Data General

ECLIPSE® S/120

Assembly Language Programmer's Reference

Warning:

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. As temporarily permitted by regulation it has not been tested for compliance with the limits for Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

ECLIPSE® S/120 Assembly Language Programmer's Reference

Data General

NOTICE

Data General Corporation (DGC) has prepared this document for use by DGC personnel, customers, and prospective customers. The information contained herein shall not be reproduced in whole or in part without DGC's prior written approval.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

DASHER, DATAPREP, ECLIPSE, ENTERPRISE, INFOS, microNOVA, NOVA, PROXI, SUPERNOVA, ECLIPSE MV/8000, TRENDVIEW, MANAP, and **PRESENT** are U.S. registered trademarks of Data General Corporation, and **AZ-TEXT, DG/L, ECLIPSE MV/6000, REV-UP, SWAT, XODIAC, GENAP, DEFINE, CEO, SLATE, microECLIPSE, BusiPEN, BusiGEN,** and **BusiTEXT** are U.S. trademarks of Data General Corporation.

Ordering No. 014-000686-00

Printed in the United States of America
Rev. 00, June 1982

Preface

The *ECLIPSE S/120 Assembly Language Programmer's Reference* is intended for use by programmers and system engineers.

Organization of This Manual

The organization of each chapter and appendix of this manual follows:

Chapter 1, "System Overview," summarizes each chapter as it describes the system block diagram. It introduces many of the system's capabilities and terms that are discussed in later chapters.

Chapter 2, "Addressing," illustrates how the S/120 computer uses information in registers and instructions to form addresses. These addresses can reference data and other instructions.

Chapter 3, "Fixed-Point Instructions," describes the fixed-point data type format and summarizes fixed-point operation. A table lists the fixed-point instructions.

Chapter 4, "Floating-Point Instructions," describes the floating-point data type format and summarizes floating-point operation. A table lists the floating-point instructions.

Chapter 5, "Stack Management," discusses stack operation, the instructions that are used to manipulate stacks, and the use of stacks in fault handling.

Chapter 6, "Program Flow Management," discusses program flow alterations (including jump and conditional skip) and program interrupts. Program flow alteration instructions are listed in a table.

Chapter 7, "Device Management," describes the two types of input/output (I/O) used on the S/120 computer: programmed I/O and data channel I/O. It discusses program interrupts, including the vectored interrupt, and also briefly describes the programmable interval timer, the real-time clock, and the asynchronous interface. This chapter also includes an I/O instruction dictionary. These instruction entries are organized by function, then listed alphabetically by mnemonic.

Chapter 8, "Memory Allocation and Protection," describes the memory allocation and protection unit (MAP) and lists the MAP instructions.

Chapter 9, "Virtual Console," describes command formats and functions used in the power-up self-test, in auto program load, in program debugging, and in changing the MAP unit or MAP status.

Chapter 10, "Instruction Dictionary," contains a detailed explanation of all instructions supported by the S/120 computer. The instruction entries are listed alphabetically by mnemonic.

Appendix A, "Standard I/O Device Codes," is a table of standard Data General device codes.

Appendix B, "Programming Aids" explains octal and hexadecimal conversion with a table for conversion to and from decimal numbers, and ASCII character codes, with a table of the ASCII character set and the related decimal, octal, and hexadecimal codes.

Appendix C, "Instruction Execution Times," lists typical execution times for the instructions supported by the S/120 computer.

Appendix D, "Programming Examples," explains shorthand methods to attain certain programming results.

Appendix E, "Compatibility with ECLIPSE Computers," discusses instruction differences between the S/120 and other ECLIPSE computers.

Appendix F, "Instruction Summary," presents a short description of each instruction for quick reference.

Typesetting Conventions

Since abbreviated instruction mnemonics include a specified device flag condition, the standard form of the instruction must be used if different conditions are desired. For example, to alter the state of the Interrupt On flag while performing a *Mask Out* instruction, you must use the full mnemonic,

DOB*[f]* *ac,***CPU**

instead of the abbreviated mnemonic (**MSKO** *ac*), in order to set bits 8 and 9 to 00.

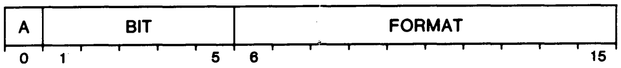
The conventions we use for instruction formats are as follows:

COMMAND Upper-case bold indicates mnemonics for commands and associated symbols that must be coded exactly as shown.

argument Lower-case italics indicate required arguments or operands for which numbers or symbols must be substituted.

[option] Lower-case italics within brackets indicate optional arguments or operands. The brackets are not coded; they merely set off the choice.

In addition, we use the following special symbols:



This diagram shows the arrangement of the 16 bits in a word. The diagram is always divided into 16 boxes, numbered 0 to 15.

Related Manuals

Other manuals describing aspects of the S/120 system are:

- ECLIPSE® *S/120 Computer System, Hardware Reference* (DGC No. 014-000690) gives a functional description of the S/120 computer system including configuration, interfacing, and theory of operation.
- ECLIPSE® *S/120 Diagnostic and Maintenance Manual* (DGC No. 015-000116) includes an introduction to the virtual console, a checklist for use with installation data sheets, and replacement procedures.
- *Introduction to Real-Time* ECLIPSE® *Computers* (DGC No. 014-000687) describes in summary form the entire line of DG ECLIPSE hardware, both processors and peripherals. It also contains a complete bibliography of DG ECLIPSE documentation and references to introductory software books.
- ECLIPSE® *Line Computers Instruction Reference Card* (DGC No. 014-000627)
- *MP/AOS Concepts and Facilities* (DGC No. 069-400200) gives an overview of the concepts and facilities of MP/AOS; describes the MP/AOS operating system, utilities, and programming languages supported by MP/AOS. For more detailed information on MP/AOS, consult the related manuals listed in *MP/AOS Concepts and Facilities.*
- *Introduction to RDOS* (DGC No. 069-400011) familiarizes beginning RDOS users with a conceptual presentation of RDOS, as the basis for further reading in other books. Also provides a map to other RDOS documentation.
- *AOS Software Documentation Guide* (DGC No. 069-000020) contains a bibliography of all the AOS software manuals, including a short description of each.

Table of Contents

Preface

Organization of This Manual **i**
Typesetting Conventions **ii**
Related Manuals **ii**

Chapter 1
System Overview

Addressing Modes **1**
Central Processing Unit **2**
System I/O Controller **2**
Memory **2**
Memory Allocation and Protection **2**
Error Checking and Correction **2**
Virtual Console **3**

Chapter 2
Addressing

Addressing Conventions **5**
Definitions **5**
Addressing Modes **6**
Absolute Addressing **6**
PC Relative Addressing **7**
Accumulator Relative Addressing **7**
Direct and Indirect Addressing **7**
Effective Address Calculation **8**
Byte Addressing **8**
Bit Addressing **9**
Reserved Memory Locations **10**

Chapter 3
Fixed-Point Instructions

Data Format **11**
Unsigned Integers **11**
Signed Integers **11**
Fixed-Point Operation **12**
ALC Format **12**
ALC Instructions **12**
ALC Instruction Execution **13**
Fixed-Point Instruction Lists **16**
Arithmetic Instructions **16**
Logical Operation Instructions **16**
Decimal Arithmetic Instructions **17**
Data Movement Instructions **17**
Byte Manipulation Instructions **18**
Bit Manipulation Instructions **18**

Chapter 4
Floating-Point Instructions

Data Format **19**
Sign **19**
Mantissa **20**
Exponent **20**
Floating-Point Operation **20**
Floating-Point Registers **20**
Guard Digits **20**
Floating-Point Status Register **20**
Floating-Point Faults **21**
Floating-Point Instruction Lists **21**
Arithmetic Instructions **22**
Floating Point Operation **22**
Data Movement Instructions **22**
Program Flow Alteration Instructions **22**
Number Conversion Instructions **23**
Status Register Instructions **24**

Chapter 5
Stack Management

Stack Operation 25
Return Block 25
Stack Pointer 25
Frame Pointer 25
Stack Limit 26
Stack Control Memory Locations 26
Stack Overflow Protection 26
Overflow Checking 26
Stack Fault Handler 27
Interrupting the Stack Instructions 27
Stack Instructions 27
Examples — Stacks 29

Chapter 6
Program Flow Management

Program Flow 31
Program Flow Alteration 31
Program Flow Alteration Instructions 32
Conditional Skip Instructions 32
Program Flow Interruption 32
Extended Operation 33
Extended Operation Instructions 33

Chapter 7
Device Management

I/O Management 35
Programmed I/O 35
Data Channel I/O 35
I/O Operations 35
I/O Format 36
Program Interrupt 37
Initiating an Interrupt and CPU Response 37
Servicing an Interrupt 37
Identifying the Interrupting Device 38
Priority Interrupts 39
Dismissing An Interrupt 39
SPU I/O Management 40
Special Mnemonics 40
Asynchronous Interface 40
Real-Time Clock Interface 42
Programmable Interval Timer 43
SPU I/O Operation 44
Power-Up Sequence 44
Auto Program Load Register 44
Power Fail/Autorestart 45
Error Checking and Correction 45
ERCC Instructions 45
Power Fail Instructions 46
Programming Example 46
I/O Instruction Dictionary 47
Standard I/O Instructions 47
CPU Device Instructions 49
Programmable Interval Timer 52
Real-Time Clock 53
Asynchronous Line Input 53
Asynchronous Interface 54
Asynchronous Line Output 54
ERCC Error Correction 55

Chapter 8
Memory Allocation and Protection

MAP Functions 57
Definitions 57
Address Translation 57
Sharing Physical Memory 57
Mapped Mode 57
User Maps 57
Data Channel MAPS 58
Unmapped Mode 59
ECLIPSE S/120 Address Translation 59
Emulator Trap 59
MAP Protection Capabilities 59
Validity Protection 59
Write Protection 59
Indirect Protection 59
I/O Protection 59
Map Protection Faults 60
Load Effective Address Mode 60
Initial Conditions 60
MAP Instructions 60

Chapter 9
Virtual Console

Entering the Virtual Console 67
Errors 67
The Rubout/Delete Key 68
The K Command 68
Virtual Console Errors 68
Commands 68
Cell Commands 69
Function Commands 70
Miscellaneous Commands 72

Chapter 10
Instruction Dictionary

Coding Aids 75
Setting the Index Field 75
ERCC Error Correction 168
Common Process 162
Mode A 162
Modes B Through E — Part I 162
Modes B through E — Part II 164

Appendix A
I/O Device Codes

Appendix B
Programming Aids

Octal and Hexadecimal Conversion 171
ASCII Character Codes 172

Appendix C
Instruction Execution Times

Appendix D
Programming Examples

Arithmetic Tests 177
Subroutines 178
Example 180

Appendix E
Compatibility With ECLIPSE Line Computers

Appendix F
Instruction Summary

Index
DG Offices
How to Order Technical Publications
Technical Products Publications Comment Form
Users' Group Membership Form

Chapter 1

System Overview

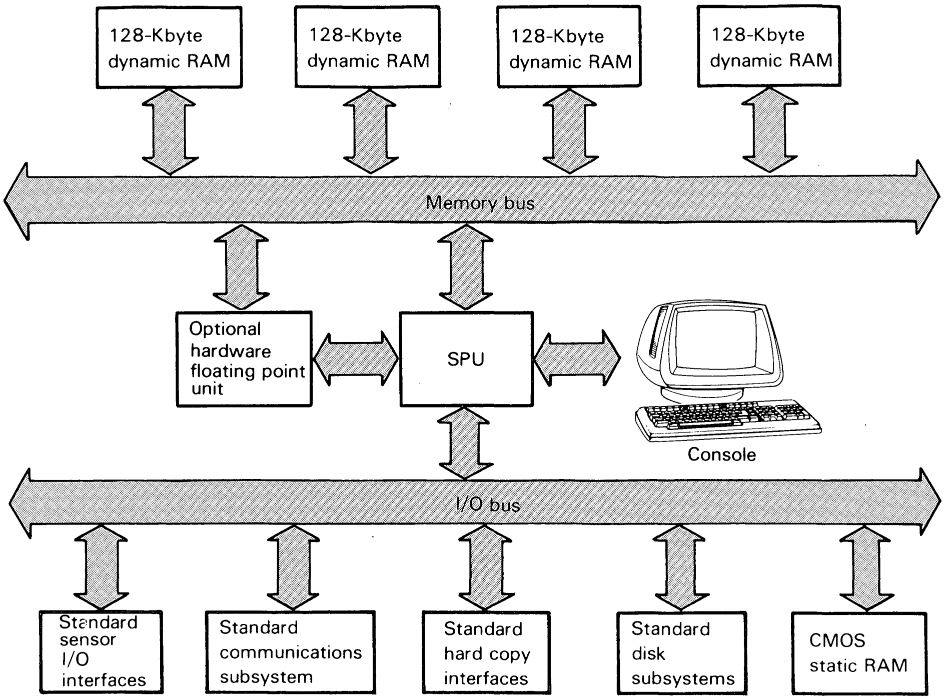


Figure 1.1 ECLIPSE S/120 computer system block diagram

The S/120 microcomputer system contains a variety of powerful standard facilities including:

- Full 16-bit ECLIPSE architecture,
- Byte and bit addressing,
- Decimal add and subtract instructions,
- Fixed-point multiply/divide instructions,
- Floating-point instructions,
- Data manipulation,
- Stack manipulation,
- Vectored interrupt,
- Data channel input/output,
- Error checking and correction
- Virtual console.

Addressing Modes

The S/120 system features direct or indirect addressing which can be done in the following modes:

Absolute: the intermediate address is the unmodified displacement.

Program Counter Relative: the intermediate address is found by adding the displacement to the address of the word containing the displacement.

Accumulator Relative: the intermediate address is found by adding the displacement to the contents of a specified accumulator (AC2 or AC3).

Refer to Chapter 2, "Addressing," for more information.

Central Processing Unit

Refer to the system block diagram in Figure 1.1. The central processing unit (CPU) together with the external microcode controllers (XMCs) implement the ECLIPSE instruction set. The instruction groups include:

- Decimal add and subtract. See Chapter 3, "Fixed-Point Instructions."
- Signed and unsigned fixed-point including efficient multiply/divide instructions. Execution times for multiply instructions average 9.5 microseconds; execution times for divide instructions range from 13 to 20.5 microseconds. See Chapter 3, "Fixed-Point Instructions."
- Data manipulation — efficient handling of bits, bytes, and character strings. See Chapter 3, "Fixed-Point Instructions."
- Floating point — normalized single-precision and double-precision floating-point arithmetic. See Chapter 4, "Floating-Point Instructions."
- Stack manipulation — extensive stack support instructions including save, return, push, and pop. See Chapter 5, "Stack Management."
- 16-level, programmed priority interrupts including vectored interrupts. See Chapter 6, "Program Flow Management" and Chapter 7, "Device Management."

System I/O Controller

The system input/output (SIO) controller monitors the system bus for I/O instructions from the CPU to devices that are internal to the SIO controller as well as to devices that are connected to the ECLIPSE I/O bus.

Devices internal to the SIO controller include the programmable interval timer (PIT), the real-time clock (RTC), the asynchronous interface (TTI/TTO), and the power monitor.

The S/120 computer supports all ECLIPSE peripherals. The data channel facility allows devices to transfer data to and from fast memory over the ECLIPSE I/O bus at speeds of 2.0 megabytes per second for input and 1.3 megabytes per second for output.

Refer to Chapter 7, "Device Management," for more information.

Memory

Random access memory (RAM) is a local storage medium whose contents can be read or modified one word at a time. The S/120 computer uses dynamic RAM of 128 Kbytes per memory board. The S/120 computer system contains up to 512 Kbytes of read/write memory.

Memory Allocation and Protection

The memory allocation and protection (MAP) feature performs logical-to-physical address translation between the CPU and memory. In addition to translating addresses, the MAP feature also performs the various protection functions. These functions are validity protection, write protection, indirect protection, and I/O protection.

- Validity protection protects one user's memory space from inadvertent access by another user.
- Write protection allows users to read the protected memory locations, but not to write into them.
- Indirect protection allows the supervisor to ensure that the CPU will not be placed in an indirection loop.
- I/O protection protects the I/O devices in the system from unauthorized access.

The emulator trap feature is available when the MAP is on. See Chapter 8, "Memory Allocation and Protection," for further information on these MAP features.

Error Checking and Correction

The S/120 error checking and correction (ERCC) facility generates and appends a 6-bit check code to each word (2 bytes) of data written to memory. In addition, the CPU checks this 6-bit code for each word read from memory. Detection of a single-bit error will cause the erroneous bit to be corrected.

Double-bit and some triple-bit errors are detected but not corrected. However, their fault addresses and error syndrome codes are recorded, and an interrupt (when enabled) is issued.

The S/120 also implements an advanced error checking and correction feature *sniffing* that continuously tests all on-board memory at the rate of one location per 16 microseconds. This results in a complete check of the S/120 memory every four seconds and minimizes the accumulation of correctable single-bit errors into multiple bit errors.

These ERCC instructions—Enable ERCC, Read Memory Fault Address, and Read Memory Fault code—are described in the "I/O Instruction Dictionary," in Chapter 7.

Virtual Console

The virtual console allows the programmer to inspect and modify the system's state. It also aids program debugging. The virtual console allows the user to:

- Stop, start, and continue program execution;
- Examine and alter CPU registers and memory locations;
- Initiate program load sequences;
- Change user maps or MAP status.

For more information, refer to Chapter 9, "Virtual Console."

Chapter 2

Addressing

Addressing Conventions

Each of the 32 K logical locations in main memory contains a 16-bit word. You use a 15-bit logical address to specify a memory location.

The maximum amount of logical address space available to the programmer is 32,768 words. The physical address space, the amount of memory in the system, may be up to 1 Mbyte. In the logical address space, the next sequential memory location after 77777_8 is location 0.

The memory allocation and protection unit (MAP) controls the relationship between a logical address space and the physical address space. When the MAP is enabled, it intercepts each memory reference and translates the 15-bit logical address into a 20-bit physical address. Unless the MAP itself is being programmed, the translation process is invisible to the programmer. For more information on the MAP, see Chapter 8, "Memory Allocation and Protection."

Definitions

The following definitions introduce some of the concepts of word addressing.

Addressing modes — four methods of addressing that use a displacement from some reference point to find the desired address. These are absolute, program counter, AC2 relative, and AC3 relative. Different modes use different reference points.

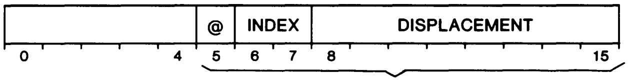
Indirect addressing — a method of addressing that uses the contents of one address as a pointer to another address; in turn, this second address may be used as a pointer to yet another address. A series of indirect addresses is called an indirection chain.

Index bits — bits in the instruction that specify the reference point used to generate an address.

Effective address calculation — the process of converting the index, indirect, and displacement bits (defined below) into an address to be used by the instruction. This process generates a logical address.

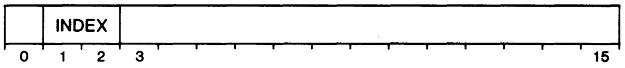
Intermediate address — the address obtained by the effective address calculation before testing for indirection.

Short address field — 11 bits in the instruction which define the intermediate address. Bit 5 is the *indirect bit*, bits 6 and 7 are the *index bits*, and bits 8 through 15 are the *displacement bits*.

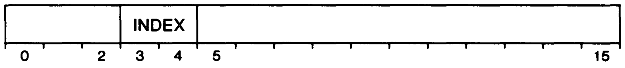


short address field

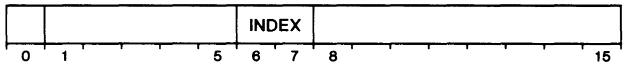
Extended address field — 2 index bits in the instruction, plus 16 bits of the next sequential word, that define the intermediate address. Depending on the instruction, either bits 1 and 2, bits 3 and 4, or bits 6 and 7 are the index bits. In the word following the instruction, bit 0 is the indirect bit and bits 1 through 15 are the displacement bits. Extended address instructions are 2-word instructions.

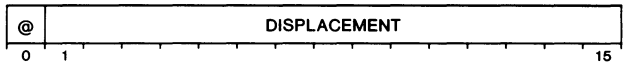


or



or





Indirect bit — one bit of the instruction or address that controls the indirection chain at each step of the addressing process.

Displacement bits — eight bits of the instruction that specify the displacement distance, in memory locations, between some reference point (determined by the mode) and the desired address.

When the index bits are 00, the displacement is considered to be an unsigned integer. When the index bits are 01, 10, or 11, the displacement is considered a signed integer.

	Range of Displacement Field	
Index Bits	Short Class	Extended Class
00	0 to 377_8 or 0 to 255_{10}	0 to 77777_8 or 0 to $32{,}767_{10}$
01	-200_8 to 177_8	-40000_8 to 37777_8
10 or 11	-128 to $+127_{10}$	$-16{,}384$ to $+16{,}383_{10}$

Table 2.1 Index bits with corresponding range of displacement field

Lower page zero – locations 0–377_8 in memory.

Page — a memory storage area of 2 kilobytes. The MAP unit allows any group of up to 32 pages to be organized as a single logical address space.

Addressing Modes

Word addressing can be done in the following modes:

- Absolute (lower page zero locations 0-377_8) mode (Mode 0) — index bits are 00.
- PC (program counter) relative (Mode 1) — index bits are 01.
- AC2 relative (Mode 2) — index bits are 10.
- AC3 relative (Mode 3) — index bits are 11.

In addition, direct or indirect addressing can be used in any of these modes. By choosing the proper mode at the appropriate time, you can obtain access to any address in your logical address space.

Figure 2.1 illustrates the four addressing modes.

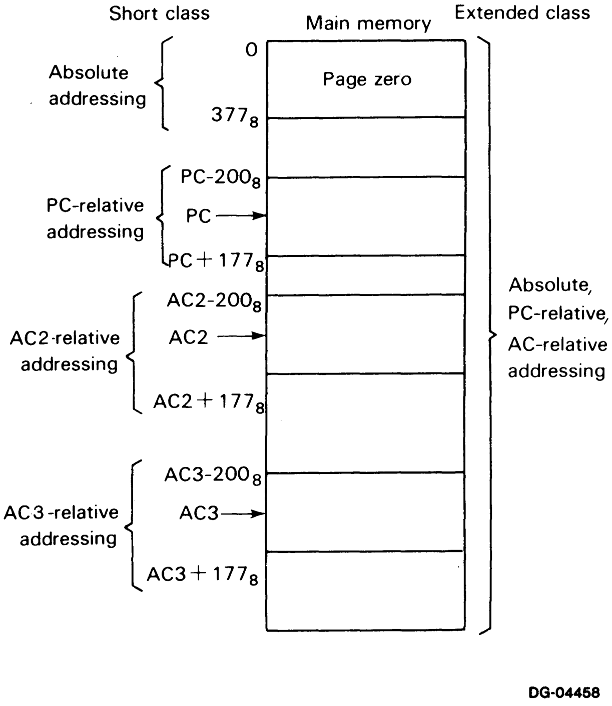


Figure 2.1 Addressing modes

Absolute Addressing

In absolute addressing mode, the intermediate address is set equal to the unmodified displacement. As a result, the short class of instructions specify locations in the range 0 to 377_8 in the absolute mode (short class instructions are restricted to eight bits in the displacement).

Lower page zero thus becomes very important because any memory-reference instruction can address this area. You can use it as a common storage area for items that you frequently reference throughout a program. Note, however, that we reserve some of these locations for special purposes. See Table 2.3 for a list of these locations.

Extended class instructions can reference any logical memory address from 0 to 77777_8 using the absolute addressing mode.

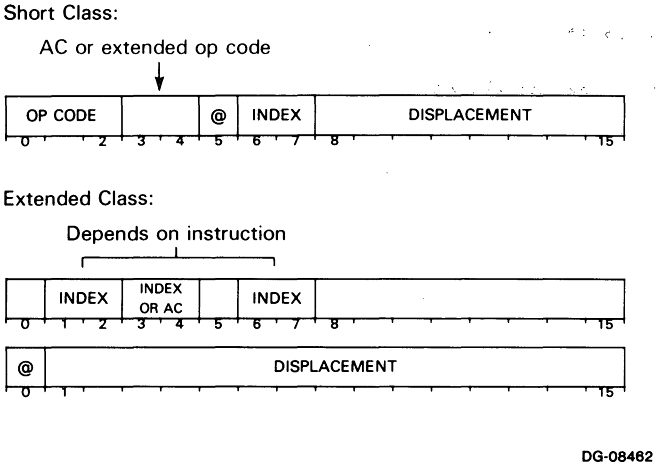


Figure 2.2 Short and extended classes of instructions

PC Relative Addressing

In PC relative addressing mode, the intermediate address is found by adding the displacement which is a two's complement number to the value of the program counter. The value of the program counter is equal to the address of the word containing the displacement.

Accumulator Relative Addressing

In accumulator relative addressing mode, the intermediate address is found by adding the displacement in a two's complement number to the value in the accumulator indicated by the index bits. You may use either AC2 or AC3.

Address Mode	Index	Intermediate Logical Address *	Displacement Range (octal words)
Absolute	00	|D|	0 to 377_8
		|D|	0 to 77777_8
PC Relative	01	PC+D	-200_8 to 177_8
		PC-D	-4000_8 to 37777_8
		PC+1+D	-200_8 to 177_8
		PC+1-D	-4000_8 to 37777_8
AC2 Relative	10	AC2+D	-200_8 to 177_8
		AC2-D	-4000_8 to 37777_8
		AC2+1+D	-200_8 to 177_8
		AC2+1-D	-4000_8 to 37777_8
AC3 Relative	11	AC3+D	-200_8 to 177_8
		AC3-D	-4000_8 to 37777_8
		AC3+1+D	-200_8 to 177_8
		AC3+1-D	-4000_8 to 37777_8

Table 2.2 Addressing mode range

**Bit 0 of PC, AC2, and AC3 is ignored when calculating the intermediate logical address.*

Direct and Indirect Addressing

After the intermediate address is produced from the displacement and index bits, it is translated from a logical address to a physical address. The processor uses the indirect bit (bit 0 of the intermediate address) to determine the final address.

An indirect bit of 0 specifies direct addressing. This means that the intermediate address is the effective address.

An indirect bit of 1 specifies indirect addressing. Indirect addressing uses the intermediate address to obtain an indirect pointer. Bits 1-15 of this pointer are a new intermediate address. If bit 0 of the pointer is 1, then the new intermediate address is used to obtain the next indirect pointer in the indirection chain. If bit 0 of the pointer is 0, then the indirection chain ends and the new intermediate address becomes the effective address. With the MAP indirection protection enabled, the indirect chain is limited to 15 levels. The flowchart in Figure 2.3 explains the address calculation.

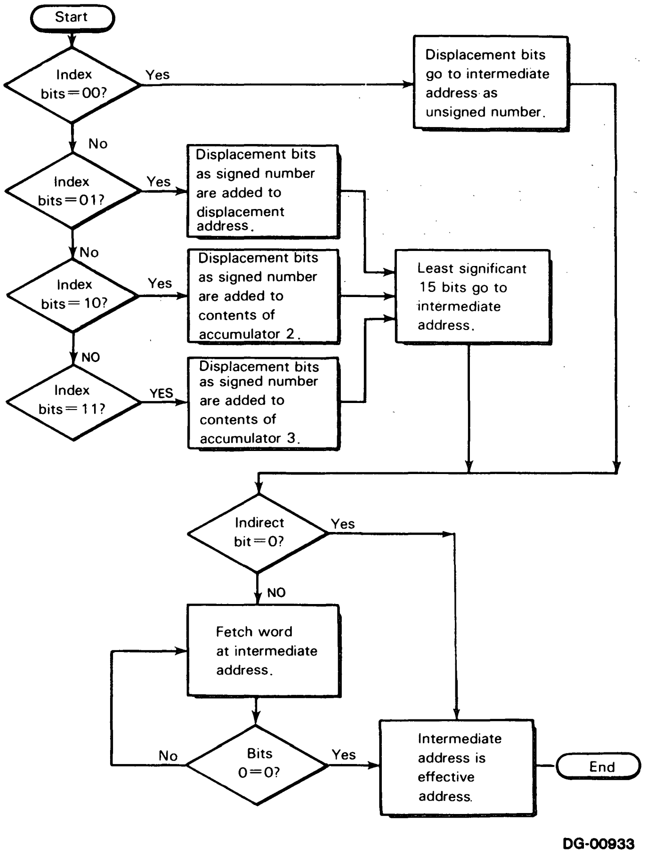

Start
Index bits=00?
Yes
Displacement bits go to intermediate address as unsigned number.
No
Index bits=01?
Yes
Displacement bits as signed number are added to displacement address.
No
Index bits=10?
Yes
Displacement bits as signed number are added to contents of accumulator 2.
NO
Index bits=11?
YES
Displacement bits as signed number are added to contents of accumulator 3.
Least significant 15 bits go to intermediate address.
Indirect bit=0?
Yes
NO
Fetch word at intermediate address.
No
Bits 0=0?
Yes
Intermediate address is effective address.
End
DG-00933

Figure 2.3 Effective address calculation

Effective Address Calculation

Figure 2.3 illustrates how the processor calculates an effective address. First it determines the addressing mode of the addressing reference and constructs an intermediate address accordingly. Next it checks for any indirection. If there is no indirection, the effective address takes on the value of the intermediate address. If there is indirection, the processor calculates a new intermediate address. Once indirection is resolved, the effective address takes on the value of the last-calculated intermediate address.

> **NOTE:** *A memory address is always 15 bits long. When the results of an addition overflows 15 bits, the overflow is ignored.*

Examples:

Short Class

Program Counter = 77774_8
Displacement $+\ 012_8$

Result $= 000006_8$; not 1000006_8

Extended Class

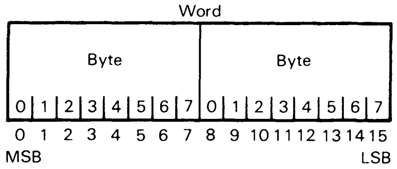
Program Counter = 077774_8
Displacement $+\ 077774_8$

Result $= 077770_8$; not 177770_8

Byte Addressing

A word contains two bytes of eight bits each (see Figure 2.4). Data General uses the following bit-numbering convention:

The most significant bit (MSB) is bit 0 for both bytes and words. The least significant bit (LSB) is bit 7 for bytes and bit 15 for words.



DG-07577

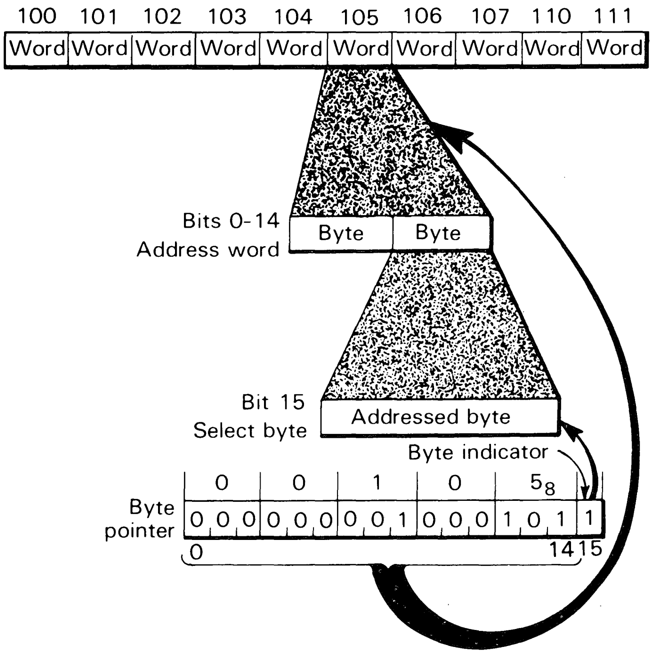
Figure 2.4 Word addressing format

Byte manipulation instructions use a 16-bit *byte pointer* as an address to the desired byte of a word. Bits 0-14 of the byte pointer contain the memory address of the 2-byte word. Bit 15, the *byte indicator*, indicates which byte of the addressed word will be used.

If bit 15 of the byte pointer is 0, the most significant byte (bits 0-7) of the addressed word is used.

If bit 15 of the byte pointer is 1, the least significant byte (bits 8-15) of the addressed word is used.

> **NOTE:** *A byte address is always a direct address, never an indirect address.*



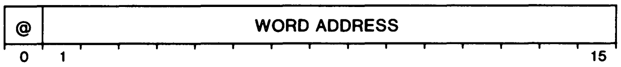
DG-00930

Figure 2.5 Byte addressed as 000213

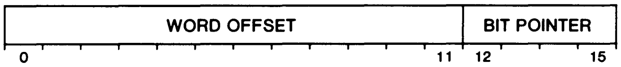
Bit Addressing

Bit addressing uses a word pointer, consisting of a word address, a word offset, and a bit pointer to address a bit in memory (Figure 2.6). The format, loaded into two accumulators specified in the bit instruction, is

ACS Contents



ACD Contents



The source accumulator (ACS) specified in the instruction word contains a word address (possibly indirect) which, upon completion of any indirection chain, is used as the *base address.* To obtain the effective address of the desired word, this base address is added to the unsigned, positive number contained in bits 0-11. These bits are the *word offset* of the destination accumulator (ACD) specified in the instruction.

Bits 12-15 of ACD contain the unsigned number of the position of the desired bit within the addressed word. None of this manipulation affects the original contents of the two accumulators.

> **NOTE:** *If the two accumulators, specified in the bit instruction, are the same accumulator, then the word address is assumed to be zero, and the word in memory is addressed directly using the word offset. For example, address bit 5 of the word at memory location 104 is shown in Figure 2.6: Base=101_8, Offset=3_8, Bit desired=5_8, ACS=000101, and ACD=000065.*

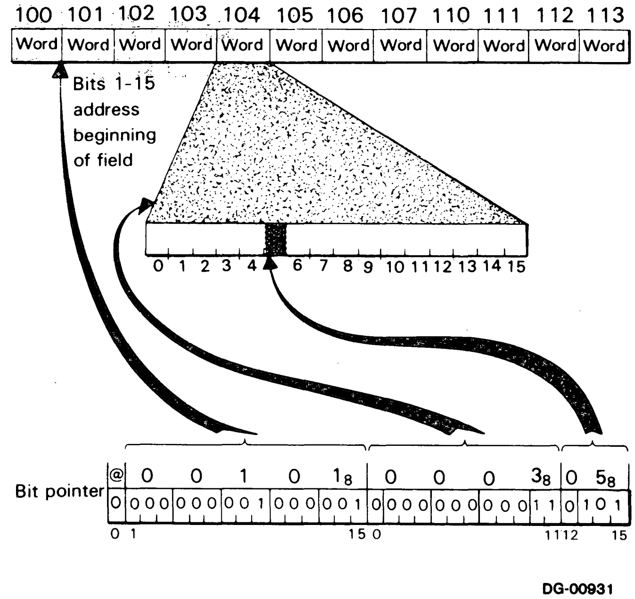


Figure 2.6 Addressing bit 5 of word 104

Reserved Memory Locations

Within lower page zero (0 to 377_8), some memory locations have been reserved as storage for data which have special meaning for the system processing unit (SPU). The locations are program accessible. Table 2.3 lists these locations, their addresses, names, and functions.

Location Address (Octal)	Location Name	Location Function
00000	I/O return address	Return address from I/O interrupt. Also first instruction of auto-restart routine.
00001	I/O handler address	Address of the I/O interrupt handler. Indirectable.
00002	SC handler address	Address of the system call instruction handler. Indirectable.
00003	Map fault handler address	Address of the protection fault handler. Indirectable.
00004	Vector stack pointer	Address of the top of the vector stack. Nonindirectable.
00005	Current mask	Current interrupt priority mask.
00006	Vector stack limit	Address of the last normally usable location in the vector stack.
00007	Vector stack fault address	Address of the vector stack fault handler. Indirectable.
00010	Block pointer	Pointer to context block saved at the time of a page fault or hardware breakpoint.
00011	Emulator trap handler address	Address of the emulator trap handler. Indirectable. (If address = 0, a NOP is performed.) Address of the top of the stack. Nonindirectable.
00040	Stack pointer	Address of the start of the current stack
00041	Frame pointer	frame minus one. Nonindirect. Address of stack upper limit.
00042	Stack limit	Address of the stack fault routine.
00043	Stack fault address	Indirectable. Address of the beginning of XOP
00044	XOP origin address	table. Nonindirectable.
00045	Floating-point fault address	Address of the floating-point fault handler. Indirectable.

Table 2.3 Reserved memory locations

Chapter 3

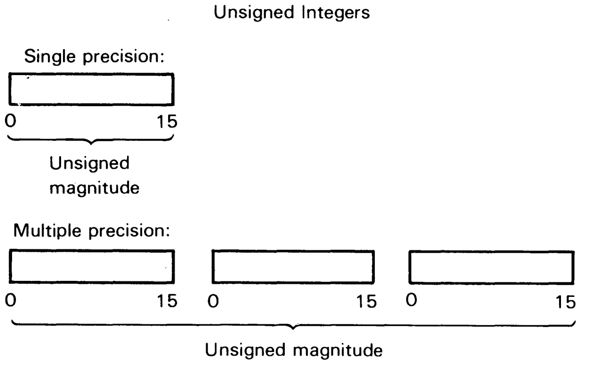
Fixed-Point Instructions

Data Format

Fixed-point numbers are unsigned and signed binary integers.

Unsigned Integers

An unsigned integer is represented by using all of the bits of one or more 16-bit words to represent the magnitude. Single-precision integers are one word (16 bits) long, and multiple-precision integers are two or more words long. See Figure 3.1.



DG-05506

Figure 3.1 Representation of unsigned integers

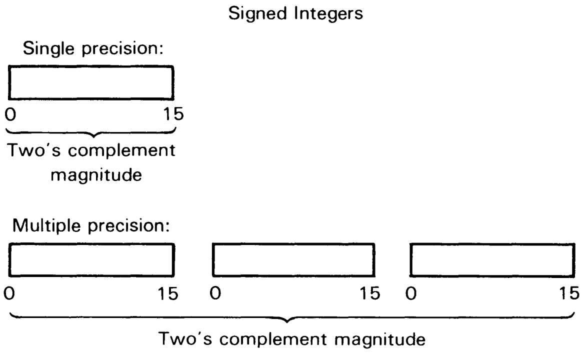
Signed Integers

A signed integer uses a two's complement representation to distinguish between positive and negative values. (See Figure 3.2.)

Table 3.1 is an example of two's complement arithmetic. A positive integer contains a zero in bit 0; a negative integer contains a one in bit 0.

To form the negative of 4:		
4	=	0 000 000 000 000 100
one's complement	=	1 111 111 111 111 011
add 1	+	1
−4	=	1 111 111 111 111 100
To form the negative of 1715_8:		
1715_8	=	0 000 001 111 001 101
one's complement	=	1 111 110 000 110 010
add 1	+	1
-1715_8	=	1 111 110 000 110 011
To form the negative of -1715_8:		
-1715_8	=	1 111 110 000 110 011
one's complement	=	1 000 001 111 001 100
add 1	+	1
1715_8	=	0 000 001 111 001 101
To form the negative of 0_8:		
0	=	0 000 000 000 000 000
one's complement	=	1 111 111 111 111 111
add 1	+	1
0	=	0 000 000 000 000 000

Table 3.1 Example of two's complement representation



DG-08656

Figure 3.2 Representation of signed integers

Table 3.2 shows the possible range of single- and double-precision numbers represented by these formats:

	Single Precision	Double Precision
Unsigned	0 to 65,535	0 to 4,294,967,295
Signed	−32,768 to +32,767	−2,147,483,648 to +2,147,483,647

Table 3.2 Range of single- and double-precision integers

Fixed-Point Operation

The arithmetic logic unit (ALU) is utilized by instructions that perform arithmetic or logical manipulation on operands, move data between accumulators, or check for skip conditions.

There are eight instructions which utilize the ALU for more than one function at a time and share a common format. These instructions are referred to as arithmetic/logic class (ALC) instructions.

ALC Format

Arithmetic/logic class (ALC) instructions **ADC, ADD, AND, COM, INC, MOV, NEG,** and **SUB** perform a group of general functions in addition to the function specified by the instruction. These general functions are encoded in four fields in the ALC instructions. They are:

- Set carry bit (0, 1, complement, or no change);
- Shift (one bit right, one bit left, swap bytes, or no change);
- Skip test;
- Load or No Load.

The format for the 2-accumulator/multiple operation is:

MNEMONIC*[c][sh][#] acs,acd[,skip]*

which assembles as:

1	ACS	ACD	OP CODE	SH	C	#	SKIP
0	1 2	3 4	5 7	8 9	10 11	12	13 15

Bits	Name	Description
0	—	Always set to 1.
1,2	ACS	Specifies the source accumulator.
3,4	ACD	Specifies the destination accumulator.
5-7	Op Code	Contains the operation code.
8,9	Shift	Specifies the action of the shifter.
10,11	Carry	Initializes the carry bit.
12	Load/No Load	Specifies whether or not to load the result obtained into the destination accumulator.
13-15	Skip	Specifies the skip test.

ALC Instructions

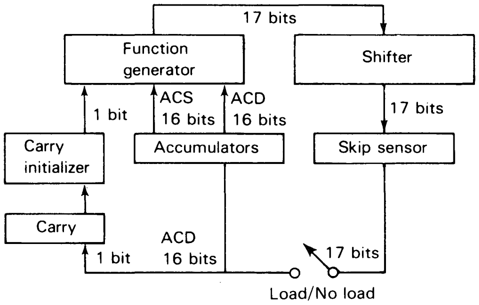
Table 3.3 lists the ALC instructions and briefly describes each one.

Mnem	Instructions	Action
ADC	Add Complement	Adds the logical complement of one unsigned integer to another unsigned integer.
ADD	Add	Adds the contents of one accumulator to the contents of another.
AND	AND	Forms the logical AND of the contents of two accumulators.
COM	Complement	Forms the logical complement of the contents of an accumulator.
INC	Increment	Increments the contents of an accumulator.
MOV	Move	Moves the contents of an accumulator through the ALU.
NEG	Negate	Forms the two's complement of the contents of an accumulator.
SUB	Subtract	Subtracts the contents of one accumulator from the contents of another.

Table 3.3 ALC instructions

ALC Instruction Execution

The logical organization of the ALU is illustrated in Figure 3.3.



DG-00927

Figure 3.3 Arithmetic logic unit

When an ALC instruction begins execution, it loads the contents of carry and the contents of the accumulator(s) to be processed into the ALU. The distinct stages of ALU operation are discussed separately. Refer to Figure 3.4 for the ALC instruction operation sequence.

Carry

The ALU begins its manipulation of the data by determining an initial value for carry. This new value is based upon three things: the old value of carry, bits 10-11 of the ALC instruction, and the ALC instruction being executed. The ALU first determines the effect of the instruction bits 10-11 on the old value of carry. Table 3.4 shows each of the mnemonics that can be appended to the instruction mnemonic, the value of bits 10-11 for each choice, and the action each one takes.

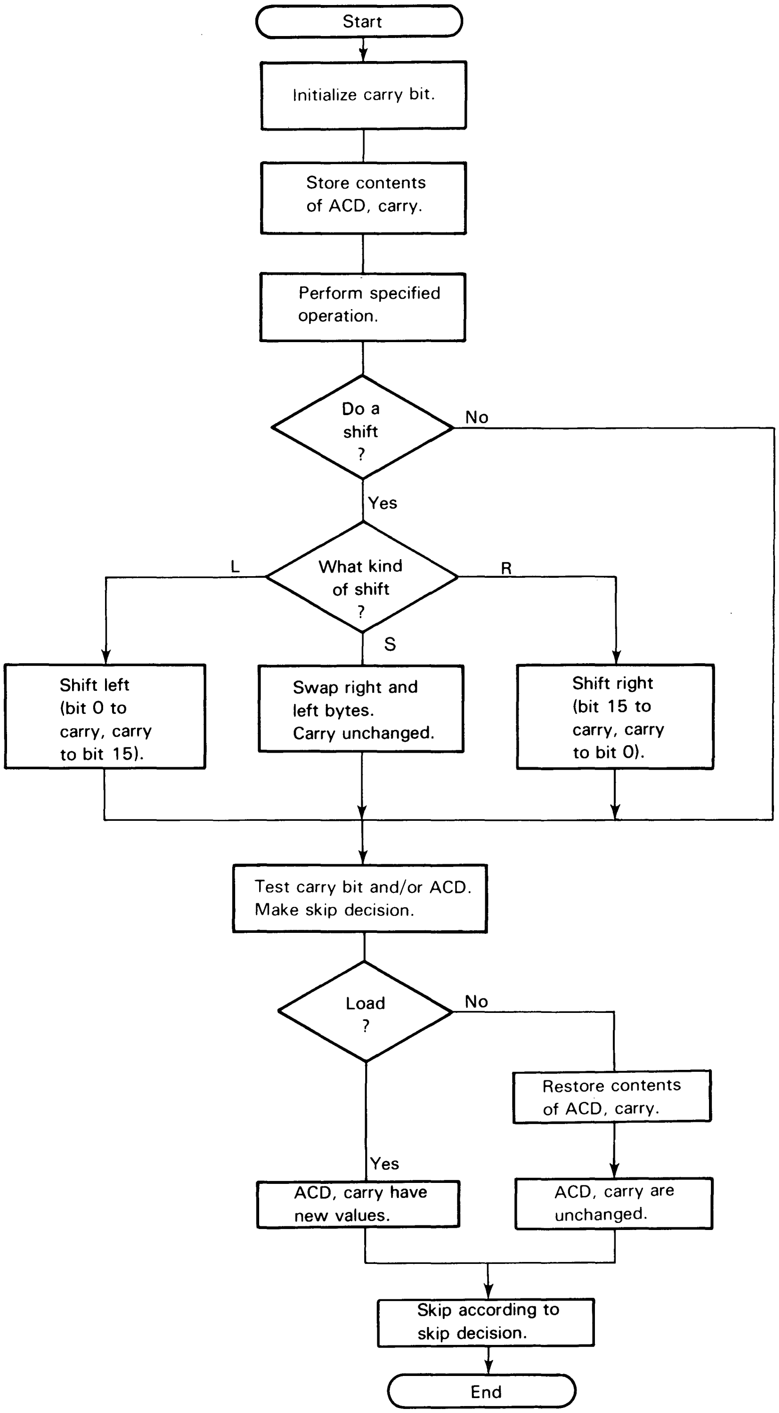
Symbol	Carry (C) Bit Value	Operation
Omitted	00	Leave carry unchanged.
Z	01	Initialize carry to 0.
O	10	Initialize carry to 1.
C	11	Complement carry.

Table 3.4 Carry mnemonics

ALC Specified Function

The ALU next evaluates the effect of the specific function (bits 5-7) upon the data. For the instructions *Move, AND,* and *Complement*, the ALU performs the function on the data word(s) and saves the result. The value of carry is as it was calculated above. For the instructions *Add, Add Complement, Subtract, Negate,* and *Increment*, the result of the function's action upon the data word(s) may be larger than $2^{16} - 1$. An overflow results. In this situation, the ALU saves the 16 least significant bits of the function result, but it complements the value of carry calculated above.

NOTE: *At this stage of operation, the ALU loads neither the saved value of the function result into the destination accumulator nor the calculated value of carry into carry.*


Start
Initialize carry bit.
Store contents of ACD, carry.
Perform specified operation.
Do a shift ?
No
Yes
What kind of shift ?
L
R
S
Shift left (bit 0 to carry, carry to bit 15).
Swap right and left bytes. Carry unchanged.
Shift right (bit 15 to carry, carry to bit 0).
Test carry bit and/or ACD. Make skip decision.
Load ?
No
Restore contents of ACD, carry.
Yes
ACD, carry have new values.
ACD, carry are unchanged.
Skip according to skip decision.
End

DG-08293

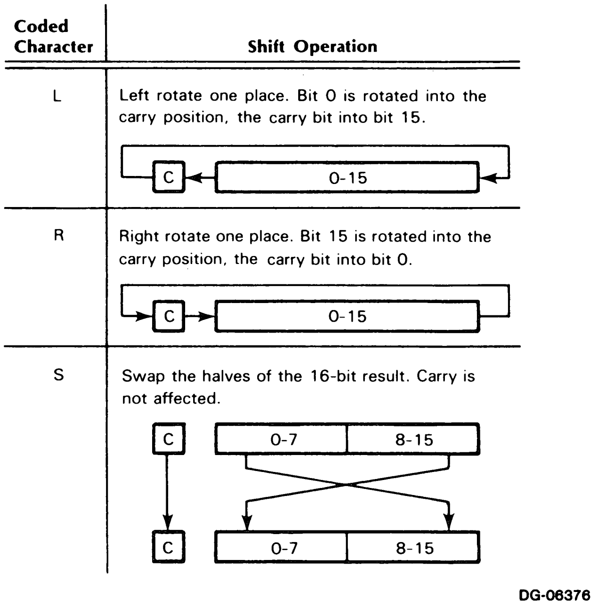
Figure 3.4 ALC instruction sequences

Shift Operations

Next the ALU performs any specified shift operation on the 17 bits output from the function generator (16 bits of data plus the calculated value of the carry bit). Depending on which shift operation is specified in the instruction, the function generator output can be rotated left or right one bit, or it can have its bytes swapped. Table 3.5 shows the different shift operations that can be performed, the value of bits 8-9 for each choice, and the action each choice takes. Figure 3.5 shows how each shift operation works.

Symbol	Shift (SH) Bit Value	Operation
Omitted	00	Do not shift the result of the ALC operation.
L	01	Rotate left the 17-bit combination of carry and ALC operation result.
R	10	Rotate right the 17-bit combination of carry and ALC operation result.
S	11	Swap the two 8-bit halves of the ALC operation result without affecting carry.

Table 3.5 Shift mnemonics



Coded Character	Shift Operation
L	Left rotate one place. Bit 0 is rotated into the carry position, the carry bit into bit 15.
R	Right rotate one place. Bit 15 is rotated into the carry position, the carry bit into bit 0.
S	Swap the halves of the 16-bit result. Carry is not affected.

DG-08376

Figure 3.5 Shift operations

Skip Tests

The ALU can test the result of the shift operation for one of a variety of conditions, and skip or not skip the next word depending upon the result of the test. Table 3.6 shows the tests that can be performed, the value of bits 13-15 for each choice, and the action each choice takes.

Symbol	Skip Bit Value	Operation
Omitted	000	No skip (never skip).
SKP	001	Skip unconditionally.
SZC	010	Skip if carry is zero.
SNC	011	Skip if carry is nonzero.
SZR	100	Skip if ALC result is zero.
SNR	101	Skip if ALC result is nonzero.
SEZ	110	Skip if either ALC result or carry is zero.
SBN	111	Skip if both ALC result and carry are nonzero.

Table 3.6 Skip mnemonics

Load/No Load

If the no-load bit (bit 12) is zero, the ALU loads the result of the shift operation into the destination accumulator and loads the new value of carry into carry. If the no-load bit is one, then the ALU does not load the result of the shift operation into the destination accumulator and does not load the new value of carry into carry; however, the skip tests do take place. This no-load option is particularly convenient to use when you want to test for some condition without overwriting the contents of the destination accumulator. Table 3.7 shows how to code the load/no-load operation.

Symbol	Load/No Load Value	Operation
Omitted	0	Load the result of the shift operation into ACD.
#	1	Do not load the ALC operation result into ACD; restore carry to value it had before shifting.

Table 3.7 Load/no-load symbols

NOTE: *These instructions must have neither the No-Load/Never-Skip nor the No-Load/Skip-Always options specified at the same time.*

Fixed-Point Instruction Lists

Arithmetic Instructions

The fixed-point arithmetic instruction set performs binary arithmetic on operands in accumulators. The operands may be 4 or 16 bits in length and may be signed or unsigned. The instructions appearing in Table 3.8 perform integer arithmetic (often referred to as fixed-point arithmetic).

These operations include add, subtract, multiply, divide, increment, negate, and halve.

Mnem	Instructions	Action
ADDI	Extended Add Immediate	Adds a signed integer in the range of −32,768 to +32,767 to the contents of an accumulator.
ADI	Add Immediate	Adds an unsigned integer in the range of 1 to 4 to the contents of an accumulator.
DIV	Unsigned Divide	Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator.
DIVS	Signed Divide	Divides the signed 32-bit integer in two accumulators by the signed contents of a third accumulator.
DIVX	Sign Extend and Divide	Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* **DIVS** on the result.
HLV	Halve	Divides the contents of an unsigned accumulator by 2.
MUL	Unsigned Multiply	Multiplies the unsigned contents of two accumulators and adds the results to the unsigned contents of a third accumulator.
MULS	Signed Multiply	Multiplies the signed contents of two accumulators and adds the results to the signed contents of a third accumulator.
SBI	Subtract Immediate	Subtracts an unsigned integer in the range of 1 to 4 from the contents of an accumulator.

Table 3.8 Fixed-point arithmetic instructions

The following ALC instructions are usually classified as fixed-point instructions:

ADC
ADD
INC
NEG
SUB

Refer to the descriptions of individual instructions in Chapter 10.

Note that the results of some of the integer arithmetic instructions can affect the value of carry. Overflow conditions complement this value.

Logical Operation Instructions

The logical instruction set performs various logical operations on the contents of accumulators, or the numbers contained in the immediate field and the contents of accumulators.

All of the logical operations instructions are shown in Table 3.9.

The ALC instructions **AND** and **COM**, listed in Table 3.3, are classified as logical operations. Refer to the descriptions of individual instructions in Chapter 10.

Mnemonic	Instructions	Action
ANC	AND With Complemented Source	Forms the logical AND of the contents of one accumulator and the logical complement of the contents of another accumulator.
ANDI	AND Immediate	Forms the logical AND of a 16-bit number contained in the instruction and the contents of an accumulator.
DHXL	Double Hex Shift Left	Shifts the 32-bit contents of two accumulators left one to four hex digits depending on the value of a 2-bit number contained in the instruction.
DHXR	Double Hex Shift Right	Shifts the 32-bit contents of two accumulators right one to four hex digits depending on the value of a 2-bit number contained in the instruction.
DLSH	Double Logical Shift	Shifts the 32-bit contents of two accumulators left or right depending on the contents of a third accumulator.
HXL	Hex Shift Left	Shifts the contents of an accumulator left one to four hex digits depending on the value of a 2-bit number contained in the instruction.
HXR	Hex Shift Right	Shifts the contents of an accumulator right one to four hex digits depending on the value of a 2-bit number contained in the instruction.
IOR	Inclusive OR	Forms the logical inclusive OR of the contents of two accumulators.
IORI	Inclusive OR Immediate	Forms the logical inclusive OR of a 16-bit number contained in the instruction and the contents of an accumulator.
LSH	Logical Shift	Shifts the contents of an accumulator left or right depending on the contents of another accumulator.
XOR	Exclusive OR	Forms the logical exclusive OR of the contents of two accumulators.
XORI	Exclusive OR Immediate	Forms the logical exclusive OR of a 16-bit number contained in the instruction and the contents of an accumulator.

Table 3.9 Logical operations

Decimal Arithmetic Instructions

The decimal arithmetic instruction set performs addition and subtraction on decimal numbers. The decimal arithmetic instructions are shown in Table 3.10.

Mnemonic	Instructions	Action
DAD	Decimal Add	Adds together the decimal digits found in bits 12-15 of two accumulators.
DSB	Decimal Subtract	Subtracts the decimal digit in bits 12-15 of one accumulator from the decimal digit in bits 12-15 of another accumulator.

Table 3.10 Decimal arithmetic

Data Movement Instructions

The data movement instruction set contains instructions that load and store data between accumulators and memory; move or exchange data between accumulators; load the results of an effective address calculation into an accumulator; and move blocks of data between memory locations.

The extended forms of the data movement instructions contain a byte pointer in the instruction coding to reference bytes. The short forms use an accumulator to hold the byte pointer.The ALC instruction *move* (**MOV**) is listed in Table 3.3. Refer to the descriptions of individual instructions in Chapter 10. The data movement instructions are listed in Table 3.11.

Mnemonic	Instructions	Action
BAM	Block Add and Move	Moves word in memory from one location to another, adding a constant to each word.
BLM	Block Move	Moves words in memory from one location to another.
LDA, ELDA	Load Accumulator	Loads data from memory to an accumulator.
LEF, ELEF	Load Effective Address	Places an effective address in an accumulator.
STA, ESTA	Store Accumulator	Stores data in memory from an accumulator.
XCH	Exchange Accumulators	Exchanges the contents of two accumulators.

Table 3.11 Data movement

Byte Manipulation Instructions

The byte instruction set contains instructions that store and load bytes between accumulators and memory, or move strings of bytes between memory locations with various control options.

When an instruction moves a byte to an accumulator, it also clears the most significant byte of the destination accumulator. When an instruction moves a byte from an accumulator to memory, it leaves unchanged the other byte contained in that word of memory.

The extended forms of the byte instructions contain a byte pointer in the instruction coding to reference bytes. The short forms use an accumulator to hold the byte pointer. See Byte Addressing, Chapter 2. Byte manipulation instructions are listed in Table 3.12.

Mnemonic	Instructions	Action
CMP	Character Compare	Compares one string of characters in memory to another string.
CMT	Character Move Until True	Moves a string of bytes from one area of memory to another until a table-specified delimiter character is encountered or the source string is exhausted.
CMV	Character Move	Moves a string of bytes from one area of memory to another under control of the values in the four accumulators.
CTR	Character Translate	Translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second string of bytes.
LDB, ELDB	Load Byte	Places a byte of information into an accumulator.
STB, ESTB	Store Byte	Stores the right byte of an accumulator into a byte of memory.

Table 3.12 Byte manipulation instructions

Bit Manipulation Instructions

The instructions that manipulate bits:

- Locate a bit in memory and set it to 0 or 1;
- Add a number to the contents of one accumulator based on the number of ones or high-order zeroes found in another accumulator;
- Test a bit, skipping the next word if the specified condition is true. See Bit Addressing, Chapter 2. Bit manipulation instructions are listed in Table 3.13.

Mnemonic	Instruction	Action
BTO	Set Bit To One	Sets the bit addressed by the bit pointer to one.
BTZ	Set Bit To Zero	Sets the bit addressed by the bit pointer to zero.
COB	Count Bits	Counts the number of ones in one accumulator and adds that number to the second accumulator.
LOB	Locate Lead Bit	Counts the number of high-order zeroes in one accumulator and adds that number to the second accumulator.
LRB	Locate And Reset Lead Bit	Performs a *Locate Lead Bit* **LOB** instruction and sets the lead bit to zero.
SNB	Skip On Nonzero Bit	Skips the next sequential word if the bit addressed by the bit pointer is one.
SZB	Skip On Zero Bit	Skips the next sequential word if the bit addressed by the bit pointer is zero.
SZBO	Skip On Zero Bit And Set To One	Sets the bit addressed by the bit pointer to one and skips the next sequential word if the bit was originally zero.

Table 3.13 Bit manipulation instructions

Chapter 4

Floating-Point Instructions

Floating-point format allows the use of very large numbers and fractions, and floating-point operations are faster than multiple-precision fixed-point operations.

Data Format

Floating-point numbers occupy either two words (single precision) or four words, (double precision). The format of single- and double-precision floating-point numbers is shown in Figure 4.1. The floating-point number is composed of:

- A sign;
- A mantissa, which is the fractional part of the number, adjusted to be greater than or equal to 1/16 and less than 1 after every operation;
- An exponent, which is adjusted to maintain the correct value of the number.

The magnitude of a floating-point number is defined as

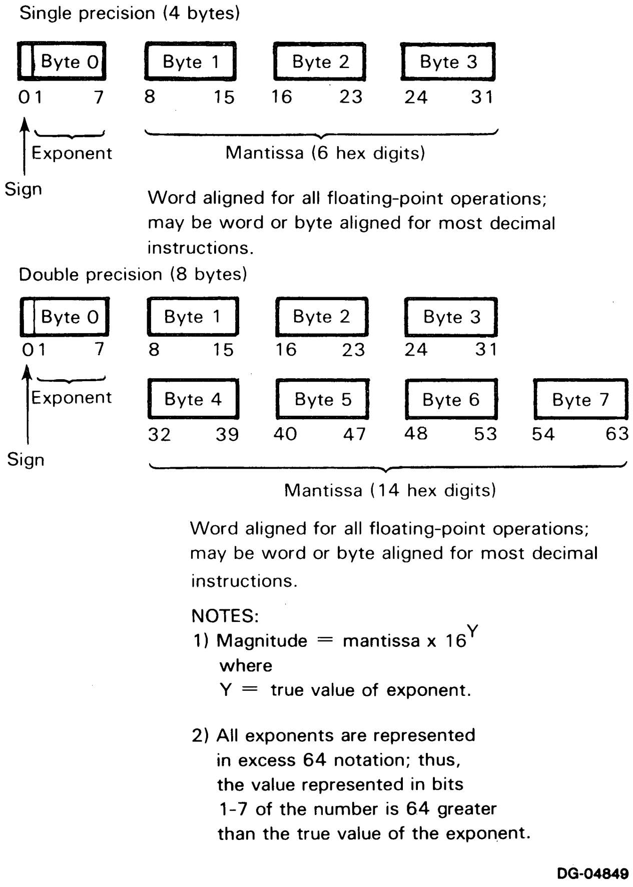
MANTISSA X 16^y

where y is the true value of the exponent.

Floating-point zero is represented by a number with all bits zero, known as pure zero. When a calculation results in a zero mantissa, the number is automatically converted to pure zero. If a number has a zero mantissa but neither a zero sign nor exponent, it is called impure zero. When representing zero as a floating-point number, use pure zero; impure zero produces undefined results in calculations.

Sign

Bit 0 of the first byte is the sign bit. If the sign bit is zero, the number is positive. If the sign bit is one, the number is negative.



Word aligned for all floating-point operations; may be word or byte aligned for most decimal instructions.

Word aligned for all floating-point operations; may be word or byte aligned for most decimal instructions.

NOTES:

1) Magnitude = mantissa x 16^Y
where
Y = true value of exponent.

2) All exponents are represented in excess 64 notation; thus, the value represented in bits 1-7 of the number is 64 greater than the true value of the exponent.

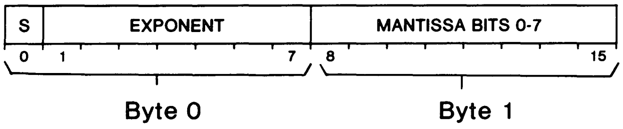
Figure 4.1 Single- and double-precision floating-point numbers

Mantissa

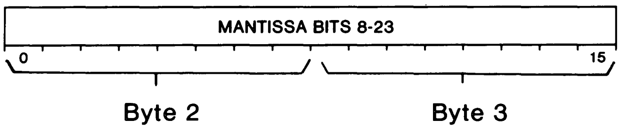
The mantissa is an unsigned fraction. The mantissa of a single-precision number occupies bytes 1 to 3; the mantissa of a double-precision number occupies bytes 1 to 7. The single- and double-precision formats follow.

Single Precision

Word 1

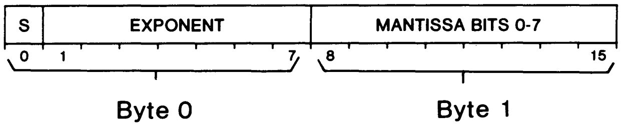


Word 2

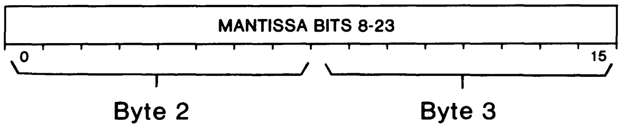


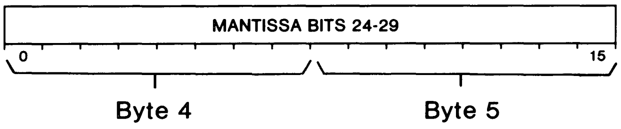
Double Precision

Word 1

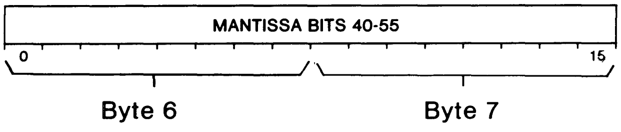


Word 2





Word 4



The binary point is located to the left of the first bit of the mantissa.

To keep the mantissa in the range of 1/16 to 1, the results of each floating-point calculation are normalized.

A mantissa is normalized by shifting one or more hex digits (four bits) 1-1 left or 1 digit right until the four most significant bits (the left-most four bits of byte 1) represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one.

Exponent

Bits 1-7 of the first byte contain the exponent. All exponents are represented in excess 64 representation. This means that the value of the number represented in bits 1-7 is 64 greater than the true value of the exponent. The range of true value of the exponent field is 0 to 127. The range of true value of the exponent is −64 to 63. See Table 4.1.

Exponent Field	True Value of Exponent
0	−64
64	0
127	63

Table 4.1 Excess 64 representation

Floating-Point Operation

Floating-point instructions assume normalized input numbers. Results are undefined for unnormalized input.

Floating-Point Registers

There are five registers available to the programmer in the floating-point processor. These are the four floating-point accumulators (FPACs) and the floating-point status register (FPSR). The FPACs are numbered FAC0, FAC1, FAC2, and FAC3. The FPSR is a 32-bit register that contains information about the present status of the floating-point processor.

Guard Digits

In order to increase the accuracy of floating-point operations, a guard digit is appended to the least significant bit of each mantissa. A guard digit is one hex digit (four bits) that initially contains zeroes.

When a floating-point operation between two floating-point operands is specified, the processor first appends one guard digit.

After appending the guard digit, the processor performs the specified operation. The result of the operation is called the intermediate result. The processor normalizes this value if necessary by shifting the intermediate result left one hex digit (four bits) at a time until the four most significant bits (bits 0-3 of the mantissa) represent a nonzero quantity. Zeroes are filled in on the right. For every hex digit shifted, the processor decrements the exponent of the intermediate result by one.

Floating-Point Status Register

The floating-point status register (FPSR) is a 15-bit register that contains two 16-bit words that give information about the present status of the floating-point processor.

The format of the first word of the FPSR is:

ANY	OVF	UNF	DVZ	MOF	TE	Z	N	0	RES	0	0	FPMOD	
0	1	2	3	4	5	6	7	8	9	10	11	12	15

The format of the second word is:

0	FLOATING-POINT PROGRAM COUNTER	
0	1	15

Bits	Name	Description
0	ANY	Indicates that any of bits 1-4 are set. If ANY = one, then the contents of the floating-point program counter are valid. If ANY = zero, then the contents of the floating-point program counter are undefined.
1	OVF	Overflow indicator — Set during the processing of a floating-point number if an exponent overflow occurred; the result is correct except that the exponent is 128 too small.
2	UNF	Underflow indicator — Set during the processing of a floating-point number if an exponent overflow occurred; the result is correct except that the exponent is 128 too large.
3	DVZ	Divide by Zero — Set during the processing of a floating-point division if a zero divisor was detected. The division was aborted and the operands remain unchanged.
4	MOF	Mantissa Overflow — Set during the processing of a floating-point number if a significant bit was shifted out of the high-order end of the mantissa.
5	TE	Trap Enable — If this bit is set to one, the setting of any of bits 0-4 will result in a floating-point trap.
6	Z	Zero — The result of the last floating-point operation was equal to zero.
7	N	Negative — The result of the last floating-point operation was less than zero.
8	—	Reserved for future use.
9	RES	Resume — Used internally by microcode. Should be ignored and not modified by the programmer when saving and restoring the floating-point status register. When initializing floating-point status, this should be set to zero.
10,11	—	Reserved for future use.
12-15	FPMOD	Floating-point implementation code.*
16		Reserved for future use.
17-31	FPPC	Floating-point program counter. In the event of a floating-point fault, this is the address of the floating-point instruction that caused the fault. If there is no floating-point fault, the FPPC is undefined.

Table 4.2 Floating-point format bit description

Some floating-point operations cause the following fault conditions.

Overflow — exponent overflow occurred. (The result is correct except that the exponent is 128 too small.)

Underflow — exponent underflow occurred. (The result is correct except that the exponent is 128 too large.)

Divide by Zero — zero divisor detected; division aborted.

Mantissa Overflow — a bit was shifted out of the most-significant bit of the mantissa during an **FSCAL** instruction, or the result of an **FFAS** or **FFMD** instruction does not fit into the destination.

Floating-Point Faults

If the program has set the trap enabling bit (5) in the floating-point status register to one, a floating-point fault condition will initiate a floating-point trap.

Before the next sequential instruction is executed, a return block is pushed onto the stack and the program jumps indirect via location 45_8. Location 45_8 contains the address of the floating-point fault handler. The return block pushed has the following format.

Word Pushed	Description
1	AC0
2	AC1
3	AC2
4	AC3
5	Bit 0 = Carry bit; Bits 1-15 = Return address

NOTE: *The return address pushed in word five is the address following the floating-point instruction that caused the fault.*

When a floating-point fault occurs and the trap enable bit is one, the trap enable bit is set to zero before control is transferred to the floating-point fault handler. The trap enable bit should be set to one before normal processing resumes.

Floating-Point Instruction Lists

The floating-point instruction set may be divided into arithmetic, data movement, program flow alteration, number conversion, and floating-point status register (FPSR) instructions. Floating-point instructions assume normalized input numbers; unnormalized input results in undefined output.

The FPSR is updated after each completed floating-point instruction. At this time the FPSR is checked for possible fault conditions.

*This code is 10_8.

The floating-point instructions are shown in Tables 4.2 through 4.6. Note that several instructions have two forms, one ending in S (for single-precision floating-point format), and one ending in D (for double-precision floating-point format). The function of the two forms is otherwise identical.

Arithmetic Instructions

The floating-point arithmetic instruction set contains instructions to perform arithmetic functions on both single- and double-precision floating-point numbers. See Table 4.3.

Mnem	Instructions	Action
FAMS, FAMD	Add (Memory To FPAC)	Adds the floating-point number in memory to the floating-point number in an FPAC.
FAS, FAD	Add (FPAC To FPAC)	Adds the floating-point number in one FPAC to the floating-point number in another FPAC.
FCMP	Compare Floating Point	Compares two floating-point numbers and sets the Z and N flags accordingly.
FDMS, FDMD	Divide (FPAC By Memory)	Divides the floating-point number in an FPAC by a floating-point number in memory.
FDS, FDD	Divide (FPAC By FPAC)	Divides the floating-point number in one FPAC by the floating-point number in another FPAC.
FHLV	Halve	Divides the floating-point number in FPAC by two.
FMMS, FMMD	Multiply (Memory By FPAC)	Multiplies the floating-point number in memory by the floating-point number in an FPAC.
FMS, FMD	Multiply (FPAC By FPAC)	Multiplies the floating-point number in one FPAC by the floating-point number in another FPAC.
FNEG	Negate	If FPAC does not contain true zero, it inverts the sign bit.
FSMS, FSMD	Subtract (Memory From FPAC)	Subtracts the floating-point number in memory from the floating-point number in an FPAC.
FSS, FSD	Subtract (FPAC From FPAC)	Subtracts the floating-point number in one FPAC from the floating-point number in another FPAC.

Table 4.3 Arithmetic instructions

Floating Point Operation

Each user must provide stack space for floating point operations.

To protect against stack overflow in a program using floating-point arithmetic operations, allocate the following number of words to the stack:

Mnemonic	Number of Words used by Instruction
FMD, FMMD	4
FDS, FDMS	6
FDD, FDMD	14

Data Movement Instructions

The floating-point data movement instruction set contains instructions to load or store floating-point numbers between memory and a floating-point accumulator (FPAC), or move a floating-point number from one accumulator to another FPAC.

Mnemonic	Instructions	Action
FLDS, FLDD	Load Floating-Point	Moves a floating-point number from memory to a specified FPAC.
FMOV	Move Floating-Point	Moves the contents of one FPAC to another FPAC.
FSTS, FSTD	Store Floating-Point	Stores the contents of a specified FPAC into memory.

Table 4.4 Data movement instructions

Program Flow Alteration Instructions

The floating-point program flow alteration instruction set contains instructions to conditionally skip the next sequential instruction, depending upon certain flag settings in the FPSR. Program flow alteration instructions are listed in Table 4.5.

Mnem	Instructions	Action
FNS	No Skip	Executes the next sequential word.
FSA	Skip Always	Skips the next sequential instruction.
FSEQ	Skip On Zero	Skips the next sequential word if the Z flag in the FPSR is one.
FSGE	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if the N flag of the FPSR is zero.
FSGT	Skip On Greater Than Or Equal To Zero	Skips the next sequential word if both the Z and N flags of the FPSR are zero.
FSLE	Skip On Less Than Or Equal To Zero	Skips the next sequential word if either the Z flag or the N flag of the FPSR is one.
FSLT	Skip On Less Than Zero	Skips the next sequential word if the N flag of the FPSR is one.
FSND	Skip On No Zero Divide	Skips the next sequential word if the divide by zero (DVZ) flag of the FPSR is zero.
FSNE	Skip On Nonzero	Skips the next sequential word if the Z flag of the FPSR is zero.
FSNER	Skip On No Error	Skips the next sequential word if bits 1-4 of the FPSR are all zero.
FSNM	Skip On No Mantissa Overflow	Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is zero.
FSNO	Skip On No Overflow	Skips the next sequential word if the overflow (OVF) flag of the FPSR is zero.
FSNOD	Skip On No Overflow And No Zero Divide	Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the FPSR are zero.
FSNU	Skip On No Underflow	Skips the next sequential word if the underflow (UNF) flag of the FPSR is zero.
FSNUD	Skip On No Underflow And No Zero Divide	Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the FPSR are zero.
FSNUO	Skip On No Underflow And No Overflow	Skips the next sequential word if both the underflow (UNF) flag and the overflow (OVF) flag of the FPSR are zero.

Table 4.5 Program flow alteration instructions

Number Conversion Instructions

The floating-point number conversion instruction set contains instructions that return an absolute value, change the exponent of a floating-point number, convert a floating-point number to an integer or an integer to a floating-point number, and integerize or normalize a floating-point number. Number conversion instructions are listed in Table 4.6.

Mnem	Instructions	Action
FAB	Absolute Value	Sets the sign bit of an FPAC to zero.
FEXP	Load Exponent	Places bits 1-7 of AC0 in bits 1-7 of the specified FPAC. If FPAC contains zero, it will remain a true zero.
FFAS	Fix To AC	Converts the integer portion of a floating-point number to a signed two's complement integer and places the result in an accumulator.
FFMD	Fix To Memory	Converts the integer portion of a floating-point number to double-precision integer format and stores the result in two memory locations.
FINT	Integerize	Sets the fractional portion of the floating-point number in the specified FPAC to zero and normalizes the result.
FLAS	Float From AC	Converts a signed two's complement number in an accumulator to a floating-point number.
FLMD	Float From Memory	Converts the contents of two memory locations in integer format to floating-point format and places the result in a specified FPAC.
FNOM	Normalize	Normalizes the floating-point number in FPAC.
FRH	Read High Word	Places the 16 most significant bits of an FPAC in AC0.
FSCAL	Scale	Shifts the mantissa of the floating-point number in FPAC either right or left, depending upon the contents of bits 1-7 of AC0.

Table 4.6 Number conversion instructions

Status Register Instructions

The floating-point status register instruction set contains instructions that store the FPSR contents into memory, load the FPSR from memory, clear all errors, and enable and disable floating-point traps.

Mnem	Instructions	Action
FCLE	Clear Errors	Sets bits 0-4 of the FPSR to zero.
FLST	Load Floating-Point Status	Moves the contents of two memory locations to the FPSR.
FSST	Store Floating-Point Status	Moves the contents of the FPSR to two memory locations.
FTD	Trap Disable	Sets the trap enable (TE) flag of the FPSR to zero.
FTE	Trap Enable	Sets the trap enable (TE) flag of the FPSR to one.

Table 4.7 Floating-point status register instructions

Chapter 5

Stack Management

Stack Operation

A stack is a series of consecutive locations in memory. Stack instructions add items in sequential order to the stack (*push*) or retrieve them in reverse order (*pop*). The processor maintains this last in/first out (LIFO), or "push down," stack.

The stack stores temporary data, as well as return blocks, which contain information that the processor uses when entering and returning from subroutines. The stack is managed by four reserved storage locations: stack pointer, frame pointer, stack limit, and stack fault pointer. An important byproduct of the stack facility is that storage locations are reserved only when needed. Once a procedure is finished with its portion of the stack, those memory locations may be reclaimed by another procedure for further use.

Stack instructions store the contents of accumulators on the stack, change the stack register that controls the stack, define new stacks, and perform other tasks.

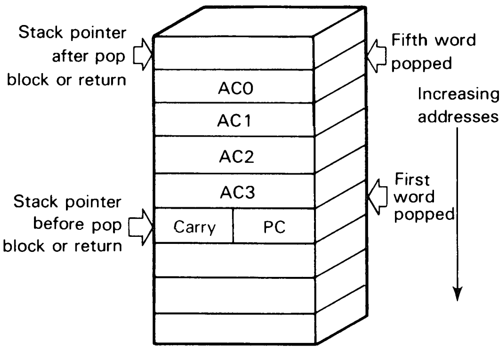
Return Block

Return blocks are used to enter or exit from subroutines. The contents of the return block may vary slightly depending upon which instruction pushes the block, but the purpose of the block is always the same — to allow an orderly return from a called routine. For a discussion of subroutine call and return, refer to Appendix D, Programming Examples.

The contents of the return block, depending on the instruction used, are the contents of the four accumulators (AC0, AC1, AC2, AC3), the program counter, and the carry bit. Figure 5.1 illustrates a standard return block. The instructions that use a standard return block are **SYC, VCT** (modes C and E), **XOP, XOP1,** and **POPB. RSTR, RTN, SAVE, FPOP,** and **FPSH** instructions affecting the stack are contained in Tables 5.3 and 5.4.

> NOTE: *If an undefined instruction is encountered while operating in the mapped mode, a return block is pushed onto the stack. The program then jumps indirectly through location 11_8. This location can contain the indirect address of an emulator routine.*

The instructions affecting the stack are contained in Tables 5.3 and 5.4.


Stack pointer after pop block or return
AC0
AC1
AC2
AC3
Carry
PC
Stack pointer before pop block or return
Fifth word popped
Increasing addresses
First word popped

DG-08275

Figure 5.1 Return block

Stack Pointer

The stack pointer (SP) is the address of the *top* (highest) memory location of the stack reached, thus far. When you set up the stack, you set the value of the stack pointer to one less than the address of the first location in the stack. After initialization, each time you push a word onto the stack the stack increments by one. The stack pointer always points to the last element on the stack. For example, when you pop the top word from the stack, the pointer decrements by one. If you push or pop a 5-word return block, the stack pointer increments or decrements by five.

Address 400_8 is considered a standard starting address for a stack. Location 40_8 contains the current value of the stack pointer.

Frame Pointer

The *Save* (**SAVE**) and *Return* (**RTN**) instructions use the frame pointer to store and restore the value of the stack pointer when entering or leaving subroutines. The frame pointer points to the top of the last return block pushed. The frame pointer is not incremented or decremented by operations that affect the stack pointer.

If the frame pointer is initially set to the same value as the stack pointer, it becomes a useful reference, since it preserves the original value of the stack pointer. Location 40_8 contains the value of the frame pointer.

The frame pointer may also define the boundary between words placed on the stack by different routines in a program. A routine may then use the frame pointer to refer back into the stack. In this way, the routine may retrieve data left in the stack by the preceding procedure.

Stack Limit

The stack limit is the upper limit of the stack area. After each push or save operation, the value of the stack pointer is compared with the value of the stack limit. If the stack pointer is greater than the stack limit, an overflow condition exists and a stack fault occurs. Therefore, the stack limit should be initialized to a value greater than the stack pointer. Figure 5.2 illustrates the location of the stack pointer and stack limit in main memory.

Location 42_8 contains the value of the stack limit.

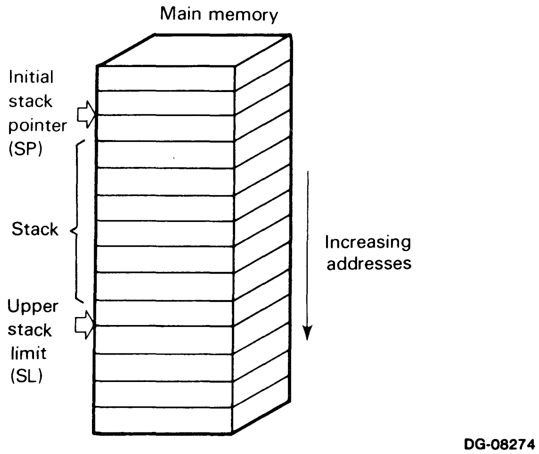


Figure 5.2 Main memory

Stack Control Memory Locations

Table 5.1 lists stack control memory locations.

Memory Location	Contents
40_8	Stack Pointer
41_8	Stack Frame Pointer
42_8	Stack Limit
43_8	Stack Fault Pointer

Table 5.1 Stack control memory locations

Stack Overflow Protection

Stack overflow occurs when a program pushes data into the area beyond the stack limit. The stack limit protects the integrity of the program against stack overflow. See "Stack Limit."

If a stack instruction pushes data onto the stack beyond the stack limit, a 5-word return block is pushed onto the stack. Control is transferred to the stack fault handler at the stack fault address. Since stack overflow is detected only after completion of a push operation, set the stack limit up to 23 words less than the address of the last location in the stack.

To protect against stack overflow in a program not using floating-point instructions:

- Initialize the stack limit to 10 less than the address of the last location in the stack.

For floating-point push and pop stack operations:

- Initialize the stack limit to 23 less than the address of the last location in the stack.

For floating-point arithmetic operations (excluding push and pop), refer to Table 5.2 for the correct word per instruction allocation to the stack.

Mnemonic	Number Words Used by Instruction
FMD, FMMD	4
FDS, FDMS	6
FDD, FDMD	14

Table 5.2 Allocation of words to the stack

To disable overflow protection, set the stack limit to 177777_8.

Overflow Checking

During the course of checking for stack overflow, the stack pointer and the stack limit are treated as unsigned, 16-bit integers and are compared. If overflow has occurred, the processor:

- Sets bit 0 of the stack pointer to zero,
- Sets bit 0 of the stack limit to one,
- Pushes a 5-word return block onto the stack,
- Executes a Jump Indirect to the stack fault address (location 43_8).

Mnemonic	Instructions	Action
FPOP	Pop Floating- Point State	Pops a floating-point return block off user stack.
FPSH	Push Floating- Point State	Pushes a floating-point return block onto user stack.
POP	Pop Multiple Accumulators	Pops up to four words from stack into accumulators.
POPB	Pop Block	Pops words from stack.
POPJ	Pop PC and Jump	Pops word from stack into program counter.
PSH	Push Multiple Accumulators	Pushes accumulators onto stack.
PSHJ	Push Jump	Pushes next address onto stack.
PSHR	Push Return Address	Increments present address by two and pushes onto stack.
RSTR	Restore	Returns program control from certain I/O interrupts.
RTN	Return	Returns program control from saved subroutines.
SAVE	Save	Pushes return block onto stack and saves more stack storage space.
SYC	System Call	Pushes return block and indirectly places the address of the system call handler into the program counter.
VCT	Vector On Interrupting Device	Interrupt handler. Transfers control to a specific I/O device.
XOP	Extended Operation	Pushes return block onto stack and accesses routines.
XOP1	Extended Operation	Pushes return block onto stack and accesses routines.

Table 5.3 Stack instruction description

Bit 0 of the stack pointer and stack limit are set as indicated so that the stack limit will (temporarily) be larger than the stack pointer. In this way, the return block pushed by the overflow mechanism will not be interpreted as yet another overflow fault, causing a loop condition. The program counter in the return block points to the instruction that immediately follows the stack instruction that caused the fault.

Stack Fault Handler

The stack fault handler routine is created by the programmer and addressed by the stack fault address. The stack fault handler routine receives control in the event of a stack overflow and then determines the nature of the stack fault.

Bit 0 of the stack pointer and the stack limit should be reset to their original values. Any further action required of the stack fault handler, such as allocating more stack space or terminating the program, should be taken at this time.

Location 43_8 contains the address (possibly indirect) of the stack fault handler routine.

Interrupting the Stack Instructions

Stack instructions, with the exception of the floating-point stack instructions, are noninterruptible. The stack pointer and program counter are not updated until the completion of the floating-point pop or push stack instructions; therefore, any interrupt service routines that return control to the interrupted program, through the program counter stored in memory location 0, will correctly restart the two floating-point stack instructions.

Stack Instructions

The following two tables list the instructions that pop and push words onto the stack. Table 5.3 presents a brief description of each instruction with the number of words affected. Table 5.4 lists the minimum safety margin for the stack limit, which instruction works in conjunction with another, and what is pushed on or popped off the stack.

Mnemonic	Stack Limit (minimum safety margin for overflow protection)	Works in Conjunction With	Word(s) Pushed/Popped (in order of push/pop)	Number of Words	
				Pushed	Popped
FPOP	0	FPSH	FPAC3 FPAC2 FPAC1 FPAC0 FPSR (bits 16-31) FPSR (bits 0-15)		18
FPSH	23	FPOP	FPSR (bits 0-15) FPSR (bits 16-31) FPAC0 FPAC1 FPAC2 FPAC3	18	
POP	0	PSH	AC<0-3>		1-4
POPB	0	SYC, VCT (without stack change)	Return Block		5
POPJ	0	PSHJ, PSHR	Program Counter		1
PSH	6-9	POP	AC<0-3>	1-4	
PSHJ	6	POPJ	Next Address	1 1	
PSHR	6	POPJ	Program Counter+2		
RSTR	14	VCT (with stack change)	Return Block Stack Fault Pointer Stack Limit Frame Pointer Stack Pointer		9
RTN	0	SAVE	Bit 0 = Carry bit Bits 1-15 = PC Frame Pointer AC2 AC1 AC0		5
SAVE	10+	RTN	AC0 AC1 AC2 Frame Pointer Bit 0 = Carry bit Bits 1-15 = AC3	5	
SYC	10	POPB	Return Block	5	
VCT				0 or 5	
Mode A	0	JMP@0	—		
Mode B	0	JMP@0	—		
Mode C	5	JMP@0	Return Block		
Mode D	0	POPB	Stack Pointer Frame Pointer Stack Limit Stack Fault Pointer		
Mode E	9	RSTR	Return Block Stack Pointer Frame Pointer Stack Limit Stack Fault Pointer		
XOP	10	POPB	Return Block	5	
XOP1	10	POPB	Return Block	5	

Table 5.4 Stack instructions

Return Block	=AC0
	=AC1
	=AC2
	=AC3
	Bit 0 = Carry bit
	Bits 1-15 = PC

Examples — Stacks

Figure 5.3 illustrates a 50_8-word stack set up with overflow protection enabled for fixed-point arithmetic. The following assembly language instructions may be used:

```
.TITL STACK
.EXTN STH      ;Declare STH external
.LOC 400       ;Go to location 400
.BLK 50        ;Allocate $50_8$ words
.LOC 40        ;Go to stack control words
377            ;Stack pointer
377            ;Frame pointer
434            ;Stack limit
STKHR          ;Address of stack fault
.END
```

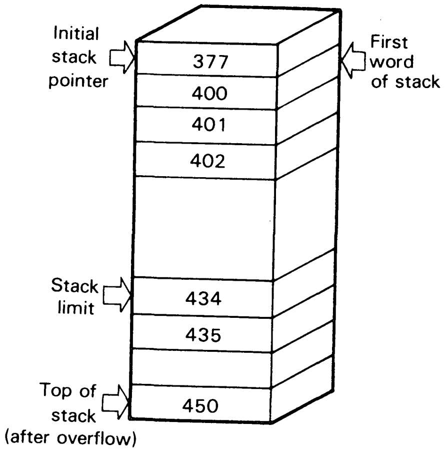


Figure 5.3 Condition of stack after overflow routine

Figure 5.4 shows a stack area of 100_8 words with no overflow protection.

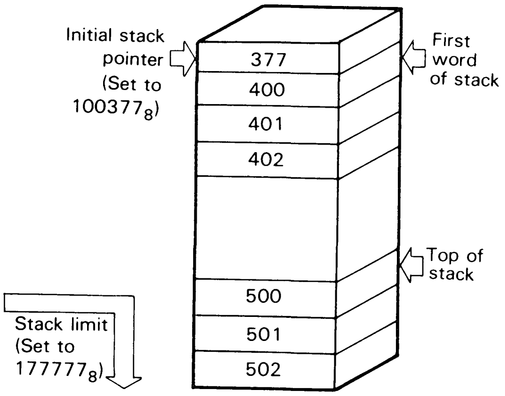

Initial stack pointer (Set to 100377_8)
377
400
401
402
500
501
502
First word of stack
Top of stack
Stack limit (Set to 177777_8)
DG-08278

Figure 5.4 Condition of stack with no overflow protection

NOTE: *To disable stack overflow protection, set the stack limit to* 177777_8.

Chapter 6

Program Flow Management

Program Flow

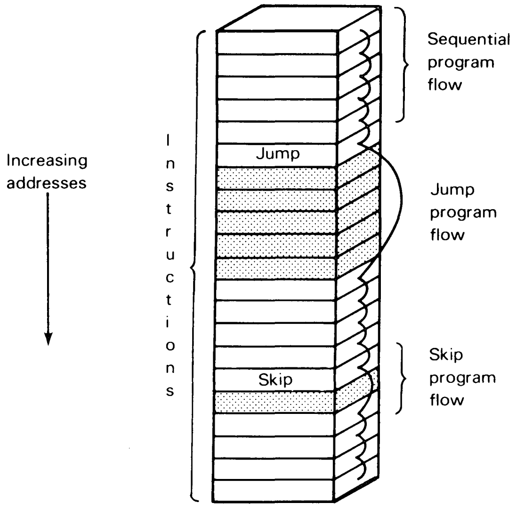
Each central processing unit (CPU) instruction is contained in two 16-bit words. Programs for the CPU consist of sequences of instructions stored in memory. The order in which these instructions are executed depends on the 15-bit memory address in the program counter (PC). During program execution, the CPU sends this address to memory and fetches the instruction contained in that memory location. During the execution of an instruction, information moves between the CPU internal registers and memory or between the CPU registers and input/output (I/O) device buffers.

When the instruction is completed, the CPU increments the PC by one and fetches the contents of the next sequential memory location. You can address the complete logical address space (locations 0 through 77777_8 inclusive).

To address the word following location 77777_8, address location 0. To address the word preceding location 0, address location 77777_8.

Program Flow Alteration

Sequential operation can be altered by a jump, a conditional skip instruction, or by program interrupts. Jump instructions load a new address into the PC, while conditional skip instructions increment the PC by two if the condition tested is true. In either case, sequential operation continues from the updated contents of the PC. Note that if you attempt to skip over an instruction which is multiple words in length, the second word of the instruction is executed as an instruction. Figure 6.1 illustrates program flow without interrupts.



DG-00543

Figure 6.1 Program flow without interrupts

Table 6.1 lists the program flow alteration instructions and gives a brief description of each.

Program Flow Alteration Instructions

The program flow alteration instruction set (Table 6.1) consists of instructions that change the contents of the PC either by specifying new contents for the PC or by causing the PC to be conditionally incremented by one (or by two if it is a 2-word instruction) and then continuing sequential operation with the updated PC.

Mnem	Instructions	Action
CLM	Compare To Limits	Compares a signed integer with two other numbers and skips if first integer is between the other two.
DSPA	Dispatch	Compares a signed integer with two other numbers and skips if first integer is not between the others; otherwise, uses the integer as an index into a table and places indexed value in the program counter.
HALT, HALTA	Halt	Stops program execution with programmed I/O and data channel interrupts enabled.
JMP, EJMP	Jump	Places an effective address in the program counter.
JSR, EJSR	Jump To Subroutine	Increments the program counter and stores the incremented value in AC3; then places a new address in the program counter.
SYC	System Call	Pushes a return block onto the stack; places address of System Call handler in the program counter.
VCT	Vector On Interrupting Device Code	Identifies highest priority interrupt; passes control through a table to a handler routine for the device.
XOP, XOP1	Extended Operation	Pushes a return block onto the stack, indexes into the **XOP** table, and transfers control to another procedure.

Table 6.1 Program flow alteration instructions

Conditional Skip Instructions

Table 6.2 presents the conditional skip instructions.

Mnem	Instructions	Action
DSZ, EDSZ	Decrement And Skip If Zero	Decrements the addressed word, then skips if the decremented value is zero.
ISZ, EISZ	Increment And Skip If Zero	Increments the addressed word, then skips if the incremented value is zero.
SGE	Skip If ACS Greater Than Or Equal To ACD	Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.
SGT	Skip If ACS Greater Than ACD	Compares two signed integers in accumulators; skips if the first is greater than the second.
SKP [*t*]	I/O Skip	Skips if the I/O condition *t* is true.
SNB	Skip On Nonzero Bit	References a single bit in memory through the bit pointer; skips if bit is 1.
SZB	Skip On Zero Bit	References a single bit in memory through the bit pointer; skips if bit is 0.
SZBO	Skip On Zero Bit, Set To 1	References a single bit in memory through the bit pointer; skips if bit is 0 and sets the bit to 1.

Table 6.2 Conditional skip instructions

Program Flow Interruption

When program interrupts occur (from I/O device controllers, the internal real-time clock, or a stack overflow error), the CPU stores the next sequential program address in location zero. Then the CPU loads the PC with the starting address of the interrupt handler routine and continues sequential operation. (See Program Interrupt in Chapter 7 for specific memory locations.)

Figure 6.2 illustrates program flow with interrupt.

Refer to Appendix D, Programming Examples, for an example and illustrations of a subroutine call and return.

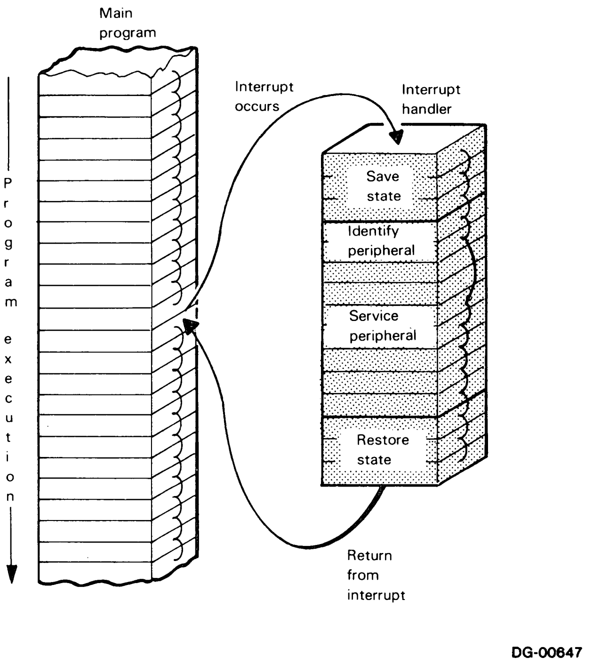


Figure 6.2 Program flow with interrupt

Extended Operation

The Extended Operation instruction (**XOP**) provides an efficient method of transferring control to and from procedures. It enables the user to transfer control to any one of 48 procedure entry points.

Extended Operation Instructions

There are two extended operation instructions in the ECLIPSE S/120 instruction set. They are shown in Table 6.3.

Mnem	Instructions	Action
XOP	Extended Operation	Pushes a return block on the stack; places the address of the specified accumulators into AC2 and AC3; and transfers control to 1 or 32 other procedures through the **XOP** table.
XOP1	Extended Operation	Same as **XOP** except that 32 is added to the entry number before entering the **XOP** table, and only 16 table entries can be specified.

Table 6.3 Extended operation instructions

Chapter 7

Device Management

I/O Management

Data may be transferred using two methods. The first method is between input/output (I/O) devices and accumulators using programmed I/O; the second is directly between I/O devices and memory, using data channel I/O.

Most I/O devices are controlled through the manipulation of Busy and Done flags. Flag values are changed through the use of optional flag command mnemonics. The effects of the flag commands are device, dependent.

When the Busy and Done flags are both zero, the I/O device is idle and cannot perform any operations. To start a device, the program must set Busy to one and Done to zero. When the device has finished its operation and is ready to start another, it sets Busy to zero and Done to one.

Programmed I/O

Programmed I/O transfers data one word at a time under direct program control. This type of I/O allows data to be examined individually as they are transferred.

Data Channel I/O

Data channel I/O permits data to be transferred in blocks of words, with program control necessary only at the start and end of the operation. The transfer is made directly to or from memory. Data channel I/O is an efficient method of transferring blocks of data between memory and an I/O device.

Data channel transfers are set up by a series of instructions that specify the following:

- address of the first word to transfer,
- direction of the transfer (read/write),
- total number of words to transfer.

When a data channel device is ready to send or receive data, it issues a data channel request. At the beginning of every memory cycle, the I/O channel synchronizes any requests that are then being made and controls the transfers between the I/O bus and the memory. When a request is honored, a word is transferred directly between the device and memory over the data channel.

All requests are honored according to the relative position of the requesting device on the I/O bus. Data channel service begins with the device that is physically closest on the bus to the CPU. The next closest device is serviced next, and so on, until all requests have been honored. New requests are synchronized concurrently with the servicing of older requests. If a device continually requests data channel service, it prevents all devices further out on the bus from gaining access to the channel.

The ECLIPSE S/120 data channel facility allows devices to transfer data to and from fast memory over the ECLIPSE I/O bus at speeds of approximately 2.0 megabytes per second for input and approximately 1.3 megabytes per second for output.

For more information on the data channel, see the *Peripherals Programmer's Reference Manual* (DGC No. 014-000632), and the *Interface Designer's Reference* for NOVA and ECLIPSE Line Computers (DGC No. 014-000629).

I/O Operations

The CPU can address one of as many as 64 I/O controllers connected to the ECLIPSE I/O bus by means of the device code occupying bits 10-15 of an I/O instruction. The basic I/O instruction set is used to control I/O devices, to set up data channel operation, and to pass data to and from these devices. These I/O instructions are listed in Tables 7.1 and 7.2.

Mnem	Instruction	Action
DIA	Data In A	Transfers data from the A buffer of an I/O device to an accumulator.
DIB	Data In B	Transfers data from the B buffer of an I/O device to an accumulator.
DIC	Data In C	Transfers data from the C buffer of an I/O device to an accumulator.
DIS	Data In Status	Returns the status of an I/O device to an accumulator.
DOA	Data Out A	Transfers data from an accumulator to the A buffer of an I/O device.
DOB	Data Out B	Transfers data from an accumulator to the B buffer of an I/O device.
DOC	Data Out C	Transfers data from an accumulator to the C buffer of an I/O device.
NIO	No I/O Transfer	Changes a flag without causing any other effect.
SKP	I/O Skip	Tests a flag and skips the next sequential word if the test condition is true.

Table 7.1 Standard I/O instructions

Mnem	Instruction	Action
INTA (DIB, CPU)	Interrupt Acknowledge	Returns the device code of an interrupting device.
INTDS (NIOC, CPU)	Interrupt Disable	Sets Interrupt On flag to zero.
INTEN (NIOS, CPU)	Interrupt Enable	Sets Interrupt On flag to one.
IORST (DIC, CPU)	Reset	Sets all Busy and Done flags and the priority mask to zero.
MSKO (DOB, CPU)	Mask Out	Changes the priority mask.
READS (DIA, CPU)	Read Switches	Places the contents of the virtual console register into an accumulator.
VCT	Vector on Interrupting Device	Identifies the highest priority interrupt and passes control through a table to a device handler.

Table 7.2 CPU I/O instructions

I/O Format

The format for the I/O instructions is:

Mnemonic*[f]* *ac,***device**

which assembles as:

0	1	1	AC	OP CODE	CTRL	DEVICE CODE
0	1	2	3 4	5 7	8 9	10 15

Bits	Function
0-2	Always 011.
3,4	Specify accumulator.
5-7	Specify operation code.
8,9	Control certain flags in the device.
10-15	Specify the device code.

I/O instructions are encoded in the following four fields. They are Accumulator, Instruction, Control, and Device.

Accumulator Field (bits 3-4) — specifies one of the four accumulators that will either contain the data to be moved to a device or receive the data from the device.

Instruction Field (bits 5-7) — specifies the I/O instruction (what is to be done with the data; that is, move data from an accumulator to a device, move data from a device to an accumulator, do nothing with the data, or test the device flags).

Control Field (bits 8-9) — specifies *f* or *t* depending upon the type of instruction. The *f* function can be specified for all I/O instructions except *Skip.* An *f* function tells the CPU to manipulate the state of the Busy and Done flags of an I/O device. The Start, Clear, or Pulse commands specified in the control field by *f* are shown in Table 7.3. Refer to the specific instruction entries in the I/O instruction dictionary at the end of this chapter.

> **NOTE:** *If an attempt is made to test the status of the Busy and Done flags of a nonexistent device, the CPU recognizes the Busy and Done flag bits as zero.*

Optional Mnemonic	Sets Bits 8-9 to	Instruction	Action
—	00		Does not affect the Busy and Done flags.
f=S	01	Start	Start the device by setting Busy to one and Done to zero.
f=C	10	Clear	Idle the device by setting both Busy and Done to zero.
f=P	11	Pulse	The effect depends on the device.

Table 7.3 I/O control flags

The *t* function can be specified with the Skip instructions to cause the instruction to perform tests on the Busy and Done flags of an I/O device. Table 7.4 lists the possible test conditions and mnemonics for each. Refer to the **SKP, CPU** entry in the I/O instruction dictionary at the end of this chapter.

Optional Mnemonic	Sets Bits 8-9 to	Action
t=BN	00	Tests for Busy = 1
t=BZ	01	Tests for Busy = 0
t=DN	10	Tests for Done = 1
t=DZ	11	Tests for Done = 0

Table 7.4 I/O test flag (SKP instruction only)

Device Field (bits 10-15) — specifies one of the possible 64 I/O devices to be addressed. A complete list of assembler-recognizable mnemonics assigned by Data General is provided in Appendix A.

Refer to *Peripherals Programmer's Reference Manual* (DGC No. 014-000632) for details about programming specific devices.

Program Interrupt

The I/O interrupt system in the S/120 computer manages programmed interrupts by permitting the program to ignore I/O devices until one requires service. After handling all data channel requests, the processor completes execution of any incomplete instruction, services any further data channel requests that were synchronized while the instruction was executing, then services outstanding I/O interrupt requests. When all requests have been serviced, program execution continues.

I/O interrupt control instructions offer the programmer the following selection of I/O control schemes:

- No interrupts— the CPU checks I/O device status under programmed control.
- Interrupts with no priority system— the CPU services one device at a time in the order determined by the timing of the interrupt and the physical location of its controller in the computer chassis.
- Interrupts with a priority system— the CPU services an interrupt from a selected device in the order described above, but a higher priority device can interrupt a lower priority device's interrupt service routine. The interrupt handler does this by manipulating the devices priority mask bits using the **MSKO** instruction.

The following aspects of program interrupt are discussed below:

- Initiating an interrupt,
- Servicing an interrupt,
- Vectored interrupt,
- Control flags,
- Priority interrupts,
- Dismissing an interrupt.

Initiating an Interrupt and CPU Response

When a device requires service, it initiates a program interrupt request by setting the Done flag to one. If the device's Interrupt Disable flag is set to zero, the CPU receives the request. If the Interrupt Disable flag is set to one, the device will not request service until the device's Interrupt Disable flag is reset.

If the interrupts are enabled, that means that the **ION** flag is set to one. When that occurs, the CPU will service a program interrupt upon completion of an instruction or a data channel request.

The CPU responds to a program interrupt request as follows:

- Sets the **ION** flag to zero to protect against further interrupts.
- Stores the contents of the program counter in location zero so that the interrupted program can resume after servicing the interrupt.
- Jumps indirect through location one.

Servicing an Interrupt

The purpose of an interrupt service handler is to

- Save the state of the CPU,
- Identify the interrupting device,
- Transfer control to the appropriate service routine,
- Restore the state of the CPU.

Identifying the Interrupting Device

If interrupt-driven operation is selected, the programmer can select one of the following methods of identifying the interrupting device:

- Placing the interrupter's device code in an accumulator with an **INTA** instruction. (See the **INTA** instruction in the I/O dictionary.)
- Identifying the interrupting device, saving return information and jumping through a table to an individual device's interrupt handling routine with a **VCT** instruction.
- Testing the device's control flags with an *I/O Skip* (**SKP**) instruction.

Vectored Interrupt

The *Vector On Interrupting Device Code* (**VCT**) instruction can simplify the design of an interrupt handler by doing many of the required steps in one instruction. It can also perform different levels of tasks as needed within the interrupt handler.

The **VCT** instruction has five different modes that can be used in different circumstances. Refer to the **VCT** instruction in Chapter 10. Depending on the selected mode, it can perform any or all of the following operations:

- Save the state of the computer,
- Store the user stack parameters,
- Create a new stack,
- Reset the priority mask.

When selecting one of these modes, you must weigh the importance of its operations against the time used for each interrupt. You are not committed to one mode throughout the interrupt handler. It is possible to use different **VCT** instruction modes at different times to serve different needs.

Mode A is used when a device requires immediate interrupt service. This would be the case for unbuffered devices with very short latency times, or for real-time processes that require immediate access. This mode does not save any information on the state of the computer.

Modes B through E create a priority structure which permits some interrupting devices to interrupt the service of certain others. This takes more time than mode A service, but it permits immediate service for some devices even if a slower device is already being serviced.

Modes D and E both initiate a new stack. You should use them only when operating in an unmapped mode, since they set up a new vector stack for use by the interrupt handler and store the (old) user stack parameters in it. Once this new stack has been set up, there is no reason to try to set it up again if a new interrupt occurs before the old one has finished. Mode E also pushes a return block onto the stack to make a return to the first interrupt handler easier. Modes B and C do not initiate a new stack and are therefore appropriate to use when a device interrupts the interrupt processing of another device (mapped mode). Mode C also pushes a new return block onto the stack.

Interrupt On Flag

The Interrupt On (**ION**) flag in the CPU indicates the status of the interrupt system. When the **ION** flag is set to one, the interrupt facility is enabled and the CPU can respond to an interrupt request. When the **ION** flag is set to zero, the interrupt facility is disabled and the CPU ignores all interrupt requests. The *f* function of any I/O instruction addressed to the CPU (device code 77_8) sets the **ION** flags as shown in Table 7.5. The Skip instruction addressed to the CPU tests the state of **ION** as shown in Table 7.6.

Mnem	Sets Bits 8-9 to	Instruction	Action
—	00	None	Does not affect the state of the Interrupt On flag.
f= S	01	Start	Set the Interrupt On flag to one.
f= C	10	Clear	Set the Interrupt On flag to zero.
f= P	11	Pulse	Clears virtual console interrupts. Does not affect the state of the Interrupt On flag. Used only with VCT.

Table 7.5 Setting Interrupt On (ION) flag in CPU (device 77_8)

Mnem	Sets Bits 8 and 9 to	Action
t=DN	10	Skip next sequential instruction if Power Fail equals one (power is failing).
t=DZ	11	Skip next sequential instruction if Power Fail equals zero.
t=BN	00	Skip next sequential instruction if Interrupt On equals one.
t=BZ	01	Skip next sequential instruction if Interrupt On equals zero.

Table 7.6 Testing Interrupt On (ION) flag in CPU (device 77_8)

Priority Interrupts

The CPU services an interrupt from a selected device in the order determined by the timing of the interrupt and the physical location of its controller in the ECLIPSE S/120 chassis. A higher priority device can interrupt a lower priority device's interrupt service routine. The interrupt handler does this by manipulating the device's priority mask bits using the *Mask Out* instruction. Refer to the **MSKO** entry in this chapter's I/O instruction dictionary.

Interrupt Priority Mask

The interrupt priority mask is a 16-bit word. Each I/O device in a system is assigned a mask bit that governs the device's Interrupt Disable flag. When the **MSKO** instruction sets a particular bit in the mask to one, the Interrupt Disable flag in the corresponding device is set to one and the device is masked out or disabled. This means that the device cannot generate program interrupts. Those devices whose corresponding mask bits are zero have a higher priority than the device being serviced. The CPU interrupts service to the lower priority device to honor an interrupt request from a higher priority device.

The mask bits are assigned to the devices on the hardware level, so you cannot program them. You can, however, control the order of priority of these bits. In your program, you can use the priority mask to rank your I/O devices in any order. Note that certain I/O devices which operate at roughly the same speed are assigned the same mask bit; these devices will always have the same priority. Appendix A lists the mask bit assignments in addition to the device code assignments.

Priority Interrupt Handler

To use a priority interrupt instruction in your system, the interrupt handler must be re-entrant. This means that if a device service routine is interrupted by a higher priority device, there will be no loss of the information the handler needs to restore the state of the machine. For a handler to be re-entrant, it must be able to save the contents of location 0 (the return address) and the current priority mask each time it is entered at a higher level. It should also be able to perform the following sequence of operations.

- Save the state of the processor (accumulators, carry, stack pointer and frame pointer, contents of location 0, and the current priority mask),
- Check for current number of interrupts,
- Identify the device requesting the interrupt,
- Transfer control to the interrupting device's service routine,
- Establish and store a new priority mask,
- Enable interrupts,
- Service the device,
- Disable interrupts,
- Restore the state of the processor,
- Enable interrupts,
- Transfer control to the return address saved from location 0.

To establish a system of priorities, place a *Mask Out* (**MSKO**) instruction in the interrupt service handler for each device. This instruction changes the priority mask, thus controlling which devices can interrupt. Devices that should not interrupt the device being serviced are masked out if their mask bits are set to 1. In addition, all pending interrupt requests from devices controlled by that bit are disabled. The other mask bits, corresponding to devices that can interrupt, are set to 0.

Dismissing An Interrupt

After servicing the interrupting device, either the peripheral service routine or the main interrupt handler should perform the following sequence of events:

- Clear the device's Done flag to dismiss the interrupt just honored. (If you leave this out, the undismissed interrupt will cause another interrupt when you attempt to transfer control back to the interrupted program.)
- Restore the state of the CPU.
- Set the **ION** flag to one to enable interrupts. (Although interrupts are enabled with one instruction, the CPU will not respond to an interrupt request until the next instruction executes.)
- Return to the interrupted program. (This usually is done by a jump indirect through location zero, since this is where the CPU placed the valueof the program counter when it began to service the interrupt.)

SPU I/O Management

The S/120 system processing unit contains the central processing unit (CPU). Table 7.7 lists the I/O instructions which deal with the CPU. All of the instructions grouped in this table are addressed to device code 77_8.

Mnem	Instruction	Action
DIS CPU	Read Processor Status	Returns the status of the processor, including the following conditions: power fail interrupt on, break key reset, power-up reset, halt instructions, interrupt request, and program load key.
DOAP CPU	CPU Acknowledge	Clears and sets virtual console interrupts.
HALTA (DOC, CPU)	Halt	Stops the processor.
INTA (DIB, CPU)	Interrupt Acknowledge	Returns the device code of an interrupting device.
INTDS (NIOC, CPU)	Interrupt Disable	Sets the CPU Interrupt On flag to zero.
INTEN (NIOS, CPU)	Interrupt Enable	Sets the CPU Interrupt On flag to one.
IORST (DIC, CPU)	Reset	Sets all Busy and Done flags and the priority mask to zero.
MSKO (DOB, CPU)	Mask Out	Changes the priority mask.
READS (DIA[*f*]CPU)	Read Switches	Places the contents of the virtual console register into an accumulator.
SKP, CPU	CPU Skip	Tests the Interrupt On or Power Fail flag and skips the next sequential word if the test condition is true.

Table 7.7 CPU device instructions

Special Mnemonics

Most of the instructions in Table 7.7 have two forms. If you use the standard form of the instruction, **DOB**, for example, then you can specify a function *f* to act upon the **ION** flag. If you use the special form of the instruction **MSKO**, you cannot specify a function *f* to act upon the **ION** flag. Refer to the specific instruction entries in this chapter's I/O instruction dictionary for more information.

The SPU contains three I/O interfaces:

- An asynchronous communications interface (TTI, TTO).
- A programmable real-time clock interface (RTC).
- A programmable interval timer (PIT).

The CPU responds to TTI, TTO, RTC, and PIT as it does to any I/O device.

Asynchronous Interface

The asynchronous interface is a programmed I/O controller which contains both a double-buffered transmitter and receiver, allowing full-duplex communications between the CPU and a serial, asynchronous terminal.

> **NOTE:** *The S/120 asynchronous communications interface receives and transmits 8-bit data characters without parity. If the system console device being used with the S/120 operates with a data character length of seven bits, you should configure the device to operate with "mark parity." When receiving data characters from a 7-bit system console device, software should mask out the parity bit after the character has been loaded into an accumulator. The parity bit is the most significant bit of the character and is contained in bit 8 of the specified accumulator.*

Controller Registers

The following registers are available to the program:

An 8-bit receiver-holding register,
An 8-bit transmitter-holding register.

The receiver-holding register stores the assembled character that is received over the communications line in serial form. It makes the character available to the program until the receiver overwrites the contents of the register with the next assembled character.

The transmitter-holding register stores the character sent to the interface by the program. When *Clear To Send* is asserted by the terminal, the transmitter disassembles the character and sends it over the communications line in serial form.

Instruction Set

Device Mnemonic	
Transmitter	**TTO**
Receiver	**TTI**
Device Code	
Transmitter	11_8
Receiver	10_8
Priority Mask Bit	
Transmitter	15
Receiver	14

A list of asynchronous interface instructions are shown in Table 7.8.

Mnemonic	Instruction	Action
DIA[*f*] *ac*, TTI	Read Character	Reads a character received by the command line.
DOA[*f*] *ac*, TTO	Write Character	Writes a character to the line.

Table 7.8 Asynchronous interface instructions

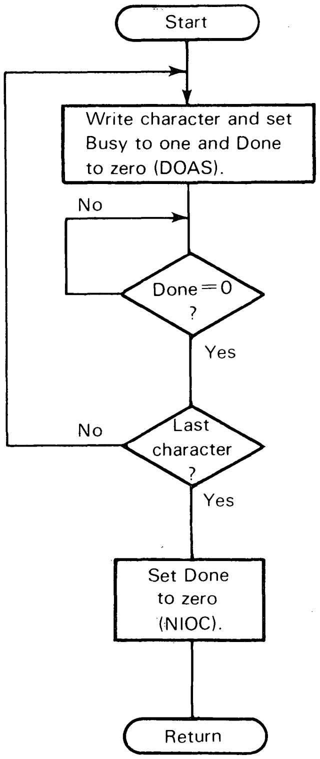
Programming

The transmitter (TTO) and receiver (TTI) function as separate devices. Each has its own set of Busy and Done flags which are manipulated both by the program and the devices.

The asynchronous interface controller transmits and receives 8-bit characters.

Programming the interface consists of:

Writing characters,
Reading characters.



DG-09006

Figure 7.1 Write character

Writing Characters

Before sending a character to the transmitter, check its Busy flag. If it is one, wait until it is zero. When Busy is zero, issue a *Write Character* instruction with a Start command (**DOAS**).

Reading Characters

To initiate character reception, use the *No I/O Transfer* instruction with a Start command (**NIOS**). This sets the receiver's Busy flag to one and Done flag to zero. When the receiver has a character for the program, it can then set the Done flag interrupt to one and initiate an interrupt request if its Interrupt Disable flag is zero.

When Done is one, issue a *Read Character* instruction with either a Start (**DIAS**) or Clear (**DIAC**) command. This loads the character in the receiver holding register into the specified accumulator. The Start command restarts the receiver. The Clear command terminates reception by setting both Busy and Done to zero.

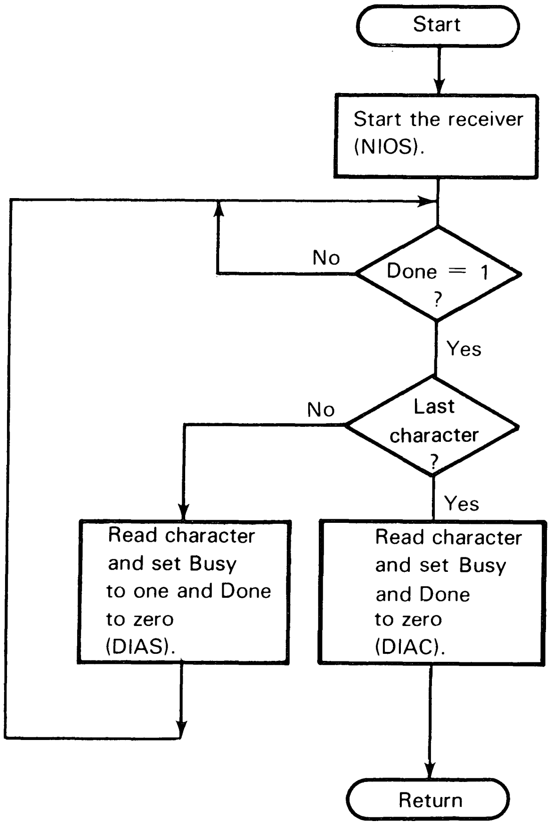
> **NOTE:** *The S/120 asynchronous communications interface receives and transmits 8-bit data characters without parity. If the system console device being used with the S/120 operates with a data character length of seven bits, you should configure the device to operate with "mark parity." When receiving data characters from a 7-bit system console device, software should mask out the parity bit after the character has been loaded into an accumulator. The parity bit is the most significant bit of the character and is contained in bit 8 of the specified accumulator.*

Timing

After the receiver Done flag is set to one, the character in the receiver holding register is available to the program for a time interval determined by the transmission rate (baud). To avoid possible data loss, the program must respond to the interrupt request by reading the character within the time interval indicated in Table 7.9. The time intervals tabulated are based on the assumption that characters transmitted at 50 to 110 baud contain 10 bits (including 2 stop bits), and characters transmitted at 134.5 to 38,400 baud contain 9 bits (including 1 stop bit).

Baud	Maximum Allowable Programmed I/O Latency (ms)
50	219.00
75	146.00
110	100.00
134.5	74.35
150	66.66
200	54.75
300	33.33
600	16.66
1200	8.33
1800	5.55
2000	5.00
2400	4.16
4800	2.08
9600	1.04
19,200	0.52
38,400	0.26

Table 7.9 Timing considerations


Start
Start the receiver (NIOS).
No
Done = 1 ?
Yes
No
Last character ?
Yes
Read character and set Busy to one and Done to zero (DIAS).
Read character and set Busy and Done to zero (DIAC).
Return

DG-08311

Figure 7.2 Read character

After the transmitter Done flag is set to one, the program should provide another character within the time period indicated in Table 7.10 to maintain the maximum transmission rate.

Power-Up Response

After power up, the transmitter Busy and Done flags and the receiver Busy and Done flags are zero.

Real-Time Clock Interface

The real-time clock provides a programmable selection of precise time bases for the S/120 computer system. Four frequencies are available: 10 Hz, 100 Hz, 1000 Hz, and line frequency.

Controller Registers

The interface contains a 2-bit frequency select register that is program-accessible. This register holds a 2-bit code that selects one of the four available frequencies: 10 Hz, 100 Hz, 1000 Hz, and line frequency.

Instruction Set

Device Mnemonic	RTC
Device Code	14_8
Priority Mask Bit	13
Frequencies	Line, 10 Hz, 100 Hz, 1000 Hz

A single I/O instruction programs the real-time clock. The real-time clock instruction is shown in Table 7.10. For more details refer to the I/O instruction dictionary at the end of this chapter. The **DOA***[f] ac*, **RTC** instruction loads the appropriate two bits of the specified accumulator field into the frequency select register.

The real-time clock is controlled by manipulating the interface Busy and Done flags.

Mnemonic	Instruction	Action
DOA*[f] ac*, RTC	Select RTC Frequency	Selects the frequency of RTC interrupts

Table 7.10 Real-time clock instruction

Mnem	Set Bits 8 and 9 to	Instruction	Action
—	00	—	No effect
f=S	01	Start	Enables RTC
f=C	10	Clear	Disables RTC
f=P	11	Pulse	No effect

Table 7.11 RTC flag commands

Programming

Programming the real-time clock consists of

- Selecting the real-time clock frequency,
- Enabling real-time clock interrupt requests,
- Servicing real-time clock interrupt requests.

To select the clock frequency, issue a *Select Frequency* instruction (**DOA**) *[f] ac*, **RTC**.

To enable a real-time clock interrupt request, issue a Start command to set Busy to one and Done to zero. Since the clock is free-running, the generation of the interrupt request may occur at any time up to one clock period after the Busy flag is set to one. When the clock period expires, the real-time clock sets Busy to zero and Done to one, thus initiating an interrupt request if the interface's Interrupt Disable flag is reset.

When servicing real-time clock interrupt requests, issue a *No I/O Transfer* instruction with either a Start command (**NIOS**) or a Clear command (**NIOC**) instruction. The Start command enables an interrupt request at the expiration of the current clock period. The Clear command inhibits subsequent interrupt requests by setting both Busy and Done to zero.

Timing

The first interrupt request initiated by the real-time clock can occur at any time up to the full clock period. If the program responds to real-time clock interrupt requests before each succeeding clock period expires, all subsequent RTC interrupt requests will occur at the clock frequency.

Power-Up Response

After power up or when an *I/O Reset* (**IORST**) instruction is performed, the line frequency rate is selected and both the Busy and Done flags Bre zero.

Programmable Interval Timer

The programmable interval timer (PIT) is a CPU-independent time base which can be programmed to initiate program interrupts at fixed intervals ranging from 100 microseconds to 6.5536 seconds in increments of 100 microseconds. The PIT can also be jumpered to run at other intervals. It can also be sampled with I/O instructions at any point in its cycle to determine the time until the next interrupt. The PIT is used in multiprogram operating systems to allocate CPU time to different programs on a "time slice" basis.

Controller Registers

The PIT consists of a 16-bit count register and a 16-bit counter. During operation, the PIT counter is loaded with the contents of the initial count register and is then incremented at 100 microsecond intervals. A Busy and a Done flag control the operation of the device. Table 7.12 lists the instructions used to program the PIT.

Instruction Set

Device Mnemonic	PIT
Device Code	43_8
Piority Mask Bit	6

Mnemonic	Instruction	Action
DOA PIT	Specify Initial Count	Selects the value which will be loaded into the counter each time the PIT is started.
DIA PIT	Read Count	Reads the current value of the PIT counter.

Table 7.12 PIT instructions

Programming

To obtain a particular time interval between program interrupt requests, load the two's complement of the number of clock intervals between interrupt requests into the initial count register. Refer to Table 7.13 for a list of PIT rates. The time between program interrupt requests will be the value selected by the contents of the initial count register. Refer to the specific instruction entries in the I/O instruction dictionary in this chapter. For the counter rate switch settings, refer to the *ECLIPSE S/120 Computer System, Hardware Reference Series* (DGC No. 014-000690).

	Time Interval	
Selected Frequency (KHZ)	Minimum	Maximum
1000	1 microsecond	65.536 milliseconds
100	10 microseconds	655.36 milliseconds
10	100 microseconds	6.5536 seconds
1	1 millisecond	65.536 seconds

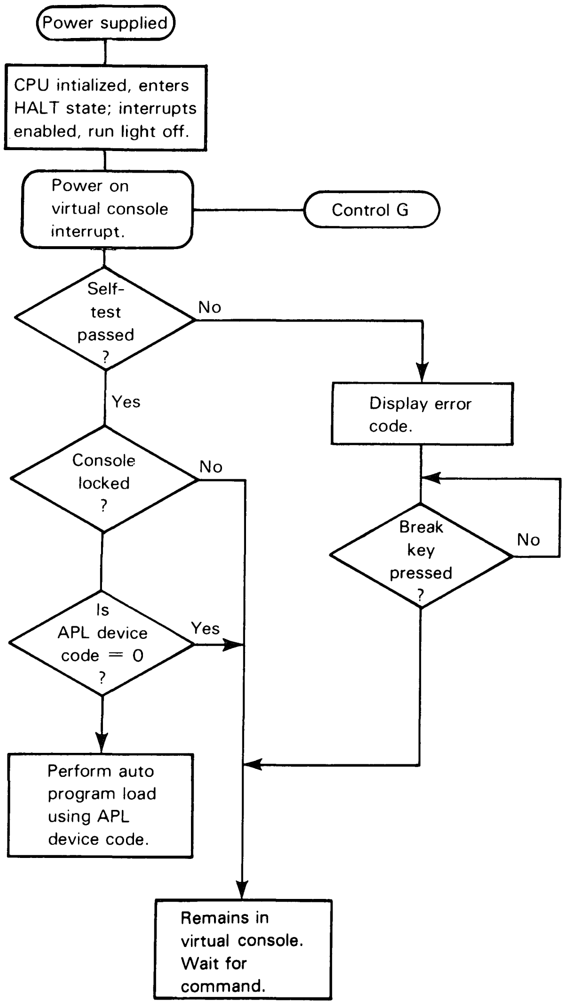
Table 7.13 PIT rates

SPU I/O Operation

Power-Up Sequence

After power is applied to the System Processing Unit (SPU), the CPU is initialized and enters the HALT state. At this time, all Busy and Done flags are set to zero; I/O interface registers are cleared; the state machines in the ECLIPSE I/O interface are initialized; bits 0, 9 through 12, 13, 14, and 15 of the MAP status register are cleared; and the CPU status register is cleared, except for the power-up bit (bit 4), which is set to one.

From the HALT state, the CPU enters the virtual console, which clears the power-up bit and performs a short — about 0.75-second duration — diagnostic routine. If an error is detected, an error message is displayed. (The error messages are described in Chapter 9, "Virtual Console.")


Power supplied
CPU intialized, enters HALT state; interrupts enabled, run light off.
Power on virtual console interrupt.
Control G
Self-test passed ?
No
Yes
Display error code.
Console locked ?
No
Break key pressed ?
No
Is APL device code = 0 ?
Yes
Perform auto program load using APL device code.
Remains in virtual console. Wait for command.

DG-08657

Figure 7.3 Power-up sequence for the ECLIPSE S/120 CPU

As shown in Figure 7.3, after power-up the CPU remains in the virtual console if the control panel is unlocked or if the automatic program load (APL) device code is equal to zero. If the panel is locked, an automatic program load is performed from the device specified by the APL jumpers.

After a program load has been performed, the device code of the device loaded from is placed in the virtual console switch register. A **READS** instruction can then be used to return the device code.

Auto Program Load Register

The auto load register indicates which device boots the system. The auto program load register definitions are listed in Table 7.14. Table 7.15 lists the instructions to write output register. When the local output register is written to, it turns on the front console run light. Refer to the **READS** instruction in the I/O instruction dictionary. For more information on auto program load, refer to Chapter 9, "Virtual Console."

Device Code	1_8
Priority Mask Bit	None

Mnem	Bit	Instruction	Action (If Set to 1)
DCH	0	Data Channel	Auto load program is loaded from the data channel.
LCK	1	Lock	Front panel is locked.
DLY	2	Delay	A one-minute delay in the execution of the auto load.
—	3-9	—	Reserved.
—	10-15	Device Code	Specifies device to perform the auto load.

Table 7.14 Read auto load register definitions

Mnem	Bit	Instruction	Action (If Set to 1)
R	0-14	Reserved	Reserved for future use.
LGT	15	Run Light	Turns on front panel run light.

Table 7.15 Write to local output register instructions

Power Fail/Autorestart

When line power loss is imminent, the SPU receives a power-fail interrupt request. When an abnormal power condition occurs, the power supply asserts a signal that causes the system I/O IC to issue an interrupt request to the CPU. The CPU should respond to the request by entering a user-supplied, power fail interrupt routine. A typical power fail routine saves the state of the processor and loads a return instruction into physical location 0, and halts. An example of such a routine is given in the section "Programming Example." The power fail routine can do one of two things:

The user powerfail routine can simply execute a **Halt** instruction, or if battery backup is present, the powerfail routine should save the state of the processor in system memory, load a return instruction into physical memory location 0 and then execute a **Halt** instruction. If no battery backup is present, the routine should simply halt the processor. If the **Halt** instruction is executed within one millisecond and if the Halt Dispatch SIO operating characteristic is selected (refer to SIO jumpering in the *ECLIPSE S/120 Computer System Hardware Reference* (DGC 014-000690) the virtual console is entered, otherwise a hard halt occurs.

As long as the batteries have not been exhausted during the power disruption, the virtual console will automatically clear the powerfail interrupt when power is restored. The action taken by the virtual console then depends on whether the control panel is locked. If the control panel is locked, then the instruction contained in physical memory location 0 is executed, returning control to the user's program. If the control panel is unlocked, the virtual console will retain control and issue a prompt character to the system console when power returns. The user can then continue by typing **R** on the system console (refer to the virtual console chapter), if all of the following conditions are true:

- Battery backup is present,
- The state of the processor is saved,
- A return instruction is stored in physical memory location 0 when the power failure occurred.

Otherwise, the user can initiate a program load or take other appropriate action. If the batteries have been exhausted the virtual console performs the power-up response as shown in Figure 7.3. (Refer to the Power-Up Sequence section in this chapter) The battery backup, when present and fully charged, will maintain system memory validity for a minimum of one hour.

Error Checking and Correction

The ECLIPSE S/120 error checking and correction facility is designed for applications requiring either a high degree of reliability for the main memory of a system, or a "fail-soft" capability in the event of memory errors. The ERCC facility will detect and correct all single-bit memory errors.

The error checking and correction (ERCC) facility generates and appends a 6-bit check code to each word (2 bytes) of data written to memory. During memory read operations, the ERCC facility processes the 22-bit memory word to determine if an error has occurred. If a single-bit error has occurred, the erronous bit is corrected before the word is transferred. The corrected word is also written to memory along with a new check code. In addition, the fault address and an error syndrome code are recorded for transfer to the CPU. An interrupt request may be issued. This fault address and error syndrome code can be used to identify a marginal or failing system memory RAM chip.

Double-bit and some triple-bit errors are detected, but they are not corrected. However, their fault address and error syndrome codes are recorded and an interrupt request may be issued.

In a process called *sniffing*, the ERCC facility also detects and corrects single-bit errors during memory refresh cycles (when enabled). However, their fault addresses and error syndrome codes are not recorded, and no interrupt is issued.

ERCC Instructions

One I/O instruction sets the mode of operation of the ERCC facility. ERCC contains a Done flag which is set to 1 after an error has been detected and the ERCC initiates an interrupt request. Two instructions interrogate ERCC after the detection and correction of an error.

The ERCC facility has no Busy flag and no mask bit in the priority mask. The device code for the ERCC facility is 2. The assembler recognizes the mnemonic **ERCC** for this device code.

ERCC instructions use a specified accumulator to receive data or contain the control information.

Table 7.16 shows the ERCC instructions.

Mnem	Instruction	Action
DOA	Enable ERCC	Enables the ERCC facility according to the setting of bits 13-15 of the specified accumulator.
DIA	Read Memory Fault Address	Returns the low-order bits of the memory location which has produced an error.
DIB	Read Memory Fault Code	Returns a 6-bit error code that tells which bit was in error. Also returns the two most significant bits of the memory fault address.

Table 7.16 ERCC instructions

Power Fail Instructions

The power fail instructions test the state of the power fail flag. They use device code 77_8.

The power fail facility has no priority mask bit in the priority mask. It responds to **INTA** and **VCT** instructions with device code 0.

Table 7.17 references the power fail instructions. Refer to the *CPU Skip* (**SKP CPU**) instruction entry in the Instruction Dictionary for more information.

Mnemonic	Instruction	Action
SKPDN CPU	Skip if Power Flag is One	Skip if power failure has occurred.
SKPDZ CPU	Skip if Power Flag is Zero	Skip if power is ON.

Table 7.17 Power fail instructions

Programming Example

The following sequence of instructions is an example of a power fail/autorestart program routine.

```
IHANP:  PSH     0,3      ;save accumulators
        SUBCL   0,0      ;save carry
        PSH     0,0
        SKPDN   CPU
        JMP     NPFI     ;not power fail interrupt
        LDA     0,0      ;get program counter
        PSH     0,0
        LDA     0,RSI    ;get restart instruction
        STA     0,0
        ELEF    0,PFAR   ;get restart address
        PSH     0,0      ;save on stack
        HALT             ;halt
PFAR:   POP     0,0      ;get address of interrupt
        STA     0,0      ;save
        POP     0,0
        MOVR    0,0      ;restore carry
        POP     3,0      ;restore accumulators
        INTEN            ;enable interrupt system
        JMP     @0,0     ;return to program
RSI:    POPJ             ;restart instruction
```

I/O Instruction Dictionary

This dictionary lists the I/O instructions for the S/120 computer. The functional grouping of I/O instructions are arranged in alphabetical order by mnemonic. Each instruction entry includes

- the mnemonic recognized by the assembler,
- the number and format of any arguments,
- the bit format,
- the description of how the instruction works.

In general, I/O instructions can be executed only when both LEF mode and I/O protection are disabled. See the Load Effective Address Mode section in Chapter 8.

> **NOTE:** *Instructions such as* **XCT**, **RSTR**, *or* **POPB** *which may be used with I/O instructions are listed in the general instruction dictionary in Chapter 10.*

Standard I/O Instructions

You can use the following standard I/O instructions with any I/O device, using the appropriate device code.

Table 7.18 lists optional mnemonics which you may want to add to some of your I/O instructions. These mnemonics are explained in detail earlier in this chapter; the instruction entries will tell you whether you can use one of these optional mnemonics with an instruction.

Optional Mnemonic	Sets Bits 8-9 to	Instruction	Action
—	00		Leaves device's Busy and and Done flags unchanged.
f=S	01	Start	Starts the device by setting Busy to one and Done to zero.
f=C	10	Clear	Idles the device by setting both Busy and Done to zero.
f=P	11	Pulse	The effect depends on the device.

Table 7.18 Optional I/O mnemonics

Data In A

DIA[*f*] *ac,device*

0	1	1	AC	0	0	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from the A buffer of an I/O device to an accumulator.

The contents of the A input buffer in the specified device are placed in the specified AC. Sets the Busy and Done flags according to the function specified by *f*.

The format of the AC after the transfer is device, dependent.

If the specified device does not exist, the AC will contain 177777_8 after the transfer.

Data In B

DIB[*f*] *ac,device*

0	1	1	AC	0	1	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from the B buffer of an I/O device to an accumulator.

Places the contents of the B input buffer of the specified device in the specified AC. Sets the Busy and Done flags according to the function specified by *f*.

The format of the AC after the transfer is device dependent. If the specified device does not exist, the AC contains 177777_8 after the transfer.

Data In C

DIC[*f*] *ac,device*

0	1	1	AC	1	0	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from the C buffer of an I/O device to an accumulator.

Places the contents of the C input buffer in the specified device in the specified AC. Sets the Busy and Done flags according to the function specified by *f*.

The format of the AC after the transfer is device dependent. If the specified device does not exist, the AC contains 177777_8 after the transfer.

Data In Status

DIS *ac,device*

0	1	1	AC	1	1	1	0	0	DEVICE CODE
0	1	2	3 4	5	6	7	8	9	10 15

Returns the status of the addressed device and places this data into the specified accumulator.

The accumulator must be specified as 1, 2, or 3. The **DIS** instruction uses the same operation code as the **SKP** instruction. **DIS** 0 **CPU** is equivalent to the **SKP** 0 **CPU** instruction.

The information contained in the specified accumulator is in the following format for I/O devices:

D	B	1	RESERVED FOR FUTURE USE
0	1	2	3 15

D Device is done if set to one.
B Device is busy if set to one.

> **NOTE:** *If the device does not exist, the contents of the specified AC will be* 037777_8.

Data Out A

DOA*[f]* *ac,device*

0	1	1	AC	0	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from an accumulator to the A buffer of an I/O device.

Places the contents of the specified AC in the A output buffer of the specified device. Sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

Data Out B

DOB*[f]* *ac,device*

0	1	1	AC	1	0	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from an accumulator to the B buffer of an I/O device.

Places the contents of the specified AC in the B output buffer of the specified device. Sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

Data Out C

DOC*[f]* *ac,device*

0	1	1	AC	1	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from an accumulator to the C buffer of an I/O device.

Places the contents of the specified AC in the C output buffer of the specified device. Sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

No I/O Transfer

NIO *[f]* *ac,device*

0	1	1	0	0	0	0	0	F	DEVICE CODE
0	1	2	3	4	5	6	7	8 9	10 15

Used when a Busy or Done flag must be changed with no other operation taking place.

Sets the Busy and Done flags in the specified device according to the function specified by *f*.

I/O Skip

SKP *[t] device*

0	1	1	0	0	1	1	1	T	DEVICE CODE
0	1	2	3	4	5	6	7	8 9	10 15

If the test condition, *t*, is true for the device specified by the device code, the instruction skips the next sequential word. The possible values of *t* are listed in the table.

Mnemonic	Value	Test
BN	00	Test for Busy = 1
BZ	01	Test for Busy = 0
DN	10	Test for Done = 1
DZ	11	Test for Done = 0

Instruction: **SKPBZ** TTO
Checks the setting of the Teletype Busy flag. If the Busy flag is set to 0 (Teletype is busy), the next sequential word is skipped.

CPU Device Instructions

I/O instructions addressed to device code 77_8 are grouped as CPU device instructions.

Priority Mask Bit — None

CPU Status

DIS*[f]* *ac,***CPU**

0	1	1	AC		1	1	1	0	0	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Returns the status of the CPU status register and places this data into the specified accumulator.

> **NOTE: DIS** *0* **CPU** *is equivalent to the* **SKP** *0* **CPU** *instruction.*

The information contained in the specified accumulator is in the format:

PF	ION	1	BRK	POP	HLT	DH	IRQ	PL	-	TRP	SUP	MEM		-	DG
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Mnem	Bit	Name	Function (If Set to 1)
PF	0	Power Fail	Power Fail flag set.
ION	1	Interrupt On	Interrupt On flag set.
1	2	—	Set to one.
BRK	3	Break Key Interrupt	Interrupt resulting from depression of console Break key. Used only by the virtual console.
PUP	4	Power Up Reset	Power came up since last DOAP CPU. Used only by the virtual console.
HLT	5	Halt	**Halt** instruction was executed. Used only by the virtual console.
DH	6	Halt Dispatch	If one, indicates that a Halt instruction will cause the processor to enter the virtual console. If zero, indicates that a Halt instruction will cause the processor to halt.
IRQ	7	Interrupt	An interrupt is being requested.
PL	8	Program Load	Indicates the program load console key has been depressed.
—	9	—	Undefined.
TRP	10	Trap	Indicates that the virtual console single-instruction mode is in use.
SUR	11	Survived Memory Data Valid	Indicates that memory data is valid following a power disruption.
MEM TYP	12-13	Memory Type	Indicates capacity of implemented system memory as follows: 01 256 Kbytes 10 128 Kbytes 11 512 Kbytes
—	14	—	Undefined.
DG	15	Diagnostic Test	Indicates that the virtual console is looping; for diagnostic purposes only.

CPU Acknowledge

DOAP*[ac]* *ac*,**CPU**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1

Clears the virtual console interrupts from the CPU status register.

> **NOTE:** *If a Halt instruction is performed, the virtual console clears the power fail interrupt. Never set bit 15.*

Halt

HALT
HALTA *ac*
DOC*[f]* *ac*,**CPU**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		1	1	0	F		1	1	1	1	1	1

Stops user program execution and returns to the virtual console if the Halt dispatch function is enabled by jumpering. Refer to Chapter 9, "Virtual Console" for more information.

The **DOC***[f]* *ac*,**CPU** instruction sets the Interrupt On flag according to the function specified in the *f* field, then stops the processor. If the Halt dispatch function is not enabled, then the processor, while stopped, will honor data channel requests but will not honor program interrupt requests.

> **NOTE:** *The assembler recognizes the mnemonic* **HALT** *as equivalent to* **HALTA** *0.*

Interrupt Acknowledge

INTA
DIB *[f]ac*, **CPU**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	1	1	F		1	1	1	1	1	1

Places a 6-bit device code in bits 10-15 of the specified accumulator. This device code identifies the highest priority device currently requesting an interrupt. Sets bits 0-9 of the specified accumulator to one.

After the transfer, the **DIB** mnemonic sets the Interrupt On flag according to the function specified by *f*.

The **INTA** mnemonic places the device code into bits 10-15 of the specified accumulator without affecting the **ION** flag.

Interrupt Disable

INTDS
NIOC CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	0	0	0	1	0	1	1	1	1	1	1

Sets Interrupt On flag to zero to disable program interrupts.

Interrupt Enable

INTEN
NIOS CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1

Sets Interrupt On flag to one to enable interrupts.

If this instruction changes the state of the Interrupt On flag from zero to one, the CPU allows one more instruction to execute before the first I/O interrupt can occur. However, if the instruction to be executed can be interrupted, then interrupts can occur as soon as the instruction begins to execute.

> **NOTE:** *If the instruction uses only one CPU cycle, then the CPU allows two instructions to execute before the first I/O interrupt can occur. Refer to Appendix C for a list of CPU cycles.*

Reset

IORST
DIC*[f]* *ac*,**CPU**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		1	0	1	F		1	1	1	1	1	1

Sets the Busy and Done flags of all I/O devices to zero.

The **IORST** mnemonic sets the **ION** flag to zero. The assembler recognizes the mnemonic **IORST** as equivalent to **DICC** 0, **CPU**. Positions in ACD and ACS are not both zero.

The **DIC** mnemonic sets the 16-bit priority mask to zero. Sets the **ION** flag according to the function specified by *f*. If you use this mnemonic, you must code an accumulator to avoid an assembly error. The processor, however, ignores the accumulator field. The specified accumulator remains unchanged.

The processor performs the equivalent of an **IORST** instruction at power-up, when you press the RESET switch (if the console is not locked), and when you type **I** from the virtual console.

> **NOTE:** **IORST** *will not affect any bits in the FPSR.*

Mask Out

MSKO *ac*
DOB*[f]* *ac*,**CPU**

0	1	1	AC	1	0	0	F	1	1	1	1	1	1
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Sets the priority mask.

The **DOB***[f] ac,* **CPU** instruction places the contents of the specified accumulator into the priority mask. After the transfer, the Interrupt On flag is set according to the function specified by *f*. The contents of AC remain unchanged.

If the device priority bit equals 1, the instruction sets the I/O device Interrupt Disable flag. All I/O device controllers respond to **MSKO, IORST,** and **INTA**. Only the CPU responds to such instructions as **INTEN, INTDS, HALT,** and **VCT**.

> **NOTE:** *A one in any bit disables interrupt requests from devices which use that bit as a mask.*

Read Virtual Console Registers

READS *ac*
DIA*[f]* *ac*,**CPU**

0	1	1	AC	0	0	1	F	1	1	1	1	1	1
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Places the contents of the virtual console register into and accumulator.

After the transfer, sets the Interrupt On flag according to the function specified by *f*. For more information, refer to Chapter 9, "Virtual Console".

CPU Skip

SKP*[t]* *ac*,**CPU**

0	1	1	0	0	1	1	1	T	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8 9	10	11	12	13	14	15

Skips the next sequential word if the test condition specified by *t* is true. The possible skip test conditions (test for interrupt *t*) are summarized in the following table.

Mnemonic	Value	Test
BN	00	Skip if interrupts are enabled.
BZ	01	Skip if interrupts are disabled.
DN	10	Skip if power flag is one.
DZ	11	Skip if power flag is zero.

Using the **SKP***(t)ac* **CPU** instruction for testing of the Power Fail flag allows the power-fail option to provide a "fail-soft" capability in the event of an unexpected power loss.

Programmable Interval Timer

I/O instructions addressed to device code 43_8 are grouped as programmable interval timer device instructions.

Priority Mask Bit — 6

IORST sets the Busy and Done flags, the interrupt request flag, the initial count register, the count output buffer, and the interrupt mask bit (bit 6) to 0; it then stops the counting cycle.

Mnem	Sets Bits 8 and 9 to	Instruction	Action
—	00	—	None
f=S	01	Start	Starts the counting cycle.
f=C	10	Clear	Stops the counting cycle.
f=P	11	Pulse	No effect.

Table 7.19.1 PIT flag commands

Read Count

DIA*[f]* *ac*,**PIT**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	0	1	F		1	0	0	0	1	1

Places the value of the programmable interval timer's counter in bits 0-15 of the specified accumulator destroying the accumulator's previous contents. After the data transfer, performs the function specified by *f*. The format of the specified accumulator is

0															15
PRESENT COUNT (2'S COMPLEMENT)															

Bits	Name	Contents or Function
0-15	Count	Current value of the PIT counter within one count cycle.

Table 7.19

Specify Initial Count

DOA*[f]* *ac*,**PIT**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC		0	1	0	F		1	0	0	0	1	1

Loads bits 0-15 of the specified accumulator into the programmable interval timer's initial count register. After the data transfer, performs the function specified by *f*. The contents of the specified accumulator remain unchanged. Format of the accumulator is

0															15
INITIAL COUNT (2'S COMPLEMENT)															

Bits	Name	Contents or Function
0-15	Initial Count	Two's complement of the number of 100-microsecond intervals between interrupts

Real-Time Clock

I/O instructions addressed to device code 14_8 are grouped as real-time clock device instructions.

Priority Mask Bit — 13

IORST sets the Busy and Done flags, the interrupt mask bit (bit 13) and the clock frequency select bits to zero; and then disables RTC interrupts.

Mnem	Set Bits 8 and 9 to	Instruction	Action
—	00	—	No effect
f=S	01	Start	Enables RTC
f=C	10	Clear	Disables RTC
f=P	11	Pulse	No effect

Table 7.20 RTC flag commands

Select Frequency

DOA*[f]* *ac*,**RTC**

0	1	1	AC		0	1	0	F		0	0	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Selects the frequency of the real-time clock. Loads the contents of bits 14 and 15 of the specified accumulator into the frequency select register. After the data transfer, performs the function specified by *f*. The contents of the accumulator remain unchanged. The format of the specified accumulator is

RESERVED		FS	
0	13	14	15

Bits	Name	Function
0-13	—	Reserved for future use.
14,15	Frequency select	Selects the desired frequency: Bits 14 15 0 0 = Line frequency 0 1 = 10 Hz 1 0 = 100 Hz 1 1 = 1000 Hz

Asynchronous Line Input

The I/O instruction addressed to device code 10_8 is the asynchronous line input instruction.

Priority Mask Bit — 14

IORST sets the following receiver flags to zero: Busy, Done, and Break. It forces the output of the data terminal ready register to the low state.

Mnem	Set Bits 8 and 9 to	Instruction	Action
—	00	—	None
f=S	01	Start	Sets the receiver Busy flag to one and the following receiver flags to zero: Done, Overrun, and Parity Error.
f=C	10	Clear	Sets the receiver Busy flag to one and the following receiver flags to zero: Done, Overrun, and Parity Error.
f=P	11	Pulse	Sets the Break flag to zero.

Table 7.21 Asynchronous line input flag commands

Asynchronous Interface

Read Character

DIA*[f]* *ac*,**TTI**

0	1	1	AC		0	0	1	F		0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Reads a character received by the communications line. Loads the contents of the receiver-holding register into the specified accumulator. After the data transfer, performs the function specified by *f*.

> **NOTE:** *The S/120 asynchronous communications interface receives and transmits 8-bit data characters without parity. If the system console device being used with the S/120 operates with a data character length of seven bits, you should configure the device to operate with "mark parity." When receiving data characters from a 7-bit system console device, software should mask out the parity bit after the character has been loaded into an accumulator. The parity bit is the most significant bit of the character and is contained in bit 8 of the specified accumulator.*

The format of the specified accumulator is

RESERVED	CHARACTER
0 — 7	8 — 15

Bits	Name	Function
0-7	—	Reserved for future use.
8-15	Character	The character most recently received, right justified in the accumulator.

Asynchronous Line Output

The I/O instruction addressed to device code 11_8 is the asynchronous line output instruction.

Priority Mask Bit —15

IORST sets the following receiver flags to zero: Busy, Done, and Break. It forces the output of the data terminal ready register to the low state.

Mnem	Set Bits 8 and 9 to	Instruction	Action
—	00	—	None
f=S	01	Start	Sets the transmitter Busy flag to one and the Done flag to zero.
f=C	10	Clear	Sets the transmitter Busy and Done flags to zero.
f=P	11	Pulse	No effect

Table 7.22 Asynchronous line output flag commands

Write Character

DOA*[f]* *ac*,**TTO**

0	1	1	AC		0	1	0	F		0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Writes a character to the communications line. Loads bits 8-15 of the specified accumulator into the transmitter holding register. After the data transfer, performs the function specified by *f*. The contents of the specified accumulator remain unchanged. The format of the specified accumulator is

	CHARACTER
0 — 7	8 — 15

Bits	Name	Function
0-7	—	Reserved for future use.
8-15	Character	The character to be transmitted, right justified.

ERCC Error Correction

I/O instructions addressed to device code 2_8 are grouped as ERCC Error Correction instructions.

Priority Mask Bit - None

IORST Sets the interrupt request flag, the Done flag, and the ERCC control flags (bits 14 and 15) to 0; disables error checking and correction.

Mnem	Sets bits 8-9 to	Instruction	Action
—	00	None	
S	01	Start	Sets the interrupt request flag and the done flag to 0.
C	10	Clear	No effect
P	11	Pulse	No effect

Table 7.23 ERCC Error Correction Flag Commands

Read Memory Fault Address

DIA*[f]* *ac*,**ERCC**

0	1	1	AC	0	0	1	F	0	0	0	0	1	0
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Places the sixteen low-order bits of the physical address of the fault location into the specified accumulator. (The previous contents of that accumulator are overwritten.) The instruction sets the Done flag as specified by the flag command.

The following shows the format of the contents of the specified accumulator.

LOW-ORDER ADDRESS BITS
0 15

Bits	Name	Contents or Function
0-15	Low-order Address	Sixteen low-order bits of the physical address of the memory fault location.

NOTE: *The physical address is meaningless unless it is read after the ERCC facility requests an interrupt and before a* Start *or* **IORST** *flag command sets the Done flag to 0.*

Read Memory Fault Code

DIB*[f]* *ac*,**ERCC**

0	1	1	AC	0	1	1	F	0	0	0	0	1	0
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Places a 6-bit error identification code in bits 0-5 of the specified accumulator. The instruction first sets bits 6-11 of the accumulator to 0 and places the four high-order bits of the physical address of the fault location in bits 12-15. Next, the instruction sets the Done flag as specified by the flag command.

The following table shows the format of the contents of the specified accumulator.

FAULT		HIGH-ORDER ADDR
0 5	6 11	12 15

Bits	Name	Contents or Function	
0-5	Fault Code	6-bit code, identifies the bit in error.	
		000000	No error.
		000001	Check bit 5.
		000010	Check bit 4.
		000011	Two bit error.
		000100	Check bit 3.
		000101	Two bit error.
		000110	Two bit error.
		000111	Multiple bit error.
		001000	Check bit 2.
		001001	Two bit error.
		001010	Two bit error
		001011	Data bit 15.
		001100	Two bit error.
		001101	Data bit 13.
		001110	Data bit 7.
		001111	Two bit error.
		010000	Check bit 1.
		010001	Two bit error.
		010010	Two bit error.
		010011	Multiple bit error.
		010100	Two bit error.
		010101	Data bit 12.
		010110	Data bit 6.
		010111	Two bit error.
		011000	Two bit error.
		011001	Data bit 10.
		011010	Data bit 4.
		011011	Two bit error.

Bits	Name	Contents or Function
0-5	Fault Code	6-bit code, identifies the bit in error
		011100 Data bit 0.
		011101 Two bit error.
		011110 Two bit error.
		011111 Multiple bit error.
		100000 Check bit 0.
		100001 Two bit error.
		100010 Two bit error.
		100011 Data bit 14
		100100 Two bit error.
		100101 Data bit 11.
		100110 Data bit 5.
		100111 Two bit error.
		101000 Two bit error.
		101001 Data bit 9.
		101010 Data bit 3.
		101011 Two bit error.
		101100 Multiple bit error.
		101101 Two bit error.
		101110 Two bit error.
		101111 Multiple bit error.
		110000 Two bit error.
		110001 Data bit 8.
		110010 Data bit 2.
		110011 Two bit error.
		110100 Data bit 1.
		110101 Two bit error.
		110110 Two bit error.
		110111 Multiple bit error.
		111000 Multiple bit error.
		111001 Two bit error.
		111010 Two bit error.
		111011 Multiple bit error.
		111100 Two bit error.
		111101 Multiple bit error.
		111110 Multiple bit error.
		111111 Two bit error.
6-11	—	Reserved for future use.
12-15	High Order Address	Four high-order bits of the physical address of the fault location.

NOTE: *The address is meaningless unless read after the ERCC facility requests an interrupt and before a* Start *or* **IORST** *flag command sets the Done flag to 0.*

Enable ERCC

DOA*[f]* *ac*,**ERCC**

0	1	1	AC		0	1	0	0	0	0	0	0	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the ERCC facility to function according to bits 13-15 of the specified accumulator. Next, the instruction sets the Done flag and then the Interrupt Request flag, as specified by the flag command. The instruction disregards bits 0-12.

The following shows the format of the contents of the specified accumulator.

	MODE
0 12	13 15

Bits	Name	Contents or Function
0-12	—	Reserved for future use.
13-15	MODE	Control the ERCC feature as follows: 000 • Write checkcode; disable checking and correction during memory memory read; disable interrupts; enable checking and correction during memory refresh. 001 Disable writing checkcode; disable checking and correction during memory read; disable interrupts; enable checking and correction during memory refresh. 010 Write checkcode; enable checking and correction during memory read and refresh; disable interrupts 011 Write checkcode; enable checking and correction during memory memory read and refresh; enable interrupts. 100 Write checkcode; disable checking and correction during memory read and refresh; disable interrupts. 101 Disable writing checkcode; disable checking and correction during memory read and refresh; disable interrupts 110 Write checkcode; enable checking and correction during memory memory read; disable interrupts; ~~disable interrupts~~; disable checking and correction during memory refresh. 111 Write checkcode; enable checking and correction during ~~memory~~ memory read; enable interrupts; disable checking and correction during memory refresh.

Chapter 8

Memory Allocation and Protection

MAP Functions

The memory allocation and protection (MAP) unit provides the necessary hardware to control and use more than 64 kilobytes of physical memory. In addition, the MAP provides protection functions to maintain the integrity of a large system.

> **NOTE:** *In the following section,* MAP *refers to the memory allocation and protection unit, whereas* map *refers to a set of memory translation functions used by the MAP unit.*

A MAP unit gives several users access to the resources of the computer by dividing the memory space available into blocks assigned to each user. Each time a user accesses memory, the MAP translates the address that the user sees—the logical address—to an address that the memory sees—the physical address. This is all transparent to the user. With software to control the priorities of the MAP and the CPU, several users can access the computer without being aware of the presence of the others.

Definitions

The following definitions will help you understand mapping.

Logical Address — the address used by the user in all programming. The logical address space is 32,768 words long and is addressed by a 15-bit address.

Physical Address — the address used by the MAP to address the physical memory. The maximum size of the physical address space in an S/120 system is 512 kilobytes and it is addressed by an 18-bit address.

Address Translation — the process of translating logical addresses into physical addresses.

Memory Space — the addresses (physical or logical) assigned to a particular user.

Page — 1024 (2000_8) words (2 kilobytes) in memory.

User Map — the set of memory address translation functions defined for a particular process. They translate logical addresses to physical addresses for every memory reference.

Data Channel Map — the set of address translation functions defined by the user-specified map. They translate logical addresses to physical addresses when data channel devices address the memory.

Supervisor — the part of the operating system which controls system function's such as the operation of the MAP unit.

Address Translation

The primary function of the MAP unit is address translation. A user map assigns each logical page of a user to a corresponding physical page. If a user's map is changed, the address space visible to the user is unchanged, but the map now translates each logical address into a different physical address.

A user's physical pages can be in any order in physical memory. This means that the supervisor can select unused pages for a new user without concern for maintaining any particular arrangement. This also allows a more complete use of the physical memory since no contiguous blocks of memory larger than 2 kilobytes are required.

Sharing Physical Memory

The MAP allows several users to use the same section of physical memory. This is useful if several users want to use a common routine such as trigonometric tables.

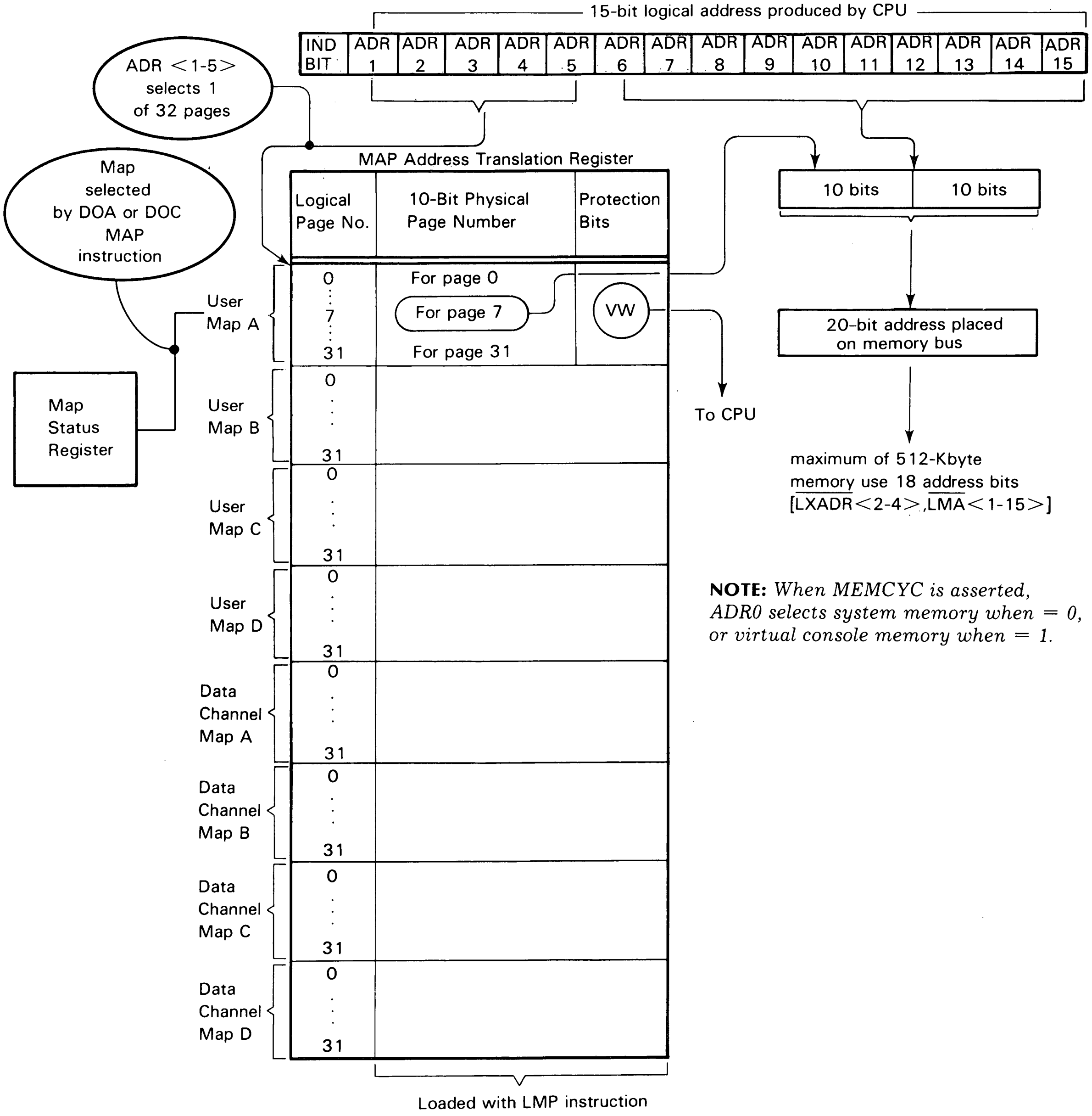
Mapped Mode

In mapped mode, the MAP unit provides two types of maps:

User maps
Data channel map

User Maps

Each user requires a separate user map. The MAP can hold four user maps, but only one can be enabled at any one time. This means that when four users exist, the processor specifies the user map for each and loads them


15-bit logical address produced by CPU
IND BIT
ADR 1
ADR 2
ADR 3
ADR 4
ADR 5
ADR 6
ADR 7
ADR 8
ADR 9
ADR 10
ADR 11
ADR 12
ADR 13
ADR 14
ADR 15
ADR <1-5> selects 1 of 32 pages
Map selected by DOA or DOC MAP instruction
Map Status Register
MAP Address Translation Register
Logical Page No.
10-Bit Physical Page Number
Protection Bits
User Map A
User Map B
User Map C
User Map D
Data Channel Map A
Data Channel Map B
Data Channel Map C
Data Channel Map D
0
7
31
For page 0
For page 7
For page 31
VW
To CPU
10 bits
10 bits
20-bit address placed on memory bus
maximum of 512-Kbyte memory use 18 address bits [LXADR<2-4>,LMA<1-15>]
NOTE: When MEMCYC is asserted, ADR0 selects system memory when = 0, or virtual console memory when = 1.
Loaded with LMP instruction

DG-08294

Figure 8.1 MAP address translation

into the MAP. The supervisor can then enable one or another as needed. If there are more than four users, new user maps must be loaded as needed. This is simplified by the use of the **LMP** instruction which loads a complete map with one instruction and uses relatively little time.

Data Channel MAPS

The data channel MAPS can access memory without direct control from the user's program. Thus, the data channel can service a user who is not the currently executing user. This allows the I/O activity of one user to be overlapped with the execution of another user. The MAP can hold four data channel maps. Enabling data channel mapping enables all four data channel maps at the same time. The choice of which map to use for data channel references is made by the I/O controller making the reference. Those controllers not equipped to make this distinction use data channel map A by default. See the *Programmer's Reference Manual Peripherals* (DGC No. 015-000021).

NOTE: *If an instruction changes the current map state and the two maps involved do not contain equivalent mapping, the next instruction will be fetched from and executed in the new map state.*

Unmapped Mode

The MAP can also operate in unmapped mode. In this mode, no address translation occurs, so that addresses issued by the CPU reference locations in the first 32 kilobytes of physical memory (locations in physical pages 0 through 32). If, while operating in unmapped mode, the program requires access to some other part of memory, the Page 31 register can be used to accomplish this. Refer to the **DOB, MAP** instruction in the instruction dictionary.

ECLIPSE S/120 Address Translation

Figure 1.2 illustrates the address translation performed by the S/120 MAP unit. Each user's 64-kilobyte logical address space consists of 32 1024-word (2-kilobyte) pages. A program can load an address translation map consisting of 32 12-bit words for each of up to four users and one map for a data channel. Each 12-bit word in a user's map includes 10 bits that specify the physical page and two bits that indicate a MAP protection code. One map code bit marks the page as write protected or write enabled and the other bit specifies that the page is validity protected or unprotected.

Emulator Trap

The ECLIPSE S/120 emulator trap allows users to emulate undefined instructions. If an undefined instruction is encountered while operating in the mapped mode, a return block is pushed onto the stack. The program then jumps indirectly through location 11_8. This location can contain the indirect address of an emulator routine. If the contents of location 11_8 are zero, an undefined instruction simply results in no operation (**NOP**).

MAP Protection Capabilities

In addition to address translation, the MAP provides four types of protection. The MAP flags the nature of each protection violation and causes a MAP protection fault. The four types of MAP protection are:

Validity protection
Write protection
Indirect protection
I/O protection

Validity Protection

Validity protection protects one user's memory space from inadvertent access by another user, thereby preserving the integrity and privacy of the user's memory space. When a user's map is specified, the blocks of logical addresses required by the user's program are linked to blocks of physical addresses. The remaining (unused) logical blocks are declared invalid to that user, and any attempt to access them will cause a validity protection fault.

Validity protection is always enabled, so the supervisor's responsibility is limited to declaring the appropriate blocks of logical addresses. If the MAP feature attempts to translate an invalid logical address for the user, a MAP protection fault occurs. In this case, the state of the processor is saved and the program jumps to the programmer-supplied MAP fault handling routine.

NOTE: *No validity traps occur on MAP single-cycle references, but memory is protected.*

Write Protection

Write protection allows users to read the protected memory locations, but not to write into them. In this way, the integrity of common areas of memory can be protected. An attempt to write into a write protected area of memory will cause a protection fault.

Blocks of logical memory may be write protected when the map is specified. Write protection can be enabled or disabled at any time by the supervisor.

For example, a set of trigonometric functions is stored in a section of memory accessible to all users. This section should be write protected so that users can read the functions but cannot change them.

Indirect Protection

Indirect protection allows the supervisor to ensure that the CPU will not be placed in an indirection loop. When in an indirection loop without indirect protection, the CPU would be unable to proceed with any further instructions, thus effectively halting the system.

With indirect protection enabled, a chain of ~~16~~ 15 indirect references causes a MAP protection fault. Indirect protection can be enabled or disabled at any time by the supervisor.

I/O Protection

I/O protection protects the I/O devices in the system from unauthorized access. If a user with I/O protection enabled attempts to execute an I/O instruction, an I/O protection fault will occur. Enabling I/O protection will prevent execution of the *Load Map* (**LMP**) instruction. I/O protection can be enabled or disabled at any time.

Validity violation clears LEF, I/O, DEFER and WP modes, plus other ~~error~~ violation errors.

Map Protection Faults

When a user violates one of the enabled types of protection, a protection fault occurs, as follows:

- The current user map is disabled.
- A 5-word return block is pushed onto the system stack. The program counter pushed will point to the word following the instruction which caused the MAP fault.
- Control is transferred to the protection fault handler by an indirect jump through memory location 3 which should contain the fault handler routine address.

The protection fault handler routine may determine the type of fault that occurred, using the *Read Map Status* **DIA MAP** instruction, before taking the appropriate action.

Any attempt to read beyond the maximum physical address space will result in undefined data being returned.

Any attempt to write beyond the maximum physical address space will have no effect and will not produce an error.

Load Effective Address Mode

The *Load Effective Address* **LEF** instruction has the same format as I/O instructions. The MAP has a LEF mode bit which determines whether an I/O format instruction will be interpreted as an I/O or a **LEF** instruction. When the MAP is enabled and the LEF mode bit is one (LEF mode enabled), all I/O format instructions are interpreted as *Load Effective Address* instructions. When the LEF mode bit is zero, all I/O format instructions are interpreted as I/O instructions.

The *Load Effective Address* instruction is very useful for loading a constant into an accumulator. In addition, a user operating in the LEF mode is denied access to any I/O devices, because all I/O and **LEF** instructions are interpreted as **LEF** instructions in this mode. This means that LEF mode can be used for I/O protection. The *Load Map* instruction, however, does not use the I/O format and therefore can still be executed.

Initial Conditions

At power up, the user maps and the data channel maps are undefined, the MAP is in unmapped mode, and unmapped logical page 31 is mapped to physical page 31. This means that all addresses issued by the CPU reference physical pages 0 to 32. While operating in unmapped mode, page 31 register may be used to access some other part of memory. Refer to the **DOB, MAP** instruction in Chapter 9.

After an *I/O Reset*, the MAP is in unmapped mode, the data channel maps are disabled, and unmapped logical page 31 is mapped to physical page 31.

MAP Instructions

The MAP instructions control the actions of the MAP. They are used by the supervisor program to change the mapping functions or to check the status of the various maps.

> NOTE: *MAP instructions* can *be executed in mapped mode if I/O protection and LEF mode are disabled for the user. When executed in mapped mode, the* Read Map Status, Initiate Page Check, *and* Page Check *instructions will return the desired information without changing the map. The* Map Single Cycle *instruction will disable the user map after the next memory reference.*
>
> *Enabling only LEF mode will convert all I/O instructions (including MAP instructions) to* **LEF** *instructions. The* Load Map *instruction, however, does not use the I/O format and therefore can still be executed. Enabling I/O protection will prevent execution of the* Load Map *instruction. Bit 1 of the MAP status register indicates the state of the MAP. If bit 1 is set to one, the MAP is enabled. If bit 1 is set to zero, the MAP is disabled. For further details, refer to the* **DIA MAP** *instruction in Chapter 10, Instruction Dictionary.*

The MAP instructions are shown in Table 8.1. All except *Load Map* **LMP** are in I/O format using the device mnemonic **MAP** .

Mnem	Instructions	Function
DIA	Read Map Status	Reads the status of the current map.
DIC	Page Check	Provides the identity and some characteristics of the physical page that corresponds to the logical page identified by the immediately preceding **DOC** *ac* **MAP** instruction.
DOA	Load Map Status	Defines the parameters of a new map.
DOB	Map Supervisor Page 31	Specifies the physical page corresponding to logical page 31 of unmapped address space.
DOC	Initiate Page Check	Identifies a logical page; selects the map without changing status.
LMP	Load Map	Loads successive words from memory into the MAP where they are used to define a user or data channel map.
NIOP	Map Single Cycle	Maps one memory reference using the last user map.

Table 8.1 MAP instructions

Load Map

LMP

1	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Loads successive words from memory into the MAP.

Words are loaded in consecutive, ascending order according to their addresses.

Three accumulators affect the **LMP** instruction:

- AC0 must contain zero.
- AC1 contains an unsigned integer which is the number of words to be loaded into the MAP.
- AC2 contains the address of the first word to be loaded. If bit zero is one, the instruction follows the indirection chain and places the resulting effective address into AC2.
- AC3 is ignored and its contents remain unchanged.

For each word loaded, the instruction decrements the number in AC1 by one and increments the address in AC2 by one.

Upon completion of the **LMP** instruction:

- AC0 remains unchanged.
- AC1 contains zero.
- AC2 contains the address of the word following the last word loaded.

The words loaded into the MAP define the address translation functions for the various user and data channel maps. The contents of the MAP field (bits 6-8) of the MAP status register determine which map is affected by the **LMP** instruction. You can alter this field by using either the *Load Map Status* (**DOA** ac,**MAP**) or the *Initiate Page Check* (**DOC** ac,**MAP**) instruction.

The format of the words loaded into the MAP is

WP	LOGICAL	PHYSICAL
0	1 5	6 15

Bits	Name	Contents or Function
0	Write Protect	Write protect for user maps. Map faults may not occur during memory references initiated by data channel.
1-5	Logical	Logical page number.
6-15	Physical	Physical page number.

NOTE: *To declare a logical page invalid, set the Write Protect bit to one and all of bits 6-15 to one.*

NOTE: *The* **LMP** *instruction is interruptible in the same manner as the* **BAM** *instruction.*

If you issue this instruction while in mapped mode, with I/O protection enabled, the map and accumulators are not altered and a MAP fault occurs.

If the **LMP** instruction alters the translation of the page indicated by the program counter for the next instruction fetch, this causes the instruction to be fetched from the new translation.

Load MAP Status

DOA *ac*,MAP

0	1	1	AC		0	1	0	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The contents of the specified AC are placed in the MAP status register. The contents of the AC remain unchanged.

The format of the specified AC is

NME	RESERVED					MAP			LEF	I/O	WP	IND	NME	DCH	UE
0	1				5	6		8	9	10	11	12	13	14	15

Bits	Name	Contents or Function
0,13	NME	Depending on the bit settings, the next user map enabled will be that for: Bit 1 Bit 13 User enabled 0 0 A 0 1 B 1 0 C 1 1 D
1-5	—	Reserved for future use.
6-8	MAP	Specifies which map will be loaded by the next **LMP** instruction as follows: Bit 6 Bit 7 Bit 8 User Enabled 0 0 0 A 0 0 1 C 0 1 0 B 0 1 1 D 1 0 0 Data Channel A 1 0 1 Data Channel C 1 1 0 Data Channel B 1 1 1 Data Channel D ~~1 0 1 Reserved~~ ~~1 1 0 Reserved~~ ~~1 1 1 Reserved~~
9	LEF	If one, the **LEF** instruction will be enabled for the next user.
10	I/O	If one, I/O protection will be enabled for the next user.
11	WP	If one, write protection will be enabled for the next user.
12	IND	If one, indirect protection will be enabled for the next user.
14	DCH Enable	If one, the mapping of data channel addresses will be enabled immediately after this instruction.
15	User Enable	If one, mapping of CPU addresses will commence with the first memory reference after the next indirect reference or return type instruction (**POPB, POPJ, RTN, RSTR**).

NOTE: *If the* **DOA MAP** *instruction sets the User Enable bit to one, the interrupt system is inhibited, and the MAP waits for an indirect reference or a return-type instruction. Either event releases the interrupt system and allows the MAP to begin translating addresses (using the user map specified by bits 0 and 13 of the MAP status register.*

Address translation resumes:

- *after the first level of the next indirect reference; or*
- *after a* **POPB, POPJ, RTN,** *or* **RSTR** *instruction.*

Read MAP Status

DIA *ac*,MAP

0	1	1	AC	0	0	1	0	0	0	0	0	0	1	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Reads the status of the current map.

Places the contents of the MAP status register in the specified AC. The previous contents of the AC are overwritten. The format of the information placed in the specified AC is

> **NOTE:** *The* **IORST** *instruction will clear bits 0 through 5, and ~~13 through~~ 14 and 15 of the MAP status register.* **IORST** *also turns off the map.*

NME	MPN	I/O	WP	IND	SC	MAP	LEF	I/O	WP	IND	NME	DCH	UM
0	1	2	3	4	5	6 8	9	10	11	12	13	14	15

Bits	Name	Contents or Function
0,13	NME	Depending on the bit settings, the last *Load Map Status* **DOA MAP** instruction enabled: Bit 0 Bit 13 User Enabled 0 0 A 0 1 B 1 0 C 1 1 D
1	MPN	MAP state -- one indicates mapping is enabled; zero indicates mapping is disabled.
2	I/O	If one, the last protection fault was an I/O protection fault.
3	WP	If one, the last protection fault was a write protection fault.
4	IND	If one, the last protection fault was an indirect protection fault.
5	SC	If one, the last map reference was a *MAP Single Cycle* (**NIOP MAP**) instruction.
6-8	MAP	Specifies which map was loaded by the last **LMP** instruction as follows: Bit 6 Bit 7 Bit 8 User Enabled 0 0 0 A 0 0 1 C 0 1 0 B 0 1 1 D ~~1 0 0 Data Channel~~ 1 0 0 Data Channel A 1 0 1 Data Channel C 1 1 0 Data Channel B 1 1 1 Data Channel D ~~1 0 1 Reserved~~ ~~1 1 0 Reserved~~ ~~1 1 1 Reserved~~
9	LEF	If one, the **LEF** instruction was enabled for the last user.
10	I/O	If one, I/O protection was enabled for the last user.
11	WP	If one, write protection was enabled for the last user.
12	IND	If one, indirect protection was enabled for the last user.
14	DCH	If one, the mapping of data channel addresses has been enabled.
15	UM	User mode. If one, the last I/O interrupt occurred while in user mode (map enabled).

Initiate Page Check

DOC *ac*,MAP

0	1	1	AC		1	1	0	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The contents of the specified AC are transferred to the MAP feature for later use by the *Page Check* or *Load MAP* instruction. The contents of the specified AC remain unchanged. The format of the specified AC is

	LOGICAL	MAP	
0	1 5	6 8	9 15

Bits	Name	Contents or Function
0	—	Reserved for future use.
1-5	Logical Page	Number of the logical page for which the check is requested.
6-8	Map	Specify which map should be used for check as follows:
		Bit 6 Bit 7 Bit 8 User Enabled
		0 0 0 A
		0 0 1 C
		0 1 0 B
		0 1 1 D
		1 0 0 Data Channel A
		1 1 0 Data Channel C
		1 1 0 Data Channel B
		1 1 1 Data Channel D
		~~1 0 1 Reserved~~
		~~1 1 0 Reserved~~
		~~1 1 1 Reserved~~
9-15		Reserved for future use.

Page Check

DIC *ac*,MAP

0	1	1	AC		1	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The number of the physical page which corresponds to the logical page specified by the preceding **DOC MAP** instruction is placed in bits 6 to 15 of the specified AC. Places additional information about the correspondence in bits 0-5. The previous contents of the AC are overwritten. The format of the information placed in the specified AC is

WP	MAP		PHYSICAL
0	1 3	4 5	6 15

Bits	Name	Contents or Function
0	WP	The write protect bit for the logical page which corresponds to the physical page specified by bits 6-15.
1-3	MAP	The map which was used to perform the translation between logical page number and physical page number as follows:
		Bit 1 Bit 2 Bit 3 User Enabled
		0 0 0 A
		0 0 1 C
		0 1 0 B
		0 1 1 D
		1 0 0 Data Channel A
		1 0 1 Data Channel C
		1 1 0 Data Channel B
		1 1 1 Data Channel D
		~~1 0 1 Reserved~~
		~~1 1 0 Reserved~~
		~~1 1 1 Reserved~~
4-5		Reserved for future use.
6-15	Physical Page	The number of the page which corresponds to the logical page given in the preceding **DOC MAP** instruction.

NOTE: *If all physical page bits including the write protect bit are one, then the logical page is validity protected.*

Map Supervisor Page 31

DOB *ac*,MAP

0	1	1	AC		1	0	0	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Specifies that mapping take place for a single page of an unmapped address space. Mapping is always done for locations 76000_8 though 77777_8 (logical page 31). This is the only page which can be mapped when in unmapped address space. You can use this instruction to access a page of a user's memory space when in unmapped mode. The MAP supervisor Page 31 instruction can only be used with the MAP off.

Bits 6-15 of the specified AC are transferred to the MAP feature. These bits specify a physical page number to which logical page 31 will be mapped when in the supervisor mode.

The contents of the specified AC remain unchanged. The format of the specified AC is

RESERVED		PHYSICAL	
0	5	6	15

Bits	Name	Contents or Function
0-5	—	Reserved for future use.
6-15	Physical Page	The number of the physical page to which logical page 31 should be mapped when in supervisor mode.

> NOTE: *If supervisor page 31 translation is altered while instructions are being fetched through supervisor page 31, instructions will be fetched from the new translation.* **IORST** *resets logical address translation to physical address translation.*

Map Single Cycle

Disable User Mode

NIOP MAP

0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The effect of this instruction depends upon the mode from which it is issued.

> NOTE: *The interrupt system is disabled from the beginning of the* MAP Single Cycle *instruction until after the next* **LDA, ELDA, STA,** *or* **ESTA** *instruction.*

From user (mapped) mode:

If the LEF mode and I/O protection are disabled, the **NIOP** instruction turns off the MAP. All subsequent memory references are unmapped until the map is reactivated with a *Load Map Status* **DOA MAP** instruction.

From the unmapped mode:

The user map is enabled for one memory reference. The first memory reference of the next **LDA, ELDA, STA,** or **ESTA** instruction is mapped. After the memory cycle is mapped, the user map is again disabled.

For example, if AC2 contains 405_8 and the following instruction sequence is issued:

```
.
.
NIOP   MAP    ;MAP SINGLE CYCLE
LDA    3,2,2
.
.
```

INTS are inhibit
'til instruction
after [E]LDA/[E]ST

then the logical address 407_8 will be mapped using the last enabled user map (specified by bits 0 and 13 of the MAP status register at the time of the memory reference). The word contained in the corresponding physical location will be placed in AC3.

However, if the following instruction sequence is issued:

```
.
.
NIOP   MAP    ;MAP SINGLE CYCLE
LDA    3,@2,2
.
.
```

then the logical address 407_8 will be mapped using the user map for the last enabled user. The contents of the corresponding physical location will be used as the first level of an indirection chain. The next memory cycle, which is the second level of the indirection chain, will not be mapped.

Chapter 9

Virtual Console

The virtual console is a program that can aid you in working with the ECLIPSE S/120 computer system. It allows you to interact with the computer through your terminal. You enter simple commands on a terminal keyboard to examine and/or modify any processor register or memory location. A breakpoint feature allows you to stop the execution of a program at selected places for debugging.

> **NOTE:** *Input/Output (I/O) interrupts are inhibited when the virtual console is executing. Also, when the virtual console is executing, the RUN light on the chassis is not lighted. When a user program is executing, the RUN light is lighted. I/O protection is not enabled when the virtual console is executing.*

The virtual console resides in read-only memory (ROM) chips on the ECLIPSE S/120 system processing unit (SPU) board. The virtual console has access to 512 bytes of static random-access memory (RAM), also on the SPU board, for use as a *scratchpad.* Since neither virtual console ROM nor scratchpad RAM is part of the normal address space, they are transparent to the user.

Entering the Virtual Console

Upon power up, the virtual console firmware first performs a short (0.75-second) self-test routine; then, if the front panel is locked, it examines the contents of the automatic program load (APL) register. If the APL register contains device code 0, the virtual console retains control. If it contains any other device code, the central processing unit (CPU) performs an immediate program load from that device. (See "Program Load Commands" and the "Power-Up Self-Test" sections.)

In addition to power up, the virtual console is entered when:

- A **HALT** instruction is executed (unless **HALT** dispatch is disabled).
- The user presses the BREAK key of the system console (if the Break function on the controller board is not disabled by jumpering). The virtual console returns to the user program if the console is locked.
- The user presses the RESET switch on the front console and the console is not locked.
- The program completes execution of an instruction in the one-step mode (See "Program Debugging" section).
- A breakpoint is encountered (See "Program Debugging section").

> **NOTE:** *When the user presses the RESET switch to enter the virtual console, the contents of the program counter are lost.*

The contents of the ACs, the program counter, and the carry bit are displayed each time the virtual console is entered, except at power up and when the user presses the RESET switch.

Once the virtual console has been entered and the user presses the PR LOAD switch on the front panel, the virtual console will attempt to program load from the device selected by the APL register.

Once called, the virtual console displays a ! on the terminal. This is called the virtual console *prompt*; it indicates that the virtual console is ready to accept a command. A single character preceeding the prompt indicates the current state of the memory allocation and protection (MAP) unit:

!	The MAP is currently off and no address translation will occur.
A!	The MAP is on and user A has been selected.
B!	The MAP is on and user B has been selected.
C!	The MAP is on and user C has been selected.
D!	The MAP is on and user D has been selected.

When a particular user has been selected, the current state of that user's map will be used in all address translations. You can change the MAP status from within the virtual console as explained in "Changing the MAP or MAP Status" later in this chapter.

Errors

This section discusses two methods for correcting typographical errors: the Rubout/Delete key and the **K** command, The final topic in the section is virtual console errors, which occur when commands are issued incorrectly. The context of these errors wll be clarified when each command is discussed later in the chapter.

The Rubout/Delete Key

The Rubout/Delete key deletes the last character you typed. The virtual console echoes the deletion with an underscore (_). Typing additional Rubouts deletes digits from right to left.

If you type any Rubouts immediately after opening a cell, the virtual console deletes the rightmost digits of the cell's contents as though you had just typed them yourself. You may then type in new values for these digits. Refer to the "Command Formats" section in this chapter for more information on the Rubout key.

> **CAUTION:** *The Rubout/Delete key has no effect on floating-point accumulators. If you type these characters, the virtual console will issue a New Line prompt, without changing any data.*

The K Command

If you wish to cancel the entire line that you just entered, type a **K**. The virtual console prints a ? followed by a New Line with a prompt, and also closes the current cell if it is open. The ? followed by a New Line with a prompt is also printed if you type a character that the virtual console does not recognize.

Virtual Console Errors

If you attempt to open a nonexistent memory cell, the data displayed as its *contents* will be meaningless. To determine whether a location exists, enter a new value in the memory cell and then reopen it. If it does not contain the value just entered, then the location is nonexistent.

The virtual console types a ? followed by a New Line with a prompt in the following conditions:

- An **R** command is issued without an argument.
- A set breakpoint command specifies an invalid address.
- A delete breakpoint command specifies a number greater than 7.
- A set breakpoint command is issued after all eight breakpoints have been assigned.
- An *n***M** or *n***C** command is issued and no map is on, or *n* specifies a logical page number greater than 37_8.
- An *n***F** command is issued and *n* specifies a number greater than 3.

In all of the above cases, the command involved will not be executed. In fact, the virtual console does nothing, except discard data that was just entered with an erroneous command.

Commands

In the following sections, the virtual console commands are categorized into three groups:

- Cell commands
- Function commands
- Miscellaneous commands

A virtual console command consists of a single character. Some commands require a leading *argument*, which is an octal number or an expression. Valid numbers and expression are:

Digits
must be in the range from 0 through 7. (If the argument is an address, it must be in the range from 0 through 77777.)

Hex digits
ranging from 0 through 9 or A through F.

Period
the character "." replaces the value of the last address used.

Signs plus (+) or minus (−)
may be typed after any valid number and must be followed by a valid number. The virtual console will compute the arithmetic result and enter it in place of the original expression.

Delete or Rubout
the delete key may be used to delete any single digit. The virtual console displays an underscore character (_)to indicate that the preceding character has been deleted. The delete key will *not* delete the +, − or . symbols. If delete is used after a + or a − , it has no effect. If it is used after a period, it deletes the rightmost digit of the last address. The virtual console only retains six digits at any time; therefore, the delete key will not resolve all errors.

For clarity, all examples in this chapter show data entered by the user in bold type. On the terminal, user input and program response are not differentiated.

Cell Commands

Several virtual console instructions operate on *cells*. A cell is either a memory location, *memory cell*, or an internal register, *internal cell*, such as an accumulator, or a floating-point accumulator (*FPAC cell*). Each internal register accessible by the virtual console is assigned an internal cell number. Table 9.1 lists these registers and their numbers.

Number	Cell
0-3	Accumulators AC0 through AC3, respectively.
4	The address of the break instruction at which the program halted, if the virtual console was entered on encountering a break instruction; or the contents of the program counter, if the virtual console was entered in any other way.
5	Carry bit: bit 15 is equal to 0 when carry equals 0; equal to 1 when carry equals 1.
6	CPU (interrupt and virtual console interrupt) status.
7	System console status word.
10	Virtual console register.
11	MAP status register.
12	Floating-point status register.
13	Floating-point program counter.
14	Search mask.*

Table 9.1 Internal cells

**Refer to "Search Command" in this chapter for the contents of this register.*

To examine or modify any cell, you must open it using one of the commands listed in Table 9.2. Opening a cell causes its address and contents to be printed, in octal, at your terminal. Addresses and memory or internal cell contents are displayed in octal format, while floating-point cell contents are displayed in hexadecimal format.

Command	Function
*n*A	Opens the internal cell specified by *n*.
expr/ [1]	Opens the memory location specified by octal number *expr*.
Carriage Return [2]	Closes the current cell and opens the next consecutive cell.
New Line[3]	Closes the current cell but does not open another.
/	Closes the current cell and opens the memory cell whose address is equal to the contents of the current memory or internal cell; after a New Line, opens location 0.

Table 9.2 Virtual console cell commands

[1] *The symbol* expr/ *represents any valid octal number or expression. See "Command Formats" section.*

[2] *Current cell means the last cell opened.*

[3] *Line Feed on non-ANSI standard keyboards.*

Internal cell number 10, the virtual console register (Table 9.1) can be accessed while in user mode with a **READS** instruction. This call always contains the device code of the last device from which a program load was performed.

When you open a memory cell, the virtual console interprets the address according to the current setting of the user MAP. That is, the number you enter is interpreted as a 15-bit address, then translated into a physical address. If the memory cell address you enter contains more than five octal digits (15-bits), only the last five digits are used. Leading zeroes are not necessary. If for example, you want to open logical memory location 5, type

```
5/
```

Once you have opened a cell, you may change its contents by typing the octal number or expression whose value is to be placed in the cell. When you are changing an internal cell or a memory cell, you do not have to type leading zeroes. When you are changing floating-point accumulator cells, the digits are entered most-significant position first, therefore it is not necessary to enter trailing zeroes. Terminate the expression with a Carriage Return, Line Feed, or New Line. Note that if you type Carriage Return, the next cell will also be opened. This is convenient when you enter data into several consecutive locations.

> NOTE: *If you open a cell and immediately type* **H** *or* **L**, *the contents of the cell are used as the value of* expr *for that command.*

If you type an expression starting with a + or -, the value of the expression will be added to or subtracted from the current contents of the cell.

> NOTE: *You cannot type an expression starting with + or - when altering floating-point accumulator (fpac) cells.*

Examples showing resolution of expressions, when the last address entered was 100 are:

- 100000_1 will be replaced by 100001.
- **1000000** will be replaced by 000000 *(virtual console only remembers six digits).*
- **.-3** will be replaced by 75.
- **.7** will be replaced by 1007.
- **0-7** will be replaced by 177771.
- **6+.-3** will be replaced by 103 *(6+100-3=103).*
- **75+_5**will be replaced by 102 *(75+5=102 — the + is not deleted.).*
- **60+._**will be replaced by 70 (60+10=70 — the _ erased the rightmost 0 of the last address, 100).

> NOTE: *When altering FPAC cells, you cannot type an expression starting with = or -, or delete any digits already entered.*

If you enter an illegal digit, the virtual console issues a prompt and does not change the cell content. If you enter more than 16 bits when altering internal cells or memory cells, only the last 16 bits entered are used. If you enter 16 bits when altering floating-point cells, the virtual console automatically changes the cell content and issues a prompt.

Examples showing the use of /, New Line, and Carriage Return, with data entered by the user appearing in boldface are:

A! **3A** 000003A 000100 <**CR**>
AC3 contains 100. Internal cell 4 is opened as shown on next line.

000004A 000704 /0000704 024132 <**NL**>
PC contained 704. Memory location 704 contains 24132. The next memory location is not opened.

A! **5A** 000005A 0000001 <**NL**>
User changed the carry bit to 1. The next cell is not opened.

A! **100**/000100 025037 .<**NL**>
Changes the contents of memory location 100 to current address. The next memory location is not opened.

A! **100**/000100 000100 <**CR**>
Confirms the preceding commands. Memory location 101 is opened as shown on next line.

000101 000602 +**1**<**NL**>
Increments contents of 101. The next memory location is not opened.

A! **.**/000101 000603
Comfirms preceding step.

In all memory location examples above, there was no map protection for the logical page. When the logical page is protected, the type of protection is indicated with a character displayed between the memory address and the data as follows.

A! **50**/ 000050 X DDDDDD

In this example

X = type of protection; it can be V, W, WP, or a *space*.
V = validity protected logical page; data displayed is invalid.
W = write-protected logical page; data cannot be changed.
WP = write-protected logical page; and user write fault is enabled, data cannot be changed.
space = unprotected logical page.
DDDDDD = data

Function Commands

Table 9.3 lists the virtual console function commands. Sections that follow explain these commands in detail.

Command	Function
A	Displays contents of AC0 through AC3, program counter, and carry bit.
*expr***B C**	Inserts a breakpoint at the memory location specified by octal number *expr*. (If no *expr* is entered, all breakpoints will be displayed along with their assigned numbers.) Displays contents of all data channel maps.
*n***C**	Displays and/or alters the currently-selected data channel map path specified by *n* (n = 0-37_8).
*n***D**	Deletes breakpoint number *n* where *n* is a number between 0 and 7. (If no *n* is specified, all breakpoints are deleted.)
↑**G**	Executes the power-up self-test sequence.*
*n***H**	Performs a program load from the data channel device whose device code is *n*.
I	Executes an *I/O Reset* (**IORST**) instruction.
K	Cancels the entire line just typed and prints a question mark (?).
*n***L**	Performs a program load from the programmed I/O device whose device code is *n*.
M	Displays contents of all user maps.
*n***M**	Displays and/or alters the currently-selected user map page specified by *n* (n= 0-37_8).
O	Steps through one instruction of the user's program.
P	Starts program execution at the memory location specified by the contents of internal cell number 4. (See Table 9.1.)
*n***P**	Same as **P** above, except the next *n* breakpoints are ignored.
*expr***R**	Starts program execution at the memory location specified by octal number *expr*.
*n***S**	Searches the currently-selected logical space for the value specified by *n*.
U	Changes the user map. Displays a colon (:), after which the user must enter A, B, C, or D to specify a map. If any other character is entered, the map is turned off.
↑**V**	Performs a user read/write memory test at all locations. *

Table 9.3 Virtual console function commands

**Both ↑G and ↑ V represent the simultaneous depression of the console CTRL and G or V keys.*

Breakpoints and Program Control

The virtual console breakpoint facility allows you to place breakpoints at up to eight locations in your program. When the program encounters the breakpoint during execution, it enters the virtual console so that you can examine or modify any cells. Breakpoints are an aid in debugging a program, since they enable you to stop your program at locations where there may be problems and then resume execution with no loss of data.

NOTE: *Breakpoints will not work and should not be used if the Halt Dispatch is not enabled. Refer to Halt Dispatch jumper in the "Installation and Jumpering" section of ECLIPSE S/120* Computer System Hardware Reference *DGC 014-000690.*

Setting Breakpoints

To set a breakpoint, type *expr* **B**. This command sets the breakpoint at the address specified by argument *expr* according to the *current user map*. Breakpoints can be used only in the user map from which the user program will be started.

The virtual console assigns numbers to breakpoints in reverse order — that is, breakpoint 7 is assigned first, then 6, and so on. The unassigned breakpoint with the highest number is always assigned first. For example, if numbers 7 and 5 are assigned, the next will be 6, not 4.

To delete a breakpoint you must use the number assigned to it as described below. Typing **B** with no specified address causes the virtual console to list all the current breakpoints along with their assigned numbers.

Examples:

A!**423B**	Places a breakpoint at address 423 in user map A.
A!**B**	Requests list of all current breakpoints.
7 75324	Breakpoint 7 is at address 75324.
3 423	Breakpoint 3 is at address 423.
A!**623B**?	Requests a breakpoint at a valid location, but apparently all eight breakpoints are in use. User must delete a breakpoint before setting another.
A!	A prompt always follows a "?".

NOTE: *Do not place two breakpoints at the same location.*

NOTE: *Breakpoints* **must** *never be set in addresses that are to be executed when the* Load Effective Address *mode is enabled or I/O protected.*

Deleting Breakpoints

The **D** command deletes a breakpoint. The command's format is *n***D** where *n* specifies the breakpoint number. This command deletes the breakpoint regardless of the current state of the map. If no argument ("n") is specified, **D** command deletes all breakpoints.

Examples

A!**3D**	Deletes breakpoint number 3.
A!**12D**?	Only eight breakpoints (numbers 0-7) are valid; -- no other number is allowed.

Encountering a Breakpoint

When a breakpoint is encountered during execution of a user program, the virtual console is entered. The address of the instruction at which the breakpoint was set is displayed and placed in internal cell number 4, and the instruction at the location of the breakpoint is not executed. Then the virtual console displays a prompt, indicating that the user can now inspect and modify any internal cell or memory location.

Single Stepping

Use the command **O** to one-step through a program. Issued when the virtual console has control, the **O** command sets a flag that causes a virtual console interrrupt to occur as soon as the first main (user) program instruction has executed. The virtual console then returns to the main program location specified by the contents of internal cell 4; the main program executes one instruction, and then the virtual console resumes control.

You can cause the virtual console to step through *n* instructions by typing *n***O**, where *n* is an octal number that can range between 0 and 177777. The virtual console then executes those *n* instructions. As each instruction is executed, the virtual console prints the address of the following instruction, the contents of the ACs and the condition of the carry bit, that is, the contents of the program counter, ACs and the carry flip-flop at the time of instruction execution. This is a convenient way to locate skips or branches. After the *n* count of instructions have been executed, the virtual console resumes control and issues a prompt.

NOTE: *The fact that a user breakpoint may have been set for an instruction will have no effect on the execution of the* **O** *command. When single stepping, interrupts will not be honored if enabled before entering a virtual console.*

NOTE: *The BREAK key interrupts the virtual console before it finishes stepping through all* n *instructions. In response, the virtual console displays the address of the instruction following the one last executed and then displays a prompt.*

Resuming Program Execution

The virtual console has two commands that allow you to resume program execution after the virtual console has been entered through a breakpoint, after single-stepping, or after the BREAK key has been pressed: **P** and **R**.

Typing **P** restarts program execution at the location specified by the contents of internal cell number 4, which specifies the return address (see Table 9.1). Typing *n***P** restarts program execution in the same manner; however, the next *n* breakpoints are ignored.

NOTE: *When the virtual console is entered through a breakpoint, internal cell 4 contains the address of the location of the breakpoint. This should be the next instruction to be executed in order to resume normal* *program flow. When the virtual console is entered any other way, internal cell 4 contains the value of the PC + 1; this should also be the next instruction executed in order to resume normal flow. In either case, the* **P** *instruction produces the required result.*

You can also return to a program by typing *expr***R**. In this case, program execution resumes at the location specified by *expr*. When the **R** command is issued, the virtual console inserts all previously specified breakpoints, clears virtual console interrupts, and resumes program execution at the logical address <*expr*> in the currently selected MAP. The **R** command does **not** cause an I/O or system reset. The number specified by *expr* must be a valid address in user memory. If this argument is not in the user range, or is not supplied, the virtual console simply displays ? and then issues a prompt.

Miscellaneous Commands

Changing the MAP or MAP Status

To change user maps, use the **U** command. In response to this command, the console immediately prints a ":". Now you must type a single character: A, B, C, or D. Typing *any* other character (including Carriage Return, New Line), turns off the MAP. After you type a character, the virtual console displays a prompt reflecting the new state of the MAP.

You can change the MAP status from within the virtual console by opening internal cell 11, the MAP status register, with a **11A** command. Type in the new status, then close the cell with a New Line. *Now issue a* **U** *command, whether or not you want to change maps.* (If you do not want to change maps, simply enter the character for the current map.) The new map status is effective only after a **U** command has been issued.

Examples:

```
!U:B
```

No user map selected; user selected the B user map.

```
B!11A000011A 000000 77<NL>
```

MAP status register cleared; user entered 77.

```
B!11A000011A 000077 <NL>
```

Confirms preceeding step.

```
B!U:B
```

New map status takes effect.

```
B!U:<NL>
!
```

Turns MAP off.

Displaying, Altering, and Dumping User MAPs

Using the **M** command, displays, and/or alters the physical page to which a logical page of the currently selected user map is mapped. This command also displays the physical page to which all logical pages of all user maps are mapped.

To display the physical page to which a logical page of the currently selected user map is mapped, type *n***M**. In this format, *n* represents the octal logical page number. The physical page is displayed in the following format:

WPPPPP

W Signifies write protection: 1 = write-protected, 0 = not write-protected.

PPPPP Represents the physical page number in octal (0 - 1777).

After the physical page number is displayed, you can alter the physical page for the same logical page by typing the new page number in the same format followed by <NL> or <CR>.

To display the physical page to which all logical pages of all user maps are mapped, type **M**.

Examples:

```
A!32M 000132 100120<NL>
A!
```

User map A, logical page 26 ($32_8 = 26_{10}$) is mapped to physical page 90 ($132_8 = 90_{10}$) and is not write-protected. User remapped logical page 26 to physical page 80 ($120_8 = 80_{10}$) and set write protection.

```
A!5M100030 <CR>
000006 000077 <NL>
A!
```

User map A, logical page 5 is mapped to physical page 24 ($30_8 = 24_{10}$) and is write protected. User map A, logical page 6 is mapped to physical page 63 ($77_8 = 63_{10}$) and is not write-protected.

Displaying, Altering, and Dumping Data Channel MAPs

The **C** command displays and/or alters the physical page to which a logical page of the currently selected data channel map is mapped. It also displays the physical page to which all logical pages of all data channel maps are mapped.

To display the physical page to which a logical page of the currently-selected data channel map is mapped, type *n***C**. The display of the physical page is in the same format described in the section "Displaying, Altering, and Dumping User MAPs."

To display the physical page to which all logical pages of all data channel maps are mapped, type **C**.

Auto Program Load

Before pressing the PROGRAM LOAD switch on the console, you must prepare the I/O device to read data.

Follow the program load instructions for your operating system in one of the following manuals:

- *Loading and Generating MP/AOS* (DGC No. 069-400207)
- *How to Load and Generate Your RDOS System* (DGC No. 069-400013)
- *How to Load and Generate Your Advanced Operating System* (DGC No. 093-000217).

When the user presses the PROGRAM LOAD switch on the front panel, the virtual console loads the program from the device selected by the auto program load register. This load register contains the device code plus a bit which identifies whether it is a programmed I/O or a data channel program load. If you wish to perform a program load from a device other than the device selected by the auto program load register, proceed as follows.

- Type *n***L** to cause the CPU to perform a program load from a programmed I/O device whose device code is equal to the octal number *n*. (Typing device code 0 causes a re-entry to the virtual console; device code 77 is reserved.)
- If *n* L is typed, the programmed I/O device supplies data in bits 8-15. The program load program stores each pair of bytes as a single word in memory: the odd byte becomes the left half of the word, and the even byte becomes the right half.

The program load program ignores leading null characters and does not begin storing any words until it reads a nonzero synchronization byte. The first word following this synchronization byte must be the two's complement of the number of words to be loaded, including the word count number. The program load program stores a maximum of 32 kilobytes to be read beginning at memory location 100_8.

Type *n***H** to cause the CPU to perform a program load from a data channel device whose device code is equal to *n*. An **IORST** instruction is issued to clear all devices. This also sets the word count and address registers to zero. The data channel begins at address zero and loops at location 377_8 until the data transfer places a word into that location. The program then executes the transferred word as an instruction. Typically, the instruction is to halt or to jump to the data that have just been transferred.

> **NOTE:** *Some data channel devices transfer more than 256 words at a time. It is up to the device or the program to control the transfer after 256 words have been read.*

Once a program load has begun, the virtual console is no longer in control.

After a program load has been performed - whether initiated by *n***L**, *n***H**, or **PR LOAD-** — the device code of the device loaded from is placed in the virtual console register. The device code shifted left one bit position, is also placed in AC0. A **READS** instruction can be used at any time to retrive the device code of the last load device from the virtual console register.

The I/O Reset Command

The **I** command causes the virtual console immediately to execute an I/O **RESET** instruction to all I/O devices. This clears all I/O device controller flags (Busy = 0, Done = 0).

Example:

```
B!I
!
```

User map B was selected when the user issued a **I** command. I/O Reset disables the MAP and clears the Map status register.

```
!11A000011A 0000000<NL>
!
```

Confirms Map status register is cleared.

The Search Command

The *expr***S** command provides you with a means to search the currently specified logical space (64 kilobytes) for the value *expr*. The contents of each searched memory location are *ANDed* with a search mask contained in internal cell 14 before the comparison is made. The address and contents of each location where a comparison is found will be displayed as

AAAAAA DDDDDD

In this format,
AAAAAA = the physical memory address
DDDDDD = the contents

Example This example seaches the currently-specified logical page for all occurences of any I/O instruction to device code 22:

```
A!14A 000014A ?????? 160077<CR>
```

Sets the search mask to 160077 which will limit the comparision to all I/O instructions.

```
A!060022S
```

Search for all I/O instructions to device 22.

```
15643 60122
17423 61322
23654 62222
```

I/O instructions to device code 22 were found in three locations of the current logical space.

The Power-Up Self-Test

The ↑G command causes the virtual console to execute the power-up self-test sequence.

> **CAUTION:** *The memory-test portion of the power-up self-test is destructive to user and virtual console memory; therefore, a program load* **must** *be performed after the self-test.*

Errors that occur are flagged as follows:

I/O fault:
the virtual console displays an I.

Virtual console memory fault:
the virtual console displays an H.

User memory fault:
the virtual console displays an M.

Error checking and correction fault:
the virtual console displays an E.

If the virtual console displays any error flag, the user must depress the BREAK key to continue. If no errors are encountered and the front panel is locked, the virtual console performs a program load. The virtual console will retain control and issue a prompt under the following conditions:

- The user depresses the BREAK key after an error
- The APL register contains device code 0.
- The front panel is unlocked.

> **NOTE:** *The power-up self-test is not a definitive hardware test. Its primary purpose is to detect faults that would prevent the loading of diagnostic programs.*

The User Read/Write Memory Test

The ↑V command causes the virtual console to execute a test sequence of all user read/write memory locations.

> **CAUTION:** *The test is destructive to user read/write memory; therefore, a program load* **must** *be performed after the* ↑V *command.*

The virtual console displays **P** each time all locations of user memory have been tested. Any error that occurs causes the error location to be displayed as shown below, and the test continues.

XXX YYYY

In this display

XXX Indicates the physical page number that failed. The number ranges from 0 to 377_8

YYYY Specifies the location within page that failed. The number ranges from 0 to 1777_8

To stop the user memory test depress the BREAK key. The test will stop at the completion of the current pass.

Chapter 10

Instruction Dictionary

This dictionary contains instructions listed in alphabetical order according to assembler mnemonics. Refer to the I/O instruction dictionary in Chapter 7 for specific I/O instruction information.

Included for each instruction are:

- the assembler mnemonic,
- the format of any arguments involved,
- the bit format produced,
- a functional description of each instruction.

Coding Aids

We use certain conventions and abbreviations throughout this chapter to help you properly code each instruction for Data General's assembler. Briefly, they are:

Symbol	Means
[] or *[]*	Square brackets indicate that the enclosed symbol (for example, *[,skip]*) is an optional operand or mnemonic. Code it only if you want to specify the option.
Bold	Code operands or mnemonics printed in boldface exactly as shown. For example, code the mnemonic for the *Move* instruction: **MOV**.
italic	For each operand or mnemonic in italics, replace the item with a number or symbol that provides the assembler value you need for that item (for example, the proper accumulator number, an address, etc.)

Table 10.1 Type conventions

We use the following abbreviations throughout this chapter.

Abbr	Meaning
i	Signed two's complement integer in the range of −32,768 to 32,767; or unsigned in the range of 0 to 65,535.
N	Integer in the range of 0 to 3.
n	Integer in the range of 1 to 4.
AC	Accumulator.
ACS	Source accumulator.
ACD	Destination accumulator.
FPAC	Floating-point accumulator.
FACS	Floating-point source accumulator.
FACD	Floating-point destination accumulator.

Table 10.2 Abbreviations

Setting the Index Field

To set the index field, code a comma followed by an integer between 0 and 3. This will set the index field to the value you specified. You can also use the period (.) to set the index field to 01 (PC relative). The period can be read as the *address of the current instruction.* When you use a period, you usually follow it with a plus or minus sign and the displacement value, such as **.+3** or **.−12.**

If you are coding extended class instructions, note that using a period (for example, **EJMP .+5**) does not produce the same effect as coding a comma followed by a 1 (**EJMP 5,1**). When using a period, the displacement is added to the address of the instruction (the first word of a 2-word instruction). When using a comma, the displacement is added to the address of the word containing the displacement (the second word of a 2-word instruction). Therefore, **EJMP .+5** is equivalent to **EJMP 4,1**.

Add Complement

ADC*[c][sh][#]   acs,acd[,skip]*

1	ACS	ACD	1	0	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Adds the logical complement of an unsigned integer to another unsigned integer.

Sets carry to the specified value. Adds the logical complement of the unsigned, 16-bit integer in ACS to the unsigned, 16-bit integer in ACD. If the addition produces a result greater than $2^{16}-1(65535_{10})$, then the value of carry is complemented. Places the 17-bit result (carry and function result) in the shifter.

Tests the skip condition. Performs the specified shift operation. If the no-load bit is zero, loads the 17-bit value into the carry bit and ACD. If the skip condition is true, skips the next sequential word.

> **NOTE:** *If the number in ACS is less than the number in ACD, the instruction complements the value of carry before shifting.*

Example

Operation	Before	After
ADC 1,0 Add the complement of 010101_8 to 345_8	$AC0=000345_8$ $AC1=010101_8$ Carry=0	$AC0=170243_8$ $AC1=010101_8$ Carry=0

Add

ADD   *[c][sh][#]acs,acd,[,skip]*

1	ACS	ACD	1	1	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Performs unsigned integer addition.

Sets carry to the specified value. Adds the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD. If the result is greater than $2^{16}-1$ $(65{,}535_{10})$, complements carry. Places the 17-bit value (carry and the result of the add) into the shifter. Performs the specified shift operation.

Tests the skip condition. If the no-load bit is zero, places the 17-bit value in the carry bit and ACD. If the skip condition is true, skips the next sequential word.

Example

Operation	Before	After
ADD 1,0 Add 345_8 and 010101_8	$AC0 = 000345_8$ $AC1 = 010101_8$ Carry = 0	$AC0 = 170243_8$ $AC1 = 010101_8$ Carry = 0

Extended Add Immediate

ADDI *i,ac*

1	1	1	AC		1	1	1	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

IMMEDIATE FIELD	
16	31

Adds a signed integer in the range -32768_{10} to $+32767_{10}$ to the contents of an accumulator.

Adds the signed 16-bit, two's complement number contained in the immediate field to the signed 16-bit, two's complement number contained in the specified accumulator. Places the result in the accumulator. Carry remains unchanged.

Example

Operation	Before	After
ADDI 303, 1 Add 303_8 and 000345_8.	ACO = 000345_8	ACO = 000650_8

Add Immediate

ADI *n,ac*

1	N		AC		0	0	0	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Adds an unsigned integer in the range of 1 to 4 to the contents of an accumulator.

Adds the unsigned, 16-bit number in the specified accumulator to the contents of the immediate field *n* plus 1. Carry remains unchanged.

> **NOTE:** *DGC assemblers compute N before loading the immediate field. You code* n, *which is N + 1. Code the exact value you want to add.*

Example

Operation	Before	After
ADI 4,2 Add 4 to 177775_8.	AC2 = 177775_8	AC2 = 000001_8

AND With Complemented Source

ANC *acs,acd*

0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15
1	ACS	ACD	0	0	1	1	0	0	0	1	0	0	0

Forms the logical AND of the complement of the contents of ACS and the contents of ACD.

Sets carry to the specified value. Forms the logical AND of the one's complement of the contents of ACS and the contents of ACD. Sets a bit in the result to one if the corresponding bit position in ACS is zero and the corresponding bit position in ACD is one.

Places the result in ACD. Leaves ACS unchanged.

Example

Operation	Before	After
ANC 0,1 AND the complement of AC0 with AC1.	AC0 = 177775_8	AC0 = 177775_8

AND

AND*[c][sh][#]* *acs,acd[,skip]*

0	1–2	3–4	5	6	7	8–9	10–11	12	13–15
1	ACS	ACD	1	1	1	SH	C	#	SKIP

Forms the logical AND of the contents of two accumulators.

Sets the value of carry to the specified value. Places carry and the logical AND of ACS and ACD in the shifter. Each bit placed in the shifter is one only if the corresponding bit in both ACS and ACD is one; otherwise, the resulting bit is zero. The instruction then performs the specified shift operation and places the result in carry and ACD if the no-load bit is zero. If the skip condition is true, the next sequential word is skipped.

AND Immediate

ANDI *i,ac*

1	1	0	AC		1	1	1	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

IMMEDIATE FIELD	
16	31

Forms the logical AND of the contents of the immediate field and the contents of the specified accumulator.

Block Add And Move

BAM

1	0	0	1	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Moves memory words from one location to another, adding a constant contained in AC0 to each one.

Moves words sequentially from one memory location to another treating them as unsigned, 16-bit integers.

The instruction adds two unsigned, 16-bit integers, one from AC0 and one from the source location, and transfers the new word to the destination location. Carry remains unchanged.

Bits 1-15 of AC2 contain the address of the source location. Bits 1-15 of AC3 contain the address of the destination location. The address in bits 1-15 of AC2 or AC3 is an indirect address if bit 0 of that accumulator is one. In that case, the instruction follows the indirection chain before placing the resultant effective address into the accumulator.

The unsigned 15-bit number in AC1 is equal to the number of words to be moved. This number must be greater than zero and less than or equal to 32,768 (77777_8). If the number in AC1 is outside this range, no data are moved and the contents of the accumulators remain unchanged.

AC	Contents
0	Addend
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the count in AC1 is decremented by one and the source and destination addresses in AC2 and AC3 are incremented by one. Upon completion of the instruction, the contents of AC0 remain unchanged, AC1 contains zeroes, and AC2 and AC3 point to the address following the last word in their respective fields.

Words are moved in consecutive, ascending order according to their addresses. The next address after 77777_8 is zero for both fields. The fields may overlap in any way.

> **NOTE:** *Because of the potentially long time that may be required to perform this instruction, it is interruptible. If a* **BAM** *instruction is interrupted, the program counter is decremented by one before being placed in location 0 so that it points to the interrupted instruction. Any interrupt service routine that returns control to the interrupted program via the address stored in location 0 will correctly restart the* **BAM** *instruction.*

When updating the source and destination addresses, the **BAM** instruction forces bit 0 of the result to zero. This ensures that upon return from an interrupt, the instruction will not try to resolve an indirect address in either AC2 or AC3.

Example

Operation	Before	After

Move 20_8 words from memory locations 077770+ to memory locations 034450+, adding 31_8 to each.	AC0 = 000031
	AC1 = 000020
	AC2 = 077770
	AC3 = 034450
		AC0 = 000031
		AC1 = 000000
		AC2 = 000011
		AC3 = 034471

Block Move

BLM

1	0	1	1	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Moves memory words from one location to another.

The **BLM** instruction is the same as the **BAM** instruction in all respects except that no addition is performed and AC0 is not used.

> **NOTE:** *The* **BLM** *instruction is interruptible in the same manner as the* **BAM** *instruction.*

Set Bit To One

BTO *acs,acd*

1	ACS		ACD		1	0	0	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Forms a bit pointer from ACS and ACD and sets the addressed bit in memory to one, leaving ACS and ACD unchanged.

ACS contains the most-significant 16 bits of the bit pointer, and ACD contains the least-significant 16 bits. If you specify ACS and ACD as the same accumulator, the contents of that accumulator become the least significant 16 bits of the bit pointer. The most-significant 16 bits are zero.

> **NOTE:** *Bit 0 of the bit pointer must be zero. The bit pointer contained in ACS and ACD must not make an indirect memory reference.*

Example

Operation	Before	After

BTO 0,1
Set bit 7 at 23456_8
to 1.
	AC0 = 023450_8
	AC1 = 000147_8
	23456_8=010103_8
		AC0 = 023450_8
		AC1 = 000147_8
		23456_8=010503_8

Set Bit To Zero

BTZ *acs,acd*

1	ACS		ACD		1	0	0	0	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the addressed bit to zero.

Forms a 32-bit pointer from the contents of ACS and ACD. ACS contains the most significant 16 bits and ACD contains the least significant 16 bits of the bit pointer.

If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the least significant 16 bits of the bit pointer and assumes that the most significant 16 bits are zero.

The instruction then sets the addressed bit in memory to zero, leaving the contents of ACS and ACD unchanged.

Compare To Limits

CLM *acs,acd*

1	ACS		ACD		1	0	0	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Compares a signed integer with two other integers and skips if the first integer is between the other two.

Compares the signed two's complement integer in ACS to two signed, two's complement limit values, *L* and *H*. If the number in ACS is greater than or equal to *L* and less than or equal to *H*, the next sequential word is skipped. If the number in ACS is less than *L* or greater than *H*, the next sequential word is executed.

If you specify ACS and ACD as different accumulators, the address of the limit value *L* is contained in bits 1-15 of the ACD. The limit value *H* is contained in the word following *L*. Bit 0 of ACD is ignored.

If you specify ACS and ACD as the same accumulator, that accumulator must contain the integer to be compared. The limit values *L* and *H* are in the two words following the instruction. *L* is the first word, and *H* is the second word. The next sequential word is the third word following the instruction.

Since the number contained in AC1 is between the limit values the next instruction is skipped.

Example

Compare 000331_8 to *L* stored at 001234_8 and *H* stored at 001235_8.

Instructions: **CLM** 1, 0
Skipped instruction
Executed instruction
AC0 = 001234_8
AC1 = 000331_8
Location 001234_8 = 000017_8
Location 001235_8 = 001112_8

Since the number in AC1 is between the limit values, the processor skips the next sequential word.

Character Compare
CMP

1	1	0	1	1	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Compares two strings of bytes and returns a code reflecting the results of the comparison.

Compares two strings, one byte at a time. Treats each byte as an unsigned 8-bit binary quantity in the range $0\text{-}255_{10}$. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the lower valued string. Both strings remain unchanged.

The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addressed) for each string.

AC0 specifies the length and direction of comparison for string 2. If the string is compared from its lowest memory location to its highest, AC0 contains the unsigned value of the number of bytes in string 2. If the string is compared from its highest memory location to its lowest, AC0 contains the two's complement number of the number of bytes in string 2.

AC1 specifies the length and direction of comparison for string 1. If the string is compared from its lowest memory location to its highest, AC1 contains the unsigned value of the number of bytes in string 1. If the string is compared from its highest memory location to its lowest, AC1 contains the two's complement number of the number of bytes in string 1.

AC2 contains a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in decending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

The strings may overlap in any way. Overlap will not affect the results.

After the instruction, AC0 contains the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table below.

Code Returned in AC1	Result
−1	string 1 < string 2
0	string 1 = string 2
+1	string 1 > string 2

AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality were found) or to the byte following string 2 (if string 2 were exhausted).

AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality were found) or to the byte following string 1 (if string 1 were exhausted). If the two strings are of unequal length, the instruction pads the shorter string with space characters $<040>_8$ and continues the comparison.

If the lengths of both strings 1 and 2 are zero, the instruction returns zero in AC1.

Example

Operation	Before	After

Compare the byte string at memory locations 000345 to 000400 to the byte string starting at location 011123, from the lowest to the highest location.	AC0 = 000066_8
	AC1 = 000066_8
	AC2 = 000713_8
	AC3 = 022247_8
		AC0 = 000000
		AC1 = code from table
		AC2 = 001001_8
		AC3 = 22335_8

Character Move Until True

CMT

1	1	1	0	1	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range 0-255_{10}) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter, and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is one, the byte pending is a delimiter. In this case, the instruction does not copy the pending byte, so the instruction terminates.

The instruction processes the string in the same direction, either from lowest memory locations to highest (ascending order), or from highest memory locations to lowest (descending order). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

The accumulator affects the **CMT** instruction as follows:

- AC0 contains the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.
- AC1 specifies the length of the strings and the direction of processing. If the source string is moved to the destination field in ascending order, AC1 contains the unsigned value of the number of bytes in the source string. If the source string is moved to the destination string in descending order, AC1 contains the negative two's complement number of bytes in the source string.
- AC2 contains a byte pointer to the first byte to be written to in the destination field. When the process is performed in ascending order, AC2 points to the lowest byte in the destination string. When the process is performed in descending order, AC2 points to the highest byte in the destination string.
- AC3 contains a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, AC3 points to the lowest byte in the source string. When the process is performed in descending order, AC3 points to the highest byte in the source string.
- The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects. If AC2=AC3, the instruction reads until it finds a character or the instruction returns to the address of a delimiter.

Upon completion:

- AC0 contains the resolved address of the delimiter table.
- AC1 contains the number of bytes that were not moved.
- AC2 contains a byte pointer to the byte following the last byte written in the destination field.
- AC3 contains a byte pointer either to the delimiter or to the first byte following the exhausted source string.

NOTE: *When the source and destination addresses are the same, no data are moved. Any other type of overlap may produce unusual side effects. If AC1 contains the number zero at the beginning of this instruction, no bytes are fetched and none are stored.*

Example

Operation	**Before**	**After**

Check the string starting at memory location 001132	AC0 = 010203
	AC1 = 000042
		010203
		000000 (if no delimiter found)

for a delimiter, beginning at the	AC2 = 002265	002337 (if no delimiter found)
string's lowest byte, and replace the string to its original location. The string consists of 42_8 bytes.	AC3 = 002265	002337 (if no delimiter found)

① carry is indicator
② AC0 returns # chars to compare, including current char –

Character Move

CMV

1	1	0	1	0	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Moves a string of bytes from one area of memory to another. Under control of the four accumulators, the **CMV** instruction moves a string of bytes from one area of memory to another and returns a value in the carry bit reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination string, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each string.

- AC0 specifies the length and direction of processing for the destination string. If the string is processed from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in the destination string. If the string is processed from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in the destination string.
- AC1 specifies the length of the strings and the direction of processing. If the source string is to be moved to the destination string in ascending order, AC1 contains the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination string in descending order, AC1 contains the two's complement of the number of bytes in the source string.
- AC2 contains a byte pointer to the first byte to be written into in the destination string. When the field is written in ascending order, the value of AC2 points to the lowest byte. When the field is written in descending order, AC2 points to the highest byte.
- AC3 contains a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, AC3 points to the lowest byte. When the field is copied in descending order, AC3 points to the highest byte.

The strings may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion:

- AC0 contains zero,
- AC1 contains the number of bytes left to fetch from the source field.
- AC2 contains a byte pointer to the byte following the destination field.
- AC3 contains a byte pointer to the byte following the last byte fetched from the source field.

NOTE: *If AC0 contains the number zero at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is zero at the beginning of this instruction, the destination string is filled with space characters.*

Example

Operation	Before	After

Move a string of bytes starting at memory location 003321, to the location starting at memory location 001111. The string to be moved is 30_8 bytes long, the destination string is 40_8 bytes long, and both are to be processed in ascending order.	AC0 = 000040
	AC1 = 000030
	AC2 = 002223
	AC3 = 006643
	Carry = X
		AC0 = 000000
		AC1 = 000000
		AC2 = 002263
		AC3 = 006673
		Carry = 0

The last 10_8 bytes of the destination string are filled with space characters, and since the source string is shorter than the destination string, a zero is returned in carry.

interruptable.

Count Bits

COB *acs,acd*

1	ACS	ACD	1	0	1	1	0	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Adds a number equal to the number of ones in ACS to the signed, 16-bit, two's complement number in ACD. The contents of ACS and the value of carry are unchanged.

> **NOTE:** *If you specify ACS and ACD as the same accumulator, the contents of ACS will be changed.*

Example

Operation	Before	After

COB 1, 0	AC0 = 123450
	AC1 = 000103
		AC0 = 123453
		AC1 = 000103

Complement

COM*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	0	0	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Forms the logical complement of the contents of an accumulator.

Sets carry to the specified value. Forms the logical complement of the number in ACS and places the 17-bit value (carry and function result) in the shifter. Performs the specified shift operation and places the result in carry and ADC if the no-load bit is zero. If the skip condition is true, the next sequential word is skipped.

Character Translate

CTR

1	1	1	0	0	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

Operating in one of two modes — translate and move, or translate and compare — the **CTR** instruction, in conjunction with the four accumulators, translates a string of bytes, one byte at a time, from one data representation to another data representation. It uses each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value and is then either moved to another area of memory or compared to a second translated string.

The instruction processes both strings from the lowest memory location to highest (ascending).

> **NOTE:** *The fields may overlap in any way. However, the instruction processes bytes one at a time, so certain types of overlap may produce unusual side effects.*

Translate and Move Mode

The instruction translates each byte in the source string and moves it to the destination string. This mode is specified by a one in bit 0 of AC1.

The accumulators affecting the **CTR** instruction are:

- AC0 contains the address, direct or indirect, of a word which contains a byte pointer to the first byte in the translation table.
- AC1 contains the two's complement of the number of bytes in both strings.

  NOTE: *Since bit 0 of AC1 contains a 1 for translate and move mode, the maximum number of bytes to be moved is 32,767*$_{10}$.

- AC2 contains a byte pointer to the first byte in the destination string.
- AC3 contains a byte pointer to the first byte in the source string.

Upon completion:

- AC0 contains the resolved address of the word that contains the byte pointer to the translation table.
- AC1 contains 0.
- AC2 contains a byte pointer to the byte following the last byte written into the destination string.
- AC3 contains a byte pointer to the byte following the last byte moved in the source string.

Example

Operation	Before	After

CTR Translate a 50-byte string, starting at memory location 010321, using a pre-defined translation table, and return the string to its original memory location. Memory location 002123 contains the byte pointer to the translation table.	AC0 = 002123$_8$
	AC1 = 177716$_8$
	AC2 = 020643$_8$
	AC3 = 020643$_8$
		AC0 = 002123$_8$
		AC1 = 000000$_8$
		AC2 = 020725$_8$
		AC3 = 020725$_8$

The source string is now overwritten and its location is filled with a translated string.

Translate and Compare Mode

This mode is specified by a zero in bit 0 of AC1. The instruction translates each byte in both string 1 and string 2 and compares the translated values. Each translated byte is treated as an unsigned, 8-bit binary quantity in the range 0 to 255$_{10}$. If two translated bytes are not equal, the string whose byte has the smaller numerical value is defined as the lower valued string. Both strings remain unchanged.

The accumulators affecting the **CTR** instruction are:

- AC0 contains the address, direct or indirect, of a word which contains a byte pointer to the first byte in the translation table.
- AC1 contains the unsigned value of the number of bytes in the strings.

  NOTE: *Since bit 0 of AC1 contains a 0 for translate and compare mode, the maximum number of bytes that may be compared is 32,767*$_{10}$.

- AC2 contains a byte pointer to the first byte in string 2.
- AC3 contains a byte pointer to the first byte in string 1.

Upon completion:

- AC0 contains the resolved address of the word that contains the byte pointer to the translation table.
- AC1 contains a return code as calculated in the following table.

Code	Result
−1	Translated value of string 1< translated value of string 2
0	Translated value of string 1= translated value of string 2
+1	Translated value of string 1> translated value of string 2

If the length of both strings 1 and 2 is zero, the compare option returns a zero in AC1.

- AC2 contains a byte pointer to either the failing byte in string 2 if an inequality is found, or the byte following the last byte in string 2 if the strings are identical.
- AC3 contains a byte pointer to either the failing byte in string 1 if an inequality is found, or the byte following the last byte in string 1 if the strings are identical.

Decimal Add

DAD *acs,acd*

1	ACS	ACD	0	0	0	1	0	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses the value of carry for a decimal carry.

Adds the unsigned decimal digit contained in ACS bits 12–15 to the unsigned decimal digit contained in ACD bits 12–15. Adds the value of carry to this result. Places the decimal units' position of the final result in ACD bits 12–15, and the decimal carry in the carry bit. The contents of ACS and bits 0–11 of ACD remain unchanged.

> **NOTE:** *No validation of the input digits is performed. Therefore, if bits 12–15 of either ACS or ACD contain a number greater than 9, the results will be unpredictable.*

Example

Assume that bits 12–15 of AC2 contain nine; bits 12–15 of AC3 contain seven; and the value of carry is zero. After the instruction **DAD 2,3** is executed, AC2 remains the same; bits 12–15 of AC3 contain six; and the value of carry is one, indicating a decimal carry from this *Decimal Add*. (Refer to Figure 10.1.)

	Before						After					
AC2	0	000	000	000	001	001	0	000	000	000	001	001
AC3	0	000	000	000	000	111	0	000	000	000	000	110
Carry	0						1					

DG-06798

Figure 10.1 Decimal addition

Double Hex Shift Left

DHXL *n,ac*

1	N	AC	0	1	1	1	0	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Shifts the 32 bits contained in AC (the most significant 16 bits) and AC+1 (the least significant 16 bits) left N+1 hex digits. Bits shifted out are lost. Vacated bit positions become zeros. If you specify AC as AC3, AC+1 is AC0. Carry remains unchanged.

> **NOTE:** *DGC assemblers compute N. You code* n, *which is N+1. Code the exact number of hex digits you want shifted. If* n *is equal to four, all of AC+1 shifts into AC. AC+1 fills with zeros.*

Example

Operation	Before	After

DHXL 1,0
Shift the 32-bit number contained in AC0 (most significant bits) and AC1 (least significant bits) left one hex digit.
	AC0 = 001160_8
	AC1 = 050010_8
		AC0 = 023405_8
		AC1 = 000200_8

Double Hex Shift Right

DHXR *n,ac*

1	N	AC	0	1	1	1	1	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Shifts the 32 bits contained in AC (the most-significant 16 bits) and AC+1 (the least-significant 16 bits) right N+1 hex digits. Bits shifted out are lost. Vacated bit positions become zeros. If you specify AC as AC3, AC+1 is AC0. Carry remains unchanged.

> **NOTE:** *DGC assemblers compute N. You code* n, *which is N+1. Code the exact number of hex digits you want shifted. If* n *is equal to four, all of AC shifts into AC+1. AC+1 fills with zeros.*

Example

Operation	Before	After

DHXR 2,0
Divide 23450000103_8 by 256_{10}. That's the same as shifting it right two hex digits.
	AC0 = 001160_8
	AC1 = 050010_8
		AC0 = 023405_8
		AC1 = 000200_8

Data In A

DIA*[f]* *ac,device*

0	1	1	AC	0	0	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from the A buffer of an I/O device to an accumulator.

The contents of the A input buffer in the specified device are placed in the specified AC. Sets the Busy and Done flags according to the function specified by *f.*

The format of the AC after the transfer is device, dependent.

If the specified device does not exist, the AC will contain 177777_8 after the transfer.

Example

Operation	Before	After
DIAC 1,TTI	AC1 = 010101_8	AC1 = 000112_8
Teletype		
A input buffer	112_8	112_8
Done flag	1_8	0

Read Memory Fault Address

DIA*[f]* *ac,***ERCC**

0	1	1	AC	0	0	1	F	0	0	0	0	1	0
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Places the sixteen low-order bits of the physical address of the fault location into the specified accumulator. (The previous contents of that accumulator are overwritten.) The instruction sets the Done flag as specified by the flag command.

The following shows the format of the contents of the specified accumulator.

LOW-ORDER ADDRESS BITS
0 15

NOTE: *The physical address is meaningless unless it is read after the ERCC facility requests an interrupt and before a* Start *or* **IORST** *flag command sets the Done flag to 0.*

Read MAP Status

DIA *ac*,MAP

0	1	1	AC		0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Reads the status of the current map.

Places the contents of the MAP status register in the specified AC. The previous contents of the AC are overwritten. The format of the information placed in the specified AC is

> **NOTE:** *The* **IORST** *instruction will clear bits 0 through 5 and 13 through 15 of the MAP status register.* **IORST** *also turns off the map.*

NME	MPN	I/O	WP	IND	SC	MAP			LEF	I/O	WP	IND	NME	DCH	UM
0	1	2	3	4	5	6		8	9	10	11	12	13	14	15

Bits	Name	Contents or Function
0,13	NME	Depending on the bit settings, the last *Load Map Status* **DOA MAP** instruction enabled: Bit 0 Bit 13 User Enabled 0 0 A 0 1 B 1 0 C 1 1 D
1	MPN	MAP state -- one indicates mapping is enabled; zero indicates mapping is disabled.
2	I/O	If one, the last protection fault was an I/O protection fault.
3	WP	If one, the last protection fault was a write protection fault.
4	IND	If one, the last protection fault was an indirect protection fault.
5	SC	If one, the last map reference was a *MAP Single Cycle* (**NIOP MAP**) instruction.
6-8	MAP	Specifies which map was loaded by the last **LMP** instruction as follows: Bit 6 Bit 7 Bit 8 User Enabled 0 0 0 A 0 0 1 C 0 1 0 B 0 1 1 D 1 0 0 Data Channel 1 0 0 Data Channel A 1 0 1 Data Channel C 1 1 0 Data Channel B 1 1 1 Data Channel D 1 0 1 Reserved 1 1 0 Reserved 1 1 1 Reserved
9	LEF	If one, the **LEF** instruction was enabled for the last user.
10	I/O	If one, I/O protection was enabled for the last user.
11	WP	If one, write protection was enabled for the last user.
12	IND	If one, indirect protection was enabled for the last user.
14	DCH	If one, the mapping of data channel addresses has been *enabled.*
15	UM	User mode. If one, the last I/O interrupt occurred while in user mode (map enabled).

Data In B

DIB*[f]* *ac,device*

0	1	1	AC	0	1	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from the B buffer of an I/O device to an accumulator.

Places the contents of the B input buffer of the specified device in the specified AC. Sets the Busy and Done flags according to the function specified by *f*.

The format of the AC after the transfer is device dependent. If the specified device does not exist, the AC contains 177777_8 after the transfer.

Read Memory Fault Code

DIB*[f]* *ac,***ERCC**

0	1	1	AC	0	1	1	F	0	0	0	0	1	0
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Places a 6-bit error identification code in bits 0-5 of the specified accumulator. The instruction first sets bits 6-11 of the accumulator to 0 and places the four high-order bits of the physical address of the fault location in bits 12-15. Next, the instruction sets the Done flag as specified by the flag command.

The following table shows the format of the contents of the specified accumulator.

FAULT		HIGH-ORDER ADDR
0 5	6 11	12 15

Bits	Name	Contents or Function	
0-5	Fault Code	6-bit code, identifies the bit in error.	
		000000	No error.
		000001	Check bit 5.
		000010	Check bit 4.
		000011	Two bit error.
		000100	Check bit 3.
		000101	Two bit error.
		000110	Two bit error.
		000111	Multiple bit error.
		001000	Check bit 2.
		001001	Two bit error.
		001010	Two bit error
		001011	Data bit 15.
		001100	Two bit error.
		001101	Data bit 13.
		001110	Data bit 7.
		001111	Two bit error.
		010000	Check bit 1.
		010001	Two bit error.
		010010	Two bit error.
		010011	Multiple bit error.
		010100	Two bit error.
		010101	Data bit 12.
		010110	Data bit 6.
		010111	Two bit error.
		011000	Two bit error.
		011001	Data bit 10.
		011010	Data bit 4.
		011011	Two bit error.

Bits	Name	Contents or Function
0-5	Fault Code	6-bit code, identifies the bit in error
		011100 Data bit 0.
		011101 Two bit error.
		011110 Two bit error.
		011111 Multiple bit error.
		100000 Check bit 0.
		100001 Two bit error.
		100010 Two bit error.
		100011 Data bit 14
		100100 Two bit error.
		100101 Data bit 11.
		100110 Data bit 5.
		100111 Two bit error.
		101000 Two bit error.
		101001 Data bit 9.
		101010 Data bit 3.
		101011 Two bit error.
		101100 Multiple bit error.
		101101 Two bit error.
		101110 Two bit error.
		101111 Multiple bit error.
		110000 Two bit error.
		110001 Data bit 8.
		110010 Data bit 2.
		110011 Two bit error.
		110100 Data bit 1.
		110101 Two bit error.
		110110 Two bit error.
		110111 Multiple bit error.
		111000 Multiple bit error.
		111001 Two bit error.
		111010 Two bit error.
		111011 Multiple bit error.
		111100 Two bit error.
		111101 Multiple bit error.
		111110 Multiple bit error.
		111111 Two bit error.
6-11	—	Reserved for future use.
12-15	High Order Address	Four high-order bits of the physical address of the fault location.

NOTE: *The address is meaningless unless read after the ERCC facility requests an interrupt and before a* Start *or* **IORST** *flag command sets the Done flag to 0.*

Data In C

DIC*[f]* *ac,device*

0	1	1	AC	1	0	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from the C buffer of an I/O device to an accumulator.

Places the contents of the C input buffer in the specified device in the specified AC. Sets the Busy and Done flags according to the function specified by *f*.

The format of the AC after the transfer is device dependent. If the specified device does not exist, the AC contains 177777_8 after the transfer.

Page Check

DIC *ac*,MAP

0	1	1	AC		1	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The number of the physical page which corresponds to the logical page specified by the preceding **DOC MAP** instruction is placed in bits 6–15 of the specified AC. Places additional information about the correspondence in bits 0–5. The previous contents of the AC are overwritten. The format of the information placed in the specified AC is

WP	MAP				PHYSICAL	
0	1	3	4	5	6	15

Bits	Name	Contents or Function
0	WP	The write protect bit for the logical page which corresponds to the physical page specified by bits 6-15.
1-3	Map	The map which was used to perform the translation between logical page number and physical page number as follows: Bit 1 Bit 2 Bit 3 User Enabled 0 0 0 A 0 0 1 C 0 1 0 B 0 1 1 D 1 0 0 Data Channel A 1 0 1 Data Channel C 1 1 0 Data Channel B 1 1 1 Data Channel D 1 0 1 Reserved 1 1 0 Reserved 1 1 1 Reserved
4-5	—	Reserved for future use.
6-15	Physical page	The number of the page which corresponds to the logical page given in the preceding **DOC MAP** instruction.

NOTE: *If all physical page bits including the write protect bit are one, then the logical page is validity protected.*

CPU Status

DIS*[f]* *ac*,CPU

0	1	1	AC		1	1	1	0	0	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Returns the status of the CPU status register and places this data into the specified accumulator.

NOTE: DIS *0* **CPU** *is equivalent to the* **SKP** *0* **CPU** *instruction.*

The information contained in the specified accumulator is in the format:

PF	ION	1	BRK	PUP	HLT	DH	IRQ	PL	-	TRP	SUR	MEM		-	DG
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Mnem	Bit	Name	Function (If Set to 1)
PF	0	Power Fail	Power Fail flag set.
ION	1	Interrupt On	Interrupt On flag set.
1	2	—	Set to one.
BRK	3	Break Key Interrupt	Interrupt resulting from depression of console Break key. Used only by the virtual console.
PUP	4	Power Up Reset	Power came up since last DOAP CPU. Used only by the virtual console.
HLT	5	Halt	**Halt** instruction was executed. Used only by the virtual console.
DH	6	Halt Dispatch	If one, indicates that a Halt instruction will cause the processor to enter the virtual console. If zero, indicates that a Halt instruction will cause the processor to halt.
IRQ	7	Interrupt	An interrupt is being requested.
PL	8	Program Load	Indicates the program load console key has been depressed.
—	9	—	Undefined.
TRP	10	Trap	Indicates that the virtual console single-instruction mode is in use.
SUR	11	Survived Memory Data Valid	Indicates that memory data is valid following a power disruption.
MEM TYP	12-13	Memory Type	Indicates capacity of implemented system memory as follows: 01 256 Kbytes 10 128 Kbytes 11 512 Kbytes
—	14	—	Undefined.
DG	15	Diagnostic Test	Indicates that the virtual console is looping; for diagnostic purposes only.

Data In Status

DIS*[f]* *ac,device*

0	1	1	AC	1	1	1	0	0	DEVICE CODE
0	1	2	3 4	5	6	7	8	9	10 15

Returns the status of the addressed device and places this data into the specified accumulator.

The accumulator must be specified as 1, 2, or 3. The **DIS** instruction uses the same operation code as the **SKP** instruction. **DIS** 0 **CPU** is equivalent to the **SKP** 0 **CPU** instruction.

The information contained in the specified accumulator is in the following format for I/O devices:

D	B	1	RESERVED FOR FUTURE USE
0	1	2	3 15

D Device is done if set to one.
B Device is busy if set to one.

> **NOTE:** *If the device does not exist, the contents of the specified AC will be 037777_8.*

Unsigned Divide

DIV

1	1	0	1	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the unsigned, 32-bit integer in AC0 (the most-significant 16 bits) and AC1 (the least-significant 16 bits) by the unsigned, 16-bit number in AC2.

The quotient and remainder are both unsigned, 16-bit numbers. The quotient occupies AC1, and the remainder occupies AC0.

Sets carry to zero. Leaves AC2 unchanged.

> **NOTE:** *Before performing the divide, the instruction compares the number in AC0 to the number in AC2. (This is an unsigned compare.) If the number in AC0 is greater than or equal to that in AC2, an overflow results. The carry bit becomes one, and the instruction terminates. All operands remain unchanged.*

Example

Operation	Before	After

DIV
Divide 6 by 2.
	AC0 = 000000_8
	AC1 = 000006_8
	AC2 = 000002_8
	Carry = 0
		AC0 = 000000_8
		AC1 = 000003_8
		AC2 = 000002_8
		Carry = 0

DIV
Divide
7777600001_8
by 077777_8.
	AC0 = 037777_8
	AC1 = 000001_8
	AC2 = 077777_8
	Carry = 0
		AC0 = 000000_8
		AC1 = 077777_8
		AC2 = 0
		Carry = 0

DIV
Divide
37777400001_8
by 177777_8
Overflow results.
	AC0 = 177776_8
	AC1 = 000001_8
	AC2 = 077777_8
	Carry = 0
		AC0 = 177776_8
		AC1 = 000001_8
		AC2 = 077777_8
		Carry = 1

DIV
Divide 7 by 2.
Get a remainder of 1.
	AC0 = 000000_8
	AC1 = 000007_8
	AC2 = 000002_8
	Carry = 0
		AC0 = 000001_8
		AC1 = 000003_8
		AC2 = 000002_8
		Carry = 0

Signed Divide

DIVS

1	1	0	1	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the 32-bit, two's complement number in AC0 (the most-significant 16 bits) and in AC1 (the least-significant 16 bits) by the 16-bit, two's complement number in AC2.

The quotient and remainder are both 16-bit, two's complement numbers. The quotient occupies AC1, and the remainder occupies AC0. The rules of algebra determine the quotient's sign. The remainder's sign is always the same as the dividend's sign, except that a zero quotient or a zero remainder is always positive.

Sets carry to zero. Leaves AC2 unchanged.

> **NOTE:** *If the quotient is too large to fit into AC1, an overflow results. The carry bit becomes one, and the instruction terminates. When this happens, the contents of AC0 and AC1 are unpredictable.*

Example

Operation	Before	After

DIVS
Divide 6 by 2.
	AC0 = 000000_8
	AC1 = 000006_8
	AC2 = 000002_8
		AC0 = 000000_8
		AC1 = 000003_8
		AC2 = 000002_8

DIVS
Divide
7777600001_8
by 077777_8.
	AC0 = 037777_8
	AC1 = 000001_8
	AC2 = 077777_8
		AC0 = 000000_8
		AC1 = 077777_8
		AC2 = 077777_8

DIVS
Divide 1 by -1
	AC0 = 000000_8
	AC1 = 000001_8
	AC2 = 177777_8
		AC0 = 000000_8
		AC1 = 177777_8
		AC2 = 177777_8

DIVS
Divide -7 by -2.
Get a remainder
of- 1.
	AC0 = 177777_8
	AC1 = 177777_8
	AC2 = 177776_8
		AC0 = 177777_8
		AC1 = 000003_8
		AC2 = 177776_8

Sign Extend And Divide

DIVX

1	0	1	1	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Extends the sign of one accumulator into a second accumulator and performs a **DIVS** operation on the result.

Extends the sign of the number in AC1 into AC0 by placing a copy of bit 0 of AC1 in each bit of AC0. After extending the sign, the instruction performs a **DIVS** operation.

> **NOTE:** *If the quotient is too large to fit into AC1, an overflow results. The processor sets carry to one and terminates the instruction operation. The contents of AC0 and AC1 are unpredictable.*

Double Logical Shift

DLSH *acs,acd*

1	ACS	ACD	0	1	0	1	1	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Shifts the 32-bit number in ACD (the most significant 16 bits) and in ACD+1 (the least significant 16 bits) either left or right. AC3+1 is AC0.

The sign of the 8-bit, two's complement number in bits 8–15 of ACS determines the direction of the shift. If this number is positive, the shift is to the left. If it is negative, the shift is to the right. If the number is zero, there is no shift. Bits 0–7 of ACS remain unchanged.

The magnitude of the 8-bit, two's complement number in bits 8–15 of ACS determines the number of bits to be shifted. Bits shifted out are lost. Vacated bit positions become zeroes.

Carry remains unchanged. ACS also remains unchanged unless, of course, ACS is ACD+1.

> **NOTE:** *If the magnitude of the number in bits 8–15 of ACS is greater than* 31_{10}, *all bits of ACD and ACD+1 become zero. Carry and ACS remain unchanged.*

Examples

Operation	Before	After

DLSH 0,1
Shift left one bit
	AC0 = 000001_8
	AC1 = 012345_8
	AC2 = 054321_8
		AC0 = 000001_8
		AC1 = 024712_8
		AC2 = 130642_8

DLSH 0,1
Shift right one bit
	AC0 = 000377_8
	AC1 = 024712_8
	AC2 = 130642_8
		AC0 = 000377_8
		AC1 = 012345_8
		AC2 = 054321_8

Data Out A

DOA*[f]* *ac,device*

0	1	1	AC	0	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from an accumulator to the A buffer of an I/O device.

Places the contents of the specified AC in the A output buffer of the specified device. Sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

Enable ERCC

DOA*[f]* *ac*,**ERCC**

0	1	1	AC	0	1	0	0	0	0	0	0	0	1	0
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Sets the ERCC facility to function according to bits 13-15 of the specified accumulator. Next, the instruction sets the Done flag and then the Interrupt Request flag, as specified by the flag command. The instruction disregards bits 0-12.

The following shows the format of the contents of the specified accumulator.

	MODE
0 12	13 15

Bits	Name	Contents or Function
0-12	—	Reserved for future use.
13-15	MODE	Control the ERCC feature as follows: 000 • Write checkcode; disable checking and correction during memory memory read; disable interrupts; enable checking and correction during memory refresh. 001 Disable writing checkcode; disable checking and correction during memory read disable interrupts; enable checking and correction during memory refresh. 010 Write checkcode; enable checking and correction during memory read and refresh; disable interrupts 011 Write checkcode; enable checking and correction during memory memory read and refresh; enable interrupts. 100 Write checkcode; disable checking and correction during memory read and refresh; disable interrupts. 101 Disable writing checkcode; disable checking and correction during memory read and refresh; disable interrupts 110 Write checkcode; enable checking and correction during memory memory read; disable interrupts; disable interrupts; disable checking and correction during memory refresh. 111 Write checkcode; enable checking and correction during memory memory read; enable interrupts; disable checking and correction during memory refresh.

Load MAP Status

DOA *ac*,**MAP**

0	1	1	AC	0	1	0	0	0	0	0	0	0	1	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Defines the parameters of a new map.

The contents of the specified AC are placed in the MAP status register. The previous contents of the AC remain unchanged.

The format of the specified AC is

NME	RESERVED	MAP	LEF	I/O	WP	IND	NME	DCH	UE
0 1	2 5	6 8	9	10	11	12	13	14	15

Bits	Name	Contents or Function
0,13	NME	Depending on the bit settings, the next user map enabled will be that for: Bit 0 Bit 13 User Enabled 0 0 A 0 1 B 1 0 C 1 1 D
1-5	Reserved	Reserved for future use.
6-8	MAP	Specifies which map will be loaded by the next **LMP** instruction as follows: Bit 6 Bit 7 Bit 8 User Enabled 0 0 0 A 0 0 1 C 0 1 0 B 0 1 1 D 1 0 0 Data Channel 1 0 0 Data Channel A 1 0 1 Data Channel C 1 1 0 Data Channel B 1 1 1 Data Channel D 1 0 1 Reserved 1 1 0 Reserved 1 1 1 Reserved
9	LEF	If one, the **LEF** instruction will be enabled for the next user.
10	I/O	If one, I/O protection will be enabled for the next user.
11	WP	If one, write protection will be enabled for the next user.
12	IND	If one, indirect protection will be enabled for the next user.
14	DCH Enable	If one, the mapping of data channel addresses will be enabled immediately after this instruction.
15	User Enable	If one, mapping of CPU addresses will commence with the first memory reference after the next indirect reference or return type instruction (**POPB, POPJ, RTN, RSTR**).

> **NOTE:** *If the* **DOA MAP** *instruction sets the User Enable bit to 1, the interrupt system is inhibited, and the MAP waits for an indirect reference or a return-type instruction. Either event releases the interrupt system and allows the MAP to begin translating addresses (using the user map specified by bits 0 and 13 of the MAP status register).*

Address translation resumes:

- *after the first level of the next indirect reference; or*
- *after a* **POPB, POPJ, RTN,** *or* **RSTR** *instruction.*

CPU Acknowledge

DOAP *ac,***CPU**

0	1	1	AC	0	1	0	1	1	1	1	1	1	1	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Clears the virtual console interrupts from the CPU status register.

> **NOTE:** *If a Halt instruction is performed, the virtual console clears the power fail interrupt. Never set bit 15.*

Data Out B

DOB*[f]* *ac,device*

0	1	1	AC	1	0	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from an accumulator to the B buffer of an I/O device.

Places the contents of the specified AC in the B output buffer of the specified device. Sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

Map Supervisor Page 31

DOB*[f]* *ac*,**MAP**

0	1	1	AC	1	0	0	0	0	0	0	0	0	1	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Specifies that mapping take place for a single page of an unmapped address space. Mapping is always done for locations 76000_8 through 77777_8 (logical page 31). This is the only page which can be mapped when in unmapped address space. You can use this instruction to access a page of a user's memory space when in unmapped mode. The MAP supervisor Page 31 instruction can only be used with the MAP off.

Bits 6–15 of the specified AC are transferred to the MAP feature. These bits specify a physical page number to which logical page 31 will be mapped when in the supervisor mode.

The contents of the specified AC remain unchanged. The format of the specified AC is

RESERVED	PHYSICAL
0 5	6 15

Bits	Name	Contents or Function
0-5	---	Reserved for future use.
6-15	Physical Page	The number of the physical page to which logical page 31 should be mapped when in supervisor mode.

NOTE: *If supervisor page 31 translation is altered while instructions are being fetched through supervisor page 31, instructions will be fetched from the new translation environment.* **IORST** *resets logical address translation to physical address translation.*

Data Out C

DOC*[f]* *ac,device*

0	1	1	AC	1	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from an accumulator to the C buffer of an I/O device.

Places the contents of the specified AC in the C output buffer of the specified device. Sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

Initiate Page Check

DOC *ac*,MAP

0	1	1	AC	1	1	0	0	0	0	0	0	0	1	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

The contents of the specified AC are transferred to the MAP feature for later use by the **DIC MAP** or **LMP** instruction. The contents of the specified AC remain unchanged. The format of the specified AC is

	LOGICAL	MAP	RESERVED
0	1 5	6 8	9 15

Bits	Name	Contents or Function
0	—	Reserved for future use.
1-5	Logical Page	Number of the logical page for which the check is requested.
6-8	Map	Specify which map should be used for check as follows:
		Bit 6 Bit 7 Bit 8 User Enabled
		0 0 0 A
		0 0 1 C
		0 1 0 B
		0 1 1 D
		1 0 0 Data Channel
		1 0 0 Data Channel A
		1 0 1 Data Channel C
		1 1 0 Data Channel B
		1 1 1 Data Channel D
		1 0 1 Reserved
		1 1 0 Reserved
		1 1 1 Reserved
9-15	—	Reserved for future use.

Decimal Subtract

DSB *acs,acd*

1	ACS	ACD	0	0	0	1	1	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses carry as a decimal borrow.

Subtracts the unsigned decimal digit contained in ACS bits 12–15 from the unsigned decimal digit contained in ACD bits 12–15. Subtracts the complement of carry from this result. Places the decimal units' position of the final result in ACD bits 12–15 and the complement of the decimal borrow in carry. In other words, if the final result is negative, the instruction indicates a borrow by setting carry to zero. If the final result is positive, the instruction indicates no borrow by setting carry to one. The contents of ACS and bits 0–11 of ACD remain unchanged.

Example

Assume that bits 12–15 of AC2 contain 9; bits 12–15 of AC3 contain 7; and carry contains zero. After the instruction **DSB 3,2** is executed, AC3 remains the same; bits 12–15 of AC2 contain one; and carry is set to one, indicating no borrow from the **DSB** instruction. (See Figure 10.2)

	Before	After
AC2	0 | 000 | 000 | 000 | 001 | 001	0 | 000 | 000 | 000 | 000 | 001
AC3	0 | 000 | 000 | 000 | 000 | 111	0 | 000 | 000 | 000 | 000 | 111
Carry	0	1

DG-06799

Figure 10.2 Decimal subtraction

Dispatch

DSPA *ac* @

1	1	0	AC		1	INDEX		0	0	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT	
16	17	31

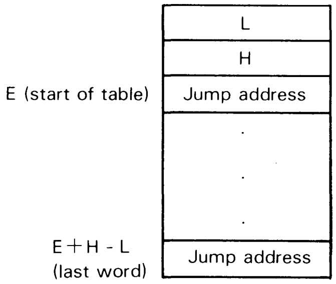
Conditionally transfers control to a location whose address is selected from a table in memory. Figure 10.3 illustrates the operation of this instruction.

Computes the effective address E. This is the address of H, the high limit, and L, the low limit. Each is a two's complement number. H is in location $E-1$, and L is in location $E-2$. The last entry in the dispatch table is in location $E+H-L$. Figure 10.4 illustrates the dispatch table.

Compares the two's complement *number* in the specified accumulator to the limit values. If the *number* is less than L or greater than H, operation continues with the word immediately following the **DSPA** instruction. If the *number* is greater than L and less than or equal to H, the processor fetches the word at location $E-L+$*number*.

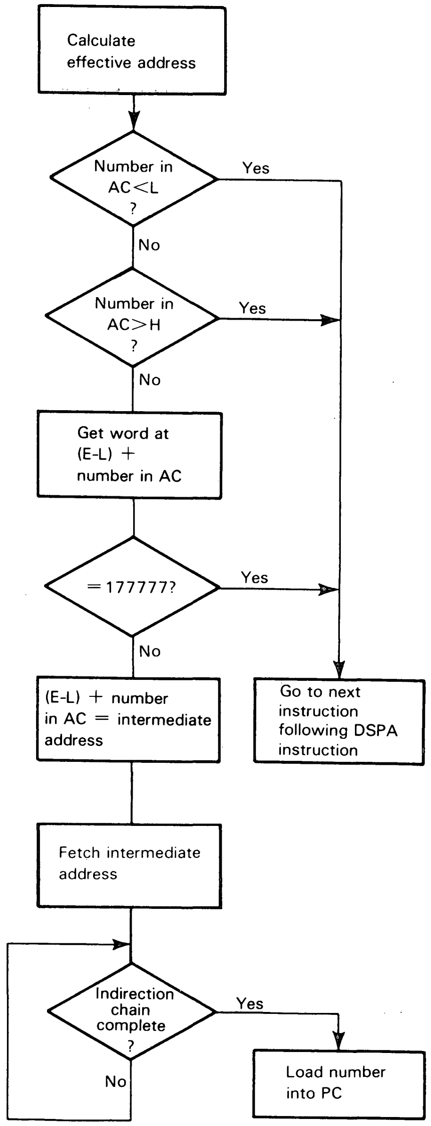
If the fetched word is 177777_8, operation continues with the word immediately following the **DSPA** instruction. Otherwise, the fetched word is the intermediate address in an effective address calculation.

After resolving the indirection, if any, the instruction places the effective address in the program counter. Operation continues with the word addressed by the updated program counter.



DG-09007

Figure 10.3 DSPA operation


Calculate effective address
Number in AC<L ?
Yes
No
Number in AC>H ?
Yes
No
Get word at (E-L) + number in AC
=177777?
Yes
No
(E-L) + number in AC = intermediate address
Go to next instruction following DSPA instruction
Fetch intermediate address
Indirection chain complete ?
Yes
No
Load number into PC

DG-08279

Figure 10.4 Dispatch table

Decrement And Skip If Zero

DSZ *[@]displacement[,index]*

0	0	0	1	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Decrements the addressed word, then skips if the decremented value is zero.

Computes the effective address. Decrements by one the addressed word and writes the result back into that location. If the updated value of the location is zero, the instruction skips the next sequential word. The CPU controls the memory bus until the instruction is completed.

Extended Decrement And Skip If Zero

EDSZ *[@]displacement[,index]*

1	0	0	1	1	1	INDEX	0	0	1	1	1	0	0	0
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
16	17 31

Decrements the addressed word, then skips if the decremented value is zero.

Computes the effective address. Decrements by one the contents of the addressed word and writes the result back into that location. If the updated value of the word is zero, the instruction skips the next sequential word. The CPU controls the memory bus until the instruction is completed.

Extended Increment And Skip If Zero

EISZ *[@]displacement[,index]*

1	0	0	1	0	1	INDEX	0	0	1	1	1	0	0	0
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
16	17 31

Increments the addressed word, then skips if the incremented value is zero.

Computes the effective address. Increments by one the contents of the location specified by the effective address, and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction skips the next sequential word. The CPU controls the memory bus until the instruction is completed.

Extended Jump

EJMP *[@]displacement[,index]*

1	0	0	0	0	1	INDEX	0	0	1	1	1	0	0	0
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
16	17 31

Computes the effective address and places it in the program counter. Operation continues with the word addressed by the updated program counter.

Example

Instruction:	**EJMP** address Next Sequential Instruction . . .
Address:	Next Executed Instruction

After execution of the **EJMP** instruction, the assembler replaces address with a 15-bit displacement. The program continues by executing the instruction contained at address.

Extended Jump To Subroutine

EJSR *[@]displacement[,index]*

1	0	0	0	1	1	INDEX		0	0	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT	
16	17	31

Increments and stores the value of the program counter in AC3, then places a new address in the program counter.

Computes the effective address. Places the address of the next sequential word (the word following the **JSR** instruction) in AC3. Places the effective address in the program counter. Operation continues with the word addressed by the updated program counter.

> **NOTE:** *The instruction computes the effective address before loading AC3 with the incremented program counter.*

Example

Instruction: **EJSR SUBR**
Next Instruction
.
.
.
SUBR: First **SUBR** (subroutine) Instruction

After execution of the **EJSR** instruction, the program continues operation with the first **SUBR** instruction and AC3 contains the address of the next instruction after the **EJSR** operation.

Extended Load Accumulator

ELDA *ac,[@]displacement[,index]*

1	0	1	AC		1	INDEX		0	0	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT	
16	17	31

Moves a copy of the contents of a memory word into the specified accumulator.

Calculates the effective address. Places the contents of the addressed location in the specified accumulator. The addressed location remains unchanged.

Extended Load Byte

ELDB *ac,displacement[,index]*

1	0	0	AC	1	INDEX	0	1	1	1	1	0	0	0
0	1	2	3 4	5	6 7	8	9	10	11	12	13	14	15

	DISPLACEMENT
16	17 31

Copies a byte from memory into an accumulator.

Forms a byte pointer from the displacement in the following way: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement to give a memory address. The byte indicator designates which byte of the addressed word will be loaded into bits 8–15 of the specified accumulator. The instruction sets bits 0–7 of the specified accumulator to zero.

The instruction overwrites the previous contents of the specified accumulator, but it does not alter either the index value or the displacement.

The argument *index* selects the source of the index value. It may have values in the range of 0–3. The meaning of each value is shown in the following table.

Index Bits	Index Value
00	0
01	Address of the displacement field (word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Example

Place the low byte of memory location 000740 into AC1.
Instruction: INST: **ELDB** 1,@BYTAB

```
.
.
.
BYTAB: 001701
.
.
.
000740: 016753
```

	Before	**After**
	AC1= 000000	AC1 = 000353

Load Effective Address

ELEF *ac,[@]displacement[,index]*

1	1	1	AC	1	INDEX	0	0	1	1	1	0	0	0
0	1	2	3 4	5	6 7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
16	17 31

Places an effective address in an accumulator.

Computes the effective address and places it in bits 1–15 of the specified accumulator. Sets bit 0 of the accumulator to zero. The previous contents of the accumulator are overwritten.

Example

```
ELEF  0,TABLE  ; The logical address of TABLE
               ; is placed in AC0.
ELEF  1,-55,3  ; Subtracts 000055₈ from
               ; the unsigned integer in AC3 and
               ; places the result in AC1.
ELEF  0,+0     ; Places the logical address of this
               ; ELEF
               ; instruction in AC0.
```

Extended Store Accumulator

ESTA *ac,[@]displacement[,index]*

1	1	0	AC		1	INDEX		0	0	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT	
16	17	31

Stores the contents of an accumulator into a memory location.

Computes the effective address. Places the contents of the specified accumulator in the addressed location. The contents of the specified accumulator remain unchanged.

Example

Operation	Before	After
ESTA 1, 1123 Store the contents of AC1 into memory location 001123.	AC1 = 010101 Location 001123 = XXXXXX	AC1 = 010101 Location 001123010101

Extended Store Byte

ESTB *ac,displacement[,index]*

1	0	1	AC		1	INDEX		0	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

	DISPLACEMENT	
16	17	31

Copies into memory the byte contained in the right half of an accumulator.

Forms a byte pointer from the displacement as follows: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement field to give a memory address. The byte indicator determines which byte of the addressed location will receive bits 8–15 of the specified accumulator.

The argument *index* selects the source of the index value. It may have values in the range of 0–3; the meaning of each value is shown in the table.

Index Bits	Index Value
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of AC2
11	Contents of AC3

Absolute Value

FAB *fpac*

1	1	0	FPAC		1	1	0	0	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the sign bit of FPAC to zero. Leaves bits 1–63 of FPAC unchanged. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FPAC.

Add Double (FPAC to FPAC)

FAD *facs,facd*

1	FACS		FACD		0	0	0	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Adds the floating-point number in FACS to the floating-point number in FACD and places the normalized result in FACD. Overwrites the previous contents of FACD, leaves the contents of FACS unchanged, and updates the **Z** and **N** flags in the FPSR to reflect the new contents of FACD.

Floating-point addition consists of an exponent comparison and a mantissa addition. The exponents of the two numbers are compared, and the mantissa of the number with the smaller exponent is shifted right until its exponent equals the larger number's exponent. This mantissa alignment is accomplished by taking the absolute value of the difference between the two exponents and shifting the mantissa right that number of hex digits. Hex shifting is repeated until both exponents are equal or all significant digits are shifted out of the mantissa. If all significant digits are shifted out of the mantissa, the operation is equivalent to adding the number with the larger exponent to zero. This requires a shift of at least 14 hex digits.

After alignment, the mantissas are added together. The result of this addition is termed the intermediate result. One guard digit is provided for the intermediate result, which is used if normalization is required. The sign of the intermediate result is determined from the signs of the two operands by the rules of algebra. If the mantissa addition produces a carry out of the most significant bit, the mantissa in the intermediate result is shifted right one hex digit and the exponent is incremented by one. If this shift produces an exponent overflow, the **OVF** bit is set in the FPSR, and the number in FACD is correct, except that the exponent is 128 too small.

If there is no mantissa overflow, the mantissa of the intermediate result is examined for leading hex zeros. If the mantissa is found to be all zeros, a true zero is placed in the FACD and the instruction terminates.

If the mantissa is nonzero, the intermediate result is normalized, and then placed in the FACD. If the normalization results in an exponent underflow, the **UNF** bit is set in the FPSR and the instruction is terminated. The number in the FACD is correct except that the exponent is 128 too large.

Add Double (Memory to FPAC)

FAMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	0	0	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Adds a double-precision, floating-point number in memory to the contents of a floating-point accumulator. The normalized result is located in FPAC.

Computes the effective address which addresses a 4-word (double-precision) operand in memory. The floating-point number addressed is added to the floating-point number in FPAC with the normalized, double-precision result placed in FPAC. The previous contents of FPAC are overwritten, the contents of the addressed memory location remain unchanged, and the **Z** and **N** flags of the FPSR are updated to reflect the new contents of FPAC.

Addition is accomplished as described in **FAD.**

Add Single (Memory to FPAC)

FAMS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	0	0	0	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Adds a single-precision, floating-point number from memory to the contents of a floating-point accumulator.

Computes the effective address which addresses a 2-word (single-precision) operand in memory. The floating-point number addressed is added to the floating-point number in FPAC with the normalized, single-precision result placed in FPAC. The previous contents of FPAC are overwritten, the contents of the addressed memory location remain unchanged, and the **Z** and **N** flags of the FPSR are updated to reflect the new contents of FPAC.

Single-precision addition is accomplished as described in **FAD** with the exception that the mantissa of the smaller number may be shifted a maximum of six hex digits before the number is considered as zero.

> **NOTE:** *Only 32 bits of the FPAC are used with one guard digit during single-precision addition. One guard digit is appended to each number involved in the operation. In the final result, bits 32–63 are set to zero.*

Add Single (FPAC to FPAC)

FAS *facs,facd*

1	FACS	FACD	0	0	0	0	0	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Adds the floating-point number contained in one FPAC to the contents of another FPAC.

Adds the floating-point number in FACS to the floating-point number in FACD and places the normalized, single-precision result in FACD, overwriting the previous contents of FACD. The **Z** and **N** flags of the FPSR are updated to reflect the new contents of FACD; the contents of FACS remain unchanged.

Single-precision addition is accomplished as described in **FAD** with the exception that the mantissa of the smaller number may be shifted a maximum of six hex digits before the number is considered as zero.

> **NOTE:** *Only 32 bits of the FPAC are used with one guard digit during single-precision addition. One guard digit is appended to each number involved in the operation. In the final result, bits 32–63 are set to zero.*

Clear Errors

FCLE

1	1	0	1	0	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets bits 0–4 of the FPSR to zero.

Compare Floating Point

FCMP *facs,facd*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	FACS		FACD		1	1	1	0	0	1	0	1	0	0	0

Compares two floating-point numbers and sets the **Z** and **N** flags in the FPSR accordingly.

Algebraically compares the floating-point numbers in FACS and FACD to each other and updates the **Z** and **N** flags in the FPSR to reflect the result. Leaves the contents of FACS and FACD unchanged. The results of the compare and the corresponding flag settings are shown in the table below.

Z	N	Result
1	0	FACS=FACD
0	1	FACS>FACD
0	0	FACS<FACD

NOTE: *Unnormalized operands give unspecified results.*

Divide Double (FPAC by FPAC)

FDD *facs,facd*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	FACS		FACD		0	0	1	1	1	1	0	1	0	0	0

Divides the floating-point number in FACD by the floating-point number in FACS and places the normalized result in FACD.

Divides the floating-point number in FACD by the floating-point number in FACS and places the normalized, double-precision result in FACD. The previous contents of FACD are overwritten, the contents of FACS remain unchanged, and the **Z** and **N** flags of the FPSR are updated to reflect the new contents of FACD.

If the mantissa in FACS is zero, the **DVZ** bit (divide by zero) is set in the FPSR and the instruction is terminated. The number in FACD remains unchanged.

If the mantissa in FACS is nonzero, the two mantissas are compared. If the mantissa of the number in FACD is greater than or equal to the mantissa of the number in FACS, the mantissa of the number in FACD is shifted right one hex digit and the exponent of the number in FACD is increased by one.

The mantissa in FACD is then divided by the mantissa in FACS and the quotient is the mantissa of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra.

The exponent in FACS is subtracted from the exponent in FACD and 64 is added to this result (the addition of 64 maintains the excess 64 notation). The result of the exponent manipulation becomes the exponent of the intermediate result. The result is normalized and placed in FACD. If the exponent processing produces either overflow or underflow, the corresponding bit in the FPSR is set (**OVF** or **UNF**, respectively); the exponent in FACD is correct, except that the exponent is 128 too small for overflow and 128 too large for underflow.

Divide Double (FPAC by Memory)

FDMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	1	1	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Divides the floating point number in FPAC by a double-precision (64-bit), floating-point number in memory and places the normalized result in FPAC.

Divides the floating-point number in FPAC by the 4-word (double-precision) floating-point number addressed by the effective address and places the normalized, double-precision result in FPAC. The previous contents of FPAC are overwritten, the contents of memory remain unchanged, and the **Z** and **N** flags of the FPSR are updated to reflect the new contents of FPAC.

Division using the **FDMD** instruction is similar to division using the **FDD** instruction with the exception that references to FACS should be the effective address.

Divide Single (FPAC by Memory)

FDMS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	1	1	0	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Divides the floating-point number in FPAC by a single-precision (32-bit), floating-point number in memory and places the normalized result in FPAC.

Divides the floating-point number in FPAC by the 2-word (single-precision) floating-point number addressed by the effective address and places the normalized, single-precision result in FPAC. The previous contents of FPAC are overwritten, and the contents of memory remain unchanged. The **Z** and **N** flags of the FPSR are updated to reflect the new contents of FPAC.

Division using the **FDMS** instruction is similar to division using the **FDD** instruction with the exception that references to FACS should be the effective address.

> **NOTE:** *The instruction initially sets bits 32–63 of the FPAC to zero. In the final result, bits 32–63 are set to zero.*

Divide Single (FPAC by FPAC)

FDS *facs,facd*

1	FACS		FACD		0	0	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the floating-point number in FACD by the floating-point number in FACS and places the normalized, single-precision result in FACD.

The previous contents of FACD are overwritten, and the contents of FACS remain unchanged. The **Z** and **N** flags of the FPSR are updated to reflect the new contents of FACD.

Division with the **FDS** instruction is similar to division with the **FDD** instruction.

NOTE: *Only bits 0 through 31 of the FPAC are used.*

Load Exponent

FEXP *fpac*

1	0	1	FPAC		1	1	0	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Places bits 1–7 of AC0 in bits 1–7 of the specified FPAC. Ignores bits 0 and 8–15 of AC0. Bits 0 and 8–63 of FPAC and the entire contents of AC0 are unchanged. Leaves bits 1–7 of FPAC unchanged if FPAC contains true zero. The **Z** and the **N** flags of the FPSR are updated to reflect the contents of the FPAC.

NOTE: *The exponent contained in bits 1–7 of AC0 is assumed to be in Excess 64 representation.*

Fix To AC

FFAS *ac,fpac*

0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15
1	AC	FPAC	1	0	1	1	0	1	0	1	0	0	0

Converts the integer portion of the floating-point number contained in the specified FPAC to a signed two's complement integer and places the result in an accumulator.

Forms the absolute value of the integer portion of the floating-point number in FPAC. Extracts the 15 least-significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result in the specified accumulator, sets the **Z** and **N** flags in the floating-point status register to ~~zero,~~ and leaves the contents of FPAC unchanged. NO — Z & N get set in most cases

If the number in FPAC is less than ~~-32,767~~ - 32768 or greater than +32,767, this instruction sets the **MOF** flag in the floating-point status register to one.

> **NOTE:** *If the lower 15 bits of the integer formed from the number in FPAC are all 0, the sign bit of the result will be zero regardless of the sign of the original number.*

if exponent > 15

if fraction, Z & N not modified

Fix To Memory

FFMD *fpac,[@]displacement[,index]*

0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15
1	INDEX	FPAC	1	0	1	1	1	1	0	1	0	0	0

0	1–15
@	DISPLACEMENT

Converts the integer portion of a floating-point number to double-precision integer format and stores the result in two memory locations.

Forms the absolute value of the integer portion of the floating-point number in FPAC. Extracts the 31 least-significant bits from this value and, if the number in FPAC is negative, forms the two's complement of the integer. Then places the result into the locations addressed by E, sets the **Z** and **N** flags in the floating-point status register to 0, and leaves the contents of FPAC unchanged. MSBs ?

If the number in FPAC is less than ~~-2147,483,647~~ -2,147,483,648 or greater than +2,147,483,647, this instruction sets the **MOF** flag in the floating-point status register to one.

> **NOTE:** *If the lower 31 bits of the integer formed the number in FPAC are all 0, the sign bit of the result will be zero.*

Halve

FHLV *fpac*

1	1	1	FPAC		1	1	0	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the floating-point number in FPAC by 2.

Shifts the mantissa contained in FPAC right one bit position, fills the vacated bit position with a zero, and places the bit shifted out in the guard digit. Normalizes the number and places the result in FPAC. Sets the **UNF** flag in the FPSR to one if the normalization process causes an exponent underflow. The number in FPAC is then correct, except that the exponent is 128 too large. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FPAC.

Integerize

FINT *fpac*

1	1	0	FPAC		1	1	0	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Zeros the fractional portion (if any) of the number contained in the specified FPAC, and then normalizes the number. The instruction updates the **Z** and **N** flags in the FPSR to reflect the new contents of the specified FPAC.

> **NOTE:** *If the absolute value of the number contained in the specified FPAC is less than one, the specified FPAC is set to true zero.*

Float From AC

FLAS *ac,fpac*

1	AC	FPAC	1	0	1	0	0	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Converts a two's complement number to floating-point format.

Converts the signed two's complement number contained in the specified accumulator to a single-precision floating-point number, places the result in the specified FPAC, and sets the 32 least significant bits of the FPAC to zero. Leaves the contents of the specified accumulator unchanged and overwrites the previous contents of the FPAC. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FPAC.

The range of numbers that can be converted is −32,768 to +32,767.

Load Floating-Point Double

FLDD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	1	0	0	0	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Moves four words out of memory into a specified FPAC.

Computes the effective address and places the double-precision floating-point number at that address in FPAC. Overwrites the previous contents of FPAC.

NOTE: *The data moved are not normalized* ~~*and*~~ but *the* **Z** *and* **N** *bits of the FPSR are* ~~*not*~~ *modified.*

Load Floating-Point Single

FLDS *fpac,[@]displacement[,index]*

1	INDEX		FPAC		1	0	0	0	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT	
0	1	15

Moves two words out of memory into a specified FPAC.

Computes the effective address and places the single-precision, floating-point number at that address in FPAC. Overwrites the previous contents of FPAC. The 32 least significant bits of FPAC are set to zero.

NOTE: *The data moved is not normalized ~~and~~ but the* **Z** *and* N *bits of the FPSR are ~~not~~ modified.*

Float From Memory

FLMD *fpac,[@]displacement[,index]*

1	INDEX		FPAC		1	0	1	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT	
0	1	15

Converts the contents of two memory locations to floating-point format and places the result in a specified FPAC.

Computes the effective address, converts the 32-bit, signed, two's complement number addressed to a double-precision floating-point number, and places the result in the specified FPAC. Overwrites the previous contents of FPAC, and updates the **Z** and **N** flags in the FPSR to reflect the new contents of the FPAC.

The range of numbers that can be converted is −2,147,483,648 to +2,147,483,647.

Load Floating-Point Status

FLST *[@]displacement[,index]*

1	0	1	INDEX	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Moves the contents of two specified memory locations to the FPSR.

Computes the effective address, places the addressed 32-bit operand in the FPSR, and sets the condition codes to the values of the loaded bits.

If the **FLST** instruction sets bits 0 to 4 of the FPSR to one, then the address of the **FLST** instruction is loaded into the ~~FPAC.~~ FPSR's PC NO– NOT ACCORDING TO DIAGS… ~~[illegible]~~

For more information on the FPSR, refer to Chapter 4, "Floating-Point Instructions" under Floating-Point Status Register.

> **NOTE:** *If bits 0 and 5 of the FPSR are set to one, a floating-point fault condition will initiate a floating-point trap.*

Multiply Double (FPAC by FPAC)

FMD *facs,facd*

1	FACS	FACD	0	0	1	0	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Multiplies a floating-point number by a floating-point number.

Multiplies the double-precision floating-point number in FACD by the double-precision floating-point number in FACS. Places the normalized, double-precision result in FACD. Updates the **Z** and **N** flags of the FPSR to reflect the new contents of FACD. The contents of FACS remain unchanged.

The mantissas of the two numbers are multiplied together to give the mantissa of the intermediate result. If normalization is required, one guard digit is provided for the intermediate result.

The exponents of the two operands are added together and 64 is subtracted (the subtraction of 64 maintains the Excess 64 notation).

The result of the exponent manipulation becomes the exponent of the intermediate result. The sign of the intermediate result is determined from the sign of the two operands by the rules of algebra.

If the exponent processing produces either overflow or underflow, the result is held until normalization. If normalization does not correct this condition, the corresponding flag (**OVF** or **UNF**) in the FPSR is set to one. The number in FACD is correct except that the exponent is 128 too small for exponent overflow and 128 too large for exponent underflow.

Multiply Double (FPAC by Memory)

FMMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	1	0	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Multiplies a floating-point number in FPAC by a double-precision, floating-point number in memory.

Multiplies the floating-point number in FPAC by the double-precision, floating-point number in the location specified by the effective address and places the normalized, double-precision result in FPAC. The previous contents of FPAC are overwritten and the contents of the effective address are unchanged. The **Z** and **N** flags of the FPSR are set to reflect the new contents of FPAC.

Multiplication using the **FMMD** instruction follows the format for multiplication using the **FMD** instruction.

Multiply Single (FPAC by Memory)

FMMS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	1	0	0	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Multiplies a floating-point number in FPAC by a single-precision, floating-point number in memory.

Multiplies the single-precision floating-point number in FPAC by the single-precision, floating-point number in the location specified by the effective address and places the normalized, single-precision result in FPAC. The previous contents of FPAC are overwritten and the contents of the effective address are unchanged. The **Z** and **N** flags of the FPSR are set to reflect the new contents of FPAC.

Multiplication using the **FMMS** instruction follows the format for multiplication using the **FMD** instruction.

> **NOTE:** *The instruction initially sets bits 32–63 of the FPAC to zero. In the final result, bits 32–63 are set to zero.*

Move Floating-Point

FMOV *facs,facd*

1	FACS		FACD		1	1	1	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Moves the contents of one FPAC to another FPAC.

Places the contents of FACS in FACD and overwrites the previous contents of FACD. The contents of FACS are unchanged. The **Z** and **N** flags in the FPSR are set to reflect the new contents of FACD.

Multiply Single (FPAC by FPAC)

FMS *facs,facd*

1	FACS		FACD		0	0	1	0	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Multiplies the single-precision floating-point number in FACD by the single-precision floating-point number in FACS and places the normalized, single-precision result in FACD.

The **Z** and **N** flags of the FPSR are updated to reflect the new contents of FACD; the contents of FACS remain unchanged.

Multiplication using the **FMS** instruction follows the format for multiplication using the **FMD** instruction.

> **NOTE:** *The instruction initially sets bits 32–63 of the FPAC to zero. In the final result, bits 32–63 are set to zero.*

Negate

FNEG *fpac*

1	1	1	FPAC		1	1	0	0	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Inverts the sign bit of FPAC. Bits 1–63 of FPAC remain unchanged. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FPAC. If FPAC contains pure zero, the sign bit remains unchanged.

Normalize

FNOM *fpac*

1	0	0	FPAC		1	1	0	0	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Normalizes a floating-point number.

Normalizes the floating-point number contained in FPAC. If an exponent underflow occurs, the **UNF** flag in the FPSR is set. This indicates that the exponent in FPAC is 128 too large. The **Z** and **N** flags of the FPSR are set to reflect the new contents of FPAC.

> **NOTE:** *If all the mantissa bits of the number in FPAC are zero, then the sign and exponent bits are set to zero (pure zero).*

The **FNOM** instruction is the only instruction that will accept impure zero input (zero mantissa with nonzero exponent or sign).

No Skip

FNS

1	0	0	0	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The next sequential word is executed.

Pop Floating-Point State

FPOP

1	1	1	0	1	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pops an 18-word floating-point return block off the user stack and alters the state of the floating-point unit. The words popped and their destinations are shown in Figure 10.5.

> **NOTE:** *These instructions are interruptible. Because the FACD, stack pointer, and program counter are not updated until the completion of these instructions, any interrupt service routines that return control to the interrupted program via the program counter stored in location 0 will correctly restart these instructions.*

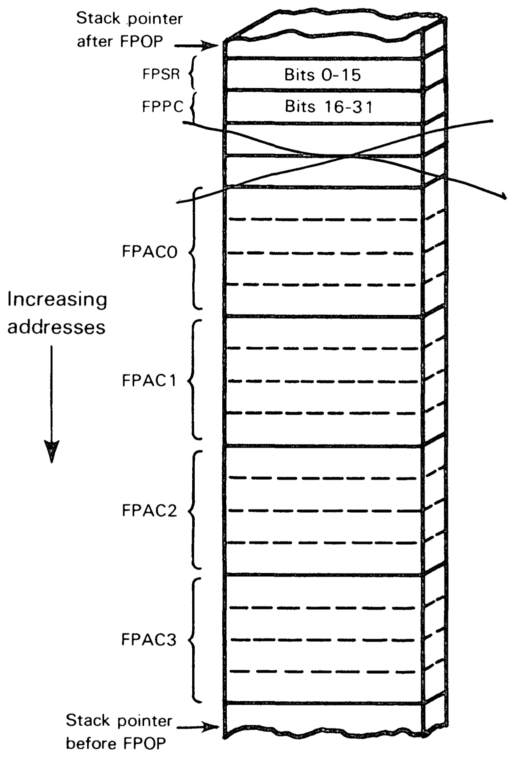


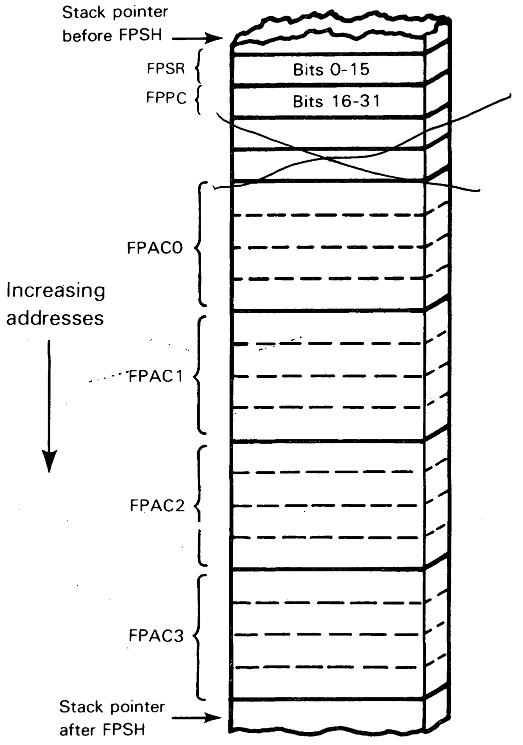
Figure 10.5 Words popped off user stack

Push Floating-Point State

FPSH

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	0	0	1	1	0	1	1	1	0	1	0	0	0

Pushes an 18-word floating-point return block onto the user stack, leaving the contents of the floating-point accumulators and the FPSR unchanged. The format of the 18 words pushed is shown in Figure 10.6.



DG-00604

Figure 10.6 Format of 18-word return block

Read High Word

FRH *fpac*

0	1	2	3-4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	FPAC	1	1	0	0	0	1	0	1	0	0	0

Places the 16 most significant bits of FPAC in AC0 and overwrites the previous contents of AC0. This does not change the contents of FPAC or the **Z** and **N** flags in the FPSR.

Skip Always

FSA

1	0	0	0	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skip the next sequential word.

Scale

FSCAL *fpac*

1	0	0	FPAC		1	1	0	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Shifts the mantissa of a floating-point number and replaces its exponent.

Shifts the mantissa of the floating-point number in FPAC either right or left, depending upon the contents of bits 1–7 of AC0. The contents of AC0 remain unchanged.

Treats bits 1–7 of AC0 as an exponent in excess 64 representation. Computes the difference between this exponent and the exponent in FPAC by subtracting the exponent in FPAC from the number contained in AC0 (bits 1–7).

- If the difference is zero, or the contents of FPAC are pure zero, the instruction stops.
- If the difference is positive, the instruction shifts the mantissa contained in FPAC right that number of hex digits.
- If the difference is negative, the instruction shifts the mantissa contained in FPAC left that number of hex digits. The **MOF** flag in the FPSR is always set if the difference is negative.

Bits shifted out of either end of the mantissa are overwritten. If the entire mantissa is shifted out of FPAC, the instruction sets FPAC to pure zero.

After this shift, the contents of bits 1–7 of AC0 replace the exponent contained in FPAC. If FPAC has previously been set to pure zero, the contents of FPAC are not replaced.

The instruction sets the **Z** and **N** flags of the FPSR to reflect the new contents of FPAC.

Subtract Double (FPAC from FPAC)

FSD

1	FACS		FACD		0	0	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Subtracts the double-precision floating-point number in FACS from the double-precision floating-point number in FACD and places the normalized result in the FACD. Overwrites the previous contents of FACD and leaves the contents of FACS unchanged. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FACD.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating-point addition. (See **FAD**.)

Skip On Zero

FSEQ

1	0	0	1	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the **Z** flag of the FPSR is one.

Skip On Greater Than Or Equal To Zero

FSGE

1	0	1	0	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the **N** flag of the FPSR is zero.

Skip On Greater Than Zero

FSGT

1	0	1	1	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if both the **Z** and **N** flags of the FPSR are zero.

Skip On Less Than Or Equal To Zero

FSLE

1	0	1	1	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential instruction if either the **Z** flag or the **N** flag of the FPSR is one.

Skip On Less Than Zero

FSLT

1	0	1	0	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the **N** flag of the FPSR is one.

Subtract Double (Memory from FPAC)

FSMD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	0	1	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 … 15

Subtracts the floating-point number in the source location from the floating-point number in FPAC and places the normalized result in the FPAC. Overwrites the previous contents of FPAC and leaves the contents of the source location unchanged. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FPAC.

The instruction computes the effective address, which addresses a 4-word (double-precision) operand.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating-point addition. (See **FAMD.**)

Subtract Single (Memory from FPAC)

FSMS *fpac,[@]displacement[,index]*

1	INDEX	FPAC	0	1	0	1	0	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 … 15

Subtracts the floating-point number in the source location from the floating-point number in FPAC and places the normalized result in the FPAC. Overwrites the previous contents of FPAC and leaves the contents of the source location unchanged. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FPAC.

The instruction computes the effective address, which addresses a 2-word (single-precision) operand.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating-point addition. (See **FAMS.**)

Skip On No Zero Divide

FSND

1	1	0	0	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential program word if the divide by zero (**DVZ**) flag of the FPSR is set to zero.

Skip On Nonzero

FSNE

1	0	0	1	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the **Z** flag of the FPSR is zero.

Skip On No Error

FSNER

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	0	1	0	1	0	1	0	0	0

Skips the next sequential word if bits 1–4 of the FPSR are all zero.

Skip On No Mantissa Overflow

FSNM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	0	1	1	0	1	0	1	0	1	0	0	0

Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is zero.

Skip On No Overflow
FSNO

1	1	1	0	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the overflow (**OVF**) flag of the FPSR is zero.

Skip On No Overflow and No Zero Divide
FSNOD

1	1	1	0	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if both the overflow (**OVF**) flag and the divide by zero (**DVZ**) flag of the FPSR are zero.

Skip On No Underflow

FSNU

1	1	0	1	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the underflow (**UNF**) flag of the FPSR is zero.

Skip On No Underflow And No Zero Divide

FSNUD

1	1	0	1	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if both the underflow (**UNF**) flag and the divide by zero (**DVZ**) flag of the FPSR are zero.

Skip On No Underflow And No Overflow
FSNUO

1	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if both the underflow (**UNF**) flag and overflow (**OVF**) flag of the FPSR are zero.

Subtract Single (FPAC from FPAC)
FSS *facs,facd*

1	FACS		FACD		0	0	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Subtracts the floating-point number in FACS from the floating-point number in FACD and places the normalized result in the FACD. Overwrites the previous contents of FACD and leaves the contents of FACS unchanged. Updates the **Z** and **N** flags in the FPSR to reflect the new contents of FACD.

The subtraction is performed by inverting the sign bit of the source operand and adding. After the sign inversion, the operation is equivalent to floating-point addition.

Store Floating-Point Status

FSST *,[@]displacement[,index]*

1	0	0	INDEX	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Moves the contents of the FPSR to two specified memory locations.

Computes the effective address and places the 32-bit contents of the FPSR in the two consecutive addressed memory locations. Leaves the contents of the FPSR unchanged.

For more information on the FPSR, refer to Chapter 4, "Floating-Point Instructions" under Floating-Point Status Register.

Store Floating-Point Double

FSTD *fpac,[@]displacement[,index]*

1	INDEX	FPAC	1	0	0	1	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Stores the contents of a specified FPAC into four memory locations.

Computes the effective address and places the floating-point number contained in FPAC in memory beginning at the addressed location. Overwrites the previous contents of the addressed memory location and leaves the contents of FPAC and the condition codes in the FPSR unchanged.

NOTE: *The data moved is not normalized.*

Store Floating-Point Single

FSTS *fpac,[@]displacement[,index]*

1	INDEX		FPAC		1	0	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

@	DISPLACEMENT	
0	1	15

Stores the contents of a specified FPAC into two memory locations.

Computes the effective address and places the floating-point number contained in FPAC in memory beginning at the addressed location. Overwrites the previous contents of the addressed memory location and leaves the contents of FPAC and the condition codes in the FPSR unchanged. For single precision, only the 32 most significant bits of FPAC are stored.

NOTE: *The data moved is not normalized.*

Trap Disable

FTD

1	1	0	0	1	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the trap enable bit of the FPSR to zero.

Trap Enable

FTE

1	1	0	0	0	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the trap enable bit of the FPSR to one.

> **NOTE:** *When a floating-point fault occurs and the trap enable bit is one, the trap enable bit is set to zero before control is transferred to the floating-point error handler. The trap enable bit should be set to one before normal processing is resumed.*

Halt

HALT
HALTA *ac*
DOC*[f]* *ac,***CPU**

0	1	1	AC		1	1	0	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Stops user program execution and returns to the virtual console program if the Halt Dispatch function is enabled by jumpering.

The **DOC***[f] ac,***CPU** instruction sets the Interrupt On flag according to the function specified in the *f* field, then stops the processor. If the Halt Dispatch function is not enabled, then while stopped, the processor will honor data channel requests, but will not honor program interrupt requests.

> **NOTE:** *The assembler recognizes the mnemonic* **HALT** *as equivalent to* **HALTA** *0.*

Halve

HLV *ac*

1	1	0	AC		1	1	0	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the 16-bit two's complement number in the specified accumulator by two. Truncates the result and stores it in the specified accumulator.

If the number in the accumulator is positive, the instruction performs the division by shifting the number right one bit. If the number in the accumulator is negative, the instruction performs the division by negating the number, shifting it right one bit, then negating it again.

Examples

Operation	Before	After
HLV 0 Divide 4523_8 by 2.	AC0 = 004523_8	AC0 = 002251_8
HLV 1 Divide −2 by 2.	AC1 = 177776_8	AC1 = 177777_8

Hex Shift Left

HXL *n,ac*

1	N		AC		0	1	1	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Shifts the contents of the specified accumulator left N+1 hex digits. Bits shifted out are lost. Vacated bit positions become zeroes.

> **NOTE:** *DGC assemblers compute N. You code* n, *which is N+1. Code the exact number of hex digits you want shifted. If* n *is four, AC fills with zeroes.*

Example

Operation	Before	After
HXL 2,1 Shift the 16-bit number in AC1 left 2 hex digits.	AC1 = 004523_8	AC0 = 051400_8

Hex Shift Right

HXR *n,ac*

1	N	AC	0	1	1	0	1	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Shifts the contents of the specified accumulator right N+1 hex digits. Bits shifted out are lost. Vacated bit positions become zeroes.

> **NOTE:** *DGC assemblers compute N. You code* n, *which is N+1. Code the exact number of hex digits you want shifted. If* n *is four, AC fills with zeroes.*

Example

Operation	Before	After
HXR 2,1 Shift the 16-bit number in AC1 right 2 hex digits.	AC1 = 004523_8	AC0 = 000011_8

Increment

INC[*c*][*sh*][*#*] *acs,acd[,skip]*

1	ACS	ACD	0	1	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Increments the contents of an accumulator.

Sets carry to the specified value. Increments the unsigned, 16-bit number in ACS by one. If the incremented value is greater than $2^{16}-1$, complements carry. Places the 17-bit value (carry and the result of the increment) in the shifter. Performs the specified shift operation.

Tests the skip condition. If the no-load bit is zero, loads the 17-bit value into the carry bit and ACD. If the skip condition is true, skips the next sequential word.

Interrupt Acknowledge

INTA
DIB *[f] ac,* **CPU**

0	1	1	AC		0	1	1	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Places a 6-bit device code in bits 10-15 of the specified accumulator. This device code identifies the highest priority device currently requesting an interrupt. Sets bits 0-9 of the specified accumulator to one.

After the transfer, the **DIB** mnemonic sets the Interrupt On flag according to the function specified by *f*.

The **INTA** mnemonic places the device code into bits 10-15 of the specified accumulator without affecting the ION flag.

Interrupt Disable

INTDS
NIOC CPU

0	1	1	0	0	0	0	0	1	0	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets Interrupt On flag to zero. This disables program interrupts.

Interrupt Enable

INTEN
NIOS CPU

0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets Interrupt On flag to one. This enables interrupts.

If this instruction changes the state of the Interrupt On flag from zero to one, the CPU allows one more instruction to execute before the first I/O interrupt can occur. However, if the instruction to be executed can be interrupted, then interrupts can occur as soon as the instruction begins to execute.

NOTE: *If the instruction uses only one CPU cycle, then the CPU allows two instructions to execute before the first I/O interrupt can occur. Refer to Appendix C, "Instruction Execution Times" for a list of CPU cycles.*

Inclusive OR

IOR *ac,acd*

1	ACS		ACD		0	0	1	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Forms the logical inclusive OR of the contents of ACS and the contents of ACD. Places the result in ACD.

This instruction sets a bit position in ACD to one if the corresponding bit positions in ACD and ACS are not both zero.

Example

Operation	Before	After

IOR 0, 1
Inclusive ORs the contents of AC0 with the contents of AC1.
	AC0 = 002245_8
	AC1 = 010133_8
		AC0 = 002245_8
		AC1 = 012377_8

Inclusive OR Immediate

IORI *i,ac*

1	0	0	AC		1	1	1	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

IMMEDIATE FIELD	
16	31

Forms the logical inclusive OR of the contents of the immediate field and the contents of the specified AC and places the result in the specified AC.

This instruction sets a bit position in the specified accumulator to one if the corresponding bit positions in the accumulator and the immediate field are not both zero.

Reset

IORST
DIC*[f]* *ac,***CPU**

0	1	1	AC		1	0	1	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the Busy and Done flags of all I/O devices to zero.

The **IORST** mnemonic sets the **ION** flag to zero. The assembler recognizes the mnemonic **IORST** as equivalent to **DICC** 0, **CPU**. Positions in ACD and ACS are not both zero.

The **DIC** mnemonic sets the **ION** flag according to the function specified by *f*. If you use this mnemonic, you must code an accumulator to avoid an assembly error. The processor, however, ignores the accumulator field. The specified accumulator remains unchanged.

The processor performs the equivalent of an **IORST** instruction at power-up, when you press the RESET switch (if the console is not locked), and when you type **I** from the virtual console.

NOTE: IORST *will not affect any bits in the FPSR.*

Increment And Skip If Zero

ISZ *[@]displacement[,index]*

0	0	0	1	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Increments the addressed word, then skips if the incremented value is zero.

Computes the effective address. Increments the addressed word by one. Stores the result in the same location. If the incremented value is zero, the instruction skips the next sequential word.

The CPU controls the memory bus until the instruction is completed.

Jump

JMP *[@]displacement[,index]*

0	0	0	0	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Computes the effective address and places it in the program counter. Operation continues with the word addressed by the updated program counter.

Jump To Subroutine

JSR *[@]displacement[,index]*

0	0	0	0	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Computes the effective address, then places the address of the next sequential word in AC3. Places the effective address in the program counter. Operation continues with the word addressed by the updated program counter.

> **NOTE:** *The instruction computes the effective address before it places the incremented program counter in AC3.*

Load Accumulator

LDA *ac,[@]displacement[,index]*

0	0	1	AC	@	INDEX	DISPLACEMENT
0	1	2	3 4	5	6 7	8 15

Copies a word from memory to an accumulator.

Computes the effective address. Places the word addressed by the effective address into the specified accumulator. The contents of the addressed location remain unchanged.

Load Byte

LDB *acs,acd*

1	ACS	ACD	1	0	1	1	1	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Uses a byte pointer contained in ACS to load a byte from memory into bits 8–15 of ACD. Sets bits 0–7 of ACD to zero. The contents of ACS remain unchanged, unless, of course, ACS and ACD are the same accumulator.

Load Effective Address

LEF *ac,[@]displacement[,index]*

0	1	1	AC	@	INDEX	DISPLACEMENT
0	1	2	3 4	5	6 7	8 15

Computes an effective address, placing it into an accumulator.

Places the calculated effective address into bits 1–15 of the specified AC. Bit 0 of the AC is set to zero.

> **NOTE:** *The* **LEF** *instruction can only be used in a mapped system while in mapped mode.*

With the LEF mode bit set to one, all I/O and **LEF** instructions will be interpreted as **LEF** instructions.

With the LEF mode bit set to zero, all I/O and **LEF** instructions will be interpreted as I/O instructions.

Be sure that I/O protection is enabled or the LEF mode bit is set to one before using the **LEF** instruction. If you issue a **LEF** instruction in the I/O mode, with protection disabled, the instruction will be interpreted and executed as an I/O instruction, with possible undesirable results.

```
LEF   0,TABLE      ; The logical address of
                   ; TABLE is placed in AC0.
LEF   1,-55,3      ; Subtracts 000055_8
                   ; from the unsigned 15-bit integer
                   ; in AC3 and the result is
                   ; placed in AC1.
LEF   0, . +0      ; Places the address of this
                   ; LEF
                   ; instruction in AC0.
```

Load Map
LMP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0

Loads successive words from memory into the MAP.

Words are loaded in consecutive, ascending order according to their addresses.

The accumulators affecting the **LMP** instruction are:

- AC0 must contain zero.
- AC1 contains an unsigned integer which is the number of words to be loaded into the MAP.
- AC2 contains the address of the first word to be loaded. If bit 0 is one, the instruction follows the indirection chain and places the resultant effective address into AC2.
- AC3 is ignored and its contents remain unchanged.

For each word loaded, the instruction decrements the count in AC1 by one and increments the source address in AC2 by one.

Upon completion of the **LMP** instruction:

- AC0 remains unchanged.
- AC1 contains zero.
- AC2 contains the address of the word following the last word loaded.

The words loaded into the MAP define the address translation functions for the various user and data channel maps. The contents of the MAP field (bits 6–8) of the MAP status register determine which map is affected by the **LMP** instruction. You can alter this field by using either the **DOA MAP** or the **DOC MAP** instruction.

The format of the words loaded into the MAP is

WP	LOGICAL	PHYSICAL
0	1 – 5	6 – 15

Bits	Name	Contents or Function
0	Write Protect	Write protect for user maps. Map faults may not occur during memory references initiated by the data channel.
1-5	Logical	Logical page number.
6-15	Physical	Physical page number.

NOTE: *To declare a logical page invalid, set the Write Protect bit to 1 and bits 6–15 to 1.*

NOTE: *The* **LMP** *instruction is interruptible in the same manner as the* **BAM** *instruction.*

If you issue this instruction while in mapped mode, with I/O protection enabled, the map and accumulator contents are not altered, and a MAP fault occurs.

If the **LMP** instruction alters the translation of the page indicated by the program counter for the next instruction fetch, this causes the instruction to be fetched from the new translation environment.

Locate Lead Bit

LOB *acs,acd*

1	ACS		ACD		1	0	1	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Adds a number equal to the number of high-order zeroes in the contents of ACS to the signed, 16-bit, two's complement number contained in ACD. The contents of ACS and the value of carry remain unchanged.

> **NOTE:** *If ACS and ACD are specified as the same accumulator, the instruction functions as described above, except that since ACS and ACD are the same accumulator, the contents of ACS will be changed.*

Locate and Reset Lead Bit

LRB *acs,acd*

1	ACS		ACD		1	0	1	0	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Performs **LOB** instruction and sets the lead bit to zero.

Adds a number equal to the number of most significant zeroes in the contents of ACS to the signed, 16-bit, two's complement number contained in ACD. Sets the leading 1 in ACS to zero. The value of carry remains unchanged.

> **NOTE:** *If ACS and ACD are specified to be the same accumulator, then the instruction sets the leading 1 in that accumulator to zero, and no count is taken.*

Logical Shift

LSH *acs,acd*

1	ACS	ACD	0	1	0	1	0	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Shifts the 16-bit number in ACD either left or right.

The sign of the 8-bit, two's complement number in bits 8–15 of ACS determines the direction of the shift. If this number is positive, the shift is to the left. If it's negative, the shift is to the right. If the number is zero, there is no shift. Bits 0–7 of ACS remain unchanged.

The magnitude of the 8-bit, two's complement number in bits 8–15 of ACS determines the number of bits to be shifted. Bits shifted out are lost. Vacated bit positions become zeros.

The carry bit and ACS remain unchanged.

> **NOTE:** *If the magnitude of the number in bits 8–15 of ACS is greater than 15_{10}, all bits of ACD become zero.*

Examples

Operation	Before	After

LSH 0,1
Shift left one bit.
	AC0 = 000001_8
	AC1 = 012345_8
		AC0 = 000001_8
		AC1 = 024712_8

LSH 0,1
Shift right one bit.
	AC0 = 0003777_8
	AC1 = 024712_8
		AC0 = 000377_8
		AC1 = 012345_8

Move

MOV*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	0	1	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Moves the contents of an accumulator through the arithmetic logic unit (ALU).

Sets carry to the specified value. Places carry and the contents of ACS in the shifter. Performs the specified shift operation.

Test the skip operation. If the no-load bit is zero, loads the result of the shift into the carry bit and ACD. If the skip condition is true, skips the next sequential word.

Mask Out

MSKO *ac*

DOB*[f]* *ac*,**CPU**

0	1	1	AC		1	0	0	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the priority mask.

The **DOB***[f] ac,* **CPU** instruction places the contents of the specified accumulator into the priority mask. After the transfer, the CPU Interrupt On flag is set according to the function specified by *f*. The contents of AC remain unchanged.

If the device priority bit equals 1, the instruction sets the I/O device Interrupt Disable flag. All I/O device controllers respond to **MSKO, IORST,** and **INTA**. Only the CPU responds to such instructions as **INTEN, INTDS, HALT,** and **VCT**.

> NOTE: *A one in any bit disables interrupt requests from devices which use that bit as a mask.*

Modify Stack Pointer

MSP *ac*

1	0	0	AC		1	1	0	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Changes the value of the stack pointer and tests for potential overflow.

Adds the 16-bit two's complement number in the specified accumulator to the stack pointer and places the result in the stack pointer (location 40_8).

Checks for overflow by comparing the number now in location 40_8 to the stack limit. If the new stack pointer is greater than the stack limit, restores the stack pointer's original value and pushes a return block into the stack. The final stack pointer is the original stack pointer plus five. Ther return block is in the following format.

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	Bit 0 equals carry. Bits 1 to 15 equal PC (the address of MSP instruction).

The program counter in the return block contains the address of the **MSP** instruction.

After pushing the return block, the program counter contains the address of the stack fault routine. Control transfers to the stack fault routine.

Unsigned Multiply

MUL

1	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Multiplies the unsigned, 16-bit number in AC1 by the unsigned, 16-bit number in AC2 to yield an unsigned, 32-bit product. Adds the unsigned, 16-bit number in AC0 to the product. The result is an unsigned, 32-bit number that occupies AC0 (the most-significant 16 bits) and AC1 (the least significant 16-bits). AC2 and carry remain unchanged. Because the result is a double-length number, overflow cannot occur.

Examples

Operation	Before	After

MUL
Multiply 3 by 2.
	AC0 = 000000_8
	AC1 = 000003_8
	AC2 = 000002_8
		AC0 = 000000_8
		AC1 = 000006_8
		AC2 = 000002_8

MUL
Multiply 077777_8 by 077777_8.
	AC0 = 000000_8
	AC1 = 077777_8
	AC2 = 077777_8
		AC0 = 037777_8
		AC1 = 000001_8
		AC2 = 077777_8

MUL
Multiply 177777_8 by 177777_8.
	AC0 = 00000_8
	AC1 = 177777_8
	AC2 = 177777_8
		AC0 = 177776_8
		AC1 = 000001_8
		AC2 = 077777_8

MUL
Multiply 3 by 2 and add a remainder of 1.
	AC0 = 000001_8
	AC1 = 000003_8
	AC2 = 000002_8
		AC0 = 000000_8
		AC1 = 000007_8
		AC2 = 000002_8

Signed Multiply

MULS

1	1	0	0	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Multiplies the unsigned, 16-bit two's complement number in AC1 by the 16-bit two's complement number in AC2 to yield an 32-bit two's complement product. Adds the 16-bit, two's complement number in AC0 to the product. The result is a 32-bit two's complement number that occupies AC0 (the most-significant 16 bits) and AC1 (the least-significant 16 bits). AC2 and carry remain unchanged. Because the result is a double-length number, overflow cannot occur.

Examples

Operation	Before	After

MULS
Multiply 3 by 2.
	AC0 = 000000_8
	AC1 = 000003_8
	AC2 = 000002_8
		AC0 = 000000_8
		AC1 = 000006_8
		AC2 = 000002_8

MULS
Multiply 077777_8 by 077777_8.
	AC0 = 000000_8
	AC1 = 077777_8
	AC2 = 077777_8
		AC0 = 037777_8
		AC1 = 000001_8
		AC2 = 077777_8

MULS
Multiply -1 by -1.
	AC0 = 00000_8
	AC1 = 177777_8
	AC2 = 177777_8
		AC0 = 000000_8
		AC1 = 000001_8
		AC2 = 177777_8

MULS
Multiply 3 by −2 and add a remainder of −1.
	AC0 = 177777_8
	AC1 = 000003_8
	AC2 = 177776_8
		AC0 = 177777_8
		AC1 = 177771_8
		AC2 = 177776_8

Negate

NEG*[c][sh][#]*   *acs,acd[,skip]*

1	ACS	ACD	0	0	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Forms the two's complement of the contents of an accumulator.

Sets carry to the specified value. Takes the two's complement of the 16-bit number in ACS. Places the 17-bit value (carry and the result of the negate) in the shifter. Performs the specified shift operation.

Tests the skip condition. If the no-load bit is zero, places the 17-bit value in the carry bit and ACD. If the skip condition is true, skips the next sequential word.

> **NOTE:** *If ACS contains zero, the instruction complements carry.*

No I/O Transfer

NIO *[f]*   *ac,device*

0	1	1	0	0	0	0	0	F	DEVICE CODE
0	1	2	3	4	5	6	7	8 9	10 15

Used when a Busy or Done flag must be changed with no other operation taking place.

Sets the Busy and Done flags in the specified device according to the function specified by *f*.

Map Single Cycle
Disable User Mode
NIOP MAP

0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The effect of this instruction depends upon the mode from which it is issued.

> NOTE: *The interrupt system is disabled from the beginning of the* MAP Single Cycle *instruction until after the next* **LDA, ELDA, STA,** *or* **ESTA** *instruction.*

From user (mapped) mode:
If the LEF mode and I/O protection are disabled, the NIOP instruction turns off the MAP. All subsequent memory references are unmapped until the map is reactivated with a *Load Map Status* instruction.

From the unmapped mode:
The user map is enabled for one memory reference. The first memory reference of the next **LDA, ELDA, STA,** or **ESTA** instruction is mapped. After the memory cycle is mapped, the user map is again disabled.

For example, if AC2 contains 405_8 and the following instruction sequence is issued:

```
.
.
NIOP    MAP     ;MAP SINGLE CYCLE
LDA     3,2,2
.
.
```

then the logical address 407_8 will be mapped using the last enabled user map (specified by bits 0 and 13 of the MAP status register at the time of the memory reference). The word contained in the corresponding physical location will be placed in AC3.

However, if the following instruction sequence is issued:

```
.
.
NIOP    MAP     ;MAP SINGLE CYCLE
LDA     3,@2,2
.
.
```

then the logical address 407_8 will be mapped using the user map for the last enabled user. The contents of the corresponding physical location will be used as the first level of an indirection chain. The next memory cycle, which is the second level of the indirection chain, will not be mapped.

Pop Multiple Accumulators
POP *acs,acd*

1	ACS		ACD		1	1	0	1	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pops one to four words off the stack and places them in the specified accumulators.

Loads the accumulators in descending order, starting with ACS and ending with ACD. ACD−1 is AC3. If you specify ACS and ACD as the same accumulator, the instruction pops one word off the stack and places it in that accumulator.

Decrements the stack pointer by the number of accumulators loaded. Leaves the frame pointer unchanged.

Example

Instruction: **POP** 3,1

	Before	After
AC0	xxxxxx	Untouched
AC1	xxxxxx	Third word popped
AC2	xxxxxx	Second word popped
AC3	xxxxxx	First word popped

xxxxxx denotes unknown contents which are overwritten.

Pop Block
POPB

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	1	1	1	1	1	1	0	0	1	0	0	0

Pops five words off the stack and places them in predetermined locations. The words popped and their destinations are as follows:

Word Popped	Contents
1	Bit 0 = carry bit Bits 1-15 = PC
2	AC3
3	AC2
4	AC1
5	AC0

Continues operation with the word addressed by the updated program counter. The frame pointer remains unchanged.

Pop PC And Jump
POPJ

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0

Pops the top word off the stack and places it in the program counter. Continues execution with the word addressed by the updated value of the program counter.

Decrements the stack pointer by one. Leaves the frame pointer unchanged.

Push Multiple Accumulators

PSH *acs,acd*

1	ACS	ACD	1	1	0	0	1	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Pushes the contents of one to four accumulators onto the stack.

Pushes accumulators in ascending order starting with ACS and ending with ACD. AC3 + 1 is AC0. If you specify ACS and ACD as the same accumulator, the instruction pushes that accumulator onto the stack.

Increments the stack pointer by the number of accumulators pushed. Leaves the frame pointer unchanged.

Checks for overflow after completing the entire push operation.

Push Jump

PSHJ *,[@]displacement[,index]*

1	0	0	0	0	1	INDEX	1	0	1	1	1	0	0	0
0	1	2	3	4	5	6 7	8	9	10	11	12	13	14	15

@	DISPLACEMENT
0	1 15

Pushes the address of the next sequential instruction onto the stack, computes the effective address, and places it in the program counter. Continues execution with the word addressed by the updated value of the program counter.

Push Return Address

PSHR

1	0	0	0	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes the address of this instruction *plus 2* onto the stack.

Read Virtual Console Registers

READS *ac*

DIA*[f]* *ac*,CPU

0	1	1	AC		0	0	1	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Places the contents of the virtual console register into an accumulator.

After the transfer, sets the Interrupt On flag according to the function specified by *f*.

HS	LCK	RESERVED		DEVICE CODE	
0	1	2	9	10	15

Mnem	Bit	Instruction	Action (If Set to 1)
HS	0	High Speed	Auto load program is loaded from a high-speed device.
LCK	1	Lock	Front panel is locked.
—	2-9	—	Reserved.
—	10-15	Device Code	Specifies device to perform the auto load.

Restore
RSTR

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	0	1	1	1	1	1	1	0	0	1	0	0	0

Pops nine words off the stack and places them in predetermined locations. The words popped and their destinations are as follows:

Word Popped	Contents
1	Bit 0 equals carry bit Bits 1-15 = PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Stack fault pointer
7	Stack limit
8	Frame pointer
9	Stack pointer

Continues operation with the word addressed by the updated program counter.

Return
RTN

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	0	1	1	1	1	1	1	0	0	1	0	0	0

Pops a return block off the stack.

Returns control from subroutines that issue a **SAVE** instruction at their entry points. The **SAVE** instruction loads the current value of the stack pointer into the frame pointer. The **RTN** instruction places the contents of the fram pointer into the stack pointer and pops five words from the stack.

Word Popped	Destination
First	Bit 0 equals carry bit Bits 1 to 15 equals program counter
Second	AC3
Third	AC2
Fourth	AC1
Fifth	AC0

The operations occur as follows:

1. The contents of the frame pointer are loaded into the stack pointer.
2. Bit 0 of the location addressed by the new stack pointer is loaded into carry and bits 1 to 15 are loaded into the program counter. The stack pointer decrements by one.
3. The contents of the location now addressed by the stack pointer are loaded into AC3. The stack pointer decrements by one.
4. The contents of the location now addressed by the stack pointer are loaded into AC2. The stack pointer decrements by one.
5. The contents of the location now addressed by the stack pointer are loaded into AC1. The pointer decrements by one.
6. The contents of the location now addressed by the stack pointer are loaded into AC0. The stack pointer decrements by one. The stack pointer now points to the new top of stack.
7. The contents of AC3 are loaded into the frame pointer.

Continues operation with the word addressed by the updated program counter.

Save

SAVE *i*

1	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

IMMEDIATE FIELD	
0	15

Saves the information required by the **RTN** instruction.

Pushes a return block onto the stack. Then, places the value of the incremented stack pointer in the frame pointer and in AC3. Leaves AC0, AC1, and AC2 unchanged. The following table shows the format of the 5-word return block.

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Frame pointer before the save
5	Bit 0 equals carry bit Bit 1 to 15 equals bits 1 to 15 of AC3

The operations occur as follows:

1. The stack pointer increments by one. The contents of AC0 are written into the location now addressed by the stack pointer.
2. The stack pointer increments by one. The contents of AC1 are written into the location now addressed by the stack pointer.
3. The stack pointer increments by one. The contents of AC2 are written into the location now addressed by the stack pointer.
4. The stack pointer increments by one. A zero is written into bit 0, and the contents of the frame pointer are written into bits 1 to 15 of the location now addressed by the stack pointer.
5. The stack pointer increments by one. The carry is written into bit 0, and bits 1 to 15 of AC3 are written into bits 1 to 15 of the location now addressed by the stack pointer.
6. Checks for stack overflow. An overflow occurs if the stack pointer exceeds the stack limit. If an overflow occurs, pushes a stack fault block on the stack, overwriting the first return block.
7. If an overflow has not occurred, puts the stack pointer in the frame pointer and in AC3.
8. If an overflow has not occurred, adds the immediate field to the stack pointer.

The **SAVE** instruction allocates a portion of the stack for use by the procedure which executed the **SAVE**. After storing the stack pointer in the frame pointer and pushing the return block, the **SAVE** instruction adds the unsigned 16-bit integer contained in its immediate field to the stack pointer. This unsigned integer is called the frame size.

The frame size defines a portion of the stack not normally accessed by push and pop operations. The procedure issuing the **SAVE** instruction uses this area for temporary storage of variables, counters, etc. The frame pointer whose value goes into AC3 points to this storage area.

After pushing the 5-word return block, the **SAVE** instruction checks for stack overflow. If the contents of the stack pointer exceed the contents of the stack limit, a stack overflow occurs.

Use the **SAVE** instruction after a **JSR** instruction. **JSR** transfers control to a subroutine and stores the return address in AC3. **SAVE** pushes AC3 on the stack, then puts the frame pointer in AC3.

Subtract Immediate

SBI *n,ac*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	N		AC		0	0	0	0	1	0	0	1	0	0	0

Subtracts an unsigned integer in the range of one to four from the contents of an accumulator.

Subtracts the contents of the immediate field plus 1 from the unsigned, 16-bit number in the specified accumulator. This is done by adding one to the contents of the immediate field, forming its two's complement, and adding the result to the contents of the accumulator. Places the answer in the specified accumulator. The carry bit remains unchanged.

> **NOTE:** *DGC assemblers compute N before loading the immediate field. You code* n, *which is N+1. Code the exact value you want to subtract.*

Skip If ACS Greater Than Or Equal To ACD

SGE *acs,acd*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	ACS		ACD		0	1	0	0	1	0	0	1	0	0	0

Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

Algebraically compares the two's complement number in ACS to the two's complement number in ACD. Skips the next sequential word if the number in ACS is greater than or equal to the number in ACD. The contents of ACS and ACD remain unchanged.

> **NOTE:** *The* **SGT** *and* **SGE** *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use either the* **SUB** *or* **ADC** *instruction.*

Skip If ACS Greater Than ACD

SGT *acs,acd*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	ACS		ACD		0	1	0	0	0	0	0	1	0	0	0

Compares two signed integers in two accumulators and skips if the first is greater than the second.

Algebraically compares the two's complement number in ACS to the two's complement number in ACD. Skips the next sequential word if the number in ACS is greater than the number in ACD. The contents of ACS and ACD remain unchanged.

> **NOTE:** *SGT treats the contents of the specified accumulators as two's complement integers. To compare unsigned integers, use either* **SUB** *or* **ADC** *and set the [*skip*] field appropriately.*

CPU Skip

SKP*[t]* *ac,***CPU**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	1	1	1	T		1	1	1	1	1	1

Skips the next sequential word if the test condition specified by *t* is true. The following table shows the possible skip test conditions.

Mnemonic	Value	Test
BN	00	Skip if interrupts are enabled.
BZ	01	Skip if interrupts are disabled.
DN	10	Skip if power flag is one.
DZ	11	Skip if power flag is zero.

Using the *CPU Skip* instruction for testing of the Power Fail flag allows the power-fail option to provide a "fail-soft" capability in the event of an unexpected power loss.

I/O Skip

SKP*[t] device*

0	1	1	0	0	1	1	1	T	DEVICE CODE
0	1	2	3	4	5	6	7	8 9	10 15

If the test condition, *t*, is true for the device specified by the device code, the instruction skips the next sequential word. The possible values of *t* are listed in the table.

Mnemonic	Value	Test
BN	00	Test for Busy = 1
BZ	01	Test for Busy = 0
DN	10	Test for Done = 1
DZ	11	Test for Done = 0

Instruction: **SKPBZ** TTO
Checks the setting of the console interface Busy flag. If the Busy flag is set to 0 (Teletype is not busy), the next sequential word is skipped.

Skip On Nonzero Bit

SNB *acs,acd*

1	ACS	ACD	1	0	1	1	1	1	1	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the addressed bit is set to one.

Forms a bit pointer from ACS and ACD. ACS contains the effective address. Bits 0-11 of ACD are the word offset, and bits 12-15 specify the bit. ACS and ACD remain unchanged.

If you specify ACS and ACD as the same accumulator, the effective address is zero and the accumulator contains the word offset and specifies the bit.

Store Accumulator

STA *ac,[@]displacement[,index]*

0	1	0	AC	@	INDEX	DISPLACEMENT
0	1	2	3 4	5	6 7	8 15

Stores the contents of an accumulator into a memory location.

Places the contents of the specified accumulator in the word addressed by the effective address. Overwrites the previous contents of the addressed location. The contents of the specified accumulator remain unchanged.

Store Byte

STB *acs,acd*

1	ACS	ACD	1	1	0	0	0	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Stores the right byte (bits 8-15) of ACD in an addressed memory byte. ACS contains the byte pointer. ACS and ACD remain unchanged.

Subtract

SUB*[c][sh][#]   acs,acd[,skip]*

1	ACS	ACD	1	0	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Performs unsigned interger subtraction.

Sets carry to the specified value. Subtracts the unsigned, 16-bit number in ACS from the unsigned, 16-bit number in ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. If the operation produces a carry of one out of the most-significant bit (bit 0), the instruction complements carry. Places the 17-bit value (carry and the result of the subtraction) in the shifter. Performs the specified shift operation.

Tests the skip condition. If the no-load bit is zero, places the 17-bit value in the carry bit and ACD. If the skip condition is true, skips the next sequential word.

> **NOTE:** *If, before execution, the 16-bit, unsigned number in ACS is less than or equal to the 16-bit, unsigned number in ACD, the instruction complements carry.*

System Call

SYC *acs,acd*

1	ACS	ACD	1	1	1	0	1	0	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Pushes a return block and transfers control to the system call handler.

Disables the MAP, pushes the standard return block on the stack, and jumps indirect through location 2. ACS and ACD remain unchanged.

The program counter in the return block contains the address of the instruction immediately following **SYC**.

I/O interrupts cannot occur between the time the **SYC** instruction is executed and the time the next instruction is executed.

> **NOTE:** *If both accumulators are specified as AC0, the instruction does not push a return block onto the stack. AC0 remains unchanged.*

Data General assemblers recognize the mnemonic **SCL** as equivalent to **SYC1,1**, and **SVC** as equivalent to **SYC 0,0.**

Skip On Zero Bit

SZB *acs,acd*

1	ACS		ACD		1	0	0	1	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

If the addressed bit is zero, the next sequential word is skipped.

Forms a bit pointer from ACS and ACD. ACS contains the effective address. Bits 0-11 of ACD are the word offset, and bits 12-15 specify the bit. ACS and ACD remain unchanged.

If you specify ACS and ACD as the same accumulator, the effective address is zero. The accumulator contains the word offset and specifies the bit.

Skip On Zero Bit And Set To One

SZBO *acs,acd*

1	ACS		ACD		1	0	0	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

If the addressed bit is zero, sets the bit to one and skips the next sequential word.

Forms a bit pointer from ACS and ACD. ACS contains the effective address. Bits 0 to 11 of ACD are the word offset, and bits 12 to 15 specify the bit. ACS and ACD remain unchanged.

If you specify ACS and ACD as the same accumulator, the effective address is zero. The accumulator contains the word offset and specifies the bit. The CPU controls the memory bus until the instruction is completed.

> **NOTE:** *You can use this instruction to implement bit maps. You can use bit maps to allocate facilities (such as memory blocks and I/O devices) to several processes or tasks that interrupt one another or that run in a multiprocessor environment.*

Vector On Interrupting Device Code

VCT *,[@]displacement[,index]*

0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

S	DISPLACEMENT
0	1 … 15

Returns the device code of the interrupting device and uses that code as an index into a table. The value found in the table is used in one of two ways: it can be a pointer to the appropriate interrupt handler (mode A), or as a pointer to another table (modes B through E). This second table points to the interrupt handler and contains a new priority mask. Depending on the mode used, the instruction can also save the state of the machine by pushing certain information onto the stack, creating a new vector stack, setting up a priority structure, and enabling interrupts.

The flowchart in Figure 10.9 is a complete diagram of the operation of the **VCT** instruction. Note that all modes use the *vector table* to find the next address used. Mode A uses the vector table entry as the address of the interrupt handler and passes control to it immediately. Modes B through E all use the vector table address as a pointer into a *device control table (DCT)*, where the address of the interrupt handler is found, along with a new priority mask.

Three control bits determine the mode of the **VCT** instruction which will be used. (See Figure 10.10.) Their names and locations are:

Stack Change Bit (S) — Bit 0 of the second word of the **VCT** instruction;

Direct Bit (D) — Bit 0 of the selected vector table entry;

Push Bit (P) — Bit 0 of the first word of the selected device control table.

if no device interrupting, $\overline{\text{DATA0-15}}$ lines return device code 177777, or 077 when masked down if MicroEclipse bus used

The value of these bits collectively determine the mode of the **VCT** instruction. The bits determine the mode as follows:

Direct	Stack	Push	Mode
0	—	—	A
1	0	0	B
1	0	1	C
1	1	0	D
1	1	1	E

Common Process

All modes perform the initial steps of the **VCT** instruction. These steps begin when the instruction returns the interrupting device code. The instruction adds the device code to the address of the start of the vector table (bits 1–15 of the second instruction word). The result is the address of an entry within the vector table. The instruction fetches the contents of this vector table entry and examines bit 0 of the entry (the direct bit). If the direct bit is zero, mode A is selected; otherwise, one of the other modes (B-E) is selected.

Mode A

The instruction performs the functions of mode A if the direct bit is zero. The values of the other control bits do not matter. In mode A, the instruction uses bits 1–15 of the fetched vector table entry as the address of the interrupt handler of the interupting device. Control transfers immediately to the interrupt handler with all interrupts disabled.

Modes B Through E — Part I

The direct bit has the value one for all of these modes. The values of the push bit and the stack change bit determine which of the four modes will take place. The action of these modes can be divided into two parts: a first part, whose action varies from mode to mode; and a second part, whose action is identical for every mode. We discuss each first part separately, then the common second part.

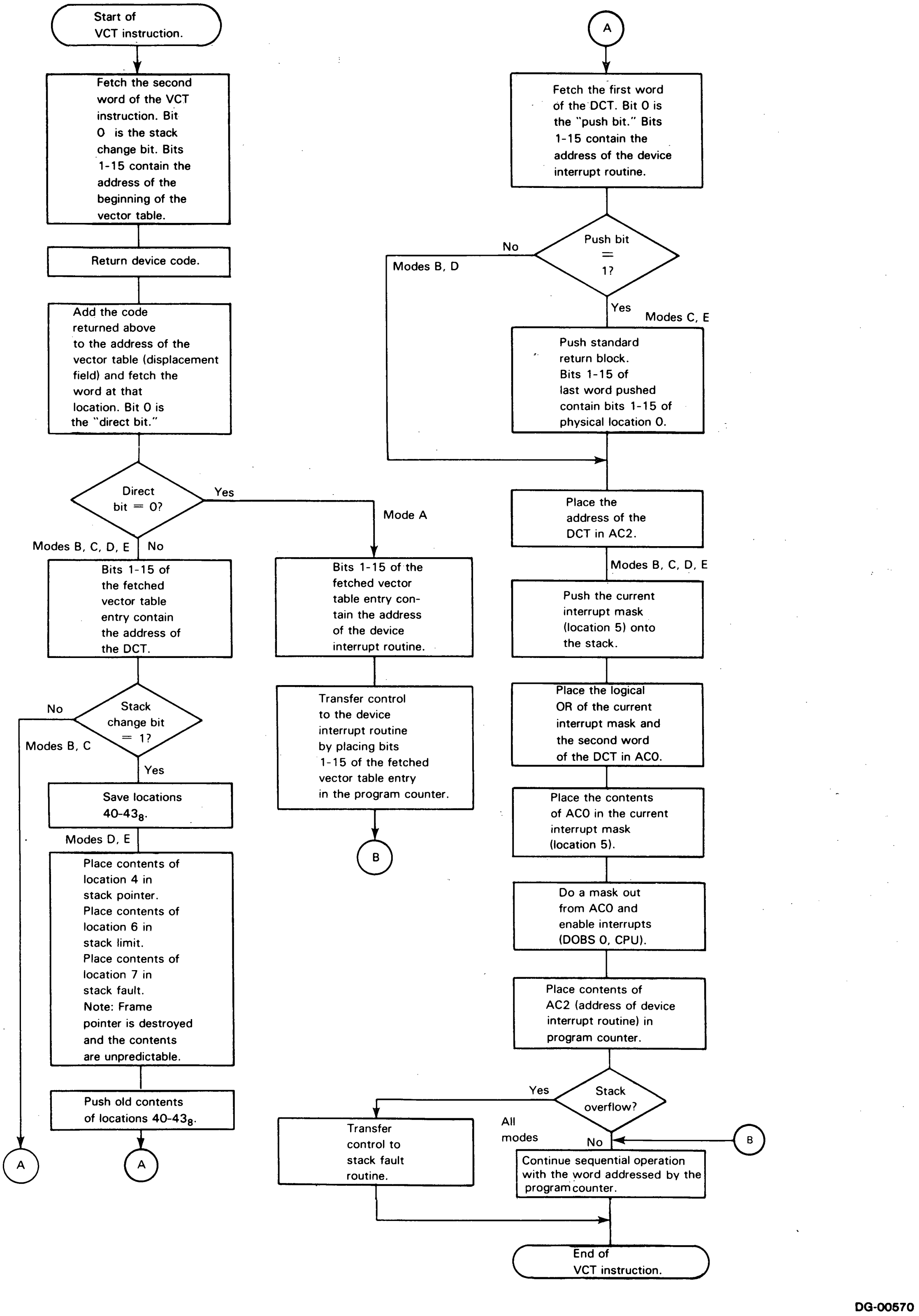
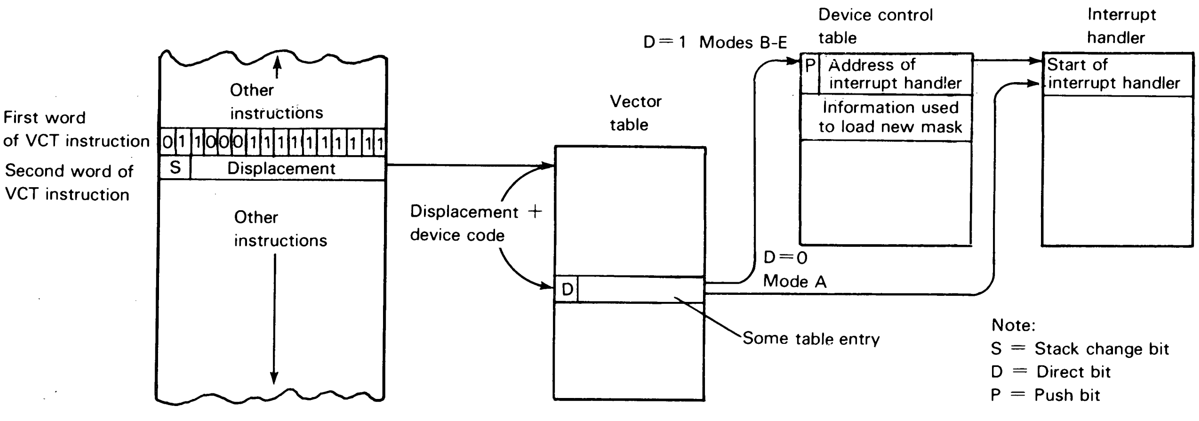

Start of VCT instruction.
Fetch the second word of the VCT instruction. Bit 0 is the stack change bit. Bits 1-15 contain the address of the beginning of the vector table.
Return device code.
Add the code returned above to the address of the vector table (displacement field) and fetch the word at that location. Bit 0 is the "direct bit."
Direct bit = 0?
Yes
Mode A
Modes B, C, D, E
No
Bits 1-15 of the fetched vector table entry contain the address of the DCT.
Bits 1-15 of the fetched vector table entry contain the address of the device interrupt routine.
Stack change bit = 1?
No
Modes B, C
Yes
Save locations 40-43$_8$.
Modes D, E
Place contents of location 4 in stack pointer. Place contents of location 6 in stack limit. Place contents of location 7 in stack fault. Note: Frame pointer is destroyed and the contents are unpredictable.
Push old contents of locations 40-43$_8$.
A
A
Transfer control to the device interrupt routine by placing bits 1-15 of the fetched vector table entry in the program counter.
B
A
Fetch the first word of the DCT. Bit 0 is the "push bit." Bits 1-15 contain the address of the device interrupt routine.
Push bit = 1?
No
Modes B, D
Yes
Modes C, E
Push standard return block. Bits 1-15 of last word pushed contain bits 1-15 of physical location 0.
Place the address of the DCT in AC2.
Modes B, C, D, E
Push the current interrupt mask (location 5) onto the stack.
Place the logical OR of the current interrupt mask and the second word of the DCT in AC0.
Place the contents of AC0 in the current interrupt mask (location 5).
Do a mask out from AC0 and enable interrupts (DOBS 0, CPU).
Place contents of AC2 (address of device interrupt routine) in program counter.
Stack overflow?
Yes
Transfer control to stack fault routine.
All modes
No
B
Continue sequential operation with the word addressed by the program counter.
End of VCT instruction.
DG-00570

Figure 10.7 Vector instruction flowchart

Mode B takes place when the stack change bit and the push bit both have the value of zero. The instruction uses the vector table entry as the address of the device control table *(DCT)* for the interrupting device. Bits 1-15 of the first word of the DCT contain the address of The desired interrupt handler (bit 0 is the push bit). The second word of the DCT contains information used to construct the new interrupt priority mask. Succeeding words (if any) contain information to be used by the device interrupt handler.



DG-05741

Figure 10.8 Overview of the vector instruction

Mode C takes place when the stack change bit has the value zero and the push bit has the value one. This mode performs the functions of mode B; in addition, mode C pushes a standard 5-word return block onto the standard stack. The return block contains the contents of the four accumulators, the value of carry, and the contents of physical location zero (the program counter return value).

Mode D takes place when the stack change bit has the value one and the push bit has the value zero. This mode performs the functions of mode B; in addition, mode D sets up a new stack for the interrupt handler (using the contents of locations 4, 6, and 7) and pushes the old contents of physical locations 40–43$_8$ (the user stack control words) onto the new stack.

Mode E takes place when the stack change bit and the push bit both have the value one. This mode combines the functions of modes C and D. That is, mode E performs the functions of mode B, sets up a new stack, and pushes a 5-word return block and the old stack control words onto the new stack.

Modes B through E — Part II

Modes B through E use the same procedure for the remainder of the **VCT** instruction. During this procedure, the instruction pushes the current priority mask (location 5) onto the stack. Next, the instruction updates location 5 and performs a **MSKO** instruction, using the logical OR of the current mask and the second word of the DCT. The instruction then sets the Interrupt On flag to one and passes control to the selected device interrupt handler. Note that the CPU permits one more instruction to execute (in this case, the first instruction of the interrupt handler) before the next I/O interrupt can occur. Returns from VCT routines may be accomplished with:

Mode	Return Instruction
A	JMP@0
B	JMP@0
C	POPB
D	(Restore saved stack parameters) JMP@0
E	RSTR

Exchange Accumulators

XCH *acs,acd*

1	ACS		ACD		0	0	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Exchanges the contents of two accumulators.

Places the original contents of ACS in ACD and the original contents of ACD in ACS.

Execute

XCT *ac*

1	0	1	AC		1	1	0	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Executes the instruction contained in AC as if it were in main memory in the location occupied by the **XCT** instruction.

If the instruction in AC is an **XCT** instruction which executes the instruction in AC, the processor is placed in a 1-instruction loop. Because of this possibility, an I/O interrupt can occur just before the processor executes the instruction AC. If an I/O interrupt occurs at this time, the program counter in the return block pushed on the stack contains the address of the **XCT** instruction. This capability gives you an effective *Wait for I/O Interrupt* instruction.

> **NOTE:** *If the specified accumulator contains the first word of a 2-word instruction, the word following the* **XCT** *instruction is used as the second word. Normal sequential operation then continues from the second word after the* **XCT** *instruction. Do not use the* **XCT** *instruction to execute an instruction that requires all four accumulators, such as* **CMV, CMT, CMP, CTR,** *or* **BAM**.

Extended Operation

XOP *acs,acd,operation #*

1	ACS	ACD	OPERATION #	0	1	1	0	0	0
0	1 2	3 4	5 9	10	11	12	13	14	15

Pushes a return block onto the stack. Places ACS's stack address in AC2; places ACD's stack address in AC3. Memory location 44_8 must contain the **XOP** origin address, the starting address of a 32_{10} word table of addresses. These addresses are the starting location of the various **XOP** operations.

Adds the operation number in the **XOP** instruction to the **XOP** origin address to produce the address of a word in the **XOP** table. The instruction fetches that word and treats it as the intermediate address in the effective address calculation. If an indirection chain is followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the **XOP** origin address remain unchanged.

The format of the return block pushed by the **XOP** instruction is shown in Figure 10.11.

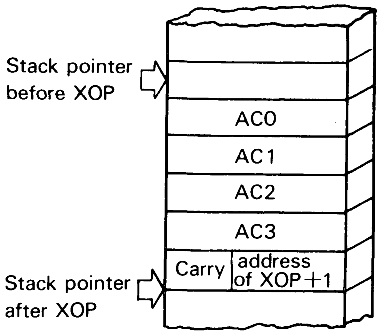


Figure 10.9 Format of return block pushed by XOP

This return block is configured so that the **XOP** procedure can return control to the calling program through the **POPB** instruction.

Alternate Extended Operation

XOP1 *acs,acd,operation #*

1	ACS	ACD	0	OPERATION #	1	1	1	0	0	0
0	1 2	3 4	5	6 9	10	11	12	13	14	15

This instruction operates like the **XOP** instruction except that it adds 32_{10} to the entry number before it adds the entry number to the **XOP** origin address. In addition, it can specify only 16 entry locations.

Exclusive OR

XOR *acs,acd*

1	ACS		ACD		0	0	1	0	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Forms the logical exclusive OR of the contents of ACS and the contents of ACD. Places the result in ACD.

Sets a bit to one if the corresponding bit positions in the two operands are unlike. Otherwise, sets the bit to zero. The contents of ACS remain unchanged.

Example

Operation	Before	Action

XOR 0,1
Exclusive ORs the contents of ACO with the contents of AC1.
	ACO=003156_8
	AC1=020701_8
		ACO=023657_8
		AC1=020701_8

Exclusive OR Immediate

XORI *i,ac*

1	0	1	AC		1	1	1	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

IMMEDIATE	
16	31

Forms the logical exclusive OR of the contents of the immediate field and the contents of the specified AC. Places the result in the specified AC.

Appendix A

I/O Device Codes

Octal Device Code	Mnem	Priority Mask Bit	Device Name
00	—	—	Unused
01	WCS	—	Writeable control store option [or
	[APL]	—	APL register]
02	ERCC	—	Error checking and correction
	MAP		Memory allocation and protection
03			
04			
05	BMC	—	Burst multiplexor channel
06	MCAT	12	Multiprocessor adapter transmitter
			Multiprocessor adapter receiver
07	MCAR	12	TTY input
			TTY output
10	TTI	14	
11	TTO	15	
12	PTR	11	Paper tape reader
13	PTP	13	Paper tape punch
14	RTC	13	Real-time clock
15	PLT	12	Incremental plotter
16	CDR	10	Card reader
17	LPT	12	Line printer
20	DSK	9	Fixed-head disc
21	ADCV	8	A-D converter
22	MTA	10	Magnetic tape
23	DACV	None	D-A converter
24	DCM	0	Data communications multiplexor
25			Fixed-head DG/Disc
26	DKB	9	DG/Disk storage subsystem
27	DPF	7	Asynchronous hardware multiplexor
30	QTY	14	
30	SLA	14	Synchronous line adapter
31[1]	IBM1	13	IBM 360/370 interface
32	IBM2	13	IBM 360/370 interface
33	DKP	7	Moving head disk
34	CAS[1]	10	Cassette tape
	DCU[4]	4	Data control unit
34	MX1	11	Multiline asynchronous controller
			Multiline asynchronous controller
35	MX2	11	Interprocessor bus—half-duplex
			IPB watchdog timer
36	IPB	6	IPB full-duplex input
			Synchronous communication receiv-
37	IVT	6	er
40[2]	DPI	8	
40	SCR	8	

Octal Device Code	Mnem	Priority Mask Bit	Device Name
41[3]	DPO	8	IPB full-duplex output
41	SCT	8	Synchronous communication trans-
			mitter
42	DIO	7	Digital I/O
43	DIOT	6	Digital I/O timer
43	PIT	11	Programmable interval timer
44	MXM	12	Modem control for MX1/MX2
45			Second multiprocessor transmitter
46	MCAT1	12	Second multiprocessor receiver
			Second TTY input
47	MCAR1	12	
50	TTI1	14	
51	TTO1	15	Second TTY output
52	PTR1	11	Second paper tape reader
53	PTP1	13	Second paper tape punch
54	RTC1	13	Second real-time clock
55	PLT1	12	Second incremental plotter
56	CDR1	10	Second card reader
57	LPT1	12	Second line printer
60	DSK1	9	Second fixed-head disk
61	ADCV1	8	Second A-D converter
62	MTA1	10	Second magnetic tape
63	DACV1	None	Second D-A converter
64			
65	IOP1	5[5]	Host to IOP interface
66	DKB1	9	Second fixed-head DG/Disk
67	DPF1	7	Second DG/Disk storage subsystem
70	QTY1	14	Second asynchronous hardware
			multiplexor
70	SLA1	14	Second synchronous line adapter
			Second IBM 360/370 interface
71[1]			Second IBM 360/370 interface
		13	Second moving head disk
72			
	DKP1	7	
73			
74	CAS1	10	Second cassette tape
74[1]		11	Second multiline asynchronous con-
75		11	troller
76	DPU	4	Second multiline asynchronous con-
77	CPU	—	troller
			DCU to host interface
			Central processor and console func-
			tions

[1] *Code returned by INTA and used by VCT.*

[2] *Can be set up with any unused even device code equal to 40 or above.*

[3] *Can be set up with any unused odd device code equal to 41 or above.*

[4] *Can be set to any unused device code between 1 and 76.*

[5] *Micro interrupts are not maskable.*

Appendix B
Programming Aids

Octal and Hexadecimal Conversion

To convert a number from octal or hexadecimal to decimal, locate in each column of the appropriate table the decimal equivalent for the octal or hex digit in that position. Add the decimal equivalents to obtain the decimal number.

To convert a decimal number to octal or hexadecimal:

1. Locate the largest decimal value in the appropriate table that will fit into the decimal number to be converted;
2. Note its octal or hex equivalent and column position;
3. Find the decimal remainder.

Repeat the process on each remainder. When the remainder is 0, all digits will have been generated. Table B.1 and B.2 provide octal and hexadecimal conversion aids.

	8^8	8^4	8^3	8^2	8^1	8^0
0	0	0	0	0	0	0
1	32,768	4,096	512	64	8	1
2	65,536	8,192	1,024	128	16	2
3	98,304	12,228	1,536	192	24	3
4	131,072	16,384	2,048	256	32	4
5	163,840	20,480	2,560	320	40	5
6	196,608	24,576	3,072	384	48	6
7	229,376	28,672	3,584	448	56	7

Table B.1 Octal conversion table

	16^5	16^4	16^3	16^2	16^1	16^0
0	0	0	0	0	0	0
1	1,048,576	65,536	4,096	256	16	1
2	2,097,152	131,072	8,192	512	32	2
3	3,145,728	196,608	12,288	768	48	3
4	4,194,304	262,144	16,384	1,024	64	4
5	5,242,880	327,680	20,480	1,280	80	5
6	6,291,456	393,216	24,576	1,536	96	6
7	7,340,032	458,752	28,672	1,792	112	7
8	8,388,608	524,288	32,768	2,048	128	8
9	9,437,184	589,824	36,864	2,304	144	9
A	10,485,760	655,360	40,960	2,560	160	10
B	11,534,336	720,896	45,056	2,816	176	11
C	12,582,912	786,432	49,152	3,072	192	12
D	13,631,488	851,968	53,248	3,328	208	13
E	14,680,064	917,504	57,344	3,584	224	14
F	15,728,640	983,040	61,440	3,840	240	15

Table B.2 Hexadecimal conversion table

ASCII Character Codes

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT.TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑—	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ OR ∧
95	137	5F	← OR —
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

DG-05495

Figure B.1 ASCII Character Codes

Appendix C

Instruction Execution Times

The following tables list typical execution times for all instructions. All times are in microseconds. Maximum I/O interrupt latency is 110 microseconds.

Instruction	Execution Time (microsec.)	CPU Cycles	Notes
ADC	0.50	1	1
ADD	0.50	1	1
ADDI	1.00	2	
ADI	0.50	1	
ANC	0.50	1	
AND	0.50	1	1
ANDI	1.00	2	
BAM	6.50 + 2.5/word	13 + 5/word	2,3
BLM	3.50 + 2.0/word	7 + 4/word	2,3
BTO	5.50	11	4
BTZ	4.50	9	4
CLM	3.50	7	1
CMP	8.50 + 7.0/byte	17 + 14/byte	3
CMT	1.50 + 9.0/byte	3 + 18/byte	3
CMV	5.50 + 5.5/byte	11 + 11/byte	3
COB	7.50	15	4
COM	0.50	1	1
CTR	5.50 + 7.0 or 9.5/byte	11 + 14 or 19/byte	5
DAD	8.00	16	
DHXL	7.50	15	5
DHXR	7.00	14	5
DIA,B,C	3.50	7	
DIS	3.50	7	
DIV	12.00	24	
DIVS	20.50	41	
DIVX	19.50	39	
DLSH	6.50	13	4
DOA,B,C	3.50	7	
DSB	8.00	16	
DSPA	6.00	12	2
DSZ	2.00	4	2,1
EDSZ	2.00	4	2,1
EISZ	2.00	4	2,1
EJMP	1.50	3	2
EJSR	1.50	3	2
ELDA	1.00	2	2
ELDB	3.50	7	
ELEF	1.00	2	2
ESTA	1.00	2	2
ESTB	3.50	7	
FAB	11.00	22	7
FAD	87.00	174	7
FAMD	92.00	184	7,8
FAMS	67.00	134	7,8
FAS	65.00	130	7
FCLE	3.00	6	
FCMP	29.00	58	
FDD	900.00	1800	7,9
FDMD	900.00	1800	7,8,9
FDMS	190.00	380	7,8,9
FDS	190.00	380	7,9
FEXP	13.00	26	7
FFAS	46.00	92	7
FFMD	45.50	90	7, 8
FHLV	30.00	60	7, 9
FINT	20.50	41	7
FLAS	31.00	62	

Table C.1

Notes

[1] *If skip occurs, add 0.50.*

[2] *If indirect chain followed, add 0.50 + (number of indirects−1)*1.00.*

[3] *For each item moved, add the amount shown.*

[4] *Execution time is operand-dependent.*

[5] *If ACS<>ACD, add 1.00 + 2(Number of indirects).*

[6] *Byte moves require 7.0 µs/byte; compares require 9.5 µs/byte.*

[7] *This instruction can take a floating point trap, which would add 19µs to the execution time.*

[8] *This instruction does an effective address calculation, which can add 15µs to the execution time.*

[9] *Floating-point divide execution times depend on the number of zero and one bits in the quotient (the more one's contained in the quotient, the longer the execution time).*

[10] *For each word moved, add 1.50.*

[11] *For stack overflow, add 9.00 + (Note 2).*

[12] *For each accumulator pushed/popped, add 0.50.*

[13] *Vector execution times depend on the mode employed.*

[14] *Add instruction execution time.*

Instruction	Execution Time (microsec.)	CPU Cycles	Notes
FLDD	16.50	32	8
FLDS	15.00	30	8
FLMD	31.00	62	8
FLST	11.50	23	7, 8
FMD	266.00	532	7
FMMD	266.00	532	7, 8
FMMS	80.50	161	7, 8
FMOV	17.00	34	7
FMS	80.50	161	7
FNEG	12.50	25	7
FNOM	43.00	86	
FNS	1.00	2	
FPOP	32.00	64	
FPSH	31.50	63	
FRH	6.00	12	
FSA	1.50	3	
FSCAL	48.00	96	7
FSD	87.00	174	7
FSEQ	5.00	10	
FSGE	4.50	9	
FSGT	5.00	10	
FSLE	5.00	10	
FSLT	4.50	9	
FSMD	92.00	184	7, 8
FSMS	67.00	134	7, 8
FSND	5.00	10	
FSNE	5.00	10	
FSNER	5.00	10	
FSNM	5.00	10	
FSNO	5.00	10	
FSNOD	5.00	10	
FSNU	5.00	10	
FSNUD	5.00	10	
FSNUO	5.00	10	
FSS	65.00	130	7
FSST	7.50	15	8
FSTD	12.50	25	8
FSTS	9.50	19	8
FTD	3.00	6	
FTE	5.00	10	7
HALT	6.50	13	
HLV	1.50	3	
HXL	2.00	4	5
HXR	2.00	4	5
INC	0.50	1	1
INTA	3.50	7	
IOR	1.50	3	
IORI	1.50	3	
IORST	9.00	18	
ISZ	2.00	4	2, 1
JMP	1.50	3	2
JSR	1.50	3	2
LDA	1.00	2	2
LDB	1.50	3	
LEF	1.00	2	2
LMP	4.50	9	2,10
LOB	4.00	8	5
LRB	5.00	10	5
LSH	8.50	17	5
MOV	0.50	1	1
MSKO	3.50	7	
MSP	3.00	6	11
MUL	9.50	19	
MULS	9.50	19	

Table C.1

Notes

[1] *If skip occurs, add 0.50.*

[2] *If indirect chain followed, add 0.50 + (number of indirects−1)*1.00.*

[3] *For each item moved, add the amount shown.*

[4] *Execution time is operand-dependent.*

[5] *If ACS<>ACD, add 1.00 + 2(Number of indirects).*

[6] *Byte moves require 7.0 µs/byte; compares require 9.5 µs/byte.*

[7] *This instruction can take a floating point trap, which would add 19µs to the execution time.*

[8] *This instruction does an effective address calculation, which can add 15µs to the execution time.*

[9] *Floating-point divide execution times depend on the number of zero and one bits in the quotient (the more one's contained in the quotient, the longer the execution time).*

[10] *For each word moved, add 1.50.*

[11] *For stack overflow, add 9.00 + (Note 2).*

[12] *For each accumulator pushed/popped, add 0.50.*

[13] *Vector execution times depend on the mode employed.*

[14] *Add instruction execution time.*

Instruction	Execution Time (microsec.)	CPU cycles	Notes
NEG	0.50	1	1
NIO	3.50	7	
POP	1.50	3	12
POPB	6.50	13	
POPJ	3.00	6	
PSH	3.00	6	11, 12
PSHJ	4.50	9	11
PSHR	4.50	9	
RSTR	10.50	21	
RTN	6.00	12	
SAVE	8.00	16	11
SBI	0.50	1	
SGE	0.50	1	1
SGT	0.50	1	1
SKP	1.50	3	1
SNB	4.50	9	1,4
STA	1.00	2	2
STB	1.50	3	
SUB	0.50	1	1
SYC	10.50	21	2
SZB	4.00	8	1,4
SZBO	6.00	12	1,4
VCT	8.00 to 34.00	16 to 68	13
XCH	1.50	3	
XCT	2.00	4	14
XOP	20.50	41	
XOP1	21.50	43	
XOR	2.50	5	
XORI	2.50	5	

Table C.1

Notes

[1] *If skip occurs, add 0.50.*

[2] *If indirect chain followed, add 0.50 + (number of indirects−1)*1.00.*

[3] *For each item moved, add the amount shown.*

[4] *Execution time is operand-dependent.*

[5] *If ACS<>ACD, add 1.00 + 2(Number of indirects).*

[6] *Byte moves require 7.0 µs/byte; compares require 9.5 µs/byte.*

[7] *This instruction can take a floating point trap, which would add 19µs to the execution time.*

[8] *This instruction does an effective address calculation, which can add 15µs to the execution time.*

[9] *Floating-point divide execution times depend on the number of zero and one bits in the quotient (the more one's contained in the quotient, the longer the execution time).*

[10] *For each word moved, add 1.50.*

[11] *For stack overflow, add 9.00 + (Note 2).*

[12] *For each accumulator pushed/popped, add 0.50.*

[13] *Vector execution times depend on the mode employed.*

[14] *Add instruction execution time.*

Appendix D

Programming Examples

Arithmetic Tests

You can use the *Subtract* instruction to clear an accumulator by subtracting it from itself:

```
SUB     2,2     ;Clears AC2 and complements carry
SUBO    2,2     ;Clears both AC2 and carry
```

Subtract is also useful for comparing quantities:

```
SUB#    2,3 SNR ;Skips if AC2 and AC3 are unequal, but ;does
                 not affect either accumulator
```

You can subtract one from an accumulator, without using a constant from memory:

```
NEG     AC,AC
COM     AC,AC
```

The Move instruction is particularly useful for checking accumulator contents.

The following examples show how this may be done.

Test an accumulator for zero.

```
MOV     AC1,       ;Tests AC1 for zero
        AC1,SZR
JMP     ...        ;Not zero
...     ...        ;Zero
```

Check if two accumulators are both zero.

```
MOV     ACS,ACS,SNR
SUB     ACS,ACD,SZR
JMP     ...             ;Not equal
...     ...             ;Equal
```

Test an accumulator for −1.

```
COM#    AC,AC,SZR
JMP     ...             ;Not−1
...     ...             ;−1
```

Test an accumulator for two or greater.

```
MOVZR#  AC,AC,SNR
JMP     ...             ;Less than 2
...     ...             ;2 or greater
```

Assume that it is known that an accumulator contains 0, 1, 2, or 3; find out which value.

```
MOVZR#  AC,AC,SEZ
JMP     THREE           ;Was 3
MOV     AC,AC,SNR
JMP     ZERO            ;Was 0
MOVZR#  AC,AC,SZR
JMP     TWO             ;Was 2
...     ...             ;Was 1
```

Check if both bytes in an accumulator are equal.

```
MOVS    ACS,ACD
SUB     ACS,ACD,SZR
JMP     ...             ;Not equal
...     ...             ;Equal
```

Compare the signed, two's complement integer contained in ACS to 0.

```
MOV#    ACS,ACS,SZR     ;Skip if contents of ACS = 0
MOV#    ACS,ACS,SNR     ;Skip if contents of ACS ≠ 0
ADDO#   ACS,ACS,SBN     ;Skip if contents of ACS > 0
MOVL#   ACS,ACS,SZC     ;Skip if contents of ACS ≥ 0
MOVL#   ACS,ACS,SNC     ;Skip if contents of ACS < 0
ADDO#   ACS,ACS,SEZ     ;Skip if contents of ACS ≤ 0
```

Test AC1 for the unsigned integer 16 −1 (177777, signed −1).

```
COM#    1,1,SZR  ;Skip the next instruction if AC1 ;contains all
                 ones.
```

As the result is not loaded in the above example, you can specify any accumulator as the destination, for example,

```
COM#    1,3,SZR
```

Assume that AC0 contains a signed, 16-bit, two's complement integer. The following three instructions will place an indicator of the sign of the number in AC0. If the number is greater than zero, AC0 is set to +1. If the number is less than zero, AC0 is set to −1. If the number is equal to zero, AC0 remains 0. The previous contents of the carry bit are overwritten.

```
ADDO    AC0,AC0,SBN    ;Skip if > 0
ADCC    AC0,AC0,SNC    ;AC0 gets -1
SUBCL   AC0,AC0        ;Copy carry into bit 15
```

This example checks an ASCII character to make sure that it is a decimal digit. The character is in AC0 and is unchanged by the test. The contents of accumulators AC1 and AC2 are overwritten.

```
LDA     AC1,C60        ;ASCII zero
LDA     AC2,C71        ;ASCII nine
ADCZ#   AC2,AC0,SNC    ;Skips if (AC0>9)
ADCZ#   AC0,AC1,SZC    ;Skips if (AC0>0)
JMP     ...            ;Not digit =
...     ...            ;Digit
C60:    60             ;ASCII 0
C71:    71             ;ASCII 9
```

Multiply an AC by the indicated value.

```
MOV       ACx,ACx      ;Multiply by 1
MOVZL     ACx,ACx      ;Multiply by 2
MOVZL     ACx,ACy      ;Multiply by 3
ADD       ACy,ACx
ADDZL     ACx,ACx      ;Multiply by 4
MOV       ACx,ACy      ;Multiply by 5
ADDZL     ACx,ACx
ADD       ACy,ACx
MOVZL     ACx,ACy      ;Multiply by 6
ADDZL     ACy,ACx
ADDZL     ACx,ACx      ;Multiply by 8
MOVZL     ACx,ACx
```

Multiplication by other factors of two can be achieved with the *Logical Shift* instruction; multiplication by factors of 16 can be accomplished with the *Hex Shift Left* instruction.

You may want to negate the double-length number whose high-order word is in AC0 and low-order word in AC1. You negate the low-order part, but simply complement the high-order part, unless the low-order part is zero.

```
NEG     1,1,SNR     —
NEG     0,0,SKP     ;Low-order zero
COM     0,0         ;Low-order nonzero
```

Note that the magnitude parts of the sequence of negative numbers from the most negative toward zero are the positive numbers zero and upward. So in multiple-precision arithmetic, low-order words can be treated simply as positive numbers.

In unsigned addition, a carry indicates that the low-order result is too large, and the high-order part must increased. The number in AC2 and AC3 is added to the number in AC0 and AC1.

```
ADDZ    3,1,SZC
INC     2,2
ADD     2,0
```

In two's complement subtraction, a carry should occur unless the subtrahend is too large. We could increment as in addition, but since incrementing in the high-order bit is precisely the difference between a one's complement and a two's complement, simply subtract the number in AC2 and AC3 from that in AC0 and AC1.

```
SUBZ    3,1,SZC
SUB     2,0,SKP
ADC     2,0
```

Together, *Add Complement* and *Subtract* allow a program to compare the magnitudes of unsigned integers.

```
SUB#     ACS,ACD,SZR    ;Skip if (ACS) = (ACD)
SUB#     ACS,ACD,SNR    ;Skip if (ACS) ≠ (ACD)
ADCZ#    ACS,ACD,SNC    ;Skip if (ACS) < (ACD)
SUBZ#    ACS,ACD,SNC    ;Skip if (ACS) ≤ (ACD)
SUBZ#    ACS,ACD,SZC    ;Skip if (ACS) > (ACD)
ADCZ#    ACS,ACD,SZC    ;Skip if (ACS) ≥ (ACD)
```

Subroutines

The transfer of control between routines is made easier and more orderly by using the stack facility. There are three general ways to effect calls and returns, but more complex ways may be derived. The three basic methods of call and return are discussed here.

The first method transfers control to the subroutine via the **JSR** instruction. The subroutine executes the **SAVE** instruction at the subroutine entry point and returns control through the **RTN** instruction.

```
CALL:    JSR     SUBR     ;In calling program
         . . .
         . . .
         . . .
SUBR:    SAVE i           ;Subroutine entry
         . . .
         . . .
         . . .
RETRN:   RTN              ;Return to
                          ;calling program
```

This method has the following characteristics:

1. AC3 of the calling program is overwritten by **JSR**.
2. The call is only one word.
3. Upon return to the calling program, AC3 contains the calling program's frame pointer.
4. A **SAVE** instruction is required at each entry point.
5. Arguments are easily passed on the stack, because **SAVE** sets up the frame pointer for the called routine and **RTN** places the frame pointer for the calling routine in AC3.

The second method transfers control to the subroutine through the **JSR** instruction. Figure D.1 illustrates this subroutine method. The subroutine executes the **PSH** instruction to save the return address and returns control through the **POPJ** instruction.

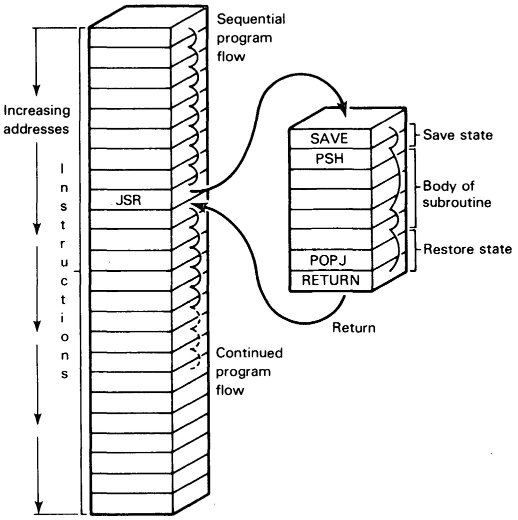


Figure D.1 Subroutine call and return

```
         JSR     SUBR     ;In calling program
         . . .
         . . .
         . . .
SUBR:    PSH     3,3      ;Subroutine
         . . .
         . . .
         . . .
RET:I    POPJ             ;Return to
                          ;calling program
```

Appendix E

Compatibility With ECLIPSE Line Computers

The microECLIPSE® series computers are compatible with the ECLIPSE line of computers, up to and including the ECLIPSE S/140 series computers. Any program currently running on an ECLIPSE S/140 computer will run on a microECLIPSE® series computer with the following changes:

Unique features—Data In Status returns the status of the addressed device and places this data into the specified accumulator.

Emulator trap—The CPU in the ECLIPSE S/120 system has a hardware provision for instruction emulation. If the CPU encounters an undefined instruction while operating in the mapped mode, it automatically makes a jump through location 11_8, provided that the contents are not zero. This location can contain the indirect location of an emulator routine.

Execution timing—The program may not be dependent on instruction execution times or I/O transfer times. Times for the ECLIPSE S/140 series computers may be faster than a MicroECLIPSE® computer, depending upon the application.

Reserved memory locations—Memory location 11 is now the location to which the processor will jump for an emulator trap and should contain the address fault handler routine.

Virtual Console—Commands are entered on a terminal keyboard. For more information on virtual console compatibility see "Virtual Console Features" in this appendix.

Automatic increment/Automatic decrement locations — Memory locations 20_8 to 27_8 and 30_8 to 37_8 are not available for this purpose on microECLIPSE® series computers.

Floating-point manipulation—The return address pushed during a floating-point trap is the address of the instruction following the instruction that caused the trap.

The floating point program counter is valid only when ANY is set.

Bit 9 of the FPSR is the resume bit. It should be ignored and not modified when saving and restoring the floating-point status register. When initializing the floating-point status, this should be set to zero.

Stack—No underflow protection is provided automatically. This can be accomplished with a user subroutine.

Bit 0 of the stack pointer will be set on a stack overflow if the return block pushed by the stack overflow routine wraps around from the top location (77777_8) to the bottom location 0 of memory.

MAP—If an instruction changes the current map state, the next instruction will be fetched from and executed in the new map state. See **DOA MAP**.

During a MAP fault, the program counter produces unpredictable results.

There are four user maps available.

Any attempt to read beyond the maximum physical address space will return undefined data. Any attempt to write beyond the maximum physical address space will have no effect.

Error Checking— ECLIPSE S/120 computer detects and corrects all single-bit memory errors. Double-bit and some triple-bit errors are detected but not corrected. However, their fault addresses and error syndrome codes are recorded, and an interrupt (when enabled) is issued.

The S/120 also implements an advanced error checking and correction feature *sniffing* that continuously tests all on-board memory.

Instructions

DIA MAP—Bit 0 will contain the extra map select bit, bit 1 will be set to 0 if the map is off, and will be set to 1 if the map is on.

No validity traps occur on map single-cycle references.

DIVS and DIVX— Carry will be set to one only if a overflow condition occurs.

IORST—Will not affect any bits in the floating point status register.

SKP—The AC field (bits 3 and 4) must be zero.

LMP—When I/O protection is on, a map fault occurs and no accumulators are altered.

AC0 must be equal to 0 or 3.

NIO—The AC field (bits 3 and 4) must be zero.

XOP1— Operates like the *Extended operation* instruction except that it adds 32 to the entry number before it adds the entry number to the **XOP** origin address.

Virtual Console Compatibility

Feature	Processor				
	ECLIPSE S/20	ECLIPSE S/120	ECLIPSE S/250	ECLIPSE S/130	ECLIPSE M600
Type of Break Point	ND	ND	ND	ND	ND
Number of Break Points	8	8	1	1	1
Single Step	Y	Y	Y	Y	Y
FPAC Support	N	Y	N	N	N
Break Key	Y	Y	N	N	N
Open Memory	Y	Y	Y	Y	Y
Open Accumulators	Y	Y	Y	Y	Y
Device Boot	H/L	U/D	Y	Y	Y
Field Cassette	Y	Y	N	N	N
Load MAP	N	N	N	N	N
Change MAP user	Y	Y	N	N	N
Power on Test	Y	Y	Y	Y	Y
IORST	I	I	R	R	R
Search with mask	N	Y	N	N	N
Complete Math	Y	Y	N	N	N
Uses User RAM	N	N	N	N	N
In User Space	N	N	N	N	N

ND= nondeleting
H/L=22H
U/D=User/DCH
R=Part of run command
I= I command

Appendix F

Instruction Summary

The following index alphabetically lists each instruction by assembler-recognizable mnemonic. It gives the format, data type used, action performed, and location contents before and after instruction execution.

The number located beneath each instruction mnemonic is the *base value* for that instruction. This base value is the 6-digit octal number that represents how the instruction would be assembled if all its options were omitted and all its operands were zero. For example, the base value for **ADD** is 103000. This represents an **ADD0,0** instruction.

Instruction Format	Operation	Before	After

ADC *[c][sh][#] acs,acd[,skip]*
102000
	Fixed-point
	ACS+ACD=ACD
		ACS=unsigned integer
		ACD=unsigned integer
			Unchanged
			Result

ADD *[c][sh][#] acs,acd[,skip]*
103000
	Fixed-point
	ACS+ACD=ACD
		ACS=unsigned integer
		ACD=unsigned integer
			Unchanged
			Result

ADDI *i,ac*
163770
	Fixed-point AC+I=AC	AC=unsigned integer
		I=unsigned integer
			Result
			Unchanged

ADI *n,ac* 100010	Fixed-point AC+*n*=AC	AC=unsigned integer *n*=unsigned integer (1-4)	Result Unchanged

ANC *acs,acd*
100610
	Fixed-point ACS AND ACD=ACD	ACS=unsigned integer
		ACD=unsigned integer
			Unchanged
			Result

AND *[c][sh][#] acs,acd[,skip]*
103400
	Fixed-point ACS AND ACD=ACD	ACS=unsigned integer
		ACD=unsigned integer
			Unchanged
			Result

ANDI *i,ac* 143770	Fixed-point AC AND immediate field=AC	AC=unsigned integer immediate field=unsigned integer	Result Unchanged

BAM
113710
	Fixed-point memory location+AC0=memory location	AC0=addend
		AC1=number of words
		AC2=source address
		AC3=destination address
			Addend
			0
			Last+1
			Last+1

BLM
133710
	Fixed-point memory location = memory location	AC1=number of words
		AC2=source address
		AC3=destination address
			0
			Last+1
			Last+1

BTO *acs,acd*
102010
	Bit (ACS and ACD)
	(Bit=1)
		ACS=word pointer
		ACD=word offset +bit pointer
		Memory location = address bit
			Unchanged
			Unchanged
			1

BTZ *acs,acd*
102110
	Bit (ACS and ACD)
	(Bit=0)
		ACS=word pointer
		ACD=word offset +bit pointer
		Memory location = Address bit
			Unchanged
			Unchanged
			0

CLM *acs,acd*
102370
	Fixed-point ACS>L and ACS<H=skip	ACS=2's complement number
		ACD=L address
			Unchanged
			Unchanged

	If ACS=ACD	ACS=2's complement number
		L=next word
		H=next word
			Unchanged
			Unchanged
			Unchanged

CMP
157650
	Character string1 compared to string2	AC0=String2 number of bytes
		AC1=String1 number of bytes
		AC2=String2 byte pointer
		AC3=String1 byte pointer
			0 or unpredictable result
			Result
			Last+1
			Last+1

Instruction Format	Operation	Before	After

CMT
167650
	Character memory location = memory location
	Delimiter Check made
		AC0=delimiter table address
		AC1=length and direction of string
		AC2=destination byte pointer
		AC3=source byte pointer
			Delimiter table address
			remaining number of bytes in string
			Last+1 or to failing byte
			Last+1 or to failing byte

CMV
153650
	Character memory location = memory location
	Carry = relative length
		AC0=destination number of bytes
		AC1=source number of bytes
		AC2=destination byte pointer
		AC3=source byte pointer
			0
			0 or unpredictable result
			Last+1
			Last+1

CTR
163650
	Character memory location = memory location
	Translates
	or
	String1 is compared to string2
	Translates
		AC0=Translation table byte pointer
		AC1=I and 2's complement bytes
		AC2=destination byte pointer
		AC3=source byte pointer
		AC0=translation table byte pointer
		AC1=length of string and number of bytes
		AC2=string2 byte pointer
		AC3=string1 byte pointer
			Unchanged
			0
			Last+1
			Last+1
			Unchanged
			Result
			Last+1 or to failing byte
			Last+1 or to failing byte

COB *acs,acd*
102610
	Bit ACS (1's)+ACD =ACD	ACS=unsigned integer
		ACD=2's complement number
			Unchanged
			Result

COM *[c][sh][#] acs,acd[,skip]*
100000
	Fixed-point ACS=ACD	ACS=unsigned integer
		ACD=2's complement number
			Unchanged
			Result

DAD *acs,acd*
100210
	Decimal ACS+ACD=ACD	ACS=binary-coded decimal
		ACD=binary-coded decimal
			Unchanged
			Result

DHXL *n,ac* 101610	Fixed-point AC and AC+1 hex shift left	AC=high order of number AC+1=low order of number *n*=unsigned integer (1-4)	Result Result Unchanged
DHXR *n,ac* 101710	Fixed-point AC and AC+1 Hex shift right	AC=high order of number AC+1=low order of number *n*=unsigned integer (1-4)	Result Result Unchanged
DIA *[f] ac,device*	I/O device (A buffer) = AC	A Buffer=unsigned integer	AC=Result
DIB *[f] ac,device*	I/O device (B buffer) = AC	B Buffer=unsigned integer	AC=Result
DIC *[f] ac,device*	I/O device (C buffer) = AC	C Buffer=unsigned integer	AC=Result
DIS *[t] ac,device*	I/O device status = AC		AC=Result
DIV 153710	Fixed-point (AC0 and AC1)/AC2	AC0=high-order number unsigned integer AC1=low-order number unsigned integer AC2=divisor unsigned integer	Remainder Quotient Unchanged

Instruction Format	Operation	Before	After

DIVS
157710
	Fixed-point (AC0 and AC1)/AC2	AC0=high-order 2's complement
		AC1=low-order 2's complement
		AC2=divisor 2's complement
			Remainder
			Quotient
			Unchanged

DIVX
137710
	Fixed-point (AC0 and AC1)/AC2	AC0=sign of AC1
		AC1=2's complement
		AC2=divisor 2's complement
			Remainder
			Quotient
			Unchanged

DLSH *acs,acd* 101310	Fixed-point ACD and (ACD+1) shift left/right	ACS=2's complement for shift ACD=high order ACD+1=low order	Unchanged Result Result
DOA *[f] ac,device*	I/O AC=device (A buffer)	AC=unsigned integer	Unchanged A buffer = Result
DOB *[f] ac,device*	I/O AC=device (B buffer)	AC=unsigned integer	Unchanged B buffer = Result
DOC *[f] ac,device*	I/O AC=device (C buffer)	AC=unsigned integer	Unchanged C buffer = Result

DSB *acs,acd*
100310
	Decimal ACD−ACS=ACD	ACS=binary coded decimal
		ACD=binary coded decimal
			Unchanged
			Result

DSPA *ac,[@]displ.[,index]*
142710
	Fixed-point AC<L or AC>H then (E−L)+unsigned integer address	AC=2's complement
		PC=PC
			Unchanged
			Address of PC

DSZ *[@]displacement[,index]* 014000	Fixed-point memory location = memory location − 1. If the value is 0, then skip.	Memory location = Unsigned integer	Unsigned integer
EDSZ *[@]displacement[,index]* 116070	Fixed-point memory location = memory location − 1. If the value = 0, then skip.	Memory locaction = Unsigned integer	Unsigned integer
EISZ *[@]displacement[,index]* 112070	Fixed-point memory location = memory location + 1. If the value = 0, then skip.	Memory location = Unsigned integer	Unsigned integer
EJMP *[@]displacement[,index]* 102070	Fixed-point calculated effective address =PC	PC=PC	Calculated effective address

EJSR *[@]displacement[,index]*
106070
	Fixed-point calculated effective address =PC	PC=PC
		AC3=unknown
			Calculated effective address
			PC+1

ELDA *ac,[@]displ.[,index]* 122070	Fixed-point memory location = AC	AC=unknown Memory location = unsigned integer	Unsigned integer Unchanged
ELDB *ac,displacement[,index]* 102170	Byte memory location = AC (right)	AC=unknown Memory location = unsigned integer	Unsigned integer Unchanged
ELEF *ac,[@]displ.[,index]*	Fixed-point calculated effective address = AC	AC= unknown	Calculated effective address Bit 0 = 0
ESTA *ac,[@]displ.[,index]* 142070	Fixed-point AC=memory location	Memory location = unknown AC=unsigned integer	AC Unchanged

Instruction Format	Operation	Before	After
ESTB *ac,displacement[,index]* 122170	Byte AC right half = memory location	Memory location=unknown	AC (right)
		AC=unsigned integer	Unchanged
FAB *fpac* 143050	Floating-point absolute value of (FPAC) =FPAC	FPAC=floating-point number	Absolute value of sign (floating-point number)
		FPSR(N,Z)	Updated
FAD *facs,facd* 100150	Floating-point FACS+FACD =FACD	FACS=Floating-point number	Unchanged
		FACD=Floating-point number	Floating-point (double precision)
		FPSR(N,Z)	Updated
FAMD *fpac,[@]displ.[,index]* 101150	Floating-point memory location+FPAC =FPAC	Memory location = double floating-point number D	Unchanged
		FPAC=Floating-point number	Floating-point (double precision)
		FPSR(N,Z)	Updated
FAMS *fpac,[@]displ.[,index]* 101050	Floating-point memory location+FPAC =FPAC	Memory location = single floating-point number S	Unchanged
		FPAC=Floating-point number	Floating-point (single precision)
		FPSR(N,Z)	Updated
FAS *facs,facd* 100050	Floating-point FACS+FACD =FACD	FACS=Floating-point number	Unchanged
		FACD=Floating-point number	Floating-point (single precision)
		FPSR(N,Z)	Updated
FCLE 153350	Floating-point FPSR bits 0-4=0	FPSR	
		ANY=unknown	0
		OVF=unknown	0
		UNF=unknown	0
		DVZ=unknown	0
		MOF=unknown	0
FCMP *facs,facd* 103450	Floating-point FACS compared to FACD	FACS=Floating-point number	Unchanged
		FACD=Floating-point number	Unchanged
		FPSR(N,Z)	Updated
FDD *facs,facd* 100750	Floating-point FACD/FACS =FACD	FACS=Floating-point number	Unchanged
		FACD=Floating-point number	Floating-point (double-precision)
		FPSR(N,Z)	Updated
FDMD *fpac,[@]displ.[,index]* 101750	Floating-point FPAC/memory location =FPAC	Memory location = double floating-point number D	Unchanged
		FPAC=Floating-point number	Floating-point number (double-precision)
		FPSR(N,Z)	Updated
FDMS *fpac,[@]displ.[,index]* 101650	Floating-point FPAC/memory location =FPAC	Memory location = single floating-point number S	Unchanged
		FPAC=Floating-point number	Floating-point number (single-precision)
		FPSR(N,Z)	Updated
FDS *facs,facd* 100650	Floating-point FACD/FACS =FACD	FACS=Floating-point number	Unchanged
		FACD=Floating-point number	Floating-point number (single-precision)
		FPSR(N,Z)	Updated

Instruction Format	Operation	Before	After
FEXP *fpac*	Floating-point AC0=FPAC	AC0=unknown	Unchanged
123150		FPAC=Floating-point number	Floating-point number (new exponent)
		FPSR(N,Z)	Updated
FFAS *ac,fpac*	Floating-point integer (FPAC) =AC	FPAC=Floating-point number	Unchanged
102650		AC0=unknown	Fixed-point number
FFMD *fpac,[@]displ.[,index]*	Floating-point absolute value of sign (FPAC) =memory location	FPAC=Floating point number	Unchanged
102750		Memory location = unknown	Fixed-point (double-precision)
FHLV *fpac*	Floating-point FPAC=FPAC/2	FPAC=Floating-point number	Result
163150		FPSR(N,Z)	Updated
FINT	Floating-point integer (FPAC) =FPAC	FPAC=Floating-point number	Result
143150		FPSR(N,Z)	Updated
FLAS *ac,fpac*	Floating-point AC=FPAC	AC=2's complement number	Unchanged
102450		FPAC=unknown	Floating-point (single-precision)
		FPSR(N,Z)	Updated
FLDD *fpac,[@]displ.[,index]*	Floating-point memory location = FPAC	Memory location = double floating-point number	Unchanged
102150		FPAC=unknown	Floating-point (double-precision)
		FPSR(N,Z)	Updated
FLDS *fpac,[@]displ.[,index]*	Floating-point memory location=FPAC	Memory location = single floating-point number	Unchanged
102050		FPAC=unknown	Floating-point number (single-*precision*)
		FPSR(N,Z)	Updated
FLMD *fpac,[@]displ.[,index]*	Floating-point memory location=FPAC	Memory location = 2's complement double floating-point number	Unchanged
102550		FPAC=unknown	Floating-point number (double-precision)
		FPSR(N,Z)	Updated
FLST *[@]displacement[,index]*	Floating-point memory location=FPSR	Memory location = single floating-point number	Unchanged
123350		FPSR(all)	Updated
FMD *facs,facd*	Floating-point (FACD)(FACS) =FACD	FACS=Floating-point number	Unchanged
100550		FACD=Floating-number	Floating-point number (double-precision)
		FPSR(N,Z)	Updated
FMMD *fpac,[@]displ.[,index]*	Floating-point (FPAC)(memory location) =FPAC	Memory location = double floating-point number	Unchanged
101550		FPAC=Floating-point number	Floating-point number(double-precision)
		FPSR(N,Z)	Updated

Instruction Format	Operation	Before	After
FMMS *fpac,[@]displ.[,index]* 101450	Floating-point (FPAC) (memory location) =FPAC	Memory location = single floating-point number	Unchanged
		FPAC=Floating-point number	Floating-point number (single-precision)
		FPSR(N,Z)	Updated
FMOV *facs,facd* 103550	Floating-point FACS=FACD	FACS=Floating-point number	Unchanged
		FACD=unknown	FACS
		FPSR(N,Z)	Updated
FMS *facs,facd* 100450	Floating-point (FACD)(FACS) =FACD	FACS=Floating-point number	Unchanged
		FACD=Floating-point number	Floating-point number (single-precision)
		FPSR(N,Z)	Updated
FNEG *fpac* 163050	Floating-point −FPAC=FPAC	FPAC=Floating-point number	Floating-point number
		FPSR(N,Z)	Updated
FNOM *fpac* 103050	Floating-point norm (FPAC)=FPAC	FPAC=Floating-point number	Floating-point number
		FPSR(N,Z)	Updated
FNS 103250	Floating-point never skip	PC=PC	PC
FPOP 167350	Floating-point stack=pop	stack=18 words	FPAC3 FPAC2 FPAC1 FPAC0 FPSR
FPSH 163350	Floating-point push=stack		Stack= FPSR FPAC0 FPAC1 FPAC2 FPAC3
FRH *fpac* 123050	Floating-point FPAC (high order) =AC0	FPAC=Floating-point number	Unchanged
		AC0=unknown	Floating-point number High 16 bits
FSA 107250	Floating-point skip always	PC=PC	PC+1
FSCAL *fpac* 103150	Floating-point AC0− FPAC (exponent) =FPAC (mantissa is shifted) AC0=FPAC (exponent)	AC0=unsigned integer	Unchanged
		FPAC=Floating-point number	Result
		FPSR(N,Z)	Updated
FSD *facs,facd* 100350	Floating-point FACS−FACD =FACD	FACS=Floating-point number	Updated
		FACD=Floating-point number	Floating-point number (double-precision)
		FPSR(N,Z)	Updated
FSEQ 113250	Floating-point FPSR (if Z=0 then skip)		
FSGE 127250	Floating-point FPSR (if N=0 then skip)		

Instruction Format	Operation	Before	After
FSGT 137250	Floating-point FPSR (if Z and N=0 then skip)		
FSLE 133250	Floating-point FPSR (if Z or N=0 then skip)		
FSLT 123250	Floating-point FPSR (if N=1 then skip)		

FSMD *fpac,[@]displ.[,index]*
101350
	Floating-point FPAC−memory location =FPAC	Memory location = double floating-point
		FPAC=Floating-point number
		FPSR(N,Z)
			Unchanged
			Floating-point number (double-precision)
			Updated

FSMS *fpac,[@]displ.[,index]*
101250
	Floating-point FPAC−memory location =FPAC	Memory location = single floating-point
		FPAC=Floating-point number
		FPSR(N,Z)
			Unchanged
			Floating-point number (single-precision)
			Updated

FSND 147250	Floating-point FPSR (if DVZ=0 then skip)		
FSNE 117250	Floating-point FPSR (if Z=0 then skip)		
FSNER 177250	Floating-point FPSR (if 1-4=0 then skip)		
FSNM 143250	Floating-point FPSR (if MOF=0 then skip)		
FSNO 163250	Floating-point FPSR (if OVF=0 then skip)		
FSNOD 167250	Floating-point FPSR (if OVF and DVZ=0 then skip)		
FSNU 153250	Floating-point FPSR (if UNF=0 then skip)		
FSNUD 157250	Floating-point FPSR (if UNF and DVZ=0 then skip)		
FSNUO 173250	Floating-point FPSR (if UNF and OVF=0 then skip)		

FSS *facs,facd*
100250
	Floating-point
	FACD−FACS=FACD
		FACS=Floating-point number
		FACD=Floating-point number
		FPSR(N,Z)
			Unchanged
			Floating-point number (single-*precision*)
			Updated

FSST *[@]displacement[,index]* 103350	Floating-point FPSR= memory location	Memory location = unknown FPSR=FPSR	FPSR Unchanged
FSTD *fpac,[@]displ.[,index]* 102350	Floating-point FPAC= memory location	FPAC=Floating-point number Memory location = unknown	Unchanged Floating-point number (double-precision)
FSTS *fpac,[@]displ.[,index]* 102250	Floating-point FPAC= memory location	FPAC=Floating-point number Memory location = unknown	Unchanged Floating-point number (single-precision)

Instruction Format	Operation	Before	After
FTD 147350	Floating-point FPSR (5=0)	FPSR (Trap enable bit)	0
FTE 143350	Floating-point FPSR (5=1)	FPSR (Trap enable bit)	1
HALT	Fixed-point stop		
HLV *ac* 143370	Fixed-point AC/2=AC	AC=2's complement	Result
HXL *n,ac* 101410	Fixed-point hex shift left =AC	*n*=unsigned integer (1-4) AC=unsigned integer	Unchanged Result
HXR *n,ac* 101510	Fixed-point hex shift right =AC	*n*=unsigned integer (1-4) AC=unsigned integer	Unchanged Result

INC *[c][sh][#] acs,acd[,skip]*
101400
	Fixed-point ACS+1=ACD	ACS=unsigned integer
		ACD=unknown
			Unchanged
			Result

IOR *acs,acd*
100410
	Fixed-point ACS or ACD=ACD	ACS=unsigned integer
		ACD=unsigned integer
			Unchanged
			Result

IORI *i,ac*
103770
	Fixed-point i or AC=AC	I=unsigned integer
		AC=unsigned integer
			Unchanged
			Result

ISZ *[@]displacement[,index]* 010000	Fixed-point if memory location = memory location +1 then skip	Memory location = unsigned integer	Unsigned integer+1

JMP *[@]displacement[,index]*
000000
	Fixed-point calculated effective address	PC=PC
		AC3=unknown
			Calculated effective address

JSR *[@]displacement[,index]* 004000	Fixed-point calculated effective address=PC	PC=PC	PC=calculated effective address
LDA *ac,[@]displ.[,index]* 020000	Fixed-point memory location = AC	AC=unknown	AC3=PC+1 Unchanged

LDB *acs,acd*
102710
	Byte memory location = ACD	ACS=Byte pointer
		ACD=unknown
		Memory location=byte
			Unchanged
			ACD=byte
			Memory location = unchanged

LEF *ac,[@]displ.[,index]* 060000	Fixed-point calculated effective address = AC	AC=unknown	Address

LMP
113410
	Map memory location = map	AC1=unsigned integer loaded
		AC2=1st address
			0
			Last+1

LOB *acs,acd*
102410
	Fixed-point ACS(0s)+ACD =ACD	ACS=unsigned integer
		ACD=2's complement number
			Unchanged
			Result

LRB *acs,acd*
102510
	Fixed-point ACS(0s)+ACD =ACD
	ACS (high order 0=1)
		ACS=unsigned integer
		ACD=2's complement number
			New unsigned integer
			Result

LSH *acs,acd*
101210
	Fixed-point ACD(shifted) =ACD	ACS=2's complement (±)
		ACD=unsigned integer
			Unchanged
			Result

MOV *[c][sh][#] acs,acd[,skip]*
101000
	Fixed-point ACS=ACD	ACS=unsigned integer
		ACD=unsigned integer
			Unchanged
			ACS

Instruction Format	Operation	Before	After
MSP *ac* 103370	Fixed-point stack pointer + AC = stack pointer	AC=2's complement number Stack pointer=unsigned integer	Unchanged Result

MUL
143710
	Fixed-point
	(AC1)(AC2)=unsigned integer
	unsigned integer + AC0= AC0
	and AC1
		AC0=intermediate unsigned integer
		AC1=unsigned integer
		AC2=unsigned integer
			High result
			Low result
			Unchanged

MULS
147710
	Fixed-point
	(AC1)(AC2)=unsigned integer
	unsigned integer + AC0= AC0 and AC1
		AC0=intermediate 2's complement number
		AC1=2's complement number
		AC2=2's complement number
			High result
			Low result
			Unchanged

NEG *[c][sh][#] acs,acd[,skip]*
163050
	Fixed-point −ACS=ACD	ACS=unsigned integer
		ACD=unknown
			Unchanged
			Result

NIO *[f] device*	I/O device flags set		
POP *acs,acd* 103210	Fixed-point stack=ACS> ACD	Stack=unknown	ACS−ACD
POPB 107710	Fixed-point stack=destination	Stack=5 words	−5 words. Return block

POPJ
117710
	Fixed-point stack=PC	Stack=1 word
		PC=unknown
			−1 word. Top word of stack

PSH *acs,acd* 103110	Fixed-point ACS>ACD= stack	Stack=unknown	ACS−ACD

PSHJ *[@]displacement[,index]*
102270
	Fixed-point PC+1=stack
	Calculated effective address=PC
		PC=PC
		Stack=unknown
			Calculated effective address
			PC+1

PSHR 103710	Fixed-point PC+2=stack	Stack=unknown	PC+2
RSTR 167710	Fixed-point stack=destination	Stack=9 words	−9 words. Destination= Return block + Stack fault address + Stack limit + Frame pointer + Stack pointer.

RTN
127710
	Fixed-point stack=destination	Stack pointer = stack pointer
		Stack=5 words
		Destination=unknown
			Stack pointer
			−5 words
			Carry +PC = 1st word
			AC3 = 2nd word
			AC2=3rd word
			AC1 = 4th word
			AC0 = 5th word

SAVE *i* 163710	Fixed-point 5 words+I =stack	Stack=unknown	AC0 AC1 AC2 Frame pointer Carry+AC3
SBI *n,ac* 100110	Fixed-point AC−*n*=AC	AC=unsigned integer *n*=unsigned integer (1-4)	Result Unchanged

SGE *acs,acd*
101110
	Fixed-point if ACS=ACD then skip	ACS=2's complement
		ACD=2's complement
			Unchanged
			Unchanged

Instruction Format	Operation	Before	After
SGT *acs,acd*	Fixed-point if ACS=ACD then skip	ACS=2's complement	Unchanged
101010		ACD=2's complement	Unchanged
STA *ac,[@]displ.[,index]*	Fixed-point AC=memory location	AC=unsigned integer	Unchanged
040000		memory location = unknown	Unsigned integer
STB *acs,acd*	Byte AC(right) =memory location	ACS=byte pointer	Unchanged
103010		ACD=byte	Unchanged
SKP *[t] device*	If t is true, then skip		Memory location=byte
SNB *acs,acd*	If addressed bit is set to one, then skip.	ACS=word pointer	Unchanged
102770		ACD=word offset and bit pointer	Unchanged
		Memory location=unknown	Unchanged
SUB *[c][sh][#] acs,acd[,skip]*	Fixed-point acd−ACS=ACD	ACS=unsigned integer	Unchanged
102400		ACD=unsigned integer	Result
SYC *acs,acd*	Fixed-point 5 words =stack@location 2=pc	ACS=unknown	Unchanged
103510		ACD=unknown	Unchanged
		PC=PC	@location 2
		Stack=unknown	Return block
SZB *acs,acd*	If addressed bit is set to zero, then skip.	ACS=word pointer	Unchanged
102210		ACD=word offset and bit pointer	Unchanged
		Memory location=unknown	Unchanged
SZBO *acs,acd*	If addressed bit is zero, set bit to one and skip.	ACS=word pointer	Unchanged
102310		ACD=word offset and bit pointer	Unchanged
		Memory location=unknown	1
VCT *[@]displacement[,index]*	Fixed-point. See Instruction		
XCH *acs,acd*	Fixed point ACS=ACD	ACS=unsigned integer	ACD
100710	ACD=ACS	ACD=unsigned integer	ACS
XCT *ac*	Fixed point AC=PC	PC=PC	AC instruction
123370		AC=Instruction	Unchanged
XOP *acs,acd,operation #*	Fixed-point unsigned integer + XOP table address =PC	44=table address	Unchanged
100030		Stack=unknown	Return block
		PC=PC	XOP unsigned integer
		AC2=unknown	Stack
			Address of ACS
		AC3=unknown	Stack
			Address of ACD
		AC1=unknown	AC1=Unchanged
		AC0=unknown	AC0=Unchanged
XOP1 *acs,acd,operation #*	Fixed-point. See XOR		
100070			
XOR *acs,acd*	Fixed-point ACS or ACD=ACD	ACS=unsigned integer	Unchanged
100510		ACD=unsigned integer	Result
XORI *i,ac*	Fixed-point i or AC=AC	AC=unsigned integer	Result
123770		I=unsigned integer	Unchanged

Arithmetic Operators

+	addition
−	subtraction or negation
/	division
()()	multiplication
inter	intermediate number

Index

Underscored page numbers identify dictionary entries.

A

- Abbreviations 75
- Absolute addressing
  - extended address field 6
  - intermediate address 6
  - short address field 6
- Absolute Value (FAB) 107
- AC relative addressing, see Accumulator relative addressing 7
- AC2 addressing, see Accumulator relative addressing 7
- Accumulator
  - destination accumulator (ACD) 9
  - source accumulator (ACS) 9
- Accumulator relative addressing, see Addressing 7
- ACS (source accumulator), definition 9
- ADC
  - Add Complement 12, 76
- ADD
  - Add 12, 76
- Add (ADD) 76
- Add Complement (ADC) 76
- Add Double 107, 108
- Add Immediate (ADI) 77
- Add instructions
  - Add 76
  - Add Complement 76
  - Add Immediate 77
  - Block Add and Move 79
  - Decimal Add 87
  - Extended Add Immediate 77
  - Extended Increment and Skip If Zero 103
  - Floating-Point Add Double—FPAC to FPAC 107
  - Floating-Point Add Double—Memory to FAPC 108
  - Floating-Point Add Single—FPAC to FPAC 109
  - Floating-Point Add Single—Memory to FPAC 108
  - Increment 137
- Add Single 108, 109
- ADDI
  - Extended Add Immediate 16, 77
- Address
  - stack fault 26, 27
- Address translation, definition 57
- Addressing
  - absolute 6, 6
  - AC relative 6
  - AC2 relative (mode 2) 6
  - AC3 relative (mode 3) 6
  - accumulator relative 7
  - conventions 5
  - direct and indirect 6, 7
  - effective address (EFA) calculation 8
  - extended class instructions 6
  - I/O controllers 35
  - indirect 5, 6, 7
  - intermediate 5, 6
  - logical 5, 57
  - maps 2
  - memory address calculation 8
  - modes 6
  - page boundary 6
  - page zero 6
  - PC relative addressing 6, 7
  - physical 5, 57
  - physical and logical compared 57
  - reserved memory locations 10
  - short class instructions 6
  - summary 2
  - translation 57
- Addressing mode range 7
- Addressing modes, definition 5
- ADI
  - Add Immediate 16, 77
- ALC, functions 13
- ALC format 12
- ALC instructions
  - Add 12
  - Add Complement 12
  - AND 12
  - Complement 12
  - Increment 12
  - Move 12
  - Negate 12
  - Subtract 12
  - summary 12
- Allocating memory, see also protecting memory 57
- Altering user MAPS 72

Altering data channel MAPS 72
Alternate Extended Operation (XOP1) 166
ALU
  carry 13
  shift 14
  skip 15
ALU operations 12
ANC
  AND With Complemented Source 17, 78
AND
  AND 12, 78
AND Immediate (ANDI) 17, 79
AND With Complemented Source (ANC) 78
ANDI
  AND Immediate 17, 79
Arithmetic logic class instructions, see ALC instructions 12
Arithmetic logic unit, see ALU 13
ASCII character codes 172
Asynchronous communications interface 40
Asynchronous interface 40
  controller registers 40
  instruction set 40
  Read Character 41
  timing 41
  Write Character 41
Asynchronous interface instructions
  Read Character 40
  Write Character 40
Asynchronous line controller (ALC) 40
Asynchronous Line Input 56, 92
  flag commands 53
  Read Character 54
Asynchronous Line Output 56, 92
Asynchronous line output
  flag commands 54
  Write Character 54
Auto program load 72
Auto program load register 44

B

B command 71
BAM
  Block Add and Move 17, 79
Battery backup 45
Baud rate 41, 41
Bit
  addressing 9
  displacement 5
  indirect 5
Bit manipulation instructions
  Count Bits 18
  Locate and Reset Lead Bit 18
  Locate Lead Bit 18
  Set Bit to One 18
  Set Bit to Zero 18
  Skip On Nonzero Bit 18
  Skip On Zero Bit 18
  Skip On Zero Bit and Set to One 18
  summary 18
Bit pointer 9
BLM
  Block Move 17, 80
Block Add and Move (BAM) 79
Block Move (BLM) 80
Breakpoints 70
  deleting 71
  encountering 71
  setting 71
BTO
  Set Bit to One 18, 80
BTZ
  Set Bit to Zero 18, 81
Busy flag 42
  values 35, 36, 37
Byte
  addressing 8, 8
  indicator 8
  pointer 8, 8
Byte indicator 8
Byte manipulation instructions
  Character Compare 18
  Character Move 18
  Character Move Until True 18
  Character Translate 18
  Extended Load Byte 18
  Extended Store Byte 18
  Load Byte 18
  Store Byte 18

C

C command 72
Carry 13
Carry mnemonics 13
Cell commands 69
Character Compare, CMP 82
Character instructions
  Character Move 84
  Character Compare 82
  Character Move Until True 83
  Character Translate 86
Character Move (CMV) 84
Character Move Until True (CMT) 83
Character Translate (CTR) 86
Checking, stack overflow 26

- Clear Errors (FCLE) 109
- CLM
  - Compare to Limits 32, 81
- CMP, Character Compare 18, 82
- CMT
  - Character Move Until True 18, 83
- CMV
  - Character Move 18, 84
- COB
  - Count Bits 18, 85
- Coding aids 75
- COM
  - Complement 12, 85
- Compare Floating-Point (FCMP) 110
- Compare instructions
  - Character Compare 82
  - Compare Floating-Point 110
- Compare to limits (CLM) 81
- Compatibility 181
- Complement (COM) 85
- Conditional skip instructions
  - Decrement and Skip if Zero 32
  - I/O Skip 32
  - Increment and Skip if Zero 32
  - Skip if ACS Greater Than ACD 32
  - Skip if ACS Greater Than or Equal 32
  - Skip on Nonzero Bit 32
  - Skip on Zero Bit 32
  - Skip on Zero Bit, Set to One 32
- Control flags, see also Flag values 37
- Conventions 75
- Convert instructions
  - Character Translate 86
  - Fix to AC 113
  - Fix to Memory 113
  - Float From AC 115
  - Float From Memory 116
  - Integerize 114
  - Normalize 120
  - Scale 123
- Count Bits (COB) 85
- CPU Acknowledge (DOAP CPU) 50, 98
- CPU device instructions
  - CPU Acknowledge 40, 50
  - CPU Skip 40, 51, 157
  - Halt 40, 50, 135
  - Interrupt Acknowledge 40, 50, 138
  - Interrupt Disable 40, 50, 138
  - Interrupt Enable 40, 50, 139
  - Mask Out 40, 51, 147
  - Read Processor Status 40
  - Read Switches 40
  - Read Virtual Console Register 51, 153
  - Reset 40, 50, 140
- CPU device mnemonics 49
- CPU I/O instructions
  - Interrupt Acknowledge 36
  - Interrupt Disable 36
  - Interrupt Enable 36
  - Mask Out 36
  - Read Switches 36
  - Reset 36
  - Vector on Interrupting Device 36
- CPU I/O operation, see also System I/O operation 40
- CPU Skip (SKP CPU) 51, 157
- CPU Status (DIS CPU) 49, 93
- CTR
  - Character Translate 18, 86

D

- D command 71
- DAD
  - Decimal Add 17, 87
- Data channel 73
  - I/O 35
  - operations 35
  - speed of operation 35
- Data channel I/O 35
- Data channel Map 58
- Data channel map
  - definition 57
  - initial conditions 60
- Data channel MAPS 72
- Data format
  - fixed-point 11
  - floating-point 19
- Data In A (DIA) 47, 89
- Data In B
  - DIB 47, 91
- Data in B (DIB) 47, 89
- Data In C (DIC) 47, 92
- Data In Status (DIS) 48, 94
- Data management and stack instructions
  - Alternate Extended Operation 166
  - Extended Operation 166
  - Modify Stack Pointer 147
  - Pop Block 151
  - Pop Multiple Accumulators 150
  - Pop PC and Jump 151
  - Push Floating-Point State 122
  - Push Jump 152
  - Push Multiple Accumulators 152
  - Push Return Address 153
  - Restore 154

  Return 154
  Save 155
  System Call 160
  Vector On Interrupting Device 162
Data management and stack instructions, Pop Floating-Point State 121
Data movement instructions
  Block Add and Move 17
  Block Move 17
  Extended Load Accumulator 17
  Extended Load Effective Address 17
  Extended Store Accumulator 17
  Load Accumulator 17
  Load Effective Address 17
  Store Accumulator 17
Data Out A (DOA) 48, 96
Data Out B (DOB) 48, 98
Data Out C (DOC) 48, 99
Data transfer
  data channel I/O 35
  programmed I/O 35
Decimal Add (DAD) 87
Decimal arithmetic instructions
  add 17
  subtract 17
Decimal Subtract (DSB) 100
Decrement and Skip If Zero (DSZ) 102
Definition, divide by zero 21
Definitions
  accumulators 9
  address translation 57
  addressing modes 5
  data channel I/O 35
  displacement bits 5
  effective address calculation 5
  excess 64 representation 20
  exponent 20
  extended address field 5
  extended operation 33
  floating-point 19
  floating-point format 19
  floating-point magnitude 19
  floating-point overflow 21
  floating-point status register 20
  floating-point underflow 21
  FPSR 20
  frame pointer 25
  guard digits 20
  impure zero 19
  index bits 5
  indirect addressing 5
  indirect bit 5
  intermediate address 5
  load effective address 60
  logical address 57
  lower page zero 5
  mantissa 20
  mantissa overflow 21
  memory space 57
  page 6, 57
  physical address 57
  programmed I/O 35
  pure zero 19
  push 25
  return block 25
  short address field 5
  sign 19
  stack 25
  stack control memory locations 26
  stack limit 26, 26
  stack overflow 26
  stack pointer 25
  supervisor 57
  user map 57
  validity protection 59
  word 5
  write protection 59, 59
Delete/rubout key 68
Device codes 169
DHXL
  Double Hex Shift Left 17, 88
DHXR
  Double Hex Shift Right 17, 88
DIA
  Data In A 35, 47, 89
DIA CPU, Read Virtual Console 153
DIA ERCC
  Read Memory Fault Address 56, 92
DIA MAP
  Read MAP Status 60, 63, 90
DIA PIT, Read Count 52
DIA TTI
  Read Character 40, 54, 56, 92
DIB
  Data In B 35, 47, 91
DIB CPU
  Interrupt Acknowledge 50, 138
DIB ERCC
  Read Memory Fault Code 56, 92
DIC
  Data In C 35, 47, 92
DIC CPU
  Reset 50, 140
DIC MAP
  Page Check 60, 64, 93
Direct addressing 7
Dirty zero, see Impure zero 19

DIS
  Data In Status 35, 48, 94
DIS CPU
  Read Processor Status 40
  CPU Status 93
DIS CPU (CPU Status) 49
Disable User Mode (NIOP MAP) 150
Disabling, stack overflow protection 26
Dismissing an interrupt, see also Program interrupt 39
Dispatch (DSPA) 101
Displacement bit, definition 5
Displaying user MAPS 72
Displaying data channel MAPS 72
DIV
  Unsigned Divide 16, 94
Divide by zero, definition 21
Divide Double—FPAC by FPAC (FDD) 110
Divide Double—FPAC by Memory (FDMD) 111
Divide instructions
  Divide Double—FPAC by FPAC 110
  Divide Double—FPAC by Memory 111
  Divide Single—FPAC by FPAC 112
  Divide Single—FPAC by Memory 111
  Halve 136
  Halve—FPAC 114
  Sign Extend and Divide 95
  Signed Divide 95
  Unsigned Divide 94
Divide Single—FPAC by FPAC (FDS) 112
Divide Single—FPAC by Memory (FDMS) 111
DIVS
  Signed Divide 16, 95
DIVX
  Sign Extend and Divide 16, 95
DLSH
  Double Logical Shift 17, 96
DOA
  Data Out A 35, 48, 96
DOA ERCC
  Enable ERCC 56, 92
DOA MAP
  Load MAP Status 60, 62, 97
DOA PIT, Initial Count 52
DOA RTC
  Select RTC Frequency 42, 53
DOA TTO
  Write Character 40, 54, 56, 92
DOAP CPU
  CPU Acknowledge 40, 50, 98
DOB
  Data Out B 35, 48, 98
DOB CPU, Mask Out 51, 147
DOB MAP
  MAP Supervisor Page 31 60, 65 99
DOC
  Data Out C 35, 48, 99
DOC CPU
  Halt 50, 135
DOC MAP
  Initiate Page Check 60, 100
DOC MAP (Initiate Page Check) 64
Done flag 42
  values 35, 36, 37
Double Hex Shift Left (DHXL) 88
Double Hex Shift Right (DHXR) 88
Double Logical Shift (DLSH) 96
DSB, Decimal Subtract 17
DSPA
  Dispatch 32, 101
DSZ
  Decrement and Skip If Zero 102
  Decrement and Skip if Zero 32
Dumping user MAPS 72
Dumping data channel MAPS 72

E

ECLIPSE compatibility 181
EDSZ
  Decrement and Skip If Zero (Extended) 102
  Decrement and Skip if Zero (Extended) 32
EFA (effective address calculation), definition 8
Effective address calculation, definition 5
EISZ
  Increment and Skip If Zero (Extended) 103
  Increment and Skip if Zero (Extended) 32
EJMP
  Extended Jump 32, 103
EJSR
  Extended Jump to Subroutine 32, 104
ELDA
  Extended Load Accumulator 17, 104
ELDB
  Extended Load Byte 18, 105
ELEF, Extended Load Effective Address 17
Enable ERCC 56, 92
  DOA ERCC 45, 55, 168
ERCC flag commands 56, 92
ERCC Instructions 45
  Enable ERCC 45, 55, 168
  Read memory fault address 45, 55, 168
  Read memory fault code 45, 55, 168

Error Checking and Correction 45, 55, 168
Error correction 68
ESTA
Error instruction 55, 168
  Extended Store Accumulator 17, 106
ESTB
  Extended Store Byte 18, 106
Examples
  arithmetic tests 177
  stack 29
  subroutines 177
Excess 64 representation, definition 20
Exchange Accumulators (XCH) 165
Exclusive OR (XOR) 167
Exclusive OR Immediate (XORI) 167
Execute (XCT) 165
Exponent, definition 20
Extended Add Immediate (ADDI) 77
Extended address field 75
Extended Decrement and Skip If Zero (EDSZ) 102
Extended Increment and Skip If Zero (EISZ) 103
Extended instructions
  Extended Add Immediate 77
  Extended Decrement and Skip If Zero 102
  Extended Increment and Skip If Zero 103
  Extended Jump 103
  Extended Jump to Subroutine 104
  Extended Load Accumulator 104
  Extended Load Byte 105
  Extended Operation 166
  Extended Store Accumulator 106
  Extended Store Byte 106
Extended Jump (EJMP) 103
Extended Jump to Subroutine (EJSR) 104
Extended Load Accumulator (ELDA) 104
Extended Load Byte (ELDB) 105
Extended Operation (XOP) 33 166
Extended Store Accumulator (ESTA) 106
Extended Store Byte (ESTB) 106

F

FAB
  Absolute Value 23, 107
FAD, Add Double—FPAC to FPAC 22, 107
FAM, Add Double (Memory to FPAC) 22
FAMD, Add Double—Memory to FPAC 108
FAMS, Add Single—Memory to FPAC 22, 108
FAS, Add Single—FPAC to FPAC 22, 109
Fault, stack fault handler 26
FCLE
  Clear Errors 24, 109
FCMP
  Compare Floating-Point 22, 110
FDD
  Divide Double—FPAC by FPAC 22, 110
  stack initialization 26
FDM
  Divide Double (FPAC by Memory) 22
  Divide Single (FPAC by Memory) 22
FDMD
  Divide Double—FPAC by Memory 111
  stack initialization 26
FDMS
  Divide Single—FPAC by Memory 111
  stack initialization 26
FDS
  Divide Single—FPAC by FPAC 22, 112
  stack initialization 26
FEXP
  Load Exponent 23, 112
FFAS
  Fix to AC 23, 113
FFMD
  Fix to Memory 23, 113
FHL, Halve (Floating-Point) 22
FHLV, Halve 114
FINT
  Integerize 23, 114
Fix to AC (FFAS) 113
Fix to Memory (FFMD) 113
Fixed-point
  data format 11, 12
Fixed-point arithmetic instructions
  Add Immediate 16
  Extended Add Immediate 16
  Halve 16
  Sign Extend and Divide 16
  Signed Divide 16
  Signed Multiply 16
  Subtract Immediate 16
  summary 16
  Unsigned Divide 16
  Unsigned Multiply 16
Fixed-point instruction lists
  arithmetic instructions 16
  bit manipulation 18
  byte manipulation 18
  data movement instructions 17
  decimal arithmetic 17
  logical operations 17
Fixed-point instructions
  Add Immediate 77
  Exchange Accumulators 165
  Extended Add Immediate 77
  Extended Store Accumulator 106
  Sign Extend and Divide 95
  Signed Divide 95
  Signed Multiply 148
  Store Accumulator 159
  Subtract 160
  Subtract Immediate 156
  Unsigned Divide 94
  Unsigned Multiply 148

- Flag commands 56, 56, 92, 92
  - asynchronous line input 53
  - asynchronous line output 54
  - PIT 52
  - RTC 53
- Flag instructions 56, 92
- Flag values
  - Busy 35, 36, 37, 42
  - Done 35, 36, 37, 42
  - ION 36, 37, 38
  - real-time clock 42
  - setting ION flag in CPU 38
  - testing ION flag in CPU 38
- FLAS
  - Float From AC 23, 115
- FLDD
  - Load Floating-Point Double 22, 115
- FLDS
  - Load Floating-Point Single 22, 116
- FLMD
  - Float From Memory 23, 116
- Float From AC (FLAS) 115
- Float From Memory (FLMD) 116
- Float instructions
  - Float From AC 115
  - Float From Memory 116
- Floating-point
  - composition 19
  - data format 19
  - definition 19
  - divide by zero 21
  - double-precision format 19
  - exponent 19
  - faults 21
  - guard digits 20
  - magnitude 19
  - mantissa 19, 20
  - mantissa overflow 21
  - operation 20
  - overflow 21
  - pure zero 19
  - registers 20
  - sign bit 19
  - single-precision format 19
  - stack allocation 20
  - status registers 20
  - trap 21
  - underflow 21
  - zero 19
- Floating-point arithmetic instructions 22
- Floating-point data movement instructions 22
- Floating-Point Divide Double (FPAC by FPAC), stack initialization 26
- Floating-Point Divide Double (FPAC by Memory), stack initialization 26
- Floating-Point Divide Single (FPAC by FPAC), stack initialization 26
- Floating-Point Divide Single (FPAC by Memory), stack initialization 26
- Floating-point fault, return block 21
- Floating-point guard digits, definition 20
- Floating-point instruction, status register instructions 24
- Floating-point instruction lists 22
- Floating-Point Multiply Double, stack initialization 26
- Floating-point number conversion instructions 23
- Floating-point overflow, definition 21
- Floating-point pop, stack initialization 26
- Floating-point program alteration instructions 23
- Floating-point push, stack initialization 26
- Floating-point registers, FPAC 20
- Floating-point status register
  - definition 20, 20
- Floating-point status register instructions 24
- Floating-point underflow, definition 21
- FLST
  - Load Floating-Point Status 24, 117
- FMS
  - Multiply Single (FPAC by FPAC) 22
- FMD
  - Multiply Double—FPAC by FPAC 22, 117
  - stack initialization 26
- FMM
  - Multiply Double (Memory by FPAC) 22
  - Multiply Single (Memory by FPAC) 22
- FMMD
  - Multiply Double—FPAC by Memory 118
  - stack initialization 26
- FMMS, Multiply Single—FPAC by Memory 118
- FMOV
  - Move Floating-Point 22, 119
- FMS, Multiply Single—FPAC by FPAC 119
- FNE, Negate (Floating-Point) 22
- FNEG, Negate 120

- FNOM
  - Normalize 23, 120
- FNS
  - No Skip 23, 121
- Format
  - ALC 12
  - of commands 75
- FPOP
  - Pop Floating-Point State 29, 121
- FPSH
  - Push Floating-Point State 29, 122
- FPSR
  - definition 20
  - see also Floating-point status register 20
- Frame pointer
  - definition 25
  - initialization 25
- FRH
  - Read High Word 23, 122
- FSA
  - Skip Always 23, 123
- FSCAL
  - Scale 23, 123
- FSD, Subtract Double—FPAC from FPAC 22, 124
- FSDN, Skip On No Zero Divide 128
- FSEQ
  - Skip On Zero 23, 124
- FSGE
  - Skip On Greater Than or Equal to Zero 23, 125
- FSGT
  - Skip On Greater Than Zero 23, 125
- FSLE
  - Skip On Less Than or Equal to Zero 23, 126
- FSLT
  - Skip On Less Than Zero 23, 126
- FSM, Subtract Single (Memory from FPAC) 22
- FSMD, Subtract Double—Memory from FPAC 22, 127
- FSMS, Subtract Single—Memory from FPAC 127
- FSND, Skip On No Zero Divide 23
- FSNE
  - Skip On Nonzero 23, 128
- FSNER
  - Skip On No Error 23, 129
- FSNM
  - Skip On No Mantissa Overflow 23, 129
- FSNO
  - Skip On No Overflow 23, 130
- FSNOD
  - Skip On No Overflow and No Zero Divide 23, 130
- FSNU
  - Skip On No Underflow 23, 131
- FSNUD
  - Skip On No Underflow and No Zero Divide 131
  - Skip On No Underfow and No Zero Divide 23
- FSNUO
  - Skip On No Underflow and No Overflow 23, 132
- FSS, Subtract Single—FPAC from FPAC 22, 132
- FSST
  - Store Floating-Point Status 24, 133
- FSTD
  - Store Floating-Point Double 22, 133
- FSTS
  - Store Floating-Point Single 22, 134
- FTD
  - Trap Disable 24, 134
- FTE
  - Trap Enable 24, 135
- Function commands, virtual console 70

G

- G command 74
- Guard digits, see Floating-point guard digits 20

H

- H command 73
- HALT
  - Halt 32, 50, 135
- HALTA
  - Halt 32, 40, 135
- Halve (FHLV) 114
- Halve (HLV) 136
- Hex Shift Left (HXL) 136
- Hex Shift Right (HXR) 137
- Hexadecimal and octal conversion 171
- HLV
  - Halve 16, 136

- HXL
  - Hex Shift Left 17, 136
- HXR
  - Hex Shift Right 17, 137

I

- I command 73
- I/O, format 36
- I/O bus 2
- I/O control flags 36
- I/O controllers, addressing of 35
- I/O device codes 169
- I/O device controller, program flow interruption 32
- I/O format 36
  - accumulator field 36
  - control field 36
  - device field 36
  - instruction field 36
- I/O instructions 35
- I/O interfaces 40
- I/O interrupt instructions
  - Interrupt Acknowledge 138
  - Mask Out 147
  - Vector On Interrupting Device 162
- I/O latency 41
- I/O management system, summary 2
- I/O operations 35
- I/O Skip (SKP) 48, 158
- I/O test flag (SKP instruction only) 37
- Impure zero, definition 19
- INC
  - Increment 12, 137
- Inclusive OR (IOR) 139
- Inclusive OR Immediate (IORI) 140
- Increment (INC) 137
- Increment and Skip If Zero (ISZ) 141
- Index bits
  - definition 5
- Index field
- Index register addressing, see Accumulator relative addressing
- Indirect addressing 7
  - definition 5
- Indirect bit, definition 5
- Indirect chain, definition 7
- Indirection protection, definition 59
- Initiate Page Check (DOC MAP) 64, 100
- Input/Output, see I/O 2
- Instruction 75
  - action performed 183
  - contents after execution 183, 183
  - data type used 183
  - examples 177
  - execution times 173
  - format 183
  - summary 183
- Instruction error checking and correction 45, 55, 168
- Instructions 56, 92
  - ALC 12
  - asynchronous interface 40
  - asynchronous line input 53
  - asynchronous line output 54
  - bit manipulation 18
  - conditional skip 32, 32
  - CPU device 40, 49
  - CPU I/O 36
  - decimal arithmetic 17
  - Extended Operation 33
  - fixed-point arithmetic 16
  - floating-point data movement 22
  - floating-point status register 24
  - logical operations 17
  - MAP 60
  - number conversion 23
  - power fail 46
  - program flow alteration 32
  - programmable interval timer 52
  - real-time clock 53
  - stack 27
  - standard I/O 35, 47
- INTA (DIB
  - CPU), Interrupt Acknowledge 36, 40, 138
- INTDS, Interrupt Disable 36, 40, 138
- Integerize (FINT) 114
- Integers
  - double-precision range 11
  - see also Fixed-point 11
  - single-precision range 11
- INTEN, Interrupt Enable 139
- INTEN (NIOS CPU), Interrupt Enable 36, 40
- Intermediate address
  - definition 5
  - extended address field 5
  - short address field 5
- Internal cells 69
- Interrupt Acknowledge (DIB CPU) 50, 138
- Interrupt Acknowledge (INTA) 138
- Interrupt Disable (INTDS) 138
- Interrupt Disable (NIOC CPU) 50, 138, 138
- Interrupt Enable (INTEN) 139

- Interrupt Enable (NIOS CPU) 50, 139, 139
- Interrupt handler, see also Program interrupt 37
- Interrupt instructions
  - Interrupt Acknowledge 138
  - Trap Disable 134
  - Trap Enable 135
- Interrupt On flag 38
- Interrupt priority mask 39
  - see also Program interrupt 39
- Interrupt service 37
- Instructions
  - ALC 12
  - byte manipulation 18
  - data movement 17
  - floating-point program flow alteration 22
- ION flag
  - setting or testing 38
  - values 36, 37, 38
- IOR
  - Inclusive OR 17, 139
- IORI
  - Inclusive OR Immediate 17, 140
- IORST (DIC CPU), Reset 36, 40, 140
- ISZ
  - Increment and Skip If Zero 32, 141

J

- JMP
  - Jump 32, 141
- JSR
  - Jump to Subroutine 32, 142
- Jump (JMP) 141
- Jump instructions
  - Dispatch 101
  - Extended Jump 103
  - Extended Jump to Subroutine 104
  - Jump 141
  - Jump to Subroutine 142
  - Pop PC and Jump 151
- Jump to Subroutine (JSR) 142

K

- K command 68

L

- L command 72
- LDA
  - Load Accumulator 17, 142
- LDB
  - Load Byte 18, 143
- LEF
  - Load Effective Address 17, 143
- LEF mode, see Load Effective Address
- LMP
  - Load MAP 60, 61, 144
- Load Accumulator (LDA) 142
- Load Byte (LDB) 143
- Load Effective Address (LEF) 143
- Load Effective Address mode, definition 60
- Load Exponent (FEXP) 112
- Load Floating-Point Double (FLDD) 115
- Load Floating-Point Single (FLDS) 116
- Load Floating-Point Status (FLST) 117
- Load instructions
  - Load Accumulator 142
  - Load Byte 143
  - Load Effective Address 143
  - Load Exponent 112
  - Load Floating-Point Double 115
  - Load Floating-Point Single 116
  - Load Floating-Point Status 117
  - Load Map 144
  - Load Map Status 97
- Load Map (LMP) 61, 144
- Load MAP Status, (DOA MAP) 62, 97
- Load/no load 15
- LOB
- Locate and Reset Lead Bit (LRB) 145
- Locate instructions
  - Locate and Reset Lead Bit 145
  - Locate Lead Bit 145
- Logic instructions, AND Immediate 79
- Logic instructions
  - Absolute Value 107
  - AND 78
  - AND With Complemented Source 78
  - Character Move Until True 83
  - Compare Floating-Point 110
  - Complement 85
  - Count Bits 85
  - Double Hex Shift Left 88
  - Double Hex Shift Right 88
  - Double Logical Shift 96
  - Exclusive OR 167
  - Hex Shift Left 136
  - Hex Shift Right 137
  - Inclusive OR 139
  - Inclusive OR Immediate 167
  - Locate and Reset Lead Bit 145
  - Locate Lead Bit 145
  - Logical Shift 146
  - Negate 149
  - Negate (floating-point) 120
  - Set Bit to One 80
  - Set Bit to Zero 81
- Logical address
  - compared to physical 57
  - definition 57
- Logical operation instructions 17
- Logical Shift (LSH) 146

- LRB
  - Locate and Reset Lead Bit 18, 145
- LSH
  - Logical Shift 17, 146

M

- M command 72
- Mantissa
  - definition 20
  - format 20
  - normalization 20
- Mantissa overflow, definition 21
- MAP, changing user 72
- Map
  - data channel 57, 57, 58
- MAP, initial conditions 60
- Map
  - modes 57
  - user 57, 57
- MAP functions
  - address translation 57
  - sharing physical memory 57
- MAP instructions
  - Disable User Mode 150
  - Initiate Page Check 60, 64, 100
  - Load Map 60, 61, 144
  - Load Map Status 60, 62, 97
  - Map Single Cycle 60, 65, 150
  - Map Supervisor Page 31 60, 65, 99
  - Page Check 60, 64, 93
  - Read MAP Status 60, 63, 90
- Map operations, load effective address 60
- MAP protection
  - I/O 59
  - Indirect 59
  - see also MAP 59
  - Validity 59
  - Write 59
- Map protection faults 60
- MAP Single Cycle (NIOP MAP) 65, 150
- MAP Supervisor Page 31 (DOB MAP) 65, 99
- Mapped mode 57
- Mapping
  - user address space 57, 57
- MAPS
  - displaying data channel 72
  - displaying user MAPS 72
- Mask, see also Program interrupt 39
- Mask Out (DOB CPU) 51, 147
- Mask Out (MSKO) 147
- Memory
  - address calculation 8
  - reserved memory locations 10
- Memory allocation, summary 2
- Memory allocation and protection
  - see MAP 57
  - summary 2
- Memory cells 69
- Memory protection, summary 2
- Memory space, definition 57
- MicroECLIPSE compatibility 181
- microECLIPSE features 1
- Mnemonics 40
- Mode 0 addressing, see Addressing, absolute 6
- Mode 1 addressing, see Addressing 7
- Mode 2 addressing, see Accumulator relative addressing 7
- Mode 3 addressing, see Accumulator relative addressing 7
- Modify Stack Pointer (MSP) 147
- MOV
  - Move 12, 146
- Move (MOV) 146
- Move Floating-Point (FMOV) 119
- Move instructions
  - Block Add and Move 79
  - Block Move 80
  - Character Move 84
  - Character Move Until True 83
  - Double Hex Shift Left 88
  - Double Hex Shift Right 88
  - Exchange Accumulators 165
  - Extended Load Accumulator 104
  - Extended Load Byte 105
  - Extended Store Accumulator 106
  - Extended Store Byte 106
  - Hex Shift Left 136
  - Hex Shift Right 137
  - Load Accumulator 142
  - Load Byte 143
  - Load Exponent 112
  - Load Floating-Point Double 115
  - Load Floating-Point Single 116
  - Move 146
  - Move Floating-Point 119
  - Pop Block 151
  - Pop Multiple Accumulators 150
  - Read High Word 122
  - Store Accumulator 159

  Store Byte 159
  Store Floating-Point Single 134
  Trap Disable 134
MSKO, dismissal 39
MSKO (DOB CPU), Mask Out 36, 40
MSP, Modify Stack Pointer 147
MUL
  Unsigned Multiply 16, 148
MULS
  Signed Multiply 16, 148
Multiply Double—FPAC by FPAC (FMD) 117
Multiply Double—FPAC by Memory (FMMD) 118
Multiply instructions
  Multiply Double—FPAC by FPAC 117
  Multiply Double—FPAC by Memory 118
  Multiply Single—FPAC by FPAC 119
  Multiply Single—FPAC by Memory 119
  Signed Multiply 148
  Unsigned Multiply 148
Multiply Single—FPAC by FPAC (FMS) 119
Multiply Single—FPAC by Memory (FMMS) 118

N

NEG
  Negate 12, 149
Negate (FNEG) 120
Negate (NEG) 149
NIO
  No I/O Transfer 35, 48, 149
NIOC CPU
  Interrupt Disable 50, 138
NIOP MAP
  Disable User Mode 150
  Map Single Cycle 60, 65, 150
NIOS CPU
  Interrupt Enable 50, 139
No I/O Transfer (NIO) 48, 149
No Skip (FNS) 121
Normalization, of mantissa 20
Normalize (FNOM) 120

O

O command 71
Octal and hexadecimal conversion 171
Operation, floating-point 20
Operations
  arithmetic logic unit 12
  carry 13
  data channel 35
  floating-point 19
  I/O 35, 36
  integer 11
  load 15
  no load 15
  program interrupt 37
  shift 14
  skip 15
  stack 25
Optional I/O mnemonics 47
Overflow checking 26
Overflow protection 26, 26

P

P command 74
Page
  definition 6, 57
Page Check (DIC MAP) 64, 93
Page zero 5
PC relative addressing, see addressing 7
Physical address
  compared to logical 57
  definition 57
Physical memory sharing 57
PIT
  see Programmable interval timer
Pointer
  frame 25
  stack 25
Pop, definition 25
POP
  Pop Multiple Accumulators 29, 150
Pop Block (POPB) 151
Pop Floating-Point State (FPOP) 121
Pop instructions
  Pop Block 151
  Pop Floating-Point State 121
  Pop Multiple Accumulators 150
  Pop PC and Jump 151
Pop Multiple Accumulators (POP) 150
Pop PC and Jump (POPJ) 151
POPB
  Pop Block 29, 151
POPJ
  Pop PC and Jump 29, 151
Power fail 45
Power fail instructions
  Skip If Power Flag Is One 46
  Skip If Power Flag Is Zero 46
Power fail/autorestart, example 46
Power-up response 42
Power-up sequence for the CPU 44
Priority interrupt handler 39
Priority interrupts 39
Priority mask, definition 39
Program counter relative addressing, see PC relative addressing
Program debugging 70
Program flow 31
Program flow alteration 31

- Program flow alteration instructions
  - Compare to Limits 32
  - Dispatch 32, 101
  - Extended Operation 32, 166
  - Halt 32, 135
  - Jump 32, 141
  - Jump to Subroutine 32, 142
  - System Call 32, 160
  - Vector On Interrupting Device 32, 162
- Program flow interruption 32
- Program interrupts 37, 39
  - priority interrupt handler 39
  - priority mask 39
  - with priority 39
- Program load register, see Auto program load register 44
- Program return 71, 72
- Programmable interval timer 40, 43
  - controller registers 43
  - flag commands 52
  - instruction set 43
  - programming 43
- Programmable interval timer instructions
  - Read Count 52
  - Specify Initial Count 52
- Programmable real-time clock 40
- Programmed I/O 35
  - definition 35
- Programmed I/O controller 40
- Programming aids 172
- Programming examples 177
- Protecting memory
  - see also Allocating memory 59
  - summary 2
- Protecting stack overflow, see also Stack overflow 26
- Protection faults 60
- PSH
  - Push Multiple Accumulators 29, 152
- PSHJ
  - Push Jump 29, 152
- PSHR
  - Push Return Address 29, 153
- Pure zero, definition 19
- Push, definition 25
- Push Floating-Point State (FPSH) 122
- Push instructions
  - Push Floating-Point State 122
  - Push Jump 152
  - Push Multiple Accumulators 152
  - Push Return Address 153
- Push Jump (PSHJ) 152
- Push Multiple Accumulators (PSH) 152
- Push Return Address (PSHR) 153

R

- R command 72
- Read auto load register definitions 44
- Read Character 56, 92
- Read Character (DIA TTI) 54
- Read Count (DIA PIT) 52
- Read High Word (FRH) 122
- Read MAP Status
  - (DIA MAP) 63, 90
- Read memory fault address 56, 92
  - DIA ERCC 45, 55, 168
- Read memory fault code 56, 92
  - DIB ERCC 45, 55, 168
- Read Virtual Console (READS) 153
- Read Virtual Console (DIA CPU) 153
- Read Virtual Console Register (READS) 51
- READS
  - Read Switches 40
  - Read Virtual Console Register 51, 153
- READS (DIA, CPU), Read Switches 36
- Real-time clock
  - flag commands 53
  - instruction set 42
  - power-up response 43
  - program flow interruption 32
  - programming 43
  - timing 43
- Real-time clock flag commands 42
- Real-time clock instruction
  - Select Frequency 53
  - Select RTC Frequency 42
- Real-time clock interface 42
  - controller registers 42
- Registers
- Reserved memory locations 10
- Reset (DIC CPU) 50, 140
- Reset (IORST) 140
- Restore, RSTR 154
- Return (RTN) 154
- Return block, definition 25
- RSTR
  - Restore 29, 154
- RTC 40
  - see also Real-time clock 42
- RTN
  - Return 29, 154
- Rubout/delete key 68

S

S command 73
S/120 computer features 1
SAVE
  Save 29, 155
SBI
  Subtract Immediate 16, 156
Scale (FSCAL) 123
Select Frequency (DOA RTC) 53
Self-test 74
Servicing an interrupt 37
Set Bit to One (BTO) 80
Set Bit to Zero (BTZ) 81
SGE
  Skip If ACS Greater Than or Equal to ACD 32, 156
SGT
  Skip If ACS Greater Than ACD 32, 157
Shift mnemonics 14
Shift operations 14
Sign, definition 19
Sign Extend and Divide (DIVX) 95
Signed Divide (DIVS) 95
Signed instructions
  Signed Divide 95
  Signed Multiply 148
Signed Multiply (MULS) 148
Single stepping 71
Skip 15
Skip (conditional)
  Decrement and Skip if Zero 32
  I/O Skip 32
  Increment and Skip if Zero 32
  Skip if ACS Greater Than ACD 32
  Skip if ACS Greater Than or Equal 32
  Skip on Nonzero Bit 32
  Skip on Zero Bit 32
  Skip on Zero Bit, Set to One 32
Skip Always (FSA) 123
Skip If ACS Greater Than ACD (SGT) 157
Skip If ACS Greater Than or Equal to ACD (SGE) 156
Skip instructions
  Skip If ACS Greater Than ACD 157
  Skip If ACS Greater Than or Equal to ACD 156
  Skip On Greater Than or Equal to Zero 125
  Skip On Greater Than Zero 125
  Skip On Less Than or Equal to Zero 126
  Skip On Less Than Zero 126
  Skip On No Error 129
  Skip On No Mantissa Overflow 129
  Skip On No Overflow 130
  Skip On No Overflow and No Zero Divide 130
  Skip On No Underflow 131
  Skip On No Underflow and No Overflow 130
  Skip On No Underflow and No Zero Divide 131
  Skip On No Zero Divide 128
  Skip On Nonzero 128
  Skip On Nonzero Bit 158
  Skip On Zero 124
  Skip On Zero Bit 161
  Skip On Zero Bit and Set to One 161
Skip instructions (nonconditional skip), No Skip 121
Skip instructions (nonconditional), Skip Always 123
Skip mnemonics 15
Skip On Greater Than or Equal to Zero (FSGE) 125
Skip On Greater Than Zero (FSGT) 125
Skip On Less Than or Equal to Zero (FSLE) 126
Skip On Less Than Zero (FSLT) 126
Skip On No Error (FSNER) 129
Skip On No Mantissa Overflow (FSNM) 129
Skip On No Overflow (FSNO) 130
Skip On No Overflow and No Zero Divide (FSNOD) 130
Skip On No Underflow (FSNU) 131
Skip On No Underflow and No Overflow (FSNUO) 132
Skip On No Underflow and No Zero Divide (FSNUD) 131
Skip On No Zero Divide (FSDN) 128
Skip On Nonzero (FSNE) 128
Skip On Nonzero Bit (SNB) 158
Skip On Zero (FSEQ) 124
Skip On Zero Bit (SZB) 161
Skip On Zero Bit and Set to One (SZBO) 161
SKP
  I/O Skip 32, 35, 48, 158
SKP CPU
  CPU Skip 40, 51, 157
SKPDN CPU, Skip if Power Flag Is One 46
SKPDZ CPU, Skip If Power Flag Is Zero 46
SNB
  Skip On Nonzero Bit 18, 32, 158
Special mnemonics 40
Specify Initial Count (DOA PIT) 52
SPU I/O operation, see also System I/O operation 40
STA
  Store Accumulator 17, 159
Stack
  initialization 25
  operation 25
  usage 25
Stack and data management, Alternate Extended Operation 166
Stack and data management instructions
  Extended Operation 166
  Modify Stack Pointer 147
  Pop Block 151
  Pop Floating-Point State 121

  Pop Multiple Accumulators 150
  Pop PC and Jump 151
  Push Floating-Point State 122
  Push Jump 152
  Push Multiple Accumulators 152
  Restore 154
  Return 154
  Save 155
  System Call 160
  Vector On Interrupting Device 162
Stack and data managment instructions, Push Return Address 153
Stack control memory locations 26
Stack definition 25
Stack examples 29
Stack fault 26
Stack fault address 26
Stack fault handler 26, 27
Stack initialization 25, 26, 26, 26, 26
  floating-point pop 26
  floating-point push 26
Stack instruction interrupts 27
Stack instructions 29
Stack limit
  definition 26, 26
Stack operation
  control memory locations 26
  frame pointer 25
  initialization 25, 26, 26, 26
  overflow checking 26
  pop 25
  protection 26
  push 25
  stack limit 26, 26
  stack pointer 25
Stack overflow
  checking 26
  definition 26
  protection 26
  stack fault address 26
  stack fault handler 26, 27
Stack overflow error, program flow interruption 32
Stack overflow protection 26
  disabling 26
Stack pointer
  definition 25
  initialization 25
Stack storage locations 25
Standard I/O instruction
  Data in A 47, 89
  Data in B 47, 89
Standard I/O instructions
  Data In B 47, 91
  Data In C 47, 92
  Data in Status 47, 48 92
  I/O Skip 48
Status instructions
  Load Floating Point Status 117
  Store Floating-Point Status 133
  Trap Enable 135
Status register, see Floating-point status register 20
STB
  Store Byte 18, 159
Store Accumulator (STA) 159
Store Byte (STB) 159
Store Floating-Point Double (FSTD) 133
Store Floating-Point Single (FSTS) 134
Store Floating-Point Status (FSST) 133
Store instructions
  Store Accumulator 159
  Store Byte 159
  Store Floating-Point Double 133
  Store Floating-Point Single 134
  Store Floating-Point Status 133
SUB
  Subtract 12, 160
Subroutine instructions
  Alternate Extended Operation 166
  Extended Jump to Subroutine 104
  Extended Operation 166
  Jump to Subroutine 142
  Push Return Address 153
  Restore 154
  Return 154
  Save 155
Subtract (SUB) 160
Subtract Double—FPAC from FPAC (FSD) 124
Subtract Double—Memory from FPAC (FSMD) 127
Subtract Immediate (SBI) 156
Subtract instructions
  Decimal Subtract 100
  Decrement and Skip If Zero 102
  Extended Decrement and Skip If Zero 102
  Subtract 160
  Subtract Double—Memory from FPAC 127
  Subtract Immediate 156
  Subtract Single—FPAC from FPAC 132
  Subtract Single—Memory from FPAC 127
Subtract Single—FPAC from FPAC (FSS) 132
Subtract Single—Memory from FPAC (FSMS) 127
Summaries 2
Summary, ECLIPSE S/20 features 1
Supervisor, definition 57
SYC

  System Call 29, 32, 160
System block diagram 1
System Call (SYC) 160
System I/O controller 2
System I/O operation
  asynchronous interface 2
  asynchronous line controller 40
  auto program load register 44
  data channel transfer 2, 2
  power-up sequence 44
  programmable interval timer 2, 43
  real-time clock 2, 42
System overview 1
SZB
  Skip On Zero Bit 18, 32, 161
SZBO
  Skip On Zero Bit and Set to One 18, 32, 161

T

Timing, see Asynchronous interface 41
Translating, addresses 57
Trap Disable (FTD) 134
Trap Enable (FTE) 135
True zero, see Pure zero
TTI 40
TTO 40
Two's complement 11

U

U command 72
Unmapped mode 59
Unsigned Divide (DIV) 94
Unsigned instructions
  Unsigned Divide 94
  Unsigned Multiply 148
Unsigned Multiply (MUL) 148
User map 57
  definition 57
  initial conditions 60
User MAPS 72

V

V command 74
Validity protection, definition 59
VCT
  Vector On Interrupting Device Code 29, 32, 26, 162
Vector instruction modes 38
Vector instruction overview 38
Vector On Interrupting Device Code (VCT) 162
Vectored interrupt 38
Virtual console
  altering data channel MAPS 72
  altering user MAPS 72
  auto program load 72
  B command 71
  breakpoints 71, 71, 71
  C command 72
  cells 69
  changing the MAP 72
  D command 71
  data channel 73
  displaying data channel MAPS 72
  displaying user MAPS 72
  dumping data channel MAPS 72
  dumping user MAPS 72
  entering 67
  error correction 68
  function commands 70
  G command 74
  H command 73
  I command 73
  interrupt 71
  K command 68
  L command 72
  M command 72
  O command 71
  P command 74
  power-up self-test 74
  program debugging 70
  prompts 67
  R command 72
  read/write memory test 74
  resuming program execution 71, 72
  rubout/delete key 68
  S command 73
  self-test routine 67
  single stepping 71
  U command 72
  uses 67
  V command 74
Virtual console compatibility 181

W

Word 5
  addressing format 8
  definition 8
Word offset 9
Word pointer 9
Write Character (DOA TTO) 54, 56, 92
Write protection, definition 59

X

XCH
Exchange Accumulators 17, 165
XCT, Exectute 165
XOP
Extended Operation 29, 32, 33, 166
XOP1
Alternate Extended Operation 32, 33, 166
XOR, Exclusive OR 17, 167
XORI
Exclusive OR Immediate 17, 167

DG OFFICES

NORTH AMERICAN OFFICES

Alabama: Birmingham
Arizona: Phoenix, Tucson
Arkansas: Little Rock
California: Anaheim, El Segundo, Fresno, Los Angeles, Oakland, Palo Alto, Riverside, Sacramento, San Diego, San Francisco, Santa Barbara, Sunnyvale, Van Nuys
Colorado: Colorado Springs, Denver
Connecticut: North Branford, Norwalk
Florida: Ft. Lauderdale, Orlando, Tampa
Georgia: Norcross
Idaho: Boise
Iowa: Bettendorf, Des Moines
Illinois: Arlington Heights, Champaign, Chicago, Peoria, Rockford
Indiana: Indianapolis
Kentucky: Louisville
Louisiana: Baton Rouge, Metairie
Maine: Portland, Westbrook
Maryland: Baltimore
Massachusetts: Cambridge, Framingham, Southboro, Waltham, Wellesley, Westboro, West Springfield, Worcester
Michigan: Grand Rapids, Southfield
Minnesota: Richfield
Missouri: Creve Coeur, Kansas City
Mississippi: Jackson
Montana: Billings
Nebraska: Omaha
Nevada: Reno
New Hampshire: Bedford, Portsmouth
New Jersey: Cherry Hill, Somerset, Wayne
New Mexico: Albuquerque
New York: Buffalo, Lake Success, Latham, Liverpool, Melville, New York City, Rochester, White Plains
North Carolina: Charlotte, Greensboro, Greenville, Raleigh, Research Triangle Park
Ohio: Brooklyn Heights, Cincinnati, Columbus, Dayton
Oklahoma: Oklahoma City, Tulsa
Oregon: Lake Oswego
Pennsylvania: Blue Bell, Lancaster, Philadelphia, Pittsburgh
Rhode Island: Providence
South Carolina: Columbia
Tennessee: Knoxville, Memphis, Nashville
Texas: Austin, Dallas, El Paso, Ft. Worth, Houston, San Antonio
Utah: Salt Lake City
Virginia: McLean, Norfolk, Richmond, Salem
Washington: Bellevue, Richland, Spokane
West Virginia: Charleston
Wisconsin: Brookfield, Grand Chute, Madison

INTERNATIONAL OFFICES

Argentina: Buenos Aires
Australia: Adelaide, Brisbane, Hobart, Melbourne, Newcastle, Perth, Sydney
Austria: Vienna
Belgium: Brussels
Bolivia: La Paz
Brazil: Sao Paulo
Canada: Calgary, Edmonton, Montreal, Ottawa, Quebec, Toronto, Vancouver, Winnipeg
Chile: Santiago
Columbia: Bogata
Costa Rica: San Jose
Denmark: Copenhagen
Ecuador: Quito
Egypt: Cairo
Finland: Helsinki
France: Le Plessis-Robinson, Lille, Lyon, Nantes, Paris, Saint Denis, Strasbourg
Guatemala: Guatemala City
Hong Kong
India: Bombay
Indonesia: Jakarta, Pusat
Ireland: Dublin
Israel: Tel Aviv
Italy: Bologna, Florence, Milan, Padua, Rome, Tourin
Japan: Fukuoka, Hiroshima, Nagoya, Osaka, Tokyo, Tsukuba
Jordan: Amman
Korea: Seoul
Kuwait: Kuwait
Lebanon: Beirut
Malaysia: Kuala Lumpur
Mexico: Mexico City, Monterrey
Morocco: Casablanca
The Netherlands: Amsterdam, Rijswijk
New Zealand: Auckland, Wellington
Nicaragua: Managua
Nigeria: Ibadan, Lagos
Norway: Oslo
Paraguay: Asuncion
Peru: Lima
Philippine Islands: Manila
Portugal: Lisbon
Puerto Rico: Hato Rey
Saudi Arabia: Jeddah, Riyadh
Singapore
South Africa: Cape Town, Durban, Johannesburg, Pretoria
Spain: Barcelona, Bibao, Madrid
Sweden: Gothenburg, Malmo, Stockholm
Switzerland: Lausanne, Zurich
Taiwan: Taipei
Thailand: Bangkok
Turkey: Ankara
United Kingdom: Birmingham, Bristol, Glasgow, Hounslow, London, Manchester
Uruguay: Montevideo
USSR: Espoo
Venezuela: Maracaibo
West Germany: Dusseldorf, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart

DataGeneral

Ordering Technical Publications

TIPS is the Technical Information and Publications Service—a new support system for DGC customers that makes ordering technical manuals simple and fast. Simple, because TIPS is a central supplier of literature about DGC products. And fast, because TIPS specializes in handling publications.

TIPS was designed by DG's Educational Services people to follow through on your order as soon as it's received. To offer discounts on bulk orders. To let you choose the method of shipment you prefer. And to deliver within a schedule you can live with.

How to Get in Touch with TIPS

Contact your local DGC education center for brochures, prices, and order forms. Or get in touch with a TIPS administrator directly by calling (617) 366-8911, extension 4086, or writing to

Data General Corporation
Attn: Educational Services, TIPS Administrator
MS F019
4400 Computer Drive
Westborough, MA 01580

TIPS. For the technical manuals you need, when you need them.

DGC Education Centers

Boston Education Center
Route 9
Southboro, Massachusetts 01772
(617) 485-7270

Washington, D.C. Education Center
7927 Jones Branch Drive, Suite 200
McLean, Virginia 22102
(703) 827-9666

Atlanta Education Center
6855 Jimmy Carter Boulevard, Suite 1790
Norcross, Georgia 30071
(404) 448-9224

Los Angeles Education Center
5250 West Century Boulevard
Los Angeles, California 90045
(213) 670-4011

Chicago Education Center
703 West Algonquin Road
Arlington Heights, Illinois 60005
(312) 364-3045

Data General

Technical Products Publications
Comment Form

Please help us improve our future publications by answering the questions below. Use the space provided for your comments.

Title:______________________________

Document No. 014-000686-00

Yes	No			
□	□	Is this manual easy to read?	○ You (can, cannot) find things easily. ○ Language (is, is not) appropriate. ○ Technical terms (are, are not) defined as needed.	○ Other:
		In whát ways do you find this manual useful?	○ Learning to use the equipment ○ As a reference ○ As an introduction to the product	○ To instruct a class. ○ Other:
□	□	Do the illustrations help you?	○ Visuals (are,are not) well designed. ○ Labels and captions (are,are not) clear. ○ Other:	
□	□	Does the manual tell you all you need to know? What additional information would you like?		
□	□	Is the information accurate? (If not please specify with page number and paragraph.)		

Name: ____________________ Title: ____________________

Company: ____________________ Division: ____________________

Address: ____________________ City: ____________________

State: __________ Zip: __________ Telephone: __________ Date: __________

Data General

Data General Corporation, Westboro, Massachusetts 01580

CUT ALONG DOTTED LINE

FOLD FOLD

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

CUT ALONG DOTTED LINE

Data General

users group

Installation Membership Form

Name ____________________ Position ____________________ Date __________

Company, Organization or School ____

Address ____________________ City ____________________ State ________ Zip __________

Telephone: Area Code __________ No. ______________ Ext. ____________________

1. Account Category

- □ OEM
- □ End User
- □ System House
- □ Government
- □ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other ______		
(Specify) ______		
______		

3. Software

□ AOS
□ DOS
□ MP/OS
□ RDOS
□ Other

Specify ______________

____

4. Languages

□ Algol
□ DG/L
□ Cobol
□ PASCAL
□ Business BASIC
□ BASIC
□ Assembler
□ Fortran
□ RPG II
□ PL/1
□ Other

Specify ______________

____

5. Mode of Operation

- □ Batch (Central)
- □ Batch (Via RJE)
- □ On-Line Interactive

6. Communications

□ HASP
□ RJE80
□ RCX 70
□ CAM
□ XODIAC
□ Other

Specify ______________

____

7. Application Description

○ ______________

____

____

____

____

____

8. Purchase

From whom was your machine(s) purchased?

□ **Data General Corp.**
□ Other
Specify ______________

____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ ______________

____

Data General

Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

FOLD FOLD

TAPE TAPE

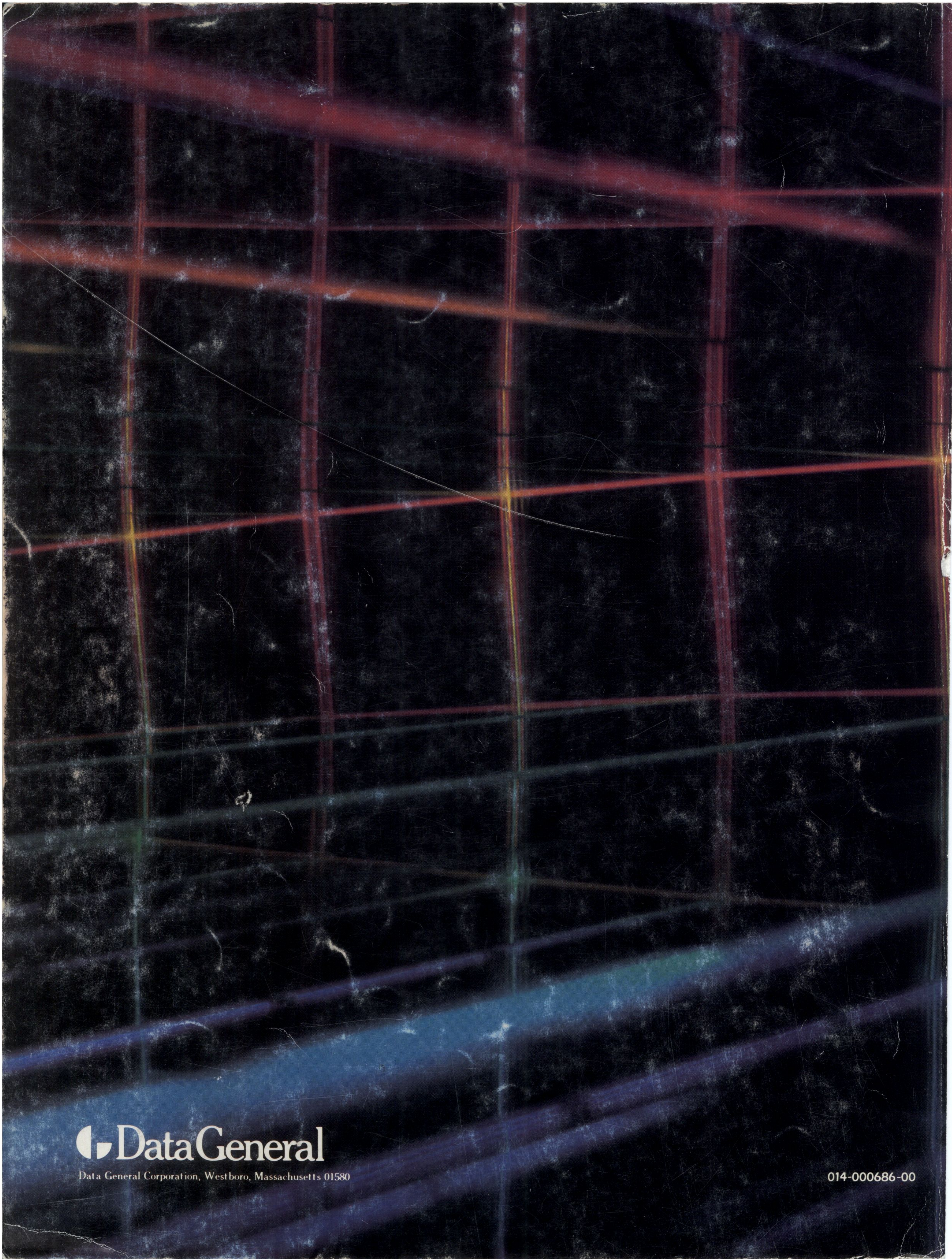
FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

DataGeneral
ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581



Data General

Data General Corporation, Westboro, Massachusetts 01580