Tektronix®
COMMITTED TO EXCELLENCE

This manual supports the following TEKTRONIX products:

8550 Options	8540 Options	Products
2P	2P	8300E26
3U	3U	8300P26

This manual supports a software/firmware module that is compatible with:

DOS/50 Version 2 (8550)
OS/40 Version 1 (8540)

PLEASE CHECK FOR CHANGE INFORMATION AT THE REAR OF THIS MANUAL.

8500 MODULAR MDL SERIES 68000 EMULATOR SPECIFICS USERS MANUAL

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

070-3970-00
Product Group 61

Serial Number ____________________

First Printing FEB 1982

Section 7L

68000 EMULATOR SPECIFICS

TABLES

ILLUSTRATIONS

Section 7L

68000 EMULATOR SPECIFICS

INTRODUCTION

This supplement is designed to be inserted into Section 7 of the 8550 System Users Manual (DOS/50 Version 2) or the 8540 System Users Manual. This Emulator Specifics section explains the features of the 8550 and 8540 that are unique to the 68000 emulator. Throughout this section, "your System Users Manual" refers to the 8550 System Users Manual or 8540 System Users Manual.

The 68000 demonstration run is designed to be used with Section 1, the Learning Guide of your System Users Manual; the rest of this section contains reference material.

As a user of the 68000 emulator, you should be familiar with the material in the MC68000 16-Bit Microprocessor User's Manual, by Motorola. In addition, you should be familiar with the internal operation of the 68000. Information is available in the booklet MC68000 Article Reprints, by Motorola. Three of the more pertinent articles from that booklet have been reprinted here in the subsection "Reprints". Some of the effects of the 68000's design on the behavior of the emulator are discussed under the topic, "Special Considerations".

GENERAL INFORMATION

EMULATOR HARDWARE CONFIGURATION

Throughout this Emulator Specifics section, the term "68000 emulator" refers to a 68000 Emulator Processor board configured with a 68000 Prototype Control Probe. In emulation mode 0, the prototype control probe must be connected to the main emulator boards. In modes 1 and 2, the prototype control probe must be connected to both the emulator and your prototype. For instructions on installing your emulator boards and probe, refer to your 68000 Emulator Processor and Prototype Control Probe Installation Service Manual.

MICROPROCESSORS SUPPORTED

The 68000 emulator emulates the Motorola MC68000 microprocessor.

EMULATION MODES

The 68000 supports emulation modes 0, 1, and 2, as described in the Emulation section of your System Users Manual.

CLOCK RATES

In emulation mode 0, the emulator clock rate is 8MHz. In emulation modes 1 and 2, the prototype clock rate may range from 2MHz to 8MHz.

SYMBOLIC DEBUG

The 68000 emulator supports the use of symbolic debug. Most of the displays in this manual include symbolic debug information.

EMULATOR-SPECIFIC PARAMETERS, COMMANDS, AND DISPLAYS

BYTE/WORD PARAMETER

Several commands allow you to operate on memory on a byte-oriented or word-oriented basis. This choice is represented by the -B or -W parameter. For the 68000 emulator, the default setting is -W, except for the MOV, RD and WRT commands, where the default is -B.

REGISTER DESIGNATORS

Table 7L-1 alphabetically lists the symbols used by DOS/50 and OS/40 to designate the registers and flags used by the 68000. The table provides the following information for each symbol:

- a description of the register or flag that the symbol represents
- the size of the register or flag
- the value assigned to the register or flag by the RESET command
- whether the register or flag can be assigned a value by the S (Set) command.

Table 7L-1
68000 Registers and Flags

Symbol	Description	Size (Bits)	Value After RESET	Altered by S Command?
A0--A6	seven address registers	32	unchanged	yes
C	Carry bit of CCR	1	unchanged	yes
CCR	Condition Code portion of SR	5	unchanged	yes
D0--D7	eight data registers	32	unchanged	yes
I	Interrupt mask of SR	3	7	yes
N	Negative bit of CCR	1	unchanged	yes
PC	Program Counter	24	contents of SP:000004 *	no
S	Supervisor/User bit of SR	1	1 (on)	yes
SR	Status Register	16	27XX	yes
SSP	Supervisor Stack Pointer (A7)	32	contents of SP:000000 *	yes
T	Trace bit of SR	1	0 (off)	yes
USP	User Stack Pointer (A7)	32	unchanged	yes
V	oVerflow bit of CCR	1	unchanged	yes
X	eXtend bit of CCR	1	unchanged	yes
Z	Zero bit of CCR	1	unchanged	yes

* SP: is Supervisor Program space.

THE 68000 STATUS REGISTER (SR)

The 16-bit Status Register (SR) has a system byte (high order) and a user byte (low order). The system byte contains the Trace Mode and Supervisor State bits, and contains the three bits used as the Interrupt Mask. The user byte contains five status flag bits, which are used primarily for branch control within a program, and for error detection. Figure 7L-1 shows the Status Register.

```
                System Byte                       User Byte

       _____________/\_______________  ______________/\_______________
      /                              \/                               \
        15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
      | T |   | S |   |   | I2| I1| I0|   |   |   | X | N | Z | V | C |
      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        |       |         \_____ _____/             |   |   |   |   |
        |       |               v                   |   |   |   |   |
        |       |               |                   |   |   |   |   |
  Trace +       |               |                   |   |   |   |   |
                |               |                   |   |   |   |   |
  Supervisor ---+               |                   |   |   |   |   |
    /User                       |                   |   |   |   |   |
                                |                   |   |   |   |   |
  Interrupt  -------------------+                   |   |   |   |   |
    Mask                                            |   |   |   |   |
                                                    |   |   |   |   |
             / Extend ------------------------------+   |   |   |   |
             |                                          |   |   |   |
             | Negative --------------------------------+   |   |   |
  Condition  |                                              |   |   |
     Codes   < Zero ----------------------------------------+   |   |
             |                                                  |   |
             | Overflow ----------------------------------------+   |
             |                                                      |
             \ Carry -----------------------------------------------+
```

Fig. 7L-1. Status Register.

DOS/50 AND OS/40 COMMANDS

AL---Allocate Memory to Logical Memory Map

The AL command allocates 4K-bytes blocks of program memory to the processor's logical memory space. The MAC (Memory Allocation Controller) option must be installed. For example, the command AL SP:UP:0 allocates one block of program memory for location 0--0FFF of the two memory spaces SP and UP. Thus, for the range 0--0FFF, these two memory spaces will be in the same physical memory. References to either of those two memory spaces in that range will access the same physical location. For example, the address 0A40 in the supervisor program space (SP) will be at the same physical location as the address 0A40 in the user program space (UP).

With the 68000 emulator, the AL command may be used only if the Memory Allocation Controller (MAC) option is installed. The use of the AL command does not follow that described by the Command Dictionary of your System Users Manual. Refer to the Emulation section of your System Users Manual for a detailed description of the AL command when the MAC option is installed.

BK---Sets or Displays Breakpoint Conditions

With the 68000 emulator, the BK command syntax and parameters are not as described in the Command Dictionary of your System Users Manual. Use the following information instead.

SYNTAX

```
bk

or

   {1  }
   {2  }
   {3  }
bk {all}  clr

or

        {1 [-a]}
        {2 [-a]}               [rd] [by]
bk [-c] {3     } [expression] [wt] [wd]
```

PARAMETERS

1, 2, and 3. The number specifies the desired breakpoint.

ALL. This specifies all breakpoints.

CLR. This parameter clears the specified breakpoint(s).

-C. This parameter causes execution to continue after each breakpoint occurs. If -C is not specified (the default), the BK command stops execution after a breakpoint occurs. To resume program execution, enter the G command without parameters.

-A. This mode sets breakpoint 1 to arm breakpoint 2. When the mode is set, breakpoint 2 will not occur unless breakpoint 1 has already happened. Arming mode may be set when entering either breakpoint 1 or breakpoint 2, but one of these two breakpoints must already be defined. When you redefine either breakpoint 1 or 2 this setting is cancelled.

expression. This parameter is an expression that represents the address where program execution is to be interrupted. The expression may include don't-care bits and/or a memory space designator. The address expression may also be omitted. For example, BK 1,,BY WT will cause a break on the first byte write.

RD and WT. These parameters designate that a breakpoint occurs when a memory read (or write) occurs at the specified address. The default is any access (read or write).

BY and WD. These parameters designates that a breakpoint occurs when a byte (or word) operation occurs at the specified address. The default is any access (byte or word).

NOTE

When you use TRA, and breakpoints are set, the break may occur before the address where the breakpoint is set. This occurs because the address has been identified going into the 68000 prefetch pipeline. Be sure to check the display to see if the last instruction executed was the one on which you wanted to break. If it is not, enter G again, and the next instruction will be executed. Check the display again, and repeat the G command, if necessary. It is recommended that you put NOP statements in your program around the statements where you want to break.

D---Dump; Displays Memory Contents

The D command allows memory space designators in the address expressions.

DI---Disassembles Object Code into Mnemonics

The DI command translates object code in memory into assembly language instructions. It displays addresses, object code, assembly language mnemonics, and operands.

DI Display Format. In general, the format of the disassembly follows the conventions of both the Motorola cross-assembler and the TEKTRONIX 8500 Series B 68000 Assembler. An example of a disassembled instruction is:

```
000712   13C0  MOVE.B  D0,F00007H
```

In this example:

000712	is the memory location of the instruction being disassembled.
13C0	is the opcode.
MOVE	is the opcode mnemonic.
.B	is the size extension. In the DI display, the size extension may be shown on some instructions where it may not be required or allowed for assembly.
D0	is the source operand.
F00007H	is the destination operand.

Exceptions from the Assembler Format. The opcode variations ADDA, ADDI, ANDI, CMPA, CMPI, EORI, MOVEA, ORI, SUBA, and SUBI do not appear in the disassembly. The assembler chooses the correct A or I opcode variation by examining the operands in the instruction, rather than by the opcode suffix. Thus, ADDI #035FH,D0 and ADD #035FH,D0 would both generate the same opcode, and their disassembly will be displayed as ADD.W #035FH,D0.

The opcodes for BT and DBF are disassembled as the equivalent mnemonics, BRA and DBRA, respectively.

The mnemonics EMT_A and EMT_F are displayed whenever an attempt is made to disassemble opcodes in the ranges A000--AFFF and F000--FFFF. These opcode ranges are reserved by Motorola for future enhancements.

A line of asterisks (********) is displayed in the instruction field if an attempt is made to disassemble an illegal opcode.

A sample disassembly is shown in Fig. 7L-2.

```
> DI 3000 3022 <CR>
  ADDRESS    DATA  MNEMONIC

  DEMO+000000
  003000     4280  CLR.L   D0

  DEMO+000002
  003002     323C  MOVE.W  #3H,D1

  DEMO+000006
  003006     207C  MOVE.L  #1000H,A0

  DEMO+00000C
  00300C     227C  MOVE.L  #2000H,A1

  LOOP
  003012     31FA  MOVE.W  3024H,3026H

  DEMO+000018
  003018     22D8  MOVE.L  (A0)+,(A1)+

  DEMO+00001A
  00301A     51C9  DBF     D1,3012H

  DEMO+00001E
  00301E     4E71  NOP

  DEMO+000020
  003020     4E71  NOP

  SELF
  003022     60FE  BT      3022H
```

Fig. 7L-2. Sample disassembly.

This display was generated when SYMD was ON.

DS---Display Contents of Emulator Processor Registers

The DS command displays the 68000 registers. The display contains PC(next), the fifteen 32-bit general registers, the system and user stack pointers, and the system status register. The status register is displayed in hexadecimal, and in binary with each bit labelled. Refer to the following example:

```
> DS

PC=00132C
D0=0000000F  D1=0001FF00  D2=00000000  D3=00000000
D4=00BC48FF  D5=00000000  D6=00000000  D7=00000000
A0=00F00000  A1=00000000  A2=00000000  A3=00001000
A4=00000004  A5=00000008  A6=00000000 SSP=00DC1000 USP=00100000

             T.S. .III ...X NZVC
SR=850A ---> 1.0. .101 ...0 1010
```

The long and short forms of the DS display are the same: the -L modifier has no effect.

EX---Displays or Alters Memory Contents

The EX (EXamine) command allows memory space designators in the address expression.

An error will occur if you attempt to examine memory that is not on a word boundary while in word mode (-W).

F---Fills Program/Prototype Memory with Data

The F command allows memory space designators in the address expression.

An error will occur if you attempt to fill memory that is not on a word boundary while in word mode (-W).

G---Begins Program Execution

The G (Go) command starts the emulator at the 24-bit address specified. The processor examines the S bit of the 68000 status register to determine whether to start in supervisor or user program space. Memory space designators are not allowed; however, you may select the program space with the Set command.

If breakpoints have been set, it may be necessary to invoke G more than once to actually execute the instruction on which you wish the break to occur. This is because the 68000 performs prefetching and does not have a fetch signal. Refer to the discussions under "Special Considerations" later in this section. Also, refer to the BK command discussion.

MAP---Sets or Displays Memory Map Assignments

The MAP command enables you to assign blocks of memory to either program or prototype memory, and to designate blocks of memory as read-only. The 68000 MAP command differs from the description given in the Command Dictionary as follows:

- The -M modifier is not allowed. Displays are in tabular form only.
- Entering MAP with no parameters causes the display of the current memory map assignments for the default memory spaces (MEMSP M).
- Entering MAP -A displays the current memory map assignments for all valid memory spaces.
- Entering MAP followed by one or more memory space designators displays the current mapping for the indicated memory space(s).
- The block size default is 4K bytes.
- Unlike most other emulators, the 68000 emulator supports write protection for prototype memory through use of the URO parameter.
- Multiple memory space designators are allowed as part of the loaddr parameter. They are not allowed in the hiaddr parameter. The hiaddr parameter defaults to the same memory space(s) as loaddr.

If you attempt to change the program/prototype map assignments and are not currently in emulation mode 1, a warning message is displayed indicating that you are not in mode 1. The mapping of read-only and read/write is valid in any emulation mode.

Here is a sample 68000 MAP assignment and display:

```
> MAP PRW 000000 70FFFF <CR>
> MAP URW SD:710000 7FFFFF <CR>
> MAP -A <CR>

UD 000000-FFFFFF  PRW

UP 000000-FFFFFF  PRW

SD 000000-70FFFF  PRW
SD 710000-7FFFFF  URW
SD 800000-FFFFFF  PRW

SP 000000-FFFFFF  PRW
```

MEMSP---Defines Default Memory Space

The 68000 emulator's default MEMSP settings are as follows:

```
> MEMSP <CR>
  Default single memory space......SP
  Default multiple memory spaces...UD UP SD SP
```

MOV---Moves Data Between Program and Prototype Memory

The MOV command allows memory space designators in the address expressions.

For example, the command:

```
> MOV UD:3640 4340 SD:2000 <CR>
```

moves User Data space 3640--4340 to Supervisor Data space beginning at address 2000.

An error occurs if you attempt to move data that is not on a word boundary while in word mode (-W). The default mode for the MOV command is -B.

P---Alters Memory Contents

The P (Patch) command allows a memory space designator in the address expression.

RD---Reads from Emulator Port

Because the 68000's I/O is memory-mapped, the 68000 RD command works like the D command, except that prototype memory is referenced, and only one word is displayed.

RESET---Reinitializes Emulator

The RESET command simulates a hardware reset by modifying the registers of the 68000 as follows:

- The Trace bit is turned off (0).
- The Supervisor bit is turned on (1).
- The interrupt level is set to 7.
- The stack pointer is loaded from Supervisor Program memory location 000000.
- PC(next) is set to the value in Supervisor Program memory location 000004.

Here is an example of register contents before invoking RESET:

```
     > DS <CR>

     PC=00132C
     DO=0000000F  D1=0001FF00  D2=00000000  D3=00000000
     D4=00BC48FF  D5=00000000  D6=00000000  D7=00000000
     A0=00F00000  A1=00000000  A2=00000000  A3=00001000
     A4=00000004  A5=00000008  A6=00000000 SSP=00DC1000 USP=00100000

                  T.S. .III ...X NZVC
     SR=850A ---> 1.0. .101 ...0 1010
```

After RESET, the display changes. Assume for this example that the contents of SP:000000 is 000000, and that the contents of SP:000004 is 020000. The arrows show the altered registers and bits:

```
     > RESET <CR>
     > DS <CR>

           |
           |
           |
           v
        ======
     PC=020000
     DO=0000000F  D1=0001FF00  D2=00000000  D3=00000000
     D4=00BC48FF  D5=00000000  D6=00000000  D7=00000000
     A0=00F00000  A1=00000000  A2=00000000  A3=00001000
     A4=00000004  A5=00000008  A6=00000000 SSP=00000000 USP=00100000
                                                 ========
                  T.S. .III ...X NZVC                ^
     SR=270A ---> 0.1. .111 ...0 1010                |
        ==        = =   ===                          |
        ^         ^ ^    ^                           |
        |         | |    |
        |         | |    |
        |         | |    |
```

S---Assigns Value to Register or Symbol

The S (Set) command changes the values of the 68000's registers. The symbols allowed and the registers they represent are shown in Table 7L-2.

Table 7L-2
Register Symbols Accepted by S Command

Symbol	Register
D0--D7	the eight 32-bit data registers
A0--A6	the seven 32-bit address registers
SR (*a)	the 16-bit Status Register
T	the Trace bit of the SR
S	the Supervisor bit of the SR
I	the three Interrupt level bits of the SR
CCR	the 5-bit Condition Code part of the SR
X	the eXtend bit of the CCR
N	the Negative bit of the CCR
Z	the Zero bit of the CCR
V	the oVerflow bit of the CCR
C	the Carry bit of the CCR

*a---The emulator does not check whether you have specified values for the unused bits in the Status Register.

SEA---Searches Memory for Value or String

Try to limit your value and string searches to the smallest portion of memory necessary. The hiaddr parameter of the SEA command defaults to the end of memory. Therefore, it is strongly recommended that you specify the hiaddr parameter of the SEA command. Otherwise, you may experience extremely lengthy search times, due to the 68000's large memory capability.

SEL---Selects the Emulator

The following command selects the 68000 emulator:

```
     > SEL 68000 <CR>
```

The system responds with the software version number and version date. The emulator hardware need not be in the system when you SELect it.

NOTE

The 68000 emulator cannot be selected while you are programming a PROM.

TRA---Controls Display of Executed Instructions

The TRA command sets the conditions for displaying trace lines during program execution. Memory space designators may be used when defining the loaddr parameter. However, they may not be used in hiaddr. If a memory space is not designated, the default is to all memory spaces. Here is a sample 68000 TRA display:

```
     > SYMD -SL ON <CR>
     > TRA ALL <CR>
     > G MAIN <CR>

     MAIN
     UP:001000  227C  MOVE.L  #F00000H,A1
         PC=001006
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000505   A1=00F00000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     PROG+000006
     UP:001006  207C  MOVE.L  #500H,A0
         PC=00100C
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00F00000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     PROG+00000C
     UP:00100C  323C  MOVE.W  #4H,D1
         PC=001010
         D0=0000000F   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00F00000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000
     PROG+000010
     UP:001010  4280  CLR.L   D0
         PC=001012
         D0=00000000   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00F00000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0004

     PROG+000012
     UP:001012  D018  ADD.B   (A0)+,D0
         PC=001014
         D0=00000001   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000501   A1=00F00000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000
       <BREAK     TRACE,ESC>
```

The lines below each assembly language instruction show the program counter, the data and address registers, the stack pointers and the status register. The line above the instruction shows the program label or "section + offset."

Notes and Exceptions. When any TRA selections are in effect, your program executes at much less than normal speed, even in those parts of the program that are not traced. If execution speed is important, but you want to step through part of a program, you can use the following TTA command:

```
     > EVE 1 A=loaddr hiaddr -C <CR>
```

This makes the 68000 emulator pause and print the register contents after executing each instruction in the specified range. Instructions outside the range will be executed at full speed if TRA is OFF.

When TRA ALL is set and execution of the user program comes within 10 bytes of a non-allocated or NOMEMed section of memory, instruction disassembly is no longer performed. This occurs because the disassembler tries to read enough words to disassemble the longest possible instruction. Instruction execution proceeds normally and breaks if the memory boundary is reached.

The emulator does not use the 68000 status register trace bit (T). The user has full use of this bit.

If your program contains only absolute sections, the trace display will not show the "section + offset" line, even if SYMD -SL is ON. However, labels are still displayed.

WRT---Writes to Emulator I/O Port

The WRT command allows you to write to a memory location as an I/O port. It has the same function as the P command, except that WRT always writes to prototype memory, and does not do a read-back check. A word write to an odd boundary causes an error.

REAL-TIME PROTOTYPE ANALYZER

You may not use the Real-Time Prototype Analyzer (RPTA) with the 68000 emulator. An error message is issued if you attempt to use the RTPA while you are using the 68000 emulator.

TRIGGER TRACE ANALYZER (TTA) COMMANDS AND PARAMETERS

The Trigger Trace Analyzer provides real-time tracing and break conditions for the 68000 emulator running at up to 8MHz.

Memory space designators are allowed as part of the address expression, but are ignored in the actual programming of the TTA. To reference a particular memory space, you must use the BUS command.

The Trigger Trace Analyzer Users Manual describes the TTA and its commands.

EVE and BUS Command Parameters

Table 7L-3 shows the bus signal symbols which may be used as parameters for the BUS command, and for the B parameter of the EVE command. The DISP command also uses these symbol in its display.

Table 7L-3
EVE/BUS Signal Symbols

Symbol	Description
UP (*a)	User Program
UD (*a)	User Data
SP (*a)	Supervisor Program
SD (*a)	Supervisor Data
U (*a)	any operation occurring in User mode
S (*a)	any operation occurring in Supervisor mode
P (*a)	any operation occurring in Program memory
D (*a)	any Data operation
INTA	INTerrupt Acknowledge
IPL0 - IPL7	InterruPt Levels
BY	BYte operation
WD	WorD operation
RD	ReaD operation
WT	WriTe operation
VPA	Valid Peripheral Address
BERR	Bus ERRor
HLT	Halt

(*a)---The DISP command displays these symbols as part of the address field instead of the bus signal field.

CONS---Set Consecutive Events

The EMU and FET parameters are not supported.

DISP---Display Contents of Acquisition Memory

The DISP command displays the contents of the TTA Acquisition Memory. This memory acquires a record of bus activity that occurred while your program was running.

Because the 68000 has a prefetch pipeline, and no fetch signal as discussed under "Special Considerations", the DISP command attempts to disassemble every word as an instruction, unless it is obviously not code. This ensures that every word that really is an instruction is disassembled. However, it also generates superfluous disassembly lines, which should be ignored.

The DISP command also displays the signal symbols shown in Table 7L-3 as part of its ADDRESS and BUS fields. When an INTA (interrupt acknowledge) cycle occurs, the notation **: will be displayed in place of an memory space designator, since the function code lines (FCx) do not show a valid memory space.

Figure 7L-3 shows a sample display of the following program lines when the program was run with TRA OFF:

```
CLR.L   D0
MOVE.W  #1H,D1
MOVE.L  #1000H,A0
```

```
> DISP <CR>

ADDRESS    DATA                                 7-PROBE-0   BUS

PROG+000000
SP:003000  4280   CLR.L    D0                   0000 0000   WD IPLO RD

PROG+000002
SP:003002  323C   MOVE.W   #1H,D1               0000 0000   WD IPLO RD

PROG+000004
SP:003004  0001   OR.B     #7CH,D1              0000 0000   WD IPLO RD

PROG+000006
SP:003006  207C   MOVE.L   #1000H,A0            0000 0000   WD IPLO RD

PROG+000008
SP:003008  0000   OR.B     #0H,D0               0000 0000   WD IPLO RD

PROG+00000A
SP:00300A  1000   MOVE.B   D0,D0                0000 0000   WD IPLO RD
```

Fig. 7L-3. Sample DISP display.

Note that the third, fifth, and sixth lines are superfluous. They result from disassembly on the operands of the other instructions.

TS---Display Status of TTA Triggers

In a TS display, the "bus" signals may not be identical to the parameters you enter with a BUS command or the B parameter of the EVE command. However, the signals displayed are functionally equivalent to the parameters you specified. The format of the display is as described in the 8500 Trigger Trace Analyzer Users Manual.

When an INTA (interrupt acknowledge) cycle occurs, the notation **: will be displayed in place of a memory space designator, since the function code lines (FCx) do not show a valid memory space.

Supervisor mode (S) will be indicated any time interrupt acknowledge (INTA) is shown.

SERVICE CALLS

Service calls (SVCs) allow your program to use many system capabilities of your 8540, 8550, or 8560. The 68000 emulator supports service calls in all three modes.

An SVC is invoked with any byte instruction which writes to the address range specified by the SVC command. The operand of the instruction directs the system to a specified memory address called the SRB pointer. (The pointer points to the SRB, the Service Request Block.) The SRB pointer tells the system where to find the data (stored in the SRB) that informs the system which service to perform. The SRB pointer and the SRB may be located in any of the four memory spaces.

Table 7L-4 shows the default addresses for the eight SRB pointers. You can use the SVC command to alter these addresses and their associated port values to suit your program requirements.

The memory space of the SRB vector can be specified with a memory space designator in the address parameter of the SVC command. The default memory space is the value of MEMSP S. If you do not use the SVC command to specify the SRB vector, the vector defaults to supervisor data space (SD).

Refer to the Command Dictionary of your System Users Manual for syntax and use of the SVC command.

SVC Address Range

The 68000 uses memory-mapped I/O. In order for a byte-write instruction to invoke a Service Call, its address operand (the "port") must be in the proper SVC address range. This range may be anywhere in memory; however, it is recommended that the range used be above address 400H. The default address range is F00000--F00007. You can change this range with the SVC command.

Multiple memory spaces are allowed for the "port" parameter in the SVC command. If memory space designators are omitted, the value of MEMSP M at the time the SVC command is executed is used. In the SVC command, the least significant digit of the "port" range will be set to 0 (i.e., 0FFF7 will be rounded to 0FFF0).

Example. The following command changes the SVC address range to 1000--1007 and causes the SRB vector to start at location F00 in Supervisor Data space.

```
> SVC,,SD:F00 1000 <CR>
```

Then, to invoke SVC1, include the following instructions in your program:

```
MOVE.B D0,1007H
NOP
NOP
```

Table 7L-4
68000 Service Calls

SVC Number	Service Call (*a)		Default Location of SRB Pointer
	Mnemonic (*b) (*c)	Hexadecimal	
1	MOVE.B D0,(GEN.L)F00007H	13C000F00007	C0,C1,C2,C3
	NOP	4E71	
	NOP	4E71	
2	MOVE.B D0,(GEN.L)F00006H	13C000F00006	C4,C5,C6,C7
	NOP	4E71	
	NOP	4E71	
3	MOVE.B D0,(GEN.L)F00005H	13C000F00005	C8,C9,CA,CB
	NOP	4E71	
	NOP	4E71	
4	MOVE.B D0,(GEN.L)F00004H	13C000F00005	CC,CD,CE,CF
	NOP	4E71	
	NOP	4E71	
5	MOVE.B D0,(GEN.L)F00003H	13C000F00004	D0,D1,D2,D3
	NOP	4E71	
	NOP	4E71	
6	MOVE.B D0,(GEN.L)F00002H	13C000F00002	D4,D5,D6,D7
	NOP	4E71	
	NOP	4E71	
7	MOVE.B D0,(GEN.L)F00001H	13C000F00001	D8,D9,DA,DB
	NOP	4E71	
	NOP	4E71	
8	MOVE.B D0,(GEN.L)F00000H	13C000F00000	DC,DD,DE,DF
	NOP	4E71	
	NOP	4E71	

(*a) The default SVC address range (F00000--F00007) is assumed.
(*b) The MOVE.B instruction is used in this table. However, any byte-write instruction can be used to invoke an SVC.
(*c) The 68000-specific assembler directive, GEN.L, generates a long word address.

NOTE

Include two NOP instructions immediately following the byte-write instruction. The NOPs fill the 68000 prefetch pipeline so that other instructions following the SVC will not be lost.

When SVCs are enabled, the addresses used by the SVCs should not be used in any write instruction except to invoke an SVC. A read instruction will not invoke an SVC.

SRB Format

The 68000 emulator uses the LAS (Large Address Space) format for SRBs and SRB pointers. This format is described in the Service Calls section of your System Users Manual. Fig. 7L-4 illustrates the format of a 68000 SRB pointer for SVC1.

```
        C0         C1         C2         C3
        +----------+----------+----------+----------+
        |          |                                |
        | Memory   |                                |
        | Space    |          24-Bit Address        |
        | Byte     |                                |
        |          |                                |
        +----------+----------+----------+----------+
```

Fig. 7L-4. A 68000 SRB pointer located at C0--C3.

Table 7L-5 list the value of the Memory Space byte that corresponds to each memory space. For more information on memory spaces, refer to the heading "Memory Spaces" under "Special Considerations" later in this section.

Table 7L-5
Encoding of the Memory Space Byte

Memory Space	Code
current default	0000
UD:	0001
UP:	0010
SD:	0100
SP:	1000

SVC Demonstration

Figure 7L-5 lists a 68000 program that uses four SVC functions: Assign Channel, Read ASCII, Write ASCII, and Abort. The program's algorithm is explained in the Service Calls section of your System Users Manual, which also demonstrates a version of the program written in 8080A/8085A assembly language. Using the program in Fig. 7L-5, you can perform a parallel demonstration with the 68000 B Series Assembler and 68000 emulator.

```
; SSSSS V   V CCCCC
; S      V   V C
; SSSSS  V V  C        DEMONSTRATION:   68000 EMULATOR
;     S  V V  C
; SSSSS   V   CCCCC
;
        ORG     0C0H     ;Beginning of SRB vector.
        LONG    SRB1FN   ;SRB1 LAS specification.
        LONG    SRB2FN   ;SRB2 LAS specification.
        LONG    SRB3FN   ;SRB3 LAS specification.
        LONG    SRB4FN   ;SRB4 LAS specification.
        LONG    SRB5FN   ;SRB5 LAS specification.
                         ;End of SRB vector.
;
        ORG     400H     ;Set up SRB areas.
;
                         ;SRB1 = Assign 'CONI' to Channel 0.
SRB1FN  BYTE    10H      ;Assign
        BYTE    00H      ;   to Channel 0.
SRB1ST  BLOCK   01H      ;Status returned here.
        BLOCK   01H      ;Reserved.
        BLOCK   02H      ;Reserved.
        WORD    05H      ;Length of 'CONI' + <CR>.
        LONG    CONI     ;LAS pointer to 'CONI' + <CR>.
                         ;End of SRB1.
;
                         ;SRB2 = Assign 'LPT' to Channel 1.
SRB2FN  BYTE    10H      ;Assign
        BYTE    01H      ;   to Channel 1.
SRB2ST  BLOCK   01H      ;Status reserved here.
        BLOCK   01H      ;Reserved.
        BLOCK   02H      ;Reserved.
        WORD    04H      ;Length of 'LPT' + <CR>.
        LONG    LPT      ;LAS pointer to 'LPT' + <CR>.
                         ;End of SRB2.
;
                         ;SRB3 = Read ASCII line from CONI (Channel 0).
SRB3FN  BYTE    01H      ;Read ASCII
        BYTE    00H      ;   from Channel 0.
SRB3ST  BLOCK   01H      ;Status returned here.
        BLOCK   01H      ;Reserved.
        BLOCK   02H      ;Byte count returned here.
        WORD    100H     ;256 bytes in our buffer.
        LONG    BUFFER   ;LAS pointer to our buffer.
                         ;End of SRB3.
;
                         ;SRB4 = Write ASCII line to LPT (Channel 1).
SRB4FN  BYTE    02H      ;Write ASCII
        BYTE    01H      ;   to Channel 1.
SRB4ST  BLOCK   01H      ;Status returned here.
        BLOCK   01H      ;Reserved.
        BLOCK   02H      ;Byte count returned here.
        WORD    100H     ;256 bytes in our buffer.
        LONG    BUFFER   ;LAS pointer to our buffer.
                         ;End of SRB4.
```

Fig. 7L-5. 68000 SVC demonstration program listing (part 1 of 2).

```
;
                                ;SRB5 = Abort (Close all channels and terminate).
SRB5FN  BYTE     1FH            ;Abort.
        BLOCK    0BH            ;Reserved.
                                ;End of SRB5.
;
BUFFER  BLOCK    100H           ;Our I/O area.
CONI    ASCII    'CONI'         ;ASCII of 'CONI'
        BYTE     0DH            ;   + <CR>.
LPT     ASCII    'LPT'          ;ASCII of 'LPT'
        BYTE     0DH            ;   + <CR>.
                                ;End of data definitions.
;
;       Beginning of executable code.
;
START   ORG      1000H                    ;Entry point into program.
        MOVE.B   D0,(GEN.L)0F00007H       ;Call SVC1 to
        NOP                               ;   assign 'CONI'
        NOP                               ;   to Channel 0.
        TST.B    SRB1ST                   ;Check status to see if all went well.
        BNE      ABORT                    ;No?  Stop everything.
        MOVE.B   D0,(GEN.L)0F00006H       ;Call SVC2 to
        NOP                               ;   assign 'LPT'
        NOP                               ;   to Channel 1.
        TST.B    SRB2ST                   ;Check status to see if all went well.
        BNE      ABORT                    ;No?  Stop everything.
LOOP    MOVE.B   D0,(GEN.L)0F00005H       ;Call SVC3 to read
        NOP                               ;   a line from 'CONI'
        NOP                               ;   into the buffer.
        TST.B    SRB3ST                   ;Check status to see if all went well.
        BNE      ABORT                    ;No?  Stop everything.
        MOVE.B   D0,(GEN.L)0F00004H       ;Call SVC4 to write
        NOP                               ;   a line to 'LPT'
        NOP                               ;   from the buffer.
        TST.B    SRB4ST                   ;Check status to see if all went well.
        BEQ      LOOP                     ;Yes?  Go back to read another line.
                                          ;No?  Fall through to termination.
ABORT   MOVE.B   D0,(GEN.L)0F00003H       ;Call SVC5
        NOP                               ;   to do the abort.
        NOP
        END      START                    ;End of the program.
```

Fig. 7L-5. 68000 SVC demonstration program listing (part 2 of 2).

This program shows the use of four service calls. The program's algorithm is explained in the Service Calls section of your System Users Manual. The program accepts a line of ASCII characters from the system terminal. Then, when it receives a RETURN character, the program writes the line to the line printer and accepts another line. (On the 8550, output to the line printer is buffered. No text is printed until the 8550's line printer buffer is full or the program ends.) To terminate the program, enter a CTRL-Z while the program is waiting for input.

Figure 7L-6 shows another way to code the executable portion of the program. By inserting the statement MOVE.L #0F00000,A1 at the beginning of the code, and the block of EQU statements at the end, you can save object code space and reference the SVCs symbolically.

```
;
;       Beginning of executable code.
;
START   ORG     1000H               ;Entry point into program.
        MOVE.L  #0F00000,A1         ;Set a register to the SVC
                                    ;   trigger location.
        MOVE.B  D0,SVC1(A1)         ;Call SVC1 to
        NOP                         ;   assign 'CONI'
        NOP                         ;   to Channel 0.
        TST.B   SRB1ST              ;Check status to see if all went well.
        BNE     ABORT               ;No?  Stop everything.
        MOVE.B  D0,SVC2(A1)         ;Call SVC2 to
        NOP                         ;   assign 'LPT'
        NOP                         ;   to Channel 1.
        TST.B   SRB2ST              ;Check status to see if all went well.
        BNE     ABORT               ;No?  Stop everything.
LOOP    MOVE.B  D0,SVC3(A1)         ;Call SVC3 to read
        NOP                         ;   a line from 'CONI'
        NOP                         ;   into the buffer.
        TST.B   SRB3ST              ;Check status to see if all went well.
        BNE     ABORT               ;No?  Stop everything.
        MOVE.B  D0,SVC4(A1)         ;Call SVC4 to write
        NOP                         ;   a line to 'LPT'
        NOP                         ;   from the buffer.
        TST.B   SRB4ST              ;Check status to see if all went well.
        BEQ     LOOP                ;Yes?  Go back to read another line.
                                    ;No?  Fall through to termination.
ABORT   MOVE.B  D0,SVC5(A1)         ;Call SVC5
        NOP                         ;   to do the abort.
        NOP
                                    ;Define SVC symbols.
SVC1    EQU     7
SVC2    EQU     6
SVC3    EQU     5
SVC4    EQU     4
SVC5    EQU     3
;
        END     START               ;End of the program.
```

Fig. 7L-6. Alternate executable code for SVC demonstration program.

SPECIAL CONSIDERATIONS

Some of the characteristics of the 68000 microprocessor greatly affect the behavior and appearance of several commands; in particular, BK, TRA, and DISP. These characteristics are discussed in the following paragraphs, and in much greater detail in the reprints at the back of this section.

Fetching and the Prefetch Pipeline

The 68000 microprocessor has a prefetch pipeline which speeds up the instruction fetch-decode-execute process. The processor does not, however, have a fetch signal available. These two factors combine to cause differences in many of the emulator's displays.

Figure 7L-7 shows a simplified diagram of the pipeline.

```
   --------------------
   |                  |
   |   Instruction    |
   |    Register      |
   |                  |
   ---------+----------
            |
            |
            v
 ------------------------          ------------------------
 |                      |          |                      |
 |                      | Address  |                      |
 |     Instruction      +---------->|    Micro and Nano    |
 |                      |          |                      |
 |       Decode         |<---------+    Control Stores    |
 |                      |  Branch  |                      |
 |                      |  Select  |                      |
 --+--+------------------          ----+-+-----------------
   |  |  ALU         ^                 | |
   |  |  Function    |                 | |  Timing & Switch
   |  |  & Register  | Conditionals    | |  Control Signals
   |  |  Selection   |                 \ /
   v  v              |                  v
 --------------------+-------------------------------------
 |                                                        |
 |                                                        |
 |                     Execution Unit                     |
 |                                                        |
 |                                                        |
 ----------------------------------------------------------
```

Fig. 7L-7. 68000 instruction pipeline block diagram.

The pipeline consists of an Instruction Register, an Instruction Decoder, and an Execution Unit. The Instruction Register holds the most recently fetched instruction word. The Instruction Decoder, using the Micro and Nano Control Stores, decodes the instruction. When the instruction reaches the Execution Unit, it is executed.

It is the responsibility of the instruction that is executing to ensure two things before it finishes its execution:

1. that the next instruction word is accessed with sufficient time for complete decoding by the end of the current instruction; and

2. that the instruction word following the next instruction is fetched by the end of the current instruction

Since there is no instruction fetch signal, the only time the emulator knows that an instruction has been fetched into the instruction pipeline is when it sees the appropriate values on the bus. The emulator doesn't know whether the instruction is actually executing from this information.

Interrupts

The 68000 has seven interrupt levels. Higher-numbered interrupts have higher priority. The level 7 interrupt is a non-maskable interrupt (NMI).

Interrupts are controlled by the three interrupt mask bits (I2--I0) in the Status Register.

When an interrupt request is made, the 68000 compares the level of the interrupt with the interrupt mask. If the new interrupt has a level higher than the mask setting, the interrupt is recognized. When this occurs, the 68000 has to stack its Program Counter and Status Register so that it can return to the interrupted task after the interrupt has been processed.

This stacking is done in the following time sequence: First, PC(L) is stacked, then the interrupt is acknowledged. Next, the status register is stacked, and last, PC(H). Because PC(L) is stacked before the emulator knows that an interrupt has occurred, this Supervisor Data write always appears on a DISP display of your program run.

Program example. The program example in Fig. 7L-8 is used to illustrate the effects of the 68000 characteristics previously described.

The program was loaded beginning at address 3000H. After moving values into three registers, the program enters the LOOP and executes it twice. The two NOPs are then executed, followed by the branch to SELF. Execution of this instruction would continue indefinitely, but breakpoint 1 was set at SELF before the program was executed.

```
        CLR.L    D0
        MOVE.W   #1H,D1
        MOVE.L   #1000H,A0
        MOVE.L   #2000H,A1
LOOP    MOVE.W   A,B
        MOVE.L   (A0)+,(A1)+
        DBF      D1,LOOP
        NOP
        NOP
SELF    BT       SELF
A       BLOCK    02H
B       BLOCK    02H
```

Fig. 7L-8. Program example.

Figures 7L-9 and 7L-10 use the DISP command to show the contents of the TTA acquisition memory for this program run with TRA OFF and TRA ON, respectively.

Refer to the following comments while examining these two figures:

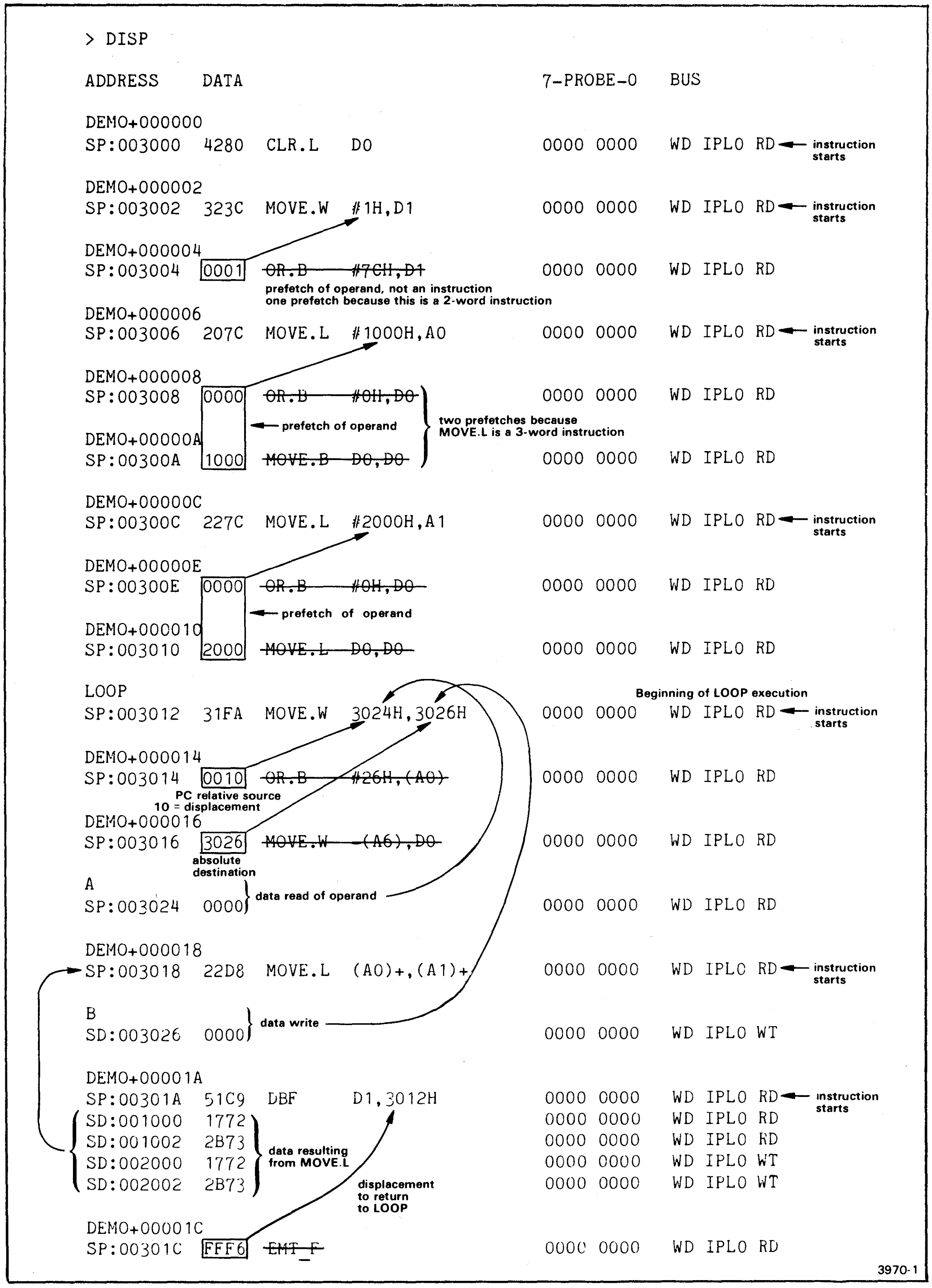
Each of the superfluous disassembly lines is crossed out. These lines resulted from disassembly of the operand(s) in the preceding instruction. Boxes are drawn around the operands in the object code field, and arrows show where they came from in the preceding instruction.

Data reads and data writes from A, B, and the MOVE.L (A0)+,(A1)+ instruction are interleaved with fetches from the instruction stream.

The effects of prefetching appear in the figures. Notes in each figure indicate where the instructions start, and which ones were actually executed.

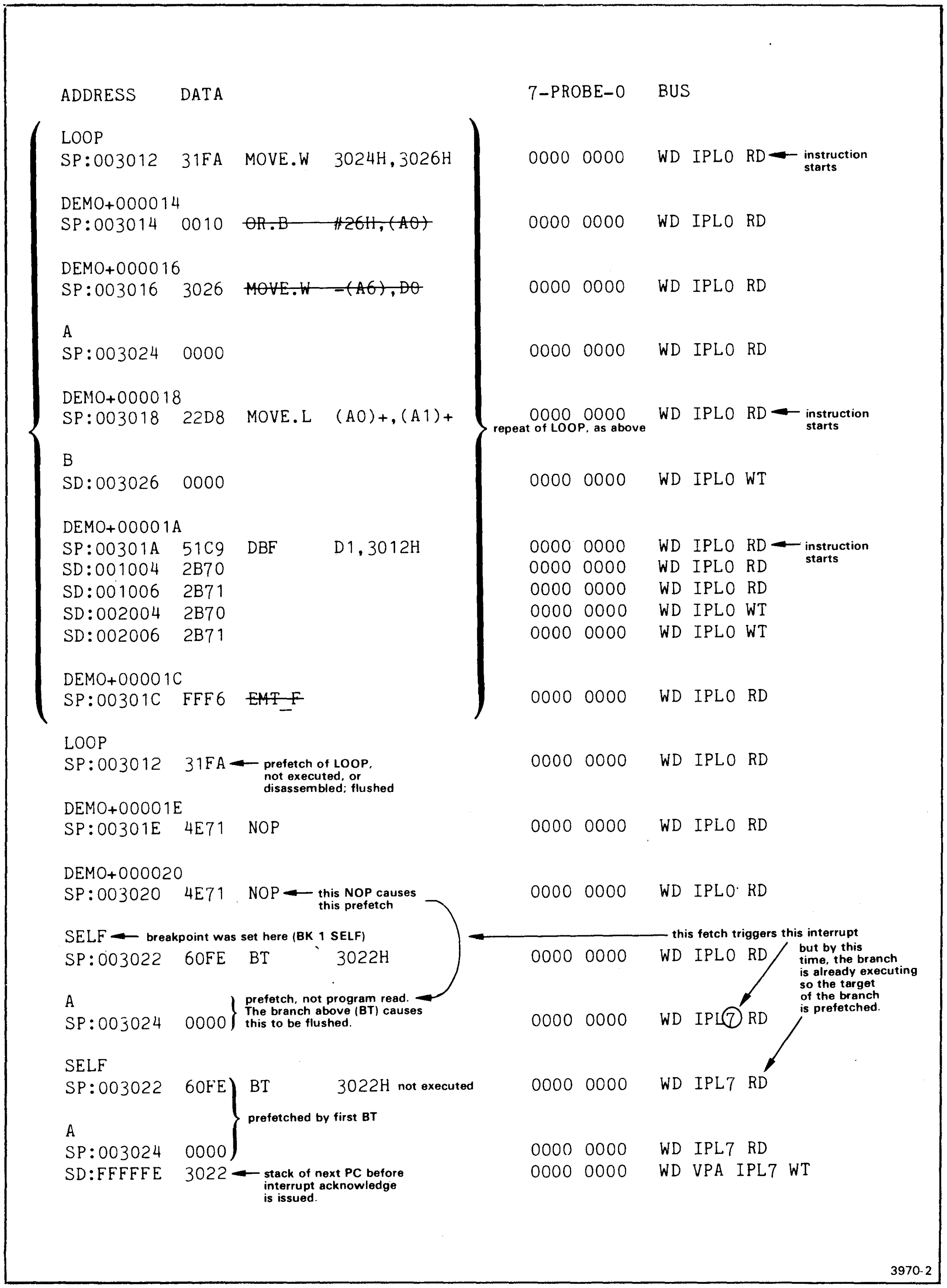
At the end of both figures, you can see the next PC(L) stacked in SD space before the interrupt acknowledge to the break interrupt occurred. In Figure 7L-10, where TRA is ON, the display shows that PC(L) was also stacked each time an interrupt to print a trace line occurred.

The run with TRA ON terminated two instructions before the run with TRA OFF, because of the decrease in execution speed with TRA ON. When TRA was OFF, the two additional instructions reached the Execution Unit before the break interrupt occurred. Note that with TRA ON, the instruction on which the breakpoint was set was not executed.



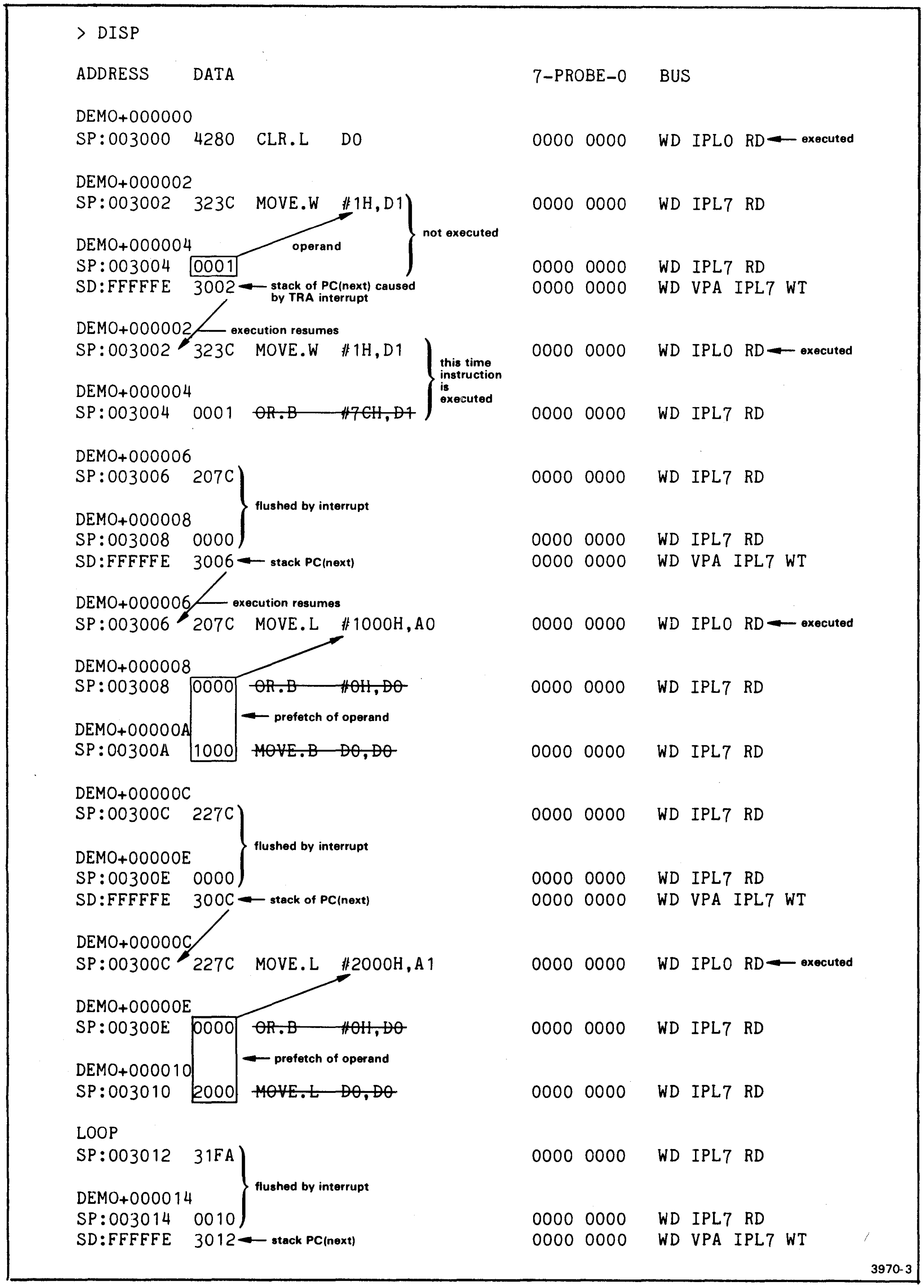
```
> DISP

ADDRESS     DATA                                    7-PROBE-0   BUS

DEMO+000000
SP:003000   4280   CLR.L    D0                      0000 0000   WD IPLO RD

DEMO+000002
SP:003002   323C   MOVE.W   #1H,D1                  0000 0000   WD IPLO RD

DEMO+000004
SP:003004   0001   ~~OR.B     #7CH,D1~~             0000 0000   WD IPLO RD

DEMO+000006
SP:003006   207C   MOVE.L   #1000H,A0               0000 0000   WD IPLO RD

DEMO+000008
SP:003008   0000   ~~OR.B     #0H,D0~~              0000 0000   WD IPLO RD

DEMO+00000A
SP:00300A   1000   ~~MOVE.B   D0,D0~~               0000 0000   WD IPLO RD

DEMO+00000C
SP:00300C   227C   MOVE.L   #2000H,A1               0000 0000   WD IPLO RD

DEMO+00000E
SP:00300E   0000   ~~OR.B     #0H,D0~~              0000 0000   WD IPLO RD

DEMO+000010
SP:003010   2000   ~~MOVE.L   D0,D0~~               0000 0000   WD IPLO RD

LOOP
SP:003012   31FA   MOVE.W   3024H,3026H             0000 0000   WD IPLO RD

DEMO+000014
SP:003014   0010   ~~OR.B     #26H,(A0)~~           0000 0000   WD IPLO RD

DEMO+000016
SP:003016   3026   ~~MOVE.W   -(A6),D0~~            0000 0000   WD IPLO RD

A
SP:003024   0000                                    0000 0000   WD IPLO RD

DEMO+000018
SP:003018   22D8   MOVE.L   (A0)+,(A1)+             0000 0000   WD IPLO RD

B
SD:003026   0000                                    0000 0000   WD IPLO WT

DEMO+00001A
SP:00301A   51C9   DBF      D1,3012H                0000 0000   WD IPLO RD
SD:001000   1772                                    0000 0000   WD IPLO RD
SD:001002   2B73                                    0000 0000   WD IPLO RD
SD:002000   1772                                    0000 0000   WD IPLO WT
SD:002002   2B73                                    0000 0000   WD IPLO WT

DEMO+00001C
SP:00301C   FFF6   ~~EMT F~~                        0000 0000   WD IPLO RD
```

Fig. 7L-9. Example program run with TRA OFF (part 1 of 2).



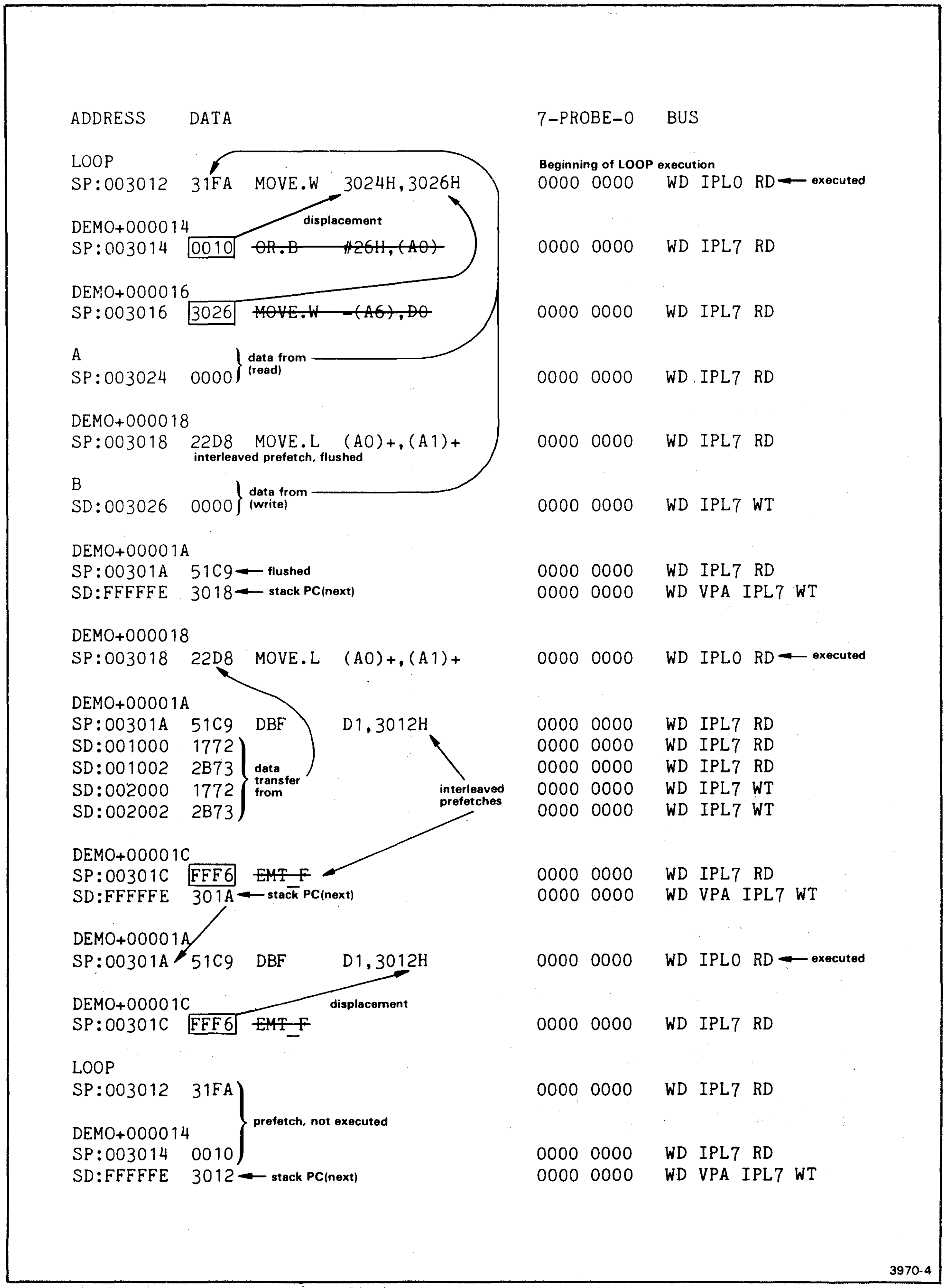
```
ADDRESS      DATA                                   7-PROBE-0     BUS

LOOP
SP:003012    31FA   MOVE.W    3024H,3026H           0000 0000     WD IPLO RD <-- instruction
                                                                                 starts
DEMO+000014
SP:003014    0010   ~~OR.B      #26H,(A0)~~             0000 0000     WD IPLO RD

DEMO+000016
SP:003016    3026   ~~MOVE.W    -(A6),D0~~              0000 0000     WD IPLO RD

A
SP:003024    0000                                   0000 0000     WD IPLO RD

DEMO+000018
SP:003018    22D8   MOVE.L    (A0)+,(A1)+           0000 0000     WD IPLO RD <-- instruction
                                          repeat of LOOP, as above               starts
B
SD:003026    0000                                   0000 0000     WD IPLO WT

DEMO+00001A
SP:00301A    51C9   DBF       D1,3012H              0000 0000     WD IPLO RD <-- instruction
SD:001004    2B70                                   0000 0000     WD IPLO RD     starts
SD:001006    2B71                                   0000 0000     WD IPLO RD
SD:002004    2B70                                   0000 0000     WD IPLO WT
SD:002006    2B71                                   0000 0000     WD IPLO WT

DEMO+00001C
SP:00301C    FFF6   ~~EMT F~~                           0000 0000     WD IPLO RD

LOOP
SP:003012    31FA <-- prefetch of LOOP,             0000 0000     WD IPLO RD
                      not executed, or
                      disassembled; flushed
DEMO+00001E
SP:00301E    4E71   NOP                             0000 0000     WD IPLO RD

DEMO+000020
SP:003020    4E71   NOP <-- this NOP causes         0000 0000     WD IPLO RD
                            this prefetch

SELF <-- breakpoint was set here (BK 1 SELF)   <---------- this fetch triggers this interrupt
SP:003022    60FE   BT        3022H                 0000 0000     WD IPLO RD   but by this
                                                                               time, the branch
A                   } prefetch, not program read.                              is already executing
SP:003024    0000   } The branch above (BT) causes  0000 0000     WD IPL(7) RD so the target
                    } this to be flushed.                                      of the branch
                                                                               is prefetched.
SELF
SP:003022    60FE } BT        3022H not executed    0000 0000     WD IPL7 RD
                  } prefetched by first BT
A                 }
SP:003024    0000 }                                 0000 0000     WD IPL7 RD
SD:FFFFFE    3022 <-- stack of next PC before       0000 0000     WD VPA IPL7 WT
                      interrupt acknowledge
                      is issued.
```

Fig. 7L-9. Example program run with TRA OFF (part 2 of 2).



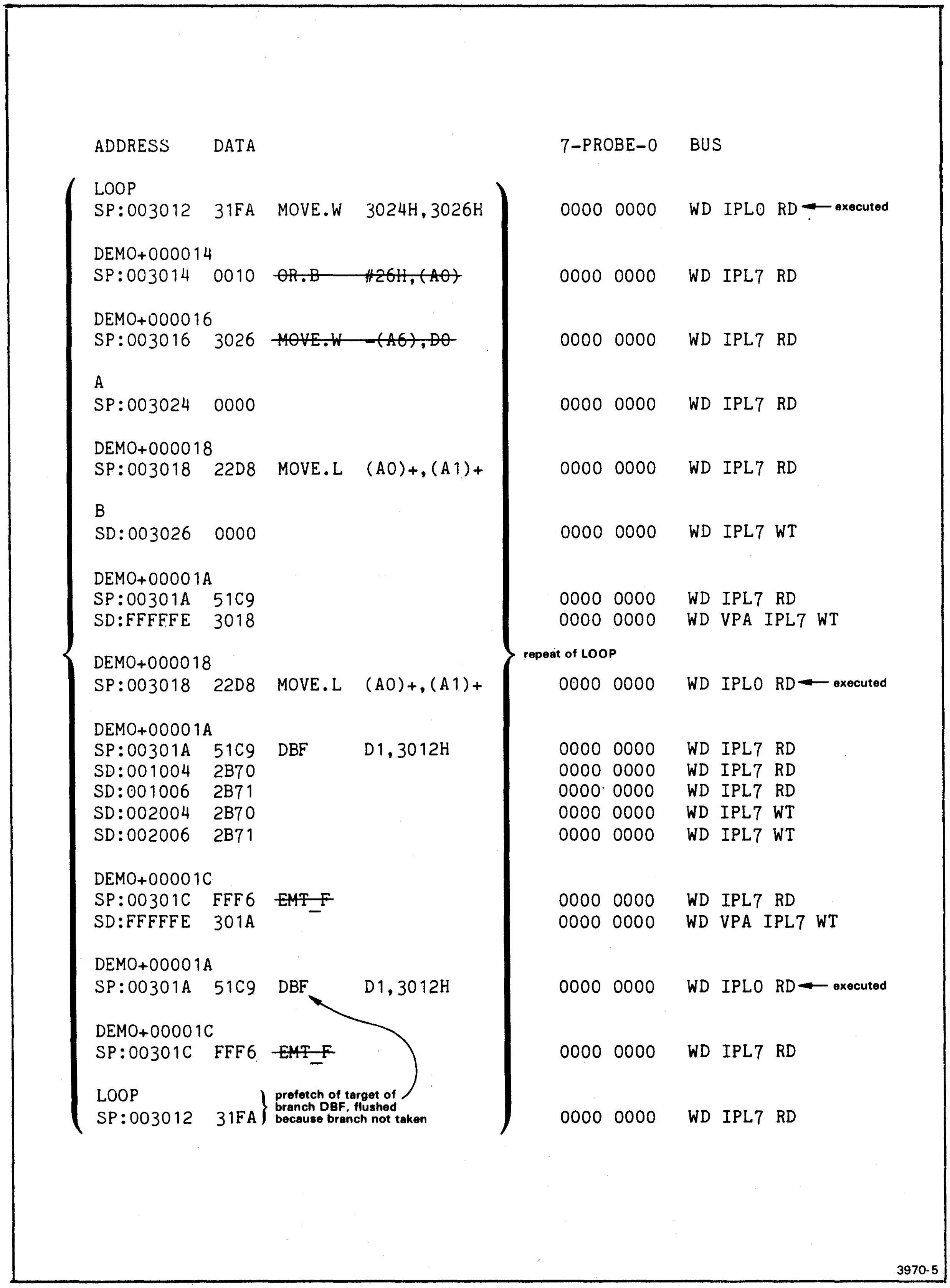
```
> DISP

ADDRESS     DATA                               7-PROBE-0    BUS

DEMO+000000
SP:003000   4280  CLR.L    D0                  0000 0000    WD IPL0 RD <— executed

DEMO+000002
SP:003002   323C  MOVE.W   #1H,D1  \           0000 0000    WD IPL7 RD
                                    } not executed
DEMO+000004              operand   /
SP:003004   0001                               0000 0000    WD IPL7 RD
SD:FFFFFE   3002 <— stack of PC(next) caused   0000 0000    WD VPA IPL7 WT
                    by TRA interrupt

DEMO+000002 <— execution resumes
SP:003002   323C  MOVE.W   #1H,D1  \           0000 0000    WD IPL0 RD <— executed
                                    } this time
DEMO+000004                         } instruction
                                    } is
SP:003004   0001  ~~OR.B     #7CH,D1~~ / executed  0000 0000    WD IPL7 RD

DEMO+000006
SP:003006   207C \                             0000 0000    WD IPL7 RD
                  } flushed by interrupt
DEMO+000008       }
SP:003008   0000 /                             0000 0000    WD IPL7 RD
SD:FFFFFE   3006 <— stack PC(next)             0000 0000    WD VPA IPL7 WT

DEMO+000006 <— execution resumes
SP:003006   207C  MOVE.L   #1000H,A0           0000 0000    WD IPL0 RD <— executed

DEMO+000008
SP:003008   0000  ~~OR.B     #0H,D0~~            0000 0000    WD IPL7 RD
                 <— prefetch of operand
DEMO+00000A
SP:00300A   1000  ~~MOVE.B   D0,D0~~             0000 0000    WD IPL7 RD

DEMO+00000C
SP:00300C   227C \                             0000 0000    WD IPL7 RD
                  } flushed by interrupt
DEMO+00000E       }
SP:00300E   0000 /                             0000 0000    WD IPL7 RD
SD:FFFFFE   300C <— stack of PC(next)          0000 0000    WD VPA IPL7 WT

DEMO+00000C
SP:00300C   227C  MOVE.L   #2000H,A1           0000 0000    WD IPL0 RD <— executed

DEMO+00000E
SP:00300E   0000  ~~OR.B     #0H,D0~~            0000 0000    WD IPL7 RD
                 <— prefetch of operand
DEMO+000010
SP:003010   2000  ~~MOVE.L   D0,D0~~             0000 0000    WD IPL7 RD

LOOP
SP:003012   31FA \                             0000 0000    WD IPL7 RD
                  } flushed by interrupt
DEMO+000014       }
SP:003014   0010 /                             0000 0000    WD IPL7 RD
SD:FFFFFE   3012 <— stack PC(next)             0000 0000    WD VPA IPL7 WT
```

Fig. 7L-10. Example program run with TRA ON (part 1 of 4).



```
ADDRESS      DATA                                  7-PROBE-0   BUS

LOOP
SP:003012   31FA   MOVE.W   3024H,3026H            0000 0000   WD IPL0 RD

DEMO+000014
SP:003014   0010   ~~OR.B      #26H,(A0)~~         0000 0000   WD IPL7 RD

DEMO+000016
SP:003016   3026   ~~MOVE.W    -(A6),D0~~          0000 0000   WD IPL7 RD

A
SP:003024   0000                                   0000 0000   WD IPL7 RD

DEMO+000018
SP:003018   22D8   MOVE.L   (A0)+,(A1)+            0000 0000   WD IPL7 RD

B
SD:003026   0000                                   0000 0000   WD IPL7 WT

DEMO+00001A
SP:00301A   51C9                                   0000 0000   WD IPL7 RD
SD:FFFFFE   3018                                   0000 0000   WD VPA IPL7 WT

DEMO+000018
SP:003018   22D8   MOVE.L   (A0)+,(A1)+            0000 0000   WD IPL0 RD

DEMO+00001A
SP:00301A   51C9   DBF      D1,3012H               0000 0000   WD IPL7 RD
SD:001000   1772                                   0000 0000   WD IPL7 RD
SD:001002   2B73                                   0000 0000   WD IPL7 RD
SD:002000   1772                                   0000 0000   WD IPL7 WT
SD:002002   2B73                                   0000 0000   WD IPL7 WT

DEMO+00001C
SP:00301C   FFF6   ~~EMT F~~                       0000 0000   WD IPL7 RD
SD:FFFFFE   301A                                   0000 0000   WD VPA IPL7 WT

DEMO+00001A
SP:00301A   51C9   DBF      D1,3012H               0000 0000   WD IPL0 RD

DEMO+00001C
SP:00301C   FFF6   ~~EMT F~~                       0000 0000   WD IPL7 RD

LOOP
SP:003012   31FA                                   0000 0000   WD IPL7 RD

DEMO+000014
SP:003014   0010                                   0000 0000   WD IPL7 RD
SD:FFFFFE   3012                                   0000 0000   WD VPA IPL7 WT
```

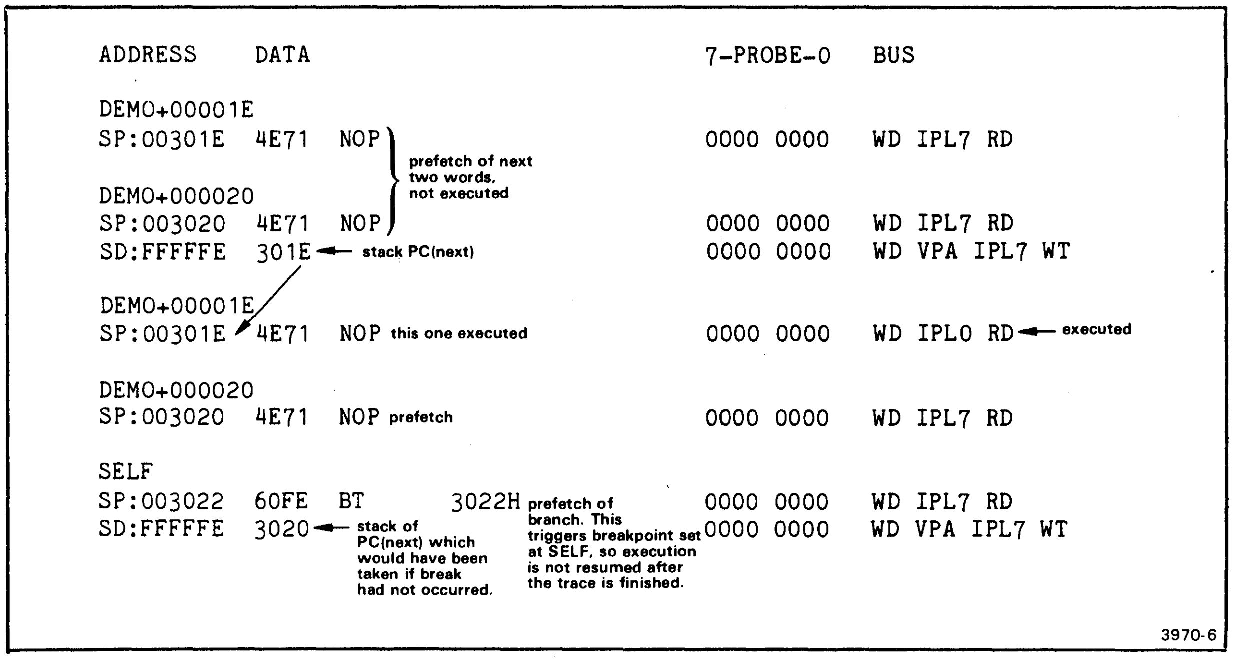
Fig. 7L-10. Example program run with TRA ON (part 2 of 4).



```
ADDRESS     DATA                                   7-PROBE-0   BUS

LOOP
SP:003012   31FA   MOVE.W    3024H,3026H           0000 0000   WD IPL0 RD ← executed

DEMO+000014
SP:003014   0010   ~~OR.B      #26H,(A0)~~             0000 0000   WD IPL7 RD

DEMO+000016
SP:003016   3026   ~~MOVE.W    -(A6),D0~~              0000 0000   WD IPL7 RD

A
SP:003024   0000                                   0000 0000   WD IPL7 RD

DEMO+000018
SP:003018   22D8   MOVE.L    (A0)+,(A1)+           0000 0000   WD IPL7 RD

B
SD:003026   0000                                   0000 0000   WD IPL7 WT

DEMO+00001A
SP:00301A   51C9                                   0000 0000   WD IPL7 RD
SD:FFFFFE   3018                                   0000 0000   WD VPA IPL7 WT
                                                 repeat of LOOP
DEMO+000018
SP:003018   22D8   MOVE.L    (A0)+,(A1)+           0000 0000   WD IPL0 RD ← executed

DEMO+00001A
SP:00301A   51C9   DBF       D1,3012H              0000 0000   WD IPL7 RD
SD:001004   2B70                                   0000 0000   WD IPL7 RD
SD:001006   2B71                                   0000 0000   WD IPL7 RD
SD:002004   2B70                                   0000 0000   WD IPL7 WT
SD:002006   2B71                                   0000 0000   WD IPL7 WT

DEMO+00001C
SP:00301C   FFF6   ~~EMT F~~                           0000 0000   WD IPL7 RD
SD:FFFFFE   301A                                   0000 0000   WD VPA IPL7 WT

DEMO+00001A
SP:00301A   51C9   DBF       D1,3012H              0000 0000   WD IPL0 RD ← executed

DEMO+00001C
SP:00301C   FFF6   ~~EMT F~~                           0000 0000   WD IPL7 RD

LOOP                } prefetch of target of
SP:003012   31FA    } branch DBF, flushed              0000 0000   WD IPL7 RD
                    } because branch not taken
```

3970-5

Fig. 7L-10. Example program run with TRA ON (part 3 of 4).



```
ADDRESS      DATA                                    7-PROBE-0   BUS

DEMO+00001E
SP:00301E   4E71   NOP  prefetch of next             0000 0000   WD IPL7 RD
                        two words,
DEMO+000020             not executed
SP:003020   4E71   NOP                               0000 0000   WD IPL7 RD
SD:FFFFFE   301E <- stack PC(next)                   0000 0000   WD VPA IPL7 WT

DEMO+00001E
SP:00301E   4E71   NOP this one executed             0000 0000   WD IPL0 RD <- executed

DEMO+000020
SP:003020   4E71   NOP prefetch                      0000 0000   WD IPL7 RD

SELF
SP:003022   60FE   BT          3022H prefetch of     0000 0000   WD IPL7 RD
SD:FFFFFE   3020 <- stack of         branch. This    0000 0000   WD VPA IPL7 WT
                    PC(next) which   triggers breakpoint set
                    would have been  at SELF, so execution
                    taken if break   is not resumed after
                    had not occurred. the trace is finished.
```

3970-6

Fig. 7L-10. Example program run with TRA ON (part 4 of 4).

Memory Spaces

The 68000 supports four memory spaces: User Data (UD), User Program (UP), Supervisor Data (SD) and Supervisor Program (SP). You may partition memory, using the MAP command, so that certain address blocks are accessed only for a particular type of reference. The default is for all of memory to be accessible by all four types of memory spaces.

The processor determines the type of reference by examining the three Function Code pins (FC2--FC0). The reference made based on the encoding of these three pins is shown in Table 7L-6.

Table 7L-6
Classification of Memory Space References

Function Code			
FC2	FC1	FC0	Type of Reference
0	0	0	(Reserved)
0	0	1	User Data
0	1	0	User Program
0	1	1	(Reserved)
1	0	0	(Reserved)
1	0	1	Supervisor Data
1	1	0	Supervisor Program
1	1	1	Interrupt Acknowledge

Memory Space Partitioning

The 68000 emulator supports memory space partitioning. You must have the Memory Allocation Controller (MAC) option installed in your system in order to use memory partitioning in program memory. If your prototype supports memory partitioning, you do not need the MAC option to access memory partitions.

This discussion includes an example program that shows the kind of statements you need to include in your program to use partitioned memory spaces. The example program shows you how to define the 68000 interrupt vectors and interrupt handlers, how to start a User program, and how to return to the Supervisor from the User's routine.

After the program has been linked and loaded as described in this example, it uses the four memory spaces:

- Supervisor Program (SP) space contains the reset vector and interrupt handlers, starts the User job, and exits when the User job is finished.
- Supervisor Data (SD) space contains the interrupt vectors, the SRB used by the exit SVC in the Supervisor Program, and the Supervisor stack.
- User Program (UP) space contains a program that opens a channel, writes to it, and returns to Supervisor control.
- User Data (UD) space contain the SRBs and other data used by the SVCs in the User Program. It also contains the User stack.
- The example program ends with the SVC trigger definitions which are available to all memory spaces (non-partitioned).

The Example Program. A listing of the example program source, called MEMPAR.SRC, is shown in Figure 7L-11. There are many other ways to code routines that perform these tasks; but this example will help you understand what must be included. Comments within the code explain what the program is doing.

```
;--------------------------------------------------------------------------
;
;  This example program shows the use of memory partitioning.
;
;--------------------------------------------------------------------------
;
                LIST    DBG             ;Pass symbols to the linker
                GEN.L                   ;Default to long addresses
;
;--------------------------------------------------------------------------
;
; ------Define the vectors.  The locations of the reset and interrupt
;       vectors in memory are shown in Table 7L-7.
;
; ------Define the reset vector.
;       (The reset vector is in Supervisor Program space.)
;
                SECTION RESET.VEC
;
                ADDRESS STACK           ;Initial SSP
                ADDRESS START           ;Initial PC
;
; ------Define the interrupt vectors.
;       (The rest of the vector table is in Supervisor Data space.)
;
                SECTION INTERRUPT.VEC
;
                ORG     08H             ;This is where the Bus Error vector
V.BUS.ERR       ADDRESS BUS.ERR         ;   starts.
V.ADDRESS.ERR   ADDRESS ADDRESS.ERR
V.ILLEGAL.INS   ADDRESS ILLEGAL.INS
V.ZDIVIDE       ADDRESS ZDIVIDE
V.CHK.INS       ADDRESS CHK.INS
V.TRAPV.INS     ADDRESS TRAPV.INS
V.PRIVILEGE     ADDRESS PRIVILEGE
V.TRACE.BIT     ADDRESS TRACE.BIT
V.EMT.A         ADDRESS EMT.A
V.EMT.F         ADDRESS EMT.F
;
                BLOCK   50H             ;Reserve space for vectors between
                                        ;   V.EMT.F and V.TRAP.INS.
V.TRAP.INS      BLOCK   04H             ;Trap 0 set by Supervisor Program.
;
                BLOCK   3CH             ;Reserve space for other vectors.
;
V.SVC1          ADDRESS OPEN.SRB        ;Define the SRB vectors.
V.SVC2          ADDRESS PRINT.SRB
V.SVC3          ADDRESS EXIT.SRB
```

Fig. 7L-11. Memory partitioning example program (part 1 of 5).

```
;-----------------------------------------------------------------------------
;
; ------The following section will be linked to run
;       in Supervisor Program memory.
;
                SECTION SUPER.PROG
;
; ------Define the interrupt handlers.
;
BUS.ERR         EQU     $                   ;>  Normally,
ADDRESS.ERR     EQU     $                   ;>  these statements
ILLEGAL.INS     EQU     $                   ;>  would reference
ZDIVIDE         EQU     $                   ;>  the appropriate
CHK.INS         EQU     $                   ;>  interrupt handlers.
TRAPV.INS       EQU     $                   ;>  Since this is just
PRIVILEGE       EQU     $                   ;>  an example of
TRACE.BIT       EQU     $                   ;>  how they are set up,
EMT.A           EQU     $                   ;>  we just do the
EMT.F           EQU     $                   ;>  dummy NOP which follows.
                NOP                         ;Dummy routine for this example.
                RTE
;
```

Fig. 7L-11. Memory partitioning example program (part 2 of 5).

```
; ------Start of executable code.
;       (Note: SSP and Supervisor mode must have been set by Reset.)
;
START           EQU     $
;
; ------Prepare to start User job.
;
;       Since this program will be linked to run in partitioned
;       memory spaces, the addresses will have eight additional
;       high-order bits (two hex digits) appended:
;
;               UD: space starts at 01000000H
;               UP: space starts at 02000000H
;               SD: space starts at 04000000H
;               SP: space starts at 08000000H
;
;       The assembler will be able to strip off these extra bits when
;       the operand is an address field, but it does not know whether
;       an immediate value should be 24 or 32 bits.  So you must use
;       the BITS assembler directive to explicitly remove the extra bits.
;
                MOVE.L  #BITS(SUSPEND,0,24),V.TRAP.INS  ;Put return address
                                                        ;  in TRAP vector,
                                                        ;  taking low 24 bits.
                LEA     #USER.STACK,A1  ;Define User stack.
                                        ;Here, the assembler knows it's
                MOVE    A1,USP          ;  looking at an address.
;
                PEA     USER.JOB        ;Push User start address on stack.
                MOVE.W  #0700H,-(A7)    ;Push User SR on stack.
                RTE                     ;Start User job.
;
;
; ------ <<<< A TRAP 0 instruction by user causes a return to this point.>>>>
;
SUSPEND         EQU     $
                MOVE.B  D0,SVC3         ;Invoke the Exit SVC.
                NOP
                NOP
;
; ------End of Supervisor Program execution.
;
;----------------------------------------------------------------------------
;
; ------Supervisor Data memory space definition.
;
                SECTION SUPER.DATA
;
EXIT.SRB        EQU     $
                BYTE    1AH             ;1A is the Exit SVC function code.
;
                ORG     /2              ;Start the stack on an even address.
                BLOCK   400H            ;Reserve supervisor stack space.
STACK           EQU     $
;
;----------------------------------------------------------------------------
```

Fig. 7L-11. Memory partitioning example program (part 3 of 5).

```
;---------------------------------------------------------------------------
;
; ------User Program section.
;
                SECTION USER.PROG
;
USER.JOB        MOVE.B  D0,SVC1         ;Invoke SVC1 to open channel.
                NOP
                NOP
                MOVE.B  D0,SVC2         ;Invoke SVC2 to write to channel.
                NOP
                NOP
                TRAP    #0              ;Trap back to Supervisor state.
;
;--------------------------------------------------------------------------
;
; ------User Data section.
;
                SECTION USER.DATA
;
OPEN.SRB        EQU     $
                BYTE    50H             ;Open for write to
                BYTE    01H             ;   Channel 1.
                BYTE    00H             ;Status.
                BYTE    00H             ;Reserved.
                WORD    0000H           ;Not used.
                WORD    0000H           ;Not used.
                LONG    OPEN.STRNG      ;Pointer to 'CONO' string.
;
PRINT.SRB       EQU     $
                BYTE    02H             ;Write ASCII and wait
                BYTE    01H             ;   on Channel 1.
                BYTE    00H             ;Status.
                BYTE    00H             ;Reserved.
                WORD    0000H           ;Not used.
                WORD    80H             ;Buffer length.
                LONG    PRINT.STRNG     ;Pointer to string.
;
OPEN.STRNG      ASCII   'CONO'          ;ASCII of 'CONO', plus
                BYTE    0DH             ;   a carriage return.
PRINT.STRNG     ASCII   'MEMORY PARTITIONING EXAMPLE'   ;ASCII of the printed
                BYTE    0DH             ;   string, plus carriage return.
;
                ORG     /2              ;Start User stack on an even byte.
                BLOCK   400H            ;Reserve User stack space.
USER.STACK      EQU     $
;
```

Fig. 7L-11. Memory partitioning example program (part 4 of 5).

```
;
;-----------------------------------------------------------------------
;
; ------SVC trigger definitions.
;
;       We'll want these to be in all memory spaces (non-partitioned).
;
                SECTION SVCTRIG
;
SVC8            BLOCK   1
SVC7            BLOCK   1
SVC6            BLOCK   1
SVC5            BLOCK   1
SVC4            BLOCK   1
SVC3            BLOCK   1
SVC2            BLOCK   1
SVC1            BLOCK   1
;
                END     START
```

Fig. 7L-11. Memory partitioning example program (part 5 of 5).

The memory location assigned to each exception vector by the 68000 is listed in Table 7L-7.

Table 7L-7
68000 Exception Vector Assignment

Vector Number	Address (Hex)	Memory Space	Vector Assignment
0	000	SP	Reset: Initial SSP (*a)
	004	SP	Reset: Initial PC (*a)
2	008	SD	Bus Error
3	00C	SD	Address Error
4	010	SD	Illegal Instruction
5	014	SD	Zero Divide
6	018	SD	CHK Instruction
7	01C	SD	TRAPV Instruction
8	020	SD	Privilege Violation
9	024	SD	Trace
10	028	SD	Line 1010 Emulator
11	02C	SD	Line 1111 Emulator
12 (*b)	030	SD	(Unassigned, Reserved)
13 (*b)	034	SD	(Unassigned, Reserved)
14 (*b)	038	SD	(Unassigned, Reserved)
15	03C	SD	Uninitialized Interrupt Vector
16--23 (*b)	040	SD	(Unassigned, Reserved)
	05F		
24	060	SD	Spurious Interrupt (*c)

Table 7L-7 (con't)

Vector Number	Address (Hex)	Memory Space	Vector Assignment
25	064	SD	Level 1 Interrupt Autovector
26	068	SD	Level 2 Interrupt Autovector
27	06C	SD	Level 3 Interrupt Autovector
28	070	SD	Level 4 Interrupt Autovector
29	074	SD	Level 5 Interrupt Autovector
30	078	SD	Level 6 Interrupt Autovector
31	07C	SD	Level 7 Interrupt Autovector
32--47	080 0BF	SD	TRAP Instruction Vectors (*d)
48--63 (*b)	0C0 0FF	SD	(Unassigned, Reserved)
64--255	100 3FF	SD	User Interrupt Vectors

(*a) The reset vector (0) requires four words, unlike other vectors which only require two words. It is located in Supervisor Program (SP) space.
(*b) Vectors 12, 13, 14, 16--23, and 48--63 are reserved by Motorola for future enhancements. No user peripheral devices should be assigned to these numbers.
(*c) The spurious interrupt vector is taken when a bus error is indicated during interrupt processing.
(*d) TRAP #n uses vector number 32+n.

The remainder of this discussion steps you through the procedure you follow to assemble, link, allocate memory, load and run the example program.

Assembling and Linking the Program. Assemble MEMPAR.SRC with the following command:

```
     > ASM MEMPAR.OBJ,,MEMPAR.SRC <CR>
```

Assume that you have a linker command file called MEMPAR.LNK which contains the following linker command options:

```
          -l f
          -d
          -O MEMPAR.OBJ
          -o MEMPAR.LOA
          -m RVEC=08000000-08000007
          -m IVEC=04000000-040003FF
          -m SVCTRAP=00F00000-00F00007
          -m UD=01000000-0100FFFF
          -m UP=02000000-0200FFFF
          -m SD=04000400-0400FFFF
          -m SP=08000008-0800FFFF
          -L SEC=RESET.VEC BASE RVEC
          -L SEC=INTERRUPT.VEC BASE IVEC
          -L SEC=SVCTRIG BASE SVCTRAP
          -L SEC=SUPER.CODE BASE SP
          -L SEC=SUPER.DATA BASE SD
          -L SEC=USER.CODE BASE UP
          -L SEC=USER.DATA BASE UD
```

Link using this command:

```
          > LINK -C MEMPAR.LNK <CR>
```

The linker options are explained in the following paragraphs:

The -l f option gives you a full linker listing, and -d puts symbol information for symbolic debug in the load module. MEMPAR.OBJ and MEMPAR.LOA are the names of the object module from the assembler, and the load module output by the linker, respectively.

The -m option assigns logical memory names to blocks of program memory. Here, RVEC is the block in Supervisor Program space for the reset vector; IVEC is the block in Supervisor Data space for the exception vectors; and SVCTRAP is the block for the SVC trigger locations. When the program is loaded, SVCTRAP will be in the default memory space as selected by the MEMSP S command.

The names UD and UP are assigned to two 64K blocks of memory. The first is in User Data space and the second in User Program space. SD is the block starting after the IVEC block in Supervisor Data space. SP starts after the RVEC block in Supervisor Program space.

The -L option locates the program sections in the memory blocks just defined. Since there is only one section in each memory block, you can use the BASE parameter. This causes each section to be located at the beginning of its memory block. If there was more than one section in a memory block, you would used the RANGE parameter to locate those sections somewhere within the desired block.

Allocate Memory. You must allocate memory to load and execute the program in mode 0. Enter the following command line:

```
> AL UD:0 1FFF ; AL UP:0 1FFF ; AL SD:0 1FFF ; AL SP:0 1FFF ; AL 0F00000 <CR>
```

You can check the memory allocations by entering the AL command with no parameters:

```
> AL <CR>
 00000000   -  00000FFF    UD:  ...  ...  ...
 00000000   -  00000FFF    ...  UP:  ...  ...
 00000000   -  00000FFF    ...  ...  SD:  ...
 00000000   -  00000FFF    ...  ...  ...  SP:
 00001000   -  00001FFF    UD:  ...  ...  ...
 00001000   -  00001FFF    ...  UP:  ...  ...
 00001000   -  00001FFF    ...  ...  SD:  ...
 00001000   -  00001FFF    ...  ...  ...  SP:
 00F00000   -  00F00FFF    UD:  UP:  SD:  SP:
  9  BLOCK(S) ALLOCATED    23 BLOCK(S) FREE
```

Each of the four memory spaces has two blocks allocated to it exclusively. The block used for the SVC trigger locations has been allocated to all four memory spaces.

Now you can load the program:

```
> LO MEMPAR.LOA <CR>
```

The processor must be in Supervisor mode to start running the program. Use the RESET command to set Supervisor mode and put the start address in the reset vector.

```
> RESET <CR>
```

Start program execution with the G command:

```
> G <CR>
MEMORY PARTITIONING EXAMPLE
>
```

Program execution begins at the label START in the Supervisor Program. The Supervisor starts the User job. The User job opens a channel to the system terminal and prints the string "MEMORY PARTITIONING EXAMPLE." The User then returns control to the Supervisor, which exits, and control returns to the operating system.

The 68000 STOP Instruction

In Mode 0. A break is always generated when a STOP instruction is executed in mode 0. The <BREAK STOP> message and the registers are displayed, and PC(next) points to the instruction after the STOP.

With TRA ON. If TRA is ON, a STOP instruction will cause a break in all three modes.

In Mode 1 or 2 with TRA OFF. When the 68000 emulator is running in mode 1 or 2 with TRA OFF and encounters a STOP instruction, it stops and waits for an interrupt. However, if a system interrupt occurs instead of a user interrupt, a break is generated, and the registers and <BREAK STOP> message are displayed. If you want to execute the STOP again, you must adjust the PC, which is pointing to the instruction after the STOP. This can be accomplished with a command like, "G PC-4".

In addition, since these system interrupts are usually keyboard or timer interrupts, you can avoid them (1) by not typing on the keyboard while the emulator is running, and (2) by using the TTA timing options rather than the CLOCK command.

If a user interrupt occurs after a STOP break, it will still be latched, and will be honored at the next G command.

NOTE

> The preceding paragraphs assume that jumper J2144 on the EMU 2 board is in its normal position. In the optional position, a STOP instruction always generates a break. Refer to the subsection, "Jumpers", in this section for further information.

When an Interrupt Occurs Near a STOP. When the emulator is running with TRA OFF, and breaks because of a breakpoint, or a TTA, SVC, or MAC interrupt in the vicinity of a STOP instruction, the STOP will not be detected. Because of this, the 68000 emulator does not adjust the PC when a STOP is detected.

JUMPERS

The 68000 Emulator Processor Module and Prototype Control Probe have several user-selectable configuration jumpers. The following paragraphs describe the functional characteristics of each of these jumpers. All jumpers are in the "normal" position (1-2) when shipped from the factory. These jumpers affect the operation of the emulator in all three emulation modes unless otherwise indicated.

EMU 1 BOARD

P1080---Emulator Halt Control Selector

After a 68000 HALT condition occurs (double bus error or double address error), P1080 determines whether control returns to the operating system, or remains with the prototype.

In the normal position (1-2), control is returned to the system in all three emulation modes.

In the optional position (2-3), control is returned to the system only in mode 0. In modes 1 and 2, control remains with the prototype. Since the microprocessor must exit the halted condition, the system will hang unless the prototype circuitry resets the microprocessor.

EMU 2 BOARD

J2144---Break Cycle Control Selector

NOTE

EMU 2 must be removed from your development system to access J2144.

J2144 controls the break cycle of the emulator after the emulator executes a STOP instruction.

In the normal position (1-2), when the emulator is running in emulation mode 0, and a STOP instruction is executed, the emulator will break and return control to the operating system.

In the optional position (2-3), the emulator will always break and return control to the operating system when a STOP instruction is executed.

NOTE

When J2144 is in its normal (1-2) position and the emulator is operating in emulation mode 1 or 2, the system will appear to hang. You must type CTRL-C or issue a prototype interrupt, to return control to the operating system.

INTERFACE BUFFER BOARD

The Buffer board contains six configuration jumpers. To access these jumpers, you must perform the Prototype Control Probe Assembly/Disassembly procedure described in the 68000 Emulator Service Manual.

P1---Data Transfer ACKnowledge (DTACK) Delay

P1 inserts or removes a delay of the prototype's Data Transfer ACKnowledge (DTACK) to the 68000 microprocessor when the emulator is in mode 1.

NOTE

The configuration of P1 depends on the configuration of jumpers J1045 and J2045 on the Mobile Microprocessor board. These jumpers are discussed later in this section.

In the normal position (1-2), the prototype's DTACK is delayed at the rate determined by J1045 and J2045. This prevents overdriving of the program memory's access time.

In the optional position (2-3), J1045 and J2045 are bypassed, so that the prototype returns DTACK without delay.

NOTE

When P1 is in the optional (2-3) position, data may be invalid or lost if program memory is accessed faster than its time limitations allow.

P2 and P3---Prototype Bus Arbitration Control

P2 and P3 determine when the prototype is allowed to control the 68000 bus.

In the normal position (1-2), the prototype's Bus Request and Bus Grant Acknowledge signals to the 68000 microprocessor are disabled whenever the emulator returns control to the operating system.

In the optional position (2-3), the prototype's Bus Request and Bus Grant Acknowledge signals are allowed to request and hold the 68000 bus, even when the emulator has started its Dump and Restore, and has returned control to the operating system.

NOTE

In the optional (2-3) positions for P2 and P3, the system may hang. To return to normal operation, you must release the bus.

P6---Address Strobe Control

P6 determines when the 68000 microprocessor address strobe is driven to the prototype circuit.

In the normal position (1-2), the 68000 microprocessor address strobe is driven in all cycles except: (1) during an emulator Dump and Restore, and (2) during an interrupt acknowledge of a Non-Maskable Interrupt (NMI) issued by the emulator.

In the optional position (2-3), the 68000 microprocessor address strobe is driven in all cycles except during an interrupt acknowledge of an NMI from the emulator.

P7---DTACK Timeout Control

P7 controls how the emulator will behave when no prototype DTACK occurs within 1 ms. This jumper is used when any of the following conditions exist:

- P8 is in normal (1-2) position, memory is mapped to the prototype, and no prototype DTACK is generated.
- P8 is in optional (2-3) position and no DTACK is generated by the prototype.
- The development system is operating in mode 2 and no prototype DTACK is generated.

In the normal position (1-2), the system will hang until a DTACK is received from the prototype, or until a break condition occurs.

In the optional position (2-3), the system will hang until a DTACK is received from the prototype, at which time the system will continue operating as usual. A break condition will not clear the system.

P8---Internal Generation of DTACK in Mode 1

P8 allows or prevents the internal generation of a Data Transfer ACKnowledge (DTACK) signal by the 68000 emulator while in emulation mode 1.

In the normal position (1-2), the prototype's DTACK is used until memory has been mapped. If memory is mapped to program memory, then an internal DTACK is generated. If memory is mapped to the prototype, DTACK must be generated by the prototype.

In the optional position (2-3), no internal generation of a DTACK signal is allowed, regardless of mapping. All DTACK signals must originate from the prototype.

NOTE

> When P8 requires a prototype DTACK, the prototype must generate a DTACK within 1 ms. If DTACK is not generated within 1 ms, then a DTACK timeout occurs. Refer to the preceding discussion of jumper P7.

INTERFACE CONTROL BOARD

The Interface Control board contains two configuration jumpers. To access these jumpers, follow the Prototype Control Probe Assembly/Disassembly procedure described in the 68000 Emulator Service Manual.

J4011---Save Non-Maskable Interrupts

J4011 controls whether Non-Maskable Interrupts (NMIs) are saved during Dump and Restore (D/R) routines.

In the normal position (1-2), NMIs are saved during D/R routines (for example, when the development system has control and the emulator is not running).

In the optional position (2-3), NMIs are not saved under any circumstances.

NOTE

> Saved NMIs are issued to the 68000 microprocessor when the development system relinquishes control, and the emulator begins program execution.

J6021---Save Prototype Interrupts

J6021 controls whether prototype circuit interrupts (interrupt levels other than level 7) are saved during Dump and Restore (D/R) routines.

In the normal position (1-2), prototype interrupts, if held until acknowledged, are saved during D/R routines (for example, when the development system has control and the emulator is not running).

In the optional position (2-3), prototype circuit interrupts are not saved under any circumstances.

NOTE

> Saved interrupts are issued to the 68000 microprocessor when the development system relinquishes control, and the emulator begins program execution.

MOBILE MICROPROCESSOR BOARD

Two configuration jumpers are located on the Mobile Microprocessor board. To access these jumpers, perform the Prototype Control Probe Assembly/Disassembly procedure as described in the 68000 Emulator Service Manual.

J1045 and J2045---Delay of DTACK Assertion

Two different DTACK (Data Transfer ACKnowledge) signals may be issued to the emulator: the 68000 microprocessor's DTACK and the prototype's DTACK. With both J1045 and J2045 in their optional positions, assertion of all 68000 microprocessor DTACK signals to the emulator is delayed. In addition, assertion of the prototype's DTACK signals to the emulator is delayed only when operating in mode 1 with jumper P1 in its normal position. (Refer to the discussion of Interface Buffer board jumper P1 earlier in this section.)

CAUTION

> Use of J1045 and J2045 in their normal positions may cause invalid data or loss of data if program memory is accessed faster than its limitations allow. However, no component damage will result.

The positioning of these jumpers depends on the program memory configuration installed in your development system, as shown in Table 7L-8.

Table 7L-8
J1045 and J2045 Configurations (*a)

Memory Configuration	Jumper Configuration (*b)		Characteristic
	J1045	J2045	
32K Program Memory board	A1 to A	A1 to A	Normal (no delay)
64K or 128K Static Program Memory board	A1 to A	A1 to A	Normal (no delay)
64K or 128K Static Program Memory board and Memory Allocation Ctrlr.	A1 to A	A1 to A	Normal (no delay)
32K Program Memory board and Memory Allocation Ctrlr.	A5 to B	A1 to A	Option (one wait state delay at > 6.4MHz) (*c)

(*a)---Jumper configurations listed are for < 8 MHz operation.
(*b)---Jumper configurations not listed are for future use.
(*c)---One wait state is equivalent to one extra clock cycle per memory cycle.

EMULATOR TIMING

The signals between the prototype and the emulating microprocessor are buffered. Therefore, some timing differences exist between the 68000 emulator and a 68000 microprocessor inserted directly into the prototype. Table 7L-9 lists the emulator/microprocessor timing differences for the 68000. Figures 7L-12 and 7L-13 are timing diagrams that correspond to the signals listed in Table 7L-9.

Table 7L-9
68000 Emulator/Microprocessor Timing Differences

Number	Characteristic	Symbol	Processor Min	Processor Max	Emulator Min	Emulator Max	Unit
1	Clock Period	'cyc	125	500	125	500	ns
2	Clock Width Low	'CL	55	250	55	250	ns
3	Clock Width High	'CH	55	250	55	250	ns
4	Clock Fall Time	'Cf	—	10	—	10	ns
5	Clock Rise Time	'Cr	—	10	—	10	ns
6	Clock Low to Address	'CLAV	—	70	—	92	ns
6A	Clock High to FC(H) Valid	'CHFCV	—	70	—	92	ns
7A	Clock High to Address High Impedance (Max.)	'CHAZx	—	80	—	114	ns
7B	Clock High to Data High Impedance (Max.)	'CHDZx	—	80	—	130	ns
8	Clock High to Address/FC(H) Invalid (Min.)	'CHAZn	0	—	14	—	ns
9[1]	Clock High to AS(L), DS(L) Low (Max.)	'CHSLx	—	60	—	86	ns
10	Clock High to AS(L), DS(L) Low (Min.)	'CHSLn	0	—	14	—	ns
11[2]	Address to AS(L), DS(L) (Read) Low/AS(L) Write	'AVSL	30	—	24	—	ns
11A[2]	FC(H) Valid to AS(L), DS(L) (Read) Low/AS(L) Write	'FCVSL	60	—	44	—	ns
12[1]	Clock Low to AS(L), DS(L) High	'CLSH	—	70	—	96	ns
13[2]	AS(L), DS(L) High to Address/FC(H) Invalid	'SHAZ	30	—	20	—	ns
14[2,5]	AS(L), DS(L) Width Low (Read)/AS(L) Write	'SL	240	—	240	—	ns
14A[2]	DS(L) Width Low (Write)	—	115	—	115	—	ns
15[2]	AS(L), DS(L) Width High	'SH	150	—	150	—	ns
16	Clock High to AS(L), DS(L) High Impedance	'CHSZ	—	80	—	93	ns
17[2]	AS(L), DS(L) High to R(H)/W(L) High	'SHRH	40	—	40	—	ns
18[1]	Clock High to R(H)/W(L) High (Max.)	'CHRHx	—	70	—	96	ns
19	Clock High to R(H)/W(L) High (Min.)	'CHRHn	0	—	14	—	ns
20[1]	Clock High to R(H)/W(L) Low	'CHRL	—	70	—	96	ns
21[2]	Address Valid to R(H)/W(L) Low	'AVRL	20	—	14	—	ns
21A[2]	FC(H) Valid to R(H)/W(L) Low	'FCVRL	60	—	54	—	ns
22[2]	R(H)/W(L) Low to DS(L) Low (Write)	'RLSL	80	—	80	—	ns
23	Clock Low to Data Out(H) Valid	'CLDO	—	70	—	95	ns
25[2]	DS(L) High to Data Out(H) Invalid	'SHDO	30	—	20	—	ns
26[2]	Data Out(H) Valid to DS(L) Low (Write)	'DOSL	30	—	20	—	ns
27[6]	Data In to Clock Low (Setup Time)	'DICL	15	—	17	—	ns
28[2]	AS(L), DS(L) High to DTACK(L) High	'SHDAH	0	120	0	94	ns
29	DS(L) High to Data Invalid (Hold Time)	'SHDI	0	—	0	—	ns
30	AS(L), DS(L) High to BERR(L) High	'SHBEH	0	—	0	—	ns
31[2,6]	DTACK(L) Low to Data In (Setup Time)	'DALDI	—	90	—	88	ns
32	HALT(L) and RESET(L) Input Transition Time	'RHrf	0	200	0	200	ns
33	Clock High to BG(L) Low	'CHGL	—	70	—	93	ns
34	Clock High to BG(H) High	'CHGH	—	70	—	93	ns
35[7]	BR(L) Low to BG(L) Low	'BRLGL	1.5	3	1.5	3	Clk Per
36[7]	BR(L) High to BG(L) High	'BRHGH	1.5	3	1.5	3	Clk Per
37	BGACK(L) Low to BG(L) High	'GALGH	1.5	3	1.5	3	Clk Per
38	BG(L) Low to Bus High Impedance (With AS(L) High)	'GLZ	—	80	—	148	ns
39	BG(L) Width High	'GH	1.5	—	1.5	—	Clk Per
46	BGACK(L) Width	'BGL	1.5	—	1.5	—	Clk Per
47[6,8]	Asynchronous Input Setup Time	'ASI	20	—	20	—	ns
48	BERR(L) Low to DTACK(L) Low[3]	'BELDAL	50	—	50	—	ns
53	Data Hold from Clock High	'CHDO	0	—	0	—	ns
55	R(H)/W(L) to Data Bus Impedance Change	'RLDO	30	—	20	—	ns
56	HALT(H)/RESET(L) Pulse Width[4]	'HRPW	10	—	10		Clk Per

[1] For a loading capacitance of less than or equal to 50 picofarads, subtract 5 nanoseconds from the values given in these columns.

[2] Actual value depends on clock period.

[3] If #47 is satisfied for both DTACK(L) and BERR(L), #48 may be 0 ns.

[4] After Vcc has been applied for 100 ms.

[5] For T6E, BF4, and R9M mask sets #14 and #14A are one clock period less than the given number.

[6] If the asynchronous setup time (#47) requirements are satisfied, the DTACK(L) low-to-data setup time (#31) requirement can be ignored. The data must only satisfy the data-in to clock-low setup time (#27) for the following cycle.

[7] For the Probe-tip add 20 nanoseconds to the clock periods listed.

[8] VPA = 50 nanoseconds for the Probe-tip.

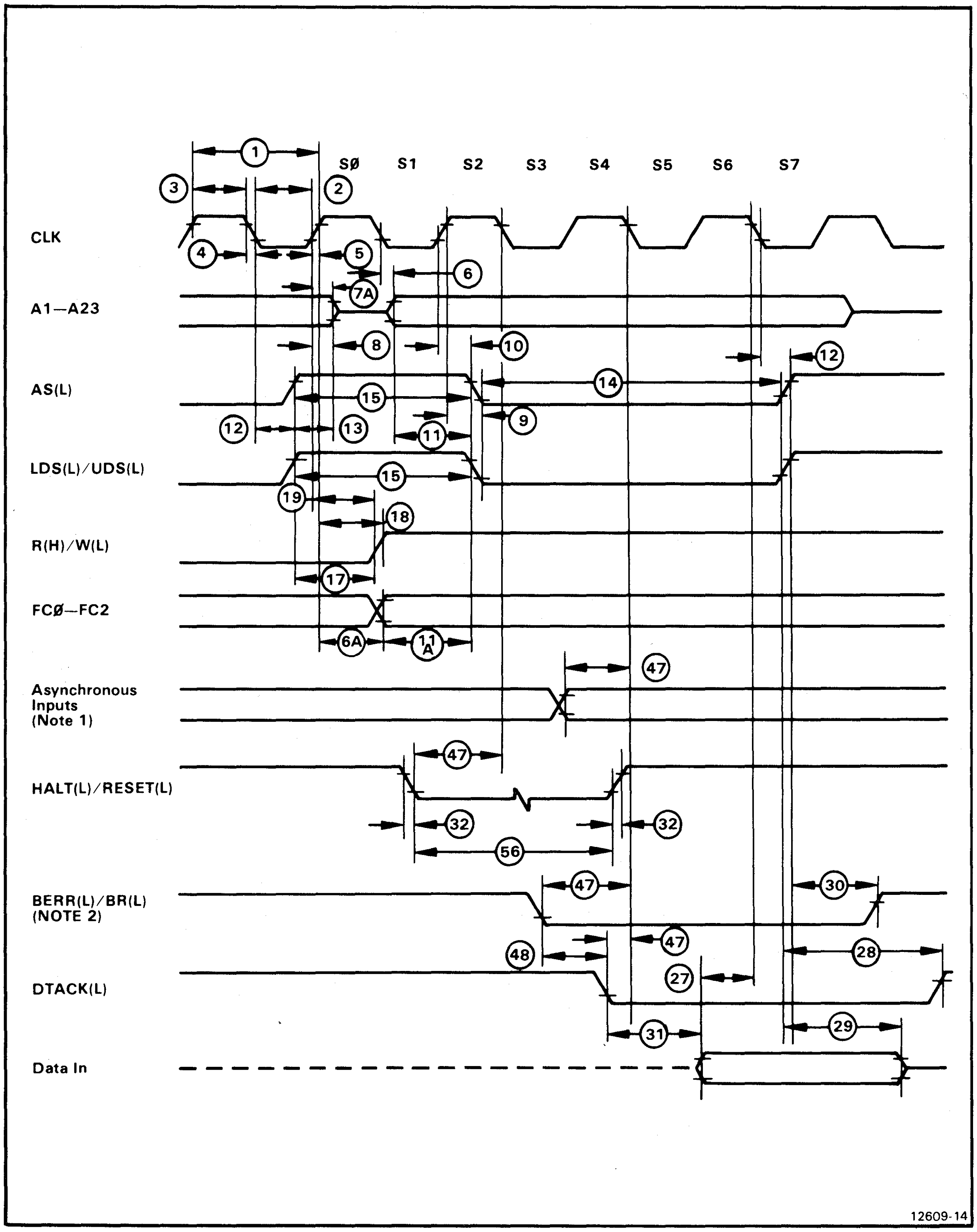


Fig. 7L-12. 68000 timing diagram, read cycle.

Notes and circled numbers refer to Table 7L-9.

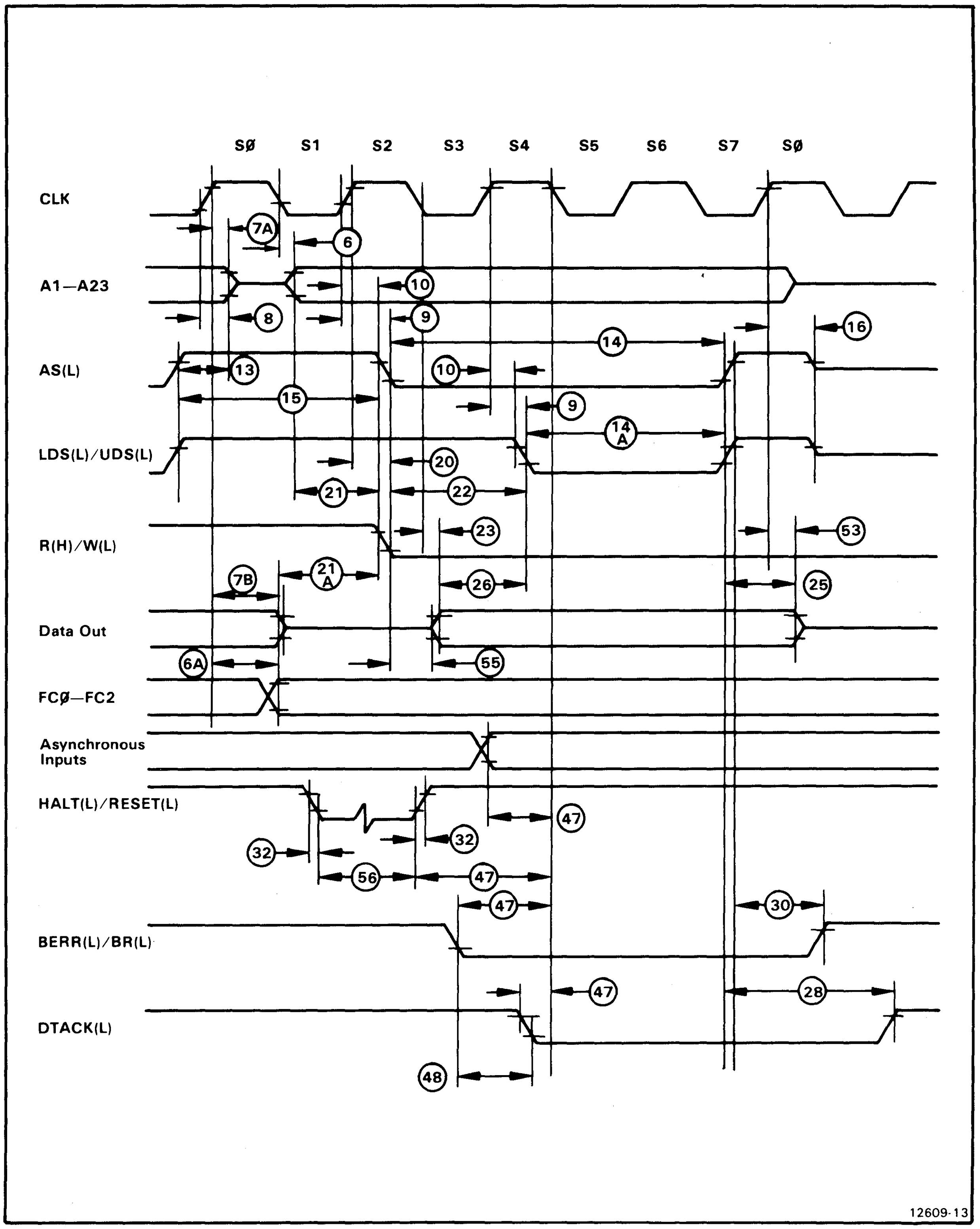


Fig. 7L-13. 68000 timing diagram, write cycle.

Circled numbers refer to Table 7L-9.

PROBE/PROTOTYPE INTERFACE DIAGRAM

Figure 7L-14 is a block diagram of the interface between the prototype and the 68000 Prototype Control Probe.

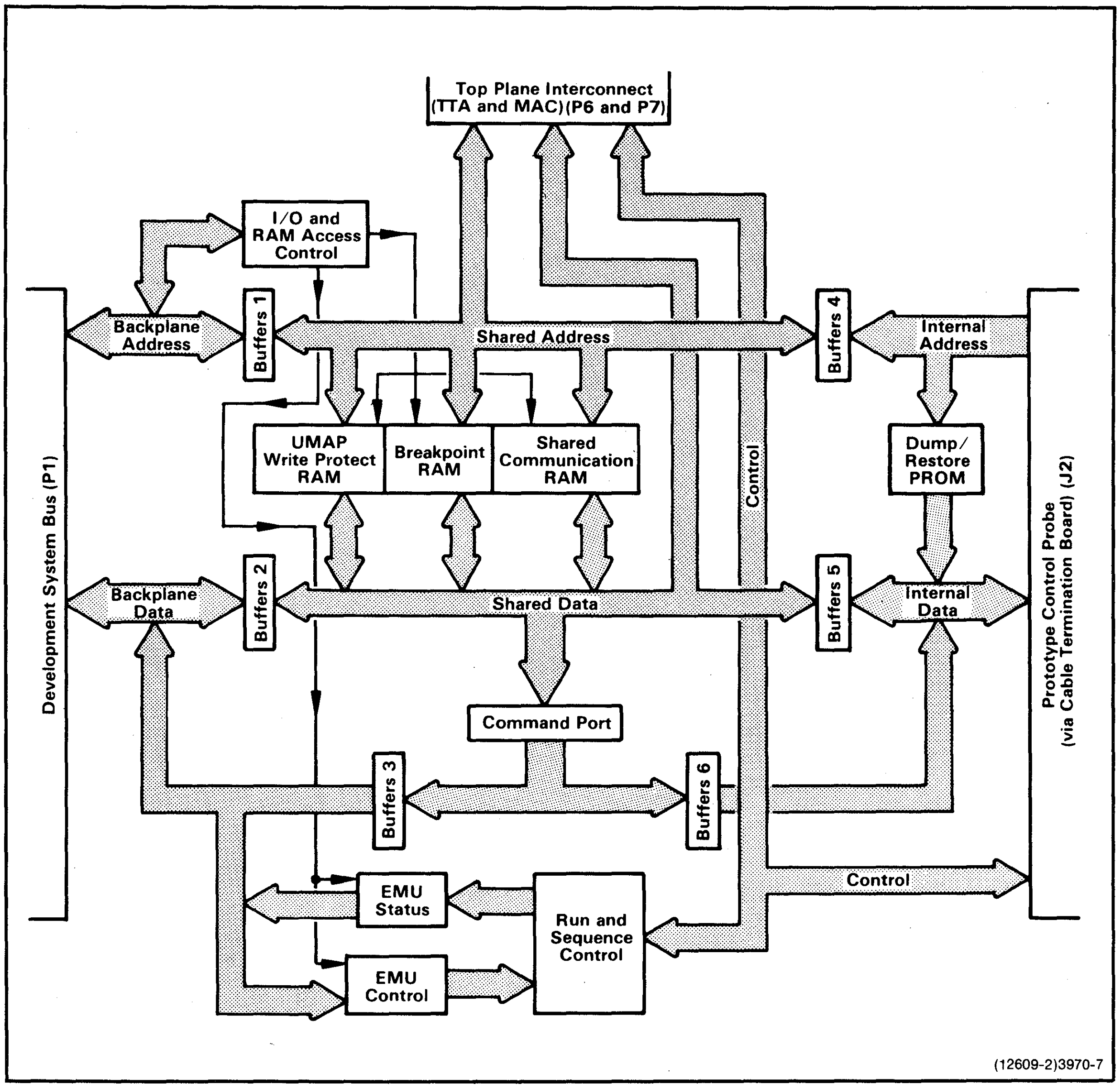


Fig. 7L-14. 68000 Prototype/Control Probe interface.

This figure provides a functional overview of signal buffering between the prototype and the emulating microprocessor. A more detailed circuit description can be found in the 68000 Emulator Processor and Prototype Control Probe Service Manual.

INSTALLING YOUR 68000 EMULATOR SOFTWARE

8540 SOFTWARE INSTALLATION PROCEDURE

The ROMs that contain the control software for your 68000 emulator must be installed in your 8540's System ROM Board. Refer to your 8540 Installation Guide for instructions on how to install these ROMs.

8550 SOFTWARE INSTALLATION PROCEDURE

Your emulator software installation disk contains two types of software:

- emulator control software, which you install onto your DOS/50 system disk so that DOS/50 can control your emulator hardware;
- emulator diagnostic software, which you install onto your 8550 system diagnostic disk so that diagnostic tests can be run on your emulator as well as on other 8550 system hardware.

This subsection describes how to install the control software and diagnostic software for your 68000 emulator.

To complete these installation procedures you need the following items:

- an 8550 system (with or without a 68000 emulator)
- a DOS/50 system disk with a write-enable tab over the write-protect slot
- a 68000 emulator software installation disk with no write-enable tab
- (for installation of diagnostic software) an 8550 system diagnostic disk with a write-enable tab over the write-protect slot.

Each installation procedure takes about five minutes.

Start Up and Set the Date

Turn on your 8550 system. (For start-up instructions, refer to the Learning Guide of your System Users Manual.) Place your system disk in drive 0 and shut the drive 0 door. When you see the > prompt on your system terminal, place your installation disk in drive 1 and shut the drive 1 door.

Use the DAT command to set the date and time. For example, if it is 11:05 am on October 12, 1982, type:

```
> DAT 12-OCT-82/11:05 <CR>
```

The system uses this information when it sets the CREATION time attribute of each file copied from your installation disk.

Install the Emulator Control Software

The command file INSTALL2, which installs the emulator control software, resides on the installation disk. To execute the command file, simply type its filespec:

```
     > /VOL/EMU.68000/INSTALL2 <CR>
```

DOS/50 responds with the following message:

```
     * During this installation procedure, one or more of the
     * following messages may appear.   IGNORE THESE MESSAGES:
     *
     *       Error 6E - Directory alteration invalid
     *       Error 7E - Error in command execution
     *       Error 1D - File not found
     *
     * If any OTHER error message appears, see your
     * Users Manual for further instructions.
     *
     * If no other error message appears, you'll receive a
     * message when the installation procedure is complete.
     *
     T,OFF
```

In this installation procedure, you may disregard error messages 6E, 7E, and 1D; these messages have no bearing on the success of the installation. However, if a message other than 6E, 7E, or 1D appears, take the following steps:

1. Make sure you are using the right disks.
2. Make sure your system disk has a write-enable tab.
3. Make sure there are at least 16 free files and 150 free blocks on your system disk.
4. Begin the installation procedure again.

If the installation procedure fails again, copy down the error message and contact your Tektronix service representative.

The "T,OFF" command suppresses subsequent output to your system terminal (except error messages) until INSTALL2 finishes executing. Within about five minutes, INSTALL2 will finish and your system terminal will display the following message:

```
     *
     * Your installation has been completed.
     >
```

Install the Emulator Diagnostic Software

Note the Name of Your Diagnostic Disk. In order to install the emulator diagnostic software, you must know the name of your 8550 system diagnostic disk. Remove your emulator installation disk from drive 1 and insert the diagnostic disk. Enter the following command to list the names of the two disks mounted in your 8550:

```
> ATT /VOL/* WHERE <CR>
sysvol          WHERE=FLX0  <-- DOS/50 system disk
8550DIAGx.x     WHERE=FLX1  <-- 8550 system diagnostic disk
```

Note the name of your diagnostic disk. (It should be something like "8550DIAG2.0".)

Insert Your Emulator Installation Disk into Drive 1. INSTALLDIAGS, the command file that installs the diagnostics, resides on the installation disk. Remove your diagnostic disk from drive 1 and insert your installation disk. Invoke the INSTALLDIAGS command file and pass it the name of your diagnostic disk, which you just noted:

```
> /VOL/EMU.68000/INSTALLDIAGS 8550DIAGx.x <CR>
```

DOS/50 responds with the following messages:

```
*
*********************************************
*     DIAGNOSTIC INSTALLATION PROCEDURE     *
*********************************************
*
*     During this installation procedure, the following error
*     message will appear once.  IGNORE THIS MESSAGE:
*
*          Error 2A Parameter required
*
*     If any OTHER error message appears or this appears more
*     than once, see your Users Manual for further instructions.
*
*     If no other error message appears, you'll receive a message
*     when the installation is complete.
*
T,OFF
COP:             Error 2A Parameter required
*
*-----> Remove the DOS/50 System Disc
*-----> Insert the 8550 System Diagnostic Disc
*-----> Type CO -A
*
SUSP,-A

>>
```

Insert Your Diagnostic Disk into Drive 0. Remove your DOS/50 system disk from drive 0 and insert your 8550 system diagnostic disk. Then enter the command CO -A to continue execution of the command file:

```
     >> CO -A <CR>
```

After a few minutes, the following message is displayed:

```
     COP,-BN,/VOL/EMU.68000/DIAGS/68000TST.SAV,/VOL/8550DIAGx.x/68000TST.SAV
     *
     *-----> Remove 8550 System Diagnostic Disc
     *-----> Insert DOS/50 System Disc
     *-----> Type CO -A
     *
     SUSP,-A
```

Insert Your DOS/50 System Disk into Drive 0. Remove your diagnostic disk from drive 0 and insert your DOS/50 system disk. Then type CTRL-C and enter the CO -A command again:

```
     >> CO -A <CR>
```

The command file finishes with the following message:

```
     USER,,NO.NAME
     ********************************************
     *     DIAGNOSTIC INSTALLATION COMPLETE     *
     ********************************************
     >
```

In this installation procedure, error message 2A should appear once. If any other error message appears, check your disks and begin the diagnostic installation procedure again. If the installation procedure fails again, copy down the error message and contact your Tektronix service representative.

Once your software is installed, you can:

- o remove your disks and turn off your 8550 system, or
- o install more software, or
- o continue with the 68000 Emulator Demonstration Run that follows in this section. If you do this, you do not have to restart the system or reset the date and time.

NOTE

At this point, "NONAME" is the current user. To change the current user back to "yourname," enter USER,,yourname.

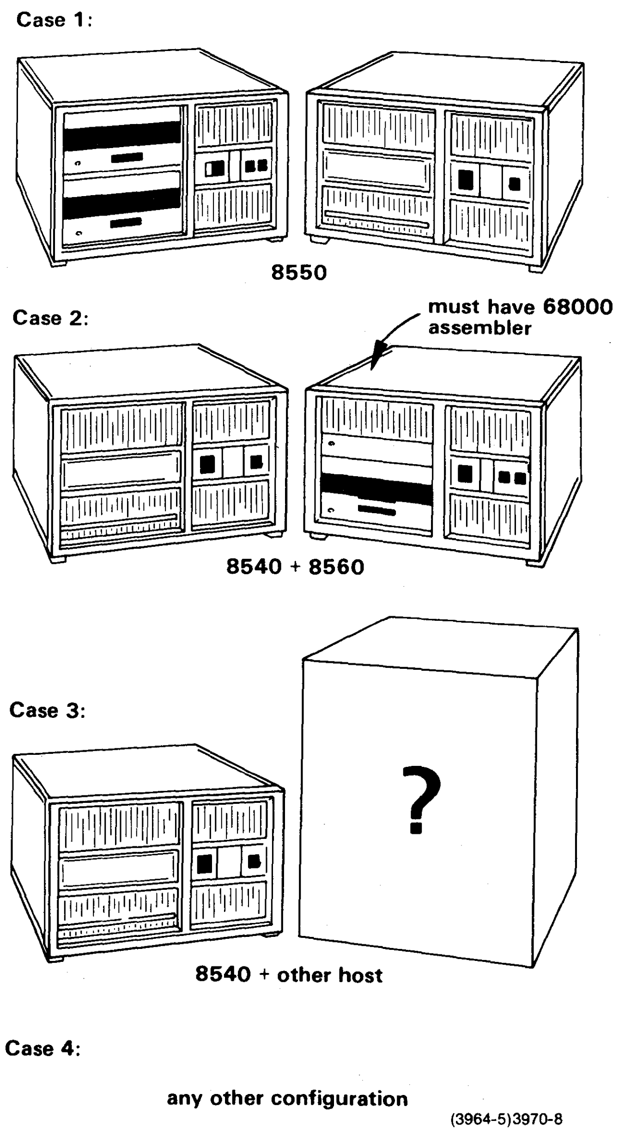
68000 DEMONSTRATION RUN

INTRODUCTION

This demonstration run shows you how to load, execute, and monitor a simple 68000 assembly language program on your 8540 or 8550. To perform this demonstration, your 68000 emulator hardware and control software must be installed in your 8540 or 8550. Throughout this demonstration run, the term "68000 assembler" refers to a B Series 68000 Assembler.

Figure 7L-15 shows the source and object code for the demonstration program.

This demonstration run includes procedures for four different system configurations:



Case 1: If you have an 8550, the source code and object code for the demonstration program are provided on the installation disk that contains your 68000 emulator control software. This demonstration shows you how to assemble the program on your 8550. (If your system disk does not have a 68000 assembler, you must skip that part of the demonstration.)

Case 2: If you have an 8540/8560 system, and your 8560 has a 68000 assembler, you can create and assemble the program on the 8560 and download it to the 8540. This demonstration shows how.

Case 3: If you have an 8540 that is connected to a host computer other than an 8560, we cannot give you a specific list of commands for creating and assembling the program on your host. However, Fig. 7L-16 gives the program object code in Extended Tekhex format. You can create the Tekhex file using your host's assembler or text editor, and then download the file to the 8540 via the 8540's optional COM interface.

Case 4: If none of the other cases applies to you, you can patch the program into memory using the P command. This demonstration shows how.

Once the program is loaded or patched into memory, you can execute the program on your emulator.

NOTE

The 8540 commands shown in this demonstration can also be used on an 8550 that is connected to an 8560 or another host computer.

```
ASM          68000                                              Page    1
Xnn.nn-nn  (8550)                                         dd-mmm-yy/hh:mm:ss

 1                            LIST    DBG                 ;Turn on symbolic debug
 2                                                        ;   option.
 3                            SECTION DEMO
 4                  700 R     ORG     700H                ;Begin summing routine.
 5 00000700 207C0000   START  MOVEA.L #TABLE,A0           ;Set the table pointer.
            0500      R
 6 00000706 323C0004          MOVE.W  #TSIZE-1,D1         ;Set the pass counter.
 7 0000070A 4280              CLR.L   D0                  ;Clear the register to be
 8                                                        ;   used for summation.
 9 0000070C D018       LOOP   ADD.B   (A0)+,D0            ;Add byte from table to D0
10 0000070E 51C9FFFC          DBRA    D1,LOOP             ;Decrement, and branch
11                                                        ;   if not 5 passes yet.
12 00000712 13C000F0          MOVE.B  D0,(GEN.L)0F00007H ;Else exit, trigger SVC 1.
            0007
13 00000718 4E71              NOP                         ;Two NOPs for SVC.
14 0000071A 4E71              NOP                         ;End of summing routine.
15                     ;
16 0000071C 1A         EXIT   BYTE    1AH                 ;1AH = function code
17                                                        ;   for the exit SVC.
18                   C0 R     ORG     0C0H                ;Define SRB vector space.
19 000000C0 0000071C R        LONG    EXIT                ;Define the SRB pointer.
20                     ;
21                    5  TSIZE EQU    5                   ;Set table size = 5.
22                  500 R     ORG     500H                ;Put TABLE at 500H.
23 00000500           5  TABLE BLOCK  TSIZE               ;Space for TABLE.
24                     ;
25                  700       END     START
== ======== ========= ================================= ==========================
|     |         |                     |                            |
|  address  object code          source code                   comments
|
+-- source code line number
```

Fig. 7L-15. 68000 demonstration run program.

This display appears on the first page of your 68000 Series B Assembler listing using an 8550.

```
(A)

%436263700207C00000500323C00044280D01851C9FFFC13C000F000074E714E711A
%106292C00000071C
%443DB4DEMO010371D14LOOP370C15START370015TABLE350025TSIZE1514EXIT371C
%0981B3700

(B)

FIRST DATA BLOCK: object code for addresses 700--71B

header
  ¦ load address    object code
  ¦     ¦                ¦
======----==========================================================
%436263700207C00000500323C00044280D01851C9FFFC13C000F000074E714E711A


SECOND DATA BLOCK: object code for addresses C0--C4

header
  ¦   load  object
  ¦ address  code
  ¦    ¦      ¦
======---========
%106292C00000071C


SYMBOL BLOCK

header       section
  ¦  section definition
  ¦   name   field      symbol definition fields
  ¦     ¦      ¦                    ¦
======-----=======--------------------------------------------
%443DB4DEMO010371D14LOOP370C15START370015TABLE350025TSIZE1514EXIT371C


TERMINATION BLOCK

header
  ¦   transfer
  ¦   address
  ¦     ¦
======----
%0981B3700
```

Fig. 7L-16. 68000 demonstration program: Extended Tekhex format.

Figure 7L-16A lists an Extended Tekhex load module that contains the object code and program symbols for the demonstration program. Figure 7L-16B gives the meanings of the different fields in the message blocks. If you have a host computer other than an 8560, you can create this load module and download it to your 8540 or 8550.

EXAMINE THE DEMONSTRATION PROGRAM

The demonstration program adds five numbers from a table stored in locations 500--504 in program memory, and puts the sum in register D0. (You will place values in the table later in this demonstration.) The 8085A Emulator Demonstration Run in the Learning Guide of your System Users Manual contains a flowchart that illustrates the steps of the program.

The source code contains two kinds of statements: assembler directives (like ORG, WORD, BYTE, and GEN.L), and 68000 assembly language instructions. Most assembler directives are microprocessor-independent and are explained in the 8085A Emulator Demonstration Run. The only assembler directive that is 68000-specific is the GEN.L directive. This causes a long word address to be encoded. The 68000 assembly language instructions are discussed in the following paragraphs.

Set Table Pointer. The MOVEA.L #TABLE,A0 instruction moves the address of the table (500) into register A0. As a result, A0 points to the first element of the table. The label START is used by the END directive to specify that the MOVEA.L #TABLE,A0 instruction is the first to be executed.

Set Pass Counter. Register D1 is used as the pass counter. The MOVE.W #TSIZE-1,D1 instruction moves the value 5-1=4 into register D1. This causes the number of passes to be 5, since the DBRA instruction used for the branch will loop until the value in D1 is -1. Each time a number is taken from the table and added to register D0, register D1 is decremented.

Clear Summation Register. The CLR.L D0 instruction zeros register D0 so that you can start adding numbers from the table.

Add Byte from Table and Point to Next Byte. The next instruction, ADD.B (A0)+,D0, adds the byte addressed by A0 to register D0. After the byte is added, A0 is incremented to point to the next byte in the table. For example, A0 is initialized to contain 500. After the add is performed, the + part of the instruction causes A0 to be incremented to 501, the address of the second byte in the table. The label LOOP represents the address of the ADD.B instruction; this label is used by the DBRA D1,LOOP instruction.

Decrement Pass Counter and Loop If Not Yet Five Passes. The DBRA D1,LOOP instruction decrements register D1, the pass counter. Then it jumps to the LOOP label if D1 does not contain -1. If D1 does contain -1, the program proceeds to the next instruction, MOVE.B D0,(GEN.L)0F00007H.

Exit. The MOVE.B D0,(GEN.L)0F00007H and two NOP instructions constitute a service call (SVC) that causes an exit from the program. Any byte-write instruction to the address F00007 would cause an SVC, and the contents of D0 are not affected. For more information on SVCs, refer to the Service Calls section of your System Users Manual.

ASSEMBLE AND LOAD THE DEMONSTRATION PROGRAM

Now it's time to create the program so you can run it on your emulator. One of the following discussions describes the set of steps that is appropriate for your hardware configuration:

- For 8550 users --- Case 1: Assemble and Load on the 8550
- For 8560 users --- Case 2: Assemble and Load on the 8560; Download to the 8540
- For 8540 users with host computers other than the 8560 --- Case 3: Download from Your Host to the 8540
- For other hardware configurations --- Case 4: Patch the Program into Memory

Work through the discussion that is appropriate for you. Once you have put the program into program memory, turn to the heading, "Run the Demonstration Program", later in this section.

CASE 1: ASSEMBLE AND LOAD ON THE 8550

This discussion shows you how to copy the demonstration program from your 68000 emulator software installation disk, assemble the program, and load it into 8550 program memory.

Start Up and Log On

Turn on your 8550 system. (For start-up instructions, refer to the paragraph, "Start Up the 8550 and Its Peripherals", in the Learning Guide of your System Users Manual.) Place your system disk in drive 0 and shut the drive 0 door. When your system displays the ">" prompt, place your 68000 emulator software installation disk in drive 1 and shut the drive 1 door.

Use the DAT command to set the current date and time. For example, if it were 2:30 pm on October 12, 1982, you would enter the following command line:

```
> DAT 12-OCT-82/2:30 PM <CR>
```

Use the SEL command to tell DOS/50 to use the emulator and assembler software designed for the 68000:

```
> SEL 68000 <CR>
```

The system responds with the current version number:

```
68000 emulator V n.nn   mm/dd/yy
```

The SEL command automatically sets the emulation mode to 0.

Copy the Demonstration Run Program from the Installation Disk

Enter the following command lines to create an empty directory called DEMO on your system disk, and to make DEMO the current directory. The BR command creates a brief name, ROOT, to mark the old current directory. At the end of this demonstration, you will return to this ROOT directory and delete the DEMO directory and its contents.

```
> BR ROOT /USR <CR>
> CREATE DEMO <CR>
> USER DEMO <CR>
```

Now use the COP command to copy all the files in the DEMO2 directory on the installation disk to the DEMO directory you just created:

```
> COP /VOL/EMU.68000/DEMO2/* * <CR>
```

Remove your installation disk from drive 1 and put it away.

Now list the files you have just copied to the current directory:

```
> L <CR>

FILENAME

ASM
LOAD

Files used       124
Free files       132
Free blocks      813
Bad blocks         0
```

The file named ASM contains the assembly language source code for this demonstration program, and the file named LOAD contains the executable object code. This copy of LOAD is used in the demonstration only if you do not have a 68000 assembler, and thus cannot create your own object file and load file from the source file.

Examine the Demonstration Program

Enter the following command line to display the source file, ASM, on the system terminal:

```
> CON ASM <CR>
        LIST     DBG                   ;Turn on symbolic debug
                                       ;   option.
        SECTION DEMO
        ORG      700H                  ;Begin summing routine.
START   MOVEA.L #TABLE,A0              ;Set the table pointer.
        MOVE.W  #TSIZE-1,D1            ;Set the pass counter.
        CLR.L    D0                    ;Clear the register to be
                                       ;   used for summation.
LOOP    ADD.B    (A0)+,D0              ;Add byte from table to D0.
        DBRA     D1,LOOP               ;Decrement, and branch
                                       ;   if not 5 passes yet.
        MOVE.B  D0,(GEN.L)0F00007H ;Else exit, trigger SVC 1.
        NOP                            ;Two NOPs for SVC.
        NOP                            ;End of summing routine.
;
EXIT    BYTE     1AH                   ;1AH = function code
                                       ;   for the exit SVC.
        ORG      0C0H                  ;Define SRB vector space.
        LONG     EXIT                  ;Define the SRB pointer.
;
TSIZE   EQU      5                     ;Set table size = 5.
        ORG      500H                  ;Put TABLE at 500H.
TABLE   BLOCK    TSIZE                 ;Space for TABLE.
;
        END      START
```

Assemble the Source Code

If you do not have a 68000 assembler on your system disk, you cannot perform this step, so skip the next four commands (ASM, COP, LINK, and L).

The ASM (assemble) command translates assembly language (source code) into binary machine language (object code). The ASM command also creates an assembler listing that correlates the object code with the source code. Enter the following command line to assemble the source code in the file ASM, and to create the listing and object files, ASML and OBJ:

```
     > ASM OBJ ASML ASM <CR>
           ^    ^    ^
           |    |    |
           |    |    +-- source file
           |    |
           |    +------- assembler listing file
           |
           +------------ object file

     ASM 68000 Xnn.nn-nn Copyright (C) 19nn Tektronix, Inc.
     *****Pass 2

     25 Lines Read
     25 Lines Processed
     0 Errors
```

Make sure the printer is turned on and properly connected. Then, copy the assembler listing to the line printer with the following command.

```
     > COP ASML LPT <CR>
```

The fields of the assembler listing are shown in Fig. 7L-15. The entries in the symbol table are also displayed, as shown in Fig. 7L-17. For a detailed explanation of assembler listings, consult your Assembler Users Manual.

```
ASM          68000 SYMBOL TABLE                            Page    2
Xnn.nn-nn  (8550)                                dd-mmm-yy/hh:mm:ss


Scalars

TSIZE---------00000005

Section = DEMO, Aligned to 00000000, Size = 0000071D

EXIT----------0000071C   LOOP----------0000070C   START---------00000700

TABLE---------00000500

Section = %OBJ, Aligned to 00000000, Size = EMPTY


   25 Lines Read
   25 Lines Processed
    0 Errors
```

Fig. 7L-17. Symbol table listing.

Link the Object Code

The linker creates an executable load file from one or more object files. Enter the following linker command to create a load file called LOAD from your object file, OBJ:

```
> LINK -O OBJ -o LOAD -d <CR>
```

The system responds with the version number, listing status, and transfer address:

```
Tektronix Linker Vnn.nn-nn  (8550)
Copyright (C) 19nn Tektronix, Inc.
Listing file not generated
Transfer address:       700
```

If you wish to get a full linker listing written to your system terminal, include the -l f option in the LINK command line. The linker command options -O and -o specify the object file and load file, respectively. The -d command option causes the linker to pass the program symbols from the object file to the load file for use in program debugging.

The files generated by the ASM and LINK commands should now be on your disk. Enter this command to list the files in your current directory:

```
> L <CR>

FILENAME

ASM
LOAD
OBJ
ASML

Files used      126
Free files      130
Free blocks     811
Bad blocks        0
```

Notice that there are now four files listed in your directory. OBJ and ASML were created by the assembler, and LOAD was created by the linker.

Load the Program into Memory

Now it's time to load the object code from the load file LOAD into program memory.

Allocate Memory. If you have the Memory Allocation Controller (MAC) option installed, you need to allocate memory for the program. (If you do not have the MAC option, do not enter the AL command that follows.) The AL command allocates memory space to program memory. The default condition at start-up is zero blocks allocated to program memory. Enter the following command line:

```
     > AL 0 ; AL 0F00000 ; AL 0FFFFFF <CR>
       1 BLOCK(S) ALLOCATED  000000   000FFF
       1 BLOCK(S) ALLOCATED  F00000   F00FFF
       1 BLOCK(S) ALLOCATED  FFF000   FFFFFF
```

These commands allocate 12K bytes (3 blocks) of program memory for the logical addresses used by the demonstration program. The first block is used by the program and the SRB pointer; the second block will contain the SRB (Service Request Block) used by the SVC (Service Call); and the third block is used by the 68000 system stack. For more information on memory allocation and use of the AL command, refer to the Emulation section of your System Users Manual.

Zero Out Memory. Before you load the code, use the F (Fill) command to fill program memory with zeros. Later, when you examine memory, the zeros will make it easy to identify the beginning and end of your code. (Zeroing out memory has no effect on how the program is loaded.) Enter the following command line to fill memory at addresses C0--7FF with zeros:

```
     > F 0C0 7FF 0000 <CR>
```

Check that Memory Was Filled with Zeros. Check the contents of memory with the D (Dump) command. The display shows the data in hexadecimal format, as well as the corresponding ASCII characters. Display the contents of memory addresses C0--CF and 700--7FF with the following commands:

```
> D 0C0 <CR>
       0    2    4    6    8    A    C    E
0000C0 0000 0000 0000 0000 0000 0000 0000 0000   ................

> D 700 7FF <CR>
       0    2    4    6    8    A    C    E
000700 0000 0000 0000 0000 0000 0000 0000 0000   ................
000710 0000 0000 0000 0000 0000 0000 0000 0000   ................
000720 0000 0000 0000 0000 0000 0000 0000 0000   ................
000730 0000 0000 0000 0000 0000 0000 0000 0000   ................
000740 0000 0000 0000 0000 0000 0000 0000 0000   ................
000750 0000 0000 0000 0000 0000 0000 0000 0000   ................
000760 0000 0000 0000 0000 0000 0000 0000 0000   ................
000770 0000 0000 0000 0000 0000 0000 0000 0000   ................
000780 0000 0000 0000 0000 0000 0000 0000 0000   ................
000790 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007A0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007B0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007C0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007D0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007E0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007F0 0000 0000 0000 0000 0000 0000 0000 0000   ................
```

Load the Object Code into Memory. Load the object code for the demonstration program into program memory with the following command:

```
     > LO <LOAD <CR>
          ====
            |
        load file
```

Load the Program Symbols. Recall that the source code for the demonstration program contained the directive, LIST DBG. Because of this directive, the object file contains a list of the symbols that appeared in the source code, and their associated values. And, since you included the -d command when you invoked the linker, these symbols were passed to the load file.

Now, you can use the SYMLO command to load the symbols into the symbol table in 8550 system memory:

```
     > SYMLO -S <LOAD <CR>
```

The -S option means that both address symbols and scalar symbols are loaded. If you omit the -S, only address symbols are loaded. (A scalar is a number that is not an address; for example, TSIZE, the length of the table.) Later in this demonstration, whenever you use a symbol in a command, DOS/50 refers to the symbol table to find the value that the symbol represents.

You have assembled and linked the demonstration program and loaded it into memory. Now skip forward to the heading, "Run the Demonstration Program."

CASE 2: ASSEMBLE ON THE 8560; DOWNLOAD TO THE 8540

This discussion shows you how to create the demonstration program source code and assemble it on the 8560, and then download the object code to 8540 (or 8550) program memory. If your 8560 does not have a 68000 assembler, you cannot do this part of the demonstration, so skip forward to the heading, "Case 4: Patch the Program into Memory", for instructions.

Start Up and Log In

Start up your 8540, make sure that it is in TERM mode, and log in to the 8560 TNIX operating system. Refer to your 8560 System Users Manual for detailed instructions.

Since you're logged in to TNIX, your system prompt is "$". (Later in the demonstration, we show the system prompt as ">", for people using 8540s and 8550s in LOCAL mode.) Every command you enter is processed by TNIX. If you enter an OS/40 command, TNIX passes it to the 8540.

Enter the following commands to select the 68000 assembler on the 8560, and the 68000 emulator on the 8540:

```
     $ uP=68000; export uP <CR>
     $ sel 68000 <CR>
```

The sel command automatically sets the emulation mode to 0.

Create the Demonstration Program

Enter the following TNIX command lines to create an empty directory called demo and to make it the working directory. You'll create your source file and related files in this directory.

```
     $ mkdir demo <CR>
     $ cd demo <CR>
```

Now use the TNIX editor, ed, to create the demonstration program source file. This command line invokes the editor and specifies that you want to create a file called asm:

```
     $ ed asm <CR>
     ?asm
```

The editor responds "?asm" to remind you that asm does not already exist. Notice that the editor does not prompt you when it's ready for input.

Enter the Text. Now enter the editor command a (append text) and type in the program. Use the BACKSPACE key to erase typing mistakes.

```
a <CR>

        column  column
          9       17
          |       |
          v       v
          LIST    DBG                 ;Turn on symbolic debug <CR>
                                      ;   option.  <CR>
          SECTION DEMO <CR>
          ORG     700H                ;Begin summing routine.  <CR>
START     MOVEA.L #TABLE,A0           ;Set the table pointer.  <CR>
          MOVE.W   #TSIZE-1,D1        ;Set the pass counter.  <CR>
          CLR.L    D0                 ;Clear the register to be <CR>
                                      ;   used for summation.  <CR>
LOOP      ADD.B   (A0)+,D0            ;Add byte from table to D0.  <CR>
          DBRA    D1,LOOP             ;Decrement, and branch <CR>
                                      ;   if not 5 passes yet.  <CR>
          MOVE.B  D0,(GEN.L)0F00007H  ;Else exit, trigger SVC 1.  <CR>
          NOP                         ;Two NOPs for SVC.  <CR>
          NOP                         ;End of summing routine.  <CR>
; <CR>
EXIT      BYTE    1AH                 ;1AH = function code <CR>
                                      ;   for the exit SVC.  <CR>
          ORG     0C0H                ;Define SRB vector space.  <CR>
          LONG    EXIT                ;Define the SRB pointer.  <CR>
; <CR>
TSIZE     EQU     5                   ;Set table size = 5.  <CR>
          ORG     500H                ;Put TABLE at 500H.  <CR>
TABLE     BLOCK   TSIZE               ;Space for TABLE.  <CR>
; <CR>
          END     START <CR>
. <CR>
```

At the end of your text, enter a period on a line by itself. The editor will now accept new commands.

Check for Errors. Type the following editor command to display the text you have entered. Check for typing mistakes.

```
1,$p <CR>
| ||
| |+-- print command: displays the lines
| |    in the designated range
| |
| +--- designates last line in file
|
+----- designates first line in file
```

If you made any mistakes, you can correct them now. If you're not familiar with the editor, Table 7L-10 lists the commands you need to add, delete, and replace lines. For more information on the TNIX editor, refer to your 8560 System Users Manual.

Table 7L-10
Basic 8560 Editing Commands

Command	Function
mm,nnp <CR>	Displays lines mm through nn
nn <CR>	Makes line nn the current line
d <CR>	Deletes the current line
a <CR> <line(s) of text> . <CR>	Adds text below the current line
c <CR> <line(s) of text> . <CR>	Replaces the current line with the text you type in

Once your text is correct, enter the w command to write the text to the source file, asm:

```
w <CR>
760
```

The editor responds with the number of characters it wrote to the file.

Finally, enter the q command to quit the editor and return to TNIX:

```
q <CR>
$ <--- TNIX prompt
```

Assemble the Source Code

The TNIX asm (assemble) command translates assembly language (source code) into binary machine language (object code). The asm command also creates an assembler listing that you use to correlate the object code with the source code. Enter the following command line to assemble the source code in the file asm and create the listing and object files asml and obj:

```
$ asm obj asml asm <CR>
      ^   ^    ^
      |   |    |
      |   |    +-- source file
      |   |
      |   +------- assembler listing file
      |
      +----------- object file

ASM 68000 Xnn.nn-nn Copyright (C) 19nn Tektronix, Inc.
*****Pass 2

25 Lines Read
25 Lines Processed
0 Errors
```

Print the assembler listing on the 8560's line printer with the following command:

```
$ lp1r asml <CR>
```

Examine page 1 of your listing. Did the assembler issue any error messages? There should be none. However, if your source code contains errors, take the following steps:

1. Refer to your Assembler Users Manual to see what the error messages mean.

2. Enter the command ed asm to get back into the editor and fix the mistakes in your source code. Exit the editor with the w and q commands, as before.

3. Enter the command asm obj asml asm to re-assemble your source code.

Link the Object Code

The linker creates an executable load file from one or more object files. Enter the following command to create a load file called load from your object file, obj. Be sure to capitalize the parameters exactly as shown.

```
$ link -d -O obj -o load <CR>
```

The system responds with the linker version, listing file status, and transfer address.

```
Tektronix Linker Vnn.nn-nn  (8560)
Copyright (C) 19nn Tektronix, Inc.
Listing file not generated
Transfer address:     700
```

If you wish to get a full linker listing, include the -l f option on the link command line. The linker options -O and -o specify the object file and load file, respectively. The -d command option causes the linker to pass the program symbols from the object file to the load file, for use in program debugging.

The files generated by the asm and link commands should now be in your working directory, demo. Enter the following command to list the files in your working directory:

```
$ ls <CR>
asm
asml
load
obj
```

Notice that there are now four files listed in your directory: obj and asml were created by the assembler, and load was created by the linker.

Download the Program to the 8540

Now it's time to download the object code produced by the 8560's linker into 8540 program memory.

Allocate Memory. If you have the Memory Allocation Controller (MAC) option installed, you need to allocate memory for the program. (If you do not have the MAC option, do not enter the AL command that follows.) The AL command allocates memory space to program memory. The default condition at start-up is zero blocks allocated to program memory. Enter the following command line:

```
> al 0 ; al 0f00000 ; al 0ffffff <CR>
  1 BLOCK(S) ALLOCATED  000000   000FFF
  1 BLOCK(S) ALLOCATED  FFF000   FFFFFF
  1 BLOCK(S) ALLOCATED  F00000   F00FFF
```

This command allocates 12K bytes (3 blocks) of program memory for the logical addresses used by the demonstration program. The first block is used by the program and the pointer to the SRB (Service Request Block); the second block will contain the SRB used by the Service Call; and the third block is used by the 68000 system stack. For more information on memory allocation and use of the AL command, refer to the Emulation section of your System Users Manual.

Zero Out Memory. Before you download any code, use the OS/40 F (Fill) command to fill 8540 program memory with zeros. Later, when you examine memory, the zeros make it easy to identify the beginning and end of your code. (Zeroing out memory has no effect on how the program is loaded.) Enter the following command line to fill memory addresses C0--7FF with zeros:

```
$ f 0c0 7ff 0000 <CR>
```

Check that Memory Was Filled with Zeros. Check the contents of memory with the OS/40 D (Dump) command. The display shows the data in hexadecimal format, as well as the corresponding ASCII characters. Display the contents of memory addresses C0--CF and 700--7FF with the following commands:

```
> d 0c0 <CR>
        0    2    4    6    8    A    C    E
0000C0 0000 0000 0000 0000 0000 0000 0000 0000  ................

> d 700 7ff <CR>
        0    2    4    6    8    A    C    E
000700 0000 0000 0000 0000 0000 0000 0000 0000  ................
000710 0000 0000 0000 0000 0000 0000 0000 0000  ................
000720 0000 0000 0000 0000 0000 0000 0000 0000  ................
000730 0000 0000 0000 0000 0000 0000 0000 0000  ................
000740 0000 0000 0000 0000 0000 0000 0000 0000  ................
000750 0000 0000 0000 0000 0000 0000 0000 0000  ................
000760 0000 0000 0000 0000 0000 0000 0000 0000  ................
000770 0000 0000 0000 0000 0000 0000 0000 0000  ................
000780 0000 0000 0000 0000 0000 0000 0000 0000  ................
000790 0000 0000 0000 0000 0000 0000 0000 0000  ................
0007A0 0000 0000 0000 0000 0000 0000 0000 0000  ................
0007B0 0000 0000 0000 0000 0000 0000 0000 0000  ................
0007C0 0000 0000 0000 0000 0000 0000 0000 0000  ................
0007D0 0000 0000 0000 0000 0000 0000 0000 0000  ................
0007E0 0000 0000 0000 0000 0000 0000 0000 0000  ................
0007F0 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Download the Object Code. Enter the following command line to download the object code from the 8560 file load to 8540 program memory:

```
    $ lo <load <CR>
          ====
           !
       load file
```

Download the Program Symbols. Recall that the source code for the demonstration program contains the directive LIST DBG. Because of this directive, the object file contains a list of the symbols that appear in the source code, and the values associated with those symbols. And, because you included the -d option in the link command line, those symbols were passed to the load file. Use the OS/40 SYMLO command to download those symbols into the symbol table in 8540 system memory:

```
    $ symlo -s <load <CR>
```

The -S option means that both address symbols and scalar symbols are downloaded. If you omit the -S, only address symbols are downloaded. (A scalar is a number that is not an address; for example, TSIZE, the length of the table.)

Later in this demonstration, whenever you use a symbol in an OS/40 command line, OS/40 refers to the symbol table to find the value that the symbol represents.

You've assembled and linked the demonstration program and downloaded it into memory. Now skip forward to the heading, "Run the Demonstration Program."

CASE 3: DOWNLOAD FROM YOUR HOST TO THE 8540

This discussion gives some general instructions for downloading the demonstration program from a host computer other than an 8550 or 8560 to 8540 (or 8550) program memory. If your 8540 is not equipped with the optional COM Interface Package, you cannot complete this part of the demonstration, so skip forward to the heading, "Case 4: Patch the Program into Memory" for instructions. COM Interface software is standard on the 8550.

Since we don't know what host computer you are using, we can only provide a general outline for creating the demonstration program and downloading it to the 8540. Once you have determined the command sequence that is appropriate for your host, record this information in the space provided in Fig. 7L-18.

<u>Create the Extended Tekhex Load Module</u>

<u>Prepare the 8540</u>

<u>Establish Communication</u>

<u>Download the Load Module</u>

<u>Terminate Communication</u>

Fig. 7L-18. Host computer commands for preparing demonstration program.

Create the Extended Tekhex Load Module

In order for the object code to be downloaded to the 8540, it must be in Extended Tekhex format, as shown in Fig. 7L-16. You can create the load module in one of two ways:

1. Use your host computer's text editor, and key the load module in by hand.

2. Use your host computer's 68000 assembler:

   a. Translate the demonstration program into the language of your host's 68000 assembler.

   b. Create and assemble the source file.

   c. Link the object code, if necessary.

   d. Translate the object code produced by the assembler or linker into Extended Tekhex format. The Intersystem Communication section of your System Users Manual provides a general algorithm for conversion to Extended Tekhex format.

Prepare the 8540

Start up your 8540 and enter the following command to select the 68000 emulator:

```
> SEL 68000 <CR>
```

The SEL command automatically sets the emulation mode to 0.

Allocate Memory. If you have the Memory Allocation Controller (MAC) option installed, you need to allocate memory for the program. (If you do not have the MAC option, do not enter the AL command that follows.) The AL command allocates memory space to program memory. The default condition at start-up is zero blocks allocated to program memory. Enter the following command:

```
> AL 0 ; AL 0F00000 ; AL 0FFFFFF <CR>
  1 BLOCK(S) ALLOCATED  000000   000FFF
  1 BLOCK(S) ALLOCATED  FFF000   FFFFFF
  1 BLOCK(S) ALLOCATED  F00000   F00FFF
```

This command allocates 12K bytes (3 blocks) of program memory for the logical addresses used by the demonstration program. The first block is used by the program and the pointer to the SRB (Service Request Block); the second block will contain the SRB used by the SVC (Service Call); and the third block is used by the 68000 system stack. For more information on memory allocation and use of the AL command, refer to the Emulation section of your System Users Manual.

Zero Out Memory. Before you download any code, use the OS/40 F (Fill) command to fill 8540 program memory with zeros. Later, when you examine memory, the zeros make it easy to identify the beginning and end of your code. (Zeroing out memory has no effect on how the program is loaded.) Enter the following command line to fill memory addresses C0--7FF with zeros:

```
     > F 0C0 7FF 0000 <CR>
```

Check that Memory Was Filled with Zeros. Check the contents of memory with the OS/40 D (Dump) command. The display shows the data in hexadecimal format, as well as the corresponding ASCII characters. Display the contents of memory addresses C0--CF and 700--7FF with the following commands:

```
> D 0C0 <CR>
        0    2    4    6    8    A    C    E
0000C0 0000 0000 0000 0000 0000 0000 0000 0000   ................

> D 700 7FF <CR>
        0    2    4    6    8    A    C    E
000700 0000 0000 0000 0000 0000 0000 0000 0000   ................
000710 0000 0000 0000 0000 0000 0000 0000 0000   ................
000720 0000 0000 0000 0000 0000 0000 0000 0000   ................
000730 0000 0000 0000 0000 0000 0000 0000 0000   ................
000740 0000 0000 0000 0000 0000 0000 0000 0000   ................
000750 0000 0000 0000 0000 0000 0000 0000 0000   ................
000760 0000 0000 0000 0000 0000 0000 0000 0000   ................
000770 0000 0000 0000 0000 0000 0000 0000 0000   ................
000780 0000 0000 0000 0000 0000 0000 0000 0000   ................
000790 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007A0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007B0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007C0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007D0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007E0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007F0 0000 0000 0000 0000 0000 0000 0000 0000   ................
```

Download the Load Module to the 8540

Be sure that your 8540 and your host computer are connected via an RS-232-C-compatible communications link. Then perform the following steps to download the Tekhex load module to 8540 program memory. (Refer to the Intersystem Communication section of your System Users Manual to determine the commands and parameters that are appropriate for your host computer.)

a. Enter the 8540 COM command to establish communication. (The parameters of the COM command are host-specific.) Log on to your host and execute any necessary host initialization commands.

b. Enter the command line that downloads the Tekhex load module to the 8540. This command line consists of a host computer command that performs the download, followed by a null character (CTRL-@ on most terminals) and a carriage return. COM places the object code in 8540 program memory, and puts the program symbols into the symbol table in 8540 system memory.

c. Log off your host, then terminate COM command execution by entering the null character, and then pressing the ESC key.

Once you've downloaded the program to the 8540, skip forward to the heading, "Run the Demonstration Program."

CASE 4: PATCH THE PROGRAM INTO MEMORY

This discussion shows you how to patch the demonstration program into 8540 (or 8550) program memory using the P command, and then add the program symbols into the symbol table using the ADDS command.

Ordinarily, you would load the object code and symbols from a binary or hexadecimal load file, as illustrated for Cases 1, 2, and 3. The procedure presented here is not normally used for preparing a program for execution. Use this procedure only if you have no standard means for preparing the program, but would still like to try out your emulator.

Start Up the 8540

Start up your 8540 and enter the following command to select the 68000 emulator:

```
> SEL 68000 <CR>
```

The SEL command automatically sets the emulation mode to 0.

Allocate Memory. If you have the Memory Allocation Controller (MAC) option installed, you need to allocate memory for the program. (If you do not have the MAC option, do not enter the AL command that follows.) The AL command allocates memory space to program memory. The default condition at start-up is zero blocks allocated to program memory. Enter the following command:

```
> AL 0 ; AL 0F00000 ; AL 0FFFFFF <CR>
  1 BLOCK(S) ALLOCATED  000000   000FFF
  1 BLOCK(S) ALLOCATED  FFF000   FFFFFF
  1 BLOCK(S) ALLOCATED  F00000   F00FFF
```

This command allocates 12K bytes (3 blocks) of program memory for the logical addresses used by the demonstration program. The first block is used by the program and the pointer to the SRB (Service Request Block); the second block will contain the SRB used by the SVC (Service Call); and the third block is used by the 68000 system stack. For more information on memory allocation and use of the AL command, refer to the Emulation section of your System Users Manual.

Zero Out Memory. Before you download any code, use the OS/40 F (Fill) command to fill 8540 program memory with zeros. Later, when you examine memory, the zeros make it easy to identify the beginning and end of your code. (Zeroing out memory has no effect on how the program is loaded.) Enter the following command line to fill memory addresses C0--7FF with zeros:

```
> F 0C0 7FF 0000 <CR>
```

Check that Memory Was Filled with Zeros. Check the contents of memory with the OS/40 D (Dump) command. The display shows the data in hexadecimal format, as well as the corresponding ASCII characters. Display the contents of memory addresses C0--CF and 700--7FF with the following commands:

```
> D 0C0 <CR>
       0    2    4    6    8    A    C    E
0000C0 0000 0000 0000 0000 0000 0000 0000 0000   ................

> D 700 7FF <CR>
       0    2    4    6    8    A    C    E
000700 0000 0000 0000 0000 0000 0000 0000 0000   ................
000710 0000 0000 0000 0000 0000 0000 0000 0000   ................
000720 0000 0000 0000 0000 0000 0000 0000 0000   ................
000730 0000 0000 0000 0000 0000 0000 0000 0000   ................
000740 0000 0000 0000 0000 0000 0000 0000 0000   ................
000750 0000 0000 0000 0000 0000 0000 0000 0000   ................
000760 0000 0000 0000 0000 0000 0000 0000 0000   ................
000770 0000 0000 0000 0000 0000 0000 0000 0000   ................
000780 0000 0000 0000 0000 0000 0000 0000 0000   ................
000790 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007A0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007B0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007C0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007D0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007E0 0000 0000 0000 0000 0000 0000 0000 0000   ................
0007F0 0000 0000 0000 0000 0000 0000 0000 0000   ................
```

Patch the Object Code into Memory

The OS/40 P (Patch) command stores a sequence of bytes into memory, replacing the previous memory contents. Enter the following command to store the object code for the first three instructions in the program (MOVEA, MOVE, and CLR) starting at location 700:

```
    > P 700 207C00000500 323C0004 4280 <CR>
        === ============ ======== ====
         |       |           |      |
         |       |           |  CLR.L D0
         |       |           |
         |       |       MOVE.W #TSIZE-1,D1
         |       |
         |   MOVEA.L #TABLE,A0
         |
        patch address
```

Now patch in the next five instructions (ADD, DBRA, MOVE, and two NOPs), and the Exit SVC function code ...

```
    > P 70E D018 51C9FFFC 13C000F00007 4E71 4E71 1A <CR>
```

Finally, patch in the SRB information for the Exit SVC at address C0:

```
    > P 0C0 0000071C <CR>
```

You'll examine the contents of memory later in this demonstration.

Put Symbols into the Symbol Table

Later in this demonstration, you will use symbols from the demonstration program (START, LOOP, TSIZE, TABLE, and EXIT) when communicating with OS/40. Whenever you use a symbol in a command line, OS/40 refers to a symbol table in 8540 system memory to find the values that the symbol stands for. Enter the following command line to add the program symbols to the symbol table, along with their values:

```
    > ADDS START=700 LOOP=70C -S TSIZE=5 TABLE=500 EXIT=71C <CR>
```

The ADDS command cannot provide all the symbol-related information that is provided by the SYMLO command (as in Cases 1 and 2) or the COM command (as in Case 3). Because this information is missing, some of the displays you produce later in this demonstration will not match the symbolic displays shown in this manual. For more information on the ADDS command, refer to the Command Dictionary of your System Users Manual.

You've patched the demonstration program into program memory and placed the program symbols in the symbol table. Now it's time to run the program.

RUN THE DEMONSTRATION PROGRAM

From now until the end of the demonstration, the commands you are to enter are shown in lowercase. If you are not logged in to an 8560, you may enter commands in either lowercase or uppercase. If you are using an 8560, you must enter the name of every command in lowercase, and your system prompt is "$", not ">".

Now that you've loaded the program into memory, you need to:

- verify that the program was loaded correctly; and
- put values into the table in memory, for the program to add.

Check Memory Contents Again. Before you loaded the program, you filled memory locations C0--7FF with zeros. Look again at the memory areas used by the program with the following command lines:

```
> d 0c0 <CR>
       0    2    4    6    8    A    C    E
0000C0 0000 071C 0000 0000 0000 0000 0000 0000   ................

> d 700 71f <CR>
       0    2    4    6    8    A    C    E
000700 207C 0000 0500 323C 0004 4280 D018 51C9   |....2<..B...Q.
000710 FFFC 13C0 00F0 0007 4E71 4E71 1A00 0000   ........NqNq....
```

The object code is loaded in two different blocks:

- The 68000 machine instructions are loaded at address 700 (specified by the first ORG directive).
- Information for the Exit SVC is loaded at address C0 (specified by the second ORG directive).

The contents of the table at address 500 are still undefined, but you will put some values into the table in just a few minutes.

Turn Symbolic Debug On. Enter the following command to turn on symbolic debug. This causes symbols from your code to be displayed when disassembly is performed. (The -S and -L options are already set by default.)

```
> SYMD ON <CR>
```

Disassemble the Object Code. The DI (DIsassemble) command displays memory contents both in hexadecimal notation and in assembly language mnemonics. You can use the DI command to verify that the object code in memory corresponds to your source code. Enter the following command to disassemble the area of memory occupied by the executable part of your program:

```
> di 700 71a <CR>

ADDRESS    DATA  MNEMONIC

START
000700     207C  MOVE.L  #500H,A0

DEMO+000706
000706     323C  MOVE.W  #4H,D1

DEMO+00070A
00070A     4280  CLR.L   D0

LOOP
00070C     D018  ADD.B   (A0)+,D0

DEMO+00070E
00070E     51C9  DBRA    D1,70CH

DEMO+000712
000712     13C0  MOVE.B  D0,F00007H

DEMO+000718
000718     4E71  NOP

DEMO+00071A
00071A     4E71  NOP
```

Compare the DI display with the assembler listing you generated earlier, or refer back to Fig. 7L-15.

The DI display contains two lines for each disassembled instruction. The second line contains the absolute location of the instruction (ADDRESS), the machine language instruction itself (DATA), the instruction mnemonic (MNEMONIC), and the instruction operands. The first line contains symbolic representations for the location. The symbolic location enables you to correlate the display with your assembler listing. The symbols START and LOOP correspond to the labels START and LOOP in the source code.

For those lines of the display where the location does not correspond to a label in the symbol table, DI substitutes the section name plus the address of the instruction relative to the beginning of the section as shown in the location counter field of your assembler listing. (Since section DEMO begins at address 0, the offset is 0, and the relative address is the same as the absolute address in this display. This offset feature is much more useful for sections that don't start at address 0.)

If you didn't load the pertinent symbols and related information into the symbol table (using a command such as SYMLO), the DI command cannot supply this symbolic information.

Now, you've seen that your system can use the symbol table to translate numbers into symbols, to make a display easier to read. Your system can also translate a symbol in a command line into an address. For example, since your system knows that the symbol START is equivalent to the address 700, you could have entered the DI command in any of the following ways:

```
     di 700 71A
     di START 71A
     di start start+1a
     di 700 START+1a
```

Notice that a symbol can be entered in either lowercase or uppercase.

The feature that enables DOS/50 and OS/40 to correlate symbols from your program with the numbers they represent is termed symbolic debug.

Put Values into the Table in Memory. The demonstration program sums five numbers from a table in memory. Use the P (Patch) command to store the numbers 1, 2, 3, 4, and 5 in the table. Do you remember what the address of the table is? It doesn't matter, as long as you remember that the symbol TABLE represents that address.

```
     > p -b table 0102030405 <CR>
            ===== ==========
              |       |
        address of   string of bytes to be stored
        table: 500   at addresses 500--504
```

Check the Contents of the Table. Use the D command with the -B (byte-oriented) parameter to display the contents of the table. (When you don't specify an upper boundary for the area to be dumped, the D command dumps 16 bytes.)

```
          +------ lower address: 500
          |
          |  +-- upper address: omitted
          |  |   (defaults to lower address + OF)
          |  |
        ===== =
  > d -b table  <CR>
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
  000500 01 02 03 04 05 00 00 00 00 00 00 00 00 00 00 00   ................
```

Notice that bytes 500--504 (the table) contain the values you patched in. Bytes 505--50F were zeroed earlier by the F command.

The following command dumps only the contents of the table:

```
     > d -b table table+tsize-1 <CR>
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
     000500 01 02 03 04 05                                   .....
```

Start Program Execution

Enter the G (Go) command to start program execution at location 700, the transfer address specified by the END directive in the source code.

```
    > g <CR>
```

The program executes, and when the Exit SVC occurs, the program breaks (stops). Register D0 contains the sum of the numbers in the memory table: 1+2+3+4+5=0F. You can use the DS command to examine the register contents:

```
    > ds <CR>
    PC=00071C
    D0=0000000F  D1=0000FFFF  D2=00000000  D3=00000000
    D4=00000000  D5=00000000  D6=00000000  D7=00000000
    A0=00000505  A1=00F00000  A2=00000000  A3=00000000
    A4=00000000  A5=00000000  A6=00000000 SSP=00000000 USP=00000000

                 T.S. .III ...X NZVC
    SR=0000 ---> 0.0. .000 ...0 0000
```

MONITOR PROGRAM EXECUTION

You have assembled, loaded, and executed the demonstration program. The rest of this demonstration shows you some commands for monitoring program execution. You can watch the changes in the emulator's registers and observe the effect of each instruction as the program proceeds.

Trace All Instructions. The TRA (TRAce) command lets you observe the changes in the 68000 registers as the program proceeds. When you enter a TRA command and then start execution with the G command, display lines are sent to the system terminal. As each instruction executes, the display line shows the instruction (as in the DI display) and the contents of the registers after that instruction has executed. Enter the following command to trace all of the program's instructions:

```
     > tra all <CR>
```

Enter the command G START (or G 700) to resume program execution back at the beginning of the program:

```
     > g start <CR>
```

As the program executes, the following trace is displayed. Remember that you can type CTRL-S to suspend the display and CTRL-Q to resume the display.

```
     START
     UP:000700  207C  MOVE.L  #500H,A0
         PC=000706
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+000706
     UP:000706  323C  MOVE.W  #4H,D1
         PC=00070A
         D0=0000000F   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+00070A
     UP:00070A  4280  CLR.L   D0
         PC=00070C
         D0=00000000   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0004
```

```
LOOP
UP:00070C  D018  ADD.B    (A0)+,D0
    PC=00070E
    D0=00000001   D1=00000004   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000501   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

DEMO+00070E
UP:00070E  51C9  DBRA     D1,70CH
    PC=00070C
    D0=00000001   D1=00000003   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000501   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

LOOP
UP:00070C  D018  ADD.B    (A0)+,D0
    PC=00070E
    D0=00000003   D1=00000003   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000502   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

DEMO+00070E
UP:00070E  51C9  DBRA     D1,70CH
    PC=00070C
    D0=00000003   D1=00000002   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000502   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

LOOP
UP:00070C  D018  ADD.B    (A0)+,D0
    PC=00070E
    D0=00000006   D1=00000002   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000503   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

DEMO+00070E
UP:00070E  51C9  DBRA     D1,70CH
    PC=00070C
    D0=00000006   D1=00000001   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000503   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000
```

```
     LOOP
     UP:00070C  D018  ADD.B   (A0)+,D0
         PC=00070E
         D0=0000000A   D1=00000001   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000504   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+00070E
     UP:00070E  51C9  DBRA    D1,70CH
         PC=00070C
         D0=0000000A   D1=00000000   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000504   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     LOOP
     UP:00070C  D018  ADD.B   (A0)+,D0
         PC=00070E
         D0=0000000F   D1=00000000   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000505   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+00070E
     UP:00070E  51C9  DBRA    D1,70CH
         PC=000712
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000505   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+000712
     UP:000712  13C0  MOVE.B  D0,F00007H
         PC=000718
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000505   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000
       <BREAK     TRACE, SVC>
```

After register D0 is cleared, it begins to store the sum of the numbers being added. The ADD.B instruction adds a number from the table into D0. At the end of the program, D0 contains the sum of the numbers you put into the table.

Register D1, the pass counter, is set to contain 4 (TSIZE-1) at the beginning of the program. It decreases by one (because of the DBRA instruction) each time a number is added to D0. The program ends after register D1 reaches minus one (FFFF).

Register A0, set to contain 500 (TABLE) at the start of the program, increases by one each time a number is added to the accumulator. At the end of the program, register A0 has been incremented five times and contains 505.

Trace to the Line Printer. By adding the parameter >LPT to a command, you can direct that command's output to the line printer instead of to the system terminal. First verify that your line printer is properly connected and powered up. Then enter the following command to execute the program with trace output directed to the line printer:

```
> g start >LPT <CR>
```

NOTE

If you're operating in TERM mode with an 8560, use one of the following commands in place of the command shown:

- g start ¦ lp1r sends the display to the 8560 line printer.
- g start \>LPT sends the display to the line printer on the 8540 or 8550.

Trace Jump Instructions Only. Another way to monitor the program's execution is to look only at the jump instructions. By tracing the jump instructions, you can still observe the changes in the registers, but you save time and space by not tracing the instructions within the loop. Enter the following command to trace only the jump instructions when the loop is being executed:

```
> tra jmp loop 70E <CR>
           ==== ===
             ¦   ¦
             ¦   +-- upper address } Within this range,
             ¦                     } only jump instructions
             +------ lower address } are traced.
                     (70C)
```

Check the Status of the Trace. The TRA command without any parameters displays the trace conditions that are currently set. Because you can have up to three trace selections in effect at the same time, it is useful to be able to see which selections are active. Check your trace status with the following command line:

```
> tra <CR>
TRACE    ALL,DEMO+000000,FFFFFF
TRACE    JMP,LOOP,DEMO+00070E
```

As you've specified, TRA ALL is in effect for addresses 0--70B, TRA JMP is in effect for addresses 70C--70E, and TRA ALL is again in effect for addresses 70F--FFFFFF.

Again, start your program with the G command. The following trace is displayed:

```
     > g start <CR>

     START
     UP:000700  207C  MOVE.L  #500H,A0
         PC=000706
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+000706
     UP:000706  323C  MOVE.W  #4H,D1
         PC=00070A
         D0=0000000F   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+00070A
     UP:00070A  4280  CLR.L   D0
         PC=00070C
         D0=00000000   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0004

     DEMO+00070E
     UP:00070E  51C9  DBRA    D1,70CH
         PC=00070C
         D0=00000001   D1=00000003   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000501   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+00070E
     UP:00070E  51C9  DBRA    D1,70CH
         PC=00070C
         D0=00000003   D1=00000002   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000502   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+00070E
     UP:00070E  51C9  DBRA    D1,70CH
         PC=00070C
         D0=00000006   D1=00000001   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000503   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000
     DEMO+00070E
     UP:00070E  51C9  DBRA    D1,70CH
         PC=00070C
         D0=0000000A   D1=00000000   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000504   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000

     DEMO+000712
     UP:000712  13C0  MOVE.B  D0,F00007H
         PC=000718
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000505   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000
       <BREAK     TRACE, SVC>
```

As with the TRA ALL display, observe that register D1 (the pass counter) is decremented; register A0 (the table pointer) is incremented; and D0 stores the sum of the numbers from the table. With the TRA JMP selection in effect, the instructions within the loop are not displayed.

Set a Breakpoint after a Specific Instruction. Now that you've seen how the program adds the numbers together, here's a new task: add only the third and fourth numbers from the table. To perform this task, you want the pass counter to contain 1, and the table pointer to contain 502 (the address of the third number in the table). You can accomplish these changes without altering the object code in memory. First, stop program execution after the pass counter and the table pointer have been set. Next, while the program is stopped, enter new values for the pass counter and table pointer. When execution resumes, the program will treat the new values as if they were the original programmed values.

Enter the following command line to trace all of the instructions as the program executes:

```
     > tra all <CR>
```

Check the status of the trace with the following command line:

```
     > tra <CR>
     TRACE    ALL,DEMO+000000,FFFFFF
```

The TRA ALL command just entered makes the previous trace selections obsolete.

Now, set a breakpoint so that the program stops after the table pointer and pass counter have been set. The next command causes the program to stop after the address of the MOVE.W instruction (706) has been seen on the bus. This happens first when the address is prefetched. You may have to use the G command several times to actually execute the instruction where the breakpoint has been set. Be sure to check the last disassembled instruction line to see which instruction was last executed.

```
     > bk 1 706 <CR>
       = ====
       |  |
       |  +-- breakpoint address
       |
       +----- breakpoint number
              (can be 1, 2, or 3)
```

Check the breakpoint setting with the BK command:

```
     > bk <CR>
     BK 1 DEMO+000706
     BK 2  CLR
     BK 3  CLR
```

Use the G command to start program execution:

```
     > g start <CR>

     START
     UP:000700  207C  MOVE.L  #500H,A0
         PC=000706
         D0=0000000F   D1=0000FFFF   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000
       <BREAK     TRACE, BKPT1>
```

All instructions up to and including the instruction last executed are displayed. The break occurred when the emulator detected the breakpoint address going into the 68000 prefetch pipeline. However, by examining the trace, you can see that the instruction on which you want to break has not yet executed. So, enter the G command again.

```
     > g <CR>

     DEMO+000706
     UP:000706  323C  MOVE.W  #4H,D1
         PC=00070A
         D0=0000000F   D1=00000004   D2=00000000   D3=00000000
         D4=00000000   D5=00000000   D6=00000000   D7=00000000
         A0=00000500   A1=00000000   A2=00000000   A3=00000000
         A4=00000000   A5=00000000   A6=00000000
         SSP=00000000  USP=00000000  SR=0000
       <BREAK     TRACE, BKPT1>
```

This time, the trace shows that the MOVE.W instruction was executed. The number of times you must enter the G command before the instruction at the breakpoint address is executed depends on the number of words in the instructions and the TRA mode settings.

Set New Values in Pass Counter and Table Pointer; Check Results. Now that you've reached the breakpoint, you can change the contents of the registers while execution is stopped. The break display shows that register D1 (the pass counter) contains 4, and register A0 (the table pointer) contains the address 500. Use the S (Set) command to set the number of passes to two and set the table pointer to 502:

```
> s d1=1 a0=502 <CR>
```

The S command does not produce a display, but you can use the DS (Display Status) command to check the values in the registers you changed. DS displays the contents of each emulator and status register. Check the result of the previous S command with the following command line:

```
> ds <CR>
PC=00070A
D0=0000000F  D1=00000001  D2=00000000  D3=00000000
D4=00000000  D5=00000000  D6=00000000  D7=00000000
A0=00000502  A1=00F00000  A2=00000000  A3=00000000
A4=00000000  A5=00000000  A6=00000000 SSP=00000000 USP=00000000

             T.S. .III ...X NZVC
SR=0000 ---> 0.0. .000 ...0 0000
```

The DS display shows that the pass counter and table pointer now contain the new values.

Resume Program Execution. If you enter the G command with no parameters, program execution starts where it left off. Resume program execution after the breakpoint with the following command:

```
> g <CR>
DEMO+00070A
UP:00070A  4280   CLR.L    D0
    PC=00070C
    D0=00000000    D1=00000001    D2=00000000    D3=00000000
    D4=00000000    D5=00000000    D6=00000000    D7=00000000
    A0=00000502    A1=00000000    A2=00000000    A3=00000000
    A4=00000000    A5=00000000    A6=00000000
    SSP=00000000   USP=00000000   SR=0004

LOOP
UP:00070C  D018   ADD.B    (A0)+,D0
    PC=00070E
    D0=00000003    D1=00000001    D2=00000000    D3=00000000
    D4=00000000    D5=00000000    D6=00000000    D7=00000000
    A0=00000503    A1=00000000    A2=00000000    A3=00000000
    A4=00000000    A5=00000000    A6=00000000
    SSP=00000000   USP=00000000   SR=0000
```

```
DEMO+00070E
UP:00070E  51C9  DBRA    D1,70CH
    PC=00070C
    D0=00000003   D1=00000000   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000503   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

LOOP
UP:00070C  D018  ADD.B   (A0)+,D0
    PC=00070E
    D0=00000007   D1=00000000   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000504   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

DEMO+00070E
UP:00070E  51C9  DBRA    D1,70CH
    PC=000712
    D0=00000007   D1=0000FFFF   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000504   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000

DEMO+000712
UP:000712  13C0  MOVE.B  D0,F00007H
    PC=000718
    D0=00000007   D1=0000FFFF   D2=00000000   D3=00000000
    D4=00000000   D5=00000000   D6=00000000   D7=00000000
    A0=00000504   A1=00000000   A2=00000000   A3=00000000
    A4=00000000   A5=00000000   A6=00000000
    SSP=00000000  USP=00000000  SR=0000
  <BREAK     TRACE, SVC>
```

Notice that the program performed two passes through the loop, and that the program added the third and fourth numbers in the table: 3+4=7.

SUMMARY OF 68000 EMULATOR DEMONSTRATION RUN

You have assembled, loaded, executed, and monitored the demonstration run program. You have used the following commands:

- SEL---selects the 68000 emulator
- ASM---creates object code from an assembly language program
- LINK---links object code into a load module
- AL---allocates memory when the MAC option is used
- F---fills an area of memory with a specified value
- D---displays memory contents in ASCII and hexadecimal format
- LO---loads object code into memory
- SYMLO---loads program symbols for use in symbolic debug
- DI---disassembles memory contents into assembly language mnemonics
- P---patches a string of bytes into memory
- G---begins or resumes program execution
- TRA---selects instructions to be traced during program execution
- BK---sets a breakpoint
- S---modifies emulator registers
- DS---displays emulator registers

Delete the Demonstration Run Files

Now that you've finished the demonstration run, you can delete the source, object, listing, and load files. If you're using an 8550, the source and load files are still available to you on the 68000 emulator installation disk. If you're using an 8560, remember that once you delete the source file (asm), there is no way of recovering it.

Delete 8550 Files. If your files are on the 8550, use the following procedure to delete them. First use the USER command to move from the DEMO directory back into the directory you were in at the start of the demonstration. Recall that you marked that directory with the brief name /ROOT.

```
> USER /ROOT <CR>
```

Now enter the following command to delete the DEMO directory and the files it contains:

```
> DEL DEMO/* DEMO <CR>
Delete ASM   ?  Y <CR>
Delete LOAD  ?  Y <CR>
Delete OBJ   ?  Y <CR>
Delete ASML  ?  Y <CR>
Delete DEMO  ?  Y <CR>
```

Before deleting each file, DOS/50 asks you whether you really want to delete it. You type "Y" for yes.

Delete 8560 Files. If your files are on the 8560, use the following procedure to delete them. Enter the following command to remove all files in the working directory, including the source file:

```
$ rm * <CR>
```

Now move from the demo directory back into the parent directory and remove the demo directory itself:

```
$ cd ..  <CR>
$ rmdir demo <CR>
```

Turn Off Your System

For instructions on turning off your 8540 or 8550, refer to the Learning Guide of your System Users Manual.

ERROR MESSAGES

The following error messages are specific to the 68000 emulator and to the Memory Allocation Controller (MAC) board.

3E--- Invalid memory space designator. A memory space designator, such as SC:, has been incorrectly entered.

3F--- Illegal use of don't-care expression. A don't-care expression has been used where a unique value is required.

40--- Memory space designator illegal in expr. A memory space designator has been used in a parameter that does not allow memory space designators. For example, in a pair of parameters that represent an address range, only the first may contain a memory space designator.

42--- Invalid use of multiple memory spaces. Multiple memory spaces can only be used with the commands listed under MEMSP in the Command Dictionary.

74--- Program memory jumpered incorrectly. Using the 68000 or Z8001, the SELect command cannot set up the MAC board properly since program memory has been strapped so that addresses do not have a unique location.

88--- Signals cannot occur simultaneously. Using either the TTA, or a Z8001/Z8002, 8086, or 68000 emulator, an attempt has been made to set an event or breakpoint on bus signals that are mutually exclusive (such as a read and a write on the same line).

8F--- User memory declared non-existent. An attempt has been made to access memory which was declared nonexistent with the NOMEM command. Check memory declarations with the MEM and NOMEM commands. If the problem persists after checking your program, check your MAC board.

90--- Invalid arming mode. The -A arming modifier of the BK command needs two programmed breakpoints, but only one is currently programmed. This error occurs only when using an emulator such as the Z8001/Z8002, 8086, or 68000.

E1--- Emulator double fault or odd stack pointer. On the 68000, the emulator has halted during a user job. Possible causes are a double address or bus error, or an odd system stack pointer. Reset the registers and check the program and prototype.

E2--- Processor registers changed. Following a 68000 processor halt, the emulator had to reset the PC, SSP, and SR registers before all the registers were saved.

E6--- No MAC board in system. No Memory Allocation Controller board has been installed.

E7--- System error on MAC board. Unknown system error. Reboot and reselect. If the problem persists, contact your Tektronix service representative.

REPRINTS

The articles reprinted on the following pages contain detailed information about the operation of the 68000 microprocessor. Familiarity with this information will help your understanding of the 68000 emulator.

The three articles are:

- "Microprogrammed Implementation of a Single Chip Microprocessor," by Skip Stritter and Nick Tredennick,
- "Design and Implementation of System Features of the MC68000," by John Zolnowsky and Nick Tredennick, and
- "Instruction Prefetch on the MC68000," by John Zolnowsky.

Further reading can be found in the booklet, MC68000 Article Reprints, by Motorola.

MICROPROGRAMMED IMPLEMENTATION OF A SINGLE CHIP MICROPROCESSOR

Skip Stritter
Nick Tredennick

Motorola Semiconductor Group
Austin, Texas

This paper considers microprogramming as a tool for implementing large scale integration, single-chip microprocessors. Design trade-offs for microprogrammed control are discussed in the context of semiconductor design constraints which limit the size, speed, complexity and pin-out of circuits. Aspects of the control unit of a new generation microprocessor, which has a two level microprogrammed structure, are presented.

Introduction

The field of single-chip, large scale integration (LSI) microprocessors is advancing at an incredible rate. Progress in the underlying semiconductor technology, MOS, is driving the advance. Every two years, circuit densities are improving by a factor of two, circuit speeds are increasing by a factor of two, and at the same time speed-power products are decreasing by a factor of four. Finally, yield enhancement techniques are driving down production costs and hence product prices, thereby increasing demand and opening up new applications and markets.

One effect of this progress in semiconductor technology is advances in LSI microprocessors. The latest generation, currently being introduced by several companies, is an order of magnitude more powerful than the previous generation, the 8-bit microprocessors of three or four years ago. The new microprocessors have 16-bit data paths and arithmetic capability. They directly address multiple-megabyte memories. In terms of functional capability and speed they will out-perform all but the high end models of current 16-bit minicomputers.

As LSI microprocessor technology matures it becomes feasible to apply traditional implementation techniques, that have been proven in large computers, to the design of microcomputers. LSI microprocessor design is now at the stage where better implementation techniques are required in order to control complexity and meet tight design schedules. One such technique, microprogramming, is the subject of this paper. Most of the traditionally claimed benefits of microprogramming, e.g. regularity (to decrease complexity), flexibility (to ease design changes) and reduced design cost, apply to the implementation problems facing today's LSI microprocessor designer.

This paper describes the control structure of one of the new generation, single-chip microprocessors, the MC68000 processor from Motorola, with special attention to the constraints which LSI technology imposes on processor implementation. There are four such constraints: circuit size, circuit speed, interconnection complexity and package pin count. The implications of these constraints on the structure of a microprocessor control unit and its microcode are explored.

LSI Semiconductor Technology

Though progressing quickly, LSI technology still imposes strict constraints on the microprocessor designer.

Circuit Size and Density

There is a fairly constant bound on the size of LSI chip that can be economically produced. Even though circuit densities are improving, at any given time there is a limit on the number of gates that can be put on a chip. The major constraint on an LSI designer is to fit his design into a fixed maximum number of gates.

Circuit Speed

As with circuit density, the LSI designer has a fixed maximum circuit speed with which to work. Speed is limited primarily by the power dissipation limits of the semiconductor package. The problem is compounded by the fact that the processor technology and main memory technology in microprocessor applications are the same. The speed gap between ECL logic and core memory enjoyed by the large computer designer is not available to single-chip microprocessor designers.

Interconnect Complexity

Internal interconnections on an LSI circuit often take as much chip area as the gates they connect. Furthermore, the designer does not have the option of running jumper wires across his circuit when he runs out of surface area. In some cases, it is cheaper to duplicate functions on various sections of the chip than to provide connection to a single centralized function. Another implication of the interconnect problem is that regular structures, such as ROM arrays, can be packed much more tightly than random logic.

Pin-Out

Semiconductor packaging technology limits the number of connections an LSI chip may have to the outside world. Common packages today have 24 or 40 pins; 48 and 64 pin packages are considered large. The pin-out limitation can be overcome by time multiplexing pin use, but the resulting slowdown is usually not acceptable.

Another constraint on LSI designers, not inherent in the technology, is the intensely competitive climate of the semiconductor industry. During the development of any new product it is likely that other companies are working on comparable products. Furthermore, the first product available, of a given type, usually gets the largest share of the market. This situation places LSI designers under tight schedules. Any techniques for reducing product design times can affect product success.

Control Unit Design Tradeoffs

Combinatorial Logic versus Microprogramming

Although previous LSI microprocessor implementations at Motorola have not been microprogrammed, a microprogrammed implementation for MC68000 was considered early in the project. The benefits of microprogramming were convincing enough that once the feasibility of a microcoded implementation was established, the alternative of combinatorial logic implementation was not seriously considered. This is in spite of the fact that the implementers' proven expertise was in combinatorial implementations and they had no experience with microprogramming.

Besides several non-technical reasons for microprogramming (very tight design schedule, limited staff, etc.) there are compelling technical advantages to microprogramming, especially regularity and flexibility.

The design time constraint appears to be eased by microcoding. The regular structure of control store, in contrast with arbitrary control logic, decreases the complexity of the control unit. This in turn decreases the design time. A more complex controller can be implemented at a given design cost. Regularity of the structure simplifies the layout of the chip. Considerable time savings (possibly months) can be realized in the layout step and errors are less likely. Microprogramming allows the processor architects to delay binding some decisions. Once the basic control structure is determined, the circuit designers can go to work, even though the actual microcode may not be written. This reduces the inherent sequentiality of the design process by allowing more overlap of the design efforts of microcoders and circuit designers, and therefore shortens design time.

The regular structure of control store in a microcoded implementation has several other benefits. The regularity decreases the interconnection complexity and therefore the size of the control circuitry. In other words, an array of read only memory cells may take less chip area than the equivalent combinatorial logic. Also, the regularity of structure facilitates detailed simulation and testing.

The MC68000 processor is designed to be enhanced with new instructions in future versions. Microprogramming makes it more likely that such expansion will not involve a major redesign of the chip. The flexibility of microprogramming can also ease the problems of design changes and correction of design errors. Some such changes can be made merely by changing the control store contents with no redesign of the logical circuitry.

Besides regularity and flexibility, microprogramming provides another benefit. The clocking functions in microprogrammed control are much cleaner than those randomly distributed throughout a combinatorial maze with its associated delay, distribution and regeneration requirements. For instance, accurate delay elements are difficult to construct on an integrated circuit, which causes increased tolerances in control signals and slower clocks for combinatorial circuits.

On-Chip versus Off-Chip Control Store

Given the size constraint for LSI chips it would be very attractive to consider off-chip control store for a microprogrammed LSI processor Other constraints make this impractical, however. The pin-out limitation severely limits the width of the control word from off-chip control store. This implies that the control unit microcode must be vertical. This in turn limits the overall speed of the processor since many micro cycles are required in vertical microcode to implement a single macro instruction. The technology speed constraint does not allow brute force solution to this problem by speeding up the internal cycle time. Time multiplexing pins to sequentially access horizontal micro instructions would also slow down the processor.

The LSI-11 from Digital Equipment Corporation uses off-chip control store. The result is a fairly narrow (22 bits) micro instruction. This structure causes a fundamental limitation to the potential speed of the LSI-11, as discussed by Snow and Siewiorek.[7] Because of the above considerations an on-chip control store implementation is preferable.

Horizontal versus Vertical Microcode

The decision to use horizontal or vertical microcode involves conflicting sets of constraints. Horizontal microcode is indicated for several reasons. Vertical microcode is highly encoded and requires a significant amount of combinatorial logic to decode the micro instructions. Horizontal microcode provides fully decoded (or nearly so) fields which can directly drive the execution unit with little intervening logic. Vertical microcode also typically requires more micro cycles to emulate a given macro instruction. Because of the LSI technology circuit speed constraint these extra micro cycles cannot be hidden in each macro cycle by speeding up the internal circuitry. Thus both the interconnect constraint (elimination of random logic) and the speed constraint argue for horizontal microcode.

On the other hand, vertical microcode has advantages, the major one being reduction of the size of the control store. Horizontal micro instructions tend to be very wide and are often duplicated several times in control store.

One solution is to use a two level control store, or hybrid vertical/horizontal structure. This is similar to the two level control proposed by Grasselli.[3,6] In the two level control structure each macro instruction is emulated by a sequence of micro instructions. The micro instructions are narrow, consisting primarily of pointers to nano instructions. (Micro instructions also contain information about branching in the micro sequence.) The nano instructions are wide, providing fairly direct, decoded control of the execution unit. Nano instructions can be placed randomly in the nano store since no sequential accesses to nano instructions are required. Also only one copy of each unique nano instruction need be stored, no matter how many times it is referred to by micro instructions.

The Appendix contains an analytic treatment of two level control. A derivation of the potential savings in control store space, with examples, is given.

An extension of the two level concept is made in the Nanodata QM-1.[5,6] In that machine a micro instruction can specify a sequence of nano instructions. This approach was not taken in the MC68000 for two reasons. First, the initial microprograms showed that micro sequences tend to be very short (one, two, or three micro instructions), so sequential nano instructions cannot be used to advantage. Secondly, unless some facility for nano branches is implemented multiple copies of some nano instructions must be kept in nano store.

The MC68000 Control Unit Implementation

Actual implementation of the general structure derived above involves many other design problems. The remaining sections of the paper discuss the actual implementation chosen and the design considerations involved. Major problems to be solved were minimization of the size of control store, speed-up of the control unit, and reduction of interconnect between the control and execution units. Control store size was minimized by providing a suitable micro instruction branching capability to facilitate sharing subsequences. The control unit speed requirements made micro instruction prefetch necessary so that each nano store access and the subsequent micro store access are overlapped as much as possible.[2] Execution Unit interconnect is minimized by placing the nano store directly above the Execution Unit (with space for some decoding). Fields in the nano store are allocated such that control store output lines are close to the corresponding Execution Unit control points.

The MC68000 control unit supports an instruction set which consists of general single and dual operand instructions involving byte, word (16 bits), or double word operands. Operations are generally memory-to-register, register-to-memory, or register-to-register with some notable exceptions such as the general memory-to-memory move. In addition to standard instructions such as add, compare, and shift; the MC68000 processor is designed to support such instructions as load and store multiple registers, pack (ASCII), multiply and divide, and various forms of bit manipulation.

The MC68000 processor provides eight 32-bit address manipulation registers and eight 32-bit data manipulation registers. Address registers allow 16- and 32-bit operations and data registers allow 8-, 16-, and 32-bit operations. All address and data registers are accessible to the programmer. In addition, there is a program counter with limited user accessibility and there are several registers not available to the user which are used for temporary storage during instruction execution.

The register file is divided into three sections as shown in Fig. 1. Two buses connect all of the words in the register file. The register file sections are either isolated or concatenated using the bi-directional bus switches. This permits general register transfer operations across register sections. A limited arithmetic unit is located in each segment containing address register words and a general capability arithmetic and logical unit is located in the data low word section. This allows address and data calculations to occur simultaneously. For example, it is possible to do a register-to-register word addition concurrently with a program counter increment (the program counter is colocated

with the address register words and carry out from the arithmetic unit low is provided as carry in to the arithmetic unit high).
Special functional units for bit manipulation, packing and unpacking data are located in the data section.

Two factors combined to suggest the desirability of the configuration shown in Figure 1. The first was a very dense two-port static RAM cell which conveniently supported a two-bus structure. The second was the 16-bit data width which made 16-bit segmentation of the registers desirable.

In addition to the configuration of the Execution Unit, other factors contributed to the design of the control unit. The instruction set was specified and considered frozen. The first version assumed that op codes and instruction formats would remain static as defined, though holes were left in the original op code space to allow planned orderly expansion of the available instruction set.

Restriction to fixed instruction formats has several important consequences:

1. Certain fields, such as register designators, can be extracted directly from known positions in the instruction. This tends to reduce control store size and simplify instruction decoding.

2. Register selection and ALU functions tend to remain unchanged for the duration of a single instruction execution. These register designators and ALU functions can be extracted from the instruction (decoded, if necessary) and routed directly to the execution unit, bypassing the control store.

3. The control store need only contain information about when a register is read or written or when the ALU should operate.

Taking advantage of these observations can lead to simplification of the control and reduction of the required micro control store size.

A simplified block diagram of the MC68000 control structure, shown in Figure 2, illustrate an application of the above observations to the controller design. The basic idea is to extract from the macro instruction word all information which is macro instruction static; that is, information which does not depend on timing during the instruction execution for its usefulness. Signals which are not timing dependent bypass the control store and act directly on the Execution Unit.

In a typical microcontrol implementation, the Instruction Decode provides a starting address to the Control Store. The Control Store generates a sequence of control signals, for the Execution Unit, and its own next state information. Branching is accomplished using feedback from the Execution Unit to alter the next state information into the Control Store. At the end of execution of the macro instruction, the Control Store causes loading of the next macro instruction into the Instruction Register and transfers next state control to the Instruction Decode unit.

Traditional implementation of the MC68000 control section using a single Control Store with internal state sequencing information was investigated. It was found to be impractical because the control store was too large for a single chip implementation. Methods for reduction of the total control store area required were considered. It was determined that necessary control store area could be substantially reduced through the use of a two-level control store structure. The structure selected for the MC68000 control unit is shown in Figure 3.

In a two-level structure, the first level (micro control store) contains sequences of control word addresses for the lower level (nano control store). Dynamic operation is illustrated in Figure 4 (bus activity for an indexed address, register to memory add).
The Instruction Decode provides the starting address for a single macro instruction routine. The micro control store provides a sequence of addresses into the nano control store. The nano control store contains an arbitrarily ordered set of unduplicated machine state con-

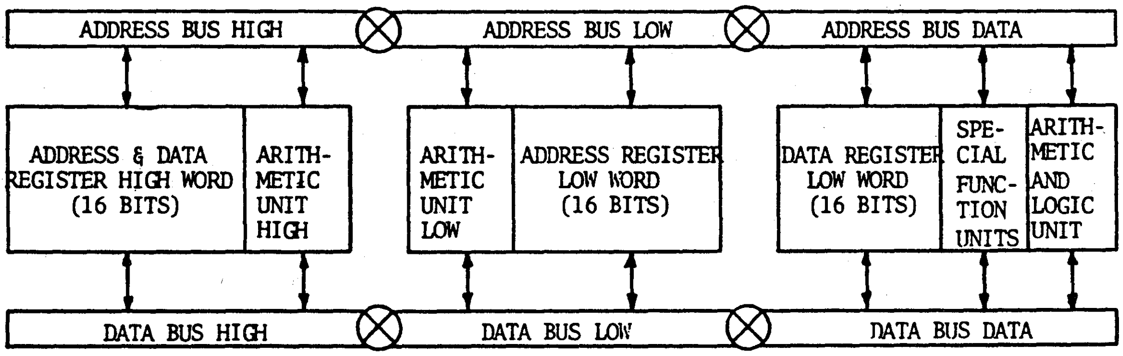


FIGURE 1. MC68000 EXECUTION UNIT GENERAL CONFIGURATION

trol words. The practicality of this structure for space reduction rests on two mutually dependent assumptions.

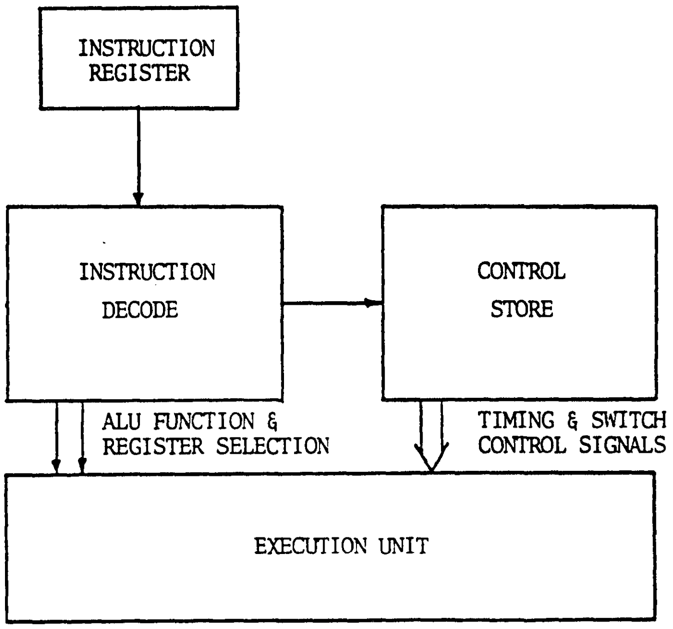


FIGURE 2. SIMPLIFIED BLOCK DIAGRAM OF THE MC68000 CONTROL STRUCTURE

First, the number of different control states actually implemented is a small fraction of the number of possible control states. For example, a reasonably horizontal control word for the MC68000 Execution Unit contains about 70 bits, implying a possible 2^{70} different control words. Most of the possible control states are not meaningful for macro instruction execution. The implementation of the complete set of macro instruction sequences for the MC68000 processor requires only about 200 to 300 ($< 2^9$) unique nano words. This set of nano words is a very small fraction of the set of possible states. Nano words are uniquely specified by no more than nine bits of address. As a result, words in the micro control store address sequences need only allocate nine bits for each nano control store address.

Second, there must be some redundant use of the necessary control words to realize a reduction in control store area. If there were a one-to-one correspondence between nano control store addresses in the micro control store and control words in the nano control store, then the nano address in the micro instruction could be replaced by the contents of the addressed nano instruction and the address bits eliminated. If, however, there are more addresses in the instruction sequences than there are unique control words, a reduction in total control store size may be possible. (For a heuristic derivation of these dependencies and possible advantages, see the appendix.) In the MC68000 control unit, for

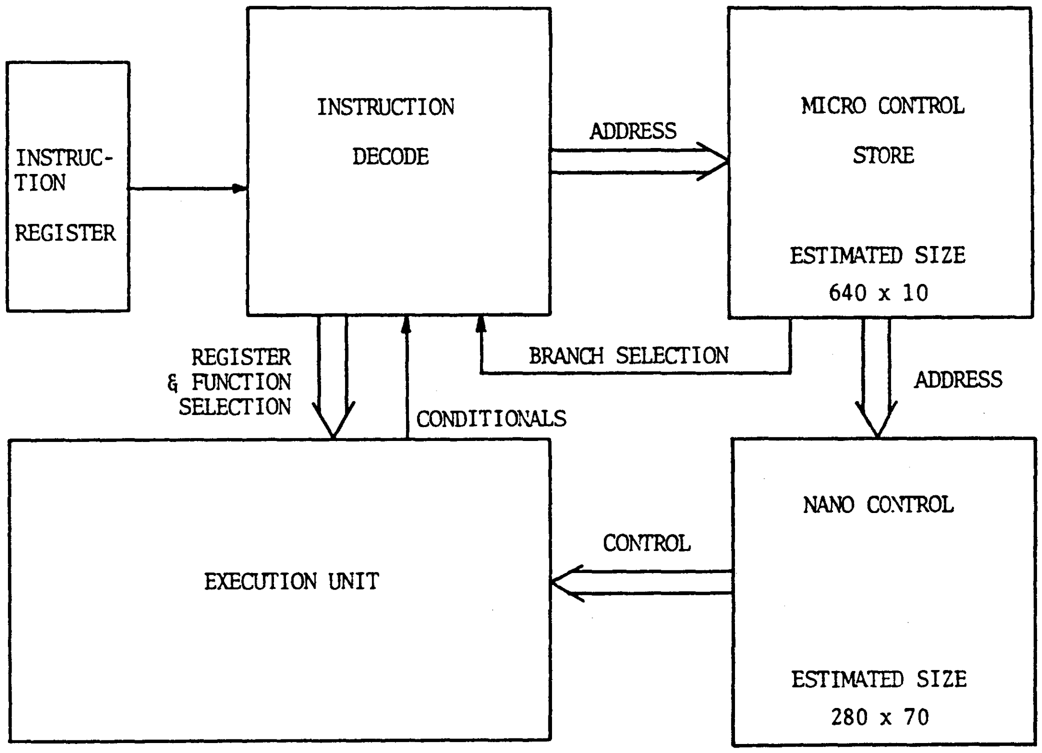


FIGURE 3. MC68000 CONTROL STRUCTURE

example, each different control word is used an average of between two and three times. There are about 650 nano addresses in a complete implementation of micro control store address sequences for the instruction set; yet there are only about 280 different control words used. A single level implementation would have required about 45K control store bits, while the two level structure uses only a little more than half as much.

Another parameter which can have a significant effect on control store size is the extent to which the control word is encoded. In a two level control structure, each control word in the nano control store is uniquely represented by an address in the micro control store. The address in the micro control store could be considered to be a maximal encoding of the control word because there is a one-to-one correspondence between unique control words and unique addresses in the micro control store address sequences. (The nano control store could be viewed as merely an orderly method for translating maximally encoded state information to a significantly more horizontal format.) At the other extreme, one bit could be allocated in the control word for each switch, or control point, that must be driven in the Execution Unit. If the control word is encoded, the decoding logic necessary between the control store output and the Execution Unit must be considered with respect to both timing and space constraints.

In the MC68000 control unit, bits in the nano control store are assigned generally on a functional basis with individual subfields assigned to specific control subfunctions. For example, separate short control fields are assigned to program counter control and arithmetic and logic unit output control.

Within a specific subfield the control bits are encoded into the minimum bits necessary to provide the required subfunction states subject to the constraint that the decode to individual control lines involve no more than approximately two logic levels. Some space is necessary between the nano control store output and the Execution Unit control points for alignment of the control store outputs with their respective control points and to combine certain timing information with appropriate control point variables. Within this space it is possible to provide minimal decoding at very little cost in additional area while the signal encoding in the nano control store saves considerable nano word width and, hence, total storage space required. In the MC68000 control unit the nano word width is approximately 70 bits, while the Execution Unit contains about 180 control points.

The decision to implement the MC68000 control unit using a two level storage structure was based on minimizing control store area. Although necessary control store area was significantly reduced, introduction of the two level concept created several problems.

One problem with a two level control store is that access to a memory is not instantaneous and in a two level structure the accesses must be sequential. The alternative, combinatorial

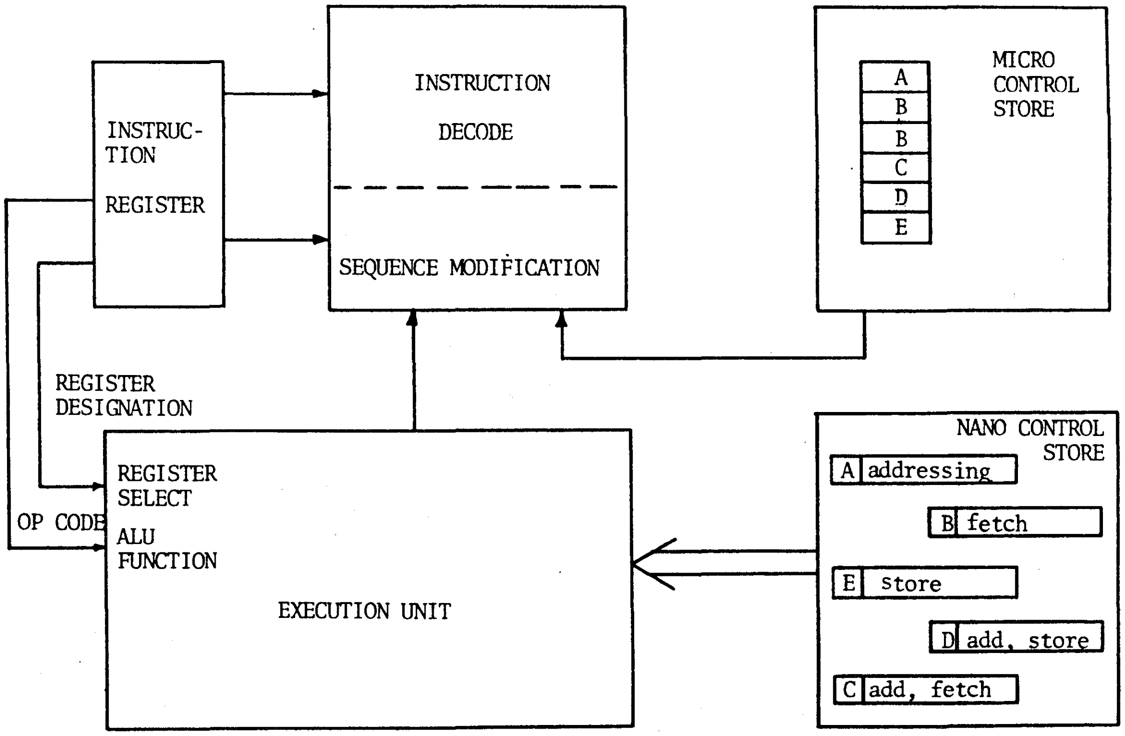


FIGURE 4. MC68000 CONTROL UNIT DYNAMIC OPERATION

decoding, does not proceed in zero time either, which partially compensates for the extra memory access. Further compensation for the extra access time requires complex control timing techniques such as instruction prefetch, access overlap, and multiple word accesses.

Another problem associated with a two level structure is the delay associated with conditional branching. Viewed in a strictly sequential fashion, a condition set in the Execution Unit must affect the selected micro control store address sequence which, in turn, affects the nano word selected. The nano word selected ultimately causes actions which can be dependent upon the value of the tested combination. Techniques used to minimize the sequential nature of this type of delay include physical organization of words within both control store levels and simultaneous access to more than a single control store word. For example, an access to a row of nano store (containing multiple nano words) can be initiated early in a cycle, with subsequent single-word selection based on conditional branch information. Also, in many cases, probable outcome of a conditional branch favors one branch more than another. In such an instance, it may be possible or even desirable to prefetch instruction words associated with the most likely branch condition. The best example of the usefulness of this idea is its application to a decrement and branch-not-zero type instruction. Branching is heavily favored and a prefetch at the destination location can greatly minimize execution delays associated with looping.

If common micro instruction routines, such as the address calculation routines, are to be shared among several macro instructions, then mechanisms must be provided which facilitate functional branches for both entering and leaving the common routines. Care must be taken to avoid delays associated with functional branches in a two level store, especially when the common routines are short (making it more difficult to overlap accesses to different routines in different control stores).

The capability to perform direct branches in the micro control store allows various macro instruction sequences to share common ending routines. It also permits more flexibility in organizing the micro routine sequences within the control store address space.

Branching mechanisms are mentioned here because they occur very commonly. In the MC68000 micro control store, the average micro instruction sequence encounters some type of branch at about one out of every two nano addresses. Implementation of efficient branching mechanisms is critical for providing fast execution times with a microprogrammed structure. The details of the branching mechanisms are, however, very specific to a particular implementation and are, therefore, not relevant beyond the general considerations already presented.

Other Issues

Minimization

Special care has been taken during the microprogramming to detect duplicate nano instructions. Beyond that, in some cases it has been possible to rewrite micro sequences, delaying or anticipating some actions, so as to use previously created nano instructions. For instance, operands are moved into temporary registers early in instruction execution. This tends to make subsequent execution cycles more independent of operand sources and, hence, improves the chance for common nano instructions. Each unique nano instruction that can be eliminated reduces nano store size.

Micro store size is minimized by careful detection of subsequences of micro instructions; for instance, those for address calculations, or storing results.

Simplicity of Structure

Frieder and Miller[2] argue that simplicity of structure makes microprogramming easier. The MC68000 arithmetic and register unit (the Execution Unit) is quite simple: all resources (registers, temporaries and functional units) are tied to each of the two internal buses. This structure simplifies microprogramming, since all transfers of information use the same mechanism.

Generality of Structure

Frieder and Miller[2] also call for generality of structure. This is probably important for general purpose emulation. In the MC68000 processor the architecture is known and fixed. Various non-general assumptions were made, for instance, about macro instruction decoding and the location of register fields in macro instructions. Because of the strict technology constraints, the control structure is optimized for emulation of a single architecture.

User Microprogramming

Current technology requires that large on-chip control store be implemented in ROM, which is much denser than alterable memory. Technology advances will certainly ease this requirement in the near future. Single-chip computers have already been built with small on-chip alterable memories. Clearly, custom microprogramming, user microprogramming and dynamically altered microprograms will be feasible on LSI microprocessors in the future.

Summary

Microprogramming is a viable tool for implementation of LSI microprocessors. The current state of the art in semiconductor technology places certain constraints on the size, speed, interconnect complexity and pin-out of today's integrated circuits. These constraints affect the form of microprogrammed control that can be used. The structure described here: two level, overlapped, hybrid vertical and horizontal microcontrol, implements a new generation microprocessor within these constraints.

Acknowledgements

This paper and the MC68000 microprocessor would have been impossible without the creative guidance of T. Gunter, the administrative guidance of G. Daniels, and the inspirational guidance of C. Crook.

References

1. Dollhoff, T. L., "The Negative Aspects of Microprogramming", Datamation (July, 1964), p. 64-66.
2. Frieder, G. and J. Miller, "An Analysis of Code Density for the Two Level Programmable Control of the Nanodata QM-1", Tenth Annual Workshop on Microprogramming (October, 1975) p. 26-32.
3. Grasselli, A., "The Design of Program-Modifiable Micro-Programmed Control Units", IRE Transactions on Electronic Computers EC-11 no. 6 (June, 1962).
4. Rosin, R. F., "Contemporary Concepts of Microprogramming and Emulation", Computing Surveys 1 no. 4 (December, 1969) p. 197-212.
5. Rosin, R. F., G. Frieder and R. Eckhouse, "An Environment for Research in Microprogramming and Emulation", Comm. ACM 15 no. 8 (August, 1972) p. 748-760.
6. Salisbury, A., Microprogrammable Computer Architectures, American Elsevier, 1976.
7. Snow, E. A. and D. Siewiorek, "Impact of Implementation Design Tradeoffs on Performance: The PDP-11, a Case Study", Dept. of EE and Comp. Sci., Carnegie-Mellon University (July, 1977).

Appendix: Control store size reduction with a two-level control store.

Assume:

n = number of individually-controlled switches in an execution unit (width of the horizontal control word)

k = total number of control states required to implement all instructions

ρ = proportion of unique control states to total number of control states

Single-Level Control Store

In a simplified model of a single-level control store there are k micro instructions, each containing a control state (n bits) and a next micro instruction address ($\lceil \log_2 k \rceil$ bits) See Figure A1.

Total size of single-level control store:

$$S_1 = k\,(n + \lceil \log_2 k \rceil)$$

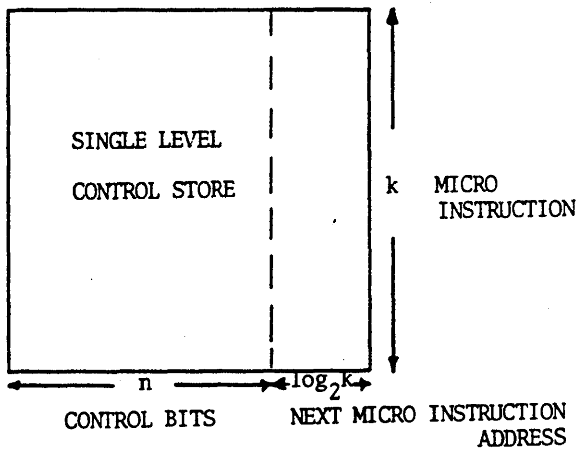


FIGURE A1. MODEL OF A SINGLE LEVEL CONTROL STORE

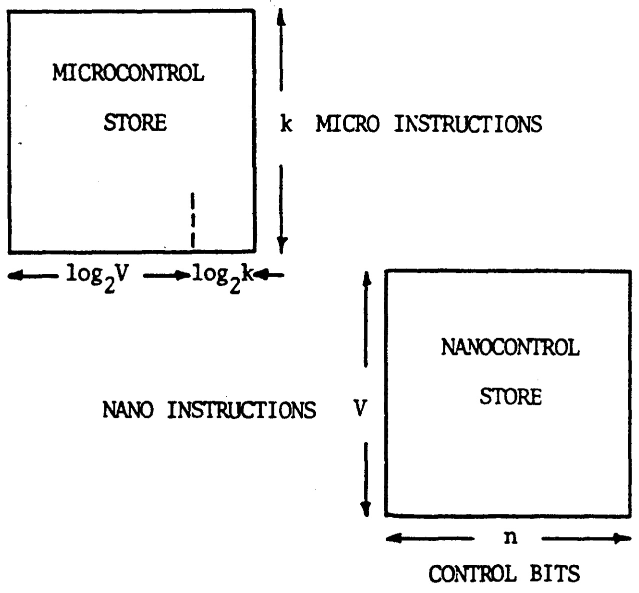


FIGURE A2. MODEL OF A TWO LEVEL CONTROL STORE

Two-Level Control Store

A simplified model of a two-level control store has a micro control store of k micro instructions with a nano address ($\lceil \log_2 v \rceil$bits) and a next micro instruction address ($\lceil \log_2 k \rceil$ bits). The nano control store has v (=ρk) nano instructions, each containing a control state (n bits).

Total size of two-level control store:

$$S_2 = k\,(\lceil \log_2 v \rceil + \lceil \log_2 k \rceil) + nv \qquad (2)$$

$$\text{where } v = \rho k \qquad (3)$$

Control Store Size Comparison

Two-level store requires less control store bits than single control store when:

$$S_2 < S_1$$

using

(1), (2) and (3) gives:

$$k(\lceil \log_2 \rho k \rceil + \lceil \log_2 k \rceil) + n\rho k < k(n + \lceil \log_2 k \rceil) \qquad (4)$$

Simplifying (4) gives:

$$\lceil \log_2 k \rceil + \lceil \log_2 \rho \rceil + n\rho < n \qquad (5)$$

Solving for n and k in (5) gives the result that two-level store is smaller than single-level control store if

$$n > \frac{\lceil \log_2 k \rceil + \lceil \log_2 \rho \rceil}{1-\rho} \qquad (6)$$

or

$$k < \frac{1}{\rho}\; 2^{n(1-\rho)} \qquad (7)$$

Example

In typical microprogrammed machines, n (the width of the horizontal control word) varies from 20 to 360. k varies from 50 to 4000. Typical values for ρ are not known.

In the MC68000 microprocessor

$n \approx 70$

$k \approx 650$

$\rho \approx .4$

$S_1 = k\,(n + \lceil \log_2 k \rceil)$

$= 52400$

$S_2 = k\,(\lceil \log_2 v \rceil + \lceil \log_2 k \rceil) + nv$

$= 30550$

$\frac{S_2}{S_1} = .58$

$\Delta S = S_1 - S_2 = 21850$ bits

Design and Implementation of System Features for the MC68000

John Zolnowsky and Nick Tredennick

MOTOROLA Semiconductor
Austin, Texas

ABSTRACT

The MC68000 combines state-of-the-art technology, advanced circuit design techniques, and computer science to achieve an architecturally advanced 16-bit microprocessor. The processor is implemented by a microprogrammed control of an execution unit. The processor incorporates advanced system features, including multi-level vectored interrupts, privilege states, illegal instruction policing, and bus cycle abort. This paper discusses the implementation of the system features and the influence of the implementation method on the processor design.

1. MC68000 Overview

1.1. Resources

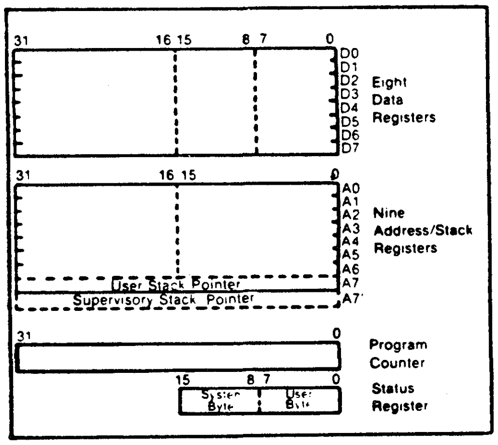


Figure 1: MC68000 Registers

Figure 1 shows the register resources of the MC68000 microprocessor. The first eight registers (D0-D7) are used as data registers for byte (8-bit), word (16-bit), and long (32-bit) data operations. The second set of nine registers (A0-A7, A7') are used as address registers, supporting both software stack operations and base addressing. There is a separate 32-bit program counter and a 16-bit status register.

Figure 2 shows the format of the status register. The trace control (T), supervisor/user (S), and Interrupt Mask (I0-I2) appear in the upper system byte. The condition codes appear in the lower user byte: extend (X), negative (N), zero (Z), overflow (V), and carry (C)

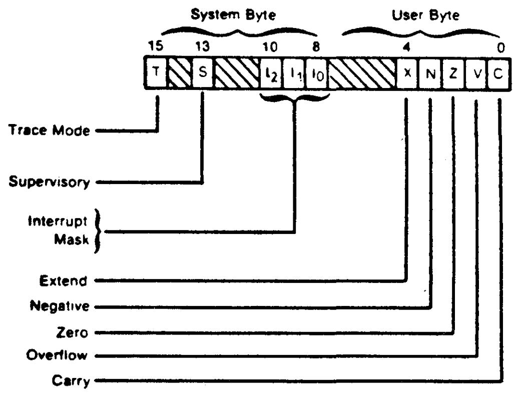


Figure 2: MC68000 status register format

MC68000 memory is organized as 16-bit words, addressable to 8-bit bytes. All address computations are done to 32-bit resolution, but only the low order 24 bits are brought out due to pin count limitations.

1.2. Instructions

The MC68000 supports five basic data types with six basic types of addressing and 56 instruction types. The supported data types are bits, BCD digits, bytes, words, and long words. The basic address types include register direct, register indirect, absolute, immediate, program counter relative, and implied. The register indirect addressing modes include the capability to do post-increment, pre-decrement, displacement, and indexed addressing. Instruction categories include data movement, arithmetic operations (add, sub, multiply, divide), logical operations (and, or, exclusive-or, not), shift and rotate operations, bit manipulation instructions, program control, and system control instructions.

1.3. Structure

To convey an understanding of the relationship between system features as desired (originally specified) and as ultimately supported, it is necessary to first describe the philosophy of the control structure which will provide the background for implementation tradeoffs. The MC68000 uses a microprogrammed control unit which is tightly coupled to the execution unit and the bus interface. (The control structure and execution unit are described in greater detail elsewhere [2].) Tight coupling permits full overlap of fetch, decode, and execute cycles. Overlap of these processing phases has impact on implementation of system features in the MC68000.

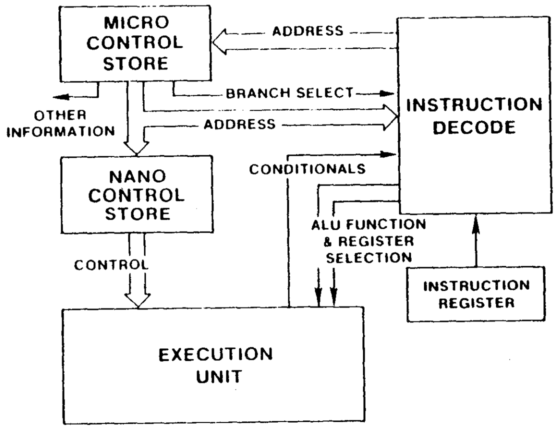


Figure 3: Block diagram of the MC68000 control unit

A basic block diagram of the two level control structure used by the MC68000 is shown in Figure 3. The micro control store contains a set of routines. Each routine is a sequence of micro orders which implements a macro instruction or a portion of a macro instruction (such as an addressing mode). The macro instruction register decode (Instruction Decode) provides a starting address to the micro control store which subsequently provides its own next addresses for a sequence of micro orders which performs operations required by a particular macro instruction. Each micro word contains an address which is used to reference a word in the nano control store. The nano control store contains the set of unique control words which is required to support the entire instruction set. Words in the nano control store are field-encoded such that with two to three levels of decoding they will directly drive control points in the execution unit. To aid in reducing the size of the control unit the MC68000 employs a residual control technique [1]. Information which remains static for the duration of a macro instruction is held in a register (not translated through the control stores) so that space in the control stores is reserved for information which changes from micro cycle to micro cycle.

```
F3 ¦ F4 ¦ F5 ¦             F6 ¦ F7
D2 ¦ D3 ¦ D4 ¦             D5 ¦ D6
E1 ¦ E2 ¦ E3 ¦ E4    E4    E4 ¦ E5
```

Figure 4: Simplified instruction execution sequence

A simplified view, as illustrated by figure 4, assumes that instructions exhibit only fetch, decode, and execute cycles. The boundary between macro instructions is controlled by the execute cycle (which may require several machine cycles to complete). The basic philosophy of the control structure is that fetch, decode, and execute cycles will be overlapped across every macro instruction boundary. This implies that the micro routine for each macro instruction must insure that:

1) The next macro instruction word is accessed with sufficient time to be fully decoded by the end of the current macro instruction.

2) The word following the next macro instruction is fetched by the end of the current macro instruction.

```
Sub
Add
Cmp
```

Figure 5: Simple sequence of instructions.

As an example, assume a simple sequence of single word instructions as shown in Figure 5. It is the responsibility of the micro routine for the subtract instruction to ensure that the add instruction is placed into IR with sufficient time to decode and that a fetch is made to the word following the add instruction. Even if the subtract instruction consists of multiple words (additional words might contain an immediate value, displacement, or an address) the above stated constraints still apply. The micro routines will make as many accesses to the instruction stream as there are words in the definition of the associated macro instruction. The accesses will, however, be skewed forward by two words to provide the necessary overlap.

2. System Features

The MC68000 includes features beyond efficient instruction execution. It also has system features for easier program and memory management, and for handling of exceptional conditions.

2.1. Privilege States

The MC68000 processor operates in one of two states of privilege: the "user" state or the "supervisor" state. The privilege state determines which operations are legal. It is used to choose between the supervisor stack pointer and the user stack pointer in instruction references.

When the processor starts a bus cycle, it classifies the reference via an encoding on the three Function Code pins. This allows external translation of addresses, control of access, and differentiation of special processor states, such as interrupt acknowledge.

Table 1: Classification of References

Function Code	Reference Class
0 0 0	(Reserved)
0 0 1	User Data
0 1 0	User Program
0 1 1	(Reserved)
1 0 0	(Reserved)
1 0 1	Supervisor Data
1 1 0	Supervisor Program
1 1 1	Interrupt Acknowledge

2.1.1. Supervisor/User State

For instruction execution, the supervisor state is determined by the S-bit of the status register; if the S-bit is on, the processor is in supervisor state, otherwise, it is in the user state. The supervisor state is the higher state of privilege. All instructions can be executed in supervisor state. The bus cycles generated by instructions executed in supervisor state are classified as supervisor references. While the processor is in the supervisor privilege state, those instructions which use either the system stack pointer implicitly or address register seven (A7) explicitly access the supervisor stack pointer.

The user state is the lower state of privilege. Most instructions execute the same in user state as in supervisor state. However, some instructions which have important system effects are made "illegal". For example, to insure that a user program cannot enter the privileged state except in a controlled manner, the instructions which modify the entire status register are privileged. The bus cycles generated by an instruction executed in user state are classified as user state references. While the processor is in the user privilege state, those instructions which use either the system stack pointer implicitly, or address register seven (A7) explicitly access the user stack pointer.

2.1.2. Change of Privilege State

Once the processor is in the user state executing instructions, only exception processing (described below) can change the privilege state. During exception processing the current setting of the S-bit of the status register is saved, and the S-bit is forced on, putting the processor in supervisor state. Thus when instruction execution resumes at the address specified to process the exception, the processor is in the supervisor privilege state.

The transition from supervisor to user state can be accomplished by any of four instructions: RTE, MOVE to Status Register, ANDI to Status Register, and EORI to Status Register. The RTE instruction fetches the new status register and program counter from the supervisor stack, loads each into its respective register, and then begins the instruction fetch at the new program counter address in the privilege state determined by the S-bit of the new status register. The MOVE, ANDI, and EORI to Status Register instructions each fetch all operands in the supervisor state, perform the appropriate update to the status register, and then fetch the next instruction at the next sequential program counter address in the privilege state determined by the new S-bit.

2.1.3. Implementation

The creation of privilege states and the subsequent system features affected implementation cost of the processor. Inclusion of separate implicit stack pointers for the user and supervisor states caused increased complexity in the register decoders which had to distinguish between a user and supervisor register for all implicit and explicit references to the system stack pointer (A7). Addition of one 32-bit register to the execution unit was not a major cost factor, but did increase the width of the execution unit. Since the supervisor mode is used to create user environments, instructions were required which allowed manipulation of the user stack pointer while in supervisor mode. This further complicated the register decoders and forced introduction of an additional specialized decoder for explicit control of the user stack pointer while in supervisor mode.

The function code pins are employed to classify processor bus cycles. Two bits in the micro control store classify an access as data space, program space, interrupt acknowledge, or unknown. The unknown state is combined with a value from a special decoder to determine whether the associated access is to data or program space. Unknown states occur in micro routines which may be shared by data space access macro instructions and instruction space access macro instructions. For example, the base plus displacement addressing mode (data space access) and the program counter plus displacement addressing mode (program space access) share the same micro routine.

The user/supervisor function code information is obtained from the status register. User/supervisor information is not provided by the micro control store due to micro control store word width constraints. Its exclusion from the micro control store implies that there must exist some means in the nano control store for direct manipulation of the user/supervisor bit in the status register during exception processing. In addition, there are several privileged instructions which can change the user/supervisor bit. Any prefetches done prior to manipulation of the status register must be discarded. Since the micro routines for manipulation of the user byte of the status register are shared with routines for manipulation of the entire status register, the delay associated with ignoring the prefetches is suffered by both instruction types.

2.2. Exception Processing

2.2.1. Processing States

The processor is always in one of three processing states: normal, exception, or halted. The normal processing state is that associated with instruction execution; the bus cycles are to fetch instructions and operands, and to store results. The STOP instruction is a special case of the normal state in which no further bus cycles are started.

The exception processing state is associated with interrupts, trap instructions, tracing and other exceptional conditions. The exception may be internally generated, by an instruction or by an unusual condition arising during the execution of an instruction. Externally, exception processing can be forced by an interrupt, by a bus error, or by a reset. Exception processing is designed to provide an efficient context switch so that the processor may handle unusual conditions.

Exceptions can be grouped according to their generation. The Group 0 exceptions are Reset, Bus Error, and Address Error. These exceptions cause the instruction currently being executed to be aborted, and the exception processing to commence at the next minor cycle of the processor. The Group 1 exceptions are trace and interrupt, as well as the privilege violations and illegal instructions. These exceptions allow the current instruction to execute to completion, but preempt the execution of the next instruction by forcing exception processing to occur. The Group 2 exceptions occur as part of the normal processing of instructions. The TRAP, TRAPV, CHK, and Zero Divide exceptions are in this group.

Table 2: Exception Groups

Group 0	Reset Bus Error Addr Error	Exception processing begins at the next minor cycle
Group 1	Trace Interrupt Illegal Privilege	Exception processing begins before the next instruction
Group 2	TRAP, TRAPV, CHK, Zero Divide	Exception processing is started by normal instruction execution

The halted processing state is an indication of catastrophic hardware failure. For example, if during the exception processing of a Bus Error another Bus Error occurs, the processor assumes that the system is unusable and halts.

2.2.2. Exception Processing Initiation

The processor hardware recognizes three distinct types of exception conditions: internal exceptions (Group 2), non-catastrophic exceptions (Group 1), and catastrophic exceptions (Group 0). Exception processing for Group 2 exceptions is initiated through normal instruction execution. Group 2 exceptions are detected and processed via micro routines without the aid of specialized additional hardware.

When a Group 1 exception arises, execution of the current macro instruction continues unaffected (including prefetch and decode of the next macro instruction). At the end of the current macro instruction, the micro routine specifies that the next micro control store address is to come from the macro instruction register decoder. However, the existence of a Group 1 exception condition will force substitution of a micro control store address for the appropriate exception processing micro routine.

Occurrence of any Group 0 exception implies that the currently executing micro routine cannot continue; the exception micro routine address preempts the current micro routine at the next minor cycle.

2.2.2.1. Exception Vectors

Exception vectors are memory locations from which the processor fetches the address of a routine which will handle that exception. All exception vectors are two words in length, except for the reset vector, which is four words in length. A vector number is an eight-bit number, which when multiplied by four gives the address of an exception vector. Vector numbers are generated internally or externally, depending on the cause of the exception. The exception vectors are assigned to low addresses in the supervisor data space.

2.2.2.2. Exception Processing Sequence

All exception processing is done in supervisor state, regardless of the setting of the S-bit in the status register. The bus cycles generated during exception processing are classified as supervisor references. All stacking operations during exception processing use the supervisor stack pointer.

Exception processing occurs in four identifiable steps. During the first step, an internal copy is made of the status register. After the copy is made, the special processor state bits in the status register are changed. The S-bit is forced on (1), putting the processor into supervisor privilege state. Also, the T-bit is forced to 0 (off), which will allow the exception handler to execute unhindered by tracing. For the reset and interrupt exceptions, the interrupt priority mask is also updated.

In the second step, the vector number of the exception is determined. For interrupts, the vector number is obtained by a processor fetch, classified as an interrupt acknowledge. For all other exceptions, internal logic provides the vector number. This vector number is then used to generate the address of the exception vector.

The third step is to save the current processor status. Only for the Reset exception is this not done. The current program counter value and the saved copy of the status register are stacked using the supervisor stack pointer. The program counter value stacked usually points to the next unexecuted instruction. Additional information defining the current context is stacked for the Bus Error and Address Error exceptions.

The last step is the same for all exceptions. The new program counter value is fetched from the exception vector. The processor then resumes instruction execution. The instruction at the address given in the exception vector is fetched, and normal instruction decoding and execution is started.

2.2.2.3. Reset

2.2.2.3.1. Description

The Reset pin provides the highest level exception. The processing of the Reset signal is designed for system initiation, and recovery from catastrophic failure. Whatever processing was in progress at the time of the reset is aborted. The processor interrupt priority mask is set at level seven. The vector number is internally generated to reference the reset exception vector at location 0 in the supervisor program space. Because no assumptions can be made about the validity of register contents, in particular the supervisor stack pointer, neither the program counter nor the status register is saved. The address contained in the first two words of the reset exception vector is used to initialize the supervisor stack pointer, and the address in the next two words is used to initialize the program counter. Finally instruction execution is started at the address in the program counter.

2.2.2.3.2. Hardware Support

Hardware support for reset permeates the machine because reset must provide machine initialization from any internal state. Activation of the reset pin preempts all other pending conditions and current activities. Normal operation of the control unit is suspended and the control unit is forced to a state from which it begins executing the reset micro routine. Bits in the nano control store are provided to allow the micro routine to obtain the reset vector address, force the machine into supervisor mode, and set the priority mask (to the level specified by a decoder - in this case, level seven). Additionally, since the register designators for the preempted instruction are unknown, the nano control store must provide bits which can directly specify selection of the implicit stack pointer for its initialization.

2.2.2.4. Interrupts

2.2.2.4.1. Description

The MC68000 provides seven levels of interrupt priorities. Devices may be chained externally within interrupt priorities, allowing an unlimited number of peripheral devices to interrupt the processor. Interrupt priority levels are numbered from one to seven, level seven being the highest priority. The status register contains a three-bit mask which indicates the current processor priority, and interrupts are inhibited for all priority levels less than or equal to the current processor priority.

An interrupt request is made to the processor by encoding the interrupt request level on the interrupt request pins; a zero indicates no interrupt request. Interrupt requests arriving at the processor do not force immediate exception processing, but are made pending. Pending interrupts are detected between instruction executions. If the priority of the pending interrupt is lower than or equal to the current processor priority, execution continues with the next instruction and the interrupt exception processing is postponed. (The recognition of level seven is slightly different, as explained below.)

If the priority of the pending interrupt is greater than the current processor priority, the exception processing sequence is started. First a copy of the status register is saved, and the privilege state is set to supervisor, tracing is suppressed, and the processor priority level is set to the level of the interrupt being acknowledged. The processor fetches the vector number from the interrupting device, classifying the access as an interrupt acknowledge and displaying the level number of the interrupt being acknowledged on the address bus. External logic can respond to the interrupt acknowledge read in one of three ways: put a vector number on the data bus, request automatic vectoring, or indicate that no device is responding (Bus Error). If external logic requests an automatic vectoring, the processor internally generates a vector number which is determined by the interrupt level number. If external logic indicates a Bus Error, the interrupt is taken to be spurious, and the generated vector number references the spurious interrupt vector. The processor then proceeds with the usual exception processing. Normal instruction execution commences in the interrupt handling routine.

Priority level seven is a special case. Level seven interrupts cannot be inhibited by the interrupt priority mask, thus providing a "non-maskable interrupt" capability. An interrupt is generated each time the interrupt request level changes from some lower level to level seven.

2.2.2.4.2. Hardware Support

On-chip logic provides detection and comparison of interrupt requests. Arrival of interrupt requests does not affect execution of the current instruction. If an interrupt of sufficient priority arrives, a pointer to the interrupt micro routine will be substituted for the micro routine pointer from IR decode at the next macro instruction boundary. An interrupt acknowledge is accomplished by the interrupt micro routine via an internal path involving no less than six separate registers. Support for translation and extension of interrupt vector addresses and creation of interrupt auto vector addresses is the responsibility of the field translate hardware in the MC68000. The micro routine uses the address from this special function unit as a pointer to the location of the program counter for the particular interrupt. Vectored, auto vectored, and spurious interrupts are all handled by the same micro routine; the differences occur in vector generation by the field translate unit.

2.2.2.5. Internally Generated Exceptions

2.2.2.5.1. Description

Traps are exceptions caused by instructions. They arise either from processor recognition of abnormal conditions during instruction execution, or from use of instructions whose normal behavior is trapping.

Some instructions are used specifically to generate traps. The TRAP instruction always forces an exception, and is useful for implementing system calls for user programs. The TRAPV and CHK instructions force an exception if the user program detects a runtime error, which may be an arithmetic overflow or a subscript out of bounds. The divide instructions will force an exception if a division operation is attempted with a divisor of zero.

Illegal instruction is the term used to refer to any of the word bit patterns which is not the bit pattern of the first word of a legal MC68000 instruction. Those word patterns with bits [15:12]=1010 or 1111 are distinguished as unimplemented instructions, and separate exception vectors are given to these patterns to permit efficient emulation. If during instruction execution such an illegal instruction is fetched, an illegal instruction exception occurs. This facility allows the operating system to detect program errors, or to emulate unimplemented instructions in software.

In order to provide system security, various instructions are privileged. An attempt to execute one of the privileged instructions while in the user privilege state will cause an exception.

To aid in program development, the MC68000 includes a facility to allow instruction by instruction tracing. In trace state, after each instruction is executed an exception is forced, allowing a debugging program to monitor the execution of the program under test.

The trace facility uses the T-bit in the supervisor portion of the status register. If the T-bit is off (0), tracing is disabled, and instruction execution proceeds from instruction to instruction as normal. If the T-bit is on (1), at the beginning of the execution of an instruction, a trace exception will be generated after the execution of that instruction is completed. If the instruction is not executed, either because an interrupt is taken, or the instruction is illegal or privileged, the trace exception does not occur. If the instruction is executed and an interrupt is pending on completion, the trace exception is processed before the interrupt exception. If the instruction generates an exception, the generated exception is processed before the trace exception.

As an extreme illustration of the above rules, consider the arrival of an interrupt during the execution of a TRAP instruction while tracing is enabled. First the trap exception is processed, then the trace exception, and finally the interrupt exception. Instruction execution resumes in the interrupt handler routine.

2.2.2.5.2. Hardware Support

Trace, privilege violation, illegal instruction, and all instructions which cause a trap are handled in much the same fashion as an auto vectored interrupt by the hardware. They all share (except for some small initial differences) a single micro routine. Again, as with interrupts, the field translate unit provides the vector address for the program counter. The decode of an illegal instruction, or a privileged instruction in user mode causes the macro instruction decode logic to generate a pointer to a special micro routine which returns the machine to supervisor mode and effects a trap. Considerable additional hardware is required to detect these errors and to create the address of the exception vector; and increased control store space is necessary for the special micro routine.

2.2.2.6. Bus Error/Address Error

2.2.2.6.1. Description

Bus Error exceptions occur when external logic requests that a Bus Error be processed by an exception. The current bus cycle which the processor is making is aborted. Whatever processing the processor was doing, instruction or exception, is terminated, and the processor immediately begins exception processing.

Exception processing for Bus Error follows the usual sequence of steps. The status register is copied, the supervisor state is entered, and the trace state is turned off. The vector number is generated to refer to the Bus Error vector. Since the processor was not between instructions when the Bus Error exception request was made, the context of the processor is more detailed. To save more of this context, additional information is saved on the supervisor stack. The program counter and the copy of the status register are of course saved. Besides the usual information, the processor saves the its internal copy of the first word of the instruction being processed, and the address which was being accessed by the aborted bus cycle. Also saved is specific information about the access: whether it was a read or a write, whether the processor was processing an instruction or not, and the classification displayed on the function code pins when the Bus Error occurred. Although this information is not sufficient in general to effect full recovery from the Bus Error, it does allow software diagnosis. Finally, the processor commences instruction processing at the address contained in the Bus Error exception vector.

If a Bus Error occurs during the exception processing for a Bus Error, Address Error, or Reset, the processor is halted, and all processing ceases. This simplifies the detection of catastrophic system failure, since the processor removes itself from the system rather than destroying all memory contents. Only the RESET pin can restart a halted processor.

Address Error exceptions occur when the processor attempts to access a word or a long word operand at an odd address. The effect is much like an internally generated Bus Error, so that the bus cycle is aborted, and the processor begins exception processing. After exception processing commences, the sequence is the same as that for Bus Error, except that the vector number refers to the Address Error vector. Likewise, if an Address Error occurs during the exception processing for a Bus Error, Address Error, or Reset, the processor is halted.

2.2.2.6.2. Hardware Support

During execution of a micro routine the program counter value is often moved to a temporary register where it is manipulated. An updated value of the program counter is returned to the program counter register prior to the end of the micro routine. Macro instructions contain from one to five words and it is not convenient or efficient to attempt to maintain an updated value in the program counter throughout all micro routines. Since occurrence of a Group 0 exception truncates execution of the current micro routine the internal state of the execution unit and, most importantly, the program counter (which is stacked during exception processing) are not well defined for the current implementation. Conditions for processing a Group 1 or a Group 2 exception are such that the program counter is always well-defined.

2.2.3. Multiple Exceptions

2.2.3.1. Description

This section describes the processing which occurs when multiple exceptions arise simultaneously. Group 0 exceptions have highest priority; Group 2 exceptions have lowest priority. Within Group 0, Reset has highest priority, followed by Bus Error and Address Error. Within Group 1, trace has priority over external interrupts, which in turn takes priority over illegal instruction and privilege violation. Since only one instruction can be executed at once, there is no priority relation within Group 2.

The priority relation between two exceptions determines which is taken, or taken first, if the conditions for both arise simultaneously. The description above of the tracing a TRAP instruction when a interrupt arrives is an example of the application of the priority relation. In another example, if a Bus Error occurs during a TRAP instruction, the Bus Error takes precedence, and the TRAP instruction processing is aborted.

2.2.3.2. Hardware Support

It is possible for several exception conditions to be present at once. An exception priority network is used to provide hardware arbitration among multiple exception conditions which can occur. The network keeps track of the arrival and status of exception conditions, forms the micro control store starting address for the highest priority exception condition, and generates the address substitution signal at the appropriate time.

3. Summary and Conclusions

The MC68000 is a register oriented architecture with system features provided by carefully defined privilege states and exception processing. The privilege states divide processing in to user and supervisor modes, with additional protection provided by functional separation of program and data space. Exception processing is defined to divide exception conditions into three logical priority groupings according to the manner in which they are handled by the hardware. In addition, a complete hierarchy is specified for hardware action in processing of multiple exceptions.

The two level microprogrammed control unit which implements the MC68000 architecture accommodates the priority groupings for exception conditions fairly easily. Additional hardware is required to provide support mechanisms associated with privilege states and exception vector generation. A priority encoder and extra logic are required to implement the hierarchical treatment of multiple exception conditions. Additional width in the nano control store words supports the implicit stack pointer references, privilege state changes, and vector address manipulation required for exception processing. Processing for the various exception conditions tends to be fairly homogeneous, so different exception conditions share complete or partial micro routine sequences (via join micro program flow). In the MC68000, clearly defined, well specified system features and a clean, regular control structure minimize the classical conflict between the architecture and the implementation.

References

[1] Alan B. Salisbury, Microprogrammable Computer Architectures, American Elsevier Publishing Company, Inc (1976), pp 47-48.

[2] E. P. Stritter and H. L. Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor", Proceedings of the 11th Annual Microprogramming Workshop, Nov. 1978, pp 8-16.

Instruction Prefetch on the MC68000

John Zolnowsky
Motorola MOS Microprocessor Design

The MC68000 uses a two-word tightly coupled prefetch mechanism to enhance performance. This mechanism is described in terms of the microcode operations involved.

DEFINITION: The execution of an instruction begins when the microroutine for that instruction is entered.

Using this definition, some features of the prefetch mechanism can be described.

1) When execution of an instruction begins, the operation word and the word following have already been fetched. The operation word is in the instruction decoder.

2) In the case of multiword instructions, as each additional word of the instruction is used internally, a fetch is made to the instruction stream to replace it.

3) The last fetch from the instruction stream is made when the operation word is discarded and decoding is started on the next instruction.

4) If the instruction is a single word instruction causing a branch, the second word is not used. But because this word is fetched by the preceding instruction, it is impossible to avoid this superfluous fetch. In the case of an interrupt or trace exception, both words are not used.

5) The program counter points to the last word fetched from the instruction stream.

February 13, 1980

The following example illustrates many of the features of prefetch. The contents of memory are assumed to be as illustrated in Figure 1.

```
                    ORG      0;
                    DATA.L   INITIAL_SSP;
                    DATA.L   BEGIN;

                    ORG      INTVECTOR;
                    DATA.L   INTHANDLR;

                    ORG      PROGRAM;
BEGIN:              NOP      ;
                    BRA      LABEL;
                    ADD      D0 TO D0;
LABEL:              SUB      DISP(A0) FROM A1;
                    CMP      D2 TO D3;
                    SGE      D7;

                    ...

INTHANDLR:          MOVE.W   xxx.L TO yyy.L;
                    NOP      ;
                    SWAP;

                    ...
```

Figure 1: Contents of Memory

The sequence we shall illustrate consists of the power-up reset, the execution of NOP, BRA, SUB, the taking of an interrupt, and the execution of the MOVE.W xxx.L to yyy.L. The order of operations described within each microroutine is not exact, but is intended for illustrative purposes only.

Microroutine	Operation	Location	Operand
Reset	Read	0	SSP High
	Read	2	SSP Low
	Read	4	PC High
	Read	6	PC Low
	Read	(PC)	NOP
	Read	+(PC)	BRA
	<begin NOP>		
NOP	Read	+(PC)	ADD
	<begin BRA>		
BRA	PC=PC+d		
	Read	(PC)	SUB
	Read	+(PC)	DISP
	<begin SUB>		
SUB	Read	+(PC)	CMP
	Read	DISP(A0)	<src>
	Read	+(PC)	SGE
	<begin CMP>	<take INT>	
INTERRUPT	Write	-(SSP)	PC Low
	Write	-(SSP)	PC High
	Read	<INT ACK>	Vector #
	Write	-(SSP)	SR
	Read	(VR)	PC High
	Read	+(VR)	PC Low
	Read	(PC)	MOVE
	Read	+(PC)	xxx High
	<begin MOVE>		
MOVE	Read	+(PC)	xxx Low
	Read	+(PC)	yyy High
	Read	xxx	<src>
	Read	+(PC)	yyy Low
	Read	+(PC)	NOP
	Write	yyy	<dest>
	Read	+(PC)	SWAP
	<begin NOP>		

Figure 2: Instruction Operation Sequence

February 13,1980

MANUAL CHANGE INFORMATION

At Tektronix, we continually strive to keep up with latest electronic developments by adding circuit and component improvements to our instruments as soon as they are developed and tested.

Sometimes, due to printing and shipping requirements, we can't get these changes immediately into printed manuals. Hence, your manual may contain new change information on following pages.

A single change may affect several sections. Since the change information sheets are carried in the manual until all changes are permanently entered, some duplication may occur. If no such change pages appear following this page, your manual is correct as printed.

Tektronix®
COMMITTED TO EXCELLENCE

MANUAL CHANGE INFORMATION

Date: 3-3-82 Change Reference: C1/382

Product: 8500 MDL: 68000 Emulator Specifics Users Manual Part No.: 070-3970-00

DESCRIPTION

TEXT CORRECTIONS

Page 7L-58 Change the second sentence under the heading Insert Your DOS/50 System Disk into Drive 0 to read as follows:

Then enter the CO -A command again:

Page 7L-58 In the first sentence of the **NOTE** change

"NONAME"

to

"NO.NAME"

Page 7L-98 Under error message 3E, change

such as SC:

to

such as SP: