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

List of Tables

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 M1 K1 N1	8 8 8 8	W W W W	Row Register Register M1 Register K1 Register N1	Connected within the core registers. Used to store data about the matrices. Write only happens once and store the value.
M2 K2 N2	8 8 8	RI RI RI	Register M2 Register K2 Register N2	Connected within the core registers. Used to store data about the matrices. Data is written and manipulated.
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 C2 C3	8 8 8	W WI WI	Register C1 Register C2 Register C3	Connected within the core, used to store addresses for DRAM access. These registers functioning as address pointers on a higher level.
RP RT	8 8	W WR	Multiplier Register Temporary Register	Used to store the result of ALU operations
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

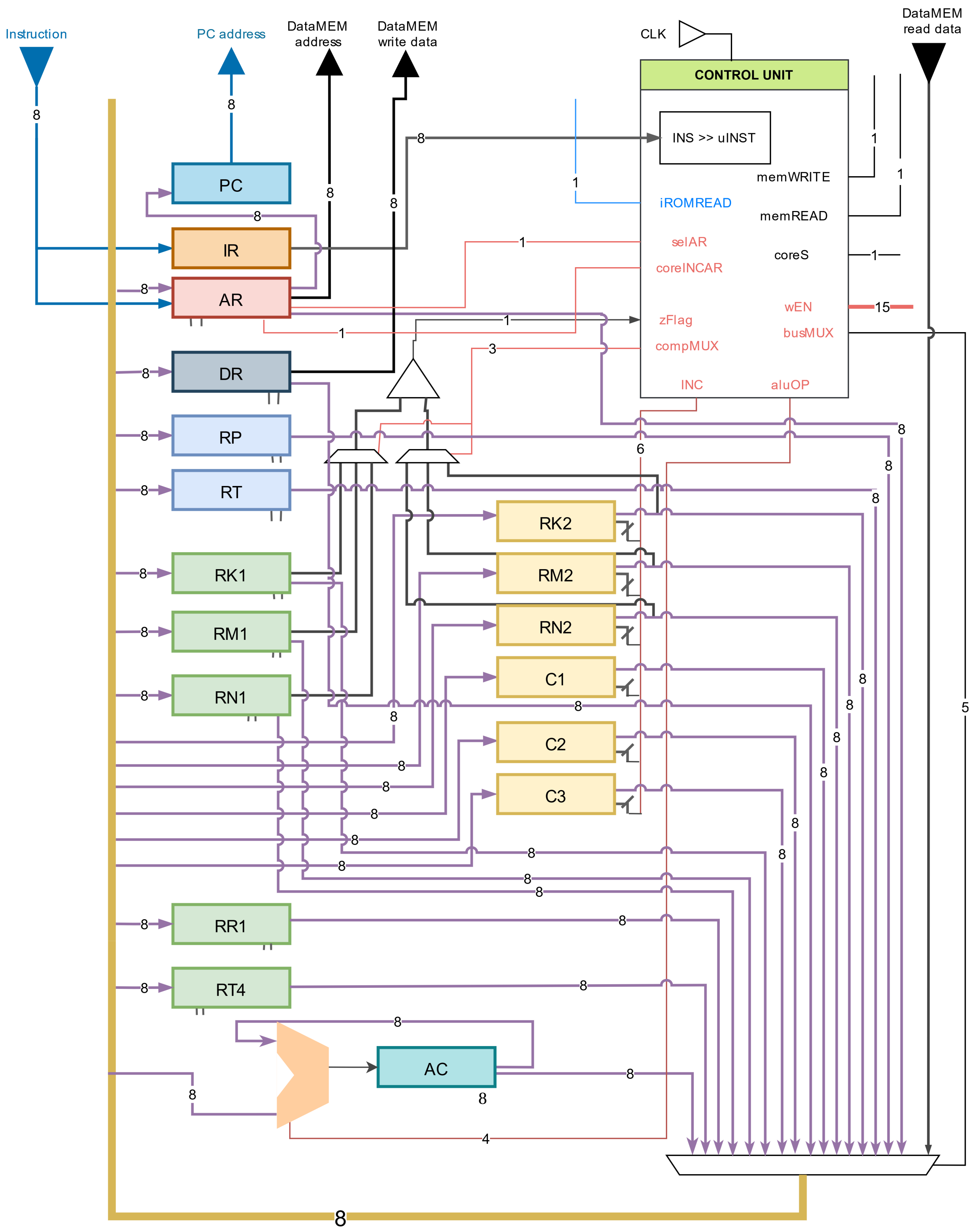


Figure 2.1: Data Path of the Core

2.2 The Instruction Set

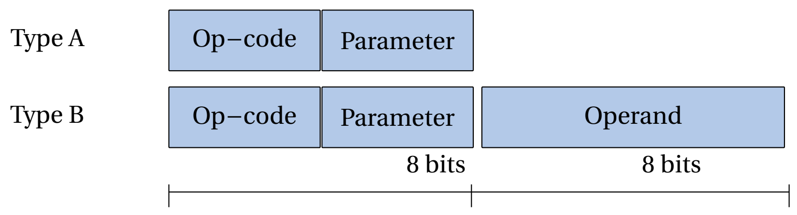


Figure 2.2: Instruction Types

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

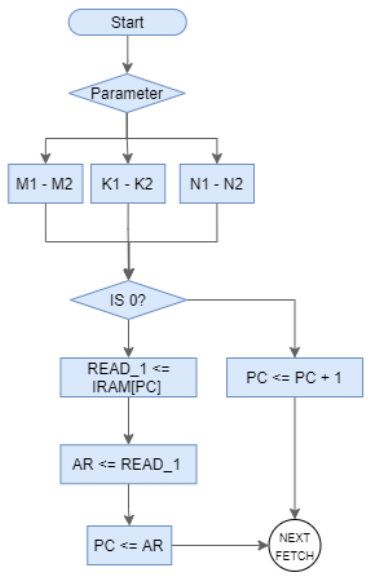


Figure 2.3: JPNZ Flowchart

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 to:
    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 to:
    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.

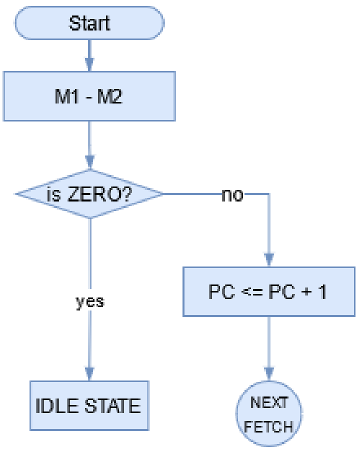


Figure 2.4: Check_Idle Flow Chart

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
Memory	0011	0- C1 1- C2	LOAD	LOAD [C1]	Load values from DRAM to AC **Parameters:** Memory location addresss	4
Memory	0100		STORE	STORE	Store vaues in DRAM	4
Memory	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
General	0111	0- RP 1- RT 2- C1 3- C3	MOVE	MOVE [RP]	Move values from AC to a read write register or a ponter **Parameters:** Register or pointer names	1
General	1110		GET	GET	Move value from RT to C3 pointer	1
General	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
Arithmetic	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
Arithmetic	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:

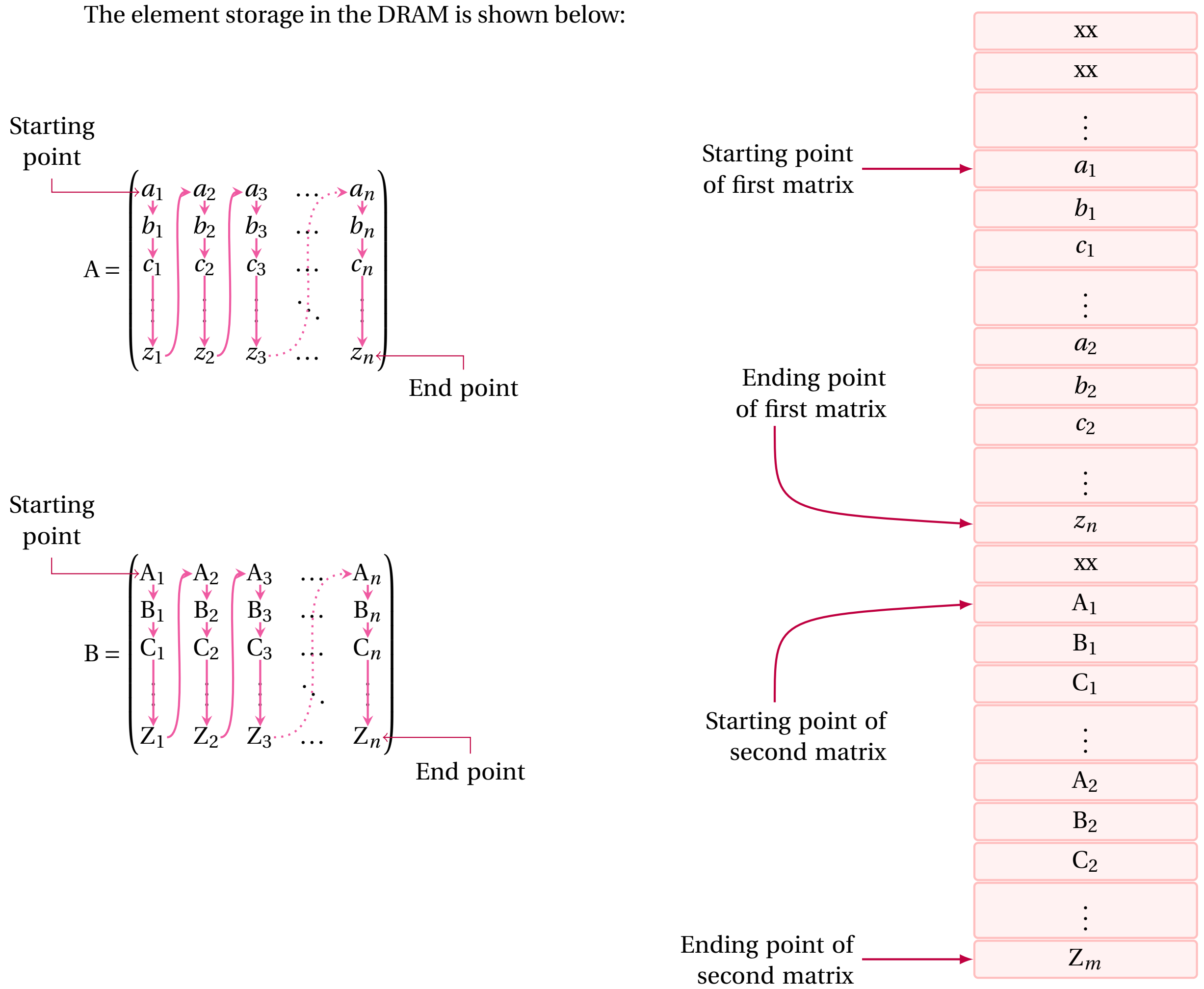
$$C_{M\times K} = A_{M\times N} \times B_{N\times K}$$

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:



Starting point
$A =$
a_1 a_2 a_3 … a_n
b_1 b_2 b_3 … b_n
c_1 c_2 c_3 … c_n
z_1 z_2 z_3 … z_n
End point
Starting point
$B =$
A_1 A_2 A_3 … A_n
B_1 B_2 B_3 … B_n
C_1 C_2 C_3 … C_n
Z_1 Z_2 Z_3 … Z_n
End point
Starting point of first matrix
Ending point of first matrix
Starting point of second matrix
Ending point of second matrix
xx
xx
a_1
b_1
c_1
a_2
b_2
c_2
z_n
xx
A_1
B_1
C_1
A_2
B_2
C_2
Z_m

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.

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

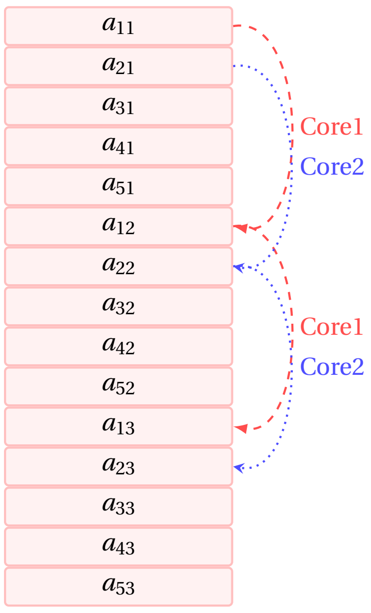


Figure 3.2: Multiple cores accessing the First Matrix in Dram

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.

3.4 The Algorithm for Matrix Multiplication

```
COPY [Rm1]
T1
RESET [ALL]
if z==1:
        CHK_IDLE
COPY [Rn1]
T2
COPY [Rk1]
T3
COPY [Rr]
T7
COPY [Rt4]
T4
ADD_MEM
MOVE [C3]
```

```
#loop[M] begin:
    ASSIGN [C2]
    T5
    #loop[K] begin:
        GET_C1
        RESET [RT]
        SET [C1]
        ADD [RM2]
        MOVE [C1]
        #loop[N] begin:
            LOAD [C1]
            SET [DR]
            MOVE [Rp]
            LOAD [C2]
            SET [DR]
            MUL [Rp]
            ADD [Rt]
            MOVE [Rt]
            SET [C1]
            ADD [Rm1]
            MOVE [C1]
            INC [C2]
            INC [Rn2]
            if (z==0){
                JPNZ [N]
            }else{
                NJMP
            }
            #loop[N]
        STORE
        INC [C3]
        RESET [Rn2]
        INC [Rk2]
        if (z==0){
            JPNZ [K]
        }else{
            NJMP
        }
        #loop [K]
    RESET [Rk2]
    INC [Rm2]
    if (z==0){
        JPNZ [M]
    }else{
        NJMP
    }
    #loop[M]
    END
```

4 Architecture

4.1 State Diagram & Micro Instructions

.

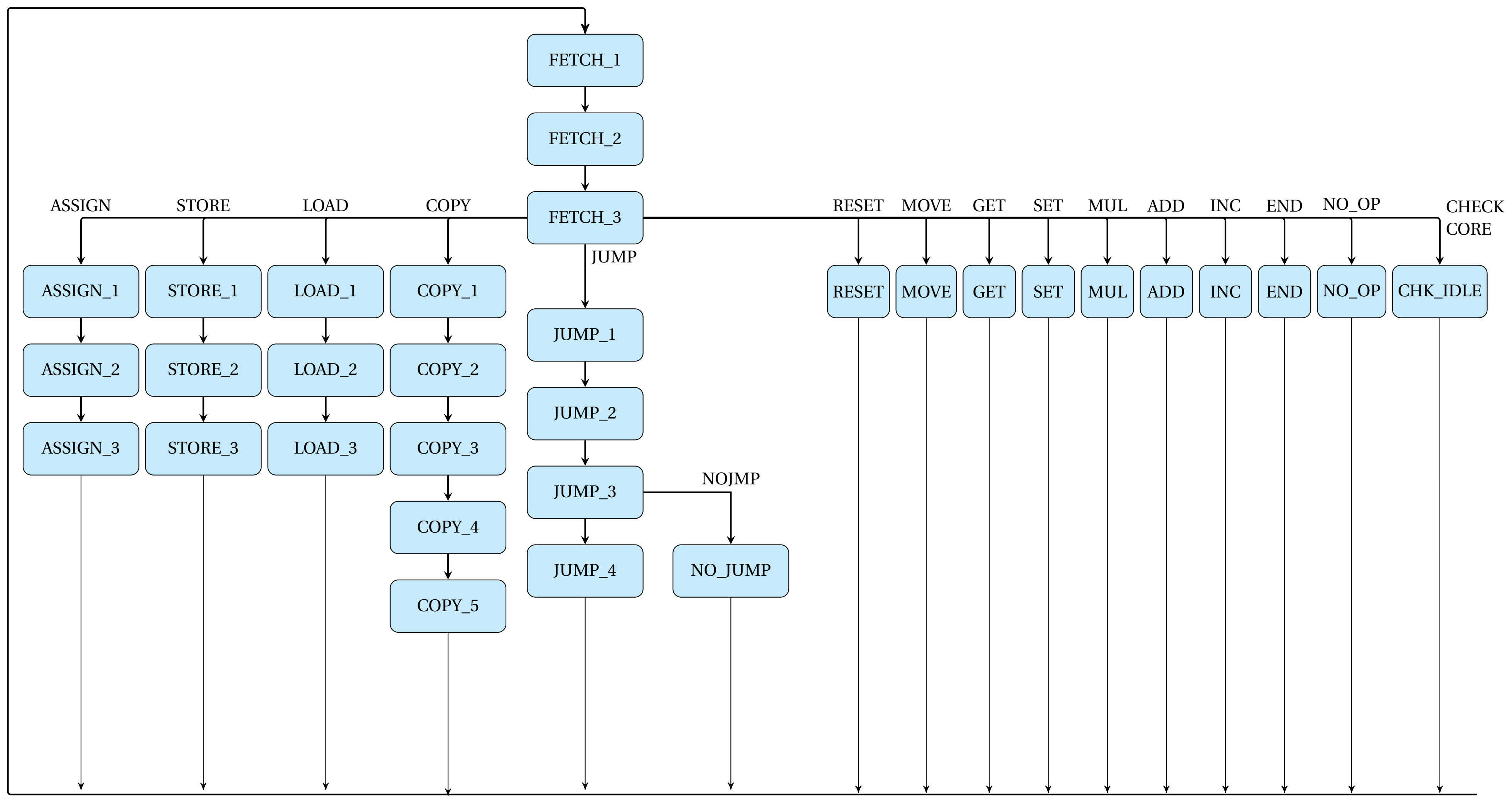


Figure 4.1: Control Unit State Diagram

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.

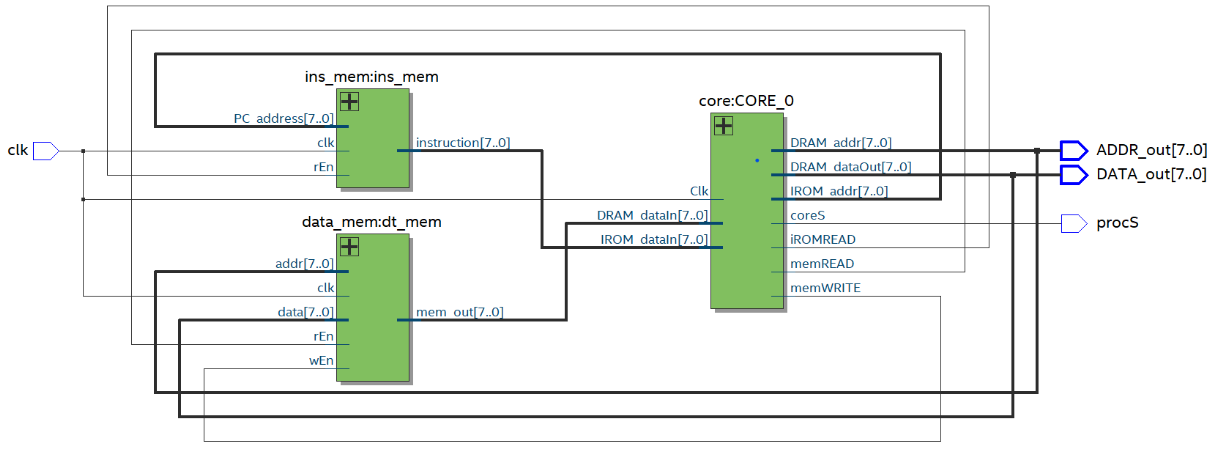


Figure 4.2: Single Core Processor

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

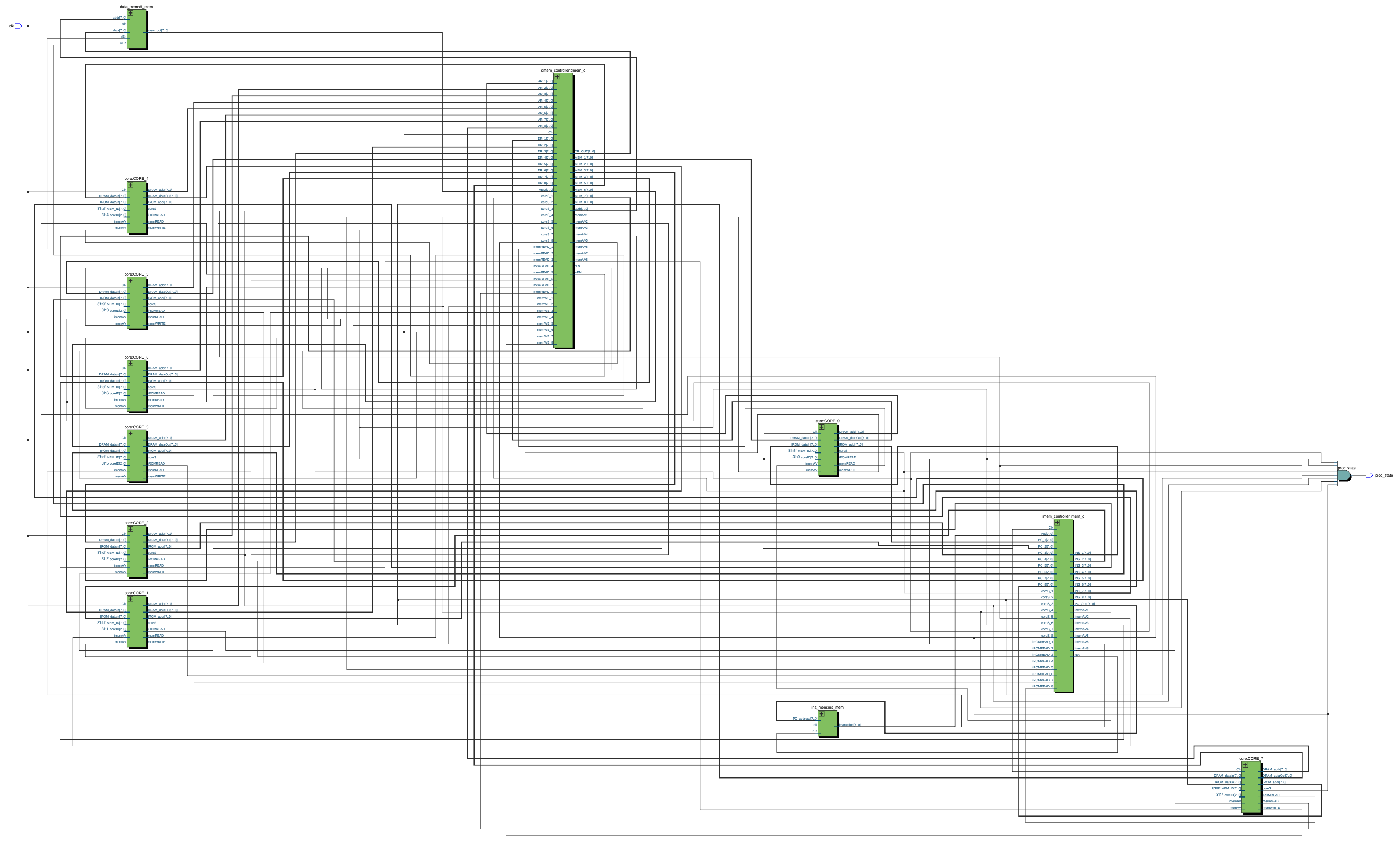
4.3 Processor - Multi Core

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

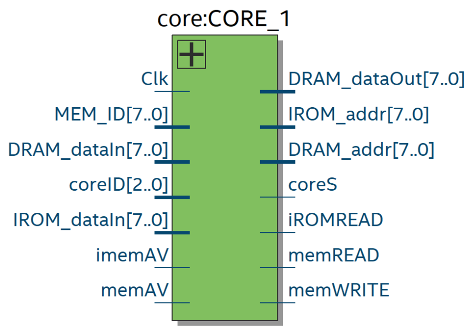


Figure 4.3: Core Module

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:

IN	coreID	:	Core identifying number
	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

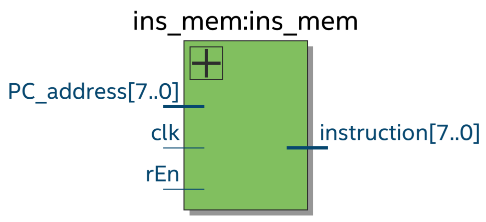


Figure 4.4: Instruction Memory Module

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

Table 4.5: IROM Controller Control Signals

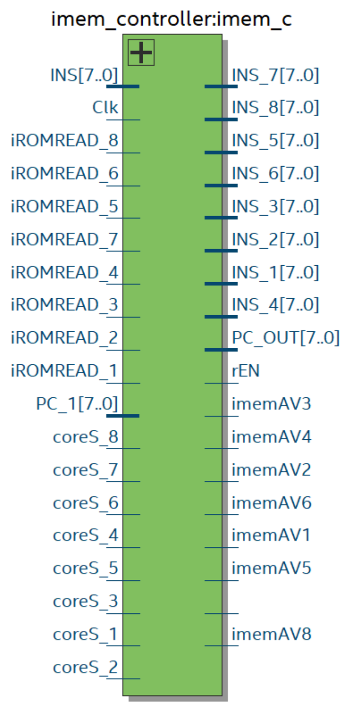


Figure 4.5: Instruction Memory Controller

4.4.4 Data Memory

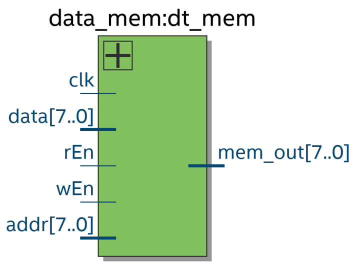
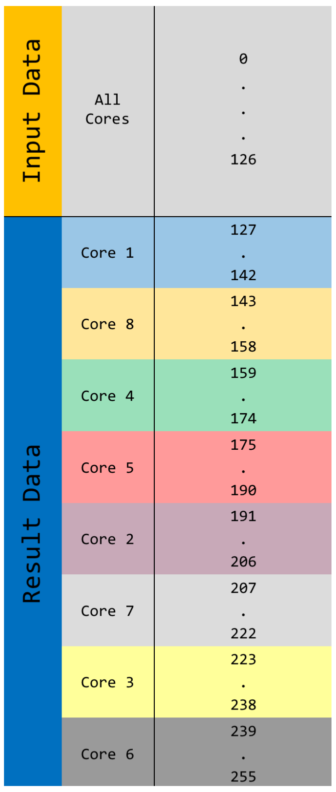


Figure 4.6: Data Memory Module

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.

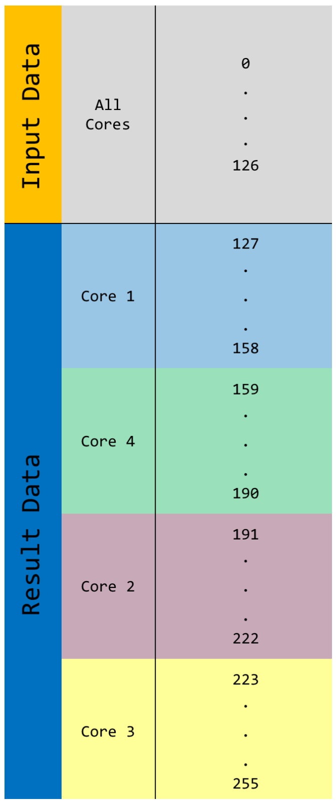
The data input as well as output for DRAM is always from the DR (Data Register) of the CPU/cores and the address pointer is always the AR (Address Register). In the implementation, the data memory data_in is directly connected to the DR but the data_out is connected to the data bus of the core.
In our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



Input Data	All Cores	0 . . . 126
Result Data	Core 1	127 . . . 158
	Core 4	159 . . . 190
	Core 2	191 . . . 222
	Core 3	223 . . . 255

(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

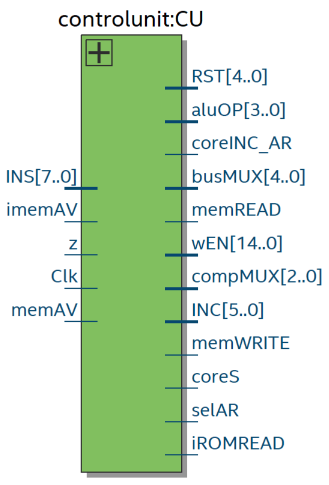


Figure 4.8: Control Unit

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

```
case (INS[7:4])
    `JMP : begin
        case (INS[3:0])
            `JMP_M : NEXT_STATE <= `JMPM_1;
            `JMP_K : NEXT_STATE <= `JMPK_1;
            `JMP_N : NEXT_STATE <= `JMPN_1;
        endcase
    end
    `COPY : begin
        NEXT_STATE <= `COPY_1;
        iROMREAD <= 1;
    end
    `LOAD : begin
        case (INS[3:0])
           `LOAD_C1 : NEXT_STATE <= `LOADC1_1;
           `LOAD_C2 : NEXT_STATE <= `LOADC2_1;
        endcase
    end
    `STORE : NEXT_STATE <= `STORE_1;
    `ASSIGN : begin
        NEXT_STATE <= `ASSIGN_1;
        iROMREAD <= 1;
    end
    `RESET : begin
        case (INS[3:0])
            `RESET_ALL : NEXT_STATE <= `RESETALL_1;
            `RESET_N2 : NEXT_STATE <= `RESETN2_1;
            `RESET_K2 : NEXT_STATE <= `RESETK2_1;
            `RESET_Rt : NEXT_STATE <= `RESETRt_1;
        endcase
    end
    `MOVE : begin
        case (INS[3:0])
            `MOVE_RP : NEXT_STATE <= `MOVEP_1;
            `MOVE_RT : NEXT_STATE <= `MOVET_1;
            `MOVE_RC1 : NEXT_STATE <= `MOVEC1_1;
            `MOVE_C3 : NEXT_STATE <= `MOVEC3_1;
        endcase
    end
    `SET : begin
        case (INS[3:0])
            `SETC1 : NEXT_STATE <= `SETC1_1;
            `SETDR : NEXT_STATE <= `SETDR_1;
        endcase
    end
    `MUL : begin
        case (INS[3:0])
        `MUL_RP : NEXT_STATE <= `MULRP_1;
        endcase
    end
    `ADD : begin
        case (INS[3:0])
            `ADD_RT : NEXT_STATE <= `ADDRT_1;
            `ADD_RR1 : NEXT_STATE <= `ADDRR1_1;
            `ADD_RM2 : NEXT_STATE <= `ADDRM2_1;
            `ADD_MEM : NEXT_STATE <= `ADDC3_1;
        endcase
    end
    `INC : begin
        case (INS[3:0])
            `INC_C2 : NEXT_STATE <= `INCC2_1;
            `INC_C3 : NEXT_STATE <= `INCC3_1;
            `INC_M2 : NEXT_STATE <= `INCM2_1;
            `INC_K2 : NEXT_STATE <= `INCK2_1;
            `INC_N2 : NEXT_STATE <= `INCN2_1;
        endcase
    end
    `END : NEXT_STATE <= `ENDOP_1;
    `CHK_IDLE : NEXT_STATE <= `CHKIDLE_1;
    `GET_C1 : NEXT_STATE <= `GETC1_1;
endcase
```

4.5.2 ALU

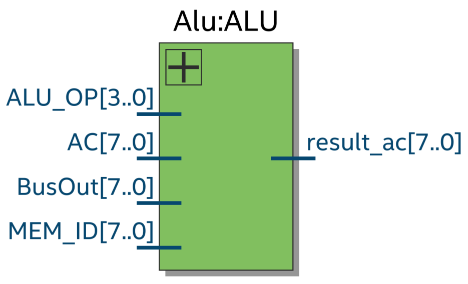


Figure 4.9: ALU Module

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:

```
always @* begin
  case (ALU_OP)
        SET:     result<=BusOut;
        MUL:     result<=AC*BusOut;
        ADD:     result<=AC+BusOut;
        default: result<=AC;
    endcase
end
```

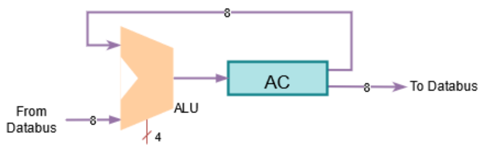


Figure 4.10: ALU & AC register

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.

```
always @(*)
begin
    case(mux_sel)
        5'b00001: select <= AC;
        5'b00010: select <= C3;
        5'b00011: select <= C2;
        5'b00100: select <= C1;
        5'b00101: select <= RN2;
        5'b00110: select <= RK2;
        5'b00111: select <= RM2;
        5'b01000: select <= RN1;
        5'b01001: select <= RK1;
        5'b01010: select <= RM1;
        5'b01011: select <= RT;
        5'b01100: select <= RP;
        5'b01101: select <= DR;
        5'b01110: select <= AR;
        5'b01111: select <= MEM;
        5'b10000: select <= RR;
        5'b10001: select <= RT4;
    endcase
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.

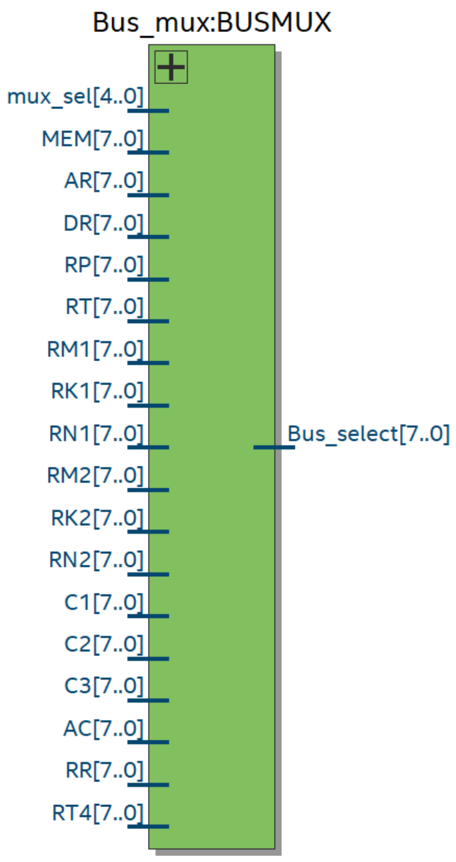


Figure 4.11: Data-bus Multiplexer Module

4.6.2 Comparator & its Muxes

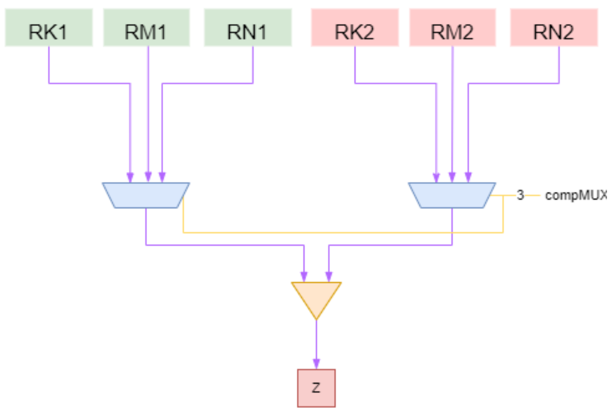


Figure 4.12: Comparator and Comp-Muxes Block Diagram

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 maximuize 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.

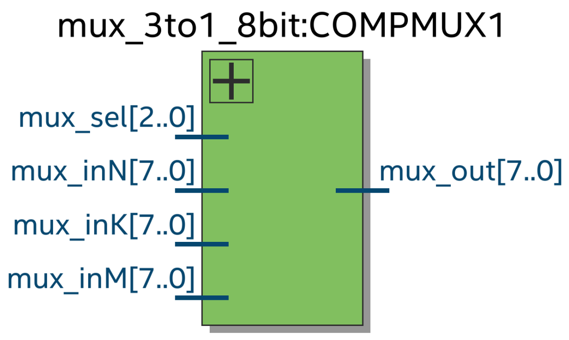


Figure 4.13: Comparator Bus Mux

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

```
always @(*)
begin
    case(mux_sel)
        3'b001:   mux_out <= mux_inN;
        3'b010:   mux_out <= mux_inK;
        3'b100:   mux_out <= mux_inM;
        default:  mux_out <= 8'bz;
    endcase
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.

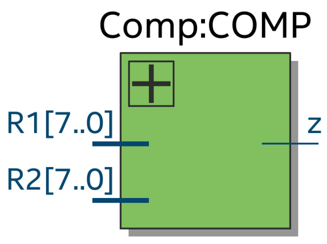


Figure 4.14: Comparator

```
always @(*) begin
    value <= R1-R2;
  flagOut <= (value == 8'b0);
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)

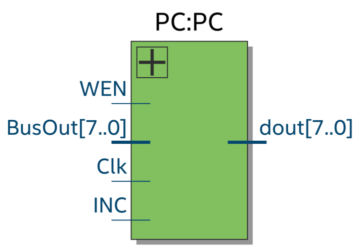


Figure 4.15: Program Counter PC Register

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.

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

```
always @(negedge Clk)
begin

    if (WEN) value <= BusOut;
    else if (INC) value <= dout + 8'b1;
end
```

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)

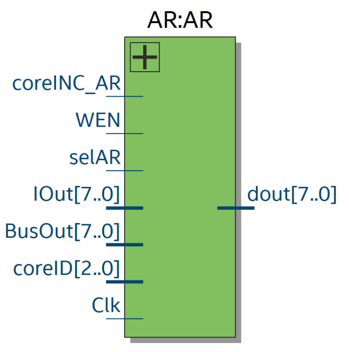


Figure 4.16: Address Register AR

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 implemenation, 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

```
always @(negedge Clk)
begin
    if(WEN==1 && selAR == 0 && coreINC_AR == 0) begin
        value <= BusOut;
    end
    if (coreINC_AR==1) begin
        value <= value + coreID;
    end
    if(WEN==1 && selAR == 1 && coreINC_AR == 0) begin
        value <= IOut;
    end
end
```

4.7.3 Read-Write Registers

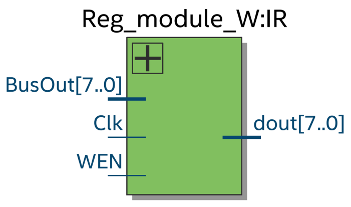


Figure 4.17: Read-Write Register

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

```
always @(negedge Clk)
begin

    if (WEN) value <= BusOut;

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

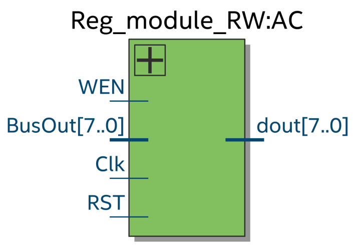


Figure 4.18: Reset-Write Register

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

```
always @(negedge Clk)
begin

    if (WEN) value <= BusOut;
    else if (RST) value <= 8'b0;
end
```

4.7.5 Write-Increment Registers

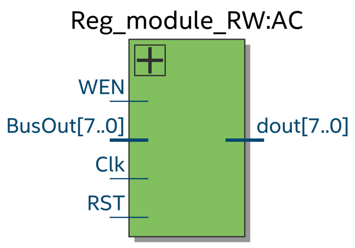


Figure 4.19: Write-Increment Register

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

```
always @(negedge Clk)
begin

    if (WEN) value <= BusOut;
    else if (INC) value <= dout + 8'b1;
end
```

4.7.6 Write-Reset-Increment Registers

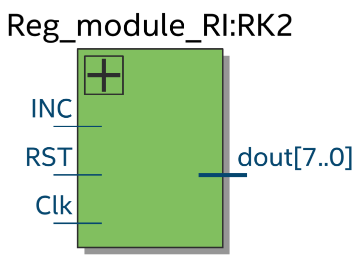


Figure 4.20: Reset-Increment Register

These registers have 3 controls and, as show below, are used as counters for each loop. They are all RESET to zero at the begining 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.

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

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

```
always @(negedge Clk)
begin

    if (RST) value <= 8'b0;
    else if (INC) value <= dout + 8'b1;
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

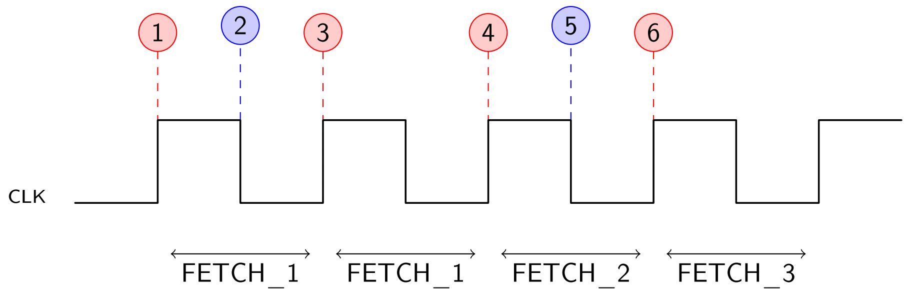


Figure 5.1: Clock cycles of FETCH

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

.

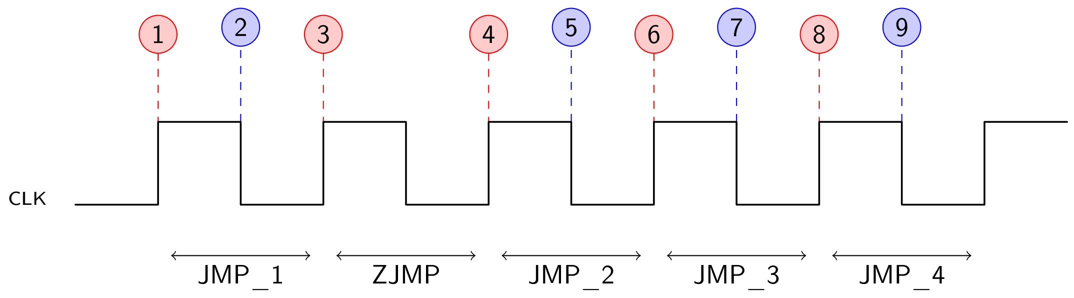


Figure 5.2: Clock cycles of JUMP (z = 0)

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

.

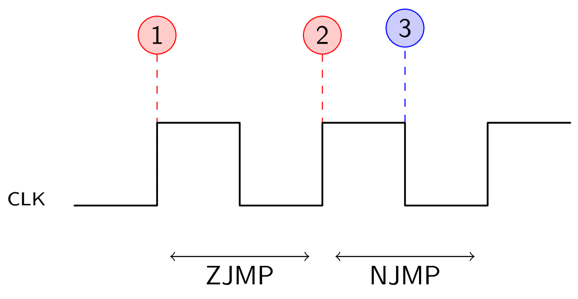


Figure 5.3: Clock cycles of JUMP (z = 1)

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

.

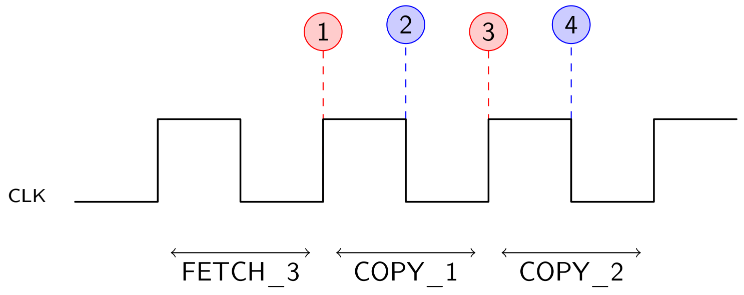


Figure 5.4: Clock cycles of COPY (fetching Copy Address)

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

.

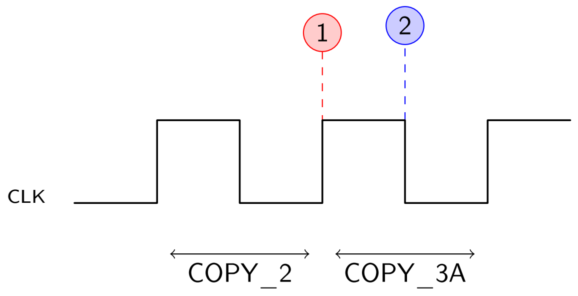


Figure 5.5: Clock cycles of COPY 3A (Increment Addresses)

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

.

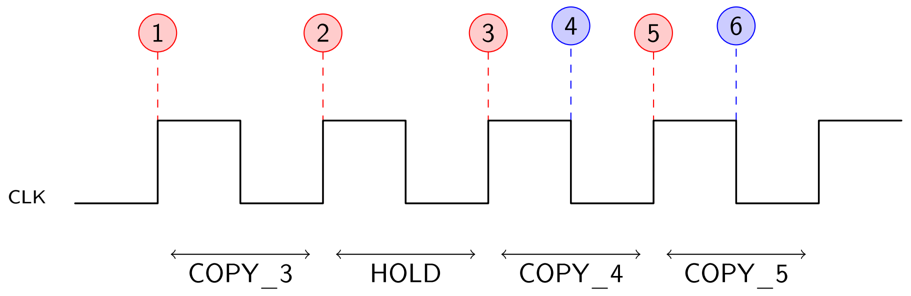


Figure 5.6: Clock cycles of storing data in registers

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

.

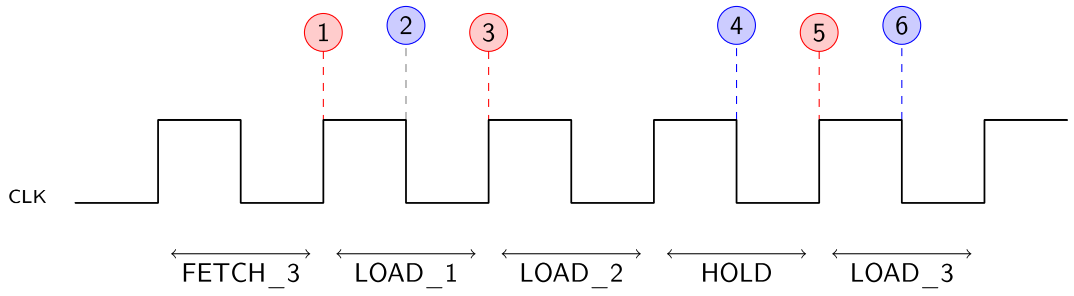


Figure 5.7: Clock cycles of LOAD

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

.

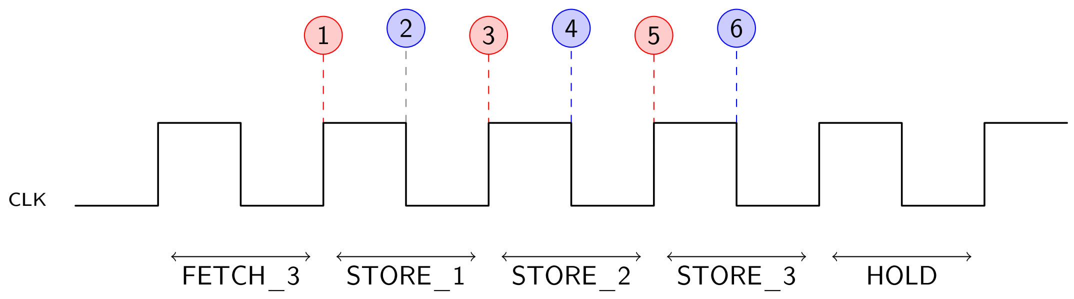


Figure 5.8: Clock cycles of STORE

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

.

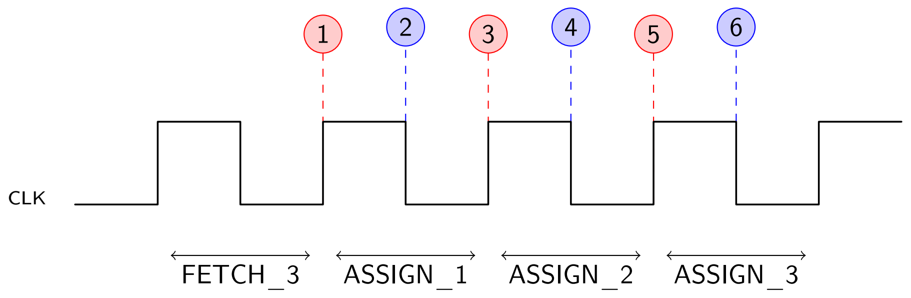


Figure 5.9: Clock cycles of ASSIGN

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

.

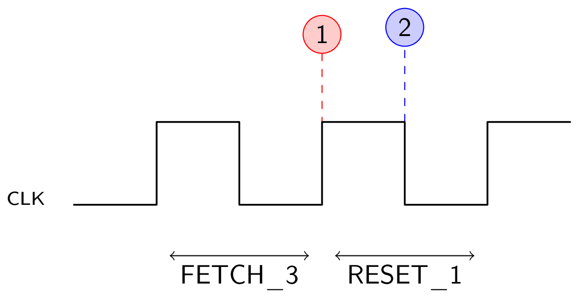


Figure 5.10: Clock cycles of RESET

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

.

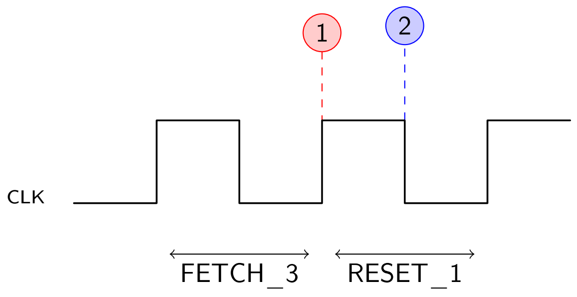


Figure 5.11: Clock cycles of MOVE

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

.

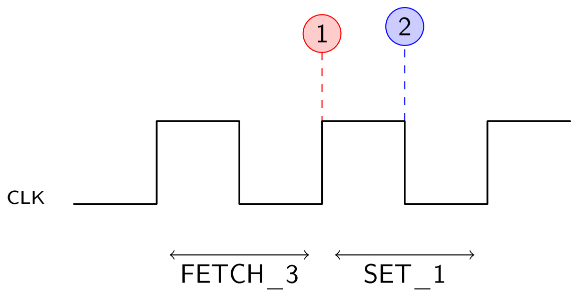


Figure 5.12: Clock cycles of SET

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

.

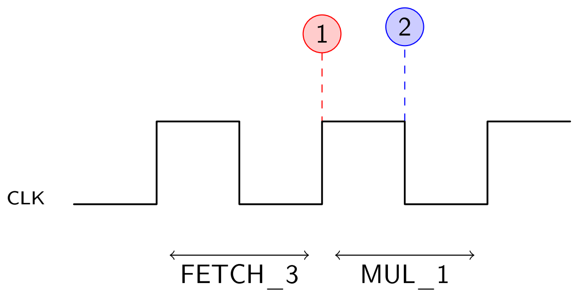


Figure 5.13: Clock cycles of MUL

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

.

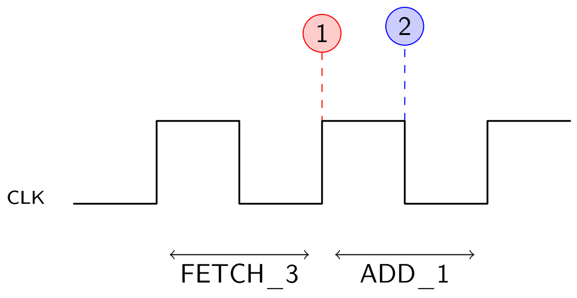


Figure 5.14: Clock cycles of ADD

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

.

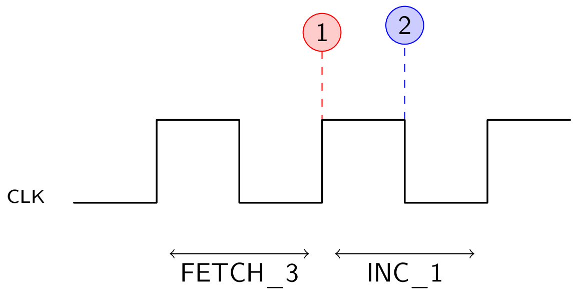


Figure 5.15: Clock cycles of INC

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

.

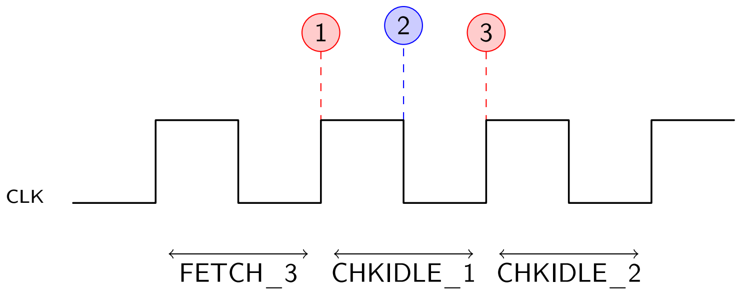


Figure 5.16: Clock cycles of CHK_IDLE

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

.

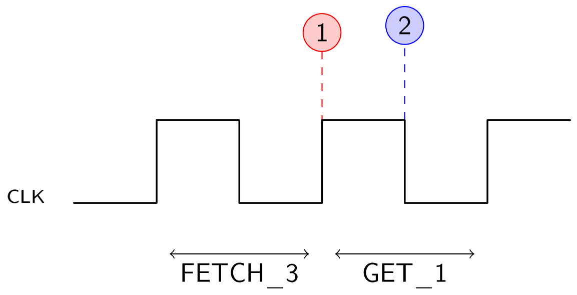


Figure 5.17: Clock cycles of GET

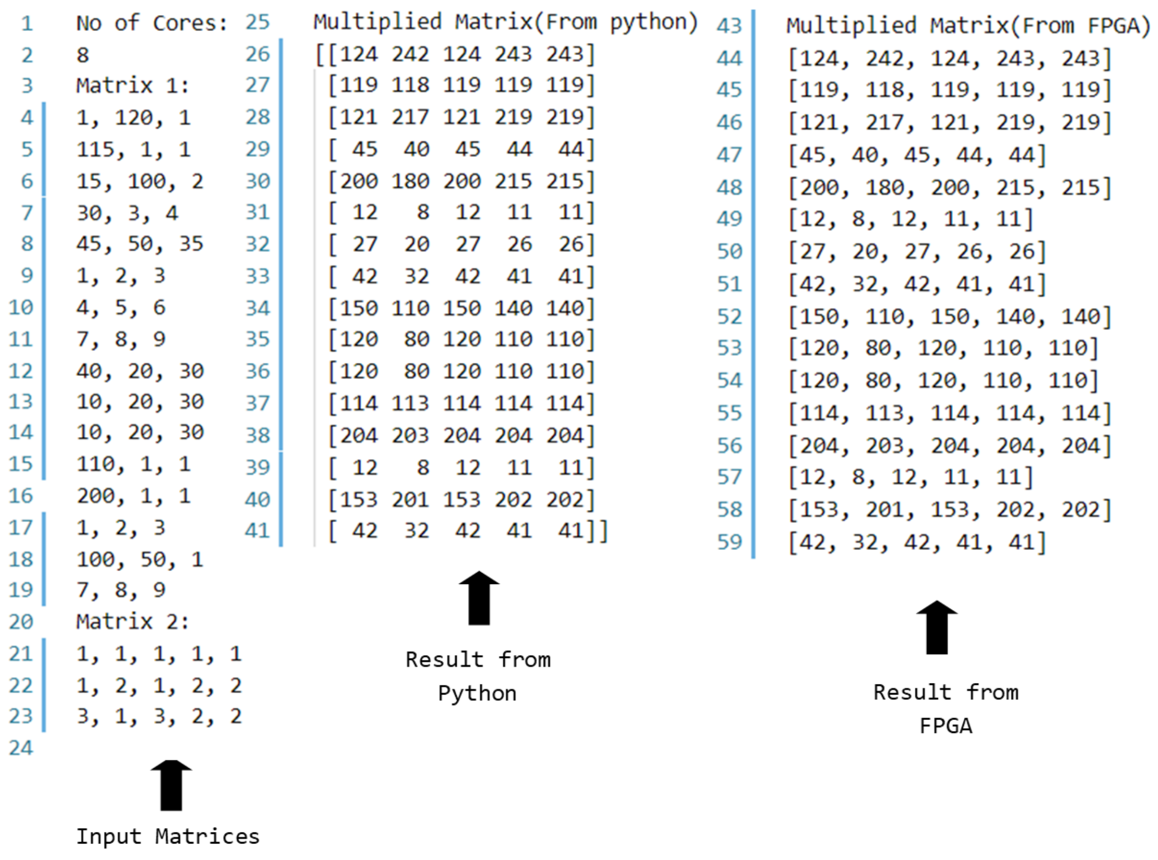
1. BusMux is enabled for the RT4 register.
   RC1 is updated with the value of RT4.
2. value is available for the Control unit from the RC1.

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.

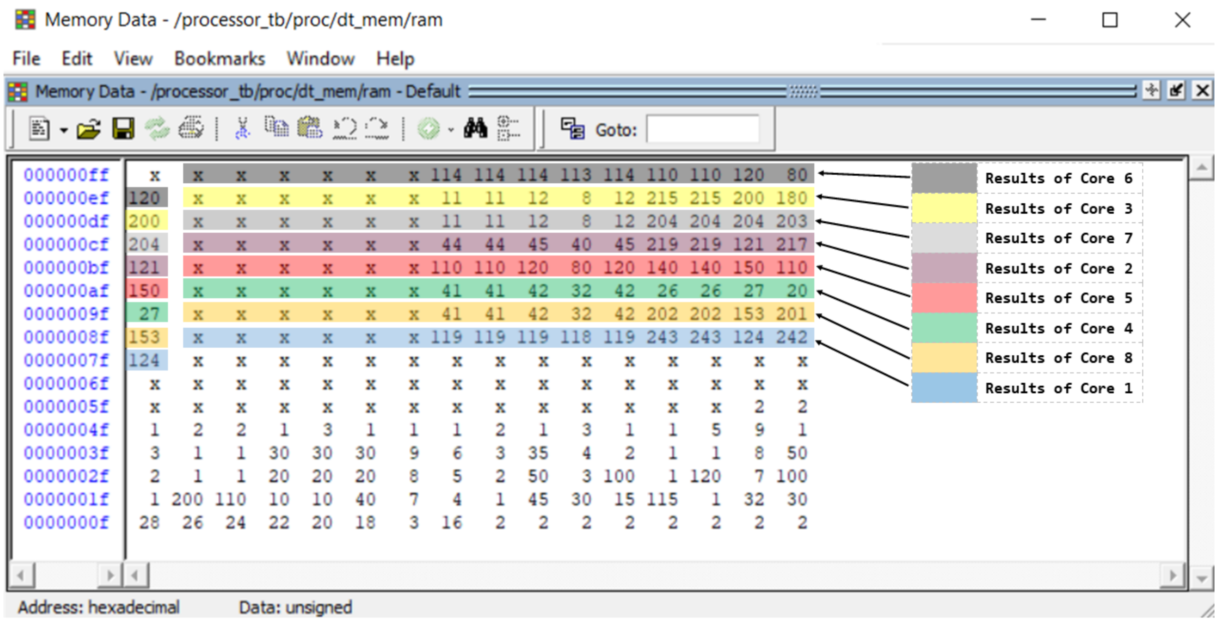


```
No of Cores:
8
Matrix 1:
1, 120, 1
115, 1, 1
15, 100, 2
30, 3, 4
45, 50, 35
1, 2, 3
4, 5, 6
7, 8, 9
40, 20, 30
10, 20, 30
10, 20, 30
110, 1, 1
200, 1, 1
1, 2, 3
100, 50, 1
7, 8, 9
Matrix 2:
1, 1, 1, 1, 1
1, 2, 1, 2, 2
3, 1, 3, 2, 2
```

```
Multiplied Matrix(From python)
[[124 242 124 243 243]
 [119 118 119 119 119]
 [121 217 121 219 219]
 [ 45  40  45  44  44]
 [200 180 200 215 215]
 [ 12   8  12  11  11]
 [ 27  20  27  26  26]
 [ 42  32  42  41  41]
 [150 110 150 140 140]
 [120  80 120 110 110]
 [120  80 120 110 110]
 [114 113 114 114 114]
 [204 203 204 204 204]
 [ 12   8  12  11  11]
 [153 201 153 202 202]
 [ 42  32  42  41  41]]
```

```
Multiplied Matrix(From FPGA)
[124, 242, 124, 243, 243]
[119, 118, 119, 119, 119]
[121, 217, 121, 219, 219]
[45, 40, 45, 44, 44]
[200, 180, 200, 215, 215]
[12, 8, 12, 11, 11]
[27, 20, 27, 26, 26]
[42, 32, 42, 41, 41]
[150, 110, 150, 140, 140]
[120, 80, 120, 110, 110]
[120, 80, 120, 110, 110]
[114, 113, 114, 114, 114]
[204, 203, 204, 204, 204]
[12, 8, 12, 11, 11]
[153, 201, 153, 202, 202]
[42, 32, 42, 41, 41]
```

Figure 6.1: Result Comparison

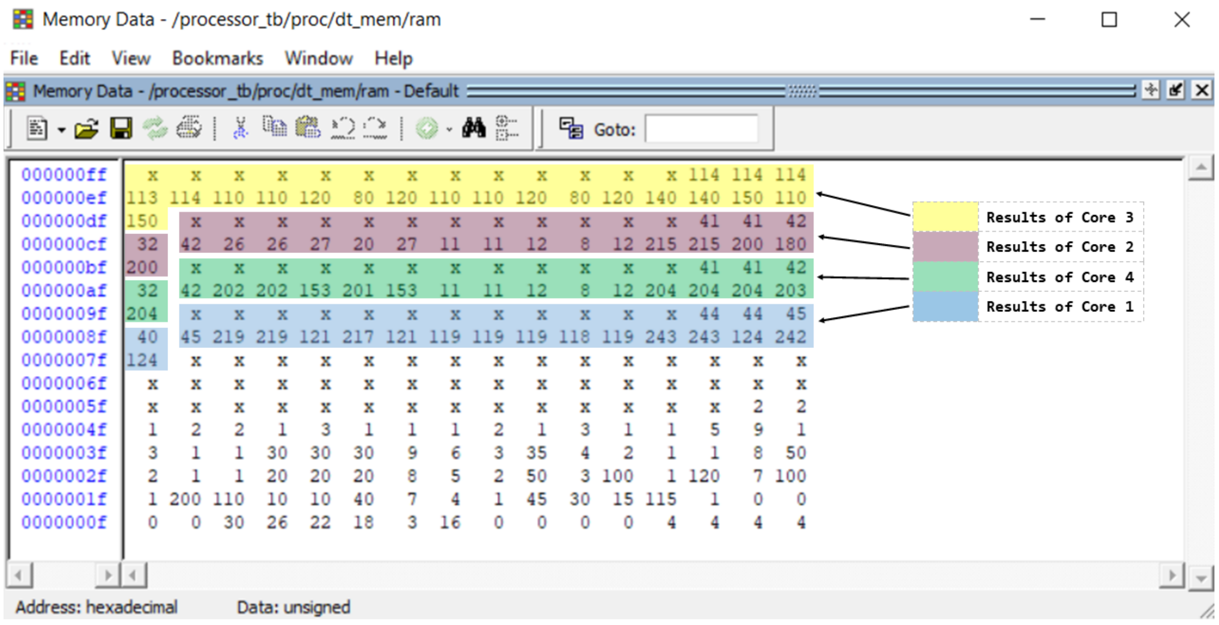

Memory Data - /processor_tb/proc/dt_mem/ram
File Edit View Bookmarks Window Help
Memory Data - /processor_tb/proc/dt_mem/ram - Default
Goto:

```
000000ff    x   x   x   x   x   x   x 114 114 114 113 114 110 110 120  80
000000ef  120   x   x   x   x   x   x  11  11  12   8  12 215 215 200 180
000000df  200   x   x   x   x   x   x  11  11  12   8  12 204 204 204 203
000000cf  204   x   x   x   x   x   x  44  44  45  40  45 219 219 121 217
000000bf  121   x   x   x   x   x   x 110 110 120  80 120 140 140 150 110
000000af  150   x   x   x   x   x   x  41  41  42  32  42  26  26  27  20
0000009f   27   x   x   x   x   x   x  41  41  42  32  42 202 202 153 201
0000008f  153   x   x   x   x   x   x 119 119 119 118 119 243 243 124 242
0000007f  124   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
0000006f    x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
0000005f    x   x   x   x   x   x   x   x   x   x   x   x   x   x   2   2
0000004f    1   2   2   1   3   1   1   1   2   1   3   1   1   5   9   1
0000003f    3   1   1  30  30  30   9   6   3  35   4   2   1   1   8  50
0000002f    2   1   1  20  20  20   8   5   2  50   3 100   1 120   7 100
0000001f    1 200 110  10  10  40   7   4   1  45  30  15 115   1  32  30
0000000f   28  26  24  22  20  18   3  16   2   2   2   2   2   2   2   2
```

⬛	Results of Core 6
🟨	Results of Core 3
⬜	Results of Core 7
🟪	Results of Core 2
🟥	Results of Core 5
🟩	Results of Core 4
🟧	Results of Core 8
🟦	Results of Core 1

Address: hexadecimal   Data: unsigned

Figure 6.2: DRAM output for 8-cores processing


Memory Data - /processor_tb/proc/dt_mem/ram
File Edit View Bookmarks Window Help
Memory Data - /processor_tb/proc/dt_mem/ram - Default
Goto:

```
000000ff    x   x   x   x   x   x   x   x   x   x   x   x   x 114 114 114
000000ef  113 114 110 110 120  80 120 110 110 120  80 120 140 140 150 110
000000df  150   x   x   x   x   x   x   x   x   x   x   x   x  41  41  42
000000cf   32  42  26  26  27  20  27  11  11  12   8  12 215 215 200 180
000000bf  200   x   x   x   x   x   x   x   x   x   x   x   x  41  41  42
000000af   32  42 202 202 153 201 153  11  11  12   8  12 204 204 204 203
0000009f  204   x   x   x   x   x   x   x   x   x   x   x   x  44  44  45
0000008f   40  45 219 219 121 217 121 119 119 119 118 119 243 243 124 242
0000007f  124   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
0000006f    x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
0000005f    x   x   x   x   x   x   x   x   x   x   x   x   x   x   2   2
0000004f    1   2   2   1   3   1   1   1   2   1   3   1   1   5   9   1
0000003f    3   1   1  30  30  30   9   6   3  35   4   2   1   1   8  50
0000002f    2   1   1  20  20  20   8   5   2  50   3 100   1 120   7 100
0000001f    1 200 110  10  10  40   7   4   1  45  30  15 115   1   0   0
0000000f    0   0  30  26  22  18   3  16   0   0   0   0   4   4   4   4
```

🟨	Results of Core 3
🟪	Results of Core 2
🟩	Results of Core 4
🟦	Results of Core 1

Address: hexadecimal   Data: unsigned

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 fouth 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.

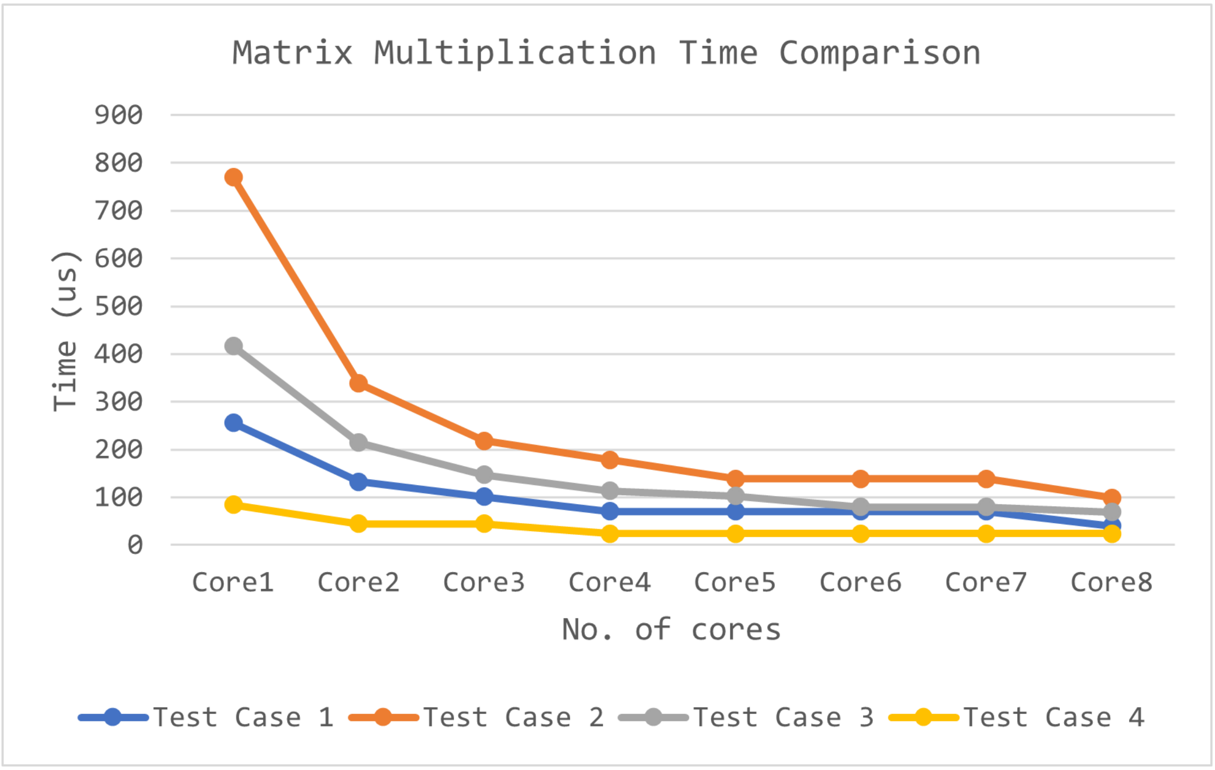


Figure 6.4: Matrix Multiplication Time Comparison

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 over come 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.

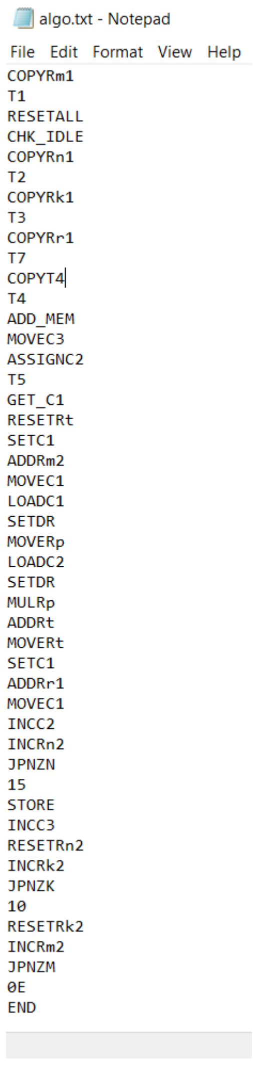
By changing the relevant parameters for memory size and register size, the capability of the processor can be extended. However, the primary purpose of this project is to compare the processing ability of different numbers of cores for matrix multiplication, which is achieved as shown in Section 6 above.

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.

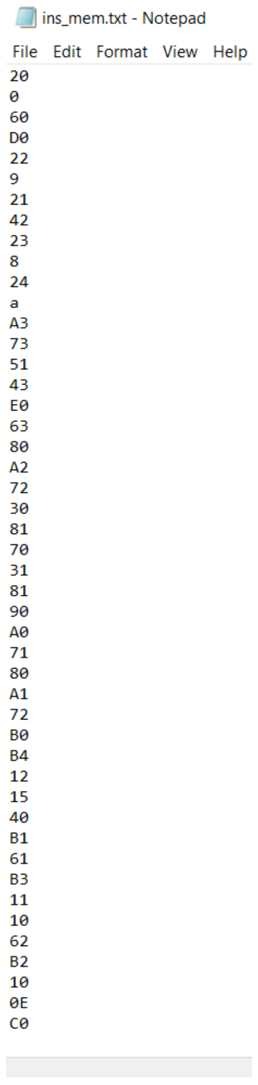
At the same time our python program creates a text file for the Data Memory, converting the raw input decimal values of the matrix elements into the base of hexadecimal.

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.



```
COPYRm1
T1
RESETALL
CHK_IDLE
COPYRn1
T2
COPYRk1
T3
COPYRr1
T7
COPYT4
T4
ADD_MEM
MOVEC3
ASSIGNC2
T5
GET_C1
RESETRt
SETC1
ADDRm2
MOVEC1
LOADC1
SETDR
MOVERp
LOADC2
SETDR
MULRp
ADDRt
MOVERt
SETC1
ADDRr1
MOVEC1
INCC2
INCRn2
JPNZN
15
STORE
INCC3
RESETRn2
INCRk2
JPNZK
10
RESETRk2
INCRm2
JPNZM
0E
END
```

(a) Algorithm Code



```
20
0
60
D0
22
9
21
42
23
8
24
a
A3
73
51
43
E0
63
80
A2
72
30
81
70
31
81
90
A0
71
80
A1
72
B0
B4
12
15
40
B1
61
B3
11
10
62
B2
10
0E
C0
```

(b) Instruction Memory File

Figure 8.1: Input and the Output of the Python Programme

9 References

1. Carpinelli, John D. Computer Systems Organization &amp; 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

```

module processor
# (parameter WIDTH = 8)
(
    input clk,
    output proc_state
);
// wire clk = CLOCK_50;
wire [(WIDTH)-1:0] INS;                                                //iROM output
wire [(WIDTH)-1:0] PC_addr;                                            //PC out to imem controller
wire [(WIDTH)-1:0] MEMWRITE_data;                                      //writing data to dram
wire [(WIDTH)-1:0] DRAM_addr;                                          //dram accessing memory location address
wire [(WIDTH)-1:0] MEMREAD_data;                                       //dram output

wire iROMREAD;
wire memWRITE;
wire memREAD;
wire imemAV1, imemAV2, imemAV3, imemAV4, imemAV5, imemAV6, imemAV7, imemAV8;
wire iROMREAD_1, iROMREAD_2, iROMREAD_3,iROMREAD_4, iROMREAD_5, iROMREAD_6, iROMREAD_7, iROMREAD_8;
wire coreS_1, coreS_2, coreS_3,coreS_4, coreS_5, coreS_6, coreS_7, coreS_8;
wire [WIDTH-1:0] PC_1, PC_2, PC_3,PC_4, PC_5, PC_6, PC_7, PC_8;
wire [WIDTH-1:0]INS_1, INS_2, INS_3,INS_4, INS_5, INS_6, INS_7, INS_8;
wire [WIDTH-1:0] AR_1, AR_2, AR_3,AR_4, AR_5, AR_6, AR_7, AR_8;
wire [WIDTH-1:0] DR_1, DR_2, DR_3, DR_4, DR_5, DR_6, DR_7, DR_8;
wire memREAD_1, memREAD_2, memREAD_3,memREAD_4, memREAD_5, memREAD_6, memREAD_7, memREAD_8;
wire memWE_1, memWE_2, memWE_3,memWE_4, memWE_5, memWE_6, memWE_7, memWE_8;
wire [WIDTH-1:0] MEM_1, MEM_2, MEM_3,MEM_4, MEM_5, MEM_6, MEM_7, MEM_8;
wire memAV1, memAV2, memAV3,memAV4, memAV5, memAV6, memAV7, memAV8;

// localparam MEMID_CORE1 = 8'd127;                                       //DRAM Store starting locations for respective cores
// localparam MEMID_CORE4 = 8'd159;
// localparam MEMID_CORE2 = 8'd191;
// localparam MEMID_CORE3 = 8'd223;

localparam MEMID_CORE1 = 8'd127;
localparam MEMID_CORE8 = 8'd143;
localparam MEMID_CORE4 = 8'd159;
localparam MEMID_CORE5 = 8'd175;
localparam MEMID_CORE2 = 8'd191;
localparam MEMID_CORE7 = 8'd207;
localparam MEMID_CORE3 = 8'd223;
localparam MEMID_CORE6 = 8'd239;



localparam COREID_1 = 3'd0;
localparam COREID_2 = 3'd1;
localparam COREID_3 = 3'd2;
localparam COREID_4 = 3'd3;
localparam COREID_5 = 3'd4;
localparam COREID_6 = 3'd5;
localparam COREID_7 = 3'd6;
localparam COREID_8 = 3'd7;

//instruction memory
ins_mem #(.ADDR_WIDTH(WIDTH), .INS_WIDTH(WIDTH)) ins_mem(.instruction(INS), .PC_address(PC_addr), .clk(clk), .rEn(iROMREAD));

//data memory
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));

imem_controller #(.WIDTH(WIDTH)) imem_c(.Clk(clk), .INS(INS), .rEN(iROMREAD), .PC_OUT(PC_addr),
 .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),
 .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),
 .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),
 .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),
 .imemAV1(imemAV1), .imemAV2(imemAV2), .imemAV3(imemAV3), .imemAV4(imemAV4), .imemAV5(imemAV5), .imemAV6(imemAV6), .imemAV7(imemAV7), .imemAV8(imemAV8));


dmem_controller #(.WIDTH(WIDTH)) dmem_c (.Clk(clk), .MEM(MEMREAD_data), .rEN(memREAD), .wEN(memWRITE), .DR_OUT(MEMWRITE_data), .addr(DRAM_addr),
.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),
.memAV1(memAV1), .memAV2(memAV2), .memAV3(memAV3), .memAV4(memAV4), .memAV5(memAV5), .memAV6(memAV6), .memAV7(memAV7), .memAV8(memAV8),
.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),
.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),
.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),
.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),
.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) );


core #(.WIDTH(WIDTH)) CORE_0 (.Clk(clk), .IROM_dataIn(INS_1), .DRAM_dataIn(MEM_1),.DRAM_dataOut(DR_1),
                            .IROM_addr(PC_1), .DRAM_addr(AR_1), .memREAD(memREAD_1), .memWRITE(memWE_1),
                            .iROMREAD(iROMREAD_1), .coreS(coreS_1),
                            .imemAV(imemAV1), .memAV(memAV1), .MEM_ID(MEMID_CORE1), .coreID(COREID_1));

core #(.WIDTH(WIDTH)) CORE_1 (.Clk(clk), .IROM_dataIn(INS_2), .DRAM_dataIn(MEM_2),.DRAM_dataOut(DR_2),
                            .IROM_addr(PC_2), .DRAM_addr(AR_2), .memREAD(memREAD_2), .memWRITE(memWE_2), .iROMREAD(iROMREAD_2), .coreS(coreS_2),
                            .imemAV(imemAV2), .memAV(memAV2), .MEM_ID(MEMID_CORE2), .coreID(COREID_2));

core #(.WIDTH(WIDTH)) CORE_2 (.Clk(clk), .IROM_dataIn(INS_3), .DRAM_dataIn(MEM_3),.DRAM_dataOut(DR_3),
                              .IROM_addr(PC_3), .DRAM_addr(AR_3), .memREAD(memREAD_3), .memWRITE(memWE_3), .iROMREAD(iROMREAD_3), .coreS(coreS_3),
                              .imemAV(imemAV3), .memAV(memAV3), .MEM_ID(MEMID_CORE3), .coreID(COREID_3));

core #(.WIDTH(WIDTH)) CORE_3 (.Clk(clk), .IROM_dataIn(INS_4), .DRAM_dataIn(MEM_4),.DRAM_dataOut(DR_4),
                              .IROM_addr(PC_4), .DRAM_addr(AR_4), .memREAD(memREAD_4), .memWRITE(memWE_4), .iROMREAD(iROMREAD_4), .coreS(coreS_4),
                              .imemAV(imemAV4), .memAV(memAV4), .MEM_ID(MEMID_CORE4), .coreID(COREID_4));

core #(.WIDTH(WIDTH)) CORE_4 (.Clk(clk), .IROM_dataIn(INS_5), .DRAM_dataIn(MEM_5),.DRAM_dataOut(DR_5),
                              .IROM_addr(PC_5), .DRAM_addr(AR_5), .memREAD(memREAD_5), .memWRITE(memWE_5), .iROMREAD(iROMREAD_5), .coreS(coreS_5),
                              .imemAV(imemAV5), .memAV(memAV5), .MEM_ID(MEMID_CORE5), .coreID(COREID_5));

core #(.WIDTH(WIDTH)) CORE_5 (.Clk(clk), .IROM_dataIn(INS_6), .DRAM_dataIn(MEM_6),.DRAM_dataOut(DR_6),
                              .IROM_addr(PC_6), .DRAM_addr(AR_6), .memREAD(memREAD_6), .memWRITE(memWE_6), .iROMREAD(iROMREAD_6), .coreS(coreS_6),
                              .imemAV(imemAV6), .memAV(memAV6), .MEM_ID(MEMID_CORE6), .coreID(COREID_6));

core #(.WIDTH(WIDTH)) CORE_6 (.Clk(clk), .IROM_dataIn(INS_7), .DRAM_dataIn(MEM_7),.DRAM_dataOut(DR_7),
                              .IROM_addr(PC_7), .DRAM_addr(AR_7), .memREAD(memREAD_7), .memWRITE(memWE_7), .iROMREAD(iROMREAD_7), .coreS(coreS_7),
                              .imemAV(imemAV4), .memAV(memAV7), .MEM_ID(MEMID_CORE7), .coreID(COREID_7));

core #(.WIDTH(WIDTH)) CORE_7 (.Clk(clk), .IROM_dataIn(INS_8), .DRAM_dataIn(MEM_8),.DRAM_dataOut(DR_8),
                              .IROM_addr(PC_8), .DRAM_addr(AR_8), .memREAD(memREAD_8), .memWRITE(memWE_8), .iROMREAD(iROMREAD_8), .coreS(coreS_8),
                              .imemAV(imemAV8), .memAV(memAV8), .MEM_ID(MEMID_CORE8), .coreID(COREID_8));
assign proc_state = (coreS_1 && coreS_2 && coreS_3 && coreS_4 && coreS_5 && coreS_6 && coreS_7 && coreS_8);

endmodule
```

10.2 Core

```
`include "proc_param.v"

module core

#(parameter WIDTH = 8)(
    input Clk,
    input [WIDTH-1:0]IROM_dataIn,                                  // --> IR or --> AR
    input [WIDTH-1:0]DRAM_dataIn,                                  // --> DR
    input [WIDTH-1:0] MEM_ID,
    input [2:0] coreID,
    input imemAV, memAV,
    output [WIDTH-1:0]DRAM_dataOut,                                // from DR
    output [WIDTH-1:0]IROM_addr,                                   // PC
    output [WIDTH-1:0]DRAM_addr,                                   // AR
    output wire memREAD, memWRITE, iROMREAD, coreS
);

```

```verilog
wire [14:0] wEN;
wire [5:0] INC;
wire [4:0] RST;
wire [2:0] compMUX;
wire [3:0] aluOP;
wire zFlag;
wire selAR;
wire [4:0]busMUX;
wire coreINC_AR;

wire [WIDTH-1:0]INS;                                                  // instruction from iROM
wire [WIDTH-1:0] COMP_IN1;
wire [WIDTH-1:0] COMP_IN2;
wire [WIDTH-1:0]PC_OUT;
wire [WIDTH-1:0]AR_OUT;
wire [WIDTH-1:0]DR_OUT;
wire [WIDTH-1:0]RP_OUT;
wire [WIDTH-1:0]RT_OUT;
wire [WIDTH-1:0]RT4_OUT;
wire [WIDTH-1:0]RR_OUT;
wire [WIDTH-1:0]RM1_OUT;
wire [WIDTH-1:0]RK1_OUT;
wire [WIDTH-1:0]RN1_OUT;
wire [WIDTH-1:0]RM2_OUT;
wire [WIDTH-1:0]RK2_OUT;
wire [WIDTH-1:0]RN2_OUT;
wire [WIDTH-1:0]C1_OUT;
wire [WIDTH-1:0]C2_OUT;
wire [WIDTH-1:0]C3_OUT;
wire [WIDTH-1:0]AC_OUT;
wire [WIDTH-1:0]ALU_OUT;
wire [WIDTH-1:0]BUSMUX_OUT;
//PC                                                      15                  5
PC #(.WIDTH(WIDTH)) PC (.Clk(Clk), .WEN(wEN[`PC_W]), .INC(INC[`PC_INC]), .BusOut(AR_OUT), .dout(PC_OUT));
//IR                                                      14
Reg_module_W #(.WIDTH(WIDTH)) IR (.Clk(Clk), .WEN(wEN[`IR_W]), .BusOut(IROM_dataIn), .dout(INS));
//AR                                            13
AR #(.WIDTH(WIDTH)) AR (.Clk(Clk), .WEN(wEN[`AR_W]), .BusOut(BUSMUX_OUT), .IOut(IROM_dataIn), .selAR(selAR), .dout(AR_OUT), .coreID(coreID), .coreINC_AR(coreINC_AR));
//DR                                                      12
Reg_module_W #(.WIDTH(WIDTH)) DR (.Clk(Clk), .WEN(wEN[`DR_W]), .BusOut(BUSMUX_OUT), .dout(DR_OUT));
//RP                                                      11
Reg_module_W #(.WIDTH(WIDTH)) RP (.Clk(Clk), .WEN(wEN[`RP_W]), .BusOut(BUSMUX_OUT), .dout(RP_OUT));
//RT                                                      10
Reg_module_RW #(.WIDTH(WIDTH)) RT (.Clk(Clk), .WEN(wEN[`RT_W]), .RST(RST[`RT_RST]), .BusOut(BUSMUX_OUT), .dout(RT_OUT));
//RT4                                                     16
Reg_module_W #(.WIDTH(WIDTH)) RT4 (.Clk(Clk), .WEN(wEN[`RT4_W]), .BusOut(BUSMUX_OUT), .dout(RT4_OUT));
//Rr                                                     16
Reg_module_W #(.WIDTH(WIDTH)) RR (.Clk(Clk), .WEN(wEN[`RR_W]), .BusOut(BUSMUX_OUT), .dout(RR_OUT));
//RM1                                                     9
Reg_module_W #(.WIDTH(WIDTH)) RM1 (.Clk(Clk), .WEN(wEN[`RM1_W]), .BusOut(BUSMUX_OUT), .dout(RM1_OUT));
//RK1                                                     8
Reg_module_W #(.WIDTH(WIDTH)) RK1 (.Clk(Clk), .WEN(wEN[`RK1_W]), .BusOut(
    BUSMUX_OUT), .dout(RK1_OUT));
//RN1                                               7
Reg_module_W #(.WIDTH(WIDTH)) RN1 (.Clk(Clk), .WEN(wEN[`RN1_W]), .BusOut(BUSMUX_OUT), .dout(RN1_OUT));
//RM2
Reg_module_RI #(.WIDTH(WIDTH)) RM2 (.Clk(Clk), .RST(RST[`RM2_RST]), .INC(INC[`RM2_INC]), .BusOut(BUSMUX_OUT), .dout(RM2_OUT));
//RK2
Reg_module_RI #(.WIDTH(WIDTH)) RK2 (.Clk(Clk), .RST(RST[`RK2_RST]), .INC(INC[`RK2_INC]), .BusOut(BUSMUX_OUT), .dout(RK2_OUT));
//RN2
Reg_module_RI #(.WIDTH(WIDTH)) RN2 (.Clk(Clk), .RST(RST[`RN2_RST]), .INC(INC[`RN2_INC]), .BusOut(BUSMUX_OUT), .dout(RN2_OUT));
//RC1                                               3
Reg_module_W #(.WIDTH(WIDTH)) RC1 (.Clk(Clk), .WEN(wEN[`RC1_W]), .BusOut(BUSMUX_OUT), .dout(C1_OUT));
//RC2                                               2                   1
Reg_module_WI #(.WIDTH(WIDTH)) RC2 (.Clk(Clk), .WEN(wEN[`RC2_W]), .INC(INC[`RC2_INC]), .BusOut(BUSMUX_OUT), .dout(C2_OUT));
//RC3                                               1                   0
Reg_module_WI #(.WIDTH(WIDTH)) RC3 (.Clk(Clk), .WEN(wEN[`RC3_W]), .INC(INC[`RC3_INC]), .BusOut(BUSMUX_OUT), .dout(C3_OUT));
//AC                                                 0
Reg_module_RW #(.WIDTH(WIDTH)) AC (.Clk(Clk), .WEN(wEN[`AC_W]), .RST(RST[`AC_RST]), .BusOut(ALU_OUT), .dout(AC_OUT));

//ALU
Alu #(.WIDTH(WIDTH)) ALU (.AC(AC_OUT), .BusOut(BUSMUX_OUT), .result_ac(ALU_OUT), .ALU_OP(aluOP), .MEM_ID(MEM_ID));

mux_3to1_8bit  COMPMUX1 (.mux_inN(RN1_OUT), .mux_inK(RK1_OUT), .mux_inM(RM1_OUT), .mux_sel(compMUX), .mux_out(COMP_IN1));
mux_3to1_8bit  COMPMUX2 (.mux_inN(RN2_OUT), .mux_inK(RK2_OUT), .mux_inM(RM2_OUT), .mux_sel(compMUX), .mux_out(COMP_IN2));
Comp #(.WIDTH(WIDTH)) COMP (.R1(COMP_IN1), .R2(COMP_IN2), .z(zFlag));

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));
//CU
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));


assign IROM_addr = PC_OUT;
assign DRAM_dataOut = DR_OUT;
assign DRAM_addr = AR_OUT;
endmodule
```

10.3 Data Memory Controller

```
module dmem_controller#(parameter WIDTH=8)(
    input Clk,
    input coreS_1, coreS_2, coreS_3,coreS_4, coreS_5, coreS_6, coreS_7, coreS_8,
    input [WIDTH-1:0] AR_1, AR_2, AR_3,AR_4, AR_5, AR_6, AR_7, AR_8,
    input [WIDTH-1:0] DR_1, DR_2, DR_3, DR_4, DR_5, DR_6, DR_7, DR_8,
    input [WIDTH-1:0]MEM,                                   //from DRAM
    input memREAD_1, memREAD_2, memREAD_3,memREAD_4, memREAD_5, memREAD_6, memREAD_7, memREAD_8,
    input memWE_1, memWE_2, memWE_3,memWE_4, memWE_5, memWE_6, memWE_7, memWE_8,
    output reg rEN,
    output reg wEN,
    output reg [WIDTH-1:0] MEM_1, MEM_2, MEM_3,MEM_4, MEM_5, MEM_6, MEM_7, MEM_8
,
    output reg [WIDTH-1:0] addr,                                    //to DRAM
    output reg [WIDTH-1:0] DR_OUT,
    output reg memAV1, memAV2, memAV3,memAV4, memAV5, memAV6, memAV7, memAV8
                    //to cores
);
localparam NORM = 5'b00000;
localparam NORMEND =5'b00001;

localparam AR_1_2 = 5'b00010;
localparam AR_2_1 = 5'b00011;
localparam AR_2_2 = 5'b00100;
localparam AR_3_1 = 5'b00101;
localparam AR_3_2 = 5'b00110;
localparam AR_4_1 = 5'b00111;
localparam AR_4_2 = 5'b01000;
localparam AR_5_1 = 5'b01001;
localparam AR_5_2 = 5'b01010;
localparam AR_6_1 = 5'b01011;
localparam AR_6_2 = 5'b01100;
localparam AR_7_1 = 5'b01101;
localparam AR_7_2 = 5'b01110;
localparam AR_8_1 = 5'b01111;
localparam AR_8_2 = 5'b10000;

localparam DR_1_1 = 5'b10001;
localparam DR_1_2 = 5'b10010;
localparam DR_1_3 = 5'b10011;
localparam DR_1_4 = 5'b10100;
localparam DR_1_5 = 5'b10101;
localparam DR_1_6 = 5'b10110;
localparam DR_1_7 = 5'b10111;

reg [4:0] NEXT_STATE_DC=NORM;
reg [4:0] STATE_DC = NORM;

always @(posedge Clk) begin
    STATE_DC=NEXT_STATE_DC;
    case (STATE_DC)
     NORM: begin
        memAV1 <= 0;
        memAV2 <= 0;
        memAV3 <= 0;
        memAV4 <= 0;
        memAV5 <= 0;
        memAV6 <= 0;
        memAV7 <= 0;
        memAV8 <= 0;
        rEN <= 0;
        wEN <= 0;
        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
            if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
    memREAD_5==1 && memREAD_6==1 && memREAD_7==1 && memREAD_8==1) begin
                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 && AR_7==AR_8) begin // Active 8 cores, same addresses
                    rEN <= 1;
                    addr <= AR_1;
                    NEXT_STATE_DC <= NORMEND;
                    memAV1 <= 1;
                    memAV2 <= 1;
                    memAV3 <= 1;
                    memAV4 <= 1;
                    memAV5 <= 1;
                    memAV6 <= 1;
                    memAV7 <= 1;
                    memAV8 <= 1;

                end
                else begin          // Active 8 cores, different addresses
                    rEN <= 1;
                    addr <= AR_1;
                    NEXT_STATE_DC <= AR_1_2;
                end
              end
              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
                  wEN <= 1;
                  addr <= AR_1;
                  DR_OUT <= DR_1;
                  memAV1 <= 0;
                  memAV2 <= 0;
                  memAV3 <= 0;
                  memAV4 <= 0;
                  memAV5 <= 0;
                  memAV6 <= 0;
                  memAV7 <= 0;
                  memAV8 <= 0;
                  NEXT_STATE_DC <= DR_1_1;
              end
          end
          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
              if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
     memREAD_5==1 && memREAD_6==1 && memREAD_7==1) begin
                  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
                    rEN <= 1;
                    addr <= AR_1;
                    NEXT_STATE_DC <= NORMEND;
                    memAV1 <= 1;
                    memAV2 <= 1;
                    memAV3 <= 1;
                    memAV4 <= 1;
                    memAV5 <= 1;
                    memAV6 <= 1;
                    memAV7 <= 1;
                end
                else begin
                    rEN <= 1;
                    addr <= AR_1;
                    NEXT_STATE_DC <= AR_1_2;
                end
              end
              else if(memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1 &&
    memWE_5==1 && memWE_6==1 && memWE_7==1) begin
                  wEN <= 1;
                  addr <= AR_1;
                  DR_OUT <= DR_1;
                  memAV1 <= 0;
                  memAV2 <= 0;
                  memAV3 <= 0;
                  memAV4 <= 0;
```

```
                memAV5 <= 0;
                memAV6 <= 0;
                memAV7 <= 0;
                NEXT_STATE_DC <= DR_1_1;
            end

        end

        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
            if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
 memREAD_5==1 && memREAD_6==1) begin
                if(AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4 && AR_4==AR_5 &&
AR_5==AR_6) begin
                  rEN <= 1;
                  addr <= AR_1;
                  NEXT_STATE_DC <= NORMEND;
                  memAV1 <= 1;
                  memAV2 <= 1;
                  memAV3 <= 1;
                  memAV4 <= 1;
                  memAV5 <= 1;
                  memAV6 <= 1;
              end
              else begin
                  rEN <= 1;
                  addr <= AR_1;
                  NEXT_STATE_DC <= AR_1_2;
              end
            end
            else if(memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1 &&
memWE_5==1 && memWE_6==1) begin
                wEN <= 1;
                addr <= AR_1;
                DR_OUT <= DR_1;
                memAV1 <= 0;
                memAV2 <= 0;
                memAV3 <= 0;
                memAV4 <= 0;
                memAV5 <= 0;
                memAV6 <= 0;
                NEXT_STATE_DC <= DR_1_1;
            end

        end

        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
            if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1 &&
 memREAD_5==1) begin
                if(AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4 && AR_4==AR_5) begin
                  rEN <= 1;
                  addr <= AR_1;
                  NEXT_STATE_DC <= NORMEND;
                  memAV1 <= 1;
                  memAV2 <= 1;
                  memAV3 <= 1;
                  memAV4 <= 1;
                  memAV5 <= 1;
                end
                else begin
                  rEN <= 1;
                  addr <= AR_1;
```

```
         NEXT_STATE_DC <= AR_1_2;
        end
      end

      else if(memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1 && memWE_5==1) begin
        wEN <= 1;
        addr <= AR_1;
        DR_OUT <= DR_1;
        memAV1 <= 0;
        memAV2 <= 0;
        memAV3 <= 0;
        memAV4 <= 0;
        memAV5 <= 0;
        NEXT_STATE_DC <= DR_1_1;
      end
    end
    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
      if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1 && memREAD_4==1) begin
        if(AR_1==AR_2 && AR_2==AR_3 && AR_3==AR_4) begin
         rEN <= 1;
         addr <= AR_1;
         NEXT_STATE_DC <= NORMEND;
         memAV1 <= 1;
         memAV2 <= 1;
         memAV3 <= 1;
         memAV4 <= 1;
        end
        else begin
         rEN <= 1;
         addr <= AR_1;
         NEXT_STATE_DC <= AR_1_2;
        end
      end

      else if(memWE_1==1 && memWE_2==1 && memWE_3==1 && memWE_4==1) begin
        wEN <= 1;
        addr <= AR_1;
        DR_OUT <= DR_1;
        memAV1 <= 0;
        memAV2 <= 0;
        memAV3 <= 0;
        memAV4 <= 0;
        NEXT_STATE_DC <= DR_1_1;
      end

    end
    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
      if(memREAD_1==1 && memREAD_2==1 && memREAD_3==1) begin
        if(AR_1==AR_2 && AR_2==AR_3) begin
         rEN <= 1;
         addr <= AR_1;
         NEXT_STATE_DC <= NORMEND;
         memAV1 <= 1;
         memAV2 <= 1;
         memAV3 <= 1;
        end
        else begin
         rEN <= 1;
```

```
                  addr <= AR_1;
                  NEXT_STATE_DC <= AR_1_2;
                end
            end

            else if(memWE_1==1 && memWE_2==1 && memWE_3==1) begin
                wEN <= 1;
                addr <= AR_1;
                DR_OUT <= DR_1;
                memAV1 <= 0;
                memAV2 <= 0;
                memAV3 <= 0;
                NEXT_STATE_DC <= DR_1_1;
            end

        end

        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
            if(memREAD_1==1 && memREAD_2==1) begin
                if(AR_1==AR_2) begin
                  rEN <= 1;
                  addr <= AR_1;
                  NEXT_STATE_DC <= NORMEND;
                  memAV1 <= 1;
                  memAV2 <= 1;
                end
                else begin
                  rEN <= 1;
                  addr <= AR_1;
                  NEXT_STATE_DC <= AR_1_2;
                end
            end
            else if(memWE_1==1 && memWE_2==1) begin
                wEN <= 1;
                addr <= AR_1;
                DR_OUT <= DR_1;
                memAV1 <= 0;
                memAV2 <= 0;
                NEXT_STATE_DC <= DR_1_1;
            end

        end

        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
            if(memREAD_1==1) begin
                  rEN <= 1;
                  addr <= AR_1;
                  NEXT_STATE_DC <= NORMEND;
                  memAV1 <= 1;
            end
            else if(memWE_1==1) begin
                wEN <= 1;
                addr <= AR_1;
                DR_OUT <= DR_1;
                memAV1 <= 1;
                NEXT_STATE_DC <= NORM;
            end
        end

    end
    NORMEND: begin
```

```
        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
            MEM_1 <= MEM;
            MEM_2 <= MEM;
            MEM_3 <= MEM;
            MEM_4 <= MEM;
            MEM_5 <= MEM;
            MEM_6 <= MEM;
            MEM_7 <= MEM;
            MEM_8 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
            memAV6 <= 0;
            memAV7 <= 0;
            memAV8 <= 0;
        end
        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
            MEM_1 <= MEM;
            MEM_2 <= MEM;
            MEM_3 <= MEM;
            MEM_4 <= MEM;
            MEM_5 <= MEM;
            MEM_6 <= MEM;
            MEM_7 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
            memAV6 <= 0;
            memAV7 <= 0;
        end
        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
            MEM_1 <= MEM;
            MEM_2 <= MEM;
            MEM_3 <= MEM;
            MEM_4 <= MEM;
            MEM_5 <= MEM;
            MEM_6 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
            memAV6 <= 0;
        end
        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
            MEM_1 <= MEM;
            MEM_2 <= MEM;
            MEM_3 <= MEM;
            MEM_4 <= MEM;
            MEM_5 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
        end
        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
            MEM_1 <= MEM;
            MEM_2 <= MEM;
            MEM_3 <= MEM;
            MEM_4 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
        end
        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
            MEM_1 <= MEM;
            MEM_2 <= MEM;
            MEM_3 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
            memAV2 <= 0;
            memAV3 <= 0;
        end
        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
            MEM_1 <= MEM;
            MEM_2 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
            memAV2 <= 0;
        end
        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
            MEM_1 <= MEM;
            memAV1 <= 0;          // memAV = 0 in AR_1
        end
        NEXT_STATE_DC <= NORM;

    end

    AR_1_2: begin
        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
            memAV6 <= 0;
            memAV7 <= 0;
            memAV8 <= 0;
            NEXT_STATE_DC <= AR_2_1;

        end
        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
            memAV6 <= 0;
            memAV7 <= 0;
            NEXT_STATE_DC <= AR_2_1;

          end

        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
            memAV6 <= 0;
            NEXT_STATE_DC <= AR_2_1;

        end

        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            memAV5 <= 0;
            NEXT_STATE_DC <= AR_2_1;

        end
        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            memAV2 <= 0;
            memAV3 <= 0;
            memAV4 <= 0;
            NEXT_STATE_DC <= AR_2_1;

        end
        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            memAV2 <= 0;
            memAV3 <= 0;
            NEXT_STATE_DC <= AR_2_1;

        end
        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            memAV2 <= 0;
            NEXT_STATE_DC <= AR_2_1;

        end
        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
            MEM_1 <= MEM;
            memAV1 <= 0;
            NEXT_STATE_DC <= NORM;

        end

      end
```

```

        AR_2_1: begin
            rEN  <= 1;
            addr  <= AR_2;
            NEXT_STATE_DC  <= AR_2_2;
            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
              memAV1  <= 1;
              memAV2  <= 1;
            end
        end

        AR_2_2: begin
          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
              MEM_2  <= MEM;
              memAV1  <= 0;
              memAV2  <= 0;
              memAV3  <= 0;
              memAV4  <= 0;
              memAV5  <= 0;
              memAV6  <= 0;
              memAV7  <= 0;
              memAV8  <= 0;
              NEXT_STATE_DC  <= AR_3_1;
          end

          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
              MEM_2  <= MEM;
              memAV1  <= 0;
              memAV2  <= 0;
              memAV3  <= 0;
              memAV4  <= 0;
              memAV5  <= 0;
              memAV6  <= 0;
              memAV7  <= 0;
              NEXT_STATE_DC  <= AR_3_1;
          end

          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
              MEM_2  <= MEM;
              memAV1  <= 0;
              memAV2  <= 0;
              memAV3  <= 0;
              memAV4  <= 0;
              memAV5  <= 0;
              memAV6  <= 0;
              NEXT_STATE_DC  <= AR_3_1;
          end
          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
              MEM_2  <= MEM;
              memAV1  <= 0;
              memAV2  <= 0;
              memAV3  <= 0;
              memAV4  <= 0;
              memAV5  <= 0;
              NEXT_STATE_DC  <= AR_3_1;
          end
          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
              MEM_2 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              NEXT_STATE_DC <= AR_3_1;
          end
          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
              MEM_2 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              NEXT_STATE_DC <= AR_3_1;
          end
          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
              MEM_2 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              NEXT_STATE_DC <= NORM;
          end

        end

        AR_3_1: begin
          rEN <= 1;
          addr <= AR_3;
          NEXT_STATE_DC <= AR_3_2;
          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
              memAV1 <= 1;
              memAV2 <= 1;
              memAV3 <= 1;
          end
        end

        AR_3_2: begin
            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
              MEM_3 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              memAV8 <= 0;
              NEXT_STATE_DC <= AR_4_1;
            end

            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
              MEM_3 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              NEXT_STATE_DC <= AR_4_1;
            end

          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
              MEM_3 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              NEXT_STATE_DC <= AR_4_1;
            end
            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
              MEM_3 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              NEXT_STATE_DC <= AR_4_1;
            end
            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
              MEM_3 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              NEXT_STATE_DC <= AR_4_1;
            end
            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
              MEM_3 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              NEXT_STATE_DC <= NORM;
            end



        end

        AR_4_1: begin
          rEN <= 1;
          addr <= AR_4;
          NEXT_STATE_DC <= AR_4_2;
          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
              memAV1 <= 1;
              memAV2 <= 1;
              memAV3 <= 1;
              memAV4 <= 1;
          end

        end

        AR_4_2:begin
            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
              MEM_4 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              memAV8 <= 0;
              NEXT_STATE_DC <= AR_5_1;
            end

            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
              MEM_4 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              NEXT_STATE_DC <= AR_5_1;
            end
            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
              MEM_4 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              NEXT_STATE_DC <= AR_5_1;
            end
            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
              MEM_4 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              NEXT_STATE_DC <= AR_5_1;
            end
            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
              MEM_4 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              NEXT_STATE_DC <= NORM;
            end

        end

        AR_5_1: begin
          rEN <= 1;
```

```
          addr <= AR_5;
          NEXT_STATE_DC <= AR_5_2;
          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
              memAV1 <= 1;
              memAV2 <= 1;
              memAV3 <= 1;
              memAV4 <= 1;
              memAV5 <= 1;
          end

        end

         AR_5_2:begin
            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
              MEM_5 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              memAV8 <= 0;
              NEXT_STATE_DC <= AR_6_1;
            end
            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
              MEM_5 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              NEXT_STATE_DC <= AR_6_1;
            end
            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
              MEM_5 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              NEXT_STATE_DC <= AR_6_1;
            end
            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
              MEM_5 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              NEXT_STATE_DC <= NORM;
            end
        end

         AR_6_1: begin
```

```
          rEN <= 1;
          addr <= AR_6;
          NEXT_STATE_DC <= AR_6_2;
          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
              memAV1 <= 1;
              memAV2 <= 1;
              memAV3 <= 1;
              memAV4 <= 1;
              memAV5 <= 1;
              memAV6 <= 1;
          end

        end

        AR_6_2:begin
            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
              MEM_6 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              memAV8 <= 0;
              NEXT_STATE_DC <= AR_7_1;
            end
            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
              MEM_6 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              NEXT_STATE_DC <= AR_7_1;
            end

            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
              MEM_6 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              NEXT_STATE_DC <= NORM;
            end
        end

        AR_7_1: begin
          rEN <= 1;
          addr <= AR_7;
          NEXT_STATE_DC <= AR_7_2;
          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
              memAV1 <= 1;
              memAV2 <= 1;
              memAV3 <= 1;
              memAV4 <= 1;
              memAV5 <= 1;
              memAV6 <= 1;
              memAV7 <= 1;
          end

        end

        AR_7_2:begin
            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
              MEM_7 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              memAV8 <= 0;
              NEXT_STATE_DC <= AR_8_1;
            end
            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
              MEM_7 <= MEM;
              memAV1 <= 0;
              memAV2 <= 0;
              memAV3 <= 0;
              memAV4 <= 0;
              memAV5 <= 0;
              memAV6 <= 0;
              memAV7 <= 0;
              NEXT_STATE_DC <= NORM;
            end
        end

        AR_8_1: begin
          rEN <= 1;
          addr <= AR_8;
          NEXT_STATE_DC <= AR_8_2;
          memAV1 <= 1;
          memAV2 <= 1;
          memAV3 <= 1;
          memAV4 <= 1;
          memAV5 <= 1;
          memAV6 <= 1;
          memAV7 <= 1;
          memAV8 <= 1;

        end

        AR_8_2:begin
          MEM_8 <= MEM;
          memAV1 <= 0;
          memAV2 <= 0;
          memAV3 <= 0;
          memAV4 <= 0;
          memAV5 <= 0;
          memAV6 <= 0;
          memAV7 <= 0;
          memAV8 <= 0;
          NEXT_STATE_DC <= NORM;

          end




          DR_1_1:begin
            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
             wEN <= 1;
             addr <= AR_2;
             DR_OUT <= DR_2;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             memAV4 <= 0;
             memAV5 <= 0;
             memAV6 <= 0;
             memAV7 <= 0;
             memAV8 <= 0;

             NEXT_STATE_DC <= DR_1_2;
            end

            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
             wEN <= 1;
             addr <= AR_2;
             DR_OUT <= DR_2;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             memAV4 <= 0;
             memAV5 <= 0;
             memAV6 <= 0;
             memAV7 <= 0;

             NEXT_STATE_DC <= DR_1_2;
            end
            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
             wEN <= 1;
             addr <= AR_2;
             DR_OUT <= DR_2;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             memAV4 <= 0;
             memAV5 <= 0;
             memAV6 <= 0;
             NEXT_STATE_DC <= DR_1_2;
            end
            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
             wEN <= 1;
             addr <= AR_2;
             DR_OUT <= DR_2;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             memAV4 <= 0;
             memAV5 <= 0;
             NEXT_STATE_DC <= DR_1_2;
            end
            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
             wEN <= 1;
             addr <= AR_2;
             DR_OUT <= DR_2;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             memAV4 <= 0;
             NEXT_STATE_DC <= DR_1_2;
            end
            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
             wEN <= 1;
             addr <= AR_2;
             DR_OUT <= DR_2;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             NEXT_STATE_DC <= DR_1_2;
            end
            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
             wEN <= 1;
             addr <= AR_2;
             DR_OUT <= DR_2;
             memAV1 <= 1;
             memAV2 <= 1;
             NEXT_STATE_DC <= NORM;
            end


          end

          DR_1_2:begin
            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
             wEN <= 1;
             addr <= AR_3;
             DR_OUT <= DR_3;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             memAV4 <= 0;
             memAV5 <= 0;
             memAV6 <= 0;
             memAV7 <= 0;
             memAV8 <= 0;

             NEXT_STATE_DC <= DR_1_3;
            end
            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
             wEN <= 1;
             addr <= AR_3;
             DR_OUT <= DR_3;
             memAV1 <= 0;
             memAV2 <= 0;
             memAV3 <= 0;
             memAV4 <= 0;
             memAV5 <= 0;
             memAV6 <= 0;
```

```
            memAV7  <=  0;

            NEXT_STATE_DC  <=  DR_1_3;
          end
          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
            wEN  <=  1;
            addr  <=  AR_3;
            DR_OUT  <=  DR_3;
            memAV1  <=  0;
            memAV2  <=  0;
            memAV3  <=  0;
            memAV4  <=  0;
            memAV5  <=  0;
            memAV6  <=  0;
            NEXT_STATE_DC  <=  DR_1_3;
          end
          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
            wEN  <=  1;
            addr  <=  AR_3;
            DR_OUT  <=  DR_3;
            memAV1  <=  0;
            memAV2  <=  0;
            memAV3  <=  0;
            memAV4  <=  0;
            memAV5  <=  0;
            NEXT_STATE_DC  <=  DR_1_3;
          end
          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
            wEN  <=  1;
            addr  <=  AR_3;
            DR_OUT  <=  DR_3;
            memAV1  <=  0;
            memAV2  <=  0;
            memAV3  <=  0;
            memAV4  <=  0;
            memAV5  <=  0;
            NEXT_STATE_DC  <=  DR_1_3;
          end
          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
            wEN  <=  1;
            addr  <=  AR_3;
            DR_OUT  <=  DR_3;
            memAV1  <=  0;
            memAV2  <=  0;
            memAV3  <=  0;
            memAV4  <=  0;
            NEXT_STATE_DC  <=  DR_1_3;
          end
          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
            wEN  <=  1;
            addr  <=  AR_3;
            DR_OUT  <=  DR_3;
            memAV1  <=  1;
            memAV2  <=  1;
            memAV3  <=  1;
            NEXT_STATE_DC  <=  NORM;
          end
        end
```

```

      DR_1_3:begin
        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
          wEN <= 1;
          addr <= AR_4;
          DR_OUT <= DR_4;
          memAV1 <= 0;
          memAV2 <= 0;
          memAV3 <= 0;
          memAV4 <= 0;
          memAV5 <= 0;
          memAV6 <= 0;
          memAV7 <= 0;
          memAV8 <= 0;

          NEXT_STATE_DC <= DR_1_4;
        end
        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
          wEN <= 1;
          addr <= AR_4;
          DR_OUT <= DR_4;
          memAV1 <= 0;
          memAV2 <= 0;
          memAV3 <= 0;
          memAV4 <= 0;
          memAV5 <= 0;
          memAV6 <= 0;
          memAV7 <= 0;
          NEXT_STATE_DC <= DR_1_4;
        end
        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
          wEN <= 1;
          addr <= AR_4;
          DR_OUT <= DR_4;
          memAV1 <= 0;
          memAV2 <= 0;
          memAV3 <= 0;
          memAV4 <= 0;
          memAV5 <= 0;
          memAV6 <= 0;
          NEXT_STATE_DC <= DR_1_4;
        end
        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
          wEN <= 1;
          addr <= AR_4;
          DR_OUT <= DR_4;
          memAV1 <= 0;
          memAV2 <= 0;
          memAV3 <= 0;
          memAV4 <= 0;
          memAV5 <= 0;
          NEXT_STATE_DC <= DR_1_4;
        end
        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
          wEN <= 1;
          addr <= AR_4;
          DR_OUT <= DR_4;
          memAV1 <= 1;
```

```
       memAV2 <= 1;
       memAV3 <= 1;
       memAV4 <= 1;
       NEXT_STATE_DC <= NORM;
     end
   end

   DR_1_4:begin
     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
       wEN <= 1;
       addr <= AR_5;
       DR_OUT <= DR_5;
       memAV1 <= 0;
       memAV2 <= 0;
       memAV3 <= 0;
       memAV4 <= 0;
       memAV5 <= 0;
       memAV6 <= 0;
       memAV7 <= 0;
       memAV8 <= 0;

       NEXT_STATE_DC <= DR_1_5;
     end
     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
       wEN <= 1;
       addr <= AR_5;
       DR_OUT <= DR_5;
       memAV1 <= 0;
       memAV2 <= 0;
       memAV3 <= 0;
       memAV4 <= 0;
       memAV5 <= 0;
       memAV6 <= 0;
       memAV7 <= 0;
       NEXT_STATE_DC <= DR_1_5;
     end
     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
       wEN <= 1;
       addr <= AR_5;
       DR_OUT <= DR_5;
       memAV1 <= 0;
       memAV2 <= 0;
       memAV3 <= 0;
       memAV4 <= 0;
       memAV5 <= 0;
       memAV6 <= 0;
       NEXT_STATE_DC <= DR_1_5;
     end
     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
       wEN <= 1;
       addr <= AR_5;
       DR_OUT <= DR_5;
       memAV1 <= 1;
       memAV2 <= 1;
       memAV3 <= 1;
       memAV4 <= 1;
       memAV5 <= 1;
       NEXT_STATE_DC <= NORM;
     end
```

```
      end
    DR_1_5:begin
        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
         wEN  <= 1;
         addr  <= AR_6;
         DR_OUT  <= DR_6;
         memAV1  <= 0;
         memAV2  <= 0;
         memAV3  <= 0;
         memAV4  <= 0;
         memAV5  <= 0;
         memAV6  <= 0;
         memAV7  <= 0;
         memAV8  <= 0;

         NEXT_STATE_DC  <= DR_1_6;
        end
        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
         wEN  <= 1;
         addr  <= AR_6;
         DR_OUT  <= DR_6;
         memAV1  <= 0;
         memAV2  <= 0;
         memAV3  <= 0;
         memAV4  <= 0;
         memAV5  <= 0;
         memAV6  <= 0;
         memAV7  <= 0;
         NEXT_STATE_DC  <= DR_1_6;
        end
        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
         wEN  <= 1;
         addr  <= AR_6;
         DR_OUT  <= DR_6;
         memAV1  <= 1;
         memAV2  <= 1;
         memAV3  <= 1;
         memAV4  <= 1;
         memAV5  <= 1;
         memAV6  <= 1;
         NEXT_STATE_DC  <= NORM;
        end
      end

    DR_1_6:begin
        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
         wEN  <= 1;
         addr  <= AR_7;
         DR_OUT  <= DR_7;
         memAV1  <= 0;
         memAV2  <= 0;
         memAV3  <= 0;
         memAV4  <= 0;
         memAV5  <= 0;
         memAV6  <= 0;
         memAV7  <= 0;
         memAV8  <= 0;

         NEXT_STATE_DC  <= DR_1_7;
        end
        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
         wEN <= 1;
         addr <= AR_7;
         DR_OUT <= DR_7;
         memAV1 <= 1;
         memAV2 <= 1;
         memAV3 <= 1;
         memAV4 <= 1;
         memAV5 <= 1;
         memAV6 <= 1;
         memAV7 <= 1;
         NEXT_STATE_DC <= NORM;
        end

      end
    DR_1_7:begin
        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
         wEN <= 1;
         addr <= AR_8;
         DR_OUT <= DR_8;
         memAV1 <= 1;
         memAV2 <= 1;
         memAV3 <= 1;
         memAV4 <= 1;
         memAV5 <= 1;
         memAV6 <= 1;
         memAV7 <= 1;
         memAV8 <= 1;

         NEXT_STATE_DC <= NORM;
        end
      end



    endcase

end



endmodule
```

10.4 Data Memory

```
// Quartus Prime Verilog Template
// Single port RAM with single read/write address

module data_mem
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=8)
(
        input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] addr,
        input wEn, clk, rEn,
        output [(DATA_WIDTH-1):0] mem_out
);

        // Declare the RAM variable
        reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

        // Variable to hold the registered read address
        reg [ADDR_WIDTH-1:0] addr_reg;

        //outfile
        integer outfile;

        initial begin
                $readmemh("D:\\Academic\\ACA\\SEM5 TRONIC ACA\\SEMESTER 5\\CSD\\FPGA\\00 - Git\\fpga-quartus\\8_CORE\\MULTI_CORE_edit\\data_mem.txt",ram);
        end

        always @ (negedge clk)
        begin
                // Write
                if (wEn) begin
                        ram[addr] <= data;
                        addr_reg <= addr;
                end
                if (rEn) begin
                        addr_reg <= addr;
                end
                $writememh("D:\\Academic\\ACA\\SEM5 TRONIC ACA\\SEMESTER 5\\CSD\\FPGA\\00 - Git\\fpga-quartus\\8_CORE\\MULTI_CORE_edit\\result.txt",ram);
        end

        assign mem_out = ram[addr_reg];

endmodule
```

10.5 Instruction Memory Controller

```
module imem_controller#(parameter WIDTH=8)(
    input Clk,
    input iROMREAD_1, iROMREAD_2, iROMREAD_3, iROMREAD_4, iROMREAD_5, iROMREAD_6, iROMREAD_7, iROMREAD_8,                //rEn signals from each core
    input coreS_1, coreS_2, coreS_3,coreS_4, coreS_5, coreS_6, coreS_7, coreS_8,                //core states from each core
    input [WIDTH-1:0] PC_1, PC_2, PC_3,PC_4, PC_5, PC_6, PC_7, PC_8,            //addresses from each core
    input [WIDTH-1:0]INS,                                                           //Instruction from IROM
    output reg rEN,                                                                 //rEn signal to IROM
    output reg [WIDTH-1:0] PC_OUT,                                                  //read address to IROM
    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
    output reg imemAV1, imemAV2, imemAV3,imemAV4, imemAV5, imemAV6, imemAV7, imemAV8                       //IROM read state signal to each core
);
localparam NORMI = 3'b000;
localparam NORMENDI = 3'b001;

reg [2:0] NEXT_STATE_IC=NORMI;
reg [2:0] STATE_IC=NORMI;



always @(negedge Clk) begin
    STATE_IC = NEXT_STATE_IC;
    case (STATE_IC)
    NORMI:
        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
            if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1
&& iROMREAD_5==1 && iROMREAD_6==1 && iROMREAD_7==1 && iROMREAD_8==1) begin
            rEN <= 1;
            PC_OUT <= PC_1;
            NEXT_STATE_IC <= NORMENDI;
            imemAV1 <= 1;
            imemAV2 <= 1;
            imemAV3 <= 1;
            imemAV4 <= 1;
            imemAV5 <= 1;
            imemAV6 <= 1;
            imemAV7 <= 1;
            imemAV8 <= 1;
        end
        else begin
            rEN <= 0;
            imemAV1 <= 0;
            imemAV2 <= 0;
            imemAV3 <= 0;
            imemAV4 <= 0;
            imemAV5 <= 0;
            imemAV6 <= 0;
            imemAV7 <= 0;
            imemAV8 <= 0;
            NEXT_STATE_IC <= NORMI;

        end
    end
    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
        if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1
&& iROMREAD_5==1 && iROMREAD_6==1 && iROMREAD_7==1) begin
           rEN <= 1;
           PC_OUT <= PC_1;
           NEXT_STATE_IC <= NORMENDI;
            imemAV1 <= 1;
            imemAV2 <= 1;
            imemAV3 <= 1;
            imemAV4 <= 1;
            imemAV5 <= 1;
            imemAV6 <= 1;
            imemAV7 <= 1;
        end
        else begin
            rEN <= 0;
            imemAV1 <= 0;
            imemAV2 <= 0;
            imemAV3 <= 0;
            imemAV4 <= 0;
            imemAV5 <= 0;
            imemAV6 <= 0;
            imemAV7 <= 0;
            NEXT_STATE_IC <= NORMI;

        end

    end

    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
        if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1
&& iROMREAD_5==1 && iROMREAD_6==1) begin
           rEN <= 1;
           PC_OUT <= PC_1;
```

```
                NEXT_STATE_IC <= NORMENDI;
                 imemAV1 <= 1;
                 imemAV2 <= 1;
                 imemAV3 <= 1;
                 imemAV4 <= 1;
                 imemAV5 <= 1;
                 imemAV6 <= 1;

            end
            else begin
                 rEN <= 0;
                 imemAV1 <= 0;
                 imemAV2 <= 0;
                 imemAV3 <= 0;
                 imemAV4 <= 0;
                 imemAV5 <= 0;
                 imemAV6 <= 0;

                 NEXT_STATE_IC <= NORMI;

            end

        end

        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
            if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1 && iROMREAD_5==1) begin
                rEN <= 1;
                PC_OUT <= PC_1;
                NEXT_STATE_IC <= NORMENDI;
                 imemAV1 <= 1;
                 imemAV2 <= 1;
                 imemAV3 <= 1;
                 imemAV4 <= 1;
                 imemAV5 <= 1;

            end
            else begin
                 rEN <= 0;
                 imemAV1 <= 0;
                 imemAV2 <= 0;
                 imemAV3 <= 0;
                 imemAV4 <= 0;
                 imemAV5 <= 0;
                 NEXT_STATE_IC <= NORMI;

            end

        end

        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
            if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1 && iROMREAD_4==1) begin
                rEN <= 1;
                PC_OUT <= PC_1;
                NEXT_STATE_IC <= NORMENDI;
                 imemAV1 <= 1;
                 imemAV2 <= 1;
                 imemAV3 <= 1;
                 imemAV4 <= 1;

             end
             else begin
                 rEN <= 0;
                 imemAV1 <= 0;
                 imemAV2 <= 0;
                 imemAV3 <= 0;
                 imemAV4 <= 0;
                 NEXT_STATE_IC <= NORMI;

             end

         end

         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
             if(iROMREAD_1==1 && iROMREAD_2==1 && iROMREAD_3==1) begin
                rEN <= 1;
                PC_OUT <= PC_1;
                NEXT_STATE_IC <= NORMENDI;
                 imemAV1 <= 1;
                 imemAV2 <= 1;
                 imemAV3 <= 1;


             end
             else begin
                 rEN <= 0;
                 imemAV1 <= 0;
                 imemAV2 <= 0;
                 imemAV3 <= 0;
                 NEXT_STATE_IC <= NORMI;

             end

         end

         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
             if(iROMREAD_1==1 && iROMREAD_2==1) begin
                rEN <= 1;
                PC_OUT <= PC_1;
                NEXT_STATE_IC <= NORMENDI;
                 imemAV1 <= 1;
                 imemAV2 <= 1;



             end
             else begin
                 rEN <= 0;
                 imemAV1 <= 0;
                 imemAV2 <= 0;
                 NEXT_STATE_IC <= NORMI;

             end

         end

         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
             if(iROMREAD_1==1) begin
                rEN <= 1;
                PC_OUT <= PC_1;
```

```
                    NEXT_STATE_IC <= NORMENDI;
                     imemAV1 <= 1;



              end
              else begin
                     rEN <= 0;
                     imemAV1 <= 0;
                     NEXT_STATE_IC <= NORMI;

              end

          end


      NORMENDI:
        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
          INS_1 <= INS;
          INS_2 <= INS;
          INS_3 <= INS;
          INS_4 <= INS;
          INS_5 <= INS;
          INS_6 <= INS;
          INS_7 <= INS;
          INS_8 <= INS;
          imemAV1 <= 1;
          imemAV2 <= 1;
          imemAV3 <= 1;
          imemAV4 <= 1;
          imemAV5 <= 1;
          imemAV6 <= 1;
          imemAV7 <= 1;
          imemAV8 <= 1;
          NEXT_STATE_IC <= NORMI;
        end

        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
          INS_1 <= INS;
          INS_2 <= INS;
          INS_3 <= INS;
          INS_4 <= INS;
          INS_5 <= INS;
          INS_6 <= INS;
          INS_7 <= INS;
          imemAV1 <= 1;
          imemAV2 <= 1;
          imemAV3 <= 1;
          imemAV4 <= 1;
          imemAV5 <= 1;
          imemAV6 <= 1;
          imemAV7 <= 1;
          NEXT_STATE_IC <= NORMI;
        end

       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
          INS_1 <= INS;
          INS_2 <= INS;
          INS_3 <= INS;
          INS_4 <= INS;
          INS_5  <= INS;
          INS_6  <= INS;
          imemAV1  <= 1;
          imemAV2  <= 1;
          imemAV3  <= 1;
          imemAV4  <= 1;
          imemAV5  <= 1;
          imemAV6  <= 1;
          NEXT_STATE_IC  <= NORMI;
        end

        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
          INS_1  <= INS;
          INS_2  <= INS;
          INS_3  <= INS;
          INS_4  <= INS;
          INS_5  <= INS;
          imemAV1  <= 1;
          imemAV2  <= 1;
          imemAV3  <= 1;
          imemAV4  <= 1;
          imemAV5  <= 1;
          NEXT_STATE_IC  <= NORMI;
        end

        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
          INS_1  <= INS;
          INS_2  <= INS;
          INS_3  <= INS;
          INS_4  <= INS;
          imemAV1  <= 1;
          imemAV2  <= 1;
          imemAV3  <= 1;
          imemAV4  <= 1;
          NEXT_STATE_IC  <= NORMI;
        end

        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
          INS_1  <= INS;
          INS_2  <= INS;
          INS_3  <= INS;
          imemAV1  <= 1;
          imemAV2  <= 1;
          imemAV3  <= 1;
          NEXT_STATE_IC  <= NORMI;
        end

        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
          INS_1  <= INS;
          INS_2  <= INS;
          imemAV1  <= 1;
          imemAV2  <= 1;
          NEXT_STATE_IC  <= NORMI;
        end

        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
          INS_1  <= INS;
          imemAV1  <= 1;
          NEXT_STATE_IC <= NORMI;
        end




      endcase

  end

  endmodule
```

10.6 Instruction Memory

```
module ins_mem
#(parameter ADDR_WIDTH=8, parameter INS_WIDTH=9)
(
    input   [(ADDR_WIDTH-1):0]  PC_address,     // Input Address
    input clk, rEn,
    output  reg [(INS_WIDTH-1):0]  instruction    // Instruction at memory
    location Address

 );

    reg [(INS_WIDTH-1):0] mem[2**ADDR_WIDTH-1:0];  // 2**ADDR_WIDTH

        initial
        begin
                $readmemh("D:\\Academic\\ACA\\SEM5 TRONIC ACA\\SEMESTER 5\\CSD\\
    FPGA\\00 - Git\\fpga-quartus\\8_CORE\\MULTI_CORE_edit\\ins_mem.txt",mem);

        end

    always @(posedge clk) begin
        if (rEn == 1) begin
            instruction <= mem[PC_address];
        end
    end

endmodule
```

10.7 Control Unit

```
`include "cu_param.v"

//Control unit outputs
// iROMREAD
// memREAD              //AR read, memory read, DR write
// memWRITE

// [14:0]wEN
//REGISTERS:
// RT4  Rr  PC  IR  AR  DR  RP  RT  RM1 RK1 RN1 C1  C2  C3  AC
// 14   13  12 _11      10  9   8  _7   6   5   4  _3   2   1   0


// [4:0]busMUX          (2**4)
// RT4    Rr  MEM   AR  DR  RP  RT  RM1 RK1 RN1 RM2 RK2 RN2 C1  C2  C3  AC
// 17     16   15   14  13  12  11  10  9   8   7   6   5   4   3   2   1


// [5:0]INC
//  PC  RM2 RK2 RN2  C2  C3
//  5   4   3   2   1   0

// [4:0] RST
// RT   RM2    RK2    RN2   AC
// 4    3      2      1     0

// [2:0]compMUX               //both muxes get the same control signal
//  M1-M2   K1-K2    N1-N2
//  2       1        0

// [3:0]aluOP
// ADDMEM  ADD      MUL     SET
// 3       2        1        0


//NEXT INSTRUCTION

module controlunit
#(parameter WIDTH = 8)
(
    input Clk,
    input z,                                    //JUMP flag
    input [WIDTH-1:0] INS,                      //Instruction from the Instruction
    memory
    input memAV,                                //DATA MEMORY AVAILABLE flag
    input imemAV,                               //INSTRUCTION MEMORY AVAILABLE flag
    output reg iROMREAD,                        //rEn of IROM
    output reg memREAD,                         //rEN of DRAM
    output reg memWRITE,                        //wEN of DRAM
    output reg [14:0] wEN,                      //wEN bitmask of core registers
    output reg selAR,                           //flag to be used in address
    fetching to AR from IROM
    output reg coreINC_AR,                      //flag to be used in increament the
    addresses by the coreID
    output reg [4:0] busMUX,                    //Bus selector
    output reg [5:0] INC,                       //Register increment bitmask
        output reg [4:0] RST,                       //Register reset bitmask
    output reg [2:0] compMUX,                   //Control signal to the comparator
    output reg [3:0] aluOP,                     //Control signal to the ALU
    output reg coreS                            //Core state flag
);


reg [7:0]NEXT_STATE=‘FETCH_1;
reg [7:0]STATE=‘FETCH_1;
reg zFlag, memAVREG, jmpMFlag;

always @(negedge Clk) begin
    zFlag = z;
    memAVREG = memAV;
end


//DEFINE ALL THE STATES OF THE CONTROL UNIT
always @(posedge Clk) begin
    STATE = NEXT_STATE;
    case(STATE)
        ‘NOOP_1 : begin                               //NO_OP
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          //iROM read before FETCH_1
        end
        `FETCH_1 : begin                            //FETCH_1     iROM[PC]
            iROMREAD <= 1;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (imemAV) begin                       //Wait unitl instruction is available to the CU
                NEXT_STATE <= `FETCH_2;
            end
            else begin
                NEXT_STATE <= `FETCH_1;
            end
        end
        `FETCH_2 : begin                            //FETCH_2     IR <= iROM[PC], PC <= PC+1
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_1000_0000_0000;         //IR WRITE
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 6'b10_0000;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_3;
        end
        `FETCH_3 : begin                            //FETCH_3      IR HAS ALREADY GOT THE INS
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            case (INS[7:4])
                `JMP : begin
                    case (INS[3:0])
                        `JMP_M : NEXT_STATE <= `JMPM_1;
                        `JMP_K : NEXT_STATE <= `JMPK_1;
                        `JMP_N : NEXT_STATE <= `JMPN_1;
                    endcase
                end
                `COPY : begin
                    NEXT_STATE <= `COPY_1;
                    iROMREAD <= 1;
                end
                `LOAD : begin
                    case (INS[3:0])
                       `LOAD_C1 : NEXT_STATE <= `LOADC1_1;
                       `LOAD_C2 : NEXT_STATE <= `LOADC2_1;
                    endcase
                end
                `STORE : NEXT_STATE <= `STORE_1;
                `ASSIGN : begin
                    NEXT_STATE <= `ASSIGN_1;
                    iROMREAD <= 1;                                   //iROM read before ASSIGN_1
                end
                `RESET : begin
                    case (INS[3:0])
                        `RESET_ALL : NEXT_STATE <= `RESETALL_1;
                        `RESET_N2 : NEXT_STATE <= `RESETN2_1;
                        `RESET_K2 : NEXT_STATE <= `RESETK2_1;
                        `RESET_Rt : NEXT_STATE <= `RESETRt_1;
                    endcase
                end
                `MOVE : begin
                    case (INS[3:0])
                        `MOVE_RP : NEXT_STATE <= `MOVEP_1;
                        `MOVE_RT : NEXT_STATE <= `MOVET_1;
                        `MOVE_RC1 : NEXT_STATE <= `MOVEC1_1;
                        `MOVE_C3 : NEXT_STATE <= `MOVEC3_1;
                    endcase
                end
                `SET : begin
                    case (INS[3:0])
                        `SETC1 : NEXT_STATE <= `SETC1_1;
                        `SETDR : NEXT_STATE <= `SETDR_1;
                        //`SETRK1 : NEXT_STATE <= `SETRK1_1;
                    endcase
                end
                `MUL : begin
                    case (INS[3:0])
                    `MUL_RP : NEXT_STATE <= `MULRP_1;
                    //`MUL_CORE : NEXT_STATE <= `MULCORE_1;
                                                   endcase
                end
                `ADD : begin
                    case (INS[3:0])
                        `ADD_RT : NEXT_STATE <= `ADDRT_1;
                        `ADD_RR1 : NEXT_STATE <= `ADDRR1_1;
                        `ADD_RM2 : NEXT_STATE <= `ADDRM2_1;
                        `ADD_MEM : NEXT_STATE <= `ADDC3_1;
                    endcase
                end
                `INC : begin
                    case (INS[3:0])
                        `INC_C2 : NEXT_STATE <= `INCC2_1;
                        `INC_C3 : NEXT_STATE <= `INCC3_1;
                        `INC_M2 : NEXT_STATE <= `INCM2_1;
                        `INC_K2 : NEXT_STATE <= `INCK2_1;
                        `INC_N2 : NEXT_STATE <= `INCN2_1;
                    endcase
                end
                `END : NEXT_STATE <= `ENDOP_1;
                `CHK_IDLE : NEXT_STATE <= `CHKIDLE_1;
                `GET_C1 : NEXT_STATE <= `GETC1_1;
            endcase
        end

        `JMPM_1 : begin                                         //`JMPM_1 REG M1 - REG M2
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 3'b100;
            aluOP <= 0;
            coreS <= 0;
            jmpMFlag <= 1;                                      //to check end operation in ZJMP
            NEXT_STATE <= `ZJMP_1;
        end
        `JMPK_1 : begin                                         //JMP_K REG K1 - REG K2
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 3'b010;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `ZJMP_1;
        end
        `JMPN_1 : begin                                         //JMP_N REG N1 - REG N2
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 3'b001;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `ZJMP_1;
        end
        `ZJMP_1 : begin                                         //ZERO CHECK (check loop termination)
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            if (zFlag == 1 && jmpMFlag == 1) begin          //end of the program
                NEXT_STATE <= `ENDOP_1;
                jmpMFlag <= 0;
                coreS <= 1;
            end
            else if (zFlag == 1 && jmpMFlag == 0) begin
                NEXT_STATE <= `NJMP_1;
            end
            else begin
                NEXT_STATE <= `JMP_2;
                iROMREAD <= 1;                              // iROM read before JMP_2
                jmpMFlag <= 0;
            end
        end
       `JMP_2 : begin                                       //JMP_2 READ_IROM[PC]
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <=`JMP_3;
        end
       `JMP_3 :begin                                        //JMP_3 AR <= IROM[PC]
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0100_0000_0000;
            selAR <= 1;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <=`JMP_4;
        end
       `JMP_4 : begin                                       //JMP_4 PC <= AR
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b001_0000_0000_0000;
            selAR <= 0;
            busMUX <= 14;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= ‘FETCH_1;
        end
        ‘NJMP_1 : begin                                     //NO‘JMP
        PC <= PC + 1
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 6’b10_0000;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= ‘FETCH_1;
        end
        ‘COPY_1 : begin                                     //‘COPY_1
        READ_IROM[PC]
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= ‘COPY_2;
        end
        ‘COPY_2 : begin                                     //‘COPY_2
        AR <= IROM[PC], PC <= PC + 1
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15’b000_0100_0000_0000;
            selAR <= 1;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 6’b10_0000;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (INS[3:0]== ‘COPY_M || INS[3:0]== ‘COPY_T) begin
                NEXT_STATE <= ‘COPYM_3A;
            end
            else begin
                NEXT_STATE <= ‘COPY_3;
            end
        end
        ‘COPYM_3A : begin                                   //COPYM3_A
          AR <= AR + Core_ID
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 1;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `COPY_3;
        end
        `COPY_3 : begin                                     //`COPY_3
      MEM_READ[AR]  ... GIVING MEMORY ADDRESS
            iROMREAD <= 0;
            memREAD <= 1;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 4'b1111;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (memAVREG) begin
                NEXT_STATE <= `COPY_4;
            end
            else begin
                NEXT_STATE <= `HOLD_1;
            end
        end
        `COPY_4 : begin                                     //`COPY_4
    DR <= MEM_READ[AR] ... ALREADY RECEIVED DATA
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0010_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 4'b1111;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (INS[3:0]== `COPY_M) begin                // RM1
                NEXT_STATE <= `COPYM_5;
            end
            else if (INS[3:0]== `COPY_K) begin           // RK1
                NEXT_STATE <= `COPYK_5;
            end
            else if (INS[3:0]== `COPY_N) begin           // RN1
                NEXT_STATE <= `COPYN_5;
            end
            else if (INS[3:0]== `COPY_R) begin           // Rr
                NEXT_STATE <= `COPYR_5;
            end
            else if (INS[3:0]== `COPY_T) begin           // RT4
                NEXT_STATE <= `COPYRT4_5;
            end
        end
        `COPYM_5 : begin                                    //`COPY M1
     REG M1 <= DR
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0100_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 4'b1101;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                                  // iROM read before
    FETCH_1
        end
        `COPYK_5 : begin                                    //`COPY K1
     REG K1 <= DR
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0010_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 4'b1101;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                                  // iROM read before
    FETCH_1
        end
        `COPYN_5 : begin                                    //`COPY N1
     REG N1 <= DR
            memREAD <= 1;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0001_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 13;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                                  // iROM read before
    FETCH_1
        end
        `COPYR_5 : begin                                    //`COPY Rr
     Rr <= DR
            memREAD <= 1;
            memWRITE <= 0;
            wEN <= 15'b010_0000_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 13;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          // iROM read before
    FETCH_1
        end
        `COPYRT4_5 : begin                          //`COPY RT4
      RT4 <= DR
            memREAD <= 1;
            memWRITE <= 0;
            wEN <= 15'b100_0000_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 13;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          // iROM read before
    FETCH_1
        end
        `LOAD_2 : begin                             //`LOAD_2
         MEM_READ[AR]
            iROMREAD <= 0;
            memREAD <= 1;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 15;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (memAVREG) begin
                NEXT_STATE <= `LOAD_3;
            end
            else begin
                NEXT_STATE <= `HOLD_1;
            end
        end
        `LOAD_3 : begin                             //`LOAD_3
         DR <= MEM_READ[AR]
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0010_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 15;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          // iROM read before
    FETCH_1
        end
        `LOADC1_1 : begin                           //`LOAD_C1
         AR <= C1
```

```
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0100_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 4;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `LOAD_2;
        end
        `LOADC2_1 : begin                               //`LOAD_C2
         AR <= C2
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0100_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 3;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `LOAD_2;
        end
        `STORE_1 : begin                                //`STORE_1
         DR <= RT
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0010_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 11;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `STORE_2;
        end
        `STORE_2 : begin                                //`STORE_2
           AR <= C3
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0100_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 2;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `STORE_3;
        end
        `STORE_3 : begin                                //`STORE_3
        MEM_WRITE[AR] <= DR
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 1;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 13;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (memAVREG) begin
                NEXT_STATE <= `FETCH_1;
                iROMREAD <= 1;                          // iROM read before
    FETCH_1
            end
            else begin
                NEXT_STATE <= `HOLD_1;                  // Hold the processor
    till the DRAM gives data
            end
        end
        `ASSIGN_1 : begin                               //ASSIGN_1
        READ_IROM[PC]
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `ASSIGN_2;
        end
        `ASSIGN_2 : begin                               //ASSIGN_3
      AR <= IROM[PC], PC <= PC+1
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0100_0000_0000;
            selAR <= 1;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 6'b10_0000;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (INS[3:0]== `ASSIGN_C1) begin
                NEXT_STATE <= `ASSIGNC1_3A;
            end
            else if (INS[3:0]== `ASSIGN_C2) begin
                NEXT_STATE <= `ASSIGNC2_3;
            end
            // else if (INS[3:0]== `ASSIGN_C3) begin          //not needed in
     new algo
            //     NEXT_STATE <= `ASSIGNC3_3;
                        //      end
        end
        `ASSIGNC1_3A : begin                            //ASSIGN_C1A
          AR <= AR + Core_ID
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_1000;
            selAR <= 0;
            coreINC_AR <= 1;
            busMUX <= 14;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `ASSIGNC1_3;
                end
            `ASSIGNC1_3 : begin                                //ASSIGN_C1
              C1 <= AR
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_1000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 14;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before
    FETCH_1
                end
                `ASSIGNC2_3 : begin                            //ASSIGN_C2
                  C2 <= AR
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_0100;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 14;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before
    FETCH_1
        end
        // `ASSIGNC3_3 : begin             //ASSIGN_C3                   C3 <=
    AR     //not needed in new algo
        //     iROMREAD <= 0;
        //     memREAD <= 0;
        //     memWRITE <= 0;
        //     wEN <= 15'b000_0000_0000_0010;
        //     selAR <= 0;
        //     coreINC_AR <= 0;
        //     busMUX <= 14;
        //     INC <= 0;
        //     RST <= 0;
        //     compMUX <= 0;
        //      aluOP  <=  0;
        //      coreS  <=  0;
        //      NEXT_STATE  <=  `FETCH_1;
        //      iROMREAD  <=  1;            //  iROM  read  before  FETCH_1
        //  end
        `RESETALL_1  :  begin                                 //RESET_ALL
            memREAD  <=  0;
            memWRITE  <=  0;
            wEN  <=  0;
            selAR  <=  0;
            coreINC_AR  <=  0;
            busMUX  <=  0;
            INC  <=  0;
            RST  <=  5'b11111;
            compMUX  <=  0;
            aluOP  <=  0;
            coreS  <=  0;
            NEXT_STATE  <=  `FETCH_1;
            iROMREAD  <=  1;                                  //  iROM  read  before
    FETCH_1
        end
        `RESETN2_1  :  begin                                  //RESET  REG  N2
            memREAD  <=  0;
            memWRITE  <=  0;
            wEN  <=  0;
            selAR  <=  0;
            coreINC_AR  <=  0;
            busMUX  <=  0;
            INC  <=  0;
            RST  <=  5'b00010;
            compMUX  <=  0;
            aluOP  <=  0;
            coreS  <=  0;
            NEXT_STATE  <=  `FETCH_1;
            iROMREAD  <=  1;                                  //  iROM  read  before
    FETCH_1
        end
        `RESETK2_1  :  begin                                  //RESET  REG  K2
            memREAD  <=  0;
            memWRITE  <=  0;
            wEN  <=  0;
            selAR  <=  0;
            coreINC_AR  <=  0;
            busMUX  <=  0;
            INC  <=  0;
            RST  <=  5'b00100;
            compMUX  <=  0;
            aluOP  <=  0;
            coreS  <=  0;
            NEXT_STATE  <=  `FETCH_1;
            iROMREAD  <=  1;                                  //  iROM  read  before
    FETCH_1
        end
        `RESETRt_1  :  begin                                  //RESET  REG  RT
            memREAD  <=  0;
            memWRITE  <=  0;
            wEN  <=  0;
            selAR  <=  0;
            coreINC_AR  <=  0;
            busMUX  <=  0;
            INC  <=  0;
            RST  <=  5'b10000;
            compMUX  <=  0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before FETCH_1
        end
        `MOVEP_1 : begin                                //MOVE TO REG P REG P <= AC
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0001_0000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 1;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before FETCH_1
        end
        `MOVET_1 : begin                                //MOVE TO REG T REG T <= AC
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_1000_0000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 1;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before FETCH_1
        end
        `MOVEC1_1 : begin                               //MOVE TO REG C1 C1 <= AC
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_1000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 1;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before FETCH_1
        end
        `MOVEC3_1 : begin                               //MOVE TO REG C3 C3 <= AC
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_0010;
            selAR <= 0;
```

```
                coreINC_AR <= 0;
                busMUX <= 1;
                INC <= 0;
                RST <= 0;
                compMUX <= 0;
                aluOP <= 0;
                coreS <= 0;
                NEXT_STATE <= `FETCH_1;
                iROMREAD <= 1;                              // iROM read before
FETCH_1
            end
            `SETC1_1 : begin                                //SET AC AS C1
             SET, AC <= C1
                memREAD <= 0;
                memWRITE <= 0;
                wEN <= 15'b000_0000_0000_0001;
                selAR <= 0;
                coreINC_AR <= 0;
                busMUX <= 4;
                INC <= 0;
                RST <= 0;
                compMUX <= 0;
                aluOP <= 4'b0001;
                coreS <= 0;
                NEXT_STATE <= `FETCH_1;
                iROMREAD <= 1;                              // iROM read before
FETCH_1
            end
            `SETDR_1 : begin                                //SET AC AS DR
              SET, AC <= DR
                memREAD <= 0;
                memWRITE <= 0;
                wEN <= 15'b000_0000_0000_0001;
                selAR <= 0;
                coreINC_AR <= 0;
                busMUX <= 13;
                INC <= 0;
                RST <= 0;
                compMUX <= 0;
                aluOP <= 4'b0001;
                coreS <= 0;
                NEXT_STATE <= `FETCH_1;
                iROMREAD <= 1;                              // iROM read before
FETCH_1
            end
            // `SETRK1_1 : begin      //SET AC AS DR                  SET, AC <= DR
            //      iROMREAD <= 0;
            //      memREAD <= 0;
            //      memWRITE <= 0;
            //      wEN <= 15'b000_0000_0000_0001;
            //      selAR <= 0;
            //      coreINC_AR <= 0;
            //      busMUX <= 9;
            //      INC <= 0;
            //      RST <= 0;
            //      compMUX <= 0;
            //      aluOP <= 4'b0001;
            //      coreS <= 0;
            //      NEXT_STATE <= `FETCH_1;
            // end
            `MULRP_1 : begin                                //MUL REG P
            AC <= REG_P * AC
                memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_0001;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 12;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 4'b0010;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before
FETCH_1
        end
        // `MULCORE_1 : begin       //AC MULTIPLY CORE ID               AC <=
CORE ID * AC
        //     iROMREAD <= 0;
        //     memREAD <= 0;
        //     memWRITE <= 0;
        //     wEN <= 15'b000_0000_0000_0001;
        //     selAR <= 0;
        //     coreINC_AR <= 0;
        //     busMUX <= 0;
        //     INC <= 0;
        //     RST <= 0;
        //     compMUX <= 0;
        //     aluOP <= 4'b1000;
        //     coreS <= 0;
        //     NEXT_STATE <= `FETCH_1;
        // end

        `ADDRT_1 : begin                                //ADD REG_T
   AC <= REG_1 + AC
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_0001;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 11;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 4'b0100;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before
FETCH_1
        end
        `ADDRR1_1 : begin                               //ADD REG_Rr
    AC <= Rr + AC
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_0001;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 16;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 4'b0100;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                              // iROM read before
```

```
FETCH_1
    end
    `ADDRM2_1 : begin                                //ADD REG_M2
    AC <= REG_M2 + AC
        memREAD <= 0;
        memWRITE <= 0;
        wEN <= 15'b000_0000_0000_0001;
        selAR <= 0;
        coreINC_AR <= 0;
        busMUX <= 7;
        INC <= 0;
        RST <= 0;
        compMUX <= 0;
        aluOP <= 4'b0100;
        coreS <= 0;
        NEXT_STATE <= `FETCH_1;
        iROMREAD <= 1;                                   // iROM read before
FETCH_1
    end
    `ADDC3_1 : begin                                 //ADD RK1*CORE_ID TO C3
               AC <= C3 + AC
        memREAD <= 0;
        memWRITE <= 0;
        wEN <= 15'b000_0000_0000_0001;
        selAR <= 0;
        coreINC_AR <= 0;
        busMUX <= 2;
        INC <= 0;
        RST <= 0;
        compMUX <= 0;
        aluOP <= 4'b1000;
        coreS <= 0;
        NEXT_STATE <= `FETCH_1;
        iROMREAD <= 1;                                   // iROM read before
FETCH_1
    end
    `INCC2_1 : begin                                 //INC REG C2
   REG C2 <= C2+1
        memREAD <= 0;
        memWRITE <= 0;
        wEN <= 0;
        busMUX <= 0;
        INC <= 6'b00_0010;
        RST <= 0;
        compMUX <= 0;
        aluOP <= 0;
        coreS <= 0;
        NEXT_STATE <= `FETCH_1;
        iROMREAD <= 1;                                   // iROM read before
FETCH_1
    end
    `INCC3_1 : begin                                 //INC REG C3
    REG C3 <= C3+1
        memREAD <= 0;
        memWRITE <= 0;
        wEN <= 0;
        selAR <= 0;
        coreINC_AR <= 0;
        busMUX <= 0;
        INC <= 6'b00_0001;
        RST <= 0;
        compMUX <= 0;
        aluOP <= 0;
```

```
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          // iROM read before FETCH_1
        end
        `INCM2_1 : begin                            //INC reg m2 REG M2 <= M2+1
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 6'b01_0000;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          // iROM read before FETCH_1
        end
        `INCK2_1 : begin                            //INC REG K2 REG K2 <= K2+1
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 6'b00_1000;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          // iROM read before FETCH_1
        end
        `INCN2_1 : begin                            //INC REG N2 REG N2 <= N2+1
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 6'b00_0100;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                          // iROM read before FETCH_1
        end
        `ENDOP_1 : begin                            //INC REG N2 REG N2 <= N2+1
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 1;
            NEXT_STATE <= `ENDOP_1;
        end
        `HOLD_1 : begin
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            if (memAVREG) begin
                case (INS[7:4])
                    `COPY : NEXT_STATE <= `COPY_4;
                    `LOAD : NEXT_STATE <= `LOAD_3;
                    `STORE : begin
                        NEXT_STATE <= `FETCH_1;
                        iROMREAD <= 1;                     // iROM read before FETCH_1
                    end
                endcase
            end
            else begin
                NEXT_STATE <= `HOLD_1;                     // Hold till data is available frome DRAM
            end
        end
        `CHKIDLE_1 : begin                                 //CHECK RM1 - RM2 REG M1 - REG M2
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 3'b100;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `CHKIDLE_2;
        end
        `CHKIDLE_2 : begin      //ZERO CHECK
            iROMREAD <= 0;
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 0;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 0;
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            if (zFlag) begin
                NEXT_STATE <= `ENDOP_1;
            end
            else begin
                NEXT_STATE <= `FETCH_1;
                iROMREAD <= 1;                              // iROM read before
    FETCH_1
            end
        end
        `GETC1_1 : begin                                    //MOVE RT4 VAL TO
    REG C1                  C1 <= RT4
            memREAD <= 0;
            memWRITE <= 0;
            wEN <= 15'b000_0000_0000_1000;
            selAR <= 0;
            coreINC_AR <= 0;
            busMUX <= 17;  //RT4
            INC <= 0;
            RST <= 0;
            compMUX <= 0;
            aluOP <= 0;
            coreS <= 0;
            NEXT_STATE <= `FETCH_1;
            iROMREAD <= 1;                                  // iROM read before
    FETCH_1
        end
    endcase
end
endmodule
```

10.8 Arithmetic and Logic Unit

```
module Alu #(
    parameter WIDTH = 8
) (
    input [WIDTH-1:0] AC, BusOut,
    input [3:0] ALU_OP,
    input [WIDTH-1:0] MEM_ID,
    output [WIDTH-1:0] result_ac
);

        localparam SET = 4'b0001;
        localparam MUL = 4'b0010;
        localparam ADD = 4'b0100;
    localparam ADDMEM = 4'b1000;


    reg [WIDTH-1:0] result;

    always @(*) begin
        case (ALU_OP)
            SET:
                result<=BusOut;
            MUL:
                result<=AC*BusOut;
            ADD:
                result<=AC+BusOut;
            ADDMEM:
                result <= AC+MEM_ID;
            default:
                result<=AC;
        endcase
    end
    assign result_ac = result;

endmodule
```

10.9 Bus Mux

```
//‘include "proc_pram.v"
//  MEM AR  DR  RP  RT  RM1 RK1 RN1 RM2 RK2 RN2 C1  C2  C3  AC
//  15   14  13  12  11  10  9   8   7   6   5   4   3   2   1

module Bus_mux
#(parameter WIDTH = 8)
(MEM, AR, DR, RP, RT, RM1, RK1, RN1, RM2, RK2, RN2, C1,  C2,  C3,  AC, RR, RT4,
    mux_sel, Bus_select);

input [4:0] mux_sel;
input [WIDTH-1:0] MEM, AR, DR, RP, RT, RM1, RK1, RN1, RM2, RK2, RN2, C1,  C2,
    C3,  AC, RR, RT4;
output [WIDTH-1:0] Bus_select;
// The output is defined as register
reg [WIDTH-1:0] select;
always @(*)
begin
    case(mux_sel)

        5'b00001:
            select <= AC;
        5'b00010:
            select <= C3;
        5'b00011:
            select <= C2;
        5'b00100:
            select <= C1;
        5'b00101:
            select <= RN2;
        5'b00110:
            select <= RK2;
        5'b00111:
            select <= RM2;
        5'b01000:
            select <= RN1;
        5'b01001:
            select <= RK1;
        5'b01010:
            select <= RM1;
        5'b01011:
            select <= RT;
        5'b01100:
            select <= RP;
        5'b01101:
            select <= DR;
        5'b01110:
            select <= AR;
        5'b01111:
            select <= MEM;
        5'b10000:
            select <= RR;
        5'b10001:
            select <= RT4;

    endcase
end

```

```
assign Bus_select = select;
endmodule
```

10.10 Comparator

```
module Comp
#(parameter WIDTH = 8)
(
    input [WIDTH-1:0] R1, R2,                    //inputs comes from AC and bus
    output wire z                                //control signal for Jump
   instructions
);

reg [WIDTH-1:0] value;
reg flagOut;
always @(*) begin
    value <= R1-R2;
        flagOut <= (value == 8'b0);
end

assign z = flagOut;

endmodule
```

10.11 Program Counter(PC)

```
module PC //write, increment enable /PC, C2, C3
#(parameter WIDTH = 8)
(Clk, WEN,INC,BusOut,dout);
input Clk, WEN,INC;
input [WIDTH-1:0] BusOut;
output [WIDTH-1:0] dout;
reg [WIDTH-1:0] value;

initial begin
    value <= 8'b0;
end
always @(negedge Clk)
begin

    if (WEN) value <= BusOut;
    else if (INC) value <= dout + 8'b1;
end

assign dout=value;

endmodule
```

10.12 Address Register (AR)

```
module AR //write enable /IR, AR, Rp, Rt, Rk1, Rm1, Rn1, C1
#(parameter WIDTH = 8)
(
input Clk, WEN, selAR, coreINC_AR,
input [WIDTH-1:0] IOut, BusOut,
input [2:0] coreID,
output [WIDTH-1:0] dout
);

reg [WIDTH-1:0] value;

always @(negedge Clk)
begin
    if(WEN==1 && selAR == 0 && coreINC_AR == 0) begin
```

```
        value <= BusOut;
    end
    if (coreINC_AR==1) begin                                    //will be used
    in the COPYRm1 & COPYT4
        value <= value + coreID;
    end
    if(WEN==1 && selAR == 1 && coreINC_AR == 0) begin
        value <= IOut;
    end
end

assign dout=value;

endmodule
```

10.13 Read-Write Register

```
module Reg_module_W //write enable /IR, AR, Rp, Rt, Rk1, Rm1, Rn1, C1
#(parameter WIDTH = 8)
(Clk, WEN, BusOut, dout);
input Clk, WEN;
input [WIDTH-1:0] BusOut;
output [WIDTH-1:0] dout;
reg [WIDTH-1:0] value;

always @(negedge Clk)
begin

    if (WEN) value <= BusOut;

end

assign dout=value;

endmodule
```

10.14 Write-Reset Register

```
module Reg_module_RW //write, reset enable //Rt, AC
#(parameter WIDTH = 8)
(Clk,RST,WEN,BusOut,dout);
input Clk, WEN,RST;
input [WIDTH-1:0] BusOut;
output [WIDTH-1:0] dout;
reg [WIDTH-1:0] value;


always @(negedge Clk)
begin

    if (WEN) value <= BusOut;
    else if (RST) value <= 8'b0;
end

assign dout=value;

endmodule
```

10.15 Write-Increment Register

```
module Reg_module_RI //increment, reset enable //Rn2, Rk2, Rm2
#(parameter WIDTH = 8)
(Clk, RST,INC,BusOut,dout);
input Clk, RST,INC;
input [WIDTH-1:0] BusOut;
output [WIDTH-1:0] dout;
reg [WIDTH-1:0] value;

always @(negedge Clk)
begin

    if (RST) value <= 8'b0;
    else if (INC) value <= dout + 8'b1;
end

assign dout=value;

endmodule
```

10.16 Write-Reset-Increment Register

```
module Reg_module_WI //write, increment enable /PC, C2, C3
#(parameter WIDTH = 8)
(Clk, WEN,INC,BusOut,dout);
input Clk, WEN,INC;
input [WIDTH-1:0] BusOut;
output [WIDTH-1:0] dout;
reg [WIDTH-1:0] value;

initial begin
    value <= 8'b0;
end
always @(negedge Clk)
begin

    if (WEN) value <= BusOut;
    else if (INC) value <= dout + 8'b1;
end

assign dout=value;

endmodule
```