

DEPARTMENT OF ELECTRONIC & TELECOMMUNICATION ENGINEERING
UNIVERSITY OF MORATUWA

EN3030: Circuits and Systems Design



FPGA BASED PROCESSOR DESIGN PROJECT

Final Report

	NAME	INDEX
1.	P.K.G.S.C Jayathilake	170268R
2.	P.C.G. Mahiepala	170368A
3.	M.T.L Mannapperuma	170373J
4.	M.K.T Sampath	170543G

Supervisor

Dr. Jayathu Samarawickrama

This report is submitted in partial fulfillment of the requirements
for the module *EN3030 Circuits and Systems Design*

July 7, 2021

Abstract

Task-oriented processor designing has increased dramatically in the last few years. Applications, such as computer vision, wireless base-band, neural networking requires more computational power and unique set of ability set. By designing a custom processor, those specific requirements can be addressed more effectively and adopt perfect architectures to achieve optimum performance on those specific aspects.

Almost every resource-consuming application required matrix multiplication. Design a custom processor to optimized matrix multiplication is the objective of this project. We proposed a multi-core processor with a special parallelization algorithm to speed up the calculation. In report discuss the architecture, components, and algorithms of the custom processor we proposed.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Central Processing Unit (CPU)	1
1.3	Microprocessor	1
1.4	Problem Statement	2
1.5	Proposed Solution Based on FPGA	2
1.6	Required Algorithms.	2
2	Designing a FPGA based custom processor	3
2.1	FPGA - An Overview	3
2.2	Data Paths	3
2.2.1	Single Core	3
2.2.2	Multi Core	4
2.3	The ISA - The Instruction Set Architecture	4
2.3.1	Instruction Set	4
2.3.2	State Diagram	6
3	Module and Components	7
3.1	Clock Module	7
3.2	AC	8
3.3	BUS Multiplexer	9
3.4	Control Unit (CU)	10
3.5	Decoder	12
3.6	DRAM (Data RAM)	13
3.7	IRAM (Instruction RAM)	14
3.8	General Purpose Register	15
3.9	ALU (Arithmetic and Logic Unit)	16
3.10	Mat_X	17
3.11	Reg_X	18
3.12	MDDR	18
3.13	PC	20
3.14	Register Selector	20
3.15	Core	22
3.16	Processor	23
4	Algorithm and Design Consideration	25
4.1	Algorithm implementation	25
4.1.1	The Flow Chart of the Algorithm	26

4.1.2	The Assembly Code	27
4.2	Compilers	28
4.2.1	Instruction Compiler	28
4.2.2	Data Compiler	28
5	Performance evaluation	29
5.1	Performance variation with Number of Cores	29
5.2	Performance variation with Input Matrix Size	30
6	Reference	31
7	Appendix	32
7.1	AC.v	32
7.2	CONTROL_UNIT.v	33
7.3	DECODER.v	45
7.4	DRAM.v	53
7.5	IRAM.v	56
7.6	GENERAL_PURPOSE_REGISTER.v	59
7.7	MAIN_ALU.v	60
7.8	MAT_X.v	61
7.9	REG_X.v	62
7.10	MDDR.v	63
7.11	PC.v	64
7.12	REGISTER_SELECTOR.v	65
7.13	CORE.v	66
7.14	PROCESSOR.v	70
7.15	compiler.py	73
7.16	matrix.py	74

List of Figures

1	Central Processing Unit (CPU)	1
2	Data paths of a Single Core	3
3	Data paths of a Multi Core Processor	4
4	State Diagram	6
5	Clock Module	7
6	Clock Signal	7
7	AC	8
8	BUS_MULTIPLEXER	9
9	Control Unit	10
10	DECODER	12
11	DRAM	13
12	IRAM	14
13	General Purpose Register	15
14	ALU	16
15	Mat_X	17
16	Reg_X	18
17	MDDR	19
18	PC	20
19	Register Selector	21
20	Core	22
21	Processor	23
22	4x4 Matrix	25
23	Resultant 3x4 Matrix	25
24	Flow Chart	26
25	Assembly Code	27
26	Instruction Compiler	28
27	Data Compiler	28
28	Performance relative to Core count	29
29	Performance relative to Matrix size	30

List of Tables

1	Instruction Set Table	4
2	AC Input/Output Table	8
3	BUS Multiplexer Input/Output Table	9
4	Control Unit Input/Output Table	11

5	Decoder Input/Output Table	12
6	DRAM Input/Output Table	13
7	IRAM Input/Output Table	14
8	General Purpose Register Input/Output Table	15
9	ALU Input/Output Table	16
10	ALU Opcode Table	16
11	Mat_X Input/Output Table	17
12	Reg_X Input/Output Table	18
13	MDDR Input/Output Table	19
14	PC Input/Output Table	20
15	Register Selector Input/Output Table	21
16	Core Input/Output Table	22
17	Processor Input/Output Table	24

1 Introduction

1.1 Overview

The objective of this project is to build a processor that can be used to multiply two matrices efficiently. The proposed processor has 4 cores that operate independently. Using multicore processing Matrix multiplication is carried out parallelly in each core. Every core is responsible for a subset of numbers in the final resultant matrix.

1.2 Central Processing Unit (CPU)

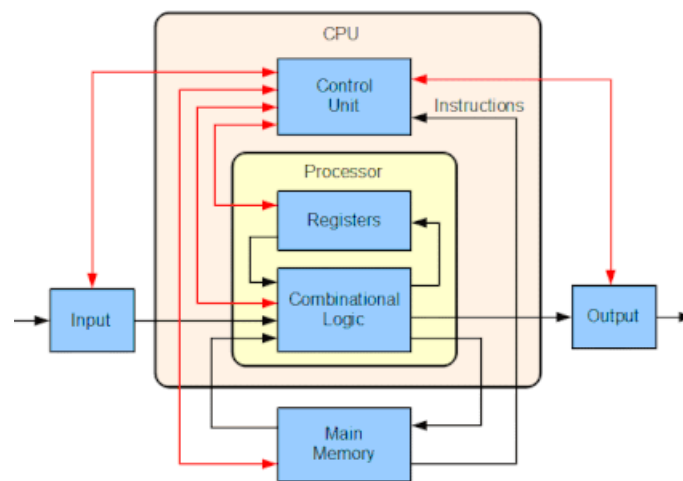


Figure 1: Central Processing Unit (CPU)

This is responsible for handling all arithmetic, logic control operations as well as it handles the input output processes according to the given instructions. This contains CU (Control Unit) and an ALU (Arithmetic and Logic Unit).

1.3 Microprocessor

The microprocessor is the unit made from incorporates the CPU, registers & memory. This unit is driven by the clock. Data is input to this unit by the memory and those data are processed according to the instructions stored in the instruction memory and then the results are stored into the memory.

1.4 Problem Statement

Multiplying two matrices is a very resource-consuming process. The standard way of multiplying an m -by- n matrix by an n -by- p matrix has complexity $O(mnp)$. If all of those are "n", it's $O(n^3)$. Therefore, this must be addressed in a different approach to optimizing the process. In this project, the problem statement is to speed up matrix multiplication using a multicore processor so that calculation can be parallelized in cores. Also, constraints in this design, matrix can only be contain positive integers.

1.5 Proposed Solution Based on FPGA

As mentioned in the previous section, the main requirement for the application of the processor is to multiply two matrixes in an efficient manner with a multicore processor. In our proposed solution two matrices can be given as the input of the processor then the processor calculates multiplication using 4 cores. Those 4 cores are identical and execute the same instructions set independently to finish the process as fast as it can.

1.6 Required Algorithms.

In this project, we had to make FPGA based processor which multiplies given two matrixes to have the resultant matrix using multi-core processor architecture. To accomplish this task, we had to use the following algorithm. The algorithm is designed in such a way that one core of the processor is responsible for calculating one value of the resultant matrix. According to the algorithm, when calculating the result of a particular location, the algorithm loops over the relevant row of the first matrix and relevant column of the second matrix and calculate the result. This algorithm allows dividing the work among multiple cores in an optimum manner.

2 Designing a FPGA based custom processor

2.1 FPGA - An Overview

FPGA stands for Field-programmable Gate Array. This is a very powerful device that has millions of logic ICs. These ICs are arranged in a grid format which can be configured by a designer or a customer after being manufactured. We used a FPGA board manufactured by Xilinx Inc and used Xilinx Vivado software for programming, testing, and simulations.

2.2 Data Paths

2.2.1 Single Core

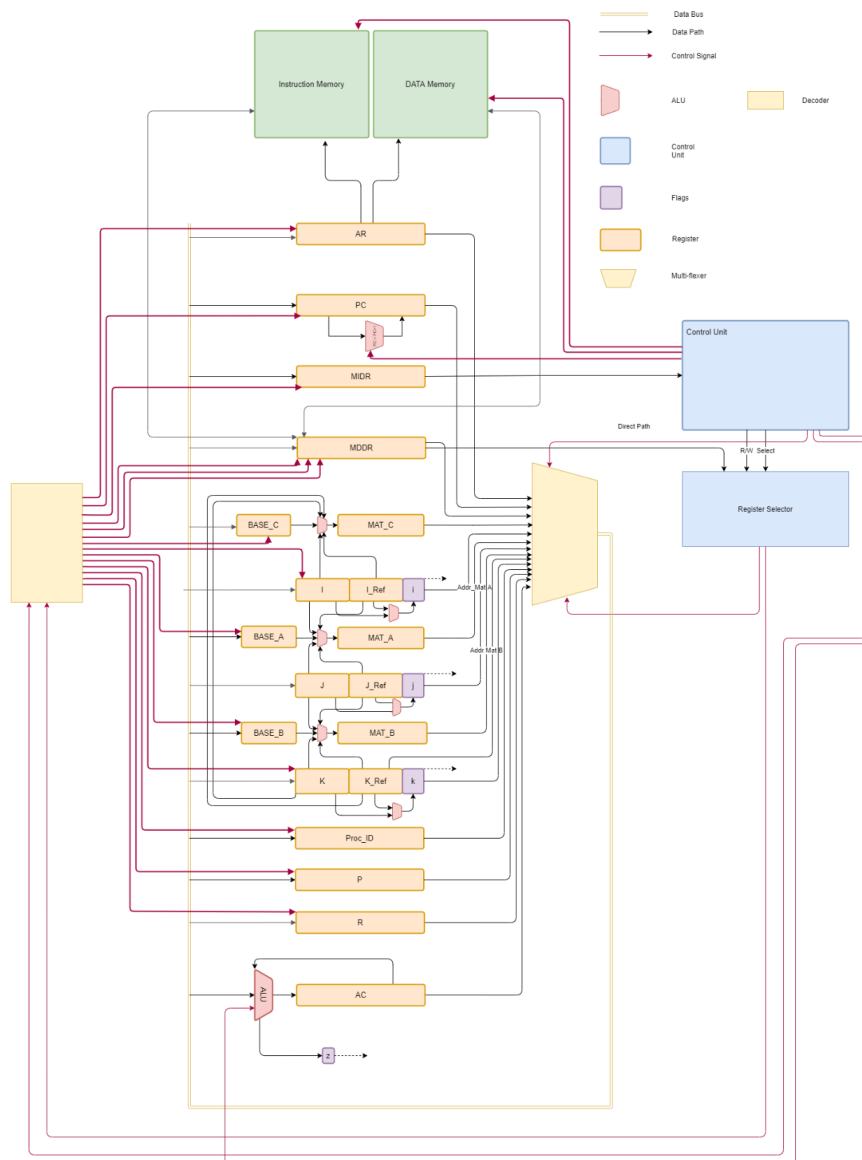


Figure 2: Data paths of a Single Core

2.2.2 Multi Core

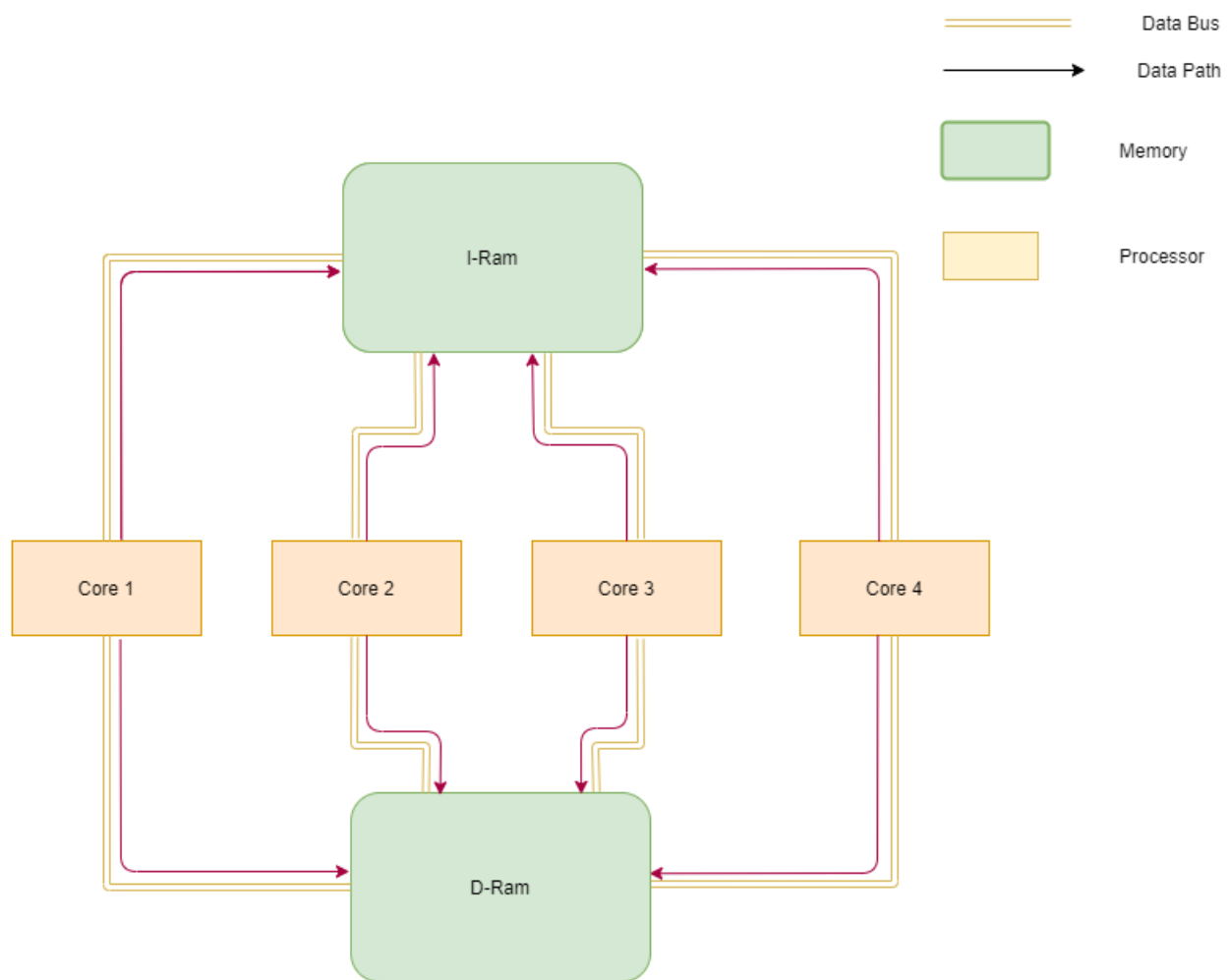


Figure 3: Data paths of a Multi Core Processor

2.3 The ISA - The Instruction Set Architecture

2.3.1 Instruction Set

Table 1: Instruction Set Table

Instruction	Opcode	Micro Instruction	Address	Operation
FETCH	0	FETCH1	0	$AR \leftarrow PC$; read; $PC \leftarrow PC + 1$
		FETCH2	1	$MDDR \leftarrow M.I$; $AR \leftarrow PC$; read
		FETCH3	2	$MIDR \leftarrow MDDR$;
NOP	3	NOP	3	
END	4	END	4	Finish = 1

Continued on next page

Table 1 – continued from previous page

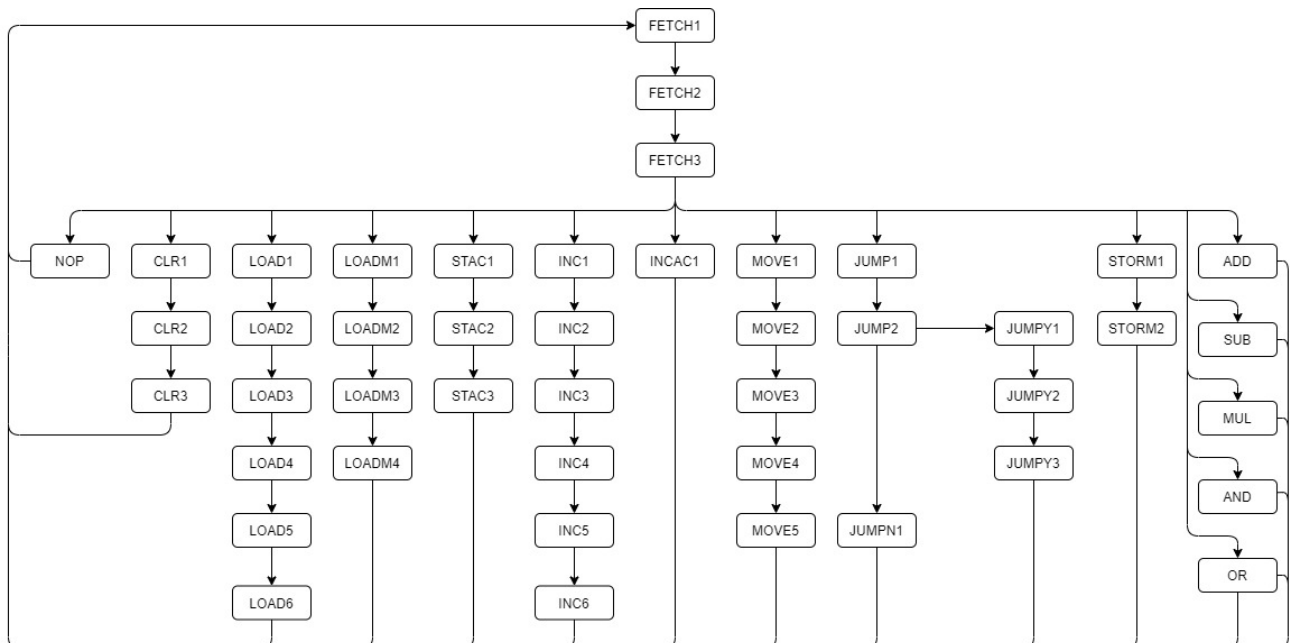
Instruction	Opcode	Micro Instruction	Address	Operation
CLR	5	CLR1	5	MDDR \Leftarrow M.I ; PC = PC + 1
		CLR2	6	AC \Leftarrow 0
		CLR3	7	R[MDDR] \Leftarrow AC
LOAD	8	LOAD1	8	MDDR \Leftarrow M.I ; PC = PC + 1
		LOAD2	9	AR \Leftarrow MDDR ; read;
		LOAD3	10	MDDR \Leftarrow M.Data ; AR \Leftarrow PC
		LOAD4	11	AC \Leftarrow MDDR ; read
		LOAD5	12	MDDR \Leftarrow M.I ; PC = PC + 1
		LOAD6	13	R[MDDR] \Leftarrow AC
LOADM	14	LOADM1	14	MDDR \Leftarrow M.I ; PC = PC + 1
		LOADM2	15	AR \Leftarrow MAT_[MDDR] ; read
		LOADM3	16	MDDR \Leftarrow M.Data
		LOADM4	17	AC \Leftarrow MDDR
STAC	18	STAC1	18	MDDR \Leftarrow M.I ; PC = PC + 1
		STAC2	19	AR \Leftarrow MDDR
		STAC3	20	MDDR \Leftarrow AC ; write
INC	21	INC1	21	MDDR \Leftarrow M.I ; PC = PC + 1
		INC2	22	AR \Leftarrow MDDR
		INC3	23	AR \Leftarrow PC ; read
		INC4	24	MDDR \Leftarrow M.I ; PC = PC + 1
		INC5	25	AC] \Leftarrow AC + R[MDDR]
		INC6	26	R[MDDR] \Leftarrow AC
INCAC	27	INCAC1	27	AC = AC + 1
JUMP	28	JUMP1	28	MDDR \Leftarrow M.I;
		JUMP2	29	PC = PC + 1
		JUMPY1	30	AR \Leftarrow PC ; read
		JUMPY2	31	MDDR \Leftarrow M;
		JUMPY3	32	PC \Leftarrow MDDR
		JUMPN1	33	PC = PC + 1
MOVE	34	MOVE1	34	MDDR \Leftarrow M.I ; PC = PC + 1
		MOVE2	35	AC \Leftarrow R[MDDR];
Continued on next page				

Table 1 – continued from previous page

Instruction	Opcode	Micro Instruction	Address	Operation
		MOVE3	36	$AR \leftarrow PC$; read
		MOVE4	37	$MDDR \leftarrow M.I$; $PC \leftarrow PC + 1$
		MOVE5	38	$R[MDDR] \leftarrow AC$
ADD	39	ADD	29	$AC \leftarrow AC + R$; if ($AC=0$) then $z=1$ else $z=0$
SUB	40	SUB	40	$AC \leftarrow AC - R$; if ($AC=0$) then $z=1$ else $z=0$
MUL	41	MUL	41	$AC \leftarrow AC * R$; if ($AC=0$) then $z=1$ else $z=0$
AND	42	AND	42	$AC \leftarrow AC \text{ and } R$; if ($AC=0$) then $z=1$ else $z=0$
OR	43	OR	43	$AC \leftarrow AC \text{ or } R$; if ($AC=0$) then $z=1$ else $z=0$
STORM	44	STORM1	44	$AR \leftarrow MAT_C$
		STORM2	45	$MDDR \leftarrow AC$; write

2.3.2 State Diagram

Single core executes one task at a time. The below figure shows the state diagram of our processor.

**Figure 4: State Diagram**

3 Module and Components

3.1 Clock Module

The processor consists of various modules and all of them need to work synchronously. This task is handled by the clock which runs continuously and consistently the whole time. Each module in the processor is connected to the clock. In our processor, we defined specifying operations to happen on the positive edge of the clock and the negative edge of the clock. figure 6 shows those operations.

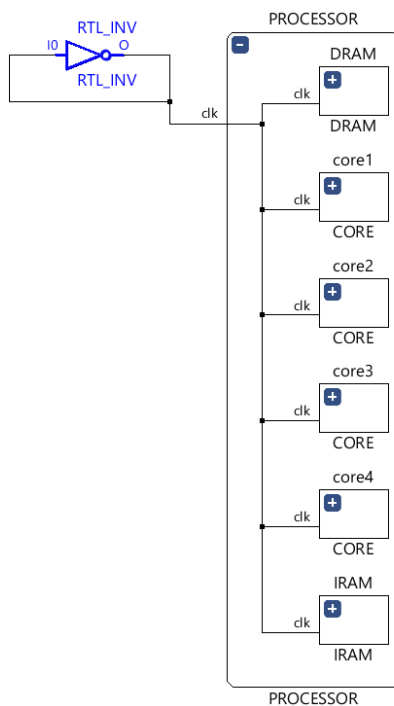


Figure 5: Clock Module

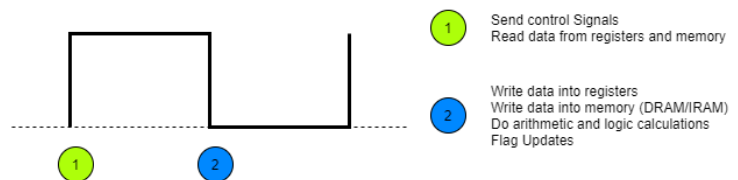


Figure 6: Clock Signal

3.2 AC

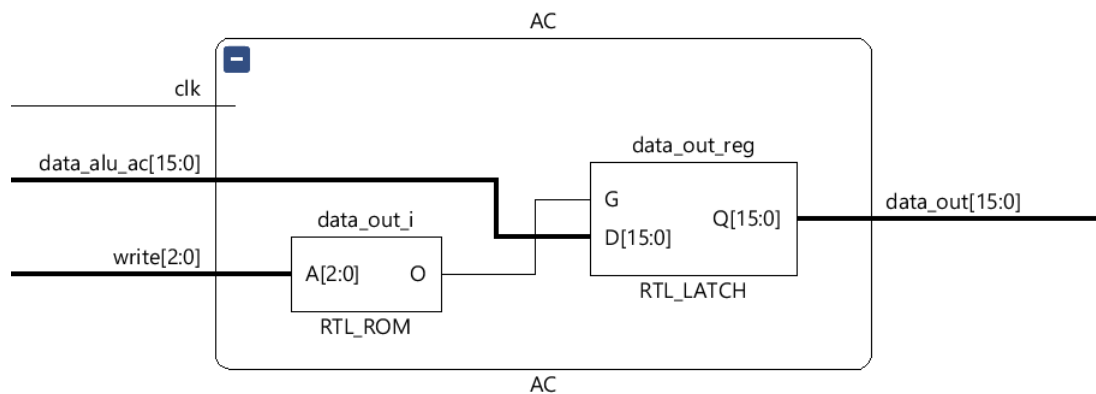


Figure 7: AC

AC is one of the special purpose registers in this processor. Which is directly connected to the ALU output. In our design, AC is not directly connected to the bus, to take inputs from the bus. However it is connected to the bus via the ALU. AC has 16-bit output and input. AC gets 3 control signals from CU, when the control signal 100 is given to the AC, it write data out of the ALU. In addition to that AC can be incremented and cleared (set to zero).

Table 2: AC Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	writealu_rstac_incac	3	Control Signal 100 = to write ALU data out to AC 010 = make AC = 0 001 = AC = AC + 1
	data_alu_ac	16	Data input from ALU out
Outputs	data_out	16	Data output

3.3 BUS Multiplexer

This module is created to get relevant register outputs to the BUS according to the CU or Register Selector control signal. It takes two inputs of control signals and outputs of each register as inputs to the multiplexer itself. As this module gets two control signals, register selector control signals have given priority. CU control signals are approved only if the register selector control signal is 0.

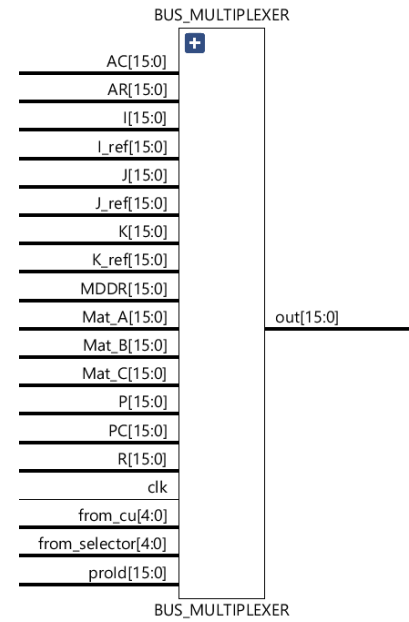


Figure 8: BUS_MULTIPLEXER

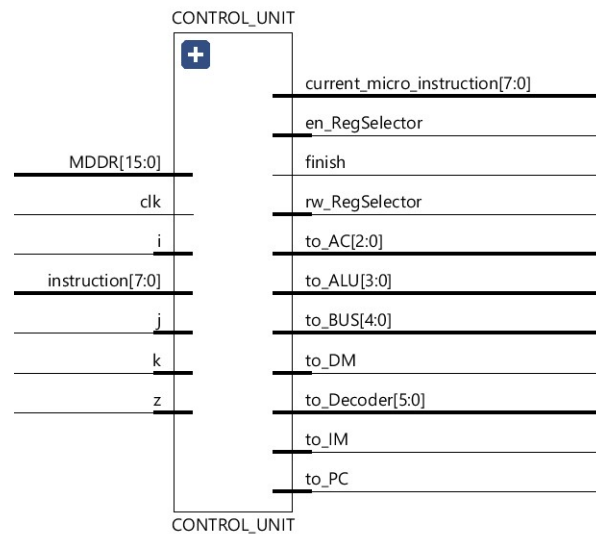
Table 3: BUS Multiplexer Input/Output Table

Inputs	Name	Size (bits)	Purpose
	clk	1	Clock Input
	from_selector	5	Control signal from Reg.Selector module
	from_cu	5	Control signal from control unit
	AC	16	Data out from AC
	AR	16	Data out from AR
	I	16	Data out from I
	I_Ref	16	Data out from I_Ref
	J	16	Data out from J
	J_Ref	16	Data out from J_Ref
	K	16	Data out from K
	K_Ref	16	Data out from K_Ref
	MDDR	16	Data out from MDDR
	Mat_A	16	Data out from Mat_A
	Mat_B	16	Data out from Mat_B
	Mat_C	16	Data out from Mat_C
Continued on next page			

Table 3 – continued from previous page

	P	16	Data out from P
	PC	16	Data out from PC
	R	16	Data out from R
	proId	16	Data out from proId
Outputs	data_out	16	Data output

3.4 Control Unit (CU)

**Figure 9: Control Unit**

The control unit controls the single core operations and defines all the instructions and their microinstructions. This works always when an instruction or z flag or CS (current instruction) is set. This module takes clock, z flag, i, j, k flags, instruction opcode, and MDDR value which is the currently loaded memory value as inputs. The control unit responds to the instructions and according to the microinstructions of that instruction, the control unit gives relevant signals to the component. As an example, when we take FETCH1 microinstruction it gives BUS MULTIPLEXER signal of 00010. This signal gets the value of the PC (program counter) and gives it to the bus. Then CU gives the decoder the signal 00001 which will add data from bus to AC. Finally, CU will give the signal 1 to incpc input in the PC which will increment the PC by 1. So, this completes the FETCH1 microinstruction, and it will go to the next microinstruction from there. After a single core finishes its tasks its finish flag will be set to high (1) and that particular core will stop executing instructions.

Table 4: Control Unit Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	MDDR	16	Data from MDDR
	instruction	8	Instruction from MIDR
	i	1	I flag input
	j	1	J flag input
	k	1	K flag input
	z	1	Z flag input
Outputs	to_AC	3	Control signal to AC
	to_ALU	4	Control signal to ALU
	to_BUS	5	Control signal to BUS
	to_DM	1	Control signal to DRAM
	to_IM	1	Control signal to IRAM
	to_PC	1	Control signal to PC
	to_Decoder	6	Control signal to decoder
	en_RegSelect	1	Enable signal to regSelector
	rw_RegSelect	1	Control signal to regSelector
	current_micro_instruction	8	Current Micro Instruction to Display
	finish	1	Finish status output

3.5 Decoder

This decoder module decodes control signals from the control unit for registers. Since there should be 20 different decoder signal types 6-bit reg select register was created to store control signal from the register selector as well as the control unit. Here we prioritize control signals from register selector from control unit because this register selector module was created to simplify selecting register read-write and enable signals. Most of the time control unit also sends read-write and register enabling signals to this register selector. That is why we prioritize this over control unit signals.

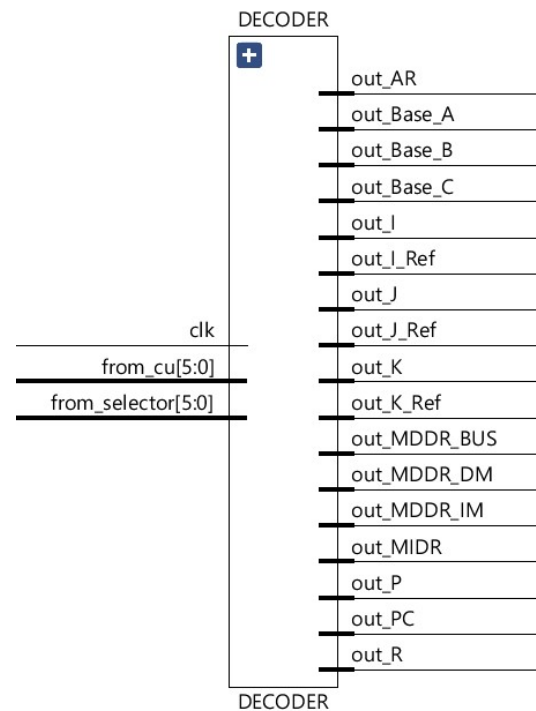


Figure 10: DECODER

Table 5: Decoder Input/Output Table

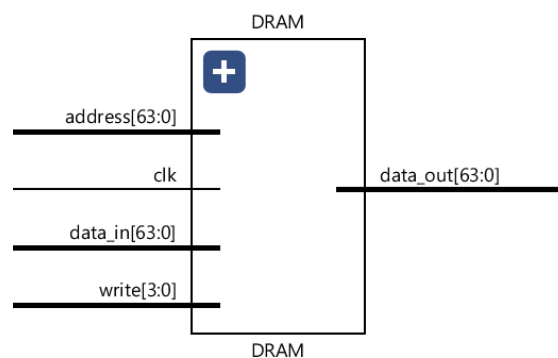
	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	from_selector	6	Control signal from Reg_Selector module
	from_cu	6	Control signal from control unit
Outputs	out_AR	1	Control signal to AR
	out_Base_A	1	Control signal to Base_A
	out_Base_B	1	Control signal to Base_B
	out_Base_C	1	Control signal to Base_C
	out_I	1	Control signal to I
	out_I_Ref	1	Control signal to I_Ref
	out_J	1	Control signal to J
	out_J_Ref	1	Control signal to J_Ref
	out_K	1	Control signal to K
	out_K_Ref	1	Control signal to K_Ref
	out_MDDR_BUS	1	Control signal to MDDR to get data from BUS

Continued on next page

Table 5 – continued from previous page

	out_MDDR_DM	1	Control signal to MDDR to get data from DRAM
	out_MDDR_IM	1	Control signal to MDDR to get data from IRAM
	out_MIDR	1	Control signal to MIDR
	out_P	1	Control signal to P
	out_PC	1	Control signal to PC
	out_R	1	Control signal to R

3.6 DRAM (Data RAM)

**Figure 11: DRAM**

DRAM is used to store data needed for processing and store the processed data. When an address is set and sends a read signal, the DRAM will output the relevant data. When the address and $data_{in}$ are set and send a write signal, the DRAM will store the data in the relevant address. The address, $data_{in}$ – $address[15 : 0]$ belong to core0. Hence any core can access DRAM at any time.

Table 6: DRAM Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	write	1	Control signal. Data will write or read to/from the DRAM from/to relevant core according to the write signal
	address	16	Address
	Data_in		Data Input
Outputs	data_out	16	Data output

3.7 IRAM (Instruction RAM)

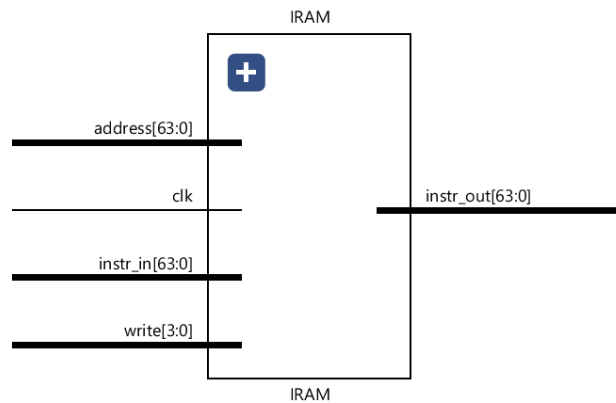


Figure 12: IRAM

IRAM is used to store data needed for processing and store the processed data. When an address is set and sends a read signal, the IRAM will output the relevant data. When the address and data are reset and send a write signal, the IRAM will store the data in the relevant address. The address, data in, and data out belong to core0. Hence any core can access IRAM at any time.

Table 7: IRAM Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	write	4	Control signal. Data will write or read to/from the IRAM from/to relevant core according to the write signal
	address	63	Address
	instr_in	63	Instruction Input
Outputs	instr_out	63	Instruction output

3.8 General Purpose Register

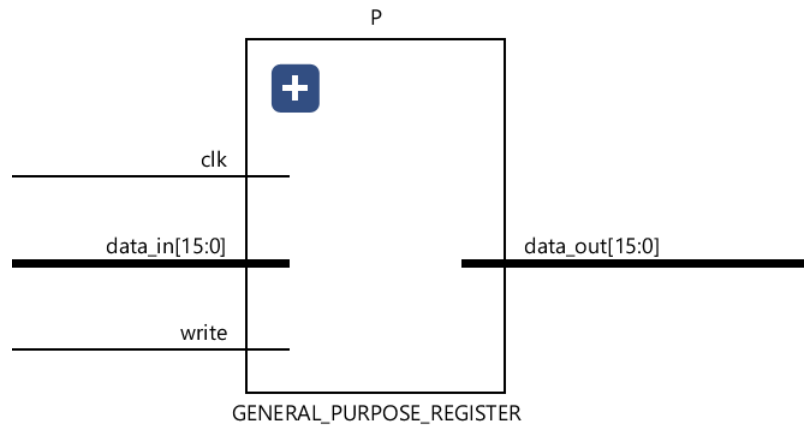


Figure 13: General Purpose Register

General Purpose Registers are used to store data temporarily. When ALU is processing data register acts as an intermediate memory location. This register cannot be cleared or incremented by itself for that we need to use INC instruction.

Table 8: General Purpose Register Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	write	4	Control signal. When write=0 register will write input data. Otherwise register is in Read mode
	data_in	63	Data Input
Outputs	data_out	63	Data output

3.9 ALU (Arithmetic and Logic Unit)

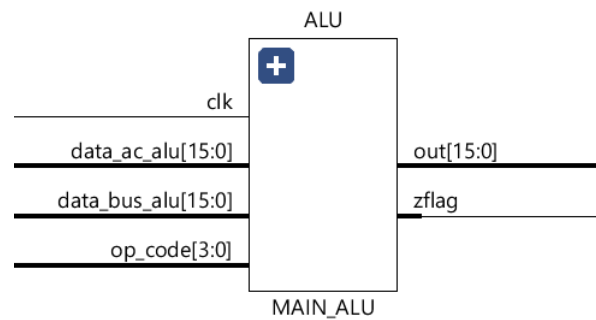


Figure 14: ALU

All Arithmetic and logical operations are handled by ALU. The result of ALU operation is stored in AC and zflag is updated accordingly.

Table 9: ALU Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	data_ac_alu	16	output of the AC to the ALU.
	data_bus_alu	16	Data input from bus
	op_code	4	Control signal which decides logic operation of ALU
Outputs	out	16	Data output to the AC
	zflag	1	Z flag z flag will be set to 1 if ALU result is zero or overflowed.

Here is the mapping of **op_codes** to ALU operations. All two input logical operations are carried out between AC input and Bus input.

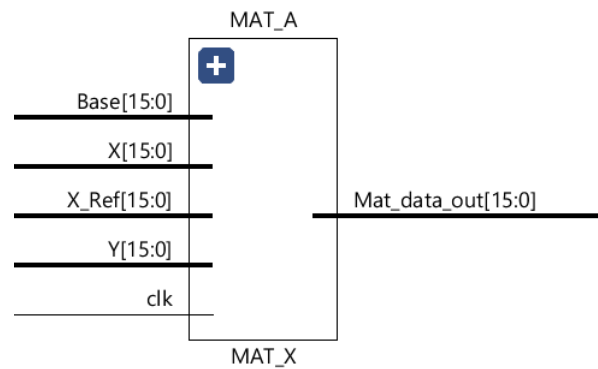
Table 10: ALU Opcode Table

Opcode bit pattern	Operation
0000	Ideal (previous ALU output = ALU output)
0001	Bypass (ALU output = bus input)
0010	Addition
0011	Substituting
Continued on next page	

Table 10 – continued from previous page

0100	Multiplication
0101	AND
0110	OR
0111	ADD 1 (ALU output = AC input + 1)
1000	Reset (ALU output is 0)

3.10 Mat_X

**Figure 15: Mat_X**

This module is a general-purpose module we used for a Matrix. In this module X means columns counter of the matrix and Y means rows counter of the defined matrix. This module is created to calculate matrix memory location when the starting address of the matrix, number of columns and row, column counter is given. $\text{Mat_data_out} = Y * X_Ref + X + \text{Base}$ is the equation we used here to calculate.

Table 11: Mat_X Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	Base	16	Input from the relevant base register.
	Y	16	Input from the relevant Reg-X register to X.
	X	16	Input from the relevant Reg-X register to Y.
	X_ref	16	Input from the relevant register to X_ref.
Outputs	Mat_data_out	16	Calculated address

3.11 Reg_X

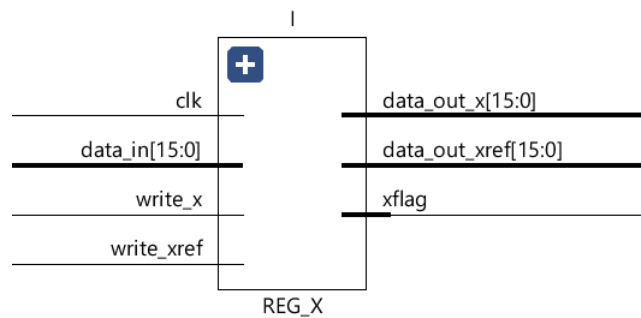


Figure 16: Reg_X

This module is created as a skeleton for I, J, and K registers. The main function of this register is to serve as a counter for looping through two input matrices. for loop through 2 input matrices required three for loops. I, J, and K registers act as loop indexes. When indexes are reached to limit (xref) respective flags (xflag) will set to 1.

Table 12: Reg_X Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	write_x	1	Control signal to write data to X value.
	write_xref	1	Control signal to write data to X_ref value.
	data_in	16	Data in from the bus.
Outputs	data_out_x	16	Data output of X
	data_out_xref	16	Data output of X_ref
	xflag	1	Flag output of X

3.12 MDDR

Data coming from both IRAM and DRAM are the first to come to this register. Data to be written to IRAM or DRAM is also stored in this. Therefore, this register has three data inputs which are from the bus, IRAM, and DRAM. This register gets 3 control signals from the decoder to decide which data input to write into it.

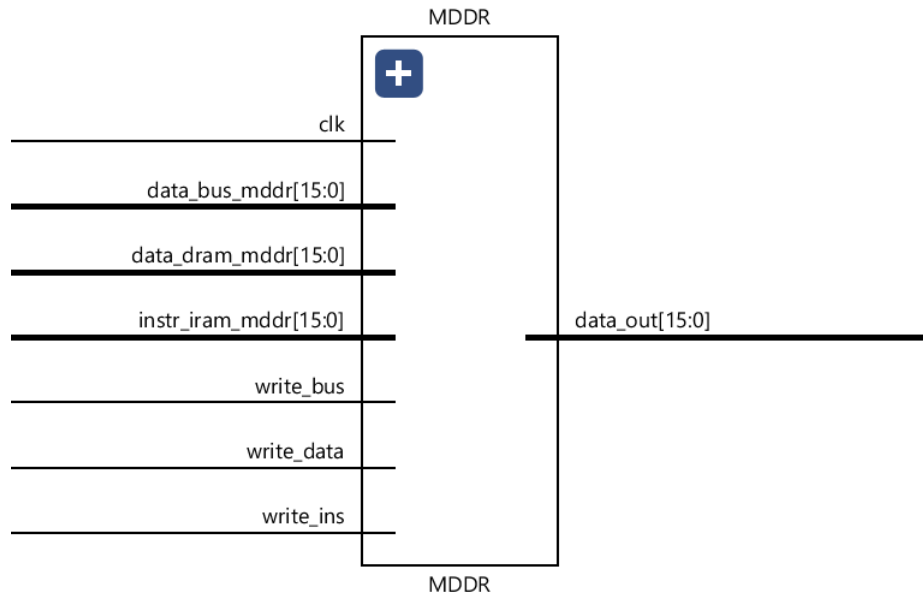


Figure 17: MDDR

Table 13: MDDR Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	write_bus	1	Control signal to write data on the bus to the MDDR.
	write_data	1	Control signal to write data from DRAM to the MDDR.
	write_ins	1	Control signal to write data from IRAM to the MDDR.
	data_bus_mddr	16	Data input from bus.
	data_dram_mddr	16	Data input from the DRAM.
	data_iram_mddr	16	Data input from the IRAM.
Outputs	data_out	16	Data output

3.13 PC

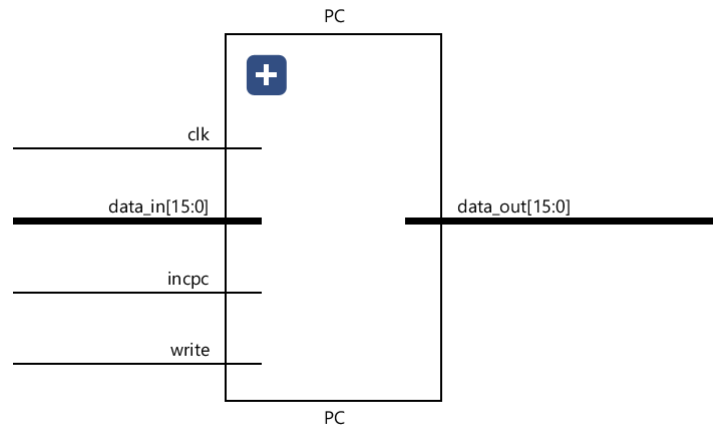


Figure 18: PC

PC is used to keep track on the current instruction running. The control unit will load instructions according to the PC.

Table 14: PC Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	write	1	Control signal. When write = 1 PC will write input data. Otherwise, register is in Read mode.
	data_in	16	Data input
Outputs	data_out	16	Data output

3.14 Register Selector

This is a specially designed unit that is used to select register with a given register address. When an instruction of program needs to access register the address of that register is provided next to that instruction. Then control unit sends enable signal to this Register Selector unit with a read-write signal and fetches the registered address to MDDR from IRAM. Then this unit act as a switch.

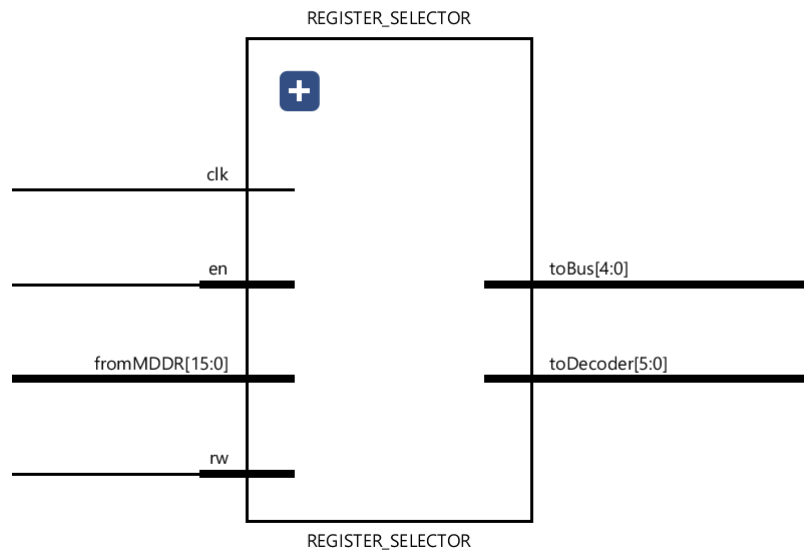


Figure 19: Register Selector

Table 15: Register Selector Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	en	1	Enable signal from control unit.
	fromMDDR	16	Data from MDDR MDDR is directly wired to this unit Address of required register fetch from this input.
	rw	1	Read-write selection signal from control unit.
Outputs	toBus	5	Output for Bus Multiplexer. If $rw = 0$, <ul style="list-style-type: none"> Address of register is set to toBus. toDecoder is set to 0.
	toDecoder	6	Output for Decoder.. If $rw = 0$, <ul style="list-style-type: none"> Address of register is set to toDecoder. toBus is set to 0.

3.15 Core

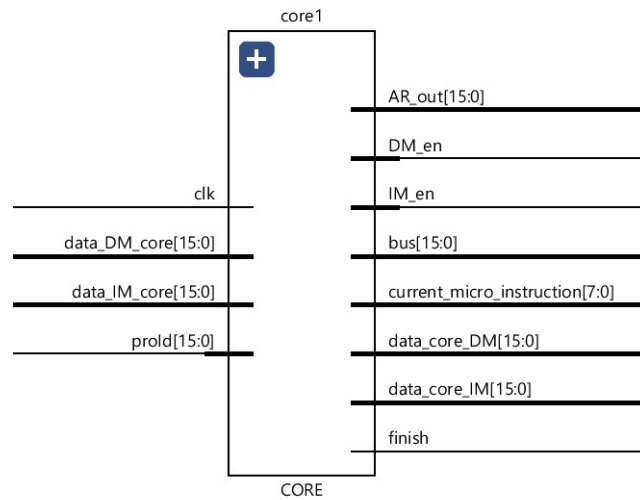


Figure 20: Core

The processor consists of four single cores. Basically, these cores can operate independently. The core can access IRAM and DRAM independently. Each core has a unique ID which is obtained from the processor. This ID is used to fetch relevant row and column values to generate the final answer (fetching algorithm).

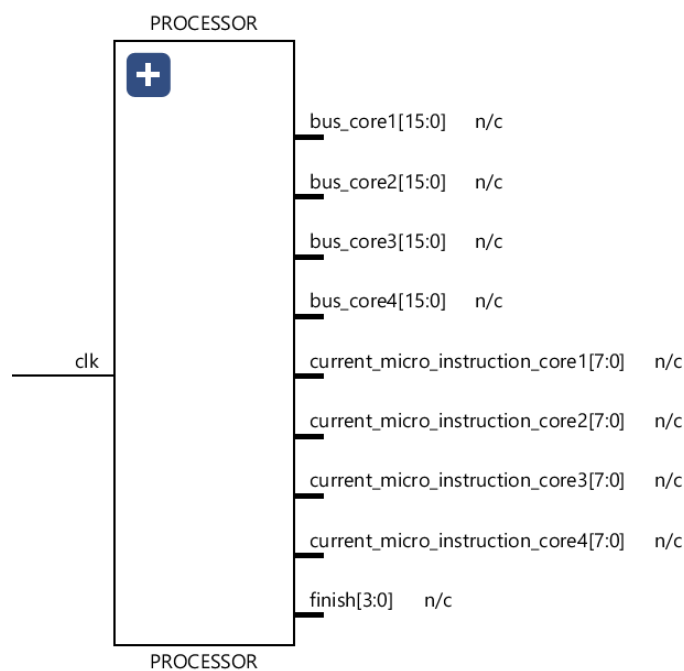
Table 16: Core Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
	data_DM_core	16	Data from DRAM
	data_IM_core	16	Instruction from IRAM
	proId	16	Unique core ID from main processor
Outputs	DM_en	1	Enable signal to DRAM. Also, this signal can be used to detect which core is requesting.
	IM_en	1	Enable signal for IRAM read Also, this signal can be used to detect which core is requesting.
	data_core_DM	16	Data to be written in DRAM
	data_core_IM	16	Data to be written in IRAM
	finish	1	When core finish its process,it set
Continued on next page			

Table 16 – continued from previous page

			finish to 1, after all four cores set finish to 1, whole process will be terminated.
	bus	16	This output is just for plotting simulation waveform
	current_micro_instruction	8	This output is just for plotting simulation waveform
	AR_out	16	This output is just for plotting simulation waveform

3.16 Processor

**Figure 21: Processor**

The processor connects all core, IRAM, and DRAM. The cores run the instructions and save the result in dram. (These outputs are used for simulation purposes)

Table 17: Processor Input/Output Table

	Name	Size (bits)	Purpose
Inputs	clk	1	Clock Input
Outputs	finish	4	Check whether the cores are finished
	bus_core1	16	Value in the bus of core1
	bus_core2	16	Value in the bus of core2
	bus_core3	16	Value in the bus of core3
	bus_core4	16	Value in the bus of core4
	current_micro_instruction_core1	16	Current micro instruction in core1
	current_micro_instruction_core2	16	Current micro instruction in core2
	current_micro_instruction_core3	16	Current micro instruction in core3
	current_micro_instruction_core4	16	Current micro instruction in core4

4 Algorithm and Design Consideration

4.1 Algorithm implementation

Our algorithm uses three loops to accomplish the task. The core loops over the matrices to calculate the multiplied matrix. Three registers have been used to keep track of the loops. To reduce the number of steps for calculations, we have implemented a custom module to calculate the addresses of a given matrix. When the base address of the matrix, x,y coordinates were given to this address calculator module, It will calculate the address without spending more than one clock cycle. By using the advantage of this custom module, we have optimized the calculations.

The algorithm used to multiply the matrices using multiple cores is almost the same as the algorithm used in a single-core processor. We have introduced a core id for each core. By using the core id the core determines the starting point of the calculation.

For an example, if the core id is 1 the core will start its calculation from 2nd cell of the resultant matrix.

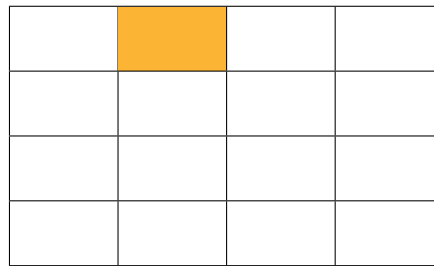
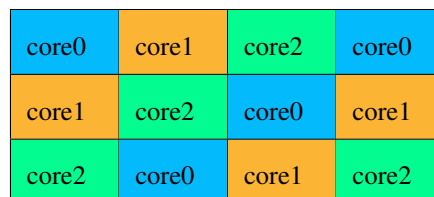


Figure 22: 4x4 Matrix

How to split the work between cores?

The job of a core is to calculate the final result of a given cell of the resultant matrix (according to the algorithm). For example, when we multiply 3x2 and 2x4 matrices, the resultant matrix should be 3x4. If there are 3 cores, the responsible core for each final result is shown below.



core0	core1	core2	core0
core1	core2	core0	core1
core2	core0	core1	core2

Figure 23: Resultant 3x4 Matrix

4.1.1 The Flow Chart of the Algorithm

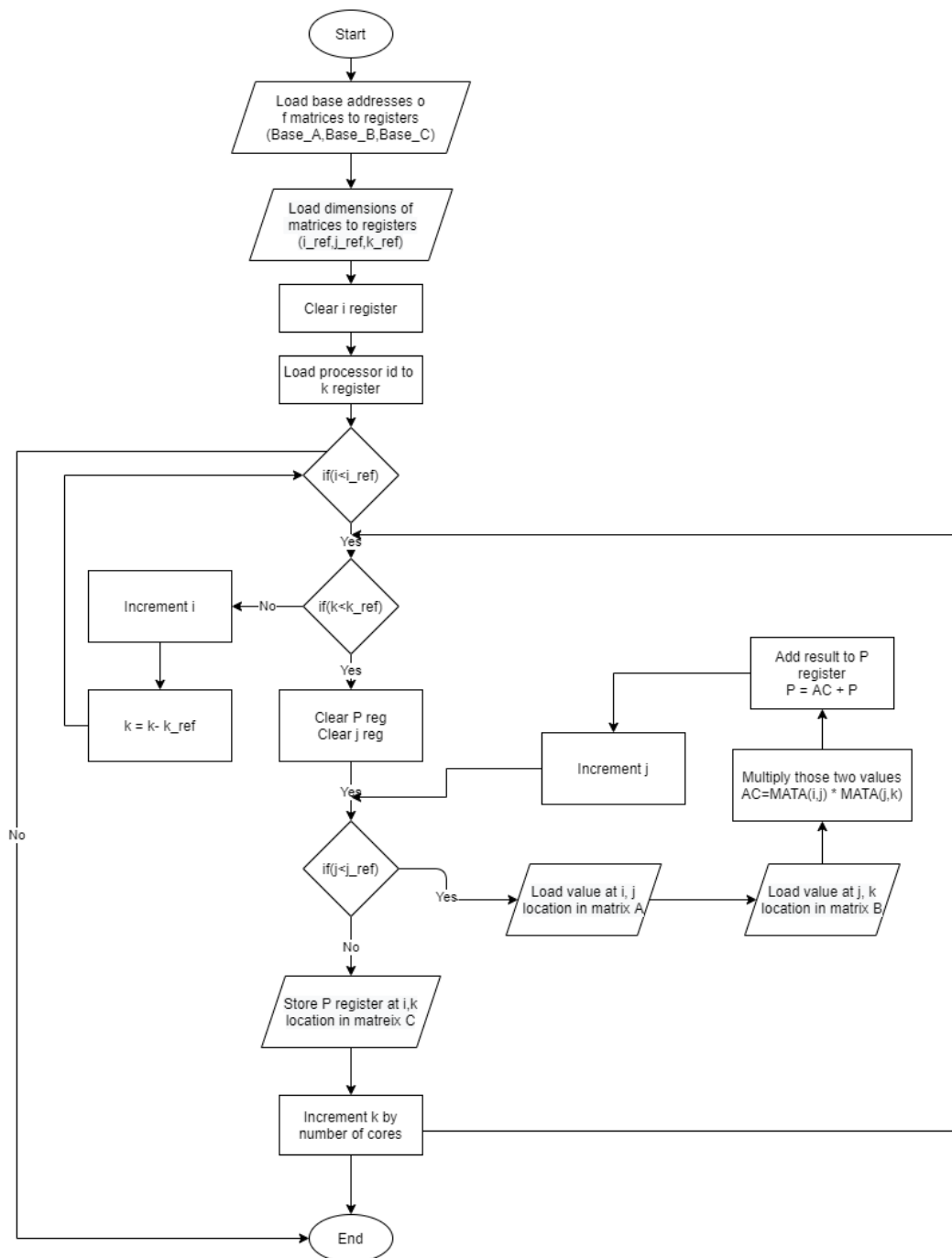


Figure 24: Flow Chart

4.1.2 The Assembly Code

```

load 0 BASE_A
load 1 BASE_B
load 2 BASE_C
load 3 i_ref
load 4 j_ref
load 15 k_ref
clr i
move PRO_ID k

:TAG6
jump i_flag TAG1
:TAG5
jump k_flag TAG2
    clr P
    clr j
    :TAG4
    jump j_flag TAG3
        loadm MAT_A
        move AC R
        loadm MAT_B
        mul
        move AC R
        move P AC
        add
        move AC P
        inc 1 j
    jump nj_flag TAG4
    :TAG3
    move P AC
    storm
    inc 3 k
jump nk_flag TAG5
:TAG2
inc 1 i
move k_ref R
move k AC
sub
move AC k
jump ni_flag TAG6
:TAG1
end

```

Figure 25: Assembly Code

4.2 Compilers

4.2.1 Instruction Compiler

We build a python compiler that takes Instructions as an input and returns binary values to be stored in IRAM. Using this compiler, we can test different programs efficiently. Python code is attached to the appendix.

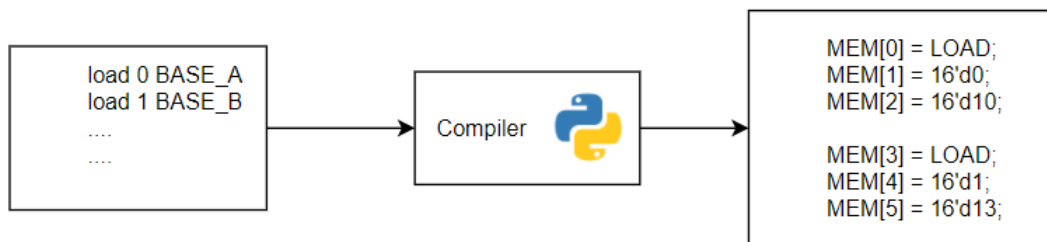


Figure 26: Instruction Compiler

4.2.2 Data Compiler

This compiler takes two matrixes as the input and returns formatted data to be stored. Using this compiler, we can test our processor with different matrices easily. Also, this compiler returns the correct answer of the multiplication so that the answers of the processor can be verified.

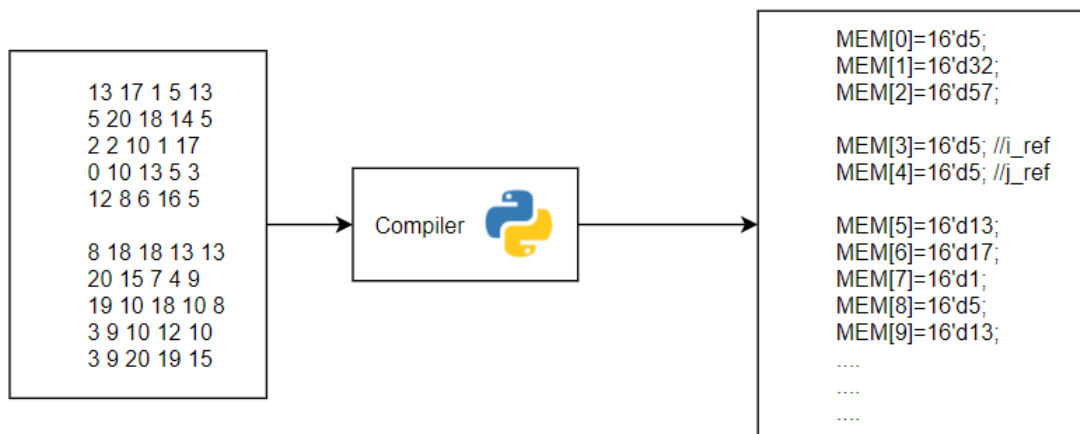


Figure 27: Data Compiler

5 Performance evaluation

5.1 Performance variation with Number of Cores

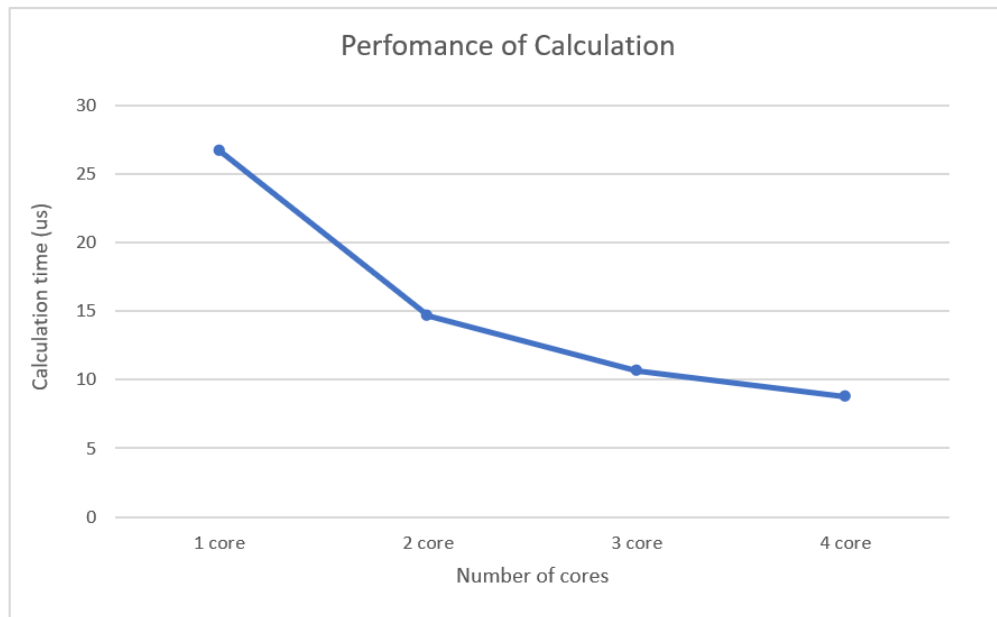


Figure 28: Performance relative to Core count

In the performance evaluation, input matrix size is fixed to two 4×2 and 2×3 matrices. The method we followed to evaluate the processor's performance against several cores is to increase the number of cores and then measure the finish time of multiplication.

The reduction of time of calculation, when increasing the number of cores is indirectly proportional as we expected.

5.2 Performance variation with Input Matrix Size

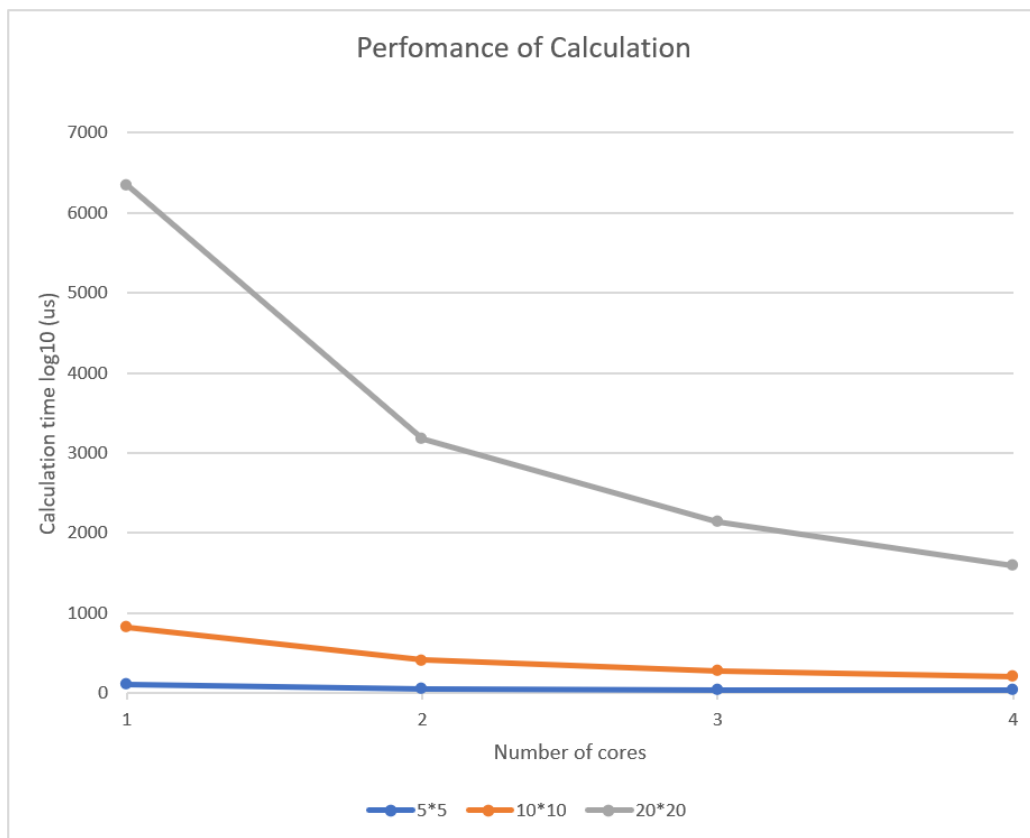


Figure 29: Performance relative to Matrix size

For further evaluation on performance improvement, we then vary the size of the input matrices while a number of cores are changing. We use 5×5 , 10×10 , and 20×20 matrices as the inputs. And change the number of cores from 1 to 4. From the result, we can see that impact of increasing the number of cores on performance increasing, increases with the size of the input matrices.

6 Reference

- [1] Xilinx: Documentation,
<https://www.xilinx.com/support.html#documentation>
- [2] Nvidia: Cuda Zone,
<https://developer.nvidia.com/cuda-zone>
- [3] Wikipedia: Computer architecture,
https://en.wikipedia.org/wiki/Computer_architecture

7 Appendix

7.1 AC.v

```
1 `timescale 1ns / 1ps
2 module AC (
3     input clk,
4     input [2:0]write,
5     input [15:0] data_alu_ac,
6     output reg [15:0] data_out );
7     always @(*)
8         begin
9             if (write==3'b100) data_out = data_alu_ac;
10        end
11 endmodule
```

7.2 CONTROL_UNIT.v

```

1 module CONTROL_UNIT(
2     input clk,
3     input [0:0] z,
4     input [0:0] i,
5     input [0:0] j,
6     input [0:0] k,
7     input [7:0] instruction,
8     input [15:0]MDDR,
9
10    output reg[3:0] to_ALU,
11    output reg[0:0] to_DM,
12    output reg[0:0] to_IM,
13    output reg[5:0] to_Decoder,
14    output reg[4:0] to_BUS,
15    output reg[0:0] to_PC,
16    output reg[2:0] to_AC,
17    output reg[0:0] en_RegSelector,
18    output reg[0:0] rw_RegSelector,
19    output reg finish,
20    output reg [7:0]current_micro_instruction);
21
22    reg [5:0] CS = 8'd3;
23    reg [5:0] NS = 8'd0;
24
25    parameter FETCH1 = 8'd0;
26    parameter FETCH2 = 8'd1;
27    parameter FETCH3 = 8'd2;
28    parameter NOP = 8'd3;
29    parameter END = 8'd4;
30    parameter CLR1 = 8'd5;
31    parameter CLR2 = 8'd6;
32    parameter CLR3 = 8'd7;
33    parameter LOAD1 = 8'd8;
34    parameter LOAD2 = 8'd9;
35    parameter LOAD3 = 8'd10;
36    parameter LOAD4 = 8'd11;
37    parameter LOAD5 = 8'd12;
38    parameter LOAD6 = 8'd13;
39    parameter LOADM1 = 8'd14;
40    parameter LOADM2 = 8'd15;
41    parameter LOADM3 = 8'd16;
42    parameter LOADM4 = 8'd17;
43    parameter STAC1 = 8'd18;
44    parameter STAC2 = 8'd19;
45    parameter STAC3 = 8'd20;
46    parameter INC1 = 8'd21;
47    parameter INC2 = 8'd22;
48    parameter INC3 = 8'd23;
49    parameter INC4 = 8'd24;
50    parameter INC5 = 8'd25;
51    parameter INC6 = 8'd26;
52    parameter INCAC1 = 8'd27;
53    parameter JUMP1 = 8'd28;
54    parameter JUMP2 = 8'd29;

```

```

55     parameter JUMPY1 = 8'd30;
56     parameter JUMPY2 = 8'd31;
57     parameter JUMPY3 = 8'd32;
58     parameter JUMPY4 = 8'd33;
59     parameter MOVE1 = 8'd34;
60     parameter MOVE2 = 8'd35;
61     parameter MOVE3 = 8'd36;
62     parameter MOVE4 = 8'd37;
63     parameter MOVE5 = 8'd38;
64     parameter ADD = 8'd39;
65     parameter SUB = 8'd40;
66     parameter MUL = 8'd41;
67     parameter AND = 8'd42;
68     parameter OR = 8'd43;
69     parameter STORM1 = 8'd44;
70     parameter STORM2 = 8'd45;
71
72     always @(posedge clk)
73     begin
74         CS <= NS;
75         current_micro_instruction<=NS;
76     end
77     always @(posedge clk)
78     begin
79         if (CS == END)
80             finish <= 1'd1;
81         else
82             finish <= 1'd0;
83     end
84     always @(CS or z or instruction)
85     begin
86         case(CS)
87             NOP: begin
88                 NS = FETCH1;
89             end
90             FETCH1: begin
91                 to_ALU = 4'd0;
92                 to_BUS = 5'd2;
93                 to_Decoder = 5'd1;
94                 to_IM = 1'd0;
95                 to_DM = 1'd0;
96                 to_PC = 1'd1;
97                 to_AC = 3'd0;
98                 en_RegSelector = 1'd0;
99                 rw_RegSelector = 1'd0;
100                NS = FETCH2;
101            end
102            FETCH2: begin
103                to_ALU = 4'd0;
104                to_BUS = 5'd2;
105                to_Decoder = 5'd18;
106                to_IM = 1'd0;
107                to_DM = 1'd0;
108                to_PC = 1'd0;

```



```
109         to_AC = 3'd0;
110         en_RegSelector = 1'd0;
111         rw_RegSelector = 1'd0;
112         NS = FETCH3;
113     end
114     FETCH3: begin
115         to_ALU = 4'd0;
116         to_BUS = 5'd4;
117         to_Decoder = 5'd3;
118         to_IM = 1'd0;
119         to_DM = 1'd0;
120         to_PC = 1'd0;
121         to_AC = 3'd0;
122         en_RegSelector = 1'd0;
123         rw_RegSelector = 1'd0;
124         NS = instruction[7:0];
125     end
126     END: begin
127         NS = END;
128     end
129     CLR1: begin
130         to_ALU = 4'd0;
131         to_BUS = 5'd4;
132         to_Decoder = 5'd6;
133         to_IM = 1'd0;
134         to_DM = 1'd0;
135         to_PC = 1'd1;
136         to_AC = 3'd0;
137         en_RegSelector = 1'd0;
138         rw_RegSelector = 1'd0;
139         NS = CLR2;
140     end
141     CLR2: begin
142         to_ALU = 4'd8;
143         to_BUS = 5'd0;
144         to_Decoder = 5'd0;
145         to_IM = 1'd0;
146         to_DM = 1'd0;
147         to_PC = 1'd0;
148         to_AC = 3'b000;
149         en_RegSelector = 1'd0;
150         rw_RegSelector = 1'd0;
151         NS = CLR3;
152     end
153     CLR3: begin
154         to_ALU = 4'd0;
155         to_BUS = 5'd0;
156         to_Decoder = 5'd0;
157         to_IM = 1'd0;
158         to_DM = 1'd0;
159         to_PC = 1'd0;
160         to_AC = 3'b000;
161         en_RegSelector = 1'd1;
162         rw_RegSelector = 1'd1;
```

```
163         NS = FETCH1;
164     end
165     LOAD1: begin
166         to_ALU = 4'd0;
167         to_BUS = 5'd0;
168         to_Decoder = 5'd6;
169         to_IM = 1'd0;
170         to_DM = 1'd0;
171         to_PC = 1'd1;
172         to_AC = 3'b000;
173         en_RegSelector = 1'd0;
174         rw_RegSelector = 1'd0;
175         NS = LOAD2;
176     end
177     LOAD2: begin
178         to_ALU = 4'd0;
179         to_BUS = 5'd4;
180         to_Decoder = 5'd1;
181         to_IM = 1'd0;
182         to_DM = 1'd0;
183         to_PC = 1'd0;
184         to_AC = 3'b000;
185         en_RegSelector = 1'd0;
186         rw_RegSelector = 1'd0;
187         NS = LOAD3;
188     end
189     LOAD3: begin
190         to_ALU = 4'd0;
191         to_BUS = 5'd2;
192         to_Decoder = 5'd19;
193         to_IM = 1'd0;
194         to_DM = 1'd0;
195         to_PC = 1'd0;
196         to_AC = 3'b000;
197         en_RegSelector = 1'd0;
198         rw_RegSelector = 1'd0;
199         NS = LOAD4;
200     end
201     LOAD4: begin
202         to_ALU = 4'd1;
203         to_BUS = 5'd4;
204         to_Decoder = 5'd0;
205         to_IM = 1'd0;
206         to_DM = 1'd0;
207         to_PC = 1'd0;
208         to_AC = 3'b100;
209         en_RegSelector = 1'd0;
210         rw_RegSelector = 1'd0;
211         NS = LOAD5;
212     end
213     LOAD5: begin
214         to_ALU = 4'd0;
215         to_BUS = 5'd0;
216         to_Decoder = 5'd6;
```

```
217         to_IM = 1'd0;
218         to_DM = 1'd0;
219         to_PC = 1'd1;
220         to_AC = 3'b000;
221         en_RegSelector = 1'd0;
222         rw_RegSelector = 1'd0;
223         NS = LOAD6;
224     end
225     LOAD6: begin
226         to_ALU = 4'd0;
227         to_BUS = 5'd20;
228         to_Decoder = 5'd0;
229         to_IM = 1'd0;
230         to_DM = 1'd0;
231         to_PC = 1'd0;
232         to_AC = 3'b000;
233         en_RegSelector = 1'd1;
234         rw_RegSelector = 1'd1;
235         NS = FETCH1;
236     end
237     LOADM1: begin
238         to_ALU = 4'd0;
239         to_BUS = 5'd0;
240         to_Decoder = 5'd6;
241         to_IM = 1'd0;
242         to_DM = 1'd0;
243         to_PC = 1'd1;
244         to_AC = 3'b000;
245         en_RegSelector = 1'd0;
246         rw_RegSelector = 1'd0;
247         NS = LOADM2;
248     end
249     LOADM2: begin
250         to_ALU = 4'd0;
251         to_BUS = 5'd0;
252         to_Decoder = 5'd1;
253         to_IM = 1'd0;
254         to_DM = 1'd0;
255         to_PC = 1'd0;
256         to_AC = 3'b000;
257         en_RegSelector = 1'd1;
258         rw_RegSelector = 1'd0;
259         NS = LOADM3;
260     end
261     LOADM3: begin
262         to_ALU = 4'd0;
263         to_BUS = 5'd0;
264         to_Decoder = 5'd5;
265         to_IM = 1'd0;
266         to_DM = 1'd0;
267         to_PC = 1'd0;
268         to_AC = 3'b000;
269         en_RegSelector = 1'd0;
270         rw_RegSelector = 1'd0;
```

```
271         NS = LOADM4;
272     end
273     LOADM4: begin
274         to_ALU = 4'd1;
275         to_BUS = 5'd4;
276         to_Decoder = 5'd0;
277         to_IM = 1'd0;
278         to_DM = 1'd0;
279         to_PC = 1'd0;
280         to_AC = 3'b100;
281         en_RegSelector = 1'd0;
282         rw_RegSelector = 1'd0;
283         NS = FETCH1;
284     end
285     STAC1: begin
286         to_ALU = 4'd0;
287         to_BUS = 5'd0;
288         to_Decoder = 5'd6;
289         to_IM = 1'd0;
290         to_DM = 1'd0;
291         to_PC = 1'd1;
292         to_AC = 3'b000;
293         en_RegSelector = 1'd0;
294         rw_RegSelector = 1'd0;
295         NS = STAC2;
296     end
297     STAC2: begin
298         to_ALU = 4'd0;
299         to_BUS = 5'd4;
300         to_Decoder = 5'd1;
301         to_IM = 1'd0;
302         to_DM = 1'd0;
303         to_PC = 1'd0;
304         to_AC = 3'b000;
305         en_RegSelector = 1'd0;
306         rw_RegSelector = 1'd0;
307         NS = STAC3;
308     end
309     STAC3: begin
310         to_ALU = 4'd0;
311         to_BUS = 5'd20;
312         to_Decoder = 5'd4;
313         to_IM = 1'd0;
314         to_DM = 1'd1;
315         to_PC = 1'd0;
316         to_AC = 3'b000;
317         en_RegSelector = 1'd0;
318         rw_RegSelector = 1'd0;
319         NS = FETCH1;
320     end
321     INC1: begin
322         to_ALU = 4'd0;
323         to_BUS = 5'd0;
324         to_Decoder = 5'd6;
```

```
325         to_IM = 1'd0;
326         to_DM = 1'd0;
327         to_PC = 1'd1;
328         to_AC = 3'b000;
329         en_RegSelector = 1'd0;
330         rw_RegSelector = 1'd0;
331         NS = INC2;
332     end
333     INC2: begin
334         to_ALU = 4'd1;
335         to_BUS = 5'd4;
336         to_Decoder = 5'd0;
337         to_IM = 1'd0;
338         to_DM = 1'd0;
339         to_PC = 1'd0;
340         to_AC = 3'b100;
341         en_RegSelector = 1'd0;
342         rw_RegSelector = 1'd0;
343         NS = INC3;
344     end
345     INC3: begin
346         to_ALU = 4'd0;
347         to_BUS = 5'd2;
348         to_Decoder = 5'd1;
349         to_IM = 1'd0;
350         to_DM = 1'd0;
351         to_PC = 1'd0;
352         to_AC = 3'b000;
353         en_RegSelector = 1'd0;
354         rw_RegSelector = 1'd0;
355         NS = INC4;
356     end
357     INC4: begin
358         to_ALU = 4'd0;
359         to_BUS = 5'd0;
360         to_Decoder = 5'd6;
361         to_IM = 1'd0;
362         to_DM = 1'd0;
363         to_PC = 1'd1;
364         to_AC = 3'b000;
365         en_RegSelector = 1'd0;
366         rw_RegSelector = 1'd0;
367         NS = INC5;
368     end
369     INC5: begin
370         to_ALU = 4'd2;
371         to_BUS = 5'd0;
372         to_Decoder = 5'd0;
373         to_IM = 1'd0;
374         to_DM = 1'd0;
375         to_PC = 1'd0;
376         to_AC = 3'b100;
377         en_RegSelector = 1'd1;
378         rw_RegSelector = 1'd0;
```

```
379         NS = INC6;
380     end
381     INC6: begin
382         to_ALU = 4'd0;
383         to_BUS = 5'd20;
384         to_Decoder = 5'd0;
385         to_IM = 1'd0;
386         to_DM = 1'd0;
387         to_PC = 1'd0;
388         to_AC = 3'b000;
389         en_RegSelector = 1'd1;
390         rw_RegSelector = 1'd1;
391         NS = FETCH1;
392     end
393     INCAC1: begin
394         to_ALU = 4'd7;
395         to_BUS = 5'd0;
396         to_Decoder = 5'd0;
397         to_IM = 1'd0;
398         to_DM = 1'd0;
399         to_PC = 1'd0;
400         to_AC = 3'b100;
401         en_RegSelector = 1'd0;
402         rw_RegSelector = 1'd0;
403         NS = FETCH1;
404     end
405     MOVE1: begin
406         to_ALU = 4'd0;
407         to_BUS = 5'd0;
408         to_Decoder = 5'd6;
409         to_IM = 1'd0;
410         to_DM = 1'd0;
411         to_PC = 1'd1;
412         to_AC = 3'b000;
413         en_RegSelector = 1'd0;
414         rw_RegSelector = 1'd0;
415         NS = MOVE2;
416     end
417     MOVE2: begin
418         to_ALU = 4'd1;
419         to_BUS = 5'd0;
420         to_Decoder = 5'd0;
421         to_IM = 1'd0;
422         to_DM = 1'd0;
423         to_PC = 1'd0;
424         to_AC = 3'b100;
425         en_RegSelector = 1'd1;
426         rw_RegSelector = 1'd0;
427         NS = MOVE3;
428     end
429     MOVE3: begin
430         to_ALU = 4'd0;
431         to_BUS = 5'd2;
432         to_Decoder = 5'd1;
```

```
433         to_IM = 1'd0;
434         to_DM = 1'd0;
435         to_PC = 1'd0;
436         to_AC = 3'b000;
437         en_RegSelector = 1'd0;
438         rw_RegSelector = 1'd0;
439         NS = MOVE4;
440     end
441     MOVE4: begin
442         to_ALU = 4'd0;
443         to_BUS = 5'd0;
444         to_Decoder = 5'd6;
445         to_IM = 1'd0;
446         to_DM = 1'd0;
447         to_PC = 1'd1;
448         to_AC = 3'b000;
449         en_RegSelector = 1'd0;
450         rw_RegSelector = 1'd0;
451         NS = MOVE5;
452     end
453     MOVE5: begin
454         to_ALU = 4'd0;
455         to_BUS = 5'd20;
456         to_Decoder = 5'd0;
457         to_IM = 1'd0;
458         to_DM = 1'd0;
459         to_PC = 1'd0;
460         to_AC = 3'b000;
461         en_RegSelector = 1'd1;
462         rw_RegSelector = 1'd1;
463         NS = FETCH1;
464     end
465     ADD: begin
466         to_ALU = 4'd2;
467         to_BUS = 5'd17;
468         to_Decoder = 5'd0;
469         to_IM = 1'd0;
470         to_DM = 1'd0;
471         to_PC = 1'd0;
472         to_AC = 3'b100;
473         en_RegSelector = 1'd0;
474         rw_RegSelector = 1'd0;
475         NS = FETCH1;
476     end
477     SUB: begin
478         to_ALU = 4'd3;
479         to_BUS = 5'd17;
480         to_Decoder = 5'd0;
481         to_IM = 1'd0;
482         to_DM = 1'd0;
483         to_PC = 1'd0;
484         to_AC = 3'b100;
485         en_RegSelector = 1'd0;
486         rw_RegSelector = 1'd0;
```

```
487         NS = FETCH1;
488     end
489     MUL: begin
490         to_ALU = 4'd4;
491         to_BUS = 5'd17;
492         to_Decoder = 5'd0;
493         to_IM = 1'd0;
494         to_DM = 1'd0;
495         to_PC = 1'd0;
496         to_AC = 3'b100;
497         en_RegSelector = 1'd0;
498         rw_RegSelector = 1'd0;
499         NS = FETCH1;
500     end
501     AND: begin
502         to_ALU = 4'd5;
503         to_BUS = 5'd17;
504         to_Decoder = 5'd0;
505         to_IM = 1'd0;
506         to_DM = 1'd0;
507         to_PC = 1'd0;
508         to_AC = 3'b100;
509         en_RegSelector = 1'd0;
510         rw_RegSelector = 1'd0;
511         NS = FETCH1;
512     end
513     OR: begin
514         to_ALU = 4'd6;
515         to_BUS = 5'd17;
516         to_Decoder = 5'd0;
517         to_IM = 1'd0;
518         to_DM = 1'd0;
519         to_PC = 1'd0;
520         to_AC = 3'b100;
521         en_RegSelector = 1'd0;
522         rw_RegSelector = 1'd0;
523         NS = FETCH1;
524     end
525     JUMP1: begin
526         to_ALU = 4'd0;
527         to_BUS = 5'd0;
528         to_Decoder = 5'd6;
529         to_IM = 1'd0;
530         to_DM = 1'd0;
531         to_PC = 1'd1;
532         to_AC = 3'b000;
533         en_RegSelector = 1'd0;
534         rw_RegSelector = 1'd0;
535         NS = JUMP2;
536     end
537     JUMP2: begin
538         to_ALU = 4'd0;
539         to_BUS = 5'd0;
540         to_Decoder = 5'd0;
```



```

541         to_IM = 1'd0;
542         to_DM = 1'd0;
543         to_PC = 1'd0;
544         to_AC = 3'b000;
545         en_RegSelector = 1'd0;
546         rw_RegSelector = 1'd0;
547         if(((MDDR==16'd0 && z==1'b1) || (MDDR==16'd2 &&
i==1'b1) || (MDDR==16'd4 && j==1'b1) || (MDDR==16'd6 && k==1'b1) ||
(MDDR==16'd1 && z==1'b0) || (MDDR==16'd3 && i==1'b0) || (MDDR==16'd5 &&
j==1'b0) || (MDDR==16'd7 && k==1'b0)))
548             NS = JUMPY1;
549         else
550             NS = JUMPN1;
551     end
552     JUMPN1: begin
553         to_ALU = 4'd0;
554         to_BUS = 5'd0;
555         to_Decoder = 5'd0;
556         to_IM = 1'd0;
557         to_DM = 1'd0;
558         to_PC = 1'd1;
559         to_AC = 3'b000;
560         en_RegSelector = 1'd0;
561         rw_RegSelector = 1'd0;
562         NS = FETCH1;
563     end
564     JUMPY1: begin
565         to_ALU = 4'd0;
566         to_BUS = 5'd2;
567         to_Decoder = 5'd1;
568         to_IM = 1'd0;
569         to_DM = 1'd0;
570         to_PC = 1'd0;
571         to_AC = 3'b000;
572         en_RegSelector = 1'd0;
573         rw_RegSelector = 1'd0;
574         NS = JUMPY2;
575     end
576     JUMPY2: begin
577         to_ALU = 4'd0;
578         to_BUS = 5'd0;
579         to_Decoder = 5'd6;
580         to_IM = 1'd0;
581         to_DM = 1'd0;
582         to_PC = 1'd1;
583         to_AC = 3'b000;
584         en_RegSelector = 1'd0;
585         rw_RegSelector = 1'd0;
586         NS = JUMPY3;
587     end
588     JUMPY3: begin
589         to_ALU = 4'd0;
590         to_BUS = 5'd4;
591         to_Decoder = 5'd2;
592         to_IM = 1'd0;

```

```
593         to_DM = 1'd0;
594         to_PC = 1'd0;
595         to_AC = 3'b000;
596         en_RegSelector = 1'd0;
597         rw_RegSelector = 1'd0;
598         NS = FETCH1;
599     end
600     STORM1: begin
601         to_ALU = 4'd0;
602         to_BUS = 5'd7;
603         to_Decoder = 5'd1;
604         to_IM = 1'd0;
605         to_DM = 1'd0;
606         to_PC = 1'd0;
607         to_AC = 3'b000;
608         en_RegSelector = 1'd0;
609         rw_RegSelector = 1'd0;
610         NS = STORM2;
611     end
612     STORM2: begin
613         to_ALU = 4'd0;
614         to_BUS = 5'd20;
615         to_Decoder = 5'd4;
616         to_IM = 1'd0;
617         to_DM = 1'd1;
618         to_PC = 1'd0;
619         to_AC = 3'b000;
620         en_RegSelector = 1'd0;
621         rw_RegSelector = 1'd0;
622         NS = FETCH1;
623     end
624 endcase
625 end
626 endmodule
```

7.3 DECODER.v

```

1 module DECODER(
2     input clk,
3     input [5:0] from_cu,
4     input [5:0] from_selector,
5     output reg [0:0] out_AR,
6     output reg [0:0] out_PC,
7     output reg [0:0] out_MIDR,
8     output reg [0:0] out_MDDR_BUS,
9     output reg [0:0] out_MDDR_IM,
10    output reg [0:0] out_MDDR_DM,
11    output reg [0:0] out_Base_C,
12    output reg [0:0] out_I,
13    output reg [0:0] out_I_Ref,
14    output reg [0:0] out_Base_A,
15    output reg [0:0] out_J,
16    output reg [0:0] out_J_Ref,
17    output reg [0:0] out_Base_B,
18    output reg [0:0] out_K,
19    output reg [0:0] out_K_Ref,
20    output reg [0:0] out_P,
21    output reg [0:0] out_R
22 );
23 reg [5:0] reg_select;
24 always @(*)
25 begin
26     if (from_selector==5'd0)reg_select=from_cu;
27     else reg_select=from_selector;
28     case (reg_select)
29         5'd0:begin
30             out_AR=0;
31             out_PC=0;
32             out_MIDR=0;
33             out_MDDR_BUS=0;
34             out_MDDR_IM=0;
35             out_MDDR_DM=0;
36             out_Base_C=0;
37             out_I=0;
38             out_I_Ref=0;
39             out_Base_A=0;
40             out_J=0;
41             out_J_Ref=0;
42             out_Base_B=0;
43             out_K=0;
44             out_K_Ref=0;
45             out_P=0;
46             out_R=0;
47         end
48         5'd1:begin
49             out_AR=1;
50             out_PC=0;
51             out_MIDR=0;
52             out_MDDR_BUS=0;
53             out_MDDR_IM=0;
54             out_MDDR_DM=0;

```

```
55         out_Base_C=0;
56         out_I=0;
57         out_I_Ref=0;
58         out_Base_A=0;
59         out_J=0;
60         out_J_Ref=0;
61         out_Base_B=0;
62         out_K=0;
63         out_K_Ref=0;
64         out_P=0;
65         out_R=0;
66     end
67     5'd2:begin
68         out_AR=0;
69         out_PC=1;
70         out_MIDR=0;
71         out_MDDR_BUS=0;
72         out_MDDR_IM=0;
73         out_MDDR_DM=0;
74         out_Base_C=0;
75         out_I=0;
76         out_I_Ref=0;
77         out_Base_A=0;
78         out_J=0;
79         out_J_Ref=0;
80         out_Base_B=0;
81         out_K=0;
82         out_K_Ref=0;
83         out_P=0;
84         out_R=0;
85     end
86     5'd3:begin
87         out_AR=0;
88         out_PC=0;
89         out_MIDR=1;
90         out_MDDR_BUS=0;
91         out_MDDR_IM=0;
92         out_MDDR_DM=0;
93         out_Base_C=0;
94         out_I=0;
95         out_I_Ref=0;
96         out_Base_A=0;
97         out_J=0;
98         out_J_Ref=0;
99         out_Base_B=0;
100        out_K=0;
101        out_K_Ref=0;
102        out_P=0;
103        out_R=0;
104    end
105    5'd4:begin
106        out_AR=0;
107        out_PC=0;
108        out_MIDR=0;
```

```
109         out_MDDR_BUS=1;
110         out_MDDR_IM=0;
111         out_MDDR_DM=0;
112         out_Base_C=0;
113         out_I=0;
114         out_I_Ref=0;
115         out_Base_A=0;
116         out_J=0;
117         out_J_Ref=0;
118         out_Base_B=0;
119         out_K=0;
120         out_K_Ref=0;
121         out_P=0;
122         out_R=0;
123     end
124     5'd5:begin
125         out_AR=0;
126         out_PC=0;
127         out_MIDR=0;
128         out_MDDR_BUS=0;
129         out_MDDR_IM=0;
130         out_MDDR_DM=1;
131         out_Base_C=0;
132         out_I=0;
133         out_I_Ref=0;
134         out_Base_A=0;
135         out_J=0;
136         out_J_Ref=0;
137         out_Base_B=0;
138         out_K=0;
139         out_K_Ref=0;
140         out_P=0;
141         out_R=0;
142     end
143     5'd6:begin
144         out_AR=0;
145         out_PC=0;
146         out_MIDR=0;
147         out_MDDR_BUS=0;
148         out_MDDR_IM=1;
149         out_MDDR_DM=0;
150         out_Base_C=0;
151         out_I=0;
152         out_I_Ref=0;
153         out_Base_A=0;
154         out_J=0;
155         out_J_Ref=0;
156         out_Base_B=0;
157         out_K=0;
158         out_K_Ref=0;
159         out_P=0;
160         out_R=0;
161     end
162     5'd7:begin
```

```
163         out_AR=0;
164         out_PC=0;
165         out_MIDR=0;
166         out_MDDR_BUS=0;
167         out_MDDR_IM=0;
168         out_MDDR_DM=0;
169         out_Base_C=1;
170         out_I=0;
171         out_I_Ref=0;
172         out_Base_A=0;
173         out_J=0;
174         out_J_Ref=0;
175         out_Base_B=0;
176         out_K=0;
177         out_K_Ref=0;
178         out_P=0;
179         out_R=0;
180     end
181     5'd8:begin
182         out_AR=0;
183         out_PC=0;
184         out_MIDR=0;
185         out_MDDR_BUS=0;
186         out_MDDR_IM=0;
187         out_MDDR_DM=0;
188         out_Base_C=0;
189         out_I=1;
190         out_I_Ref=0;
191         out_Base_A=0;
192         out_J=0;
193         out_J_Ref=0;
194         out_Base_B=0;
195         out_K=0;
196         out_K_Ref=0;
197         out_P=0;
198         out_R=0;
199     end
200     5'd9:begin
201         out_AR=0;
202         out_PC=0;
203         out_MIDR=0;
204         out_MDDR_BUS=0;
205         out_MDDR_IM=0;
206         out_MDDR_DM=0;
207         out_Base_C=0;
208         out_I=0;
209         out_I_Ref=1;
210         out_Base_A=0;
211         out_J=0;
212         out_J_Ref=0;
213         out_Base_B=0;
214         out_K=0;
215         out_K_Ref=0;
216         out_P=0;
```

```
217         out_R=0;
218     end
219     5'd10:begin
220         out_AR=0;
221         out_PC=0;
222         out_MIDR=0;
223         out_MDDR_BUS=0;
224         out_MDDR_IM=0;
225         out_MDDR_DM=0;
226         out_Base_C=0;
227         out_I=0;
228         out_I_Ref=0;
229         out_Base_A=1;
230         out_J=0;
231         out_J_Ref=0;
232         out_Base_B=0;
233         out_K=0;
234         out_K_Ref=0;
235         out_P=0;
236         out_R=0;
237     end
238     5'd11:begin
239         out_AR=0;
240         out_PC=0;
241         out_MIDR=0;
242         out_MDDR_BUS=0;
243         out_MDDR_IM=0;
244         out_MDDR_DM=0;
245         out_Base_C=0;
246         out_I=0;
247         out_I_Ref=0;
248         out_Base_A=0;
249         out_J=1;
250         out_J_Ref=0;
251         out_Base_B=0;
252         out_K=0;
253         out_K_Ref=0;
254         out_P=0;
255         out_R=0;
256     end
257     5'd12:begin
258         out_AR=0;
259         out_PC=0;
260         out_MIDR=0;
261         out_MDDR_BUS=0;
262         out_MDDR_IM=0;
263         out_MDDR_DM=0;
264         out_Base_C=0;
265         out_I=0;
266         out_I_Ref=0;
267         out_Base_A=0;
268         out_J=0;
269         out_J_Ref=1;
270         out_Base_B=0;
```

```
271         out_K=0;
272         out_K_Ref=0;
273         out_P=0;
274         out_R=0;
275     end
276     5'd13:begin
277         out_AR=0;
278         out_PC=0;
279         out_MIDR=0;
280         out_MDDR_BUS=0;
281         out_MDDR_IM=0;
282         out_MDDR_DM=0;
283         out_Base_C=0;
284         out_I=0;
285         out_I_Ref=0;
286         out_Base_A=0;
287         out_J=0;
288         out_J_Ref=0;
289         out_Base_B=1;
290         out_K=0;
291         out_K_Ref=0;
292         out_P=0;
293         out_R=0;
294     end
295     5'd14:begin
296         out_AR=0;
297         out_PC=0;
298         out_MIDR=0;
299         out_MDDR_BUS=0;
300         out_MDDR_IM=0;
301         out_MDDR_DM=0;
302         out_Base_C=0;
303         out_I=0;
304         out_I_Ref=0;
305         out_Base_A=0;
306         out_J=0;
307         out_J_Ref=0;
308         out_Base_B=0;
309         out_K=1;
310         out_K_Ref=0;
311         out_P=0;
312         out_R=0;
313     end
314     5'd15:begin
315         out_AR=0;
316         out_PC=0;
317         out_MIDR=0;
318         out_MDDR_BUS=0;
319         out_MDDR_IM=0;
320         out_MDDR_DM=0;
321         out_Base_C=0;
322         out_I=0;
323         out_I_Ref=0;
324         out_Base_A=0;
```



```
325         out_J=0;
326         out_J_Ref=0;
327         out_Base_B=0;
328         out_K=0;
329         out_K_Ref=1;
330         out_P=0;
331         out_R=0;
332     end
333     5'd16:begin
334         out_AR=0;
335         out_PC=0;
336         out_MIDR=0;
337         out_MDDR_BUS=0;
338         out_MDDR_IM=0;
339         out_MDDR_DM=0;
340         out_Base_C=0;
341         out_I=0;
342         out_I_Ref=0;
343         out_Base_A=0;
344         out_J=0;
345         out_J_Ref=0;
346         out_Base_B=0;
347         out_K=0;
348         out_K_Ref=0;
349         out_P=1;
350         out_R=0;
351     end
352     5'd17:begin
353         out_AR=0;
354         out_PC=0;
355         out_MIDR=0;
356         out_MDDR_BUS=0;
357         out_MDDR_IM=0;
358         out_MDDR_DM=0;
359         out_Base_C=0;
360         out_I=0;
361         out_I_Ref=0;
362         out_Base_A=0;
363         out_J=0;
364         out_J_Ref=0;
365         out_Base_B=0;
366         out_K=0;
367         out_K_Ref=0;
368         out_P=0;
369         out_R=1;
370     end
371     5'd18:begin
372         out_AR=1;
373         out_PC=0;
374         out_MIDR=0;
375         out_MDDR_BUS=0;
376         out_MDDR_IM=1;
377         out_MDDR_DM=0;
378         out_Base_C=0;
```

```
379         out_I=0;
380         out_I_Ref=0;
381         out_Base_A=0;
382         out_J=0;
383         out_J_Ref=0;
384         out_Base_B=0;
385         out_K=0;
386         out_K_Ref=0;
387         out_P=0;
388         out_R=0;
389     end
390     5'd19:begin
391         out_AR=1;
392         out_PC=0;
393         out_MIDR=0;
394         out_MDDR_BUS=0;
395         out_MDDR_IM=0;
396         out_MDDR_DM=1;
397         out_Base_C=0;
398         out_I=0;
399         out_I_Ref=0;
400         out_Base_A=0;
401         out_J=0;
402         out_J_Ref=0;
403         out_Base_B=0;
404         out_K=0;
405         out_K_Ref=0;
406         out_P=0;
407         out_R=0;
408     end
409     default:
410     begin
411         out_AR=0;
412         out_PC=0;
413         out_MIDR=0;
414         out_MDDR_BUS=0;
415         out_MDDR_IM=0;
416         out_MDDR_DM=0;
417         out_Base_C=0;
418         out_I=0;
419         out_I_Ref=0;
420         out_Base_A=0;
421         out_J=0;
422         out_J_Ref=0;
423         out_Base_B=0;
424         out_K=0;
425         out_K_Ref=0;
426         out_P=0;
427         out_R=0;
428     end
429 endcase
430 end
431 endmodule
```

7.4 DRAM.v

```

1 module DRAM(
2     input clk,
3     input [3:0]write,
4     input [63:0] address ,
5     input [63:0] data_in ,
6     output reg [63:0] data_out);
7     reg [15:0] MEM [65535:0];
8     initial begin
9         MEM[0]=16'd5; //mat_a_base
10        MEM[1]=16'd32; //mat_b_base
11        MEM[2]=16'd57; //mat_c_base
12
13        MEM[3]=16'd5; //i_ref
14        MEM[4]=16'd5; //j_ref
15
16        //[13, 17, 1, 5, 13]
17        MEM[5]=16'd13;
18        MEM[6]=16'd17;
19        MEM[7]=16'd1;
20        MEM[8]=16'd5;
21        MEM[9]=16'd13;
22        //[5, 20, 18, 14, 5]
23        MEM[10]=16'd5;
24        MEM[11]=16'd20;
25        MEM[12]=16'd18;
26        MEM[13]=16'd14;
27        MEM[14]=16'd5;
28        //[2, 2, 10, 1, 17]
29        MEM[15]=16'd2;
30        MEM[16]=16'd2;
31        MEM[17]=16'd10;
32        MEM[18]=16'd1;
33        MEM[19]=16'd17;
34        //[0, 10, 13, 5, 3]
35        MEM[20]=16'd0;
36        MEM[21]=16'd10;
37        MEM[22]=16'd13;
38        MEM[23]=16'd5;
39        MEM[24]=16'd3;
40        //[12, 8, 6, 16, 5]
41        MEM[25]=16'd12;
42        MEM[26]=16'd8;
43        MEM[27]=16'd6;
44        MEM[28]=16'd16;
45        MEM[29]=16'd5;
46
47        MEM[30]=16'd5; //j_ref
48        MEM[31]=16'd5; //k_ref
49
50        //[8, 18, 18, 13, 13]
51        MEM[32]=16'd8;
52        MEM[33]=16'd18;
53        MEM[34]=16'd18;
54        MEM[35]=16'd13;

```

```

55     MEM[36]=16'd13;
56     //[20, 15, 7, 4, 9]
57     MEM[37]=16'd20;
58     MEM[38]=16'd15;
59     MEM[39]=16'd7;
60     MEM[40]=16'd4;
61     MEM[41]=16'd9;
62     //[19, 10, 18, 10, 8]
63     MEM[42]=16'd19;
64     MEM[43]=16'd10;
65     MEM[44]=16'd18;
66     MEM[45]=16'd10;
67     MEM[46]=16'd8;
68     //[3, 9, 10, 12, 10]
69     MEM[47]=16'd3;
70     MEM[48]=16'd9;
71     MEM[49]=16'd10;
72     MEM[50]=16'd12;
73     MEM[51]=16'd10;
74     //[3, 9, 20, 19, 15]
75     MEM[52]=16'd3;
76     MEM[53]=16'd9;
77     MEM[54]=16'd20;
78     MEM[55]=16'd19;
79     MEM[56]=16'd15;
80
81     end
82     always @(posedge clk)
83     begin
84         if (write[0:0] == 1)
85             begin
86                 MEM[address[15:0]] <= data_in[15:0];
87                 $display("address0:- %d, mem_data_in0:-
88 %d",address[15:0],data_in[15:0]);
89             end
90             if (write[1:1] == 1)
91                 begin
92                     MEM[address[31:16]] <= data_in[31:16];
93                     $display("address1:- %d, mem_data_in1:-
94 %d",address[31:16],data_in[31:16]);
95                 end
96                 if (write[2:2] == 1)
97                     begin
98                         MEM[address[47:32]] <= data_in[47:32];
99                         $display("address2:- %d, mem_data_in2:-
100 %d",address[47:32],data_in[47:32]);
101                     end
102                     if (write[3:3] == 1)
103                         begin
104                             MEM[address[63:48]] <= data_in[63:48];
105                             $display("address3:- %d, mem_data_in3:-
106 %d",address[63:48],data_in[63:48]);
107                         end
108                         data_out[15:0] <= MEM[address[15:0]];
109                         data_out[31:16] <= MEM[address[31:16]];

```

```
106 |           data_out[47:32] <= MEM[address[47:32]];
107 |           data_out[63:48] <= MEM[address[63:48]];
108 |           end
109 | endmodule
```

7.5 IRAM.v

```

1 module IRAM(
2     input clk,
3     input [3:0]write,
4     input [63:0] address,
5     input [63:0] instr_in,
6     output reg [63:0] instr_out
7 );
8 reg [15:0] MEM [65535:0];
9 parameter FETCH = 8'd0;
10 parameter NOP = 8'd3;
11 parameter END = 8'd4;
12 parameter CLR = 8'd5;
13 parameter LOAD = 8'd8;
14 parameter LOADM = 8'd14;
15 parameter STAC = 8'd18;
16 parameter INC = 8'd21;
17 parameter INCAC = 8'd27;
18 parameter JUMP = 8'd28;
19 parameter MOVE = 8'd34;
20 parameter ADD = 8'd39;
21 parameter SUB = 8'd40;
22 parameter MUL = 8'd41;
23 parameter AND = 8'd42;
24 parameter OR = 8'd43;
25 parameter STORM = 8'd44;
26 initial begin
27
28     MEM[0] = LOAD; //load 0 BASE_A
29     MEM[1] = 16'd0;
30     MEM[2] = 16'd10;
31     MEM[3] = LOAD; //load 1 BASE_B
32     MEM[4] = 16'd1;
33     MEM[5] = 16'd13;
34     MEM[6] = LOAD; //load 2 BASE_C
35     MEM[7] = 16'd2;
36     MEM[8] = 16'd7;
37     MEM[9] = LOAD; //load 3 i_ref
38     MEM[10] = 16'd3;
39     MEM[11] = 16'd9;
40     MEM[12] = LOAD; //load 4 j_ref
41     MEM[13] = 16'd4;
42     MEM[14] = 16'd12;
43     MEM[15] = LOAD;
44     MEM[16] = 16'd31;
45     MEM[17] = 16'd15;
46     MEM[18] = CLR;
47     MEM[19] = 16'd8;
48     MEM[20] = MOVE;
49     MEM[21] = 16'd23;
50     MEM[22] = 16'd14;
51     MEM[23] = JUMP;
52     MEM[24] = 16'd2;
53     MEM[25] = 16'd86;
54     MEM[26] = JUMP;

```

```
55     MEM[27] = 16'd6;
56     MEM[28] = 16'd70;
57     MEM[29] = CLR;
58     MEM[30] = 16'd16;
59     MEM[31] = CLR;
60     MEM[32] = 16'd11;
61     MEM[33] = JUMP;
62     MEM[34] = 16'd4;
63     MEM[35] = 16'd60;
64     MEM[36] = LOADM;
65     MEM[37] = 16'd21;
66     MEM[38] = MOVE;
67     MEM[39] = 16'd20;
68     MEM[40] = 16'd17;
69     MEM[41] = LOADM;
70     MEM[42] = 16'd22;
71     MEM[43] = MUL;
72     MEM[44] = MOVE;
73     MEM[45] = 16'd20;
74     MEM[46] = 16'd17;
75     MEM[47] = MOVE;
76     MEM[48] = 16'd16;
77     MEM[49] = 16'd20;
78     MEM[50] = ADD;
79     MEM[51] = MOVE;
80     MEM[52] = 16'd20;
81     MEM[53] = 16'd16;
82     MEM[54] = INC;
83     MEM[55] = 16'd1;
84     MEM[56] = 16'd11;
85     MEM[57] = JUMP;
86     MEM[58] = 16'd5;
87     MEM[59] = 16'd33;
88     MEM[60] = MOVE;
89     MEM[61] = 16'd16;
90     MEM[62] = 16'd20;
91     MEM[63] = STORM;
92     MEM[64] = INC;
93     MEM[65] = 16'd1;
94     MEM[66] = 16'd14;
95     MEM[67] = JUMP;
96     MEM[68] = 16'd7;
97     MEM[69] = 16'd26;
98     MEM[70] = INC;
99     MEM[71] = 16'd1;
100    MEM[72] = 16'd8;
101    MEM[73] = MOVE;
102    MEM[74] = 16'd15;
103    MEM[75] = 16'd17;
104    MEM[76] = MOVE;
105    MEM[77] = 16'd14;
106    MEM[78] = 16'd20;
107    MEM[79] = SUB;
108    MEM[80] = MOVE;
```

```
109     MEM[81] = 16'd20;
110     MEM[82] = 16'd14;
111     MEM[83] = JUMP;
112     MEM[84] = 16'd3;
113     MEM[85] = 16'd23;
114     MEM[86] = END;
115
116 end
117     always @(posedge clk)
118     begin
119         if (write[0:0] == 1)
120         begin
121             MEM[address[15:0]] <= instr_in[15:0];
122         end
123         if (write[1:1] == 1)
124         begin
125             MEM[address[31:16]] <= instr_in[31:16];
126         end
127         if (write[2:2] == 1)
128         begin
129             MEM[address[47:32]] <= instr_in[47:32];
130         end
131         if (write[3:3] == 1)
132         begin
133             MEM[address[63:48]] <= instr_in[63:48];
134         end
135         instr_out[15:0] <= MEM[address[15:0]];
136         instr_out[31:16] <= MEM[address[31:16]];
137         instr_out[47:32] <= MEM[address[47:32]];
138         instr_out[63:48] <= MEM[address[63:48]];
139     end
140 endmodule
```


7.6 GENERAL_PURPOSE_REGISTER.v

```
1 module GENERAL_PURPOSE_REGISTER
2     (input clk,write,
3      input [15:0] data_in,
4      output reg [15:0] data_out
5     );
6     always @(negedge clk)
7         begin
8             if (write) data_out <= data_in;
9         end
10 endmodule
```

7.7 MAIN_ALU.v

```
1 module MAIN_ALU
2   ( input clk,
3     input [15:0] data_bus_alu,
4     input [15:0] data_ac_alu,
5     input [3:0] op_code,
6     output reg [15:0] out,
7     output reg [0:0] zflag = 1'd0
8   );
9   always @(negedge clk)
10     begin
11       case (op_code)
12         4'b0000: {zflag,out}= {zflag,out};
13         4'b0001: {zflag,out}= data_bus_alu;
14         4'b0010: {zflag,out}= data_ac_alu + data_bus_alu;
15         4'b0011: {zflag,out}= data_ac_alu - data_bus_alu;
16         4'b0100: {zflag,out}= data_ac_alu * data_bus_alu;
17         4'b0101: {zflag,out}= data_ac_alu & data_bus_alu;
18         4'b0110: {zflag,out}= data_ac_alu | data_bus_alu;
19         4'b0111: {zflag,out}= data_ac_alu + 1;
20         4'b1000: {zflag,out}= 0;
21         default: out = 0;
22       endcase
23       if(out == 16'd0) zflag = 1'd1;
24     end
25 endmodule
```

7.8 MAT_X.v

```
1 //X colum Y row
2 module MAT_X
3     (input clk,
4      input [15:0] X_Ref,
5      input [15:0] X, //Counter 1
6      input [15:0] Base, //Starting Address
7      input [15:0] Y, //Counter 2
8      output reg [15:0] Mat_data_out
9     );
10     always @(*)
11         begin
12             Mat_data_out <= Y*X_Ref + X+Base;
13         end
14 endmodule
```

7.9 REG_X.v

```
1  `timescale 1ns / 1ps
2  module REG_X
3      (input clk,write_x,write_xref,
4       input [15:0] data_in,
5       output reg [0:0] xflag,
6       output reg [15:0] data_out_x,
7       output reg [15:0] data_out_xref);
8      always @(negedge clk)
9          begin
10             if (write_x) data_out_x = data_in;
11             if (write_xref) data_out_xref = data_in;
12             if(data_out_x < data_out_xref) xflag = 1'd0;
13             else xflag = 1'd1;
14         end
15 endmodule
```

7.10 MDDR.v

```
1 module MDDR(  
2     input clk,write_ins,write_data,write_bus,  
3     input [15:0] instr_iram_mddr,  
4     input [15:0] data_dram_mddr,  
5     input [15:0] data_bus_mddr,  
6     output reg [15:0] data_out  
7 );  
8     always @(negedge clk)  
9         begin  
10             if (write_ins) data_out <= instr_iram_mddr;  
11             if (write_bus) data_out <= data_bus_mddr;  
12             if (write_data) data_out <= data_dram_mddr;  
13         end  
14 endmodule
```

7.11 PC.v

```
1 module PC
2     (input clk,write,incpc,
3      input [15:0] data_in,
4      output reg [15:0] data_out = 16'd0
5     );
6     always @(negedge clk)
7         begin
8             if (write) data_out <= data_in;
9             else if (incpc) data_out <= data_out + 1;
10        end
11 endmodule
```

7.12 REGISTER_SELECTOR.v

```
1 module REGISTER_SELECTOR(  
2     input clk,  
3     input [0:0]en,  
4     input [0:0]rw,  
5     input [15:0] fromMDDR,  
6     output reg [5:0] toDecoder = 6'd0,  
7     output reg [4:0] toBus = 4'd0  
8 );  
9 always @(*)  
10 begin  
11     if (en==0)  
12         begin  
13             toDecoder=0;  
14             toBus=0;  
15         end  
16     else  
17         begin  
18             if(rw==0)  
19                 begin  
20                     toBus = fromMDDR[4:0];  
21                     toDecoder = 6'd0;  
22                 end  
23             else if(rw==1)  
24                 begin  
25                     toDecoder = fromMDDR[5:0];  
26                     toBus = 4'd0;  
27                 end  
28             end  
29         end  
30 endmodule
```

7.13 CORE.v

```

1 module CORE(
2     input clk,
3     input [15:0]data_DM_core,//from data mem to proc(MDDR)
4     input [15:0]data_IM_core,
5     input [15:0]proId,
6     output reg [15:0]data_core_IM,
7     output reg [15:0]data_core_DM,//from processor(MDDR) to data mem (data)
8     output reg [0:0]DM_en,//from processor to data mem (control signal)
9     output reg [0:0]IM_en,
10    output reg [15:0]AR_out,
11    output reg finish,
12    output reg [15:0]bus,
13    output reg [7:0]current_micro_instruction
14
15 );
16 wire [7:0]current_micro_instruction_wire;
17 wire to_DM;
18 wire to_IM;
19 wire [0:0]wire_decoder_registers [16:0];
20 wire [15:0] bus_wire;
21 wire [15:0] MIDR_wire;
22 wire [15:0] wire_iram_mddr;
23 wire [15:0] wire_dram_mddr;
24 wire [15:0] Base_A_wire;
25 wire [15:0] Base_B_wire;
26 wire [15:0] Base_C_wire;
27 wire [15:0] data_alu_ac;
28 wire [15:0] wire_registers_busMultiplexer [12:0];
29 wire [0:0] PC_inc_wire;
30 wire [0:0] iflag_wire;
31 wire [0:0] jflag_wire;
32 wire [0:0] kflag_wire;
33 wire [0:0] zflag_wire;
34 wire [15:0] i_ref_wire;
35 wire [15:0] j_ref_wire;
36 wire [15:0] k_ref_wire;
37 wire [3:0] opcode_wire;
38 wire [2:0] ac_cu_wire;
39 wire [5:0] wire_cu_decoder;//select reg by cu
40 wire [5:0] wire_regSelector_decoder;//select read/write by reg selector m
41 wire [4:0] bus_selector_from_cu; //
42 wire [4:0] bus_selector_from_register_selector;
43 wire [0:0] en_register_selector;//form CU - enable
44 wire [0:0] rw_register_selector;//from cu - read/wrie select
45 wire [0:0] finish_wire;//from cu - read/wrie select
46
47 assign wire_iram_mddr = data_IM_core;
48 assign wire_dram_mddr = data_DM_core;
49
50 always@(*)
51 begin
52     current_micro_instruction <= current_micro_instruction_wire;
53     bus <= bus_wire;
54     IM_en <= to_IM;

```



```

55     DM_en <= to_DM;
56     AR_out <= wire_registers_busMultiplexer[0];
57     data_core_IM <= wire_registers_busMultiplexer[2];
58     data_core_DM <= wire_registers_busMultiplexer[2];
59     finish <= finish_wire ;
60     if(finish==1'd1) $display("finish");
61     //$display("AR:- %d , I:- %d, I_ref:- %d, J:- %d, J_ref:- %d, base_A:
, wire_registers_busMultiplexer[0], wire_registers_busMultiplexer[4], i_ref_wire
62     end
63
64     GENERAL_PURPOSE_REGISTER ID (.clk(clk), .write(1'd1), .data_in(proId), .d
65
66     GENERAL_PURPOSE_REGISTER AR (.clk(clk), .write(wire_decoder_registers[0]
67
68     PC PC (.clk(clk), .write(wire_decoder_registers[1]), .incpc(PC_inc_wire)
69
70     GENERAL_PURPOSE_REGISTER MIDR (.clk(clk), .write(wire_decoder_registers[
71
72     MDDR MDDR (.clk(clk), .write_ins(wire_decoder_registers[4]), .write_data(
73     .write_bus(wire_decoder_registers[3]), .instr_iram_mddr(wire_iram_mddr
74     .data_bus_mddr(bus_wire), .data_out(wire_registers_busMultiplexer[2]))
75
76     GENERAL_PURPOSE_REGISTER BASE_C (.clk(clk), .write(wire_decoder_register
77
78     REG_X I (.clk(clk), .write_x(wire_decoder_registers[7]), .write_xref(wire
79     .data_out_x(wire_registers_busMultiplexer[4]), .data_out_xref(i_ref_w
80
81     REG_X J (.clk(clk), .write_x(wire_decoder_registers[10]), .write_xref(wir
82     .xflag(jflag_wire), .data_out_x(wire_registers_busMultiplexer[5]), .d
83
84     REG_X K (.clk(clk), .write_x(wire_decoder_registers[13]), .write_xref(wir
85     .xflag(kflag_wire), .data_out_x(wire_registers_busMultiplexer[6]), .d
86
87     GENERAL_PURPOSE_REGISTER BASE_A (.clk(clk), .write(wire_decoder_register
88     .data_out(Base_A_wire));
89
90     GENERAL_PURPOSE_REGISTER BASE_B (.clk(clk), .write(wire_decoder_register
91     .data_out(Base_B_wire));
92
93     MAT_X MAT_A (.clk(clk), .X_Ref(j_ref_wire), .X(wire_registers_busMultipl
94     .Y(wire_registers_busMultiplexer[4]), .Mat_data_out(wire_registers_busl
95
96     MAT_X MAT_B (.clk(clk), .X_Ref(k_ref_wire), .X(wire_registers_busMultipl
97     .Y(wire_registers_busMultiplexer[5]), .Mat_data_out(wire_registers_busl
98
99     MAT_X MAT_C (.clk(clk), .X_Ref(k_ref_wire), .X(wire_registers_busMultipl
100    .Y(wire_registers_busMultiplexer[4]), .Mat_data_out(wire_registers_busl
101
102    GENERAL_PURPOSE_REGISTER P (.clk(clk), .write(wire_decoder_registers[15]
103    .data_out(wire_registers_busMultiplexer[7]));
104
105    GENERAL_PURPOSE_REGISTER R (.clk(clk), .write(wire_decoder_registers[16]
106    .data_out(wire_registers_busMultiplexer[8]));
107
108    MAIN_ALU ALU (.clk(clk), .data_bus_alu(bus_wire), .data_ac_alu(wire_regi

```

```

109         .op_code(opcode_wire), .out(data_alu_ac), .zflag(zflag_wire));
110
111     AC AC (.clk(clk), .write(ac_cu_wire),
112         .data_alu_ac(data_alu_ac), .data_out(wire_registers_busMultiplexer[9]
113
114     DECODER DECODER (
115         .clk(clk),
116         .from_cu(wire_cu_decoder),
117         .from_selector(wire_regSelector_decoder),
118         .out_AR(wire_decoder_registers[0]),
119         .out_PC(wire_decoder_registers[1]),
120         .out_MIDR(wire_decoder_registers[2]),
121         .out_MDDR_BUS(wire_decoder_registers[3]),
122         .out_MDDR_IM(wire_decoder_registers[4]),
123         .out_MDDR_DM(wire_decoder_registers[5]),
124         .out_BASE_C(wire_decoder_registers[6]),
125         .out_I(wire_decoder_registers[7]),
126         .out_I_Ref(wire_decoder_registers[8]),
127         .out_Base_A(wire_decoder_registers[9]),
128         .out_J(wire_decoder_registers[10]),
129         .out_J_Ref(wire_decoder_registers[11]),
130         .out_Base_B(wire_decoder_registers[12]),
131         .out_K(wire_decoder_registers[13]),
132         .out_K_Ref(wire_decoder_registers[14]),
133         .out_P(wire_decoder_registers[15]),
134         .out_R(wire_decoder_registers[16])
135     );
136     BUS_MULTIPLEXER BUS_MULTIPLEXER(
137         .clk(clk),
138         .from_selector(bus_selector_from_register_selector),
139         .from_cu(bus_selector_from_cu),
140         .AR(wire_registers_busMultiplexer[0]),
141         .PC(wire_registers_busMultiplexer[1]),
142         .R(wire_registers_busMultiplexer[8]),
143         .P(wire_registers_busMultiplexer[7]),
144         .Mat_A(wire_registers_busMultiplexer[10]),
145         .Mat_B(wire_registers_busMultiplexer[11]),
146         .MDDR(wire_registers_busMultiplexer[2]),
147         .AC(wire_registers_busMultiplexer[9]),
148         .I(wire_registers_busMultiplexer[4]),
149         .I_ref(i_ref_wire),
150         .J(wire_registers_busMultiplexer[5]),
151         .J_ref(j_ref_wire),
152         .K(wire_registers_busMultiplexer[6]),
153         .K_ref(k_ref_wire),
154         .Mat_C(wire_registers_busMultiplexer[3]),
155         .proId(wire_registers_busMultiplexer[12]),
156         .out(bus_wire)
157     );
158     CONTROL_UNIT CONTROL_UNIT(
159         .clk(clk),
160         .z(zflag_wire),
161         .i(iflag_wire),
162         .j(jflag_wire),

```

```
163     .k(kflag_wire),
164     .instruction(MIDR_wire),
165     .MDDR(wire_registers_busMultiplexer[2]),
166     .to_ALU(opcode_wire),
167     .to_DM(to_DM),
168     .to_IM(to_IM),
169     .to_Decoder(wire_cu_decoder),
170     .to_BUS(bus_selector_from_cu),
171     .to_PC(PC_inc_wire),
172     .to_AC(ac_cu_wire),
173     .en_RegSelector(en_register_selector),
174     .rw_RegSelector(rw_register_selector),
175     .finish(finish_wire),
176     .current_micro_instruction(current_micro_instruction_wire)
177 );
178 REGISTER_SELECTOR REGISTER_SELECTOR(
179     .clk(clk),
180     .en(en_register_selector),
181     .rw(rw_register_selector),
182     .fromMDDR(wire_registers_busMultiplexer[2]),
183     .toDecoder(wire_regSelector_decoder),
184     .toBus(bus_selector_from_register_selector)
185 );
186
187 endmodule
```

7.14 PROCESSOR.v

```

1  `timescale 1ns / 1ps
2
3  module PROCESSOR(
4
5  input  clk,
6      output reg [15:0]bus_core1,
7      output reg [7:0]current_micro_instruction_core1,
8      output reg [15:0]bus_core2,
9      output reg [7:0]current_micro_instruction_core2,
10     output reg [15:0]bus_core3,
11     output reg [7:0]current_micro_instruction_core3,
12     output reg [15:0]bus_core4,
13     output reg [7:0]current_micro_instruction_core4,
14     output reg [3:0]finish
15 );
16 wire [7:0]current_micro_instruction_wire_core1;
17 wire [7:0]current_micro_instruction_wire_core2;
18 wire [7:0]current_micro_instruction_wire_core3;
19 wire [7:0]current_micro_instruction_wire_core4;
20 wire [3:0]en_from_cpu_to_IM;
21 wire [3:0]en_from_cpu_to_DM;
22 wire [63:0]data_from_cpu_to_IM;
23 wire [63:0]data_from_cpu_to_DM;
24 wire [63:0]data_from_IM_to_cpu;
25 wire [63:0]data_from_DM_to_cpu;
26 wire [63:0]data_from_AR_to_M;
27 wire [15:0]bus_wire_core1;
28 wire [15:0]bus_wire_core2;
29 wire [15:0]bus_wire_core3;
30 wire [15:0]bus_wire_core4;
31 wire [3:0]finish_wire;
32 parameter proId0 =16'd0 ;
33 parameter proId1 =16'd1 ;
34 always @(*)
35 begin
36
37     current_micro_instruction_core1<=current_micro_instruction_wire_core1;
38     bus_core1 <= bus_wire_core1;
39
40     current_micro_instruction_core2<=current_micro_instruction_wire_core2;
41     bus_core2 <= bus_wire_core2;
42
43     current_micro_instruction_core3<=current_micro_instruction_wire_core3;
44     bus_core3 <= bus_wire_core3;
45
46     current_micro_instruction_core4<=current_micro_instruction_wire_core4;
47     bus_core4 <= bus_wire_core4;
48     finish<=finish_wire;
49
50 end
51 CORE core1 (
52     .clk(clk),
53     .data_DM_core(data_from_DM_to_cpu[15:0]),
54     .data_IM_core(data_from_IM_to_cpu[15:0]),
55     .proId(proId0),
56     .data_core_IM(data_from_cpu_to_IM[15:0]),

```

```

52     .data_core_DM(data_from_cpu_to_DM[15:0]),
53     .DM_en(en_from_cpu_to_DM[0:0]),
54     .IM_en(en_from_cpu_to_IM[0:0]),
55     .AR_out(data_from_AR_to_M[15:0]),
56     .finish(finish_wire[0:0]),
57     .bus(bus_wire_core1),
58     .current_micro_instruction(current_micro_instruction_wire_core1)
59 );
60 CORE core2 (
61     .clk(clk),
62     .data_DM_core(data_from_DM_to_cpu[31:16]),
63     .data_IM_core(data_from_IM_to_cpu[31:16]),
64     .proId(proId1),
65     .data_core_IM(data_from_cpu_to_IM[31:16]),
66     .data_core_DM(data_from_cpu_to_DM[31:16]),
67     .DM_en(en_from_cpu_to_DM[1:1]),
68     .IM_en(en_from_cpu_to_IM[1:1]),
69     .AR_out(data_from_AR_to_M[31:16]),
70     .finish(finish_wire[1:1]),
71     .bus(bus_wire_core2),
72     .current_micro_instruction(current_micro_instruction_wire_core2)
73 );
74 CORE core3 (
75     .clk(clk),
76     .data_DM_core(data_from_DM_to_cpu[47:32]),
77     .data_IM_core(data_from_IM_to_cpu[47:32]),
78     .proId(16'd2),
79     .data_core_IM(data_from_cpu_to_IM[47:32]),
80     .data_core_DM(data_from_cpu_to_DM[47:32]),
81     .DM_en(en_from_cpu_to_DM[2:2]),
82     .IM_en(en_from_cpu_to_IM[2:2]),
83     .AR_out(data_from_AR_to_M[47:32]),
84     .finish(finish_wire[2:2]),
85     .bus(bus_wire_core3),
86     .current_micro_instruction(current_micro_instruction_wire_core3)
87 );
88 CORE core4 (
89     .clk(clk),
90     .data_DM_core(data_from_DM_to_cpu[63:48]),
91     .data_IM_core(data_from_IM_to_cpu[63:48]),
92     .proId(16'd3),
93     .data_core_IM(data_from_cpu_to_IM[63:48]),
94     .data_core_DM(data_from_cpu_to_DM[63:48]),
95     .DM_en(en_from_cpu_to_DM[3:3]),
96     .IM_en(en_from_cpu_to_IM[3:3]),
97     .AR_out(data_from_AR_to_M[63:48]),
98     .finish(finish_wire[3:3]),
99     .bus(bus_wire_core4),
100    .current_micro_instruction(current_micro_instruction_wire_core4)
101 );
102 IRAM IRAM(
103     .clk(clk),
104     .write(en_from_cpu_to_IM),
105     .address(data_from_AR_to_M),

```

```
106         .instr_in(data_from_cpu_to_IM),
107         .instr_out(data_from_IM_to_cpu)
108     );
109     DRAM DRAM (
110         .clk(clk),
111         .write(en_from_cpu_to_DM),
112         .address(data_from_AR_to_M),
113         .data_in(data_from_cpu_to_DM),
114         .data_out(data_from_DM_to_cpu)
115     );
116 endmodule
```

7.15 compiler.py

```
1 import json
2
3 f = open("code.txt", "r")
4 r = open("result.txt", "w")
5 dictionary = json.load(open("dictionary.json"))
6
7 ToMEM = ""
8 count = 0
9 for line in f:
10     code = line.strip().split()
11     if(line.strip() == ""):
12         continue
13     ToMEM += "//Comment:- " + line.strip() + "\n"
14     for op in code:
15         if(op[0] == ":"):
16             ToMEM += "//" + op[1:] + "\n"
17             continue
18         if(op in dictionary):
19             op = dictionary[op]
20
21         if(op.strip().isnumeric()):
22             ToMEM += "MEM[{count}] = 16'd{op};\n".format(count = count, op =
op)
23         else:
24             ToMEM += "MEM[{count}] = {op};\n".format(count = count, op =
op.upper())
25         count +=1
26     ToMEM += "\n"
27 print(ToMEM)
28
29 r.write(ToMEM)
```

7.16 matrix.py

```

1 def split_list(data):
2     divide_index = data.index("")
3     return data[:divide_index], data[divide_index+1:]
4
5
6 file = open("matrix.txt", "r")
7 data = file.read()
8 file.close()
9 data = data.split("\n")
10 matA, matB = split_list(data)
11 output = []
12 roundsA = 0
13 roundsB = 0
14 i = len(matA)
15 j = 0
16 k = 0
17
18
19 for index, x in enumerate(matA):
20     matA[index] = list(map(int, matA[index].strip().split()))
21     j = len(matA[index])
22     roundsA += len(matA[index])
23 for index, x in enumerate(matB):
24     matB[index] = list(map(int, matB[index].strip().split()))
25     k = len(matB[index])
26     roundsB += len(matB[index])
27 if(j == len(matB)):
28     print("Matrices can be multiplied")
29     rounds = roundsB+roundsA+7
30     output.append("MEM[0]=16'd5; //mat_a_base")
31     output.append("MEM[1]=16'd{}; //mat_b_base".format(7+roundsA))
32     output.append("MEM[2]=16'd{}; //mat_c_base
33 \n".format(7+roundsA+roundsB))
34     output.append("MEM[3]=16'd{}; //i_ref".format(i))
35     output.append("MEM[4]=16'd{}; //j_ref \n".format(j))
36     count = 5
37     for i in matA:
38         output.append("//{}".format(i))
39         for j_item in i:
40             output.append("MEM[{}]=16'd{};".format(count, j_item))
41             count += 1
42     output.append("\nMEM[{}]=16'd{}; //j_ref".format(count, j))
43     output.append("MEM[{}]=16'd{}; //k_ref\n".format(count+1, k))
44     count += 2
45     for i in matB:
46         output.append("//{}".format(i))
47         for j_item in i:
48             output.append("MEM[{}]=16'd{};".format(count, j_item))
49             count += 1
50
51 file = open("output.txt", "w")
52 result = [[sum(a*b for a, b in zip(X_row, Y_col))
53            for Y_col in zip(*matB)] for X_row in matA]
54 output.append("\n//results should be..")

```



```
54     for item in result:
55         output.append("//{}".format(item))
56     for element in output:
57         file.write(element + "\n")
58     file.close()
59 else:
60     print("Matrices can't be multiplied")
```