Department of Electronic & Telecommunication Engineering

Biomedical engineering

University of Moratuwa

EN 3030: Circuits and Systems Design

FPGA BASED PROCESSOR DESIGN REPORT

Name	Index number
Dhinesh S.	170133B
Jathurshan P.	170248G
Mithunjha A.	170389M
Vinith K.	170654X

This report is submitted in partial fulfillment of the requirements

For the module EN 3030: Circuits and Systems Design.

April 07th, 2021

Abstract

This report contains the details of the design and implementation of the custom processor designed to utilize multiple cores to do matrix multiplication using Field Programmable Gate Array. The processor was initially designed to do the multiplication of square matrices and then was enhanced to the multiplication of rectangular matrices. The design proposed in this report is capable of doing multiplication of the matrices with the number of rows or columns up to 16 and with values of the matrix elements in the range of 0 – 63. The number of cores in the design can be increased up to six, and this limitation is caused due to the decimal to binary limitation in MATLAB, which is an external factor. The ability to vary the number of cores in the multi-core processor is an additional advantage of our design. Comparison between the time taken for matrix multiplication by different numbers of cores and MATLAB was conducted to validate the efficiency of the design. All the necessary details and documents to reproduce this design are attached to this report.

Table of Contents

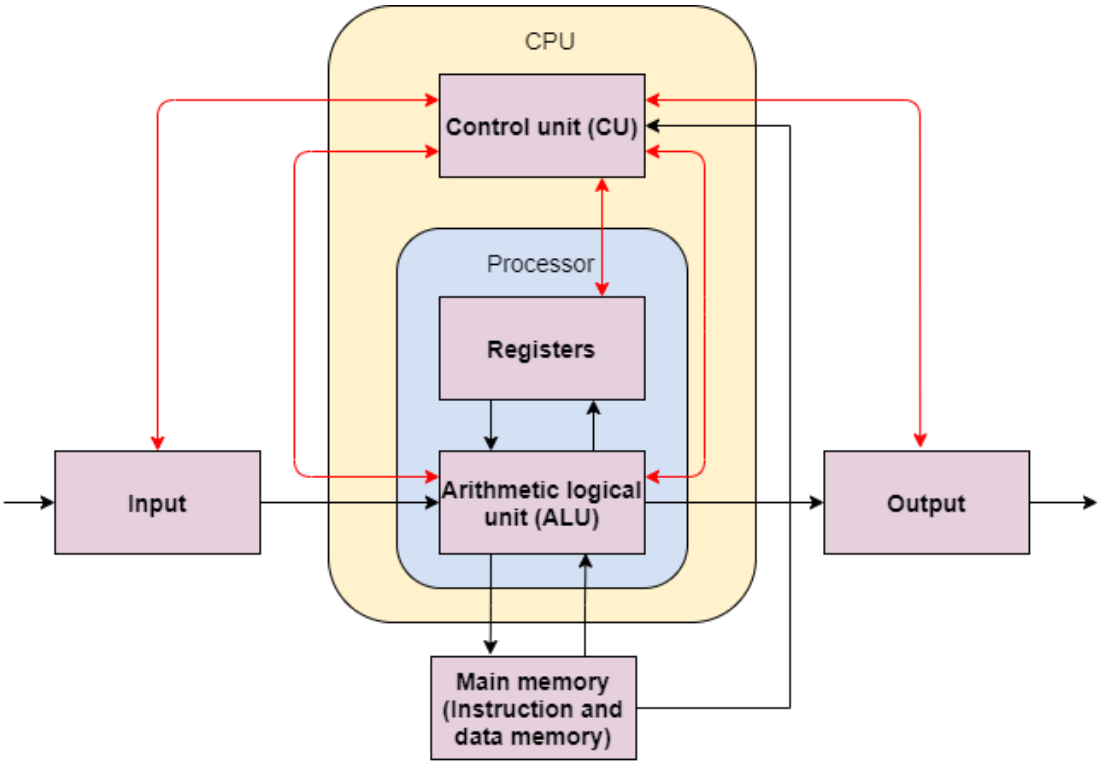
1. Introduction

1.1. Processor Design

The scope of this project is to implement an object specified microprocessor and Central processing unit (CPU) design, to multiply given two matrices using multiple cores. The implementation is done in Verilog hardware descriptive language (Verilog HDL) in Quartus Prime version 18.1 lite edition and the procedure is simulated using ModelSim. This report includes the designs of the microprocessor cores, control unit, memory modules and the overall architecture.

1.2. Central Processing Unit (CPU)

The Central Processing Unit (CPU) is an electronic circuitry that carries out the instructions fed into the computer program (Instruction memory), by performing the basic arithmetic, logical and control functions with the input/output operations as per the instructions provided. The control unit and the processor (registers and ALU) together are generally referred as the CPU. The internal structure of a general CPU with the main memory and input/output circuits is given below.



1.3. Microprocessor

Microprocessor represents the computer processor which comprises of the circuitries of arithmetic, logic and control units to function as the central processing unit. It is a multipurpose,

register based, clock driven digital IC with the capabilities of executing instructions and performing arithmetic and logical operations. Both sequential and combinational logic are present within a microprocessor. Multi core processor consists of more than one processing units, to fetch and execute the instructions efficiently. Running instructions on multiple cores at the same time increases the overall speed of program execution.

1.4. Problem Statement

The objective of the project is to design a multicore microprocessor to multiply given two matrices. The input matrix size can be varied under a given constraint. The matrix size and the elements of the matrix are given as the input and the multiplied matrix is given as the output. The constraints under which the microprocessor is designed are given below.

- Matrix dimension: square matrix with maximum size of 16.
- The input numbers: max of 63(2^6-1).
- Unsigned integers

Note: if signed integers are to be used, the value of the input should be limited from -32 to 31 as the memory allocated to store the input integers are 6 bits and therefore the output maximum size is limited to 16 bits.

The implementation is further upgraded to handle rectangular matrices of maximum length and breath of 16. For this implementation, the core design and the ISA were updated from the square matrix implementation.

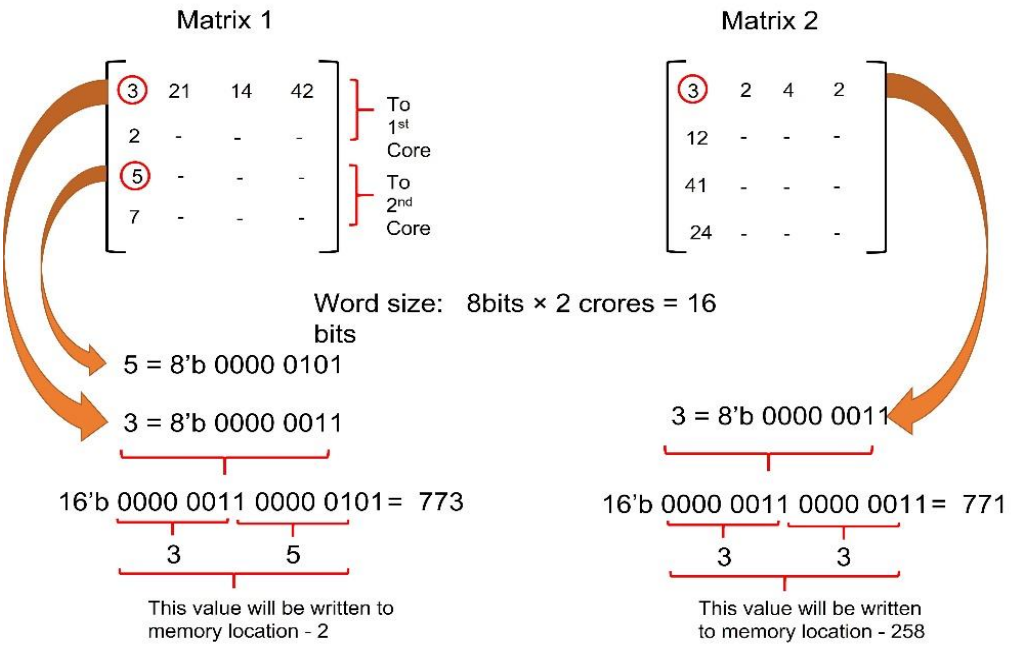
As for the verification of the designed processor, the same matrix multiplication is performed using MATLAB and the output is compared between the Verilog implementation and the MATLAB output. A comparison matrix to validate the results was designed using Python.

1.5. Proposed Solution

Our solution utilizes multicore processors for matrix multiplication, which was initially implemented for multiplication of square matrices and later was enhanced to multiplication of rectangular matrices. Initially the matrix multiplication was achieved on one core processor and then was expanded to multi cores by manipulating the storage of input matrices to the data memory.

Initially fixed memory locations were assigned to store first matrix, second matrix and the output matrix. The data memory was sliced into columns, where each core was assigned to

that certain data memory column. The rows of the first matrix will be divided between cores and will be written in the column of the specific cores. Then all the columns of the second matrix will be written in the column for each core. Let us consider an example get a better insight on our method. Consider multiplication of two 4×4 matrices using 2 cores. The rows and columns in these two matrices will be written to data memory as given below:



After assigning the values of the matrices to the memory, each core in the processor will be assigned to calculate the output of the respective rows in the output matrix. Likewise, the matrix multiplication is done using multiple cores.

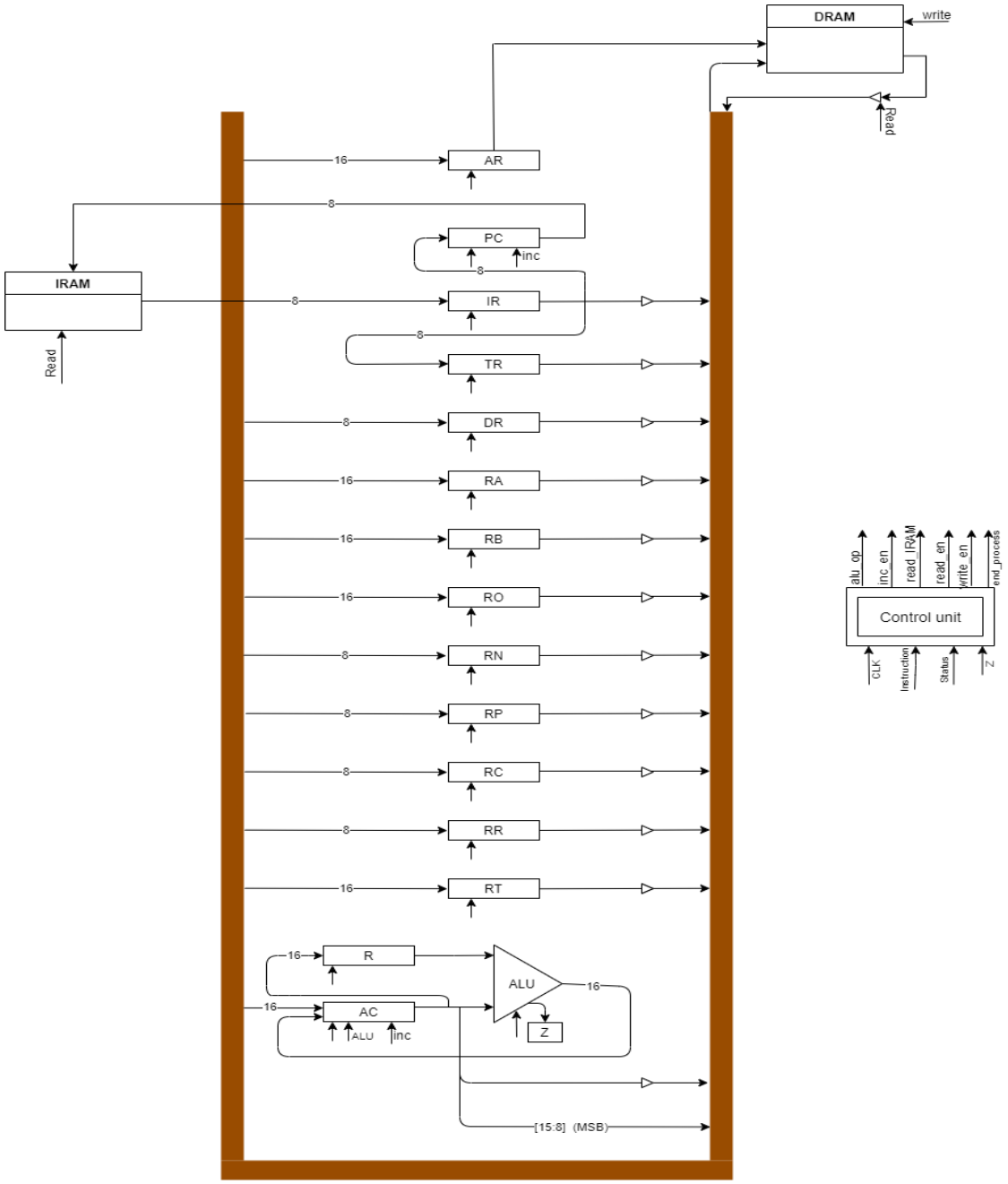
2. Design

2.1. Overview

FPGA, which stands for field programmable gate array consists of a grid of programmable logical blocks that provides the user reconfigurable interconnections to achieve the requirements. Configuration of FPGA is specified using a hardware description language (HDL). In this project, Verilog was used to develop the design and simulation was carried out using ModelSim.

2.2. Instruction Set Architecture

2.2.1. Data Path



2.2.2. Registers

1. Address Register (AR)- 16bits register to store address location of the Data RAM (DRAM)
2. Instruction Register (IR) - 8bits register to store instruction (assembly code)
3. Data Register (DR) - 8bits register to store data from DRAM
4. Temporary Register (TR) - 8bits register to handle 16bits memory addresses
5. Program Counter (PC) - 8bits register, which act as pointer to next instructions in IRAM
6. RA, RB, RO - 16bits register to store input matrices and output matrices respectively.
7. RN - 8bits register to store matrix size
8. RP, RC, RR - 8bits registers, which act as single value pointer, column pointer and row pointer respectively.
9. RT - 16bits registers to store the intermediate total values
10. Accumulator (AC) and R - 16bits register to perform arithmetic operations
11. Z - 1 bit register
12. ALU - 16bits

2.2.3. Instruction Set and Microinstruction Sequence

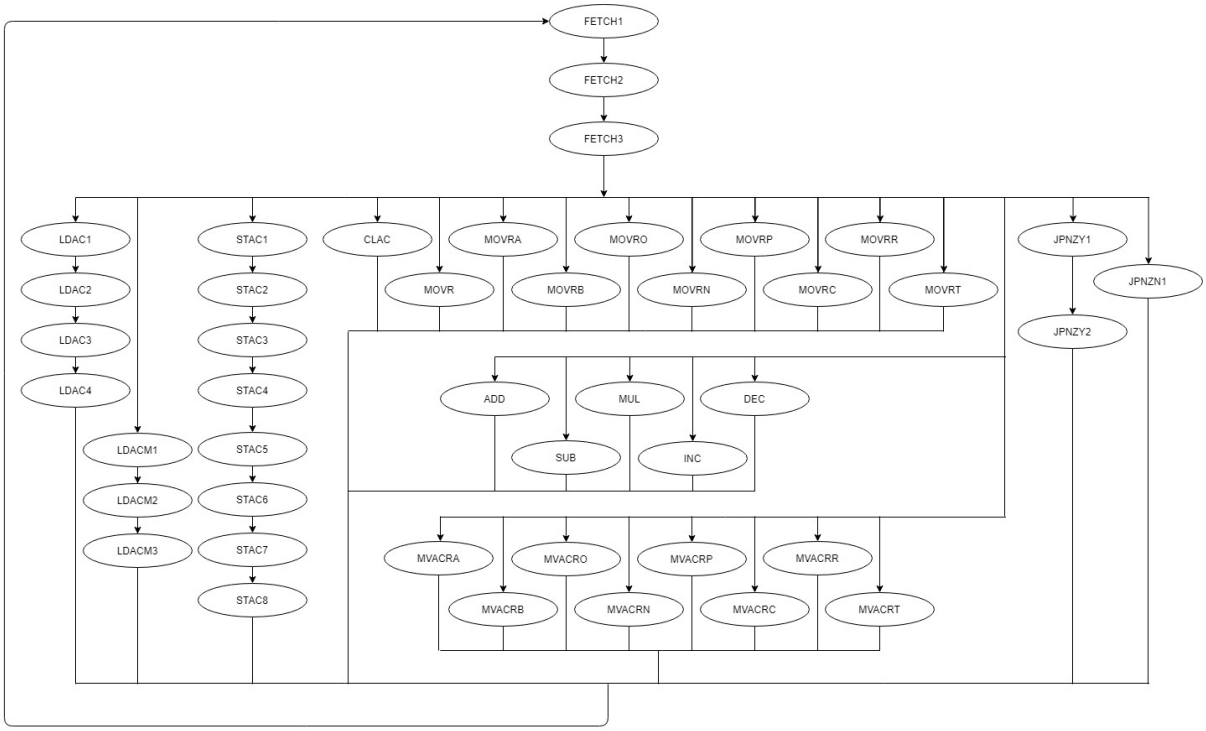
Following table gives the Instruction set architecture of the matrix multiplication processor built. Altogether 29 ISA has been used in the processor design.

ISA	ISA (steps)	ISA opcode	Micro instruction	Micro Instruction (Steps)	CU opcode
IDLE	Do Nothing	0	IDLE	Do Nothing	0
FETCH	FETCH	1	FETCH1	Read IRAM	1
			FETCH2	IR← IRAM, PC← PC+1	2
			FETCH3	Read IRAM	3
LDAC τ	AC ← τ	4	LDAC1	IR← IRAM, PC← PC+1	4
			LDAC2	TR← IR, Read IRAM	5

			LDAC3	IR ← IRAM, PC← PC+1	6
			LDAC4	AC← IR TR	7
LDACM	AC[0:8]← DRAM[AC]	8	LDACM1	AR←AC	8
			LDACM2	Read DRAM	9
			LDACM3	AC←DRAM[AR]	10
STAC	M[AC]← RT	11	STAC1	AR←AC	11
			STAC2	AC←RT	12
			STAC3	DR←AC[7:0]	13
			STAC4	DRAM←DR; Write	14
			STAC5	AC←RO, AC←AC+1	15
			STAC6	AR←AC	16
			STAC7	AC←RT	17
			STAC8	DRAM←AC[15:8]; Write	18
CLAC	AC ← 0, Z← 1	19	CLAC	AC ← 0, Z← 1	19
MOVR	R ← AC	20	MOVR	R ← AC	20
MOVRA	RA ← AC	21	MOVRA	RA ← AC	21
MOVRB	RB ← AC	22	MOVRB	RB ← AC	22
MOVRO	RO ← AC	23	MOVRO	RO ← AC	23
MOVRN	RN ← AC[0:8]	24	MOVRN	RN ← AC[0:8]	24
MOVRP	RP ← AC[0:8]	25	MOVRP	RP ← AC[0:8]	25
MOVRC	RC ← AC[0:8]	26	MOVRC	RC ← AC[0:8]	26
MOVRR	RR ← AC[0:8]	27	MOVRR	RR ← AC[0:8]	27

MOVRT	RT ← AC[0:8]	28	MOVRT	RT ← AC[0:8]	28
MVACRA	AC← RA	29	MVACRA	AC← RA	29
MVACRB	AC← RB	30	MVACRB	AC← RB	30
MVACRO	AC← RO	31	MVACRO	AC← RO	31
MVACRN	AC← RN	32	MVACRN	AC← RN	32
MVACRP	AC← RP	33	MVACRP	AC← RP	33
MVACRC	AC← RC	34	MVACRC	AC← RC	34
MVACRR	AC← RR	35	MVACRR	AC← RR	35
MVACRT	AC← RT	36	MVACRT	AC← RT	36
ADD	AC ← AC+R	37	ADD	AC ← AC+R	37
MUL	AC ← AC*R	38	MUL	AC ← AC*R	38
SUB	AC ← AC-R	39	SUB	AC ← AC-R	39
INC	AC ← AC+1	40	INC	AC ← AC+1	40
DEC	AC← AC-1	41	DEC	AC← AC-1	41
JPNZ τ	IF Z=0, GOTO τ	42	JPNZY1	IR←IRAM	42
			JPNZY2	PC←IR	43
			JPNZN1	PC←PC+1	44
ENDOP	End Operation	46	ENDOP	End Operation	46

2.2.4. State Diagram



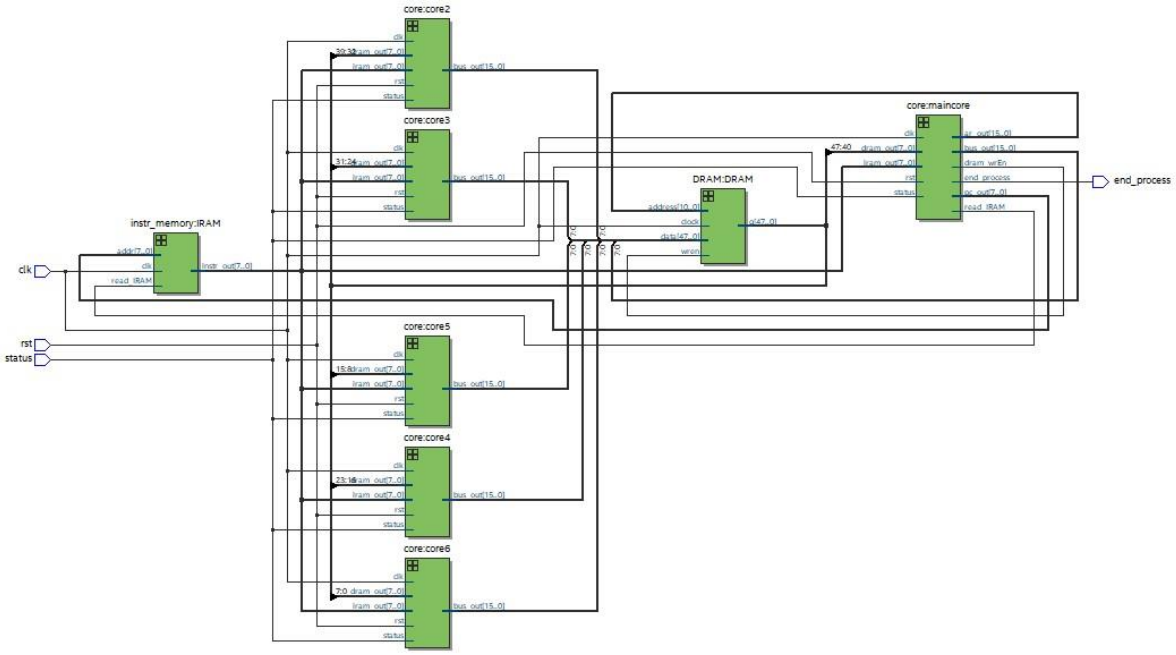
2.3. Modules and Components

2.3.1. Multicore Processor

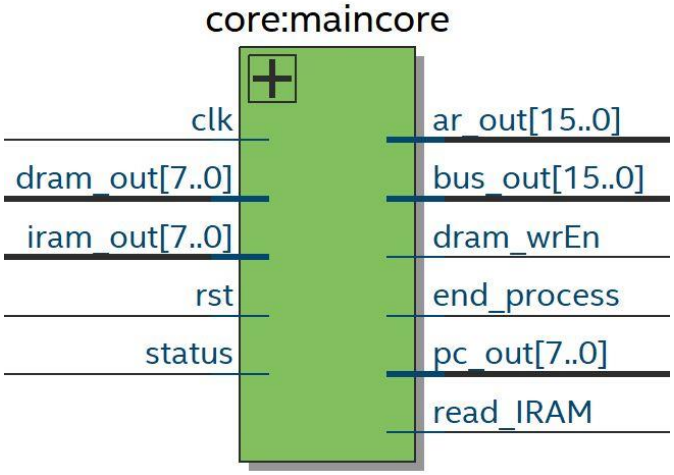
This module connects multiple core modules and memory modules (DRAM and instruction memory) to enable matrix multiplication. The inputs to the module are as follows:

1. **clk:** Clock Signal
2. **rst:** reset signal.
3. **status:** This signal indicates “Start” status to the core to initiate initial fetch cycle.

The module gives **end_process** signal as an output which indicates the end of the multiplication process. In the multi core design only the main core is connected to instruction memory through pc_out, read_IRAM and connected to DRAM through ar_out, dram_wrEn. In our design, main core controls the instructions run in the other cores by controlling IRAM and enables other cores to read from and write to DRAM. Other cores in the multi-processor module are connected to IRAM and DRAM only through iram_out, bus_out and dram_out.



2.3.2. Processor (Single core)



The processor (core) module combines all the registers, increment registers, ALU, BUS and the control unit in the architecture to enable processing part of the system. This module does not include instruction memory and data memory. The inputs to the core module are as follows:

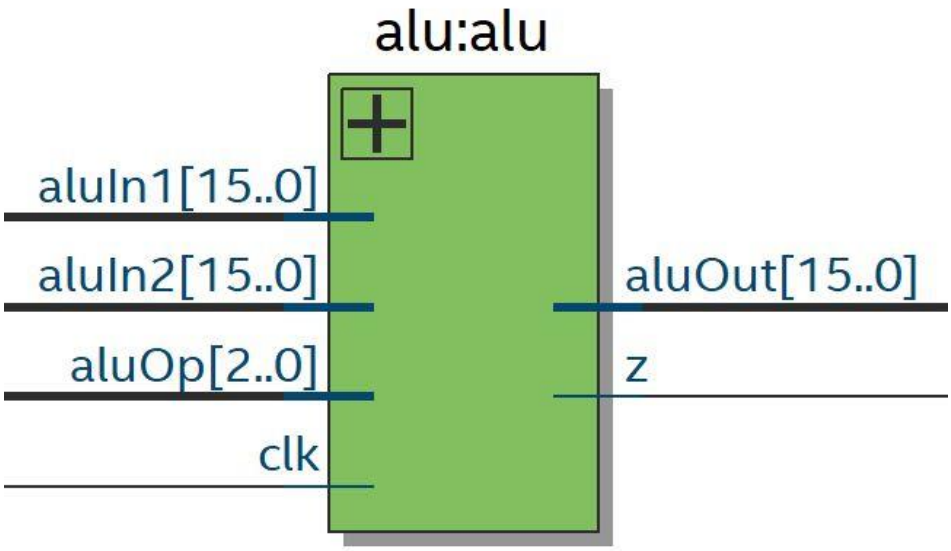
4. **clk:** Clock Signal
5. **dram_out:** The output of the DRAM, which is the memory element in the memory address pointed by AR register.

6. **iram_out:** The output of the Instruction memory, which is the instruction in the memory address pointed by PC register.
7. **rst:** reset signal.
8. **status:** This signal indicates “Start” status to the core to initiate initial fetch cycle.

The outputs of the core module are as follows:

1. **ar_out:** The address in the AR register, which should be connected to address in the DRAM.
2. **bus_out:** The data in the BUS, which should be connected to the data in DRAM.
3. **pc_out:** The address in the PC register, which should be connected to address in the instruction memory.
4. **dram_wrEn:** Output control signal from the control unit to enable DRAM read.
5. **read_IRAM:** Output control signal from the control unit to enable instruction memory read.
6. **end_process:** Output control signal from the control unit, which indicates end of the process.

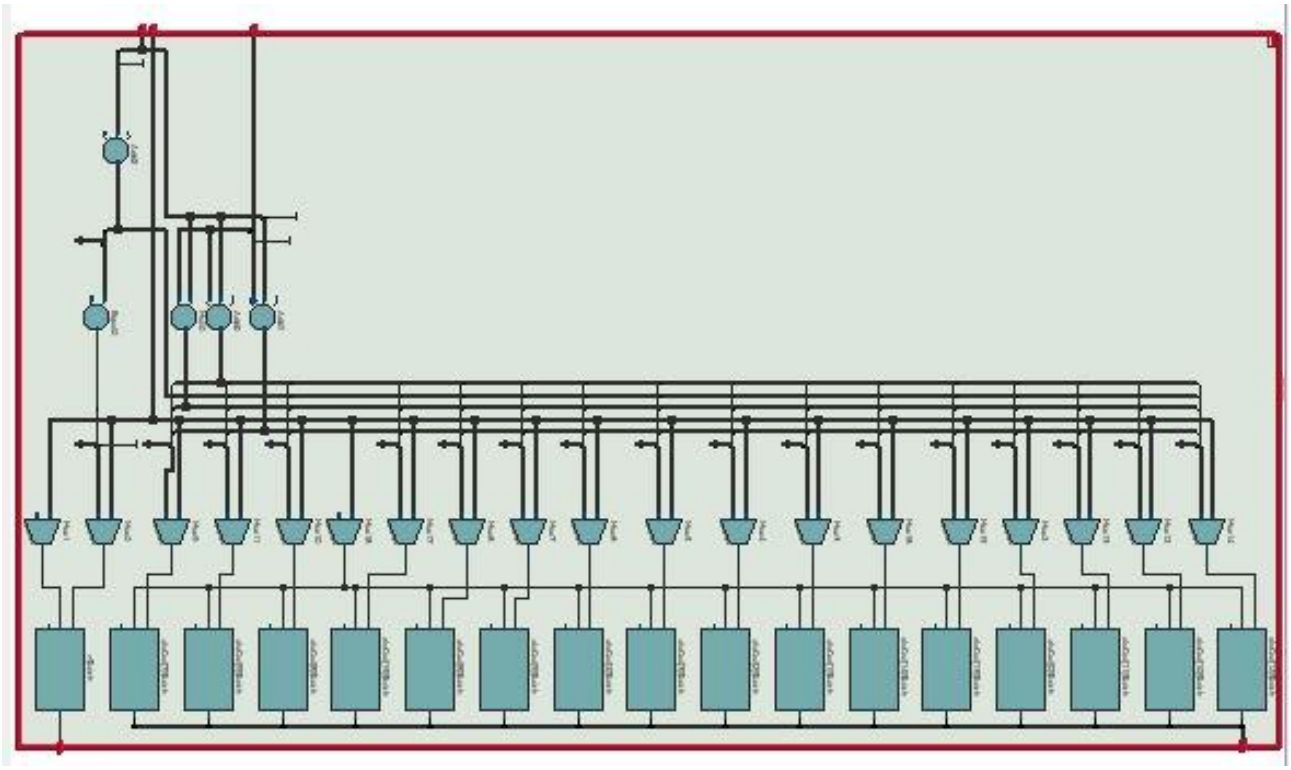
2.3.3. ALU



ALU module is used for handling all arithmetic and logical operations. In this module, input ports are connected to R (**aluIn2**) and AC (**aluIn1**) registers and the subsequent output is fed to AC (**aluOut**). It also consists of a **Z** flag, that gets updated during subtraction (DEC) and clear (CLAC) operations, for the purpose of handling jumps. The operation to be

performed by the ALU is fed using an encoded 3 bit control signal termed **aluOp**. AluOp values and the corresponding operations are as follows.

aluOp	Instruction	ALU Operation	Z flag
001	ADD	AC ← AC + R	X
010	SUB	AC ← AC – R	X
011	MUL	AC ← AC × R	X
100	DEC	AC ← AC – 1	AC – 1 = 0 → Z = 1
			AC – 1 ≠ 0 → Z = 0
101	CLAC	AC ← 0	1

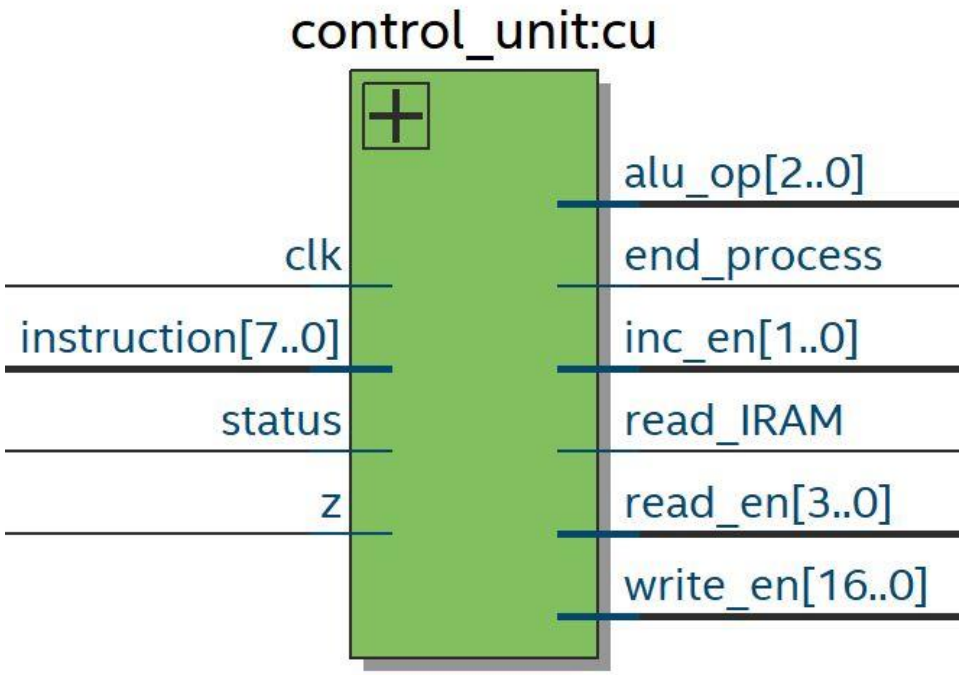


2.3.4. CU

The control unit produces necessary control signals to control each module in a core to achieve the given task and control the data flow in the processor. It takes the instructions from the instruction memory and converts it into a sequence of signals. The input to the control units are as follows:

1. **clk:** Clock Signal

2. **instruction:** Instruction opcode in the instruction register fetched from the instruction memory. The control unit generates sequence of signals based on the opcode in this input.
3. **status:** This signal indicates “Start” status to the control unit to initiate initial fetch cycle.
4. **z:** Value of z, based on output of ALU.



The signals produced by the control unit as follows:

1. **alu_op:** 3-bit control signal to control the functions of the ALU unit.
2. **inc_en:** 2-bit control signal to enable increment in PC and AC registers.
3. **read_IRAM:** Control signal to read instruction from instruction memory.
4. **read_en:** 4-bit control signal to control the dataflow from registers and DRAM to the bus.
5. **write_en:** 17-bit control signal to enable writing to the registers and DRAM.
6. **end_process:** Control signal to indicate the end of the process.

The format of the control signals generated by the control unit is given below:

read_ IRAM 1 bit	end_ process 1 bit	read_en 4 bits	write_en 17 bits	Inc_en 2 bits	Alu_op 3 bits	Next address 6 bits

Read_IRAM:

This signal is generated during fetch cycle and the instances where reading instructions from the instruction memory is necessary. When this signal is given to the instruction memory it will write the instruction at the memory location pointed by the Program Counter register to Instruction Register.

Read_en:

The 4-bit control signal, which is connected to the bus controls the data flow from the registers and DRAM to the BUS. This signal eliminates the error caused by reading multiple registers to the BUS and reading data from DRAM to BUS. The encoding values for reading data to the bus from registers and DRAM are given as follows:

Register/ DRAM	Read_en Control Signal
IR	0001
TR	0010
DR	0011
RA	0100
RB	0101
RO	0110
RN	0111
RP	1000
RC	1001
RR	1010
RT	1011
AC	1100
DRAM	1101
IR.TR	1110
AC[15:8]	1111

Write_en:

This is a 17-bit signal, where each bit is connected separately to a register / DRAM to enable writing to those modules. In all the registers other than IR, PC, TR and R, converting write_en to 1 enables writing to those registers from the BUS. IR, PC, TR and R are not

connected to the BUS, and write_en =1 enables writing from IR← Instruction memory, PC ←IR, TR←IR and R← AC respectively. Write_en signal connected to the DRAM enables writing data from BUS to the memory location indicated by AR. AC register has two write_en signals, where on is to enable writing from BUS and other from the ALU. The format of the write_en control signal as follows:

WRITE EN

16 DRAM<- BUS	15 AC<-ALU	14 BUS->AC	13 R	12 RT	11 RR	10 RC	9 RP	8 RN	7 RO	6 RB	5 RA	4 DR	3 TR	2 IR	1 PC	0 AR

Inc_en:

2-bit signal connected separately to AC and PC registers, which enables incrementation of the stored values. The format of the control signal as follows:

INC EN

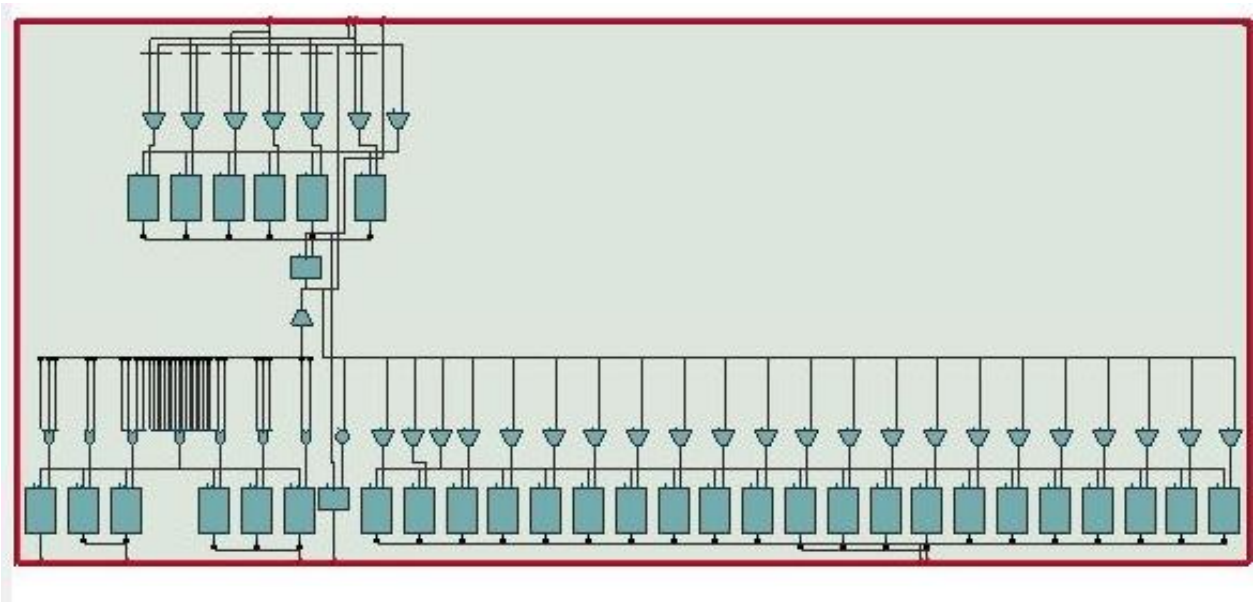
1	0
AC <- AC+1	PC <- PC + 1

Alu_op:

3-bit control signal connected to ALU of the processor and enables 8 different ALU operations, but for matrix multiplication 5 operations are sufficient. The ALU does addition, subtraction, multiplication, decrement, and also assigns its output to zero and z to 1 according to received control signal. The ALU operations for alu_op control signals are summarized in the table below:

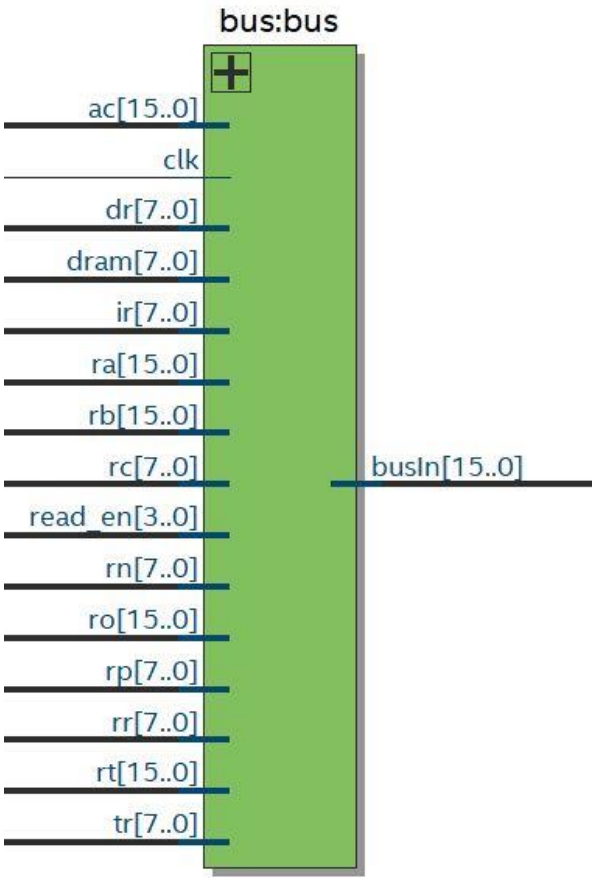
ALU Operation	**Alu_op Control Signal**
Addition	0001
Subtraction	0010
Multiplication	0011
Decrement	0100

ALU Out ← 0 Z ← 1	0101



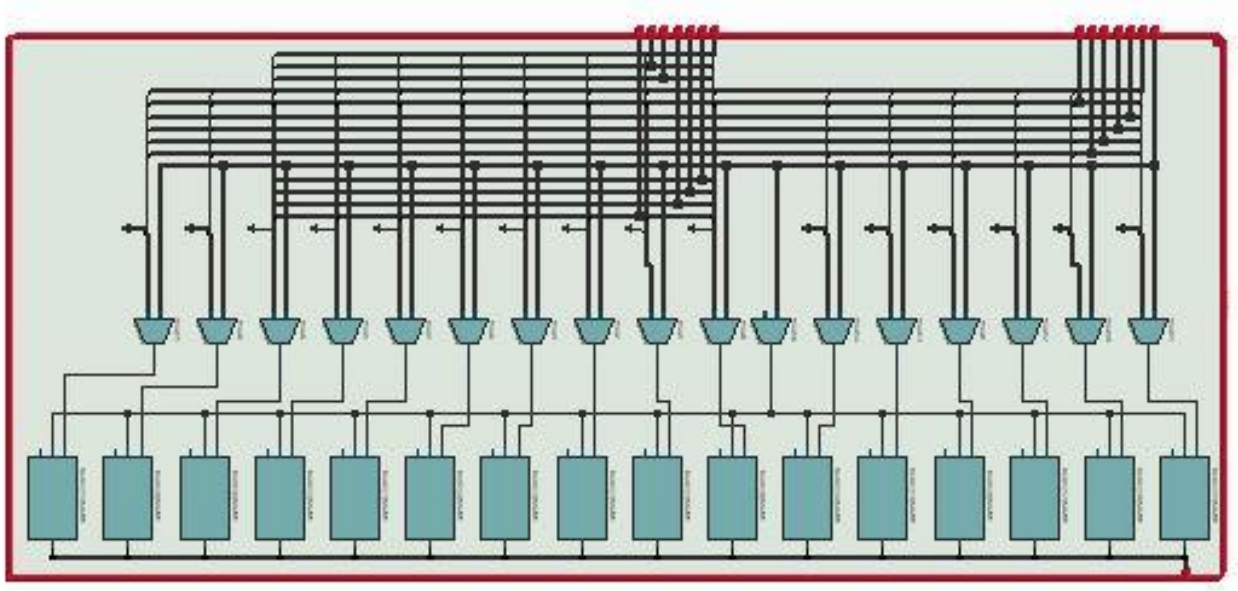
Bus

Each core has a dedicated bus to facilitate data transfer between modules. The bus can read data from one module at a time. The DEMUX implementation of the bus is given below.

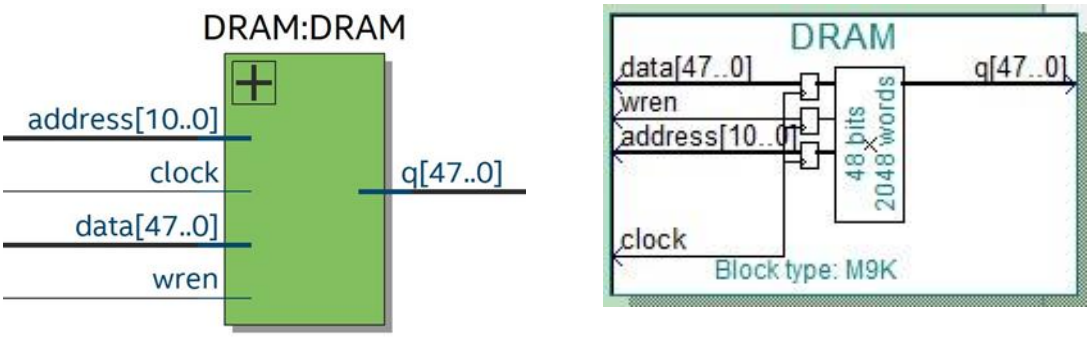


In the bus architecture, the bus takes control signals corresponding to the registers IR, TR, DR, RA, RB, RO, RN, RP, RC, RR, RT, and AC and can read data from those registers. (data memory) is also directly in connection with the bus. Since there are 15 unique cases of data reading from the bus, 4 bit flag is used to address them as given below.

1. IR:	0001	9. RC:	1001
2. TR:	0010	10. RR:	1010
3. DR:	0011	11. RT:	1011
4. RA:	0100	12. AC (8 LSB):	1100
5. RB:	0101	13. DRAM:	1101
6. RO:	0110	14. IR.TR:	1110
7. RN:	0111	15. AC (8 MSB):	1111
8. RP:	1000		



2.3.5. DRAM



The main memory in the system which stores the data. In our application, matrix size, max rows per core, matrix 1, matrix 2 are written to DRAM initially and later output matrix is stored. This is done with the aid of MATLAB to generate ".mif" file format to initiate the DRAM. The word size of the DRAM depends on the number of cores used for multiplication.

$$\text{Word size} = 8\text{bits} \times \text{Number of Cores}$$

The number of memory locations in the DRAM is 65536 (216). Values in 1st and 2nd matrix are stored in one memory location in the memory, but for output matrix two memory location per value was utilized. Also, each memory location was sliced into number of cores and allocated to each core in the processor.

The memory allocation in the DRAM as follows:

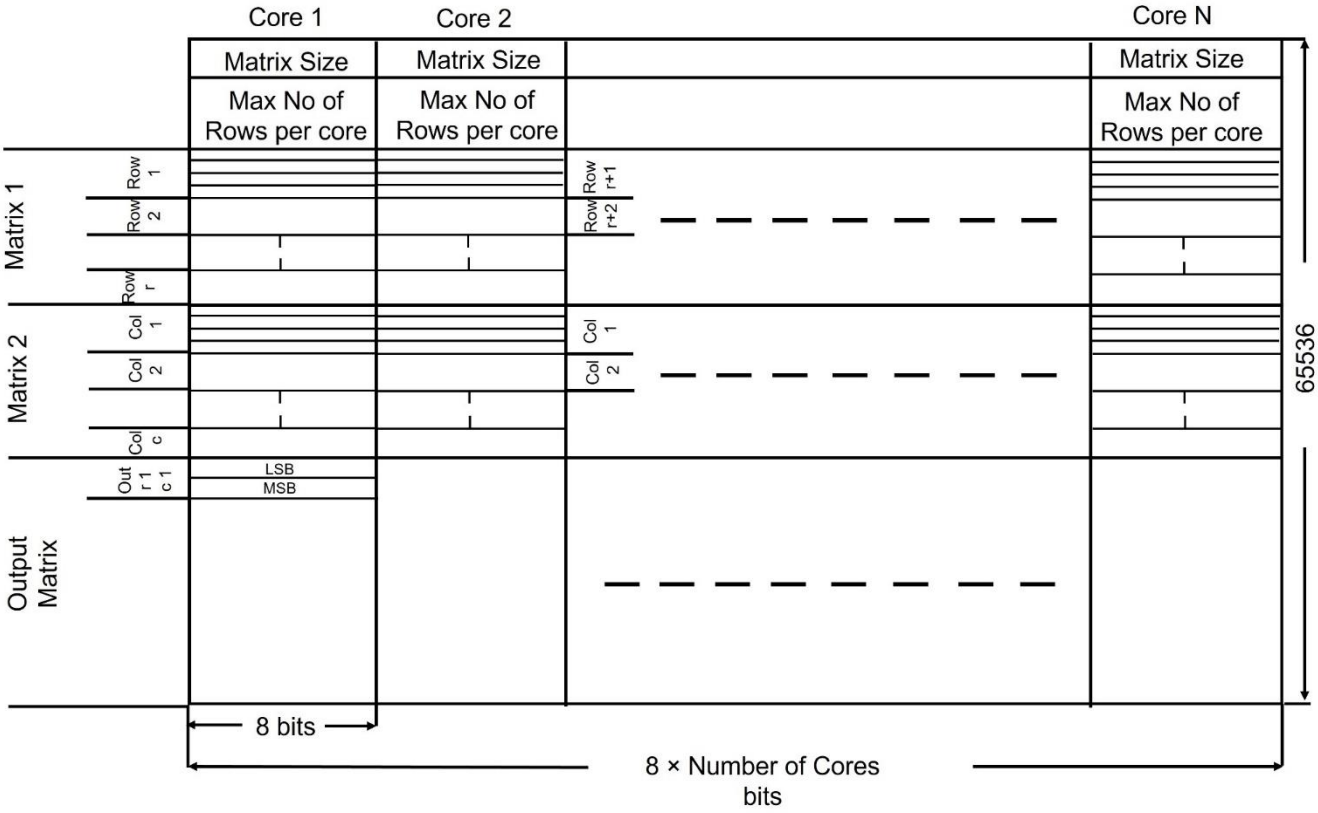
	Memory Address (16 bits)
Matrix Size	0
Maximum Rows per Column	1
Matrix 1	2 - 257
Matrix 2	258 - 514
Output Matrix	515 - 1026

The inputs to the DRAM are as follows:

1. clk: Clock Signal
2. address: Memory location address to read or write to the DRAM, which is indicated by the AR register.
3. data: Data which needs to be written to DRAM at the location indicated by AR register.

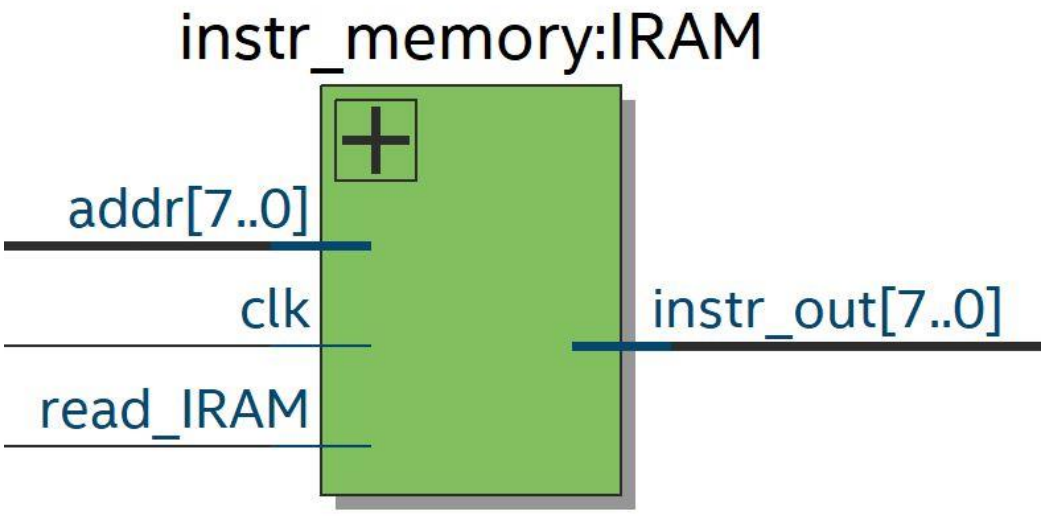
4. wren: Signal to enable memory write to the DRAM.

The output of the DRAM is q, which is the data in the memory location pointed by the AR register.

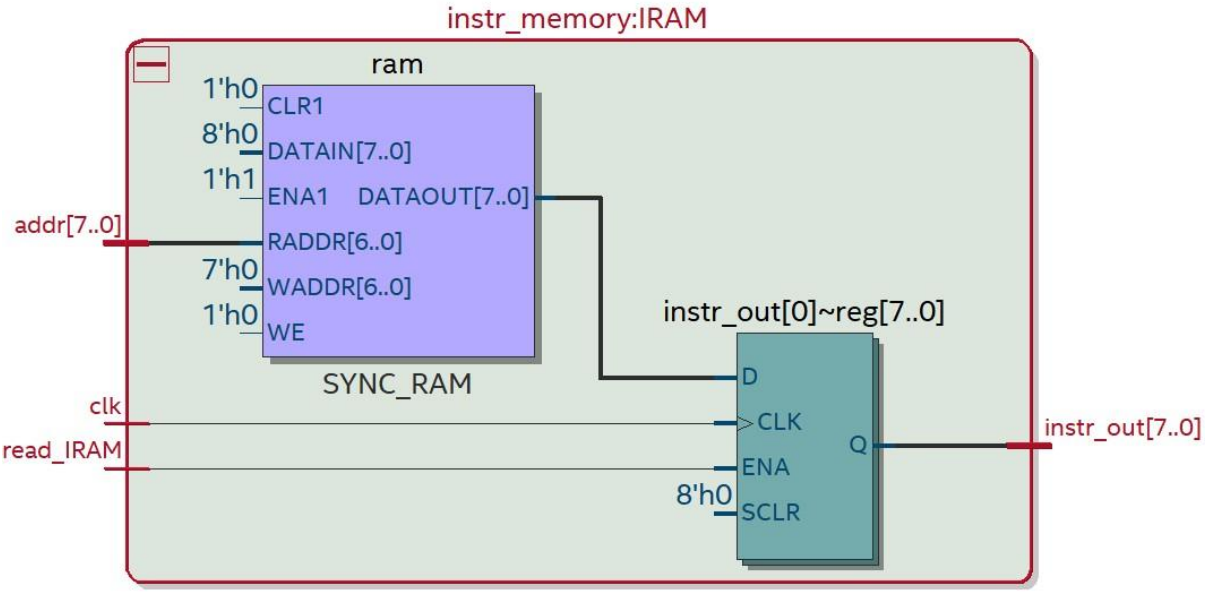


2.3.6. IRAM

An Instruction RAM (IRAM) module was designed to store the assembly codes in order. The word size is 8bits and number of memory location allocated was 95, since the length of assembly code is 91. PC register is used as the pointer to IRAM as well as the IR register used



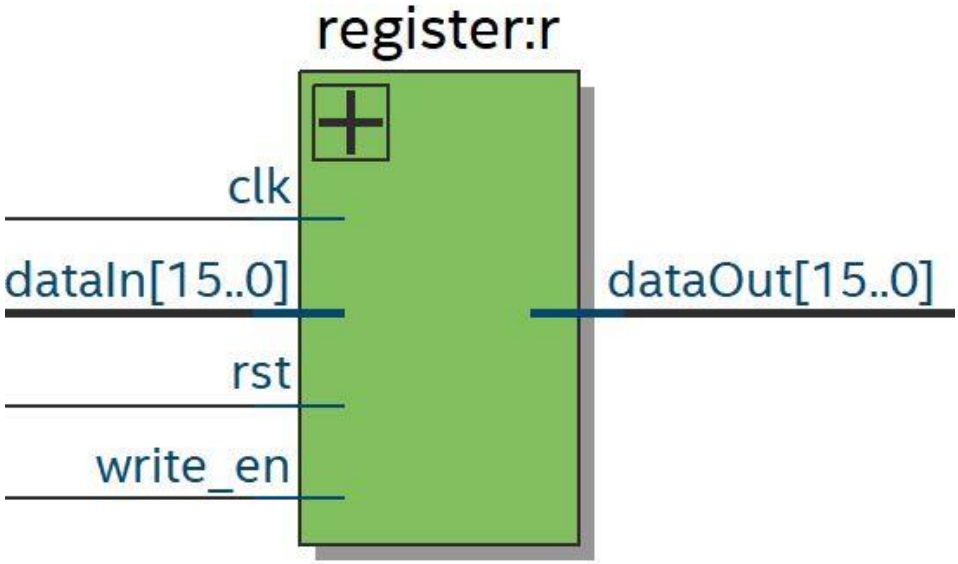
to store the instructions. The Control Unit (CU) gives the control signals to the IRAM to read IRAM. Write IRAM was not used.

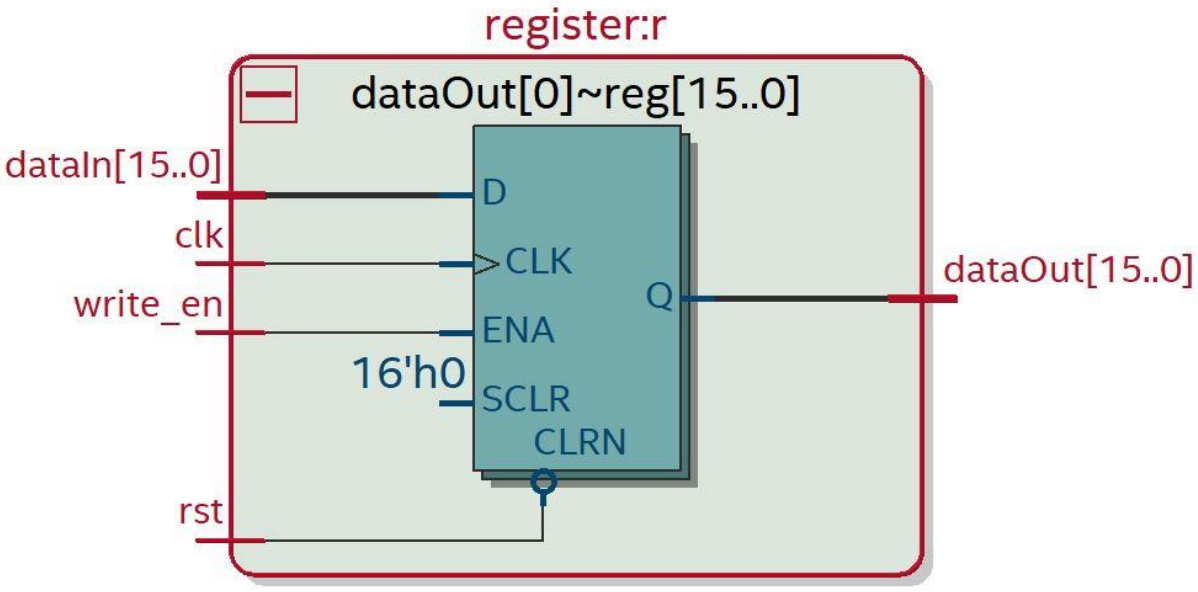


2.3.7. Registers without Increment

IR, TR, DR, RN, RP, RC, RR : 8 bits, AR, RA, RB, RO, RT, R : 16 bits.

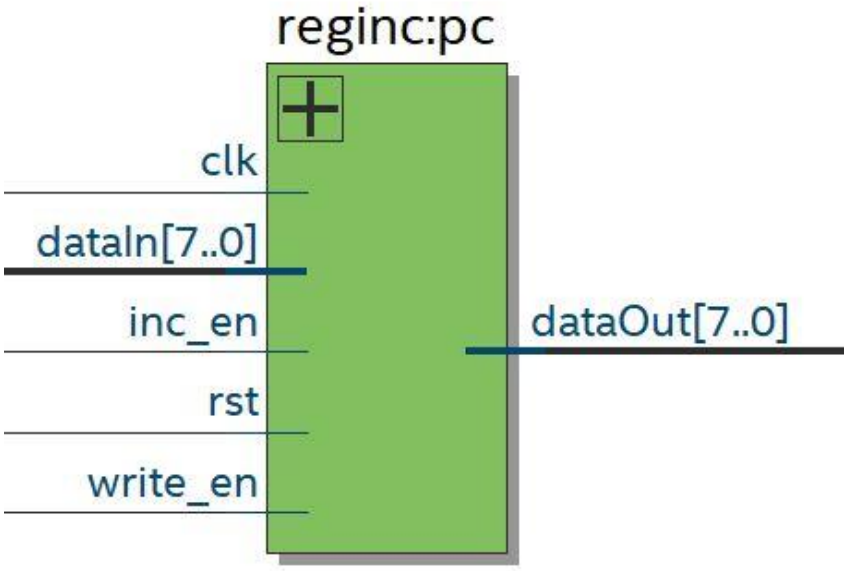
Registers that are designed for the sole purpose of temporarily storing values fall under this category. Data storing is controlled by **write_en** control signal. i.e., whenever **write_en** is high, the values in the input port (**dataIn**) will be stored for transmission.

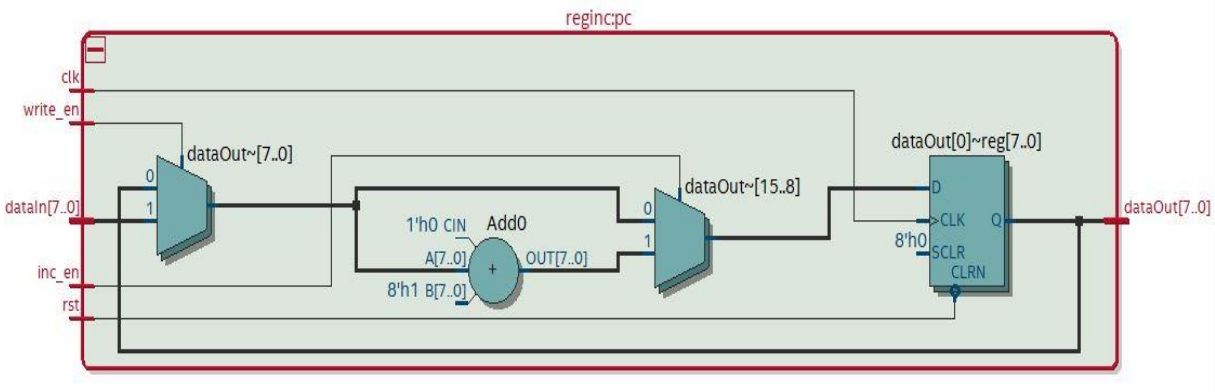




2.3.8. Registers with Increment (PC)

Registers with the capability of incrementing the stored value by one without feeding the data to the ALU fall under this category. This additional capability is coordinated by the control signal of **inc_en**. Whenever **inc_en** signal is set to high, the stored value will be updated by one after one clock cycle. If both **inc_en** and **dataIn** control signals are high, the value in the input port will be updated by one and stored to the register after one clock cycle.





2.3.9. Accumulator (AC)

AC is an enhanced register module designed for feeding an operand to ALU and also for storing the corresponding output values from ALU. AC module receives its inputs from both bus (**dataIn**) and ALU (**aluIn**) modules, which are controlled with **write_en** and **alu_en** control signals respectively. Also, AC has the capability of incrementing the input value by one and it is controlled by **inc_en** control signal. With this, AC can increment the input value and store the resultant at the same clock cycle if both **inc_en** and **write_en/ dataIn** control signals are high. The output from AC (**dataOut**) is fed to ALU, R and bus modules.

3. Design Considerations

3.1. Data Processing Techniques

Input Matrix Generation

Initially matrices should be generated to carry out the assigned task. MATLAB script was developed for that purpose where dimensions of input matrices and number of cores to be used are fed. In the MATLAB script, matrices with random integers and the specified dimensions would be generated initially. Then based on the number of cores, rows would be divided between the cores. In our design, each row of RAM will be partitioned such that each core will have a portion representing one integer. i.e., for n cores implementation, converting the first line of RAM to binary provides the concatenation of binary values of the first integers assigned to each core. With our MATLAB script, a data file of .mif type would be generated according to the aforementioned design. The file location of the generated .mif file would be indicated during the creation of DRAM module. In the topmost layer (top level entity), these values in RAM would be partitioned among the specified cores.

	Core 1 : Assigned Values	Core 2 : Assigned Values	Core 3 : Assigned Values	Values in input (.mif) file
1	00110011 (51)	00001111 (15)	00010110 (22)	001100110000111100010110 (3,346,198)
2	00101010 (42)	00010100 (20)	00001010 (10)	001010100001010000001010 (2,757,642)

Output Matrix Generation

Output values generated by the simulation are divided into two and stored in successive memory locations. In the multicore processor design, the first row consists of the least significant bits of the calculated values, while the second row comprises of the most significant bits. For e.g., for a 3 core design, each row will be 3×8→24 bits long and the 8 bits dedicated to each core provides either MSB or LSB values of the output integers. To decode that into regular unsigned integers, python script was developed which would take the number of cores and the output .mem file from simulation (ModelSim) as the inputs and generate the output matrix. Also, it compares the output values with the expected values from MATLAB and provides with the comparison matrix which indicates 1 if the values match and 0 if they do not.

	Values in output (.mem) file	Core 1 : Calculated Values	Core 2 : Calculated Values	Core 3 : Calculated Values
1	001100110000111100010110	0010101000110011 (10,803)	0001010000001111 (5,135)	0000101000010110 (2,582)
2	001010100001010000001010			
1	110010111100111111110110	0100100111001011 (18,891)	0001011011001111 (5,839)	1111101011110110 (64,246)
2	010010010001011011111010			

3.2. Assembly code for Square Matrix

```
01.  CLAC
02.  LDAC      //load the memory location of matrix size in DRAM
03.  “N1”      //16bits address loaded as 2 8 bit address
04.  “N2”
05.  LDACM     //load the matrix size value from DRAM
06.  MOVRP     //move the matrix size to RP and RN registers
07.  MOVRN
08.  LDAC
09.  “N3”      //load the maximum rows per core
10.  “N4”
11.  LDACM
12.  MOVRR     //move the maximum number of rows per core to RR
13.  LDAC      //load the memory location of first element of matrix A
14.  “M1_1”
15.  “M1_2”
16.  MOVRA     //move it to RA register
17.  LDAC      //load the memory location of first element of matrix B
18.  “M2_1”
19.  “M2_2”
```

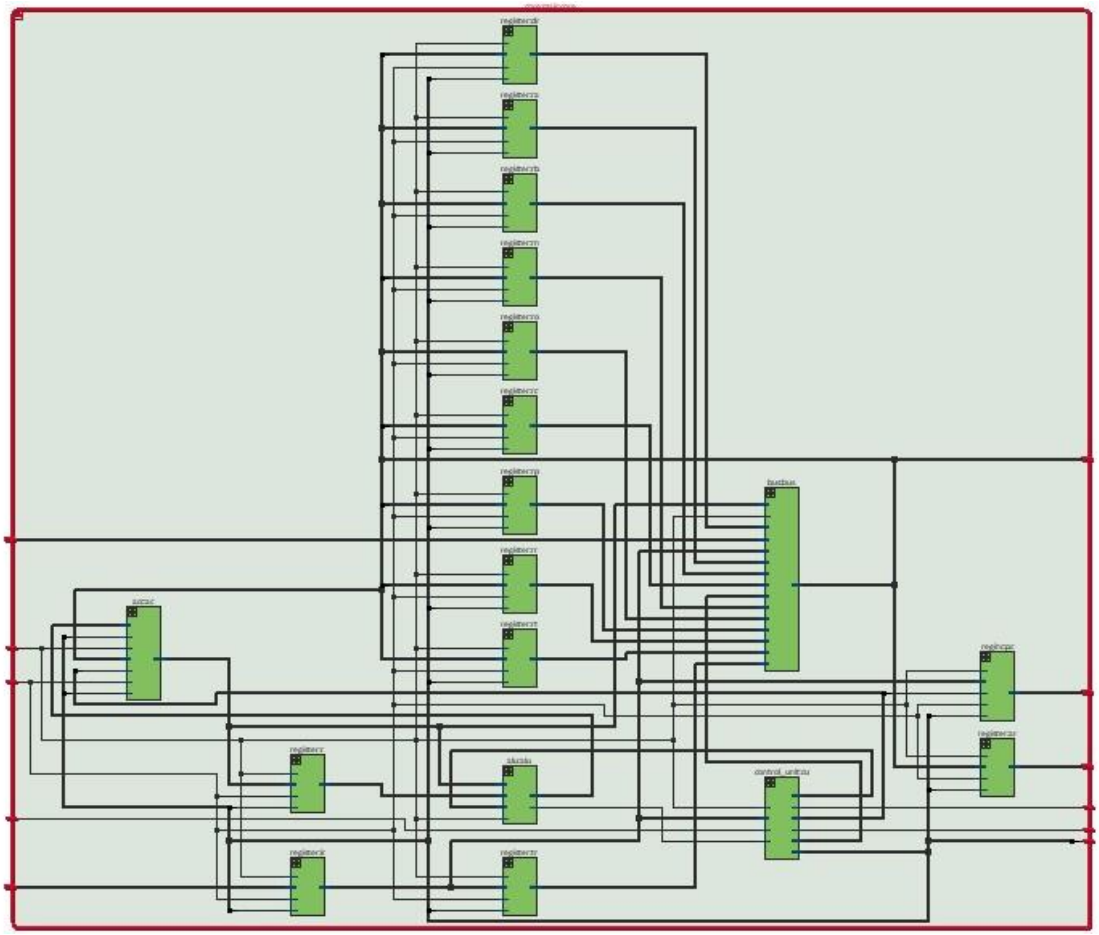
20. MOVRB ////move it to RB register
21. LDAC //load the memory location of first element of matrix O
22. “M3_1”
23. “M3_2”
24. MOVRO //move it to RO register
25. MVACRN
26. MOVRC //Column counter set to matrix size
27. CLAC //clear ACC
28. MOVRT //RT (total register) set to zero
29. MVACRN
30. MOVRP //point counter set to matrix size
31. MVACRA
32. LDACM //element of matrix A (memory location pointed by RA) is loaded.
33. MOVR
34. MVACRB //element of matrix B (memory location pointed by RB) is loaded.
35. LDACM
36. MUL //multiply the element of A and B matrix
37. MOVR
38. MVACRT
39. ADD
40. MOVRT //Multiplied value stored in RT register
41. MVACRA
42. INC //memory location pointed by RA increased by 1
43. MOVRA
44. MVACRN
45. MOVR
46. MVACRB
47. ADD //memory location pointed by RB increased by matrix size
48. MOVRB

```
49.  MVACRP
50.  DEC        //point counter decreased by one.
51.  MOVRP
52.  JUMP
53.  "31"       //loops until the RP (pointer counter) goes to zero
54.  MVACRO
55.  STAC       //store the total value in DRAM (allocated slot for output matrix)
56.  MVACRO
57.  INC        //16bits output is written into 8bits location
58.  INC
59.  MOVRO
60.  MVACRN
61.  MOVR
62.  MVACRA     //go back to first element of the row
63.  SUB
64.  MOVRA
65.  MVACRN
66.  MOVR
67.  MUL        //Calculate number of elements in the matrix
68.  MOVR
69.  MVACRB
70.  SUB
71.  INC
72.  MOVRB      //Go back to starting value
73.  MVACRC
74.  DEC
75.  MOVRC
76.  JUMP
77.  "27"       //Loop until all the element in one row of the output is calculated
```

```
78.  MVACRN
79.  MOVR
80.  MVACRA   //Goto the next row
81.  ADD
82.  MOVRA
83.  LDAC     //load the memory location of second matrix
84.  “M2_1”
85.  “M2_2”
86.  MOVRB
87.  MVACRR   //Row counter is decremented
88.  DEC
89.  MOVRR
90.  JUMP
91.  “25”
92.  ENDOP    //end of operations
```

4. Verilog Implementation

4.1. RTL Design Simulation



4.2. Quartus II Implementation

Hardware description language (HDL) is used for configuring the FPGA modules. In our case, we have used Verilog for code development and Quartus Prime 18.1 for compiling the codes. ModelSim was used to carry out simulations to check the performance of the developed codes. First all the modules were individually developed and corresponding testbenches were written. Once their performances were verified, a single core module was created as the top layer. Then single core processor module was further enhanced as the multi core processor module. Finally, testbench script was developed for the multi core processor module and was evaluated on ModelSim. All these processes were carried out for the multiplication of square matrices. Once the working of square matrix multiplication was verified, the design was upgraded for rectangular matrix multiplication, as well as test benches were also designed to verify the performance.

Flow Summary

Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	single_core
Top-level Entity Name	multi_core_processor
Family	Cyclone IV E
Device	EP4CE115F23C7
Timing Models	Final
Total logic elements	3,825
Total registers	1233
Total pins	4 / 281 (1 %)
Total virtual pins	0
Total memory bits	99,128 / 3,981,312 (2 %)
Embedded Multiplier 9-bit elements	12 / 532 (2 %)
Total PLLs	0 / 4 (0 %)

Resource Usage Summary

Estimated Total logic elements	3,825
Total combinational functions	2966
Logic element usage by number of LUT inputs	
-- 4 input functions	1673
-- 3 input functions	798
-- <=2 input functions	495
Logic elements by mode	
-- normal mode	2657
-- arithmetic mode	309
Total registers	1233
-- Dedicated logic registers	1233

-- I/O registers 0

I/O pins 4

Total memory bits 99128

Embedded Multiplier 9-bit elements 12

Maximum fan-out node clk~input

Maximum fan-out 1119

Total fan-out 16618

Average fan-out 3.88-- arithmetic mode 309

5. Performance Evaluation

5.1. Comparison with MATLAB implementation

Matrix Size	Time taken with MATLAB	No of Cores	Time taken with FPGA processor
16x16	174,000 ns	6	1,560,210 ns
	149,000 ns	5	2,079,730 ns
	146,000 ns	4	2,079,730 ns
	101,000 ns	3	3,118,770 ns
	113,000 ns	2	4,157,810 ns
	138,000 ns	1	8,313,970 ns
15x15	213,000 ns	6	1,378,410 ns
	305,200 ns	5	1,378,410 ns
	191,000 ns	4	1,837,330 ns
	255,000 ns	3	2,296,250 ns
	930,000 ns	2	3,673,010 ns
	143,000 ns	1	6,885,450 ns
2x2	136,000 ns	1	28,210 ns

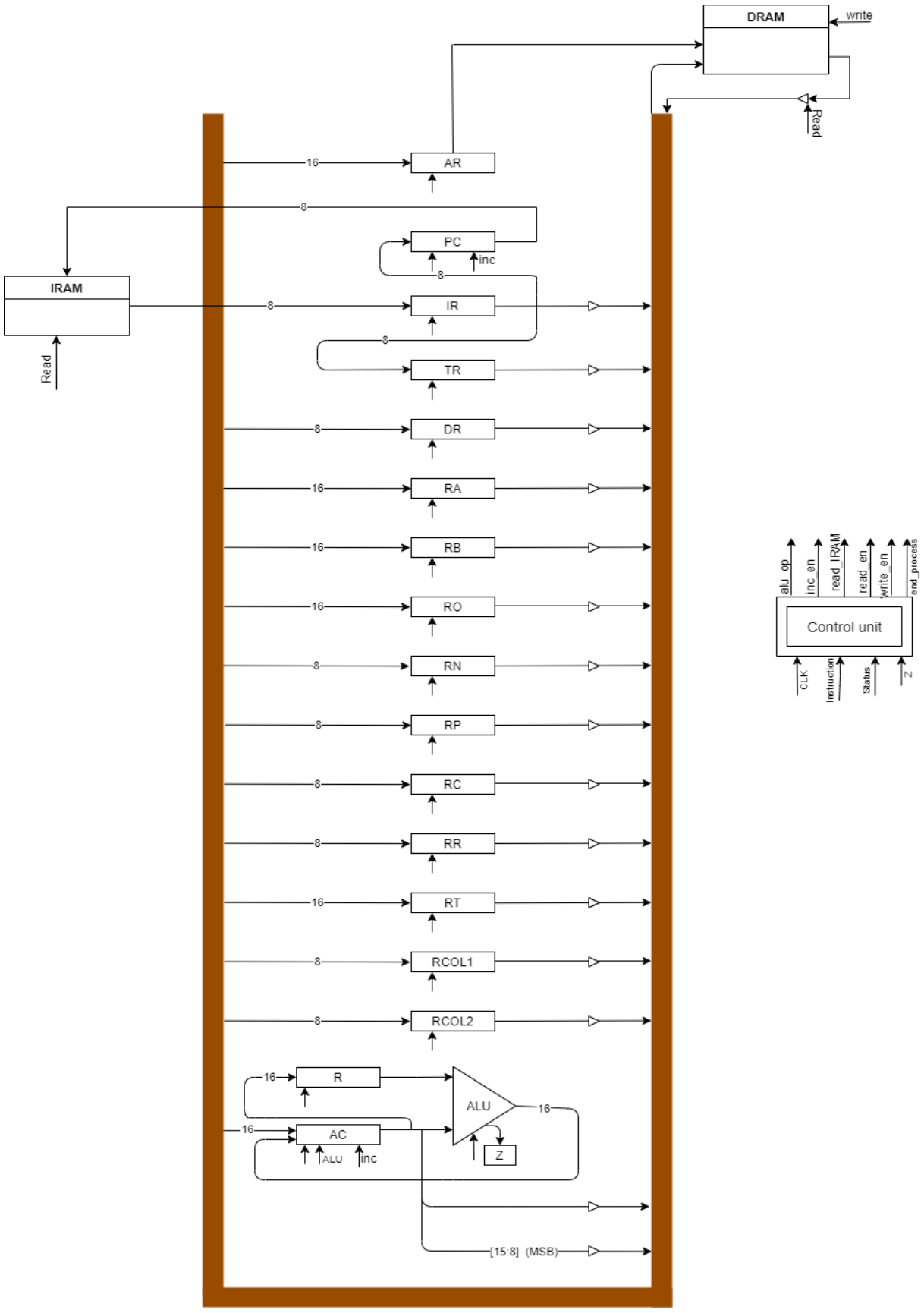
The result in the table clearly states that the incorporation of multiple cores has improved matrix multiplication. The time taken using 6 cores is around 5 fold better than time taken by a core. Similar timing between certain cores for certain matrix sizes can be observed.

Ex: Time taken for 15 × 15 matrix multiplication using 6 cores and 5 cores.

The reason for this observation is because maximum rows allocated for a core is same for 6 cores and 5 cores, which is 3 rows.

6. Design for Rectangular Matrix

6.1. Data Path



6.2. Instruction set and Micro Instructions

ISA	ISA (steps)	ISA opcode	Micro instruction	Micro Instruction (Steps)	CU opcode
MOVRCOL1	RCOL1←AC	47	MOVRCOL1	RCOL1←AC	47
MOVACRCOL1	AC ←RCOL1	48	MOVACRCOL1	AC ←RCOL1	48
MOVRCOL2	RCOL2←AC	49	MOVRCOL2	RCOL2←AC	49
MOVACRCOL2	AC← RCOL2	50	MOVACRCOL2	AC← RCOL2	50

6.3. Assembly code for rectangular matrix

1) CLAC
2) LDAC
3) 8'd0
4) 8'd0
5) LDACM
6) MOVRN
7) MOVRP
8) LDAC
9) 8'd1
10) 8'd0
11) LDACM
12) MOVRR
13) LDAC
14) 8'd2
15) 8'd0
16) LDACM
17) MOVRCOL1
18) LDAC
19) 8'd3
20) 8'd0
21) LDACM
22) MOVRCOL2
23) LDAC
24) 8'd4
25) 8'd0
26) MOVRA
27) LDAC
28) 8'd4
29) 8'd1

30) MOVRB
31) LDAC
32) 8'd4
33) 8'd2
34) MOVRO
35) MVACRCOL2
36) MOVRC
37) CLAC
38) MOVRT
39) MVACRCOL1
40) MOVRP
41) MVACRA
42) LDACM
43) MOVR
44) MVACRB
45) LDACM
46) MUL
47) MOVR
48) MVACRT
49) ADD
50) MOVRT
51) MVACRA
52) INC
53) MOVRA
54) MVACRCOL2
55) MOVR
56) MVACRB
57) ADD
58) MOVRB
59) MVACRP
60) DEC
61) MOVRP
62) JPNZ
63) 8'd40
64) MVACRO
65) STAC
66) MVACRO
67) INC
68) INC
69) MOVRO
70) MVACRCOL1
71) MOVR
72) MVACRA
73) SUB

74) MOVRA
75) MVACRCOL1
76) MOVR
77) MVACRCOL2
78) MUL
79) MOVR
80) MVACRB
81) SUB
82) INC
83) MOVRB
84) MVACRC
85) DEC
86) MOVRC
87) JPNZ
88) 8'd36
89) MVACRCOL1
90) MOVR
91) MVACRA
92) ADD
93) MOVRA
94) LDAC
95) 8'd4
96) 8'd1
97) MOVRB
98) MVACRR
99) DEC
100) MOVRR
101) JPNZ
102) 8'd34
103) ENDOP

6.4. Design modifications

Initially design process was carried out for multiplication of square matrices. Once it was successfully implemented and verified, amendments were made to the developed modules to incorporate multiplication of rectangular matrices. The major amendments and the associated modules are as follows.

1. Core → Two new registers – RCOL1, RCOL2 were created for storing the number of columns of both input matrices.
2. IRAM → Number of columns were loaded to the newly created matrices and used for controlling the jumps.

3. DRAM → Number of columns of both matrices were included in the .mif file trough MATLAB script.
4. Control Unit → Microinstructions corresponding to the newly created registers were defined.
5. Bus → Outputs of both RCOL1 and RCOL2 registers were fed to the bus.

Appendix

1. MATLAB code for generating MIF (Memory Initiation File) for FPGA designing

```
clear;
clc;
close all;

% fid=fopen('data_8bits.txt');% open the .txt file of the
data to be converted
% temp=fscanf(fid,'%x');%scan file
cores = 4;
n = 15; % matrix size (maximum is 16)
maximum_value = 63; %maximum value (since data memory word
size is 8bits)
maximum_mem_location = 256;
matrix_A = (randi(maximum_value,[n,n])); %generate matrix
with random values
transposed_matrix_A = matrix_A.';  %transpose
flatten_A = transposed_matrix_A(:); %column vector

matrix_B = (randi(maximum_value,[n,n]));
transposed_matrix_B = matrix_B.';
flatten_B = transposed_matrix_B(:);



fid=fopen('C:/Users/Vinith
Kugathasan/OneDrive/Desktop/Multi-
core/Single_Core/data.mif','w');% open the .mif file to be
written
%fid=fopen('C:/Users/Vinith
Kugathasan/OneDrive/Desktop/one_core_complete/Single_Core/
data.mif','w');% open the .mif file to be written
fprintf(fid,'WIDTH='+string(8*cores)+';\n');% write
storage bit width 8 bits;
fprintf(fid,'DEPTH=2048 ;\n');% write memory depth 2048
fprintf(fid,'ADDRESS_RADIX=UNS;\n');% write address type
is unsigned integer
%fprintf(fid,'DATA_RADIX=HEX;');
fprintf(fid,'DATA_RADIX=UNS;\n');% write The input data
type is hexadecimal
fprintf(fid,'CONTENT BEGIN\n');% starting content


n_temp = [];
for i = 1:cores
```

```
    n_temp = [n_temp flip(de2bi(n,8),2)];
end
n_out = bi2de(flip(n_temp,2));
fprintf(fid,'\t%d:%d;\n',0,n_out);

maxrow_temp = [];
maxrow = ceil(n/cores);
for i = 1:cores
    maxrow_temp = [maxrow_temp flip(de2bi(maxrow,8),2)];
end
maxrow_out = bi2de(flip(maxrow_temp,2));
fprintf(fid,'\t%d:%d;\n',1,maxrow_out);

if mod(n,cores)==0
    I1 = reshape(flatten_A,n^2/cores,cores);
    for i=0:((n^2)/cores)-1
        I2 = flip(de2bi(I1(i+1,:),8),2);
        O = bi2de(flip(reshape(I2',1,cores*8),2));
        fprintf(fid,'\t%d:%d;\n',i+2, O); %print matrix A
    end
else
    count = mod(n, cores);
    I1 = [];
    up_lim = ceil(n/cores);
    down_lim = floor(n/cores);
    index = 0;
    for i=1:cores
        if count ~= 0
            count = count - 1;
            I1 = [I1 flatten_A(index+1 :
index+(n*up_lim))];
            index = index + n*up_lim;
        else
            t = [flatten_A(index+1 : index+(n*down_lim)) ;
zeros(n,1)];
            I1 = [I1 t];
            index = index + n*down_lim;
        end
    end
    for i=0:(n*up_lim)-1
        I2 = flip(de2bi(I1(i+1,:),8),2);
        O = bi2de(flip(reshape(I2',1,cores*8),2));
        fprintf(fid,'\t%d:%d;\n',i+2, O); %print matrix A
    end
end

B_pattern = de2bi(flatten_B,8);
```

```
temp = [];
for i = 1 : cores
    temp = [temp B_pattern];
end
out_B = bi2de(temp);

for i=0:(n^2)-1
    fprintf(fid,'\t%d:%d;\n',i+2+maximum_mem_location,
out_B(i+1));
end

% for i=0:(n^2)-1
%     fprintf(fid,'\t%d:%d;\n',i+2*(n^2),0);
% end

fprintf(fid,'END;\n');
fclose(fid);%Close the file
tic
matrix_O = matrix_A*matrix_B;
toc
matrix_O
transposed_matrix_O = matrix_O.';  %transpose
flatten_O = transposed_matrix_O(:);
flatten_O;

fid=fopen( 'C:/Users/Vinith
Kugathasan/OneDrive/Desktop/out_matrix.txt','w');
for i=0:(n^2)-1
    fprintf(fid,'\t%d:%d;\n',i, matrix_O(i+1));
end
fclose(fid);%Close the file
```

2. Python code used for verification of FPGA processor output with MATLAB output

```
from os import makedirs
import math

cores = 6;

f = open("C:/Users/Vinith Kugathasan/OneDrive/Desktop/Multi-
core - Edited/Single_Core/simulation/modelsim/output_data.mem", "r")

data = (f.readlines())
data = data[3:]

for i in range (0, len(data)):
```

```
data[i] = data[i].split(" ")


data_new=[]
for i in range(0,len(data)):
row_new=[]
for j in range(0,len(data[i])):
if (data[i][j]!=''):
if '\n' in data[i][j]:
data[i][j]=data[i][j][0:-1]
if ':' in data[i][j]:
continue
row_new.append(data[i][j])
data_new = data_new + row_new

data = (data_new[0:-1])

#print(data)
matrix_size = (bin(int(data[0])).replace("0b", "")).zfill(8*cores)
column_size = (bin(int(data[3])).replace("0b", "")).zfill(8*cores)

matrix_size = int(matrix_size[0:8],2)
column_size = int(column_size[0:8],2)
#matrix_size += matrix_size%cores
#print(matrix_size)

# matrix_A = data[2:258]
# matrix_B = data[258:514]
matrix_O = data[516:]
#print(matrix_O)

row_size = cores*math.ceil(matrix_size/cores)
#print(row_size)


m = int(row_size*column_size*2/cores)
#print(m)
values=[]
#print(m)
for i in range(0,m,2):
k=(bin(int(matrix_O[i])).replace("0b", "")).zfill(8*cores)
j=(bin(int(matrix_O[i+1])).replace("0b", "")).zfill(8*cores)
#print(k,j)

for l in range(0,cores):
num = j[8*l:8*(l+1)]+ k[8*l:8*(l+1)]
values.append(int(num,2))

#print(values)
```

```
import numpy as np
matrix_list = np.zeros((row_size,column_size))

for j in range(0,len(values)):
matrix_list[j%row_size, j//row_size] = values[j]

#print(matrix_list)
process_per_core = math.ceil(matrix_size/cores)
idle_cores = process_per_core*cores - matrix_size
temp_matrix = []
out_matrix = []
steps = int(row_size*column_size/process_per_core)
#print(steps)

for i in range(0,process_per_core):
temp_matrix.append(values[i*steps:(i+1)*steps])

for i in range (0, len(temp_matrix[0])):
for j in range (0, len(temp_matrix)):
out_matrix.append(temp_matrix[j][i])

#print(out_matrix)

for j in range(0,len(out_matrix)):
matrix_list[j%row_size, j//row_size] = out_matrix[j]

max_rows = math.ceil(matrix_size/cores)
min_rows = math.floor(matrix_size/cores)




final_matrix = np.zeros((matrix_size,column_size))
final_matrix[:max_rows,:] = matrix_list[:max_rows,:]
for i in range (0,cores-1-idle_cores):
final_matrix[max_rows*(i+1):max_rows*(i+2),:] = matrix_list[max_rows*(i+1):max
_rows*(i+2),:]
for i in range (0,idle_cores):
final_matrix[(cores-idle_cores)*max_rows+(i*min_rows):(cores-
idle_cores)*max_rows+((i+1)*min_rows),:] = matrix_list[(cores-
idle_cores)*max_rows+(i*max_rows):(cores-
idle_cores)*max_rows+(i*max_rows)+min_rows,:]


print ("Expected Matrix :")
f = open("C:/Users/Vinith Kugathasan/OneDrive/Desktop/out_matrix.txt", "r")
data = (f.readlines())
```

```
real_out = np.zeros((matrix_size,column_size))
for i in range (0, len(data)):
data[i] = data[i].split(" ")
for i in range (0, len(data)):
data[i] = data[i][0].split(":")[1]
data[i] = data[i].split(";")[0]
real_out[i%matrix_size, i//matrix_size] = int(data[i])
print(real_out)

print ("Calculated Matrix :")
print(final_matrix)

comparison_matrix = np.zeros((matrix_size,column_size))

print ("Comparison :")
for i in range(matrix_size):
for j in range(column_size):
if real_out[i][j]==final_matrix[i][j]:
comparison_matrix[i][j] = 1
else:
comparison_matrix[i][j] = 0

print(comparison_matrix)
print(final_matrix.shape)
```

3. Verilog code for Matrix Multiplication

A. Multi Core Processor

```
module multi_core_processor
#(parameter cores = 6)
(
    input clk,
    input status,
    input rst,
    output end_process

);

wire [(8*cores)-1:0] dram_out; //6cores (maximum)
wire [7:0] iram_out;

wire [7:0] pc_out;
wire [15:0] ar_out;

wire [(8*cores)-1:0] bus_out;  //6cores

wire dram_wrEn;
```

```
wire read_IRAM;


core core1(.clk(clk), .dram_out(dram_out[(8*cores)-1:8*(cores-
1)]), .iram_out(iram_out), .status(status), .rst(rst),
.dram_wrEn(dram_wrEn), .read_IRAM(read_IRAM), .pc_out(pc_out),
.ar_out(ar_out), .bus_out(bus_out[(8*cores)-1:8*(cores-
1)]), .end_process(end_process));

genvar i;
generate
    for (i=0; i<=cores-2; i=i+1) begin : processor
    core coren(.clk(clk), .dram_out(dram_out[(i+1)*8-
1:i*8]), .iram_out(iram_out), .status(status), .rst(rst),
 .bus_out(bus_out[(i+1)*8-1:i*8])
    );
end
endgenerate

instr_memory IRAM(.clk(clk), .read_IRAM(read_IRAM), .addr(pc_out), .instr_o
ut(iram_out));
DRAM DRAM(.clock(clk), .address(ar_out), .data(bus_out), .wren(dram_wrEn),.
q(dram_out));

endmodule
```

B. Core

```
module core

(
    input clk,
    input [7:0] dram_out,
    input [7:0] iram_out,
    input status,
    input rst,
    output reg dram_wrEn,
    //output read_DRAM,
    output read_IRAM,
    output [7:0] pc_out,
    output [15:0] ar_out,
    output [15:0] bus_out,
    output end_process
);

wire [2:0] alu_op;
wire [18:0] write_en ;  //19bits
wire [4:0] read_en ;  //5bits
wire [1:0] inc_en;
```

```
//wire read_IRAM;
//wire read_DRAM;
//wire end_process;



wire [15:0] alu_out;
//wire [15:0] bus_out;
wire [15:0] r_out ;
wire [15:0] ac_out ;
wire [15:0] ra_out ;
wire [15:0] rb_out ;
wire [15:0] ro_out ;
wire [7:0] rcol1_out ;
wire [7:0] rcol2_out ;
wire [7:0] rn_out ;
wire [7:0] rp_out ;
wire [7:0] rc_out ;
wire [7:0] rr_out ;
wire [15:0] rt_out ;
wire [7:0] dr_out ;
wire [7:0] tr_out ;
wire [7:0] ir_out ;


wire z ;

localparam width = 8;


register ar(.dataIn(bus_out), .write_en(write_en[0]), .rst(rst), .clk(clk),
 .dataOut(ar_out));
register #(.width(width)) ir(.dataIn(iram_out), .write_en(write_en[2]), .rs
t(rst), .clk(clk), .dataOut(ir_out));
register #(.width(width)) tr(.dataIn(ir_out), .write_en(write_en[3]), .rst(
rst), .clk(clk), .dataOut(tr_out));
register #(.width(width)) dr(.dataIn(bus_out), .write_en(write_en[4]), .rst
(rst), .clk(clk), .dataOut(dr_out));
register ra(.dataIn(bus_out), .write_en(write_en[5]), .rst(rst), .clk(clk),
 .dataOut(ra_out));
register rb(.dataIn(bus_out), .write_en(write_en[6]), .rst(rst), .clk(clk),
 .dataOut(rb_out));
register ro(.dataIn(bus_out), .write_en(write_en[7]), .rst(rst), .clk(clk),
 .dataOut(ro_out));
register #(.width(width)) rn(.dataIn(bus_out), .write_en(write_en[8]), .rst
(rst), .clk(clk), .dataOut(rn_out));
register #(.width(width)) rcol1(.dataIn(bus_out), .write_en(write_en[17]),
.rst(rst), .clk(clk), .dataOut(rcol1_out));
```

```
register #(.width(width)) rcol2(.dataIn(bus_out), .write_en(write_en[18]),
.rst(rst), .clk(clk), .dataOut(rcol2_out));
register #(.width(width)) rp(.dataIn(bus_out), .write_en(write_en[9]), .rst
(rst), .clk(clk), .dataOut(rp_out));
register #(.width(width)) rc(.dataIn(bus_out), .write_en(write_en[10]), .rs
t(rst), .clk(clk), .dataOut(rc_out));
register #(.width(width)) rr(.dataIn(bus_out), .write_en(write_en[11]), .rs
t(rst), .clk(clk), .dataOut(rr_out));
register rt(.dataIn(bus_out), .write_en(write_en[12]), .rst(rst), .clk(clk)
, .dataOut(rt_out));
register r(.dataIn(ac_out), .write_en(write_en[13]), .rst(rst), .clk(clk),
.dataOut(r_out));


reginc #(.width(width)) pc(.dataIn(ir_out), .write_en(write_en[1]), .inc_en
(inc_en[0]), .rst(rst), .clk(clk), .dataOut(pc_out));
acc ac(.dataIn(bus_out), .aluIn(alu_out), .write_en(write_en[14]), .inc_en(
inc_en[1]), .rst(rst), .clk(clk), .alu_en(write_en[15]), .dataOut(ac_out));

alu alu(.clk(clk), .aluIn1(ac_out), .aluIn2(r_out), .aluOp(alu_op), .aluOut
(alu_out), .z(z));
bus bus(.clk(clk), .read_en(read_en), .ir(ir_out), .tr(tr_out), .dr(dr_out)
, .ra(ra_out), .rb(rb_out), .ro(ro_out), .rcol1(rcol1_out), .rcol2(rcol2_ou
t), .rn(rn_out), .rp(rp_out), .rc(rc_out), .rr(rr_out), .rt(rt_out), .ac(ac
_out), .dram(dram_out), .busIn(bus_out));

control_unit cu(.clk(clk), .z(z), .status(status), .instruction(ir_out), .a
lu_op(alu_op), .write_en(write_en), .inc_en(inc_en), .read_en(read_en), .re
ad_IRAM(read_IRAM), .end_process(end_process));

always@(write_en[16])
begin
dram_wrEn <= write_en[16];
end
endmodule
```

C. Control Unit

```
module control_unit
(
    input clk,
    input z,
    input status,
    input [7:0] instruction,
    output reg [2:0] alu_op,
    output reg [18:0] write_en,
    output reg [1:0] inc_en,
    output reg [4:0] read_en,
    output reg read_IRAM,
    //output reg read_DRAM,
```

```
    output reg end_process
);

reg [5:0] present = 6'd0;      //6'd0 = idle
reg [5:0] next = 6'd0;

//Micro Instructions
parameter
idle = 6'd0,
fetch1 = 6'd1,
fetch2 = 6'd2,
fetch3 = 6'd3,
ldac1 = 6'd4,
ldac2 = 6'd5,
ldac3 = 6'd6,
ldac4 = 6'd7,
ldacm1 = 6'd8,
ldacm2 = 6'd9,
ldacm3 = 6'd10,
stac1 = 6'd11,
stac2 = 6'd12,
stac3 = 6'd13,
stac4 = 6'd14,
stac5 = 6'd15,
stac6 = 6'd16,
stac7 = 6'd17,
stac8 = 6'd18,
clac = 6'd19,
movr = 6'd20,
movra = 6'd21,
movrb = 6'd22,
movro = 6'd23,
movrn = 6'd24,
movrp = 6'd25,
movrc = 6'd26,
movrr = 6'd27,
movrt = 6'd28,
mvacra = 6'd29,
mvacrb = 6'd30,
mvacro = 6'd31,
mvacrn = 6'd32,
mvacrp = 6'd33,
mvacrc = 6'd34,
mvacrr = 6'd35,
mvacrt = 6'd36,
add = 6'd37,
mul = 6'd38,
sub = 6'd39,
inc = 6'd40,
```

```
dec = 6'd41,
jpnz = 6'd42,
jpnzy1 = 6'd43,
jpnzy2 = 6'd44,
jpnzn1 = 6'd45,
endop = 6'd46,
movrcol1 = 6'd47,
mvacrcol1 = 6'd48,
movrcol2 = 6'd49,
mvacrcol2 = 6'd50;
//nop = 6'd48;


//checking the termination condition
always @(posedge clk)
begin
    present <= next;
    if (present == endop)
        end_process <= 1'd1;
    else
        end_process <= 1'd0;
end


always @(present or z or instruction or status)
case(present)
    idle: begin
        read_IRAM <= 0;
        read_en <= 5'd0;
        write_en <= 19'b0000000000000000000 ;
        inc_en <= 2'b00 ;
        alu_op <= 3'd0;
        if (status == 1)
        next <= fetch1;
        else
        next <= idle;
    end

    endop: begin
        read_IRAM <= 0;
        read_en <= 5'd0;
        write_en <= 19'b0000000000000000000 ;
        inc_en <= 2'b00 ;
        alu_op <= 3'd0;
        next <= endop;
    end

    fetch1: begin
        read_IRAM <= 1;  //READ IRAM
```

```
    read_en <= 5'd0;
    write_en <= 19'b0000000000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch2;
end


fetch2: begin
    read_IRAM <= 0;
    read_en <= 5'd0;
    write_en <= 19'b0000000000000000100 ; // IR <- IRAM
    inc_en <= 2'b01 ; //pc<-pc+1
    alu_op <= 3'd0;
    next <= fetch3;
end


fetch3: begin
    read_IRAM <= 1;    //Read IRAM
    read_en <= 5'd0;
    write_en <= 19'b0000000000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= instruction[5:0];  //get assembly code
end

ldac1: begin
    read_IRAM <= 0;
    read_en <= 5'd0;
    write_en <= 19'b0000000000000000100 ; //IR <- IRAM
    inc_en <= 2'b01 ; //pc<-pc+1
    alu_op <= 3'd0;
    next <= ldac2;
end

ldac2: begin
    read_IRAM <= 1; //Read
    read_en <= 5'd1; //BUS <- IR
    write_en <= 19'b0000000000000001000 ;  //TR <-- BUS
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= ldac3;
end

ldac3: begin
    read_IRAM <= 0;
    read_en <= 5'd0;
    write_en <= 19'b0000000000000000100 ; //IR <- IRAM
```

```
            inc_en <= 2'b01 ; //pc<-pc+1
            alu_op <= 3'd0;
            next <= ldac4;
        end

        ldac4: begin
            read_IRAM <= 0;
            read_en <= 5'd14;   //AC <- IR TR
            write_en <= 19'b0000100000000000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end

        ldacm1: begin
            read_IRAM <= 0;
            //read_DRAM <= 1;
            read_en <= 5'd12;  //BUS <- AC
            write_en <= 19'b0000000000000000001 ;  //AR <- BUS
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= ldacm2;
        end

         ldacm2 : begin
    //        read_IRAM <= 0;
    //         read_en <= 4'd0;
    //         write_en <= 17'b00000000000000000 ;  //waste a clock cycle
    //         inc_en <= 2'b00 ;
    //         alu_op <= 3'd0;
            next <= ldacm3;
         end

        ldacm3: begin
            read_IRAM <= 0;
            read_en <= 5'd13;  // BUS <- DRAM
            write_en <= 19'b0000100000000010000 ; //DR, AC ←  DRAM[AR],
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end



        stac1: begin
            read_IRAM <= 0;
            read_en <= 5'd12;  //BUS <- AC
            write_en <= 19'b0000000000000000001 ; // AR <- BUS
            inc_en <= 2'b00 ;
```

```
    alu_op <= 3'd0;
    next <= stac2;
end

stac2: begin
    read_IRAM <= 0;
    read_en <= 5'd11;  //BUS <- RT
    write_en <= 19'b0000100000000000000 ; //AC <- BUS
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= stac3;
end

stac3: begin
    read_IRAM <= 0;
    read_en <= 5'd12;  //BUS <- AC[7:0]  check 12
    write_en <= 19'b0000000000000010000 ; //DR <- BUS
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= stac4;
end



stac4: begin
    read_IRAM <= 0;
    read_en <= 5'd3;  //BUS <- DR
    write_en <= 19'b0010000000000000000 ; //DRAM <- BUS
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= stac5;
end


stac5: begin
    read_IRAM <= 0;
    read_en <= 5'd6;  //BUS <- RO
    write_en <= 19'b0000100000000000000 ; //AC<- BUS
    inc_en <= 2'b10 ; //AC <- AC+1
    alu_op <= 3'd0;
    next <= stac6;
end

stac6: begin
    read_IRAM <= 0;
    read_en <= 5'd12;  //BUS <- AC
    write_en <= 19'b0000000000000000001 ; // AR <- BUS
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
```

```
        next <= stac7;
    end

    stac7: begin
        read_IRAM <= 0;
        read_en <= 5'd11;  //BUS <-RT
        write_en <= 19'b0000100000000000000 ; // AC <-- BUS
        inc_en <= 2'b00;
        alu_op <= 3'd0;
        next <= stac8;
    end

    stac8: begin
        read_IRAM <= 0;
        read_en <= 5'd15;  //BUS<-AC[15:8]   Check with dhinesh
        write_en <= 19'b0010000000000010000 ; // DR <-- BUS, DRAM <-- BUS
        inc_en <= 2'b00 ;
        alu_op <= 3'd0;
        next <= fetch1;
    end


    clac: begin
        read_IRAM <= 0;
        read_en <= 5'd0;
        write_en <= 19'b0001000000000000000 ;
        inc_en <= 2'b00 ;
        alu_op <= 3'd5; //default AC <- 0, Z<- 1
        next <= fetch1;
    end

    movr: begin
        read_IRAM <= 0;
        read_en <= 5'd12;  //BUS <- AC
        write_en <= 19'b0000010000000000000 ; // R <- BUS
        inc_en <= 2'b00 ;
        alu_op <= 3'd0;
        next <= fetch1;
    end

    movra: begin
        read_IRAM <= 0;
        read_en <= 5'd12;  //BUS <- AC
        write_en <= 19'b0000000000000100000 ; // RA <- BUS
        inc_en <= 2'b00 ;
        alu_op <= 3'd0;
        next <= fetch1;
    end
```

```
        movrcol1: begin
            read_IRAM <= 0;
            read_en <= 5'd12;  //BUS <- AC
            write_en <= 19'b0100000000000000000 ; // RCOL <- BUS
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end

        movrcol2: begin
            read_IRAM <= 0;
            read_en <= 5'd12;  //BUS <- AC
            write_en <= 19'b1000000000000000000 ; // RCOL <- BUS
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end

        movrb: begin
            read_IRAM <= 0;
            read_en <= 5'd12;  //BUS <- AC
            write_en <= 19'b0000000000001000000 ; // RB <- BUS
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end

        movro: begin
            read_IRAM <= 0;
            read_en <= 5'd12;                       //RO <- AC
            write_en <= 19'b0000000000010000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end

        movrn: begin
            read_IRAM <= 0;
            read_en <= 5'd12;                       //RN <- AC
            write_en <= 19'b0000000000100000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end
    //
        movrp: begin
            read_IRAM <= 0;
            read_en <= 5'd12;                     //RP <- AC
            write_en <= 19'b0000000001000000000 ;
```

```
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

movrc: begin
    read_IRAM <= 0;
    read_en <= 5'd12;                    //RC <- AC
    write_en <= 19'b0000000010000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

movrr: begin
    read_IRAM <= 0;
    read_en <= 5'd12;                    //RR <- AC
    write_en <= 19'b0000000100000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

movrt: begin
    read_IRAM <= 0;
    read_en <= 5'd12;                   //RT <- AC
    write_en <= 19'b0000001000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

mvacra: begin
    read_IRAM <= 0;
    read_en <= 5'd4;             //AC <- RA
    write_en <= 19'b0000100000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

 mvacrcol1: begin
    read_IRAM <= 0;
    read_en <= 5'd16;             //AC <- RCOL
    write_en <= 19'b0000100000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end
```

```
 mvacrcol2: begin
    read_IRAM <= 0;
    read_en <= 5'd17;            //AC <- RCOL
    write_en <= 19'b0000100000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
 end

mvacrb: begin
    read_IRAM <= 0;
    read_en <= 5'd5;        //AC <- RB
    write_en <= 19'b0000100000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

mvacro: begin
    read_IRAM <= 0;
    read_en <= 5'd6;  //AC <- RO
    write_en <= 19'b0000100000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

mvacrn: begin
    read_IRAM <= 0;
    read_en <= 5'd7;  //AC <- RN
    write_en <= 19'b0000100000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

mvacrp: begin
    read_IRAM <= 0;
    read_en <= 5'd8;                //AC <- RP
    write_en <= 19'b0000100000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= fetch1;
end

mvacrc: begin
    read_IRAM <= 0;
    read_en <= 5'd9;                //AC <- RC
```

```
            write_en <= 19'b0000100000000000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end

        mvacrr: begin
            read_IRAM <= 0;
            read_en <= 5'd10;               //AC <- RR
            write_en <= 19'b0000100000000000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end

        mvacrt: begin
            read_IRAM <= 0;
            read_en <= 5'd11;                   //AC <- RT
            write_en <= 19'b0000100000000000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd0;
            next <= fetch1;
        end
    //
        add: begin
            read_IRAM <= 0;
            read_en <= 5'd0;
            write_en <= 19'b0001000000000000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd1; //AC <- AC + R
            next <= fetch1;
        end

        mul: begin
            read_IRAM <= 0;
            read_en <= 5'd0;
            write_en <= 19'b0001000000000000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd3; //AC <- AC * R
            next <= fetch1;
        end

        sub: begin
            read_IRAM <= 0;
            read_en <= 5'd0;
            write_en <= 19'b0001000000000000000 ;
            inc_en <= 2'b00 ;
            alu_op <= 3'd2; //AC <- AC - R
            next <= fetch1;
```

```
end

inc: begin
    read_IRAM <= 0;
    read_en <= 5'd0;
    write_en <= 19'b0000000000000000000 ;//
    inc_en <= 2'b10 ; //AC <- AC + 1
    alu_op <= 3'd0;
    next <= fetch1;
end

dec: begin
    read_IRAM <= 0;
    read_en <= 5'd0;
    write_en <= 19'b0001000000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd4; //AC <- AC - 1
    next <= fetch1;
end


jpnz: begin
    read_IRAM <= 0;
    read_en <= 5'd0;  //IR <-IRAM
    write_en <= 19'b0000000000000000000 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    if (z==0)
        next <= jpnzy1;
    else
        next <= jpnzn1;
end


jpnzy1: begin
    read_IRAM <= 0;
    read_en <= 5'd0;  //IR <- IRAM
    write_en <= 19'b0000000000000000100 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;
    next <= jpnzy2;
end

jpnzy2: begin
    read_IRAM <= 0;
    read_en <= 5'd1;  //PC <- IR
    write_en <= 19'b0000000000000000010 ;
    inc_en <= 2'b00 ;
    alu_op <= 3'd0;  //3'd2
```

```
        next <= fetch1;
    end

    jpnzn1: begin
        read_IRAM <= 0;
        read_en <= 5'd0;
        write_en <= 19'b0000000000000000000 ;
        inc_en <= 2'b01 ;  //PC<-PC+1
        alu_op <= 3'd0;
        next <= fetch1;
    end

endcase
endmodule
```

D. Arithmetic Logic Unit

```
module alu
(
    input clk,
    input [15:0] aluIn1, // aluIn1 => AC
    input [15:0] aluIn2,
    input [2:0] aluOp,
    output reg [15:0] aluOut,
    output reg z
);

always @(aluOp)//posedge clk)//aluOp)
begin
    case(aluOp)
        3'd1: aluOut <= aluIn1 + aluIn2; // AC + R
        3'd2: aluOut <= aluIn1 - aluIn2; // AC - R
        3'd3:   aluOut <= aluIn1 * aluIn2; // AC x R
        3'd4:
        begin
        aluOut = aluIn1 - 16'b1;            // AC - 1
        if (aluOut==0)
            z <= 1'b1;
        else
            z <= 1'b0;
        end
        3'd5:
        begin
        aluOut <= 16'b0;           // Case 0 -> 0
        z <= 1'b1;
        end
        //default : aluOut <= 16'b0;
    endcase
```

```
end
endmodule
```

E. Instruction Memory (IRAM)

```
module instr_memory(input clk,
input read_IRAM,
input [7:0] addr,//256 locations
output reg [7:0] instr_out);

reg [7:0] ram [102:0];
parameter idle = 8'd0;
parameter ldac = 8'd4;
parameter ldacm = 8'd8;
parameter stac = 8'd11;
parameter clac = 8'd19;
parameter movr = 8'd20;
parameter movra = 8'd21;
parameter movrb = 8'd22;
parameter movro = 8'd23;
parameter movrn = 8'd24;
parameter movrp = 8'd25;
parameter movrc = 8'd26;
parameter movrr = 8'd27;
parameter movrt = 8'd28;
parameter mvacra = 8'd29;
parameter mvacrb = 8'd30;
parameter mvacro = 8'd31;
parameter mvacrn = 8'd32;
parameter mvacrp = 8'd33;
parameter mvacrc = 8'd34;
parameter mvacrr = 8'd35;
parameter mvacrt = 8'd36;
parameter add = 8'd37;
parameter mul = 8'd38;
parameter sub = 8'd39;
parameter inc = 8'd40;
parameter dec = 8'd41;
parameter jpnz = 8'd42;
parameter endop = 8'd46;
parameter movrcol1 = 6'd47;
parameter mvacrcol1 = 6'd48;
parameter movrcol2 = 6'd49;
parameter mvacrcol2 = 6'd50;


initial begin
ram[0] = clac;
```

```
ram[1] = ldac;
ram[2] = 8'd0;  //load data memory location of matrix size
ram[3] = 8'd0;
ram[4] = ldacm;
ram[5] = movrn;
ram[6] = movrp;
ram[7] = ldac;
ram[8] = 8'd1;
ram[9] = 8'd0;
ram[10] = ldacm;
ram[11] = movrr;
ram[12] = ldac;
ram[13] = 8'd2;
ram[14] = 8'd0;
ram[15] = ldacm;
ram[16] = movrcol1;
ram[17] = ldac;
ram[18] = 8'd3;
ram[19] = 8'd0;
ram[20] = ldacm;
ram[21] = movrcol2;
ram[22] = ldac;
ram[23] = 8'd4;
ram[24] = 8'd0;
ram[25] = movra;
ram[26] = ldac;
ram[27] = 8'd4;//lsb
ram[28] = 8'd1;
ram[29] = movrb;
ram[30] = ldac;
ram[31] = 8'd4;//lsb
ram[32] = 8'd2;
ram[33] = movro;
ram[34] = mvacrcol2;
ram[35] = movrc;
ram[36] = clac;
ram[37] = movrt;
ram[38] = mvacrcol1;
ram[39] = movrp;
ram[40] = mvacra;
ram[41] = ldacm;
ram[42] = movr;
ram[43] = mvacrb;
ram[44] = ldacm;
ram[45] = mul;
ram[46] = movr;
ram[47] = mvacrt;
ram[48] = add;
ram[49] = movrt;
```

```
ram[50] = mvacra;
ram[51] = inc;
ram[52] = movra;
ram[53] = mvacrcol2;
ram[54] = movr;
ram[55] = mvacrb;
ram[56] = add;
ram[57] = movrb;
ram[58] = mvacrp;
ram[59] = dec;
ram[60] = movrp;
ram[61] = jpnz;
ram[62] = 8'd40;   // rechk
ram[63] = mvacro;
ram[64] = stac;
ram[65] = mvacro;
ram[66] = inc;
ram[67] = inc;
ram[68] = movro;
ram[69] = mvacrcol1;
ram[70] = movr;
ram[71] = mvacra;
ram[72] = sub;
ram[73] = movra;
ram[74] = mvacrcol1;
ram[75] = movr;
ram[76] = mvacrcol2;
ram[77] = mul;
ram[78] = movr;
ram[79] = mvacrb;
ram[80] = sub;
ram[81] = inc;
ram[82] = movrb;
ram[83] = mvacrc;
ram[84] = dec;
ram[85] = movrc;
ram[86] = jpnz;///
ram[87] = 8'd36;
ram[88] = mvacrcol1;
ram[89] = movr;
ram[90] = mvacra;
ram[91] = add;
ram[92] = movra;
ram[93] = ldac;
ram[94] = 8'd4;
ram[95] = 8'd1;
ram[96] = movrb;
ram[97] = mvacrr;
ram[98] = dec;
```

```
    ram[99] = movrr;
    ram[100] = jpnz;
    ram[101] = 8'd34;
    ram[102] = endop;
    end

    always @(posedge clk)
    begin
    if (read_IRAM == 1)
    instr_out <= ram[addr];
    end
    endmodule
```

F. Data Memory (DRAM)

```
`timescale 1 ps / 1 ps
// synopsys translate_on
module DRAM (
    address,
    clock,
    data,
    wren,
    q);

    input   [10:0]  address;
    input     clock;
    input   [47:0]  data;
    input     wren;
    output  [47:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [47:0] sub_wire0;
    wire [47:0] q = sub_wire0[47:0];

    altsyncram  altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .data_a (data),
                .wren_a (wren),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
```

```
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_b (1'b0));
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "data.mif",
        altsyncram_component.intended_device_family = "Cyclone IV E",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=
DRAM",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 2048,
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M9K",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_
READ",
        altsyncram_component.widthad_a = 11,
        altsyncram_component.width_a = 48,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

G. BUS

```
module bus
(
    input clk,
    input [4:0] read_en,    //read enable
    input [7:0] ir,
    input [7:0] tr,
    input [7:0] dr,
    input [15:0] ra,
```

```
    input [15:0] rb,
    input [15:0] ro,
    input [7:0] rcol1,
    input [7:0] rcol2,
    input [7:0] rn,
    input [7:0] rp,
    input [7:0] rc,
    input [7:0] rr,
    input [15:0] rt,
    input [15:0] ac,
    input [7:0] dram,
    output reg [15:0] busIn
    ) ;

always @(read_en or ir or tr or dr or ra or rb or ro or rn or rp or rc or r
r or rt or ac or dram or rcol1 or rcol2)

begin
case(read_en)
    5'd1: busIn <= ir;
    5'd2: busIn <= tr;
    5'd3: busIn <= dr;
    5'd4: busIn <= ra;
    5'd5: busIn <= rb;
    5'd6: busIn <= ro;
    5'd7: busIn <= rn;
    5'd8: busIn <= rp;
    5'd9: busIn <= rc;
    5'd10: busIn <= rr;
    5'd11: busIn <= rt;
    5'd12: busIn <= ac;
    5'd13: busIn <= dram;
    5'd14: busIn <= {ir,tr};
    5'd15: busIn <= ac[15:8];
    5'd16: busIn <= rcol1;
    5'd17: busIn <= rcol2;
    //default: busIn <=16'd0;
endcase
end
endmodule
```

H. Accumulator

```
module acc
(
    input [15:0] dataIn,
    input [15:0] aluIn,
    input write_en,
    input inc_en,
```

```
        input rst,
        input clk,
        input alu_en,
        output reg [15:0] dataOut

    );

    always @(posedge clk or negedge rst)
    begin
        if(~rst)
        begin
            dataOut <= 16'd0;
        end

        else if(write_en && ~alu_en)
        begin
        dataOut = dataIn;
        if(inc_en)
            dataOut = dataOut + 16'd1;
        end

        else if(alu_en && ~write_en)
        begin
        dataOut = aluIn;
        if(inc_en)
            dataOut = dataOut + 16'd1;
        end

        else if(inc_en && ~write_en && ~alu_en)
        begin
        dataOut <= dataOut + 16'd1;
        end

    end
    endmodule
```

I. PC (Register with increment)

```
module reginc
#(parameter width = 16) //default size
(
     input [width-1:0] dataIn,
     input write_en,
     input inc_en,
     input rst,
     input clk,
     output reg [width-1:0] dataOut

);
```

```
always @(posedge clk or negedge rst)
begin
    if(~rst)
    begin
        dataOut <= 0;
    end

    else
    begin
    if(write_en)
        dataOut = dataIn;
    if(inc_en)
        dataOut = dataOut + 1;
    end
end
endmodule
```

J. Register without increment

```
module register
#(parameter width = 16) //default size
(
    input [width-1:0] dataIn,
    input write_en,
    input rst,
    input clk,
    output reg [width-1:0] dataOut
);


always @(posedge clk or negedge rst)
begin
    if(~rst)
        dataOut <= 0;
    else if (write_en)
        dataOut <= dataIn;
end


endmodule
```