Department of Electronic and Telecommunication Engineering
University of Moratuwa

EN 3030 - Circuits and Systems Design

PROJECT REPORT

DESIGN AND IMPLEMENTATION OF A CUSTOM PROCESSOR FOR MATRIX MULTIPLICATION

Name	Index Number
ABEYWICKRAMA D.S.N	170009C
HETTIARACHCHI M.N	170222X
THANUKA M.B.S.D	170614C
WICKRAMASINGHE J.A.D.L	170692K

This report is submitted in partial fulfillment of the requirements
for the module EN 3030 - Circuits and Systems Design.
July 7, 2021

Contents

List of Figures

List of Tables

1 Abstract

With the breakthrough of embedded systems in todays world there are numerous number of custom single purpose processors in use. Benefits of such processors are high performance, small in size and low power consumption. In this report we demonstrate the design architecture followed and a performance evaluation of the custom processor we built using a FPGA to do matrix multiplication.

The processor we designed in general consists of 4 cores, a common instruction memory and data memory and a memory controller which controls the memory access from each core. Our ISA consists of 28 instructions and the defined memory consists of 16 bit words. Each core consists of 12 Registers,a Control Unit and an Arithmetic and Logic Unit.

And finally we provide a comprehensive performance evaluation of the speed of the processor when the number of cores increases from 1 to 4 for different matrices.

2 Introduction

This report describes the step-by-step procedure used to design a processor to perform matrix multiplication. And also, in this report we include the test codes that are used, and the simulation results and the physical hardware implementations.

2.1 Processor Design

The main requirement of this assignment is to design a processor and a CPU (Central Processing Unit) with multiple cores which can multiply two given matrices. We did this task using Verilog Hardware Description Language (HDL) and the implementations are done using the Quartus Prime Lite edition 19.1 and the Development board with Cyclone IV FPGA and the simulations are done using ModelSim-Altera. We implemented Quad Core Processor to do this task.

2.2 Central Processing Unit (CPU)

The main part of any digital computer system is the Central Processing Unit (CPU) which is the physical heart of the entire computer system. The electronic circuitry in the computer system which performs basic arithmetic functions such as addition, subtraction, multiplication, and division, and logical operations such as the selection of desired problem-solving procedure based on predefined decision criteria and the comparing data, and the input/output(I/O) operations according to the given instructions, is called the Central Processing Unit (CPU)[2].

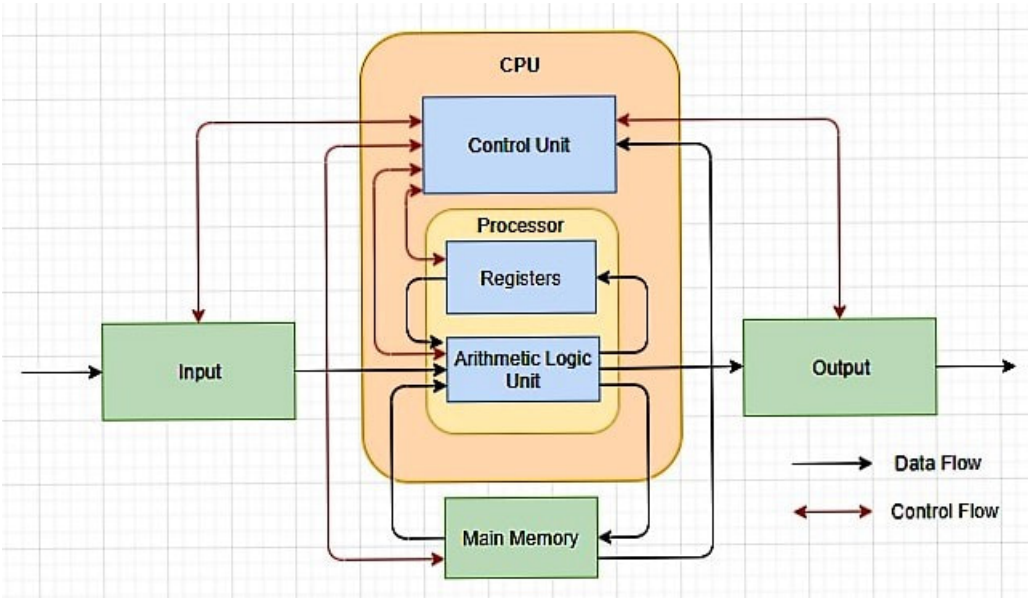

Figure 1: Basic Diagram of the CPU[1]

The Central processing unit (CPU) is generally composed of the main memory, control unit, and arithmetic-logic unit. The control unit of the central processing unit regulates and integrates the operations of the computer. It selects and retrieves instructions from the main memory in proper sequence.

2.3 Problem Statement

We have given on this project to multiply two matrices with using multiple cores. Therefore, we specially focused to implement this using multiple cores and increase the speed of the task. There are no limits for the size of the input matrices. But we defined some constrained for the size of matrices as following to avoid overflowing the DRAM which designed with a memory size of 512, 16 bits words.

$$m \times n + n \times p + m \times p < 506$$

m: no of rows in input matrix A
n: no of columns in input matrix A
p: no of columns in input matrix B

We multiplied the two input matrices using a python code for the verification of the designed processor.

2.4 Proposed Solution based on FPGA

The main requirement was to design a processor to multiply two matrices. Therefore, two input matrices should be stored in the memory unit of the processor at the starting. And also, after the multiplication of two matrices, result matrix needs to be sent back to the computer and display it. The task flow of the project is mainly as follows.

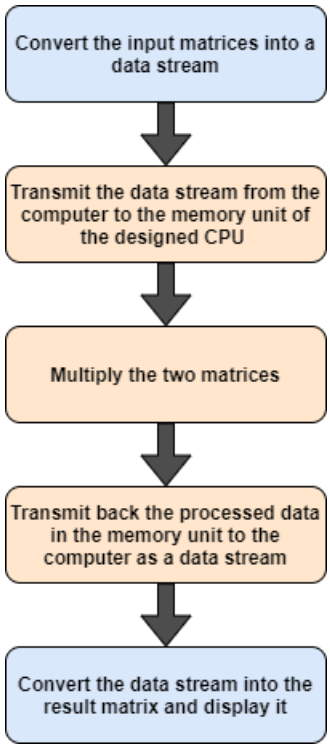

Figure 2: Task flow

First, we generated the input file in the memory initialization file (MIF) format with the addresses and 16 bits words. Then we imported the input file using the In-System memory content editor and stored the data in the memory unit, so that data can be accessed when required.

Then the FPGA programmed using the Verilog HDL code with the IRAM and the DRAM. After initializing the start, designed processor multiplied the two input matrices and store results matrix in the DRAM.

Finally, the output file exported to a directory in the local machine using the In-System memory content editor and displayed it. After deciding the task flow of the project, the following key points are the main requirements of the designing process.

1. Generate MIF files to import data from the computer and export data to the computer.
2. To store the input matrices and the result matrix in the CPU, the designing memory unit needs to be large.
3. Design ISA for the processor which is capable of multiplying two matrices.

3 Designing a FPGA based custom Processor

3.1 What is FPGA?

The Field Programmable Gate Array, which is commonly known as FPGA is an integrated circuit that consists of user programmable interconnections of internal and configurable hardware logic blocks(CLBs) to customize operations for a specific application. FPGAs can be reprogrammed to perform custom applications or functionality requirements after manufacturing. This feature differentiates the FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured to perform pre defined specific tasks. [3] For this project we have used Cyclone DE2-115 development and educational board[4] which contains a chip manufactured by Altera with other peripheral devices. We used Intel Altera Quartus Prime[5], Model Sim[6] software and Verilog as the Hardware Descriptive Language (HDL).

3.2 Instruction Set Architecture (ISA)

The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler writer which is more or less machine language. The ISA serves as the boundary between software and hardware. The instruction set provides commands to the processor, on what needs to be done. The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O.

3.2.1 Instructions Set

Instruction	Opcode	State	Address	Operation
FETCH		FETCH1	0	READ
		FETCH2	1	DR <= IRAM, PC <= PC + 1
		FETCH3	2	IR <= DR, READ
		FETCH4	3	IDLE
LDAC x	1	LDAC1	5	DR <= IRAM, PC <= PC + 1
		LDAC2	6	AR <= DR, READ
		LDAC3	7	DR <= DRAM
		LDAC4	8	AC <= DR
LOADACR	2	LOADAC1	9	AR <= R, READ
		LOADAC2	10	DR <= DRAM
		LOADAC3	11	AC <= DR
STAC	3	STAC1	12	AR <= R, DR <= AC, WRITE
		STAC2	13	DRAM[R] <= DR
MVACR	4	MVACR1	14	R <= AC
MVACA	5	MVACA1	15	A <= AC
MVACSUM	6	MVACSUM1	16	SUM <= AC
MVACI	7	MVACI1	17	I <= AC
MVSUMAC	8	MVSUMAC1	18	AC <= SUM
MVIR	9	MVIR1	19	R <= I
MVKR	10	MVKR1	20	R <= K
MVJR	11	MVJR1	21	R <= J
MVIDR	12	MVIDR1	22	R <= ID
MULR	13	MULR1	23	AC <= AC*R
MULA	14	MULA1	24	AC <= AC*A
ADDR	15	ADDR1	25	AC <= AC+R
ADDSUM	16	ADDSUM1	26	AC <= AC + SUM
COMP	17	COMP1	27	AC <= AC - R, if AC <0 z=1 else z=0
INCK	18	INCK1	28	K <= K+1
INCJ	19	INCJ1	29	J <= J+1
INCR	20	INCR1	30	R <= R+1
JMPZ a	21	JMPZY1	31	DR <= IRAM
		JMPZY2	32	PC <= DR
		JMPZN1	33	PC <= PC + 1
JUMP a	22	JUMP1	34	DR <= IRAM
		JUMP2	35	PC <= DR
RSTALL	23	RSTALL1	36	ID <= Core ID
RSTI	24	RSTI1	37	I <= ID
RSTJ	25	RSTJ1	38	J <= 0
RSTK	26	RSTK1	39	K <= 0
RSTSUM	27	RSTSUM1	40	SUM <= 0
NOP	28	NOP1	41	NO OPERATION

Table 1: Instruction Set (ISA)

3.2.2 State Diagram

The operation of this processor is based on a behavioral model called finite-state machine, which consist of a finite number of states that executes one task at any given time. In order to achieve the matrix multiplication, the machine repeatedly follows a pre-defined, finite set of sequence flows. The following diagram represents the state machine of our custom made processor.

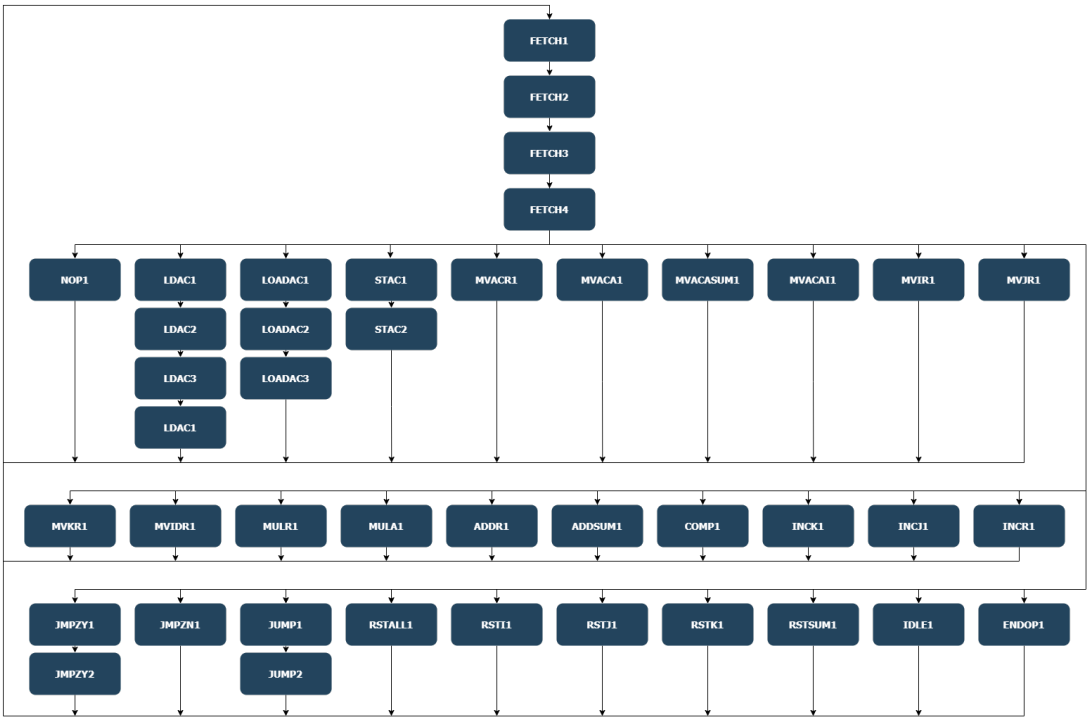

Figure 3: State Diagram

3.2.3 Processor Architecture

3.2.3.1 Processor RTL Netlist View

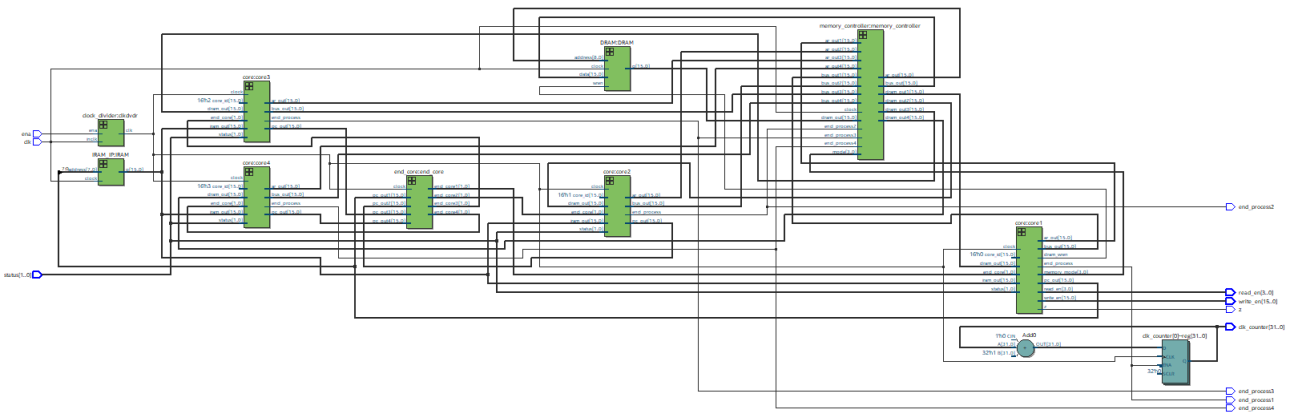

Figure 4: RTL Netlist View

3.2.3.2 Single Core Data Path

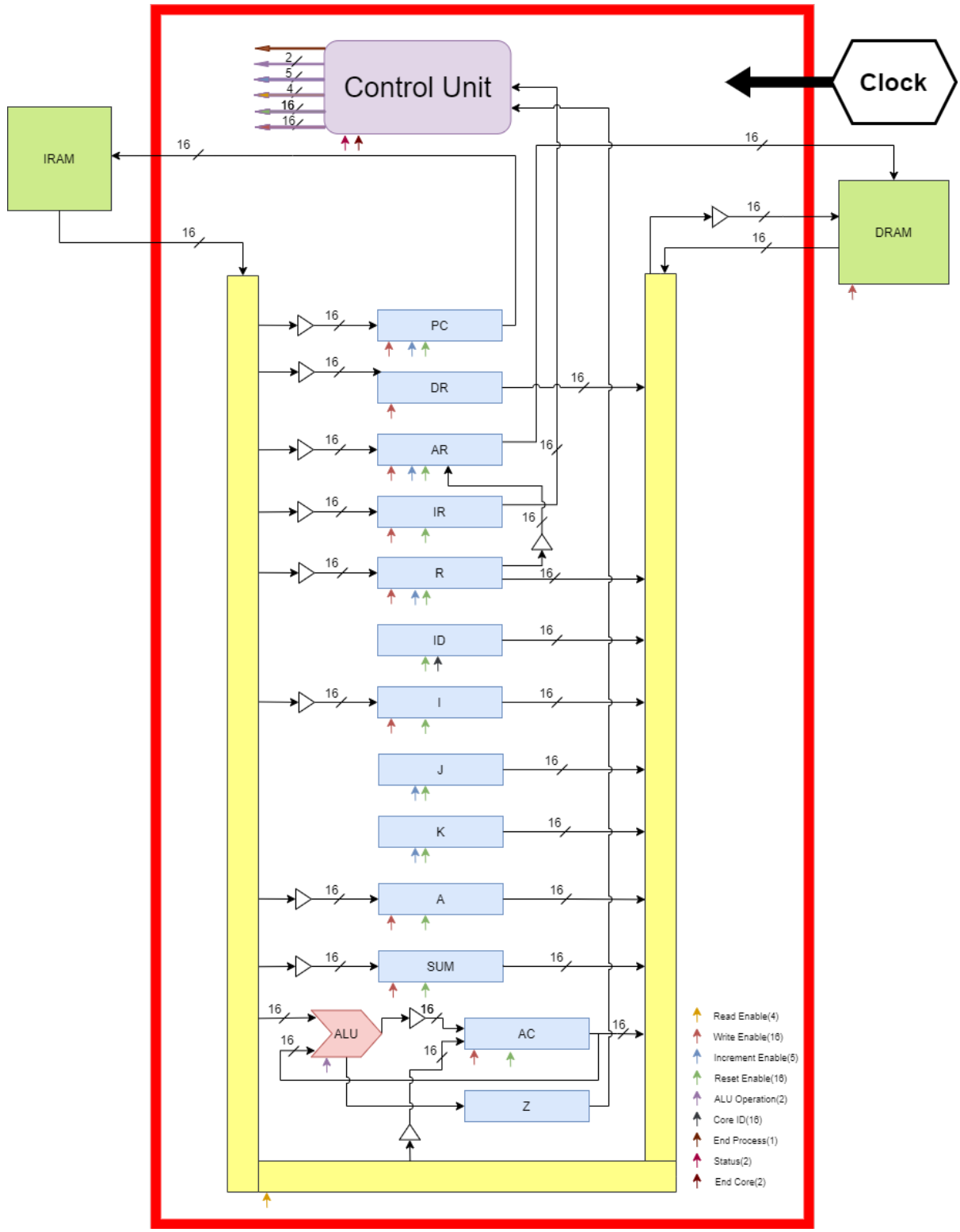

Figure 5: Single Core

3.2.3.3 Multi Core Data Path

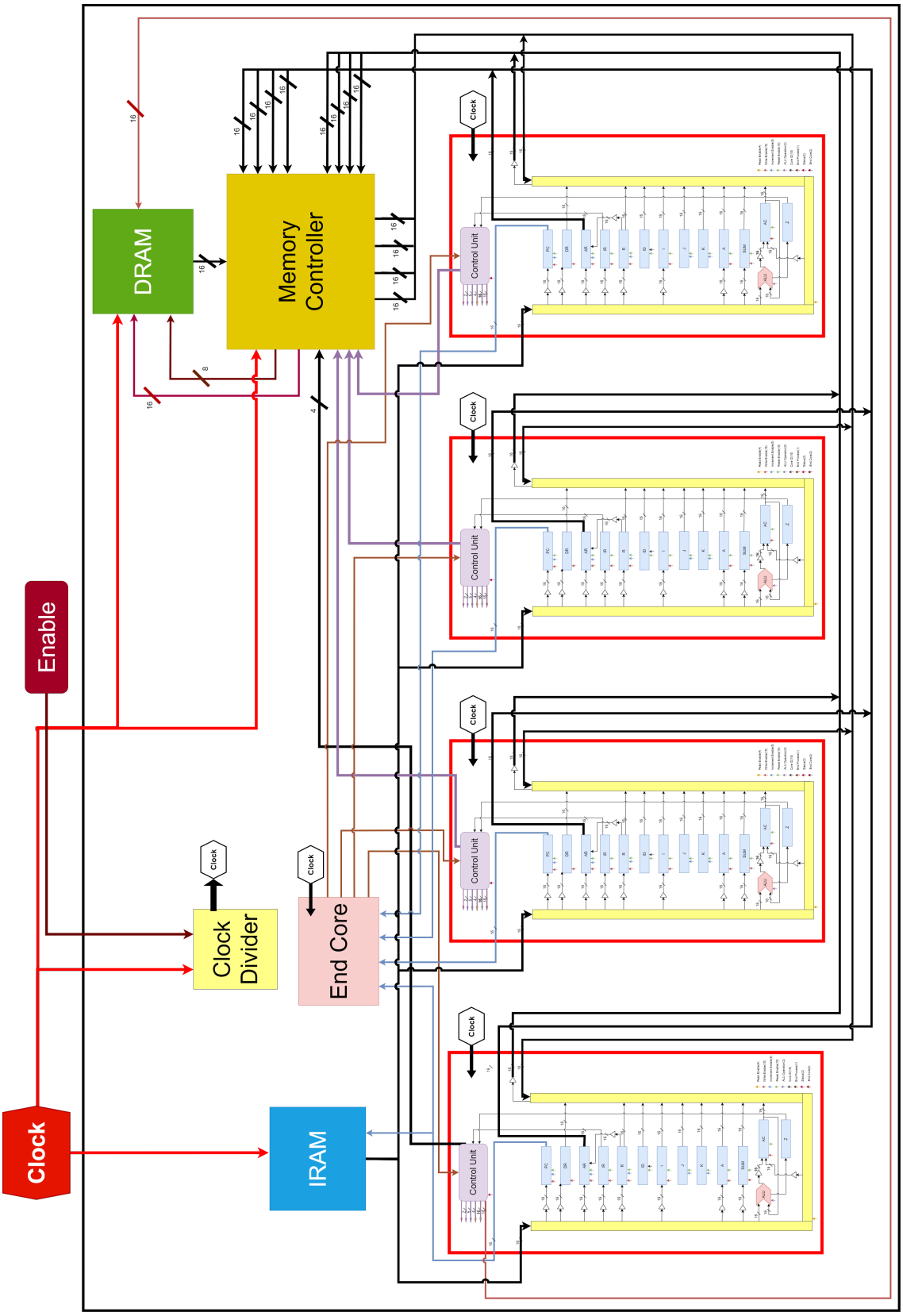

Figure 6: Multi Core

3.3 Modules and Components

3.3.1 Top level module

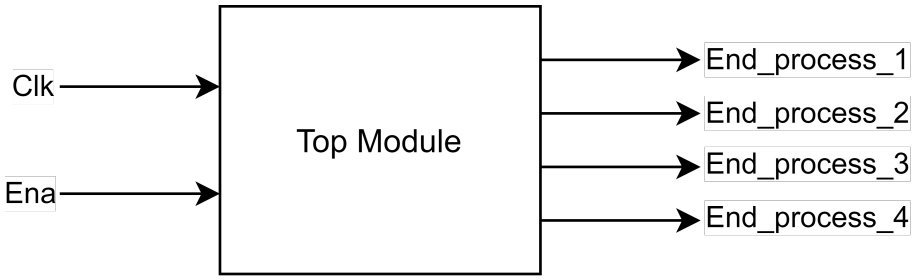

Figure 7: Top Level Module

This is the module that contains instances of all the lower-level modules that are required to implement the matrix multiplication. This module contains instances of clock_divider, core, end_core, memory_controller, IRAM and DRAM modules.

Inputs and outputs of this module are listed below.

Port	Description
Clk	Input signal that contains 50 MHz clock signal from the FPGA
Ena	Input signal to indicate the start of the process
End_process_1	Output signal to indicate the end of the process of core 1
End_process_2	Output signal to indicate the end of the process of core 2
End_process_3	Output signal to indicate the end of the process of core 3
End_process_4	Output signal to indicate the end of the process of core 4

Table 2: Inputs and outputs of top level module

3.3.2 Processor

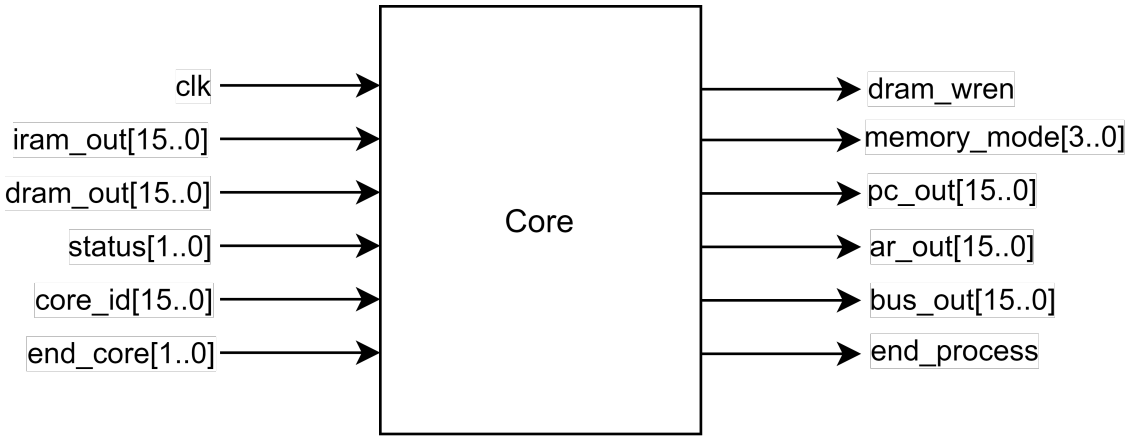

Figure 8: Processor

This module is used to instantiate multiple cores in the top module. This can be considered as the base processing unit of the project. This module contains instances for units listed below.

- Control Unit – State machine for the processor, generates control signals.
- Bus – Transport data between units
- ALU – Arithmetic and Logic Unit
- PC – Program counter
- AC – Accumulator
- AR – Address Register
- DR, IR – Data Register, Instruction Register
- ID, I, J, K, A, SUM – Special Purpose Registers
- R – General Purpose Register

Inputs and outputs of this module are listed below.

Port	**Description**
clk	Input signal that contains scaled clock signal from the clock divider
status	Input signal to indicate the start of the process
iram_out	Input signal with the output data of IRAM
dram_out	Input signal with the output data of DRAM
core_id	Input signal which contains the core id of the respective core
end_core	Input signal to terminate the processor
dram_wren	Output signal to DRAM write enable signal
iram_wren	Output signal to IRAM write enable signal
memory_mode	Output signal with the control signal for memory controller module
pc_out	Output signal with the address for the IRAM
ar_out	Output signal with the address for the DRAM
bus_out	Output signal with the data for the DRAM
end_process	Output signal which shows the end of the process

Table 3: Inputs and outputs of the Processor

3.3.3 Data Memory (DRAM)

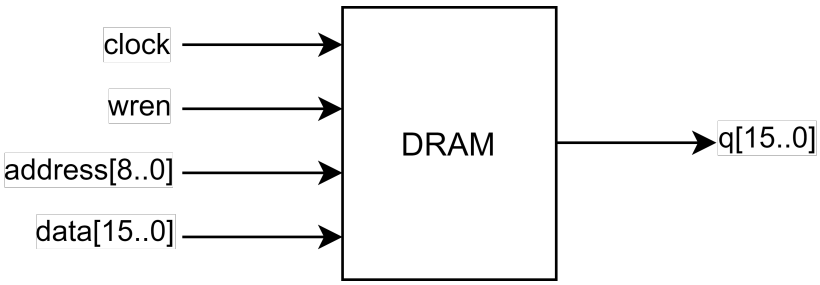

Figure 9: DRAM

This module is used to store data for the processor. Two matrices and other required data is stored in this. After the matrix multiplication final results are written in to this memory. A pre-built

module, RAM: 1-PORT module was used to design this module. Data memory was designed with a memory size of 512, where each word is 16 bits wide.

Port	Description
clock	Input signal that contains 50 MHz clock signal from the FPGA
wren	Input signal with the write enable signal
address	Input signal with the address that should be read
q	Output signal with the instruction

Table 4: Port description for the DRAM

3.3.4 Instruction Memory (IRAM)

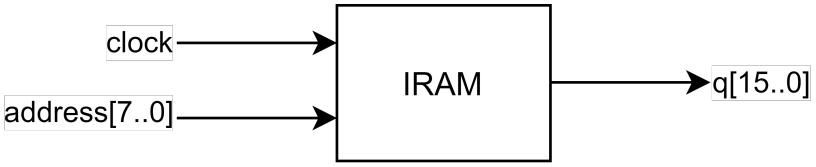

Figure 10: IRAM

This module is used to store instructions for the processor. A pre-built module, RAM: 1-PORT module was used to design this module. Instruction memory was designed with a memory size of 256, where each word is 16 bits wide.

Port	Description
clock	Input signal that contains 50 MHz clock signal from the FPGA
address	Input signal with the address that should be read
q	Output signal with the instruction

Table 5: Port description for the IRAM

3.3.5 Control Unit

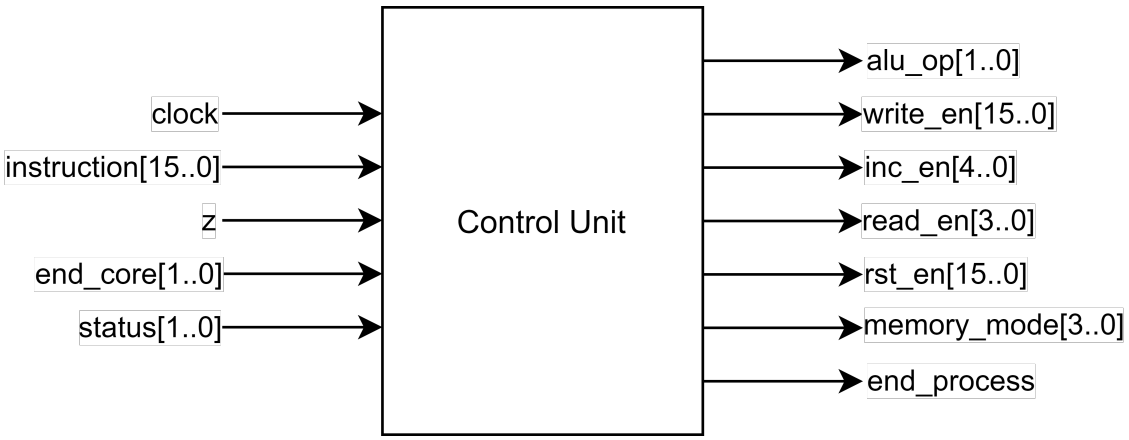

Figure 11: Control Unit

This module manages the operations in the processor. The state machine of the processor is implemented in this module. Control unit takes instructions from IRAM and generates control signals to manage other modules in the processor. Input signals and output signals of this module is given in the following table.

Port	**Description**
clock	Input signal that contains scaled clock signal from the clock divider
instruction	Input signal with the instruction from the IRAM
z	Input signal with the flag from ALU
end_core	Input signal to terminate the processor
status	Input signal to indicate the start of the process
alu_op	Output signal with the control signal for ALU
write_en	Output signal with the control signal for write enable signal for registers
read_en	Output signal with the control signal for read enable signal for BUS
inc_en	Output signal with the control signal for increment signal for registers
rst_en	Output signal with the control signal for reset signal for registers
memory_mode	Output signal with the control signal for memory controller module
end_process	Output signal which shows the end of the process

Table 6: Inputs and outputs of the Control Unit

3.3.5.1 Write Enable

This is a 16 bit wide control signal which is used to control input data of the registers and other components connected to the BUS. Each bit is related to 16 different inputs of registers. If bit value in the write_en[ ] signal equals to 1, the register will take the data from the corresponding input.

Bit position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Input	PC	DR	AR	IR	R	*	I	*	*	A	SUM	AC	DRAM	*	AR (from R)	AC (from ALU)

Table 7: Bit assignment for the write enable signal

* : bit is not used as a control signal
If there is no specific source mentioned, all the input data comes from BUS.

3.3.5.2 Reset Enable

This is a 16 bit wide control signal which is used to reset the value of the registers. If rst_en[i] equals to 1, i^{th} register value will be reset to 0 (in ID register the value will be reset to core id).

Bit position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register	PC	*	AR	IR	R	ID	I	J	K	A	SUM	AC	*	*	*	*

Table 8: Bit assignment for the Reset enable signal

* : bit is not used as a control signal

3.3.5.3 Increment Enable

This is a 5 bit wide control signal which is used to increment the value of 5 registers by 1. If inc_en[i] equals to 1, i^{th} register value will be incremented by 1.

Bit position	4	3	2	1	0
Register	PC	AR	R	J	K

Table 9: Bit assignment for the Increment enable signal

3.3.6 Arithmetic And Logic Unit (ALU)

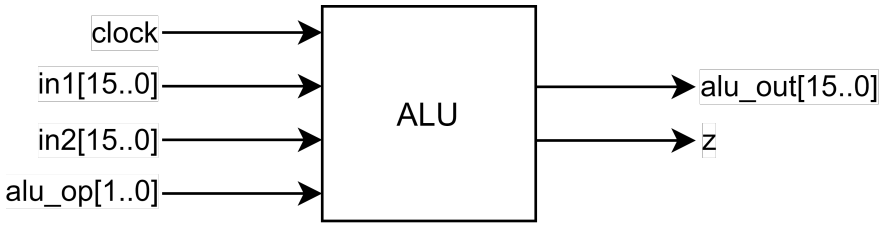

Figure 12: ALU

This module is used to manage all the arithmetic and logic operations in the processor. Input signal and output signals of the module are given below.

Port	**Description**
clock	Input signal that contains scaled clock signal from the clock divider
in1	Input signal with data from AC
in2	Input signal with data from the Bus
alu_op	Input signal with the control signal for ALU
alu_out	Output signal which contains data after the operation
z	Output signal which used as a flag after COMP operation

Table 10: Port description for the ALU

alu_op:

This is a 2 bit wide control signal which is used to select the operations in ALU.

alu_op signal	**Operation**	**Description**
01	Addition	alu_out <= in1 + in2
10	Comparison	If (in1<in2): z<=1, else: z<=0
11	Multiplication	alu_out <= in1*in2

Table 11: ALU operation signal

3.3.7 Registers

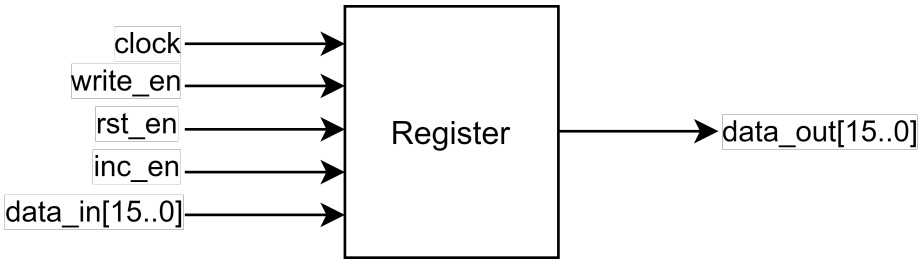

Figure 13: Register

This module is used to implement 16-bit registers in the processor. Seven special purpose registers and general-purpose register R were implemented using this module.

Port	Description
clock	Input signal that contains scaled clock signal from the clock divider
write_en	Input signal with control signal to enable input data from Bus
rst_en	Input signal with control signal to reset the register value to 0
inc_en	Input signal with control signal to increment the register value by 1
data_in	Input signal with the data from Bus
data_out	Output signal of the register

Table 12: Port description for the Register

3.3.8 Accumulator

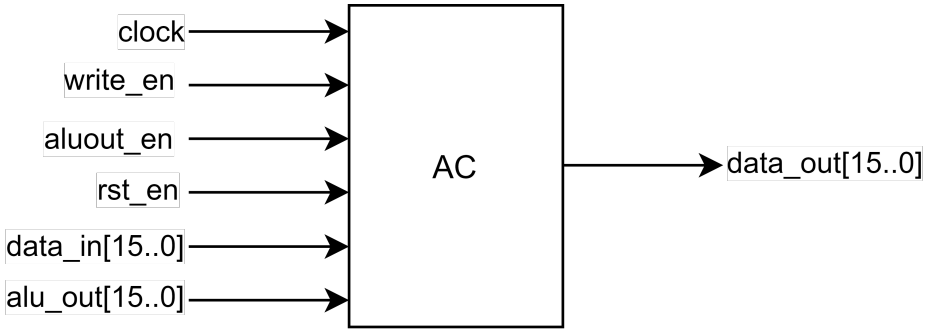

Figure 14: Accumulator

This module is used as a special purpose 16 bit register which stores data from ALU after its operations. In addition to the data input from Bus, output of the ALU is directly connected to the AC.

Port	Description
clock	Input signal that contains scaled clock signal from the clock divider
write_en	Input signal with control signal to select input data from Bus
aluout_en	Input signal with control signal to select input data from ALU output
rst_en	Input signal with control signal to reset the value of AC to 0
data_in	Input signal with the data from Bus
alu_out	Input signal with the data from ALU output
data_out	Output signal of the AC this is connected to both ALU and Bus

Table 13: Port description for the ALU

3.3.8.1 Data Registers

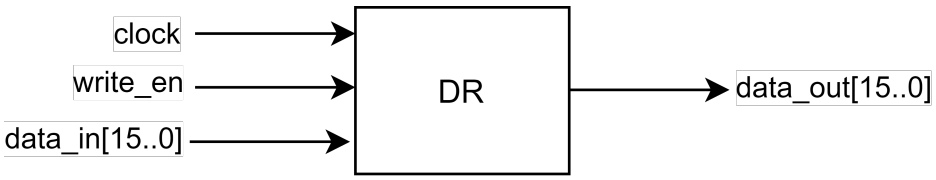

Figure 15: Data Register

This module is used as a special purpose 16-bit register, which stores the data that has been read from the DRAM or that has to be written to the DRAM.

Port	Description
clock	Input signal that contains scaled clock signal from the clock divider
write_en	Input signal with control signal to enable input data from Bus
data_in	Input signal with the data from Bus
data_out	Output signal of the DR

Table 14: Port description for the Data Register

3.3.8.2 Address Registers

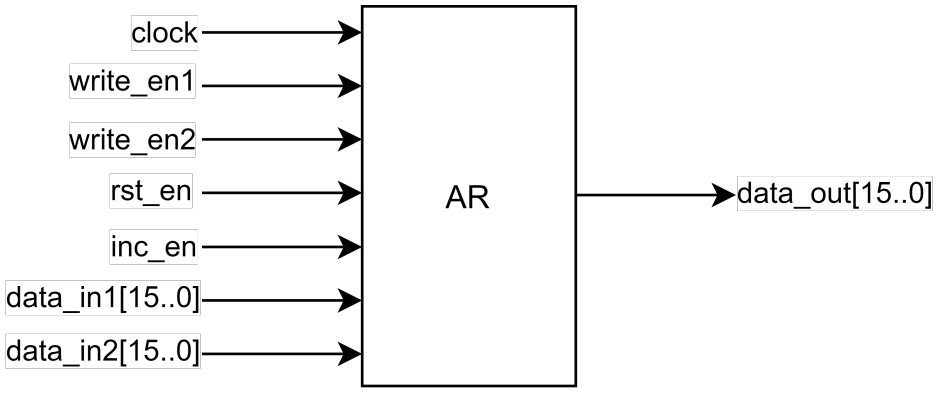

Figure 16: Address Registers

This module is used as a special purpose 16-bit register, which stores the 16-bit address of the DRAM block that is being written or read. This register gets its input data from 2 sources: bus

and R register.

Port	Description
clock	Input signal that contains scaled clock signal from the clock divider
write_en1	Input signal with control signal to select input data from Bus
write_en2	Input signal with control signal to select input data from register R
rst_en	Input signal with control signal to reset the value of AR to 0
inc_en	Input signal with control signal to increment the value of AR by 1
data_in1	Input signal with the data from Bus
data_in2	Input signal with the data from register R
data_out	Output signal of the AR

Table 15: Port description for the AR

3.3.9 Bus

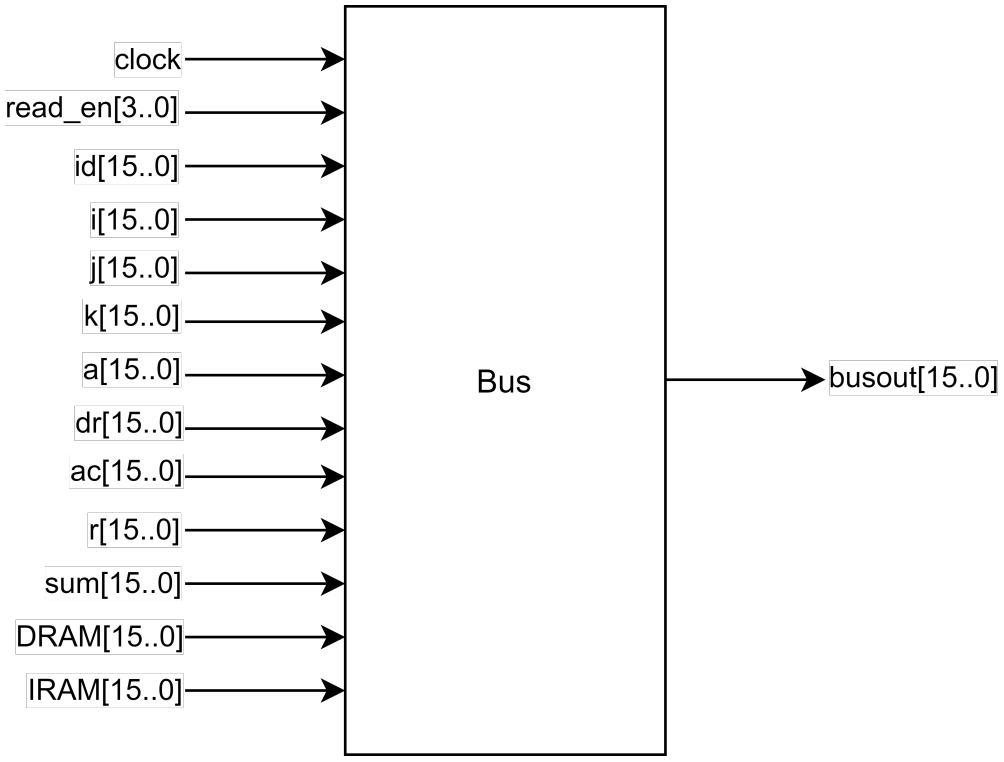

Figure 17: Bus

This module is used to transport data between registers and memory. This module will fetch data from one of the inputs according to the control signal "read_en", and keeps it until one of the inputs to the bus(register out values) changes.

3.3.9.1 Read Enable

This is a 4 bit wide control signal which is used to control the input data of the Bus. Outputs of 9 registers, data output from DRAM and data output from IRAM is connected to the Bus.

read_en	Data fetched from
0001	DR
0010	R
0011	ID
0100	I
0101	J
0110	K
0111	A
1000	SUM
1001	AC
1010	DRAM
1011	IRAM

Table 16: Bit assignment for the Read enable signal

3.3.10 Memory Controller

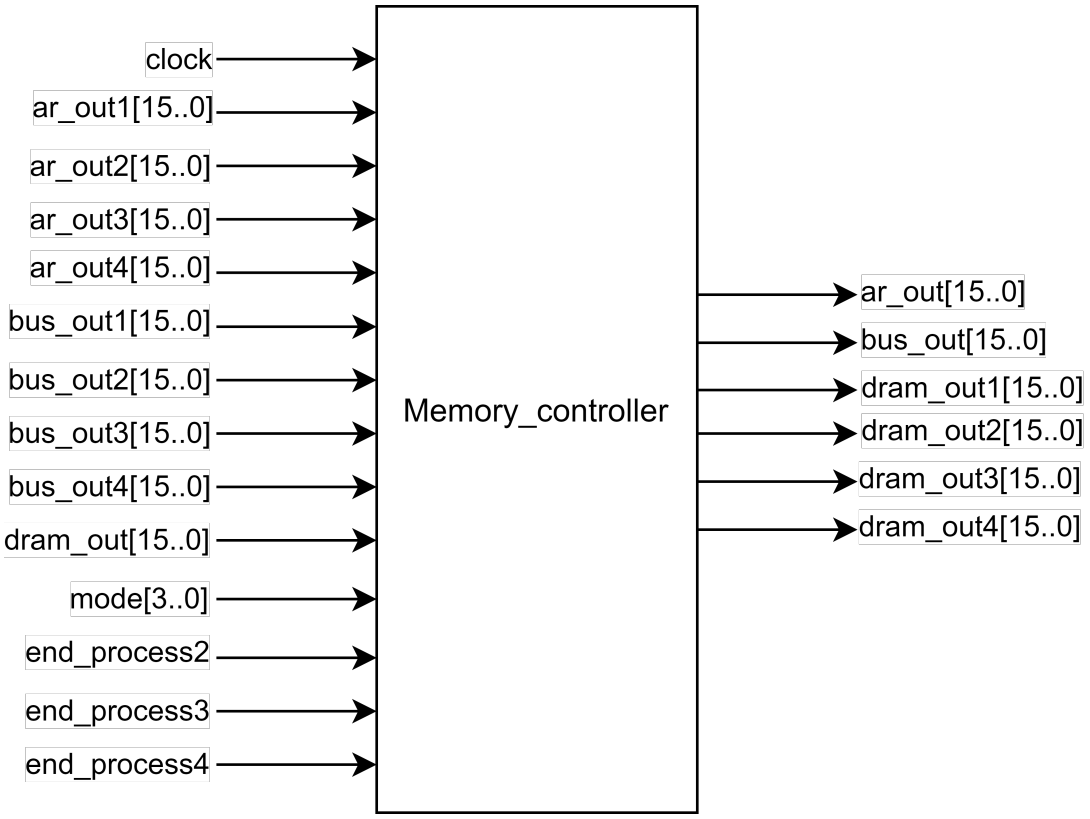

Figure 18: Memory Controller

This module was used to manage the data memory access for the multicore processor. As the design consists of 4 parallel working cores, and memory only has one port for read and write operations a new component was introduced to control the memory access process.

Port	Description
clock	Input signal that contains scaled clock signal from the clock divider
ar_out1	Input signal with the memory address from core 1
ar_out2	Input signal with the memory address from core 2
ar_out3	Input signal with the memory address from core 3
ar_out4	Input signal with the memory address from core 4
bus_out1	Input signal with the data from core 1 to be written in memory
bus_out2	Input signal with the data from core 2 to be written in memory
bus_out3	Input signal with the data from core 3 to be written in memory
bus_out4	Input signal with the data from core 4 to be written in memory
dram_out	Input signal with the data read from memory
mode	Input signal with the data from core 1 to control the memory access
end_process2	Input signal that indicates end of the process of core 2
end_process3	Input signal that indicates end of the process of core 3
end_process4	Input signal that indicates end of the process of core 4
ar_out	Output signal with the memory address to DRAM
bus_out	Output signal with the data to be written in DRAM
dram_out1	Output signal with the data from DRAM to core 1
dram_out2	Output signal with the data from DRAM to core 2
dram_out3	Output signal with the data from DRAM to core 3
dram_out4	Output signal with the data from DRAM to core 4

Table 17: Port description for the Memory Controller

This module takes the 4-bit control signal "mode" from the first core and changes memory access settings accordingly.

Mode	Operation
0000	All cores read from the same memory location
0001	Data read from memory for core 1
0010	Data read from memory for core 2
0011	Data read from memory for core 3
0100	Data read from memory for core 4
0101	Memory write for core 1
0110	Memory write for core 2
0111	Memory write for core 3
1000	Memory write for core 4

Table 18: Control Signal for memory access

3.3.11 Clock Divider

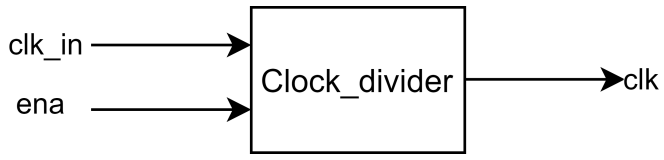

Figure 19: Register:Clock Divider

This module was used to slow down the clock speed. Since there were some malfunctions with higher clock speeds this was used to get a 10 times slower clock signal from the original 50 MHz clock.

Port	Description
clk_in	Input signal that contains 50 MHz clock signal from the FPGA
ena	Input signal which indicates the start of the operation
clk	Output signal that contains scaled clock signal from the clock divider

Table 19: Port description for the Clock Divider

3.3.12 CoreID

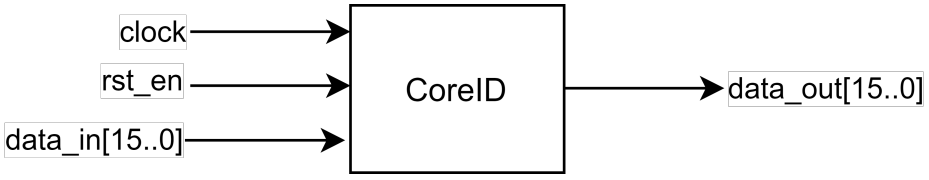

Figure 20: Register:CoreID

This module is used as a special purpose 16-bit register, which stores the ID of the corresponding core. Core ID is given to this register as an input signal.

Port	Description
clock	Input signal that contains scaled clock signal from the clock divider
rst_en	Input signal with control signal to reset the register value to the core ID
data_in	Input signal with the core ID
data_out	Output signal of the register

Table 20: Port description for the CoreID

3.3.13 End Core

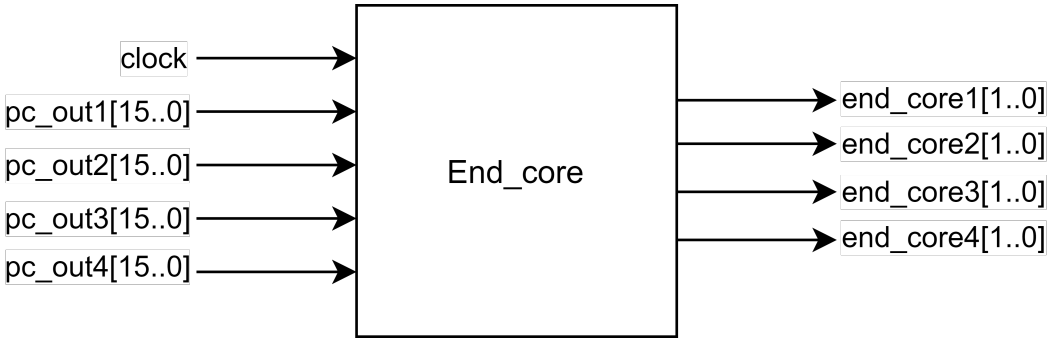

Figure 21: Register:End Core

This module is used to terminate the processing of cores. It compares "PC" values of all the other cores with PC value of core 1 to determine whether or not to end the processes of those cores.

4 Algorithms and Design Considerations

The task given was to design a multicore processor to multiply two matrices. As the first step for this task a matrix multiplication algorithm was designed only considering a single core.

4.1 Matrix Multiplication Algorithm

4.1.0.1 Single Core Implementation

In addition to the two matrices, their sizes and starting memory addresses of each matrices are stored in the memory to be taken as inputs.

Address	0	1	2	3	4	5	ap-bp	bp-cp	cp..
Data	m	n	p	ap	bp	cp	Matrix A	Matrix B	0

Table 21: DRAM (Single Core)

Matrix multiplication algorithm consists of 3 main loops. First loop iterates through number of rows in the final matrix. Second loop iterates through the elements in each row. Third loop is used to calculate one element in the matrix.

Consider matrix multiplication:

$$A(m \times n) \times B(n \times p) = C(m \times p)$$

m = number of rows in matrix A
n = number of columns in matrix A = number of rows in matrix B
p = number of columns in matrix B
ap = address of the first element of matrix A (= 7)
bp = address of the first element of matrix B (= ap + m*n)
cp = address of the first element of matrix C (= bp + n*p)
i = current row of the matrix A
j = current column of the matrix A (current row of the matrix B)
k = current column of the matrix B

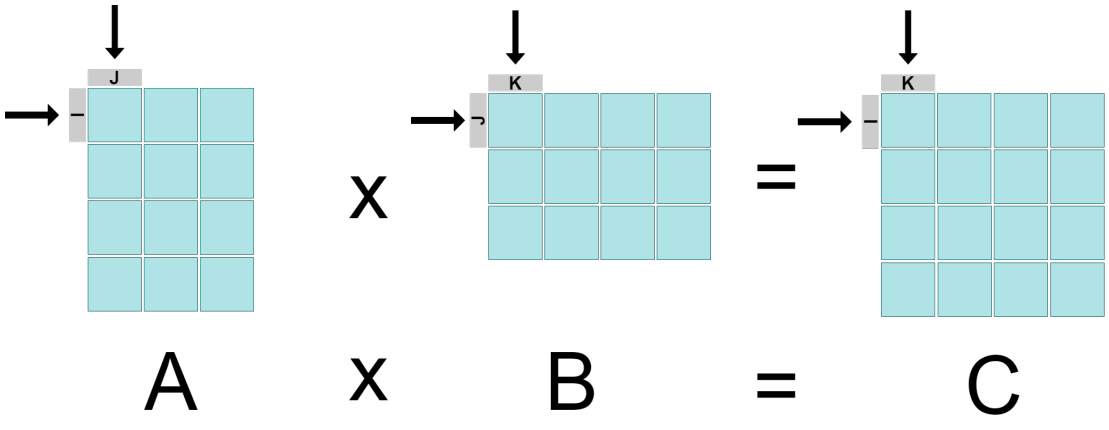

Figure 22: Matrix Multiplication

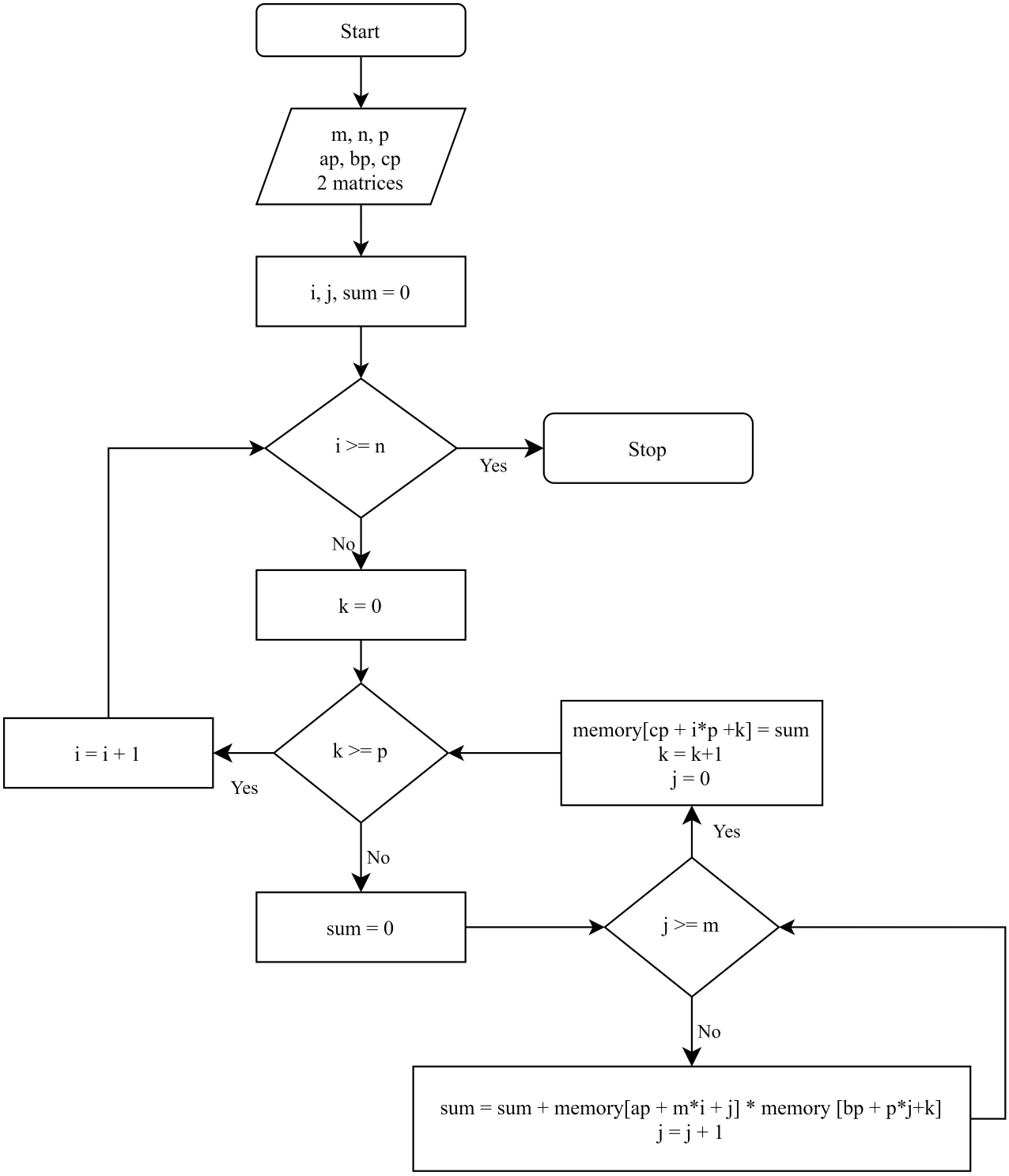

Figure 23: The Single Core Algorithm

4.1.0.2 Multi Core Implementation

The processor was designed with 4 cores and user has the ability to choose number of cores that should be used in the process. In order to achieve that, "noc" (Number of cores) parameter was added into the input data of memory.

Address	0	1	2	3	4	5	6	ap-bp	bp-cp	cp..
Data	noc	m	n	p	ap	bp	cp	Matrix A	Matrix B	0

Table 22: DRAM (Multi Core)

All the cores are given a core ID (0,1,2,3) at the beginning of the process. A control unit of each core decides whether to continue its operations or not by comparing its core ID with the "noc". In the multicore operation each active core will parallelly calculate the elements of a row in the resulting matrix.

A single core is given the task of calculating elements in several rows in each iterations. The row number can be calculated as:

Row number = core_id * iteration + noc

If the row number is larger than the total number of rows, the relevant core will end its operations.

Example:

Consider the multiplication of two 5x5 matrices A and B. It will generate a new 5x5 matrix C. If number of cores equals to 3 the processor will use 3 cores to carry out the operations.

After comparing core ID and noc 4^{th} core will end its operations. Then in the first iteration the first 3 cores will compute the values of elements in the first 3 rows of matrix C. Then in the second iteration 3^{rd} core will end its operations, and first and second cores will calculate the elements in the remaining 2 rows. In the 3^{rd} iteration 1^{st} and 2^{nd} cores will end its operations.

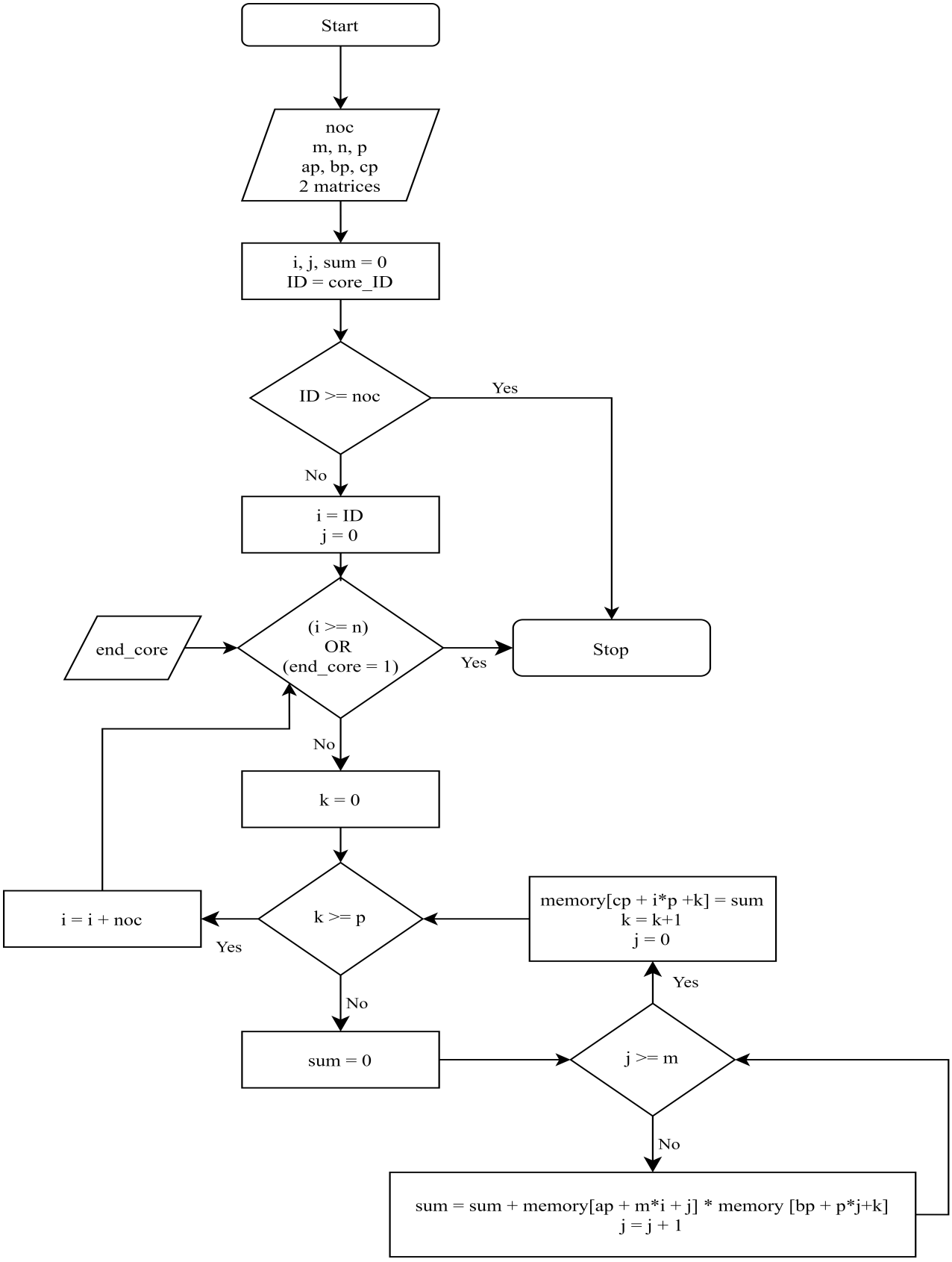

Figure 24: The Multi Core Algorithm

4.2 Assembly Code

Detailed assembly code which was used to implement multi core algorithm was given below.

```
1.  // A*B = C matrix multiplication
2.  RSTALL   // ID <- Core ID (0,1,2,3,4)   // I,j,sum <-0
3.  LDAC noc   //AC <- M[noc] = No of cores
4.  MVIDR     //R <- ID
5.  INCR      // R<-R+1 (Core_ID+1)
6.  COMP      // if AC < R, z =1, else z=0 (comparing core ID with number of cores)
7.  JMPZ mend
8.  RSTI      // I<-ID
9.  RSTJ      //J <- 0
10. loop:
11.    MVIR     //R <- I
12.    LDAC n   // no.of rows in A = no.of rows in C
13.    INCR          // I <- I + 1
14.    COMP         // (comparing current row with total number of rows)
15.    JMPZ mend      // JUMP if z=1
16.    RSTK          //K <-- 0
17.    loop1:
18.       MVKR      //R <- K
19.       LDAC p    //no.of columns in B = no.of columns in C
20.       INCR
21.       COMP
22.       JMPZ mend1  // JUMP if z=1
23.       RSTSUM     //SUM <- 0
24.       JUMP loop2
25.       mstore:
26.          MVIR    //R <- I
27.          LDAC p  //
28.          MULR    // AC <- AC*R
29.          MVKR    // R <-K
30.          ADDR    // AC <-- AC+R
31.          MVACR   //R <-AC
32.          LDAC cp //AC <-M[cp] (addr of first element of C matrix)
33.          ADDR    // AC <- AC+R
34.          MVACR   // R <- AC
35.          MVSUMAC    // AC<-SUM
36.          STAC       // M[R] <- AC
37.          INCK       // K<-K+1
38.          RSTJ
```

```
39.          JUMP loop1
40.      loop2:
41.          MVJR      //R<-J
42.          LDAC m     //no.of columns in A = no.of rows in B
43.          INCR
44.          COMP
45.          JMPZ mstore
46.          MVIR
47.          LDAC m
48.          MULR       //AC<-AC*R(i*m)
49.          MVJR       //R<-J
50.          ADDR       //AC<-AC+R
51.          MVACR      //R<-AC
52.          LDAC ap    //AC <- M[ap] (addr of first element of A matrix)
53.          ADDR      //AC<-AC+R (AC+AP)
54.          MVACR      //R <-AC
55.          LDAC R    //AC <- M[R]
56.          MVACA      //A<-AC
57.          MVJR       //R<-J
58.          LDAC p     //AC<-M[p]
59.          MULR       //AC<-AC*R
60.          MVKR       //R<-K
61.          ADDR       //AC<-AC+R
62.          MVACR      //R<-AC
63.          LDAC bp    //AC<-M[bp]
64.          ADDR      //AC<-AC+R (AC+BP)
65.          MVACR
66.          LDACR    //AC <- M[R]
67.          MULA       //AC<-AC*A
68.          ADDSUM     //AC<-AC+SUM
69.          MVACSUM    //SUM <- AC
70.          INCJ       //J<-J+1
71.          JUMP loop2
72.    mend1:
73.       MVIR       //R<-I
74.       LDAC noc
75.       ADDR
76.       MVACI
77.       JUMP loop
78. mend:
79.    ENDOP
```

5 Design Optimization

5.1 Minimizing the number of instructions in ISA

In the initial architecture for the processor 41 instructions were defined for the ISA. In our new optimized ISA these instructions were reduced to 28 instructions. This reduction was achieved by doing the following changes to the processor design.

1. There were dedicated registers to store the dimensions of the matrices (n,m,p) and the starting point memory addresses of each matrix(ap,cp,bp).Therefore when loaded from memory they were transferred from AC register to the dedicated registers using separate instructions. These registers were eliminated from our optimized design and whenever the above values are required it is loaded from the data memory and is temporarily stored in the general purpose register R. Therefore all these operations were replaced using a single instruction (MVACR) and multiple memory accesses.
2. In the initial design our memory was a 8 bit memory and hence the resulting matrix after multiplication was stored in two memory blocks therefore the register S was used to store the number of memory blocks needed for each element of the matrix. As our new design has a 16 bit memory this register and instructions relating to it was omitted.
3. We had dedicated instructions to do subtractions with changing the 'z' flag for the matrix dimensions. In our new design these are replaced by a single COMP instruction which compares the values of registers AC and R and flags 'z' accordingly.
4. The instructions to perform multiplication and addition was also optimized in our new design leaving only the desired instructions.

6 Problems Faced & Solutions

- Hardware debugging
  - Debugging using FPGA board was one of the major challenges we faced during the design phase. Model Sim software provides the ability to check any internal signal or register value during simulations. However, in the hardware implementation we had to use inbuilt indicators of FPGA board to check internal signals and register values of the processor. Some of those indicators are,
    * Seven Segment Displays : to indicate register values, check memory output, check bus output
    * LEDs : check control signals like "write_enable" and "read_enable"
    * Push buttons: to generate a manual clock
    * Switches: give custom inputs

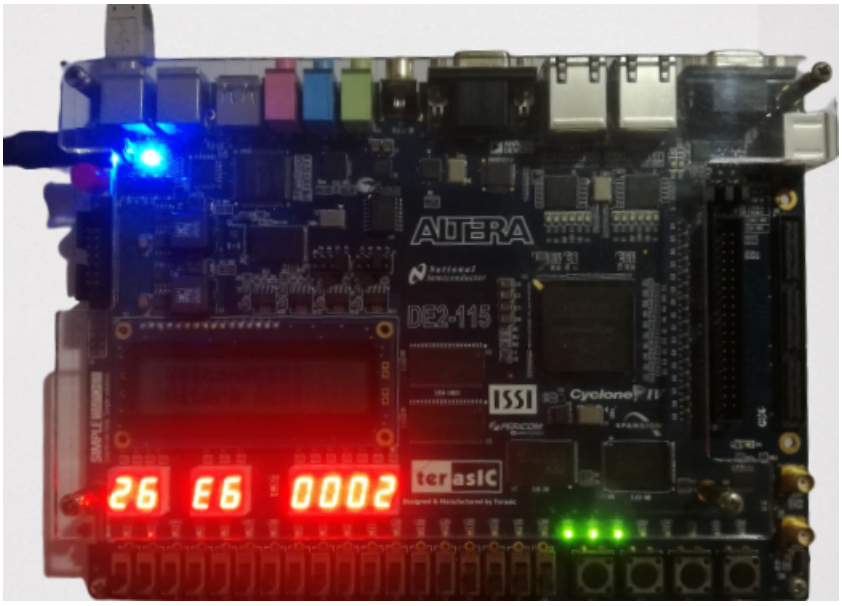

Figure 25: FPGA Development Board

- Glitches due to the high clock speed
  - Some of the modules like ALU was first tested separately prior adding to the processor. These modules worked as intended with a manually generated clock signal. However, whenever the 50 MHz clock signal was used these modules did not generate the expected outcomes. In order to overcome this we used a module named "clock_divider" to scale down the clock speed by a factor of 10.
- Long compilation times
  - Tested all the modules separately before combining them.
- Difficulty in developing algorithms and testing using FPGA development board when groups members are in different places due to Covid'19 pandemic situation.
  - We used online video conference meetings for communication and programmed modules. Also, we used ModelSim simulation tool to simulate the design without the FPGA board.

7 Assembler and Simulator

7.1 Assembler

The assembler is used to convert the assembly language code to machine code. In our project, the assembler written in python reads a source code file (.s format) which has the algorithm in assembly language code for the project. The assembler python code has the ISA and the relevant opcodes saved as a dictionary and maps each line in the assembly language code to the proper binary code and stores these in the required arrangement in a Memory Initialization File (.mif format). This MIF file has the content for the Instruction Memory of the processor.

```
WIDTH=16;
DEPTH=256;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        000  :   0000000000010111;
        001  :   0000000000000001;
        002  :   0000000000000000;
        003  :   0000000000001100;
        004  :   0000000000010100;
        005  :   0000000000010001;
        006  :   0000000000010101;
        007  :   0000000001011010;
        008  :   0000000000011000;
        009  :   0000000000011001;
        00A  :   0000000000001001;
        00B  :   0000000000000001;
        00C  :   0000000000000001;
        00D  :   0000000000010100;
        00E  :   0000000000010001;
        00F  :   0000000000010101;
        010  :   0000000001011010;
        011  :   0000000000011010;
        012  :   0000000000001010;
        013  :   0000000000000001;
        014  :   0000000000000011;
        015  :   0000000000010100;
        016  :   0000000000010001;
        017  :   0000000000010101;
        018  :   0000000001010011;
        019  :   0000000000011011;
        01A  :   0000000000010110;
        01B  :   0000000000101101;
        01C  :   0000000000001001;
```

Figure 26: Generated MIF file by the Assembler

7.2 Simulation

7.2.1 Python Simulation

The Python simulation was done at the beginning of the project in order to verify if the algorithm we developed using the defined ISA works properly and gives out the expected results. Here the DRAM was given as list where each element corresponds to a data memory block. The singlecore() function does the exact same process of a single core in our processor design and for a multicore architecture the function is instantiated based on the number of cores in our design. In the defined singlecore() function each line or a couple of lines (when a condition is checked) is written according to the order of the instructions stored in IRAM.

7.2.2 Model Sim Simulation

The Modelsim simulation was done through a test bench. Test benches are also used to verify FPGA designs. In test benches the behaviour of the hardware is depicted through waveforms. In our project we have written the test bench for the top module code. As shown in A.19 in the Appendix section, the inputs to the top module are stated as registers and an initial value is hardcoded to start the processor. A clock of period 20ns (50MHz) is defined here which is initially assigned to the value zero. The Modelsim simulator will by default show the waveforms of the inputs of the top module and the relevant outputs.

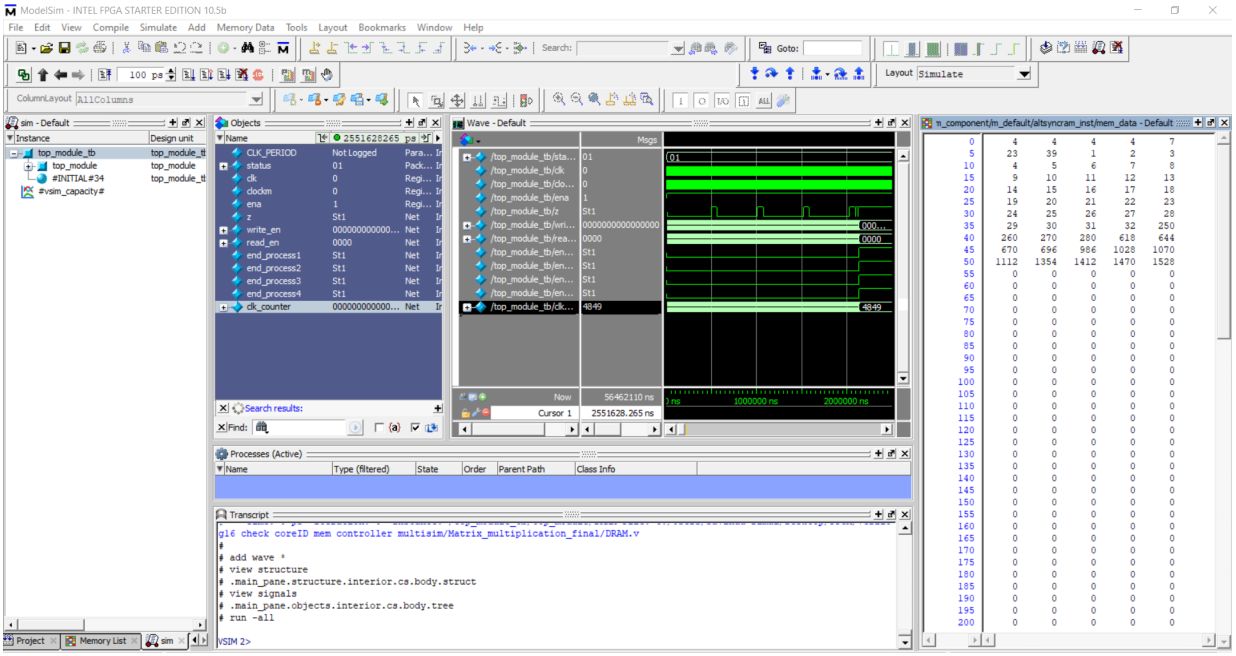

Figure 27: ModelSim Simulation

8 Performance Evaluation

After completing the multi core processor for matrix multiplication it was tested using different matrices. The performance analysis of the multicore processor was done by comparing the total time consumption.

Four square matrices were used in these tests and total number of clock cycles to complete the matrix multiplication was taken as the time measurement. Each matrix multiplication was done using different number of cores (i.e., 1,2,3,4 cores). Those results are given in the following graph.

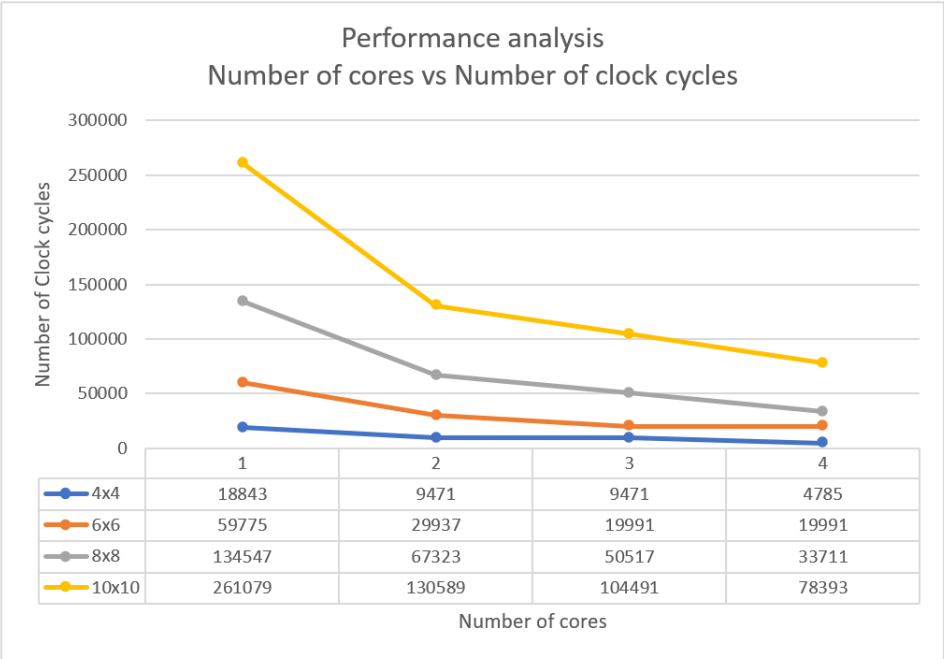

Figure 28: Performance Analysis

The time consumption for same two matrix multiplication for different number of cores depends on the number of iterations. Therefore, time consumption will decrease with the number of cores.

Example: Considering multiplication of two 6x6 matrices,

Number of cores	Number of iterations
1	6
2	3
3	2
4	2

Table 23: Performance Analysis

As a single core has the highest number of iterations, it has the highest time consumption (59775 clock cycles). Number of iterations has its lowest value when number of cores equal to 3 and 4. Therefore, it has the lowest time consumption (19991 clock cycles).

References

[1] ComputerGeek. Cpufig, 2021. https://www.computerhunger.com/what-is-the-cpu-and-its-function-components-and-diagram/.

[2] The Editors of Encyclopaedia Britannica. Cpu, 2021. https://www.britannica.com/technology/central-processing-unit.

[3] Xilinx. Fpga, 2021. https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[4] Terasic. De2-115, 2003-2013. https://www.intel.com/content/dam/www/programmable/us/en/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf.

[5] Intel. Quartus prime lite edition, 2020. https://fpgasoftware.intel.com/?edition=lite.

[6] Intel. Modelsim, 2020. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html.

A Appendix

A.1 Assembly Code

```
RSTALL
LDAC
16'D0
MVIDR
INCR
COMP
JMPZ
16'D90
RSTI
RSTJ
MVIR
LDAC
16'D1
INCR
COMP
JMPZ
16'D90
RSTK
MVKR
LDAC
16'D3
INCR
COMP
JMPZ
16'D83
RSTSUM
JUMP
16'D45
MVIR
LDAC
16'D3
MULR
MVKR
ADDR
MVACR
LDAC
16'D6
ADDR
MVACR
MVSUMAC
STAC
INCK
RSTJ
JUMP
16'D18
MVJR
LDAC
16'D2
INCR
COMP
JMPZ
16'D28
MVIR
```

```
LDAC
16'D2
MULR
MVJR
ADDR
MVACR
LDAC
16'D4
ADDR
MVACR
LDACR
MVACA
MVJR
LDAC
16'D3
MULR
MVKR
ADDR
MVACR
LDAC
16'D5
ADDR
MVACR
LDACR
MULA
ADDSUM
MVACSUM
INCJ
JUMP
16'D45
MVIR
LDAC
16'D0
ADDR
MVACI
JUMP
16'D10
ENDOP
```

A.2 Top Level Module

```
module top_module(
        input [1:0] status,
        input clk,
        input clockm,
        input ena,
        output z,
        output [15:0]write_en,
        output [3:0] read_en,
        output end_process1,
        output end_process2,
        output end_process3,
        output end_process4,
        output reg[31:0] clk_counter
        );

```

```
16          wire clock;
17          wire [15:0] iram_out;
18          wire [15:0] dram_out; wire [15:0] dram_out1; wire [15:0] dram_out2;
    wire [15:0] dram_out3; wire [15:0] dram_out4;
19          wire dram_wren;
20          wire iram_wren;
21          wire [15:0] pc_out1; wire [15:0] pc_out2; wire [15:0] pc_out3; wire
    [15:0] pc_out4;
22
23          wire [1:0] end_core1; wire [1:0] end_core2; wire [1:0]end_core3;
    wire [1:0]end_core4;
24          wire [3:0] memory_mode;
25
26          wire [15:0] ar_out; wire [15:0] ar_out1; wire [15:0] ar_out2; wire [
    15:0] ar_out3; wire [15:0] ar_out4;
27          wire [15:0] bus_out; wire [15:0] bus_out1; wire [15:0] bus_out2;
    wire [15:0] bus_out3; wire [15:0] bus_out4;
28          wire [15:0] ac_out1; wire [15:0] ac_out2; wire [15:0] ac_out3; wire
    [15:0] ac_out4;
29          wire [15:0] dr_out;
30          wire [15:0] ir_out;
31          wire [15:0] i_out1;wire [15:0] i_out2;wire [15:0] i_out3;wire [15:0]
     i_out4;
32
33
34  //counter for the performance analysis
35  initial begin
36          clk_counter = 32'd0;
37  end
38
39   always @ (posedge clock )
40          begin
41          if(~end_process1)
42          begin
43          clk_counter = clk_counter + 1;
44          end
45          end
46
47
48
49  clock_divider clkdvdr(.inclk(clk),.ena(ena),.clk(clock));
50
51  core core1(.clock(clock), .iram_out(iram_out),.dram_out(dram_out1),.status(
    status),
52  .dram_wren(dram_wren),.iram_wren(iram_wren),.pc_out(pc_out1),.ar_out(ar_out1
    ),.bus_out(bus_out1),.end_process(end_process1),.ac_out(ac_out1),
53  .write_en(write_en),.read_en(read_en),.dr_out(dr_out),.ir_out(ir_out),.z(z)
    ,.core_id(16'd0),.memory_mode(memory_mode),.end_core(end_core1),.i_out(
    i_out1));
54
55  core core2(.clock(clock), .iram_out(iram_out),.dram_out(dram_out2),.status(
    status),
56  .dram_wren(),.iram_wren(),.pc_out(pc_out2),.ar_out(ar_out2),.bus_out(
    bus_out2),.end_process(end_process2),.ac_out(ac_out2),
57  .write_en(),.read_en(),.dr_out(),.ir_out(),.z(),.core_id(16'd1),.end_core(
    end_core2),.i_out(i_out2));
58
59  core core3(.clock(clock), .iram_out(iram_out),.dram_out(dram_out3),.status(
```

```
    status),
.dram_wren(),.iram_wren(),.pc_out(pc_out3),.ar_out(ar_out3),.bus_out(
    bus_out3),.end_process(end_process3),.ac_out(ac_out3),
.write_en(),.read_en(),.dr_out(),.ir_out(),.z(),.core_id(16'd2),.end_core(
    end_core3),.i_out(i_out3));

core core4(.clock(clock), .iram_out(iram_out),.dram_out(dram_out4),.status(
    status),
.dram_wren(),.iram_wren(),.pc_out(pc_out4),.ar_out(ar_out4),.bus_out(
    bus_out4),.end_process(end_process4),.ac_out(ac_out4),
.write_en(),.read_en(),.dr_out(),.ir_out(),.z(),.core_id(16'd3),.end_core(
    end_core4),.i_out(i_out4));


end_core end_core(.clock(clock),.pc_out1(pc_out1),.pc_out2(pc_out2),.pc_out3
    (pc_out3),.pc_out4(pc_out4),
.end_core1(end_core1),.end_core2(end_core2),.end_core3(end_core3),.end_core4
    (end_core4));


memory_controller memory_controller(.clock(clk),.ar_out1(ar_out1),.ar_out2(
    ar_out2),.ar_out3(ar_out3),.ar_out4(ar_out4),
.bus_out1(bus_out1),.bus_out2(bus_out2),.bus_out3(bus_out3),.bus_out4(
    bus_out4),.dram_out(dram_out),.mode(memory_mode),
.ar_out(ar_out),.bus_out(bus_out),.dram_out1(dram_out1),.dram_out2(dram_out2
    ),.dram_out3(dram_out3),.dram_out4(dram_out4),
.end_process2(end_process2),.end_process3(end_process3),.end_process4(
    end_process4));



IRAM_IP  IRAM(.address(pc_out1),.clock(clk),.data(),.wren(),.q(iram_out));

DRAM  DRAM(.address(ar_out),.clock(clk),.data(bus_out),.wren(dram_wren),.q(
    dram_out));


endmodule
```

A.3 The Proessor

```
module core(
        input clock,
        input [15:0]iram_out,
        input [15:0]dram_out,
        input [1:0]status,
        input [15:0] core_id,
        input [1:0] end_core,

        output dram_wren,
        output iram_wren,
        output [3:0]memory_mode,
        output [15:0]pc_out,
        output [15:0]ar_out,
        output [15:0]bus_out,
        output [15:0]ac_out,
```

```
        output [15:0]write_en,
        output [3:0] read_en,
        output [15:0] dr_out,
        output [15:0] ir_out,
        output [15:0] i_out,
        output end_process,
        output z
        );


        wire[1:0] alu_op;
        wire[4:0] inc_en;
        wire[15:0] rst_en;
        wire[15:0] id_out;
        wire[15:0] j_out;
        wire[15:0] k_out;
        wire[15:0] a_out;
        wire[15:0] r_out;
        wire[15:0] sum_out;
        wire[15:0] alu_out;



//CU//
control_unit CU(.clock(clock), .z(z),.instruction(ir_out) ,.alu_op(alu_op),.write_en(write_en) ,.inc_en(inc_en),.read_en(read_en),
.rst_en(rst_en),.end_process(end_process) ,.status(status),.memory_mode(memory_mode),.end_core(end_core) );

//BUS//
bus BUS( .clock(clock),.read_en(read_en),.id(id_out),.i(i_out),
.j(j_out),.k(k_out),.a(a_out),.dr(dr_out),.ac(ac_out),.r(r_out),.sum(sum_out),.DRAM(dram_out),.IRAM(iram_out),.busout(bus_out)) ;

//ALU//
alu ALU(.clock(clock),.in1(ac_out),.in2(bus_out),.alu_op(alu_op),.out(alu_out),.z(z));

//PC//
register_16 PC(.clk(clock),.write_en(write_en[15]),.rst_en(rst_en[15]),.inc_en(inc_en[4]),.data_in(bus_out),.data_out(pc_out));

//DR//
DR DR(.clk(clock),.write_en(write_en[14]),.data_in(bus_out),.data_out(dr_out));

//AR//
AR AR(.clk(clock),.write_en1(write_en[13]),.write_en2(write_en[1]),.rst_en(rst_en[13]),.inc_en(inc_en[3]),
.data_in1(bus_out),.data_in2(r_out),.data_out(ar_out));

//IR//
register_16 IR(.clk(clock),.write_en(write_en[12]),.rst_en(rst_en[12]),.inc_en(),.data_in(bus_out),.data_out(ir_out));


//R//
register_16 R(.clk(clock),.write_en(write_en[11]),.rst_en(rst_en[11]),.
```

```
    inc_en(inc_en[2]),.data_in(bus_out),.data_out(r_out));

//ID//
CoreID ID(.clk(clock),.rst_en(rst_en[10]),.data_in(core_id),.data_out(id_out
    ));

//I//
register_16 I(.clk(clock),.write_en(write_en[9]),.rst_en(rst_en[9]),.inc_en
    (),.data_in(bus_out),.data_out(i_out));

//J//
register_16 J(.clk(clock),.write_en(write_en[8]),.rst_en(rst_en[8]),.inc_en(
    inc_en[1]),.data_in(),.data_out(j_out));

//K//
register_16 K(.clk(clock),.write_en(write_en[7]),.rst_en(rst_en[7]),.inc_en(
    inc_en[0]),.data_in(),.data_out(k_out));

//A//
register_16 A(.clk(clock),.write_en(write_en[6]),.rst_en(rst_en[6]),.inc_en
    (),.data_in(bus_out),.data_out(a_out));

//SUM//
register_16 SUM(.clk(clock),.write_en(write_en[5]),.rst_en(rst_en[5]),.
    inc_en(),.data_in(bus_out),.data_out(sum_out));

//AC //
AC AC(.clk(clock),.write_en(write_en[4]),.rst_en(rst_en[4]),.aluout_en(
    write_en[0]),.data_in(bus_out),.alu_out(alu_out),.data_out(ac_out));


assign dram_wren = write_en[3];
assign iram_wren = write_en[2];

endmodule
```

A.4 Control Unit

```
module control_unit(input clock,
input z,
input [1:0]end_core,
input [15:0] instruction ,
output reg [1:0] alu_op,
output reg [15:0] write_en ,
output reg [4:0] inc_en,
output reg [3:0] read_en,
output reg [15:0] rst_en,
output reg end_process ,
output reg[3:0] memory_mode,
input [1:0] status );


reg [6:0] present = 7'd67;
reg [6:0] next = 7'd67;


```

```
19  parameter
20  fetch0 = 7'd67,
21  fetch1 = 7'd0,
22  fetch2 = 7'd1,
23  fetch3 = 7'd2,
24  fetch4 = 7'd3,
25
26  nop1 = 7'd4,
27
28  ldac1 = 7'd5,
29  ldac2 = 7'd6,
30  ldac3 = 7'd7,
31  ldac4 = 7'd8,
32
33
34  loadac1 = 7'd9,
35  loadac2 = 7'd10,
36  loadac3 = 7'd11,
37
38  stac1 = 7'd12,
39  stac2 = 7'd13,
40
41  mvacr1 = 7'd14,
42  mvaca1 = 7'd15,
43  mvacsum1 = 7'd16,
44  mvaci1 = 7'd17,
45  mvsumac1 = 7'd18,
46  mvir1 = 7'd19,
47  mvkr1 = 7'd20,
48  mvjr1 = 7'd21,
49  mvidr1 = 7'd22,
50
51  mulr1 = 7'd23,
52  mula1 = 7'd24,
53  addr1 = 7'd25,
54  addsum1 = 7'd26,
55  comp1 = 7'd27,
56
57  inck1 = 7'd28,
58  incj1 = 7'd29,
59  incr1 = 7'd30,
60
61  jmpzy1 = 7'd31,
62  jmpzy2 = 7'd32,
63
64  jmpzn1 = 7'd42,
65  jmpzn2 = 7'd43,
66
67  jump1 = 7'd33,
68  jump2 = 7'd34,
69
70  rstall1 = 7'd35,
71  rsti1 = 7'd36,
72  rstj1 = 7'd37,
73  rstk1 = 7'd38,
74  rstsum1 = 7'd39,
75
76  idle = 7'd40,
```

```
77  endop = 7'd41,
78
79  fetchx = 7'd44,
80  ldacx = 7'd45,
81  ldacy = 7'd46,
82  ldacz = 7'd68,
83
84  jmpzyx = 7'd47,
85  jumpx = 7'd48,
86  loadacx1 =7'd49,
87  loadacx2 =7'd50,
88  loadacx3 =7'd51,
89  loadacx4 =7'd52,
90  loadacx5 =7'd53,
91  loadacx6 =7'd54,
92  loadacx7 =7'd55,
93  loadacx8 = 7'd69,
94  loadacx9 = 7'd70,
95
96  stacx1 = 7'd56,
97  stacx2 = 7'd57,
98  stacx3 = 7'd58,
99  stacx4 = 7'd59,
100 stacx5 = 7'd60,
101 stacx6 = 7'd61,
102 stacx7 = 7'd62,
103 stacx8 = 7'd63,
104 stacx9 = 7'd64,
105 stacx10 = 7'd65,
106 stacy = 7'd66;
107
108 always @(posedge clock)
109 present <= next;
110
111 always @(posedge clock)
112 begin
113 if (present == endop)
114 end_process <= 1'd1;
115 else
116 end_process <= 1'd0;
117 end
118
119 always @(present or z or instruction or status)
120 case(present)
121 idle: begin
122 read_en <= 4'd0;
123 inc_en <= 5'b00000 ; // PC AR R J K
124 write_en <= 16'b0000000000000000 ;
125 rst_en <= 16'b0000000000000000 ;
126 alu_op <= 2'd0;
127 if (status == 2'b01)
128 next <= fetch1;
129 else
130 next <= idle;
131 end
132
133 fetch0: begin
134 rst_en <= 16'b1111111111111111 ;
```

```
135 next <= fetch1;
136 end
137
138 fetch1: begin
139 read_en <= 4'd0;
140 inc_en <= 5'b00000 ;
141 write_en <= 16'b0000000000000000 ;
142 rst_en <= 16'b0000000000000000 ;
143 alu_op <= 2'd0;
144 memory_mode <= 4'd0;
145 next <= fetch2;
146 end
147
148 fetch2: begin
149 read_en <= 4'd11;
150 inc_en <= 5'b10000 ;
151 write_en <= 16'b0100000000000000 ;
152 rst_en <= 16'b0000000000000000 ;
153 alu_op <= 2'd0;
154 next <= fetchx;
155 end
156
157 fetchx: begin
158 read_en <= 4'd0;
159 inc_en <= 5'b00000 ;
160 write_en <= 16'b0000000000000000 ;
161 rst_en <= 16'b0000000000000000 ;
162 alu_op <= 2'd0;
163 next <= fetch3;
164 end
165
166
167 fetch3: begin
168 read_en <= 4'd1;
169 inc_en <= 5'b00000 ;
170 write_en <= 16'b0001000000000000 ;
171 rst_en <= 16'b0000000000000000 ;
172 alu_op <= 2'd0;
173 next <= fetch4;
174 end
175
176 fetch4: begin
177 read_en <= 4'd0;
178 inc_en <= 5'b00000 ;
179 write_en <= 16'b0000000000000000 ;
180 rst_en <= 16'b0000000000000000 ;
181 alu_op <= 2'd0;
182 if (end_core==2'b10) next <= endop;
183 else if (instruction [15:0]==16'd28)    next <= nop1;
184 else if (instruction [15:0]==16'd1)     next <= ldac1;
185 else if (instruction [15:0]==16'd2)     next <= loadac1;
186 else if (instruction [15:0]==16'd3)     next <= stac1;
187 else if (instruction [15:0]==16'd4)     next <= mvacr1;
188 else if (instruction [15:0]==16'd5)     next <= mvaca1;
189 else if (instruction [15:0]==16'd6)     next <= mvacsum1;
190 else if (instruction [15:0]==16'd7)     next <= mvaci1;
191 else if (instruction [15:0]==16'd8)     next <= mvsumac1;
192 else if (instruction [15:0]==16'd9)     next <= mvir1;
```

```
193 else if (instruction [15:0]==16'd10)     next <= mvkr1;
194 else if (instruction [15:0]==16'd11)     next <= mvjr1;
195 else if (instruction [15:0]==16'd12)     next <= mvidr1;
196 else if (instruction [15:0]==16'd13)     next <= mulr1;
197 else if (instruction [15:0]==16'd14)     next <= mula1;
198 else if (instruction [15:0]==16'd15)     next <= addr1;
199 else if (instruction [15:0]==16'd16)     next <= addsum1;
200 else if (instruction [15:0]==16'd17)     next <= comp1;
201 else if (instruction [15:0]==16'd18)     next <= inck1;
202 else if (instruction [15:0]==16'd19)     next <= incj1;
203 else if (instruction [15:0]==16'd20)     next <= incr1;
204 else if (instruction [15:0]==16'd22)     next <= jump1;
205 else if (instruction [15:0]==16'd23)     next <= rstall1;
206 else if (instruction [15:0]==16'd24)     next <= rsti1;
207 else if (instruction [15:0]==16'd25)     next <= rstj1;
208 else if (instruction [15:0]==16'd26)     next <= rstk1;
209 else if (instruction [15:0]==16'd27)     next <= rstsum1;
210 else if ((instruction [15:0]==16'd21) & (z==1)) next <= jmpzy1;
211 else if ((instruction [15:0]==16'd21) & (z==0)) next <= jmpzn1;
212 else next <= endop;
213
214 end
215
216
217 ldac1: begin
218 read_en <= 4'd11;
219 inc_en <= 5'b10000 ;
220 write_en <= 16'b0100000000000000 ;
221 rst_en <= 16'b0000000000000000 ;
222 alu_op <= 2'd0;
223 memory_mode <= 4'd0;
224 next <= ldacx;
225 end
226
227 ldacx: begin
228 read_en <= 4'd0;
229 inc_en <= 5'b00000 ;
230 write_en <= 16'b0000000000000000 ;
231 rst_en <= 16'b0000000000000000 ;
232 alu_op <= 2'd0;
233 next <= ldac2;
234 end
235
236 ldac2: begin
237 read_en <= 4'd1;
238 inc_en <= 5'b00000 ;
239 write_en <= 16'b0010000000000000 ;
240 rst_en <= 16'b0000000000000000 ;
241 alu_op <= 2'd0;
242 next <= ldacz;
243 end
244
245 ldacz: begin
246 read_en <= 4'd0;
247 inc_en <= 5'b00000 ;
248 write_en <= 16'b0000000000000000 ;
249 rst_en <= 16'b0000000000000000 ;
250 alu_op <= 2'd0;
```

```
next <= ldac3;
end

ldac3: begin
read_en <= 4'd10;
inc_en <= 5'b00000 ;
write_en <= 16'b0100000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= ldacy;
end

ldacy: begin
read_en <= 4'd0;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= ldac4;
end

ldac4: begin
read_en <= 4'd1;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000010000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= fetch1;
end

////////////////////////////////////////////////////////
loadac1: begin
read_en <= 4'd2;
inc_en <= 5'b00000 ;
write_en <= 16'b0010000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= loadac2;
end
//step1
loadac2 : begin
read_en <= 4'd10;
inc_en <= 5'b00000 ;
write_en <= 16'b0100000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd1;
next <= loadacx1;
end

loadacx1: begin
read_en <= 4'd0;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd1;
next <= loadacx2;
```

```
end


//step2
loadacx2 : begin
read_en <= 4'd10;
inc_en <= 5'b00000 ;
write_en <= 16'b0100000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd2;
next <= loadacx3;
end

loadacx3: begin
read_en <= 4'd0;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd2;
next <= loadacx4;
end

//step3
loadacx4 : begin
read_en <= 4'd10;
inc_en <= 5'b00000 ;
write_en <= 16'b0100000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd3;
next <= loadacx5;
end

loadacx5: begin
read_en <= 4'd0;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd3;
next <= loadacx6;
end

//step4
loadacx6 : begin
read_en <= 4'd10;
inc_en <= 5'b00000 ;
write_en <= 16'b0100000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd4;
next <= loadacx7;
end

loadacx7: begin
read_en <= 4'd0;
```

```
367 inc_en <= 5'b00000 ;
368 write_en <= 16'b0000000000000000 ;
369 rst_en <= 16'b0000000000000000 ;
370 alu_op <= 2'd0;
371 memory_mode <= 4'd4;
372 next <= loadacx8;
373 end
374
375 loadacx8: begin
376 read_en <= 4'd10;
377 inc_en <= 5'b00000 ;
378 write_en <= 16'b0100000000000000 ;
379 rst_en <= 16'b0000000000000000 ;
380 alu_op <= 2'd0;
381 memory_mode <= 4'd4;
382 next <= loadacx9;
383 end
384
385 loadacx9: begin
386 read_en <= 4'd0;
387 inc_en <= 5'b00000 ;
388 write_en <= 16'b0000000000000000 ;
389 rst_en <= 16'b0000000000000000 ;
390 alu_op <= 2'd0;
391 memory_mode <= 4'd4;
392 next <= loadac3;
393 end
394
395 loadac3 : begin
396 read_en <= 4'd1;
397 inc_en <= 5'b00000 ;
398 write_en <= 16'b0000000000010000 ;
399 rst_en <= 16'b0000000000000000 ;
400 alu_op <= 2'd0;
401 next <= fetch1;
402 end
403
404
405 ////////////////////////////////////////////////////////////////////
406 stac1: begin
407 read_en <= 4'd9;
408 inc_en <= 5'b00000 ;
409 write_en <= 16'b0100000000000010 ;
410 rst_en <= 16'b0000000000000000 ;
411 alu_op <= 2'd0;
412 next <= stacy;
413 end
414
415 stacy:begin
416 read_en <= 4'd1;
417 inc_en <= 5'b00000 ;
418 write_en <= 16'b0000000000000000 ;
419 rst_en <= 16'b0000000000000000 ;
420 alu_op <= 2'd0;
421 memory_mode <= 4'd5;
422 next <= stac2;
423 end
424
```

```
425 stac2: begin
426 read_en <= 4'd1;
427 inc_en <= 5'b00000 ;
428 write_en <= 16'b0000000000001000 ;
429 rst_en <= 16'b0000000000000000 ;
430 alu_op <= 2'd0;
431 memory_mode <= 4'd5;
432 next <= stacx1;
433 end
434
435 stacx1: begin
436 read_en <= 4'd1;
437 inc_en <= 5'b00000 ;
438 write_en <= 16'b0000000000000000 ;
439 rst_en <= 16'b0000000000000000 ;
440 alu_op <= 2'd0;
441 memory_mode <= 4'd5;
442 next <= stacx2;
443 end
444 ///core2
445 stacx2: begin
446 read_en <= 4'd1;
447 inc_en <= 5'b00000 ;
448 write_en <= 16'b0000000000000000 ;
449 rst_en <= 16'b0000000000000000 ;
450 alu_op <= 2'd0;
451 memory_mode <= 4'd6;
452 next <= stacx3;
453 end
454
455 stacx3: begin
456 read_en <= 4'd1;
457 inc_en <= 5'b00000 ;
458 write_en <= 16'b0000000000001000 ;
459 rst_en <= 16'b0000000000000000 ;
460 alu_op <= 2'd0;
461 memory_mode <= 4'd6;
462 next <= stacx4;
463 end
464
465 stacx4: begin
466 read_en <= 4'd1;
467 inc_en <= 5'b00000 ;
468 write_en <= 16'b0000000000000000 ;
469 rst_en <= 16'b0000000000000000 ;
470 alu_op <= 2'd0;
471 memory_mode <= 4'd6;
472 next <= stacx5;
473 end
474 ////core3
475 stacx5: begin
476 read_en <= 4'd1;
477 inc_en <= 5'b00000 ;
478 write_en <= 16'b0000000000000000 ;
479 rst_en <= 16'b0000000000000000 ;
480 alu_op <= 2'd0;
481 memory_mode <= 4'd7;
482 next <= stacx6;
```

```
end

stacx6: begin
read_en <= 4'd1;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000001000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd7;
next <= stacx7;
end

stacx7: begin
read_en <= 4'd1;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd7;
next <= stacx8;
end
////core4
stacx8: begin
read_en <= 4'd1;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd8;
next <= stacx9;
end

stacx9: begin
read_en <= 4'd1;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000001000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd8;
next <= stacx10;
end

stacx10: begin
read_en <= 4'd1;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
memory_mode <= 4'd8;
next <= fetch1;
end



////////////////////////////////////
mvacr1: begin
read_en <= 4'd9;
inc_en <= 5'b00000 ;
```

```
write_en <= 16'b0000100000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= fetch1;
end

mvaca1: begin
read_en <= 4'd9;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000001000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= fetch1;
end

mvacsum1: begin
read_en <= 4'd9;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000100000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= fetch1;
end

mvaci1: begin
read_en <= 4'd9;
inc_en <= 5'b00000 ;
write_en <= 16'b0000001000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= fetch1;
end

mvsumac1: begin
read_en <= 4'd8;
inc_en <= 5'b00000 ;
write_en <= 16'b0000000000010000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= fetch1;
end


mvir1: begin
read_en <= 4'd4;
inc_en <= 5'b00000 ;
write_en <= 16'b0000100000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
next <= fetch1;
end

mvkr1 : begin
read_en <= 4'd6;
inc_en <= 5'b00000 ;
write_en <= 16'b0000100000000000 ;
rst_en <= 16'b0000000000000000 ;
alu_op <= 2'd0;
```

```
599 next <= fetch1;
600 end
601
602 mvjr1: begin
603 read_en <= 4'd5;
604 inc_en <= 5'b00000 ;
605 write_en <= 16'b0000100000000000 ;
606 rst_en <= 16'b0000000000000000 ;
607 alu_op <= 2'd0;
608 next <= fetch1;
609 end
610
611 mvidr1: begin
612 read_en <= 4'd3;
613 inc_en <= 5'b00000 ;
614 write_en <= 16'b0000100000000000 ;
615 rst_en <= 16'b0000000000000000 ;
616 alu_op <= 2'd0;
617 next <= fetch1;
618 end
619
620 mulr1: begin
621 read_en <= 4'd2;
622 inc_en <= 5'b00000 ;
623 write_en <= 16'b0000000000000001 ;
624 rst_en <= 16'b0000000000000000 ;
625 alu_op <= 2'd3;
626 next <= fetch1;
627 end
628
629 mula1: begin
630 read_en <= 4'd7;
631 inc_en <= 5'b00000 ;
632 write_en <= 16'b0000000000000001 ;
633 rst_en <= 16'b0000000000000000 ;
634 alu_op <= 2'd3;
635 next <= fetch1;
636 end
637
638 addr1: begin
639 read_en <= 4'd2;
640 inc_en <= 5'b00000 ;
641 write_en <= 16'b0000000000000001 ;
642 rst_en <= 16'b0000000000000000 ;
643 alu_op <= 2'd1;
644 next <= fetch1;
645 end
646
647 addsum1: begin
648 read_en <= 4'd8;
649 inc_en <= 5'b00000 ;
650 write_en <= 16'b0000000000000001 ;
651 rst_en <= 16'b0000000000000000 ;
652 alu_op <= 2'd1;
653 next <= fetch1;
654 end
655
656 comp1: begin
```

```
657 read_en <= 4'd2;
658 inc_en <= 5'b00000 ;
659 write_en <= 16'b0000000000000000 ;
660 rst_en <= 16'b0000000000000000 ;
661 alu_op <= 2'd2;
662 next <= fetch1;
663 end
664
665 inck1: begin
666 read_en <= 4'd0;
667 inc_en <= 5'b00001 ;
668 write_en <= 16'b0000000000000000 ;
669 rst_en <= 16'b0000000000000000 ;
670 alu_op <= 2'd0;
671 next <= fetch1;
672 end
673
674 incj1: begin
675 read_en <= 4'd0;
676 inc_en <= 5'b00010 ;
677 write_en <= 16'b0000000000000000 ;
678 rst_en <= 16'b0000000000000000 ;
679 alu_op <= 2'd0;
680 next <= fetch1;
681 end
682
683 incr1: begin
684 read_en <= 4'd0;
685 inc_en <= 5'b00100 ;
686 write_en <= 16'b0000000000000000 ;
687 rst_en <= 16'b0000000000000000 ;
688 alu_op <= 2'd0;
689 next <= fetch1;
690 end
691
692 jmpzy1: begin
693 read_en <= 4'd11;
694 inc_en <= 5'b00000 ;
695 write_en <= 16'b0100000000000000 ;
696 rst_en <= 16'b0000000000000000 ;
697 alu_op <= 2'd0;
698 next <= jmpzyx;
699 end
700
701 jmpzyx: begin
702 read_en <= 4'd0;
703 inc_en <= 5'b00000 ;
704 write_en <= 16'b0000000000000000 ;
705 rst_en <= 16'b0000000000000000 ;
706 alu_op <= 2'd0;
707 next <= jmpzy2;
708 end
709
710 jmpzy2: begin
711 read_en <= 4'd1;
712 inc_en <= 5'b00000 ;
713 write_en <= 16'b1000000000000000 ;
714 rst_en <= 16'b0000000000000000 ;
```

```
715 alu_op <= 2'd0;
716 next <= fetch1;
717 end
718
719
720 jmpzn1: begin
721 read_en <= 4'd0;
722 inc_en <= 5'b10000 ;
723 write_en <= 16'b0000000000000000 ;
724 rst_en <= 16'b0000000000000000 ;
725 alu_op <= 2'd0;
726 next <= fetch1;
727 end
728
729
730
731 jump1: begin
732 read_en <= 4'd11;
733 inc_en <= 5'b00000 ;
734 write_en <= 16'b0100000000000000 ;
735 rst_en <= 16'b0000000000000000 ;
736 alu_op <= 2'd0;
737 next <= jumpx;
738 end
739
740 jumpx: begin
741 read_en <= 4'd0;
742 inc_en <= 5'b00000 ;
743 write_en <= 16'b0000000000000000 ;
744 rst_en <= 16'b0000000000000000 ;
745 alu_op <= 2'd0;
746 next <= jump2;
747 end
748
749 jump2: begin
750 read_en <= 4'd1;
751 inc_en <= 5'b00000 ;
752 write_en <= 16'b1000000000000000 ;
753 rst_en <= 16'b0000000000000000 ;
754 alu_op <= 2'd0;
755 next <= fetch1;
756 end
757
758
759 rstall1: begin
760 read_en <= 4'd0;
761 inc_en <= 5'b00000 ;
762 write_en <= 16'b0000000000000000 ;
763 rst_en <= 16'b0110111111100000 ;
764 alu_op <= 2'd0;
765 next <= fetch1;
766 end
767
768 rsti1: begin
769 read_en <= 4'd3;
770 inc_en <= 5'b00000 ;
771 write_en <= 16'b0000001000000000 ;
772 rst_en <= 16'b0000000000000000 ;
```

```
773 alu_op <= 2'd0;
774 next <= fetch1;
775 end
776
777 rstj1: begin
778 read_en <= 4'd0;
779 inc_en <= 5'b00000 ;
780 write_en <= 16'b0000000000000000 ;
781 rst_en <= 16'b0000000100000000 ;
782 alu_op <= 2'd0;
783 next <= fetch1;
784 end
785
786 rstk1: begin
787 read_en <= 4'd0;
788 inc_en <= 5'b00000 ;
789 write_en <= 16'b0000000000000000 ;
790 rst_en <= 16'b0000000010000000 ;
791 alu_op <= 2'd0;
792 next <= fetch1;
793 end
794
795 rstsum1: begin
796 read_en <= 4'd0;
797 inc_en <= 5'b00000 ;
798 write_en <= 16'b0000000000000000 ;
799 rst_en <= 16'b0000000000100000 ;
800 alu_op <= 2'd0;
801 next <= fetch1;
802 end
803
804 nop1: begin
805 read_en <= 4'd0;
806 inc_en <= 5'b00000 ;
807 write_en <= 16'b0000000000000000 ;
808 rst_en <= 16'b0000000000000000 ;
809 alu_op <= 2'd0;
810 next <= fetch1;
811 end
812
813 endop: begin
814 read_en <= 4'd0;
815 inc_en <= 5'b00000 ;
816 write_en <= 16'b0000000000000000 ;
817 rst_en <= 16'b0000000000000000 ;
818 alu_op <= 2'd0;
819 next <= endop;
820 end
821
822 default: begin
823 read_en <= 4'd0;
824 inc_en <= 5'b00000 ;
825 write_en <= 16'b0000000000000000 ;
826 rst_en <= 16'b0000000000000000 ;
827 alu_op <= 2'd0;
828 next <= fetch1;
829 end
830 endcase
```

```
831
832 endmodule
```

A.5 ALU

```
1  module alu(input clock,
2  input [15:0] in1,
3  input [15:0] in2,
4  input [1:0] alu_op,
5  output reg [15:0] out,
6  output reg z);
7
8
9  always @(*)
10          begin
11          case(alu_op)
12          2'd1: out <= in1 + in2;
13          2'd2: begin
14                              if(in1<in2)
15                              begin
16                              z <= 1;
17                              end
18                              else
19                              begin
20                              z <= 0;
21                              end
22                              end
23          2'd3: out <= in1*in2;
24
25          endcase
26
27          end
28
29 endmodule
```

A.6 Register

```
1  // 16 bit register with read,write,inc,rst
2  module register_16(clk,write_en,rst_en,inc_en,data_in,data_out);
3  input clk;
4  input write_en;
5  input rst_en;
6  input inc_en;
7  input [15:0] data_in;
8  output reg [15:0] data_out;
9
10 always @ (posedge clk)
11 begin
12          if (write_en == 1)
13                  data_out <= data_in;
14          if (rst_en == 1)
15                  data_out <= 16'd0;
16          if (inc_en == 1)
17                  data_out <= data_out + 16'd1;
```

```
end
endmodule
```

A.7 Data Register

```
module DR(clk,write_en,data_in,data_out);    //DR register
input clk;
input write_en;
input [15:0] data_in;
output reg [15:0] data_out;

always @ (posedge clk)
begin
        if (write_en == 1)
                data_out <= data_in;


end
endmodule
```

A.8 Address Register

```
module AR(clk,write_en1,write_en2,rst_en,inc_en,data_in1,data_in2,data_out);
      //normal register with read,write,inc,rst
input clk;
input write_en1;
input write_en2;
input rst_en;
input inc_en;
input [15:0] data_in1;
input [15:0] data_in2;
output reg [15:0] data_out;

always @ (posedge clk)
begin
        if (write_en1 == 1)
                data_out <= data_in1;
        if (write_en2 == 1)
                data_out <= data_in2;
        if (rst_en == 1)
                data_out <= 16'd0;
        if (inc_en == 1)
                data_out <= data_out + 16'd1;
end
endmodule
```

A.9 Accumulator

```
module AC(input clk,
        input write_en,
        input rst_en,
        input aluout_en,
```

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

always@ (posedge clk)
begin
        if (write_en == 1)
                data_out <= data_in;
        if (aluout_en == 1)
                data_out <= alu_out;
        if (rst_en == 1)
                data_out <= 16'd0;
end
endmodule
```

A.10 Bus

```
module bus(
        input clock,
        input [3:0] read_en,
        input [15:0] id,
        input [15:0] i,
        input [15:0] j,
        input [15:0] k,
        input [15:0] a,
        input [15:0] dr,
        input [15:0] ac,
        input [15:0] r,
        input [15:0] sum,
        input [15:0] DRAM,
        input [15:0] IRAM,
        output reg[15:0] busout ) ;


always @(id or i or j or k or a or dr or ac or r or sum or read_en)
begin
case(read_en)
4'd1: busout <= dr ;
4'd2: busout <= r;
4'd3: busout <= id ;
4'd4: busout <= i ;
4'd5: busout <= j ;
4'd6: busout <= k ;
4'd7: busout <= a ;
4'd8: busout <= sum;
4'd9: busout <= ac;
4'd10: busout <= DRAM ;
4'd11: busout <= IRAM ;
default: busout <= 16'd0;
endcase
end
endmodule
endmodule
```

A.11 Memory Controller

```
module memory_controller(
        input clock,
        input [15:0] ar_out1,
        input [15:0] ar_out2,
        input [15:0] ar_out3,
        input [15:0] ar_out4,

        input [15:0] bus_out1,
        input [15:0] bus_out2,
        input [15:0] bus_out3,
        input [15:0] bus_out4,

        input [15:0] dram_out,
        input [3:0] mode,

        input end_process2,
        input end_process3,
        input end_process4,

        output reg[15:0] ar_out,
        output reg[15:0] bus_out,

        output reg[15:0] dram_out1,
        output reg[15:0] dram_out2,
        output reg[15:0] dram_out3,
        output reg[15:0] dram_out4);


always @(posedge clock)
begin
        if (mode == 4'd1)
        begin
                ar_out <= ar_out1;
                dram_out1 <= dram_out;
        end
        else if (mode == 4'd2)
        begin
                ar_out <= ar_out2;
                dram_out2 <= dram_out;
        end
        else if (mode == 4'd3)
        begin
                ar_out <= ar_out3;
                dram_out3 <= dram_out;
        end
        else if (mode == 4'd4)
        begin
                ar_out <= ar_out4;
                dram_out4 <= dram_out;
        end

        else if (mode == 4'd5)
        begin
                ar_out <= ar_out1;
                bus_out <= bus_out1;
        end
```

```
        else if ((mode == 4'd6) & (end_process2 != 1))
        begin
                ar_out <= ar_out2;
                bus_out <= bus_out2;
        end
        else if ((mode == 4'd7) & (end_process3 != 1))
        begin
                ar_out <= ar_out3;
                bus_out <= bus_out3;
        end
        else if ((mode == 4'd8) & (end_process4 != 1))
        begin
                ar_out <= ar_out4;
                bus_out <= bus_out4;
        end
        else if (mode == 4'd0)
        begin
                ar_out <= ar_out1;
                dram_out1 <= dram_out;
                dram_out2 <= dram_out;
                dram_out3 <= dram_out;
                dram_out4 <= dram_out;
        end

end

endmodule
```

A.12 Clock Divider

```
module clock_divider(inclk,ena,clk);


 parameter maxcount=32'd10;// input 50MHz clock into output 5MHz clk

 input inclk;
 input ena;
 output reg clk=1;

 reg [31:0] count=32'd0;

 always @ (posedge inclk )
        begin
                if (ena)
                begin
                        if (count==maxcount)
                        begin
                        clk=~clk;
                        count=32'd0;
                        end
                        else
                        begin
                        count=count+1;
                        end
                end
                else
```

```
                begin
                clk=0;
                end
        end

 endmodule
```

A.13 CoreID

```
module CoreID(clk,rst_en,data_in,data_out);   //Core ID register

input clk;
input rst_en;
input[15:0] data_in;
output reg [15:0] data_out;

always @ (posedge clk)
begin
        if (rst_en == 1)
                data_out <= data_in;
end
endmodule
```

A.14 End Core

```
module end_core(
        input clock,
        input[15:0] pc_out1,
        input[15:0] pc_out2,
        input[15:0] pc_out3,
        input[15:0] pc_out4,
        output reg[1:0] end_core1,
        output reg[1:0] end_core2,
        output reg[1:0] end_core3,
        output reg[1:0] end_core4);

always @(posedge clock)
        begin
        if (pc_out1 != pc_out2)
                begin
                end_core2 <= 2'b10;
                end_core3 <= 2'b10;
                end_core4 <= 2'b10;
                end
        else if (pc_out1 != pc_out3)
                begin
                end_core3 <= 2'b10;
                end_core4 <= 2'b10;
                end
        else if (pc_out1 != pc_out4) end_core4 <= 2'b10;
        else
                begin
                        end_core1 <= 2'b00;
                        end_core2 <= 2'b00;
```

```
                        end_core3 <= 2'b00;
                        end_core4 <= 2'b00;
                end
        end

endmodule
```

A.15 ALU Test Bench

```
`timescale 1ns/1ps

module ALU_tb();

localparam CLK_PERIOD = 2;

reg clock;
reg [15:0] in1;
reg [15:0] in2;
reg [1:0] alu_op;
wire [15:0] out;
wire z;

alu_test1 alu(.clock(clock),.in1(in1),.in2(in2),.alu_op(alu_op),.out(out),.z
    (z));

initial begin
        clock = 1'b0;
        forever begin
                #(CLK_PERIOD/2);
                clock = ~clock;
        end
end

initial begin
        #10
        in1 = 16'd3;
        in2 = 16'd2;
        alu_op = 2'd1;

        #10
        in1 = 16'd3;
        in2 = 16'd3;
        alu_op = 2'd1;

        #10
        in1 = 16'd4;
        in2 = 16'd2;
        alu_op = 2'd1;

        #10
        in1 = 16'd3;
        in2 = 16'd2;
        alu_op = 2'd2;

        #10
        in1 = 16'd3;
```

```
        in2 = 16'd3;
        alu_op = 2'd2;

        #10
        in1 = 16'd4;
        in2 = 16'd6;
        alu_op = 2'd2;

        #10
        in1 = 16'd3;
        in2 = 16'd2;
        alu_op = 2'd3;

        #10
        in1 = 16'd3;
        in2 = 16'd3;
        alu_op = 2'd3;

        #10
        in1 = 16'd4;
        in2 = 16'd2;
        alu_op = 2'd3;

end

endmodule
```

A.16 Address Register Test Bench

```
`timescale 1ns/1ps

module AR_tb();

localparam CLK_PERIOD = 2;

reg clk,write_en1,write_en2,rst_en,inc_en;
reg [15:0] data_in1;
reg [15:0] data_in2;
wire [15:0] data_out;

AR AR(.clk(clk),.write_en1(write_en1),.write_en2(write_en2),
.rst_en(rst_en),.inc_en(inc_en),.data_in1(data_in1),
.data_in2(data_in2),.data_out(data_out));

initial begin
        clk = 1'b0;
        forever begin
                #(CLK_PERIOD/2);
                clk = ~clk;
        end
end

initial begin
        #10
        write_en1 = 1;
        write_en2 = 0;
```

```
        rst_en = 0;
        inc_en = 0;
        data_in1 = 16'd100;
        data_in2 = 16'd127;

        #10
        write_en1 = 0;
        write_en2 = 1;
        rst_en = 0;
        inc_en = 0;
        data_in1 = 16'd15;
        data_in2 = 16'd10;

        #10
        write_en1 = 0;
        write_en2 = 0;
        rst_en = 0;
        inc_en = 1;
        data_in1 = 16'd1;
        data_in2 = 16'd1;

        #10
        write_en1 = 0;
        write_en2 = 0;
        rst_en = 1;
        inc_en = 0;
        data_in1 = 16'd10;
        data_in2 = 16'd10;

end

endmodule
```

A.17 Bus Test Bench

```
`timescale 1ns/1ps

module bus_tb();

localparam CLK_PERIOD = 2;

reg clock;
reg [3:0] read_en;
reg [15:0] id;
reg [15:0] i;
reg [15:0] j;
reg [15:0] k;
reg [15:0] a;
reg [15:0] dr;
reg [15:0] ac;
reg [15:0] r;
reg [15:0] sum;
reg [15:0] DRAM;
reg [15:0] IRAM;
wire[15:0] busout;

```

```
22
23 bus1 bus(.clock(clock),.read_en(read_en),.id(id),.i(i),.j(j),.k(k),.a(a),.dr
       (dr),
24 .ac(ac),.r(r),.sum(sum),.DRAM(DRAM),.IRAM(IRAM),.busout(busout));
25
26 initial begin
27         clock = 1'b0;
28         forever begin
29                 #(CLK_PERIOD/2);
30                 clock = ~clock;
31         end
32 end
33
34 initial begin
35         #10
36         read_en= 4'd1;
37         id= 16'd1;
38         i= 16'd02;
39         j= 16'd03;
40         k= 16'd04;
41         a= 16'd05;
42
43         dr= 16'd07;
44         ac= 16'd08;
45         r= 16'd09;
46         sum= 16'd10;
47         DRAM= 16'd011;
48         IRAM= 16'd012;
49
50         #10
51         read_en= 4'd2;
52         id= 16'd1;
53         i= 16'd02;
54         j= 16'd03;
55         k= 16'd04;
56         a= 16'd05;
57
58         dr= 16'd07;
59         ac= 16'd08;
60         r= 16'd09;
61         sum= 16'd10;
62         DRAM= 16'd011;
63         IRAM= 16'd012;
64
65         #10
66         read_en= 4'd3;
67         id= 16'd1;
68         i= 16'd02;
69         j= 16'd03;
70         k= 16'd04;
71         a= 16'd05;
72
73         dr= 16'd07;
74         ac= 16'd08;
75         r= 16'd09;
76         sum= 16'd10;
77         DRAM= 16'd011;
78         IRAM= 16'd012;
```

```
79
80          #10
81          read_en= 4'd4;
82          id= 16'd1;
83          i= 16'd02;
84          j= 16'd03;
85          k= 16'd04;
86          a= 16'd05;
87
88          dr= 16'd07;
89          ac= 16'd08;
90          r= 16'd09;
91          sum= 16'd10;
92          DRAM= 16'd011;
93          IRAM= 16'd012;
94
95          #10
96          read_en= 4'd5;
97          id= 16'd1;
98          i= 16'd02;
99          j= 16'd03;
100         k= 16'd04;
101         a= 16'd05;
102
103         dr= 16'd07;
104         ac= 16'd08;
105         r= 16'd09;
106         sum= 16'd10;
107         DRAM= 16'd011;
108         IRAM= 16'd012;
109
110         #10
111         read_en= 4'd6;
112         id= 16'd1;
113         i= 16'd02;
114         j= 16'd03;
115         k= 16'd04;
116         a= 16'd05;
117
118         dr= 16'd07;
119         ac= 16'd08;
120         r= 16'd09;
121         sum= 16'd10;
122         DRAM= 16'd011;
123         IRAM= 16'd012;
124
125         #10
126         read_en= 4'd7;
127         id= 16'd1;
128         i= 16'd02;
129         j= 16'd03;
130         k= 16'd04;
131         a= 16'd05;
132
133         dr= 16'd07;
134         ac= 16'd08;
135         r= 16'd09;
136         sum= 16'd10;
```

```
137         DRAM= 16'd011;
138         IRAM= 16'd012;
139 
140         #10
141         read_en= 4'd8;
142         id= 16'd1;
143         i= 16'd02;
144         j= 16'd03;
145         k= 16'd04;
146         a= 16'd05;
147 
148         dr= 16'd07;
149         ac= 16'd08;
150         r= 16'd09;
151         sum= 16'd10;
152         DRAM= 16'd011;
153         IRAM= 16'd012;
154 
155         #10
156         read_en= 4'd9;
157         id= 16'd1;
158         i= 16'd02;
159         j= 16'd03;
160         k= 16'd04;
161         a= 16'd05;
162 
163         dr= 16'd07;
164         ac= 16'd08;
165         r= 16'd09;
166         sum= 16'd10;
167         DRAM= 16'd011;
168         IRAM= 16'd012;
169 
170         #10
171         read_en= 4'd10;
172         id= 16'd1;
173         i= 16'd02;
174         j= 16'd03;
175         k= 16'd04;
176         a= 16'd05;
177 
178         dr= 16'd07;
179         ac= 16'd08;
180         r= 16'd09;
181         sum= 16'd10;
182         DRAM= 16'd011;
183         IRAM= 16'd012;
184 
185         #10
186         read_en= 4'd11;
187         id= 16'd1;
188         i= 16'd02;
189         j= 16'd03;
190         k= 16'd04;
191         a= 16'd05;
192 
193         dr= 16'd07;
194         ac= 16'd08;
```

```
        r= 16'd09;
        sum= 16'd10;
        DRAM= 16'd011;
        IRAM= 16'd012;



end

endmodule
```

A.18 Register Test Bench

```
`timescale 1ns/1ps

module register_tb();

localparam CLK_PERIOD = 20;

reg clk;
reg write_en;
reg rst_en;
reg inc_en;
reg [15:0] data_in;
wire [15:0] data_out;

Register1 register(.clk(clk),.write_en(write_en),.rst_en(rst_en),
.inc_en(inc_en),.data_in(data_in),.data_out(data_out));

initial begin
        clk = 1'b0;
        forever begin
                #(CLK_PERIOD/2);
                clk = ~clk;
        end
end

initial begin
        #10
        write_en = 1;
        data_in = 16'd10;

        #10
        write_en = 0;
        data_in = 16'd15;

        #10
        inc_en = 1;

        #10
        inc_en = 0;
        rst_en = 1;

end

endmodule
```

A.19 Top Module Test Bench

```
`timescale 1ns/1ps

module top_module_tb();

localparam CLK_PERIOD = 20;

reg [1:0] status;
reg clk;
reg clockm;
reg ena;
wire z;
wire [15:0]write_en;
wire [3:0] read_en;
wire end_process1;
wire end_process2;
wire end_process3;
wire end_process4;
wire [31:0]clk_counter;

top_module top_module (
        .status(status),
        .clk(clk),
        .clockm(clockm),
        .ena(ena),
        .z(z),
        .write_en(write_en),
        .read_en(read_en),
        .end_process1(end_process1),
        .end_process2(end_process2),
        .end_process3(end_process3),
        .end_process4(end_process4),
        .clk_counter(clk_counter));

initial begin
        clk = 1'b0;
        clockm = 1'b0;
        ena <= 1'b1;
        status <= 2'b01;
        forever begin
                #(CLK_PERIOD/2);
                clk = ~clk;
                clockm = ~clockm;
        end
end


endmodule
```

A.20 Assembler code

```
assemblyf = open("C:\FPGA_project\Verilog16_1\Verilog16\multicorefinal.s", "r")
iram = open("C:\FPGA_project\Verilog16_1\Verilog16\iram.mif","w")
a = True
ISA = {
    'LDAC' : '0000000000000001',
```

```
    'LDACR' : '0000000000000010',
    'STAC' : '0000000000000011',
    'MVACR' : '0000000000000100',
    'MVACA' : '0000000000000101',
    'MVACSUM' : '0000000000000110',
    'MVACI' : '0000000000000111',
    'MVSUMAC' : '0000000000001000',
    'MVIR' : '0000000000001001',
    'MVKR' : '0000000000001010',
    'MVJR' : '0000000000001011',
    'MVIDR' : '0000000000001100',
    'MULR' : '0000000000001101',
    'MULA' : '0000000000001110',
    'ADDR' : '0000000000001111',
    'ADDSUM' : '0000000000010000',
    'COMP' : '0000000000010001',
    'INCK' : '0000000000010010',
    'INCJ' : '0000000000010011',
    'INCR' : '0000000000010100',
    'JMPZ' : '0000000000010101',
    'JUMP' : '0000000000010110',
    'RSTALL' : '0000000000010111',
    'RSTI' : '0000000000011000',
    'RSTJ' : '0000000000011001',
    'RSTK' : '0000000000011010',
    'RSTSUM' : '0000000000011011',
    'NOP' : '0000000000011100'
}
addr = 0
lines = assemblyf.readlines()
iram.write("WIDTH=16;\nDEPTH=256;\n\nADDRESS_RADIX=HEX;\nDATA_RADIX=BIN;\n\nCONTENT
    BEGIN")
iram.write("\n\t")
while addr<len(lines):
    line = "".join(lines[addr].split())
    if line == "ENDOP":
        hexaddr = format(addr, '03x')
        iram.write(str('['+hexaddr.upper()+'..0FF]  :   0000000000000000;'))
        break
    elif line == "":
        continue
    else:
        opcode = ISA[line]
        hexaddr = format(addr, '03x')
        iram.write(str(hexaddr.upper() + '  :   ' +opcode + ';'))
        iram.write("\n\t")
        if  line in ["LDAC" , "JUMP" , "JMPZ"]:
            addr += 1
            hexaddr = format(addr, '03x')
            memaddr = "".join(lines[addr][4:].split())
            memaddr = format(int(memaddr),'016b')
            iram.write(str(hexaddr.upper() + '  :   ' +memaddr + ';'))
            iram.write("\n\t")
        addr += 1

iram.write("\nEND;")

assemblyf.close()
iram.close()
```

A.21 Multicore Python Simulation

```
NoC=2

Memory = [NoC,
          4,
          4,
          4,
          0,
          0,
          0,
          1,

          1,2,3,4,
          5,6,7,8,
          9,10,11,12,
          13,14,15,16,

          17,18,19,20,
          21,22,23,24,
          25,26,27,28,
          29,30,31,32,

          0,0,0,0,
          0,0,0,0,
          0,0,0,0,
          0,0,0,0,
          0,0,0,0,
          0,0,0,0,
          0,0,0,0,
          0,0,0,0
        ]

n = Memory[1]
m = Memory[2]
p = Memory[3]
s = Memory[7]
Memory[4],Memory[5],Memory[6] = 8,(8+n*m),(8+n*m+m*p)

def singlecore(CoreID,M=Memory):
    ID = CoreID
    AC = M[0]
    R = ID
    R += 1
    if AC<R:
        z = 1
    else:
        z = 0
    if z==1:
        return None
    I = ID
    J = 0

    while True:      #loop
        R = I
        AC = M[1]
        R+=1
        if AC<R:
            z = 1
        else:
            z = 0
        if z == 1:
            return None
```

```
        K = 0

        while True: #loop1
            R = K
            AC = M[3]
            R+=1
            #AC = AC-R
            if AC<R:
                z=1
            else:
                z=0
            if z==1:
                break
            SUM = 0

            while True: #loop2
                R = J
                AC = M[2]
                R+=1
                if AC<R:
                    z=1
                else:
                    z=0
                if z == 1:
                    R = I
                    AC = M[3]
                    AC = AC*R
                    R = K
                    AC = AC+R
                    R = AC
                    #AC = M[7]
                    #AC = AC*R
                    #R = AC
                    AC = M[6]
                    AC = AC+R
                    R = AC

                    AC = SUM
                    M[R]=AC

                    K = K+1
                    J = 0
                    break

                R = I
                AC = M[2]
                AC = AC*R
                R = J
                AC += R
                R = AC
                AC = M[4]
                AC += R
                R = AC
                AC = M[R]
                A = AC

                R = J
                AC = M[3]
                AC = AC*R
                R = K
                AC +=R
                R = AC
                AC = M[5]
```

```
                    AC += R
                    R = AC
                    AC = M[R]

                    AC = AC*A
                    AC = AC+SUM
                    SUM = AC

                    J +=1
                    continue
            R = I
            AC = M[0]
            AC +=R
            I = AC

singlecore(0)
singlecore(1)
#singlecore(2)
#singlecore(3)

print(Memory[40:44])
print(Memory[44:48])
print(Memory[48:52])
print(Memory[52:56])
```