5-2-1986

Software simulation of MC68000

Ramakrishnan Iyer

Rochester Institute of Technology

School of Computer Science and Technology

Software Simulation of MC 68000

by

Ramakrishnan A. Iyer

A thesis, submitted to

The Faculty of the School of Computer Science and Technology

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Approved by: John L. Ellis

Dr. John Ellis

Margaret M. Reek

Prof. Margaret Reek

George A. Brown

Prof. George Brown

May 2, 1986

Title of Thesis : Software Simulation of MC68000

I, Ramakrishnan Iyer, hereby grant permission to the Wallace Memorial Library, of R.I.T., to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date : May 2, 1986

A. Ramakrishnan

ABSTRACT

The introduction of the Motorola MC68000 family of microprocessors ushered in a new era of microprocessors. These are single-chip microprocessors designed to function as the central processing units of sophisticated computer systems.

The prime objective of this thesis work is to develop a simulator for the MC68000 microprocessor mainly for educational purposes. The simulator would help in any test or research work utilizing 68000 assembly programs in the future. Most of the instructions in the 68000 family are implemented. Both the user mode and supervisory mode programs can be written and run against the simulator. Besides supporting most of the MC68000 features the simulator also has additional features to help debugging.

Table of Contents

ACKNOWLEDGEMENTS

I take this opportunity to thank Dr. John Ellis for his guidance and encouragement not only during the course of this thesis work but also during my entire career at this institution.

I also thank Professor Margaret Reek for her guidance and suggestions.

My special thanks are also due to Professor George Brown for his guidance during the implementation of this project.

I am extremely grateful to Dr. P. Peddy of Computer Engineering Department and Professor Thomas Young of Electrical Engineering Technology department for their help in getting useful information on the project at the initial stage and during the testing stage.

I am also thankful to Dr. Peter Anderson for his kind help in providing assistance when I had difficulty in securing extra lab time to do my thesis work.

CHAPTER 1

INTRODUCTION

Simulation of a microprocessor is not only an interesting proposition but also a useful project in any environment. Buying the hardware involves a lot of overhead (For eg: Maintenance). For general work, a simulator should be a satisfactory alternative. Even if the hardware is available the utility of simulators does not decrease. The users can work independent of the hardware if a simulator exists. Most of the basic programming problems done by students do not really need a close association with the hardware. The existence of a simulator has another advantage, a hardware update/maintenance need not jeopardize work currently in progress. These are the considerations which prompted this thesis work.

The prime objective of this thesis is to develop a simulator for the MC68000 microprocessor mainly for educational purposes. The RIT professional lab has available a 68000 assembler which could be utilized to the advantage of the users if there was a 68000 machine readily available. A 68000 simulator was a good choice for this purpose. This would not only help users in writing 68000 programs but also would help in any test or research work utilizing 68000 assembly programs in the future.

From the historic point of view, the concept of a 16-bit microprocessor at Motorola Semiconductors, Inc., was first generated in 1976. The idea was to design a 16-bit microprocessor that would be easy to program and support with the current development in microprocessors. Previous 6800/6801/6809 instructions were incorporated in the new machine. The MC68000 is a modern microprocessor by any standards. Many of the instructions, when combined with the different addressing modes, more closely resemble high-level language statements than assembly language instructions of traditional 4-bit or 8-bit microprocessor. The new MC68000 executes 40% faster than such 16-bit microprocessors as the z8000 from the Zilog and the 8086 from Intel as per the references in Appendix B.

Chapter 2 gives a brief description of the MC68000 microprocessor. It gives a general idea of the MC68000 architecture, which would help in understanding the working of the various instructions in this machine. This is followed by chapter 3, where the different instructions are explained in detail. It also includes a number of illustrations to aid in a better understanding of these instructions. The basic design of the simulator is described in chapter 4. The material includes explanation of the method followed in decoding the instruction set of the MC68000 microprocessor and explains the suitability of this simulator to the VAX 11/780 system. Chapter 5 deals with the implementation aspects of this simulator. This

section introduces the reader to the different capabilities of the MC68000 machine in general and of the simulator in particular. Additional features provided to help the debugging of programs is also explained. Chapter 6 goes over the testing procedure followed for the simulator and the conclusions derived from the tests. It also highlights the restrictions which the user should be aware of. A user manual is given in the appendix. This will help the user in understanding the procedure to be followed in using the simulator.

Development of this simulator was a challenging experience. Besides the time spent in coding, a lot of time was devoted to testing the simulator in a number of different ways. It is hoped that the simulator will serve its purpose as an educational tool. This thesis has gone into sufficient depth to provide a good understanding of the simulator. The code is also written in a simple and lucid manner and changes, if any, should be easy to make in the wake of new developments. It is my firm belief that with the help of the MC68000 manufacturer's user manual (and this thesis) the reader will have little difficulty in using the simulator.

CHAPTER 2

THE MC68000 MICROPROCESSOR- A BRIEF INTRODUCTION

The MC68000, designed by Motorola, is the first 16 bit microprocessor to have a 32 bit internal architecture. It has about 68000 transistors on a single chip.

In the early years of computers, the size of the registers, the internal instruction register paths and the external data paths were all identical. This is not true now. Large external paths require the chip package to have a larger number of pins which implies high packaging and production costs. Thus chips tend to have larger internal paths than external paths as in the case of MC68000. The external 16 bit bus is multiplexed from the 32 bits inside the engine. The sixteen 32 bit registers are partitioned into 8 address registers and 8 data registers. Address register " A7 " is used as supervisory stack pointer in the supervisory mode and as user stack pointer in the user mode. Hence the Motorola Inc., treats them as " A7 " and " A7' ". In other words, there are two " A7 " registers with only one of them in operation at a time. This gives the effect of 17 registers in total.

The MC68000 has a very regular instruction set, making available several general addressing modes for most instructions. This permits easy implementation of stacks and queues without special instructions. Two 32 bit stack pointers are provided for coding in system calls. A special

flag register can be set to move the machine into a debugging, single stepping mode for program development. Eight of the general purpose registers are used as data registers for byte (8 bits),(16 bits) and long word (32 bits) operations. The other nine general purpose registers are address registers which can function as stack pointers and base address registers. All 17 general purpose registers can serve as index registers. Although the program counter is 32 bits long, only the low order 24 bits are used in the basic 68000 design. These 24 bits provide the 68000 with an addressing range of 16 Mbytes.

The 68000 can operate on five different sizes of data : bits, 4 bit BCD, 8 bit bytes, 16 bit words and 32 bit long words. Byte data may be addressed on even or odd address boundaries; whereas, word and long-word data must be addressed on even address boundaries. The status register is 16 bits long and consists of a user byte in the lower order and a system byte in the upper part. The system byte consists of trace and status bits and the interrupt mask. The user byte consists of condition codes. The user byte can be addressed as a condition code register. The condition code register consists of overflow, carry, auxiliary carry, sign and zero flag bits. The instruction set contains 56 basic instruction types and the different variations of these basic instructions give a total of 81 instructions. There are 14 different addressing modes available for operand access.

To support multi-user and multi-tasking applications, the 68000 operates in two different states, namely," user " for normal operation and "supervisor" for system control. All instructions can be executed in the supervisor state. However, a few instructions like " RESET " and " STOP " are unavailable in the user state. Thus the system prevents one user or task from violating the system's integrity.

The lowest 8 bytes of memory hold the reset vector and reside in ROM. Additional locations in the lower 1024 bytes are allocated to interrupt vectors, error vectors and vectors for various other types of exceptions. These locations can reside either in ROM or RAM. The remainder of the 16 Mbyte memory map of the 68000 can be used any way the user wants. Because I/O is memory mapped in the 68000, there are no specific instructions for I/O.

The interrupt structure of the 68000 provides seven levels of vectored interrupts, with a mask in the status registers to lock out interrupts at or below the current priority level. When the 68000 receives an enabled interrupt request, it issues an acknowledge signal to the device requesting service. Thereafter, the interrupting device must put a vector number on the data bus. This vector selects one of a possible 192 interrupt service routines in memory. In addition, the 68000 provides seven unique auto vectors whereby devices that cannot generate a vector number can interrupt the 68000. They cause the microprocessor to autovector to interrupt service

subroutines for priority level of that device.

CHAPTER 3

DESCRIPTION OF INSTRUCTION SET

Before describing the instruction set it would be helpful to describe data organization in memory and the various addressing modes.

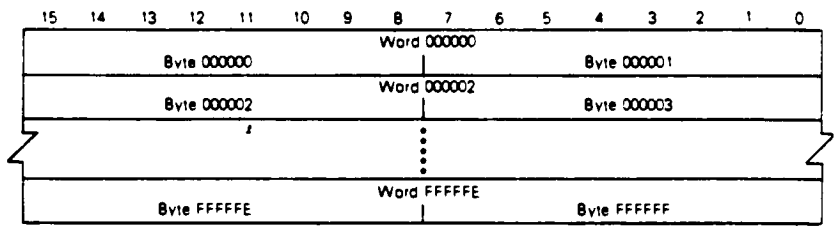
The MC68000 instruction can occupy from one to five words in the memory. The first word is an opcode word which contains the bit pattern that the 68000 decodes to determine the instruction types, the operand addressing modes, and the length of the instruction. Additional extension words are required for operand addressing modes that use constants, absolute addresses or displacement offsets.

a.Data organization in memory

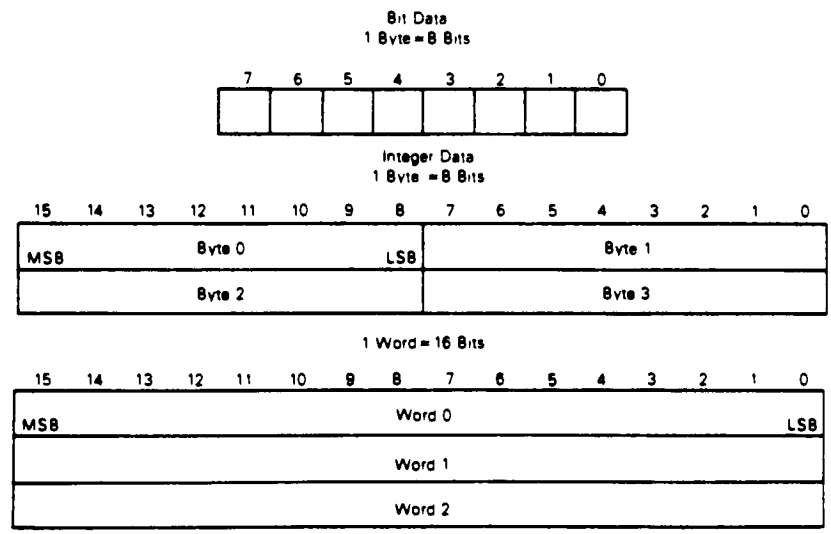
Bytes are individually addressable with the high order byte having an even address. Fig.1 on page 12 makes this organization clear. The low order byte has an odd address that is one count higher than the word address. Instructions and multibyte data are accessed only on word boundaries.

b.Addressing modes

Register Direct Modes : These effective addressing modes specify that the operand is in one of the 16 multifunction registers.



Word Organization in Memory



Data Organization in Memory

Fig.1

(1) Data Register Direct. The operand is in the data register specified by the effective address register field.

(2) Address Register Direct. The operand is in the address register specified by the effective address register field.

Memory Address Modes: These effective addressing modes specify that the operand is in memory and provide the specific address of the operand.

(1) Address Register Indirect. The address of the operand is in the address register specified by the register field.

(2) Address Register Indirect With Postincrement. The address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four depending upon whether the size of the operand is byte, word, or long. If the address register is the stack pointer and the operand size is byte, the address is incremented by two rather than one to keep the stack pointer on a word boundary.

(3) Address Register Indirect With Predecrement. The address of the operand is in the address register specified by the register field. The operand address is decremented by one, two, or four depending upon the size . However, if the address register is the stack pointer and the operand size is byte, the address is decremented by two rather than

one to keep the stack pointer on a word boundary.

(4) Address Register Indirect With Displacement. In this case the address of the operand is the sum of the address in the address register and the sign-extended 16-bit displacement integer in the extension word.

(5) Address Register Indirect With Index. This address mode requires one word of extension. See Fig.2 on page 15.

Bit 15 -- Index Register Indicator

0 -- data register

1 -- address register

Bit 14 through 12 -- Index Register Number

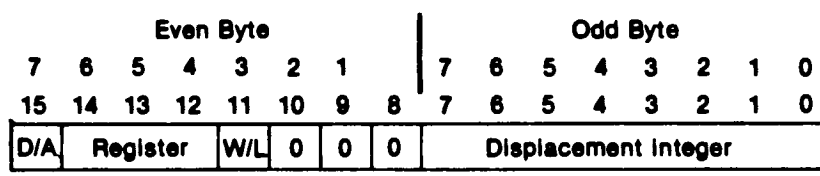
Bit 11 -- Index Size

0 -- sign-extended, low order integer in index register

1 -- long value in index register

The address of the operand is the sum of the address in the address register, the sign-extended displacement integer in the low order eight bits of the extension word, and the contents of the index register.

Special addressing modes. In this case we use the effective address register field to specify the special addressing mode instead of a register number.



Bit 15 — Index Register Indicator
  0 — data register
  1 — address register
Bits 14 through 12 — Index Register Number
Bit 11 — Index Size
  0 — sign-extended, low order word integer in index register
  1 — long value in index register

Fig.2

(1) Absolute Short address. In this case the address of the operand is in the extension word. The 16-bit address is sign-extended before it is used.

(2) Absolute Long Address. The address is formed by the concatenation of two extension words.

(3) Program Counter With Displacement. The address of the operand is the sum of the address in the program counter and the sign-extended 16-bit displacement integer in the extension word.

(4) Program Counter With Index. This is same as address register indirect with displacement mode except that program counter is used as the address register.

(5) Immediate Data. This address mode requires either one or two words of extension depending on the size of operation.

In addition, there are implied addressing modes.

For eg: In all " jsr " (jump to subroutine) the stack is implied. Similarly the program counter is also implied . There is no special instruction to indicate that the program counter is pushed onto the stack in these cases.

Description of instruction sets:

(1) ABCD (Add decimal with extend). This instruction has two forms.

a. ABCD Dy, Dx : This form uses data direct addressing for both the operands. In other words both operands must reside in internal data registers of the 68000.

b. ABCD -(Ay), -(Ax) : In this case predecrement address register indirect addressing is used to specify both the operands. This mode permits access of data stored in memory.

ABCD instruction adds the contents of the source and destination operands together with extend (X) bit of status register.

(2) ADD (Add Binary). This instruction has the following form.

ADD <ea>, Dn

ADD Dn, <ea> Note: ea denotes effective address.

The first form is generally used to process data; whereas, the second form is used to modify the address. Both the instructions add the contents of data register to the contents of the location specified by the effective address EA.

(3) ADDA (Add Address). This is similar to ADD instruction except that there is only one form having the address register as the destination.

ADD <ea>, An. This adds the source operand to the destination address register and stores the result in the address register.

(4) ADDI (Add immediate). This instruction has the following form.

ADDI <data>, <ea>

Immediate data is added to the destination operand and the result is stored in the destination operand.

(5) ADDQ (Add quick).

ADDQ <data>, <ea>

This instruction is a special variation of the ADDI instruction. The immediate data is encoded directly into the instruction word. Therefore opcode takes fewer bytes and executes faster. However the data range is only 1-8.

(6) ADDX (Add Extended). This instruction has two forms.

a. ADDX Dy, Dx : In this case contents of the data register Dy are added to that of Dx and extend bit X. The result is placed in Dx.

b. ADDX -(Ay), -(Ax) : This is a variation of the above form with the main difference being that it is a memory to memory add and hence operands are accessed using predecrement address register indirect addressing.

(7) AND (And Logical). This instruction has two forms.

a. AND <ea>, Dn

b. AND Dn, <ea>

And the source operand to the destination operand and store the result in the destination operand.

Note that immediate addressing mode is not allowed by most of the assemblers since it will be treated as ANDI instruction which is explained next.

(8) ANDI (And immediate)

ANDI <data>, <ea>

This instruction ands the immediate data to the destination operand and stores the result in the destination operand.

(9) ANDI to CCR (And Immediate to Condition Codes).

(Source) CCR ----> CCR

This instruction ands the immediate operand with the condition codes and stores the result in the low-order byte of the status register.

(10) ANDI to SR (And Immediate to the Status Register).

This is a privileged instruction. It has the following form.

Andi xxx, SR

This instruction ands the immediate operand with the contents of the status register and stores the result in the status register. Note that all bits of the status register are affected.

(11) ASL (Arithmetic Shift Left)

ASR (Arithmetic Shift Right) This instruction has three forms.

a. ASd Dx, Dy

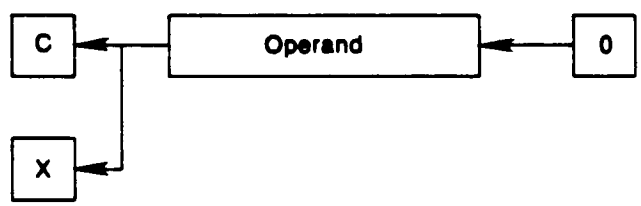
b. Asd <data>, Dy

c. Asd <ea>

This instruction arithmetically shifts the bits in the direction specified. The carry bit receives the last bit shifted out of the operand. The shift count in case (a) and (b) can be specified in two ways.

For ASL, the operand is shifted left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry and the extend bits; zeroes are shifted into the low order bit. The overflow bit indicates if any sign changes occur during the shift.

ASL:



For ASR, the operand is shifted right; the number of positions shifted is the shift count. Bits shifted out of the low order bit go to both the carry and the extend bits; the sign bit is replicated into the high order bit.

ASR:

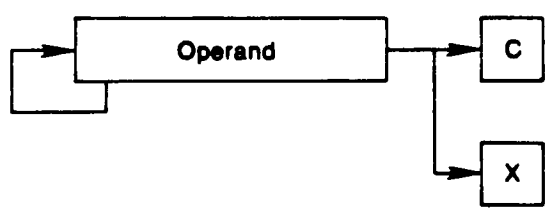


Fig.3

1. Immediate : The shift count is specified in the instruction. The range is 1-8.

2. Register: The shift count is contained in a data register specified in the instruction.

In case (b) content of memory may be shifted one bit only and the operand size is restricted to a word.

Fig.3 on page 21 further clarifies the operation of these instructions.

(12) BCC (Branch Conditionally):

This instruction has the following form.

Bcc < label >

Here "CC" is used to specify one of the many conditional relationships. This instruction passes control to the specified label only if the conditional relationship is true. Otherwise the next sequential statement is executed.

(13) BCHG (Test a Bit and Change):

This instruction has two modes as shown below

(1) BCHG Dn, <ea> ... Immediate mode

(2) BCHG <data>, <ea> ... Register mode

A bit in the destination operand is tested, and the state of the specified bit is reflected in the "z" condition bit. Thereafter the state of that bit is changed. Bit operation using the number modulo 32 is performed in case (2) and modulo 8 in case of (3).

(14) BCLR (Test a Bit and Clear):

This is similar to the above case except that the bit is always cleared in the end.

(15) BRA (Branch always):

In this case next instruction is fetched from the address formed by the sum of the current program counter and the displacement. Note that the value of program location used for this computation is the current location counter plus two. Displacement is a twos complement integer which counts the relative distance in bytes. The displacement can be 8-bit (short branch) or 16-bit (long).

(16) BSET (Test a Bit and Set)

This is similar two BCLR and BCHG instruction except that in the end the tested bit is always set.

(17) BSR (Branch to Subroutine):

This is similar to BRA instruction. However before branching the long word address of the instruction immediately following the BSR instruction is pushed onto the system stack.

(18) BTST (Test a bit):

This is similar to BSET except that the tested bit remains unchanged. Only the condition codes are set as per the tests performed on the specified bit.

(19) CHK (Check Register Against Bounds):

This is a special instruction found only in this machine. Its function is explained by the following algorithm.

if Dn < 0 or Dn > (< ea >) then TRAP

The assembler syntax is as shown below.

CHK <ea>, Dn

The content of the low order word in Dn is compared with the upper bound in <ea>. If it is less than zero or greater than the upper bound, the processor initiates an exception processing. The vector number "6" is generated to reference the CHK instruction exception vector which results in processing the exception routine located at address location hex 018.

(20) CLR (Clear an operand):

In this case the destination is cleared to all zero bits.

(21) CMP (Compare):

The source operand is subtracted from the destination operand and the result obtained is used to decide the condition codes. Note that the destination operand is not changed.

(22) CMPA (Compare address) This is similar to CMP except that the destination operand is an address register which was not the case in the CMP instruction. The size of the source operand may be specified to be word or long. Word length source operands are sign extended to 32 bit quantities before the operation is done.

(23) CMPI (Compare Immediate):

This is again similar to CMP instruction except that the source operand is always an immediate data.

(24) CMPM (Compare Memory):

This instruction has the following form.

CMPM (Ay)+, (Ax)+.

The source operand is subtracted from the destination operand and condition codes set as per the result. Note that the operands are always addressed with the postincrement mode.

RIT assembler treats the first operand as the source operand. This is different from most of the standard MC68000 assemblers. Hence when writing code for this assembler it has to be specified as shown below:

CMPM (Ax)+, (Ay)+

(25) DBcc (Test Condition, Decrement, and Branch): This instruction is extremely useful in handling loops. The algorithm of this instruction is given below:

If (condition false)

then Dn - 1 ---> Dn;

if Dn <> -1

then PC + d --> PC (d -- displacement)

else PC + 2 (Fall through to next instruction)

"cc" represents the conditions to be tested. Dn contains the count of how many times the loop is to be repeated. If the test is true, no branch is taken and the loop is terminated. If the condition is false the content of the specified data register is decremented by 1. If the register count is -1 the branch does not take place . Otherwise the program control branches back to perform the loop again.

(26) DIVS (Signed Divide):

The destination operand is divided by the source operand and the result is stored in the destination. The destination operand is a long operand, and the source is a word operand. The operation is performed using signed arithmetic. The 32 bit result is arranged such that the lower 16 bits contain the result while the upper 16 bits contain the remainder.

(27) DIVU (Unsigned arithmetic):

This is similar to DIVS except that the operation is performed using unsigned arithmetic.

(28) EOR (Exclusive OR Logical)

The source operand is exclusive ORed with the destination operand and the result stored in the destination operand.

(29) EORI (Exclusive OR immediate):

This is same as EOR except that the source operand is an immediate data.

(30) EORI to CCR (Exclusive OR Immediate to Condition Codes):

The immediate source operand is exclusive ORed with the destination operand which is a condition code register.

(31) EORI to SR (Exclusive OR Immediate to the Status Register): This is a privileged instruction. The immediate operand is exclusive ORed with the contents of the status register, and the result stored in the status register.

(32) EXG (Exchange Registers): This instruction exchanges the contents of two registers. All combinations of data register and address registers are allowed.

(33) EXT (Sign Extend): This instruction is used to sign extend a byte to a word or a word to a long. Bit 7 of the specified register is copied to bits 7-15 in case of word operation, or bit 15 of the specified register is copied to bits 16-32 in case of long operation.

(34) JMP (Jump): Execution continues at the effective address specified by the instruction.

(35) JSR (Jump to Subroutine):

This is similar to JMP except that before jumping the long address of the instruction immediately following the JSR instruction is pushed onto the stack.

(36) LEA (Load Effective Address):

The specified address register is loaded with the effective address.

(37) LINK (Link and Allocate):

This is a special instruction of its kind found only in this machine. Before the main program calls a subroutine it is necessary for the calling program to pass the values of some variables to the subroutine. These variables are often pushed onto the stack before calling the subroutine. Then during the execution of the subroutine, they can be accessed from the stack. In addition a data area has to be allocated for local storage of parameters. This is all achieved using this command. Its format is as shown below.

LINK An, <displacement> Its algorithm is :

An --> -(SP); SP --> An; SP + d --> SP

The current content of the specified address register is pushed onto the stack. The new stack pointer is then loaded into stack. This acts as the frame (local storage area) reference. Finally the displacement is added to the stack pointer. This reserves the local parameter storage area.

(38) UNLK (Unlink) :

Its format is An --> SP; (SP)+ --> An

The prior data frame is restored using this instruction. This is achieved by loading the stack pointer with the specified address register followed by loading the address register with the long value from the top of stack.

Thus original value of "An" is restored.

(39) LSR, LSL (Logical Shift): Fig.4 on page 31 illustrates the operation of these two instructions. The "c" bit is the carry bit while the "x" bit is the extend bit.

(40) Move (Move Data from source to destination):

This is a simple but a powerful instruction which moves the content of the source to the destination location. The destination location cannot be an address register.

(41) MOVE from CCR (Move from the Condition Code Register):

The contents of the status register are moved to the destination location. This is a word operation, but only the low order byte contains the condition codes.

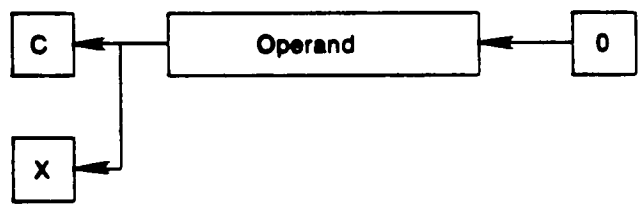
(42) MOVE to CCR (Move to condition codes):

The contents of the source operand are moved to the condition codes. As in the above case, only the low order byte is used to update the condition codes.

(43) Move to SR (Move to Status Register): This is a privileged instruction. The contents of the source operand are moved to status register.

For LSL, the operand is shifted left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry and the extend bits; zeroes are shifted into the low order bit.

LSL:



For LSR, the operand is shifted right; the number of positions shifted is the shift count. Bits shifted out of the low order bit go to both the carry and the extend bits; zeroes are shifted into the high order bit.

LSR:

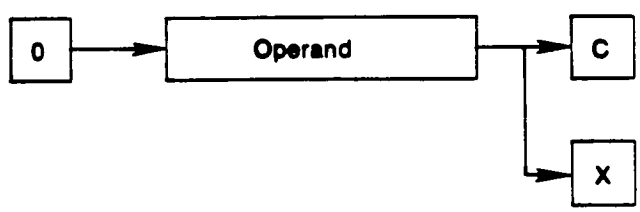


Fig.4

(44) MOVE from SR (Move from the Status Register): The contents of the status register are moved to the destination location.

(45) MOVE USP (Move User Stack Pointer): This is a privileged instruction. The contents of the user stack pointer are transferred to or from the specified address register.

(46) MOVEA (Move Address):

Move the contents of the source to the destination address register. Word size source operands are sign extended to 32 bit quantities before the operation is done.

PIT assembler treats a MOVE instruction as a MOVEA instruction if the destination operand is an address register.

(47) MOVEM (Move Multiple Registers)

The two forms of this instruction are

(a) MOVEM < register list >, < ea >

(b) MOVEM < ea >, < register list >

This instruction saves the contents of the registers specified in register list in memory or vice-versa. The register list can include any combination of data registers and address registers. The second word is called the register list mask.

Register List Mask field — Specifies which registers are to be transferred. The low order bit corresponds to the first register to be transferred; the high bit corresponds to the last register to be transferred. Thus, both for control modes and for the postincrement mode addresses, the mask correspondence is

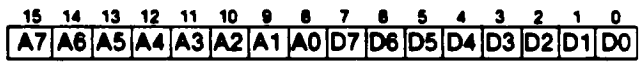


Fig.5a

while for the predecrement mode addresses, the mask correspondence is

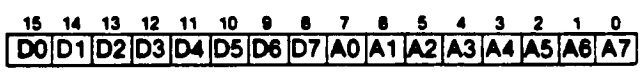


Fig.5b

The selection of these registers depends on this mask. Each bit of this mask corresponds to one of the 68000's internal registers. Setting a bit to 1 indicates that the corresponding register is included in the list and a setting of "0" indicates that it is not included. The normal meaning of the bit mask is as shown in Fig.5-a on page 33. However when address register indirect with predecrement addressing is used, the meaning of the bit mask is changed as shown in Fig.5-b on page 33.

(48) MOVEP (Move peripheral data): In this case data is transferred between a data register and alternate bytes of memory starting at the location specified and incrementing by two. The high order byte of data register is transferred first and the low order byte is transferred last. If the address is even, all the transfers are made on the high order half of the data bus; if the address is odd, all the transfers are made on the low order half of the data bus.

This instruction is actually designed for communicating with LSI peripherals that interface over an 8-bit data bus.

(49) MOVEQ (Move Quick):

This is similar to the MOVE instruction with an immediate data as the source operand. In this case data is contained in an 8-bit operand field within the operation word. However it is sign extended to a long operand before storing at the destination register.

(50) MULS (Signed multiply):

Multiply two signed 16-bit operands yielding a 32-bit signed result. Signed arithmetic is used.

(51) MULU (Unsigned multiply):

This is similar to " MULS " except that unsigned arithmetic is used.

(52) NBCD (Negate Decimal with Extend) :

This is best explained as follows:

0 - (Destination)10 - X --> Destination

The operation is performed using decimal arithmetic.

(53) NEG (Negate):

The operand addressed as destination is subtracted from zero.

(54) NEGX (Negate with extend):

The operand addressed as the destination and the extend bit are subtracted from zero.

(55) NOP (No operation):

In this case no operation is performed. Only the program counter is incremented.

(56) NOT (Logical Complement):

The ones complement of the destination is stored in the destination location.

(57) OR (Inclusive OR Logical):

The source operand is inclusive ORed with the destination operand, and the result is stored in the destination location.

(58) ORI (Inclusive OR Immediate):

Inclusive OR the immediate data to the destination operand and store the result in the destination location.

(59) ORI to CCR (Inclusive Or Immediate to condition codes):

Inclusive OR the immediate operand with the condition codes and store the result in the low-order byte of the status register.

(60) ORI to SR (Inclusive OR Immediate to the Status Register):

This is a privileged instruction. The immediate operand is exclusive ORed with the contents of the status register and the result stored the status register.

(61) PEA (Push effective address):

The effective address is computed and pushed onto the stack.

(63) RESET (Reset External Devices):

This is a method whereby external devices can be reset using software instruction. The execution continues with next instruction. As this thesis is a software simulation this instruction has limited use.

(64) ROL, ROR (Rotate without extend):

This operation is best described by Fig.6 on page 38.

The shift count may be specified as an immediate operand or in register mode.

(65) ROXL (Rotate with Extend):

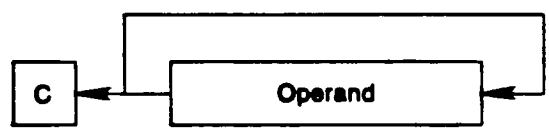
This is similar to the last mentioned instruction except that the extend bit is also included in the rotation. Fig.7 on page 39 explains the operation.

(66) RTE (Return from Exception):

This is a privileged instruction. This should be the last instruction of all exception and interrupt routines. The status register and the program counter are pulled from the system stack. The previous status register and program counter are lost.

For ROL, the operand is rotated left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry bit and back into the low order bit. The extend bit is not modified or used.

ROL:



For ROR, the operand is rotated right; the number of position shifted is the shift count. Bits shifted out of the low order bit go to both the carry bit and back into the high order bit. The extend bit is not modified or used.

ROR:

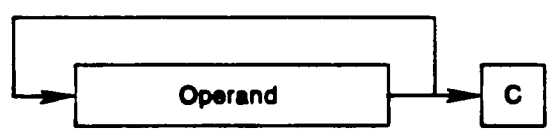
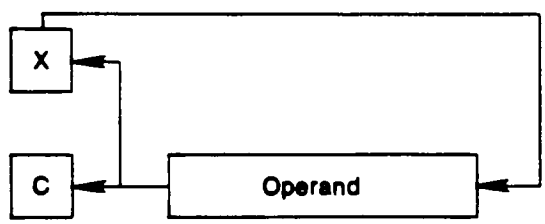


Fig.6

For ROXL, the operand is rotated left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry and extend bits; the previous value of the extend bit is shifted into the low order bit.

ROXL:



For ROXR, the operand is rotated right; the number of positions shifted is the shift count. Bits shifted out of the low order bit go to both the carry and extend bits; the previous value of the extend bit is shifted into the high order bit.

ROXR:

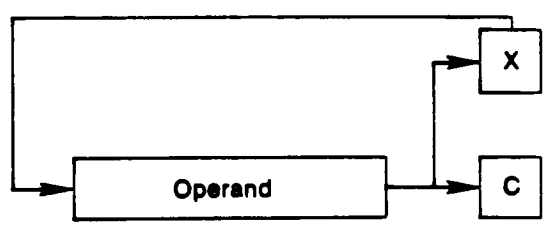


Fig.7

(67) RTR (Return and Restore Condition Codes):

This is best described by the following:

(SP)+ --> CC; (SP)+ ----> PC

The condition codes and program counter are pulled from the stack. The previous values are lost.

(68) RTS (Return from Subroutine)

The program counter is pulled from stack.

(69) SBCD (Subtract Decimal with Extend):

This is described by the following:

(Destination)10 - (source)10 - X --> Destination

The subtraction is performed using BCD arithmetic. The operand may be addressed in two different ways.

(1) SBCD Dy, Dx

(2) SBCD -(Ay), -(Ax)

In the second case the user should note that the RIT assembler treats Ax as the source operan. This is different from the standard MC68000 assembler. He has to take this factor into consideration while writing code in assembly language.

(70) Scc (Set According to condition):

In this case "cc" denotes the condition to be tested. If the condition is true, the destination is set to all ones; if the condition is false, it is set to all zeroes.

(71) STOP (Load Status Register and stop):

This is a privileged instruction. The immediate operand is moved into the entire status register. The program counter points to the next instruction, but the process stops processing and waits for an interrupt to resume operation.

(72) SUB (Subtract Binary):

The source operand is subtracted from the destination operand and the result stored in the destination operand.

(73) SUBA (Subtract address) :

This is similar to SUB except that the destination operand is always an address register; whereas, in the previous case this was not so. The BIT assembler automatically treats a SUB instruction as SUBA if the destination is an address register and does not flag it as an error.

(74) SUBI (Subtract Immediate):

This is similar to SUB except that the source operand is always an immediate value.

(75) SUBQ (Subtract quick):

This is similar to SUBI except that the source value is an immediate value in the range 1-8, and this value is encoded as part of the instruction opcode.

(76) SUBX (Subtract with extend):

This is described by the following:

(Destination) - (Source) - X --> destination.

(77) SWAP (Swap Register Halves):

Exchange the upper and lower words in a data register.

(78 TAS (Test and set an operand):

The operand is compared with zero and the condition codes set or reset as per the result. Independent of this result the most significant bit of the accessed byte is set to 1. This instruction is basically used to support multiprocessing and multitasking system environments.

(79) TRAP (Trap to exception routine):

This instruction initiates exception processing. A vector number is generated based on the low order four bits of the instruction. RIT assembler has a bug and it can recognize only the low order 3 bits of the instruction for

vector computation.

(80) TRAPV (Trap on overflow):

If the overflow is on, the processor initiates an exception routine.

(81) TST (Test an operand):

The operand is compared with zero and the condition codes set. No results are saved.

CHAPTER 4

DESIGN

The M68000 16/32 bit microprocessor reference manual (Fourth edition) formed the basis of this thesis work. From the study of the opcodes of different instructions in the MC68000 it was found that the higher order four bits of the word-sized instruction opcodes help define a definite classification of instruction sets into various groups. For example : If the value of the higher order four bits happens to be the binary code for "6" then it can be any one of the following branch instructions. (1) Bcc (Conditional branch) (2) BSR (Branch to subroutine) (3) BRA (Branch always)

If the value is a "7" then it is a "MOVEQ" instruction or if it is a "14" it is one of the rotate instructions. However, this uniformity is not consistent. For example : A "MOVE" might produce an opcode having the value of the higher order four bits as "1","2" or "3" depending on the different combinations of the effective addresses and modes. This in general formed the basis of the initial decoding sequence.

Next, the instruction sets at the second level have to be inspected to arrange another subgroup at the next level. Let us take the case of an instruction whose higher order four bits gave a value of "2". In this case we know that the instruction under consideration is either a "MOVEA", or

a plain "MOVE". This requires inspection of the remaining 12 bits in the 16-bit opcode to make distinctions among these instructions.

In the case of an instruction whose higher order four bits gave a value of "4" it was observed that a number of instructions were under this category and decoding required careful inspection of the different combinations of the remaining 12 bits in the 16-bit opcode.

This, in general, explains how the bits in the opcode were utilized in the decoding of different instructions. The pattern followed remains the same as explained in the preceding paragraphs. In some cases, though, this process was very involved and required several levels. In other words, in such cases, the decoding algorithm very much resembled a tree structure.

Let us look at a portion of the decode module on page 46. Note the case statements which branch out to different routines after decoding. Consider " case 8 ". When the higher order four bits give an "8" the program control branches to function " eightsep ". In this function the instructions in this category are separated. Hence the name " eightsep". Here the instructions are identified individually. The zeroth bit of this instruction byte is stored in variable " val0 ". The instruction " fetch (iropc) " fetches the next byte whose first two bits are inspected ("iropc stands for "instruction opcode").

```
decode()    /* instruction decode module */
{
int val;
int val1;
int val2;
int det;
int det2;
opc = getbits(iropc,7,4);  /* value of nigher order four bits */
val = getbits(iropc,7,2);
switch(opc)
{
    case 6:
            .........
             break;
     case 2 :
             ....
             ....
              break;


       ....

       ....

      /* if value of higher order four bits ( bits 15-12 ) of */
      /* the instruction opcode is a "14" then jump to routine */
      /* "eightsep"                                            */
      case 14 : rotate();
                break;

      /* if value of higher order four bits ( bits 15-12 ) of */
      /* the instruction opcode is an "8" then jump to routine */
      /* "eightsep"                                            */
      case 8 :
               eightsep();
                break;

      /* if value of higher order four bits ( bits 15-12 ) of */
      /* the instruction opcode is an "9" then jump to routine */
      /* "SUB"                                                 */
      case 9 :  SUB();
                break;

      /* if value of higher order four bits ( bits 15-12 ) of */
      /* the instruction opcode is a "0" then jump to routine */
      /* "ZOP"                                                 */
      case 0 :  ZOP();
                break;

          ....
```

```
            ....

            ....

            ....
        default:  printf("ILLEGAL  INSTRUCTION FOUND \n");
                  time = time + 34; /* update instruction execution */
                                    /* time for an "illegal instruction"*/
                                    /* trap execution                   */

                  interntrap(4);
                  break;
     }
   }
}


/* This is a routine which separates out instructions whose upper */
/* order four bits ( i.e bits 15-12 ) gives a value of "8"        */
eightsep()
{
  int val0,val1; /* variable val0 stores bit 0 of the 8-bit byte */
                 /* under inspection ( i.e bits 8  of the 16-bit */
                 /* instruction opcode ).    */
  val0 = getbits(iropc,0,1);
  oldiropc = iropc;  /* current instruction opcode byte is saved */
  fetch(&iropc);     /* fetch next instruction opcode byte   */
  val1 = getbits(iropc,7,2); /* val1 stores values of bits 7 & 6 */
                             /* of the instruction opcode        */
  if (( val0 == 0 ) && ( val1 == 3))  divu();
   else  if (( val0 == 1 ) && ( val1 == 0 )) sbcd();
   else if (( val0 == 1 ) && ( val1 == 3 ))
             divs();
          else  Or();
};
```

Note the instruction " val1 = getbits(iropc,7,2) ". Note how the contents of the variable " val0 " and " val1 " help in decoding the instructions fully. It is observed that a combination of a zero value in " val0 " and a " 3 " in " val1 " gives a " divu " (unsigned divide instruction). The decoding of " sbcd ", " divs " and " or " follow a similar pattern and is self explanatory from the code shown on pages 46-47.

The MC68000 registers are 32 bits long. The fact that VAX 11/780 at RIT is a 32 bit machine made it an ideal ground to simulate a 32 bit microprocessor. The VAX 11/780 automatically sign-extends the high-order bit; whereas, the MC68000 has the capability of addressing a byte, word, or a long word operand separately without automatic sign-extension in case of byte and word operations. This was an important consideration during the design of the code . For example , if operations were to be performed on a byte, the higher order bits were saved and at the completion of the operation were appended to the new byte value obtained after the operation.

A separate routine was designed for each instruction. This made it easier to debug and understand the code better. Moreover, each routine is a function by itself and changes, if any, can be easily made depending on addition or any enhancements to the existing capabilities of the various instruction sets. A sample program module for an instruction "EXT" (extension) is shown on page 49.

```
/* Extend sign bit of the data register  */

ext()
{
  int opmode,sign,r1;
  extern int  R[8];
  extern int iropc;    /* instruction opcode */
  extern int c,z,v,n; /* condition codes */
  extern int time;       /* instruction execution time */
  time = time + 4;
  opmode = getbits(iropc,7,2);
  r1 = getbits(iropc,2,3); /* r1 holds the register number */

  /* sign extend low order byte of data register to word */
  if ( opmode == 2 )
    {
     sign = getbits(R[r1],7,1);
     if ( sign == 1 ) /* highest order bit of the lower order byte */
                      /* is a "1"                                  */
       {
         n = 1;
         R[r1] = R[r1] | 0x0000FFFF;
       }
     else             /* highest order bit of the lower order byte */
                      /* is a zero                                 */
       {
         n = 0;
         R[r1] = R[r1] & 0xFFFF00FF;
       };
    };
  /* sign extend low order word of data register to long */
  if ( opmode == 3 )
    {
      sign = getbits(R[r1],15,1);
      if ( sign == 1 )  /* highest order bit of the lower order byte */
                      /* is a "1"                                  */
      {
        n = 1;
        R[r1] = R[r1] | 0xFFFF0000;
      }
     else  /* highest order bit of the lower order byte */
           /* is a "0"                                  */
      {
        n = 0;
        R[r1] = R[r1] & 0x0000FFFF;
      };
     };
   if ( R[r1] == 0 ) z = 1;
       else z = 0;
   v = 0;
   c = 0;
};
```

The instruction decode module is also defined as a different function for the same reason.

CHAPTER 5

IMPLEMENTATION

The software tools used for achieving this thesis work were the following :

: C compiler on the RIT VAX 11/780 system (UNIX - Version 4.2).

: The 68000 assembler and loader (User's manual explains the usage of this assembler).

The entire simulator program has been organized into different modules mainly based on the functions and types of instructions.

The main module has three main functions.

(1) Load :- This reads the object file and loads the instruction bytes into its memory.

(2) Fetch :- This function fetches one instruction byte at a time.

(3) Decode :- The byte fetched is decoded into different instructions. (Further fetches are initiated if an extension word is involved).

Before the simulator starts fetching instruction bytes, a routine "initreset" sets up the initial stack areas, stack pointers and program counter. The addresses of the different trap and interrupt routines are loaded at the .

appropriate addresses interactively. The user manual in the appendix explains this procedure with an example.

There are separate routines for different "error" routines (eg : CHK bounds, divide by zero, address error etc.) which are called by the program at different stages depending on the necessity of calling " exception " routines. These "error" routines load the appropriate exception routine addresses into the program counter.

There is a routine for calculating the effective address of the source. This calculates the effective address based on the size of operation and the effective addressing mode. Similarly, there is a routine for calculating the effective address of the destination.

There is a routine for calculating the values from memory. Given the address of the memory location and the size of operation (byte, word, or long) this routine calculates the value of the operand from memory. On the other hand, there is another routine for storing the value resulting from an operation in memory at the specified address.

The condition codes resulting from different operations are implemented in the routines for different instructions. The updated value at the end of each operation is stored in a variable "SR" (status register) so that next reference to the status register, if any, reads the updated status.

The simulator defines memory space from $0-$FFFF. This is sufficient for normal student programs. Of these the first 512 words (1024 bytes) are reserved for interrupt and trap vectors. The memory starts at address 0. This provides 255 different vector numbers, 0 and 2 through $FF. Of the 255 interrupt vectors, 192 are reserved for user interrupts. Fig.8 on page 54 shows vector assignments for exceptions. For interrupts, the vector number is provided by the interrupting external device. The actual MC68000 machine will accept seven levels of interrupts. The MC68000 has a prioritized interrupt structure. Interrupt priority levels are numbered from one to seven, with level seven being the highest priority. The status register contains a three bit mask which indicates the current priority . If the priority of the current interrupt is equal to or less than the value of this mask the 68000 ignores this interrupt request. (However there is an exception. A level 7 will still acknowledge another level 7 interrupt request). If the interrupt request has a value that is higher than the interrupt mask, the 68000 will initiate exception processing.

Interrupt processing follows a general pattern shown below :

1. The 68000 saves the 16-bit contents of the status register in an internal register.

Vector Number(s)	Address Dec	Address Hex	Space	Assignment
0	0	000	SP	Reset Initial SSP[2]
	4	004	SP	Reset Initial PC[2]
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12[1]	48	030	SD	(Unassigned, Reserved)
13[1]	52	034	SD	(Unassigned, Reserved)
14[1]	56	038	SD	(Unassigned, Reserved)
15	60	03C	SD	Uninitialized Interrupt Vector
16-23[1]	64	040	SD	(Unassigned, Reserved)
	95	05F		–
24	96	060	SD	Spurious Interrupt[3]
25	100	064	SD	Level 1 Interrupt Autovector
26	104	068	SD	Level 2 Interrupt Autovector
27	108	06C	SD	Level 3 Interrupt Autovector
28	112	070	SD	Level 4 Interrupt Autovector
29	116	074	SD	Level 5 Interrupt Autovector
30	120	078	SD	Level 6 Interrupt Autovector
31	124	07C	SD	Level 7 Interrupt Autovector
32-47	128	080	SD	TRAP Instruction Vectors[4]
	191	08F		
48-63[1]	192	0C0	SD	(Unassigned, Reserved)
	255	0FF		–
64-256	256	100	SD	User Interrupt Vectors
	1023	3FF		–

NOTES

1 Vector numbers 12, 13, 14, 16 through 23, and 48 through 63 are reserved for future enhancements by Motorola No user peripheral devices should be assigned these numbers

2 Reset vector (0) requires four words, unlike the other vectors which only require two words, and is located in the supervisor program space

3 The spurious interrupt vector is taken when there is a bus error indication during interrupt processing Refer to Paragraph 5 5 2

4 TRAP #n uses vector number 32 + n

Fig.8

2. The privilege state is set to supervisory mode and the trace mode is turned off.

3. The processor priority level is set to the level of the interrupt being acknowledged.

4. An interrupt acknowledge signal is sent to the interrupting device.

5. The interrupting device then responds by placing one of the 192 user interrupt vector numbers ($40-$FF) on the address bus.

6. The 68000 multiplies the vector number by four to convert it to a vector address.

7. The current program counter and status register are then saved on the supervisory stack (Since all the operations in progress are in the supervisory mode).

8. The 68000 loads the program counter with the calculated vector address which is the address of the interrupt routine.

Some of the earlier Motorola family devices were not able to provide vector addresses in the above manner. When these devices are used in conjunction with the 68000, the 68000 itself determines the autovector number. Since the priority levels range from 1 to 7 the autovector numbers will range from $19 to $1F.

This simulator is designed to accept only one interrupt at a time. Since this is a software simulation, there are restrictions in implementing multi-level interrupts. In this simulator "control-\ " key is used to simulate an interrupt. When the user hits this key, the processor autovectors to address location (4 * 19) $64 (which is the address of level 1 interrupt autovector in the actual 68000 processor as can be seen from Fig.8). The long word contained at this address is the address of the interrupt routine. The user has to load this address before the simulator starts fetching the instructions. The user manual section explains this procedure in detail. As in the actual machine, the processor enters into the supervisory state and pushes the current program counter and status register onto system stack before branching to the interrupt routine. As in the normal case, on encountering a "RTE" instruction in the interrupt routine, the processor changes state to user mode and returns to the main program. The saved program counter and the status register are restored from the system stack.

The MC68000 has a special reset exception. "Reset" is an external signal designed for system initiation and recovery from serious failures. On receiving this signal the processor is forced to the supervisory state and the trace state is turned off. The interrupt mask is set to the highest level so that no interrupt can interfere with the reset procedure. This exception is different from other

exceptions in that no program counter or status register is saved. The reset exception vector is four words long. Note that all other vectors are only two words long. This vector occupies addresses $00 through $07. The 68000 fetches the first two words and loads them into the system stack and loads the second two words into the program counter. Normally this is the address of the power up/restart routine.

Note that there is an instruction "RESET" whose function is not the same as a hardware reset. The "reset" instruction causes no loading of the reset vector. It allows the user to reset the system to a known state through software. In other words, it asserts the reset line to reset external devices.

This simulator uses the break key to simulate this external hardware reset. As in the case of the interrupt using " control-\ " key the user should load the " start-up " system stack pointer into the long word location starting at address $00 and the " start-up " routine address into the long word location starting at location $04. Details of this procedure are given in the user manual section.

Traps are exceptions caused by instructions. They are initiated on recognition of abnormal conditions during execution of instructions. The execution of a trap has a definite pattern.

(1) The status register is copied into an internal non-addressable register.

(2) The program enters the supervisory state and the trace mode is turned off.

(3) The vector number is internally generated.

(4) The address of the instruction after the instruction which caused trap and the status register are pushed onto the supervisory stack.

(5) Trap exception starts at the address contained in the exception vector.

Traps are of two types.

(1) Those generated due to an abnormal condition during instruction execution .

(2) Those caused by the execution of special trap instructions.

Let us consider case (1). As an example let us take the case of execution of CHK instruction. As already explained in the previous section on the 68000 instruction descriptions, this instruction checks register against some bounds. If the bounds are exceeded a trap is generated internally. Essentially what this means is that an instruction "TRAP 6" is executed internally. This causes the exception vector at address $018 (24 decimal) to be fetched and loaded into the program counter. Note that this

address points to the exception routine. The user can write his own exception routine provided he loads the address of his exception routine in location $018 when the simulator prompts for it in the initial stages. Details are given in the user's manual.

Note that generation of trap vector follows a special algorithm. A " trap n " uses vector number 4*(32 + n).

Next consider case (2) where traps are caused by trap instructions in the program. This helps in forcing an exception into one the 16 user defined trap routines. Instructions TRAP 0 through TRAP 15 causes the program to be transferred to user defined routines whose addresses are in the long word locations $80 through $BC. See fig.9 on page 60. Thus traps act as software interrupts.

There is a serious restriction in the case of traps generated through an instruction. The PIT assembler does not decode the trap instruction properly. This has already been mentioned in Chapter No.4. Hence at present one can have only 8 traps (TRAP 0 through TRAP 7). In this MC68000 simulator three traps have been used for special purposes. This is only implemented for convenience and is not a normal MC68000 feature.

(1) TRAP 5 starts trace.

Instruction	Transfers Program Control Through Vector Address:
TRAP #0	$80
TRAP #1	$84
TRAP #2	$88
TRAP #3	$8C
TRAP #4	$90
TRAP #5	$94
TRAP #6	$98
TRAP #7	$9C
TRAP #8	$A0
TRAP #9	$A4
TRAP #10	$A8
TRAP #11	$AC
TRAP #12	$B0
TRAP #13	$B4
TRAP #14	$B8
TRAP #15	$BC

Fig.9

(2) TRAP 6 stops trace.

(3) TRAP 7 acts like an END directive (It halts the program). Most MC68000 assemblers have this directive but RIT assembler does not. If found unnecessary these features may be removed and used in the normal way to transfer program control through vector addresses $94, $98 and $9C.

The MC68000 also implements a trace feature as already mentioned briefly in the introductory section. This feature assists in program development. When the trace feature is turned on, the 68000 generates an exception after each and every instruction. As in the case of traps the trace causes the program to be transferred to a user-supplied routine in memory through vector address $24. Note that the trace facility uses the T-bit in the supervisor portion of the status register. Hence tracing can be enabled or disabled only in the supervisory mode. Normally the operating system provides a debugging program in assembly language at the long word address at vector location $94. For debugging purposes this simulator a routine is written in "C" to show the contents of all registers and if necessary, the contents of memory locations. It also displays the total number of clock cycles executed so far. This gives an idea of the time needed in an actual MC68000 microprocessor. In addition, it provides a facility to change the contents of registers. This feature has been added only for convenience. The code to transfer control to vector address $94 is written but not used currently. However, in the

future this can be used if necessary.

This simulator also has an additional feature. At any time during execution of a program if the user feels that he needs to set the trace " on " or " off " he can hit the control-z key and the system prompts the user for additional information. These include

(1) Whether he desires to set the trace on/off.

(2) Whether he needs to see contents of any register .

(3) Whether he needs to change contents of memory locations.

From the table showing address assignments for exceptions on page 53 the following have not been covered in the discussion so far.

(1) Bus error (Vector number 2)

(2) Line 101 Emulator (Vector number 10)

(3) Line 1111 Emulator (Vector number 11) These features are related to hardware and hence are not implemented. The bus error signal is an externally generated input that notifies the 68000 of an error somewhere in the system.

CHAPTER 6

CONCLUSION

This section describes the results and conclusions derived from the different testing procedures. The simulator was tested at two stages. (1) Design stage. (2) Implementation.

At the implementation stage after writing code for each instruction the code was tested to determine :

(1) Expected result.

(2) Result of the condition codes.

(3) Whether the instruction works for byte, word or long word operations.

(4) Whether the instruction works for all possible addressing modes. It was during this stage that some anomalies and bugs were detected in the PIT assembler.

At the implementation stage a set of good test programs involving all the crucial operations were developed with the help of different books mentioned in the reference. These test programs were run against the simulator. This helped in detecting minor bugs in the simulator which were rectified. The set of programs used for testing are attached at the appendix C.

Consider the example shown on pages 65-67. The program finds the minimum and maximum word values in an unordered list. The minimum value is returned in memory location "minval"; the maximum value is returned in memory location "maxval". Note that "$ffff" is the smallest number while "$7fff" the largest. The script of the program run is also attached. The addresses of the locations "minval" and "maxval" are noted from the symbol table, which are $444 and $446. Look at the contents of "maxval" and "minval" at memory locations $446 and $444 respectively. It is clear that the maximum value is in "maxval" and minimum value is in "minval" at the end of the program run.

The testing process brought to light a few discrepancies. These discrepancies arose mainly due to the difference in the interpretation of some instructions by the RIT assembler.

The following are the list of instructions where deviations from the MC68000 standards were noticed.

(1) Interpretation of predecrement/postincrement modes of addressing :-

It was observed that in any instruction of type "iii -(Ay), -(Ax)" where "iii" denotes an instruction the assembler generates an opcode which treats address register Ay as destination. This discrepancy should not create much difficulty if the user takes note of this fact while writing codes in assembly language.

```
| This program finds the minimum and maximum word values in an unordered |
| list. The minimum value is returned in memory location minval; the maximum
| value is returned in memory location maxval. The starting address of the |
| list is in address register A0. The length of the list, in words, is in  |
| the list's first word location. For signed values use "bpls chkmax" &    |
| "blss cont" instead of "bccs chkmax" & "bles cont"                       |

        .data
    minval : .word 0              | minimum value location |
    maxval : .word 0              | maximum value location |
    data : .word 5,4,0x7fff,2,7,0xffff
        .text
         lea   data,a0
 minmax : movw a0@+,d1            | move element count into |
 .        subqw  #2,d1            | d1 and decrement it.    |
          movw    a0@,minval      | initially, make first   |
          movw    a0@+,maxval     | element both min and max |
 chkmin : movw    a0@+,d0          | load next element into d0 |
          cmpw     minval,d0      | is this element a new min? |
          beqs     cont
      |    bccs     chkmax      |
          bpls     chkmax
          movw     d0,minval      | Yes. Update minval  |
          bras     cont
 chkmax:  cmpw     maxval,d0       | is this element a new max? |
     |     blss    cont     |
          bles    cont
          movw    d0,maxval       | Yes. Update maxval.  |
 cont:    dbf     d1,chkmin       | End of list?  |
          trap    #6              | set trace |
          trap    #5              | stop trace |
          trap    #7
```

```
% a68* 68k29.s
% ld68* -R 400 68k29.o
% pr68* b.out
Magic Number: 263
Text Size: 68 bytes
Data Size: 16 bytes
BSS Size: 0 bytes
Symbol Table Size: 144 bytes
Text Relocation Size: 0 bytes
Data Relocation Size: 0 bytes
Entry Location: 1024

Text Segment
     0: 41 f9  0  0
     4:  4 48 32 18
     8: 55 41 33 d0
     c:  0  0  4 44
    10: 33 d8  0  0
    14:  4 46 30 18
    18: b0 79  0  0
    1c:  4 44 67 18
    20: 6a  8 33 c0
    24:  0  0  4 44
    28: 60  e b0 79
    2c:  0  0  4 46
    30: 6f  6 33 c0
    34:  0  0  4 46
    38: 51 c9 ff dc
    3c: 4e 46 4e 45
    40: 4e 47  0  0

Data Segment
     0:  0  0  0  0
     4:  0  5  0  4
     8: 7f ff  0  2
     c:  0  7 ff ff

Symbol Table
    0: ET     444 _etext
    1: ED     454 _edata
    2: EB     454 _end
    3:  T     42a chkmax
    4:  T     406 minmax
    5:  D     446 maxval
    6:  D     448 data
    7:  T     416 chkmin
    8:  D     444 minval
    9:  T     438 cont

Text Relocation Commands
Data Relocation Commands
% sim68k b.out
Do you want to load exception routine addresses into memory location.
Say y/n
n
```

```
PC-----> 43e       USP----> fefe      SSP----> fffe      SR---> 8700
R[0]---> ffff      R[1]---> ffffffff  R[2]---> 0         R[3]---> 0
R[4]---> 0         R[5]---> 0         R[6]---> 0         R[7]---> 0
A[0]---> 454       A[1]---> 0         A[2]---> 0         A[3]---> 0
A[4]---> 0         A[5]---> 0         A[6]---> 0         A[7]---> fefe
  c    z    v    n    x
  |    |    |    |    |
  0    0    0    1    0
Do you want any memory location to be displayed ?
y
 please enter lower range location in hex ----->
444
CONTENTS OF SPECIFIED MEMORY LOCATIONS ARE GIVEN BELOW
  LOCATION                    CONTENTS
    444 :   ff ff 7f ff 00 05 00 04 7f ff 00 02 00 07 ff ff
    454 :   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    464 :   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    474 :   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    484 :   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Do you want any memory location to be displayed ?
n
Do you want to change contents of any memory location. Say y/n
n
total clock cycles executed -----> 1686
% ^D
```

(2) Absolute word addressing not supported :- The RIT assembler does not support absolute short or "word" addressing. This should not create any problem as far as programming is concerned since everything is treated as a long address which is alright from the simulator point of view.

(3) "STOP" instruction not supported :- The RIT assembler does not support "stop" instruction. The user might have to load the opcode for this instruction if he needs to use them.

During testing stage of this instruction some problems were encountered while testing all instructions involving "ccr" (condition code register). It was discovered that the RIT assembler does not recognize "ccr" in instructions but instead accepts "cc". Thus the user should note that he should use "cc" and not "ccr" in his assembly code.

For eg: " movew xxx, cc" instead of "movew xxx, ccr".

While testing care was taken to see that values which are not supposed to be affected remain intact. For eg: For byte-sized operations, values of registers were checked before and after the tests to confirm that the higher order bytes remain unaffected.

Points to note :- All instructions having an immediate mode addressing are automatically treated as instructions designed specially for immediate mode.

For example : An " OR <ea>, Dn " is treated as an " ORI xxx, Dn " if <ea> denotes an immediate addressing modes.

Shortcomings from the simulator point of view are related to the fact that this is a software simulation and hence there are cases where it is not possible to exactly simulate the hardware. These cases are discussed below :-

(1) Interrupts :- There are restrictions in implementing multi-level interrupts. This simulator simulates only single level interrupts as was mentioned in the proposal. What this means is that this simulator does not have to take into consideration the priority levels. The simulator can accept only one interrupt at a time.

(2) Reset instruction :- The reset instruction has a very limited purpose on this system. The actual reset instruction sends an electrical signal to reset the external hardware system and continues execution at the next instruction. Naturally this cannot be simulated. Hence essentially this instruction is almost like a " NOP "in our case.

(3) Bus error cannot be implemented as electrical signals to external hardware are involved in this case as well.

Except for these deviations the simulator had no problems and met the required standards of a MC68000 machine.

Before concluding this section it would be interesting to view some of the features which could be added to this simulator in future. The utility of this simulator can be enhanced by adding the facility of recognizing additional interrupts. The code on page no.71 should help in starting future developments on these lines. It would also be helpful to refer to sections "TTY(4)", "IOCTL(2)" and "SIGNAL(3C)" on the UNIX Programmer's Manual in the PIT Professional Lab.

Note that this will work only in "CBREAK" and "RAW" terminal modes. The "CBREAK" mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. The "RAW" mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done. The Programmer's Manual gives details about these modes.

```
/* A format of a "C" program code which will act */
/* as a guideline to works directed towards provision */
/* of additional interrupts                          */

 include <signal.h>    /* Software signal facility */
 include <fcntl.h>     /* file control */

 void main() {
 ...
 ...
 signal( SIGIO, catcher );
 fcntl( 0, FSETOWN, getpid() ); /* A signal should go */
 fcntl( 0, FSETFL, FASYNC );    /* to this process upon */
                                /* available data       */
 ...
 ...
 };

 void catcher() {  /* Signal catcher */
 char c;
 int res;
 res = read( 0, &c, 1 );
 if ( res == 1 )
 switch (c)
  {
   ...              /* Case statements for different values */
   ...              /* of "c". Variable "c" has specific   */
   ...              /* values for different keyboard       */
   ...              /* interrupts                          */
  }
};
```

APPENDIX A

USER MANUAL

There are certain steps to be followed by the user of this simulator.

(1) To create an object module of his assembly program he should issue the command " a68* xxx.s " where xxx.s is the file name.

(2) Next he should load this code starting at memory location $400. For this he should issue the following command.

" ld68* -P 400 "xxx.o" where "xxx.o" is object file obtained from step(1). (Note: $400-$FFFE is the memory area available for user programs.) This step creates a file "b.out".

(3) Next the user should get a printable format of this b.out file using " pr68* b.out " command. This is mainly needed to study the symbol table and note the addresses of various labels used in the program.

(4) To run the program use the command " sim68k b.out

(5) Now comes the most important part which has to be handled with care for proper working of the simulator. The user should load the correct addresses of the following exception routines at the correct vector addresses using the

information obtained from step(3). The vector addresses where they are loaded are given in figure 8 on page 54.

(1) Reset

(2) Address error

(3) Illegal Instruction

(4) Zero Divide

(5) CHK instruction

(6) TRAPV instruction

(7) Privilege violation

(8) Trace (Currently not needed as explained in previous sections)

(9) Privilege violation

(10) Interrupt routine : This will be loaded at $64 as explained in earlier sections. In addition the user will have to load the addresses of various user defined trap routines . Currently only four user trap routines are allowed . Hence only vector addresses $80, $84, $88 and $8C have to be loaded if used. Now the user is all set to run his programs.

For better understanding of this process a copy of a sample program is shown on pages 74-77. It is accompanied by a script of all the commands and outputs.

```
| A program to check validity of CHK instruction & traps. |
| In this program the user has to load the address of the trap |
| routine ( called by trap #1 ) starting at label "suparea". The |
| routine at this address sets trace. The user loads this address |
| at location "$84-$87". The user has also to load the check exception |
| routine address at location "$18-$1b". For the purpose of test in this |
| program we can load an opcode for "stop" instruction.                  |


        .data
init:   .bss 1

        .text
        movw    init,d1
        trap    #1      | exception routine at address $84 |
start:
        addw    #-1,d1      | if you add #1 reg bound exceeeded when D1 = 6 |
        chk     #5,d1
        jmp     start

| A exception routine to set up trace i.e t bit which can be done only in sup
suparea: movb   a7@,d5
        orb     #0x80,d5
        movb    d5,a7@
        rte
```

```
% a68* 68k19.s
% ld68* -R 400 68k19.o
% dr68* b.out
Magic Number: 263
Text Size: 32 bytes
Data Size: 0 bytes
BSS Size: 0 bytes
Symbol Table Size: 86 bytes
Text Relocation Size: 0 bytes
Data Relocation Size: 0 bytes
Entry Location: 1024

Text Segment
     0: 32 39  0  0
     4:  4 20 4e 41
     8:  6 41 ff ff
     c: 43 bc  0  5
    10: 4e f9  0  0
    14:  4  8 1a 17
    18:  0  5  0 80
    1c: 1e 85 4e 73

Data Segment

Symbol Table
    0: ET     420 _etext
    1: ED     420 _edata
    2: EB     420 _end
    3:  T     408 start
    4:  T     416 suparea
    5:  D     420 init

Text Relocation Commands

Data Relocation Commands
% sim68k b.out
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------> 18
Enter new content in hex --------> 0
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------> 19
Enter new content in hex --------> 0
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------> 1a
Enter new content in hex --------> 1a
Do you want to load exception routine addresses into memory location.
Say y/n
y
```

```
Enter location in hex --------) 1b
Enter new content in hex --------) 0
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 1a00
Enter new content in hex --------) 4e
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 1a01
Enter new content in hex --------) 72
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 1a02
Enter new content in hex --------) 0
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 1a03
Enter new content in hex --------) 0
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 84
Enter new content in hex --------) 0
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 85
Enter new content in hex --------) 0
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 86
Enter new content in hex --------) 4
Do you want to load exception routine addresses into memory location.
Say y/n
y
Enter location in hex --------) 87
Enter new content in hex --------) 16
Do you want to load exception routine addresses into memory location.
Say y/n
n
```

```
 PC-----> 408       USP----> fefe      SSP----> fffe      SR---> 8700
 R[0]---> 0         R[1]---> 0         R[2]---> 0         R[3]---> 0
 R[4]---> 0         R[5]---> 87        R[6]---> 0         R[7]---> 0
 A[0]---> 0         A[1]---> 0         A[2]---> 0         A[3]---> 0
 A[4]---> 0         A[5]---> 0         A[6]---> 0         A[7]---> fefe
  c    z    v    n    x
  |    |    |    |    |
  0    0    0    1    0
Do you want any memory location to be displayed ?
n
Do you want to change contents of any memory location. Say y/n
n
total clock cycles executed -----> 322
 PC-----> 40c       USP----> fefe      SSP----> fffe      SR---> 8700
 R[0]---> 0         R[1]---> ffff      R[2]---> 0         R[3]---> 0
 R[4]---> 0         R[5]---> 87        R[6]---> 0         R[7]---> 0
 A[0]---> 0         A[1]---> 0         A[2]---> 0         A[3]---> 0
 A[4]---> 0         A[5]---> 0         A[6]---> 0         A[7]---> fefe
  c    z    v    n    x
  |    |    |    |    |
  0    0    0    0    0
Do you want any memory location to be displayed ?
n
Do you want to change contents of any memory location. Say y/n
n
total clock cycles executed -----> 364
 REGISTER  BOUNDS  EXCEEDED---  PROGRAM  HALTED
 STOP....PROCESSOR WAITING FOR INTERRUPT
^CBREAK KEY\^C interrupt!!!!!!!!!!!!!!
BREAK KEY\^C interrupt!!!!!!!!!!!!!!
        HARDWARE RESET
POWER-UP/RESTART  IN PROGRESS
% ^D
```

Note the output after executing the command " pr68* b.out ". The symbol table gives the addresses of the labels in the program. The routine under the label " suparea " is the supervisory mode program which sets the trace bit. This is clear from the program given. This code is entered through the trap instruction " trap 1 ". From fig.9 on page 60 it is clear that " trap 1 " transfers program control through vector address $84. It should also be noted that " chk bound exceeded " exception address is in the location $18-$1b (Figure 8). Look at the copy of the script. To run the program the command " sim68k b.out " is entered. Notice that the simulator prompts for addresses of exception routines. An address " 1a00 " is entered at location $18-$1b. This is the address of the exception routine. In this case an opcode for " stop " instruction is entered beginning at this address (The opcode entered is " 4e720000 "). From the symbol table it was observed that the address of " suparea " is " $416 ". This is entered in the memory location $84-$88 so that the program transfers control to the routine at " suparea " through " trap 1 " instruction. This sets the trace bit. Notice how the trace mode is entered and all registers are displayed. The script also shows that the processor enters the halt or " stop " state once the register bounds are exceeded. For the purpose of testing, the break key has been utilized to stop the program. Hence in the halted state, when the processor is waiting for an interrupt, pressing the break key halts the program. In the actual routine "power up/ restart " routine

is executed on recognizing an external hardware reset.

It is suggested that the user load the exception routine area with the " stop instruction " opcodes in the manner shown in the previous paragraph, if he does not have a specific exception routine while testing or running his program. This will take care of default cases where the processor halts and waits for interrupts if proper interrupt routines are not loaded.

APPENDIX B

REFERENCES

1. Leo J.Scanlon , " The 68000 : Principles and Programming ", The Blacksburg Continuing Education Series, 1983.

2. John F. Wakerly , " Microcomputer Architecture and Programming ", John Wiley and Sons, Inc., 1981.

3. Dave Bursky, " MC68000 16-Bit Microprocessor to Offer Wide Address Range, Powerful Commands ", Electronic Design 15, July 19, 1978.

4. Edward Stritter and Tom Gunter, " A Microprocessor Architecture for a Changing World ", Computer, Feb 1979 (MC68000 Article Reprints).

5. John Zolnowsky and Nick Tredennick, " Design and Implementation of System Features for the MC68000 ", Proceedings of Compcon, Fall 1979 (MC68000 Article Reprints).

6. W. Richards Adrion," Validation, Verification and Testing of Computer Software ", ACM Computing Surveys, Vol 14, Number 1, March 1982.

7. Ian H. Witten, " The new microprocessors ", Advanced Microprocessors, IEEE press selected reprint series.

8. Hoo-min D. Toong and Amar Gupta, " An Architectural Comparison of Contemporary 16-bit microprocessors ", Advanced Microprocessors, IEEE Press selected reprint series.

9. P. Heidelberger and S.S. Lavenberg, " Performance Evaluation ", IEEE Transaction on Computers, May 1984.

10. Walter A. Triebel and Avtar Singh, " 16-bit Microprocessors ", Prentice-Hall, Inc., 1985.

11. Thomas L. Harman and Barbara Lawson, " The Motorola MC68000 Microprocessor Family ", Prentice-Hall, Inc., 1985.

12. M68000 16/32-bit Microprocessor , Programmer's Reference Manual, Fourth Edition, Prentice-Hall, Inc., 1984.

TEST PROGRAMS

```
| Find the average of "n" signed numbers                          |
| The answer is in register 7 & memory location "average"         |

        .data
    n   =   5
     data : .word -65,-6,9,4,3
   |    data : .word 3,12,-2,16,10    ex of remainder   |
        .bss
         average:   .=.+4
        .text
         clrl    d7
         lea     data,a1
         nop
         movl    #n-1,d0
 nxtpt:  movw    a1@+,d1
         extl     d1
         addl     d1,d7
         dbf      d0,nxtpt
         divs     #n,d7
         trap     #6        | set trace on |
         trap     #7        | end |
```

```
| A program to check validity of CHK instruction & traps. |
| In this program the user has to load the address of the trap |
| routine ( called by trap #1 ) starting at label "suparea". The |
| routine at this address sets trace. The user loads this address |
| at location "$84-$87". The user has also to load the check exception |
| routine address at location "$18-$1b". For the purpose of test in this |
| program we can load an opcode for "stop" instruction.                  |


          .data
         init: .bss 1

          .text
         movw    init,d1
         trap    #1      | exception routine at address $84 |
start:
         addw    #-1,d1      | if you add #1 reg bound exceeeded when D1 = 6 |
         chk     #5,d1
         jmp     start

| A exception routine to set up trace i.e t bit which can be done only in sup
suparea: movb  a7@,c5
         orb   #0x80,d5
         movb  c5,a7@
         rte
```

```
| Given a number N in the range 0 < N <= 5, a program that computes its |
| factorial and saves the result in D1  |

        .data
      n : .byte  5


         .text
     moveq    #1,d7
     clrw     d6
loop: cmpb    n,d6
      beq     done
      addqw    #1,d6
      mulu     d6,d7
      bra      loop
done: movl     d7,d1
      trap     #6
      trap     #7
```

```
;Privilege violation example. User cannot use "reset" instr in his program
                .text
        nop
        nop
        jmp sup
        nop
        trap #6
        trap #7
sup:    reset
        rte
```

```
| This program finds the minimum and maximum word values in an unordered |
| list. The minimum value is returned in memory location minval; the maximum |
| value is returned in memory location maxval. The starting address of the |
| list is in address register A0. The length of the list, in words, is in |
| the list's first word location. For signed values use "bls chkmax" &     |
| "blss cont" instead of "bcs chkmax" & "ble cont"                         |

        .data
    minval : .word 0              | minimum value location |
    maxval : .word 0              | maximum value location |
    data : .word 5,4,0x7fff,2,7,0xffff
        .text
         lea   data,a0
minmax : movw a0@+,d1             | move element count into |
         subqw  #2,d1             | d1 and decrement it.    |
         movw    a0@,minval       | initially, make first   |
         movw    a0@+,maxval      | element both min and max |
chkmin : movw    a0@+,d0          | load next element into d0 |
         cmpw     minval,d0       | is this element a new min? |
         beq      cont
      |   bcc      chkmax      |
         bls      chkmax
         movw     d0,minval       | Yes. Update minval  |
         bra      cont
chkmax:  cmpw     maxval,d0       | is this element a new max? |
     |    blt     cont     |
         ble      cont
         movw     d0,maxval       | Yes. Locate maxval.  |
cont:    dbf      d1,chkmin       | End of list?  |
         trap     #6              | set trace |
         trap     #5              | stop trace |
         trap     #7
```

```
| This program calculates the binary sine value for the angle contained |
| in the low word of data register D0, using a look-up table. The signed |
| sign is returned in the low byte of data register D1. |
| Here we test whether the binary value of sine of 177 is 6 or not |

   .data
sintab: .byte  0,2,4,6,8

   .text
        clrw    d0
        movw   #177,d0
sinang:
        clrb    d1
        cmpb    #180,d0
        blss    sindos
        tas     d1
        subw    #180,d0
 sindos:
        cmpb    #91,d0
        bmis     getsin
        negw      d0
        addw     #180,d0
   getsin:
         lea     sintab,a0
         orb     a0@(0,d0:l).d1
         trap    #6    | set trace on |
```

```
| Example involving movem, suba, exg, Bne & compares |
| The program compares 16-bit positive integers in an array stored in |
| locations NUM1, NUM1 + 2, and so on, and leaves the largest one in D1 |
| Register D3 is used as a counter to determine when all the numbers |
| have been tested . The count of numbers is assumed to be in D3 before |
| the program executes. Note: The numbers are unsigned. |

            .data
        data : .word 0xFFFE,6,8,12,76,0xFFFF
            .text
             movl    #6,d3              | load count of numbers |
             moveml  #0x3080,sp@-
             movl    #data,a1
             subqw   #1,d3
             movw    a1@+,d1
loop:        movw    a1@+,d2
             cmpw    d1,d2
             bls     next
             exg     d1,d2
next:        subqw   #1,d3
             bne     loop
             moveml  sp@+,#0x020c
             trap    #6
             trap    #7
```

```
; Example--- Importance of conditional branches & tst, neg, add, and sub
; instructions.
; This routine sums the difference between two columns or vectors of integers
; addressed by A1 & A2. The length of the columns is initially held in D1
; If length of the columns is zero or an overflow occurs D4 is set to zero
; to indicate the error. Otherwise result is in D3. If the final result is
; -ve "neg" gives an absolute value.

            .data
;           col1 : .word 7,8,12,1
            col1 : .word 0,0,0,0
            col2 : .word 0x7FFF,1,22,10
;           col2 : .word 5,9,2,0

.           .text
            lea     col1,a1
            lea     col2,a2
            moveq   #4,d1
sumdif:
            clrw    d3
            clrw    d4
            tstw    d1
            beq     done

loop:       movw    a1@+,d2
            nop
            subw    a2@+,d2
            nop
            addw    d2,d3
            bvs     done
            subqw   #1,d1
            one     loop
            tstw      d3
            bge       oos
            movw      #-1,d4
            negw      d3
            bra       done
oos:
            moveq   #1,d4
done:
            trap      #6
            trap      #7
```

Test a table containing word-length operands to locate a non-zero entry. a1 contains the first address of the table and d1 contains count. If d1 contains a -1, table contains all zero values. Else a1 points to the first non-zero entry . An example for Bcc instuction

```
            .data
      data :.word  0,  0 ,0,0,0
      data :.word  0,9,0,8,9
            .text
           movl  #data,a1
           moveq #5,d1
   loop1 :
           tstw  a1@+
           bne   done1
           subqw  #1,d1
           bpl    loop1
   done1:  subql  #2,a1
           trap   #6
           trap   #7
```

```
|  16-bit bubble sort routine   |
| arrange 16-bit elements of a list in ascending order in memory |
| The starting address is in A0. The length of the list - 1 is in |
| the first word location . Note: Numbers are unsigned. |

      data
    data : .word 4,0xffff,6,9,1,2
       .bss
       .text
          movl   #data,a0
   sort:  clrb   d1                  | exchange flag = 0 |
          movw   a0@+,d3             | load word count into d3 |
   loop:
          movl   a0,a1               | load element addr. into a1 |
          subqw  #1,d3               | decrement word count |
          movw   d3,d0               | and load it into counter D0 |
   comp:  movw    a1@+,d2            | fetch word into d2 |
          cmpw    a1@,d2             | Is next word greater than this word ? |
          blss    decctr             | Yes. Continue |
          movw    a1@,a1@(-2)        | No . Exchange |
          movw    d2,a1@             | * these two words |
          tas      d1                | turn on exchange flag |
   decctr:dbf    d0,comp             | End of list ? |
          notb     d1                | Yes. Is exchange flag on ? |
          bpls    loop               | If so, star over |
          trap    #6                 | trace on |
          trap   #7                  | end   |
```

```
| Sorting an array of 16-bit signed binary numbers such that they are |
| rearranged in ascending order. The numbers are stored in memory at  |
| consecutive locations from address of "data"                        |
      .data
    n   =   5
!   data : .word 0xf,9,1,2          |
  data : .word 9,0xffff,6,4     | Note: ffff is -1 and of least magnitude |
       .even
       .text
start: movl   #data,a1
     movl    #data + 6,a3
AA:  movl    a1,a2
     addql   #2,a2
BB:  movw    a2@,d0
     cmpw    a1@,d0
     bgts    CC
     movw     a1@,a2@
     movw     d0,a1@
CC:  addql    #2,a2
     cmpl     a3,a2
     bles     BB
     addql    #2,a1
     cmpl     a3,a1
     blt      AA
     trap     #6
     trap     #7
```

```
| This program performs a binary search of a table of array for a word |
| length bit pattern called a key value. The table is composed of 5 |
| entries which are each 2 bytes long. The starting address of the table |
| is supplied in A0 and the key to locate in D0. The low-order word of D1 |
| contains the length of each entry and D2 contains the number of entries|
| in the table . If the search is successful the address of the key value |
| is returned in A6. Otherwise A6 contains zero. Since a binary search is |
| performed, the data are assumed to be sorted numerically in the table |
| being searched                                                          |

          .data
        |  data: .word 4,5,6,7  |
          data: .word 6,7,8,9,10

          .text
        lea     data,a0
        moveq   #2,d1
        moveq   #8,d0    | d0 contains the key value i.e the search value |
        moveq   #5,d2
search: subqw   #1,d2
        clrl    d3
        movl    d3,a6

ser10:  cmpw    d2,d3
        bgt     exit
        movw    d3,d4
        addw     d2,d4
        lsrw     #1,d4
        movw     d4,d5
        mulu     d1,d5
        nop
        cmpw     a0@(0,d5:w),d0
        nop
        bge      ser20

        subqw    #1,d4
        movw     d4,d2
        bra      ser10

ser20:  beq      success
        addqw    #1,d4
        movw     d4,d3
        bra      ser10

success: lea     a0@(0,d5:w),a6

exit:   trap     #6
        trap     #7
```

```
| DELETING AN ELEMENT FROM AN UNORDERED LIST      |
| This subroutine deletes the value in the low word of data register |
| d0 from an unordered list if that value is in the list. The starting addres
| of the list is in address register A0. The length of the list, in words, |
| is in the list's first word location.                                    |


        .data
    data: .word 4,1,8,9,5
        .text
        lea     data,a0
        movw     #9,d0

deleful:  movl   a0,a1          | copy starting address into A1 |
          movw    a1@+,d1       | and word count |
          subqw    #1,d1        | minus 1 into  d1 |

nextel:   cmpw    a1@+,d0       | delete victim found? |
          beqs    delete        | Yes. go delete that element |
          dbf     d1,nextel     | No. Search until end of |
          bras     alldun       | of list, then exit (element not in list ). |

| Delete an element, by moving all subsequent elements up by one word locatio

delete:   movw    a1@+,a1@(-4)  | Move one word up in list |
          dbf      d1,delete    | Have all elements been moved? |

          subqw    #1,a0@       | Yes. Decrement element count.  |
alldun:   trap     #6           | Set trace |
          trap     #5           | Stop trace |
          trap     #7
```