Reconfigurable Computing Project

Design of Math Co-Processor

Done by:

Trishnendu Ghorai (Gy-31)

Exam Roll: 510514018

Deepak Sharma (Gx-21)

Exam Roll : 510514048

Table of Content

System Architecture

Memory:-

The memory is word addressable. The size of data bus is 8 bits i.e. the memory consist of 2^8 or 256 memory word. Each memory word is of 16 bits or 2 bytes size. So the total size of memory is 2 * (2^8) = 512 bytes.

There is two type of memory in the system.

i) ROM emulated by file

ii) Working Memory emulated by register array

The compiler writes the instructions in a file and this file is used as the ROM file. At the starting of the machine what the system first does is to readback the ROM file and load it in Working Memory. And at the termination when the system halts the last thing it does is to writeback the content of working memory in the ROM file. This two operations are controlled by the READ_BACK and WRITE_BACK signal in Memory Unit (MEMU).

Register:-

The system uses total 16 registers. Each register is of same size as of Memory word i.e. 2 byte. Each register is addressable by the register number (0~F).

16 bit Arithmetic:-

The Arithmetic Unit can perform operations on 16 bit operands. It handles negative number using sign bit.

Instruction:-

The instruction set used is a 3 address instruction i.e. our system is a 3 address machine.

<Opcode> <Dest reg> <Src reg1> <Src reg2>

Addressing Mode:-

The system supports 2 addressing mode -. Register direct and Immediate addressing Mode.

OPCODE:-

We have opted for 4 bit opcode. Thus there can only be 16 different instructions the AU can perform. The first 8 instructions are for memory operations and last 8 instructions are for arithmetic operations. All Register Mode Instructions except first 4 instruction has LSB 0 where as corresponding Immediate Mode instruction has LSB 1.

Opcode MSB	Instruction type
0	Memory Operation
1	Arithmetic Operation

Opcode LSB (except first 4)	Instruction Mode
0	Register Direct
1	Immediate

The Opcodes are listed below :-

Opcode (in hex)	**Instruction**	**Operation**
0	NOP	Stall
1	HLT	Halt machine
2	LDPC	Load Program Counter
3	MV	Move (register to register)
4	LD	Load (register from memory)
5	LDI	Load Immediate
6	ST	Store (register to memory)
7	STI	Store Immediate
8	ADD	Addition
9	ADDI	Add Immediate
A	SUB	Substract
B	SUBI	Sub Immediate
C	MULT	Multiplication
D	MULTI	Mult Immediate
E	DIV	Division
F	DIVI	Division Immediate

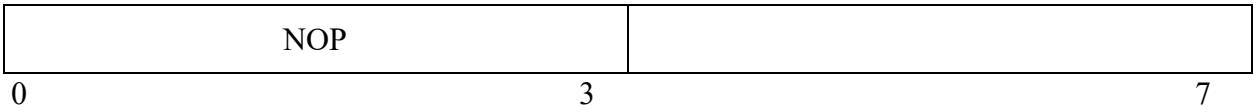
Table 1:- Opcode Table

Design of Math Co-Processor

Opcode - 0

Mnemonic - **NOP**

Size - 1 byte

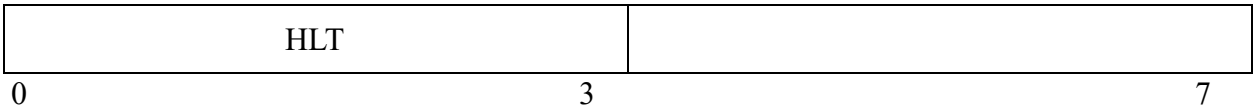


Explanation - Stalls the CPU

Opcode - 1

Mnemonic - **HLT**

Size - 1 byte

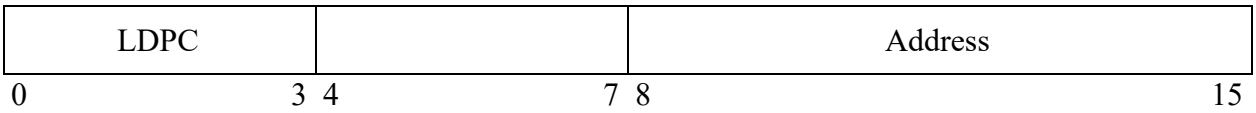


Explanation - Halts the CPU

Opcode - 2

Mnemonic - **LDPC <Address>**

Size - 2 byte

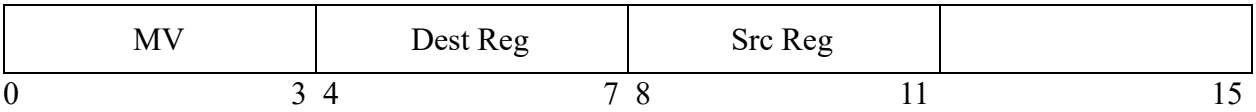


Explanation - Load Program Counter with Specific Address. This can be used to set starting address of the instructions.

Opcode - 3

Mnemonic - **MV <Dest Reg> <Src Reg>**

Size - 2 byte

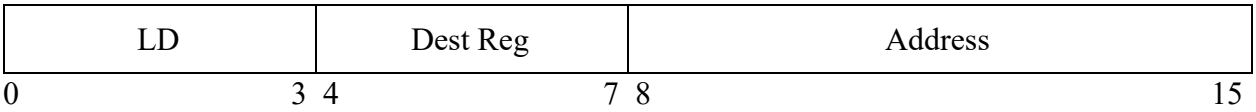


Explanation - Copies the value of Destination Register to Source Register

Opcode - 4

Mnemonic - **LD <Dest Reg> <Address>**

Size - 2 byte

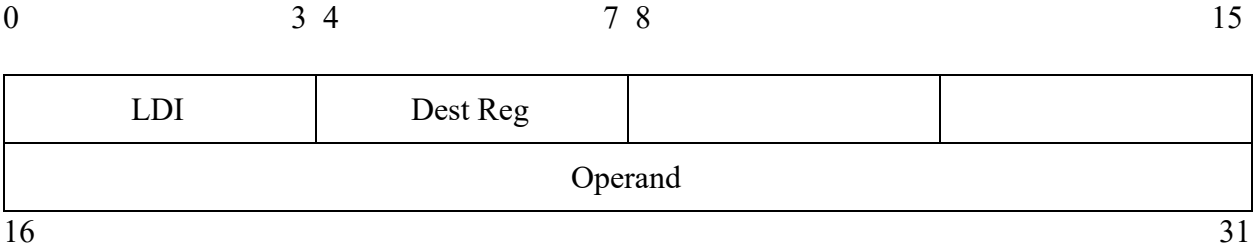


Explanation - Loads the Destination Register with the content of stored at the specified Address

Opcode - 5

Mnemonic - **LDI <Dest Reg> <Operand>**

Size - 4 byte

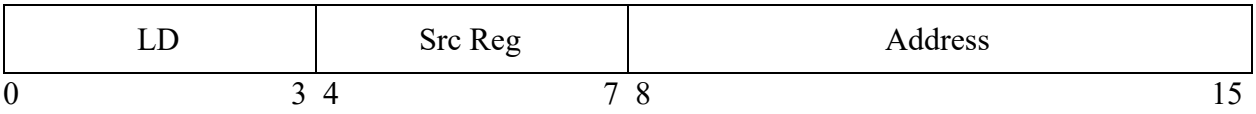


Explanation - Loads the Destination Register with the Operand value specified in the instruction

Opcode - 6

Mnemonic - **ST <Src Reg> <Address>**

Size - 2 byte

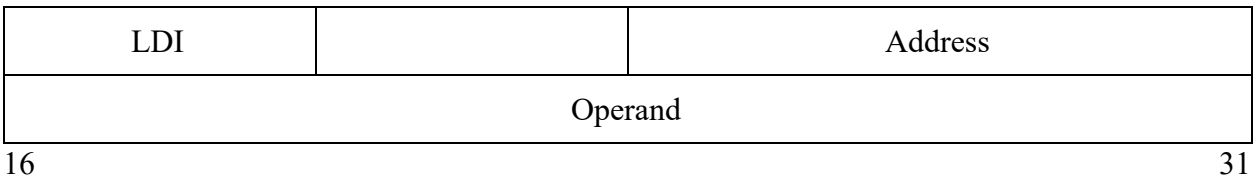


Explanation - Stores the Source Register Content to the memory address

Opcode - 7

Mnemonic - **STI <Dest Reg> <Operand>**

Size - 4 byte

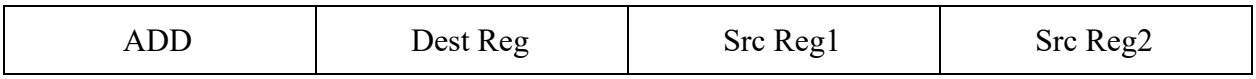


Explanation - Stores the Operand specified in the instruction the specified address

Opcode - 8

Mnemonic - **ADD <Dest Reg> <Src Reg1> <Src Reg2>**

Size - 2 byte



0 3 4 7 8 11 15

Explanation - Adds contents of Source Registers and stores them in Destination Register

Opcode - 9

Mnemonic - **ADDI <Dest Reg> <Src Reg> <Operand>**

Size - 4 byte

0 3 4 7 8 15

LDI	Dest Reg	Src Reg	
Operand			

16 31

Explanation - Adds the content of Source register and Operand specified in the instruction and stores in the Destination Register

Opcode - A

Mnemonic - **SUB <Dest Reg> <Src Reg1> <Src Reg2>**

Size - 2 byte

SUB	Dest Reg	Src Reg1	Src Reg2

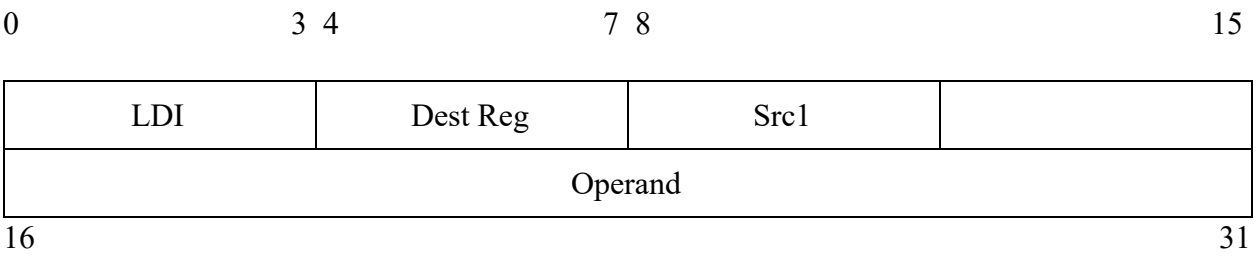
0 3 4 7 8 11 15

Explanation - Subtracts Source Register2 from Source Register1 and stores result in Destination Register

Opcode - B

Mnemonic - **SUBI <Dest Reg> <Src Reg> <Operand>**

Size - 4 byte

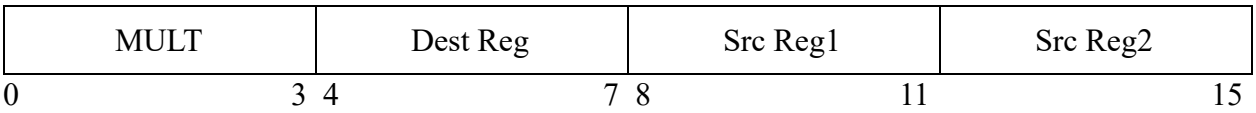


Explanation - Subtracts the content of Operand from Source register1 and specified in the instruction and stores in the Destination Register

Opcode - C

Mnemonic - **MULT <Dest Reg> <Src Reg1> <Src Reg2>**

Size - 2 byte

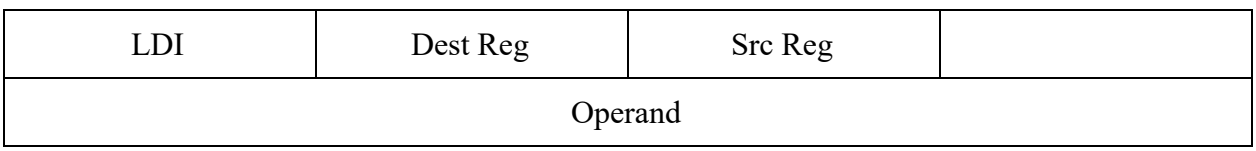


Explanation - Multiplies contents of Source Registers and stores result in Destination Register

Opcode - D

Mnemonic - **MULTI <Dest Reg> <Src Reg> <Operand>**

Size - 4 byte



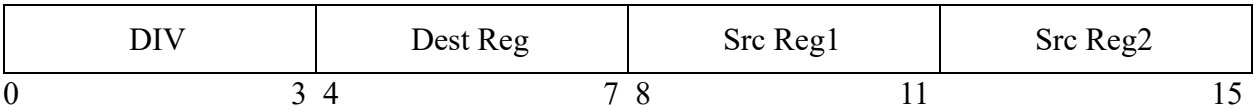
16 31

Explanation - Multiplies the content of Source register and Operand specified in the instruction and stores in the Destination Register

Opcode - E

Mnemonic - **DIV <Dest Reg> <Src Reg1> <Src Reg2>**

Size - 2 byte

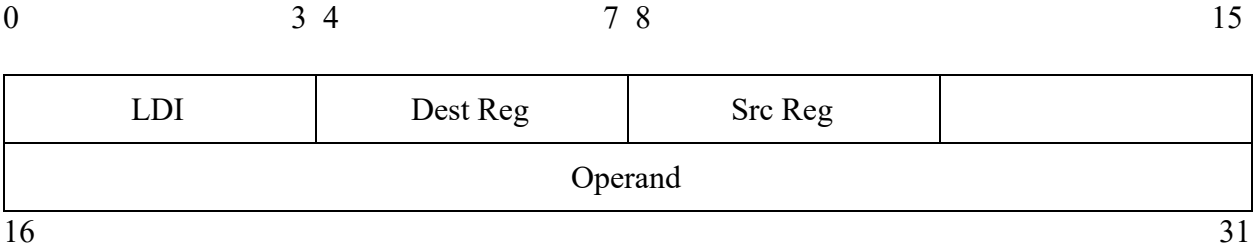


Explanation - Divides contents of Source Register1 with Source Register2 and stores result in Destination Register

Opcode - F

Mnemonic - **DIVI <Dest Reg> <Operand>**

Size - 4 byte



Explanation - Divides the content of Source register1 with Operand specified in the instruction and stores in the Destination Register

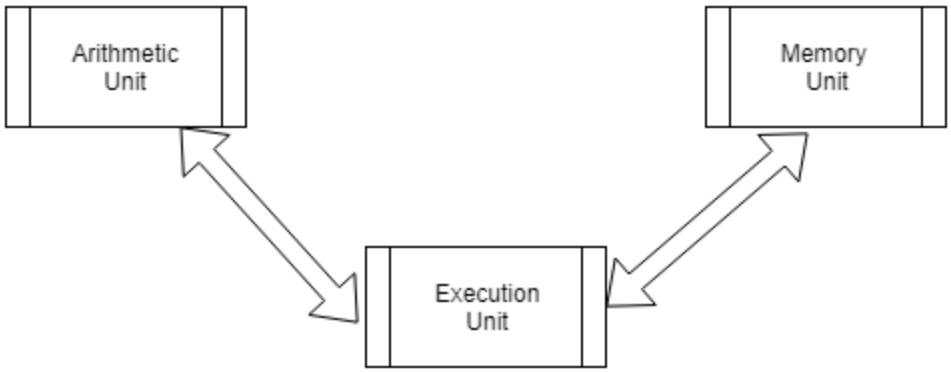
Implementation of System

In our system we have total 3 modules.

i) Arithmetic Unit (AU)

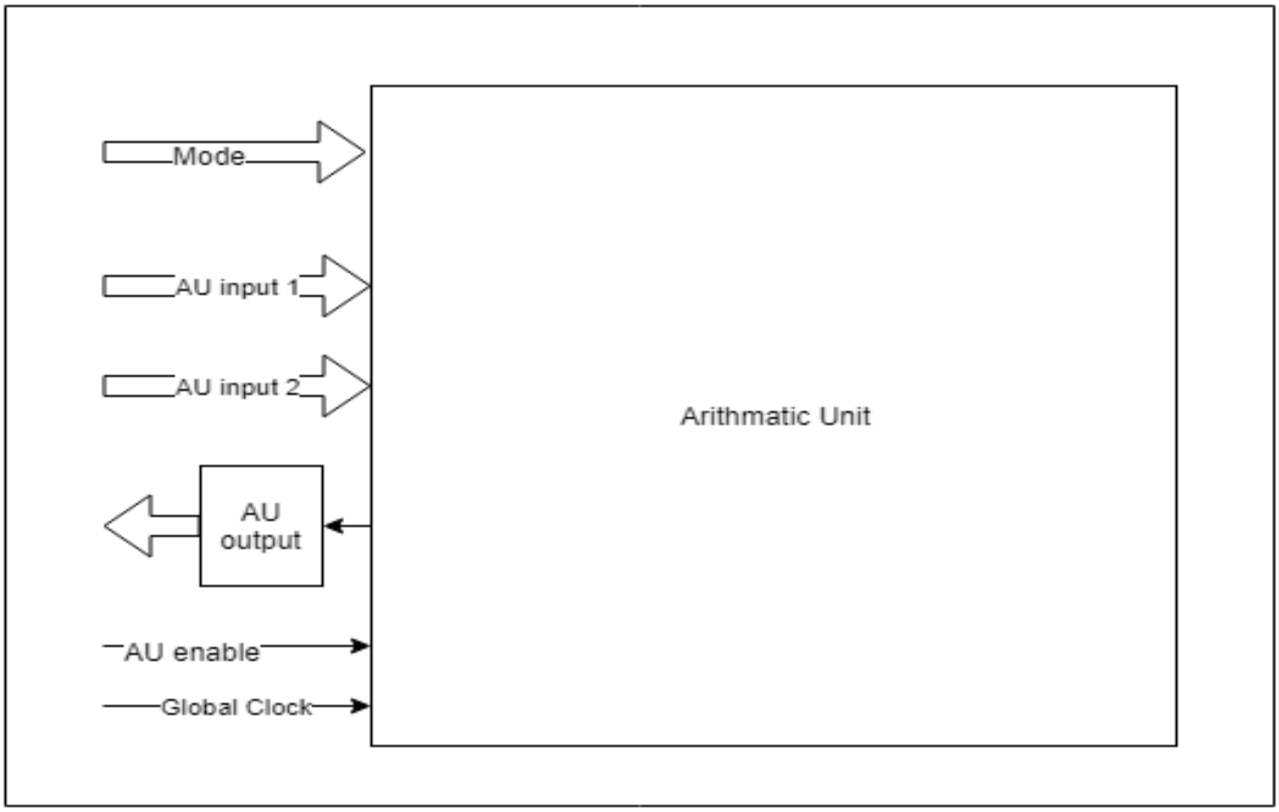
ii) Memory Unit (MEMU)

iii) Execution Unit (EU)



High Level System Diagram

Design of AU:-



Arithmetic Unit

The arithmetic unit can perform only four operation - addition, subtraction, multiplication, division. The AU handles 16 bit number but the MSB is used as sign bit. The output is given as 16 bit number and so overflows are ignored.

Handling of negative number

The negative number is handled by sign bit. The MSB is used as sign bit. The sign bit is set high for negative number. Thus all add, sub, mult, div logics need to be changed to incorporate the

sign bit representation. First sign bits are separated from the operands and the larger operand is stored in large_in and smaller operand is stored in small_in.

ADD Logic

If two operands have the same sign bit then large_in and small_in are added and the sign bit of the output sign bits of the operands.

Else small_in is subtracted from large_in and the output sign bit is the sign bit of the large_in.

SUB Logic

If two operands have the same sign bit then op2 is subtracted from op1 and the sign bit of the output sign bits of the operands.

Else small_in is subtracted from large_in and the output sign bit is the sign bit of the large_in if op1 is greater than op2, and sign bit is negation of large_in if op2 is greater than op1

MULT and DIV Logic

If two operands have the same sign bit then output sign bits of the operand otherwise sign bit is high.

Design of Math Co-Processor

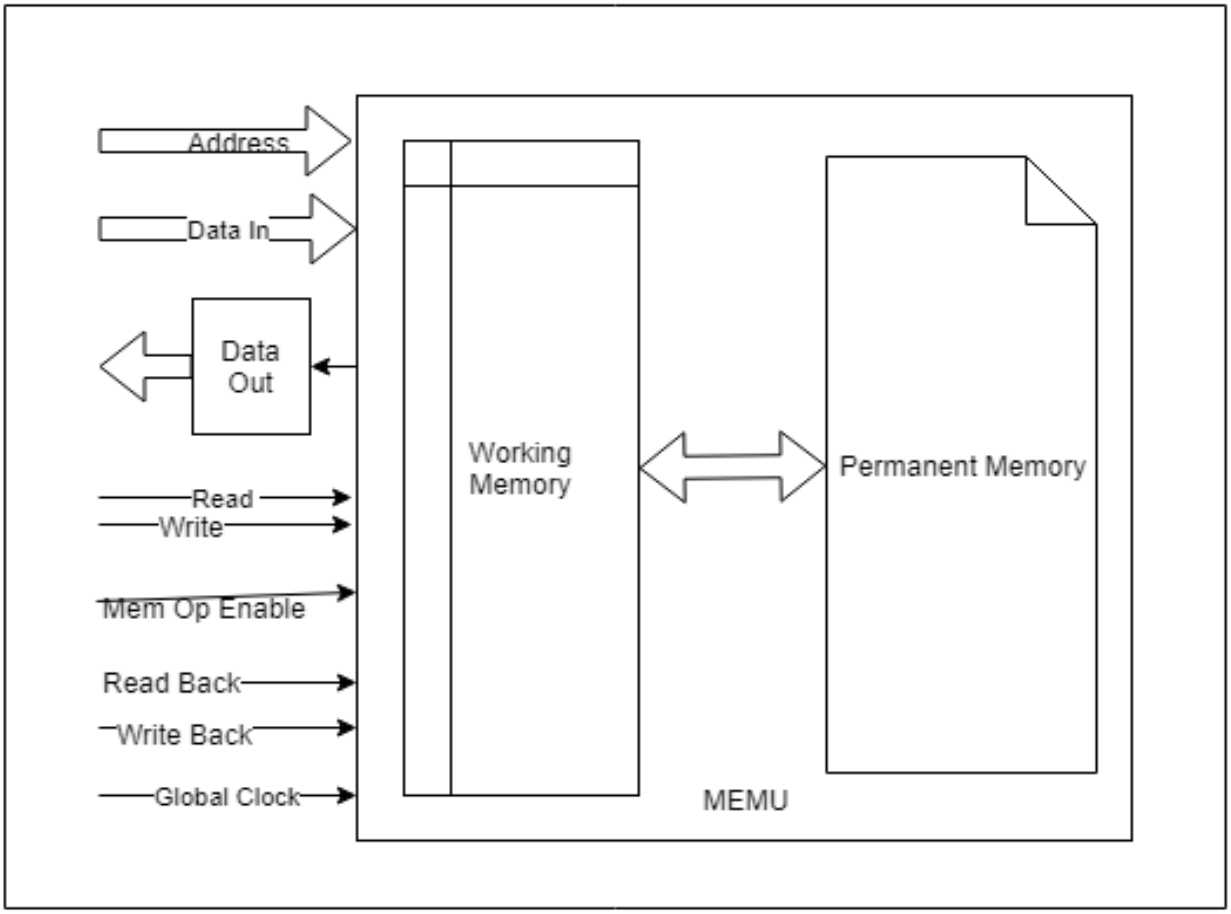
Input:-

Signal	Size	Operation
Global_clk	1	Global clock
AU_op_enable	1	AU is positive edge enable
Mode	4	Opcode for AU
AU_in_1	16	Input1
AU_in_2	16	Input2

Output:-

Register	Size	Operation
AU_out	16	Output

Design of MEMU:-



Memory Unit

The MEMU has two types of memory, permanent memory implemented using file and working memory implemented by register type array. The detailed implementation is described as follows

Design of Math Co-Processor

Permanent memory

The permanent memory is used for input instruction and output variables. When the system starts, it reads the instructions and data from permanent memory to the working memory and when the system halts, the working memory is written to the permanent memory again. This reading and writing process is controlled by two signals - Read back signal and Write back Signal. Both signal works on the positive edge.

Working memory

The execution unit(EU) can read instruction, data from working memory location and write data to working memory directly using Read and Write signal along with Memory Enable signal. Memory unit has other inputs for input address and input data and another extra register for output data.

Read Operation

For read operation first input address is fed to the input address lines, then Read signal is made high and Write signal is made low, finally after a positive edge of Memory enable signal the data is successfully read from working memory to output data register. After a successful reading Read signal is made low.

Write Operation

For write operation first address is fed to the input address lines, then Write signal is made high and Read signal is made low, finally after a positive edge of Memory enable signal the data is successfully written to working memory location. After a successful reading Write signal is made low.

Design of Math Co-Processor

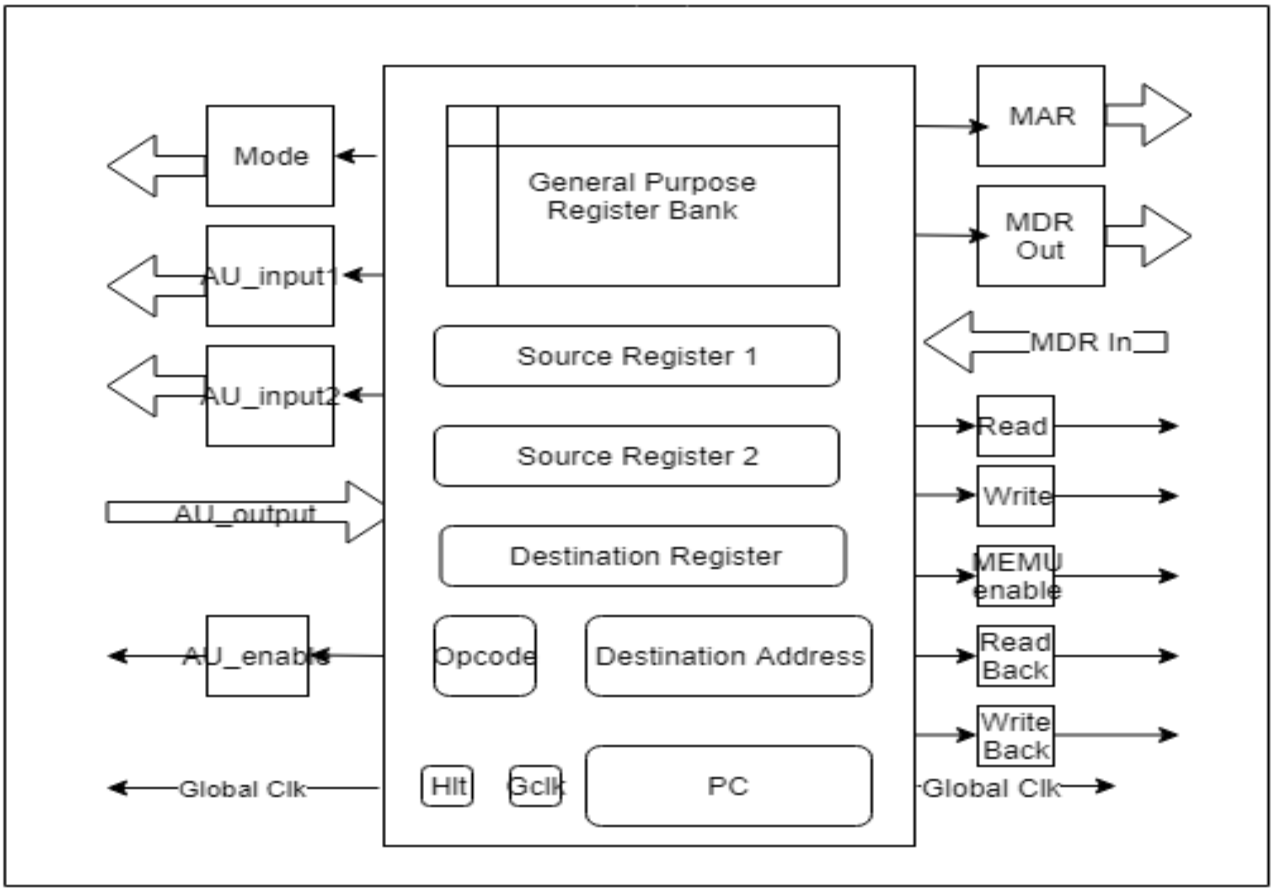
Input:-

Signal	Size	Operation
Global_clk	1	Global clock
Read_back	1	Read back from permanent memory to working memory, positive edge enable
Write_back	1	Write back from working memory to permanent memory, positive edge enable
Read	1	Reading from working memory
Write	1	Writing from working memory
Mem_op_enable	1	MEMU is positive edge enable
Address_in	16	Address
Data_in	8	Input Data

Output:-

Register	Size	Operation
Data_out	8	Output Data

Design of EXEU:-



Execution Unit

All the special and general purpose registers are part of the Execution Unit or the EXEU.

General Purpose Registers

In the instruction set we have 4 bit register address filed, thus we have 2^4 or 16 general purpose registers of 16 bit of 2 byte size in the EXEU (R0~R15). All the general purpose registers are addressable by their specific number. This registers are emulated by a fixed length array of register variables.

Special Registers

Design of Math Co-Processor

The EXEU has two special registers - Program Counter(PC) and Halt register(hlt). The PC stores the memory location of the next instruction. The hlt is set to halt the machine or terminate the execution.

Temporary Register

EXEU uses temporary registers for holding and manipulating the opcode, register address and memory address.

AU interface

It has all the necessary registers to drive the AU and wires to take the output of the AU. Complementary to AU, it has registers AU_op_enable, Mode, AU_in1, AU_in2 and wire AU_out

MEMU interface

To drive the MEMU it has the registers - Write_back_sig, Read_back_sig, Read_sig, Write_sig, Mem_op_enable, MAR, MDR_out and wire MDR_in. MDR_out sends data out from EXEU to MEMU and MDR_in taks data in the EXEU from MEMU.

Clock Signal

EXEU is also responsible to synchronize all parts of the machine by generating clock signals.

Register	Size	Operation
Global_clk	1	Global Clock source
hlt	1	halt the machine
PC	8	Address of the next instruction
reg_bank	16*(2^4)	General purpose registers

Add on Softwares

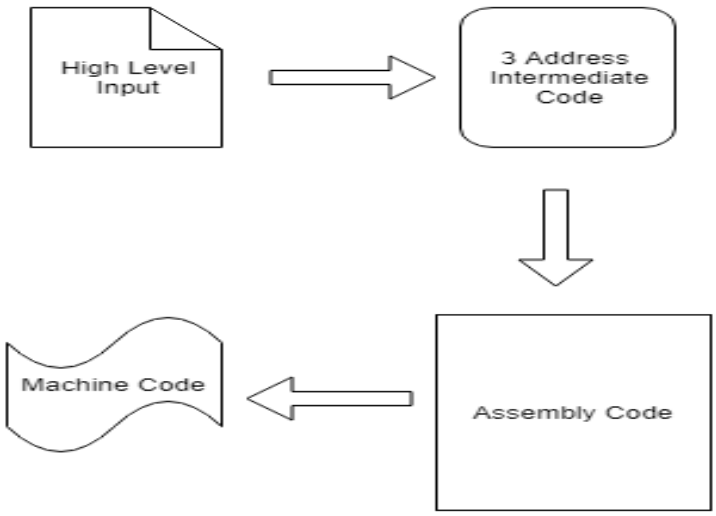
Extra softwares including Compiler and Assemblers are created for this specific machine architecture and instruction set.

Design of Compiler:-

The compiler has two stages - three address code generation and assembly code generation. The compiler uses flex and bison based c implementation for three address code generation and python based script for assembly code generation for this specific instruction set.

Design of Assembler:-

The assembler convert the assembly code to machine code in hex format which can be directly used in ROM file of the machine. The assembler is written in python.



SIMULATION

For simulation and debugging purposes ICRUS VERILOG compiler is used, for advance simulation, synthesis XILINX ISE tool is used.

Example

High Level Input

```
b = 100;
a = -5 + (b - 10) * 4;
```

Three Address Intermediate Code

```
=,b,,100
-,tmp1,0,5
-,tmp2,b,10
*,tmp3,tmp2,4
+,tmp4,tmp1,tmp3
=,a,,tmp4
end :)
```

Assembly Code

```
STI 128, 100
LDI R0, 0
SUBI R0, R0, 5
LD R1, 128
SUBI R1, R1, 10
MULTI R2, R1, 4
ADD R1, R0, R2
ST R1, 129
```

```
HLT
/* Variables are loaded at following locations of the memory:
b -> 80H
a -> 81H
*/
```

Machine Code

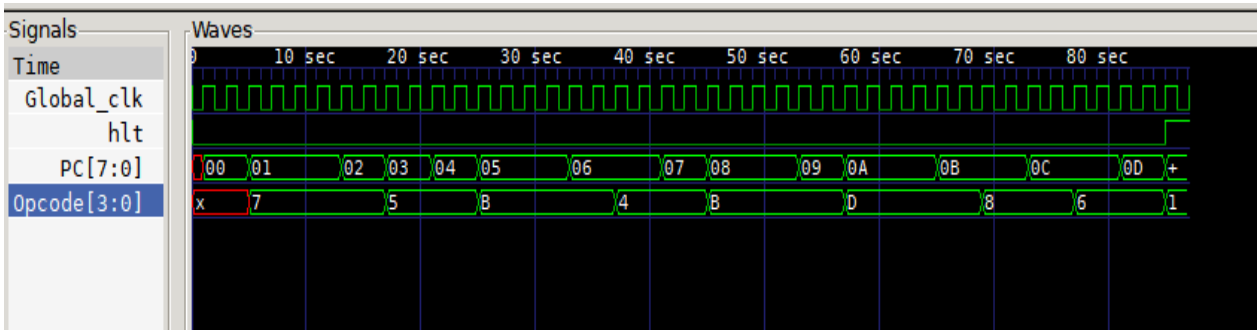
00	**7x80**
01	**0064**
02	**50xx**
03	**0000**
04	**B00x**
05	**0005**
06	**4180**
07	**B11x**
08	**000a**
09	**D21x**
0A	**0004**
0B	**8102**
0C	**6181**
0D	**1xxx**

Output

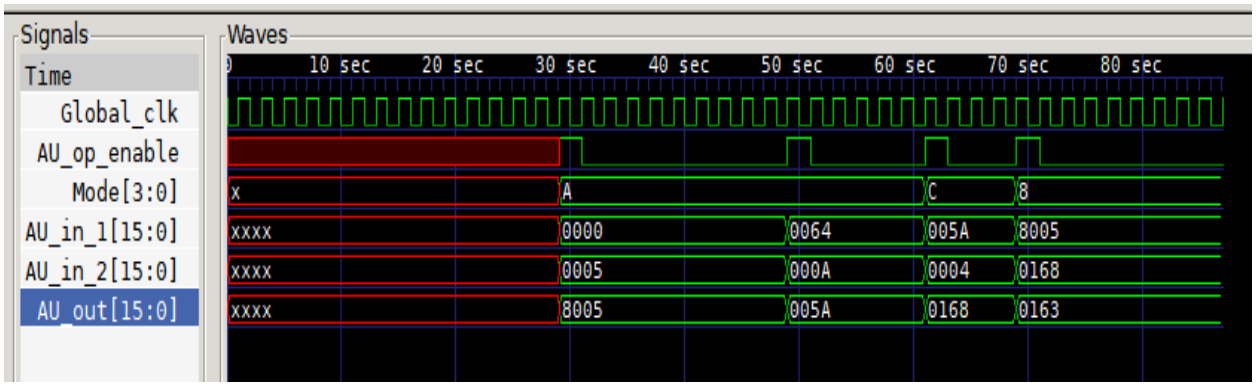
b	**80**	**0064**
a	**81**	**0163**
-	**82**	**xxxx**
-	**83**	**xxxx**
-	**84**	**xxxx**
-	**85**	**xxxx**

64H = 100

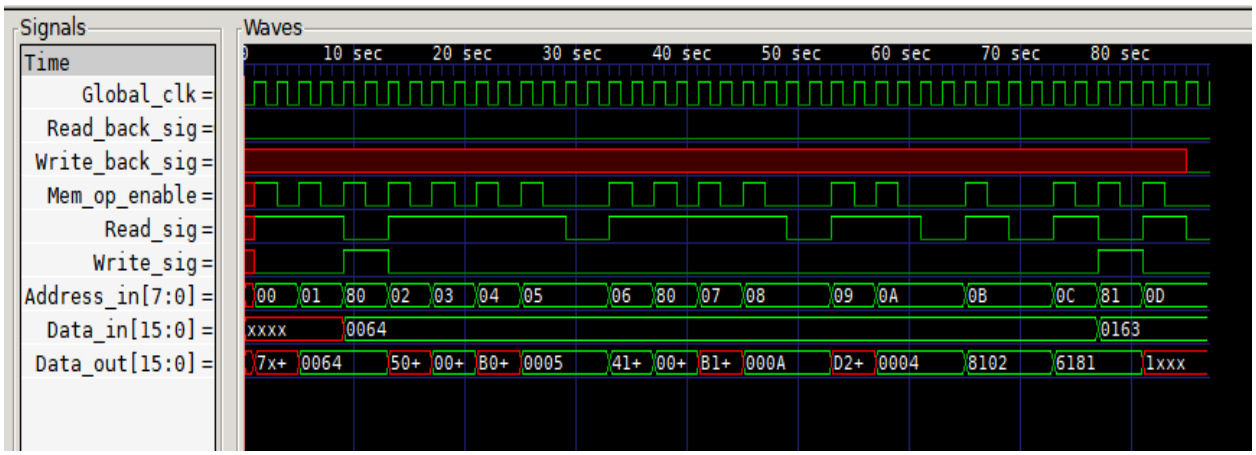
163H = 355


Signals
Waves
Time
Global_clk
hlt
PC[7:0]
Opcode[3:0]
10 sec
20 sec
30 sec
40 sec
50 sec
60 sec
70 sec
80 sec
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D
x 7 5 B 4 B D 8 6 1

EXEU


Signals
Waves
Time
Global_clk
AU_op_enable
Mode[3:0]
AU_in_1[15:0]
AU_in_2[15:0]
AU_out[15:0]
10 sec
20 sec
30 sec
40 sec
50 sec
60 sec
70 sec
80 sec
x A C 8
xxxx 0000 0064 005A 8005
xxxx 0005 000A 0004 0168
xxxx 8005 005A 0168 0163

AU


Signals
Waves
Time
Global_clk
Read_back_sig
Write_back_sig
Mem_op_enable
Read_sig
Write_sig
Address_in[7:0]
Data_in[15:0]
Data_out[15:0]
10 sec
20 sec
30 sec
40 sec
50 sec
60 sec
70 sec
80 sec
00 01 80 02 03 04 05 06 80 07 08 09 0A 0B 0C 81 0D
xxxx 0064 0163
7x+ 0064 50+ 00+ B0+ 0005 41+ 00+ B1+ 000A D2+ 0004 8102 6181 1xxx

MEMU

Appendix I

SystemArchHeader.v

```
`define DATA_WIDTH 16
`define OPCODE_WIDTH 4
`define REG_ADDR_WIDTH 4
`define MEM_WORD_WIDTH 16
`define MEM_ADDR_WIDTH 8

`define OPCODE_OFFSET `DATA_WIDTH
`define REG_DEST_OFFSET `OPCODE_OFFSET - `OPCODE_WIDTH
`define REG_SRC1_OFFSET `REG_DEST_OFFSET - `REG_ADDR_WIDTH
`define REG_SRC2_OFFSET `REG_SRC1_OFFSET - `REG_ADDR_WIDTH

`define MSB `DATA_WIDTH - 1
`define LSB 0
```

InstructionSetHeader.v

```
`include "SystemArchHeader.v"

`define NOP `OPCODE_WIDTH'h0
`define HLT `OPCODE_WIDTH'h1
`define LDPC `OPCODE_WIDTH'h2
`define MV `OPCODE_WIDTH'h3
`define LD `OPCODE_WIDTH'h4
`define LDI `OPCODE_WIDTH'h5
`define ST `OPCODE_WIDTH'h6
`define STI `OPCODE_WIDTH'h7
`define ADD `OPCODE_WIDTH'h8
`define ADDI `OPCODE_WIDTH'h9
`define SUB `OPCODE_WIDTH'hA
`define SUBI `OPCODE_WIDTH'hB
`define MULT `OPCODE_WIDTH'hC
`define MULTI `OPCODE_WIDTH'hD
`define DIV `OPCODE_WIDTH'hE
`define DIVI `OPCODE_WIDTH'hF
```

SystemDelays.v

```
//Adjust delays according to system
`define MEM_RWB_DELAY 1
`define MEM_OP_DELAY 1
`define AU_OP_DELAY 1
```

AU.v

```
`include "InstructionSetHeader.v"
`include "SystemArchHeader.v"

module AU(
            input wire Global_clk,
            input wire AU_op_enable,
            input wire [`OPCODE_WIDTH-1:0] Mode,
            input wire [`DATA_WIDTH-1:0] AU_in_1,
            input wire [`DATA_WIDTH-1:0] AU_in_2,
            output reg [`DATA_WIDTH-1:0] AU_out
      );

      reg output_signbit;
      reg [`DATA_WIDTH-1:0] tmp_in_1, tmp_in_2, tmp_swap;

      always @(posedge AU_op_enable) begin
            tmp_in_1 = AU_in_1;
            tmp_in_1[`MSB] = 0;
            tmp_in_2 = AU_in_2;
            tmp_in_2[`MSB] = 0;
            if (Mode == `MULT | Mode == `DIV) begin
                  output_signbit = AU_in_1[`MSB] ^ AU_in_2[`MSB];
            end else begin
                   if (tmp_in_1 > tmp_in_2) begin
                        output_signbit = AU_in_1[`MSB];
                   end else begin
                        if (Mode == `SUB) begin
                              output_signbit = ~AU_in_2[`MSB];
                        end else begin
                              output_signbit = AU_in_2[`MSB];
                        end
                        tmp_swap = tmp_in_1;
                        tmp_in_1 = tmp_in_2;
                        tmp_in_2 = tmp_swap;
                   end
            end
```

```
            case (Mode)
                  `NOP: begin
                                    $display("Stall");
                              end
                  `ADD: begin
                                    $display("Adding in AU");
                                    if (AU_in_1[`MSB] ^ AU_in_2[`MSB]) begin
                                          AU_out = tmp_in_1 - tmp_in_2;
                                          AU_out[`MSB] = output_signbit;
                                    end else begin
                                          AU_out = tmp_in_1 + tmp_in_2;
                                          AU_out[`MSB] = output_signbit;
                                    end

                                    $display("%d + %d = %d", AU_in_1,
AU_in_2, AU_out);
                              end

                  `SUB: begin
                                    $display("Substracting in AU");
                                    if (AU_in_1[`MSB] ^ AU_in_2[`MSB]) begin
                                          AU_out = tmp_in_1 + tmp_in_2;
                                          AU_out[`MSB] = output_signbit;
                                    end else begin
                                          AU_out = tmp_in_1 - tmp_in_2;
                                          AU_out[`MSB] = output_signbit;
                                    end
                                    $display("%d - %d = %d", AU_in_1,
AU_in_2, AU_out);
                              end

                  `MULT:begin
                                    $display("Multiplying in AU");
                                    AU_out = AU_in_1 * AU_in_2;
                                    AU_out[`MSB] = output_signbit;
                                    $display("%d * %d = %d", AU_in_1,
AU_in_2, AU_out);
                              end

                  `DIV: begin
                                    $display("Dividing in AU");
                                    AU_out = AU_in_1 / AU_in_2;
                                    AU_out[`MSB] = output_signbit;
                                    $display("%d / %d = %d", AU_in_1,
AU_in_2, AU_out);
                              end
            endcase
            //AU_out[`MSB] = output_signbit;

      end
```

```
endmodule
```

MEMU.v

```
`include "SystemArchHeader.v"
`include "SystemDelays.v"

module MEMU(
            input wire Global_clk,
            input wire Read_back_sig,
            input wire Write_back_sig,
            input wire Read_sig,
            input wire Write_sig,
            input wire Mem_op_enable,
            input wire[`MEM_ADDR_WIDTH-1:0] Address_in,
            input wire[`MEM_WORD_WIDTH-1:0] Data_in,
            output reg[`MEM_WORD_WIDTH-1:0] Data_out
      );
      reg [`MEM_WORD_WIDTH-1:0] mem [0:2**`MEM_ADDR_WIDTH-1];
      integer i, file;

      always @(posedge Read_back_sig) begin
            file = $fopen("test_rom.mem", "r");
            $display ("test_rom.mem opened for readback");
            if (file == 0) begin
            $display ("ERROR: test_rom.mem not opened");
            end else begin
                  $readmemh("test_rom.mem", mem);
                  //$display("%d -> %x%x%x%x", 15, mem[15][3:0],
mem[15][7:4], mem[15][11:8], mem[15][15:12]);
                  $display ("Successfully read back");
                  $fclose(file);
            end
      end

      always @(posedge Write_back_sig) begin
            file = $fopen("test_rom.mem", "w");
            $display ("test_rom.mem opened for wirteback");
            if (file == 0) begin
            $display ("ERROR: test_rom.mem not opened");
            end else begin
                  for(i = 0; i < 2**`MEM_ADDR_WIDTH; i+=1) begin
                        $fdisplay(file, "%x%x%x%x", mem[i][15:12],
mem[i][11:8], mem[i][7:4], mem[i][3:0]);
                  end
                  $display ("Successfully written back");
```

```
                $fclose(file);
            end
      end

      always @(posedge Mem_op_enable) begin
            if (Read_sig & !Write_sig) begin
                Data_out = mem[Address_in];
                $display("Reading from Memloc %d, Data %x%x%x%x",
Address_in, Data_out[15:12], Data_out[11:8], Data_out[7:4], Data_out[3:0]);
            end
            else if (Write_sig & !Read_sig) begin
                mem[Address_in] = Data_in;
                $display("Writing to Memloc %d, Data %x%x%x%x",
Address_in, mem[Address_in][15:12], mem[Address_in][11:8],
mem[Address_in][7:4], mem[Address_in][3:0]);
            end
            else begin
                $display("Stall");
            end
      end
endmodule
```

EXEU.v

```
`include "InstructionSetHeader.v"
`include "SystemArchHeader.v"
`include "SystemDelays.v"

module EXEU();
      integer i, total_clk_cycle;
      reg Global_clk, hlt;
      reg [`MEM_WORD_WIDTH-1:0] reg_bank [0:2**`REG_ADDR_WIDTH-1];
      reg [`MEM_ADDR_WIDTH-1:0] PC;
      reg [`OPCODE_WIDTH-1:0] Opcode;
      reg [`REG_ADDR_WIDTH-1:0] Dest_reg, Src_reg1, Src_reg2;
      reg [`MEM_ADDR_WIDTH-1:0] Dest_address;

      reg Write_back_sig, Read_back_sig, Read_sig, Write_sig, Mem_op_enable;
      reg [`MEM_ADDR_WIDTH-1:0] MAR;
      reg [`MEM_WORD_WIDTH-1:0] MDR_out;
      wire [`MEM_WORD_WIDTH-1:0] MDR_in;
      MEMU test_memu(Global_clk, Read_back_sig, Write_back_sig, Read_sig,
Write_sig, Mem_op_enable, MAR, MDR_out, MDR_in);
```

```
	reg AU_op_enable;
	reg [`DATA_WIDTH-1:0] AU_in1, AU_in2;
	wire [`DATA_WIDTH-1:0] AU_out;
	reg [`OPCODE_WIDTH-1:0] Mode;
	AU test_au(Global_clk, AU_op_enable, Mode, AU_in1, AU_in2, AU_out);

	always begin
		#1 Global_clk = ~Global_clk;
		total_clk_cycle += 1;
	end

	initial begin
		$dumpfile("my_dumpfile.vcd");
		$dumpvars;
		//Reset Global_clk
		total_clk_cycle = 0;
		Global_clk = 0;
		//Start processor
		hlt = 0;
		//Load RAM from file
		Read_back_sig = 1;
		Read_back_sig = 0;
		repeat (`MEM_RWB_DELAY) begin
			@ (posedge Global_clk);
		end
		//Load PC with starting address
		PC = `MEM_ADDR_WIDTH'h00;
		//hlt = 0;
		while (!hlt) begin
			//IF
			Read_sig = 1;
			Write_sig = 0;
			MAR = PC;
			Mem_op_enable = 1;
			Mem_op_enable = 0;
			repeat (`MEM_OP_DELAY) begin
				@ (posedge Global_clk);
			end
			$display("Executing Insruction %x%x%x%x", MDR_in[15:12],
MDR_in[11:8], MDR_in[7:4], MDR_in[3:0]);
			Opcode =
MDR_in[`OPCODE_OFFSET-1:`OPCODE_OFFSET-`OPCODE_WIDTH];

			//ID
			if (Opcode[`OPCODE_WIDTH-1]) begin		//AU
operations
				if(Opcode[0]) begin
//Immediate addressing mode operations- ADDI, SUBI, MULTI, DIVI
					Dest_reg = MDR_in[`REG_DEST_OFFSET-1:
`REG_DEST_OFFSET - `REG_ADDR_WIDTH];
					Src_reg1 =
```

```
MDR_in[`REG_SRC1_OFFSET-1:`REG_SRC2_OFFSET];
                              $display("Loading Immediate");
                              PC += 1;
                              Read_sig = 1;
                              Write_sig = 0;
                              MAR = PC;
                              Mem_op_enable = 1;
                              Mem_op_enable = 0;
                              repeat (`MEM_OP_DELAY) begin
                                    @ (posedge Global_clk);
                              end
                              AU_in1 = reg_bank[Src_reg1];
                              AU_in2 = MDR_in;
                              Mode = Opcode;
                              Mode[0] = 0;
                              AU_op_enable = 1;
                              AU_op_enable = 0;
                              repeat (`AU_OP_DELAY) begin
                                    @ (posedge Global_clk);
                              end
                              reg_bank[Dest_reg] = AU_out;
                              for(i = 0; i < 2**`REG_ADDR_WIDTH; i+=1) begin
                                    $display("Reg[%x] = %x%x%x%x", i,
reg_bank[i][15:12], reg_bank[i][11:8], reg_bank[i][7:4], reg_bank[i][3:0]);
                              end
                        end else begin
      //Register addressing mode operations- ADD, SUB, MULT, DIV
                              Dest_reg =
MDR_in[`REG_DEST_OFFSET-1:`REG_DEST_OFFSET-`REG_ADDR_WIDTH];
                              Src_reg1 =
MDR_in[`REG_SRC1_OFFSET-1:`REG_SRC2_OFFSET];
                              Src_reg2 =
MDR_in[`REG_SRC2_OFFSET-1:`REG_SRC2_OFFSET-`REG_ADDR_WIDTH];
                              AU_in1 = reg_bank[Src_reg1];
                              AU_in2 = reg_bank[Src_reg2];
                              Mode = Opcode;
                              AU_op_enable = 1;
                              AU_op_enable = 0;
                              repeat (`AU_OP_DELAY) begin
                                    @ (posedge Global_clk);
                              end
                              reg_bank[Dest_reg] = AU_out;
                              for(i = 0; i < 2**`REG_ADDR_WIDTH; i+=1) begin
                                    $display("Reg[%x] = %x%x%x%x", i,
reg_bank[i][15:12], reg_bank[i][11:8], reg_bank[i][7:4], reg_bank[i][3:0]);
                              end
                        end
                  end else begin          //MEM Operations
                        case (Opcode)
                              `NOP: begin
                                          $display("Stalling");
```

```
                                    end
                            `HLT: begin
                                    $display("Halting");
                                    hlt = 1;
                                    end
                            `MV:  begin
                                    Dest_reg =
MDR_in[`REG_DEST_OFFSET-1:`REG_DEST_OFFSET-`REG_ADDR_WIDTH];
                                    Src_reg1 =
MDR_in[`REG_SRC1_OFFSET-1:`REG_SRC2_OFFSET];
                                    $display("Moving from reg %d
to reg %d", Src_reg1, Dest_reg);
                                    reg_bank[Dest_reg] =
reg_bank[Src_reg1];
                                    for(i = 0; i <
2**`REG_ADDR_WIDTH; i+=1) begin
                                          $display("Reg[%x] =
%x%x%x%x", i, reg_bank[i][15:12], reg_bank[i][11:8], reg_bank[i][7:4],
reg_bank[i][3:0]);
                                    end
                                    end

                            `LD:  begin
                                    $display("Loading");
                                    Dest_reg =
MDR_in[`REG_DEST_OFFSET-1:`REG_DEST_OFFSET-`REG_ADDR_WIDTH];
                                    MAR =
MDR_in[`REG_SRC1_OFFSET-1:`REG_SRC1_OFFSET-`MEM_ADDR_WIDTH];
                                    Read_sig = 1;
                                    Write_sig = 0;
                                    Mem_op_enable = 1;
                                    Mem_op_enable = 0;
                                    repeat (`MEM_OP_DELAY) begin
                                          @ (posedge
Global_clk);
                                    end
                                    reg_bank[Dest_reg] = MDR_in;
                                    for(i = 0; i <
2**`REG_ADDR_WIDTH; i+=1) begin
                                          $display("Reg[%x] =
%x%x%x%x", i, reg_bank[i][15:12], reg_bank[i][11:8], reg_bank[i][7:4],
reg_bank[i][3:0]);
                                    end
                                    end

                            `LDI: begin
                                    $display("Loading
Immediate");
                                    Dest_reg =
MDR_in[`REG_DEST_OFFSET-1:`REG_DEST_OFFSET-`REG_ADDR_WIDTH];
                                    PC += 1;
```

```
                                                Read_sig = 1;
                                                Write_sig = 0;
                                                MAR = PC;
                                                Mem_op_enable = 1;
                                                Mem_op_enable = 0;
                                                repeat (`MEM_OP_DELAY) begin
                                                      @ (posedge
Global_clk);
                                                end
                                                reg_bank[Dest_reg] = MDR_in;
                                                for(i = 0; i <
2**`REG_ADDR_WIDTH; i+=1) begin
                                                      $display("Reg[%x] =
%x%x%x%x", i, reg_bank[i][15:12], reg_bank[i][11:8], reg_bank[i][7:4],
reg_bank[i][3:0]);
                                                end
                                          end
                                `ST:  begin
                                                $display("Storing");
                                                Dest_reg =
MDR_in[`REG_DEST_OFFSET-1:`REG_DEST_OFFSET-`REG_ADDR_WIDTH];
                                                MAR =
MDR_in[`REG_SRC1_OFFSET-1:`REG_SRC1_OFFSET-`MEM_ADDR_WIDTH];
                                                MDR_out =
reg_bank[Dest_reg];
                                                Read_sig = 0;
                                                Write_sig = 1;
                                                Mem_op_enable = 1;
                                                Mem_op_enable = 0;
                                                repeat (`MEM_OP_DELAY) begin
                                                      @ (posedge
Global_clk);
                                                end
                                          end
                                `STI: begin
                                                $display("Storing
Immediate");
                                                Dest_address =
MDR_in[`REG_SRC1_OFFSET-1:0];
                                                PC += 1;
                                                MAR = PC;
                                                Read_sig = 1;
                                                Write_sig = 0;
                                                Mem_op_enable = 1;
                                                Mem_op_enable = 0;
                                                repeat (`MEM_OP_DELAY) begin
                                                      @ (posedge
Global_clk);
                                                end
                                                MAR = Dest_address;
                                                MDR_out = MDR_in;
```

```
                                                Read_sig = 0;
                                                Write_sig = 1;
                                                Mem_op_enable = 1;
                                                Mem_op_enable = 0;
                                                repeat (`MEM_OP_DELAY) begin
                                                      @ (posedge
Global_clk);
                                                end
                                          end
                        endcase
                  end
            PC += 1;
            end
            //Write RAM back to file
            Write_back_sig = 1;
            Write_back_sig = 0;
            repeat (`MEM_RWB_DELAY) begin
                  @ (posedge Global_clk);
            end
            $display("Number of clock cycle = %d", total_clk_cycle);
            $finish;
      end
endmodule
```

Appendix - II

lexer.l

```
%{
#include "grammer.tab.h"
int lineno = 1;
%}
lalpha [a-z]
calpha [A-Z]
dgt [0-9]
%%
{dgt}+ {yylval.type.place = (char *)malloc(strlen(yytext)+1);
            strcpy(yylval.type.place, yytext);
            yylval.type.code = malloc(1);
            yylval.type.code[0] = 0;
            return INTCONST;}
[a-zA-Z_][a-zA-Z0-9_]* {yylval.type.place = (char
*)malloc(strlen(yytext)+1);
                                    strcpy(yylval.type.place, yytext);
                                    yylval.type.code = malloc(1);
                                    yylval.type.code[0] = 0;
                                    return ID_TOK;}
"/*"([^*]|\*+[^*/])*"*/" {}
"+=" {return PLUS_EQ_TOK;}
"-=" {return MINUS_EQ_TOK;}
"*=" {return MULT_EQ_TOK;}
"/=" {return DIVIDE_EQ_TOK;}
";" {return SEMICOLON_TOK;}
"-" {return MINUS_TOK;}
"+" {return PLUS_TOK;}
"*" {return MULT_TOK;}
"/" {return DIVIDE_TOK;}
"(" {return LPAREN_TOK;}
")" {return RPAREN_TOK;}
"=" {return EQ_TOK;}
"\n" {yylineno++;}
[ \t\n] ;
%%

int yywrap(){
    return 1;
}
```

grammer.y

```
%{
#include<stdio.h>
#define YYDEBUG 1
extern FILE *yyin;
extern int yylineno;
extern char* yytext;
extern char printtype[][10];
int bufcnt;
FILE *fpout;
int tmpcnt;
%}

%union {
    struct t{
        char *place;
        char *code;
    } type;
}

%start DEBUG
%token LPAREN_TOK RPAREN_TOK
%token EQ_TOK MINUS_TOK PLUS_TOK MULT_TOK DIVIDE_TOK PLUS_EQ_TOK
MINUS_EQ_TOK MULT_EQ_TOK DIVIDE_EQ_TOK SEMICOLON_TOK
%token MINUS_MINUS_TOK PLUS_PLUS_TOK ERROR_TOK

%token<type> INTCONST
%token<type> ID_TOK

%left PLUS_TOK MINUS_TOK
%left MULT_TOK DIVIDE_TOK
%nonassoc UMINUS

%type<type> var exp0 exp2 exp block statement DEBUG
%%

DEBUG: block {  $$.code = (char *)0; concatcode(&$1.code, "end ",":)\n");
fprintf(fpout, "%s", $1.code);}
    ;

block: block statement { $$.place = malloc(1); $$.place[0] = 0; $$.code =
(char *)0; concatcode(&$$.code, $1.code, $2.code); }
    |  %empty {$$.code = malloc(1); $$.code[0] = 0;}

statement: exp SEMICOLON_TOK
    ;


exp: ID_TOK EQ_TOK exp2     {  $$.place = malloc(1); $$.place[0] = 0;
```

```
addtocode(&$$.code, $1.code, $3.code, "=", $1.place, "", $3.place); }
    | ID_TOK PLUS_EQ_TOK exp2   {  $$.place = malloc(1); $$.place[0] = 0;
addtocode(&$$.code, $1.code, $3.code, "+", $1.place, $3.place, $1.place); }
    | ID_TOK MINUS_EQ_TOK exp2  {  $$.place = malloc(1); $$.place[0] = 0;
addtocode(&$$.code, $1.code, $3.code, "-", $1.place, $3.place, $1.place); }
    | ID_TOK MULT_EQ_TOK exp2   {  $$.place = malloc(1); $$.place[0] = 0;
addtocode(&$$.code, $1.code, $3.code, "*", $1.place, $3.place, $1.place); }
    | ID_TOK DIVIDE_EQ_TOK exp2 {  $$.place = malloc(1); $$.place[0] = 0;
addtocode(&$$.code, $1.code, $3.code, "/", $1.place, $3.place, $1.place); }
    | exp0
    ;

exp2: LPAREN_TOK exp2 RPAREN_TOK    {   $$ = $2; }
    | MINUS_TOK exp2 { addtmptoplace(&$$.place); addtocode(&$$.code,
$2.code, "", "-", $$.place, "0", $2.place); } %prec UMINUS
    | exp2 PLUS_TOK exp2    {   addtmptoplace(&$$.place);
addtocode(&$$.code, $1.code, $3.code, "+", $$.place, $1.place, $3.place); }
    | exp2 MINUS_TOK exp2   {   addtmptoplace(&$$.place);
addtocode(&$$.code, $1.code, $3.code, "-", $$.place, $1.place, $3.place);  }
    | exp2 MULT_TOK exp2    {   addtmptoplace(&$$.place);
addtocode(&$$.code, $1.code, $3.code, "*", $$.place, $1.place, $3.place);  }
    | exp2 DIVIDE_TOK exp2   {  addtmptoplace(&$$.place);
addtocode(&$$.code, $1.code, $3.code, "/", $$.place, $1.place, $3.place); }
    | exp0
    | var
    ;

exp0: ID_TOK PLUS_PLUS_TOK  { addtoplace(&$$.place, $1.place);
addtocode(&$$.code, $1.code, "", "+",$1.place,"1",$1.place); }
    | ID_TOK MINUS_MINUS_TOK    { addtoplace(&$$.place, $1.place);
addtocode(&$$.code, $1.code, "", "-",$1.place,"1",$1.place); }
    | PLUS_PLUS_TOK ID_TOK  { addtoplace(&$$.place, $2.place);
addtocode(&$$.code, $2.code, "", "+",$2.place,"1",$2.place); }
    | MINUS_MINUS_TOK ID_TOK    { addtoplace(&$$.place, $2.place);
addtocode(&$$.code, $2.code, "", "-",$2.place,"1",$2.place); }
    ;

var: ID_TOK { $$ = $1;  }
    | INTCONST  { $$ = $1;}
    ;

%%

void addtmptoplace(char **dest){
    *dest = (char *)malloc(10);
    sprintf(*dest,"tmp%d",++tmpcnt);
}

void addtoplace(char **dest, const char *src){
    *dest = (char *)malloc(strlen(src)+1);
    strcpy(*dest, src);
```

```
}

void addtocode(char **dest, const char *code1, const char *code2, const char
*op, const char *arg1, const char *arg2, const char *res){
    int l = 0;
    l = strlen(code1) + strlen(code2) + strlen(op) + strlen(arg1) +
strlen(arg2) + strlen(res) + 4;
    *dest = (char *)malloc(l+1);
    *dest[0] = (char)0;

    strcat(*dest, code1);
    strcat(*dest, code2);
    sprintf(*dest,"%s%s,%s,%s,%s\n",*dest, op, arg1, arg2, res );
}

void concatcode(char **dest, const char *code1, const char *code2){
    int l = 0;
    if(*dest)   l = strlen(*dest);

    *dest = realloc(*dest, l+strlen(code1)+strlen(code2)+1);

    if(l){
        strcat(*dest, code1);
        strcat(*dest, code2);
    }else    sprintf(*dest,"%s%s", code1, code2);

}

void concat(char *dest, char *src){
    int l1, l2;
    l1 = strlen(dest);
    l2 = strlen(src);
    dest = malloc(l1 + l2 + 1);
    sprintf(dest,"%s%s", dest, src);
}


int main(int argc, char *argv[]){
    int token;
    yydebug = 0;
    if(argc != 2){
        yytext = stdin;
    } else {
        yyin = fopen(argv[1], "r");
    }
    fpout = fopen("threeaddresscode.txt","w");
    if(!yyparse());
        fprintf(stdout, "Total %d line parsed successfully :)\n", yylineno);
    fclose(yyin);
    //printhashtable();
    return 0;
```

```
}

void yyerror (char const *s) {
    fprintf(stderr, "%s!! @line no - %d:  @symbol '%s' :(  \n",s ,yylineno,
yytext);
}
```

assembly_generator.py

```
import re

def allocreg():
   global regset
   for i in range(len(regset)):
       if regset[i]:   return i
   print("Out of registers :(")
   return -1



def isint(arg):
   return bool(re.match("^[0-9]+$", arg))

def simpleoperators(op, res, arg1, arg2):
   global tmptoreg, regset
   reg = allocreg()
   if reg == -1:   return 1

   if "tmp" not in arg1 and "tmp" not in arg2:
       if isint(arg1) and isint(arg2):
           outfile.write("LDI R"+str(reg)+", "+arg1+"\n")
           outfile.write(op+"I R"+str(reg)+", R"+str(reg)+", "+arg2+"\n")
       elif isint(arg1) and not isint(arg2):
           outfile.write("LD R"+str(reg)+",
"+str(varopset+vartomem.index(arg2))+"\n")
           outfile.write(op+"I R"+str(reg)+", R"+str(reg)+", "+arg1+"\n")
       elif not isint(arg1) and isint(arg2):
```

```
            outfile.write("LD R"+str(reg)+",
"+str(varopset+vartomem.index(arg1))+"\n")
            outfile.write(op+"I R"+str(reg)+", R"+str(reg)+", "+arg2+"\n")
        else:
            tmpreg = allocreg()
            outfile.write("LD R"+str(reg)+",
"+str(varopset+vartomem.index(arg1))+"\n")
            outfile.write("LD R"+str(tmpreg)+",
"+str(varopset+vartomem.index(arg2))+"\n")
            outfile.write(op+" R"+str(reg)+", R"+str(tmpreg)+",
R"+str(reg)+"\n")
            regset[tmptoreg[tmpreg]] = True

    elif "tmp" in arg1 and "tmp" not in arg2:
        if isint(arg2):
            if arg1 not in intermidiatecode[curline+1: ]:
regset[tmptoreg[arg1]] = True
            outfile.write(op+"I R"+str(reg)+", R"+str(tmptoreg[arg1])+",
"+arg2+"\n")
        else:
            outfile.write("LD R"+str(reg)+",
"+str(varopset+vartomem.index(arg2))+"\n")
            outfile.write(op+" R"+str(reg)+", R"+str(reg)+", "+arg1+"\n")

    elif "tmp" not in arg1 and "tmp" in arg2:
        if isint(arg1):
            if arg2 not in intermidiatecode[curline+1: ]:
regset[tmptoreg[arg2]] = True
            outfile.write(op+"I R"+str(reg)+", R"+str(tmptoreg[arg2])+",
"+arg1+"\n")
        else:
            outfile.write("LD R"+str(reg)+",
"+str(varopset+vartomem.index(arg1))+"\n")
            outfile.write(op+" R"+str(reg)+", R"+str(reg)+", "+arg2+"\n")

    else:
        outfile.write(op+" R"+str(reg)+", R"+str(tmptoreg[arg1])+",
```

```
R"+str(tmptoreg[arg2])+"\n")
       if arg1 not in intermidiatecode[curline+1: ]:
regset[tmptoreg[arg1]] = True
       if arg2 not in intermidiatecode[curline+1: ]:
regset[tmptoreg[arg2]] = True

   tmptoreg[res] = reg
   regset[reg] = False
   return 0


def memoperator(res, arg):
   if isint(arg):
       outfile.write("STI "+str(varopset+vartomem.index(res))+", "+arg+"\n")
   elif "tmp" in arg:
       outfile.write("ST R"+str(tmptoreg[arg])+",
"+str(varopset+vartomem.index(res))+"\n")
   else:
       tmpreg = allocreg()
       outfile.write("LD R"+str(tmpreg)+",
"+str(varopset+vartomem.index(arg))+"\n")
       outfile.write("ST "+arg+", "+str(varopset+vartomem.index(res))+"\n")
       regset[tmptoreg[tmpreg]] = True


regset = [True for n in range(16)]
tmptoreg = {}
vartomem = []
varopset = 128
inputfile = open('threeaddresscode.txt')
outfile = open('assemblycode.m','w')
intermidiatecode = inputfile.read().split("\r\n")
#print((len(intermidiatecode), intermidiatecode[0]))
curline = 0
flag = True
while True:
   if "end" in intermidiatecode[curline]:
```

```
        outfile.write("HLT\n")
        break
    print((curline,intermidiatecode[curline]))
    op, res, arg1, arg2 = intermidiatecode[curline].split(",")
    val = 0
    if "+" in op:   val = simpleoperators("ADD", res, arg1, arg2)
    elif "-" in op:   val = simpleoperators("SUB", res, arg1, arg2)
    elif "*" in op:   val = simpleoperators("MULT", res, arg1, arg2)
    elif "/" in op:   val = simpleoperators("DIV", res, arg1, arg2)
    elif "=" in op:
        if res not in vartomem: vartomem.append(res)
        val = memoperator(res, arg2)
    else:   outfile.write(op+" "+res+"\n")
    if(val):    break
    curline += 1
outfile.write("\n\n/* Variables are loaded at following locations:\n")
for i in range(len(vartomem)):
    outfile.write(vartomem[i]+" ->  "+str(i+varopset)+"M \n")
outfile.write("*/")
outfile.close()
```

machinecode_gen.py

```
def allocreg():
    global regset
    for i in range(len(regset)):
        if regset[i]:   return i
    print("Out of registers :(")
    return -1

def simpleoperators(op, arg1, arg2, res):
    reg = allocreg()
    if reg == -1:   return 1
    if "tmp" not in arg1 and "tmp" not in arg2:
        outfile.write("MOV "+arg1+", R"+str(reg)+"\n")
```

```
        outfile.write(op+" "+arg2+", R"+str(reg)+"\n")
    elif "tmp" in arg1 and "tmp" not in arg2:
        outfile.write("MOV R"+str(vartoreg[arg1])+", R"+str(reg)+"\n")
        if arg1 not in intermidiatecode[curline+1: ]:
regset[vartoreg[arg1]] = True
        outfile.write(op+" "+arg2+", R"+str(reg)+"\n")
    elif "tmp" not in arg1 and "tmp" in arg2:
        outfile.write("MOV R"+str(vartoreg[arg2])+", R"+str(reg)+"\n")
        if arg2 not in intermidiatecode[curline+1: ]:
regset[vartoreg[arg2]] = True
        outfile.write(op+" "+arg1+", R"+str(reg)+"\n")
    else:
        outfile.write("MOV R"+str(vartoreg[arg1])+", R"+str(reg)+"\n")
        outfile.write(op+" "+str(vartoreg[arg2])+", R"+str(reg)+"\n")
        if arg1 not in intermidiatecode[curline+1: ]:
regset[vartoreg[arg1]] = True
        if arg2 not in intermidiatecode[curline+1: ]:
regset[vartoreg[arg2]] = True

    vartoreg[res] = reg
    regset[reg] = False
    return 0

inputfile = open('threeaddresscode.txt')
outfile = open('assemblycode.m','w')
intermidiatecode = inputfile.read().split("\n")
print((len(intermidiatecode), intermidiatecode[0]))
curline = 0
regset = [True for n in range(128)]
vartoreg = {}
flag = True
while True:
    while ":" in intermidiatecode[curline]:
        if "end" in intermidiatecode[curline]:
            outfile.write("end\n")
            exit()
        else:   outfile.write(intermidiatecode[curline]+"\n")
```

```
        curline += 1
    print((curline,intermidiatecode[curline]))
    op, arg1, arg2, res = intermidiatecode[curline].split(",")
    val = 0
    if "+" in op:   val = simpleoperators("ADD", arg1, arg2, res)
    elif "-" in op:   val = simpleoperators("SUB", arg1, arg2, res)
    elif "*" in op:   val = simpleoperators("MULT", arg1, arg2, res)
    elif "/" in op:   val = simpleoperators("DIV", arg1, arg2, res)
    elif ">" in op: outfile.write("COMP "+arg1+"\n")
    else:   outfile.write(op+" "+res+"\n")
    if(val):    break
    curline += 1
outfile.close()
```