MC 68000 SOFTWARE SIMULATOR

PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT
FOR THE AWARD OF DEGREE OF
BACHELOR OF ELECTRONICS AND COMMUNICATION ENGINEERING
OF THE BHARATHIYAR UNIVERSITY
COIMBATORE-641 046

SUBMITTED BY
A. S. Ramesh
K. R. Ramesh
N. S. Senthil
S. Sreenivasa Kumar

UNDER THE GUIDANCE OF
Mr. K. RAMPRAKASH, M.E

Department of Electronics And Communication Engineering
KUMARAGURU COLLEGE OF TECHNOLOGY
COIMBATORE 641 006

1989 - 90

Department of Electronics And Communication Engineering

KUMARAGURU COLLEGE OF TECHNOLOGY

COIMBATORE - 641 006

CERTIFICATE

This is to certify that the project report entitled

MC 68000 SOFTWARE SIMULATOR

Has been submitted by

Mr. ....

in partial fulfilment for the award of Bachelor of Engineering in the ELECTRONICS AND COMMUNICATION ENGINEERING BRANCH of BHARATHIAR UNIVERSITY , COIMBATORE during the academic year 1989 – 90.

Guide

Head of the Dept.

Certified that the candidate was examined by us in the Project work viva – voce examination held on .... and the university register number was ....

Internal Examiner

External Examiner

CONTENTS

CONTENTS

ACKNOWLEDGEMENT

ACKNOWLEDGEMENT

We wish to express our heartful gratitude to Prof. PALANIVELU. our principal, whose initiative and leadership has always been a source of inspiration to us.

We are greatly indebted to Dr.S.PADMANABAN. Department of Electronics and communication Engineering for his keen interest and concern towards our project.

We wish to record our profound thanks to our guide Mr. RAMPRAKASH. Lecturer, Department of Electronics and communication Engineering for his able guidance. His suggestions and guidance at every stage has helped us greatly in this project.

We also wish to thank Prof.SHANMUGAM, HOD, Department of computer technology and informatics for providing us with the computer facilities at the KCT computer centre.

Finally we would like to thank all the staff members, laboratory assistants and non-teaching staff of the Department of Electronics and Communication Engineering and our class mates for their kind co-operation.

SYNOPSIS

SYNOPSIS

The aim of this project is to simulate the instruction set of the MC 68000 microprocessor on an IBM pc compatible computer using an high level language. The 'C' language has been used to develop this software on account of its versatility, and its specific suitability to this application. The simulator can be used as a debugging tool for internal routines, and as a learning aid for MC 68000 software. The input to the simulattor is a MC 68000 program in mnemonic form. Free format input is allowed without labels. The software has been organised as two passes. The first pass scans through the input, detects errors, if any and stores relevant data collected. The second pass simulates each instruction using this stored data. The storage facilities and other facilities offered by a personal computer can be used effectively. Input programs can be stored as files on the floppy diskettle for easy handling. Various debugging facilities such as step execution, display of registers and memory contents and break points are propvided. Editing the input programs can be done using standard editors available on the personal computer.

SPECIFICATIONS

SPECIFICATIONS

INPUT:

Input program is in mnemonic form in free format. Lables are not allowed.

DEBUGGING FACILITIES:

Step execution, break pointing, display of registers and memory contents.

OTHER DETAILS:

Simulated memory : 1 kbyte.

Simulation in user mode of MC 68000

APPROXIMATE COST OF THE PROJECT:

Rs. 2000/-

INTRODUCTION

CHAPTER - 1

INTRODUCTION

Software simulation of a processor is the process of executing programs written in the instruction set of a target processor on a host computer. The simulator goes through the operation cycle of the computer, keeps track of the contents of all the registers, flags, and memory locations. When implemented as an interactive facility it is the best design tool, which can be used for debugging internal routines, that operate primarily within the microprocessor itself with very little I/O.

The target processor has been chosen to be the MC 68000. The reason for choosing the MC 68000 for this project is that the 68K family of chips namely, the 68000, 68008, 68010, 68020 and the 68030 are becoming popular in India. But there is a shortage of good learning aids for learning these processors' software. The instruction set of MC 68000 is completely upward compatible with the later chips. It is a versatile processor with a rich variety of addressing modes.

The host system has been chosen as an IBM PC compatible. They have become common place and have become a standard tool in many laboratories. Since a large number of such compatibles have been in stalled, the software can be used by a large number of users.

can be inserted or delated anywhere with case. Since free formating is allowed, the programmer need not bother, whether he has started in the correct column and so on. Debugging facility, display of registers and memory contents, display of flags are provided. The error messages displayed are user friendly.

1.3 APPLICATIONS:-

The main application where the simulator can be used is in debugging. Since the error messages are highly user friendly, they state the error exactly enabling easy correction. The simulator does not allow non-reentrant code, which modify themselves, and codes with data accesses from program memory. This is not a limitation, but rather a feature, which will defect such badly written code. Due to the above features, this software simulator package can be used as a learning aid to learn MC 68000 software. Anyone having personal computer system, can learn the 68K assembly language without the need to go in for a 68K based system, thereby saving cost.

CHAPTER - 2

THE TARGET PROCESSOR(MC 68000)

The MC 68000 processor is the first in the 68K series of processors from motorola semiconductor in C. It is a 16 bit processor with a full 32 bit internal architecture. It is a highly versatile processor and provides 14 addressing modes. It has 56 basic instructions. It has same advanced software features which makes it easy to develop system programs to support high level languages. Some of the software features of this processor are discussed in this chapter.

2.1 REGISTERS:

The various registers available in the MC 68000 ard their sizes are given below:

* Eight 32 bit data registers registers designated as D0-D7

* Seven 32 bit address registers designated as A0-A6

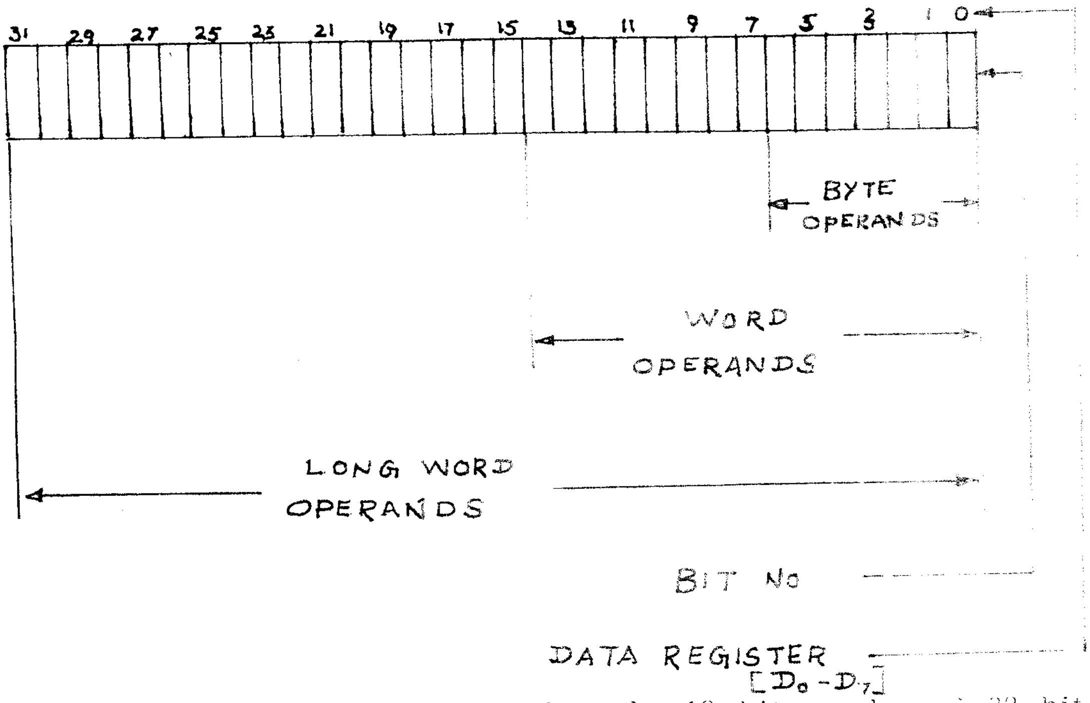
* Two 32 bit stack pointers.

* One 32 bit program counter

* One 16 bit status register.

All the data registers and address registers are general purpose accumulators. In addition to that they can be used as index registers and counters. The eight data registers can be used to handle five basic data types. They are

- * 8 bit bytes
- * 16 bit words
- * 32 bit long words
- * BCD digits
- * 1 bit values or bits



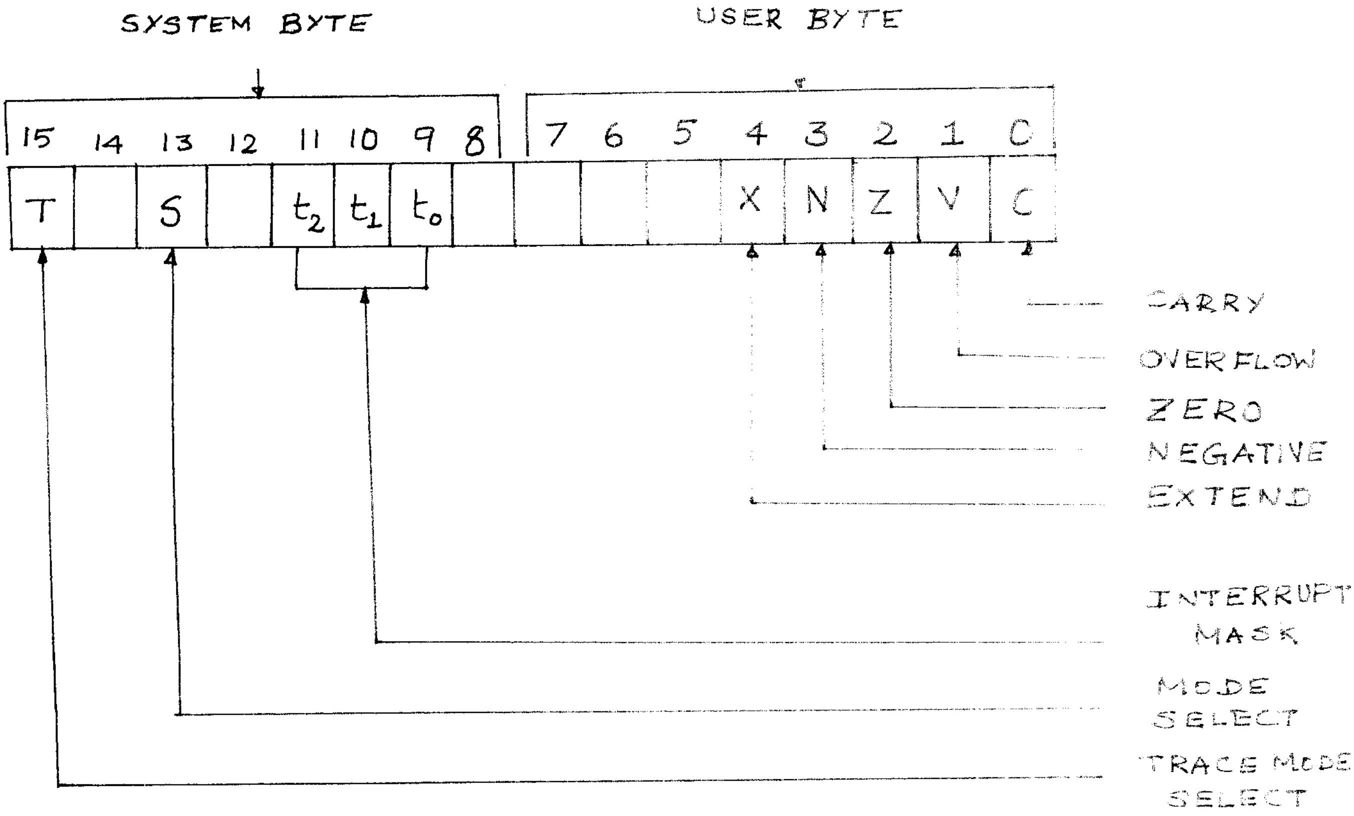
The address registers can handle only 16 bit words and 32 bit long words. Though the program counter and address registers have 32 bits, only 24 bits are used to address the memory. Table summarises the size of the address buses on each of the 68K moterola processors.

Fig (2.2):-

PROCESSOR	BUS WIDTH	ADDRESS SPACE
M C 68000	24	16 MEGABYTES
M C 68008	20	1 MEGABYTES
M C 68010	24	16 MEGABYTES
M C 68012	31	2616 A BYTES
M C 68020	32	4616 A BYTES

M C 68000 Family Address Buses

The 16 bit status register is divided in 8 bytes. The higher byte is called the SYSTEM BYTE and the lower byte is called the USER BITE. The system bite contains the interput mask (3 bits). One bit indicates the mode of operation, namely user or supervisor. Another bit indicates whether the processor is in trace mode. The user byte contains the five flags namely the X, N, Z, V and C. The X flag is similar to the cary flag C in all respects except that same instructions do not effect the extend flag X. It is used to perform multiprecision arithmetic. The N flag is set if the result is negative and reset otherwise. The Z flag is set if the result is zero and cleared otherwise. The V flag is sett if there is an arithmetic overflow. The bit assignment of the status register are shown in fig.2.3



2.2 MEMORY:

The basic memory unit of MC 68000 is a byte (8 bits). It has instructions to access bytes, words (16 bits) and long words (32 bits) from the memory. Words are stored in successive memory locations with thehigh byte appearing first. When the CPU is fetching a word, it fetches the high byte and places it in the high byte position of the appropriate register. It then fetchs the low byte and places it in the low byte position. Similarly long words are stored in four consecutive memory locations. The first two bytes make up the high word. The address bus is 24 bit wide, thus providing a direct addressing range of

Byte addresses can have any value. Word and long wordaddresses must be even numbers. All peripheral devices appear to MC 68000 processor as unique memory locations. In this sense, the processor uses memory-mapped input/output to the peripherals; a program can use the same instruction to move data to a peripheral as it does to move data to a memory location.

2.3 ADDRESSING MODES :

There are fourteen addressing modes in the MC 68000. Their description is as follows.

INHERANT ADDRESSING :

The processor knows which address to use, from the operation code itself. The operation code for these instructions are complete by themselves.

eg: NOP, RTE, etc.

DATA REGISTER DIRECT:

The addressing mode requires that the operand should reside only in a data register. There is no need to refer to operands held in memory.

eg: MOVE D2, D5 etc.,

ADDRESS REGISTER DIRECT :

This mode requires that one of the operands used be held in a address register.

eg: MOVEA D2, A3 etc.,

IMMEDIATE ADDRESSING :

In immediate addressing the data follows immediately after the operation code in memory. The effective address is simply the contents of the program counter after fetching the operation code. It has byte, word and long word immdiate addressing. In immediate addressing, the # symbol, precedes the data. The data may be hexadecimial (or) decimal digits. Presence of sign indicates a hexadecimal data.

eg: ADD. B # $ 7B, D2

ADD. B # 78, D2.

ABSOLUTE SHORT ADDRESSING :

In this mode, the low order half of the effective address follows the pcode in memory. The high order half of the effective address is obtained by extending the sign bit of the low order half of the address. If the most significant bit of the lower order half is a 1, then the higher order half will be FFFF. Otherwise it is 0000. Therefore possible address generated is in the range 00000000 to 00007F FFF (or) through FFFF8000 to FFFFFFFF.

eg: Sub. B $ 7000, D3

Sub. W $ 8F00, D5

ABSOLUTE LONG ADDRESSING :

The effective address occupies two words of program memory immediately following the opcode. High order half of the effective address is in the first word.

eg: Move. L $ F20A210, D2

ADDRESS REGISTER INDIRECT ADDRESSING:

The address of the operand to be used with the instruction is held in one of the address registers. The register indirect addressing is specified by placing the address register. Specification with in the paranthesis.

eg: Move B (A3), DO.

ADDRESS REGISTER INDIRECT WITH POST - INCREMENT.

In this type of addressing the address register is specified within the paranthesis which is followed by a plus sign. After performing the operation, the contents of the address register will be incremented by one, two or four depending on the size of the operands. This is useful in processing arrays, strings or lists.

eg: Move B (AO) +, D2.

ADDRESS REGISTER INDIRECT WITH PRE - DECREMENT :

In this type of addressing mode, the address register is specified with in the paranthesis, which is preceded by a minus sign. The contents of the register is decremented by one, two or four and then the operation is performed. This is also useful in processing of arrays, strings or lists.

eg: Move.B - (A 2), D 4.

ADDRESS REGISTER INDIRECT ADDRESSING WITH DISPLACEMENT :

This is specified with a displacement preceding the address register which is in paranthesis. The effective address is calculated by adding the displacement to the contents of the address register. The displacement is treated as a sign number.

eg: Move.B $ 1200 (AO), D3
Move. B $ 9020 (A1),D4

ADDRESS REGISTER INDIRECT WITH AND DISPLACEMENT:

In this type, the register specification is preceded by the offset value. The address register and the index register are enclosed within the paranthesis and preceded by the off set. Following this the destinsation register specification is given. The effective address is calculated by adding the contents of the address register, the index register and the displacement.

The index register is sign extended and treated as a signed number. The displacement is also treated as a sign number.

There is another type of indexed addressing, where the entire longword content of the index register can be used by appending a period followed by the letter 'L' to the index register specifications as shown below:

eg:Move.B $ 12 (A2,D5), D2
Move.B $ A2 (AO,A1.L), D5.

PROGRAM COUNTER RELATIVE WITH DISPLACEMENT:

A signed displacement or offset from the program counter is provided in the instruction, which when added to the PC value gives the effective address.

eg: Move. B $ 25 (PC), D6

" C " LANGUAGE
&
THE HOST SYSTEM

CHAPTER - 3

THE 'C' LANGUAGE AND THE HOST SYSTEM

3.1 INTRODUCTION:

This chapter discuss the various special feature of C language, in particular the feature of this language, which resulted in choice of this language for this project.

3.2 HIGHLIGHTS OF THE "C" LANGUAGE:

'C' is a high level language, but it is frequently referred to as middle level language because of its close association with system and systems programming. It is highly structured language. It is function oriented. Function called and returns are simple and arguments can be passed freely to the called function and the result can be returned to the calling function. External variables are also allowed. Different data type and storage classes are allowed. It offers many simple and efficient control flow constructions. It has got a large collection of highly useful functions in its standard library. In addition it can be expanded indefinitely by adding new user defined functiins. It also provides pointers, allows address arithmetic, file handling bitwise operations, structures, unions and their relevance to this project.

3.3 SPECIAL FEATURES OF 'C' :

'C' has large set of standard library functions for handling string which provide for comparing strings, copying strings and so on. This string string comparing function is useful in searching the mnemonic table to find a match as can be seen from the program listing.

It allows easy handling of arrays. It also allows pointer variables, arrays can be accessed by either using pointers or by indexing, thus giving a lot of flexibility in writing the program. It addition pointers have been extensively used in the program, especially in the effective address generating functions.

'C' also offers facilities to open a file, close a file and read from the file in any manner by using file pointers through functions in its standard library. This feature of 'C' leads to easy passing of the input file.

'C' allows bit wise operators in addition to the other standard operators. Bit wise anding, oring, exoring, complementing operators are standard in 'C'. It also has left shift and right shift operators which can be used to shift any arbitrary no of steps. These features of 'C' leads to easier simulation of various instructions. Especially, the bit manipulation instruction of 68000 can be simulated much more easily than in any other language.

'C' allows a concept called structure which allows associated data to be handled as a single entity and the data need not be of same type. This has been used to store the data collected in pass 1.

SIMULATOR IMPLEMENTATION

CHAPTER - 4

SIMULATOR IMPLEMENTATION:

4.1 INTRODUCTION

The simulator software has been organised into two passes pass 1 and pass 2. Pass 1 scans through the input, detect errors if any and collects all the necessary data. Pass 2 does the actual simulation.

4.2 DESIGN OF PASS I:

4.2.1 INPUT PARSING ALCORITHM:

Input parsing is the main function of pass 1 of the simulator. The input instruction has to be split into various components namely mnemonic and and operand fields. In this software development, the algorithm chosen for parsing is sequential in nature. The first instruction read from the first column. Blank spaces are skipped, labels are not allowed. therefore the first non-blank character is the first character of the mnemonic. On identifying this the entire mnemonic is read an stored in the struction. The size information at the end of the mnemonic is also extractedand stored in the structure. Next once again blanks are skipped. If the end of the line is encountered, the parsing of the current line stops. Otherwise, the first character, which is encountered after the blanks is examined. Depending on that character, a preliminary prediction of the addressing mode is done and the subsequent characters are examined to validate the assumption. If the subsequent characters are as it should be, then no errors are detected. The data, address register numbers, indirect address register numbers specified, the displacement, the index register, the direct addresses, and other details are collected and stored in the structure. In the process the exact addressing mode is determined and the corresponding code is stored in the structure. After the end of the scanning of the first operand, if a comma is encountered, the same parsing routine is re-entered again to get the second operand details. After parsing one instruction completly, the next instruction from the input file is read in and is parsed in the same manner. This process is continued till the end of file (EOF) is reached. On reaching EOF, the pass 2 is initiated.

The flowchart for pass 1 is shown in the figure 4.1

4.2.2 IMPLEMENTATION OF PASS I IN 'C'

Pass I has been implemented in a modular manner. One function has been written for parsing with a multi way decision structure embedded in it. This decision making is based on the first character of the operand. This function makes use of various other functions for specialised tasks. All the data that is collected during pass 1 is stored in an array of structure. The first element of the array corresponds to the first instruction, the second element to the second instruction and so on. Each element of the array is a structure containing data of different types. One separate function has been written to split the mnemonic and to get the size of the data and store them in the structure.

4.3 DESIGN OF PASS II

4.3.1 SIMULATION ALCORITHM:

The algorithm used to simulate the instruction set is quite simple. Various registers of the processor are simulated by declaring variables of same type and size and a small memory of 1 kb size is also simulated by declaring an array. The flags are also simulated similarly. For this algcrithm to work an important data base namely mnemonic table is needed which consists of all the mnemonics of 68000 in alphabetical order. For each instruction, the following procedure is adopted.

First, the mnemonic is taken from the structure where it has been stored, then a binary search on the sorted table is made to find this mnemonic. If a match is found then control is transferred to the appropriate function which will simulate the effects of the intruction. This is repeated for all the mnemonics.

Prior to calling the routines for simulation, the effective addresses of the two operands are generated and is made available to the simulating functions. So that they will be able to simulate the instructions straight away.

The flow chart for pass 2 is shown in the figure.

4.3.2 IMPLEMENTATION OF PASS II IN 'C':

Pass 2 is also implemented in a highly modular fashion. The mnemonic table is created using an array of structures. Each array element is a structure of two members. One member is a character array which contains the mnemonic. The other member is the function pointer. This member contains the address of the function which will simulate the effects of the corressponding mnemonic. This table is initialised as an external array of structures as shown in the program listing. The user function pointers has greatly simplified the implementation.

The searching of the table is done by another function. It receives as its arguments the string to be searched for, the table to be searched match is found it returns the index of the location in the table. This index is used to extract the function pointer associated with this index and a function code is given at this stage, the searching algorithm is binary search. The effective addresses are generated by another function. To simulate an instruction the operands are needed. In this software the operands are not supplied, but the pointers to the operands are supplied. The pointer points to the location of the operand in the host systems memory from where it is accessed. This is another specific feature of 'C' which simplify the implementation.

In addition, functions for getting a byte, word (16 bits), long words (32 bits) from an address and for storing, byte, word and lond word at a given address have been written.

4.4 EX: SIMULATION OF ADDI INSTRUCTION:

The flowchart for the simulation of the instruction ADDI (Add immediate) is given in fig. To simulate the instruction ADDI, two operands are required. These operands are not supplied directly. They are obtained from the corresponding pointers to the operands. Then the flags are initilized to zero. For the first operand, there should be the addressing mode "Immediate data" and the addressing modes "Data register direct", "Pc indirect with displacement", "Pc indirect with index and displacement", "Immediate data" should not be there for the second operand. If the above condition is satisfied, the subroutine "SADD - function to simulate subtract or add" is called and the simulation of instruction ADD is done. After the instruction is simulated, the flags and results will be displayed

4.4. EXECUTION OF SAMPLE PROGRAM: (1)

```
  MOVE.L DO  ,D1
ADDI.L # $6  ,  D1
MOVE.L D1  ,  D2
SUB Q.L # $ 4 , D2
BNE $ FFE2  (PC)
```

Let the data register D0 initially has hexadecimal number 8.

OPERATION SEQUENCE:

For the above program, initially the File is opened and the program is entered in it. Then the data is collected From the input file and stored in the structure St[ ]

MOVE is stored in the mnemonic mn[ ]

'L' indicates the long word (8 bit) and is stored in SIZE.

D0 and D1 are data registers (32 bit)

Similarly all other instructions are stored in the input data structure

PROGRAM EXECUTION:

The mnemonics of the first instruction is retrieved . then, a binary search on the stored table is made to find this mnemonic. If a match is found then control is transferred to the appropriatte function which will simulate the effects of the instruction. The same procedure is handled for all mnemonics.

EXECUTION OF STEP - 1:-

INSTRUCTION:- MOVE L D0 D1

Since both operands D0 and D1 are data registers, the addressing mode is data register direct. This instruction moves the data of word length W from D0 to D1. D0 has hexadecimal 8.

TRACE MODE :

A C C	D0	D1
00000008	00000008	00000008

Flags.

N	Z	V	C	X
0	0	0	0	0

EXECUTION OF STEP - 2

INSTRUCTION: ADDI . W # $ 6, D1

Here # indicates the immediate data

$ specifies the hexa decimal number

D1 specifies the data register - 1

L indicates the long word length

Since the hexadecimal data '06' is directly added with the contents of the data register D1, the addressing mode is immediate data. Finally the result is stored in the data register D1.

TRACE MODE:

A C C	D0	D1
0000000 E	00000008	0000000 E

Flags

N	Z	V	C	X
0	0	0	0	0

EXECUTION OF STEP 3:

INSTRUCTION : MOVE. L D1 D2

Here L specifies long word [32 bit]

D1 specifies the data register 1

D2 specifies the datta register 2

ADDRESSING MODE:

Data register direct

In this step execution, the data register - 1 contents U $ 14, are moved to the data register -2. Here also, the operands D0 and D1 are data registers, the addressing mode used here is data register direct. The trace mode displays the register contents as follows.

TRACE MODE:

```
A C C          D0
0000000 E      00000008
D1, D2
0000000 E
```

FLAGS:

```
N   Z   V   C   X
0   0   0   0   0
```

EXECUTION OF STEP 4 :-

INSTRUCTION :- SUB Q L # $ A D2

SUB Q - Subtract Quick.

This instruction subtracts the immediate data from the destination operand D2. Therefore the operand D2, ie $E is subtracted from the immediate data $ A and the result is stored in the destination register D2.

TRACE MODE :

```
A C C          D0             D1
00000004       00000008       0000000E
D2
00000004
```

FLAGS:-

```
N   Z   V   C   X
1   0   0   0   0
```

EXECUTION OF STEP 5 :

INSTRUCTION: BNE $ FFE 2 (PC)

BNE specifies "Branch on Negative"

FFE2 represents "Address of the memory location where the next instruction or the data can be obtained".

This instruction branches to the program counter if the 'N' flag is set. The address FFE2 is transferred to the program counter.

DEBUGGING FACILITIES

CHAPTER - 5

DEBUGGING FACILITIES

5.1 INTRODUCTION

This chapter discusses in brief how the various debugging facilities have been implemented in the software. The main features discussed here are breakpoints, step execution, registers display, memory contents.

5.2 STEP EXECUTION:

Step by step exacution is a very convenient technique for debugging small programs. In this technique the program is executed one instruction at a time. After execution of each program it displays the contents of registers and flags and waits. The next instruction is exceuted only when the user depresses some key. This is also called trace mode of operation. In fact the MC 68000 has mode of operation called trace mode where a trap is inserted after every instruction by the processor itself, initializing exception processing.

Here in this software, step by step execution can be imple mented easily. After the simulation of each instruction, the contents of flags and registers are displayed and the program goes into a loop waiting for a key to be depressed. Once the key is depressed the program comes out of the loop and stimulates the next instruction (1).

A simple flow chart shown in fig 5.1 illustrates this implementation.

5.3 BREAK POINTS:

Step by step execution is very tedious and not useful for large programs. For such programs breakpoint testing is used. In this technique breakpoint can be set by the user. Breakpoints are nothing but places where the program will automatically half or wait so that the user can examine the current status of the system. The program will not continue until the user orders its resumption.

The breakpoint facility is implemented quite easily. The breakpoint which were set by the user are stored in an array. After ececution of each instruction, the current value of the program counter is compared with the elements of the array till a match is found or till the array ends. If a match is found the program is temporaility halted and register contents and flags are displayed. The program resumes operation after the operator resumes the key (1).

A simple flowchart detailing this mechanism is show in figure. 5.2.

5.4 DISPLAY OF REGISTERS AND MEMORY CONTENTS:

This can also be implemented easily. A routine for displaying all the registers is written and is called whenever the user wishes to do so. Similarly a routine to display 16 bytes of memory starting from the location specified by the user is writtened and is called whenever the user wishes to see memory contents.

The flowchart detailing this mechanism is shown in figure 5.3

CONCLUSION

CHAPTER: 6

CONCLUSION

6.1: MERITS OF SOFTWARE SIMULATION:

The main merits of software simulation are listed below.

1. It can provide a complete description of the status of the computer, since the simulator program is not restricted by the micro processor chip pinout limitations or other characteristics of the under lying circuitry.

2. It can provide break points, dumps, traces and other facilities without using any of the simulated processor's memory space or control system. These facilities will therefore not interface with the user program.

3. Programs, starting points, and other conditions easy to charge.

4. All the facilities of a personal computer, including peripherals such as printer, magnetic disk storage and software are available to the microprocessor designer. For example, screen editors can be used to edit the source program, floppy diskettes to store source programs as files, and printers to get hard copies.

6.2: DEMERITS:

On the other hand, the simulator is limited by its software base and its separation from the real microprocessor.The major limitations are given below:

1. The simulator cannot cope with timing problems, since it operates at less than real-time execution speed. The simulator is quite slow.

2. The simulator cannot model the I/o section exactly, since it cannot represent external hardware or interfaces accurately.

The simulator represents the software side of debugging; it has typical advantages and limitations of a wholly software based approach. It can provide insight into program logic and other software problems, but often cannot help with timing, I/o, and other hardware problems.

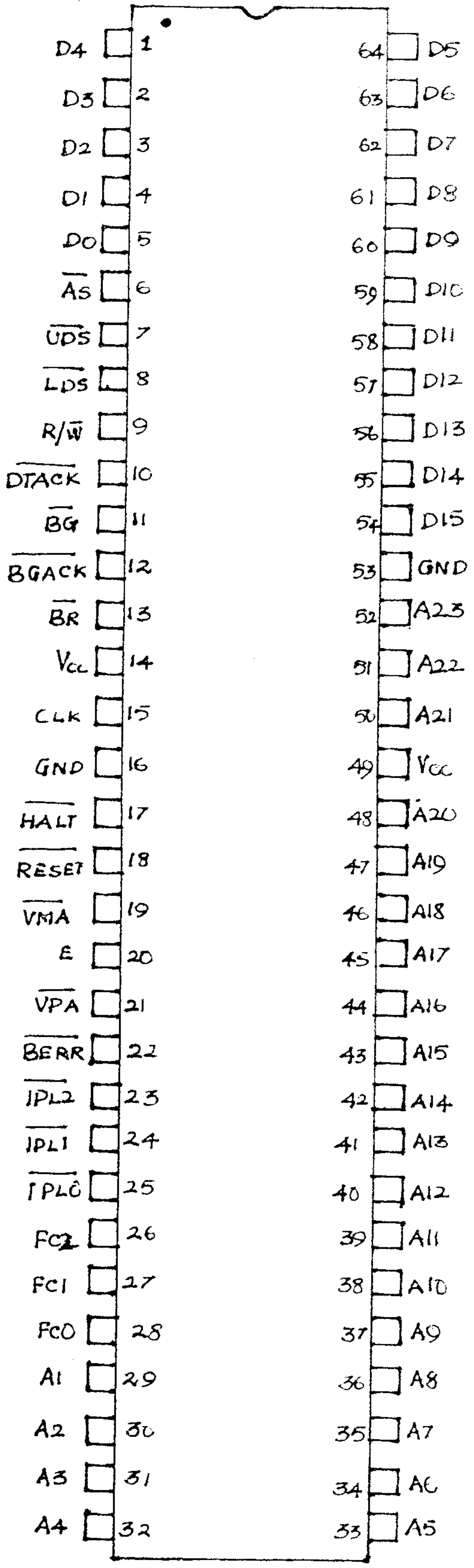
6.3: SUGGESTIONS FOR FURTHER IMPROVEMENTS:

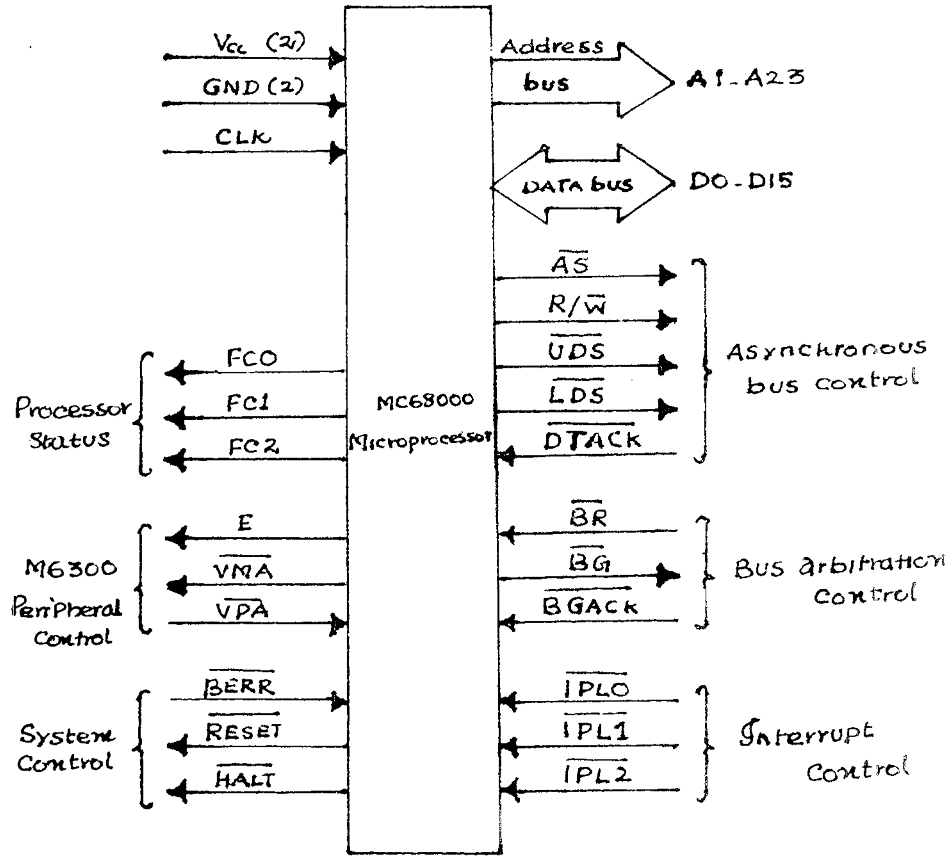
The project has got good scope for further improvements. The simulator written now does not support labels. A pre-pass I stage can be extended to the supervisor mode of execution of 68000 also. Improvements can also be made in the display modules, by providing

BIBLIOGRAPHY

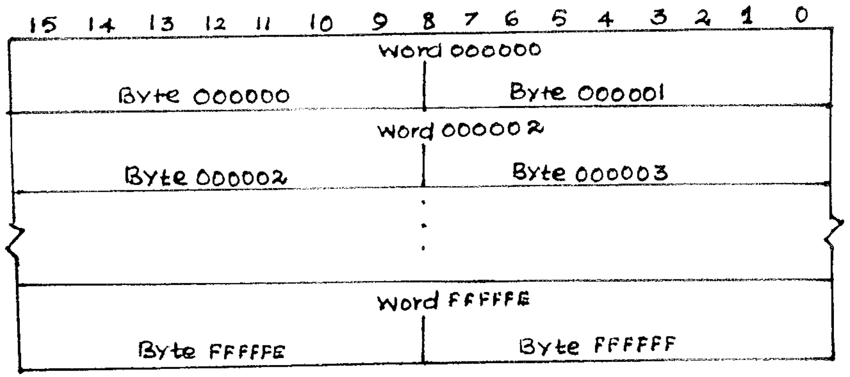
MC 68000 DETAILS

Pin assignment

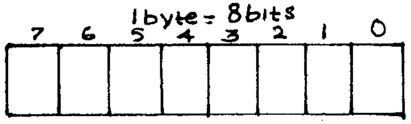




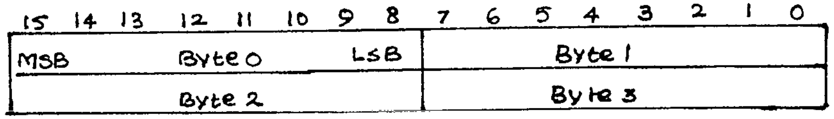
Vcc (2)
GND (2)
CLK
Address bus
A1-A23
DATA bus
D0-D15
AS
R/W
UDS
LDS
DTACK
Asynchronous bus control
FC0
FC1
FC2
Processor Status
MC68000 Microprocessor
E
VMA
VPA
M6300 Peripheral Control
BR
BG
BGACK
Bus arbitration Control
BERR
RESET
HALT
System Control
IPL0
IPL1
IPL2
Interrupt Control



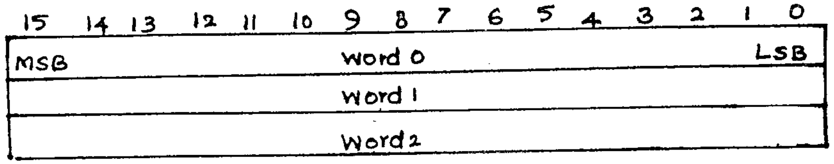
BIT DATA
1byte = 8bits

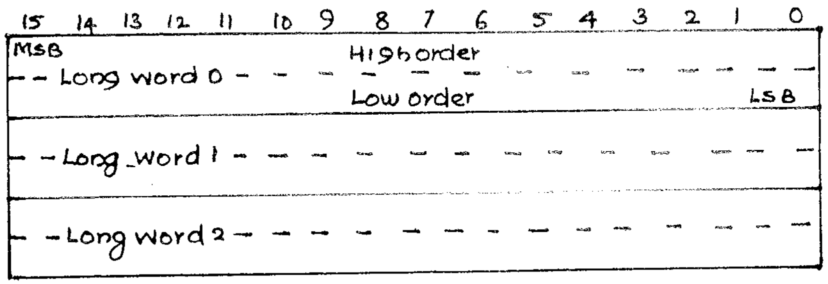


Integer data
1byte = 8bits

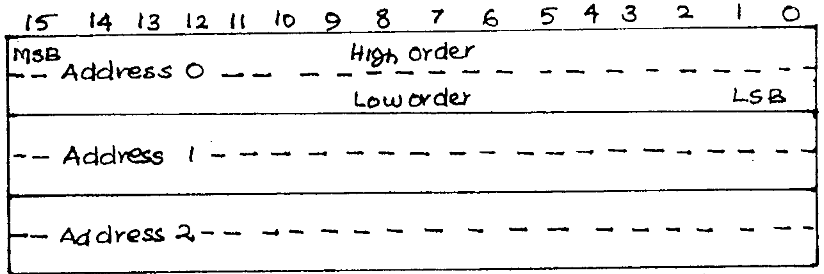


1 word = 16bits



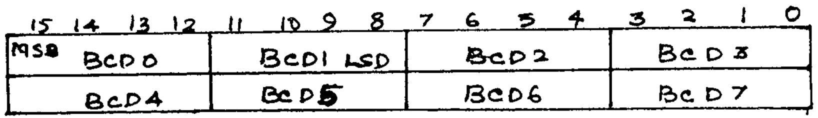


Address
1address = 32 bit



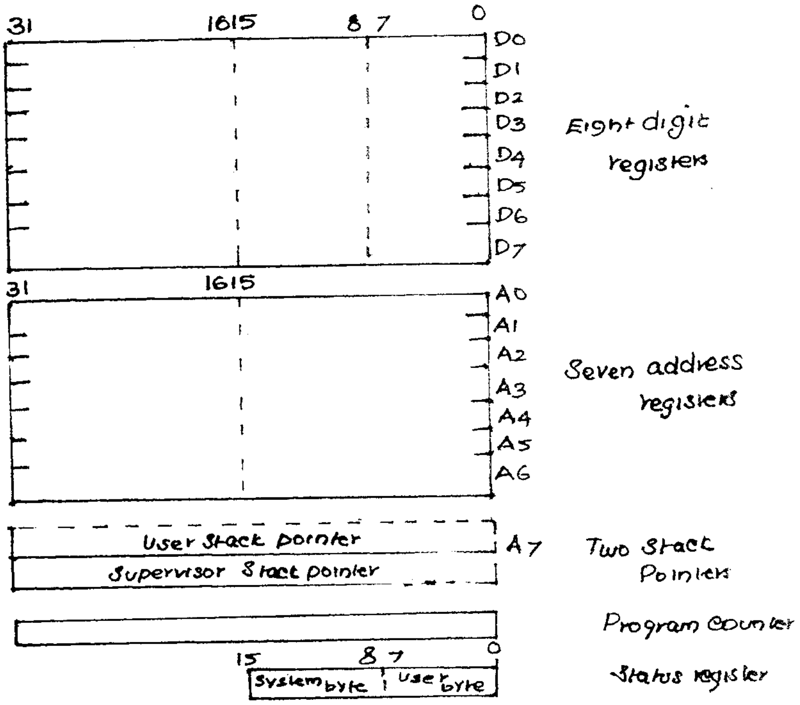
MSB, most significant bit LSB least significant bit

Decimal data
Two binary coded decimal digits = 1 byte



MSB, most significant digit - LSB least significant digit

Programming model.



INSTRUCTION TYPE	VARIATION	DESCRIPTION
ADD	ADD	Add
	ADDA	Add Address
	ADDQ	Add Quick
	ADDI	Add Immediate
	ADDC	Add With Carry
AND	AND	Logical And
CMP	CMP	Compare
	CMPA	Compare Address
	CMPM	Compare Memory
	CPI	Compare Immediate
EOR	EOR	Exclusive Or
	EORI	Exclusive Or Immediate
MOVE	MOVE	Move
	MOVEA	Move Address
	MOVEQ	Move Quick
	MOVE from SR	Move from Status Register
	MOVE to SR	Move to Status Register
	MOVE to CCR	Move to Condition Codes
	MOVE USR	Move User Stack Pointer
NEG	NEG	Negote
	NEGX	Negote With Extend
OR	OR	Logical Or
	ORI	Or Immediate
SUB	SUB	Subtract
	SUBA	Subtract Address
	SUBI	Subtract Immediate
	SUBQ	Subtract Quick
	SUBX	Subtract With Extend

MNEMONIC	DESCRIPTION
ABCD	ADD Decimal with Extend
ADD	Add
AND	Logical AND
ASL	Arithmatic Shift Left
ASR	Arithmatic Shift Right
BCC	Branch Conditionally
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Always
BSET	Bit Test and Set
BSR	Branch to Subroutine
BTST	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
DBCC	Test Cond, Decrement and Branch
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive OR
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL	Logical Shift Left
LSR	Logical Shift Right
MOVE	Move
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral Data
MULS	Signed Multiply
MULU	Signed Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NOP	No Operation
NOT	One's Complement
OR	Logical OR
PEA	Push Effective Address
RESET	Reset External Devices
ROL	Rotate Left without Extend
ROR	Rotate Left Without Extend
ROXL	Rotate Left with Extend
ROXR	Rotate Right with Extend
RTE	Return from Exception
RTR	Return and Restore
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
S_{CC}	Set Conditional

MNEMONIC	DESCRIPTION
STOP	Stop
SWAP	Swap Data Register Halves
SUB	Subtract
TAS	Test and Set Operand
TRAP	Trap
TRAPV	Trap on Overflow
TST	Test
UNLK	Unlink

MACHINE COMMANDS

MACHINE COMMANDS:

1. CTRL KB	-Marks the beginning of a block
2. CTRL KK	-Marks the end of a block
3. CTRL KW	-Copies the block elsewhere on the disk
4. CTRL KY	-Delates the block
5. CTRL P	-Connects/disconnects the printer
6. CTRL C	-Comes out of any execution
7. CTRL N	-Inserts a line
8. CTRL Y	-Delates a line
9. CTRL G	-Delates a character
10.CTRL E	-Renames a file
11.CTRL O	-Copies a file

KEYS:

1. DEL	-Deletes left character
2. INS	-Inserts character
3. F1	-Repeats character entered previously
4. F3	-Repeats words entered previously
5. F2	-Saves the program in 'C'

HARDWARE - SOFTWARE SPECIFICATIONS

HARDWARE - SOFTWARE SPECIFICATIONS:

HARDWARE:

1. Processor (CPU)	-Intel 8088, 4.77 MHZ.
2. Memory (RAM)	- 256 KB minimum,expandable to 640KB (all on CPU board) with parity check
3. Floppy drives	-2 x 360 KB (formated) 51/4" drive.
4. Hard disk drives	-Expandable to 2x10 MB or 2x20 MB
5. Monochrome display Size	-14" non-glare
Text resolution	-80 columns x 24 lines
Mounting	-tilt and swivel base
6. Colour display size	-14" non-glare
Text resolution	-80 columns x 24lines
Mounting	-16 foreground & 8 background for text.

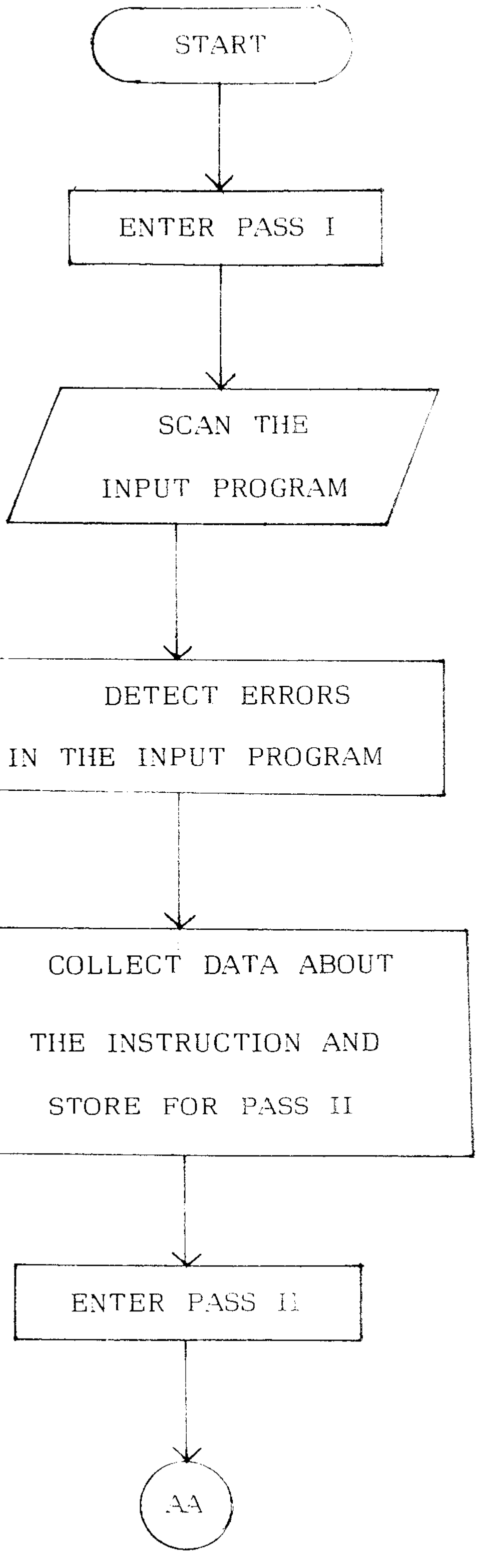
SOFTWARE

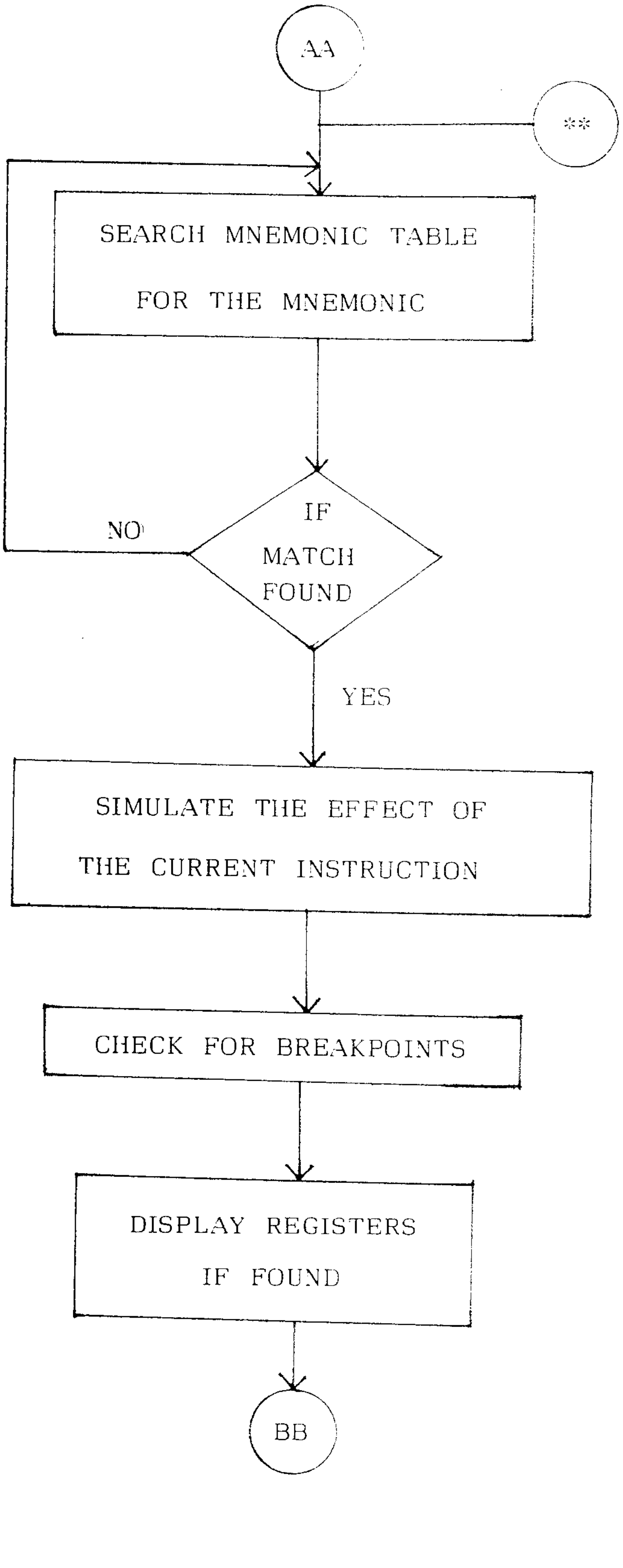
1. Operating System	-IBM DOS, Version 3.3
2. 'C' Compiler	-Turbo C Compiler, Version 2 (TC2)

FLOW CHART

FIGURE: 1.1

FLOW CHART OF THE SIMULATOR:


START
ENTER PASS I
SCAN THE INPUT PROGRAM
DETECT ERRORS IN THE INPUT PROGRAM
COLLECT DATA ABOUT THE INSTRUCTION AND STORE FOR PASS II
ENTER PASS II
AA


AA
**
SEARCH MNEMONIC TABLE FOR THE MNEMONIC
IF MATCH FOUND
NO
YES
SIMULATE THE EFFECT OF THE CURRENT INSTRUCTION
CHECK FOR BREAKPOINTS
DISPLAY REGISTERS IF FOUND
BB

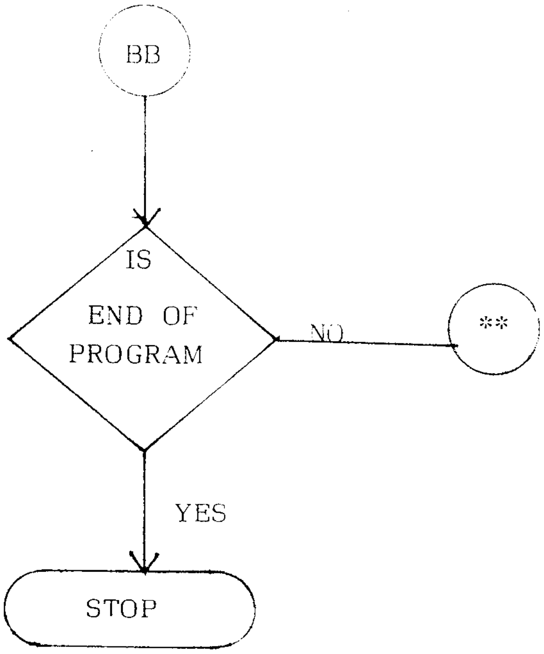
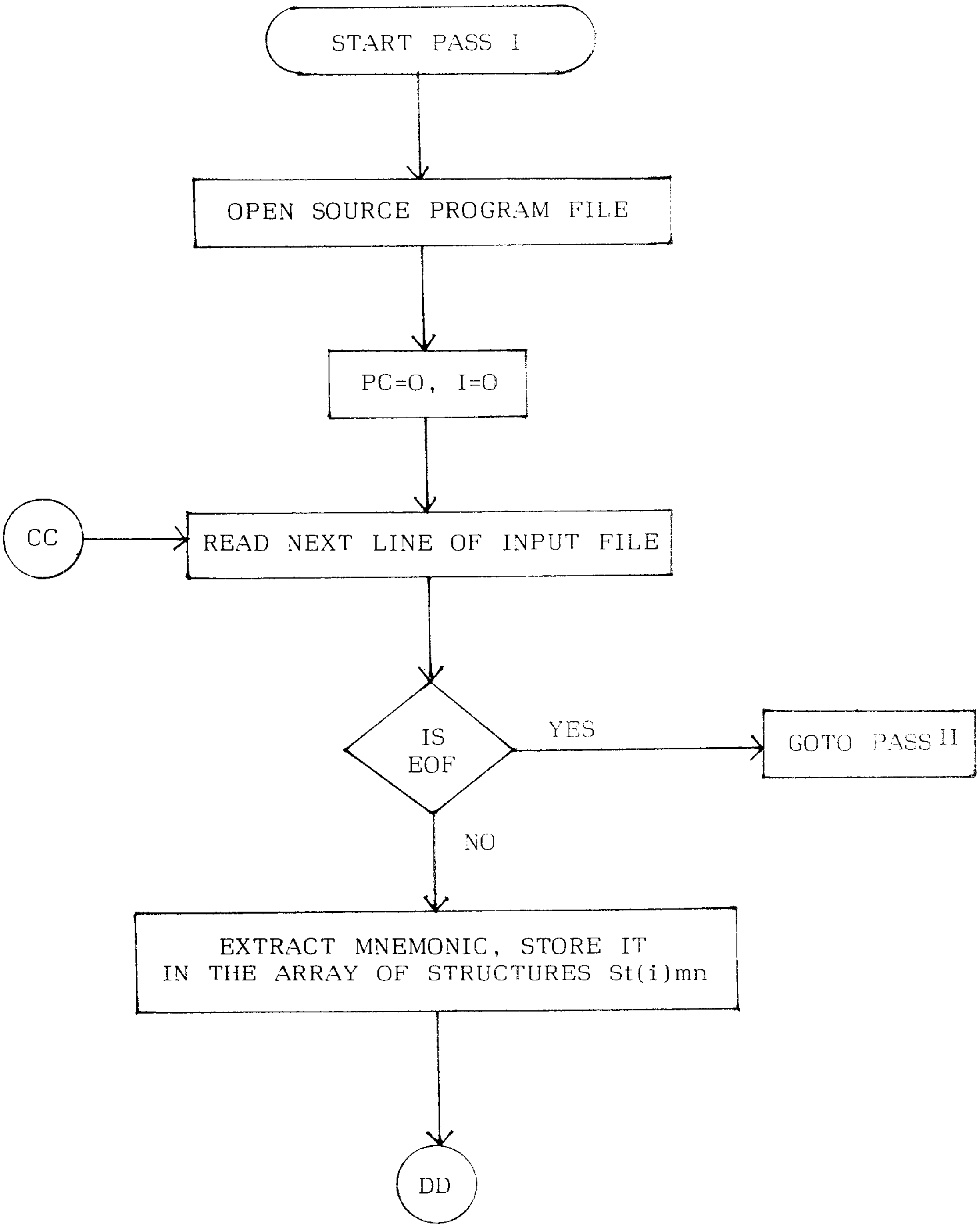
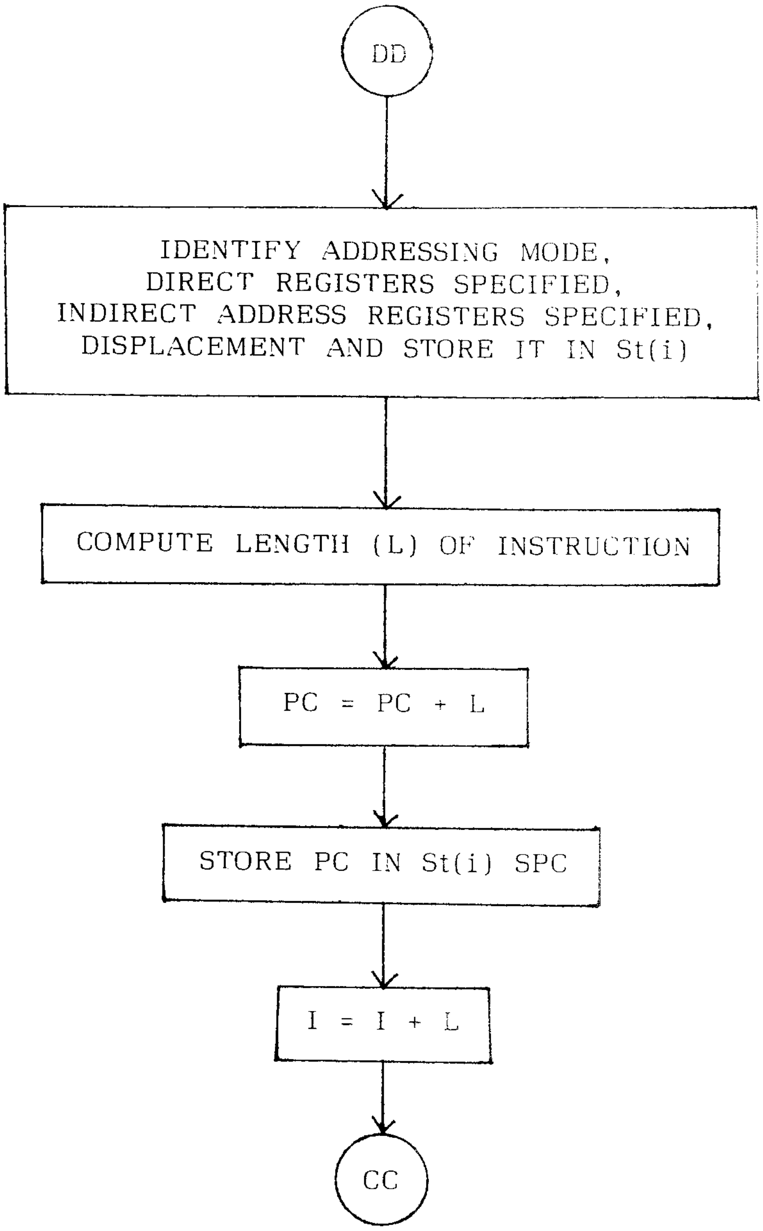

BB
IS
END OF
PROGRAM
NO
**
YES
STOP

FIGURE: 4.1

FLOW CHART OF PASS I :


START PASS I
OPEN SOURCE PROGRAM FILE
PC=O, I=O
CC
READ NEXT LINE OF INPUT FILE
IS EOF
YES
GOTO PASS II
NO
EXTRACT MNEMONIC, STORE IT IN THE ARRAY OF STRUCTURES St(i)mn
DD


DD
IDENTIFY ADDRESSING MODE, DIRECT REGISTERS SPECIFIED, INDIRECT ADDRESS REGISTERS SPECIFIED, DISPLACEMENT AND STORE IT IN St(i)
COMPUTE LENGTH (L) OF INSTRUCTION
PC = PC + L
STORE PC IN St(i) SPC
I = I + L
CC

St[i] - Array of structures

St[i].mn - Stored mnemonic

St[i].spc - Program counter value

FIGURE: 4.2

FLOW CHART OF PASS II :

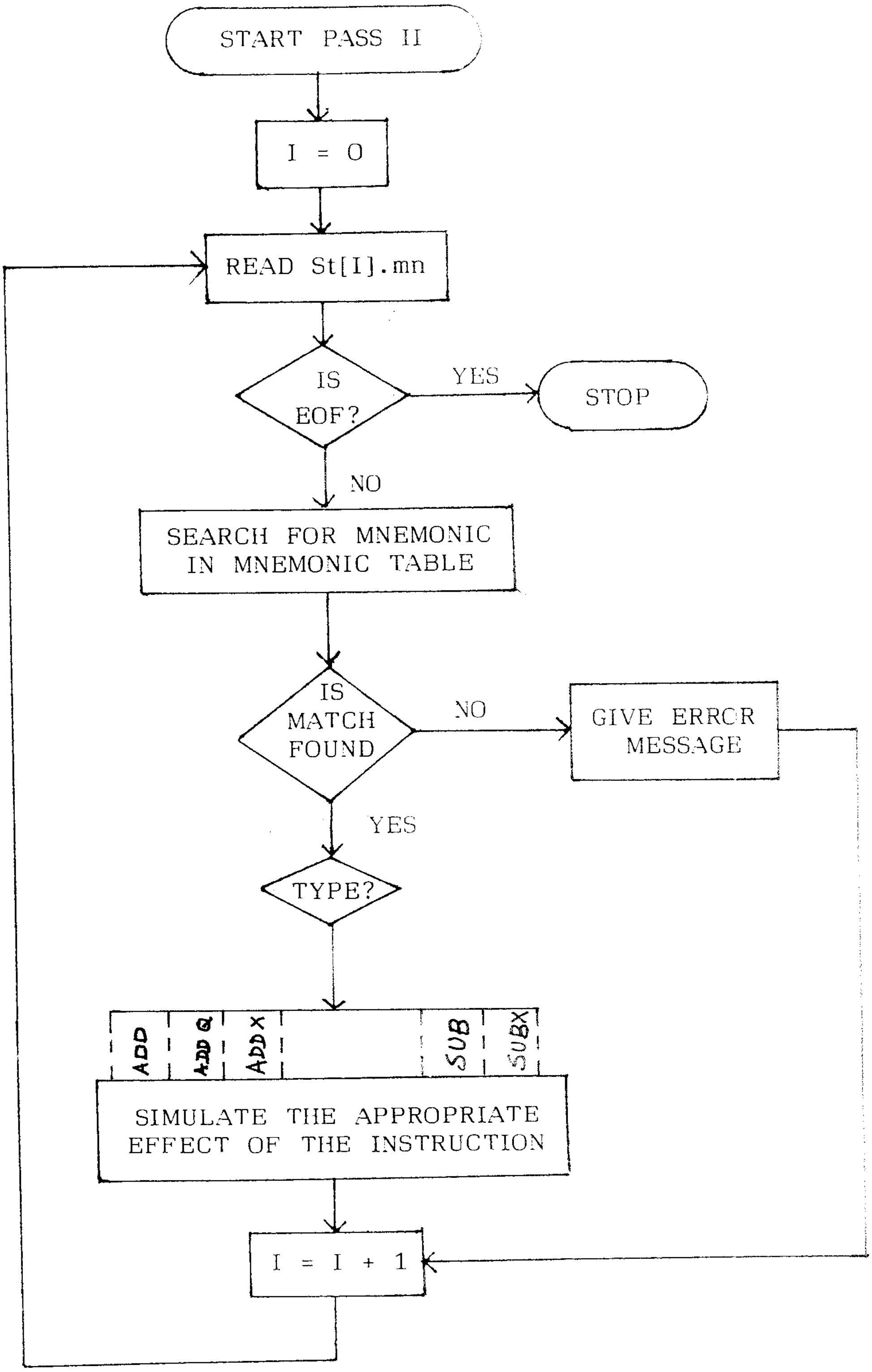
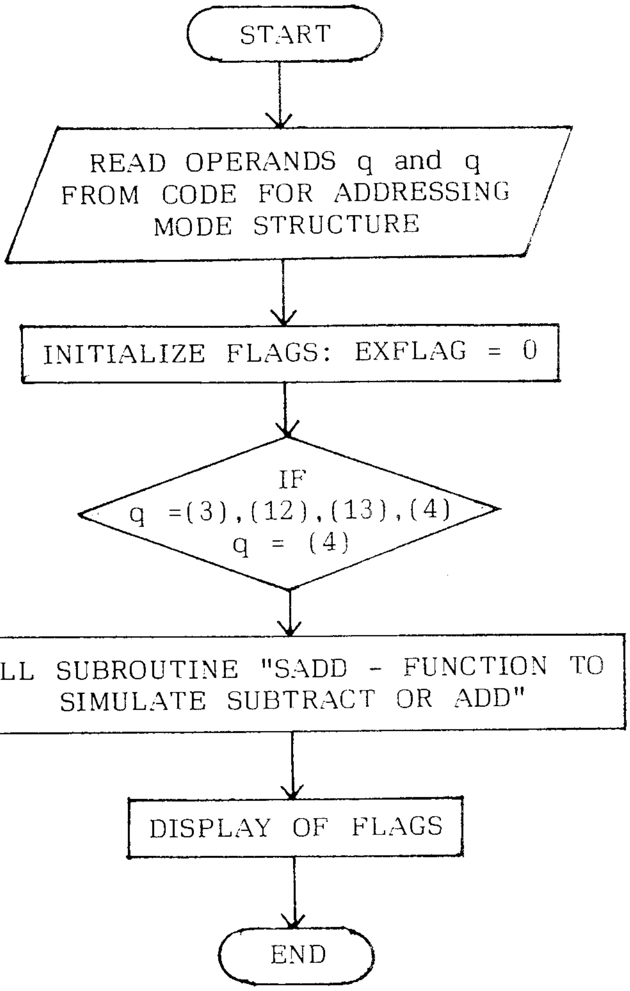


FIGURE : 4.3

FLOW CHART FOR THE SIMULATION OF ADDI INSTRUCTION:



START
READ OPERANDS q and q FROM CODE FOR ADDRESSING MODE STRUCTURE
INITIALIZE FLAGS: EXFLAG = 0
IF
q =(3),(12),(13),(4)
q = (4)
CALL SUBROUTINE "SADD - FUNCTION TO SIMULATE SUBTRACT OR ADD"
DISPLAY OF FLAGS
END

(3) - Data register direct.

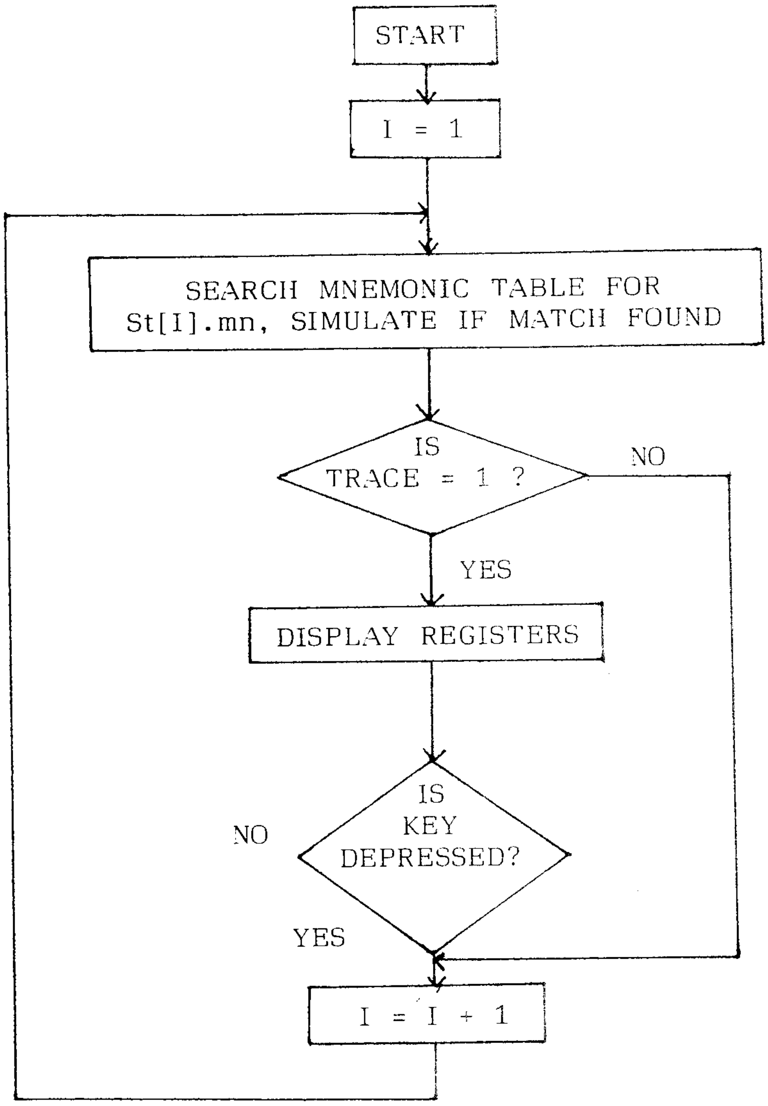
(4) - Immediate data

(12)- PC indirect with displacement

(13)- PC indirect with index and displacement

FIGURE 5.1

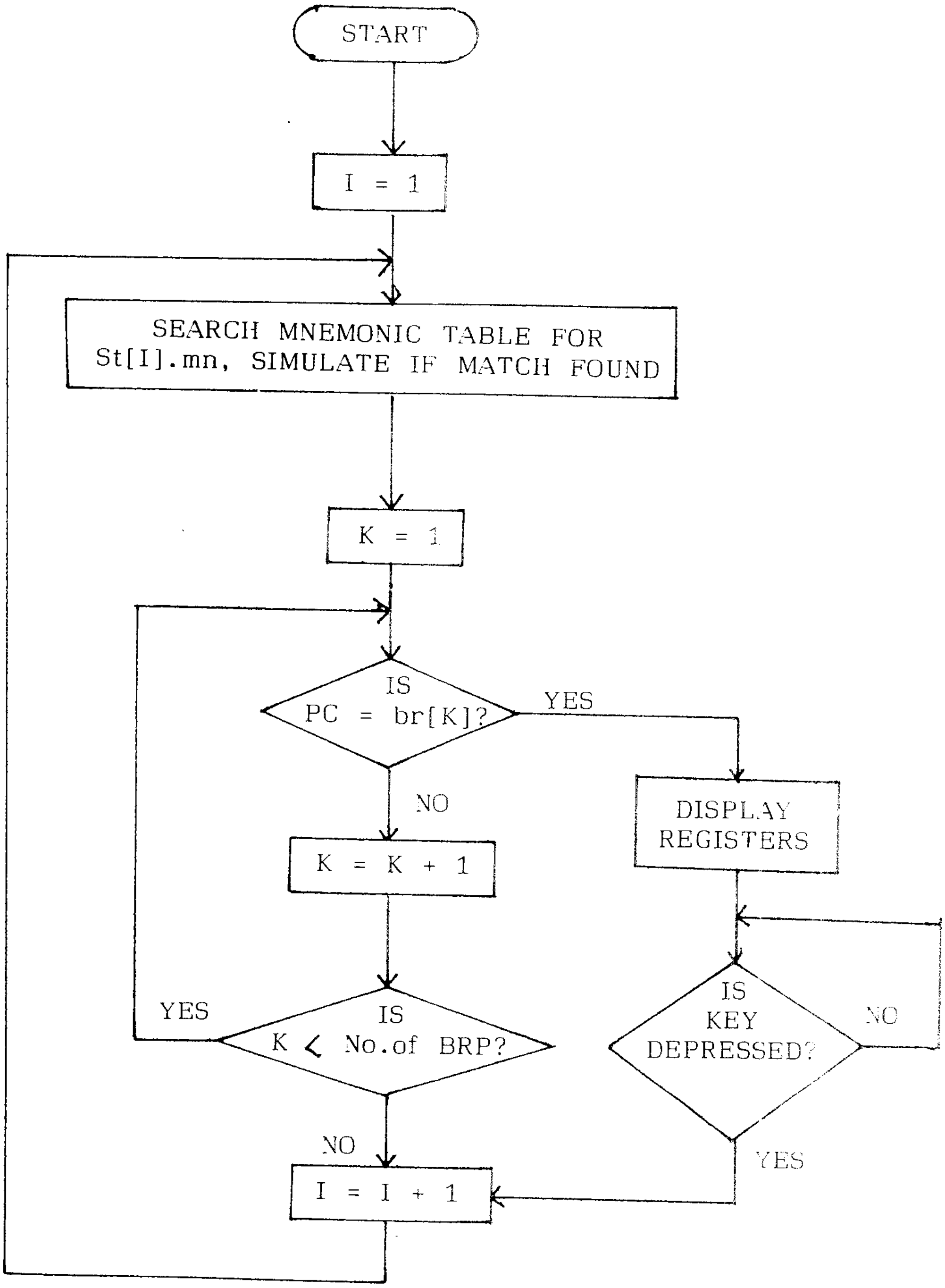
FLOW CHART OF STEP EXECUTION IMPLEMENTATION:



Trace - Step execution flag

FIGURE: 5.2

FLOW CHART OF BREAKPOINT IMPLEMENTATION:

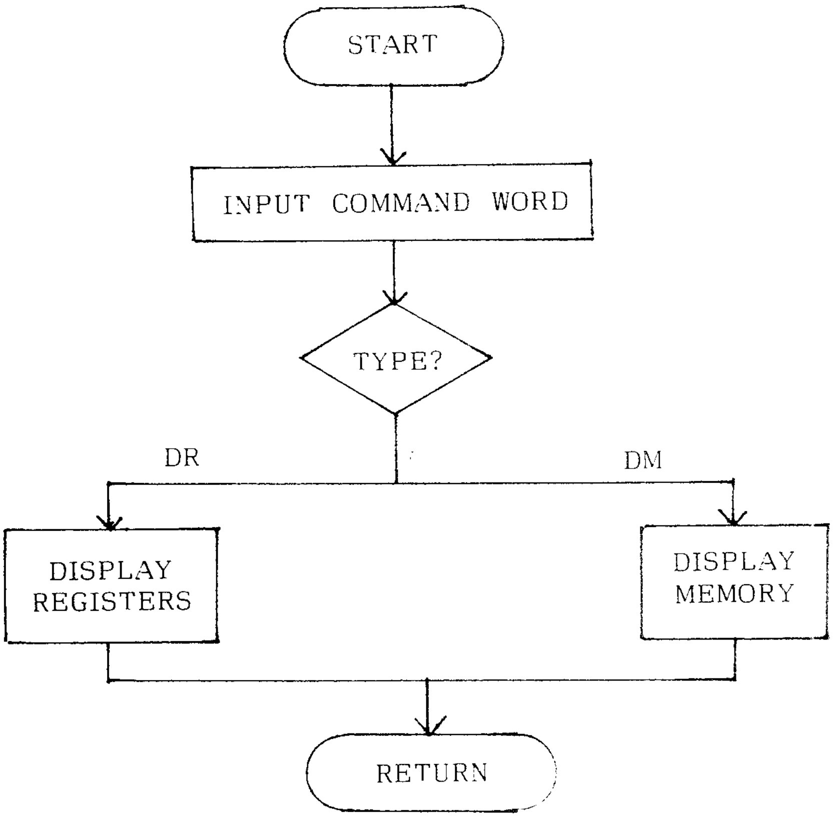


br[K] - Array of breakpoints

No.of BRP - Number of breakpoints.

FIGURE: 5.3

FLOW CHART FOR DISPLAY OF REGISTERS AND MEMORY



p-1274

PROGRAM LISTING

```
#include <ctype.h>
#include <stdio.h>  /*External declarations*/
     int noprs, oprl[3],copr[3],di[3],ai[3];
     unsigned long int opr1,eaddr[3];
     int dm[8],am[8];
     unsigned long int d[8],pc,tpc;
     unsigned long int a[8]={
            0x00000000,
            0xaaaa,
            0xbbbb,
            0xcccccccc,
            0xdddd,
            0xeee,
            0xffffffff,
            0x789
              };
            unsigned int sr;
            char mn[15];
            int ft;
            int adlen;
            int fx,fn,fz,fv,fc; /* flags */
            int vfx,vfn,vfz,vfv,vfc;
            int siz;
            int exflag;
        /* Input data is stored in this structure */
                struct inpstr {
                  char mn[15]; /* Mnemonic is stored here */
                  int snoprs;  /* No. of operands */
                  int soprl[3]; /* length of operand field */
                  int scopr[3]; /* code for addressing mode */
                  int  sdi[3]; /* Data  register  number-direct
addressing */
                  int sai[3]; /* Address register number-direct
*/
                  unsigned long int sopr1;/* Immediate data */
                  unsigned  long  int seaddr[3];  /*  Effective
address (Absolute adressing) */
                  unsigned  long  int spc; /* PC value  of  the
instruction */
                  int sft[3];
                  unsigned int smask;
                  int  sadinr[3]; /* Indirect address  register
number */
                  int sisdi[3]; /* Index register type; address
or data? */
                  int sind[3]; /* Index register number */
                  unsigned  long int sdisp[3]; /*  displacement
*/
                  int size;  /* whether Byte,  word or longword
*/
                  int sindsi[3];/* Index register size */
                    };
     struct inpstr st[50];
      unsigned short int mem[8096]; /* Memory initialization */
   /*Union containing pointers to operands */
                 union oppn {
                  unsigned short int *mm;
                  unsigned int *r16;
                  unsigned long int *rg;
                   };
              union oppn op[3];
              union oppn dop[3];
    unsigned long int l1,l2,l3,l4;
    unsigned long int br[10];
    unsigned int trace,brno,brflag;
    int fmm[3];
    int abcd(),bclr(),btst(),move();
    int movea(),movep(),moveq(),movem();
    extern int add(),addx(),addi();
    extern int addq(),adda(),sub(),subx();
    extern int subi(),suba(),subq(),bra();
    extern int bcc(),bcs(),beq(),bge();
    extern int bgt(),bhi(),ble(),bls(),blt(),bmi();
    extern int bne(),bpl(),bvc(),bvs();
    int getopr();
    int z;
   /*Mnemonic Table is initialized here */
            struct mntab {
             char monic[15]; /*Menmonic */
             int   (*funsim)();/*Pointer   to   the   function
simulating the  instruction */
              };
             struct mntab motab[]={
                "ABCD",abcd,
                "ADD",add,
                "ADDA",adda,
                "ADDI",addi,
                "ADDQ",addq,
                "ADDX",addx,
                "BCC",bcc,
                "BCLR",bclr,   /*pointer   to   the   function
functions name is bclr();*/
                "BCS",bcs,
                "BEQ",beq,
                "BGE",bge,
                "BGT",bgt,
                "BHI",bhi,
                "BLS",bls,
                "BLE",ble,
                "BLT",blt,
                "BMI",bmi,
                "BNE",bne,
                "BPL",bpl,
                "BRA",bra,
                "BTST",btst,
                "BVC",bvc,
                "BVS",bvs,
                "MOVE",move,
                "MOVEA",movea,
                "MOVEM",movem,
                "MOVEP",movep,
                "MOVEQ",moveq,
                "SUB",sub,
                "SUBA",suba,
                "SUBI",subi,
                "SUBQ",subq,
                "SUBX",subx,
                   };
            /* Similarly for other instructions */
 /*Main program segment starts here */
       main(argc,argv)
```

```
         int argc;
         char  *argv[];
          {
           char *effadd();
           char *mne();
           char *fgets();
           FILE *fp,*fopen();
           char line[50];
           int c,d,e,i,a,n,p,q,g;
           char *u,*v,*w,*x;
           fp=fopen(argv[1],"r");  /* open the assmbly source
file */
           for(i=1;i<50;i++)    /* Read a maximum of 50
instructions */
             {
               if((u=fgets(line,50,fp))==NULL) break;  /*Read
one assembly instruction */
               v=mne(u,i);  /*strip mnemonic and operand size
information */
               w=effadd(1,v,i);    /* Get first operand
information */
               if (ft==3) /* Two operands */
                {
                  effadd(2,w,i);   /* Get second operand
information */
                 st[i].sft[2]=ft;
                  }       switch(ft)  {    /* error flags */
         case 0:
           break;
         case 1:
          printf("number of operands=0\n");
          break;
         case 2:
           printf("unknown format\n");
           break;
         case 4:
           printf("invalid register specification for
operand");
           break;
         case 5:
           printf("wrong format");
           break;
         case 10:
           printf("too long hex address\n");
           break;
         case 18:
           printf("too long displacent for indexed
addressing");
           break;
         case 20:
           printf("too long decimal address\n");
           break;
       }
       st[i].spc=pc; /* Store PC value in the structure */
       if(st[i].snoprs==0)  /* Increment PC depending on the
size of the instruction */
          pc=pc+2;
        else if(st[i].snoprs==1)
          pc=pc+2+st[i].soprl[1];
       else if(st[i].snoprs==2)
          pc=pc+2+st[i].soprl[1]+st[i].soprl[2];
```

```
      }
     printf("-------------END OF PASS 1--------------\n");
     setpar();  /* Get settings for trace and breakpoint */
     z=1; /* Inst.number */
    while(1)
      {
       getopr(1,z);   /*  Generate  Pointers  to  the   first
operandds */
       dop[1].rg=op[2].rg;
       dop[1].mm=op[2].mm;
       dop[1].r16=op[2].r16;
       getopr(2,z); /* Generate Pointer to second operand */
       dop[2].rg=op[2].rg;
       dop[2].mm=op[2].mm;
       dop[2].r16=op[2].r16;
       if((n=bs(st[z].mn,motab,33))>=0)  /*Binary  search  */
(*(motab[n].funsim))();  /* Function call for simulating  */
       else
         {
            printf("unknown mnemonic\n");
         }
       g=z+1;
       tpc=st[g].spc;
       if(trace==1)   /* Trace servicing */
        {
          disrg(); /* display registers */
          while((q=getchar())!='\n')  /* Wait till return  key
is  pressed */
            ;
       }
      if(brflag==1)  /*Breakpoint servicing*/
       {
        for(q=1;q<=brno;q++)
         {
          if(br[q]==st[z+1].spc) /*Breakpoint*/
            {
             disrg(); /* display registers */
             while((q=getchar())!='\n')  /* Wait  till  return
key is pressed */
               ;
             break;
            }
          }
       }
       z++;
       if(z>=50)
         exit();
       }
 }

 /*Addressing mode identification and
      collection of relevant data*/

     char *effadd (n,inst,x)
     char *inst;
     int n;
     int x;
     {
       char *seff1(),*seff2(),*reg();
       char *ps1,*ps2,*t,*k;
       int c,i;
      char s[20];
      int hexf=0;
      t=inst;
  while (*inst==' ') /* Skip Blank spaces */
      inst++;
      c=*inst;
      switch(c)  {
      case  '\n':     /* Implicit addressing  No  operands  are
specified*/
        st[x].snoprs=0;
        st[x].scopr[n]=1;/*  Code for this addressing mode  =1
*/
        st[x].soprl[n]=0; /* Operand field length =0 */
        ft=1;
        return(inst);
      case '#': /*Immediate addressing */
        st[x].snoprs=st[x].snoprs+1;   /*  Increment  No.   of
operands */
        st[x].scopr[n]=4;  /* code =4 */
        inst++;
        while ((c=*inst)==' ') /*Skip Blanks */
           inst++;
        if (c=='$')
         {    hexf=1; /* Hexcadecimal number */
              inst++;
     }
     while  ((c=*inst)==' ')
        inst++;
     if  (hexf==0)
       {
        for (i=0; isdigit(s[i]=*inst)!=0;i++)
          inst++;
        s[i++]=' ';
           s[i]='\O';
             sscanf  (s,"%ld",&opr1); /*Get immediate data */
             st[x].sopr1=opr1;
           }
         else if(hexf==1)
           {
              for (i=0; isxdigit(s[i]=*inst)!=0; i++)
                inst++;
                 s[i++]=' ';
                 s[i]='\O';  /* Get immediate data */
                 sscanf   (s,"%lx",&opr1);   /*Get   immediate
data */
                s[x].sopr1=opr1;
              }
               if((siz==1)|| (siz==2))
                 st[x].soprl[n]=2;   /*  Set   operand   field
lenght*/
               else if(siz==3)
                 st[x].soprl[n]=4;
               break;
             case  'D':
                 st[x].snoprs=st[x].snoprs+1;
                 inst++;
                 while ((c=*inst)==' ')
                    inst++;
                 if (c>='0'  && c<='7')
                   {
                      st[x].sdi[n]=c-'0';
```

```
                       st[x].scopr[n]=2;    /*   Data   register
direct */
                       st[x].soprl[n]=0;
                       inst++;
                       while ((c=*inst)==' ')
                         inst++;
                       if   (c=='-'||  c=='/')/*register   list
specification for MOVEM instruction */
                         {
                                  k=reg(inst,n,x);   /*  Get   the
register list */
                                  vmask(x,n);
                                  return(++k);
                               }
                             break;
                          }
                         else
                           {
                              ft=4;
                              st[x].sft[n]=ft;
                              return(inst);
                           }
                     case   'A':
                           st[x].snoprs=st[x].snoprs+1;
                           inst++;
                           while((c=*inst)==' ')
                             inst++;
                           if  (c>='0'  && c<='7')
                             {
                               st[x].sai[n]=c-'0';
                               st[x].scopr[n]=   3;     /*Address
 register direct */

                               st[x].soprl[n]=0;
                               inst++;
                               while ((c=*inst)==' ')
                                inst++;
                               if (c=='-'  || c=='/') /*Register
 list  specification */
                                {
                               k=reg(inst,n,x);
                               vmask(x,n);
                               return(++k);
                              }
                             break;
                           }
                          else
                             {
                            ft=4;
                            st[x].sft[n]=ft;
                              return(inst);
                           }
          case '$':
            st[x].snoprs=st[x].snoprs+1;
            inst++;
            while (*inst==' ')
               inst++;
            for (i=0; isxdigit(s[i]=*inst)!=0;i++)
              inst++;
            if (i>6)
              {
               ft=10;
              st[x].sft[n]=ft;
              return(inst);
             }
             adlen=i;
             s[i++]=' ';
             s[i]='\0';
             sscanf(s,"%lx",&eaddr[n]);
             while  ((c=*inst)==' ')
                 inst++;
             if    (c==',' || c=='\n')  /* Absulute  addressing
*/
                 {
                    ps1=seff1(inst,n,x);
                   /* pa1= soffl(inst,n,x);*/
                    st[x].seaddr[n]=eaddr[n];
                    st[x].scopr[n]=copr[n];
                    st[x].sft[n]=ft;
                    return(ps1);
                   }
               else if (c=='(') /* Indirect with displacement */
                   {
                        inst++;
                        st[x].sdisp[n]=eaddr[n];
                        inst=seff2(inst,n,x);
                        st[x].sft[n]=ft;
                        if  (ft!=5 &&  ft!=4  &&  ft!=18)
                             break;
                        return(inst);
                    }
                   else
                   {
                        ft=5;
              st[x].sft[n]=ft;
                        return(inst);
                   }
                 case '1':
                 case '2':
                 case '3':
                 case '4':   /* Address is in decimal form */
                 case '5':
                 case '6':
                 case '7':
                 case '8':
                 case '9':
                 case '0':
                        st[x].snoprs=st[x].snoprs+1;
                        for(i=0; isdigit(s[i]=*inst)!=0; i++)
                          inst++;
                        s[i++]=' ';
                        s[i]='O';
                        sscanf(s,"%lu",&eaddr[n]);
                        if(eaddr[n]<=255)
                           adlen=2;
                        else if (eaddr[n]<=65535)
                           adlen=4;
                        else if (eaddr[n]<=16777215)
                           adlen=6;
                        else
                          {
                              ft=20;
                              st[x].sft[n]=ft;
                        return(inst);
                     }
                    while  ((c=*inst)==' ')
                        inst++;
                    if     (c==','||    c=='n')      /*Absulute
addressing */
                      {
                        ps1=seff1(inst,n,x);
                        st[x].seaddr[n]=eaddr[n];
                        st[x].scopr[n]=copr[n];
                        st[x].sft[n]=ft;
                        return(ps1);
                      }
                    else   if   (c=='(')  /*   Indirect   with
displacement */
                      {
                        inst++;
                        st[x].sdisp[n]=eaddr[n];
                        inst=seff2(inst,n,x);
                        st[x].sft[n]=ft;
                        if (ft!=5 && ft!=4 && ft!=18)
                           break;
                        return(inst);
                     }
                   else
                     {
                        ft=5;
                        st[x].sft[n]=ft;
                        return(inst);
                     }
                 case '(':
                      inst++;
                      while ((c=*inst)==' ')
                         inst++;
                      if(c=='A')  /* Address register  indirect
 */
                             {
                                 inst++;
                                 while  ((c=*inst)==' ')
                                     inst++;
                                 if (c>= '0'  && c<= '7')
                                   {
                                      st[x].snoprs=st[x].snoprs+1;
                                      st[x].sadinr[n]=c-'0';
                                      eaddr[n]=a[c-'0'];
                                      inst++;
                                      while  ((c=*inst)== ' ')
                                      inst++;
                                         if(c==')')
                                           {
                                              inst++;
                                              while
 ((c=*inst)==' ')

                                              inst++;
                                                 if(c=='+')   /*

 With post increment */

                                                    {
                                                 st[x].scopr[n]=8;/*
 Address register indirect with post increment */
                                                 inst++;
                                                 while((c=*inst)=='
       ') inst++;
                                                   if (c==',')
                                                   {
                                                    ft=3;
                                                    st[x].sft[n]=ft;
                                                    return(++inst);
                                                    }
                                                   else
        if(c=='\n')
                                                    {
                                                     ft=0;
                                                     st[x].sft[n]=ft;
                                                     return(++inst);
                                                     }
                                                 }
                                                 else if (c==',')
                                                {
                                                   ft=3;
                                                   st[x].sft[n]=ft;
                                                   st[x].scopr[n]=7;
        /* Address register indirect */
                                                   st[x].sopr1[n]=0;
                                                   return(++inst);
                                                }
                                              else if(c=='\n')
                                                {
                                                   ft=0;
                                                   st[x].sft[n]=ft;
                                                   st[x].scopr[n]=7;
                                                   st[x].sopr1[n]=0;
                                                   return(++inst);
                                                   }
                                                  else
                                                   {
                                                    ft=5;
                                                    st[x].sft[n]=ft;
                                                    return(inst);
                                                   }
                                                }
                                             else
                                               {
                                                 ft=5;
                                                 st[x].sft[n]=ft;
                                                 return(inst);
                                                 }
                                               }
                                             else
                                              {
                                                ft=4;       /*  error
        */
                                                st[x].sft[n]=ft;
                                                return(inst);
                                                }
                                             }
                                            else
                                             {
                                              ft=5;
                                              st[x].sft[n]=ft;
                                              return(inst);
                                             }
```

```
                case '-': /* Address register  indirect  with
predecrement */
                inst++;
                while ((c=*inst)== ' ')
                  inst++;
                  if (c=='(')
                    {
                        inst++;
                        while  ((c=*inst)==' ')
                        inst++;
                          if (c==  'A')
                            {
                               inst++;
                               while((c=*inst)==' ')
                               inst++;
                                  if(c>='0' && c<'7')
                                     {
                                        st[x].snoprs=st[x].snoprs+1;
                                        st[x].scopr[n]=9;    /*
Address register indirect with predecrement */
                                        st[x].sopr1[n]=0;
                                        st[x].sadinr[n]=c-'0';
                                        eaddr[n]=a[c-'0'];
                                        inst++;
                                        while  ((c=*inst)==  '
')
                                            inst++;
                                        if  (c==')')
                                            {
                                                inst++;
                                                break;
                                            }
                                          else
                                            {
                                                ft=5;
                                                st[x].sft[n]=5;
                                                return(inst);
                                            }
                                     }
                                      else
                                        {
                                            ft=4;
                                            st[x].sft[n]=ft;
                                            return(inst);
                                        }
                                     }
                                else
                                  {
                                     ft=5;
                                     st[x].sft[n]=ft;
                                     return(inst);
                                  }
                               }

                          else
                             {
                                ft=5;
                                st[x].sft[n]=ft;
                                return(inst);
                             }
                case 'S': /* Operation on status register */
                  inst++;
                  while((c=*inst)==' ')
                    inst++;
                  if(c=='R')
                    {
                       st[x].scopr[n]=15;  /* Status  register
addressing */
                       st[x].snoprs=st[x].snoprs+1;
                       st[x].soprl[n]=0;
                       inst++;
                       break;
                    }
                  else
                    {
                      ft=5;
                      return(++inst);
                    }
             case  'C': /* CCR -condition code addressing */
                  inst++;
                  while((c=*inst)==' ')
                    inst++;
                  if(c=='C')
                    {
                      inst++;
                      while((c=*inst)==' ')
                        inst++;
                      if(c=='R')
                        {
                           st[x].scopr[n]=16;   /*   Condition
code addressing */
                           st[x].snoprs=st[x].snoprs+1;
                           st[x].soprl[n]=0;
                           inst++;
                           break;
                         }
                        else
                         {
                           ft=5;
                           return(++inst);
                         }
                     }
                   else
                     {
                        ft=5;
                        return(inst);
                        }
              default:
                ft=2;
                st[x].sft[n]=2;
                return(inst);
          }
         while  ((c=*inst)==' ')
           inst++;
         if  (c==',')
           {
             ft=3;
             st[x].sft[n]=ft;
             return(++inst);
           }
         else if (c=='\n')
```

```
           { ft=0;
             st[x].sft[n]=ft;
             return(inst);
           }
         else
          {
           ft=5;
           st[x].sft[n]=ft;
           return(inst);
          }
    }
/* supporting function for addressing mode determination:
        called by effadd()*/

char *seff1(inst,n,x)
  char *inst;
  int n;
  int x;
{
  ft=3;
  if(adlen<=4)
    {
      copr[n]=5; /* absulute short addressing */
      st[x].sopr1[n]=2;
      if (eaddr[n]>0x7fff)
        {
           eaddr[n]+=0xfff0000;
           return(++inst);
        }
      else
        {
        return(++inst);
        }
     }
   else
     {
        copr[n]=6; /* Absultute long addressing */
        st[x].sopr1[n]=4;
        return(++inst);
     }
 }
/* Supporting function for addressing mode determination:
        called by effadd() */

char *seff2(inst,n,x)
        char *inst;
        int n;
        int x;

 {
        char *seff3();
        int k,l,m,c;
        char *ps3;
        while ((c=*inst)==' ')
          inst++;
        if  (c=='A')
          {
            inst++;
            while((c=*inst)==' ')
              inst++;
            if (c>'0'  && c<='7')
```

```
              {
                k=c-'0';
                st[x].sadinr[n]=k;     /*   Indirect    address
register number */
                ps3=seff3(inst,k,n,x);
                return(ps3);
              }
            else
              {
                ft=4;
                return(inst);
              }
            }
          else if (c=='P')
            {
               ++inst;
               c=*inst;
               if (c== 'C')
                 {
                    k=8;
                    st[x].sadinr[k]=k; /* PC indirect */
                    ps3=seff3(inst,k,n,x);
                    return(ps3);
                 }
              else
                 {
                    ft=5; /* error */
                    return(inst);
                 }
             }
           else
             {
                ft=5;  /* error */
                return(inst);
             }
        }
/* supporting function for finding addressing mode:
        called by seff2() */
char *seff3(inst,y,n,x)
  char *inst;
  int y,n;
  int x;
{
   int k,l,m,c;
   unsigned long int temp;
   int minus;

    inst++;
    while  ((c=*inst)==' ')
      inst++;
    if (c==')')
      {
        if (adlen<=4)
          {
             if(y<8)
               st[x].scopr[n]=10;      /*   Address    register
indirect with displacement */
             else if(y==8)
               st[x].scopr[n]=12;    /*   PC   indirect    with
displacement */
```

```
             st[x].soprl[n]=2;
               inst++;
               return(inst);
           }
         else
           {
              ft=5;  /* error */
              return(inst);
           }
      }
    else if (c==',')
      {
        inst++;
        while((c=*inst)==' ')
          inst++;
        if(c=='A')
         {
         k=0;
         st[x].sisdi[n]=0;  /* Address register indirect  with
index and displacement index register is a address register */
          }
        else if (c=='D')
          {
             k=1;
             st[x].sisdi[n]=1;/*  index  register  in  a  Data
register */
          }
        else
          {
             ft=5;
             return(inst);
          }
        inst++;
        while((c=*inst)==' ')
          inst++;
        if (c>='0'  && c<='7')
          {
            m=c-'0';
            st[x].sind[n]=m; /* Index register number */
            inst++;
            while ((c=*inst)==' ')
              inst++;
            if  (c=='.')
              {
                inst++;
                while  ((c=*inst)==' ')
                  inst++;
                if   (c=='L')   /*   use  all  bits  in  index
register(long) */
                  {
                   l=1;
                   st[x].sindsi[n]=1;  /* Index register  size
*/
                   }
                 else
                   {
                     ft=5;
                     return(inst);
                   }
                  inst++;
                  while((c=*inst)==' ')
```

```
                    inst++;
                  if (c!=')')
                    {
                      ft=5;  /* error */
                      return(inst);
                    }
                 }
              else if (c==')')
                 {
                  l=0;
                  st[x].sindsi[n]=0;  /* use 16 bits  1  index
register */
                 }
                else
                 {
                   ft=5; /* error */
                   return(inst);
                 }

                 if (adlen<=2)
                   {
                 if(y<8)
                   st[x].scopr[n]=11;   /*  Address   register
indirect withindex and displacement */
                 else if(y==8)
                   st[x].scopr[n]=13;  /*  PC  indirect   with
index and displacement */
                 st[x].soprl[n]=2;
                }
                  else
                     {
                        ft=18;
                        return (inst);
                     }
                  }
                else
                  {
                     ft=4;
                     return (inst);
                  }
              inst++;
              return(inst);
           }
              else
                 {
                    ft=5;
                    return(inst);
                 }
           }
/* Display routine */

prnall()
{
   int i,j;
   printf("noprs=%d         oprl[1]=%d          oprl[2]=%d\n",
noprs,oprl[1],oprl[2]);
   printf("copr[1]=%d
copr[2]=%d\n",copr[1],copr[2]);
   printf("di[1]=%d         di[2]=%d\n",di[1],di[2]);
   printf("ai[1]=%d         ai[2]=%d\n",ai[1],ai[2]);
   printf("opr1=%lu             opr1x=%lx\n",opr1,opr1);
   printf("eaddr[1]=%lx  eaddr[2]=%lx\n",eaddr[1],eaddr[2]);
   for (i=0; i<8; i++)
      {
        printf("A%d = %lx   D%d = %lx\n",i,a[i],i,d[i]);
      }
   printf("sr=%x\n",sr);
   printf("pc=%lx\n",pc);
   printf("flags=%1d%1d%1d%1d%1d\n",fx,fn,fz,fv,fc);
   for (i=0; i<8; i++)
  {
   printf("am[%d] = %d dm[%d] = %d\n",i,am[i],i,cm[i]);
  }
}

/* Extraction of mnemonic and size of operand */

char  *mne(mp,i)
  char *mp;
  int  i;
  {
     int n;
     while(*mp==' ')
       mp++;
     for(n=0;((st[i].mn[n]=*mp)!=' ')  &&  (*mp!='.');n++)  /*
store mnemonic */
        mp++;
     st[i].mn[n]='\0';
     if(*mp==' ')  /* No extention; therefore  word  operands
default */
      st[i].size=2;
     else if(*mp=='.')
      {
        mp++;
        if(*mp=='B') /* Byte */
          st[i].size=1;
        else if(*mp=='W') /* word */
          st[i].size=2;
        else if(*mp=='L') /* long word */
          st[i].size=3;
        else
          printf("error in mnemonic\n");
        mp++;
       }
       siz=st[i].size;
       return(mp);
}

/* Register list determination for MOVEM instruction */

char  *reg(inst,n,a)
  char *inst;
  int  n;
  int  a;
{
  int x,y,z,i,k,l,m,c;

  st[a].soprl[n]=2;
  k=st[a].scopr[n];
  for (i=0; i<=7; i++)
    {
      am[i]=0;
      dm[i]=0;
    }
      for (; ;)
       {
         if (k==2)
          {
           x=st[a].sdi[n];
           if(*inst=='-')
            {
             inst++;
             if(*inst=='D')
              {
               inst++;
               c=*inst;
               if(c>='0'   &&   c<='7')
                {
                 y=c-'0';
                 for (i=x; i<=y; i++)
                   dm[i]=1;
                 inst++;
                 if((*inst!='/')      &&     (*inst!=',')       &&
(*inst!='\n'))
                  {
                   ft=5;
                   return(inst);
                  }
                 else if((*inst==',') || (*inst=='\n'))
                  {
                    ft=3;
                    st[a].scopr[n]=14;
                    return(inst);
                  }
                 else
                  {
                    inst++;
                  }
               }
             else
               {
                ft=4;
                return(inst);
               }
            }
           else
            {
              ft=5;
              return(inst);
            }
          }
         else if((c=*inst)=='/')
           {
            inst++;
            dm[x]=1;
           }
         else if ((*inst==',') || (*inst=='\n'))
          {
           ft=3;
           dm[x]=1;
           st[a].scopr[n]=14;
           return(inst);
          }
```

```
        }
       else if (k==3)
        {
         x=st[a].sai[n];
         if (*inst=='-')
          {
           inst++;
           if (*inst=='A')
            {
             inst++;
             c=*inst;
             if(c>='0' && c<='7')
              {
               y=c-'0';
               for(i=x;i<=y;i++)
                am[i]=1;
               inst++;
               if((*inst!=',')     &&     (*inst!='/')     &&
(*inst!='\n'))
                 {
                  ft=5;
                  return(inst);
                 }
                else if ((*inst==',') || (*inst=='\n'))
                 {
                  ft=3;
                  st[a].scopr[n]=14;
                  return(inst);
                 }
                else
                  {
                   inst++;
                  }
               }
              else
               {
                ft=4;
                return(inst);
                }
             }
            else
             {
              ft=5;
              return(inst);
             }
           }
            else if ((c=*inst)=='/')
             {
              inst++;
              am[x]=1;
              }
            else if((*inst==',') || (*inst=='\n'))
              {
               ft=3;
               am[x]=1;
               st[a].scopr[n]=14;
               return(inst);
              }
           }
          else
           {
             ft=5;
             return(inst);
            }
          st[a].scopr[n]=14;
          if(*inst=='A')
           {
            k=3;
            inst++;
            c=*inst;
            if(c>='0' && c<='7')
             {
               st[a].sai[n]=c-'0';
               inst++;
             }
            else
             {
             ft=4;
             return(inst);
             }
           }
         else if (*inst=='D')
           {
            k=2;
            inst++;
            c=*inst;
            if(c>='0' && c<='7')
             {
              st[a].sdi[n]=c-'0';
              inst++;
             }
            else
             {
              ft=4;
              return(inst);
             }
           }
             else if ((*inst==',') || (*inst=='\n'))
               return(inst);
             else
               {
                ft=5;
                return(inst);
                }
          }
    }

/* Storing of collected data in a file */

    fwri()
    {
      FILE *fo,*fopen();
      int a,b,c,d;

      fo=fopen("out.txt","w");
      for(a=1;a<50;a++)
       {
         fprintf(fo,"%s                                        %d
",st[a].mn,st[a].snoprs);
         fprintf(fo,"codes=%d                                  %d
",st[a].scopr[1],st[a].scopr[2]);
         fprintf(fo,"                              errflags=%d
%d\n",st[a].sft[1],st[a].sft[2]);
         fprintf(fo,"operand1=%ld        ",st[a].soopr1);
         fprintf(fo,"pc=%lx          ",st[a].spc);
         fprintf(fo,"disments=%ld
%ld\n",st[a].sdisp[1],st[a].sdisp[2]);
         fprintf(fo,"effaddress=%ld
%lx\n",st[a].seaddr[1],st[a].seaddr[2]);
         fprintf(fo,"indi                                      add.regs=%d
%d\n",st[a].sadinr[1],st[a].sadinr[2]);
         fprintf(fo,"sdi1=%d                                       sdi2=%d
",st[a].sdi[1],st[a].sdi[2]);
         fprintf(fo,"                                          sai1=%d
sai2=%d\n",st[a].sai[1],st[a].sai[2]);
         fprintf(fo,"dreg                ind=%d                    %d
",st[a].sisdi[1],st[a].sisdi[2]);
         fprintf(fo,"                   index                   reg=%d
%d\n",st[a].sind[1],st[a].sind[2]);
         fprintf(fo,"size      of       index       reg=%d         %d
",st[a].sisdi[1],st[a].sisdi[2]);
         fprintf(fo,"    sizeofopr=%d\n",st[a].size);
         fprintf(fo,"\n\n");
         fprintf(fo,"flags=%1d%1d%1d%1d%1d\n",fx,fn,fz,fv,fc);
         fprintf(fo,"mask=%6x\n",st[a].smask);
        }
     }

/* Binary Search routine */

bs(word,tab,n) /* To search the mnemonic table */
  char *word;
  struct mntab tab[];
  int n;
 {
    int low,high,mid,cond;
    low=0;
    high=n-1;
    while(low<=high)
     {
        mid=(low+high)/2;
        if((cond=strcmp(word,tab[mid].monic))<0)
           high=mid-1;
        else if (cond>0)
           low=mid+1;
        else
           return(mid);
      }
    return(-1);
}

/*Setting pointers to the operands*/

 getopr(n,b)  /*  Set  the  Pointer  to  the  operands.   The
simulating functions make use of this pointers */
   int n,b;
 {
   int iaddr1(),iaddr2();
   int c,y;

   switch((st[b].scopr[n]))  {
       case 0:
         return(0);
        case 1:  /* Implicit addressing */
          return(0);
        case 2:  /* data register direct addressing */
          op[n].rg=&d[(st[b].sdi[n])];  /* op[n] Points to  the
variable d[i], 'i'-data register number */
          fmm[n]=0;
          return(0);
        case 3: /* Addr. register direct addressing */
          op[n].rg=&a[(st[b].sai[n])];
          fmm[n]=0;
          return(0);
        case 4: /* Immediate addressing */
          op[n].rg=&(st[b].sopr1);
          fmm[n]=0;
          return(0);
        case 5: /* Absolute shrot */
          op[n].mm=&mem[(st[b].seaddr[n])];  /* Pointer  Points
to the simulated memory location */
          fmm[n]=1;
          return(0);
        case 6: /* Absolute long */
          op[n].mm=&mem[(st[b].seaddr[n])];  /* Pointer  Points
to the simulated memory location */
          fmm[n]=1;
          return(0);
        case 7: /* Address register indirect */
          op[n].mm=&mem[(a[(st[b].sadinr[n])])];
          fmm[n]=1;
          return(0);
        case  8:  /*  Address  register  indirect  with  Post-
increment */
          op[n].mm=&mem[(a[(st[b].sadinr[n])])];
          fmm[n]=1;
          if(st[b].size==1)
             (a[(st[b].sadinr[n])])+=1; /* Increment  according
to the size of the operand */
           else if(st[b].size==2)
             (a[(st[b].sadinr[n])])+=2; /* Increment  according
to the size of the operand */
           else if (st[b].size==3)
             (a[(st[b].sadinr[n])])+=4; /* Increment  according
to the size of the operand */
           return(0);
        case 9: /* Address register indirect with pre-decrement
*/
          if(st[b].size==1)
            (a[(st[b].sadinr[n])])-=1;
          else if(st[b].size==2)
            (a[(st[b].sadinr[n])])-=2;
          else if(st[b].size==3)
            (a[(st[b].sadinr[n])])-=4;
          op[n].mm=&mem[(a[st[b].sadinr[n]])];
          fmm[n]=1;
          return(0);
        case   10:   /*   Address   register   indirect   witth
displacement */
          y=st[b].sadinr[n];
          iaddr1(y,b,n);  /* This function generater  effective
address and sets the pointers */
          return(0);
        case  11: /* Address register indirect with  index  and
```

```
displacement */
         y=st[b].sadinr[n];
         iaddr2(y,b,n);
         return(0);
       case 12: /* PC indirect with displacement */
         y=8;
         iaddr1(y,b,n);
         return(0);
       case  13: /* Address register indirect with  index  and
displacement */
         y=8;
         iaddr2(y,b,n);
         return(0);
       case 14: /* Register list specification */
         return(0);
       case 15: /* SR addressing */
         op[n].r16=&sr;
         fmm[n]=2;
         return(0);
       case 16: /* CCR addressing */
         op[n].r16=&sr;
         fmm[n]=2;
         return(0);

      }
}
/* Effective address determination:
        called by getopr() */

       iaddr1(y,b,n)
       int y,b,n;

       {
          unsigned long int dsp;

         if((st[b].sdisp[n])    &    0x8000)    /*    Negative
displacement */
          {
           dsp=(st[b].sdisp[n])   ¦   0xffff0000;   /*    Sign
extension of displacement */
           dsp=(~(dsp))+1;  /* 2's complement */
           if(y<8)
             (st[b].seaddr[n])=a[(st[b].sadinr[n])]-(dsp);  /*
Address register indirect with displacement */

           else if (y==8)
             (st[b].seaddr[n])=(st[b].spc)-(dsp)+2;   /*    PC
indirect  with displacement */
           op[n].mm=&mem[(st[b].seaddr[n])]; /* set Pointer */
           fmm[n]=1;
           return(0);
         }
       else  /* Positive displacement */
        {
         if(y<8)
          (st[b].seaddr[n])=a[(st[b].sadinr[n])]+(st[b].sdisp[n]);
         else if (y==8)
           (st[b].seaddr[n])=(st[b].spc)+(st[b].sdisp[n])+2;
         op[n].mm=&mem[(st[b].seaddr[n])];
         fmm[n]=1;
         return(0);
      }
}

/* Effective address determination:
        called by getopr() */

iaddr2(y,b,n) /* for Address register indirect with index  and
displacement */
int y,b,n;

{
   int minus;
   unsigned long int temp,dsp;

   if((st[b].sindsi[n])==0) /* 16 bits */
     {
      if((st[b].sisdi[n])==0)  /*  Index register  is  address
register */
       {
         if((a[(st[b].sind[n])]) & 0x8000) /* index  register
contents negative */
          {
            temp=a[(st[b].sind[n])] | 0xffff0000;
            temp=(~temp)+1;  /* 2's complement */
            minus=1; /* set minus flag*/
          }
        else
          {
            temp=a[(st[b].sind[n])];
            minus=0;
          }
       }
     else  if((st[b].sisdi[n])==1) /* Index register  in  Data
register */
       {
         if((d[(st[b].sind[n])]) & 0x8000) /* register */
          {
            temp=d[(st[b].sind[n])] | 0xffff0000;
            temp=(~temp)+1; /*2's complement */
            minus=1;
          }
        else /* positive */
          {
            temp=d[(st[b].sind[n])];
            minus=0;
          }
       }
   }
 else if((st[b].sindsi[n])==1) /* 32 bits */
   {
     if((st[b].sisdi[n])==0) /* index in address register */
       temp=a[(st[b].sind[n])];
     else  if((st[b].sisdi[n])==1) /* index register  in  data
register */
       temp=d[(st[b].sind[n])];
  }
if((st[b].sdisp[n]) & 0x80) /* displacement is negative */
  {
    dsp=(st[b].sdisp[n]) & 0xffffff00;
    dsp=(~(dsp))+1; /* 2 is complement */
    if(y<8)
```

```
     {
       eaddr[n]=(a[(st[b].sadinr[n])])-dsp; /* Addr.  register
indirect with index and displacement */
     }
   else if(y==8) /* PC indirect with index and displacement */
     {
      eaddr[n]=(st[b].spc)-dsp+2;
     }
  }
else  /* displacement is positive, therefore add  displacement
*/
  {
    if(y<8)
      {
        eaddr[n]=(a[(st[b].sadinr[n])])+(st[b].sdisp[n]);
      }
    else if(y==8)
      {
        eaddr[n]=(st[b].spc)+(st[b].sdisp[n])+2;
      }
  }
if((st[b].sindsi[n])==1) /* Address index register contents */
  {
   eaddr[n]+=temp;
   (st[b].seaddr[n])=eaddr[n];
   op[n].mm=&mem[(st[b].seaddr[n])];
   fmm[n]=1;
   return(0);
 }
else if((st[b].sindsi[n])==0)
 {
   if(minus==1)    /*  Address  or  subtract  index   register
contents */
     eaddr[n]-=temp;
   else if(minus==0)
     eaddr[n]+=temp;
   (st[b].seaddr[n])=eaddr[n];
   op[n].mm=&mem[(st[b].seaddr[n])];
   return(0);
  }
}

abcd()
 {
  printf("This is ABCD\n");
 }

bclr()
 {
  printf("This is BCLR\n");
 }

 btst()
  {
   printf("This is BTST\n");
  }


   unsigned long int te[3],te2[3],te3[3],te4[3];
   unsigned long int ter[3];
   unsigned long int tm1,tm2,tm3,tm4;
```

```
/*Fetching a Byte operand*/

gbyte(n)
int n;
{
              switch(fmm[n])   {
              case 0:    /* register operand */
              te[n]=(*(op[n].rg)) & 0x000000ff;
              ter[n]=te[n];
              break;
             case 1: /* memory */
              te[n]=(*(op[n].mm));
              ter[n]=te[n];
              break;
             case 2: /* SR */
              te[n]=(*(op[n].r16)) & 0x00ff;
              ter[n]=te[n];
              break;
              }
  }
/*Fetching a Word operand*/

gword(n)
int n;

{
     switch(fmm[n]) {

              case 0: /*Register addressing */
                ter[n]=(*(op[n].rg))&0x0000ffff;
                te[n]=(*(op[n].rg))&0x0000ff00;
                te[n]>>=8;
                te[n]&=0x000000ff;
                te2[n]=(*(op[n].rg))&0x000000ff;
                break;
              case 1: /* Memory addressing */
                te[n]=(*(op[n].mm));
                (op[n].mm)++;
                te2[n]=(*(op[n].mm));
                ter[n]=te[n];
                ter[n]<<=8;
                ter[n]&=0x0000ff00;
                tm1=te2[n]&0x000000ff;
                ter[n]|=tm1;   /*  The  two   bytes   are
appended and stored here.*/
                break;
              case 2: /* SR */
                ter[n]=(*(op[n].r16));
                te[n]=(*(op[n].r16))&0xff00;
                te[n]>>=8;
                te[n]&=0x000000ff; /*High byte */
                te2[n]=(*(op[n].r16))&0x00ff;/*        low
byte*/

                break;
                }
   }
/*Fetching a long word operand */

glong(n)
int n;
```

```
{
                  switch(fmm[n])    {
                  case 0: /*Register Addressing */
                    ter[n]=(*(op[n].rg));
                    te[n]=(*(op[n].rg))&0xff000000;
                    te[n]>>=24;
                    te[n]&=0x000000ff;    /*Most   significant
byte*/
                    te2[n]=(*(op[n].rg))&0x00ff0000;
                    te2[n]>>=16;
                    te2[n]&=0x000000ff;
                    te3[n]=(*(op[n].rg))&0x0000ff00;
                    te3[n]>>=8;
                    te3[n]&=0x000000ff;
                    te4[n]=(*(op[n].rg))&0x000000ff;   /*least
significant byte*/
                    break;
                  case 1:
                    te[n]=(*(op[n].mm));
                    (op[n].mm)++;
                    te2[n]=(*(op[n].mm));
                    (op[n].mm)++;
                    te3[n]=(*(op[n].mm));
                    (op[n].mm)++;
                    te4[n]=(*(op[n].mm));
                    tm1=te[n];
                    tm1<<=8;
                    tm1&=0x0000ff00;
                    tm2=te2[n]&0x000000ff;
                    tm1|=tm2;
                    tm1<<=8;
                    tm1&=0x00ffff00;
                    tm3=te3[n]&0x000000ff;
                    tm1|=tm3;
                    tm1<<=8;
                    tm1&=0xffffff00;
                    tm4=te4[n]&0x000000ff;
                    tm1|=tm4;
                    ter[n]=tm1;/* The four bytes are appended
and stored  in te[n];    */
                    break;
                  case 2:
                    printf("IAM\n");
                    break;
                    }
   }
/*Simulation of MOVE*/

move()
{
 int q,q2;
 unsigned long int ftest;
      if((st[z].scopr[2]==16)  && (st[z].size==2)) /* Move  to
CCR */
         {
           movec();
           return(0);
         }
     if((st[z].scopr[1]==15) && (st[z].size==2)) /* Move  from
SR */
        {
          mofrsr();
          return(0);
        }

      if((q=st[z].scopr[1])!=12   &&   q!=13)   /*check    for
addressing mode validity*/
       {
         if(q==3 && st[z].size==1)
          {
             printf("IAM\n");
             return(0);
          }
         if((q2=st[z].scopr[2])!=12  &&  q2!=13  &&  q2!=4  &&
q2!=3) /* check for addressing mode validity.     */
           {
              if(st[z].size==1) /*byte move*/
               {
                  gbyte(1);
                  switch(fmm[2])  {
                     case 0:  /* destination is reg */
                       (*(op[2].rg))&=0xffffff00;
                       (*(op[2].rg))|=ter[1];
                       ftest=ter[1];
                       break;
                     case 1: /* dest in memory */
                       (*(op[2].mm))=te[1];
                       ftest=te[1];
                       break;
                     case 2: /*dest is 16 bit reister */
                       (*(op[2].r16))&=0xff00;
                       (*(op[2].r16))|=ter[1];
                       ftest=ter[1];
                       break;
                       }
                    if(ftest>0x7f)
                        fn=1; /*set negative flag*/
                    else fn=0;
                    if(ftest==0)
                         fz=1; /*set zero flag*/
                    else fz=0;
                    fv=0; /*Reset 'V' and 'C' flags*/
                    fc=0; /*REset 'V' and 'C' flags*/
                   }
                 else if(st[z].size==2) /*word move*/
                   {
                     gword(1);
                     switch(fmm[2])   {

                     case 0:
                       (*(op[2].rg))&=0xffff000C;
                       (*(op[2].rg))|=ter[1];
                       break;
                     case 1:
                       (*(op[2].mm))=te[1];
                       (op[2].mm)++;
                       (*(op[2].mm))=te2[1];
                       break;
                     case 2:
                       (*(op[2].r16))=ter[1];
                       break;
                       }
                     ftest=ter[1];
                     if(ftest>0x7fff)
                       fn=1;
                     else fn=0;
                     if(ftest==0)
                       fz=1;
                     else fz=0;
                     fv=0;
                     fc=0;
                   }
                 else if(st[z].size==3) /*long word move*/
                   {
                       glong(1);
                       switch(fmm[2])    {
                         case 0:
                           (*(op[2].rg))=ter[1];
                           break;
                         case 1:
                           (*(op[2].mm))=te[1];
                           (op[2].mm)++;
                           (*(op[2].mm))=te2[1];
                           (op[2].mm)++;
                           (*(op[2].mm))=te3[1];
                           (op[2].mm)++;
                           (*(op[2].mm))=te4[1];
                           break;
                         case 2:
                           printf("IAM\n");
                           break;
                         }
                         ftest=ter[1];
                         if(ftest>0x7fffffff)
                           fn=1;
                         else fn=0;
                         if(ftest==0)
                           fz=1;
                         else fz=0;
                         fv=0;
                         fc=0;
                      }
                   }
               }
               flags();
          }

/*Simulation of MOVE data,CCR*/

movec()

{
  int q,q2,q3;
  unsigned long int tp;

  if((q=st[z].scopr[1])!=3)
    {
       gword(1);
       (*(op[2].r16))&=0xff00;
       tp=ter[1]&0x000000ff;
       (*(op[2].r16))|=tp;
       inflag();
    }
```

```
}

/*Simulation of MOVE SR, destination*/

mofrsr()

{
  int q,q2,q3;
  unsigned long int tp;

  if((q=st[z].scopr[2])!=3 && q!=12 && q!=13 && q!=4)
    {
       gword(1);
       switch(fmm[2])   {
         case  0:
           (*(op[2].rg))&=0xffff0000;
           (*(op[2].rg))|=ter[1];
           break;
         case 1:
           (*(op[2].mm))=te[1];
           (op[2].mm)++;
           (*(op[2].mm))=te2[1];
           break;
         case 2:
           (*(op[2].r16))=ter[1];
           break;
          }
      }
}

/*simulation of MOVEA*/

movea()

{
  int q,q2,q3;
  unsigned long int tp;

  if((q=st[z].scopr[2])==3 && st[z].size!=1)
    {
      if(st[z].size==2)
        {
          gword(1);
          if(ter[1]>0x7fff)
             ter[1]|=0xffff0000;
          (*(op[2].rg))=ter[1];
        }
      else if(st[z].size==3)
        {
           glong(1);
           (*(op[2].rg))=ter[1];
        }
    }
  else
    {
     printf("IAM\n");
    }
}
/*Simulation for MOVEP*/

movep()
```

```
{
 int q,q2,q3,i,j,k;
 unsigned long int tlp;
 unsigned short int tp[5];
 unsigned int tip;
    q=st[z].scopr[1];
    q2=st[z].scopr[2];
   if((q==10 && q2==2)||(q==2 && q2==10))
    {
      if(st[z].size!=1)
       {
        if(st[z].size==2)
         {
          if(q==10)
           {
          for(i=1;i<3;i++)
           {
            tp[i]=*(op[1].mm);
            op[1].mm+=2;
           }
          (*(op[2].rg))&=0xffff0000;
          tip=tp[1];
          tip<<=8;
          tip&=0xff00;
          tip|=tp[2];
          (*(op[2].rg))|=tip;
          }
        else if(q==2)
          {
            gword(1);
            *(op[2].mm)=te[1];
            op[2].mm+=2;
            *(op[2].mm)=te2[1];
          }
        }
      else if (st[z].size==3)
        {
         if(q==10)
          {
           for(i=1;i<5;i++)
            {
             tp[i]=*(op[1].mm);
             op[1].mm+=2;
            }
          tlp=tp[1];
          tlp<<=8;
          tlp&=0x0000ff00;
          tlp|=tp[2];
          tlp<<=8;
          tlp&=0x00ffff00;
          tlp|=tp[3];
          tlp<<=8;
          tlp&=0xffffff00;
          tlp|=tp[4];
          (*(op[2].rg))=tlp;
         }
       else if(q==2)
         {
          glong(1);
          *(op[2].mm)=te[1];
          op[2].mm+=2;
          *(op[2].mm)=te2[1];
          op[2].mm+=2;
          *(op[2].mm)=te3[1];
          op[2].mm+=2;
          *(op[2].mm)=te4[1];
         }
        }
      }
    else
      {
         printf("ISIZE\n");
      }
   }
 else
   {
   printf("IAM\n");
   }
}

/*Simulation of MOVEQ*/

moveq()
{
 int q,q2,q3,i,j,k;
 unsigned long tp,tp1;

 if((st[z].scopr[1]==4) && (st[z].scopr[2]==2))
  {
   gbyte(1);
   if((*(op[1].rg))>0xff)
    {
     printf("ERR\n");
     return(0);
    }
   *(op[2].rg)=ter[1];
   if(ter[1]>0x7fffffff)
     fn=1;
   else
     fn=0;
   if(ter[1]==0)
     fz=1;
   else
     fz=0;
   fv=0;
   fc=0;
 }
 else
 {
    printf("IAM\n");
    return(0);
 }
flags();
}

/*Status Register modification*/
flags()
  {
  unsigned  int  flse;  /* form SR  from  the  flag  variables
fx,fn,fz,fv,fc */
```

```
  flse=fx;
  flse<<=1;
  flse&=0x0002;
  flse|=fn;
  flse<<=1;
  flse&=0x0006;
  flse|=fz;
  flse<<=1;
  flse&=0x000e;
  flse|=fv;
  flse<<=1;
  flse&=0x001e;
  flse|=fc;
  flse&=0x001f;
  sr&=0xffe0;
  sr|=flse;
 }

inflag()  /* set fx,fn,fz,fv,fz from variable SR */
  {
    fx=sr&0x0010;
    fx>>=4;
    fx&=0x0001;
    fn=sr&0x0008;
    fn>>=3;
    fn&=0x0001;
    fz=sr&0x0004;
    fz>>=2;
    fz&=0x0001;
    fv=sr&0x0002;
    fv>>=1;
    fv&=0x0001;
    fc=sr&0x0001;
  }
/*Mask generation for MOVEM instruction*/

vmask(a,n)
  int a,n;
{
  int i,j,k,m;
  unsigned int mas, mas1, mas2;
  mas1=mas2=0;
    for(i=7;i>=0;i--)
      {
       mas1 |=dm[i];
       mas1<<=1;
       mas1&=0xfffe;
      }
       mas1>>=1;
       mas1&=0xff;
    for(i=7;i>=0;i--)
      {
       mas2|=am[i];
       mas2<<=1;
       mas2&=0xfffe;
      }
       mas2>>=1;
       mas2&=0xff;
   mas2<<=8;
   mas2&=0xff00;
   mas1&=0x00ff;
  mas=mas1|mas2;
  st[a].smask=mas;

  }

  /*Simulation of MOVEM*/

  movem()
   {
    int q,q2;

    q=st[z].scopr[1];
    q2=st[z].scopr[2];
    if(st[z].size==1)
     {
      printf("IAM\n");
      return(0);
     }
   if(((q==14)&&(q2!=14)) || ((q!=14)&&(q2==14)))
     {
      if((q2==14)&&(q!=2)&&(q!=3)&&(q!=9)&&(q!=4))
        {
         mfrm();
        }
      else                   if((q==14)&&(q2!=2)&&(q2!=3)&&(q2!=8)
&&(q2!=4)&&(q2!=12)&&(q2!=13))
             {
           mtom();
        }
      else
        {
          printf("IAM\n");
        }
      }
     else
      printf("IAM\n");
}
/*Simulation of MOVEM(from memory):
        called by movem()*/

mfrm()
{
  int q,q2,q3,i,j,k;
  unsigned int msk;

  msk=st[z].smask;
  if(st[z].size==2)
    {
      if(st[z].scopr[1]==8)
        a[st[z].sadinr[1]]-=2;
      for(i=0,j=0;i<=7;i++)
        {

         if((msk&0x0001)==1)
          {
          gword(1);
          if(ter[1]>0x7fff)
           ter[1]|=0xffff0000;
           d[i]=ter[1];
           j++;
           (op[1].mm)++;
        q3=(j*2);
       }
      msk>>=1;

     }
     if(st[z].scopr[1]==8)
       a[st[z].sadinr[1]]+=q3;
    for(i=0,k=0;i<=7;i++)
     {

      if((msk&0x0001)==1)
        {
          gword(1);
      if(ter[1]>0x7fff)
        ter[1]|=0xffff0000;
         a[i]=ter[1];
         k++;
         op[1].mm++;
        q3=(k*2);
        }
      msk>>=1;

     }
     if(st[z].scopr[1]==8)
       a[st[z].sadinr[1]]+=q3;
 }
else if (st[z].size==3)
  {
   if(st[z].scopr[1]==8)
      a[st[z].sadinr[1]]-=4;
     for(i=0,j=0;i<=7;i++)
      {

       if((msk&0x0001)==1)
        {
         glong(1);
         d[i]=ter[1];
         j++;
          (op[1].mm)++;
       q3=(j*4);
        }
       msk>>=1;

      }
       if(st[z].scopr[1]==8)
         a[st[z].sadinr[1]]+=q3;
     for(i=0,k=0;i<=7;i++)
       {

        if((msk&0x0001)==1)
         {
        glong(1);
          a[i]=ter[1];
          k++;
        (op[1].mm)++;
      q3=(k*4);
         }
        msk>>=1;

       }
      if(st[z].scopr[1]==8)
```

```
          a[st[z].sadinr[1]]+=q3;
     }
}

/* Simulation of MOVEM(to memory
        called from movem()*/

mtom()
{
  unsigned int msk;
  int q,q2,q3,q4,i,j,k;
  unsigned long int c;
    q3=0;
    q4=0;
    q2=st[z].scopr[2];
    msk=st[z].smask;
    if(st[z].size==2)
     {
       if(q2==9)
         a[st[z].sadinr[2]]+=2;
      for(i=0,j=0;i<=7;i++)
       {
        if(q2==9)
         {
          q3=msk&0x8000;
          msk<<=1;
          c=a[7-1]&0x0000ffff;
         }
       else
         {
          q3=msk&0x0001;
          msk>>=1;
          c=d[i]&0x0000ffff;
         }
        if(q3!=0)
         {
          op[1].rg=&c;
          fmm[1]=0;
          gword(1);
          *(op[2].mm)=te[1];
          (op[2].mm)++;
          *(op[2].mm)=te2[1];
          (op[2].mm)++;
          j++;
          if(q2==9)
           {
            (op[2].mm)-=4;
           }
           q4=(j*2);
          }
     }
     if(q2==9)
      {
       a[st[z].sadinr[2]]-=q4;
      }
     for(i=0,k=0;i<=7;i++)
      {
       if(q2==9)
        {
         q3=msk&0x8000;
         msk<<=1;
      c=d[7-i]&0x0000ffff;
     }
    else
     {
      q3=msk&0x0001;
      msk>>=1;
      c=a[i]&0x0000ffff;
     }
  if(q3!=0)
    {
     op[1].rg=&c;
     fmm[1]=0;
     gword(1);
     *(op[2].mm)=te[1];
     (op[2].mm)++;
     *(op[2].mm)=te2[1];
     (op[2].mm)++;
     k++;
     if(q2==9)
      {
       (op[2].mm)-=4;
      }
      q4=(k*2);
    }
 }
   if(q2==9)
     {
      a[st[z].sadinr[2]]-=q4;
     }
  }
else if(st[z].size==3)
  {
    if(q2==9)
      a[st[z].sadinr[2]]+=4;
   for(i=0,j=0;i<=7;i++)
     {
      if(q2==9)
       {
        q3=msk&0x8000;
        msk<<=1;
        c=a[7-i];
       }
      else
       {
        q3=msk&0x0001;
        msk>>=1;
        c=d[i];
       }
     if(q3!=0)
      {
        op[1].rg=&c;
        fmm[1]=0;
        glong(1);
        fillme();
        j++;
        if(q2==9)
         {
          (op[2].mm)-=8;
         }
       q4=(j*4);
      }
     }
    if(q2==9)
     {
      a[st[z].sadinr[2]]-=q4;
     }
    for(i=0,k=0;i<=7;i++)
     {
      if(q2==9)
       {
        q3=msk&0x8000;
        msk<<=1;
        c=d[7-i];
       }
      else
       {
        q3=msk&0x0001;
        msk>>=1;
        c=a[i];
       }
      if(q3!=0)
       {
        op[1].rg=&c;
        fmm[1]=0;
        glong(1);
        fillme();
        k++;
        if(q2==9)
         {
          (op[2].mm)-=8;
         }
        q4=(k*4);
      }
    }
    if(q2==9)
     {
      a[st[z].sadinr[2]]-=q4;
     }
    }
}


/*Storing data in memory*/

fillme()

{

   *(op[2].mm)=te[1];
   (op[2].mm)++;
   *(op[2].mm)=te2[1];
   (op[2].mm)++;
   *(op[2].mm)=te3[1];
   (op[2].mm)++;
   *(op[2].mm)=te4[1];
   (op[2].mm)++;
}

/*Display of Registers*/

disrg()
{
 printf("PC=%8lx         SR=%4x                \n",tpc,sr);
 printf("D0=%8lx                 D1=%8lx                      D2=%8lx
D3=%8lx\n",d[0],d[1],d[2],d[3]);
 printf("D4=%8lx                 D5=%8lx                      D6=%8lx
D7=%8lx\n",d[4],d[5],d[6],d[7]);
 printf("\n");
 printf("A0=%8lx                 A1=%8lx                      A2=%8lx
A3=%8lx\n",a[0],a[1],a[2],a[3]);
 printf("A4=%8lx                 A5=%8lx                      A6=%8lx
A7=%8lx\n",a[4],a[5],a[6],a[7]);
 printf("\n");
 printf("FLAGS=%1d %1d %1d %1d %1d\n",fx,fn,fz,fv,fc);
 printf("\n\n");
}

/*Run time parameter setting:
        Trace mode
        Breakpoints*/

setpar()
{
 int c,e,g;
 printf("Please set relevant parameters\n");
 printf("\n");
 printf("Do you want TRACE mode?\n");
 printf("   Press Y for yes pr N for no\n");
 c=getc(stdin);
 getc(stdin);
 if((c=='y')||(c=='Y'))
    trace=1;
 else if((c=='n')||(c=='N'))
    trace=0;
 printf("Do you want to set breakpoints\n");
 printf("     Press Y for yes or N for no\n");
 g=getc(stdin);
 getc(stdin);
 if((g=='y')||(g=='y'))
   brflag=1;
 else if((g=='n')||(g=='N'))
   brflag=0;
 if(brflag==1)
  {
    printf("Input breakpoints one at a time in hexa\n");
    for(e=1;e<10;e++)
     {
      brno=e;
      scanf("%lx",&br[e]);
      printf("%8lx\n",br[e]);
      if(br[e]==0)
      {
        getc(stdin);
        return(0);
      }
    }
 }
}



     sadd(suf)  /* Function to simulate subtract or add */
     int suf;
```

```
{
  unsigned long int iso1,iso2;
  unsigned long int s1,s2,s3,s4,tpry1,tpry2;
  unsigned int k1,k2;

   if(st[z].size==1)
     {
        iso1=0x000000ff;
        iso2=0x0000007f;
        gbyte(1);
        gbyte(2);
        if(suf==1)
         {
          tpry1=(~ter[1])+1;
          tpry1&=iso1;
         }
         else if(suf==0)
          {
            tpry1=ter[1];
          }
      }
    else if (st[z].size==2)
      {
        iso1=0x0000ffff;
        iso2=0x00007fff;
        gword(1);
        gword(2);
        if(suf==1)
         {
          tpry1=(~ter[1])+1;
          tpry1&=iso1;
         }
        else if(suf==0)
         {
          tpry1=ter[1];
         }
      }
    else if(st[z].size==3)
      {
        iso1=0xffffffff;
        iso2=0x7fffffff;
        glong(1);
        glong(2);
        if(suf==1)
         {
          tpry1=(~ter[1])+1;
          tpry1&=iso1;
         }
        else if(suf==0)
         {
          tpry1=ter[1];
         }
      }
    s1=tpry1;
    s2=ter[2];
    l2=s1+s2;
    if((iso1-tpry1)<ter[2])
     {
      vfc=1;
      k2=1;
      vfx=1;
    }
   else
     {
       vfc=0;
       k2=0;
       vfx=0;
     }

   if(exflag==1)
    {
     if(suf==0)
     {
       if((l2==iso1)&&(fx==1))
       {
        vfc=1;
        k2=1;
        vfx=1;
       }
      l2=l2+fx;
      }
     else if(suf==1)
       {
        if((l2!=0)&&(fx==1))
         {
           vfc=1;
           k2=1;
           vfx=1;
           l2=l2+iso1;
         }
      }
   }
  if(suf==1)
   {
    if(vfc==1)
      vfc=0;
    else if(vfc==0)
       vfc=1;
    vfx=vfc;
    }
l2&=iso1;
s3=s1&iso2;
s4=s2&iso2;
if((iso2-s3)<s4)
   k1=1;
else k1=0;
if(exflag==1)
 {
 l3=s3+s4;
 if(suf==0)
 {
    if((l3==iso2)&&(fx==1))
     {
       k1=1;
      }
     else k1=0;
   }
   else if(suf==1)
   {
    if((l3!=0)&&(fx==1))
     {
       k1=1;
```

```
          }
         else
         {
          k1=0;
         }
      }
   }
   vfv=k1^k2;
   if(l2==0)
    vfz=1;
   else vfz=0;
   if(l2>iso2)
    vfn=1;
   else vfn=0;
   if(st[z].size==1)
    {
       sbyte(l2,2);
       return(0);
    }
   else if(st[z].size==2)
     {
        sword(l2,2);
        return(0);
     }
   else if(st[z].size==3)
     {
        slong(l2,2);
        return(0);
     }
}


sbyte(s,n)
unsigned long int s;
int n;
{
   unsigned int *ar16[3];
   unsigned long int *arg[3];
   unsigned short int *amm[3];

      arg[1]=dop[1].rg;
      arg[2]=dop[2].rg;
      amm[1]=dop[1].mm;
      amm[2]=dop[2].mm;
      ar16[1]=dop[1].r16;
      ar16[2]=dop[2].r16;
      split(s);
   switch(fmm[n])    {
      case 0:
         (*(arg[n]))&=0xffffff00;
         (*(arg[n]))|=14;
         break;
      case 1:
        (*(amm[n]))=14;
        break;
      case 2:
        (*(ar16[n]))&=0xff00;
        (*(ar16[n]))|=14;
        break;
      }
  }
  sword(s,n)
  unsigned long int s;
  int n;
  {
     unsigned long int *arg[3];
     unsigned int *ar16[3];
     unsigned short int *amm[3];

        arg[1]=dop[1].rg;
        arg[2]=dop[2].rg;
        amm[1]=dop[1].mm;
        amm[2]=dop[2].mm;
        ar16[1]=dop[1].r16;
        ar16[2]=dop[2].r16;
        split(s);
     switch(fmm[n])   {
       case 0:
         (*(arg[n]))&=0xffff0000;
         (*(arg[n]))|=s;
         break;
        case 1:
          (*(amm[n]))=13;
          (amm[n])++;
          (*(amm[n]))=14;
          break;
        case  2:
          (*(ar16[n]))&=0xffff0000;
          (*(ar16[n]))|=s;
          break;
         }
}

slong(s,n)
unsigned long int s;
int n;
{
     unsigned short int *amm[3];
     unsigned int *ar16[3];
     unsigned long int *arg[3];

          arg[1]=dop[1].rg;
          arg[2]=dop[2].rg;
          amm[1]=dop[1].mm;
          amm[2]=dop[2].mm;
          ar16[1]=dop[1].r16;
          ar16[2]=dop[2].r16;
          split(s);
          switch(fmm[n])   {
            case 0:
              (*(arg[n]))=s;
              break;
            case 1:
              (*(amm[n]))==11;
              (amm[n])++;
              (*(amm[n]))=12;
              (amm[n])++;
              (*(amm[n]))=13;
              (amm[n])++;
              (*(amm[n]))=14;
              break;
```

```
             case 2:
               printf("IAM\n");
               break;
              }
        }


add()
 {

             int q,q2;
               q=st[z].scopr[1];
               q2=st[z].scopr[2];
               exflag=0;
               if((q==2)&&(q2!=3)&&(q2!=12)&&(q2!=13)&&(q2!=4))
                 sadd(0);
               else if(q2==2)
                 {
                   if((st[z].size==1)&&(q==3))
                      {
                        printf("IAM\n");
                        return(0);
                      }
                     sadd(0);
                  }
                else
                  {
                    printf("IAM\n");
                    return(0);
                  }
          fx=vfx;
          fn=vfn;
          fz=vfz;
          fc=vfc;
          flags();
          }

        addi()
        {
           int q,q2;

            q=st[z].scopr[1];
            q2=st[z].scopr[2];
            exflag=0;
            if((q==4)&&(q2!=3)&&(q2!=12)&&(q2!=13)&&(q2!=4))
              {
                 sadd(0);
              }
             else
              {
                 printf("IAM\n");
                 return(0);
              }
            copfla();
            flags();
          }

       addq()
       {
          int q,q2;
       q=st[z].scopr[1];
       q2=st[z].scopr[2];
       exflag=0;
       if((q==4)&&(q2!=12)&&(q2!=13)&&(q2!=4))
         {
           if((q2==3)&&(st[z].size==1))
            {
              printf("IAM\n");
              return(0);
            }
           if((*(op[1].rg))==0)
             (*(op[1].rg))=8;
           if((*(op[1].rg))>8)
             {
              printf("ERR\n");
              return(0);
             }
            sadd(0);
          }
         else
          {
            printf("IAM\n");
            return(0);
          }
        copfla();
        flags();
      }

 adda()
 {
  int q,q2;
  unsigned long int k1;

    q=st[z].scopr[1];
    q2=st[z].scopr[2];
    exflag=0;
    if((q2==3)&&(st[z].size!=1))
     {
       if(st[z].size==2)
         {
          gword(1);
          if(ter[1]>0x7fff)
             ter[1]|=0xffff0000;
         }
       else if (st[z].size==3)
         {
           glong(1);
         }
       glong(2);
       k1=ter[1]+ter[2];
       (*(op[2].rg))=k1;
      }
    else
      {
        printf("IAM\n");
        return(0);
      }
 }

addx()
{
    int q,q2,tzfl;

     q=st[z].scopr[1];
     q2=st[z].scopr[2];
     if(((q==2)&&(q2==2))||((q==9)&&(q2==9)))
      {
        tzfl=fz;
        exflag=1;
        sadd(0);
        fx=vfx;
        fn=vfn;
        fv=vfv;
        fc=vfc;
        if(vfz==1)
          fz=tzfl;
        else if(vfz==0)
          fz=0;
        flags();
      }
     else
      {
       printf("IAM\n");
       return(0);
      }
  }


  split(s)
  unsigned long int s;
  {
   l1=s&0xff000000;
   l1>>24;
   l1&=0x000000ff;
   l2=s&0x00ff0000;
   l2>>=16;
   l2&=0x000000ff;
   l3=s&0x0000ff00;
   l3>>=8;
   l3&=0x000000ff;
   l4=s&0x000000ff;
 }

 copfla()
 {
  fx=vfx;
  fn=vfn;
  fz=vfz;
  fv=vfv;
  fc=vfc;
 }

sub()
{
 int q,q2;
 q=st[2].scopr[1];
      q2=st[z].scopr[2];
     exflag=0;
     if((q==2)&&(q2!=3)&&(q2!=12)&&(q2!=13)&&(q2!=4))
        sadd(1);
     else if(q2==2)
       {
```

```
                if((st[z].size==1)&&(q==3))
                  {
                    printf("IAM\n");
                    return(0);
                  }
                 sadd(1);
              }
            else
              {
                printf("IAM\n");
     z].scopr[1];
            q2=st[z].scopr[2];
            if(((q==2)&&(q2==2))||((q==9)&&(q2==9)))
             {
         z].scopr[1]                 q2=st[z].scopr[2]                   if(((q==2)&&(
q=ep9r&&]q2==9)))      q2=st[z].scopr[2]                   if(((q==2)&&(q2==2))||((q==9)&
               {
                tzfl=fz;
                exflag=1;
                sadd(1);
                fx=vfx;
                fn=vfn;
                fv=vfn;
                fc=vfc;
                if(vfz==1)
                  fz=tzfl;
                else if(vfz==0)
                  fz=0;
                flags();
               }
             else
               {
                printf("IAM\n");
                return(0);
              }
          }

         subi()
         {
           int q,q2;

           q=st[z].scopr[1];
           q2=st[z].scopr[2];
           exflag=0;
           if((q==4)&&(q2!=3)&&(q2!=12)&&(q2!=13)&&(q2!=4))
             {
               sadd(1);
             }
           else
             {
              printf("IAM\n");
              return(0);
             }
            copfla();
            flags();
          }

          subq()
          {
            int q,q2;

            q=st[z].scopr[1];
```

```
 q2=st[z].scopr[2];
 exflag=0;
 if((q==4)&&(q2!=12)&&(q2!=13)&&(q2!=4))
   {
    if((q2==3)&&(st[z].size==1))
     {
      printf("IAM\n");
      return(0);
     }
    if((*(op[1].rg))==0)
      (*(op[1].rg))=8;
    if((*(op[1].rg))>8)
      {
       printf("ERR\n");
       return(0);
      }
     sadd(1);
     }
   else
     {
       printf("IAM\n");
       return(0);
     }
    copfla();
    flags();
}


suba()
{
 int q,q2;
 unsigned long int h1,h2;

     q=st[z].scopr[1];
     q2=st[z].scopr[2];
     exflag=0;
     if((q2==3)&&(st[z].size!=1))
       {
        if(st[z].size==2)
         {
          gword(1);
          if(ter[1]>0x7fff)
             ter[1]|=0xffff0000;
         }
       else if(st[z].size==3)
         {
          glong(1);
         }
       glong(2);
       h1=(~ter[1])+1;
       h2=h1+ter[2];
       (*(op[2].rg))=h2;
       }
     else
       {
        printf("IAM\n");
        return(0);
       }
  }
```

```
   bra()
   {
    int q,q2;
    if(st[z].scopr[1]==12)
     {
       for(q=1;q<50;q++)
        {
         if((st[q].spc)==(st[z].seaddr[1]))
           {

            z=q-1;
            return(0);
           }
         }
       printf("Invalid Branch\n");
    }
    else
     {
       printf("IAM\n");
     }
}

cmnt(u)
int u;

{
  int v;
  if(u==1)
    v=0;
   else if(u==0)
    v=1;
   return(v);
}

bhi()
{
  if((cmnt(fc))&(cmnt(fz)))
     bra();
  else
     return(0);
}

bls()
{
  if((fc)|(fz))
    bra();
  else return(0);
}

bcc()
{
 if(cmnt(fc))
  bra();
 else return(0);
}

bcs()
{
 if(fc)
   bra();
 else return(0);
```

```
}

bne()
{
 if(cmnt(fz))
   bra();
 else return(0);
}

beq()
{
 if(fz)
  bra();
 else return(0);
}

bvc()
{
 if(cmnt(fv))
   bra();
 else return(0);
}

bvs()
{
 if(fv)
   bra();
 else return(0);
}

bpl()
{
 if(cmnt(fn))
   bra();
 else return(0);
}

bmi()
{
 if(fn)
   bra();
 else return(0);
}

bge()
{
 if((fn&fv)|((cmnt(fn))&(cmnt(fv))))
   bra();
 else return(0);
}

blt()
{
 if((fn&(cmnt(fv)))|((cmnt(fn))&fv))
   bra();
 else return(0);
}

bgt()
{
 if((fn&fv&(cmnt(fz)))|((cmnt(fn))&(cmnt(fv))&(cmnt(fz))))
```

```
    bra();
 else return(0);
}

ble()
{
 if(fz¦(fn&(cmnt(fv)))¦((cmnt(fn))&fv))
   bra();
 else return(0);
}
```

MODEL SESSION

```
MOVE.L #$1A,D1
ADD.L D1,D0
MOVE.L D0,$200
MOVE.L $200,D5
ADD.L #$10,D5
BRA $FFE2(PC)
```

```
MOVE.L #$00000011,D0
MOVE.L #$00000022,D1
MOVEM.L D0-D1,$200
MOVEM.L $200,D2-D3
```

```
MOVE.L #$00000011,D0
MOVE.L #$00000022,D1
MOVEA.L #$200,A0
MOVEM.L D0-D1,(A0)
MOVEM.L $200,D2-D3
```

```
MOVE.W #$1003,D0
MOVE.W #$23AB,D1
ADD.W D0,D1
MOVE.B #$52,D4
ADD.B D1,D4
```

```
MOVE.L #254,D0
MOVE.L #154,D1
SUB.L D1,D0
ADD.B #155,D0
MOVEM.L D0/D1,$130
MOVEM.L $130,D2-D3
```

```
MOVE.L #$11111111,D0
MOVE.L #$22222222,D1
MOVE.L #$33333333,D2
MOVE.L #$44444444,D3
MOVE.L #$55555555,D4
MOVE.L #$66666666,D5
MOVE.L #$77777777,D6
MOVE.L #$88888888,D7
MOVEM.L D0-D7,$200
MOVEM.L $200,A0-A7
MOVEM.W A0-A7,$100
MOVE.L #0,D0
MOVE.L #0,D1
MOVE.L #0,D2
MOVE.L #0,D3
MOVE.L #0,D4
MOVE.L #0,D5
MOVE.L #0,D6
MOVE.L #0,D7
MOVEM.W $100,D0-D7
```

```
MOVE.L #$F23485A2,D0
MOVE.L #$1A,D1
ADD.L D1,D0
MOVE.L D0,$200
MOVE.L $200,D5
ADD.L #$10,D5
BRA $FFE2(PC)
```

```
MOVE.L #1000,D0
MOVE.L #1,D1
MOVEA.L #$0,A0
MOVE.L D1,(A0)+
ADDI.L #1,D1
SUBI.L #1,D0
BNE $FFF0(PC)
MOVE.L #1000,D0
MOVE.L #0,D5
MOVEA.L #$0,A0
ADD.L (A0)+,D5
SUBI.L #1,D0
BNE $FFF6(PC)
```