

UNIVERSITY OF MORATUWA
FACULTY OF ENGINEERING



DESIGN AND IMPLEMENTATION OF MULTI CORE PROCESSOR FOR MATRIX MULTIPLICATION

EN3030 : CIRCUITS AND SYSTEMS DESIGN

PROJECT REPORT

DATE OF SUBMISSION: JULY 7, 2021

NAME	Index Number
Nimashi K. H.	170407U
Nusha M. N. F.	170415R
Palihakkara A. T.	170418E
De Silva K. D. M. K.	170106V

DEPARTMENT OF ELECTRONICS & TELECOMMUNICATIONS ENGINEERING

Abstract

Design and Simulation of single core and multi core processors for matrix multiplication and a comparison of their performance.

This is a report of a project that uses the hardware description language Verilog to implement an instruction set architecture (ISA) designed by our team for the purpose of matrix multiplication. This system was implemented using Intel's Quartus Prime Lite and Altera Modelsim was used for the simulation of the implementation on an FPGA.

The project consists of three phases: ISA design, single core simulation and multi-core simulation. ISA design includes the design and optimization of the instruction set, micro instruction and datapath of the elements. Simulation of the single core processor for matrix multiplication was then done, after which the implementation for multi-core processor was done using the same ISA and similar modules as single-core, but with some modifications. The multi-core processor uses 8 cores, but less than this can be used by taking an input of the number of cores. The system was simulated and the functionality and execution time was analyzed for various test cases.

This report details the design, algorithm and logic and describes the components and modules used in building the processor. The report also compares the processing speed of our processors with the number of cores used.

Contents

List of Figures	4
List of Tables	4
1 Introduction	6
2 Instruction Set Architecture	7
2.1 Registers and Datapath	7
2.2 The Instruction Set	9
2.2.1 JPNZ	9
2.2.2 COPY	10
2.2.3 LOAD	10
2.2.4 STORE	10
2.2.5 ASSIGN	10
2.2.6 MOVE, SET & GET	11
2.2.7 RESET, INC	11
2.2.8 ADD, MUL	11
2.2.9 NOOP, CHKIDLE	12
2.3 ISA : Summary	12
3 Algorithm Design	14
3.1 Storing the Matrix in the DRAM	14
3.2 Matrix Multiplication	15
3.3 Matrix Multiplication using Multiple Cores	15
3.4 The Algorithm for Matrix Multiplication	15
4 Architecture	17
4.1 State Diagram & Micro Instructions	17
4.2 Processor - Single Core	21
4.3 Processor - Multi Core	21
4.4 System Components	23
4.4.1 Core	23
4.4.2 Instruction Memory	23
4.4.3 IROM Controller	24
4.4.4 Data Memory	25
4.4.5 DRAM Controller	26
4.5 Core Components	26
4.5.1 Control Unit	26
4.5.2 ALU	28
4.6 Muxes & Comparators	29
4.6.1 Data Bus	29
4.6.2 Comparator & its Muxes	29
4.7 Registers	31
4.7.1 Program Counter (PC)	31
4.7.2 Address Register (AR)	31
4.7.3 Read-Write Registers	32
4.7.4 Write-Reset Registers	33
4.7.5 Write-Increment Registers	33
4.7.6 Write-Reset-Increment Registers	34
5 Timing Diagrams of Instructions	35
5.1 FETCH	35
5.2 JUMP	36
5.2.1 zFlag = 0	36
5.2.2 zFlag = 1	36
5.3 COPY	37
5.3.1 Fetching Copy Address	37
5.3.2 Address assigning for each core	37

5.3.3 Fetching from Data Memory	38
5.4 LOAD	38
5.5 STORE	39
5.6 ASSIGN	39
5.7 RESET	40
5.8 MOVE	40
5.9 SET	40
5.10 MUL	41
5.11 ADD	41
5.12 INC	42
5.13 CHK_IDLE	42
5.14 GET	42
6 Results	43
7 Limitations of the system	46
8 Compiler - Python Program	47
9 References	48
10 Appendix	49
10.1 Processor	49
10.2 Core	51
10.3 Data Memory Controller	53
10.4 Data Memory	75
10.5 Instruction Memory Controller	76
10.6 Instruction Memory	82
10.7 Control Unit	82
10.8 Arithmetic and Logic Unit	102
10.9 Bus Mux	103
10.10Comparator	104
10.11Program Counter(PC)	104
10.12Address Register (AR)	104
10.13Read-Write Register	105
10.14Write-Reset Register	105
10.15Write-Increment Register	105
10.16Write-Reset-Increment Register	106

List of Figures

2.1	Data Path of the Core	8
2.2	Instruction Types	9
2.3	JPNZ Flowchart	9
2.4	Check_Idle Flow Chart	12
3.1	Storing the Matrix in DRAM	14
3.2	Multiple cores accessing the First Matrix in Dram	15
4.1	Control Unit State Diagram	17
4.2	Single Core Processor	21
4.3	Core Module	23
4.4	Instruction Memory Module	23
4.5	Instruction Memory Controller	24
4.6	Data Memory Module	25
4.7	Memory Allocations for the cores	25
4.8	Control Unit	26
4.9	ALU Module	28
4.10	ALU & AC register	28
4.11	Data-bus Multiplexer Module	29
4.12	Comparator and Comp-Muxes Block Diagram	29
4.13	Comparator Bus Mux	30
4.14	Comparator	30
4.15	Program Counter PC Register	31
4.16	Address Register AR	31
4.17	Read-Write Register	32
4.18	Reset-Write Register	33
4.19	Write-Increment Register	33
4.20	Reset-Increment Register	34
5.1	Clock cycles of FETCH	35
5.2	Clock cycles of JUMP (z = 0)	36
5.3	Clock cycles of JUMP (z = 1)	36
5.4	Clock cycles of COPY (fetching Copy Address)	37
5.5	Clock cycles of COPY 3A (Increment Addresses)	37
5.6	Clock cycles of storing data in registers	38
5.7	Clock cycles of LOAD	38
5.8	Clock cycles of STORE	39
5.9	Clock cycles of ASSIGN	39
5.10	Clock cycles of RESET	40
5.11	Clock cycles of MOVE	40
5.12	Clock cycles of SET	40
5.13	Clock cycles of MUL	41
5.14	Clock cycles of ADD	41
5.15	Clock cycles of INC	42
5.16	Clock cycles of CHK_IDLE	42
5.17	Clock cycles of GET	42
6.1	Result Comparison	43
6.2	DRAM output for 8-cores processing	44
6.3	DRAM output for 4-cores processing	44
6.4	Matrix Multiplication Time Comparison	45
8.1	Input and the Output of the Python Programme	47

List of Tables

1.1	ISA Design	6
1.2	Processor Design	6
2.1	Registers : Description & Types	7
4.2	Components of Single Core Processor	21
4.3	Components of Multi Core Processor	21
4.4	Multi-core Processor - Core Signals	23
4.5	IROM Controller Control Signals	24
4.6	Control signals	27
6.1	Test cases	45

1 Introduction

Processor design involves making decisions on the trade-off between various factors such as the size of the Instruction Set and complexity of the Processor. The REMM (Row-based Eight-core Matrix Multiplier) processor designed and simulated in this project is optimized for matrix multiplication.

Matrix Multiplication is one of the basic computing processes for various complex algorithms. The advent of various vector based processor systems shows this importance in modern computing. The REMM processor is an implementation of a multi-core matrix multiplying processor which has been designed with 15 instructions and a elegant and smooth functionality for basic matrix multiplication.

REMM works with an 8-bit wide 256 deep Instruction memory, an 8-bit wide 256 deep data memory, a data memory controller, an instruction memory controller and 8 processing units, called cores, each of which read and execute the same program instructions.

For the multi-core processor, two special modules, DRAM controller and IROM controller are included. They are setup to give synchronous data memory and instruction memory allocation to each active core. They handle the conflict created by multiple memory access by sending memory-available signals as control signals to each core.

Data memory is a commonly accessible module which is can be read by all the cores. For writing into the memory, the DRAM is divided and allocated for each of the 8 cores separately. Memory utilization is optimized considering the multi core processing at the given execution time.

The number of cores used for a given execution of the REMM can be changed by the python compiler and a comparison can be done as to how multi core processing enhances the performance and efficiency of the processing a matrix multiplication.

The corresponding compiler program which scans the text file where the algorithm is codified using human-readable language and then translated in to hexadecimal values and sent to the instruction memory.

The system was coded in Verilog HDL using Intel Quartus II Prime Lite and simulated using Altera Modelsim software.

Additionally, since the width and depth of the data memory are parameters in the core module, along with the register width, data memory width and depth, these can be modified to multiply large-scale matrices.

Table 1.1: ISA Design

No. of Instructions	:	15
No. of Micro instructions	:	59
Avg. of clock cycles/instruction (including FETCH cycle	:	5.2
No. of registers	:	18

Table 1.2: Processor Design

Data Memory	:	8×256
Instruction Memory	:	8×256
No. of cores	:	8

2 Instruction Set Architecture

Designing of the Instruction Set Architecture involves determining the necessary instructions and instruction types, the number and type of registers and the overall data path of how the different modules are connected.

2.1 Registers and Datapath

Register	Size	Type	Name	Description
PC	8	PC	Program Counter	Connected to the IROM address bar. Contains the address of the next instruction to be read from Instruction Memory.
IR	8	W	Instruction Register	Connected to the data output of the instruction memory. Receives instruction and sends it to the control unit.
AR	8	AR	Address Register	Connected to the DRAM address bar. Contains the address of the memory element to be read from Data Memory. Also connected to data bus and can be read and written.
DR	8	W	Data Register	Connected to the DRAM data bar. Anything to be written to or being read from DRAM will be stored here.
RR	8	W	Row Register	Connected within the core registers. Used to store data about the matrices. Write only happens once and store the value.
M1	8	W	Register M1	
K1	8	W	Register K1	
N1	8	W	Register N1	
M2	8	RI	Register M2	Connected within the core registers. Used to store data about the matrices. Data is written and manipulated.
K2	8	RI	Register K2	
N2	8	RI	Register N2	
T4	8	RW	Register T4	Used to store the calculated address information for DRAM access. It is used to write only once and store the value.
C1	8	W	Register C1	Connected within the core, used to store addresses for DRAM access. These registers functioning as address pointers on a higher level.
C2	8	WI	Register C2	
C3	8	WI	Register C3	
RP	8	W	Multiplier Register	Used to store the result of ALU operations
RT	8	WR	Temporary Register	
AC	8	WR	Accumulator	Used to store the most recent calculated value of the ALU.

¹ W - read/write registers

² WR - write/reset registers

³ WI - write/increment registers

⁴ WRI - write/reset/increment registers

⁵ AR & PC are unique types

Table 2.1: Registers : Description & Types

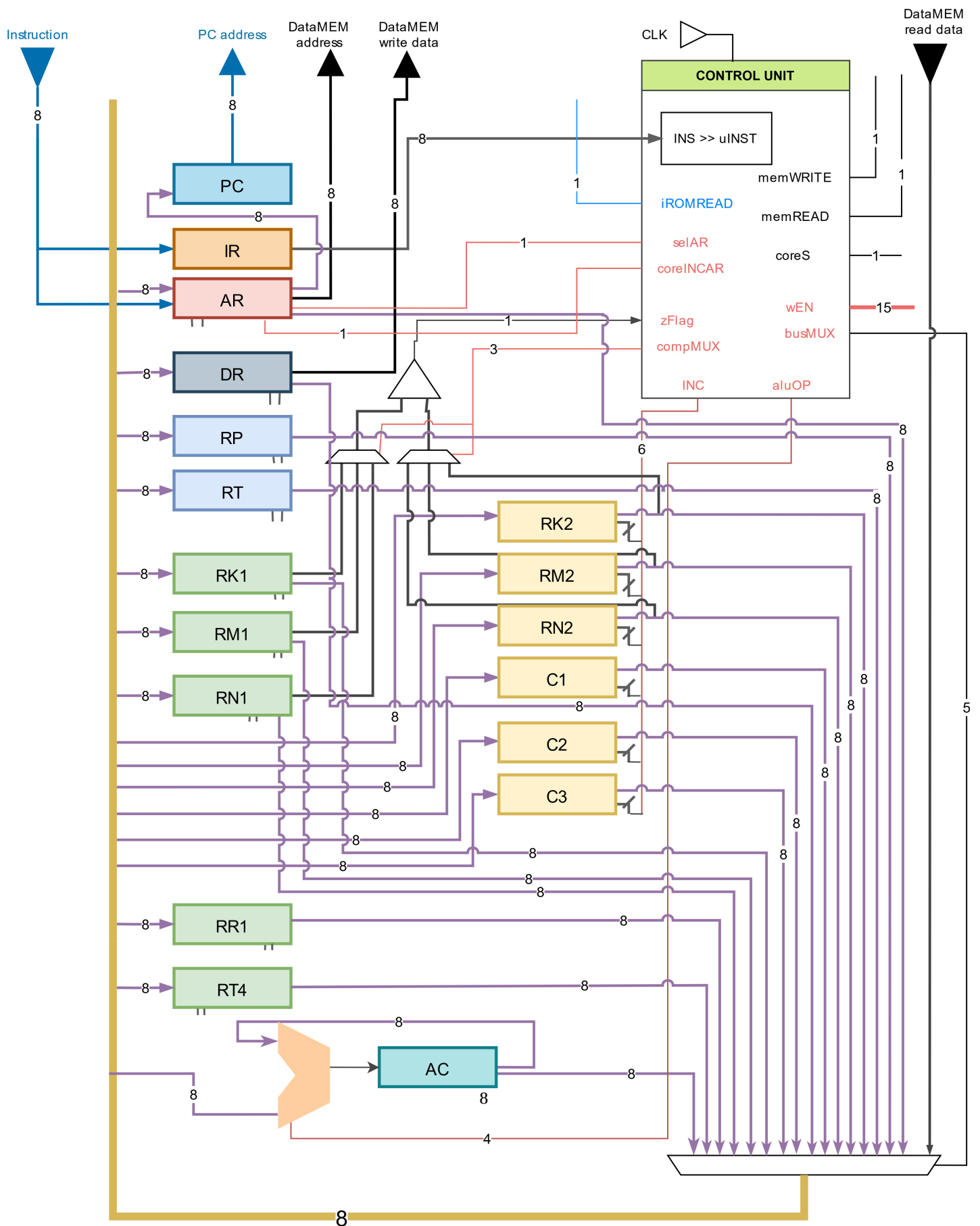


Figure 2.1: Data Path of the Core

2.2 The Instruction Set

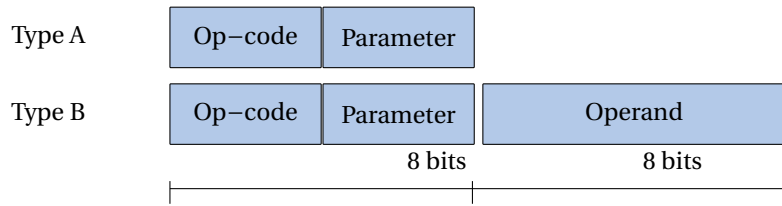


Figure 2.2: Instruction Types

The types of instructions used in this processor can be categorized into two categories:

- Type A Consists of 4 bits of Op-code & 4 bits of Parameter.
- Type B Consists of 4 bits of Op-code & 4 bits of Parameter followed by 8-bits of Operand which contains a memory address.

The instructions require a **fetch** cycle and an **execute** cycle to run. Each fetch cycle, as will be displayed in detail in Figure ??, requires 3 clock cycles in addition to the number of clock cycles taken by the instruction execution, our design maximizes the action that is done during execution, while at the same time providing enough flexibility to modify and have high-level clarity in the ISA.

2.2.1 JPNZ

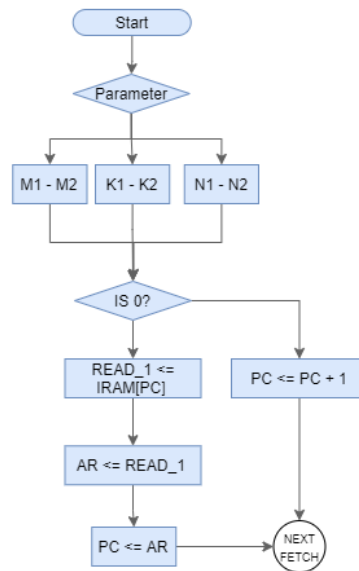


Figure 2.3: JPNZ Flowchart

JPNZ instruction stands for "Jump if Not Zero" and it uses the input of the zero flag to check whether a Jump should be carried out to continue the loop.

There are three parameters for the JPNZ instruction and each of these parameters will start a comparison of two different registers as shown. This instruction is used to smoothly continue in the two events that a given loop when doing matrix multiplication has to be repeated, or to continue to the next instruction.

It can be entered as a 16 bit instruction in the instruction memory, spanning two 8 bit storage elements. The first 8 bits contain the complete instruction and the next 8 bits contain the address of the next address that the program counter (PC) has to point to i.e. the next instruction that will be fetched.

```

Jump if:
    reg M1 - reg M2 != 0;
    reg K1 - reg K2 != 0;
    reg N1 - reg N2 != 0;
Jump to:
    LOOP1;
    LOOP2;
    LOOP2;

```

2.2.2 COPY

COPY instruction is an 8-bit instruction which copies from the DR register to the relevant other register.

This instruction has five parameters which are used to copy to five different registers. These registers are used in this customized processor to store the size of the matrices to be multiplied and other information about the matrices for easy manipulation of the matrix, such as jumping to the next row or column.

```

COPY of:
    reg DR;
COPY to:
    reg M1 <= M;
    reg K1 <= K;
    reg N1 <= N;
    reg R1 <= R;
    reg T1 <= T;

```

2.2.3 LOAD

LOAD instruction is used to read the elements from the DRAM. In this customized processor, this instruction carried out with the use of two pointer registers while the actual register pointing to the address bar of the DRAM is the Address Register AR. LOAD therefore has two parameters.

```

LOAD:
    reg AR <= reg C1;
    reg AR <= reg C2;

```

2.2.4 STORE

STORE is the instruction used to store elements in the DRAM. Elements in the register RT are stored, by first sending them to Data Register DR which is used to connect to the DRAM. his instruction has only one parameter that is necessary for this matrix multiplication customized processor.

2.2.5 ASSIGN

The ASSIGN instruction was defined to assign the addresses to the pointer registers C1 and C2. This instruction has two parameters and it is a Type B instruction which means that the following word contains the address that has to be assigned.

```

ASSIGN to:
    reg C1 <= address[element of matrix 1] ;
    reg C2 <= address[element of matrix 2] ;

```

2.2.6 MOVE, SET & GET

These instructions are used for register-to-register type data transfer. MOVE instruction is used to move what is in the ALU's register AC to the relevant register. It has four parameters which determine where the data is moved to.

```
MOVE to:
    reg RP;
    reg RT;
    reg C1;
    reg C3;
MOVE from:
    reg AC;
```

SET instruction is used to set AC value as whatever is in a given register. SET is used to read an output from an ALU operation and it has 3 parameters for this processor implementation.

```
SET to:
    reg AC;
SET from:
    reg C1;
    reg DR;
    reg K1;
```

GET is an instruction that is used to get a register value from another register: here this is specifically used to get the address T4 stored in Reg T4 to the Reg C1 pointer register. This is used for the core to return to the beginning of the matrix as an alternative to spending several clock cycles on calculating this value or fetching it from IROM.

```
GET to:
    reg C1;
GET from:
    reg T4;
```

2.2.7 RESET, INC

RESET and INC are instructions that are specific to Reset/Increment, Reset/Write and Increment/Write type registers as the register name indicates. Both of these instructions are Type A instructions with no operand. RESET enabled registers include five registers, and there are four parameters which reset them:

```
RESET all:
    reg AC;
    reg RT;
    reg RM2;
    reg RN2;
    reg RK2;
RESET:
    reg RT;
    reg RK2;
    reg RN2;
```

INC enabled registers include four registers, each of which can be incremented individually. These registers are either used to act as counters for smooth transition to the next loop during the matrix multiplication, or they are pointers to increment the DRAM address pointer.

```
INC:
    reg M2;
    reg K2;
    reg N2;
    reg C2;
    reg C3;
```

2.2.8 ADD, MUL

There are two ALU operations used in this microprocessor for matrix multiplication. They are addition and multiplication. Considering the application-specific nature of the processor, an accumulative operation for both addition and multiplication was found to be the best option.

ADD has four parameters and there are four registers which can be used to make an accumulative addition.

```

ADD t0:
    reg AC;
ADD:
    reg RT;
    reg M1;
    reg M2;

```

MUL is used for accumulative multiplication. Each element is multiplied and stored in register RP. Reg RP acts as a temporary register to hold the value that is about to be multiplied.

```

MUL t0:
    reg AC;
MUL:
    reg RP;

```

2.2.9 NOOP,CHKIDLE

These instructions are used to directly control the status of the processor.

NOOP or NO Operation is an instruction that is used to pass an empty clock cycle, after which the next instruction is fetched and executed. The processor/core is active while this operation is run and it can be used to control clock delays.

CHKIDLE (from CHeck IDLE) is an instruction that checks whether the core is idle.

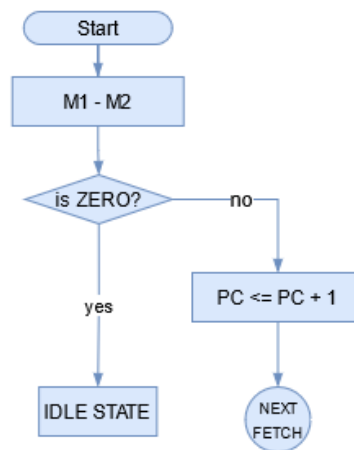


Figure 2.4: Check_Idle Flow Chart

When this instruction is passed, the flag CHKIDLE will be compared, and if the core is active, the rest of the instructions will be passed. If this returns an IDLE status, then the core is moved to END Operation mode and passes an inactive signal for the rest of the algorithm.

2.3 ISA : Summary

A summary of the instructions, function, parameters and the number of clock cycles taken per instruction (without the fetch cycle) is given below. (Each instruction takes three clock cycles to run the fetch cycle.)

Summary of the Instruction Set Architecture

	OPCODE	Parameters	OPCODE	Example	Description	CLK
	0000		NOOP	NOOP	No operation	1
	1101		CHECK_IDLE	CHK_IDLE	Make processor idle	2
Branching and Looping	0001	0- loopM 1- loopK 2- loopN	JUMP	JUMPM	Jump to instruction address Parameter: Next executing loop	5
Memory	0010	0- M1 M3 1- K1 2- N1 3- RR 4- RT	COPY	COPY [M1]	Copy values from DRAM to a read write register Parameters: Register names or allocated memory initiate number	7
	0011	0- C1 1- C2	LOAD	LOAD [C1]	Load values from DRAM to AC Parameters: Memory location addresss	4
	0100		STORE	STORE	Store vauess in DRAM	4
	0101	0- C1 1- C2	ASSIGN	ASSIGN [C1]	Assign address from DRAM to pointers Parameters: Pointer names	3
General	0110	0- ALL 1- N2 2- K2 3- RT	RESET	RESET [ALL]	Reset one or many registers Parameters: Register names	1
	0111	0- RP 1- RT 2- C1 3- C3	MOVE	MOVE [RP]	Move values from AC to a read write register or a pointer Parameters: Register or pointer names	1
	1110		GET	GET	Move value from RT to C3 pointer	1
	1000	0- C1 1- DR 2- K1	SET	SET [C1]	Move values from a register to AC Parameters: Register names	1
Arithmetic	1001		MUL	MUL	Multiply AC by given value	1
	1010	0- RT 1- M1 2- M2 3- MEM	ADD	ADD [RT]	Add given register to AC, Parameters are register names Parameters: Register names	1
	1011	0- C2 1- C3 2- M2 3- K2 4- N2	INC	INC [C2]	Increment increment-type registers and pointers by one Parameters: GP and pointers names	1

3 Algorithm Design

Since the REMM processor is built customized to handle simple matrix multiplication, the program algorithm is an important part of how the processor was designed. We selected the most optimal definitions and commands in order to get an accurate algorithm for matrix multiplication that will consume the minimum possible number of clock cycles, require less instructions, and make sense when read by humans.

Entering the matrices into the DRAM is vital to how the algorithm will run. In order for an efficient matrix multiplication at the low-level machine language, some information about the matrices being multiplied is always necessary. This includes the number of rows and columns of each of the two matrices being multiplied.

We define two matrices A and B, which will be multiplied in the following manner:

$$C_{M \times K} = A_{M \times N} \times B_{N \times K}$$

3.1 Storing the Matrix in the DRAM

When storing the matrices, the values of each of the two matrices are written column-wise into the DRAM. This can be described as writing the transposed matrix row-wise into the DRAM. This approach was selected for our design for multiple reasons:

1. To simplify the algorithm by being able to access the **second matrix's next element** and to locate the **next row of the first matrix** with a simple pointer increment. This avoids the necessity of making some unnecessary calculations to find the next element.
2. To reduce the complexity of the ISA that would have been necessary to carry out those calculations.
3. It also has the effect of increasing the efficiency of the program.

The element storage in the DRAM is shown below:

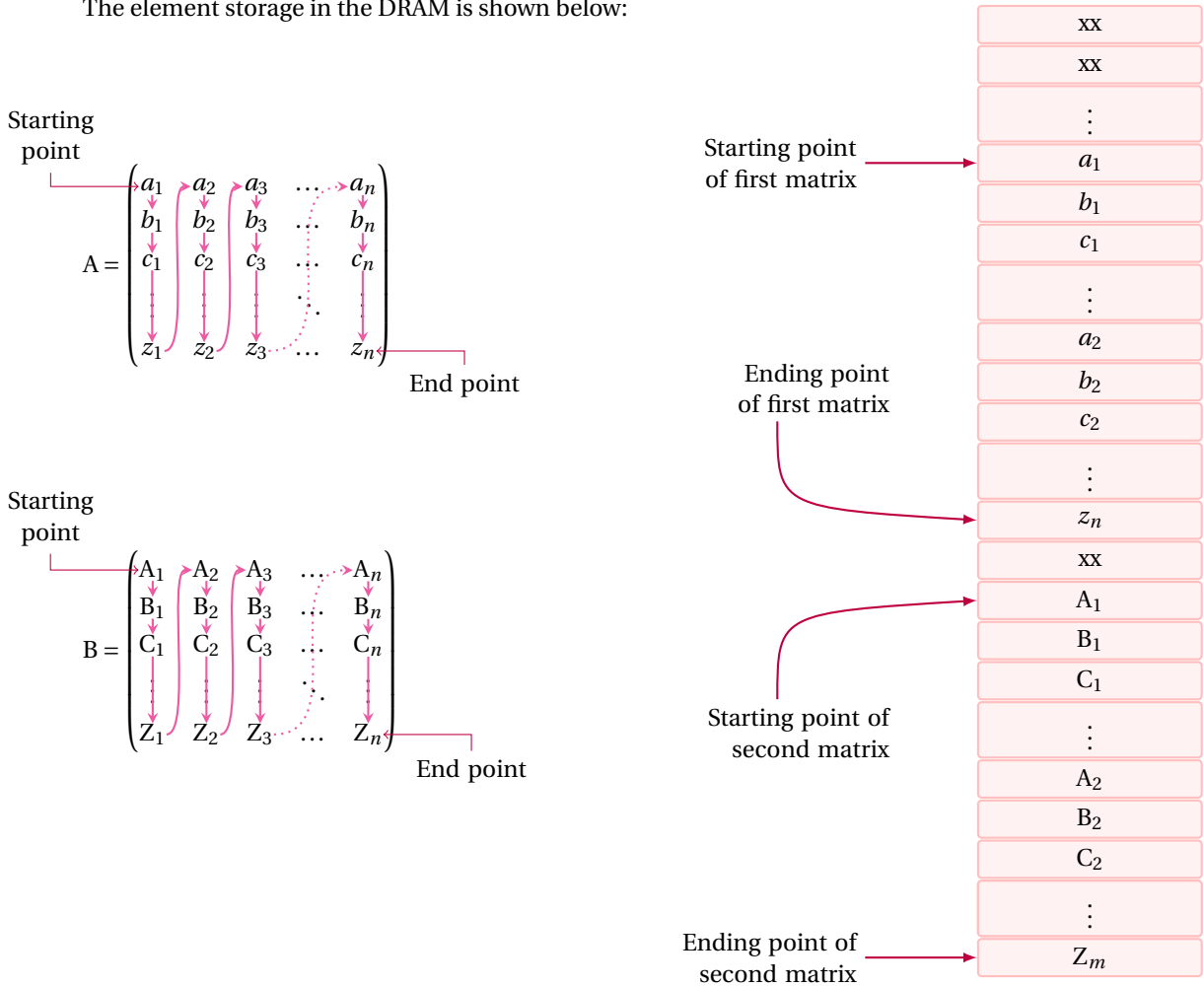


Figure 3.1: Storing the Matrix in DRAM

3.2 Matrix Multiplication

The algorithm consists of three loops which iterate according to the *branching* and *looping* control signal output by the control unit to load the corresponding elements of each of the two matrices and process the matrix multiplication, which consists of multiplying individual elements and adding them to get each resultant matrix element. The output matrix is stored row-wise.

3.3 Matrix Multiplication using Multiple Cores

The number of cores being used for execution can be changed by giving a raw input for "number of cores = " after which it will be processed by the compiler. The total number of cores in the REMM processor is eight, and up to eight cores can be selected.

In the multiple-core execution of the program, each core is allocated a number of rows so that the number of rows is most evenly distributed. This distribution is done in the compiler and stored in the Memory, to be available for each core to access. The distribution is done as shown below: In this design, each core accesses the relevant row of the first matrix in the DRAM.

```

1 #T1 is the array which holds the location
  address of First matrix first element
2 #no_of_cores is total number of executed cores
3 for i in range (no_of_cores):
4     while (counter >= (no_of_cores-i)):
5         for j in range(no_of_cores-i):
6             T1[j] += 1
7             counter -= 1

```

This simple approach allows each core to carry out the row-by-row execution of matrix multiplication operations in parallel. Accessing the second matrix will be done by all cores. Each core will output the matrix multiplication pertaining to their allocated rows.

It can be possible that some cores are not allocated a matrix. In that case, the core will verify by checking if the value it received for M number of rows is zero, and the core will activate the IDLE status.

If they are not idle, each core will read the same program from the IROM, calculate the results of their assigned row(s) and store the result output in the allocated DRAM block.

Using multiple cores in this manner will allow the time taken to be considerably faster and the algorithm will run smoothly to produce the required total result.

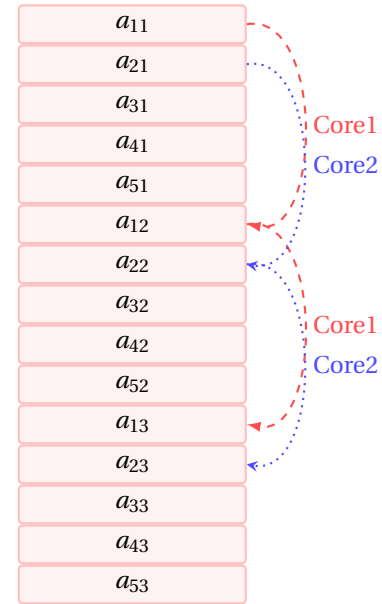


Figure 3.2: Multiple cores accessing the First Matrix in Dram

3.4 The Algorithm for Matrix Multiplication

```

1 COPY [Rm1]
2 T1
3 RESET [ALL]
4 if z==1:
5     CHK_IDLE
6 COPY [Rn1]
7 T2
8 COPY [Rk1]
9 T3
10 COPY [Rr]
11 T7
12 COPY [Rt4]
13 T4
14 ADD_MEM
15 MOVE [C3]

```



```

16 #loop[M] begin:
17     ASSIGN [C2]
18     T5
19     #loop[K] begin:
20         GET_C1
21         RESET [RT]
22         SET [C1]
23         ADD [RM2]
24         MOVE [C1]
25         #loop[N] begin:
26             LOAD [C1]
27             SET [DR]
28             MOVE [Rp]
29             LOAD [C2]
30             SET [DR]
31             MUL [Rp]
32             ADD [Rt]
33             MOVE [Rt]
34             SET [C1]
35             ADD [Rm1]
36             MOVE [C1]
37             INC [C2]
38             INC [Rn2]
39             if (z==0){
40                 JPNZ [N]
41             }else{
42                 NJMP
43             }
44             #loop[N]
45             STORE
46             INC [C3]
47             RESET [Rn2]
48             INC [Rk2]
49             if (z==0){
50                 JPNZ [K]
51             }else{
52                 NJMP
53             }
54             #loop [K]
55             RESET [Rk2]
56             INC [Rm2]
57             if (z==0){
58                 JPNZ [M]
59             }else{
60                 NJMP
61             }
62             #loop[M]
63             END

```

4 Architecture

4.1 State Diagram & Micro Instructions

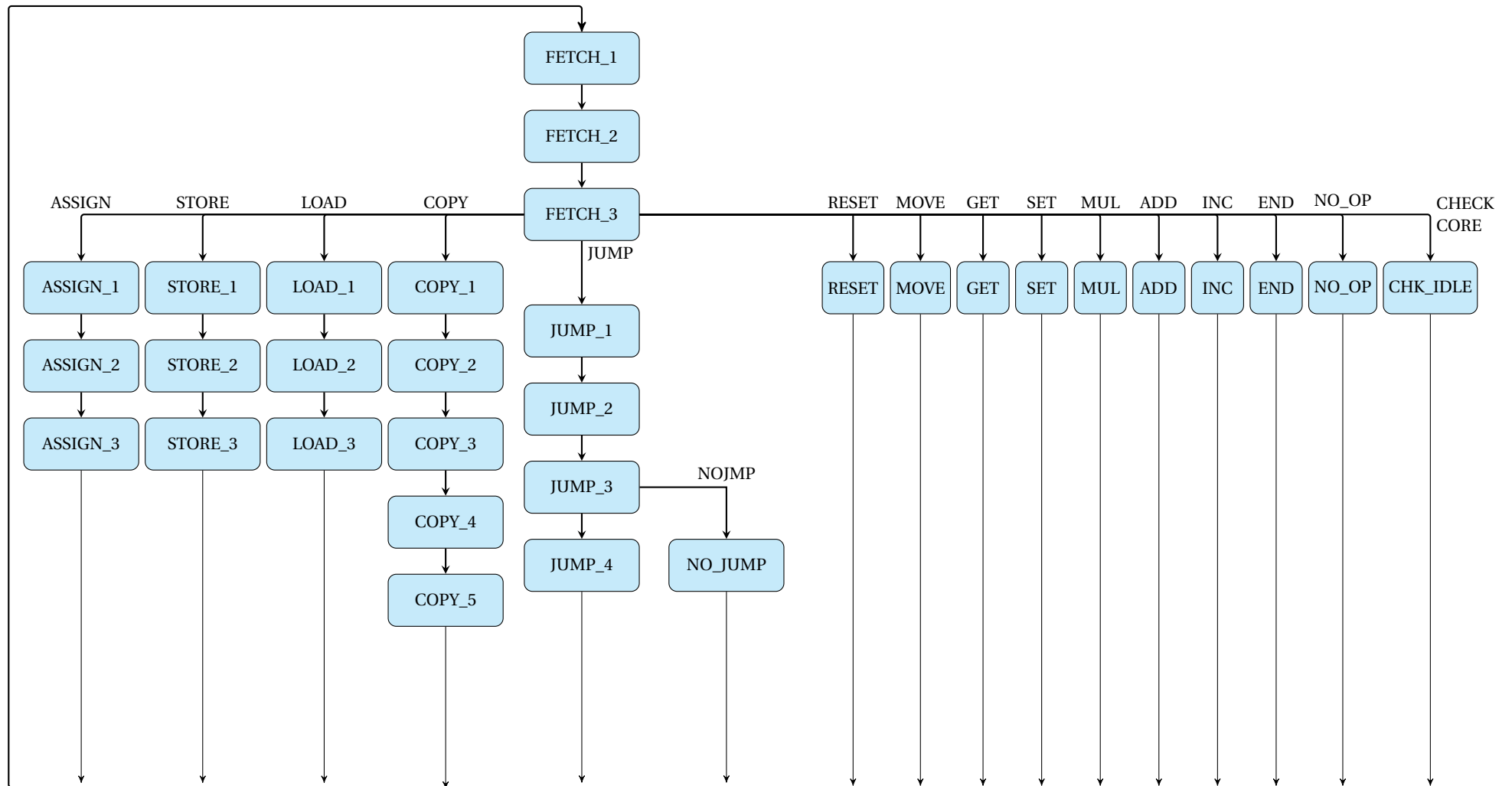


Figure 4.1: Control Unit State Diagram

Table 4.1: Micro Instructions and corresponding states

OPCODE	params	OPCODE	MICRO-INSTRUCTIONS	STEPS	NEXT_STATE	STATE
0000	0000	NOOP	NOOP	NO OPERATION	-	8'h00
			FETCH_1	PC ; READ_iROM	if(imemAV):NEXT_STATE<=FETCH_2 else: NEXT_STATE <=FETCH_1	8'h10
			FETCH_2	IR <= iROM; PC <= PC+1	NEXT_STATE <=FETCH_3	8'h11
			FETCH_3	-	NEXT_STATE <=INS[3:0]	8'h12
0001	0000	JPNZM	JPNZM_1	M1-M2	NEXT_STATE <=ZJMP_1	8'h20
0001	0001	JPNZK	JPNZK_1	K1-K2	NEXT_STATE <=ZJMP_1	8'h21
0001	0010	JPNZN	JPNZN_1	N1-N2	NEXT_STATE <=ZJMP_1	8'h22
0001	xxxx	JUMP	ZJMP_1	z RECEIVED	if (zFlag == 1 && jmpMFlag == 1): NEXT_STATE <= ENDOP if (zFlag == 1 && jmpMFlag == 0): NEXT_STATE <= NJMP_1 else: NEXT_STATE <= JPNZ_2	
0001	xxxx		JPNZ_2	READ_1 <= IRAM[PC]	NEXT_STATE <=JMP_3	8'h23
0001	xxxx		JPNZ_3	AR <= READ_1	NEXT_STATE <=JMP_4	8'h24
0001	xxxx		JPNZ_4	PC <= AR	NEXT_STATE <=FETCH_1	8'h25
0001	xxxx	NJMP	NJUMP	PC < PC+1	NEXT_STATE <=FETCH_1	8'h30
0010	xxxx	COPY	COPY_1	IRAM[PC], READ_1	NEXT_STATE <=COPY_2	8'h40
0010	xxxx		COPY_2	AR <= READ, PC <= PC + 1	if (INS[3:0]== COPY_M INS[3:0]== 'COPY_T): NEXT_STATE <= COPYM_3A else: NEXT_STATE <= COPY_3	8'h41
0010	xxxx		COPY_3	DRAM[AR] [READ_D]	if(memAV): NEXT_STATE <= COPY_4 else: NEXT_STATE <= HOLD_1	8'h42
0010	xxxx		COPY_4	DR <= READ_D	if (INS[3:0]== COPY_M):NEXT_STATE <= COPYM_5; if (INS[3:0]== COPY_K):NEXT_STATE <= COPYK_5; if (INS[3:0]== COPY_N):NEXT_STATE <= COPYN_5; if (INS[3:0]== COPY_R):NEXT_STATE <= COPYR_5; if (INS[3:0]== COPY_T):NEXT_STATE <= COPYT4_5;	8'h43

0010	0000	COPYM1	COPYM1_5	M1 <= DR	NEXT_STATE <= FETCH_1	8'h44
0010	0001	COPYK1	COPYK1_5	K1 <= DR	NEXT_STATE <= FETCH_1	8'h45
0010	0010	COPYN1	COPYN1_5	N1 <= DR	NEXT_STATE <= FETCH_1	8'h46
0010	0011	COPYRr1	COPYRR1_5	RR1 <= DR	NEXT_STATE <= FETCH_1	8'h48
0010	0100	COPYRt1	COPYRT1_5	RT1 <= DR	NEXT_STATE <= FETCH_1	8'h49
0010	0000	COPYM3	COPYM_3A	AR <= AR + CoreID	NEXT_STATE <= COPY_3	8'h47
0011	0000	LOADC1	LOADC1	AR <= C1	NEXT_STATE <= LOAD_2	8'h52
0011	0001	LOADC2	LOADC2	AR <= C2	NEXT_STATE <= LOAD_2	8'h53
0011	xxxx	LOAD	LOAD_2	READ_D <= DRAM[AR]	if(memAV): NEXT_STATE <= LOAD_3 else: NEXT_STATE <= HOLD_1	8'h50
0011	xxxx		LOAD_3	DR <= READ_D	NEXT_STATE <= FETCH_1	8'h51
0100	xxxx	STORE	STORE_1	DR <= RT	NEXT_STATE <= STORE_2	8'h60
0100	xxxx		STORE_2	AR <= C3	NEXT_STATE <= STORE_3	8'h61
0100	xxxx		STORE_3	DRAM[AR] <= DR	memWRITE BEGIN if (memAVREG): NEXT_STATE <= FETCH_1 else: NEXT_STATE <= HOLD_1	8'h62
0101	xxxx	ASSIGN	ASSIGN_1	IRAM[PC], READ_1	NEXT_STATE <= ASSIGN_1	8'h70
0101	xxxx		ASSIGN_2	AR <= READ, PC <= PC + 1	if (INS[3:0]== 'ASSIGN_C1): NEXT_STATE <= ASSIGNC1_3A else if (INS[3:0]== 'ASSIGN_C2): NEXT_STATE <= ASSIGNC2_3	8'h71
0101	0000	ASSIGNC1	ASSIGNC1_3A	AR <= AR + CoreID	NEXT_STATE <= ASSIGNC1_3	8'h75
	0001		ASSIGNC1_3	C1 <= AR	NEXT_STATE <= FETCH_1	8'h72
0101	0010	ASSIGNC2	ASSIGNC2_3	C2 <= AR	NEXT_STATE <= FETCH_1	8'h74
0110	0000	RESET	RESETALL	M2, N2, K2, Rt, AC <= 0	NEXT_STATE <= FETCH_1	8'h80
0110	0001		RESETN2	N2 <= 0	NEXT_STATE <= FETCH_1	8'h81
0110	0010		RESETK2	K2 <= 0	NEXT_STATE <= FETCH_1	8'h82
0110	0011		RESETRT	Rt<=0	NEXT_STATE <= FETCH_1	8'h83

0111	0000	MOVE	MOVERP	RP <= AC	NEXT_STATE <= FETCH_1	8'h90
0111	0001		MOVERT	RT <= AC	NEXT_STATE <= FETCH_1	8'h91
0111	0010		MOVEC1	C1 <= AC	NEXT_STATE <= FETCH_1	8'h92
0111	0011		MOVEC3	C3 <= AC	NEXT_STATE <= FETCH_1	8'h93
1101	xxxx	GET	GET_C1	C1 <= RT4	NEXT_STATE <= FETCH_1	8'hF3
1000	0000	SET	SETC1	AC <= C1	NEXT_STATE <= FETCH_1	8'hA0
1000	0001		SETDR	AC <= DR	NEXT_STATE <= FETCH_1	8'hA1
1000	0010		SET K1	AC <= K1	NEXT_STATE <= FETCH_1	8'hA2
1001	0000	MUL	MUL	AC <= Rp * AC	NEXT_STATE <= FETCH_1	8'hB0
1001	0001		MUL_CORE	AC <= AC * coreID	NEXT_STATE <= FETCH_1	8'hBF
1010	0000	ADD	ADDRt	AC <= RT + AC	NEXT_STATE <= FETCH_1	8'hC0
1010	0001		ADDM1	AC <= AC + M1	NEXT_STATE <= FETCH_1	8'hC1
1010	0010		ADDM2	AC <= AC + M2	NEXT_STATE <= FETCH_1	8'hC2
1010	0011		ADDMEM	AC <= AC + MEM_ID	NEXT_STATE <= FETCH_1	8'hC3
1011	0000	INC	INCC2	C2 <= C2+1	NEXT_STATE <= FETCH_1	8'hD0
1011	0001		INCC3	C3 <= C3+1	NEXT_STATE <= FETCH_1	8'hD1
1011	0010		INCM2	M2<=M2+1	NEXT_STATE <= FETCH_1	8'hD2
1011	0011		INCK2	K2<=K2+1	NEXT_STATE <= FETCH_1	8'hD3
1011	0100		INCN2	N2<=N2+1	NEXT_STATE <= FETCH_1	8'hD4
1101	xxxx	CHECK_IDLE	ENDIF_1	-	(if Zflag==1): NEXT_STATE <= ENDOP_1	8'hF0
1100	xxxx	END	ENDOP_1	-	NEXT_STATE <= ENDOP_1	8'hE0

4.2 Processor - Single Core

The Processor in our system is the module that holds all the other modules of the project.

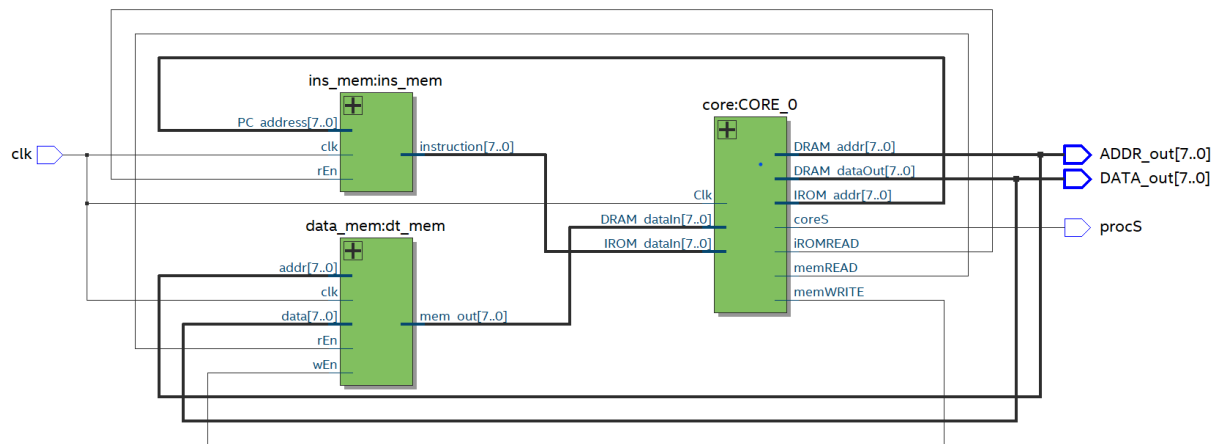


Figure 4.2: Single Core Processor

As shown in the diagram, the single core processor has just three main sub-modules. It takes only a clock as an input.

Table 4.2: Components of Single Core Processor

1-Core	:	Contains the processing and control related components
2-Instruction Memory	:	Contains the instructions
3-Data Memory	:	RAM which data can be read from and written to

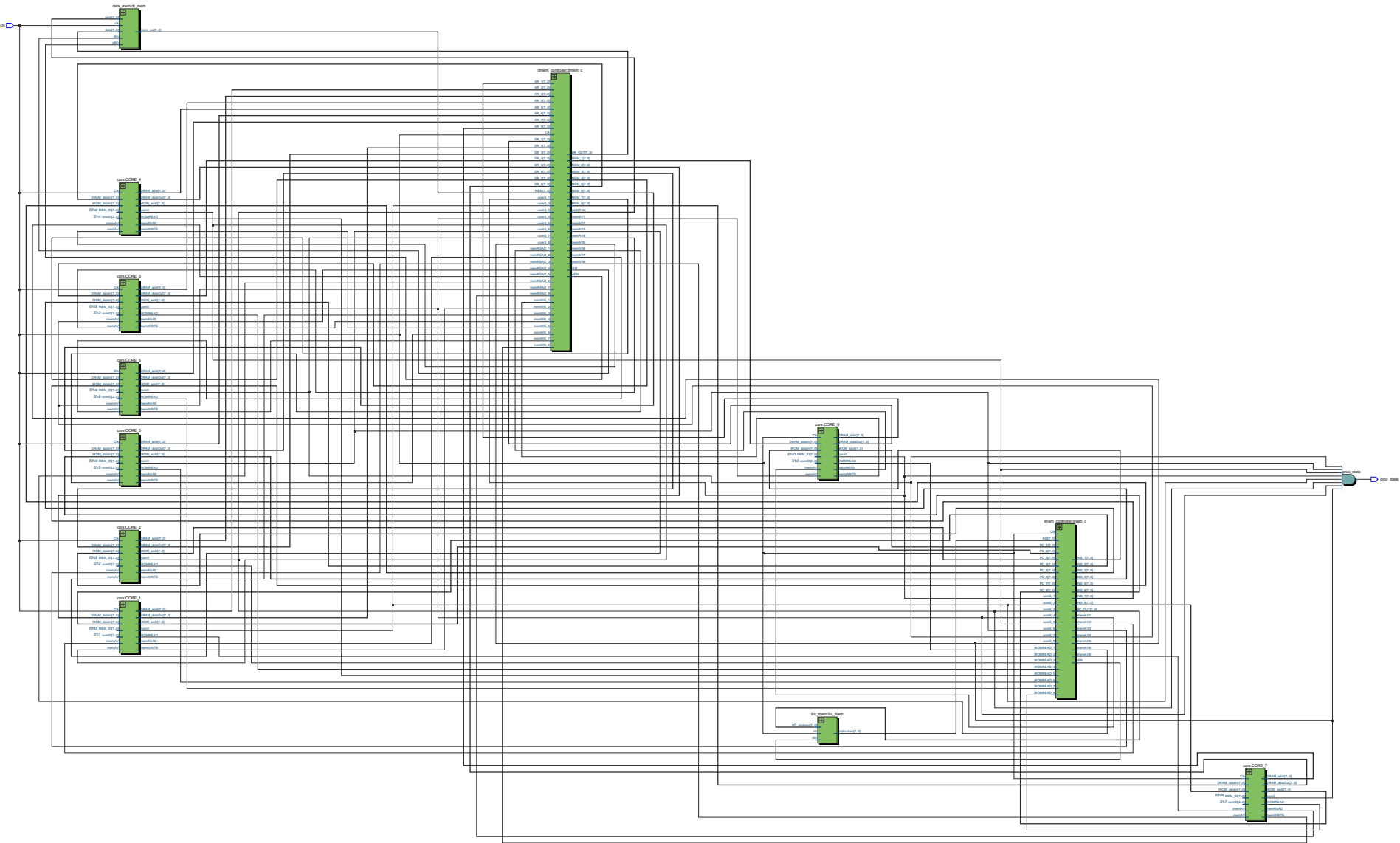
4.3 Processor - Multi Core

The multi-core processor is also a top level module, however it has several additional components:

Table 4.3: Components of Multi Core Processor

1-Several Cores	:	Each core contains processing and control related components
2-Instruction Memory	:	Contains the instructions which will be given to all the cores
3-Data Memory	:	RAM which data can be read from and written to
4-Instruction Memory Controller	:	Controls data flow between Instruction Memory and Cores
5-Data Memory Controller	:	Controls data flow between DRAM and Cores

Eight-Core Processor Top Module



4.4 System Components

This section discusses the architecture and implementation of each individual module of the processor.

4.4.1 Core

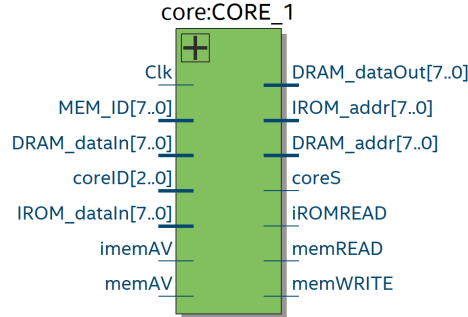


Figure 4.3: Core Module

The Core module can be compared to a single physical CPU, as it functions as a complete processing unit, capable of executing various instructions, carrying out arithmetic and logical operations and has a host of input and output ports as shown in the diagram 4.3.

It is connected to the data memory (DRAM) and the instruction memory (IROM) with an address (output) port and a data input port each. In addition to these, and the data output port used to write data to the DRAM, the core also sends read and write signals to the data memory and a read signal to read the instruction memory. The core also has some special ports to handle the multicore process handling of matrix multiplication. These include:

	coreID	:	Core identifying number
IN	MEM_ID	:	DRAM Address to STORE output from core
	imemAV	:	Signals whether instruction memory is available
	memAV	:	Signals whether data memory is available
OUT	coreS	:	Signals whether the core is active

Table 4.4: Multi-core Processor - Core Signals

In this project, we implement an eight-core processor. Each core is given access to memory only via the instruction memory and data memory controller.

4.4.2 Instruction Memory

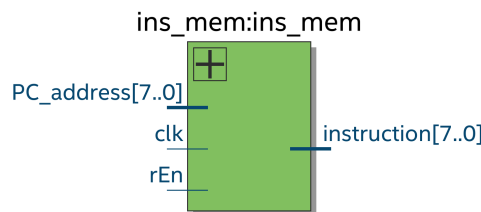


Figure 4.4: Instruction Memory Module

The Instruction Memory or IROM is the memory module that is used to store instructions. The word size of the IROM is 8 bits.

Type A instructions which are 8-bits long can be stored in one word, while 16-bit type B instructions use two word spaces to store the instruction. This is read by fetching the next word within the micro instructions of each type B instruction.

The IROM is connected to three registers in the CPU/cores. These are the PC (Program Counter), IR (Instruction Register) and AR. The PC points to the address of the IROM and the data_out of the IROM is connected to the IR and the AR directly. It should be noted that the IR is reserved for the function of reading data from IROM and pass it to the Control Unit inside the core.

Since the second word is always a memory address, either a DRAM address (in the case of COPY, STORE, LOAD & ASSIGN) or an IROM address (in the case of JPNZ), this address is read directly in to the AR. This achieves the purpose of saving a clock cycle for moving the address from IR (where instructions are usually read to) to the AR before data memory access.

In our multi-core implementation, all the cores access the same IROM and access the same set of instructions, through the instruction memory controller. That is, from the perspective of the IROM, there is no difference between communicating with a single or multiple cores. It only has to get the read enable signal and output the relevant instruction.

4.4.3 IROM Controller

The IROM controller, as mentioned before, functions as an interface to communicate between the IROM and the multiple cores (8 cores in our implementation).

Since this processor is application specific and built only for matrix multiplication, our design simplifies the instruction access by carrying out parallel instruction access for all the cores. In other words, all the cores will run each instruction in parallel. This is a simple and straightforward design that takes advantage of the fact that each core will have to carry out the exact same algorithm and thereby simply gives out all the instructions simultaneously.

If any of the cores are busy and cannot proceed to the next instruction, the controller waits until they all have come to the same point in the algorithm. Note that our IROM is read-only, so this was found to be a sufficiently efficient implementation for our program for matrix multiplication.

IN	coreS	: Core status of each of the cores
	iROMREAD	: Read signal from each of the cores
OUT	rEN	: Read Enable signal sent to IROM
	imemAV	: Signals to each core whether instruction memory is available

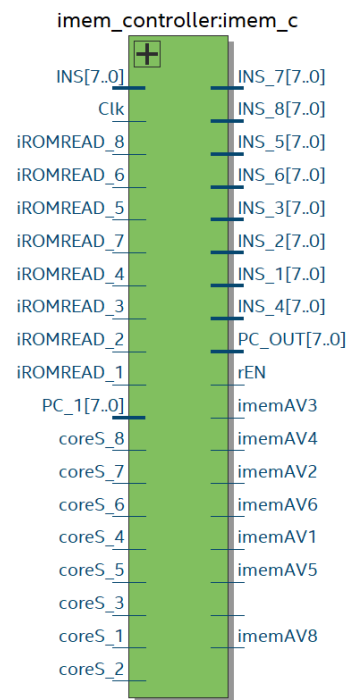


Figure 4.5: Instruction Memory Controller

Table 4.5: IROM Controller Control Signals

4.4.4 Data Memory

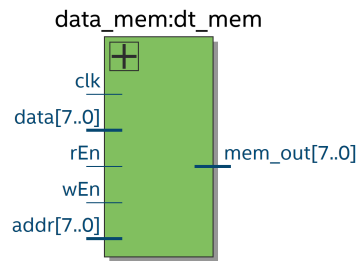


Figure 4.6: Data Memory Module

The names Data Memory, Data RAM and DRAM refer to this module. It is a 256 by 256 sized memory block, which means the address size is 8 bits and each word is 8 bits long.

The data input as well as output for DRAM is always from the DR (Data Register) of the CPU/cores and the address pointer is always the AR (Address Register). In the implementation, the data memory data_in is directly connected to the DR but the data_out is connected to the data bus of the core.

In our multi-core implementation of the data memory, it takes in a single data input and read/write signal and outputs the data output to the data bus. That is, from the perspective of the DRAM, there is no difference between communicating with a single or multiple cores. It only has to get the read/write enable signal and carry out the relevant action.

For the efficient memory allocation in the Data Memory we allocated the data addresses to each core as in the 4.7a. By that, When the processor is running with low number of cores, the Data Memory is optimized for maximum memory utilization as in 4.7b.

Input Data	All Cores	0 . . . 126
	Core 1	127 . 142
	Core 8	143 . 158
	Core 4	159 . 174
	Core 5	175 . 190
	Core 2	191 . 206
	Core 7	207 . 222
	Core 3	223 . 238
	Core 6	239 . 255
Result Data	All Cores	0 . . . 126
	Core 1	127 . 142
	Core 8	143 . 158
	Core 4	159 . 174
	Core 5	175 . 190
	Core 2	191 . 206
	Core 7	207 . 222
	Core 3	223 . 238
	Core 6	239 . 255
	Core 1	127 . 142
	Core 8	143 . 158
	Core 4	159 . 174
	Core 5	175 . 190
	Core 2	191 . 206
	Core 7	207 . 222
	Core 3	223 . 238
	Core 6	239 . 255

(a) DRAM Memory Allocation

(b) Allocation: 4-core

Figure 4.7: Memory Allocations for the cores

4.4.5 DRAM Controller

The DRAM controller acts as an interface between the multiple cores (8 cores in our implementation) and the DRAM.

It can take in different address pointer inputs from each of the cores, and carry out memory access one by one in order of priority or on a first-come, first-serve basis while keeping all the other cores in 'HOLD state (where the core is idle and waiting for the memory access to happen). Each data output from the memory is assigned to the relevant core's DR register. Data STORE is also handled similarly, with each core taking turns while others are put in the 'HOLD state.

In our multi core implementation of the processor, the DRAM controller module is optimized for the application specific purpose, and also facilitated by the instruction memory access described above to give a simple and efficient functional processor, the DRAM access by all the cores is sometimes to the same address. This cuts down on the waiting time for all the cores and the memory access is done using the same number of cycles as our single core implementation.

4.5 Core Components

4.5.1 Control Unit

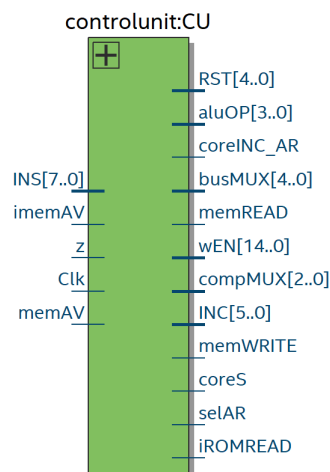


Figure 4.8: Control Unit

The Control unit functions as the main control module of our core module. It takes only one data input, an 8-bit instruction from the IR register. Additionally, it takes 3 flag inputs and a clock. The two memory checking flags function to inform the CU whether or not the memory access is available, and if not, the CU remains in 'HOLD position as previously mentioned. The Zero Flag is output by the comparator mux for the purpose of the matrix multiplication.

The CU stores control signals during an instruction execution. As all the sub components of the system run on a clock edge, there sometimes has to be a delay between data/inputs being available to a module and the action taking place. The delay caused by this was minimized in our implementation by setting some modules to take certain actions on negative edge, and the FLAGS are always read on the negative edge inside the control unit.

Sometimes this is not necessary during the processing of the instruction, but when an instruction is fetched, it needs to be processed in order to generate the control signal. The processing of the instruction is done in one single clock cycle and this processing state is defined as 'FETCH_3.

The following code demonstrates how 'FETCH_3 state proceeds with the next stage by using the instruction input.

Type	Signal	Name/Description	Size (bits)
MEMORY	iROMREAD	read instruction from memory	1
	memREAD	read data from memory	1
	memWRITE	write data to memory	1
	memAV	data memory available FLAG	1
	imemAV	instruction memory available FLAG	1
REGISTER	wEn	write Enable register	15
	INC	Increment register	6
	RST	Reset register	5
	selAR	Select AR Action	1
	coreINC_AR	Increment AR by core ID	1
MUX	busMUX	Read register to bus MUX	5
	compMUX	Register to run through comparator	3
	z	Zero FLAG	1
ALU	aluOP	ALU Operations	4

Table 4.6: Control signals

```

1  case (INS[7:4])
2    'JMP : begin
3      case (INS[3:0])
4        'JMP_M : NEXT_STATE <= 'JMPM_1;
5        'JMP_K : NEXT_STATE <= 'JMPK_1;
6        'JMP_N : NEXT_STATE <= 'JMPN_1;
7      endcase
8    end
9    'COPY : begin
10     NEXT_STATE <= 'COPY_1;
11     iROMREAD <= 1;
12   end
13   'LOAD : begin
14     case (INS[3:0])
15       'LOAD_C1 : NEXT_STATE <= 'LOADC1_1;
16       'LOAD_C2 : NEXT_STATE <= 'LOADC2_1;
17     endcase
18   end
19   'STORE : NEXT_STATE <= 'STORE_1;
20   'ASSIGN : begin
21     NEXT_STATE <= 'ASSIGN_1;
22     iROMREAD <= 1;
23   end
24   'RESET : begin
25     case (INS[3:0])
26       'RESET_ALL : NEXT_STATE <= 'RESETALL_1;
27       'RESET_N2 : NEXT_STATE <= 'RESETN2_1;
28       'RESET_K2 : NEXT_STATE <= 'RESETK2_1;
29       'RESET_Rt : NEXT_STATE <= 'RESETRt_1;
30     endcase
31   end
32   'MOVE : begin
33     case (INS[3:0])
34       'MOVE_RP : NEXT_STATE <= 'MOVEP_1;
35       'MOVE_RT : NEXT_STATE <= 'MOVET_1;
36       'MOVE_RC1 : NEXT_STATE <= 'MOVEC1_1;
37       'MOVE_C3 : NEXT_STATE <= 'MOVEC3_1;
38     endcase
39   end
40   'SET : begin
41     case (INS[3:0])
42       'SETC1 : NEXT_STATE <= 'SETC1_1;
43       'SETDR : NEXT_STATE <= 'SETDR_1;
44     endcase

```

```

45     end
46     'MUL : begin
47         case (INS[3:0])
48             'MUL_RP : NEXT_STATE <= 'MULRP_1;
49         endcase
50     end
51     'ADD : begin
52         case (INS[3:0])
53             'ADD_RT : NEXT_STATE <= 'ADDRT_1;
54             'ADD_RR1 : NEXT_STATE <= 'ADDRR1_1;
55             'ADD_RM2 : NEXT_STATE <= 'ADDRM2_1;
56             'ADD_MEM : NEXT_STATE <= 'ADDC3_1;
57         endcase
58     end
59     'INC : begin
60         case (INS[3:0])
61             'INC_C2 : NEXT_STATE <= 'INCC2_1;
62             'INC_C3 : NEXT_STATE <= 'INCC3_1;
63             'INC_M2 : NEXT_STATE <= 'INCM2_1;
64             'INC_K2 : NEXT_STATE <= 'INCK2_1;
65             'INC_N2 : NEXT_STATE <= 'INCN2_1;
66         endcase
67     end
68     'END : NEXT_STATE <= 'ENDOP_1;
69     'CHK_IDLE : NEXT_STATE <= 'CHKIDLE_1;
70     'GET_C1 : NEXT_STATE <= 'GETC1_1;
71 endcase

```

4.5.2 ALU

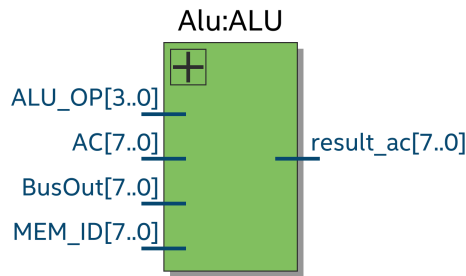


Figure 4.9: ALU Module

The Arithmetic and Logical Unit or ALU Module is responsible for two arithmetic operations in our implementation. This includes setting the AC value (for accumulated calculation), multiplication and addition of an input from the data bus with the value stored in AC. The operations are demonstrated in the code below:

```

1  always @* begin
2      case (ALU_OP)
3          SET:      result <= BusOut;
4          MUL:      result <= AC * BusOut;
5          ADD:      result <= AC + BusOut;
6          default:  result <= AC;
7      endcase
8  end

```

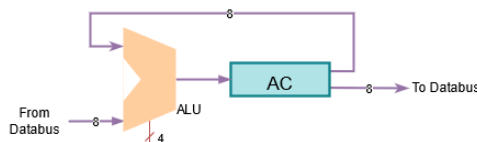


Figure 4.10: ALU & AC register

The ALU gets a 4-bit control signal that activates SET, MUL or ADD or pass mode, that last of which simply maintains the value in AC register. In this design, the register AC's input is only from the ALU, so writing any values to AC requires it to be passed through the ALU.

4.6 Muxes & Comparators

During the design of the system architecture, multiplexers and a comparator was used to build an efficient and maintain smooth communication within the core.

4.6.1 Data Bus

The Bus Mux functions as the data bus for reading data output between 17 registers and the DRAM Memory modules data_out module.

```

1  always @(*)
2  begin
3      case(mux_sel)
4          5'b00001: select <= AC;
5          5'b00010: select <= C3;
6          5'b00011: select <= C2;
7          5'b00100: select <= C1;
8          5'b00101: select <= RN2;
9          5'b00110: select <= RK2;
10         5'b00111: select <= RM2;
11         5'b01000: select <= RN1;
12         5'b01001: select <= RK1;
13         5'b01010: select <= RM1;
14         5'b01011: select <= RT;
15         5'b01100: select <= RP;
16         5'b01101: select <= DR;
17         5'b01110: select <= AR;
18         5'b01111: select <= MEM;
19         5'b10000: select <= RR;
20         5'b10001: select <= RT4;
21     endcase
22 end

```

As shown in figure 4.11, the data bus has a separate 8-bit input connection to each of the 17 registers and a five bit selection input from the control unit. The bus mux's output data is the content of the selected register (as shown above) which can be accessed by all the registers who take data_in input from the bus mux.

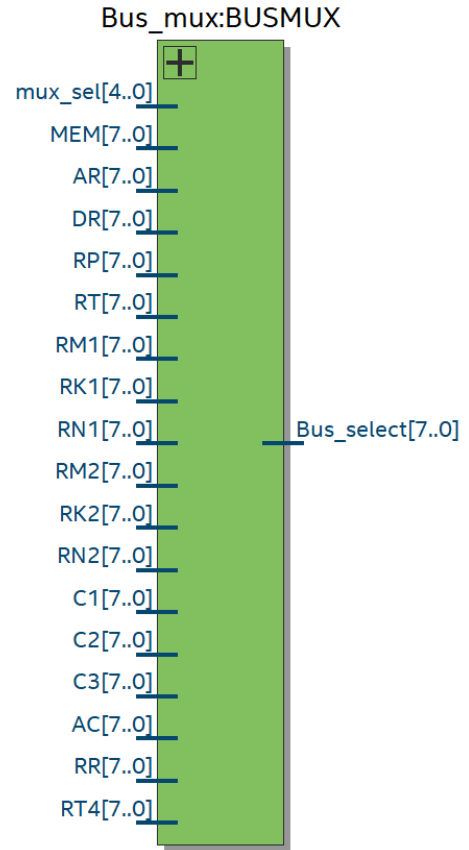


Figure 4.11: Data-bus Multiplexer Module

4.6.2 Comparator & its Muxes

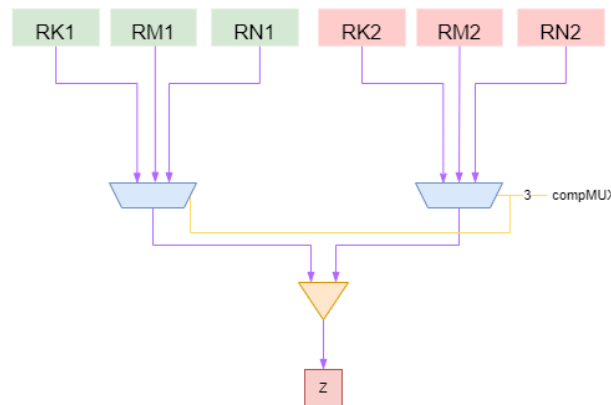


Figure 4.12: Comparator and Comp-Muxes Block Diagram

The comparator setup is a unique feature in our design that allows our implementation to:

1. Reduce an ISA instruction by having multiple parameters for the JPNZ (Jump if Not Zero) instruction and carrying out the comparison as a sub instruction. This also makes the algorithm shorter.
2. Reduce the number of operations carried out by the ALU (i.e. a subtraction) that is also non-accumulative, thus allowing us to maximize the ALU in our design.
3. Reduce the number of clock cycles of the entire algorithm. While a subtraction between two registers would include

The figure 4.12 shows the logical connection between the six registers and three modules. The control unit sends a common mux selector to both the comparator muxes after which the mux output becomes the value of the register which is needed to be compared. The comparator output is subtracted in the comparator and a z (zero flag) output is sent to the control unit. All of this happens in one clock cycle without waiting for a clock edge, which makes sure that the control unit gets the response within one clock cycle.

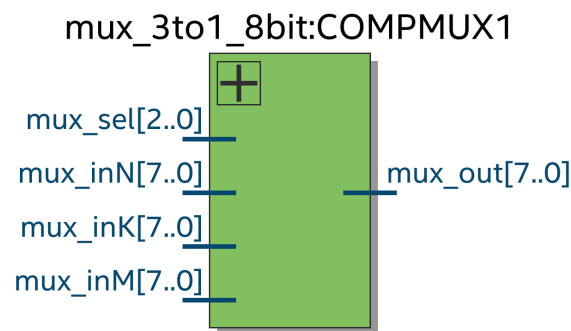


Figure 4.13: Comparator Bus Mux

The comparator mux is a 3-to-1 multiplexer of which the logic is as follows:

```

1 always @(*)
2 begin
3     case (mux_sel)
4         3'b001: mux_out <= mux_inN;
5         3'b010: mux_out <= mux_inK;
6         3'b100: mux_out <= mux_inM;
7         default: mux_out <= 8'bz;
8     endcase
9 end

```

The comparator module is a fairly simple module that takes in two 8-bit values and does the subtraction. The output given to the CU is either 0 or 1.

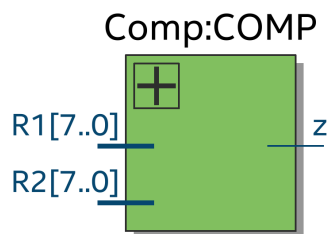


Figure 4.14: Comparator

```

1 always @(*) begin
2     value <= R1-R2;
3     flagOut <= (value == 8'b0);
4 end

```

In our design for matrix multiplication, this module is used to check whether an instruction loop should be continued or the loop should be completed and the next instruction is to be read. Since our algorithm for this implementation consists of three loops, this element is used to compare three pairs of registers. A reg port "zFlag" was created and assigned at the negative edge to the comparator output in order to ensure the "z" flag input to control unit is available at the next positive edge.

4.7 Registers

4.7.1 Program Counter (PC)

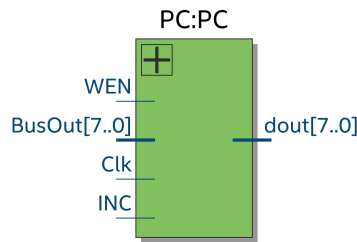


Figure 4.15: Program Counter PC Register

The Program Counter register takes inputs from AR register and outputs from PC are addresses which are sent to IROM so that the required instruction can be read.

```

1 always @(negedge Clk)
2 begin
3
4     if (WEN) value <= BusOut;
5     else if (INC) value <= dout + 8'b1;
6 end

```

- Control signals: 2
- Input : AR output
- Output : To IROM

The PC constantly increments by one during the course of selecting instructions, except when the JPNZ instruction is passed. This instruction writes a new value on to the PC. Additionally, when reading one of the Type B instructions, in order to access the operand, the Program Counter is used.

In our multicore implementation, reading the IROM happens in parallel for efficient matrix multiplication, so the PC is always the same for all the cores before an IROM read is carried out.

4.7.2 Address Register (AR)

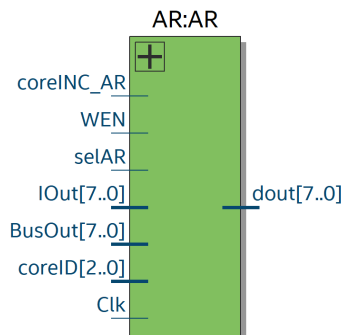


Figure 4.16: Address Register AR

The Address Register is the register that points at the address bus of the DRAM. In our implementation, the AR has two types of write instructions to write from the data bus or from IROM output. Additionally, for the multi core implementation, in order to access the relevant information by each core when reading a common instruction from the IROM, it does not increment by 1, but by Core ID. This makes it a unique register type.

- Control signals: 3
- Inputs : Data bus, IROM output, coreID
- Outputs : To DRAM address, Bus Mux

```

1 always @(negedge Clk)
2 begin
3     if(WEN==1 && selAR == 0 && coreINC_AR == 0) begin
4         value <= BusOut;
5     end
6     if (coreINC_AR==1) begin
7         value <= value + coreID;
8     end
9     if(WEN==1 && selAR == 1 && coreINC_AR == 0) begin
10        value <= IOut;
11    end
12 end

```

4.7.3 Read-Write Registers

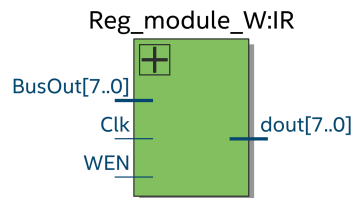


Figure 4.17: Read-Write Register

These registers have simple read and write function. They are written through the data bus and the output is sent to the data bus. They have only a write-enable signal, which takes in the data bus output while the output is constantly given to the Bus Mux.

This is the most basic register type and they are used in the following registers, which on a higher logical level takes values as shown.

- Control signals: 1
- Input : Data bus
- Output : To Bus Mux

```

1 always @(negedge Clk)
2 begin
3
4     if (WEN) value <= BusOut;
5
6 end

```

Register	Function	Data Type
IR	: Contains Instructions, sends to CU	Instruction
DR	: Contains Data read from or written to DRAM	Positive Integer
RP	: Stores MUL result from AC	Positive Integer
RR	: Holds matrix data - M	Positive Integer
Reg M1	: Holds matrix data - M per core	Positive Integer
Reg K1	: Holds matrix data - K	Positive Integer
Reg N1	: Holds matrix data - N	Positive Integer
Reg C1	: Mem Addr Pointer	Address

4.7.4 Write-Reset Registers

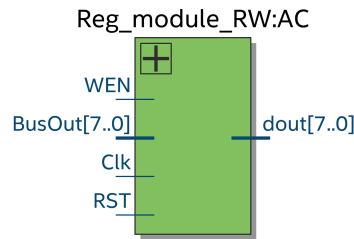


Figure 4.18: Reset-Write Register

These registers are fairly simple and take two control signals as the name suggests - write Enable and Reset. While write enable writes from Data bus, the reset simply sets the value inside it to zero.

Two of the registers that are of this type are the ALU related registers, i.e. the ACcumulator and the Temporary Register. The AC register is used to send the the ALU results to other registers.

The T4 register is used by each core in the multi-core implementation to store the location of the first element of its respective row (as assigned in the program).

- Control signals: 2
- Input : Data bus
- Output : To Bus Mux

Register	Function	Data Type
RT	: Contains temporary value from AC	Positive Integer
AC	: Contains ALU result	Positive Integer
Reg T4	: Holds Mem addr	Address

```

1 always @(negedge Clk)
2 begin
3
4     if (WEN) value <= BusOut;
5     else if (RST) value <= 8'b0;
6 end

```

4.7.5 Write-Increment Registers

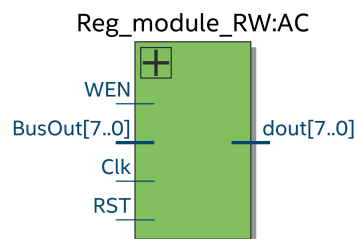


Figure 4.19: Write-Increment Register

As shown below, the write and reset enabled registers are two of the pointer type registers that contain the address of the second matrix being read and the result matrix being written respectively. Both these pointers are constantly incremented during the LOAD, STORE processes.

It can be noted that in the multicore implementation, the register C3 is given an input of the memory ID assigned to the core for writing the resultant matrix.

- Control signals: 2
- Input : Data bus
- Output : To Bus Mux

However, C2 contains the address of the second matrix, which, in our design, all the cores have to access during multiplication.

Register	Function	Data Type
Reg C2	: Mem Addr Pointer	Address
Reg C3	: Mem Addr Pointer	Address

```

1 always @(negedge Clk)
2 begin
3
4     if (WEN) value <= BusOut;
5     else if (INC) value <= dout + 8'b1;
6 end

```

4.7.6 Write-Reset-Increment Registers

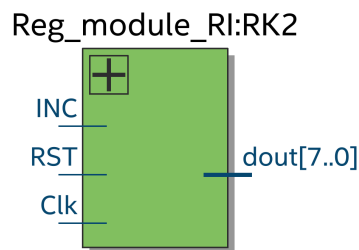


Figure 4.20: Reset-Increment Register

These registers have 3 controls and, as show below, are used as counters for each loop. They are all RESET to zero at the beginning of running the algorithm, then INCRemented continuously while comparing with the write-enable registers M1, K1, N1 respectively through the setup in Figure 4.12.

Register	Function	Data Type
Reg M2	: M loop counter	Positive Integer
Reg K2	: K loop counter	Positive Integer
Reg N2	: N loop counter	Positive Integer

- Control signals: 3
- Input : Data bus
- Output : To Bus Mux

```

1 always @(negedge Clk)
2 begin
3
4     if (RST) value <= 8'b0;
5     else if (INC) value <= dout + 8'b1;
6 end

```

5 Timing Diagrams of Instructions

Our Multi Core processor is designed such that the control signals are issued by the Control Unit at the positive clock edge and the register modules respond at the negative clock edge. By experimentation with hand-driven clock, we found the following behavior with the RAM module, ROM module, DRAM Controller module and the ROM Controller module and fine-tuned our timing to reduce the number of clock cycles required for the Memory operations:

iROMREAD control signal, is enabled from the state before Fetch_1 since there could be a clock cycle delay from the ROM controller module. By then the Control unit can make sure that the Instruction is available from the Instruction Memory in the Fetch_3 state. Also the ROM Controller Module is capable of fetching the instruction from the Instruction Memory and distributing it to each core at the same time, since our multi core architecture follows the SIMD (Single Instruction Multiple Data) approach.

RAM Controller Module is responsible for handling data, reading and writing requests from each core at the same time. In this scenario the RAM Controller Module follows a prioritizing approach for the memory access of the cores. The Data is sent one after the other, and in the meantime, Control units of the cores are put into the HOLD state until all cores finish their memory access.

5.1 FETCH

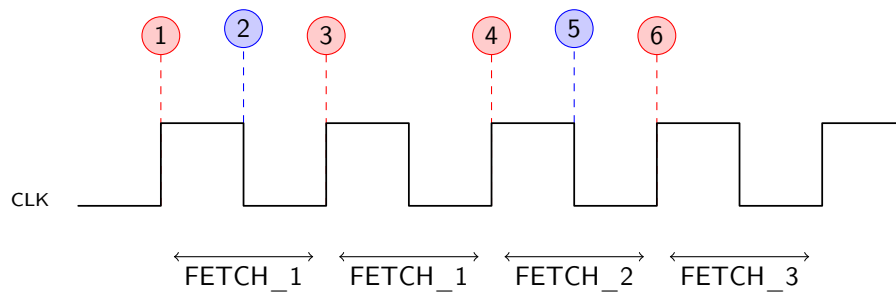


Figure 5.1: Clock cycles of FETCH

1. Control signal, iROMRead is given to read the instruction.
Check imemAV control signal to make sure that the instruction is available.
Wait for a another FETCH_1 cycle until the instruction is available.
2. Instruction is available from the Instruction Memory.
3. If imemAV control signal is 1, move to FETCH_2.
4. IR is updated with the Instruction.
PC is incremented.
5. Instruction is available for control unit from IR.
6. Fetched instruction is decoded.

5.2 JUMP

5.2.1 zFlag = 0

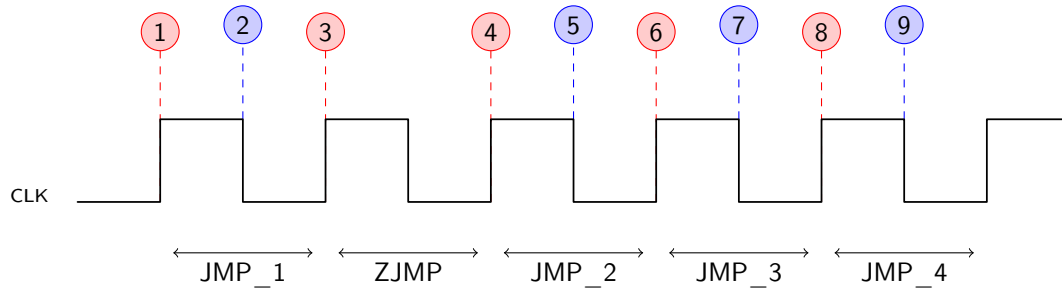


Figure 5.2: Clock cycles of JUMP (z = 0)

1. Control signal, compMUX is set to the two Multiplexers according to the which JMP instruction that control unit has. Comparator outputs the z control signal.
2. Control signal, zFlag is available for the CONTROL Unit.
3. If zFlag is 0, proceed to JMP_2.
4. Control signal, iROMREAD is enabled to get the jump location.
5. Jump address is available for AR from the Instruction Memory.
6. Control signal, selAR is given to AR to get the input from the Instruction Memory. AR is updated with the Jump Address.
7. Jump address is available for Control Unit from AR
8. PC is updated with the new jump address.
9. Jump address is available for the Instruction Memory from PC.

5.2.2 zFlag = 1

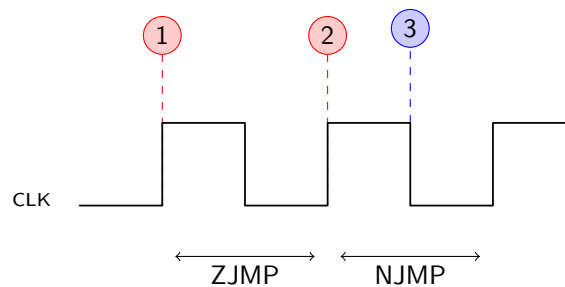


Figure 5.3: Clock cycles of JUMP (z = 1)

1. if zFlag is 1, proceed to NJMP.
2. PC is incremented.
3. PC address is available for the Instruction Memory from PC.

5.3 COPY

5.3.1 Fetching Copy Address

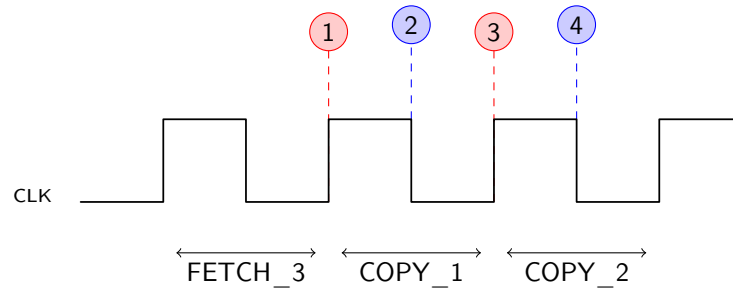


Figure 5.4: Clock cycles of COPY (fetching Copy Address)

1. Control signal, iROMRead is given to read the copy address from Instruction Memory.
2. Copy address is available from the Instruction Memory.
3. Control signal selAR, is given to AR to get the input from the Instruction Memory.
AR is updated with the Instruction.
PC is incremented.
Check for the parameter of the COPY instruction to proceed.
4. Instruction is available for control unit from AR.

5.3.2 Address assigning for each core

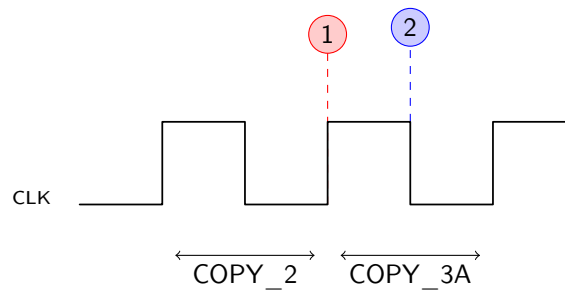


Figure 5.5: Clock cycles of COPY 3A (Increment Addresses)

1. Control signal, coreINCAR is given to the AR to increment the AR by the CORE ID of each core.
2. Incremented address is available from AR.

5.3.3 Fetching from Data Memory

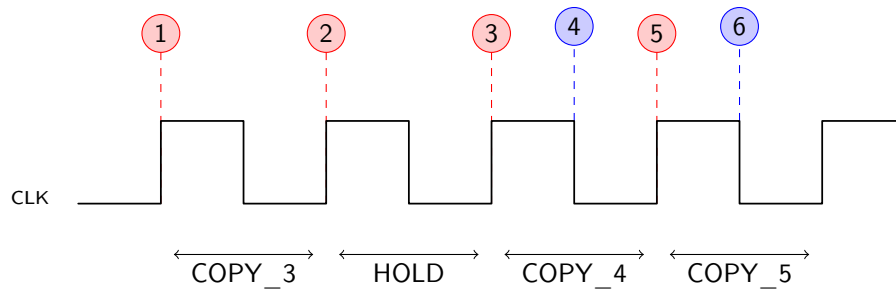


Figure 5.6: Clock cycles of storing data in registers

1. Control signal, memRead is given to read data from Data Memory.
BusMux is enabled for the Data Memory lane.
2. HOLD the processor until the data is available from the Data Memory.
3. DR is updated with the fetched data from the Data Memory.
Check for the parameter of the COPY instruction to proceed.
4. Data is available for Control unit from DR.
5. RM1, RN1, RK1, RR1 & RT1 registers are updated with the Data, according to the parameter of the Copy instruction.
6. Data is available for Control unit from RM1, RN1, RK1, RR1 & RT1.

5.4 LOAD

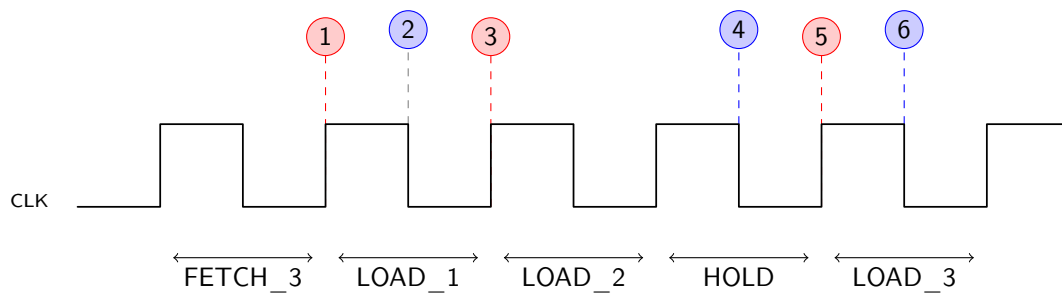


Figure 5.7: Clock cycles of LOAD

1. BusMux is enabled for the C1 register.
AR is updated with the value of C1.
2. Data address is available for the Data Memory from AR.
3. Control signal memREAD is enabled to read data from Data Memory.
Proceed to HOLD state until data is available.
4. Data is available for the Control unit from the Data Memory.
5. BusMux is enabled for the Memory line.
DR is updated with the Data.
6. Data is available for the Control unit from DR.

5.5 STORE

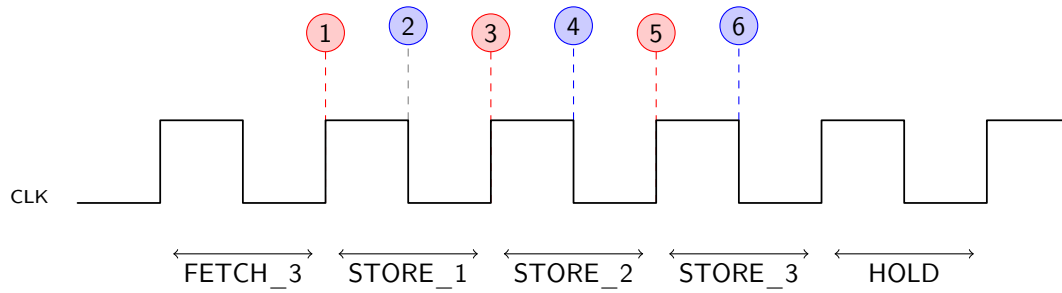


Figure 5.8: Clock cycles of STORE

1. BusMux is enabled for the RT register.
DR is updated with the value of RT.
2. Data is available for the Data Memory from DR.
3. BusMux is enabled for the C3 register.
AR is updated with the value of C3.
4. Data address is available for the Data Memory from AR.
5. Control signal memWRITE, is enabled to write data to Data Memory.
Proceed to HOLD state until data is written to Data Memory.
6. Stay on HOLD until memAV control signal is 1.

5.6 ASSIGN

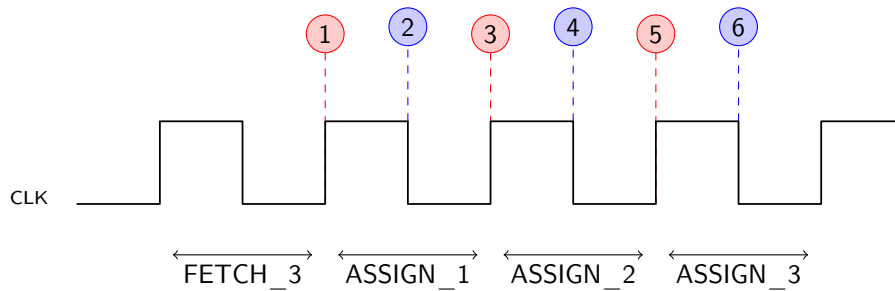


Figure 5.9: Clock cycles of ASSIGN

1. Control signal, iROMRead is given to read address from Instruction Memory.
2. Address is available for AR from the Instruction Memory.
3. Control signal, selAR is given to AR to get the input from the Instruction Memory.
AR is updated with the Assign Address.
PC is incremented.
4. Address is available for Control unit from AR.
PC address is available for the Instruction Memory from PC.
5. RC2 register is updated with the Address.
6. Address is available for Control unit from RC2.

5.7 RESET

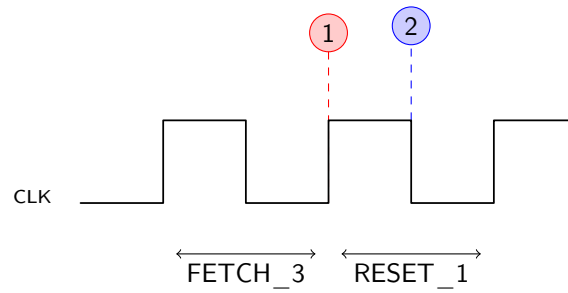


Figure 5.10: Clock cycles of RESET

1. Control signal, RST bit mask is set according to the which registers need to be reset.
2. Respective registers are reset.

5.8 MOVE

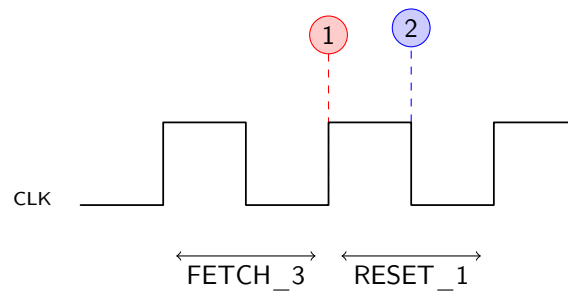


Figure 5.11: Clock cycles of MOVE

1. BusMux is enabled for AC register.
AC value is written into the respective register.
2. Value is available for the Control unit from the respective register.

5.9 SET

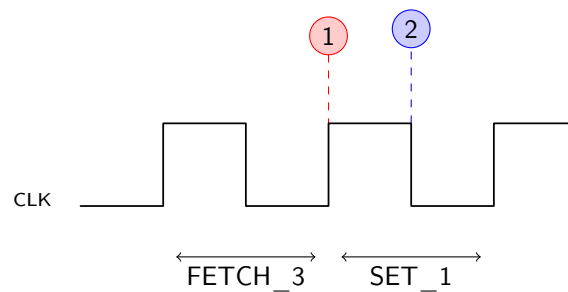


Figure 5.12: Clock cycles of SET

1. BusMux is enabled for the respective register.
AC is updated by the value of the respective register.
2. Value is available for the Control unit from the AC.

5.10 MUL

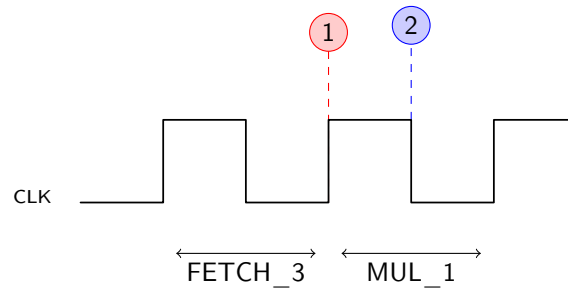


Figure 5.13: Clock cycles of MUL

1. BusMux is enabled for RP register.
Control signal, aluOP is given to the ALU for the multiplication operation.
AC is updated by the result of ALU.
2. Value is available for the Control unit from the AC.

5.11 ADD

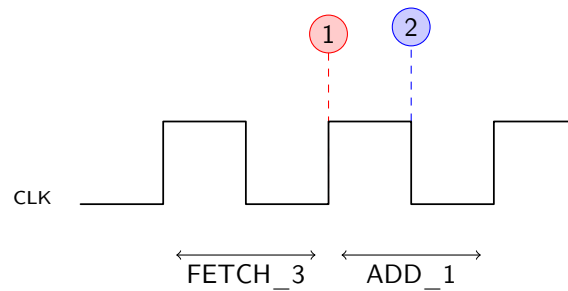


Figure 5.14: Clock cycles of ADD

1. BusMux is enabled for the respective register.
Control signal, aluOP is given to the ALU for the addition operation.
AC is updated by the result of ALU.
2. Value is available for the Control unit from the AC.

5.12 INC

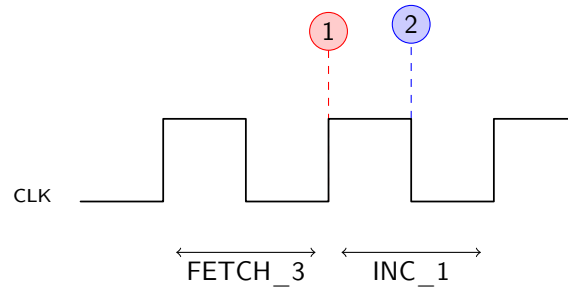


Figure 5.15: Clock cycles of INC

1. Control signal, INC bit mask is given according to the which register needs to be incremented.
2. Incremented value is available for the Control unit from the respective register.

5.13 CHK_IDLE

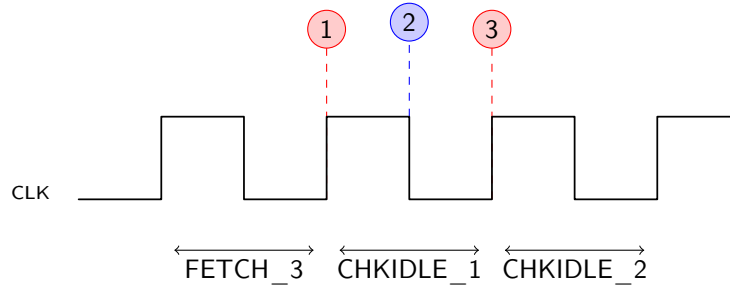


Figure 5.16: Clock cycles of CHK_IDLE

1. Control signal, compMUX is set to check the difference between the values of the register of RM1 & RM2.
2. zFlag value updated from the Comparator output.
3. Check the value of the zFlag. If it is 1, put the core in to IDLE state otherwise continue the process for the next instruction.

5.14 GET

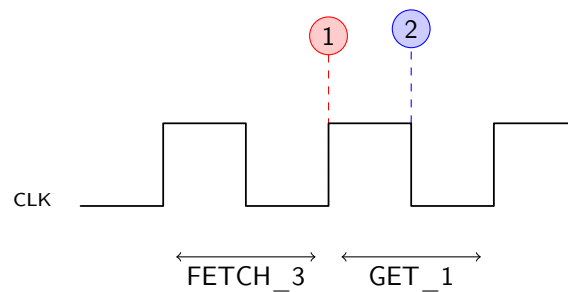


Figure 5.17: Clock cycles of GET

1. BusMux is enabled for the RT4 register.
RC1 is updated with the value of RT4.
2. value is available for the Control unit from the RC1.


6 Results


REMM processor is able to compute a result matrix of a matrix multiplication accurately. Below is the example test case and its output result.

Figure 6.2 shows the optimized DRAM memory utilization for 8-core processing. And the figure 6.3 shows the same example test case as above, carried out using a 4-core processor.

If the same input matrices are given and execution is done using 4-cores only results are same as before and memory allocation is adjusted automatically by the system itself as we codified.

1	No of Cores:	25	Multiplied Matrix(From python)	43	Multiplied Matrix(From FPGA)
2	8	26	[[124 242 124 243 243]	44	[124, 242, 124, 243, 243]
3	Matrix 1:	27	[119 118 119 119 119]	45	[119, 118, 119, 119, 119]
4	1, 120, 1	28	[121 217 121 219 219]	46	[121, 217, 121, 219, 219]
5	115, 1, 1	29	[45 40 45 44 44]	47	[45, 40, 45, 44, 44]
6	15, 100, 2	30	[200 180 200 215 215]	48	[200, 180, 200, 215, 215]
7	30, 3, 4	31	[12 8 12 11 11]	49	[12, 8, 12, 11, 11]
8	45, 50, 35	32	[27 20 27 26 26]	50	[27, 20, 27, 26, 26]
9	1, 2, 3	33	[42 32 42 41 41]	51	[42, 32, 42, 41, 41]
10	4, 5, 6	34	[150 110 150 140 140]	52	[150, 110, 150, 140, 140]
11	7, 8, 9	35	[120 80 120 110 110]	53	[120, 80, 120, 110, 110]
12	40, 20, 30	36	[120 80 120 110 110]	54	[120, 80, 120, 110, 110]
13	10, 20, 30	37	[114 113 114 114 114]	55	[114, 113, 114, 114, 114]
14	10, 20, 30	38	[204 203 204 204 204]	56	[204, 203, 204, 204, 204]
15	110, 1, 1	39	[12 8 12 11 11]	57	[12, 8, 12, 11, 11]
16	200, 1, 1	40	[153 201 153 202 202]	58	[153, 201, 153, 202, 202]
17	1, 2, 3	41	[42 32 42 41 41]]	59	[42, 32, 42, 41, 41]
18	100, 50, 1				
19	7, 8, 9				
20	Matrix 2:				
21	1, 1, 1, 1, 1				
22	1, 2, 1, 2, 2				
23	3, 1, 3, 2, 2				
24					


 Result from
Python


 Result from
FPGA



 Input Matrices

Figure 6.1: Result Comparison

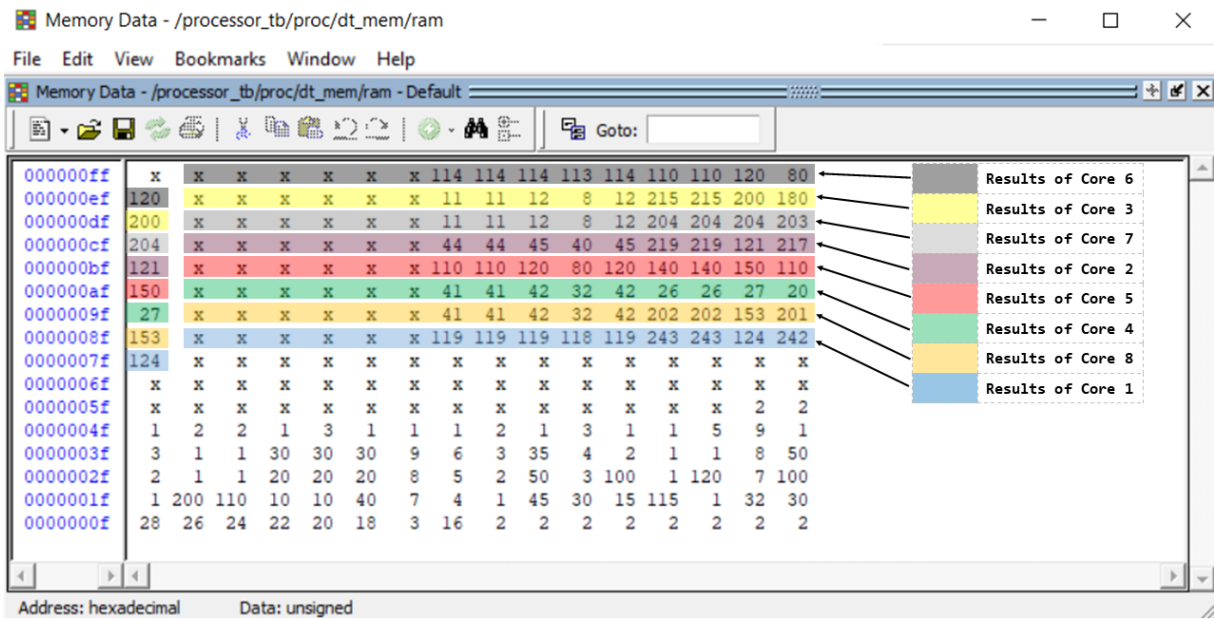


Figure 6.2: DRAM output for 8-cores processing

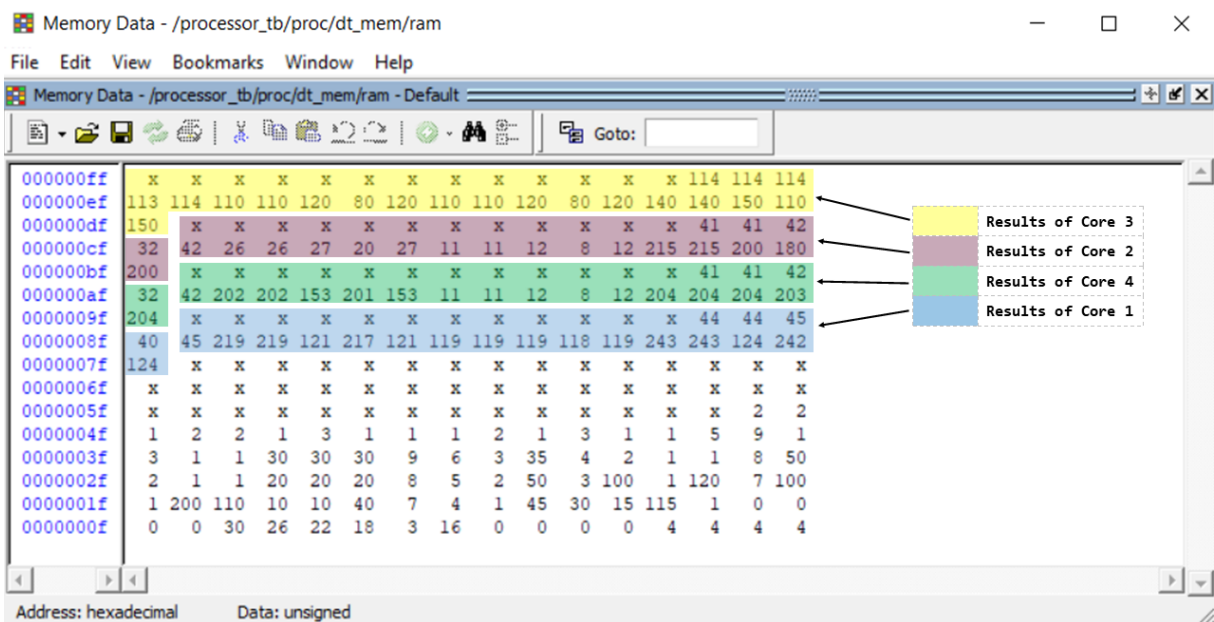


Figure 6.3: DRAM output for 4-cores processing

Test case no.	Matrix 1	Matrix 2	Result Matrix
Test case 1	8×5	5×4	8×4
Test case 2	15×3	3×8	15×8
Test case 3	36×2	2×3	36×3
Test case 4	4×3	3×4	4×4

Table 6.1: Test cases

This section summarizes the four different test cases' (6.1) output results.

As shown in the below figure:

1. The time taken to process matrix multiplication is decreasing when the number of cores are increasing one by one.
2. Significant time difference can be seen between single core processor and dual core processor to process multiplication.
3. Increasing the executing cores more than two, with each core added, change in inference speed is decreasing gradually.
4. The change in inference speed is significant until fourth core is added.

It is obvious that inference speed of 8 cores processor is significantly high when it compares with the inference speed of single core processing. But 5 cores, 6 cores, 7 cores and 8 cores processors doesn't show any significant difference in there inference speed when it compares with all four test cases.

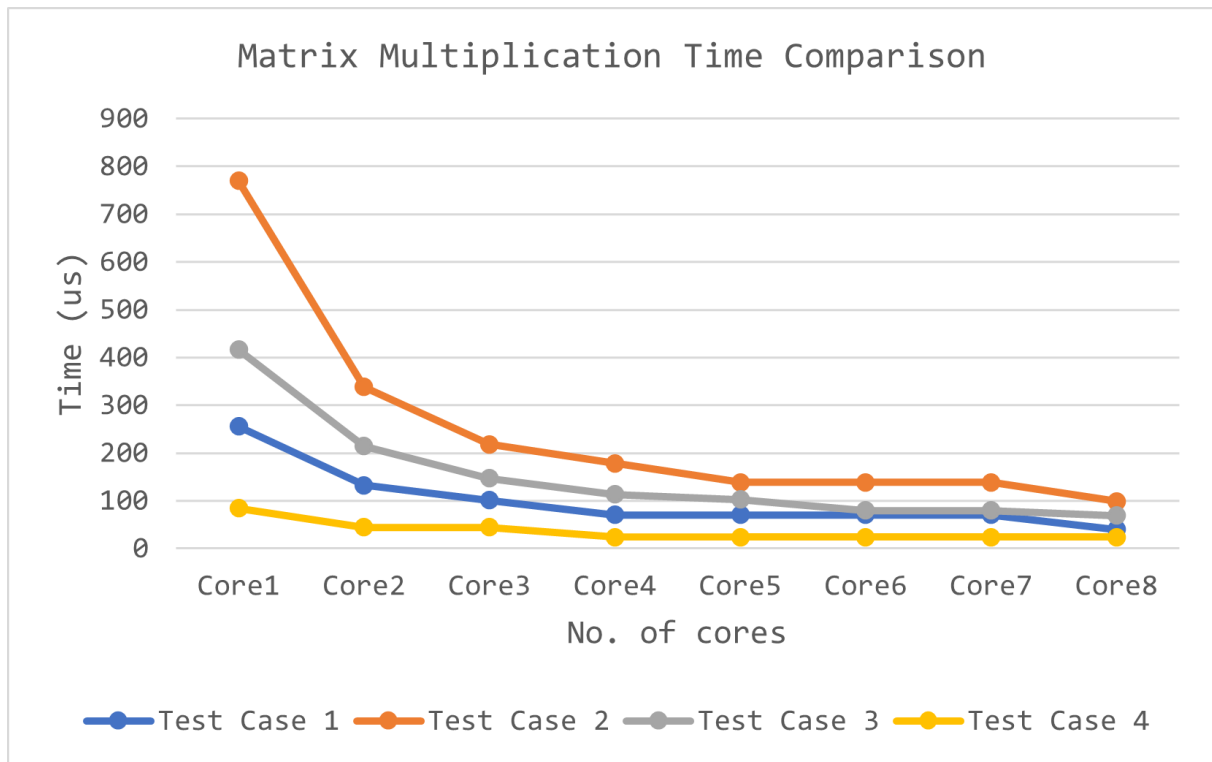


Figure 6.4: Matrix Multiplication Time Comparison

7 Limitations of the system

- Signed number operations are not possible. The system processes positive values only.
- The maximum multiplied value should be less than or equal 255.
- The maximum number of elements of the input matrices (Matrix 1 & Matrix 2) should be less than 108 to overcome the memory overflow.
- DRAM Data Memory size in this design is 8×256 which limits the number of elements that can be handled by the processor.
- Data Memory overflow can occur when executing matrix multiplication using cores less than 8 with maximum memory utilization.

By changing the relevant parameters for memory size and register size, the capability of the processor can be extended. However, the primary purpose of this project is to compare the processing ability of different numbers of cores for matrix multiplication, which is achieved as shown in Section 6 above.

8 Compiler - Python Program

We designed a python based compiler to compile our assembly code algorithm (Figure: 8.1a) into the machine code (Figure: 8.1b) before executing the Matrix Multiplication in the simulation.

At the same time our python program creates a text file for the Data Memory, converting the raw input decimal values of the matrix elements into the base of hexadecimal.

At the end of the operation we use a separate python program to read the Data Memory text file and extract the result from that into a human-readable and understandable format.

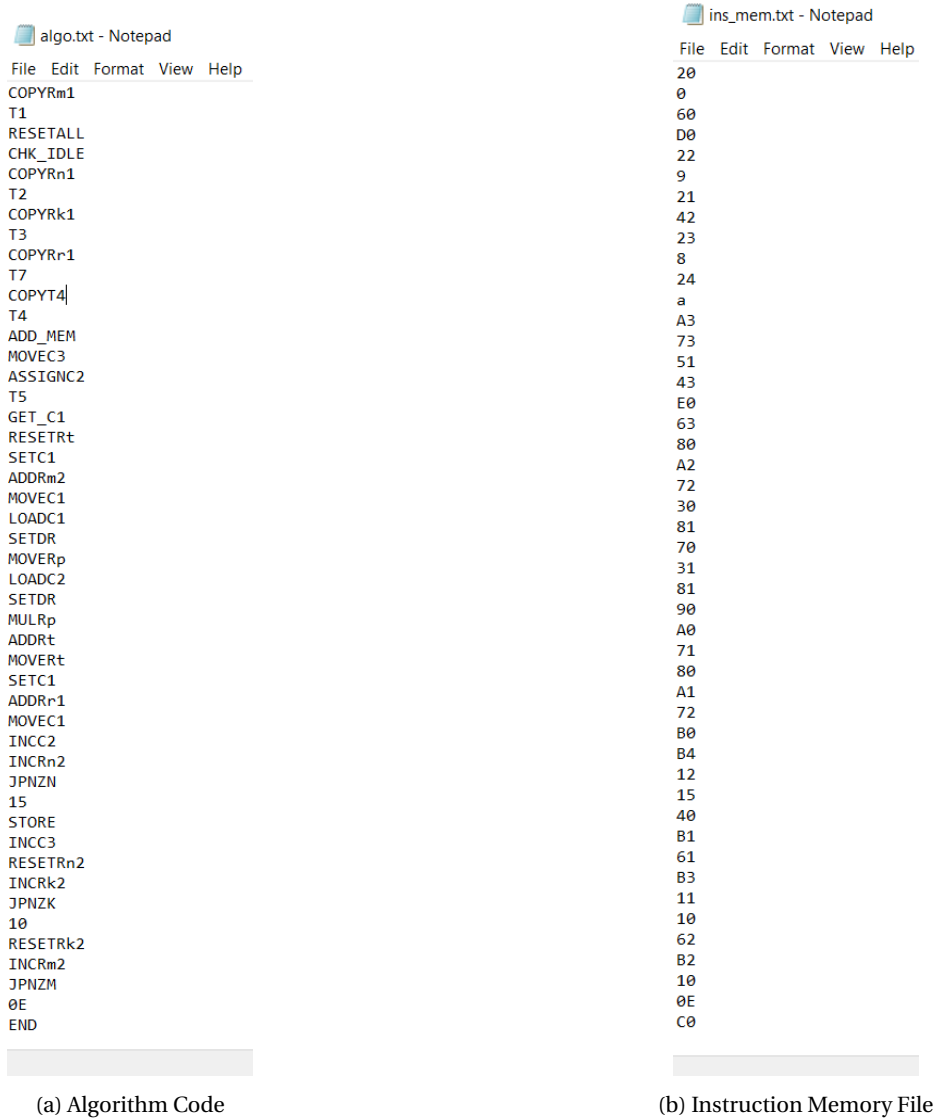


Figure 8.1: Input and the Output of the Python Programme

9 References

1. Carpinelli, John D. Computer Systems Organization & Architecture, catalog.hathitrust.org/api/volumes/oclc/44454679.html.
2. David A. Patterson and John L. Hennessy. 2008. Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) (4th. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
3. Pong P. Chu. 2008. FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version.

10 Appendix

10.1 Processor

```
1
2 module processor
3 # (parameter WIDTH = 8)
4 (
5     input clk,
6     output proc_state
7 );
8 // wire clk = CLOCK_50;
9 wire [(WIDTH)-1:0] INS;                                //iROM
10    output
11 wire [(WIDTH)-1:0] PC_addr;                            //PC out to
12    imem controller
13 wire [(WIDTH)-1:0] MEMWRITE_data;                      //writing
14    data to dram
15 wire [(WIDTH)-1:0] DRAM_addr;                          //dram
16    accessing memory location address
17 wire [(WIDTH)-1:0] MEMREAD_data;                      //dram
18    output
19
20 wire iROMREAD;
21 wire memWRITE;
22 wire memREAD;
23 wire imemAV1, imemAV2, imemAV3, imemAV4, imemAV5, imemAV6, imemAV7, imemAV8;
24 wire iROMREAD_1, iROMREAD_2, iROMREAD_3, iROMREAD_4, iROMREAD_5, iROMREAD_6,
25    iROMREAD_7, iROMREAD_8;
26 wire coreS_1, coreS_2, coreS_3, coreS_4, coreS_5, coreS_6, coreS_7, coreS_8;
27 wire [WIDTH-1:0] PC_1, PC_2, PC_3, PC_4, PC_5, PC_6, PC_7, PC_8;
28 wire [WIDTH-1:0] INS_1, INS_2, INS_3, INS_4, INS_5, INS_6, INS_7, INS_8;
29 wire [WIDTH-1:0] AR_1, AR_2, AR_3, AR_4, AR_5, AR_6, AR_7, AR_8;
30 wire [WIDTH-1:0] DR_1, DR_2, DR_3, DR_4, DR_5, DR_6, DR_7, DR_8;
31 wire memREAD_1, memREAD_2, memREAD_3, memREAD_4, memREAD_5, memREAD_6, memREAD_7,
32    memREAD_8;
33 wire memWE_1, memWE_2, memWE_3, memWE_4, memWE_5, memWE_6, memWE_7, memWE_8;
34 wire [WIDTH-1:0] MEM_1, MEM_2, MEM_3, MEM_4, MEM_5, MEM_6, MEM_7, MEM_8;
35 wire memAV1, memAV2, memAV3, memAV4, memAV5, memAV6, memAV7, memAV8;
36
37 // localparam MEMID_CORE1 = 8'd127;                    //DRAM
38    Store starting locations for respective cores
39 // localparam MEMID_CORE4 = 8'd159;
40 // localparam MEMID_CORE2 = 8'd191;
41 // localparam MEMID_CORE3 = 8'd223;
42
43
44
45 localparam MEMID_CORE1 = 8'd127;
46 localparam MEMID_CORE8 = 8'd143;
47 localparam MEMID_CORE4 = 8'd159;
48 localparam MEMID_CORE5 = 8'd175;
49 localparam MEMID_CORE2 = 8'd191;
50 localparam MEMID_CORE7 = 8'd207;
51 localparam MEMID_CORE3 = 8'd223;
52 localparam MEMID_CORE6 = 8'd239;
53
54
55
56 localparam COREID_1 = 3'd0;
57 localparam COREID_2 = 3'd1;
58 localparam COREID_3 = 3'd2;
59 localparam COREID_4 = 3'd3;
60 localparam COREID_5 = 3'd4;
61 localparam COREID_6 = 3'd5;
```

```

52 localparam COREID_7 = 3'd6;
53 localparam COREID_8 = 3'd7;
54
55 //instruction memory
56 ins_mem #(.ADDR_WIDTH(WIDTH), .INS_WIDTH(WIDTH)) ins_mem(.instruction(INS), .
    PC_address(PC_addr), .clk(clk), .rEn(iROMREAD));
57
58 //data memory
59 data_mem #(.DATA_WIDTH(WIDTH), .ADDR_WIDTH(WIDTH)) dt_mem(.data(MEMWRITE_data),
    .addr(DRAM_addr), .wEn(memWRITE), .clk(clk), .mem_out(MEMREAD_data), .rEn(
    memREAD));
60
61 imem_controller #(.WIDTH(WIDTH)) imem_c(.Clk(clk), .INS(INS), .rEn(iROMREAD), .
    PC_OUT(PC_addr),
62     .iROMREAD_1(iROMREAD_1), .iROMREAD_2(iROMREAD_2), .iROMREAD_3(iROMREAD_3), .
    iROMREAD_4(iROMREAD_4), .iROMREAD_5(iROMREAD_5), .iROMREAD_6(iROMREAD_6), .
    iROMREAD_7(iROMREAD_7), .iROMREAD_8(iROMREAD_8),
63     .coreS_1(coreS_1), .coreS_2(coreS_2), .coreS_3(coreS_3), .coreS_4(coreS_4), .
    coreS_5(coreS_5), .coreS_6(coreS_6), .coreS_7(coreS_7), .coreS_8(coreS_8),
64     .PC_1(PC_1), .PC_2(PC_2), .PC_3(PC_3), .PC_4(PC_4), .PC_5(PC_5), .PC_6(PC_6), .
    PC_7(PC_7), .PC_8(PC_8),
65     .INS_1(INS_1), .INS_2(INS_2), .INS_3(INS_3), .INS_4(INS_4), .INS_5(INS_5), .
    INS_6(INS_6), .INS_7(INS_7), .INS_8(INS_8),
66     .imemAV1(imemAV1), .imemAV2(imemAV2), .imemAV3(imemAV3), .imemAV4(imemAV4), .
    imemAV5(imemAV5), .imemAV6(imemAV6), .imemAV7(imemAV7), .imemAV8(imemAV8));
67
68
69 dmem_controller #(.WIDTH(WIDTH)) dmem_c (.Clk(clk), .MEM(MEMREAD_data), .rEn(
    memREAD), .wEn(memWRITE), .DR_OUT(MEMWRITE_data), .addr(DRAM_addr),
70     .coreS_1(coreS_1), .coreS_2(coreS_2), .coreS_3(coreS_3), .coreS_4(coreS_4), .
    coreS_5(coreS_5), .coreS_6(coreS_6), .coreS_7(coreS_7), .coreS_8(coreS_8),
71     .memAV1(memAV1), .memAV2(memAV2), .memAV3(memAV3), .memAV4(memAV4), .memAV5(
    memAV5), .memAV6(memAV6), .memAV7(memAV7), .memAV8(memAV8),
72     .AR_1(AR_1), .AR_2(AR_2), .AR_3(AR_3), .AR_4(AR_4), .AR_5(AR_5), .AR_6(AR_6), .
    AR_7(AR_7), .AR_8(AR_8),
73     .DR_1(DR_1), .DR_2(DR_2), .DR_3(DR_3), .DR_4(DR_4), .DR_5(DR_5), .DR_6(DR_6), .
    DR_7(DR_7), .DR_8(DR_8),
74     .memREAD_1(memREAD_1), .memREAD_2(memREAD_2), .memREAD_3(memREAD_3), .memREAD_4(
    memREAD_4), .memREAD_5(memREAD_5), .memREAD_6(memREAD_6), .memREAD_7(
    memREAD_7), .memREAD_8(memREAD_8),
75     .memWE_1(memWE_1), .memWE_2(memWE_2), .memWE_3(memWE_3), .memWE_4(memWE_4), .
    memWE_5(memWE_5), .memWE_6(memWE_6), .memWE_7(memWE_7), .memWE_8(memWE_8),
76     .MEM_1(MEM_1), .MEM_2(MEM_2), .MEM_3(MEM_3), .MEM_4(MEM_4), .MEM_5(MEM_5), .MEM_6
    (MEM_6), .MEM_7(MEM_7), .MEM_8(MEM_8) );
77
78
79 core #(.WIDTH(WIDTH)) CORE_0 (.Clk(clk), .IROM_dataIn(INS_1), .DRAM_dataIn(MEM_1
    ), .DRAM_dataOut(DR_1),
80     .IROM_addr(PC_1), .DRAM_addr(AR_1), .memREAD(
    memREAD_1), .memWRITE(memWE_1),
81     .iROMREAD(iROMREAD_1), .coreS(coreS_1),
82     .imemAV(imemAV1), .memAV(memAV1), .MEM_ID(
    MEMID_CORE1), .coreID(COREID_1));
83
84 core #(.WIDTH(WIDTH)) CORE_1 (.Clk(clk), .IROM_dataIn(INS_2), .DRAM_dataIn(MEM_2
    ), .DRAM_dataOut(DR_2),
85     .IROM_addr(PC_2), .DRAM_addr(AR_2), .memREAD(
    memREAD_2), .memWRITE(memWE_2), .iROMREAD(iROMREAD_2), .coreS(coreS_2),
86     .imemAV(imemAV2), .memAV(memAV2), .MEM_ID(
    MEMID_CORE2), .coreID(COREID_2));
87
88 core #(.WIDTH(WIDTH)) CORE_2 (.Clk(clk), .IROM_dataIn(INS_3), .DRAM_dataIn(MEM_3
    ), .DRAM_dataOut(DR_3),

```

```

89             .IROM_addr(PC_3), .DRAM_addr(AR_3), .memREAD(
memREAD_3), .memWRITE(memWE_3), .iROMREAD(iROMREAD_3), .coreS(coreS_3),
90             .imemAV(imemAV3), .memAV(memAV3), .MEM_ID(
MEMID_CORE3), .coreID(COREID_3));
91
92 core #(.WIDTH(WIDTH)) CORE_3 (.Clk(clk), .IROM_dataIn(INS_4), .DRAM_dataIn(MEM_4
),.DRAM_dataOut(DR_4),
93             .IROM_addr(PC_4), .DRAM_addr(AR_4), .memREAD(
memREAD_4), .memWRITE(memWE_4), .iROMREAD(iROMREAD_4), .coreS(coreS_4),
94             .imemAV(imemAV4), .memAV(memAV4), .MEM_ID(
MEMID_CORE4), .coreID(COREID_4));
95
96 core #(.WIDTH(WIDTH)) CORE_4 (.Clk(clk), .IROM_dataIn(INS_5), .DRAM_dataIn(MEM_5
),.DRAM_dataOut(DR_5),
97             .IROM_addr(PC_5), .DRAM_addr(AR_5), .memREAD(
memREAD_5), .memWRITE(memWE_5), .iROMREAD(iROMREAD_5), .coreS(coreS_5),
98             .imemAV(imemAV5), .memAV(memAV5), .MEM_ID(
MEMID_CORE5), .coreID(COREID_5));
99
100 core #(.WIDTH(WIDTH)) CORE_5 (.Clk(clk), .IROM_dataIn(INS_6), .DRAM_dataIn(MEM_6
),.DRAM_dataOut(DR_6),
101             .IROM_addr(PC_6), .DRAM_addr(AR_6), .memREAD(
memREAD_6), .memWRITE(memWE_6), .iROMREAD(iROMREAD_6), .coreS(coreS_6),
102             .imemAV(imemAV6), .memAV(memAV6), .MEM_ID(
MEMID_CORE6), .coreID(COREID_6));
103
104 core #(.WIDTH(WIDTH)) CORE_6 (.Clk(clk), .IROM_dataIn(INS_7), .DRAM_dataIn(MEM_7
),.DRAM_dataOut(DR_7),
105             .IROM_addr(PC_7), .DRAM_addr(AR_7), .memREAD(
memREAD_7), .memWRITE(memWE_7), .iROMREAD(iROMREAD_7), .coreS(coreS_7),
106             .imemAV(imemAV4), .memAV(memAV7), .MEM_ID(
MEMID_CORE7), .coreID(COREID_7));
107
108 core #(.WIDTH(WIDTH)) CORE_7 (.Clk(clk), .IROM_dataIn(INS_8), .DRAM_dataIn(MEM_8
),.DRAM_dataOut(DR_8),
109             .IROM_addr(PC_8), .DRAM_addr(AR_8), .memREAD(
memREAD_8), .memWRITE(memWE_8), .iROMREAD(iROMREAD_8), .coreS(coreS_8),
110             .imemAV(imemAV8), .memAV(memAV8), .MEM_ID(
MEMID_CORE8), .coreID(COREID_8));
111 assign proc_state = (coreS_1 && coreS_2 && coreS_3 && coreS_4 && coreS_5 &&
coreS_6 && coreS_7 && coreS_8);
112
113 endmodule

```

10.2 Core

```

1  'include "proc_param.v"
2
3  module core
4
5  #(parameter WIDTH = 8)(
6      input Clk,
7      input [WIDTH-1:0] IROM_dataIn, // --> IR or -->
      AR // --> DR
8      input [WIDTH-1:0] DRAM_dataIn, // --> DR
9      input [WIDTH-1:0] MEM_ID,
10     input [2:0] coreID,
11     input imemAV, memAV,
12     output [WIDTH-1:0] DRAM_dataOut, // from DR
13     output [WIDTH-1:0] IROM_addr, // PC
14     output [WIDTH-1:0] DRAM_addr, // AR
15     output wire memREAD, memWRITE, iROMREAD, coreS
16 );
17

```

```

18 wire [14:0] wEN;
19 wire [5:0] INC;
20 wire [4:0] RST;
21 wire [2:0] compMUX;
22 wire [3:0] aluOP;
23 wire zFlag;
24 wire selAR;
25 wire [4:0] busMUX;
26 wire coreINC_AR;
27
28 wire [WIDTH-1:0] INS; // instruction
    from iROM
29 wire [WIDTH-1:0] COMP_IN1;
30 wire [WIDTH-1:0] COMP_IN2;
31 wire [WIDTH-1:0] PC_OUT;
32 wire [WIDTH-1:0] AR_OUT;
33 wire [WIDTH-1:0] DR_OUT;
34 wire [WIDTH-1:0] RP_OUT;
35 wire [WIDTH-1:0] RT_OUT;
36 wire [WIDTH-1:0] RT4_OUT;
37 wire [WIDTH-1:0] RR_OUT;
38 wire [WIDTH-1:0] RM1_OUT;
39 wire [WIDTH-1:0] RK1_OUT;
40 wire [WIDTH-1:0] RN1_OUT;
41 wire [WIDTH-1:0] RM2_OUT;
42 wire [WIDTH-1:0] RK2_OUT;
43 wire [WIDTH-1:0] RN2_OUT;
44 wire [WIDTH-1:0] C1_OUT;
45 wire [WIDTH-1:0] C2_OUT;
46 wire [WIDTH-1:0] C3_OUT;
47 wire [WIDTH-1:0] AC_OUT;
48 wire [WIDTH-1:0] ALU_OUT;
49 wire [WIDTH-1:0] BUSMUX_OUT;
50 //PC 15 5
51 PC #(.WIDTH(WIDTH)) PC (.Clk(Clk), .WEN(wEN['PC_W']), .INC(INC['PC_INC']), .BusOut
    (AR_OUT), .dout(PC_OUT));
52 //IR 14
53 Reg_module_W #(.WIDTH(WIDTH)) IR (.Clk(Clk), .WEN(wEN['IR_W']), .BusOut(
    IROM_dataIn), .dout(INS));
54 //AR 13
55 AR #(.WIDTH(WIDTH)) AR (.Clk(Clk), .WEN(wEN['AR_W']), .BusOut(BUSMUX_OUT), .IOOut(
    IROM_dataIn), .selAR(selAR), .dout(AR_OUT), .coreID(coreID), .coreINC_AR(
    coreINC_AR));
56 //DR 12
57 Reg_module_W #(.WIDTH(WIDTH)) DR (.Clk(Clk), .WEN(wEN['DR_W']), .BusOut(
    BUSMUX_OUT), .dout(DR_OUT));
58 //RP 11
59 Reg_module_W #(.WIDTH(WIDTH)) RP (.Clk(Clk), .WEN(wEN['RP_W']), .BusOut(
    BUSMUX_OUT), .dout(RP_OUT));
60 //RT 10
61 Reg_module_W #(.WIDTH(WIDTH)) RT (.Clk(Clk), .WEN(wEN['RT_W']), .RST(RST['RT_RST
    ]), .BusOut(BUSMUX_OUT), .dout(RT_OUT));
62 //RT4 16
63 Reg_module_W #(.WIDTH(WIDTH)) RT4 (.Clk(Clk), .WEN(wEN['RT4_W']), .BusOut(
    BUSMUX_OUT), .dout(RT4_OUT));
64 //Rr 16
65 Reg_module_W #(.WIDTH(WIDTH)) RR (.Clk(Clk), .WEN(wEN['RR_W']), .BusOut(
    BUSMUX_OUT), .dout(RR_OUT));
66 //RM1 9
67 Reg_module_W #(.WIDTH(WIDTH)) RM1 (.Clk(Clk), .WEN(wEN['RM1_W']), .BusOut(
    BUSMUX_OUT), .dout(RM1_OUT));
68 //RK1 8
69 Reg_module_W #(.WIDTH(WIDTH)) RK1 (.Clk(Clk), .WEN(wEN['RK1_W']), .BusOut(

```

```

        BUSMUX_OUT), .dout(RK1_OUT));
70 //RN1 7
71 Reg_module_W #(.WIDTH(WIDTH)) RN1 (.Clk(Clk), .WEN(wEN['RN1_W']), .BusOut(
        BUSMUX_OUT), .dout(RN1_OUT));
72 //RM2
73 Reg_module_RI #(.WIDTH(WIDTH)) RM2 (.Clk(Clk), .RST(RST['RM2_RST']), .INC(INC[
        'RM2_INC']), .BusOut(BUSMUX_OUT), .dout(RM2_OUT));
74 //RK2
75 Reg_module_RI #(.WIDTH(WIDTH)) RK2 (.Clk(Clk), .RST(RST['RK2_RST']), .INC(INC[
        'RK2_INC']), .BusOut(BUSMUX_OUT), .dout(RK2_OUT));
76 //RN2
77 Reg_module_RI #(.WIDTH(WIDTH)) RN2 (.Clk(Clk), .RST(RST['RN2_RST']), .INC(INC[
        'RN2_INC']), .BusOut(BUSMUX_OUT), .dout(RN2_OUT));
78 //RC1 3
79 Reg_module_W #(.WIDTH(WIDTH)) RC1 (.Clk(Clk), .WEN(wEN['RC1_W']), .BusOut(
        BUSMUX_OUT), .dout(C1_OUT));
80 //RC2 2 1
81 Reg_module_WI #(.WIDTH(WIDTH)) RC2 (.Clk(Clk), .WEN(wEN['RC2_W']), .INC(INC[
        'RC2_INC']), .BusOut(BUSMUX_OUT), .dout(C2_OUT));
82 //RC3 1 0
83 Reg_module_WI #(.WIDTH(WIDTH)) RC3 (.Clk(Clk), .WEN(wEN['RC3_W']), .INC(INC[
        'RC3_INC']), .BusOut(BUSMUX_OUT), .dout(C3_OUT));
84 //AC 0
85 Reg_module_RW #(.WIDTH(WIDTH)) AC (.Clk(Clk), .WEN(wEN['AC_W']), .RST(RST['AC_RST
        ]), .BusOut(ALU_OUT), .dout(AC_OUT));
86
87 //ALU
88 Alu #(.WIDTH(WIDTH)) ALU (.AC(AC_OUT), .BusOut(BUSMUX_OUT), .result_ac(ALU_OUT),
        .ALU_OP(aluOP), .MEM_ID(MEM_ID));
89
90 mux_3to1_8bit COMPMUX1 (.mux_inN(RN1_OUT), .mux_inK(RK1_OUT), .mux_inM(RM1_OUT)
        , .mux_sel(compMUX), .mux_out(COMP_IN1));
91 mux_3to1_8bit COMPMUX2 (.mux_inN(RN2_OUT), .mux_inK(RK2_OUT), .mux_inM(RM2_OUT)
        , .mux_sel(compMUX), .mux_out(COMP_IN2));
92 Comp #(.WIDTH(WIDTH)) COMP (.R1(COMP_IN1), .R2(COMP_IN2), .z(zFlag));
93
94 Bus_mux #(.WIDTH(WIDTH)) BUSMUX(.MEM(DRAM_dataIn), .AR(AR_OUT), .DR(DR_OUT), .RP
        (RP_OUT), .RT(RT_OUT), .RM1(RM1_OUT), .RK1(RK1_OUT), .RN1(RN1_OUT), .RM2(
        RM2_OUT), .RK2(RK2_OUT), .RN2(RN2_OUT), .C1(C1_OUT), .C2(C2_OUT), .C3(C3_OUT
        ), .AC(AC_OUT), .mux_sel(busMUX), .Bus_select(BUSMUX_OUT), .RR(RR_OUT), .RT4
        (RT4_OUT));
95 //CU
96 controlunit #(.WIDTH(WIDTH)) CU (.Clk(Clk), .z(zFlag), .INS(INS), .iROMREAD(
        iROMREAD), .memREAD(memREAD), .memWRITE(memWRITE), .wEN(wEN), .selAR(selAR),
        .busMUX(busMUX), .INC(INC), .RST(RST), .compMUX(compMUX), .aluOP(aluOP), .
        coreS(coreS), .memAV(memAV), .imemAV(imemAV), .coreINC_AR(coreINC_AR));
97
98
99 assign IROM_addr = PC_OUT;
100 assign DRAM_dataOut = DR_OUT;
101 assign DRAM_addr = AR_OUT;
102 endmodule

```

10.3 Data Memory Controller

```

1 module dmem_controller#(parameter WIDTH=8)(
2     input Clk,
3     input coreS_1, coreS_2, coreS_3, coreS_4, coreS_5, coreS_6, coreS_7, coreS_8,
4     input [WIDTH-1:0] AR_1, AR_2, AR_3, AR_4, AR_5, AR_6, AR_7, AR_8,
5     input [WIDTH-1:0] DR_1, DR_2, DR_3, DR_4, DR_5, DR_6, DR_7, DR_8,
6     input [WIDTH-1:0] MEM, //from DRAM
7     input memREAD_1, memREAD_2, memREAD_3, memREAD_4, memREAD_5, memREAD_6,
        memREAD_7, memREAD_8,
8     input memWE_1, memWE_2, memWE_3, memWE_4, memWE_5, memWE_6, memWE_7, memWE_8,

```

```

9      output reg rEN,
10     output reg wEN,
11     output reg [WIDTH-1:0] MEM_1, MEM_2, MEM_3, MEM_4, MEM_5, MEM_6, MEM_7, MEM_8
12     ,
13     output reg [WIDTH-1:0] addr, //to DRAM
14     output reg [WIDTH-1:0] DR_OUT,
15     output reg memAV1, memAV2, memAV3, memAV4, memAV5, memAV6, memAV7, memAV8
16     //to cores
17 );
18
19 localparam NORM = 5'b00000;
20 localparam NORMEND = 5'b00001;
21
22 localparam AR_1_2 = 5'b00010;
23 localparam AR_2_1 = 5'b00011;
24 localparam AR_2_2 = 5'b00100;
25 localparam AR_3_1 = 5'b00101;
26 localparam AR_3_2 = 5'b00110;
27 localparam AR_4_1 = 5'b00111;
28 localparam AR_4_2 = 5'b01000;
29 localparam AR_5_1 = 5'b01001;
30 localparam AR_5_2 = 5'b01010;
31 localparam AR_6_1 = 5'b01011;
32 localparam AR_6_2 = 5'b01100;
33 localparam AR_7_1 = 5'b01101;
34 localparam AR_7_2 = 5'b01110;
35 localparam AR_8_1 = 5'b01111;
36 localparam AR_8_2 = 5'b10000;
37
38 localparam DR_1_1 = 5'b10001;
39 localparam DR_1_2 = 5'b10010;
40 localparam DR_1_3 = 5'b10011;
41 localparam DR_1_4 = 5'b10100;
42 localparam DR_1_5 = 5'b10101;
43 localparam DR_1_6 = 5'b10110;
44 localparam DR_1_7 = 5'b10111;
45
46 reg [4:0] NEXT_STATE_DC=NORM;
47 reg [4:0] STATE_DC = NORM;
48
49 always @(posedge Clk) begin
50     STATE_DC=NEXT_STATE_DC;
51     case (STATE_DC)
52     NORM: begin
53         memAV1 <= 0;
54         memAV2 <= 0;
55         memAV3 <= 0;
56         memAV4 <= 0;
57         memAV5 <= 0;
58         memAV6 <= 0;
59         memAV7 <= 0;
60         memAV8 <= 0;
61         rEN <= 0;
62         wEN <= 0;
63         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
64             && coreS_6==0 && coreS_7==0 && coreS_8==0) begin
65             if (memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
66                 memREAD_5==1 && memREAD_6==1 && memREAD_7==1 && memREAD_8==1) begin
67                 if (AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4 && AR_4==AR_5 && AR_5
68                     ==AR_6 && AR_6==AR_7 && AR_7==AR_8) begin // Active 8 cores, same addresses
69                     rEN <= 1;
70                     addr <= AR_1;
71                     NEXT_STATE_DC <= NORMEND;
72                     memAV1 <= 1;

```

```

67         memAV2 <= 1;
68         memAV3 <= 1;
69         memAV4 <= 1;
70         memAV5 <= 1;
71         memAV6 <= 1;
72         memAV7 <= 1;
73         memAV8 <= 1;
74
75     end
76     else begin // Active 8 cores, different addresses
77         rEN <= 1;
78         addr <= AR_1;
79         NEXT_STATE_DC <= AR_1_2;
80     end
81 end
82 else if (memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1 &&
memWE_5==1 && memWE_6==1 && memWE_7==1 && memWE_8==1) begin
83     wEN <= 1;
84     addr <= AR_1;
85     DR_OUT <= DR_1;
86     memAV1 <= 0;
87     memAV2 <= 0;
88     memAV3 <= 0;
89     memAV4 <= 0;
90     memAV5 <= 0;
91     memAV6 <= 0;
92     memAV7 <= 0;
93     memAV8 <= 0;
94     NEXT_STATE_DC <= DR_1_1;
95 end
96 end
97 else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
coreS_5==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
98     if (memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
memREAD_5==1 && memREAD_6==1 && memREAD_7==1) begin
99         if (AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4 && AR_4==AR_5 &&
AR_5==AR_6 && AR_6==AR_7) begin
100             rEN <= 1;
101             addr <= AR_1;
102             NEXT_STATE_DC <= NORMEND;
103             memAV1 <= 1;
104             memAV2 <= 1;
105             memAV3 <= 1;
106             memAV4 <= 1;
107             memAV5 <= 1;
108             memAV6 <= 1;
109             memAV7 <= 1;
110         end
111     else begin
112         rEN <= 1;
113         addr <= AR_1;
114         NEXT_STATE_DC <= AR_1_2;
115     end
116 end
117 else if (memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1 &&
memWE_5==1 && memWE_6==1 && memWE_7==1) begin
118     wEN <= 1;
119     addr <= AR_1;
120     DR_OUT <= DR_1;
121     memAV1 <= 0;
122     memAV2 <= 0;
123     memAV3 <= 0;
124     memAV4 <= 0;

```



```

125         memAV5 <= 0;
126         memAV6 <= 0;
127         memAV7 <= 0;
128         NEXT_STATE_DC <= DR_1_1;
129     end
130
131 end
132
133     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
134 coreS_5==0 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
135         if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
136 memREAD_5==1 && memREAD_6==1) begin
137             if(AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4 && AR_4==AR_5 &&
138 AR_5==AR_6) begin
139                 rEN <= 1;
140                 addr <= AR_1;
141                 NEXT_STATE_DC <= NORMEND;
142                 memAV1 <= 1;
143                 memAV2 <= 1;
144                 memAV3 <= 1;
145                 memAV4 <= 1;
146                 memAV5 <= 1;
147                 memAV6 <= 1;
148             end
149             else begin
150                 rEN <= 1;
151                 addr <= AR_1;
152                 NEXT_STATE_DC <= AR_1_2;
153             end
154         end
155     end
156     else if(memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1 &&
157 memWE_5==1 && memWE_6==1) begin
158         wEN <= 1;
159         addr <= AR_1;
160         DR_OUT <= DR_1;
161         memAV1 <= 0;
162         memAV2 <= 0;
163         memAV3 <= 0;
164         memAV4 <= 0;
165         memAV5 <= 0;
166         memAV6 <= 0;
167         NEXT_STATE_DC <= DR_1_1;
168     end
169
170 end
171
172     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
173 coreS_5==0 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
174         if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
175 memREAD_5==1) begin
176             if(AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4 && AR_4==AR_5) begin
177                 rEN <= 1;
178                 addr <= AR_1;
179                 NEXT_STATE_DC <= NORMEND;
180                 memAV1 <= 1;
181                 memAV2 <= 1;
182                 memAV3 <= 1;
183                 memAV4 <= 1;
184                 memAV5 <= 1;
185             end
186             else begin
187                 rEN <= 1;
188                 addr <= AR_1;

```

```

182         NEXT_STATE_DC <= AR_1_2;
183     end
184 end
185
186     else if (memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1 &&
memWE_5==1) begin
187         wEN <= 1;
188         addr <= AR_1;
189         DR_OUT <= DR_1;
190         memAV1 <= 0;
191         memAV2 <= 0;
192         memAV3 <= 0;
193         memAV4 <= 0;
194         memAV5 <= 0;
195         NEXT_STATE_DC <= DR_1_1;
196     end
197 end
198     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
199         if (memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1)
begin
200             if (AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4) begin
201                 rEN <= 1;
202                 addr <= AR_1;
203                 NEXT_STATE_DC <= NORMEND;
204                 memAV1 <= 1;
205                 memAV2 <= 1;
206                 memAV3 <= 1;
207                 memAV4 <= 1;
208             end
209             else begin
210                 rEN <= 1;
211                 addr <= AR_1;
212                 NEXT_STATE_DC <= AR_1_2;
213             end
214         end
215     else if (memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1)
begin
216         wEN <= 1;
217         addr <= AR_1;
218         DR_OUT <= DR_1;
219         memAV1 <= 0;
220         memAV2 <= 0;
221         memAV3 <= 0;
222         memAV4 <= 0;
223         NEXT_STATE_DC <= DR_1_1;
224     end
225 end
226
227     end
228     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
229         if (memREAD_1==1 && memREAD_2==1 && memREAD_3==1) begin
230             if (AR_1==AR_2 && AR_2==AR_3) begin
231                 rEN <= 1;
232                 addr <= AR_1;
233                 NEXT_STATE_DC <= NORMEND;
234                 memAV1 <= 1;
235                 memAV2 <= 1;
236                 memAV3 <= 1;
237             end
238             else begin
239                 rEN <= 1;

```

```

240         addr <= AR_1;
241         NEXT_STATE_DC <= AR_1_2;
242     end
243 end
244
245     else if (memWE_1==1 && memWE_2==1 && memWE_3==1) begin
246         wEN <= 1;
247         addr <= AR_1;
248         DR_OUT <= DR_1;
249         memAV1 <= 0;
250         memAV2 <= 0;
251         memAV3 <= 0;
252         NEXT_STATE_DC <= DR_1_1;
253     end
254
255 end
256
257     else if (coreS_1==0 && coreS_2==0 && coreS_3==1 && coreS_4==1 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
258         if (memREAD_1==1 && memREAD_2==1) begin
259             if (AR_1==AR_2) begin
260                 rEN <= 1;
261                 addr <= AR_1;
262                 NEXT_STATE_DC <= NORMEND;
263                 memAV1 <= 1;
264                 memAV2 <= 1;
265             end
266             else begin
267                 rEN <= 1;
268                 addr <= AR_1;
269                 NEXT_STATE_DC <= AR_1_2;
270             end
271         end
272         else if (memWE_1==1 && memWE_2==1) begin
273             wEN <= 1;
274             addr <= AR_1;
275             DR_OUT <= DR_1;
276             memAV1 <= 0;
277             memAV2 <= 0;
278             NEXT_STATE_DC <= DR_1_1;
279         end
280
281     end
282
283     else if (coreS_1==0 && coreS_2==1 && coreS_3==1 && coreS_4==1 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
284         if (memREAD_1==1) begin
285             rEN <= 1;
286             addr <= AR_1;
287             NEXT_STATE_DC <= NORMEND;
288             memAV1 <= 1;
289         end
290         else if (memWE_1==1) begin
291             wEN <= 1;
292             addr <= AR_1;
293             DR_OUT <= DR_1;
294             memAV1 <= 1;
295             NEXT_STATE_DC <= NORM;
296         end
297     end
298
299 end
300 NORMEND: begin

```

```

301         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==0) begin
302             MEM_1 <= MEM;
303             MEM_2 <= MEM;
304             MEM_3 <= MEM;
305             MEM_4 <= MEM;
306             MEM_5 <= MEM;
307             MEM_6 <= MEM;
308             MEM_7 <= MEM;
309             MEM_8 <= MEM;
310             memAV1 <= 0;           // memAV = 0 in AR_1
311             memAV2 <= 0;
312             memAV3 <= 0;
313             memAV4 <= 0;
314             memAV5 <= 0;
315             memAV6 <= 0;
316             memAV7 <= 0;
317             memAV8 <= 0;
318         end
319         else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
coreS_5==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
320             MEM_1 <= MEM;
321             MEM_2 <= MEM;
322             MEM_3 <= MEM;
323             MEM_4 <= MEM;
324             MEM_5 <= MEM;
325             MEM_6 <= MEM;
326             MEM_7 <= MEM;
327             memAV1 <= 0;           // memAV = 0 in AR_1
328             memAV2 <= 0;
329             memAV3 <= 0;
330             memAV4 <= 0;
331             memAV5 <= 0;
332             memAV6 <= 0;
333             memAV7 <= 0;
334         end
335         else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
coreS_5==0 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
336             MEM_1 <= MEM;
337             MEM_2 <= MEM;
338             MEM_3 <= MEM;
339             MEM_4 <= MEM;
340             MEM_5 <= MEM;
341             MEM_6 <= MEM;
342             memAV1 <= 0;           // memAV = 0 in AR_1
343             memAV2 <= 0;
344             memAV3 <= 0;
345             memAV4 <= 0;
346             memAV5 <= 0;
347             memAV6 <= 0;
348         end
349         else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
coreS_5==0 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
350             MEM_1 <= MEM;
351             MEM_2 <= MEM;
352             MEM_3 <= MEM;
353             MEM_4 <= MEM;
354             MEM_5 <= MEM;
355             memAV1 <= 0;           // memAV = 0 in AR_1
356             memAV2 <= 0;
357             memAV3 <= 0;
358             memAV4 <= 0;
359             memAV5 <= 0;

```

```

360         end
361         else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
362             MEM_1 <= MEM;
363             MEM_2 <= MEM;
364             MEM_3 <= MEM;
365             MEM_4 <= MEM;
366             memAV1 <= 0;           // memAV = 0 in AR_1
367             memAV2 <= 0;
368             memAV3 <= 0;
369             memAV4 <= 0;
370         end
371         else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
372             MEM_1 <= MEM;
373             MEM_2 <= MEM;
374             MEM_3 <= MEM;
375             memAV1 <= 0;           // memAV = 0 in AR_1
376             memAV2 <= 0;
377             memAV3 <= 0;
378         end
379         else if (coreS_1==0 && coreS_2==0 && coreS_3==1 && coreS_4==1 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
380             MEM_1 <= MEM;
381             MEM_2 <= MEM;
382             memAV1 <= 0;           // memAV = 0 in AR_1
383             memAV2 <= 0;
384         end
385         else if (coreS_1==0 && coreS_2==1 && coreS_3==1 && coreS_4==1 &&
coreS_5==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
386             MEM_1 <= MEM;
387             memAV1 <= 0;           // memAV = 0 in AR_1
388         end
389         NEXT_STATE_DC <= NORM;
390
391     end
392
393     AR_1_2: begin
394         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==0) begin
395             MEM_1 <= MEM;
396             memAV1 <= 0;
397             memAV2 <= 0;
398             memAV3 <= 0;
399             memAV4 <= 0;
400             memAV5 <= 0;
401             memAV6 <= 0;
402             memAV7 <= 0;
403             memAV8 <= 0;
404             NEXT_STATE_DC <= AR_2_1;
405
406         end
407         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==1) begin
408             MEM_1 <= MEM;
409             memAV1 <= 0;
410             memAV2 <= 0;
411             memAV3 <= 0;
412             memAV4 <= 0;
413             memAV5 <= 0;
414             memAV6 <= 0;
415             memAV7 <= 0;
416             NEXT_STATE_DC <= AR_2_1;

```

```

417
418     end
419
420     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
421 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
422         MEM_1 <= MEM;
423         memAV1 <= 0;
424         memAV2 <= 0;
425         memAV3 <= 0;
426         memAV4 <= 0;
427         memAV5 <= 0;
428         memAV6 <= 0;
429         NEXT_STATE_DC <= AR_2_1;
430
431     end
432
433     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
434 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
435         MEM_1 <= MEM;
436         memAV1 <= 0;
437         memAV2 <= 0;
438         memAV3 <= 0;
439         memAV4 <= 0;
440         memAV5 <= 0;
441         NEXT_STATE_DC <= AR_2_1;
442
443     end
444     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==1
445 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
446         MEM_1 <= MEM;
447         memAV1 <= 0;
448         memAV2 <= 0;
449         memAV3 <= 0;
450         memAV4 <= 0;
451         NEXT_STATE_DC <= AR_2_1;
452
453     end
454     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 && coreS_5==1
455 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
456         MEM_1 <= MEM;
457         memAV1 <= 0;
458         memAV2 <= 0;
459         memAV3 <= 0;
460         NEXT_STATE_DC <= AR_2_1;
461
462     end
463     if (coreS_1==0 && coreS_2==0 && coreS_3==1 && coreS_4==1 && coreS_5==1
464 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
465         MEM_1 <= MEM;
466         memAV1 <= 0;
467         memAV2 <= 0;
468         NEXT_STATE_DC <= AR_2_1;
469
470     end
471     if (coreS_1==0 && coreS_2==1 && coreS_3==1 && coreS_4==1 && coreS_5==1
472 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
473         MEM_1 <= MEM;
474         memAV1 <= 0;
475         NEXT_STATE_DC <= NORM;
476
477     end
478 end

```

```

474 AR_2_1: begin
475     rEN <= 1;
476     addr <= AR_2;
477     NEXT_STATE_DC <= AR_2_2;
478     if (coreS_1==0 && coreS_2==0 && coreS_3==1 && coreS_4==1 && coreS_5==1
479 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
480         memAV1 <= 1;
481         memAV2 <= 1;
482     end
483 end
484
485 AR_2_2: begin
486     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
487 && coreS_6==0 && coreS_7==0 && coreS_8==0) begin
488         MEM_2 <= MEM;
489         memAV1 <= 0;
490         memAV2 <= 0;
491         memAV3 <= 0;
492         memAV4 <= 0;
493         memAV5 <= 0;
494         memAV6 <= 0;
495         memAV7 <= 0;
496         memAV8 <= 0;
497         NEXT_STATE_DC <= AR_3_1;
498     end
499
500     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
501 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
502         MEM_2 <= MEM;
503         memAV1 <= 0;
504         memAV2 <= 0;
505         memAV3 <= 0;
506         memAV4 <= 0;
507         memAV5 <= 0;
508         memAV6 <= 0;
509         memAV7 <= 0;
510         NEXT_STATE_DC <= AR_3_1;
511     end
512
513     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
514 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
515         MEM_2 <= MEM;
516         memAV1 <= 0;
517         memAV2 <= 0;
518         memAV3 <= 0;
519         memAV4 <= 0;
520         memAV5 <= 0;
521         memAV6 <= 0;
522         NEXT_STATE_DC <= AR_3_1;
523     end
524
525     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
526 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
527         MEM_2 <= MEM;
528         memAV1 <= 0;
529         memAV2 <= 0;
530         memAV3 <= 0;
531         memAV4 <= 0;
532         memAV5 <= 0;
533         NEXT_STATE_DC <= AR_3_1;
534     end
535
536     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==1
537 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin

```

```

531         MEM_2 <= MEM;
532         memAV1 <= 0;
533         memAV2 <= 0;
534         memAV3 <= 0;
535         memAV4 <= 0;
536         NEXT_STATE_DC <= AR_3_1;
537     end
538     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 && coreS_5==1
&& coreS_6==1 && coreS_7==1 && coreS_8==1) begin
539         MEM_2 <= MEM;
540         memAV1 <= 0;
541         memAV2 <= 0;
542         memAV3 <= 0;
543         NEXT_STATE_DC <= AR_3_1;
544     end
545     if (coreS_1==0 && coreS_2==0 && coreS_3==1 && coreS_4==1 && coreS_5==1
&& coreS_6==1 && coreS_7==1 && coreS_8==1) begin
546         MEM_2 <= MEM;
547         memAV1 <= 0;
548         memAV2 <= 0;
549         NEXT_STATE_DC <= NORM;
550     end
551 end
552
553
554 AR_3_1: begin
555     rEN <= 1;
556     addr <= AR_3;
557     NEXT_STATE_DC <= AR_3_2;
558     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 && coreS_5==1
&& coreS_6==1 && coreS_7==1 && coreS_8==1) begin //when 3 cores are
working
559         memAV1 <= 1;
560         memAV2 <= 1;
561         memAV3 <= 1;
562     end
563 end
564
565 AR_3_2: begin
566     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==0) begin
567         MEM_3 <= MEM;
568         memAV1 <= 0;
569         memAV2 <= 0;
570         memAV3 <= 0;
571         memAV4 <= 0;
572         memAV4 <= 0;
573         memAV5 <= 0;
574         memAV6 <= 0;
575         memAV7 <= 0;
576         memAV8 <= 0;
577         NEXT_STATE_DC <= AR_4_1;
578     end
579
580     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==1) begin
581         MEM_3 <= MEM;
582         memAV1 <= 0;
583         memAV2 <= 0;
584         memAV3 <= 0;
585         memAV4 <= 0;
586         memAV4 <= 0;
587         memAV5 <= 0;

```



```

588         memAV6 <= 0;
589         memAV7 <= 0;
590         NEXT_STATE_DC <= AR_4_1;
591     end
592
593     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
594 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
595         MEM_3 <= MEM;
596         memAV1 <= 0;
597         memAV2 <= 0;
598         memAV3 <= 0;
599         memAV4 <= 0;
600         memAV5 <= 0;
601         memAV6 <= 0;
602         NEXT_STATE_DC <= AR_4_1;
603     end
604     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
605 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
606         MEM_3 <= MEM;
607         memAV1 <= 0;
608         memAV2 <= 0;
609         memAV3 <= 0;
610         memAV4 <= 0;
611         memAV5 <= 0;
612         NEXT_STATE_DC <= AR_4_1;
613     end
614     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==1
615 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
616         MEM_3 <= MEM;
617         memAV1 <= 0;
618         memAV2 <= 0;
619         memAV3 <= 0;
620         memAV4 <= 0;
621         NEXT_STATE_DC <= AR_4_1;
622     end
623     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 && coreS_5==1
624 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
625         MEM_3 <= MEM;
626         memAV1 <= 0;
627         memAV2 <= 0;
628         memAV3 <= 0;
629         NEXT_STATE_DC <= NORM;
630     end
631
632 end
633
634 AR_4_1: begin
635     rEN <= 1;
636     addr <= AR_4;
637     NEXT_STATE_DC <= AR_4_2;
638     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==1
639 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
640         memAV1 <= 1;
641         memAV2 <= 1;
642         memAV3 <= 1;
643         memAV4 <= 1;
644     end
645 end

```

```

646
647     AR_4_2:begin
648         if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
649             && coreS_6==0 && coreS_7==0 && coreS_8==0) begin
650             MEM_4 <= MEM;
651             memAV1 <= 0;
652             memAV2 <= 0;
653             memAV3 <= 0;
654             memAV4 <= 0;
655             memAV5 <= 0;
656             memAV6 <= 0;
657             memAV7 <= 0;
658             memAV8 <= 0;
659             NEXT_STATE_DC <= AR_5_1;
660         end
661
662         if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
663             && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
664             MEM_4 <= MEM;
665             memAV1 <= 0;
666             memAV2 <= 0;
667             memAV3 <= 0;
668             memAV4 <= 0;
669             memAV5 <= 0;
670             memAV6 <= 0;
671             memAV7 <= 0;
672             NEXT_STATE_DC <= AR_5_1;
673         end
674
675         if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
676             && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
677             MEM_4 <= MEM;
678             memAV1 <= 0;
679             memAV2 <= 0;
680             memAV3 <= 0;
681             memAV4 <= 0;
682             memAV5 <= 0;
683             memAV6 <= 0;
684             NEXT_STATE_DC <= AR_5_1;
685         end
686
687         if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
688             && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
689             MEM_4 <= MEM;
690             memAV1 <= 0;
691             memAV2 <= 0;
692             memAV3 <= 0;
693             memAV4 <= 0;
694             memAV5 <= 0;
695             NEXT_STATE_DC <= AR_5_1;
696         end
697
698         if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==1
699             && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
700             MEM_4 <= MEM;
701             memAV1 <= 0;
702             memAV2 <= 0;
703             memAV3 <= 0;
704             memAV4 <= 0;
705             NEXT_STATE_DC <= NORM;
706         end
707
708     end
709
710     AR_5_1: begin
711         rEN <= 1;
712

```

```

704     addr <= AR_5;
705     NEXT_STATE_DC <= AR_5_2;
706     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==1 && coreS_7==1 && coreS_8==1) begin
707         memAV1 <= 1;
708         memAV2 <= 1;
709         memAV3 <= 1;
710         memAV4 <= 1;
711         memAV5 <= 1;
712     end
713
714 end
715
716     AR_5_2:begin
717         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==0) begin
718             MEM_5 <= MEM;
719             memAV1 <= 0;
720             memAV2 <= 0;
721             memAV3 <= 0;
722             memAV4 <= 0;
723             memAV5 <= 0;
724             memAV6 <= 0;
725             memAV7 <= 0;
726             memAV8 <= 0;
727             NEXT_STATE_DC <= AR_6_1;
728         end
729         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==1) begin
730             MEM_5 <= MEM;
731             memAV1 <= 0;
732             memAV2 <= 0;
733             memAV3 <= 0;
734             memAV4 <= 0;
735             memAV5 <= 0;
736             memAV6 <= 0;
737             memAV7 <= 0;
738             NEXT_STATE_DC <= AR_6_1;
739         end
740         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==1 && coreS_8==1) begin
741             MEM_5 <= MEM;
742             memAV1 <= 0;
743             memAV2 <= 0;
744             memAV3 <= 0;
745             memAV4 <= 0;
746             memAV5 <= 0;
747             memAV6 <= 0;
748             NEXT_STATE_DC <= AR_6_1;
749         end
750         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==1 && coreS_7==1 && coreS_8==1) begin
751             MEM_5 <= MEM;
752             memAV1 <= 0;
753             memAV2 <= 0;
754             memAV3 <= 0;
755             memAV4 <= 0;
756             memAV5 <= 0;
757             NEXT_STATE_DC <= NORM;
758         end
759     end
760
761     AR_6_1: begin

```

```

762     rEN <= 1;
763     addr <= AR_6;
764     NEXT_STATE_DC <= AR_6_2;
765     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==1 && coreS_8==1) begin
766         memAV1 <= 1;
767         memAV2 <= 1;
768         memAV3 <= 1;
769         memAV4 <= 1;
770         memAV5 <= 1;
771         memAV6 <= 1;
772     end
773
774 end
775
776 AR_6_2:begin
777     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==0) begin
778         MEM_6 <= MEM;
779         memAV1 <= 0;
780         memAV2 <= 0;
781         memAV3 <= 0;
782         memAV4 <= 0;
783         memAV5 <= 0;
784         memAV6 <= 0;
785         memAV7 <= 0;
786         memAV8 <= 0;
787         NEXT_STATE_DC <= AR_7_1;
788     end
789     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==1) begin
790         MEM_6 <= MEM;
791         memAV1 <= 0;
792         memAV2 <= 0;
793         memAV3 <= 0;
794         memAV4 <= 0;
795         memAV5 <= 0;
796         memAV6 <= 0;
797         memAV7 <= 0;
798         NEXT_STATE_DC <= AR_7_1;
799     end
800
801     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==1 && coreS_8==1) begin
802         MEM_6 <= MEM;
803         memAV1 <= 0;
804         memAV2 <= 0;
805         memAV3 <= 0;
806         memAV4 <= 0;
807         memAV5 <= 0;
808         memAV6 <= 0;
809         NEXT_STATE_DC <= NORM;
810     end
811 end
812
813 AR_7_1: begin
814     rEN <= 1;
815     addr <= AR_7;
816     NEXT_STATE_DC <= AR_7_2;
817     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==1) begin
818         memAV1 <= 1;
819         memAV2 <= 1;

```

```

820         memAV3 <= 1;
821         memAV4 <= 1;
822         memAV5 <= 1;
823         memAV6 <= 1;
824         memAV7 <= 1;
825     end
826
827 end
828
829 AR_7_2:begin
830     if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==0) begin
831         MEM_7 <= MEM;
832         memAV1 <= 0;
833         memAV2 <= 0;
834         memAV3 <= 0;
835         memAV4 <= 0;
836         memAV5 <= 0;
837         memAV6 <= 0;
838         memAV7 <= 0;
839         memAV8 <= 0;
840         NEXT_STATE_DC <= AR_8_1;
841     end
842     if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0
&& coreS_6==0 && coreS_7==0 && coreS_8==1) begin
843         MEM_7 <= MEM;
844         memAV1 <= 0;
845         memAV2 <= 0;
846         memAV3 <= 0;
847         memAV4 <= 0;
848         memAV5 <= 0;
849         memAV6 <= 0;
850         memAV7 <= 0;
851         NEXT_STATE_DC <= NORM;
852     end
853 end
854
855 AR_8_1: begin
856     rEN <= 1;
857     addr <= AR_8;
858     NEXT_STATE_DC <= AR_8_2;
859     memAV1 <= 1;
860     memAV2 <= 1;
861     memAV3 <= 1;
862     memAV4 <= 1;
863     memAV5 <= 1;
864     memAV6 <= 1;
865     memAV7 <= 1;
866     memAV8 <= 1;
867
868 end
869
870 AR_8_2:begin
871     MEM_8 <= MEM;
872     memAV1 <= 0;
873     memAV2 <= 0;
874     memAV3 <= 0;
875     memAV4 <= 0;
876     memAV5 <= 0;
877     memAV6 <= 0;
878     memAV7 <= 0;
879     memAV8 <= 0;
880     NEXT_STATE_DC <= NORM;

```

```

881
882     end
883
884
885
886
887     DR_1_1:begin
888         if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0 &&
coreS_6==0 && coreS_7==0 && coreS_8==0) begin
889             wEN <= 1;
890             addr <= AR_2;
891             DR_OUT <= DR_2;
892             memAV1 <= 0;
893             memAV2 <= 0;
894             memAV3 <= 0;
895             memAV4 <= 0;
896             memAV5 <= 0;
897             memAV6 <= 0;
898             memAV7 <= 0;
899             memAV8 <= 0;
900
901             NEXT_STATE_DC <= DR_1_2;
902         end
903
904         else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
905             wEN <= 1;
906             addr <= AR_2;
907             DR_OUT <= DR_2;
908             memAV1 <= 0;
909             memAV2 <= 0;
910             memAV3 <= 0;
911             memAV4 <= 0;
912             memAV5 <= 0;
913             memAV6 <= 0;
914             memAV7 <= 0;
915
916             NEXT_STATE_DC <= DR_1_2;
917         end
918         else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
919             wEN <= 1;
920             addr <= AR_2;
921             DR_OUT <= DR_2;
922             memAV1 <= 0;
923             memAV2 <= 0;
924             memAV3 <= 0;
925             memAV4 <= 0;
926             memAV5 <= 0;
927             memAV6 <= 0;
928             NEXT_STATE_DC <= DR_1_2;
929         end
930         else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
931             wEN <= 1;
932             addr <= AR_2;
933             DR_OUT <= DR_2;
934             memAV1 <= 0;
935             memAV2 <= 0;
936             memAV3 <= 0;
937             memAV4 <= 0;
938             memAV5 <= 0;
939             NEXT_STATE_DC <= DR_1_2;

```

```

940     end
941     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
942         wEN <= 1;
943         addr <= AR_2;
944         DR_OUT <= DR_2;
945         memAV1 <= 0;
946         memAV2 <= 0;
947         memAV3 <= 0;
948         memAV4 <= 0;
949         NEXT_STATE_DC <= DR_1_2;
950     end
951     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 && coreS_5
==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
952         wEN <= 1;
953         addr <= AR_2;
954         DR_OUT <= DR_2;
955         memAV1 <= 0;
956         memAV2 <= 0;
957         memAV3 <= 0;
958         NEXT_STATE_DC <= DR_1_2;
959     end
960     else if (coreS_1==0 && coreS_2==0 && coreS_3==1 && coreS_4==1 && coreS_5
==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
961         wEN <= 1;
962         addr <= AR_2;
963         DR_OUT <= DR_2;
964         memAV1 <= 1;
965         memAV2 <= 1;
966         NEXT_STATE_DC <= NORM;
967     end
968
969
970 end
971
972 DR_1_2:begin
973     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0 &&
coreS_6==0 && coreS_7==0 && coreS_8==0) begin
974         wEN <= 1;
975         addr <= AR_3;
976         DR_OUT <= DR_3;
977         memAV1 <= 0;
978         memAV2 <= 0;
979         memAV3 <= 0;
980         memAV4 <= 0;
981         memAV5 <= 0;
982         memAV6 <= 0;
983         memAV7 <= 0;
984         memAV8 <= 0;
985
986         NEXT_STATE_DC <= DR_1_3;
987     end
988     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
989         wEN <= 1;
990         addr <= AR_3;
991         DR_OUT <= DR_3;
992         memAV1 <= 0;
993         memAV2 <= 0;
994         memAV3 <= 0;
995         memAV4 <= 0;
996         memAV5 <= 0;
997         memAV6 <= 0;

```

```

998         memAV7 <= 0;
999
1000         NEXT_STATE_DC <= DR_1_3;
1001     end
1002     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
1003         wEN <= 1;
1004         addr <= AR_3;
1005         DR_OUT <= DR_3;
1006         memAV1 <= 0;
1007         memAV2 <= 0;
1008         memAV3 <= 0;
1009         memAV4 <= 0;
1010         memAV5 <= 0;
1011         memAV6 <= 0;
1012         NEXT_STATE_DC <= DR_1_3;
1013     end
1014     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
1015         wEN <= 1;
1016         addr <= AR_3;
1017         DR_OUT <= DR_3;
1018         memAV1 <= 0;
1019         memAV2 <= 0;
1020         memAV3 <= 0;
1021         memAV4 <= 0;
1022         memAV5 <= 0;
1023         NEXT_STATE_DC <= DR_1_3;
1024     end
1025     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
1026         wEN <= 1;
1027         addr <= AR_3;
1028         DR_OUT <= DR_3;
1029         memAV1 <= 0;
1030         memAV2 <= 0;
1031         memAV3 <= 0;
1032         memAV4 <= 0;
1033         memAV5 <= 0;
1034         NEXT_STATE_DC <= DR_1_3;
1035     end
1036     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
1037         wEN <= 1;
1038         addr <= AR_3;
1039         DR_OUT <= DR_3;
1040         memAV1 <= 0;
1041         memAV2 <= 0;
1042         memAV3 <= 0;
1043         memAV4 <= 0;
1044         NEXT_STATE_DC <= DR_1_3;
1045     end
1046     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==1 && coreS_5
==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
1047         wEN <= 1;
1048         addr <= AR_3;
1049         DR_OUT <= DR_3;
1050         memAV1 <= 1;
1051         memAV2 <= 1;
1052         memAV3 <= 1;
1053         NEXT_STATE_DC <= NORM;
1054     end
1055 end

```



```

1056
1057     DR_1_3:begin
1058         if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0 &&
coreS_6==0 && coreS_7==0 && coreS_8==0) begin
1059             wEN <= 1;
1060             addr <= AR_4;
1061             DR_OUT <= DR_4;
1062             memAV1 <= 0;
1063             memAV2 <= 0;
1064             memAV3 <= 0;
1065             memAV4 <= 0;
1066             memAV5 <= 0;
1067             memAV6 <= 0;
1068             memAV7 <= 0;
1069             memAV8 <= 0;
1070
1071             NEXT_STATE_DC <= DR_1_4;
1072         end
1073         else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
1074             wEN <= 1;
1075             addr <= AR_4;
1076             DR_OUT <= DR_4;
1077             memAV1 <= 0;
1078             memAV2 <= 0;
1079             memAV3 <= 0;
1080             memAV4 <= 0;
1081             memAV5 <= 0;
1082             memAV6 <= 0;
1083             memAV7 <= 0;
1084             NEXT_STATE_DC <= DR_1_4;
1085         end
1086         else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
1087             wEN <= 1;
1088             addr <= AR_4;
1089             DR_OUT <= DR_4;
1090             memAV1 <= 0;
1091             memAV2 <= 0;
1092             memAV3 <= 0;
1093             memAV4 <= 0;
1094             memAV5 <= 0;
1095             memAV6 <= 0;
1096             NEXT_STATE_DC <= DR_1_4;
1097         end
1098         else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
1099             wEN <= 1;
1100             addr <= AR_4;
1101             DR_OUT <= DR_4;
1102             memAV1 <= 0;
1103             memAV2 <= 0;
1104             memAV3 <= 0;
1105             memAV4 <= 0;
1106             memAV5 <= 0;
1107             NEXT_STATE_DC <= DR_1_4;
1108         end
1109         else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==1 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
1110             wEN <= 1;
1111             addr <= AR_4;
1112             DR_OUT <= DR_4;
1113             memAV1 <= 1;

```

```

1114         memAV2 <= 1;
1115         memAV3 <= 1;
1116         memAV4 <= 1;
1117         NEXT_STATE_DC <= NORM;
1118     end
1119 end
1120
1121 DR_1_4:begin
1122     if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0 &&
coreS_6==0 && coreS_7==0 && coreS_8==0) begin
1123         wEN <= 1;
1124         addr <= AR_5;
1125         DR_OUT <= DR_5;
1126         memAV1 <= 0;
1127         memAV2 <= 0;
1128         memAV3 <= 0;
1129         memAV4 <= 0;
1130         memAV5 <= 0;
1131         memAV6 <= 0;
1132         memAV7 <= 0;
1133         memAV8 <= 0;
1134
1135         NEXT_STATE_DC <= DR_1_5;
1136     end
1137     else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
1138         wEN <= 1;
1139         addr <= AR_5;
1140         DR_OUT <= DR_5;
1141         memAV1 <= 0;
1142         memAV2 <= 0;
1143         memAV3 <= 0;
1144         memAV4 <= 0;
1145         memAV5 <= 0;
1146         memAV6 <= 0;
1147         memAV7 <= 0;
1148         NEXT_STATE_DC <= DR_1_5;
1149     end
1150     else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
1151         wEN <= 1;
1152         addr <= AR_5;
1153         DR_OUT <= DR_5;
1154         memAV1 <= 0;
1155         memAV2 <= 0;
1156         memAV3 <= 0;
1157         memAV4 <= 0;
1158         memAV5 <= 0;
1159         memAV6 <= 0;
1160         NEXT_STATE_DC <= DR_1_5;
1161     end
1162     else if(coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==1 && coreS_7==1 && coreS_8==1) begin
1163         wEN <= 1;
1164         addr <= AR_5;
1165         DR_OUT <= DR_5;
1166         memAV1 <= 1;
1167         memAV2 <= 1;
1168         memAV3 <= 1;
1169         memAV4 <= 1;
1170         memAV5 <= 1;
1171         NEXT_STATE_DC <= NORM;
1172     end

```

```

1173     end
1174 DR_1_5:begin
1175     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0 &&
1176         coreS_6==0 && coreS_7==0 && coreS_8==0) begin
1177         wEN <= 1;
1178         addr <= AR_6;
1179         DR_OUT <= DR_6;
1180         memAV1 <= 0;
1181         memAV2 <= 0;
1182         memAV3 <= 0;
1183         memAV4 <= 0;
1184         memAV5 <= 0;
1185         memAV6 <= 0;
1186         memAV7 <= 0;
1187         memAV8 <= 0;
1188
1189         NEXT_STATE_DC <= DR_1_6;
1190     end
1191     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
1192         ==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
1193         wEN <= 1;
1194         addr <= AR_6;
1195         DR_OUT <= DR_6;
1196         memAV1 <= 0;
1197         memAV2 <= 0;
1198         memAV3 <= 0;
1199         memAV4 <= 0;
1200         memAV5 <= 0;
1201         memAV6 <= 0;
1202         memAV7 <= 0;
1203         NEXT_STATE_DC <= DR_1_6;
1204     end
1205     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
1206         ==0 && coreS_6==0 && coreS_7==1 && coreS_8==1) begin
1207         wEN <= 1;
1208         addr <= AR_6;
1209         DR_OUT <= DR_6;
1210         memAV1 <= 1;
1211         memAV2 <= 1;
1212         memAV3 <= 1;
1213         memAV4 <= 1;
1214         memAV5 <= 1;
1215         memAV6 <= 1;
1216         NEXT_STATE_DC <= NORM;
1217     end
1218 end
1219 DR_1_6:begin
1220     if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0 &&
1221         coreS_6==0 && coreS_7==0 && coreS_8==0) begin
1222         wEN <= 1;
1223         addr <= AR_7;
1224         DR_OUT <= DR_7;
1225         memAV1 <= 0;
1226         memAV2 <= 0;
1227         memAV3 <= 0;
1228         memAV4 <= 0;
1229         memAV5 <= 0;
1230         memAV6 <= 0;
1231         memAV7 <= 0;
1232         memAV8 <= 0;
1233
1234         NEXT_STATE_DC <= DR_1_7;

```

```

1232     end
1233     else if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5
==0 && coreS_6==0 && coreS_7==0 && coreS_8==1) begin
1234         wEN <= 1;
1235         addr <= AR_7;
1236         DR_OUT <= DR_7;
1237         memAV1 <= 1;
1238         memAV2 <= 1;
1239         memAV3 <= 1;
1240         memAV4 <= 1;
1241         memAV5 <= 1;
1242         memAV6 <= 1;
1243         memAV7 <= 1;
1244         NEXT_STATE_DC <= NORM;
1245     end
1246
1247     end
1248     DR_1_7:begin
1249         if (coreS_1==0 && coreS_2==0 && coreS_3==0 && coreS_4==0 && coreS_5==0 &&
coreS_6==0 && coreS_7==0 && coreS_8==0) begin
1250             wEN <= 1;
1251             addr <= AR_8;
1252             DR_OUT <= DR_8;
1253             memAV1 <= 1;
1254             memAV2 <= 1;
1255             memAV3 <= 1;
1256             memAV4 <= 1;
1257             memAV5 <= 1;
1258             memAV6 <= 1;
1259             memAV7 <= 1;
1260             memAV8 <= 1;
1261
1262             NEXT_STATE_DC <= NORM;
1263         end
1264     end
1265
1266
1267
1268     endcase
1269
1270 end
1271
1272
1273
1274 endmodule

```

10.4 Data Memory

```

1  // Quartus Prime Verilog Template
2  // Single port RAM with single read/write address
3
4  module data_mem
5  #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=8)
6  (
7      input [(DATA_WIDTH-1):0] data,
8      input [(ADDR_WIDTH-1):0] addr,
9      input wEn, clk, rEn,
10     output [(DATA_WIDTH-1):0] mem_out
11 );
12
13     // Declare the RAM variable
14     reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
15
16     // Variable to hold the registered read address

```

```

17     reg [ADDR_WIDTH-1:0] addr_reg;
18
19     //outfile
20     integer outfile;
21
22     initial begin
23         $readmemh("D:\\Academic\\ACA\\SEM5 TRONIC ACA\\SEMESTER 5\\CSD\\
FPGA\\00 - Git\\fpga-quartus\\8_CORE\\MULTI_CORE_edit\\data_mem.txt",ram);
24     end
25
26     always @ (negedge clk)
27     begin
28         // Write
29         if (wEn) begin
30             ram[addr] <= data;
31             addr_reg <= addr;
32         end
33         if (rEn) begin
34             addr_reg <= addr;
35         end
36         $writememh("D:\\Academic\\ACA\\SEM5 TRONIC ACA\\SEMESTER 5\\CSD
\\FPGA\\00 - Git\\fpga-quartus\\8_CORE\\MULTI_CORE_edit\\result.txt",ram);
37     end
38
39     assign mem_out = ram[addr_reg];
40
41 endmodule

```

10.5 Instruction Memory Controller

```

1 module imem_controller#(parameter WIDTH=8)(
2     input Clk,
3     input iROMREAD_1, iROMREAD_2, iROMREAD_3, iROMREAD_4, iROMREAD_5, iROMREAD_6
, iROMREAD_7, iROMREAD_8, //rEn signals from each core
4     input coreS_1, coreS_2, coreS_3, coreS_4, coreS_5, coreS_6, coreS_7, coreS_8,
//core states from each core
5     input [WIDTH-1:0] PC_1, PC_2, PC_3, PC_4, PC_5, PC_6, PC_7, PC_8,
//addresses from each core
6     input [WIDTH-1:0] INS, //
Instruction from IROM
7     output reg rEN, //rEn
signal to IROM
8     output reg [WIDTH-1:0] PC_OUT, //read
address to IROM
9     output reg [WIDTH-1:0] INS_1, INS_2, INS_3, INS_4, INS_5, INS_6, INS_7, INS_8,
//read instructions to cores
10    output reg imemAV1, imemAV2, imemAV3, imemAV4, imemAV5, imemAV6, imemAV7,
imemAV8 //IROM read state signal to each core
11 );
12 localparam NORMI = 3'b000;
13 localparam NORMENDI = 3'b001;
14
15 reg [2:0] NEXT_STATE_IC=NORMI;
16 reg [2:0] STATE_IC=NORMI;
17
18
19
20 always @(negedge Clk) begin
21     STATE_IC = NEXT_STATE_IC;
22     case (STATE_IC)
23     NORMI:
24         if ((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) && (
coreS_5==0) && (coreS_6==0) && (coreS_7==0) && (coreS_8==0)) begin
25             if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1

```

```

26      && iROMREAD_5==1 && iROMREAD_6==1 && iROMREAD_7==1 && iROMREAD_8==1) begin
27          rEN <= 1;
28          PC_OUT <= PC_1;
29          NEXT_STATE_IC <= NORMENDI;
30          imemAV1 <= 1;
31          imemAV2 <= 1;
32          imemAV3 <= 1;
33          imemAV4 <= 1;
34          imemAV5 <= 1;
35          imemAV6 <= 1;
36          imemAV7 <= 1;
37          imemAV8 <= 1;
38      end
39      else begin
40          rEN <= 0;
41          imemAV1 <= 0;
42          imemAV2 <= 0;
43          imemAV3 <= 0;
44          imemAV4 <= 0;
45          imemAV5 <= 0;
46          imemAV6 <= 0;
47          imemAV7 <= 0;
48          imemAV8 <= 0;
49          NEXT_STATE_IC <= NORMI;
50      end
51      end
52      else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) &&
53      (coreS_5==0) && (coreS_6==0) && (coreS_7==0) && (coreS_8==1)) begin
54          if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1
55          && iROMREAD_5==1 && iROMREAD_6==1 && iROMREAD_7==1) begin
56              rEN <= 1;
57              PC_OUT <= PC_1;
58              NEXT_STATE_IC <= NORMENDI;
59              imemAV1 <= 1;
60              imemAV2 <= 1;
61              imemAV3 <= 1;
62              imemAV4 <= 1;
63              imemAV5 <= 1;
64              imemAV6 <= 1;
65              imemAV7 <= 1;
66          end
67          else begin
68              rEN <= 0;
69              imemAV1 <= 0;
70              imemAV2 <= 0;
71              imemAV3 <= 0;
72              imemAV4 <= 0;
73              imemAV5 <= 0;
74              imemAV6 <= 0;
75              imemAV7 <= 0;
76              NEXT_STATE_IC <= NORMI;
77          end
78      end
79      end
80      else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) &&
81      (coreS_5==0) && (coreS_6==0) && (coreS_7==1) && (coreS_8==1)) begin
82          if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1
83          && iROMREAD_5==1 && iROMREAD_6==1) begin
84              rEN <= 1;
85              PC_OUT <= PC_1;

```

```

84         NEXT_STATE_IC <= NORMENDI;
85         imemAV1 <= 1;
86         imemAV2 <= 1;
87         imemAV3 <= 1;
88         imemAV4 <= 1;
89         imemAV5 <= 1;
90         imemAV6 <= 1;
91
92     end
93     else begin
94         rEN <= 0;
95         imemAV1 <= 0;
96         imemAV2 <= 0;
97         imemAV3 <= 0;
98         imemAV4 <= 0;
99         imemAV5 <= 0;
100        imemAV6 <= 0;
101
102        NEXT_STATE_IC <= NORMI;
103
104    end
105
106 end
107
108     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) &&
109 (coreS_5==0) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
110         if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1
111 && iROMREAD_5==1) begin
112             rEN <= 1;
113             PC_OUT <= PC_1;
114             NEXT_STATE_IC <= NORMENDI;
115             imemAV1 <= 1;
116             imemAV2 <= 1;
117             imemAV3 <= 1;
118             imemAV4 <= 1;
119             imemAV5 <= 1;
120
121         end
122     else begin
123         rEN <= 0;
124         imemAV1 <= 0;
125         imemAV2 <= 0;
126         imemAV3 <= 0;
127         imemAV4 <= 0;
128         imemAV5 <= 0;
129         NEXT_STATE_IC <= NORMI;
130
131     end
132
133 end
134
135     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) &&
136 (coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
137         if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1)
138 begin
139             rEN <= 1;
140             PC_OUT <= PC_1;
141             NEXT_STATE_IC <= NORMENDI;
142             imemAV1 <= 1;
143             imemAV2 <= 1;
144             imemAV3 <= 1;
145             imemAV4 <= 1;

```

```

143         end
144     else begin
145         rEN <= 0;
146         imemAV1 <= 0;
147         imemAV2 <= 0;
148         imemAV3 <= 0;
149         imemAV4 <= 0;
150         NEXT_STATE_IC <= NORMI;
151     end
152 end
153
154 end
155
156     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==1) &&
157 (coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
158         if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1) begin
159             rEN <= 1;
160             PC_OUT <= PC_1;
161             NEXT_STATE_IC <= NORMENDI;
162             imemAV1 <= 1;
163             imemAV2 <= 1;
164             imemAV3 <= 1;
165         end
166     else begin
167         rEN <= 0;
168         imemAV1 <= 0;
169         imemAV2 <= 0;
170         imemAV3 <= 0;
171         NEXT_STATE_IC <= NORMI;
172     end
173 end
174
175 end
176
177     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==1) && (coreS_4==1) &&
178 (coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
179         if(iROMREAD_1==1 && iROMREAD_2==1) begin
180             rEN <= 1;
181             PC_OUT <= PC_1;
182             NEXT_STATE_IC <= NORMENDI;
183             imemAV1 <= 1;
184             imemAV2 <= 1;
185         end
186     else begin
187         rEN <= 0;
188         imemAV1 <= 0;
189         imemAV2 <= 0;
190         NEXT_STATE_IC <= NORMI;
191     end
192 end
193
194 end
195
196 end
197
198     else if((coreS_1==0) && (coreS_2==1) && (coreS_3==1) && (coreS_4==1) &&
199 (coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
200         if(iROMREAD_1==1) begin
201             rEN <= 1;
202             PC_OUT <= PC_1;

```



```

203         NEXT_STATE_IC <= NORMENDI;
204         imemAV1 <= 1;
205
206
207
208     end
209     else begin
210         rEN <= 0;
211         imemAV1 <= 0;
212         NEXT_STATE_IC <= NORMI;
213
214     end
215
216 end
217
218
219 NORMENDI:
220     if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) && (
221 coreS_5==0) && (coreS_6==0) && (coreS_7==0) && (coreS_8==0)) begin
222         INS_1 <= INS;
223         INS_2 <= INS;
224         INS_3 <= INS;
225         INS_4 <= INS;
226         INS_5 <= INS;
227         INS_6 <= INS;
228         INS_7 <= INS;
229         INS_8 <= INS;
230         imemAV1 <= 1;
231         imemAV2 <= 1;
232         imemAV3 <= 1;
233         imemAV4 <= 1;
234         imemAV5 <= 1;
235         imemAV6 <= 1;
236         imemAV7 <= 1;
237         imemAV8 <= 1;
238         NEXT_STATE_IC <= NORMI;
239     end
240
241     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) && (
242 coreS_5==0) && (coreS_6==0) && (coreS_7==0) && (coreS_8==1)) begin
243         INS_1 <= INS;
244         INS_2 <= INS;
245         INS_3 <= INS;
246         INS_4 <= INS;
247         INS_5 <= INS;
248         INS_6 <= INS;
249         INS_7 <= INS;
250         imemAV1 <= 1;
251         imemAV2 <= 1;
252         imemAV3 <= 1;
253         imemAV4 <= 1;
254         imemAV5 <= 1;
255         imemAV6 <= 1;
256         imemAV7 <= 1;
257         NEXT_STATE_IC <= NORMI;
258     end
259
260     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) && (
261 coreS_5==0) && (coreS_6==0) && (coreS_7==1) && (coreS_8==1)) begin
262         INS_1 <= INS;
263         INS_2 <= INS;
264         INS_3 <= INS;
265         INS_4 <= INS;

```

```

263     INS_5 <= INS;
264     INS_6 <= INS;
265     imemAV1 <= 1;
266     imemAV2 <= 1;
267     imemAV3 <= 1;
268     imemAV4 <= 1;
269     imemAV5 <= 1;
270     imemAV6 <= 1;
271     NEXT_STATE_IC <= NORMI;
272 end
273
274     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) && (
coreS_5==0) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
275         INS_1 <= INS;
276         INS_2 <= INS;
277         INS_3 <= INS;
278         INS_4 <= INS;
279         INS_5 <= INS;
280         imemAV1 <= 1;
281         imemAV2 <= 1;
282         imemAV3 <= 1;
283         imemAV4 <= 1;
284         imemAV5 <= 1;
285         NEXT_STATE_IC <= NORMI;
286     end
287
288     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==0) && (
coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
289         INS_1 <= INS;
290         INS_2 <= INS;
291         INS_3 <= INS;
292         INS_4 <= INS;
293         imemAV1 <= 1;
294         imemAV2 <= 1;
295         imemAV3 <= 1;
296         imemAV4 <= 1;
297         NEXT_STATE_IC <= NORMI;
298     end
299
300     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==0) && (coreS_4==1) && (
coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
301         INS_1 <= INS;
302         INS_2 <= INS;
303         INS_3 <= INS;
304         imemAV1 <= 1;
305         imemAV2 <= 1;
306         imemAV3 <= 1;
307         NEXT_STATE_IC <= NORMI;
308     end
309
310     else if((coreS_1==0) && (coreS_2==0) && (coreS_3==1) && (coreS_4==1) && (
coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
311         INS_1 <= INS;
312         INS_2 <= INS;
313         imemAV1 <= 1;
314         imemAV2 <= 1;
315         NEXT_STATE_IC <= NORMI;
316     end
317
318     else if((coreS_1==0) && (coreS_2==1) && (coreS_3==1) && (coreS_4==1) && (
coreS_5==1) && (coreS_6==1) && (coreS_7==1) && (coreS_8==1)) begin
319         INS_1 <= INS;
320         imemAV1 <= 1;

```

```

321         NEXT_STATE_IC <= NORMI;
322     end
323
324
325
326
327
328     endcase
329
330 end
331
332 endmodule

```

10.6 Instruction Memory

```

1 module ins_mem
2     #(parameter ADDR_WIDTH=8, parameter INS_WIDTH=9)
3     (
4         input    [(ADDR_WIDTH-1):0]    PC_address,        // Input Address
5         input    clk, rEn,
6         output   reg [(INS_WIDTH-1):0]  instruction        // Instruction at memory
7         location Address
8     );
9
10    reg [(INS_WIDTH-1):0] mem[2**ADDR_WIDTH-1:0]; // 2**ADDR_WIDTH
11
12    initial
13    begin
14        $readmemh("D:\\Academic\\ACA\\SEM5 TRONIC ACA\\SEMESTER 5\\CSD\\
15        FPGA\\00 - Git\\fpga-quartus\\8_CORE\\MULTI_CORE_edit\\ins_mem.txt",mem);
16    end
17
18    always @(posedge clk) begin
19        if (rEn == 1) begin
20            instruction <= mem[PC_address];
21        end
22    end
23
24 endmodule

```

10.7 Control Unit

```

1 'include "cu_param.v"
2
3 //Control unit outputs
4 // iROMREAD
5 // memREAD //AR read, memory read, DR write
6 // memWRITE
7
8 // [14:0]wEN
9 //REGISTERS:
10 // RT4 Rr PC IR AR DR RP RT RM1 RK1 RN1 C1 C2 C3 AC
11 // 14 13 12 _11 10 9 8 _7 6 5 4 _3 2 1 0
12
13
14 // [4:0]busMUX (2**4)
15 // RT4 Rr MEM AR DR RP RT RM1 RK1 RN1 RM2 RK2 RN2 C1 C2 C3 AC
16 // 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
17
18
19 // [5:0]INC
20 // PC RM2 RK2 RN2 C2 C3

```

```

21 // 5 4 3 2 1 0
22
23 // [4:0] RST
24 // RT RM2 RK2 RN2 AC
25 // 4 3 2 1 0
26
27 // [2:0] compMUX //both muxes get the same control signal
28 // M1-M2 K1-K2 N1-N2
29 // 2 1 0
30
31 // [3:0] aluOP
32 // ADDMEM ADD MUL SET
33 // 3 2 1 0
34
35
36 //NEXT INSTRUCTION
37
38 module controlunit
39 #(parameter WIDTH = 8)
40 (
41     input Clk,
42     input z, //JUMP flag
43     input [WIDTH-1:0] INS, //Instruction from the Instruction
44     memory memAV, //DATA MEMORY AVAILABLE flag
45     input imemAV, //INSTRUCTION MEMORY AVAILABLE flag
46     output reg iROMREAD, //rEn of IROM
47     output reg memREAD, //rEN of DRAM
48     output reg memWRITE, //wEN of DRAM
49     output reg [14:0] wEN, //wEN bitmask of core registers
50     output reg selAR, //flag to be used in address
51     fetching to AR from IROM
52     output reg coreINC_AR, //flag to be used in increment the
53     addresses by the coreID
54     output reg [4:0] busMUX, //Bus selector
55     output reg [5:0] INC, //Register increment bitmask
56     output reg [4:0] RST, //Register reset bitmask
57     output reg [2:0] compMUX, //Control signal to the comparator
58     output reg [3:0] aluOP, //Control signal to the ALU
59     output reg coreS, //Core state flag
60 );
61
62 reg [7:0] NEXT_STATE='FETCH_1;
63 reg [7:0] STATE='FETCH_1;
64 reg zFlag, memAVREG, jmpMFlag;
65
66 always @(negedge Clk) begin
67     zFlag = z;
68     memAVREG = memAV;
69 end
70
71 //DEFINE ALL THE STATES OF THE CONTROL UNIT
72 always @(posedge Clk) begin
73     STATE = NEXT_STATE;
74     case(STATE)
75         'NOOP_1 : begin //NO_OP
76             memREAD <= 0;
77             memWRITE <= 0;
78             wEN <= 0;
79             selAR <= 0;
80             coreINC_AR <= 0;

```

```

81         busMUX <= 0;
82         INC <= 0;
83         RST <= 0;
84         compMUX <= 0;
85         aluOP <= 0;
86         coreS <= 0;
87         NEXT_STATE <= 'FETCH_1;
88         iROMREAD <= 1;                                //iROM read before FETCH_1
89     end
90     'FETCH_1 : begin                                    //FETCH_1      iROM[PC]
91         iROMREAD <= 1;
92         memREAD <= 0;
93         memWRITE <= 0;
94         wEN <= 0;
95         selAR <= 0;
96         coreINC_AR <= 0;
97         busMUX <= 0;
98         INC <= 0;
99         RST <= 0;
100        compMUX <= 0;
101        aluOP <= 0;
102        coreS <= 0;
103        if (imemAV) begin                                //Wait until instruction is
available to the CU
104            NEXT_STATE <= 'FETCH_2;
105        end
106        else begin
107            NEXT_STATE <= 'FETCH_1;
108        end
109    end
110    'FETCH_2 : begin                                    //FETCH_2      IR <= iROM[PC
], PC <= PC+1
111        iROMREAD <= 0;
112        memREAD <= 0;
113        memWRITE <= 0;
114        wEN <= 15'b000_1000_0000_0000;                //IR WRITE
115        selAR <= 0;
116        coreINC_AR <= 0;
117        busMUX <= 0;
118        INC <= 6'b10_0000;
119        RST <= 0;
120        compMUX <= 0;
121        aluOP <= 0;
122        coreS <= 0;
123        NEXT_STATE <= 'FETCH_3;
124    end
125    'FETCH_3 : begin                                    //FETCH_3      IR HAS
ALREADY GOT THE INS
126        iROMREAD <= 0;
127        memREAD <= 0;
128        memWRITE <= 0;
129        wEN <= 0;
130        busMUX <= 0;
131        INC <= 0;
132        RST <= 0;
133        compMUX <= 0;
134        aluOP <= 0;
135        coreS <= 0;
136        case (INS[7:4])
137            'JMP : begin
138                case (INS[3:0])
139                    'JMP_M : NEXT_STATE <= 'JMPM_1;
140                    'JMP_K : NEXT_STATE <= 'JMPK_1;

```

```

141         'JMP_N : NEXT_STATE <= 'JMPN_1;
142     endcase
143 end
144 'COPY : begin
145     NEXT_STATE <= 'COPY_1;
146     iROMREAD <= 1;
147 end
148 'LOAD : begin
149     case (INS[3:0])
150         'LOAD_C1 : NEXT_STATE <= 'LOADC1_1;
151         'LOAD_C2 : NEXT_STATE <= 'LOADC2_1;
152     endcase
153 end
154 'STORE : NEXT_STATE <= 'STORE_1;
155 'ASSIGN : begin
156     NEXT_STATE <= 'ASSIGN_1;
157     iROMREAD <= 1;
158 before ASSIGN_1
159 end
160 'RESET : begin
161     case (INS[3:0])
162         'RESET_ALL : NEXT_STATE <= 'RESETALL_1;
163         'RESET_N2 : NEXT_STATE <= 'RESETN2_1;
164         'RESET_K2 : NEXT_STATE <= 'RESETK2_1;
165         'RESET_Rt : NEXT_STATE <= 'RESETRt_1;
166     endcase
167 end
168 'MOVE : begin
169     case (INS[3:0])
170         'MOVE_RP : NEXT_STATE <= 'MOVEP_1;
171         'MOVE_RT : NEXT_STATE <= 'MOVET_1;
172         'MOVE_RC1 : NEXT_STATE <= 'MOVEC1_1;
173         'MOVE_C3 : NEXT_STATE <= 'MOVEC3_1;
174     endcase
175 end
176 'SET : begin
177     case (INS[3:0])
178         'SETC1 : NEXT_STATE <= 'SETC1_1;
179         'SETDR : NEXT_STATE <= 'SETDR_1;
180         // 'SETRK1 : NEXT_STATE <= 'SETRK1_1;
181     endcase
182 end
183 'MUL : begin
184     case (INS[3:0])
185         'MUL_RP : NEXT_STATE <= 'MULRP_1;
186         // 'MUL_CORE : NEXT_STATE <= 'MULCORE_1;
187     endcase
188 end
189 'ADD : begin
190     case (INS[3:0])
191         'ADD_RT : NEXT_STATE <= 'ADDRT_1;
192         'ADD_RR1 : NEXT_STATE <= 'ADDRR1_1;
193         'ADD_RM2 : NEXT_STATE <= 'ADDRM2_1;
194         'ADD_MEM : NEXT_STATE <= 'ADDC3_1;
195     endcase
196 end
197 'INC : begin
198     case (INS[3:0])
199         'INC_C2 : NEXT_STATE <= 'INCC2_1;
200         'INC_C3 : NEXT_STATE <= 'INCC3_1;
201         'INC_M2 : NEXT_STATE <= 'INCM2_1;
202         'INC_K2 : NEXT_STATE <= 'INCK2_1;
203         'INC_N2 : NEXT_STATE <= 'INCN2_1;

```

```

203         endcase
204     end
205     'END : NEXT_STATE <= 'ENDOP_1;
206     'CHK_IDLE : NEXT_STATE <= 'CHKIDLE_1;
207     'GET_C1 : NEXT_STATE <= 'GETC1_1;
208 endcase
209 end
210
211 'JMPM_1 : begin // 'JMPM_1
212     REG M1 - REG M2
213     iROMREAD <= 0;
214     memREAD <= 0;
215     memWRITE <= 0;
216     wEN <= 0;
217     selAR <= 0;
218     coreINC_AR <= 0;
219     busMUX <= 0;
220     INC <= 0;
221     RST <= 0;
222     compMUX <= 3'b100;
223     aluOP <= 0;
224     coreS <= 0;
225     jmpMFlag <= 1; //to check end
226 operation in ZJMP
227     NEXT_STATE <= 'ZJMP_1;
228 end
229 'JMPK_1 : begin // JMP_K
230     REG K1 - REG K2
231     iROMREAD <= 0;
232     memREAD <= 0;
233     memWRITE <= 0;
234     wEN <= 0;
235     selAR <= 0;
236     coreINC_AR <= 0;
237     busMUX <= 0;
238     INC <= 0;
239     RST <= 0;
240     compMUX <= 3'b010;
241     aluOP <= 0;
242     coreS <= 0;
243     NEXT_STATE <= 'ZJMP_1;
244 end
245 'JMPN_1 : begin // JMP_N
246     REG N1 - REG N2
247     iROMREAD <= 0;
248     memREAD <= 0;
249     memWRITE <= 0;
250     wEN <= 0;
251     selAR <= 0;
252     coreINC_AR <= 0;
253     busMUX <= 0;
254     INC <= 0;
255     RST <= 0;
256     compMUX <= 3'b001;
257     aluOP <= 0;
258     coreS <= 0;
259     NEXT_STATE <= 'ZJMP_1;
260 end
261 'ZJMP_1 : begin //ZERO CHECK (check
262 loop termination)
263     iROMREAD <= 0;
264     memREAD <= 0;
265     memWRITE <= 0;

```

```

261      wEN <= 0;
262      selAR <= 0;
263      coreINC_AR <= 0;
264      busMUX <= 0;
265      INC <= 0;
266      RST <= 0;
267      compMUX <= 0;
268      aluOP <= 0;
269      if (zFlag == 1 && jmpMFlag == 1) begin //end of the program
270          NEXT_STATE <= 'ENDOP_1;
271          jmpMFlag <= 0;
272          coreS <= 1;
273      end
274      else if (zFlag == 1 && jmpMFlag == 0) begin
275          NEXT_STATE <= 'NJMP_1;
276      end
277      else begin
278          NEXT_STATE <= 'JMP_2;
279          iROMREAD <= 1; // iROM read before
280          jmpMFlag <= 0;
281      end
282  end
283  'JMP_2 : begin // JMP_2
284      READ_IROM[PC]
285      iROMREAD <= 0;
286      memREAD <= 0;
287      memWRITE <= 0;
288      wEN <= 0;
289      selAR <= 0;
290      coreINC_AR <= 0;
291      busMUX <= 0;
292      INC <= 0;
293      RST <= 0;
294      compMUX <= 0;
295      aluOP <= 0;
296      coreS <= 0;
297      NEXT_STATE <= 'JMP_3;
298  end
299  'JMP_3 :begin // JMP_3
300      AR <= IROM[PC]
301      iROMREAD <= 0;
302      memREAD <= 0;
303      memWRITE <= 0;
304      wEN <= 15'b000_0100_0000_0000;
305      selAR <= 1;
306      coreINC_AR <= 0;
307      busMUX <= 0;
308      INC <= 0;
309      RST <= 0;
310      compMUX <= 0;
311      aluOP <= 0;
312      coreS <= 0;
313      NEXT_STATE <= 'JMP_4;
314  end
315  'JMP_4 : begin // JMP_4
316      PC <= AR
317      iROMREAD <= 0;
318      memREAD <= 0;
319      memWRITE <= 0;
320      wEN <= 15'b001_0000_0000_0000;
321      selAR <= 0;
322      busMUX <= 14;

```



```

320         INC <= 0;
321         RST <= 0;
322         compMUX <= 0;
323         aluOP <= 0;
324         coreS <= 0;
325         NEXT_STATE <= 'FETCH_1;
326     end
327     'NJMP_1 : begin //NO'JMP
328         PC <= PC + 1
329         iROMREAD <= 0;
330         memREAD <= 0;
331         memWRITE <= 0;
332         wEN <= 0;
333         selAR <= 0;
334         coreINC_AR <= 0;
335         busMUX <= 0;
336         INC <= 6'b10_0000;
337         RST <= 0;
338         compMUX <= 0;
339         aluOP <= 0;
340         coreS <= 0;
341         NEXT_STATE <= 'FETCH_1;
342     end
343     'COPY_1 : begin // 'COPY_1
344         READ_IROM[PC]
345         iROMREAD <= 0;
346         memREAD <= 0;
347         memWRITE <= 0;
348         wEN <= 0;
349         selAR <= 0;
350         coreINC_AR <= 0;
351         busMUX <= 0;
352         INC <= 0;
353         RST <= 0;
354         compMUX <= 0;
355         aluOP <= 0;
356         coreS <= 0;
357         NEXT_STATE <= 'COPY_2;
358     end
359     'COPY_2 : begin // 'COPY_2
360         AR <= IROM[PC], PC <= PC + 1
361         iROMREAD <= 0;
362         memREAD <= 0;
363         memWRITE <= 0;
364         wEN <= 15'b000_0100_0000_0000;
365         selAR <= 1;
366         coreINC_AR <= 0;
367         busMUX <= 0;
368         INC <= 6'b10_0000;
369         RST <= 0;
370         compMUX <= 0;
371         aluOP <= 0;
372         coreS <= 0;
373         if (INS[3:0]== 'COPY_M || INS[3:0]== 'COPY_T) begin
374             NEXT_STATE <= 'COPYM_3A;
375         end
376         else begin
377             NEXT_STATE <= 'COPY_3;
378         end
379     end
380     'COPYM_3A : begin //COPYM3_A
381         AR <= AR + Core_ID
382         iROMREAD <= 0;

```

```

379         memREAD <= 0;
380         memWRITE <= 0;
381         wEN <= 0;
382         selAR <= 0;
383         coreINC_AR <= 1;
384         busMUX <= 0;
385         INC <= 0;
386         RST <= 0;
387         compMUX <= 0;
388         aluOP <= 0;
389         coreS <= 0;
390         NEXT_STATE <= 'COPY_3;
391     end
392     'COPY_3 : begin // 'COPY_3
393         MEM_READ[AR] ... GIVING MEMORY ADDRESS
394         iROMREAD <= 0;
395         memREAD <= 1;
396         memWRITE <= 0;
397         wEN <= 0;
398         selAR <= 0;
399         coreINC_AR <= 0;
400         busMUX <= 4'b1111;
401         INC <= 0;
402         RST <= 0;
403         compMUX <= 0;
404         aluOP <= 0;
405         coreS <= 0;
406         if (memAVREG) begin
407             NEXT_STATE <= 'COPY_4;
408         end
409         else begin
410             NEXT_STATE <= 'HOLD_1;
411         end
412     end
413     'COPY_4 : begin // 'COPY_4
414         DR <= MEM_READ[AR] ... ALREADY RECEIVED DATA
415         iROMREAD <= 0;
416         memREAD <= 0;
417         memWRITE <= 0;
418         wEN <= 15'b000_0010_0000_0000;
419         selAR <= 0;
420         coreINC_AR <= 0;
421         busMUX <= 4'b1111;
422         INC <= 0;
423         RST <= 0;
424         compMUX <= 0;
425         aluOP <= 0;
426         coreS <= 0;
427         if (INS[3:0] == 'COPY_M) begin // RM1
428             NEXT_STATE <= 'COPYM_5;
429         end
430         else if (INS[3:0] == 'COPY_K) begin // RK1
431             NEXT_STATE <= 'COPYK_5;
432         end
433         else if (INS[3:0] == 'COPY_N) begin // RN1
434             NEXT_STATE <= 'COPYN_5;
435         end
436         else if (INS[3:0] == 'COPY_R) begin // Rr
437             NEXT_STATE <= 'COPYR_5;
438         end
439         else if (INS[3:0] == 'COPY_T) begin // RT4
440             NEXT_STATE <= 'COPYRT4_5;
441         end

```

```

440         end
441         'COPYM_5 : begin                                     // 'COPY M1
REG M1 <= DR
442         memREAD <= 0;
443         memWRITE <= 0;
444         wEN <= 15'b000_0000_0100_0000;
445         selAR <= 0;
446         coreINC_AR <= 0;
447         busMUX <= 4'b1101;
448         INC <= 0;
449         RST <= 0;
450         compMUX <= 0;
451         aluOP <= 0;
452         coreS <= 0;
453         NEXT_STATE <= 'FETCH_1;
454         iROMREAD <= 1;                                     // iROM read before
FETCH_1
455         end
456         'COPYK_5 : begin                                     // 'COPY K1
REG K1 <= DR
457         memREAD <= 0;
458         memWRITE <= 0;
459         wEN <= 15'b000_0000_0010_0000;
460         selAR <= 0;
461         coreINC_AR <= 0;
462         busMUX <= 4'b1101;
463         INC <= 0;
464         RST <= 0;
465         compMUX <= 0;
466         aluOP <= 0;
467         coreS <= 0;
468         NEXT_STATE <= 'FETCH_1;
469         iROMREAD <= 1;                                     // iROM read before
FETCH_1
470         end
471         'COPYN_5 : begin                                     // 'COPY N1
REG N1 <= DR
472         memREAD <= 1;
473         memWRITE <= 0;
474         wEN <= 15'b000_0000_0001_0000;
475         selAR <= 0;
476         coreINC_AR <= 0;
477         busMUX <= 13;
478         INC <= 0;
479         RST <= 0;
480         compMUX <= 0;
481         aluOP <= 0;
482         coreS <= 0;
483         NEXT_STATE <= 'FETCH_1;
484         iROMREAD <= 1;                                     // iROM read before
FETCH_1
485         end
486         'COPYR_5 : begin                                     // 'COPY Rr
Rr <= DR
487         memREAD <= 1;
488         memWRITE <= 0;
489         wEN <= 15'b010_0000_0000_0000;
490         selAR <= 0;
491         coreINC_AR <= 0;
492         busMUX <= 13;
493         INC <= 0;
494         RST <= 0;
495         compMUX <= 0;

```

```

496         aluOP <= 0;
497         coreS <= 0;
498         NEXT_STATE <= 'FETCH_1;
499         iROMREAD <= 1;                                     // iROM read before
500     FETCH_1
501     end
502     'COPYRT4_5 : begin                                     // 'COPY RT4
503     RT4 <= DR
504         memREAD <= 1;
505         memWRITE <= 0;
506         wEN <= 15'b100_0000_0000_0000;
507         selAR <= 0;
508         coreINC_AR <= 0;
509         busMUX <= 13;
510         INC <= 0;
511         RST <= 0;
512         compMUX <= 0;
513         aluOP <= 0;
514         coreS <= 0;
515         NEXT_STATE <= 'FETCH_1;
516         iROMREAD <= 1;                                     // iROM read before
517     FETCH_1
518     end
519     'LOAD_2 : begin                                       // 'LOAD_2
520     MEM_READ[AR]
521         iROMREAD <= 0;
522         memREAD <= 1;
523         memWRITE <= 0;
524         wEN <= 0;
525         selAR <= 0;
526         coreINC_AR <= 0;
527         busMUX <= 15;
528         INC <= 0;
529         RST <= 0;
530         compMUX <= 0;
531         aluOP <= 0;
532         coreS <= 0;
533         if (memAVREG) begin
534             NEXT_STATE <= 'LOAD_3;
535         end
536         else begin
537             NEXT_STATE <= 'HOLD_1;
538         end
539     end
540     'LOAD_3 : begin                                       // 'LOAD_3
541     DR <= MEM_READ[AR]
542         memREAD <= 0;
543         memWRITE <= 0;
544         wEN <= 15'b000_0010_0000_0000;
545         selAR <= 0;
546         coreINC_AR <= 0;
547         busMUX <= 15;
548         INC <= 0;
549         RST <= 0;
550         compMUX <= 0;
551         aluOP <= 0;
552         coreS <= 0;
553         NEXT_STATE <= 'FETCH_1;
554         iROMREAD <= 1;                                     // iROM read before
555     FETCH_1
556     end
557     'LOADC1_1 : begin                                     // 'LOAD_C1
558     AR <= C1

```

```

552         iROMREAD <= 0;
553         memREAD <= 0;
554         memWRITE <= 0;
555         wEN <= 15'b000_0100_0000_0000;
556         selAR <= 0;
557         coreINC_AR <= 0;
558         busMUX <= 4;
559         INC <= 0;
560         RST <= 0;
561         compMUX <= 0;
562         aluOP <= 0;
563         coreS <= 0;
564         NEXT_STATE <= 'LOAD_2;
565     end
566     'LOADC2_1 : begin                                     // 'LOAD_C2
567         AR <= C2
568         iROMREAD <= 0;
569         memREAD <= 0;
570         memWRITE <= 0;
571         wEN <= 15'b000_0100_0000_0000;
572         selAR <= 0;
573         coreINC_AR <= 0;
574         busMUX <= 3;
575         INC <= 0;
576         RST <= 0;
577         compMUX <= 0;
578         aluOP <= 0;
579         coreS <= 0;
580         NEXT_STATE <= 'LOAD_2;
581     end
582     'STORE_1 : begin                                     // 'STORE_1
583         DR <= RT
584         iROMREAD <= 0;
585         memREAD <= 0;
586         memWRITE <= 0;
587         wEN <= 15'b000_0010_0000_0000;
588         selAR <= 0;
589         coreINC_AR <= 0;
590         busMUX <= 11;
591         INC <= 0;
592         RST <= 0;
593         compMUX <= 0;
594         aluOP <= 0;
595         coreS <= 0;
596         NEXT_STATE <= 'STORE_2;
597     end
598     'STORE_2 : begin                                     // 'STORE_2
599         AR <= C3
600         iROMREAD <= 0;
601         memREAD <= 0;
602         memWRITE <= 0;
603         wEN <= 15'b000_0100_0000_0000;
604         selAR <= 0;
605         coreINC_AR <= 0;
606         busMUX <= 2;
607         INC <= 0;
608         RST <= 0;
609         compMUX <= 0;
610         aluOP <= 0;
611         coreS <= 0;
        NEXT_STATE <= 'STORE_3;
    end
    'STORE_3 : begin                                     // 'STORE_3

```

```

MEM_WRITE[AR] <= DR
612     iROMREAD <= 0;
613     memREAD <= 0;
614     memWRITE <= 1;
615     wEN <= 0;
616     selAR <= 0;
617     coreINC_AR <= 0;
618     busMUX <= 13;
619     INC <= 0;
620     RST <= 0;
621     compMUX <= 0;
622     aluOP <= 0;
623     coreS <= 0;
624     if (memAVREG) begin
625         NEXT_STATE <= 'FETCH_1;
626         iROMREAD <= 1; // iROM read before
FETCH_1
627     end
628     else begin
629         NEXT_STATE <= 'HOLD_1; // Hold the processor
till the DRAM gives data
630     end
631 end
632 'ASSIGN_1 : begin //ASSIGN_1
READ_IROM[PC]
633     iROMREAD <= 0;
634     memREAD <= 0;
635     memWRITE <= 0;
636     wEN <= 0;
637     selAR <= 0;
638     coreINC_AR <= 0;
639     busMUX <= 0;
640     INC <= 0;
641     RST <= 0;
642     compMUX <= 0;
643     aluOP <= 0;
644     coreS <= 0;
645     NEXT_STATE <= 'ASSIGN_2;
646 end
647 'ASSIGN_2 : begin //ASSIGN_3
AR <= IROM[PC], PC <= PC+1
648     iROMREAD <= 0;
649     memREAD <= 0;
650     memWRITE <= 0;
651     wEN <= 15'b000_0100_0000_0000;
652     selAR <= 1;
653     coreINC_AR <= 0;
654     busMUX <= 0;
655     INC <= 6'b10_0000;
656     RST <= 0;
657     compMUX <= 0;
658     aluOP <= 0;
659     coreS <= 0;
660     if (INS[3:0]== 'ASSIGN_C1) begin
661         NEXT_STATE <= 'ASSIGNC1_3A;
662     end
663     else if (INS[3:0]== 'ASSIGN_C2) begin
664         NEXT_STATE <= 'ASSIGNC2_3;
665     end
666     // else if (INS[3:0]== 'ASSIGN_C3) begin //not needed in
new algo
667     //     NEXT_STATE <= 'ASSIGNC3_3;
668     //     end

```

```

669     end
670     'ASSIGNC1_3A : begin                                     //ASSIGN_C1A
        AR <= AR + Core_ID
671         iROMREAD <= 0;
672         memREAD <= 0;
673         memWRITE <= 0;
674         wEN <= 15'b0000_0000_0000_1000;
675         selAR <= 0;
676         coreINC_AR <= 1;
677         busMUX <= 14;
678         INC <= 0;
679         RST <= 0;
680         compMUX <= 0;
681         aluOP <= 0;
682         coreS <= 0;
683         NEXT_STATE <= 'ASSIGNC1_3;
684         end
685         'ASSIGNC1_3 : begin                                     //ASSIGN_C1
            C1 <= AR
686             memREAD <= 0;
687             memWRITE <= 0;
688             wEN <= 15'b0000_0000_0000_1000;
689             selAR <= 0;
690             coreINC_AR <= 0;
691             busMUX <= 14;
692             INC <= 0;
693             RST <= 0;
694             compMUX <= 0;
695             aluOP <= 0;
696             coreS <= 0;
697             NEXT_STATE <= 'FETCH_1;
698             iROMREAD <= 1;                                     // iROM read before
        end
    FETCH_1
699         end
700         'ASSIGNC2_3 : begin                                     //ASSIGN_C2
            C2 <= AR
701             memREAD <= 0;
702             memWRITE <= 0;
703             wEN <= 15'b0000_0000_0000_0100;
704             selAR <= 0;
705             coreINC_AR <= 0;
706             busMUX <= 14;
707             INC <= 0;
708             RST <= 0;
709             compMUX <= 0;
710             aluOP <= 0;
711             coreS <= 0;
712             NEXT_STATE <= 'FETCH_1;
713             iROMREAD <= 1;                                     // iROM read before
        end
    FETCH_1
714     end
715     // 'ASSIGNC3_3 : begin                                     //ASSIGN_C3
    AR //not needed in new algo                                C3 <=
716     //     iROMREAD <= 0;
717     //     memREAD <= 0;
718     //     memWRITE <= 0;
719     //     wEN <= 15'b0000_0000_0000_0010;
720     //     selAR <= 0;
721     //     coreINC_AR <= 0;
722     //     busMUX <= 14;
723     //     INC <= 0;
724     //     RST <= 0;
725     //     compMUX <= 0;

```

```

726         //      aluOP <= 0;
727         //      coreS <= 0;
728         //      NEXT_STATE <= 'FETCH_1;
729         //      iROMREAD <= 1;          // iROM read before FETCH_1
730     // end
731     'RESETALL_1 : begin                //RESET_ALL
732         memREAD <= 0;
733         memWRITE <= 0;
734         wEN <= 0;
735         selAR <= 0;
736         coreINC_AR <= 0;
737         busMUX <= 0;
738         INC <= 0;
739         RST <= 5'b11111;
740         compMUX <= 0;
741         aluOP <= 0;
742         coreS <= 0;
743         NEXT_STATE <= 'FETCH_1;
744         iROMREAD <= 1;                // iROM read before
745     FETCH_1
746     end
747     'RESETN2_1 : begin                //RESET REG N2
748         memREAD <= 0;
749         memWRITE <= 0;
750         wEN <= 0;
751         selAR <= 0;
752         coreINC_AR <= 0;
753         busMUX <= 0;
754         INC <= 0;
755         RST <= 5'b00010;
756         compMUX <= 0;
757         aluOP <= 0;
758         coreS <= 0;
759         NEXT_STATE <= 'FETCH_1;
760         iROMREAD <= 1;                // iROM read before
761     FETCH_1
762     end
763     'RESETK2_1 : begin                //RESET REG K2
764         memREAD <= 0;
765         memWRITE <= 0;
766         wEN <= 0;
767         selAR <= 0;
768         coreINC_AR <= 0;
769         busMUX <= 0;
770         INC <= 0;
771         RST <= 5'b00100;
772         compMUX <= 0;
773         aluOP <= 0;
774         coreS <= 0;
775         NEXT_STATE <= 'FETCH_1;
776         iROMREAD <= 1;                // iROM read before
777     FETCH_1
778     end
779     'RESETRt_1 : begin                //RESET REG RT
780         memREAD <= 0;
781         memWRITE <= 0;
782         wEN <= 0;
783         selAR <= 0;
784         coreINC_AR <= 0;
785         busMUX <= 0;
786         INC <= 0;
787         RST <= 5'b10000;
788         compMUX <= 0;

```



```

786         aluOP <= 0;
787         coreS <= 0;
788         NEXT_STATE <= 'FETCH_1;
789         iROMREAD <= 1;                                     // iROM read before
790     FETCH_1
791     end
792     'MOVEP_1 : begin                                       //MOVE TO REG P
793         REG P <= AC
794         memREAD <= 0;
795         memWRITE <= 0;
796         wEN <= 15'b000_0001_0000_0000;
797         selAR <= 0;
798         coreINC_AR <= 0;
799         busMUX <= 1;
800         INC <= 0;
801         RST <= 0;
802         compMUX <= 0;
803         aluOP <= 0;
804         coreS <= 0;
805         NEXT_STATE <= 'FETCH_1;
806         iROMREAD <= 1;                                     // iROM read before
807     FETCH_1
808     end
809     'MOVET_1 : begin                                       //MOVE TO REG T
810         REG T <= AC
811         memREAD <= 0;
812         memWRITE <= 0;
813         wEN <= 15'b000_0000_1000_0000;
814         selAR <= 0;
815         coreINC_AR <= 0;
816         busMUX <= 1;
817         INC <= 0;
818         RST <= 0;
819         compMUX <= 0;
820         aluOP <= 0;
821         coreS <= 0;
822         NEXT_STATE <= 'FETCH_1;
823         iROMREAD <= 1;                                     // iROM read before
824     FETCH_1
825     end
826     'MOVEC1_1 : begin                                       //MOVE TO REG C1
827         C1 <= AC
828         memREAD <= 0;
829         memWRITE <= 0;
830         wEN <= 15'b000_0000_0000_1000;
831         selAR <= 0;
832         coreINC_AR <= 0;
833         busMUX <= 1;
834         INC <= 0;
835         RST <= 0;
836         compMUX <= 0;
837         aluOP <= 0;
838         coreS <= 0;
839         NEXT_STATE <= 'FETCH_1;
840         iROMREAD <= 1;                                     // iROM read before
841     FETCH_1
842     end
843     'MOVEC3_1 : begin                                       //MOVE TO REG C3
844         C3 <= AC
845         memREAD <= 0;
846         memWRITE <= 0;
847         wEN <= 15'b000_0000_0000_0010;
848         selAR <= 0;

```

```

841         coreINC_AR <= 0;
842         busMUX <= 1;
843         INC <= 0;
844         RST <= 0;
845         compMUX <= 0;
846         aluOP <= 0;
847         coreS <= 0;
848         NEXT_STATE <= 'FETCH_1;
849         iROMREAD <= 1;                                // iROM read before
FETCH_1
850     end
851     'SETC1_1 : begin                                    //SET AC AS C1
852         SET, AC <= C1
853         memREAD <= 0;
854         memWRITE <= 0;
855         wEN <= 15'b0000_0000_0000_0001;
856         selAR <= 0;
857         coreINC_AR <= 0;
858         busMUX <= 4;
859         INC <= 0;
860         RST <= 0;
861         compMUX <= 0;
862         aluOP <= 4'b0001;
863         coreS <= 0;
864         NEXT_STATE <= 'FETCH_1;
865         iROMREAD <= 1;                                // iROM read before
FETCH_1
866     end
867     'SETDR_1 : begin                                    //SET AC AS DR
868         SET, AC <= DR
869         memREAD <= 0;
870         memWRITE <= 0;
871         wEN <= 15'b0000_0000_0000_0001;
872         selAR <= 0;
873         coreINC_AR <= 0;
874         busMUX <= 13;
875         INC <= 0;
876         RST <= 0;
877         compMUX <= 0;
878         aluOP <= 4'b0001;
879         coreS <= 0;
880         NEXT_STATE <= 'FETCH_1;
881         iROMREAD <= 1;                                // iROM read before
FETCH_1
882     end
883     // 'SETRK1_1 : begin                                //SET AC AS DR      SET, AC <= DR
884     //     iROMREAD <= 0;
885     //     memREAD <= 0;
886     //     memWRITE <= 0;
887     //     wEN <= 15'b0000_0000_0000_0001;
888     //     selAR <= 0;
889     //     coreINC_AR <= 0;
890     //     busMUX <= 9;
891     //     INC <= 0;
892     //     RST <= 0;
893     //     compMUX <= 0;
894     //     aluOP <= 4'b0001;
895     //     coreS <= 0;
896     //     NEXT_STATE <= 'FETCH_1;
897     // end
898     'MULRP_1 : begin                                    //MUL REG P
899     AC <= REG_P * AC
900     memREAD <= 0;

```

```

898         memWRITE <= 0;
899         wEN <= 15'b000_0000_0000_0001;
900         selAR <= 0;
901         coreINC_AR <= 0;
902         busMUX <= 12;
903         INC <= 0;
904         RST <= 0;
905         compMUX <= 0;
906         aluOP <= 4'b0010;
907         coreS <= 0;
908         NEXT_STATE <= 'FETCH_1;
909         iROMREAD <= 1;                                     // iROM read before
910     end
911     // 'MULCORE_1 : begin                                     //AC MULTIPLY CORE ID          AC <=
912     CORE ID * AC
913     //         iROMREAD <= 0;
914     //         memREAD <= 0;
915     //         memWRITE <= 0;
916     //         wEN <= 15'b000_0000_0000_0001;
917     //         selAR <= 0;
918     //         coreINC_AR <= 0;
919     //         busMUX <= 0;
920     //         INC <= 0;
921     //         RST <= 0;
922     //         compMUX <= 0;
923     //         aluOP <= 4'b1000;
924     //         coreS <= 0;
925     //         NEXT_STATE <= 'FETCH_1;
926     // end
927     'ADDRT_1 : begin                                         //ADD REG_T
928     AC <= REG_1 + AC
929     memREAD <= 0;
930     memWRITE <= 0;
931     wEN <= 15'b000_0000_0000_0001;
932     selAR <= 0;
933     coreINC_AR <= 0;
934     busMUX <= 11;
935     INC <= 0;
936     RST <= 0;
937     compMUX <= 0;
938     aluOP <= 4'b0100;
939     coreS <= 0;
940     NEXT_STATE <= 'FETCH_1;
941     iROMREAD <= 1;                                     // iROM read before
942     end
943     'ADDRR1_1 : begin                                       //ADD REG_Rr
944     AC <= Rr + AC
945     memREAD <= 0;
946     memWRITE <= 0;
947     wEN <= 15'b000_0000_0000_0001;
948     selAR <= 0;
949     coreINC_AR <= 0;
950     busMUX <= 16;
951     INC <= 0;
952     RST <= 0;
953     compMUX <= 0;
954     aluOP <= 4'b0100;
955     coreS <= 0;
956     NEXT_STATE <= 'FETCH_1;
957     iROMREAD <= 1;                                     // iROM read before

```

```

956     FETCH_1
957     end
958     'ADDRM2_1 : begin //ADD REG_M2
959     AC <= REG_M2 + AC
960     memREAD <= 0;
961     memWRITE <= 0;
962     wEN <= 15'b000_0000_0000_0001;
963     selAR <= 0;
964     coreINC_AR <= 0;
965     busMUX <= 7;
966     INC <= 0;
967     RST <= 0;
968     compMUX <= 0;
969     aluOP <= 4'b0100;
970     coreS <= 0;
971     NEXT_STATE <= 'FETCH_1;
972     iROMREAD <= 1; // iROM read before
973
974     FETCH_1
975     end
976     'ADDC3_1 : begin //ADD RK1*CORE_ID TO C3
977     AC <= C3 + AC
978     memREAD <= 0;
979     memWRITE <= 0;
980     wEN <= 15'b000_0000_0000_0001;
981     selAR <= 0;
982     coreINC_AR <= 0;
983     busMUX <= 2;
984     INC <= 0;
985     RST <= 0;
986     compMUX <= 0;
987     aluOP <= 4'b1000;
988     coreS <= 0;
989     NEXT_STATE <= 'FETCH_1;
990     iROMREAD <= 1; // iROM read before
991
992     FETCH_1
993     end
994     'INCC2_1 : begin //INC REG C2
995     REG C2 <= C2+1
996     memREAD <= 0;
997     memWRITE <= 0;
998     wEN <= 0;
999     busMUX <= 0;
1000    INC <= 6'b00_0010;
1001    RST <= 0;
1002    compMUX <= 0;
1003    aluOP <= 0;
1004    coreS <= 0;
1005    NEXT_STATE <= 'FETCH_1;
1006    iROMREAD <= 1; // iROM read before
1007
1008    FETCH_1
1009    end
1010    'INCC3_1 : begin //INC REG C3
1011    REG C3 <= C3+1
1012    memREAD <= 0;
1013    memWRITE <= 0;
1014    wEN <= 0;
1015    selAR <= 0;
1016    coreINC_AR <= 0;
1017    busMUX <= 0;
1018    INC <= 6'b00_0001;
1019    RST <= 0;
1020    compMUX <= 0;
1021    aluOP <= 0;

```

```

1011         coreS <= 0;
1012         NEXT_STATE <= 'FETCH_1;
1013         iROMREAD <= 1;                                     // iROM read before
1014     FETCH_1
1015     end
1016     'INCM2_1 : begin                                       //INC reg m2
1017     REG M2 <= M2+1
1018         memREAD <= 0;
1019         memWRITE <= 0;
1020         wEN <= 0;
1021         selAR <= 0;
1022         coreINC_AR <= 0;
1023         busMUX <= 0;
1024         INC <= 6'b01_0000;
1025         RST <= 0;
1026         compMUX <= 0;
1027         aluOP <= 0;
1028         coreS <= 0;
1029         NEXT_STATE <= 'FETCH_1;
1030         iROMREAD <= 1;                                     // iROM read before
1031     FETCH_1
1032     end
1033     'INCK2_1 : begin                                       //INC REG K2
1034     REG K2 <= K2+1
1035         memREAD <= 0;
1036         memWRITE <= 0;
1037         wEN <= 0;
1038         selAR <= 0;
1039         coreINC_AR <= 0;
1040         busMUX <= 0;
1041         INC <= 6'b00_1000;
1042         RST <= 0;
1043         compMUX <= 0;
1044         aluOP <= 0;
1045         coreS <= 0;
1046         NEXT_STATE <= 'FETCH_1;
1047         iROMREAD <= 1;                                     // iROM read before
1048     FETCH_1
1049     end
1050     'INCN2_1 : begin                                       //INC REG N2
1051     REG N2 <= N2+1
1052         memREAD <= 0;
1053         memWRITE <= 0;
1054         wEN <= 0;
1055         selAR <= 0;
1056         coreINC_AR <= 0;
1057         busMUX <= 0;
1058         INC <= 6'b00_0100;
1059         RST <= 0;
1060         compMUX <= 0;
1061         aluOP <= 0;
1062         coreS <= 0;
1063         NEXT_STATE <= 'FETCH_1;
1064         iROMREAD <= 1;                                     // iROM read before
1065     FETCH_1
1066     end
1067     'ENDOP_1 : begin                                       //INC REG N2
1068     REG N2 <= N2+1
1069         memREAD <= 0;
1070         memWRITE <= 0;
1071         wEN <= 0;
1072         selAR <= 0;
1073         coreINC_AR <= 0;

```

```

1066         busMUX <= 0;
1067         INC <= 0;
1068         RST <= 0;
1069         compMUX <= 0;
1070         aluOP <= 0;
1071         coreS <= 1;
1072         NEXT_STATE <= 'ENDOP_1;
1073     end
1074     'HOLD_1 : begin
1075         iROMREAD <= 0;
1076         memREAD <= 0;
1077         memWRITE <= 0;
1078         wEN <= 0;
1079         selAR <= 0;
1080         coreINC_AR <= 0;
1081         busMUX <= 0;
1082         INC <= 0;
1083         RST <= 0;
1084         compMUX <= 0;
1085         aluOP <= 0;
1086         coreS <= 0;
1087         if (memAVREG) begin
1088             case (INS[7:4])
1089                 'COPY : NEXT_STATE <= 'COPY_4;
1090                 'LOAD : NEXT_STATE <= 'LOAD_3;
1091                 'STORE : begin
1092                     NEXT_STATE <= 'FETCH_1;
1093                     iROMREAD <= 1; // iROM read before
1094             end
1095             endcase
1096         end
1097         else begin
1098             NEXT_STATE <= 'HOLD_1; // Hold till data is
1099             available frome DRAM
1100         end
1101     end
1102     'CHKIDLE_1 : begin //CHECK RM1 - RM2
1103         REG M1 - REG M2
1104         iROMREAD <= 0;
1105         memREAD <= 0;
1106         memWRITE <= 0;
1107         wEN <= 0;
1108         selAR <= 0;
1109         coreINC_AR <= 0;
1110         busMUX <= 0;
1111         INC <= 0;
1112         RST <= 0;
1113         compMUX <= 3'b100;
1114         aluOP <= 0;
1115         coreS <= 0;
1116         NEXT_STATE <= 'CHKIDLE_2;
1117     end
1118     'CHKIDLE_2 : begin //ZERO CHECK
1119         iROMREAD <= 0;
1120         memREAD <= 0;
1121         memWRITE <= 0;
1122         wEN <= 0;
1123         selAR <= 0;
1124         coreINC_AR <= 0;
1125         busMUX <= 0;
1126         INC <= 0;
1127         RST <= 0;

```

```

1126         compMUX <= 0;
1127         aluOP <= 0;
1128         if (zFlag) begin
1129             NEXT_STATE <= 'ENDOP_1;
1130         end
1131         else begin
1132             NEXT_STATE <= 'FETCH_1;
1133             iROMREAD <= 1; // iROM read before
1134         end
1135     end
1136     'GETC1_1 : begin //MOVE RT4 VAL TO
1137         REG C1        C1 <= RT4
1138         memREAD <= 0;
1139         memWRITE <= 0;
1140         wEN <= 15'b0000_0000_0000_1000;
1141         selAR <= 0;
1142         coreINC_AR <= 0;
1143         busMUX <= 17; //RT4
1144         INC <= 0;
1145         RST <= 0;
1146         compMUX <= 0;
1147         aluOP <= 0;
1148         coreS <= 0;
1149         NEXT_STATE <= 'FETCH_1;
1150         iROMREAD <= 1; // iROM read before
1151     end
1152 endcase
1153 end
endmodule

```

10.8 Arithmetic and Logic Unit

```

1  module Alu #(
2      parameter WIDTH = 8
3  ) (
4      input [WIDTH-1:0] AC, BusOut,
5      input [3:0] ALU_OP,
6      input [WIDTH-1:0] MEM_ID,
7      output [WIDTH-1:0] result_ac
8  );
9
10     localparam SET = 4'b0001;
11     localparam MUL = 4'b0010;
12     localparam ADD = 4'b0100;
13     localparam ADDMEM = 4'b1000;
14
15
16     reg [WIDTH-1:0] result;
17
18     always @(*) begin
19         case (ALU_OP)
20             SET:
21                 result <= BusOut;
22             MUL:
23                 result <= AC*BusOut;
24             ADD:
25                 result <= AC+BusOut;
26             ADDMEM:
27                 result <= AC+MEM_ID;
28             default:
29                 result <= AC;
30         endcase

```

```

31     end
32     assign result_ac = result;
33
34 endmodule

```

10.9 Bus Mux

```

1  // 'include "proc_pram.v"
2  //  MEM AR  DR  RP  RT  RM1 RK1 RN1 RM2 RK2 RN2 C1  C2  C3  AC
3  //  15   14  13  12  11  10  9   8   7   6   5   4   3   2   1
4
5  module Bus_mux
6  #(parameter WIDTH = 8)
7  (MEM, AR, DR, RP, RT, RM1, RK1, RN1, RM2, RK2, RN2, C1, C2, C3, AC, RR, RT4,
   mux_sel, Bus_select);
8
9  input [4:0] mux_sel;
10 input [WIDTH-1:0] MEM, AR, DR, RP, RT, RM1, RK1, RN1, RM2, RK2, RN2, C1, C2,
   C3, AC, RR, RT4;
11 output [WIDTH-1:0] Bus_select;
12 // The output is defined as register
13 reg [WIDTH-1:0] select;
14 always @(*)
15 begin
16     case(mux_sel)
17
18         5'b00001:
19             select <= AC;
20         5'b00010:
21             select <= C3;
22         5'b00011:
23             select <= C2;
24         5'b00100:
25             select <= C1;
26         5'b00101:
27             select <= RN2;
28         5'b00110:
29             select <= RK2;
30         5'b00111:
31             select <= RM2;
32         5'b01000:
33             select <= RN1;
34         5'b01001:
35             select <= RK1;
36         5'b01010:
37             select <= RM1;
38         5'b01011:
39             select <= RT;
40         5'b01100:
41             select <= RP;
42         5'b01101:
43             select <= DR;
44         5'b01110:
45             select <= AR;
46         5'b01111:
47             select <= MEM;
48         5'b10000:
49             select <= RR;
50         5'b10001:
51             select <= RT4;
52
53     endcase
54 end
55

```



```

56 assign Bus_select = select;
57 endmodule

```

10.10 Comparator

```

1  module Comp
2  #(parameter WIDTH = 8)
3  (
4      input [WIDTH-1:0] R1, R2,           //inputs comes from AC and bus
5      output wire z                     //control signal for Jump
6      instructions
7  );
8  reg [WIDTH-1:0] value;
9  reg flagOut;
10 always @(*) begin
11     value <= R1-R2;
12     flagOut <= (value == 8'b0);
13 end
14
15 assign z = flagOut;
16
17 endmodule

```

10.11 Program Counter(PC)

```

1  module PC //write, increment enable /PC, C2, C3
2  #(parameter WIDTH = 8)
3  (Clk, WEN, INC, BusOut, dout);
4  input Clk, WEN, INC;
5  input [WIDTH-1:0] BusOut;
6  output [WIDTH-1:0] dout;
7  reg [WIDTH-1:0] value;
8
9  initial begin
10     value <= 8'b0;
11 end
12 always @(negedge Clk)
13 begin
14
15     if (WEN) value <= BusOut;
16     else if (INC) value <= dout + 8'b1;
17 end
18
19 assign dout=value;
20
21 endmodule

```

10.12 Address Register (AR)

```

1  module AR //write enable /IR, AR, Rp, Rt, Rk1, Rm1, Rn1, C1
2  #(parameter WIDTH = 8)
3  (
4  input Clk, WEN, selAR, coreINC_AR,
5  input [WIDTH-1:0] IOut, BusOut,
6  input [2:0] coreID,
7  output [WIDTH-1:0] dout
8  );
9
10 reg [WIDTH-1:0] value;
11
12 always @(negedge Clk)
13 begin
14     if(WEN==1 && selAR == 0 && coreINC_AR == 0) begin

```

```

15         value <= BusOut;
16     end
17     if (coreINC_AR==1) begin //will be used
18         in the COPYRm1 & COPYT4
19         value <= value + coreID;
20     end
21     if(WEN==1 && selAR == 1 && coreINC_AR == 0) begin
22         value <= IOut;
23     end
24 end
25 assign dout=value;
26
27 endmodule

```

10.13 Read-Write Register

```

1 module Reg_module_W //write enable /IR, AR, Rp, Rt, Rk1, Rm1, Rn1, C1
2 #(parameter WIDTH = 8)
3 (Clk, WEN, BusOut, dout);
4 input Clk, WEN;
5 input [WIDTH-1:0] BusOut;
6 output [WIDTH-1:0] dout;
7 reg [WIDTH-1:0] value;
8
9 always @(negedge Clk)
10 begin
11
12     if (WEN) value <= BusOut;
13
14 end
15
16 assign dout=value;
17
18 endmodule

```

10.14 Write-Reset Register

```

1 module Reg_module_RW //write, reset enable //Rt, AC
2 #(parameter WIDTH = 8)
3 (Clk, RST, WEN, BusOut, dout);
4 input Clk, WEN, RST;
5 input [WIDTH-1:0] BusOut;
6 output [WIDTH-1:0] dout;
7 reg [WIDTH-1:0] value;
8
9
10 always @(negedge Clk)
11 begin
12
13     if (WEN) value <= BusOut;
14     else if (RST) value <= 8'b0;
15 end
16
17 assign dout=value;
18
19 endmodule

```

10.15 Write-Increment Register

```

1 module Reg_module_RI //increment, reset enable //Rn2, Rk2, Rm2
2 #(parameter WIDTH = 8)
3 (Clk, RST, INC, BusOut, dout);
4 input Clk, RST, INC;

```

```

5  input [WIDTH-1:0] BusOut;
6  output [WIDTH-1:0] dout;
7  reg [WIDTH-1:0] value;
8
9  always @(negedge Clk)
10 begin
11
12     if (RST) value <= 8'b0;
13     else if (INC) value <= dout + 8'b1;
14 end
15
16 assign dout=value;
17
18 endmodule

```

10.16 Write-Reset-Increment Register

```

1  module Reg_module_WI //write, increment enable /PC, C2, C3
2  #(parameter WIDTH = 8)
3  (Clk, WEN, INC, BusOut, dout);
4  input Clk, WEN, INC;
5  input [WIDTH-1:0] BusOut;
6  output [WIDTH-1:0] dout;
7  reg [WIDTH-1:0] value;
8
9  initial begin
10     value <= 8'b0;
11 end
12 always @(negedge Clk)
13 begin
14
15     if (WEN) value <= BusOut;
16     else if (INC) value <= dout + 8'b1;
17 end
18
19 assign dout=value;
20
21 endmodule

```