ADDKAC	ADDK

2.3.9 Clock

The clock signal is used to synchronize and coordinate the actions of the process where it oscillates in between the high and low states. This is especially used in synchronous sequential logic which changes its state changes only at discrete times of the clock signals. In the proposed solution, all the sequential implementations operate in either positive edge or the negative edge of the clock cycle and the process initiation in each core occurs at a positive edge.

2.3.10 Clock Corrector: Control Unit

Before the issuing of START signal by user and after getting END signal from the core, core should remain in the *idle* state. To achieve this, we have implemented Clock Corrector module as shown in Fig. 10. This makes the clock for control unit sensitive to START and END signals of the process. In a nutshell, this module is responsible to supply the standard clock as the clock for CU from the start to end of the process and otherwise remains 0.

Particularly this method will give the capability to RESET all other registers except control unit register without starting the computation process and keep the processor in idle state when there is no computations.

Apart from that, this module ensure that regardless of the instance the user starts the process, the initiation of the processing of each core occurs at a positive edge of the clock signal.

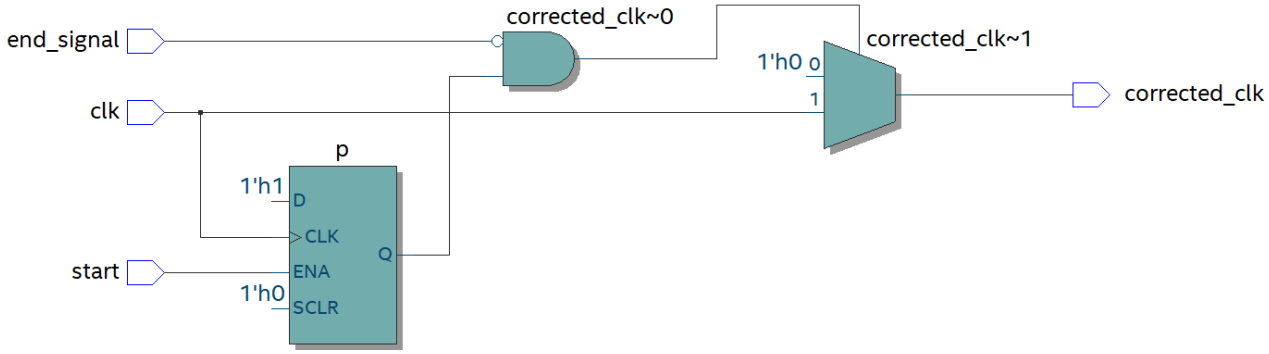

Figure 10: Implementation of Clock corrector for Control Unit

The corrected clock for CU when compares to the standard clock is shown in Figure 11

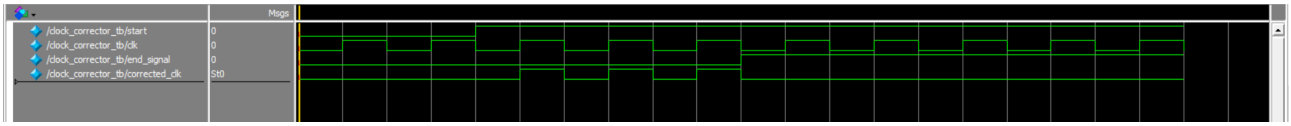

Figure 11: Comparison between corrected clock for control unit with the standard clock

2.3.11 Data Memory (DM)/ Multi-port Data Memory

DM is the main memory storage used to store the two input matrices and the resultant matrix. This is located outside the processor and has a width of 16 bits and a height of 1000 lines.

Since this is a multi-core solution, our DM is made such that it can be accessed parallely each core is directed towards separate locations to avoid any memory conflict. Hence, this implemented by multi-port memory structure where there is multiple AR ports, read-enable ports and data-out ports, one for each core. Currently, it is implemented to ten cores, however it can be expanded to any number of cores due to its scalability. In the proposed solution, at the initiation of the process, the data is always written into the memory in a positive clock edge.

The inputs and output of the DM are as shown in table 7.

Table 7: Inputs and Outputs of the DM

Input	AR-out	Output from the AR where it goes into IM and DM. This is a 16 bit address of a DM location in which data must be stored or retrieved.This input is loaded into the DM when the control signal DREAD is issued.
	output-BUS	Output from the Bus which includes the data to be stored which are sent from the rest of the registers via bus, in addition to AR. This input is stored into the DM when the control signal BUSMEM is issued along with the DWRITE.
	Clk	Clock Signal
Output	input-BUS	Input to the bus where the requested data from DM is fed into the bus when the buffer is enabled using the control signal MEMBUSD

Figure 12 illustrates the Data Memory of the proposed solution.

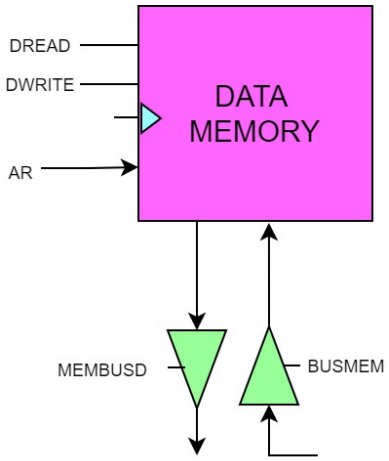

Figure 12: Data Memory

2.3.12 Instruction Memory (IM)

The IM is stored with the instructions to be executed in the correct order and is located outside the processor. In the proposed solution it is stored with 36 lines (16 bits wide), in the order as implemented in the assembly code (Section 3.3). It is a Read Only Memory (ROM) where new information cannot be written into it.

The input and output of the IM are as shown in table 8.

Table 8: Inputs and Outputs of the IM

Input	AR-out out	Output from the AR where it goes to IM and DM. This is a 16 bit address of a IM location.
Output	input-BUS	Input to the bus where the requested instruction from IM is fed into the bus when the buffer is enabled using the control signal MEMBUSI

Figure 13 depicts the Instruction Memory of the proposed solution.

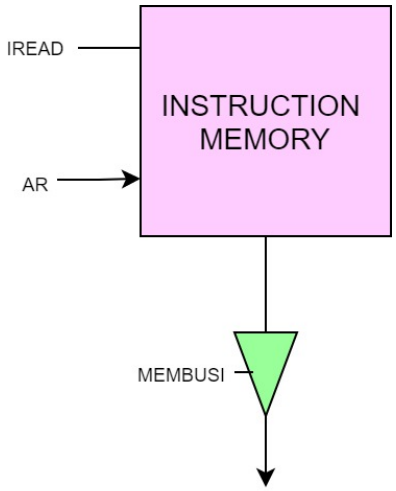

Figure 13: Instruction Memory

2.3.13 Top Module

The top module facilitates the interconnections and the initiations of the multiple cores, Multi-port Data Memory and the Instruction Memory. It also allows to take out the final output matrix from the system.

3 Algorithm

3.1 Computation inside a core

1. Initialization

   (a) Store data to data memory

      i. **Matrix 1**: Flatten matrix will be saved in the Data memory (M_D/ DM) as shown in Fig. 14. Saving will be started from 0^{th} memory location.

      ii. **Matrix 2**: Flatten matrix of Transpose of Matrix 2 will be saved in the Data memory (M_D) as shown in Fig. 15. Saving will be started from K^{th} memory location. Here $K = (mem_{size} - 2)//3$.

      iii. **Number of cores (c)**: Number of cores (c) will be saved in the memory. This will be used to identify allocated computations of each core.

      iv. **Matrix size (n)**: Matrix size (n) will be saved in the memory. This will be used to identify allocated computations of each core.

   (b) Initialize the core with $core\_id$
   Here the $core\_id$ represents a unique identity of each core. This is vary from $[0, c-1]$. Here c is the total number of cores.

2. Identifying the allocated computations of the core utilizing the $core\_id$

   As shown in Fig. 18 core identifies the allocated computations through the $core\_id$.

3. Computing allocated locations of output matrix

Algorithm 1: Computation in p^{th} core

Result: M_D with computed output matrix
Initialization: $core\_id \leftarrow p, c \leftarrow M_D, n \leftarrow M_D, Matrix_1 \leftarrow M_D, Matrix_2 \leftarrow M_D$;
$i_{count} \leftarrow$ calculate using n, c ;
$i_{start} \leftarrow core\_id \times i_c$;
$i_{end} \leftarrow (core\_id + 1) \times i_c$;
for $i; i_{start} \leq i < i_{end}; i++$ **do**
  for $j; 0 \leq j < n; j++$ **do**
    $output\_value = 0$;
    for $count; 0 \leq count < n; count++$ **do**
      $output\_value += matrix_1[i][count] \times matrix_2[j][count]$;
    end
    save $Matrix_{out}[i, j] = output\_value$ to M_D;
  end
end

Fig. 17 shows the clarification regarding the variables $i, j, count$.

4. Saving to memory

   Output matrix will be saved in the Data memory (M_D) starting from $2 \times K$ memory location as shown in Fig. 16. Here $K = (mem_{size} - 2)//3$.

Figure 18 depicts the process of memory access in the proposed solution.

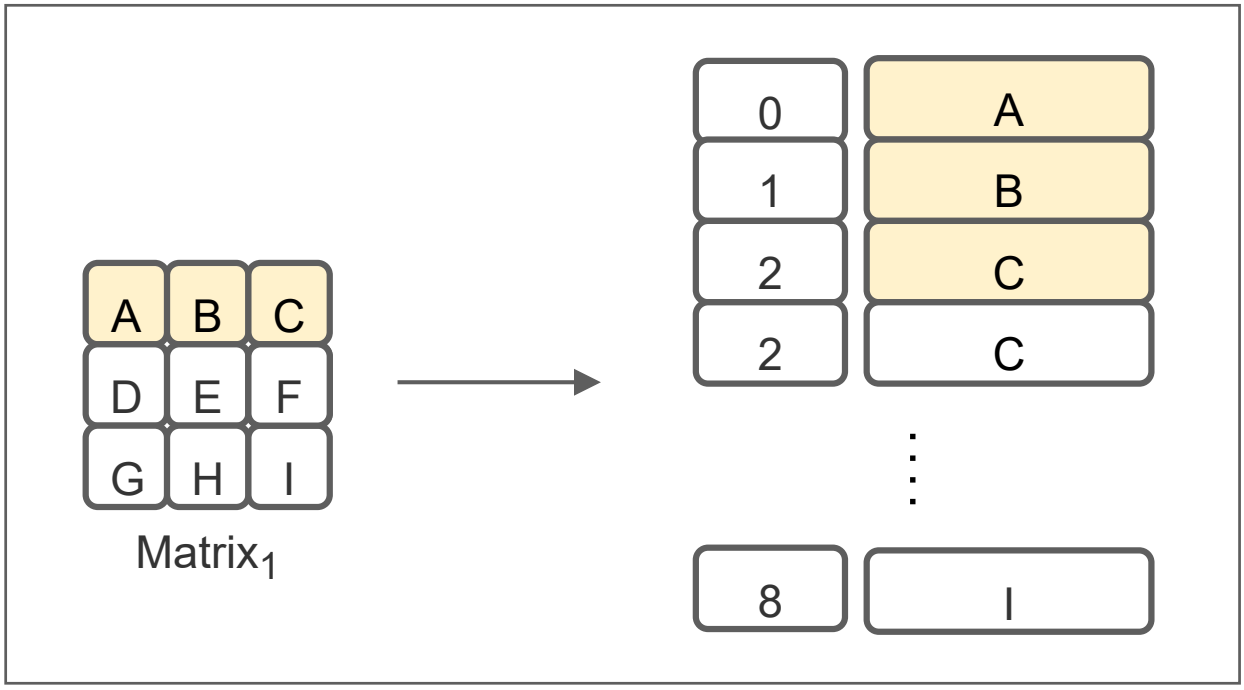

Figure 14: Storing $Matrix_1$ to the data memory

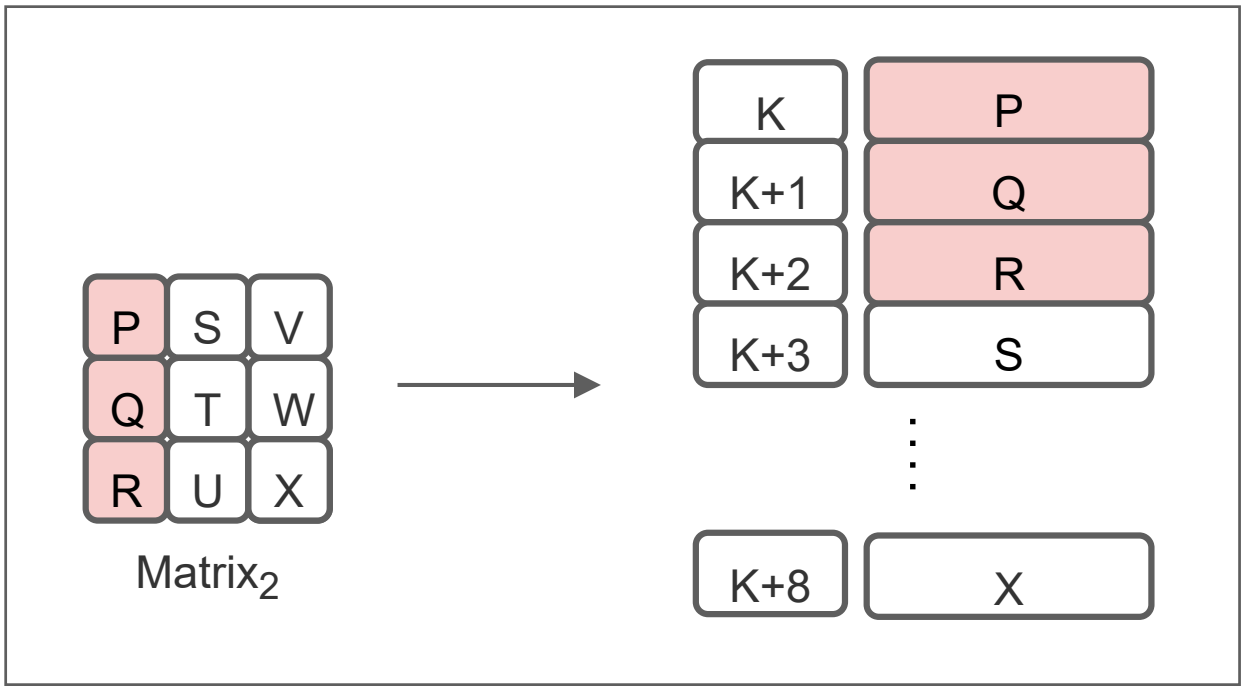

Figure 15: Storing $Matrix_2$ to the data memory

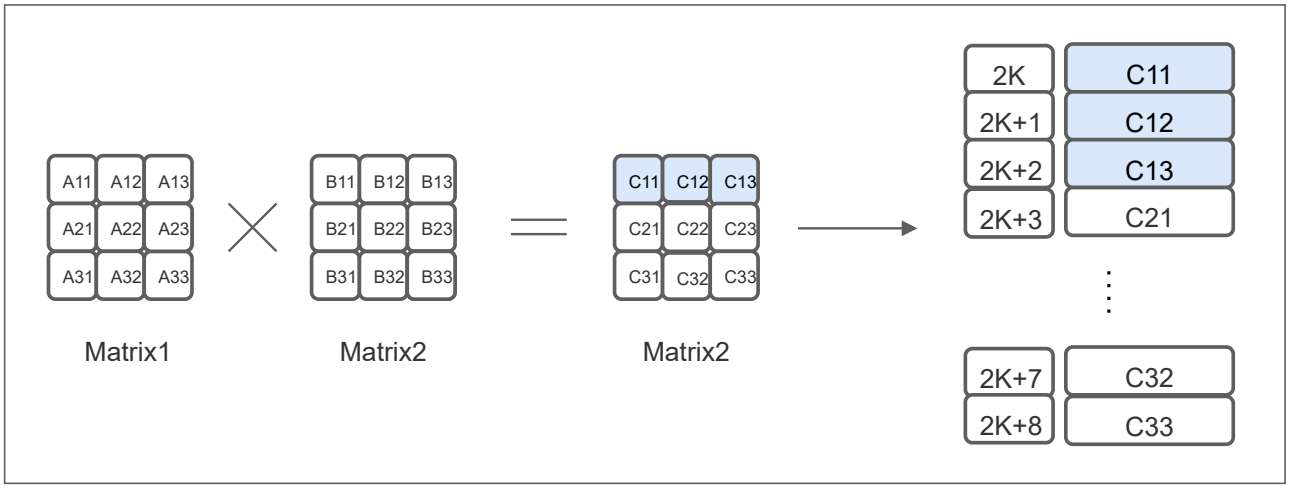

Figure 16: Storing $Matrix_out$ to data the memory

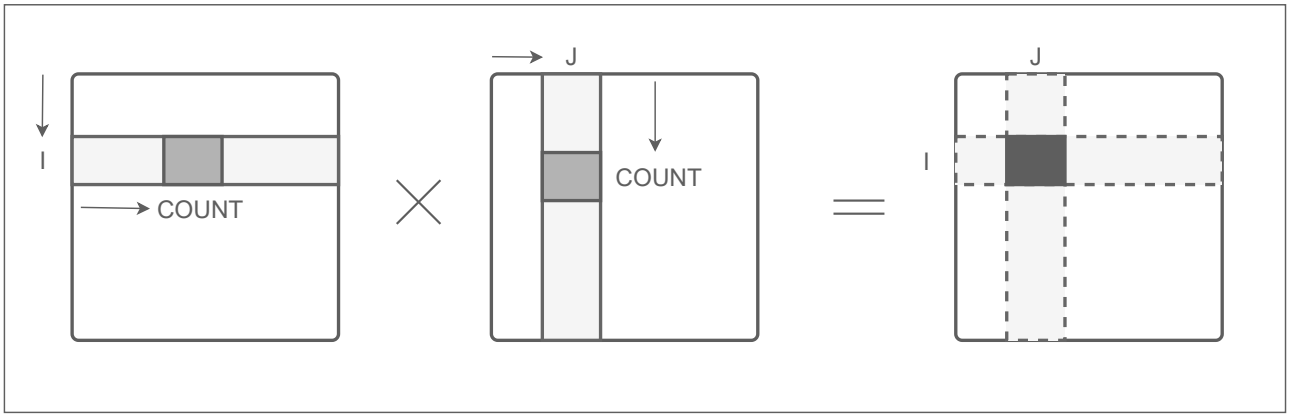

Figure 17: Variables $i, j, count$ in Algorithm 1

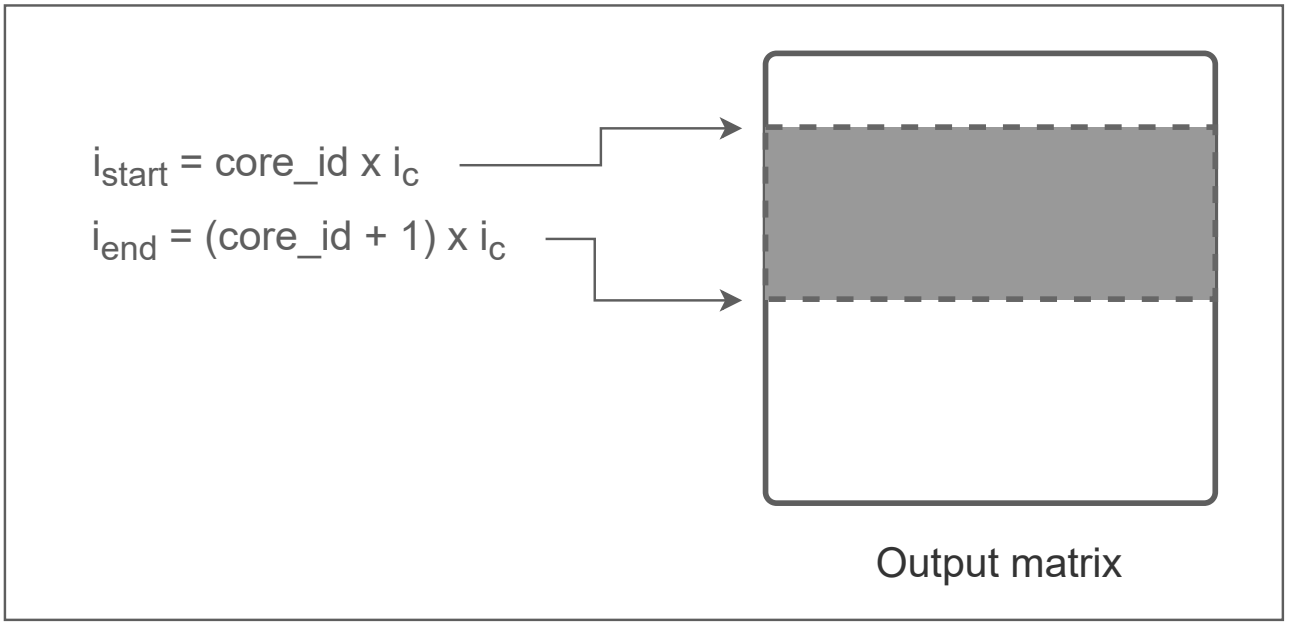

Figure 18: Identifying allocated computations of core with *core_id*

3.2 Assembly Code

The Assembly code which is stored in IM for the proposed solution is as follows;

1. SETN
2. α_n
3. SETC
4. α_c

5. RNGI
6. ENDI
7. STRTI
8. RSTJ
9. SETTP1
10. SETTP2
11. SETTP3
12. LDACTP1
13. MVACTR
14. LDACTP2
15. MUL
16. MVTR2
17. ADD
18. MVACTR2
19. MVCOUNT
20. LDN
21. SUB
22. JNPZ
23. 11
24. LDTR2
25. STAC
26. MVJ
27. LDN
28. SUB
29. JNPZ
30. 8
31. MVIE
32. MVI

33. SUB
34. JNPZ
35. 7
36. END

4 Verilog Implementation

In this section, verilog implementation detials, simulation detials and final RTL design is discussed.

4.1 Simulation using Modelsim/Questasim

Simulation is really vital when testing the verilog implementation of the design. It is quite fast and easy compared to fully fabricating the design in an actual FPGA board. In our team we have used both a customized version of Modelsim [6] for Intel boards and Questasim 10.6 [7] by Siemens. When configuring a new project in Quartus software we can add the simulator choice accordingly.

Each module is tested thoroughly using separate test benches. This approach enabled to detect bugs within the module and to move to higher hierarchical designs later. Then full core is tested from another test bench. Multi core implementation of 10 cores is finally tested and evaluated which is included in the results section.

Waveform feature in the Questasim was really helpful to debug the errors. Using correct wires and registers to the waveform allowed to detect memory errors, timing errors, etc. Also $display command is used to print important values on the terminal in the software for debugging. To measure runtime $realtime command is used. Also in Questasim once the verilog code of modules are compiled, any further change done in a test bench doesn't require you to compile the whole design again. Simply updating, recompiling and simulating the edited test bench is sufficient. This reduced testing time in a huge amount.

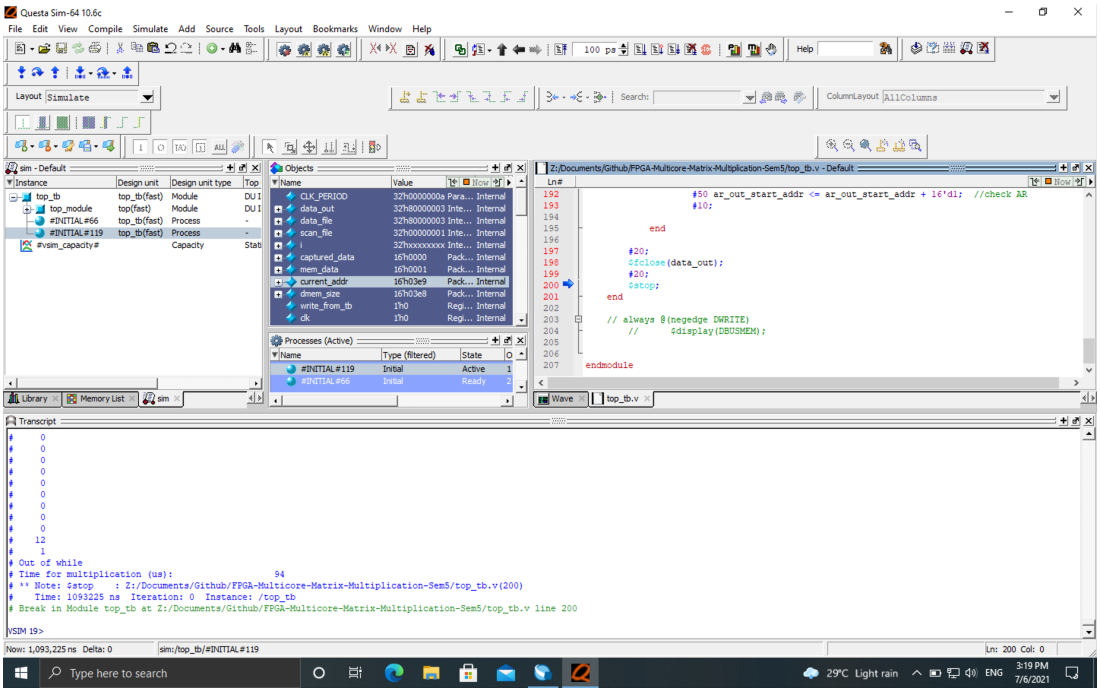

Figure 19: Simulation in Questasim

4.2 Quartus Implementation

Our processor is developed using Verilog HDL language and used Intel Quartus [8] Prime software to fabricate the design. The Verilog files can be imported to the Quartus project easily. Also Analysis and Synthesis feature can be used to compile the code and view the final RTL diagram. Quartus software allows to easily integrate Questasim simulation software which enables to open it directly from Quartus. Default testbenches can be added and configured too. Furthermore errors and warnings are shown after the Analysis and Synthesis is completed. We can go to the error directly from double clicking on the error shown on the terminal. In Quartus selecting the top module correctly is vital for the final RTL design view. Figure 20 shows a screenshot of the Quartus prime lite software with the compiled final design.

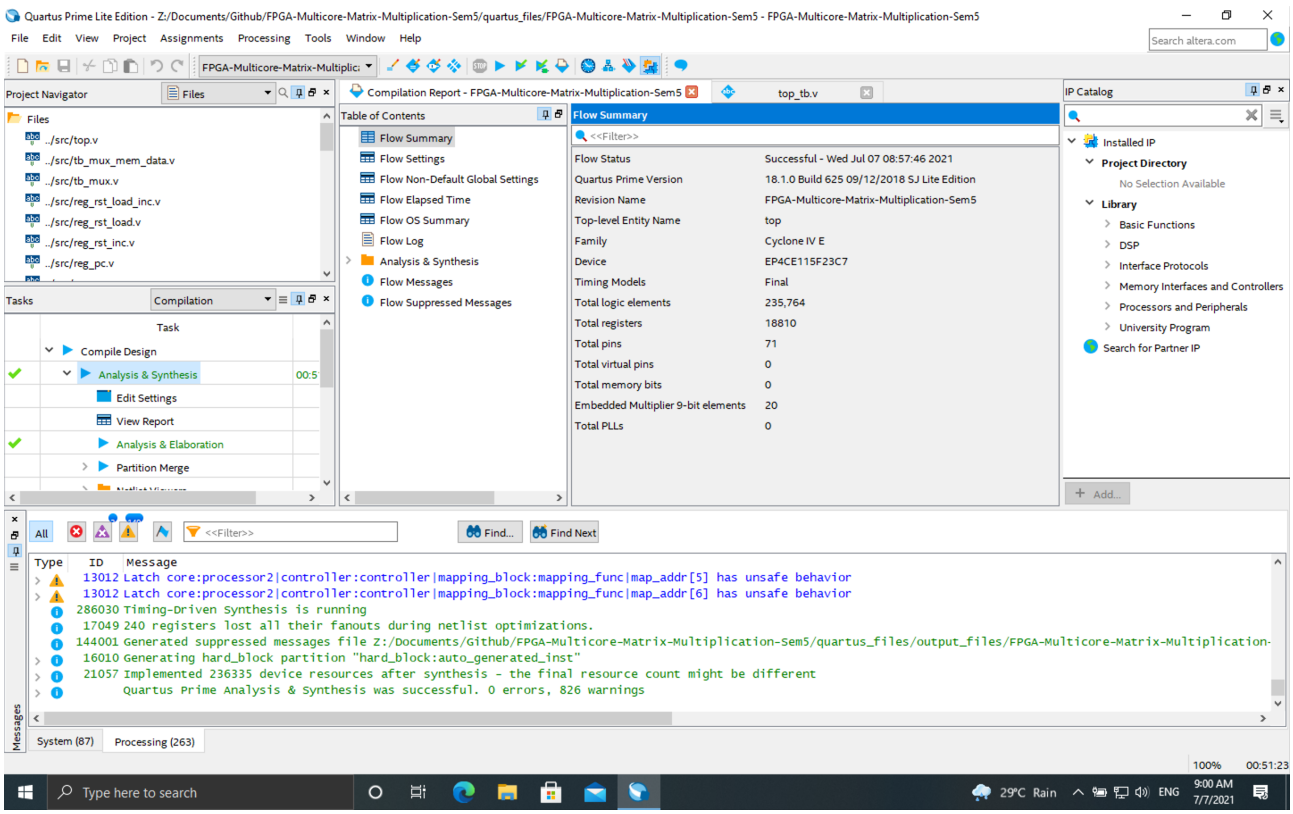

Figure 20: Using Quartus prime lite software

For Verilog programming we have used Visual Studio Code as it has intellisense feature and code auto-completion. Furthermore we used version control systems such as GitHub to collaboratively manage work among the team as we all contributed through online. Figure 21 shows a snapshot of VSCode.

Furthermore Altera Cyclone IV EP4CE115F23C7 is selected as the chip for the final design fabrication. Final task was to use multi core design to multiply two matrices together and display the results. For this we have used text file based method to write the initial matrices to the memory and to retrieve the final results from the memory. This is achieved through

the test bench by accessing the main memory using a separate Mux. Our algorithm explained above allows to utilise specified number of cores even though we have fabricated 8-10 cores in the analysis stage.

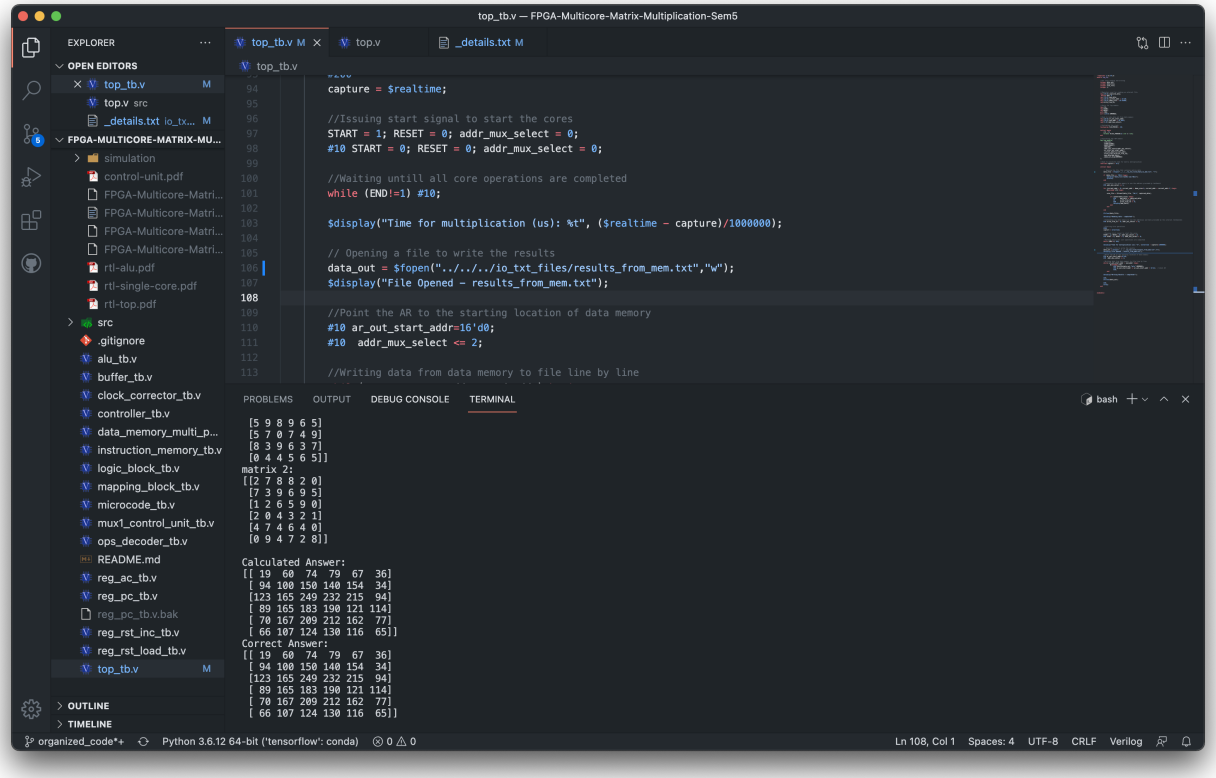

Figure 21: Using VSCode for coding

4.2.1 Design Flow Summary

Table 9 shows the design flow summary of 10 core design with a memory of 1000 length and a width of 2 bytes.

Table 9: Design Flow Summary

Flow Status	**Successful - Wed Jul 07 08:57:46 2021**
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FPGA-Multicore-Matrix-Multiplication-Sem5
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE115F23C7
Timing Models	Final
Total logic elements	235,764
Total pins	71
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	20
Total PLLs	0

4.2.2 Resource Usage Summary

Table 10 shows the resource usage summary of 10 core design with a memory of 1000 length and a width of 2 bytes.

Table 10: Resource Usage Summary

Resource	**Usage**
Total Registers	18810
– I/O Registers	0
– Dedicated logic registers	18810
Total fan-out	982304
Total combinational functions	234794
Maximum fan-out node	clk∼input
Maximum fan-out	18750
Logic elements by node	
– normal node	230444
– arithmetic node	4350
Logic element usage by number of LUT inputs	
– 3 input functions	6736
– 4 input functions	224371
– <=2 input functions	3687
I/O Pins	71
Estimated Total logic elements	235,764
Embedded Multiplier 9-bit elements	20
Average fan-out	3.87

4.2.3 Resource Utilization by Entity

Table 11 shows the Resource Utilization by Entity of 10 core design with a memory of 1000 length and a width of 2 bytes.

Table 11: Resource Utilization by Entity

Compilation Hierarchy Node	**Combinational ALUTs**	**Dedicated Logic Registers**
|top	234794 (0)	18810 (0)
|core:processor1|	1407 (5)	281 (0)
|alu:ALU|	632 (63)	0 (0)
|lpm_divide:Div0|	270 (0)	0 (0)
|lpm_divide_lkm:auto_generated|	270 (0)	0 (0)
|sign_div_unsign_dnh:divider|	270 (0)	0 (0)
|alt_u_div_eaf:divider|	270 (269)	0 (0)

Continued on next page

Table 11 : Continued from previous page

Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers
|add_sub_8pc:add_sub_1|	1 (1)	0 (0)
|lpm_divide:Mod0|	299 (0)	0 (0)
|lpm_divide_ocm:auto_generated|	299 (0)	0 (0)
|sign_div_unsign_dnh:divider|	299 (0)	0 (0)
|alt_u_div_eaf:divider|	299 (298)	0 (0)
|add_sub_8pc:add_sub_1|	1 (1)	0 (0)
|lpm_mult:Mult0|	0 (0)	0 (0)
|mult_7dt:auto_generated|	0 (0)	0 (0)
|buffer:DR_buf|	1 (1)	0 (0)
|buffer:ID_buf|	120 (120)	0 (0)
|buffer:MEMBUSI_buf|	67 (67)	0 (0)
|buffer:TR_buf|	11 (11)	0 (0)
|clock_corrector_module:clock_corrector|	1 (1)	1 (1)
|controller:controller|	186 (0)	7 (0)
|cu_reg_rst_load:register|	7 (7)	7 (7)
|logic_block:logic_box|	1 (1)	0 (0)
|mapping_block:mapping_func|	71 (71)	0 (0)
|microcode:micromemory|	81 (81)	0 (0)
|mux1_control_unit:mux1|	25 (25)	0 (0)
|mux2_control_unit:mux2|	1 (1)	0 (0)
|ops_decoder:decoder|	19 (19)	0 (0)
|reg_ac:AC|	158 (158)	16 (16)
|reg_pc:PC|	48 (48)	16 (16)
|reg_rst_inc:COUNT|	16 (16)	16 (16)
|reg_rst_inc:END|	2 (2)	1 (1)
|reg_rst_inc:J|	16 (16)	16 (16)
|reg_rst_load:AR|	18 (18)	16 (16)
|reg_rst_load:C|	1 (1)	16 (16)
|reg_rst_load:DR|	1 (1)	16 (16)
|reg_rst_load:IC|	1 (1)	16 (16)
|reg_rst_load:IE|	1 (1)	16 (16)
|reg_rst_load:IR|	17 (17)	16 (16)
|reg_rst_load:N|	1 (1)	16 (16)
|reg_rst_load:TP3|	16 (16)	16 (16)
|reg_rst_load:TR2|	17 (17)	16 (16)
|reg_rst_load:TR|	4 (4)	16 (16)
|reg_rst_load_inc:I|	16 (16)	16 (16)
|reg_rst_load_inc:TP1|	16 (16)	16 (16)
|reg_rst_load_inc:TP2|	16 (16)	16 (16)
|core:processor2|	1438 (5)	281 (0)

Continued on next page

Table 11 : Continued from previous page

Compilation Hierarchy Node	**Combinational ALUTs**	**Dedicated Logic Registers**
|core:processor3|	1438 (5)	281 (0)
|core:processor4|	1436 (5)	281 (0)
|core:processor5|	1435 (5)	281 (0)
|core:processor6|	1438 (5)	281 (0)
|core:processor7|	1432 (5)	281 (0)
|core:processor8|	1433 (5)	281 (0)
|core:processor9|	1430 (5)	281 (0)
|core:processor10|	1431 (5)	281 (0)
|data_memory_multi_port:DM_4|	220413 (220413)	16000 (16000)
|dmem_mux:data_mem_mux|	21 (21)	0 (0)
|instruction_memory:IM|	26 (26)	0 (0)
|tb_mux_mem_data:mem_data_select_mux|	16 (16)	0 (0)

4.2.4 RTL Design

RTL viewer in Intel Quartus Prime software allows to visualize our Verilog modules and their interconnections. This comes in handy to find miss connections and wrong connections between modules. Below are some RTL views of the core components of our design.

10 Core Top level design

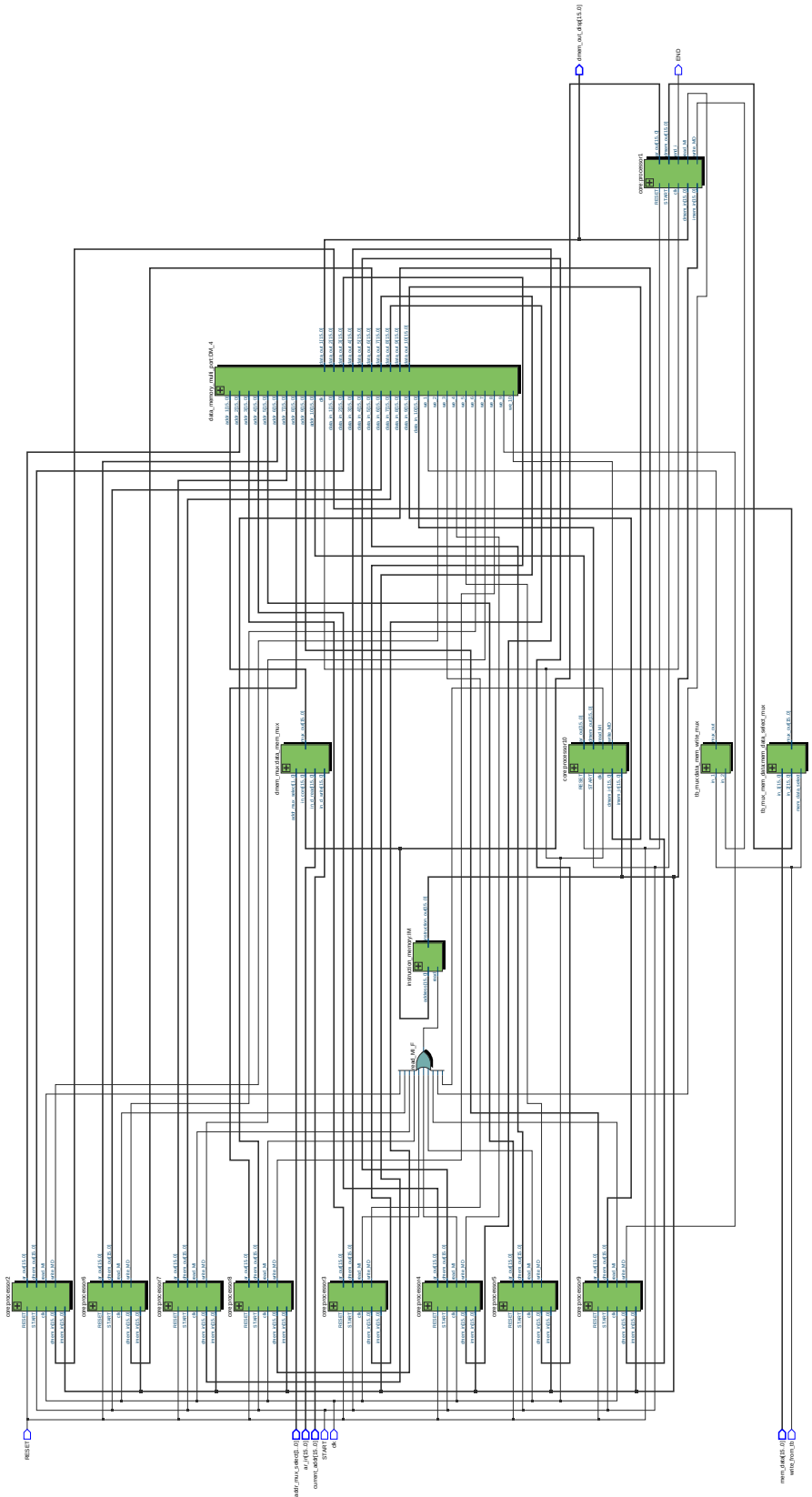

Figure 22: 10 Core top level design

Single Core RTL design

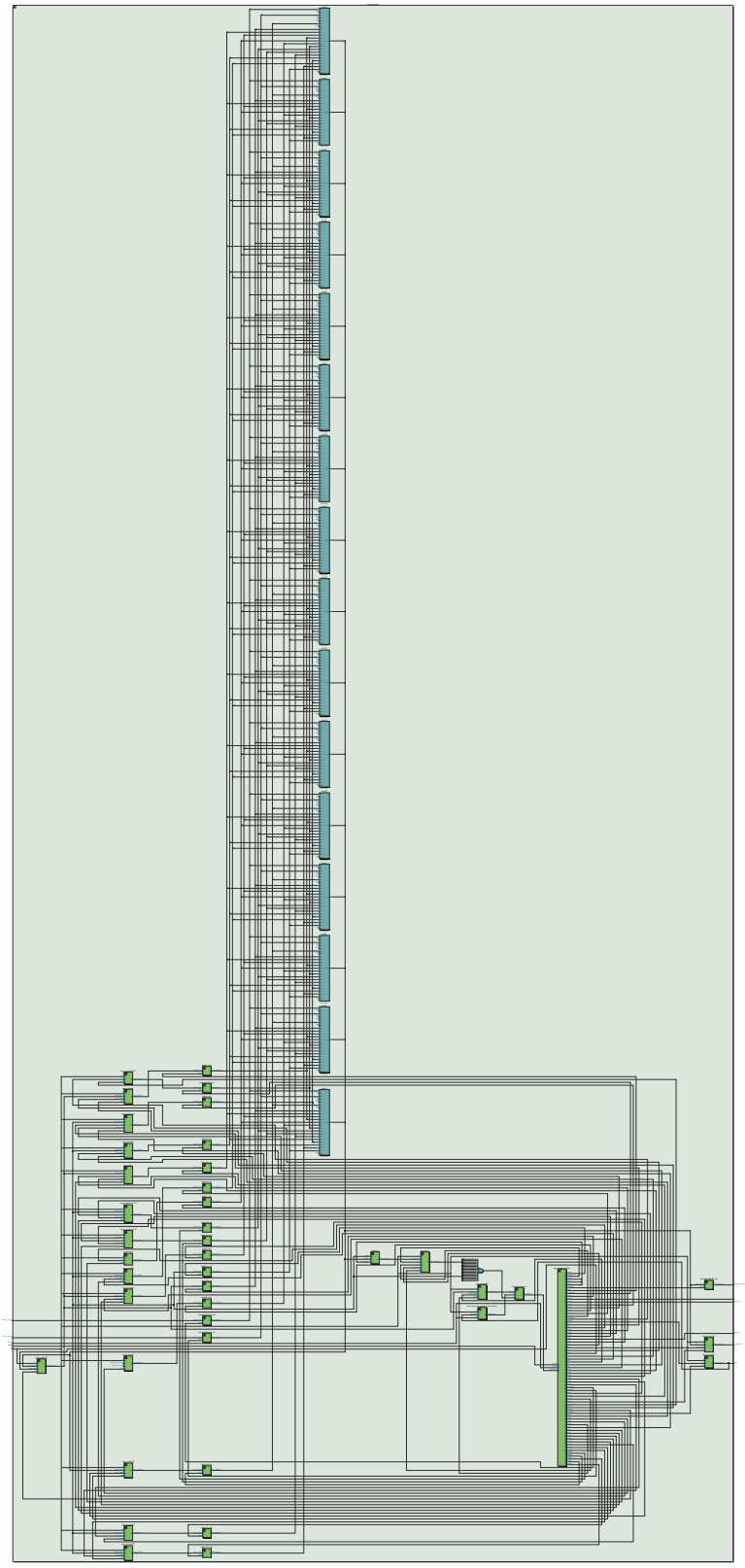

Figure 23: Single Core RTL design

Control Unit design

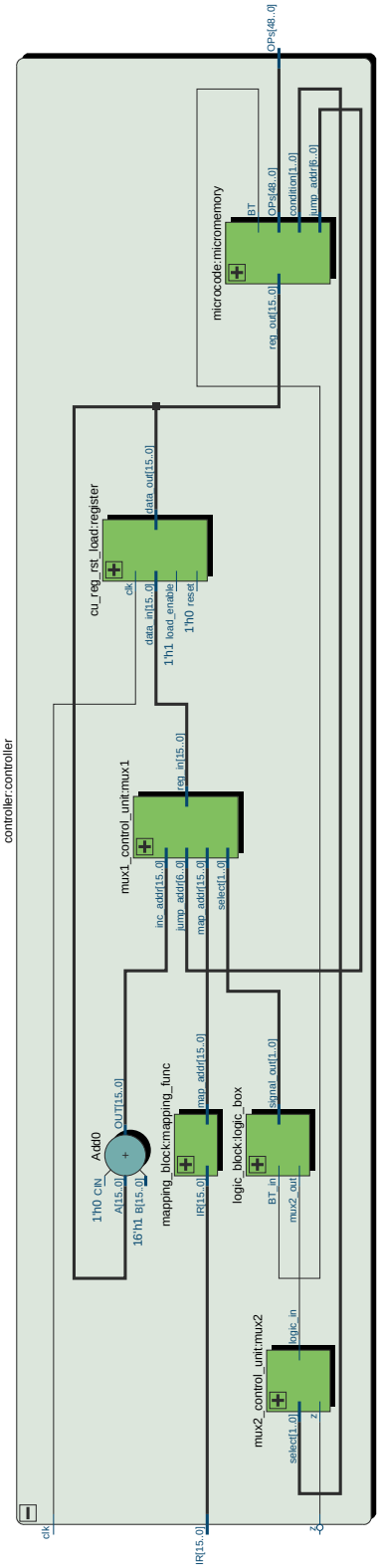

Figure 24: Control Unit design

ALU design

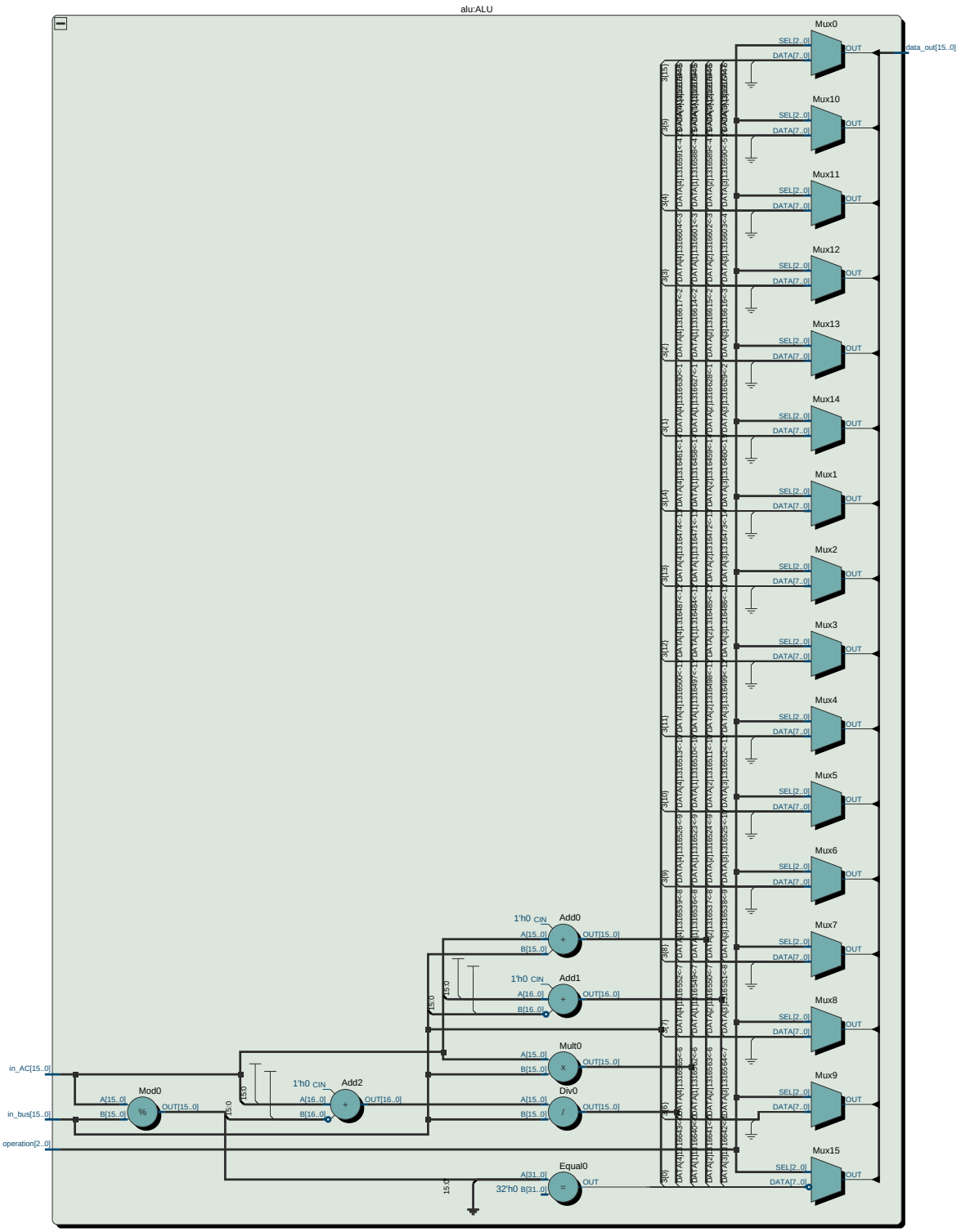

Figure 25: ALU design

5 Performance Evaluation

5.1 Detailed Comparison

To evaluate the performance of the designed multi-core processor we have selected random initialized matrices with sizes 4X4, 8X8 and 16X16 and analysed the time taken for matrix multiplication when number of cores are increased from 1 to 10.

The results are listed below.

1. For N = 4,

   Some screenshots taken during the testing stage.

```
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python save_matrix2.py --n 4 --c 1
written to data_to_mem.txt
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python check_matrix.py --n 4
matrix 1:
[[ 1 10  2  7]
 [10  4  7  6]
 [ 7  4 19 14]
 [ 2  4 12 13]]
matrix 2:
[[ 5  0 10 13]
 [ 2 11  0 19]
 [14  7 11  5]
 [14 12  7 16]]

Calculated Answer:
[[151 208  81 325]
 [240 165 219 337]
 [505 345 377 486]
 [368 284 243 370]]
Correct Answer:
[[151 208  81 325]
 [240 165 219 337]
 [505 345 377 486]
 [368 284 243 370]]

Is answer correct : True
```

Figure 26: Matrix results for 4X4 matrix with 1 core

```
#     4
#     1
# Reading data - completed!
# Time for multiplication (us):                  4
# File Opened - results_from_mem.txt
# Writing Results - completed!
# ** Note: $stop    : D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v(127)
#    Time: 191705 ns  Iteration: 0  Instance: /top_tb
# Break in Module top_tb at D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v line 127
```

Figure 27: Calculation time for 4X4 matrix with 1 core

```
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python save_matrix2.py --n 4 --c 4
written to data_to_mem.txt
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python check_matrix.py --n 4
matrix 1:
[[15 10 18 12]
 [16  3  8  4]
 [18 13  6  7]
 [18  0 14  6]]
matrix 2:
[[16 18  6  6]
 [ 2  7 13  0]
 [17  2  0 14]
 [ 8  4 13 16]]

Calculated Answer:
[[662 424 376 534]
 [430 341 187 272]
 [472 455 368 304]
 [574 376 186 400]]
Correct Answer:
[[662 424 376 534]
 [430 341 187 272]
 [472 455 368 304]
 [574 376 186 400]]

Is answer correct : True
```

Figure 28: Matrix results for 4X4 matrix with 4 cores

```
#     4
#     4
# Reading data - completed!
# Time for multiplication (us):                    1
# File Opened - results_from_mem.txt
# Writing Results - completed!
# ** Note: $stop    : D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v(127)
#    Time: 160715 ns  Iteration: 0  Instance: /top_tb
# Break in Module top_tb at D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v line 127
```

Figure 29: Calculation time for 4X4 matrix with 4 cores

Table 12: Results observed for 4x4 matrices

N	No. of Cores	Time(us)
4	1	4
	2	2
	3	2
	4	1
	5	1
	6	1
	7	1
	8	1
	9	1
	10	1

2. For N = 8,

   Some screenshots taken during the testing stage.

```
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python save_matrix2.py --n 8 --c 1
written to data_to_mem.txt
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python check_matrix.py --n 8
matrix 1:
[[14  0  8 19 10 13 15  0]
 [ 7  0 15 12 19 17  0  2]
 [15  2 15  7 11  9  7 13]
 [15  3  7 13  8 18 17  2]
 [12  3  1 12 12 17 18  6]
 [13  1 10 16 14  5 15  9]
 [ 9  7  2 13 18 10 11 14]
 [ 1 12  2 11  8  3 18  0]]
matrix 2:
[[ 9  8  8 16 13  7  8 15]
 [14 18 12 19  7  4 17 10]
 [ 2 14 16 10 12  1 11  0]
 [10 17 16 17  1 11 18 18]
 [ 9  6  2  9 13  4  9 14]
 [11 12  1 17  2 17 10 15]
 [13  3 15 14 19  2  0  3]
 [ 6 16 11 16  6  6 13  8]]

Calculated Answer:
[[ 760  808  802 1148  738  606  762  932]
 [ 583  820  565  958  576  573  804  858]
 [ 630  888  775 1105  768  494  803  785]
 [ 824  840  787 1236  788  651  760  938]
 [ 837  794  717 1208  769  638  730  951]
 [ 741  867  889 1164  843  506  812  881]
 [ 812  932  761 1228  750  584  891  986]
 [ 630  577  649  826  584  297  534  544]]
Correct Answer:
[[ 760  808  802 1148  738  606  762  932]
 [ 583  820  565  958  576  573  804  858]
 [ 630  888  775 1105  768  494  803  785]
 [ 824  840  787 1236  788  651  760  938]
 [ 837  794  717 1208  769  638  730  951]
 [ 741  867  889 1164  843  506  812  881]
 [ 812  932  761 1228  750  584  891  986]
 [ 630  577  649  826  584  297  534  544]]

Is answer correct : True
```

Figure 30: Matrix results for 8X8 matrix with 1 core

```
#      8
#      1
# Reading data - completed!
# Time for multiplication (us):                  29
# File Opened - results_from_mem.txt
# Writing Results - completed!
# ** Note: $stop    : D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v(127)
#    Time: 441985 ns  Iteration: 0  Instance: /top_tb
# Break in Module top_tb at D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v line 127
```

Figure 31: Calculation time for 8X8 matrix with 1 core

```
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python save_matrix2.py --n 8 --c 10
written to data_to_mem.txt
PS D:\FPGA\Project\FPGA based Multicore Matrix Multiplier\io_txt_files> python check_matrix.py --n 8
matrix 1:
[[12 10 18  4 12  0  0 13]
 [ 5 19  7  1 15  3 14  6]
 [ 3 19 13  5 14  9 13  7]
 [12 11  2 12 19 12 10  1]
 [ 9  2  0  6  0 16  4  5]
 [ 7 15 13  2 19  0  8 12]
 [19  6 17  9 17 13 10  0]
 [ 5  4 14  6  1 14  9  2]]
matrix 2:
[[19  6  2  8  6  9  1 10]
 [ 2 17 10  1  5  5 18  1]
 [ 7  2  5  0 12 11  1  8]
 [ 1 14 11 17 13 16  8  9]
 [15  8 17 12  1 10  9  4]
 [ 2 11  3  3 11 14 18 16]
 [13 16 19 12  9  7 15  1]
 [15 17 11  8  0  7 13  5]]

Calculated Answer:
[[753 651 605 422 402 631 519 423]
 [686 860 842 481 396 565 839 286]
 [693 975 905 535 564 751 972 446]
 [730 892 836 703 548 812 858 538]
 [340 497 283 312 354 474 506 431]
 [825 835 858 525 390 643 753 351]
 [912 815 800 674 715 954 753 699]
 [397 550 438 324 532 598 561 467]]
Correct Answer:
[[753 651 605 422 402 631 519 423]
 [686 860 842 481 396 565 839 286]
 [693 975 905 535 564 751 972 446]
 [730 892 836 703 548 812 858 538]
 [340 497 283 312 354 474 506 431]
 [825 835 858 525 390 643 753 351]
 [912 815 800 674 715 954 753 699]
 [397 550 438 324 532 598 561 467]]

Is answer correct : True
```

Figure 32: Matrix results for 8X8 matrix with 10 cores

```
#      8
#     10
# Reading data - completed!
# Time for multiplication (us):                    4
# File Opened - results_from_mem.txt
# Writing Results - completed!
# ** Note: $stop    : D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v(127)
#    Time: 186835 ns  Iteration: 0  Instance: /top_tb
# Break in Module top_tb at D:/FPGA/Project/FPGA based Multicore Matrix Multiplier/top_tb.v line 127
```

Figure 33: Calculation time for 8X8 matrix with 10 cores

Table 13: Results observed for 8x8 matrices

N	No. of Cores	Time(us)
8	1	29
	2	15
	3	11
	4	7
	5	7
	6	7
	7	7
	8	4
	9	4
	10	4

3. For N = 16,

Table 14: Results observed for 16x16 matrices

N	No. of Cores	Time(us)
16	1	219
	2	109
	3	82
	4	55
	5	55
	6	41
	7	41
	8	27
	9	27
	10	27

5.2 Summary of results

Fig. 34 shows the quantitative comparison of the performance results for different number of cores. 3 diagrams shows the performance results for different matrix sizes.

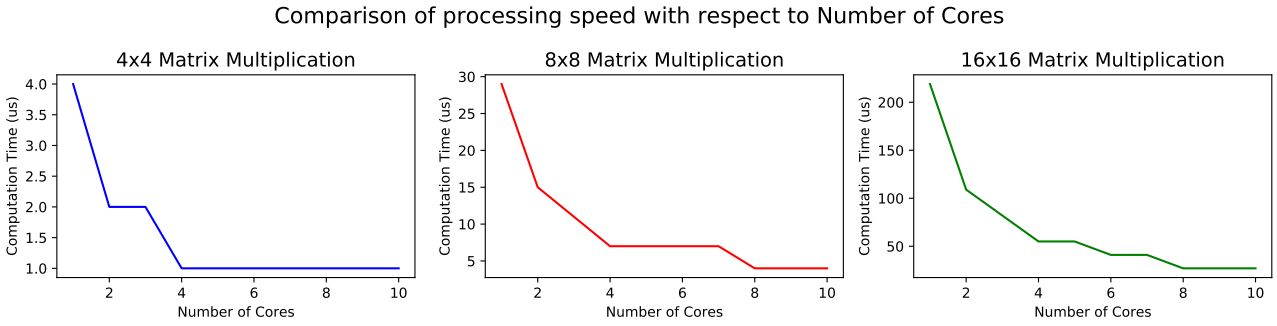

Figure 34: Performance comparison for different number of cores and different matrix sizes

5.3 Conclusion

To this end, we have implemented and tested 10-core processor for paralleled matrix multiplication with 16-bit valued data. Because of the independent processing of each core, the capability of adding more cores is straight forward.

With the results shown in Fig. 34, it can be seen that the computation time for matrix multiplication is decreasing when increasing the number of cores for all matrix sizes. The implemented processor is capable of doing;

- (4x4) 16-bit valued matrix multiplication within around **1 us** utilizing 4 cores.
- (8x8) 16-bit valued matrix multiplication within around **4 us** utilizing 8 cores.
- (16x16) 16-bit valued matrix multiplication within around **27 us** utilizing 10 cores.

Even though initial reduction of computation time is significant, number of cores will no longer be affected to the computation time when the number of cores passes the size of matrices.

6 Acknowledgement

A special thanks goes to our lecturer, Dr. Jayathu Samarawickrama for not only providing us with all the necessary theoretical knowledge regarding processor design but also for advising us on the potential issues, trade-offs and solutions when designing an Instruction Set Architecture.

References

[1] "Cpu: Central processing unit | ap csp (article)."

[2] "Matrix multiplication," May 2021.

[3] "What is an fpga? programming and fpga basics - intel® fpgas."

[4] "What is an fpga? field programmable gate array."

[5] J. D. Carpinelli, *Computer systems organization & architecture.* Addison-Wesley, 2001.

[6] "Modelsim software." https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html.

[7] "Questasim software." https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/.

[8] "Intel quartus prime software." https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

Appendix

The full implementation can be found in GitHub repository.

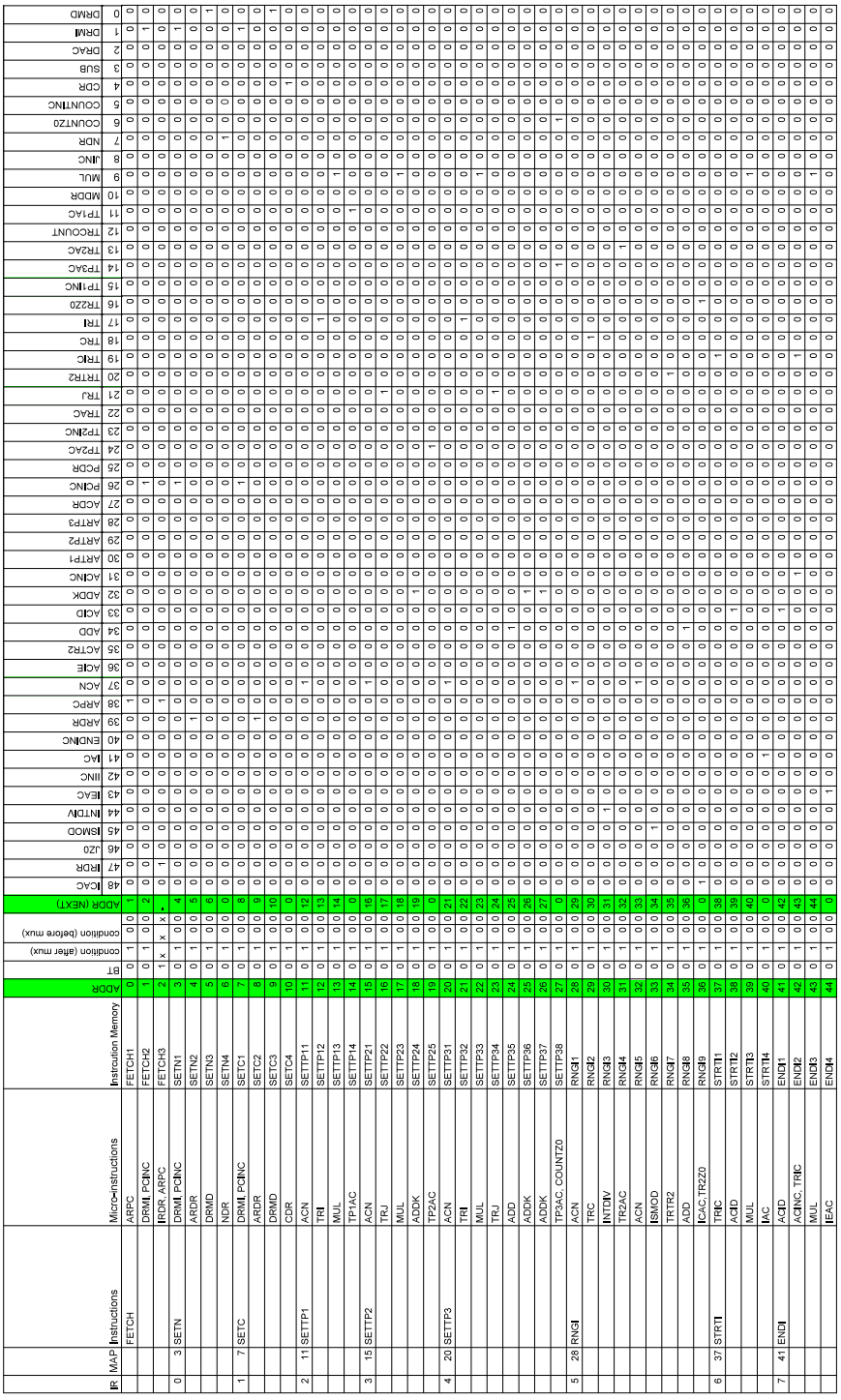

Figure 35: uOPs for microinstructions Part 1

8	45	LDACTP1	ARTP1, COUNTINC	LDACTP11	45	0	1	0	0	46	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
			DRMD	LDACTP12	46	0	1	0	0	47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
			ACDR	LDACTP13	47	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	48	LDACTP2	ARTP2	LDACTP21	48	0	1	0	0	49	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			DRMD	LDACTP22	49	0	1	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
			ACDR	LDACTP23	50	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	51	LDACTP3	ARTP3	LDACTP31	51	0	1	0	0	52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			DRMD	LDACTP32	52	0	1	0	0	53	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
			ACDR	LDACTP33	53	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	54	LDN	ACN	LDN1	54	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	55	LDTR2	ACTR2	LDTR21	55	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	56	MVACTR	TRAC	MVACTR1	56	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	57	MVACTR2	TR2AC	MVACTR21	57	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
13	58	MVCOUNT	TRCOUNT, TP1INC, TP2INC	MVCOUNT1	58	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
14	59	MVJ	TRJ	MVJ1	59	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	60	MVI	TRI	MVI1	60	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	61	MVIE	ACIE, IINC	MVIE1	61	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	62	MVTR2	TRTR2	MVTR21	62	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	63	STAC	ARTP3, TR2Z0	STAC1	63	0	1	0	0	64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			DRAC	STAC2	64	0	1	0	0	65	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
			MDDR, JINC	STAC3	65	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
21	66	JNPZ		JNPZ1	66	0	z	0	1	69	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	67	JNPZY	DRMI	JNPZY1	67	0	1	0	0	68	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
			PCDR	JNPZY2	68	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	69	JNPZN	PCINC	JNPZN1	69	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	70	RSTJ	JZ0	RSTJ1	70	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	71	END	ENDINC	END1	71	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	72	MUL_	MUL	MUL1	72	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
18	73	ADD_	ADD	ADD1	73	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	74	SUB_	SUB	SUB1	74	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Figure 36: uOPs for microinstructions Part 2

Modules

Top Module

```
module top(
    input clk,
    input START,
    input RESET,

    //For writing and reading data memory from external files
    input [1:0] addr_mux_select,
    input [15:0] ar_in,
    input [15:0] current_addr,
    input write_from_tb,
    input [15:0] mem_data,

    output END,
  output [15:0] dmem_out_disp

);

//Wires for core connections

// Intermediate wires for core 1
wire [15:0] dmem_in;
wire [15:0] imem_in;
wire [15:0] dmem_out;
wire [15:0] ar_out;

// Intermediate wires for core 2
wire [15:0] dmem_in2;
wire [15:0] dmem_out2;
wire [15:0] ar_out2;

// Intermediate wires for core 3
wire [15:0] dmem_in3;
wire [15:0] dmem_out3;
wire [15:0] ar_out3;

// Intermediate wires for core 4
wire [15:0] dmem_in4;
wire [15:0] dmem_out4;
wire [15:0] ar_out4;

// Intermediate wires for core 5
wire [15:0] dmem_in5;
wire [15:0] dmem_out5;
wire [15:0] ar_out5;

// Intermediate wires for core 6
wire [15:0] dmem_in6;
wire [15:0] dmem_out6;
wire [15:0] ar_out6;

```

```
// Intermediate wires for core 7
wire [15:0] dmem_in7;
wire [15:0] dmem_out7;
wire [15:0] ar_out7;

// Intermediate wires for core 8
wire [15:0] dmem_in8;
wire [15:0] dmem_out8;
wire [15:0] ar_out8;

// Intermediate wires for core 9
wire [15:0] dmem_in9;
wire [15:0] dmem_out9;
wire [15:0] ar_out9;

// Intermediate wires for core 10
wire [15:0] dmem_in10;
wire [15:0] dmem_out10;
wire [15:0] ar_out10;


//Control signals
wire read_MI_F;
wire read_MD, read_MI, write_MD;    //core 1
wire read_MD2, read_MI2, write_MD2; //core 2
wire read_MD3, read_MI3, write_MD3; //core 3
wire read_MD4, read_MI4, write_MD4; //core 4
wire read_MD5, read_MI5, write_MD5; //core 5
wire read_MD6, read_MI6, write_MD6; //core 6
wire read_MD7, read_MI7, write_MD7; //core 7
wire read_MD8, read_MI8, write_MD8; //core 8
wire read_MD9, read_MI9, write_MD9; //core 9
wire read_MD10, read_MI10, write_MD10; //core 10

//Used to write data in data memory to an external file
assign dmem_out_disp = dmem_in;

wire [15:0] d_mem_addr;
assign memory_in_addr = d_mem_addr;

// [Instruction Memory]
instruction_memory IM(
    .read(read_MI_F),
    .address(ar_out),
    .instruction_out(imem_in)
);

// To select addresses coming from cores and test bench for writing and
   reading from external file
dmem_mux data_mem_mux(
    .in_core(ar_out),
    .in_d_write(current_addr),
    .in_d_read(ar_in),
    .addr_mux_select(addr_mux_select),
    .mux_out(d_mem_addr)
```

```
105 );
106
107 // To select write enable signals coming from cores and test bench for
       writing and reading from external files
108 wire write_to_data_mem;
109
110 tb_mux  data_mem_write_mux(
111     .in_1(write_from_tb),
112     .in_2(write_MD),
113     .mux_out(write_to_data_mem)
114 );
115
116 // To select data coming from cores and test bench for writing and reading
       from external files
117 wire [15:0] data_for_mem;
118
119 tb_mux_mem_data mem_data_select_mux(
120     .in_1(mem_data),
121     .in_2(dmem_out),
122     .mem_data_select(write_from_tb),
123     .mux_out(data_for_mem)
124 );
125
126 //Allowing all cores to access the instructions
127 assign read_MI_F = read_MI || read_MI2 || read_MI3 || read_MI4 || read_MI5
       || read_MI6 || read_MI7 || read_MI8 ||read_MI9 || read_MI10 ;
128
129 //Initiating Cores
130 // [CORE 1]
131 core #(.core_id(0)) processor1 (
132     .clk(clk),
133     .START(START),
134     .RESET(RESET),
135     .read_MI(read_MI),
136     .imem_in(imem_in),
137     .dmem_in(dmem_in),
138     .dmem_out(dmem_out),
139     .ar_out(ar_out),
140     .read_MD(read_MD),
141     .write_MD(write_MD),
142     .end_i(END)
143  );
144
145 // [CORE 2]
146 core #(.core_id(1)) processor2 (
147     .clk(clk),
148     .START(START),
149     .RESET(RESET),
150     .read_MI(read_MI2),
151     .imem_in(imem_in),
152     .dmem_in(dmem_in2),
153     .dmem_out(dmem_out2),
154     .write_MD(write_MD2),
155     .ar_out(ar_out2)
156  );
```

```
157
158 // [CORE 3]
159 core #(.core_id(2)) processor3 (
160     .clk(clk),
161     .START(START),
162     .RESET(RESET),
163     .read_MI(read_MI3),
164     .imem_in(imem_in),
165     .dmem_in(dmem_in3),
166     .dmem_out(dmem_out3),
167     .write_MD(write_MD3),
168     .ar_out(ar_out3)
169  );
170
171 // [CORE 4]
172 core #(.core_id(3)) processor4 (
173     .clk(clk),
174     .START(START),
175     .RESET(RESET),
176     .read_MI(read_MI4),
177     .imem_in(imem_in),
178     .dmem_in(dmem_in4),
179     .dmem_out(dmem_out4),
180     .write_MD(write_MD4),
181     .ar_out(ar_out4)
182  );
183
184  // [CORE 5]
185 core #(.core_id(4)) processor5 (
186     .clk(clk),
187     .START(START),
188     .RESET(RESET),
189     .read_MI(read_MI5),
190     .imem_in(imem_in),
191     .dmem_in(dmem_in5),
192     .dmem_out(dmem_out5),
193     .write_MD(write_MD5),
194     .ar_out(ar_out5)
195  );
196
197   // [CORE 6]
198 core #(.core_id(5)) processor6 (
199     .clk(clk),
200     .START(START),
201     .RESET(RESET),
202     .read_MI(read_MI6),
203     .imem_in(imem_in),
204     .dmem_in(dmem_in6),
205     .dmem_out(dmem_out6),
206     .write_MD(write_MD6),
207     .ar_out(ar_out6)
208  );
209
210    // [CORE 7]
211 core #(.core_id(6)) processor7 (
```

```
212     .clk(clk),
213     .START(START),
214     .RESET(RESET),
215     .read_MI(read_MI7),
216     .imem_in(imem_in),
217     .dmem_in(dmem_in7),
218     .dmem_out(dmem_out7),
219     .write_MD(write_MD7),
220     .ar_out(ar_out7)
221  );
222
223   // [CORE 8]
224 core #(.core_id(7)) processor8 (
225     .clk(clk),
226     .START(START),
227     .RESET(RESET),
228     .read_MI(read_MI8),
229     .imem_in(imem_in),
230     .dmem_in(dmem_in8),
231     .dmem_out(dmem_out8),
232     .write_MD(write_MD8),
233     .ar_out(ar_out8)
234  );
235
236
237   // [CORE 9]
238 core #(.core_id(8)) processor9 (
239     .clk(clk),
240     .START(START),
241     .RESET(RESET),
242     .read_MI(read_MI9),
243     .imem_in(imem_in),
244     .dmem_in(dmem_in9),
245     .dmem_out(dmem_out9),
246     .write_MD(write_MD9),
247     .ar_out(ar_out9)
248  );
249
250    // [CORE 10]
251 core #(.core_id(9)) processor10 (
252     .clk(clk),
253     .START(START),
254     .RESET(RESET),
255     .read_MI(read_MI10),
256     .imem_in(imem_in),
257     .dmem_in(dmem_in10),
258     .dmem_out(dmem_out10),
259     .write_MD(write_MD10),
260     .ar_out(ar_out10)
261  );
262
263
264 // Initiating Multi Port Memory
265 data_memory_multi_port DM_4(
266     .clk(clk),
```

```
    .we_1(write_to_data_mem),
    .addr_1(d_mem_addr),
    .data_in_1(data_for_mem),
    .data_out_1(dmem_in),
    .we_2(write_MD2),
    .addr_2(ar_out2),
    .data_in_2(dmem_out2),
    .data_out_2(dmem_in2),
    .we_3(write_MD3),
    .addr_3(ar_out3),
    .data_in_3(dmem_out3),
    .data_out_3(dmem_in3),
    .we_4(write_MD4),
    .addr_4(ar_out4),
    .data_in_4(dmem_out4),
    .data_out_4(dmem_in4),
    .we_5(write_MD5),
    .addr_5(ar_out5),
    .data_in_5(dmem_out5),
    .data_out_5(dmem_in5),
    .we_6(write_MD6),
    .addr_6(ar_out6),
    .data_in_6(dmem_out6),
    .data_out_6(dmem_in6),
    .we_7(write_MD7),
    .addr_7(ar_out7),
    .data_in_7(dmem_out7),
    .data_out_7(dmem_in7),
    .we_8(write_MD8),
    .addr_8(ar_out8),
    .data_in_8(dmem_out8),
    .data_out_8(dmem_in8),
    .we_9(write_MD9),
    .addr_9(ar_out9),
    .data_in_9(dmem_out9),
    .data_out_9(dmem_in9),
    .we_10(write_MD10),
    .addr_10(ar_out10),
    .data_in_10(dmem_out10),
    .data_out_10(dmem_in10)
);

endmodule
```

Listing 1: Top Module (Corresponding testbench is top_tb.v

Core Module

```
module core #(parameter core_id = 0) (
    input clk,  //Corrected clock
    input START,
    input RESET,

    //Outputs related with external memories
    input [15:0] dmem_in, imem_in,
```

```
8       output [15:0] dmem_out,
9       output [15:0] ar_out,
10
11      //output control_signals
12      output read_MD, read_MI, write_MD,
13      output end_i
14
15  );
16
17  wire corrected_clk;
18  clock_corrector_module clock_corrector(
19      .start(START),
20      .clk(clk),
21      .end_signal(end_i),
22      .corrected_clk(corrected_clk)
23  );
24
25
26  //Bus
27  wire [15:0] bus;
28
29  wire [15:0] ac_out;
30  wire [15:0] alu_out;
31
32  //Intermediate Control signals
33  wire DREAD, IREAD, DWRITE, BUSMEM, MEMBUSD, MEMBUSI, TRBUS, DRBUS,  PCBUS,
       NBUS,  CBUS, JBUS;
34  wire IDBUS, COUNTBUS,  TP1BUS, TP2BUS, TP3BUS, ICBUS, IBUS, IEBUS, TR2BUS,
       ACBUS, LDAR;
35  wire LDTR, LDDR, LDPC, LDN, LDC, LDIR, LDTP1, LDTP2, LDTP3, LDIC, LDI, LDIE,
        LDTR2;
36  wire LDAC, RSTAR, RSTTR, RSTDR, RSTPC, RSTN, RSTC, RSTIR, RSTJ, RSTID,
       RSTCOUNT, RSTTP1;
37  wire RSTTP2, RSTTP3, RSTIC, RSTI, RSTIE, RSTTR2, RSTAC, RSTEND, INCPC, INCJ,
        INCCOUNT;
38  wire INCTP1, INCTP2, INCI, INCAC, INCEND, ALU0, ALU1, ALU2, ADDKAC;
39
40  //Memory read write outputs from the Core
41  assign read_MD = DREAD;
42  assign read_MI = IREAD;
43  assign write_MD = DWRITE;
44
45  //Buffers to access data and instruction memory
46  buffer MEMBUSD_buf(.data_in(dmem_in), .select(MEMBUSD), .data_out(bus));
47  buffer BUSMEMD_buf(.data_in(bus), .select(BUSMEM), .data_out(dmem_out));
48  buffer MEMBUSI_buf(.data_in(imem_in), .select(MEMBUSI), .data_out(bus));
49
50  //Initialise registers and buffers
51
52  reg_rst_load AR(.clk(clk), .reset(RSTAR), .load_enable(LDAR), .data_in(bus),
        .data_out(ar_out));
53
54  wire [15:0] tr_out;
55  reg_rst_load TR(.clk(clk), .reset(RSTTR), .load_enable(LDTR), .data_in(bus),
        .data_out(tr_out));
```

```
56 buffer TR_buf(.data_in(tr_out), .select(TRBUS), .data_out(bus));
57
58 wire [15:0] dr_out;
59 reg_rst_load DR(.clk(clk), .reset(RSTDR), .load_enable(LDDR), .data_in(bus),
       .data_out(dr_out));
60 buffer DR_buf(.data_in(dr_out), .select(DRBUS), .data_out(bus));
61
62 wire [15:0] pc_out;
63 reg_pc PC(.clk(clk), .reset(RSTPC), .load_enable(LDPC), .inc(INCPC), .
     data_in(dr_out), .data_out(pc_out));
64 buffer PC_buf(.data_in(pc_out), .select(PCBUS), .data_out(bus));
65
66 wire [15:0] n_out;
67 reg_rst_load N(.clk(clk), .reset(RSTN), .load_enable(LDN), .data_in(dr_out),
       .data_out(n_out));
68 buffer N_buf(.data_in(n_out), .select(NBUS), .data_out(bus));
69
70 wire [15:0] c_out;
71 reg_rst_load C(.clk(clk), .reset(RSTC), .load_enable(LDC), .data_in(dr_out),
       .data_out(c_out));
72 buffer C_buf(.data_in(c_out), .select(CBUS), .data_out(bus));
73
74 wire [15:0] ir_out;
75 reg_rst_load IR(.clk(clk), .reset(RSTIR), .load_enable(LDIR), .data_in(
     dr_out), .data_out(ir_out));
76
77 wire [15:0] j_out;
78 reg_rst_inc J(.clk(clk), .reset(RSTJ), .inc(INCJ), .data_out(j_out));
79 buffer J_buf(.data_in(j_out), .select(JBUS), .data_out(bus));
80
81 //Core ID register and buffer
82 wire [15:0] id_out;
83 reg_rst_load ID(.clk(clk), .reset(RSTID), .load_enable(1), .data_in(core_id)
     , .data_out(id_out));
84 buffer ID_buf(.data_in(id_out), .select(IDBUS), .data_out(bus));
85
86 wire [15:0] count_out;
87 reg_rst_inc COUNT(.clk(clk), .reset(RSTCOUNT), .inc(INCCOUNT), .data_out(
     count_out));
88 buffer COUNT_buf(.data_in(count_out), .select(COUNTBUS), .data_out(bus));
89
90 wire [15:0] tp1_out;
91 reg_rst_load_inc TP1(.clk(clk), .reset(RSTTP1), .load_enable(LDTP1), .inc(
     INCTP1), .data_in(ac_out), .data_out(tp1_out));
92 buffer TP1_buf(.data_in(tp1_out), .select(TP1BUS), .data_out(bus));
93
94 wire [15:0] tp2_out;
95 reg_rst_load_inc TP2(.clk(clk), .reset(RSTTP2), .load_enable(LDTP2), .inc(
     INCTP2), .data_in(ac_out), .data_out(tp2_out));
96 buffer TP2_buf(.data_in(tp2_out), .select(TP2BUS), .data_out(bus));
97
98 wire [15:0] tp3_out;
99 reg_rst_load TP3(.clk(clk), .reset(RSTTP3), .load_enable(LDTP3), .data_in(
     ac_out), .data_out(tp3_out));
100 buffer TP3_buf(.data_in(tp3_out), .select(TP3BUS), .data_out(bus));
```

```
101
102 wire [15:0] ic_out;
103 reg_rst_load IC(.clk(clk), .reset(RSTIC), .load_enable(LDIC), .data_in(
        ac_out), .data_out(ic_out));
104 buffer IC_buf(.data_in(ic_out), .select(ICBUS), .data_out(bus));
105
106 wire [15:0] i_out;
107 reg_rst_load_inc I(.clk(clk), .reset(RSTI), .load_enable(LDI), .inc(INCI), .
        data_in(ac_out), .data_out(i_out));
108 buffer I_buf(.data_in(i_out), .select(IBUS), .data_out(bus));
109
110 wire [15:0] ie_out;
111 reg_rst_load IE(.clk(clk), .reset(RSTIE), .load_enable(LDIE), .data_in(
        ac_out), .data_out(ie_out));
112 buffer IE_buf(.data_in(ie_out), .select(IEBUS), .data_out(bus));
113
114 wire [15:0] tr2_out;
115 reg_rst_load TR2(.clk(clk), .reset(RSTTR2), .load_enable(LDTR2), .data_in(
        ac_out), .data_out(tr2_out));
116 buffer TR2_buf(.data_in(tr2_out), .select(TR2BUS), .data_out(bus));
117
118 reg_ac AC( .clk(clk), .inc(INCAC), .inck(ADDKAC), .load_enable(LDAC), .reset
        (RSTAC), .data_in(alu_out), .data_out(ac_out) );
119 buffer AC_buf(.data_in(ac_out), .select(ACBUS), .data_out(bus));
120
121 reg_rst_inc END(.clk(clk), .reset(RSTEND), .inc(INCEND), .data_out(end_i));
122
123 //ALU
124 wire [2:0] operation;
125 assign operation[0] = ALU0;
126 assign operation[1] = ALU1;
127 assign operation[2] = ALU2;
128
129 alu ALU(.in_bus(bus), .in_AC(ac_out), .operation(operation), .data_out(
        alu_out));
130
131 //Z
132 wire z;
133 assign z= ~(ac_out[0]||ac_out[1]||ac_out[2]||ac_out[3]||ac_out[4]||ac_out
        [5]||ac_out[6]||ac_out[7]||ac_out[8]||ac_out[9]||ac_out[10]||ac_out[11]||
        ac_out[12]||ac_out[13]||ac_out[14]||ac_out[15]);
134
135 //Control Unit
136 wire [48:0] OPs;
137 controller controller(.clk(corrected_clk), .IR(ir_out), .z(z), .OPs(OPs));
138
139 //OPS decoder used to obtain control signals
140 ops_decoder decoder(.uOPs(OPs), .START(START), .RESET(RESET), .DREAD(DREAD),
        .IREAD(IREAD), .DWRITE(DWRITE), .BUSMEM(BUSMEM), .MEMBUSD(MEMBUSD), .
        MEMBUSI(MEMBUSI),
141 .TRBUS(TRBUS), .DRBUS(DRBUS), .PCBUS(PCBUS), .NBUS(NBUS), .CBUS(CBUS), .
        JBUS(JBUS), .IDBUS(IDBUS), .COUNTBUS(COUNTBUS), .TP1BUS(TP1BUS), .TP2BUS
        (TP2BUS),
142 .TP3BUS(TP3BUS), .ICBUS(ICBUS), .IBUS(IBUS), .IEBUS(IEBUS), .TR2BUS(TR2BUS),
        .ACBUS(ACBUS), .LDAR(LDAR), .LDTR(LDTR), .LDDR(LDDR), .LDPC(LDPC), .LDN(
```

```
    LDN), .LDC(LDC),
143 .LDIR(LDIR), .LDTP1(LDTP1), .LDTP2(LDTP2), .LDTP3(LDTP3), .LDIC(LDIC), .LDI(
    LDI), .LDIE(LDIE), .LDTR2(LDTR2), .LDAC(LDAC), .RSTAR(RSTAR), .RSTTR(
    RSTTR), .RSTDR(RSTDR),
144 .RSTPC(RSTPC), .RSTN(RSTN), .RSTC(RSTC), .RSTIR(RSTIR), .RSTJ(RSTJ), .RSTID(
    RSTID), .RSTCOUNT(RSTCOUNT), .RSTTP1(RSTTP1), .RSTTP2(RSTTP2), .RSTTP3(
    RSTTP3), .RSTIC(RSTIC),
145 .RSTI(RSTI), .RSTIE(RSTIE), .RSTTR2(RSTTR2), .RSTAC(RSTAC), .RSTEND(RSTEND),
     .INCPC(INCPC), .INCJ(INCJ), .INCCOUNT(INCCOUNT), .INCTP1(INCTP1), .
    INCTP2(INCTP2), .INCI(INCI),
146 .INCAC(INCAC), .INCEND(INCEND), .ALU0(ALU0), .ALU1(ALU1), .ALU2(ALU2), .
    ADDKAC(ADDKAC)
147 );
148
149
150 endmodule
```

Listing 2: Core Module

Multi Port Memory Module

```
1  module data_memory_multi_port
2  (
3    input [15:0] data_in_1, data_in_2, data_in_3, data_in_4, data_in_5,
     data_in_6, data_in_7, data_in_8, data_in_9, data_in_10,
4    input [15:0] addr_1, addr_2, addr_3, addr_4, addr_5, addr_6, addr_7,
     addr_8, addr_9, addr_10,
5    input we_1, we_2, we_3, we_4, we_5, we_6, we_7, we_8, we_9, we_10, clk,
6    output [15:0] data_out_1, data_out_2, data_out_3, data_out_4, data_out_5,
     data_out_6, data_out_7, data_out_8, data_out_9, data_out_10
7  );
8
9  // Declare the RAM variable
10 reg [15:0] ram[0:999];
11
12 assign data_out_1 = ram[addr_1];
13 assign data_out_2 = ram[addr_2];
14 assign data_out_3 = ram[addr_3];
15 assign data_out_4 = ram[addr_4];
16 assign data_out_5 = ram[addr_5];
17 assign data_out_6 = ram[addr_6];
18 assign data_out_7 = ram[addr_7];
19 assign data_out_8 = ram[addr_8];
20 assign data_out_9 = ram[addr_9];
21 assign data_out_10 = ram[addr_10];
22
23
24 always @ (posedge clk)
25   begin
26     if (we_1) ram[addr_1] <= data_in_1;
27     if (we_2) ram[addr_2] <= data_in_2;
28     if (we_3) ram[addr_3] <= data_in_3;
29     if (we_4) ram[addr_4] <= data_in_4;
30     if (we_5) ram[addr_5] <= data_in_5;
31     if (we_6) ram[addr_6] <= data_in_6;
```

```
    if (we_7) ram[addr_7] <= data_in_7;
    if (we_8) ram[addr_8] <= data_in_8;
    if (we_9) ram[addr_9] <= data_in_9;
    if (we_10) ram[addr_10] <= data_in_10;

  end

endmodule
```

Listing 3: Multi Port Data Memory (Corresponding testbench is data_memory_multi_port_tb.v)

Instruction Memory Module

```
module instruction_memory(
  input read,
  input [15:0] address,
  output [15:0] instruction_out
);

//Initiating the instruction memory
reg [15:0] inst_memory [0:99];

//Assembly code
initial
    begin
        inst_memory[0] = 0;       //SETN
        inst_memory[1] = 998 ;    //Address N
        inst_memory[2] = 1 ;      //SETC
        inst_memory[3] = 999;     //Address C
        inst_memory[4] = 5 ;      //RNGI
        inst_memory[5] = 7 ;      //ENDI
        inst_memory[6] = 6 ;      //STRTI
        inst_memory[7] = 24 ;     //RSTJ
        inst_memory[8] = 2  ;     //SETTP1
        inst_memory[9] = 3 ;      //SETTP2
        inst_memory[10] = 4 ;    //SETTP3
        inst_memory[11] = 8  ;   //LDACTP1
        inst_memory[12] = 12;     //MVACTR
        inst_memory[13] = 9 ;    //LDACTP2
        inst_memory[14] = 17 ;    //MUL
        inst_memory[15] = 26 ;    //MVTR2
        inst_memory[16] = 18 ;    //ADD
        inst_memory[17] = 27 ;    //MVACTR2
        inst_memory[18] = 13 ;    //MVCOUNT
        inst_memory[19] = 11 ;    //LDN
        inst_memory[20] = 19 ;    //SUB
        inst_memory[21] = 21;     //JNPZ
        inst_memory[22] = 11 ;    // 11
        inst_memory[23] = 28 ;    //LDTR2
        inst_memory[24] = 20 ;    //STAC
        inst_memory[25] = 14 ;    //MVJ
        inst_memory[26] = 11 ;    //LDN
        inst_memory[27] = 19 ;    //SUB
        inst_memory[28] = 21 ;    //JNPZ
```

```
        inst_memory[29] = 8 ;     //8
        inst_memory[30] = 16 ;    //MVIE
        inst_memory[31] = 15 ;    //MVI
        inst_memory[32] = 19 ;    //SUB
        inst_memory[33] = 21 ;    //JNPZ
        inst_memory[34] = 7  ;    //7
        inst_memory[35] = 25 ;    //END
    end

assign instruction_out = inst_memory[address];


endmodule
```

Listing 4: Instruction Memory Module (Corresponding testbench is instruction_memory_tb.v)

Multiplexer module used for selecting data for data memory from cores and an external file

```
module tb_mux_mem_data (
  input [15:0] in_1,
  input [15:0] in_2,
    input mem_data_select,

  output  [15:0] mux_out
);

assign mux_out = mem_data_select ? in_1 : in_2;

endmodule
```

Listing 5: Multiplexer Module used for selecting data for data memory

Multiplexer module used for selecting the write enable signals coming for data memory from cores and an external file

```
module tb_mux (
  input in_1,
  input in_2,

  output mux_out
);

assign mux_out = in_1 ? in_1 : in_2;

endmodule
```

Listing 6: Multiplexer Module used for selecting write enables signal for data memory

Multiplexer module used for selecting reading /writing addresses for data memory from cores and an external file

```
module dmem_mux(
```

```
  input [15:0] in_core,
  input [15:0] in_d_write,
    input [15:0] in_d_read,
    input [1:0] addr_mux_select,
  output reg [15:0] mux_out
);

always @(in_core or in_d_write or in_d_read or addr_mux_select)
    begin
        case(addr_mux_select)
            0:mux_out    <=   in_core;
            1:mux_out    <=   in_d_write;
            2:mux_out    <=   in_d_read;
            default: mux_out <=   in_core;

        endcase
    end

endmodule
```

Listing 7: Multiplexer Module used for selecting reading/writing addresses for data memory

Control Unit Module

```
module controller(
  input clk,
    input [15:0] IR,
    input z,
  output [48:0] OPs
);


//Map function
wire [15:0] map_addr;

mapping_block mapping_func(
  .IR(IR),
  .map_addr(map_addr)
);

wire [6:0] jump_addr;
wire [15:0] inc_addr;
wire [15:0] mux_out;
wire [15:0] reg_out;
wire [1:0] addr_select;

assign inc_addr = reg_out+1;

mux1_control_unit mux1(
  .map_addr(map_addr),
  .jump_addr(jump_addr),
  .inc_addr(inc_addr),
  .reg_in(mux_out),
  .select(addr_select)
);
```

```
32
33 cu_reg_rst_load register(
34   .clk(clk),
35   .data_in(mux_out),
36   .data_out(reg_out),
37   .load_enable(1),
38   .reset(0)
39 );
40
41 wire [1:0] condition;
42 wire BT;
43
44 microcode micromemory(
45   .reg_out(reg_out),
46   .condition(condition),
47   .BT(BT), .OPs(OPs),
48   .jump_addr(jump_addr)
49 );
50
51 wire [15:0] logic_in;
52
53 mux2_control_unit mux2(
54   .z(z),
55   .select(condition),
56   .logic_in(logic_in)
57 );
58
59 logic_block logic_box(
60   .BT_in(BT),
61   .mux2_out(logic_in),
62   .signal_out(addr_select)
63 );
64
65
66 endmodule
67
68
69
70
71
72
73
```

Listing 8: Control Unit Module (Corresponding testbench is controller_tb.v)

Clock Corrector Module

```
1 module clock_corrector_module(
2     input start,
3     input clk,
4     input end_signal,
5   output corrected_clk
6 );
7
8 reg p =  0;
```

```
9
10 always @(posedge clk)
11   begin
12       if (start)
13       p <= 1;
14      end
15
16 assign corrected_clk = (p && !end_signal) ? clk : 0;
17
18 endmodule
```

Listing 9: Clock Corrector Module (Corresponding testbench is clock_corrector_tb.v)

Arithmethic and Logic Unit Module

```
1  module alu(
2    input [15:0] in_bus,
3    input [15:0] in_AC,
4    input [2:0] operation,
5    output reg [15:0] data_out
6  );
7
8  //Defining symbols to identify logical opeartors
9  localparam pass = 0;
10 localparam mul = 1;
11 localparam add = 2;
12 localparam sub = 3;
13 localparam int_div = 4;
14 localparam is_mod = 5;
15
16
17 always @(in_bus or in_AC or operation)
18   begin
19     case(operation)
20       pass:data_out = in_bus;
21       mul: data_out = in_AC * in_bus;
22       add: data_out = in_AC + in_bus;
23       sub: data_out = in_AC - in_bus;
24       int_div: data_out= (in_AC - in_AC % in_bus)/in_bus;
25       is_mod: data_out= ((in_AC % in_bus) != 0);
26
27       default : data_out = in_bus;
28     endcase
29   end
30 endmodule
```

Listing 10: Arithmetic and Logic Unit Module (Corresponding testbench is alu_tb.v)

Ops Decoder Module

```
1  module ops_decoder(
2    input wire [48:0] uOPs,
3    input START, RESET,
4    //Control signals
```

```
5   output wire DREAD, IREAD, DWRITE, BUSMEM, MEMBUSD, MEMBUSI, TRBUS, DRBUS,
      PCBUS, NBUS,  CBUS, JBUS,
6   output wire IDBUS, COUNTBUS,  TP1BUS, TP2BUS, TP3BUS, ICBUS, IBUS, IEBUS,
     TR2BUS, ACBUS, LDAR,
7   output wire LDTR, LDDR, LDPC, LDN, LDC, LDIR, LDTP1, LDTP2, LDTP3, LDIC,
     LDI, LDIE, LDTR2,
8   output wire LDAC, RSTAR, RSTTR, RSTDR, RSTPC, RSTN, RSTC, RSTIR, RSTJ,
     RSTID, RSTCOUNT, RSTTP1,
9   output wire RSTTP2, RSTTP3, RSTIC, RSTI, RSTIE, RSTTR2, RSTAC, RSTEND,
     INCPC, INCJ, INCCOUNT,
10  output wire INCTP1, INCTP2, INCI, INCAC, INCEND, ALU0, ALU1, ALU2, ADDKAC
11 );
12
13
14 assign DREAD = uOPs[0];
15 assign IREAD = uOPs[1];
16 assign DWRITE = uOPs[10];
17 assign BUSMEM = uOPs[10];
18 assign MEMBUSD = uOPs[0];
19 assign MEMBUSI = uOPs[1];
20 assign TRBUS = uOPs[45]||uOPs[44]||uOPs[34]||uOPs[9]||uOPs[3];
21 assign DRBUS = uOPs[39]||uOPs[27]||uOPs[10];
22 assign PCBUS = uOPs[38];
23 assign NBUS = uOPs[37];
24 assign CBUS = uOPs[18];
25 assign JBUS = uOPs[21];
26 assign IDBUS = uOPs[33];
27 assign COUNTBUS = uOPs[12];
28 assign TP1BUS = uOPs[30];
29 assign TP2BUS = uOPs[29];
30 assign TP3BUS = uOPs[28];
31 assign ICBUS = uOPs[19];
32 assign IBUS = uOPs[17];
33 assign IEBUS = uOPs[36];
34 assign TR2BUS = uOPs[35]||uOPs[20];
35 assign ACBUS = uOPs[22]||uOPs[2];
36 assign LDAR = uOPs[39]||uOPs[38]||uOPs[30]||uOPs[29]||uOPs[28];
37 assign LDTR = uOPs[22]||uOPs[21]||uOPs[20]||uOPs[19]||uOPs[18]||uOPs[17]||
     uOPs[12];
38 assign LDDR = uOPs[2]||uOPs[1]||uOPs[0];
39 assign LDPC = uOPs[25];
40 assign LDN = uOPs[7];
41 assign LDC = uOPs[4];
42 assign LDIR = uOPs[47];
43 assign LDTP1 = uOPs[11];
44 assign LDTP2 = uOPs[24];
45 assign LDTP3 = uOPs[14];
46 assign LDIC = uOPs[48];
47 assign LDI = uOPs[41];
48 assign LDIE = uOPs[43];
49 assign LDTR2 = uOPs[13];
50 assign LDAC = uOPs[45]||uOPs[44]||uOPs[37]||uOPs[36]||uOPs[35]||uOPs[34]||
     uOPs[33]||uOPs[27]||uOPs[9]||uOPs[3];
51 assign RSTAR = RESET;
52 assign RSTTR = RESET;
```

```
assign RSTDR = RESET;
assign RSTPC = RESET;
assign RSTN = RESET;
assign RSTC = RESET;
assign RSTIR = RESET;
assign RSTJ = uOPs[46]||RESET;
assign RSTID = 0; // Check thi
assign RSTCOUNT = uOPs[6]||RESET;
assign RSTTP1 = RESET;
assign RSTTP2 = RESET;
assign RSTTP3 = RESET;
assign RSTIC = RESET;
assign RSTI = RESET;
assign RSTIE = RESET;
assign RSTTR2 = uOPs[16]||RESET;
assign RSTAC = RESET;
assign RSTEND = START||RESET;
assign INCPC = uOPs[26];
assign INCJ = uOPs[8];
assign INCCOUNT = uOPs[5];
assign INCTP1 = uOPs[15];
assign INCTP2 = uOPs[23];
assign INCI = uOPs[42];
assign INCAC = uOPs[31];
assign INCEND = uOPs[40];
assign ALU0 = uOPs[9]||uOPs[3]||uOPs[45];
assign ALU1 = uOPs[34]||uOPs[3];
assign ALU2 = uOPs[44]||uOPs[45];
assign ADDKAC = uOPs[32];


endmodule
```

Listing 11: OPs Decoder Module (Corresponding testbench is ops_decoder_tb.v)

Buffer Module

```
module buffer(
   input [15:0] data_in,
  input select,
  output [15:0] data_out
   );

assign data_out = select ? data_in : 16'bz;

endmodule
```

Listing 12: Buffer Module (Corresponding testbench is buffer_tb.v)

Register Module with Reset and Load

```
module reg_rst_load(
  input clk,
  input reset,
```

```
  input load_enable,
  input [15:0] data_in,

  output reg [15:0] data_out
);

always @(posedge clk)
    begin
        if (load_enable)
            data_out <= data_in;
        if (reset)
            data_out <= 16'b0;
    end

endmodule
```

Listing 13: Register Module with Reset and Load (Corresponding testbench is reg_rst_load_tb.v)

Register Module with Reset and Increment

```
module reg_rst_inc(
  input clk,
  input inc,
  input reset,

  output reg [15:0] data_out
);

always @(posedge clk)
    begin
        if (inc)
            data_out <= data_out + 16'd1;
        if (reset)
            data_out <= 16'b0;
    end

endmodule
```

Listing 14: Register Module with Reset and Increment (Corresponding testbench is reg_rst_inc_tb.v)

Register module with reset, load and increment

```
module reg_rst_load_inc(
  input clk,
  input inc,
  input load_enable,
  input reset,
    input [15:0] data_in,

    output reg [15:0] data_out
);

```

```
always @(posedge clk)
    begin
        if (load_enable)
            data_out <= data_in;
        else if (inc)
            data_out <= data_out + 16'd1;
        if (reset)
            data_out <= 16'b0;
    end

endmodule
```

Listing 15: Register module with reset

AC register module

```
module reg_ac(
  input clk,
  input inc,
  input inck,
  input load_enable,
  input reset,

    input [15:0] data_in,
  output reg [15:0] data_out
);

localparam k=332;

always @(posedge clk)
    begin
        if (load_enable)
            data_out <= data_in;
        if (inc)
            data_out <= data_out + 16'd1;
        if (inck)
            data_out <= data_out + k;
        if (reset)
            data_out <= 16'b0;
    end

endmodule
```

Listing 16: AC register (Corresponding testbench is reg_ac_tb.v)

PC register module

```
module reg_pc(
  input clk,
  input inc,
  input load_enable,
  input reset,

    input [15:0] data_in,
```

```
  output reg [15:0] data_out
);

always @(posedge clk)
    begin
        if (load_enable)
            data_out <= data_in;
        if (inc)
            data_out <= data_out + 16'd1;
        if (reset)
            data_out <= 16'b0;
    end

initial begin
    data_out = 0;
end

endmodule
```

Listing 17: PC register (Corresponding testbench is reg_pc_tb.v)

Module that maps instructions with addresses

```
module mapping_block(
  input [15:0] IR,
    output reg [15:0] map_addr
);

//Instructions
localparam SETN = 0;
localparam SETC = 1;
localparam SETTP1 = 2;
localparam SETTP2 = 3;
localparam SETTP3 = 4;
localparam RNGI = 5;
localparam STRTI = 6;
localparam ENDI = 7;
localparam LDACTP1 = 8;
localparam LDACTP2 = 9;
localparam LDACTP3 = 10;
localparam LDN = 11;
localparam LDTR2 = 28;
localparam MVACTR = 12;
localparam MVACTR2 = 27;
localparam MVCOUNT = 13;
localparam MVJ = 14;
localparam MVI = 15;
localparam MVIE = 16;
localparam MVTR2 = 26;
localparam STAC = 20;
localparam JNPZ = 21;
localparam JNPZY = 22;
localparam JNPZN = 23;
localparam RSTJ = 24;
localparam END = 25;
```

```
localparam MUL_ = 17;
localparam ADD_ = 18;
localparam SUB_ = 19;

//Mapping algorithm
always @(IR)
  begin
    case(IR)
      SETN: map_addr <= 3;
      SETC: map_addr <= 7;
      SETTP1: map_addr <= 11;
      SETTP2: map_addr <= 15;
      SETTP3: map_addr <= 20;
      RNGI: map_addr <= 28;
      STRTI: map_addr <= 37;
      ENDI: map_addr <= 41;
      LDACTP1: map_addr <= 45;
      LDACTP2: map_addr <= 48;
      LDACTP3: map_addr <= 51;
      LDN: map_addr <= 54;
      LDTR2: map_addr <= 55;
      MVACTR: map_addr <= 56;
      MVACTR2: map_addr <= 57;
      MVCOUNT: map_addr <= 58;
      MVJ: map_addr <= 59;
      MVI: map_addr <= 60;
      MVIE: map_addr <= 61;
      MVTR2: map_addr <= 62;
      STAC: map_addr <= 63;
      JNPZ: map_addr <= 66;
      JNPZY: map_addr <=67;
      JNPZN: map_addr <= 69;
      RSTJ: map_addr <= 70;
      END: map_addr <= 71;
      MUL_: map_addr <= 72;
      ADD_: map_addr <= 73;
      SUB_: map_addr <= 74;
    endcase
  end

endmodule
```

Listing 18: Module that maps instructions with addresses (Corresponding testbench is mapping_block_tb.v)

Module that contains micro instructions

```
module microcode(
    input [15:0] reg_out,
  output reg [1:0] condition,
  output reg BT,
  output reg [48:0] OPs,
  output reg [6:0] jump_addr
);

```

```
reg [58:0] ucode [0:74];

initial
    begin
    ucode[0] = {3'b000,7'd1,49'd274877906944};
    ucode[1] = {3'b000,7'd2,49'd67108866};
    ucode[2] = {3'b100,7'd0,49'd141012366262272};
    ucode[3] = {3'b000,7'd4,49'd67108866};
    ucode[4] = {3'b000,7'd5,49'd549755813888};
    ucode[5] = {3'b000,7'd6,49'd1};
    ucode[6] = {3'b000,7'd0,49'd128};
    ucode[7] = {3'b000,7'd8,49'd67108866};
    ucode[8] = {3'b000,7'd9,49'd549755813888};
    ucode[9] = {3'b000,7'd10,49'd1};
    ucode[10] = {3'b000,7'd0,49'd16};
    ucode[11] = {3'b000,7'd12,49'd137438953472};
    ucode[12] = {3'b000,7'd13,49'd131072};
    ucode[13] = {3'b000,7'd14,49'd512};
    ucode[14] = {3'b000,7'd0,49'd2048};
    ucode[15] = {3'b000,7'd16,49'd137438953472};
    ucode[16] = {3'b000,7'd17,49'd2097152};
    ucode[17] = {3'b000,7'd18,49'd512};
    ucode[18] = {3'b000,7'd19,49'd4294967296};
    ucode[19] = {3'b000,7'd0,49'd16777216};
    ucode[20] = {3'b000,7'd21,49'd137438953472};
    ucode[21] = {3'b000,7'd22,49'd131072};
    ucode[22] = {3'b000,7'd23,49'd512};
    ucode[23] = {3'b000,7'd24,49'd2097152};
    ucode[24] = {3'b000,7'd25,49'd17179869184};
    ucode[25] = {3'b000,7'd26,49'd4294967296};
    ucode[26] = {3'b000,7'd27,49'd4294967296};
    ucode[27] = {3'b000,7'd0,49'd16448};
    ucode[28] = {3'b000,7'd29,49'd137438953472};
    ucode[29] = {3'b000,7'd30,49'd262144};
    ucode[30] = {3'b000,7'd31,49'd17592186044416};
    ucode[31] = {3'b000,7'd32,49'd8192};
    ucode[32] = {3'b000,7'd33,49'd137438953472};
    ucode[33] = {3'b000,7'd34,49'd35184372088832};
    ucode[34] = {3'b000,7'd35,49'd1048576};
    ucode[35] = {3'b000,7'd36,49'd17179869184};
    ucode[36] = {3'b000,7'd0,49'd281474976776192};
    ucode[37] = {3'b000,7'd38,49'd524288};
    ucode[38] = {3'b000,7'd39,49'd8589934592};
    ucode[39] = {3'b000,7'd40,49'd512};
    ucode[40] = {3'b000,7'd0,49'd2199023255552};
    ucode[41] = {3'b000,7'd42,49'd8589934592};
    ucode[42] = {3'b000,7'd43,49'd2148007936};
    ucode[43] = {3'b000,7'd44,49'd512};
    ucode[44] = {3'b000,7'd0,49'd8796093022208};
    ucode[45] = {3'b000,7'd46,49'd1073741856};
    ucode[46] = {3'b000,7'd47,49'd1};
    ucode[47] = {3'b000,7'd0,49'd134217728};
    ucode[48] = {3'b000,7'd49,49'd536870912};
    ucode[49] = {3'b000,7'd50,49'd1};
    ucode[50] = {3'b000,7'd0,49'd134217728};
```

```
    ucode[51] = {3'b000,7'd52,49'd268435456};
    ucode[52] = {3'b000,7'd53,49'd1};
    ucode[53] = {3'b000,7'd0,49'd134217728};
    ucode[54] = {3'b000,7'd0,49'd137438953472};
    ucode[55] = {3'b000,7'd0,49'd34359738368};
    ucode[56] = {3'b000,7'd0,49'd4194304};
    ucode[57] = {3'b000,7'd0,49'd8192};
    ucode[58] = {3'b000,7'd0,49'd8425472};
    ucode[59] = {3'b000,7'd0,49'd2097152};
    ucode[60] = {3'b000,7'd0,49'd131072};
    ucode[61] = {3'b000,7'd0,49'd4466765987840};
    ucode[62] = {3'b000,7'd0,49'd1048576};
    ucode[63] = {3'b000,7'd64,49'd268500992};
    ucode[64] = {3'b000,7'd65,49'd4};
    ucode[65] = {3'b000,7'd0,49'd1280};
    ucode[66] = {3'b001,7'd69,49'd0};
    ucode[67] = {3'b000,7'd68,49'd2};
    ucode[68] = {3'b000,7'd0,49'd33554432};
    ucode[69] = {3'b000,7'd0,49'd67108864};
    ucode[70] = {3'b000,7'd0,49'd70368744177664};
    ucode[71] = {3'b000,7'd0,49'd1099511627776};
    ucode[72] = {3'b000,7'd0,49'd512};
    ucode[73] = {3'b000,7'd0,49'd17179869184};
    ucode[74] = {3'b000,7'd0,49'd8};

  end

//Issuing first micro code at the start
initial {BT, condition, jump_addr, OPs} = ucode[0];

always @(reg_out)
  {BT, condition, jump_addr, OPs}= ucode[reg_out];

endmodule
```

Listing 19: Module that contains micro instructions (Corresponding testbench is microcode_tb.v)

Register Module in Control Unit

```
module cu_reg_rst_load(
  input clk,
  input reset,
  input load_enable,
  input [15:0] data_in,
  output reg [15:0] data_out
);

always @(negedge clk)
    begin
        if (load_enable)
            data_out <= data_in;
        if (reset)
            data_out <= 16'b0;
    end
```

```
16
17 endmodule
```

Listing 20: Register Module in Control Unit

Multiplexer module in control unit that select addresses accordingly

```
1  module mux1_control_unit(
2    input [15:0] map_addr,
3    input [6:0] jump_addr,
4    input [15:0] inc_addr,
5    input [1:0] select,
6    output reg [15:0] reg_in
7  );
8
9  always @(map_addr or jump_addr or inc_addr or select)
10     begin
11         case(select)
12             0:reg_in    <=   inc_addr;
13             1:reg_in    <=   jump_addr;
14             2:reg_in    <=   map_addr;
15             default: reg_in <=   jump_addr;
16         endcase
17     end
18
19 endmodule
```

Listing 21: Multiplexer module in control unit that select addresses accordingly (Corresponding testbench is mux1_control_unit_tb.v)

Multiplexer module in control unit that select logic values accordingly

```
1  module mux2_control_unit(
2    input z,
3    input [1:0] select,
4    output reg logic_in
5  );
6
7  always @(z or select)
8    begin
9      case(select)
10       0:logic_in=1;
11       1:logic_in=z;
12       2:logic_in=~z;
13       default:logic_in= 1;
14     endcase
15   end
16
17 endmodule
```

Listing 22: Multiplexer module in control unit that select logic values accordingly

Logic module in control unit

```
module logic_block(
    input BT_in,
    input mux2_out,
  output [1:0] signal_out
);

assign signal_out[1] = BT_in;
assign signal_out[0]= ~BT_in && mux2_out;

endmodule
```

Listing 23: Logic module in control unit (Corresponding testbench is logic_block_tb.v)

Test Benches

Top module testbench

```
`timescale 1 ns/10 ps
module top_tb;

    //For file reading and writing
  integer data_out;
    integer data_file;
    integer scan_file;
    integer i;


    //Registers used for reading an external file
    reg [15:0] captured_data;
    `define NULL 0
    reg [15:0] mem_data;
  reg [15:0] current_addr = 16'd0;
    reg [15:0] dmem_size = 16'd1000;
    reg write_from_tb;

    //Ports for Top module
    reg clk;
    reg START;
    reg RESET;
    wire END;
  wire [15:0] DMEMBUS;

    //Used to take data out from data memory
    reg [15:0] ar_out_start_addr;
    reg [15:0] end_addr = 16'd997;
    reg [1:0] addr_mux_select;

    //Initiating the clock
    localparam CLK_PERIOD = 10;

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

```
38
39      //Initiating the TOP module
40      top top_module(
41          .clk(clk),
42          .START(START),
43          .RESET(RESET),
44          .END(END),
45          .addr_mux_select(addr_mux_select),
46          .ar_in(ar_out_start_addr),
47          .current_addr(current_addr),
48          .write_from_tb(write_from_tb),
49          .mem_data(mem_data),
50      .dmem_out_disp(DMEMBUS)
51      );
52
53      //Used to calculate time for matrix multiplication
54      realtime capture = 0.0;
55
56      initial begin
57
58          // Opening the file that contains matrix data
59          data_file = $fopen("../../io_txt_files/data_to_mem.txt", "r");
60
61          if (data_file == `NULL) begin
62              $display("data_file handle was NULL");
63              $finish;
64          end
65
66          //Commanding the data memory to use the address provided by
    testbench
67          #10 addr_mux_select <= 1;
68
69          for (current_addr = 0; current_addr < dmem_size+1; current_addr =
    current_addr+1 ) begin
70              @(posedge clk) begin
71
72              scan_file = $fscanf(data_file, "%d\n", captured_data);
73
74                  if (!$feof(data_file)) begin
75                      #10     mem_data <= captured_data;
76                      #5      write_from_tb <=1;
77                      #20     write_from_tb <= 0;
78                      $display(mem_data);
79                  end
80              end
81
82          end
83
84          $fclose(data_file);
85
86          $display("Reading data - completed!");
87
88          //Commanding the data memory to use the addresses and data provided
    by the internal testbenches
89          #10 write_from_tb = 0; addr_mux_select <= 0;
```

```
90
91
92          //Starting Core operations
93          #200
94          capture = $realtime;
95
96          //Issuing start signal to start the cores
97          START = 1; RESET = 0; addr_mux_select = 0;
98          #10 START = 0; RESET = 0; addr_mux_select = 0;
99
100      //Waiting untill all core operations are completed
101         while (END!=1) #10;
102
103         $display("Time for multiplication (us): %t", ($realtime - capture)
    /1000000);
104
105         // Opening a file to write the results
106         data_out = $fopen("../../io_txt_files/results_from_mem.txt","w");
107      $display("File Opened - results_from_mem.txt");
108
109         //Point the AR to the starting location of data memory
110         #10 ar_out_start_addr=16'd0;
111         #10  addr_mux_select <= 2;
112
113         //Writing data from data memory to file line by line
114         while( ar_out_start_addr < end_addr) begin
115                 @(posedge clk);
116                     #50 $fwrite(data_out,"%d\n",DMEMBUS);
117                     #50 ar_out_start_addr <= ar_out_start_addr + 16'd1;  //
    check AR
118                     #10;
119             end
120
121         $display("Writing Results - completed!");
122
123         #20;
124         $fclose(data_out);
125
126         #20;
127         $stop;
128      end
129
130
131
132 endmodule
```

Listing 24: Top module testbench

Data memory testbench

```
1 `timescale 1 ns/10 ps
2
3 module data_memory_multi_port_tb;
4
5     reg [15:0] data_in_1;
```

```
  reg [15:0] data_in_2;
  reg [15:0]data_in_3;
  reg [15:0]data_in_4;
  reg [15:0] addr_1;
  reg [15:0] addr_2;
  reg [15:0]addr_3;
  reg [15:0]addr_4;
  reg we_1;
  reg we_2;
  reg we_3;
  reg we_4;
  reg clk;
  wire [15:0] data_out_1;
  wire [15:0] data_out_2;
  wire [15:0] data_out_3;
  wire [15:0] data_out_4;

    localparam CLK_PERIOD = 10;

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

    data_memory_multi_port multiport_mem(
        .data_in_1(data_in_1),
        .data_in_2(data_in_2),
        .data_in_3(data_in_3),
        .data_in_4(data_in_4),
        .addr_1(addr_1),
        .addr_2(addr_2),
        .addr_3(addr_3),
        .addr_4(addr_4),
        .we_1(we_1),
        .we_2(we_2),
        .we_3(we_3),
        .we_4(we_4),
        .clk(clk),
        .data_out_1(data_out_1),
        .data_out_2(data_out_2),
        .data_out_3(data_out_3),
        .data_out_4(data_out_4)
    );

initial begin

    #10 we_1 = 1;  we_2 = 0; we_3 = 0; we_4 = 0; data_in_1= 1; addr_1 = 10;
   data_in_2= 0; addr_2 = 0; data_in_3= 0; addr_3 = 0; data_in_4= 0; addr_4
   = 0;

    #10 we_1 = 1;  we_2 = 1; we_3 = 1; we_4 = 1; data_in_1= 1; addr_1 = 10;
   data_in_2= 2; addr_2 = 20; data_in_3= 3; addr_3 = 30; data_in_4= 4;
   addr_4 = 40;

    #10 we_1 = 0;  we_2 = 0; we_3 = 0; we_4 = 0; data_in_1= 0; addr_1 = 20;
```

```
    data_in_2= 2; addr_2 = 20; data_in_3= 3; addr_3 = 30; data_in_4= 4;
    addr_4 = 40;

end

endmodule
```

Listing 25: Data memory testbench

Instruction memory testbench

```
`timescale 1 ns/10 ps

module instruction_memory_tb;

    reg read;
    reg [15:0] address;
    wire [15:0] instruction_out;

    instruction_memory IM(.read(read), .address(address), .instruction_out(
    instruction_out));

    initial
        begin
        #10 read <= 1 ; address <= 0;
        #10 read <= 1 ; address <= 1;
        #10 read <= 1 ; address <= 2;
        #10 read <= 1 ; address <= 3;
        #10 read <= 1 ; address <= 4;
        #10 read <= 1 ; address <= 5;
        #10 read <= 1 ; address <= 6;
        #10 read <= 1 ; address <= 7;
        #10 read <= 1 ; address <= 8;
        #10 read <= 1 ; address <= 9;
        #10 read <= 1 ; address <= 10;
        #10 read <= 1 ; address <= 11;
        #10 read <= 1 ; address <= 12;
        #10 read <= 1 ; address <= 13;
        #10 read <= 1 ; address <= 14;
        #10 read <= 1 ; address <= 15;
        #10 read <= 1 ; address <= 16;
        #10 read <= 1 ; address <= 17;
        #10 read <= 1 ; address <= 18;
        #10 read <= 1 ; address <= 19;
        #10 read <= 1 ; address <= 20;
        #10 read <= 1 ; address <= 21;
        #10 read <= 1 ; address <= 22;
        #10 read <= 1 ; address <= 23;
        #10 read <= 1 ; address <= 24;
        #10 read <= 1 ; address <= 25;
        #10 read <= 1 ; address <= 26;
        #10 read <= 1 ; address <= 27;
        #10 read <= 1 ; address <= 28;
        #10 read <= 1 ; address <= 29;
        #10 read <= 1 ; address <= 30;
```

```
        #10 read <= 1 ; address <= 31;
        #10 read <= 1 ; address <= 32;
        #10 read <= 1 ; address <= 33;
        #10 read <= 1 ; address <= 34;
        #10 read <= 1 ; address <= 35;

        end

endmodule
```

Listing 26: Instruction memory testbench

Buffer testbench

```
`timescale 1 ns/10 ps

module buffer_tb;

    reg [15:0] data_in;
    reg select;
    wire [15:0] data_out;

    buffer tri_state_buffer(
        .data_in(data_in),
        .select(select),
        .data_out(data_out)
    );

    initial begin
        #10 data_in = 0; select = 0;
        #10 data_in = 1; select = 0;
        #10 data_in = 0; select = 1;
        #10 data_in = 1; select = 1;
    end

endmodule
```

Listing 27: Buffer testbench

OPs decoder testbench

```
`timescale 1 ns/10 ps

module ops_decoder_tb;

    reg [48:0] OPs;
    reg START, RESET;
  wire DREAD,IREAD,DWRITE,BUSMEM,MEMBUSI;

    ops_decoder decoder(
        .uOPs(OPs),
        .START(START),
        .RESET(RESET),
        .DREAD(DREAD),
```

```
        .IREAD(IREAD),
        .DWRITE(DWRITE),
        .BUSMEM(BUSMEM),
        .MEMBUSI(MEMBUSI)
    );

    initial
        begin
            #10 OPs <= 0;
            #10 OPs <= 1;
            #10 OPs <= 2;
            #10 OPs <= 2678;
        end

endmodule
```

Listing 28: OPs decoder testbench

Clock corrector module testbench

```
`timescale 1 ns/10 ps

module clock_corrector_tb;

    reg start;
    reg clk;
  wire corrected_clk;

    clock_corrector_module clock_corrector(
        .start(start),
        .clk(clk),
        .corrected_clk(corrected_clk)
    );

    localparam CLK_PERIOD = 10;

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

    initial
        begin
            #20 start <= 1 ;
        end

endmodule
```

Listing 29: Clock corrector module testbench

Register module with reset and load testbench

```
`timescale 1 ns/10 ps

```

```
module reg_rst_load_tb;

    reg clk;
    reg reset;
  reg load_enable;
    reg [15:0] data_in;
  wire [15:0] data_out;

  reg_rst_load register(
        .clk(clk),
        .reset(reset),
        .load_enable(load_enable),
        .data_in(data_in),
        .data_out(data_out)
    );

    localparam CLK_PERIOD = 10;

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

  initial begin
       #10 reset <= 0 ; load_enable <= 0; data_in <= 16'd2;

       #10 reset <= 0 ; load_enable <= 1; data_in <= 16'd2;

       #10 reset <= 0 ; load_enable <= 1; data_in <= 16'd5;

       #10 reset <= 1 ; load_enable <= 1; data_in <= 16'd2;

  end

endmodule
```

Listing 30: Register module with reset and load testbench

Register module with reset and increment testbench

```
`timescale 1 ns/10 ps

module reg_rst_inc_tb;

    reg clk;
    reg reset;
    reg inc;

  wire [15:0] data_out;

  reg_rst_inc register(
        .clk(clk),
        .reset(reset),
        .inc(inc),
        .data_out(data_out)
```

```
    );

    localparam CLK_PERIOD = 10;

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


  initial begin
       #10 reset <= 1 ; inc <= 0;

       #10 reset <= 0 ; inc <= 1;

       #10 reset <= 0 ; inc <= 1;

  end

endmodule
```

Listing 31: Register module with reset and increment testbench

AC Register testbench

```
`timescale 1 ns/10 ps

module reg_ac_tb;

    reg clk;
    reg inc;
   reg inck;
    reg load_enable;
    reg reset;

    reg [15:0] data_in;
  wire [15:0] data_out;

  reg_ac register(
        .clk(clk),
        .inc(inc),
    .inck(inck),
        .load_enable(load_enable),
        .reset(reset),
        .data_in(data_in),
        .data_out(data_out)
    );

    localparam CLK_PERIOD = 10;

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

```

```
31
32    initial begin
33        #10 reset <= 0 ; load_enable <= 0; data_in <= 16'd2; inc <= 0; inck
      <= 0;
34
35        #10 reset <= 0 ; load_enable <= 1; data_in <= 16'd2; inc <= 0; inck
      <= 0;
36
37        #10 reset <= 0 ; load_enable <= 0; data_in <= 16'd2; inc <= 1; inck
      <= 0;
38
39        #10 reset <= 1 ; load_enable <= 0; data_in <= 16'd2; inc <= 1; inck
      <= 0;
40
41      #10 reset <= 0 ; load_enable <= 0; data_in <= 16'd2; inc <= 0; inck <=
      1;
42
43    end
44
45  endmodule
```

Listing 32: AC Register testbench

PC Register testbench

```
1  `timescale 1 ns/10 ps
2
3  module reg_pc_tb;
4
5      reg clk;
6      reg inc;
7      reg load_enable;
8      reg reset;
9
10     reg [15:0] data_in;
11   wire [15:0] data_out;
12
13   reg_pc register(
14         .clk(clk),
15         .reset(reset),
16         .load_enable(load_enable),
17         .inc(inc),
18         .data_in(data_in),
19         .data_out(data_out)
20     );
21
22     localparam CLK_PERIOD = 10;
23
24     initial begin
25         clk = 0;
26         forever #(CLK_PERIOD/2) clk <= ~clk;
27     end
28
29   initial begin
30       #50 reset <= 0 ; inc <= 1; data_in <= 0; load_enable = 0;
```

```
31
32        #10 reset <= 0 ; inc <= 0; data_in <= 0; load_enable = 0;
33
34        #10 reset <= 1 ; inc <= 0; data_in <= 0; load_enable = 0;
35    end
36
37 endmodule
```

Listing 33: PC Register testbench

Arithmetic and logic unit testbench

```
1  `timescale 1 ns/10 ps
2
3  module alu_tb;
4
5    reg [15:0] in_bus;
6    reg [15:0] in_AC;
7    reg [2:0] operation;
8    wire [15:0] data_out;
9
10   alu alu_test(
11   .in_bus(in_bus),
12     .in_AC(in_AC),
13     .operation(operation),
14     .data_out(data_out)
15     );
16
17   initial begin
18     #100 in_bus=15'd2 ; in_AC= 15'd5; operation=0 ;
19     #100 in_bus=15'd2 ; in_AC= 15'd5; operation=1 ;
20     #100 in_bus=15'd2 ; in_AC= 15'd5; operation=2 ;
21     #100 in_bus=15'd2 ; in_AC= 15'd5; operation=3 ;
22     #100 in_bus=15'd2 ; in_AC= 15'd5; operation=4 ;
23     #100 in_bus=15'd2 ; in_AC= 15'd5; operation=5 ;
24
25   end
26
27 endmodule
```

Listing 34: Arithmetic and logic unit testbench

Control unit testbench

```
1  `timescale 1 ns/10 ps
2
3  module controller_tb;
4
5
6     reg clk;
7     reg [15:0] IR;
8     reg z;
9    wire [48:0] OPs;
10
```

```
    localparam CLK_PERIOD = 10;

    controller controller(.clk(clk), .IR(IR), .z(z), .OPs(OPs));

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


    initial begin
       #10 IR <= 0 ; z <= 0;
       #10 IR <= 21 ; z <= 0;
       #10 IR <= 21 ; z <= 0;
       #10 IR <= 21 ; z <= 0;

  end


endmodule
```

Listing 35: Control testbench

Microcode testbench

```
`timescale 1 ns/10 ps

module microcode_tb;

   reg [15:0] reg_out;
   wire [1:0] condition;
   wire BT;
   wire [58:0] OPs;
   wire [6:0] jump_addr;

   microcode micromemory(
      .reg_out(reg_out),
      .condition(condition),
      .BT(BT),
      .OPs(OPs),
      .jump_addr(jump_addr)
   );

   initial begin

     #10 reg_out= 15'd0;
     #10 reg_out= 15'd1;
     #10 reg_out= 15'd2;
     #10 reg_out= 15'd3;
     #10 reg_out= 15'd4;
     #10 reg_out= 15'd5;
     #10 reg_out= 15'd6;

end

```

```
endmodule
```

Listing 36: Microcode testbench

Multiplexer testbench

```
`timescale 1 ns/10 ps

module mux1_control_unit_tb;

    reg [15:0] map_addr;
  reg [6:0] jump_addr;
    reg [15:0] inc_addr;
    reg [1:0] select;
  wire [15:0] reg_in;

  mux1_control_unit mux1_control_unit(
        .map_addr(map_addr),
        .jump_addr(jump_addr),
        .inc_addr(inc_addr),
        .select(select),
        .reg_in(reg_in)
    );


  initial begin
        #10 map_addr=1 ; jump_addr= 45; inc_addr= 3 ; select= 0 ;
    #10 map_addr=1 ; jump_addr= 45; inc_addr= 3 ; select= 1 ;
    #10 map_addr=1 ; jump_addr= 45; inc_addr= 3 ; select= 2 ;

  end

endmodule
```

Listing 37: Multiplexer testbench

Logic block testbench

```
`timescale 1 ns/10 ps

module logic_block_tb;

    reg BT_in;
    reg mux2_out;
  wire [1:0] signal_out;

    logic_block logic(.BT_in(BT_in), .mux2_out(mux2_out), .signal_out(
   signal_out));

    initial
        begin
            #10 BT_in <= 0 ; mux2_out <= 0;
            #10 BT_in <= 0 ; mux2_out <= 1;
            #10 BT_in <= 1 ; mux2_out <= 0;
```

```
        end

endmodule
```

Listing 38: Logic block testbench

Mapping block testbench

```
`timescale 1 ns/10 ps

module mapping_block_tb;

    reg [15:0] IR;
    wire [15:0] map_addr;

    mapping_block map_function(
        .IR(IR),
        .map_addr(map_addr)
    );

    initial begin
        #10 IR = 0;
        #10 IR = 1;
        #10 IR = 2;
        #10 IR = 3;
        #10 IR = 4;
        #10 IR = 5;
        #10 IR = 6;
        #10 IR = 7;
        #10 IR = 8;
        #10 IR = 9;
        #10 IR = 10;
        #10 IR = 11;
        #10 IR = 12;
        #10 IR = 13;
        #10 IR = 14;
        #10 IR = 15;
        #10 IR = 16;
        #10 IR = 17;
        #10 IR = 18;
        #10 IR = 19;
        #10 IR = 20;
        #10 IR = 21;
        #10 IR = 22;
        #10 IR = 23;
        #10 IR = 24;
        #10 IR = 25;
        #10 IR = 26;
        #10 IR = 27;
        #10 IR = 28;
    end

endmodule
```

Listing 39: Mapping block testbench

Python codes

Code for rearranging user data into algorithm expected format

```
1
2  import numpy as np
3  import argparse
4
5  from numpy.core.fromnumeric import size
6
7  def pad_matrix(matrix1, matrix2, n):
8    n1, n2 = matrix1.shape
9    n3, n4 =  matrix2.shape
10
11   #n_max= max(n1, n2, n3, n4)
12
13   matrix1_pad= np.zeros((n,n))
14   matrix2_pad= np.zeros((n,n))
15
16   matrix1_pad[:n1, :n2]= matrix1
17   matrix2_pad[:n3, :n4]= matrix2
18
19   return matrix1_pad, matrix2_pad
20
21
22 def save_matrix(matrix1, matrix2, mem_size=1000, K=332, n=3, c=1,
       write_filename= 'out.txt'):
23   matrix1 = np.array(matrix1)
24   matrix2 = np.array(matrix2)
25
26   matrix1, matrix2 = pad_matrix(matrix1, matrix2, n)
27
28   matrix1_flat = matrix1.flatten().astype('int')
29   matrix2_flat = matrix2.T.flatten().astype('int')
30
31   to_mem=np.zeros((mem_size+1,)).astype('int')
32   for idx in range(0, n*n):
33     to_mem[idx] = matrix1_flat[idx]
34
35   for idx in range(0, n*n):
36     to_mem[K+idx] = matrix2_flat[idx]
37
38   to_mem[-2]=c
39   to_mem[-3]=n
40
41   to_mem= '\n'.join(to_mem.astype('str'))
42
43   with open(write_filename, 'w') as f:
44     f.write(to_mem)
45   print(f'written to {write_filename}')
46
47
48 parser = argparse.ArgumentParser(description='Write matrices')
49 # parser.add_argument('--matrix1', type=str)
```

```
# parser.add_argument('--matrix2', type=str)
parser.add_argument('--mem_size', type=int, default= 1000)
parser.add_argument('--K', type= int, default= 332)
parser.add_argument('--n', type= int)
parser.add_argument('--c', type= int)
parser.add_argument('--filename', type=str, default= 'data_to_mem.txt')

args = parser.parse_args()


matrix1 = np.random.randint(20, size= (args.n,args.n))
matrix2 = np.random.randint(20, size= (args.n,args.n))


save_matrix(matrix1,matrix2,args.mem_size, args.K, args.n, args.c, args.
    filename)
```

Listing 40: save_matrix.py

Code for displaying matrix output

```

import numpy as np
import argparse

def check_matrices(file_name= 'results_from_mem.txt', n=3, K=100):
  with open(file_name) as f:
    data = f.read()

  all_data = data.split('\n')[:-1]

  matrix1  = np.array([int(x) for x in all_data[0:n*n]]).reshape(n,n)
  matrix2  = np.array([int(x) for x in all_data[K:K+n*n]]).reshape(n,n).T
  out_matrix = np.array([int(x) for x in all_data[2*K:2*K+n*n]]).reshape(n,n
    )

  correct_matrix= np.matmul(matrix1, matrix2)

  print(f'matrix 1: \n{matrix1}\nmatrix 2: \n{matrix2}\n\nCalculated Answer:
    \n{out_matrix}\nCorrect Answer: \n{correct_matrix}')
  print(f'\nIs answer correct : {(correct_matrix== out_matrix).all()}')

parser = argparse.ArgumentParser(description='Check matrices')
parser.add_argument('--filename', type=str, default= 'results_from_mem.txt')
parser.add_argument('--n', type=int)
parser.add_argument('--K', type= int, default= 332)

args = parser.parse_args()

check_matrices(args.filename, args.n, args.K)
```

Listing 41: check_matrix.py